Partial Visibility for Stylized Lines

Forrester Cole
Princeton University

Adam Finkelstein
Princeton University

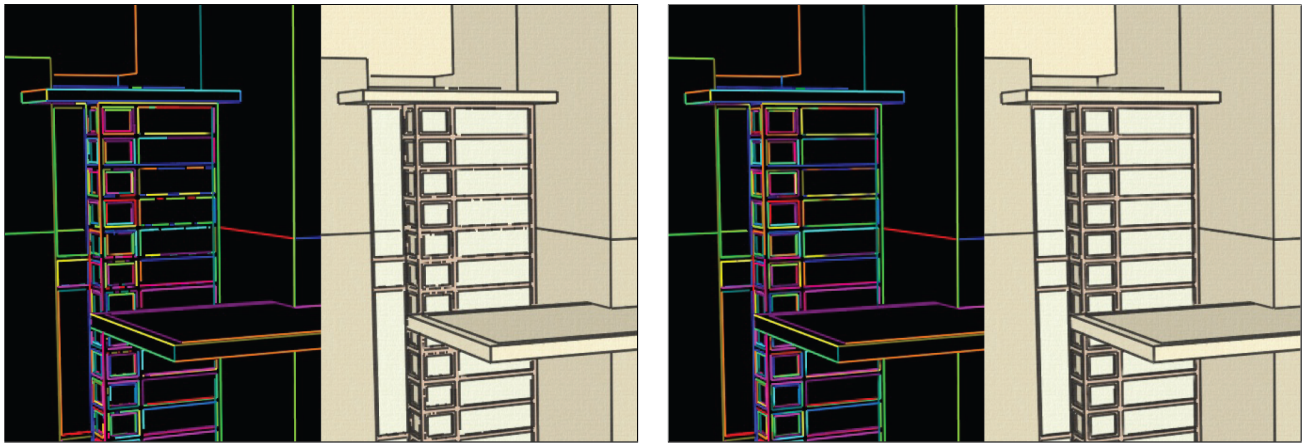

Figure 1: Anti-aliased line visibility. *Left pair:* Aliasing in the visibility test for lines (visualized over black) causes breaks and other artifacts in the rendered lines (drawn on beige model). *Right pair:* Supersampling and ID peeling removes these artifacts from the rendered lines.

Abstract

A variety of non-photorealistic rendering styles include lines extracted from 3D models. Conventional visibility algorithms make a binary decision for each line fragment, usually by a depth test against the polygons of the model. This binary visibility test produces aliasing where lines are partially obscured by polygons or other lines. Such aliasing artifacts are particularly objectionable in animations and where lines are drawn with texture and other stylization effects. We introduce a method for anti-aliasing the line visibility test by supersampling, analogous to anti-aliasing for polygon rendering. Our visibility test is inexpensive using current graphics hardware and produces partial visibility that largely ameliorates objectionable aliasing artifacts. In addition, we introduce a method analogous to depth peeling that further addresses artifacts where lines obscure other lines.

Keywords: NPR, Line Drawing, Visibility, Hidden Line Removal

1 Introduction

Common to many non-photorealistic rendering (NPR) techniques is the use some form of stylized lines. Such lines may include static 3D features such as creases and texture boundaries, as well as view-dependent features such as silhouettes, suggestive contours and suggestive highlights. Using the conventional graphics pipeline, such features may be drawn as solid, straight lines in 3D, and their visibilty can be resolved using the standard z-buffer algorithm (typically offset slightly relative to the polygons to disambiguate visibility where the lines and polygons are colocated). However, the z-buffer approach cannot be used for lines drawn with stylization effects such as varying thickness, over- or under-shoot, wavy path, and texture (e.g., Figure 6) because such effects cause the lines to be drawn in areas of the image near but not exactly identical to the location where visibility should be tested. Therefore, algorithms for drawing stylized lines from 3D models generally compute visibility for the lines prior to rendering them.

Typical line visibility algorithms generate a binary decision for every line fragment. Unfortunately such tests are subject to aliasing, leading to rendering artifacts such as those shown on the left in Figure 1. This paper describes an anti-aliasing approch to line visibility that results in partial visibility at every line fragment and, as shown on the right, largely ameliorates aliasing artifacts in the rendered lines. The specific contributions are:

- The notion of rendering lines with *partial visibility* as a mechanism for anti-aliasing.
- An algorithm for computing partial visibility of lines by super-sampling from an item buffer.
- The use of *ID peeling* in an item buffer to improve rendering quality where lines overlap in image space, and providing a quality-performance tradeoff when readback of the item buffer becomes expensive.

2 Background and Related Work

For an overview of line visibililty approaches, especially with regard to silhouettes, see the survey by Isenberg et al. [2003]. One general strategy combines visibility and rendering by simply causing the visible lines to appear in the image buffer, for example the techniques of Raskar and Cohen [1999] or more recently Lee et al. [2007], both of which worked at interactive frame rates

by using hardware rendering. These approaches limit stylization because by the time visibility has been calculated, the lines are already drawn. On the other hand, *explicit* computation of line visibility has been the subject of research since the 1960's. For example, Appel [1967] introduced the notion of *quantitative invisibility* (QI), and computed it by finding changes in visibility at certain (typically rare) locations. This approach was further improved and adapted to NPR by Markosian et al. [1997] who showed it could be performed at interactive frame rates for models of moderate complexity.

Appel's algorithm and its variants can be difficult to implement and are somewhat brittle when the lines are not in general position. Thus, Northrup and Markosian [2000] adapted the use of an *item buffer* (which had previously been used to accelerate ray tracing [Weghorst et al. 1984]) for the purpose of line visibility, calling it an "ID reference image" in this context. Several subsequent NPR systems have adopted this approach, e.g. [Kalnins et al. 2002; Kalnins et al. 2003; Cole et al. 2006], and the algorithm described in this paper also builds on this strategy. Kaplan [2007] described a method for computing QI using an item buffer, and we believe we could compute "partial" QI by combining his method with ours.

Any binary visibility test, including the item buffer approach, will lead to aliasing artifacts, analogous to those that appear for polygons when sampled into a pixel grid. The classic polygon aliasing artifacts are the "jaggies" that appear along the boundaries of a polygon, where the polygon covers only a fraction of a pixel. Considerable effort has been devoted to antialiasing for polygons [Foley et al. 1990]. A common strategy for addressing such artifacts is to supersample, for example with the use of the *A*-buffer [Carpenter 1984]. This paper demonstrates that such methods, originally developed for polygons, can be adapted to anti-alias line visibility with a similar quality-performance tradeoff.

This paper also describes *ID peeling*, which is based on the *depth peeling* approach described by Everitt [2001]. Depth peeling was originally used to correctly render transparent objects without depth sorting. As described in Section 3.2, the ID peeling can be adapted to allow the item buffer to store more than one line per pixel.

Finally we note that partially visible lines have already been used for various stylistic effects in NPR. For example Winkenbach and Salesin [1994] and Hertzmann and Zorin [2000] controlled line density among hatching lines by varying line weight and opacity.

3 Algorithm

The basis of our line rendering pipeline is the item buffer method of Northrup and Markosian [2000]. An item buffer is an off-screen buffer that contains visibility information for a set of 3D lines. To create an item buffer, the polygonal model is first drawn into the depth buffer. Each individual line is then drawn into the color buffer with a unique color (as in Figure 1), while testing against the model's depth buffer. For a model M, a set of 3D lines L, and associated colors C, the item buffer is created as follows:

```
def drawItemBuffer(M, L, C):
    set color mask = false, depth mask = true
    draw M
    set color mask = true, depth mask = false
    draw each l in L, colored by C
```

Lines are drawn with depth mask (writing to the depth buffer) disabled to prevent z-fighting between lines. Drawing the item buffer without depth writing usually does not cause additional visual artifacts, because when all lines are drawn in a similar style, it is usually not possible to tell which line is in front and which is behind.

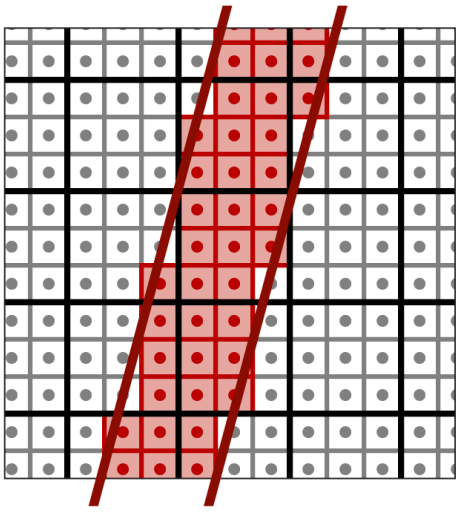

Figure 3: 9× *Supersampling.* Black lines are pixel boundaries, gray lines are subpixel sample boundaries. The edges of a fully visible line are dark red. Red subpixel samples fall within this line. Even though the line is fully visible, no single pixel (black box) contains nine red samples.

Each pixel of the item buffer contains the unique color of a single visible line fragment at that pixel. While efficient and fairly accurate, this approach suffers from two major flaws, one general and one peculiar to the item buffer. The general flaw is that an item buffer has limited resolution, and cannot be trivially anti-aliased due to the special meaning of the line colors. Any visibility algorithm (e.g. raytracing) suffers from this restriction, and our supersampling implementation also generalizes to visibility approaches besides the item buffer. The particular flaw is that only a single line color can exist as a given pixel, even if more than one line fragment is visible at that pixel. This restriction is due to the limited size of the graphics card's framebuffer, and ID peeling is a way to circumvent this hardware restriction.

3.1 Supersampling

The conventional approach to anti-aliasing for rendering is to take several sub-pixel samples and average their colors to obtain the final pixel color. This approach fails in the case of the item buffer, however, because color is used to encode the line indices. Averaging two colors results in a spurious line index. In order to supersample the item buffer, we need an aggregating operation that preserves the proper line indices.

Our approach is first to render the item buffer at high resolution – between two and six times the full-screen resolution. This can be done by either increasing the screen resolution and drawing the lines with width equal to the supersampling factor (width 2 for 2×2 supersampling), or drawing multiple copies of the item buffer with subpixel offsets. We chose the latter, because although drawing the geometry multiple times can degrade performance, we have noticed that thick lines in OpenGL behave unpredictably and may vary from platform to platform. Subpixel offsets also allow the possibility of jittered supersampling with random offsets, though we have not implemented such a scheme.

A fully visible line fragment may lie across the subpixel samples of multiple adjacent pixels (Figure 3). If each pixel contains n samples, we label each sample from $1..n$. A fragment's visibility is

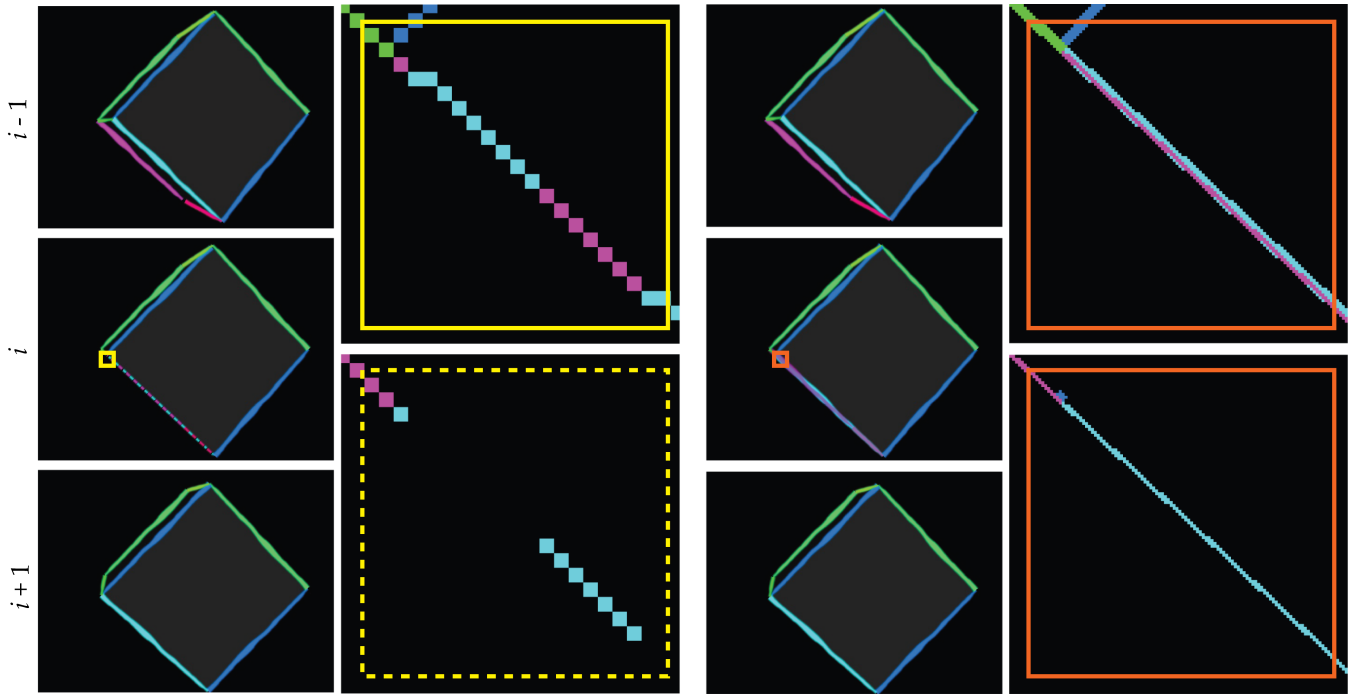

Figure 2: *Aliasing at visibility changes.* Item buffer aliasing and overwriting at changes in visibility can cause obvious artifacts in stylized lines (left). For a single item buffer (enlarged solid yellow), the pink edge interferes with the aqua edge as it transitions from visible to invisible. A hypothetical second item buffer layer (enlarged dotted yellow) could recover the pixels overwritten by other lines. With both supersampling and ID peeling the aqua line is intact and the pink line is partially visible on top of it (right: two layers of item buffer enlarged in orange).

determined by the number of sample labels covered in a 3x3 pixel neighborhood around the fragment. For example, if $n = 4$ and the line covers samples $1,3$ in one pixel and samples $2,4$ in an adjacent pixel, the fragment has full visibility. Because of the neighborhood check, this test can overestimate the visibility of a line fragment by up to one pixel.

3.2 ID Peeling

A conventional item buffer holds only a single ID per pixel. In even simple models, however, multiple lines will often project to the same item buffer pixel (Figure 2). This problem becomes worse as the model becomes more complex. For a large number of cases, however, only a small number of lines will fall on any single pixel (Table 2). We can exploit this property by adapting the technique of depth peeling [Everitt 2001].

In depth peeling, multiple layers of depth information are obtained by rendering the scene multiple times, each time allowing a fragment to pass the depth test only if it is farther from the camera than the closest fragment at the same position in the previous layer. Our version is similar, except that instead of using a second depth test, we allow a line fragment to pass only if its *index* is lower than the highest index at the corresponding pixel in the previous layer (assuming lines are drawn in ascending order). If the maximum number of lines overlapping a single pixel is n, we can recover the full visibility information in n passes.

The result is a set of n item buffers that when taken together, provide complete visibility information for each line (Figure 2).

4 Results

Supersampling and ID peeling together repair most of the visual artifacts associated with visibility testing using an item buffer (e.g., Figures 1 and 2). However, both methods impose a performance cost. Table 1 shows the effect of supersampling and ID peeling on framerates for the Falling Water model. The Falling Water model is a relatively complex model with many parallel and overlapping lines, so it provides a good "stress test" for our algorithm. Both supersampling and ID peeling impose a sub-linear performance cost, though ID peeling dominates. Tripling the number of layers (from 1 to 3) roughly halves frame rate, while for the same performance hit the number of samples may be increased from 1 to 9.

We have found that for almost all models and views, nearly full visibility information can be recovered with three or four item buffer layers, though to remove all artifacts under animation more layers may be required. More layers are also required at high supersampling levels, as supersampling tends to increase the item buffer depth complexity.

	Base	4×	9×	16×	25×
1 Layer	17.6	13.6	9.7	7.1	5.3
2 Layers	12.5	9.0	6.1	4.2	3.0
3 Layers	9.8	6.9	4.5	3.1	2.2

Table 1: *Frames per second for supersampling and ID peeling.* Timings are from a rotating a full view of the house model from Figure 1 at 800x600 window resolution. The model has approximately 10,000 line paths. Tested system had a 2.3GHz Athlon 64 CPU and an NVIDIA 8800GTS GPU.

Model \ Layers:	1	2	3	4	5	6
Box	0.89	0.11	0	0	0	0
House view 1	0.78	0.16	0.05	0.01	0	0
House view 2	0.57	0.24	0.09	0.06	0.02	0.01

Table 2: *Fraction of item buffer pixels that overlap multiple lines.* Of the item buffer pixels that touch any line, the vast majority touch four or fewer lines. The box view is Figure 2b, House view 1 is Figure 1, and House view 2 is the same view, but zoomed out until the entire model is visible (see Figure 5). No supersampling was used for this experiment.

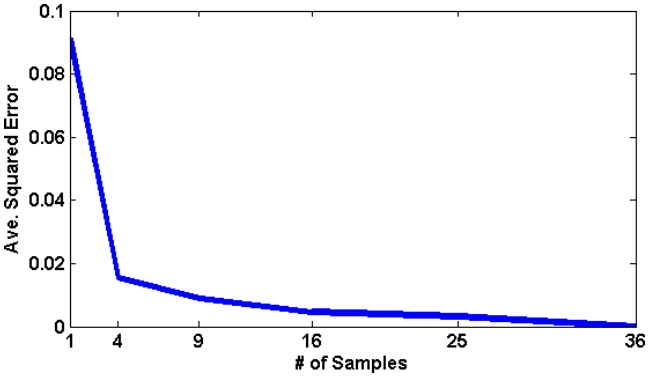

Figure 4: *Supersampling Error.* The average squared error in visibility against the number of samples taken, for the view in Figure 1. Error is computed against supersampling at 36x, which is considered zero error. Eight item buffer layers are used. The knee in the curve appears at 4× supersampling, and there is little gain after 16× supersampling.

Table 2 shows the percentage of item buffer pixels that touch one or more lines. For simple models such as the box, there are no pixels that contain more than two lines. For complex models such as the Falling Water, some pixels can overlap many lines, though for the view shown in Figure 1, 99% of all pixels that overlap any line overlap three or fewer lines. For a more difficult view, where the model is zoomed out until it fits entirely on the screen (Figure 5), 91% of all line pixels overlap three or fewer lines. Exceptional cases exist: for example, one could imagine zooming out until the entire model fit under a single pixel. In such pathological cases, however, perfect visibility information is usually not important.

The gain from supersampling generally falls off rapidly after only four samples, as measured by the average squared difference in visibility from an image created with 36 samples (Figure 4). Qualitatively, we find there are usually small visual differences that can be detected up to 16 samples, but after that point the visual impact of additional samples is minimal.

Finally, the accompanying video demonstrates the impact of these enhancements under animation. An animated cube shows aliasing artifacts that are addressed first by ID peeling and then by supersampling. The next example shows these effects for the more challenging Falling Water model, which includes many tiny overlapping lines. We note that in this example, 9× supersampling is used together with 8 ID layers, resulting in reduced aliasing. However, we believe that in this case, more sampling would further improve the result, but would exceed the memory capacity of our graphics card.

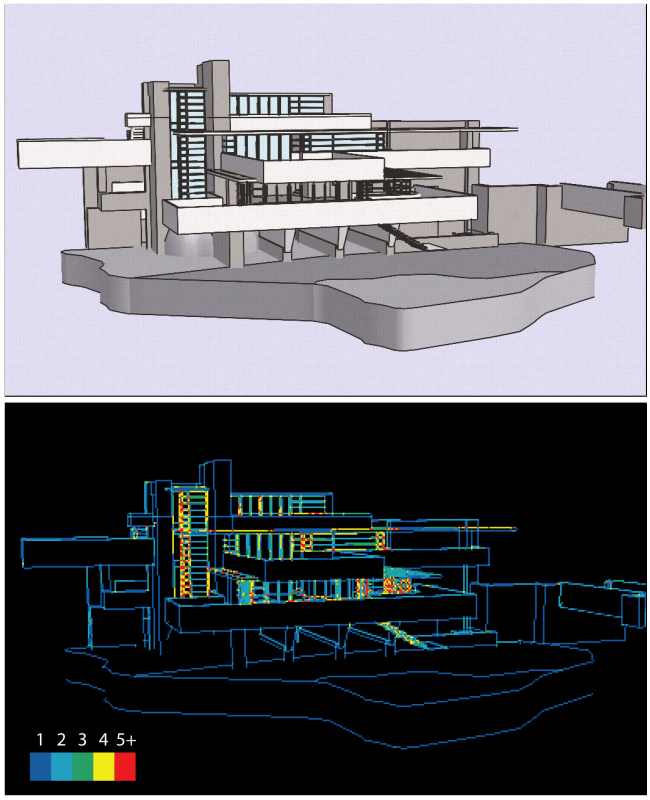

Figure 5: *ID Peeling.* Top: the "House 2" view. Bottom: visualization of item buffer layers. Colors indicate number of overlapping lines at each pixel.

5 Conclusion and Future Work

Supersampling and ID peeling together drastically reduce the number of objectionable visual artifacts when rendering stylized lines. For interactive applications, most of the benefit can be had by using only 4× supersampling and 3-4 item buffer layers. For offline animation, there is no reason not to use many samples and many item buffer layers to achieve the best possible quality.

There are several directions for future work in this area. First, our supersampling and ID peeling are currently too slow to achieve the best possible quality at interactive rates. While we were conscious of performance when creating our implementation, we left several possibilities for further optimization open.

For complex models and high supersampling rate, drawing the entire set of lines once per sample can become expensive. In these cases, it may be advantageous to draw a single, scaled-up item buffer and deal with the vagaries of thick line drawing.

The major and relatively fixed cost of the item buffer algorithm is the readback from the GPU and the processing on the CPU. Deeper layers of ID peeling tend to contain very few non-zero pixels. It may be possible to gain efficiency by using a hierarchical representation such as a quadtree where empty branches can be pruned.

In the longer run, our goal is to move the entire line visibility processing pipeline onto the GPU and avoid CPU-side processing as much as possible. Current graphics hardware should contain the functionality necessary to achieve this, but mapping the item buffer algorithm onto the GPU efficiently remains a challenge.

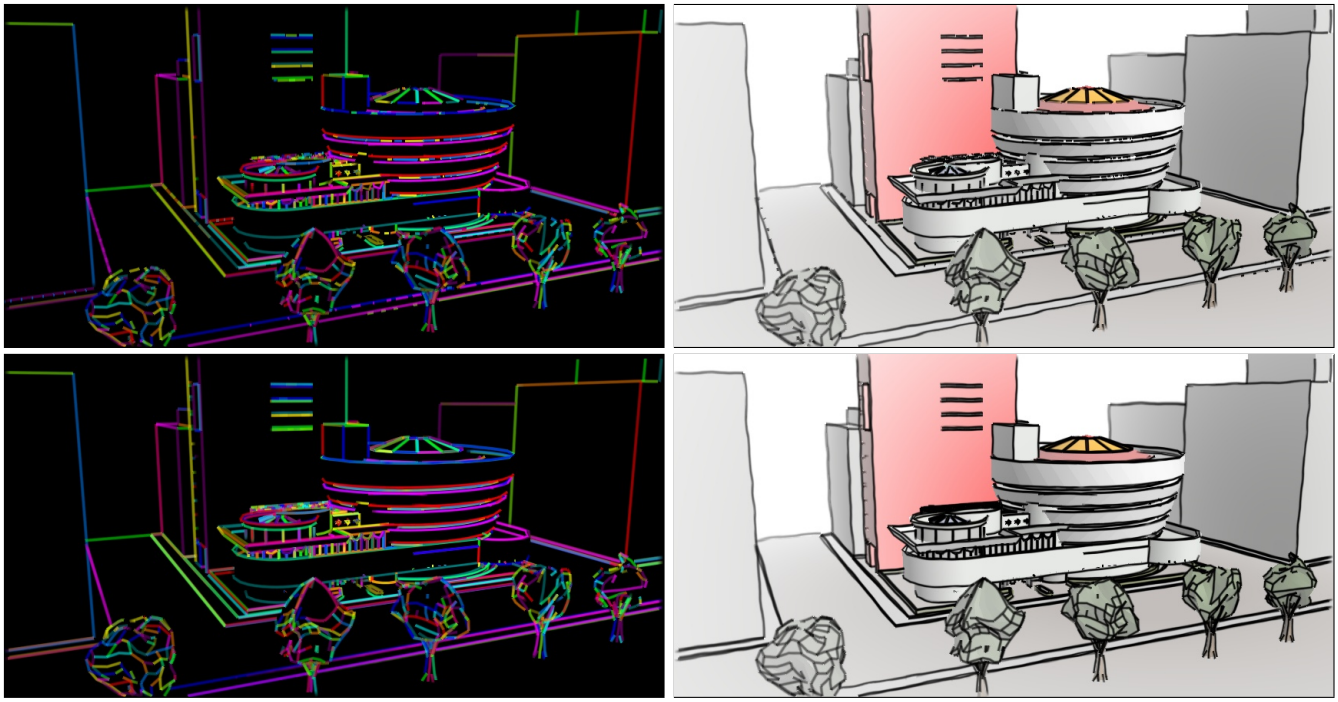

Figure 6: *Guggenheim Museum.* In typical views of complex models, there exist lines at the cusp of visibility (e.g., the rings of the tower). Top left: conventional item buffer visualization. Top right: resulting image. Bottom pair: 9× supersampling and 5 ID peeling layers.

References

APPEL, A. 1967. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 22nd national conference of the ACM*, 387–393.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18*, 3, 103–108.

COLE, F., DECARLO, D., FINKELSTEIN, A., KIN, K., MORLEY, K., AND SANTELLA, A. 2006. Directing gaze in 3D models with stylized focus. *Eurographics Symposium on Rendering* (June), 377–387.

EVERITT, C., 2001. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 517–526.

ISENBERG, T., FREUDENBERG, B., HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications 23*, 4 (July/Aug.), 28–37.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: drawing strokes directly on 3d models. In *Proceedings of SIGGRAPH 2002*, 755–762.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent stylized silhouettes. *ACM Transactions on Graphics 22*, 3 (July), 856–861.

KAPLAN, M. 2007. Hybrid quantitative invisibility. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, 51–52.

LEE, Y., MARKOSIAN, L., LEE, S., AND HUGHES, J. F. 2007. Line drawings via abstracted shading. *ACM Transactions on Graphics 26*, 3 (July), 18:1–18:5.

MARKOSIAN, L., KOWALSKI, M. A., GOLDSTEIN, D., TRYCHIN, S. J., HUGHES, J. F., AND BOURDEV, L. D. 1997. Real-time nonphotorealistic rendering. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 415–420.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic silhouettes: a hybrid approach. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, 31–37.

RASKAR, R., AND COHEN, M. 1999. Image precision silhouette edges. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 135–140.

WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. 1984. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (Jan.), 52–69.

WINKENBACH, G., AND SALESIN, D. H. 1994. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 1994*, 91–100.

Fast High-Quality Line Visibility

Forrester Cole
Princeton University

Adam Finkelstein
Princeton University

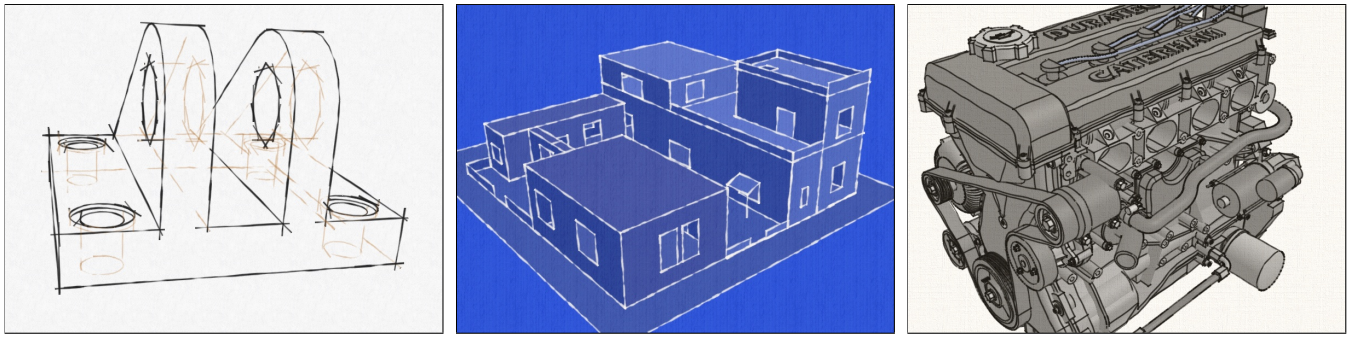

Figure 1: *Examples of models rendered with stylized lines.* Stylized lines can provide extra information with texture and shape, and are more aesthetically appealing than conventional solid or stippled lines.

Abstract

Lines drawn over or in place of shaded 3D models can often provide greater comprehensibility and stylistic freedom that shading alone. A substantial challenge for making stylized line drawings from 3D models is the visibility computation. Current algorithms for computing line visibility in models of moderate complexity are either too slow for interactive rendering, or too brittle for coherent animation. We present a method that exploits graphics hardware to provide fast and robust line visibility. Rendering speed for our system is usually within a factor of two of an optimized rendering pipeline using conventional lines, and our system provides much higher visual quality and flexibility for stylization.

Keywords: NPR, Line Drawing, Visibility, Hidden Line Removal

1 Introduction

Stylized lines play a role in many applications of non-photorealistic rendering (NPR) for 3D models (Figure 1). Lines can be used alone to depict shape, or in conjunction with polygons to emphasize features such as silhouettes, creases, and material boundaries. While graphics libraries such as OpenGL provide basic line drawing capabilities, their stylization options are limited. Desire to include effects such as texture, varying thickness, or wavy paths has lead to techniques to draw lines using textured triangle strips (*strokes*), for example those of Markosian, et al. [1997]. Stroke-based techniques provide a broad range of stylizations, as each stroke can be arbitrarily shaped and textured.

A major difficulty in drawing strokes is visibility computation. Conventional, per-fragment depth testing is insufficient for drawing broad strokes (Figure 2). Techniques such as the *item buffer* introduced by Northrup and Markosian [2000] can be used to compute visibility of lines prior to rendering strokes, but are much slower than conventional OpenGL rendering and are vulnerable to aliasing artifacts. While techniques exist to reduce these artifacts, they induce an even greater loss in performance. This paper presents a new method for testing visibility that removes the primary cause of aliasing in current techniques, and brings performance much closer to that of conventional rendering by moving the entire line visibility and drawing pipeline onto graphics hardware.

The specific contributions of this paper are:

- The description of an entirely GPU-based pipeline for hidden line removal and stylized stroke rendering.
- The introduction of the *segment atlas* as a data structure for efficient and accurate line visibility computation.

Applications for this approach include any context where interactive rendering of high-quality lines from 3D models is appropriate, including games, design and architectural modeling, medical and scientific visualization and interactive illustrations.

2 Background and Related Work

The most straightforward way to augment a shaded model with lines using the conventional rendering pipeline is to draw the polygons slightly offset from the camera and then to draw the lines, clipped against the model via the z-buffer. This is by far the most common approach, used by programs ranging from CAD and architectural modeling to 3D animation software, and because it leverages the highly-optimized pipeline implemented by graphics cards it imposes little overhead over drawing the shaded polygons alone. Unfortunately the lines resulting from this process admit only minimal stylistic control (color, fixed width, and in some implementations screen-space dash patterns).

Another general strategy combines visibility and rendering by simply causing the visible lines to appear in the image buffer, for example the techniques of Raskar and Cohen [1999] or more recently

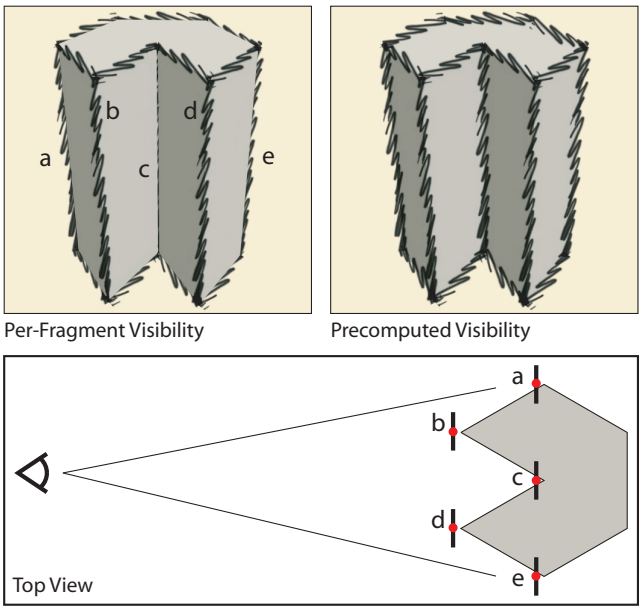

Figure 2: *Naive depth testing per-fragment vs. precomputed visibility.* When drawing wide lines, only lines that lie entirely outside the model will be drawn correctly (b and d). Lines a, c, e are partially occluded by the model, even when some polygon offset is applied. Visibility testing along the spine of the lines (red dots) prior to rendering strokes solves the problem.

Lee et al. [2007], both work at interactive frame rates by using hardware rendering. For example, the Raskar and Cohen method draws back-facing polygons in black, slightly displaced towards the camera from the front-facing polygons, so that black borders appear at silhouettes. Such approaches also limit stylization because by the time visibility has been calculated, the lines are already drawn.

To depict strokes with obvious character (e.g. texture, wobbles, varying width, deliberate breaks or dash patterns, tapered endcaps, overshoot, or haloes) Northrup and Markosian [2000] introduced a simple rendering trick wherin the OpenGL lines are supplanted by textured triangle strips. The naive approach to computing visibility for such strokes would be to apply a z-buffer test to the triangle strips describing strokes – a strategy that fails where many of the strokes interpenetrate the model (Figure 2). Therefore, NPR methods utilizing this type of stylization generally need to compute line visibility prior to rendering the lines. Line visibility has been the subject of research since the 1960's. Appel [1967] introduced the notion of *quantitative invisibility*, and computed it by finding changes in visibility at certain (typically rare) locations. This approach was further improved and adapted to NPR by Markosian et al. [1997] who showed it could be performed at interactive frame rates for models of modest complexity.

Appel's algorithm and its variants can be difficult to implement and are somewhat brittle when the lines are not in general position. Thus, Northrup and Markosian [2000] adapted the use of an *item buffer* (which had previously been used to accelerate ray tracing [Weghorst et al. 1984]) for the purpose of line visibility, calling it an "ID reference image" in this context. Several subsequent NPR systems have adopted this approach, e.g. [Kalnins et al. 2002; Kalnins et al. 2003; Cole et al. 2006]. For an overview of line visibility approaches (especially with regard to silhouettes, which present a particular challenge because they lie at the cusp of visibility), see the survey by Isenberg et al. [2003]. Any binary visibility test, including the item buffer approach, will lead to aliasing artifacts, analogous to those that appear for polygons when sampled into a pixel grid. To ameliorate aliasing artifacts Cole and Finkelstein [2008] showed how to adapt to line drawing the supersampling and depth-peeling strategies previous described for polygons, introducing the notion of *partial visibility* for lines.

While the item buffer approach can determine line visibility at interactive frame rates of moderate complexity, it is slow for large models. Moreover, computation of partial visibility – which significantly improves visual quality, especially under animation – imposes a further burden on frame rates. Our current method provides high-quality hidden line removal (with or without partial visibility) at interactive frame rates for complex models.

3 Algorithm

Our algorithm begins with a set of lines extracted from the model. Most of our experiments have focused on lines that are fixed on the model, for example creases or texture boundaries. However, our system also supports the extraction of silhouette edges from a pool of faces whose normals are interpolated (e.g. the rounded top of the clevis on the left in Figure 1). Our goal is to determine which portions of these segments are visible.

Our line visibility pipeline has three major stages, illustrated in Figure 3: line projection and clipping (Section 3.1), creation of the segment atlas (Section 3.2), and visibility testing (Section 3.3). All stages execute on the GPU, and all data required for execution resides in GPU memory in the form of OpenGL framebuffer objects or vertex buffer objects. The input to the algorithm is a set of N line strips (which we call *paths*), each divided into one or more segments. The output of the algorithm is a segment atlas containing per-sample visibility information for each segment. Finally, after visibility has been determined via this pipeline, there are two general strategies for rendering the lines, as described in Section 3.4.

3.1 Projection and Clipping

The first stage of the visibility pipeline begins with a set of candidate line segments, projects them, and clips them to the viewing frustum. Ideally, we would use the GPU's clipping hardware to clip each segment. However, in current graphics hardware the output of the clipper is not available until the fragment program stage, after rasterization has already been performed. We therefore use a fragment program to project and clip the segments. The input to the program is a six-channel framebuffer object packed with the 3D coordinates of the endpoints of each segment $(\mathbf{p},\mathbf{q})$ (Figure 3a). This buffer must be updated at each frame with the positions of any moving line segments. The output of the program is a nine-channel buffer containing the 4D homogeneous clip coordinates $(\mathbf{p}',\mathbf{q}')$ and the number of visibility samples l (Figure 3b). The number of visibility samples l is determined by:

$$l = \lceil ||\mathbf{p}'_w - \mathbf{q}'_w|| / k \rceil \tag{1}$$

where $(\mathbf{p}'_w,\mathbf{q}'_w)$ are the 2D window coordinates of the segment endpoints, and k is a screen-space sampling rate. The factor k trades off positional accuracy in the visibility test against segment atlas size. We usually set $k = 1$ or 2, meaning visibility is determined every 1 or 2 pixels along each line; there is diminishing visual benefit in determining with any greater accuracy the exact position at which a line becomes occluded.

A value of $l = 0$ is returned for segments that are entirely outside the viewing frustum. Segments for which $l \leq 1$ (i.e., sub-pixel sized segments) are discarded for efficiency if not part of a path, but otherwise must be kept or the path will appear disconnected.

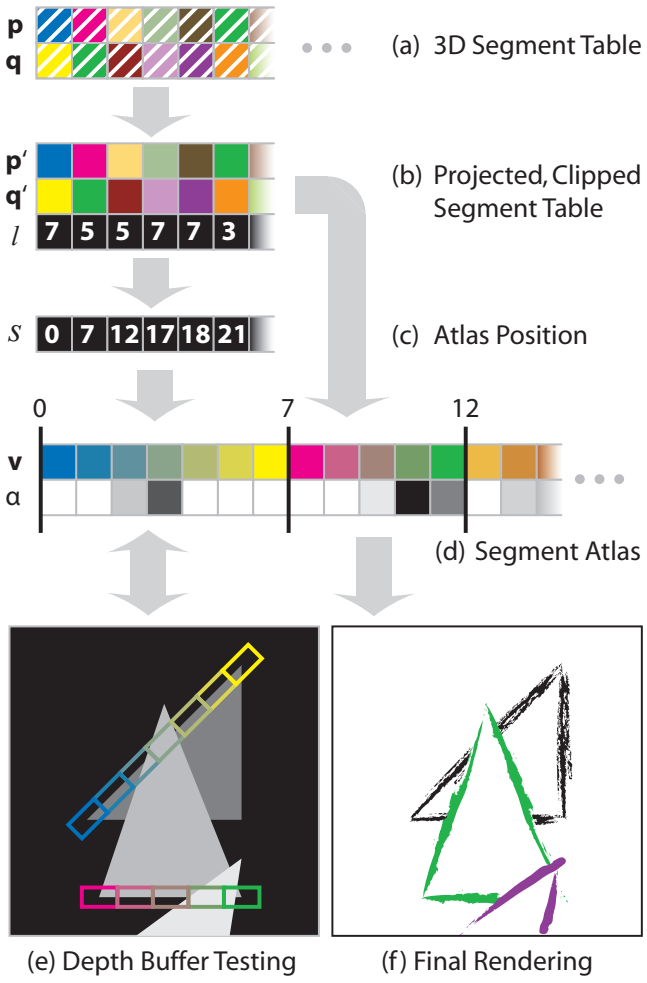

Figure 3: *Pipeline.* (a) The 3D line segments $(\mathbf{p}_i, \mathbf{q}_i)$ are stored in a table. (b) A fragment program projects and clips each segment to produce $(\mathbf{p}'_i, \mathbf{q}'_i)$, and determines a number of samples l_i proportional to its screen space length. (c) A scan operation computes the atlas positions s from the running sum of l. (d) Sample positions $\mathbf{v}$ are interpolated from $(\mathbf{p}', \mathbf{q}')$ and written into the segment atlas at offset s. Visibility values α for each sample are determined by probing the depth buffer (e) at $\mathbf{v}$, and are used to generate the final rendering (f). Note the schematic colors used throughout for the blue-yellow and pink-green segments.

While not a specific contribution of our method, we note that performing projection and clipping in this manner makes it very easy to rapidly extract silhouette edges from a portion of a mesh whose normals are interpolated, such as the rounded top of the clevis on the left in Figure 1. During clipping, neighboring face normals may be checked for a silhouette edge condition (one front-facing and one back-facing polygon). If the edge is not a silhouette, it is discarded by setting $l = 0$. This method is similar to the approach of Brabec and Seidel [2003] for computing shadow volumes on the GPU.

3.2 Segment Atlas Creation

The segment atlas is a table of segment samples. Each segment is allocated l entries in the atlas, and each entry consists of a clip space position $\mathbf{v}$ and a visibility value α (Figure 3d). The interpolated sample positions $\mathbf{v}$ are created by drawing single-pixel wide lines into the atlas, using the conventional OpenGL line drawing commands. A fragment program performs the interpolation of $\mathbf{p}'$ and $\mathbf{q}'$ and the perspective division step to produce each $\mathbf{v}$, simultaneously checking the visibility at the sample (Section 3.3).

Before the segment atlas can be constructed, we need to determine the offset s of each segment into this data structure, which is the running sum of the sample counts l (Figure 3c). The sum is calculated by performing an exclusive scan operation on l [Sengupta et al. 2007]. Once the atlas position s is computed, each segment may be drawn in the atlas independently and without overlap.

The most natural representation for the segment atlas is as a very long, 1D texture. Unfortunately, current GPUs do not allow for arbitrarily long 1D textures, at least as targets for rendering. The segment atlas can be mapped to two dimensions by wrapping the atlas positions at a predetermined width w, usually the maximum texture width W allowed by the GPU ($W = 4096$ or 8192 texels is common). The 2D atlas $\mathbf{s}$ is given by:

$$\mathbf{s} = (s \bmod w, \lfloor s/w \rfloor) \tag{2}$$

The issue then becomes how to deal with segments that extend outside the texture, i.e., segments for which $(s \bmod w) + l > w$. One way to address this problem is to draw the segment atlas twice, once normally and once with the projection matrix translated by $(-w, 1)$. Long segments will thus be wrapped across two consecutive lines in the atlas. Specifically, suppose L is the largest value of l, which can be conservatively capped at the screen diagonal distance divided by k. If $w > L$, drawing the atlas twice is sufficient, because we are guaranteed that each segment requires at most one wrap. Drawing twice incurs a performance penalty, but as the visibility fragment program is usually the bottleneck (and is still run only once per sample) the penalty is usually small.

For some rendering applications, however, it is considerably more convenient if segments do not wrap (Section 3.4). In this case, we establish a gutter in the 2D segment atlas by setting $w = W - L$. The atlas position is then only drawn once. This approach is guaranteed to waste $W - L$ texels per atlas line. Moreover, this waste exacerbates the waste due to our need to preallocate a large block of memory for the segment atlas without knowing how full it will become. Nevertheless, the memory usage of the segment atlas (which is limited by the number of lines drawn on the screen) is typically dominated by that of the 3D and 4D segment tables (which must hold all lines in the scene).

3.3 Visibility Testing

As mentioned in Section 3.2 the visibility test for each sample is performed during rasterization of the segments into the segment atlas. The visibility of a sample is computed by comparing the projected depth value of the sample with the depth value of the nearest polygon under the sample, much like a conventional z-buffer scheme. As noted by Cole and Finkelstein [2008], aliasing in the visibility test for lines can cause severe visual artifacts, especially under animation. Unlike the item buffer approach, the segment atlas method is not vulnerable to interference among lines, making a multi-layered segment atlas unnecessary. However, there are still two potential sources of aliasing error: aliasing of the per-sample depth test, and aliasing in the depth buffer with respect to the original polygons. Both these sources of aliasing can be addressed with supersampling.

During the drawing of the atlas, a fragment program computes an interpolated homogeneous clip space coordinate for each sample and performs the perspective division step. The resulting clip space z value is then compared to a depth buffer of the scene polygons. Using a single test for this comparison produces aliasing

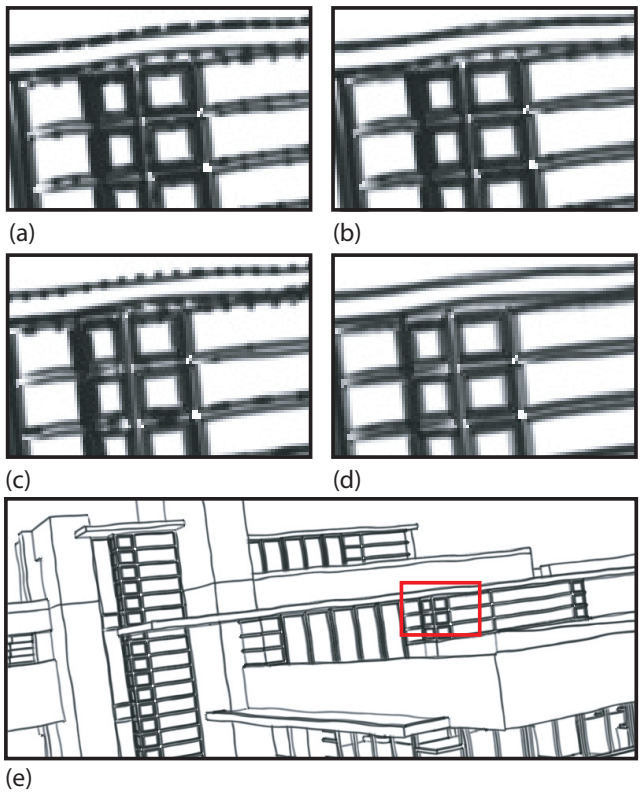

Figure 4: *Aliasing in visibility test.* Results for varying number of samples and scale of depth buffer. (a) 1 sample, 1x depth buffer. (b) 16 samples, 1x depth buffer. (c) 1 sample, 3x depth buffer. (d) 16 samples, 3x depth buffer. Red box in (e) indicates location of magnified area in the Falling Water model.

(Figure 4a). Adding additional probes in a box filter configuration around the sample gives a more accurate occlusion value for the line sample (Figure 4b). Additional depth probes are cheap, but not free. The impact of increased sampling is more visible in complex scenes, with large segment atlases (see Table 1).

Any number of depth probes will not produce an accurate result if the underlying depth buffer has aliasing error. While impossible to eliminate entirely, this source of aliasing can be reduced through supersampling of the depth buffer by increasing the viewport resolution. Since typical applications are seldom fillrate bound for simple operations like drawing the depth buffer, increasing the size of the buffer typically has little impact on performance outside of an increase in memory usage. While simply scaling the depth buffer without adding additional depth probes for each sample produces a marginal increase in image quality (Figure 4c), combining depth buffer scaling and depth test supersampling largely eliminates aliasing artifacts (Figure 4d).

3.4 Stroke Rendering

After visibility is computed, all the information necessary to draw strokes is available in the projected and clipped segment table and the segment atlas. The most efficient way to render the strokes is to generate, on the host, a single quad per segment. A vertex program then positions the vertices of the quad relative to $(\mathbf{p}', \mathbf{q}')$, taking into account the pen width and proper mitreing for multi-segment paths. A fragment program textures the quad with a 2D pen texture, and modulates the texture with the corresponding 1D visibility values

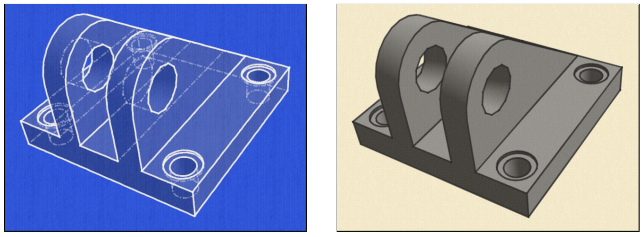

Figure 5: *Variation in style.* A different texture may be used for lines that fail the visibility test (*left*), allowing visualization of hidden structures. Our method also produces attractive results for solid, simple styles (*right*).

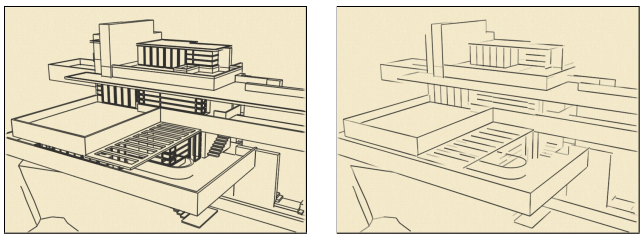

Figure 6: *Line density control.* One reason to read back the segment atlas to the host is to control screen space line density. Left: no density control. Right: line density reduction as in [Cole2006].

from the segment atlas. Additional stylization effects such as line overshoot can be added easily in the vertex program stage (Figure 5). All results shown in this paper were generated using this rendering method, with the exception of Figure 6.

In some cases it may be desirable to read back the segment atlas visibility values for processing on the host. One example could be to implement a stroke-based line density control scheme (e.g., [Grabli et al. 2004; Cole et al. 2006]). An example of the latter method for line density control, implemented in our system as a post-process to the visible paths, is shown in Figure 6. Reading back and processing the entire segment atlas is inefficient, since for reasonably complex models the vast majority of line samples in any given frame will have zero visibility. Thus we apply a stream compaction operation [Horn 2005] to the segment atlas visibility values. This yields a packed buffer with only visible samples remaining, which is suitable for readback to the host. An added benefit of the segment atlas approach compared to the item buffer approach is that the line samples in this compacted buffer are ordered by path and segment, and can therefore be efficiently converted to geometry. By comparison, the visibility samples in an item buffer are ordered by screen space position, and must be sorted or otherwise processed before use. For models of moderate complexity the performance of this rendering approach is roughly comparable to that of the GPU-rendered approach described above, with a additional fixed cost of ~ 20 ms per frame for stream compaction and read-back.

For either rendering strategy described above, the geometry is stylized via 2D images of marks in the style of pen, pencil, charcoal, etc. We use periodic textures parameterized uniformly in screenspace. Changes in this paremeterization from frame to frame influence temporal coherence of the lines, as can be observed in the accompanying video. Since the emphasis in this paper is on visibility, we use the simple strategy of fixing the "zero" parameter value along the length of the stroke at its screen-space center. A more sophisticated strategy that seeks temporal coherence from frame to frame was described by Kalnins et al. [2003].

Model	# tris	# segs	OGL	IB1	IB2	SA1	SA2
clevis	1k	1.5k	1000+	87	20	149	149
house	15k	14k	300+	24	3.4	119	97
ship	300k	300k	42	9.6	0.52	30	26
office	330k	300k	32	7.0	0.35	25	16
ship+s	-	500k	-	-	-	20	14
office+s	-	400k	-	-	-	22	13

Table 1: *Frame rates (fps) for various models rendering methods.* All frames rendered at 1024×768. Timings for clevis and house are averaged over an orbit of the model. Timings for ship and office are averaged over a walkthrough sequence (accompanying video). The "+s" indicates silhouettes were extracted and drawn. OGL: conventional OpenGL lines. IB1: single item buffer [Northrup2000]. IB2: $9\times$ supersampled item buffer with 3 layers [Cole2008]. SA1: single probe segment atlas (comparable to IB1). SA2: 9 probe segment atlas with $2\times$ scaled depth buffer (comparable to IB2).

4 Results

We implemented the segment atlas approach using OpenGL and GLSL, taking care to manage GPU-side memory operations efficiently. For comparison we also implemented an optimized conventional OpenGL rendering pipeline using line primitives, and the item buffer approach of Northrup and Markosian [2000], and the improved item buffer approach of Cole and Finkelstein [2008]. We did not use NVIDIA's CUDA architecture, because the segment atlas drawing step uses conventional line rasterization and the rasterization hardware is unavailable from CUDA.

Table 1 shows frame rates for four models ranging from 1k-500k line segments. These numbers were generated on a commodity Dell PC running Windows XP with an Intel Core 2 Duo 2.4 GHz CPU and 2GB RAM, and an NVIDIA 8800GTS GPU with 512MB RAM. For small models, our approach pays a moderate overhead cost, and therefore is at least several factors slower than conventional OpenGL rendering (though absolute speed is still high). For the more complex models, however, our method is within 50% of conventional OpenGL, while providing high image quality (SA2). Our low image quality setting (SA1) is within 75%, and still provides good quality, though with some aliasing artifacts.

Our method is always considerably faster than the item buffer based approach, but the most striking difference is when comparing the high quality modes of each method. The item buffer approach with $9\times$ supersampling and 3 layers, as suggested by [Cole and Finkelstein 2008], gives similar image quality to our method with 9 depth probes and $2\times$ scaled depth buffer. Our method, however, delivers a performance increase of up to $50\times$ for complex models.

As mentioned in Section 3.1, our method also allows for easy extraction and rendering of silhouette edges on the GPU. The last two rows of Table 1 show the performance impact for our method when extracting and rendering silhouettes. The increase in cost is roughly proportional to the increase in the total number of potential line segments. We did not implement silhouette extraction for the other methods, however, silhouette extraction can be a costly operation when performed on the CPU.

While accurate timing of the stages of our algorithm is difficult due to the deep OpenGL pipeline, the major costs of the algorithm ($\sim$80-90% of total) lie in the sample visibility testing stage, depth buffer drawing stage (for complex models), and segment atlas setup. Projection, clipping, and stroke rendering are minor costs.

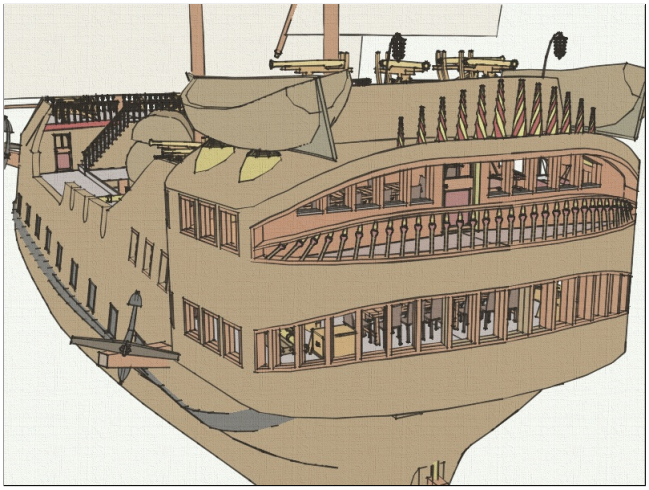

Figure 7: *Ship model.* The ship model has 300k triangles and 500k total line segments, and can be rendered at high-quality and interactive frame rates using our method.

5 Conclusion and Future Work

The proposed algorithm allows rendering of high-quality stylized lines at speeds approaching those of the conventional OpenGL rendering pipeline. The algorithm provides improved temporal coherence and less aliasing (sparkle) than previous approaches for drawing stylized lines, making it suitable for animation of complex scenes. Compared with previous approaches for computing line visibility, it is robust and conceptually simple. We believe this approach will be useful for interactive applications such as games and interactive design and modeling software, where previously the performance penalty for using stylized lines has been prohibitive.

Future work in this area may include extending the approach to include further integration of **line density control** methods such as proposed in [Grabli et al. 2004; Cole et al. 2006]. As mentioned in Section 3.4 our current system allows for such algorithms by reading the visible line paths back onto the CPU and then processing the visible lines using previous methods. However, we believe that it will be possible to handle the entire pipeline on the GPU. One challenge is that these approaches will need to be adapted to deal with partial visibility of lines.

While not a direct extension of our method, we would also like it to handle other **view-dependent lines** such as smooth silhouettes [Hertzmann and Zorin 2000], suggestive contours [DeCarlo et al. 2003], and apparent ridges [Judd et al. 2007]. Including these line types at a reasonable performance cost may require an extraction algorithm that executes on the GPU. In contrast to lines that are fixed on the model, consistent parameterization of such lines from frame to frame presents its own challenge [Kalnins et al. 2003].

While currently fast, we believe there are opportunities to further improve the **scalability** of our approach. In our implementation, all segments are recorded explicitly in the segment table, which we show give interactive performance for models with hundreds of thousands of segments. Many large models make use of scene graph hierarchies with instancing – which affords two opportunities for improved scalability. First, an improvement to the segment table would allow for line set instancing, which would make more efficient use of texture memory on-card. Second, hierarchical representations can be used to quickly reject sections of the model that not potentially visible.

Acknowledgments

We would like to thank Michael Burns and the Google 3D Warehouse for the models shown in this paper. This work was sponsored in part by the NSF grant IIS-0511965.

References

APPEL, A. 1967. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 22nd national conference of the ACM*, 387–393.

BRABEC, S., AND SEIDEL, H.-P. 2003. Shadow volumes on programmable graphics hardware. In *EUROGRAPHICS 2003*, vol. 22 of *Computer Graphics Forum*, Eurographics, 433–440.

COLE, F., AND FINKELSTEIN, A. 2008. Partial visibility for stylized lines. In *NPAR 2008*.

COLE, F., DECARLO, D., FINKELSTEIN, A., KIN, K., MORLEY, K., AND SANTELLA, A. 2006. Directing gaze in 3D models with stylized focus. *Eurographics Symposium on Rendering* (June), 377–387.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Trans. Graph. 22*, 3, 848–855.

GRABLI, S., DURAND, F., AND SILLION, F. 2004. Density measure for line-drawing simplification. In *Proceedings of Pacific Graphics*.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proceedings of SIGGRAPH 2000*, 517–526.

HORN, D. 2005. Stream reduction operations for gpgpu applications. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, ch. 36, 573–589.

ISENBERG, T., FREUDENBERG, B., HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications 23*, 4 (July/Aug.), 28–37.

JUDD, T., DURAND, F., AND ADELSON, E. H. 2007. Apparent ridges for line drawing. *ACM Trans. Graph. 26*, 3, 19.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: drawing strokes directly on 3d models. In *Proceedings of SIGGRAPH 2002*, 755–762.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent stylized silhouettes. *ACM Transactions on Graphics 22*, 3 (July), 856–861.

LEE, Y., MARKOSIAN, L., LEE, S., AND HUGHES, J. F. 2007. Line drawings via abstracted shading. *ACM Transactions on Graphics 26*, 3 (July), 18:1–18:5.

MARKOSIAN, L., KOWALSKI, M. A., GOLDSTEIN, D., TRYCHIN, S. J., HUGHES, J. F., AND BOURDEV, L. D. 1997. Real-time nonphotorealistic rendering. In *Proceedings of SIGGRAPH 1997*, 415–420.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic silhouettes: a hybrid approach. In *NPAR 2000*, 31–37.

Figure 8: *Office model.* The office model has five levels, each with detailed furniture, totaling 330k triangles and 400k line segments.

RASKAR, R., AND COHEN, M. 1999. Image precision silhouette edges. In *Proceedings of SI3D 1999*, ACM Press, New York, NY, USA, 135–140.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware 2007*, 97–106.

WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. 1984. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (Jan.), 52–69.

Two Fast Methods for High-Quality Line Visibility

Forrester Cole and Adam Finkelstein

Abstract—Lines drawn over or in place of shaded 3D models can often provide greater comprehensibility and stylistic freedom than shading alone. A substantial challenge for making stylized line drawings from 3D models is the visibility computation. Current algorithms for computing line visibility in models of moderate complexity are either too slow for interactive rendering, or too brittle for coherent animation. We introduce two methods that exploit graphics hardware to provide fast and robust line visibility. First, we present a simple shader that performs a visibility test for high-quality, simple lines drawn with the conventional implementation. Next, we offer a full optimized pipeline that supports line visibility and a broad range of stylization options.

Index Terms—Visible line, surface algorithms, non-photorealistic rendering.

1 Introduction

Stylized lines play a role in many applications of nonphotorealistic rendering (NPR) for 3D models. Lines can be used alone to depict shape, or in conjunction with polygons to emphasize features such as silhouettes, creases, and material boundaries. While graphics libraries such as OpenGL provide basic line drawing capabilities, their stylization options are limited. Desire to include effects such as texture, varying thickness, or wavy paths has lead to techniques that draw lines using textured triangle strips (*strokes*), for example, those of Markosian, et al. [1]. Stroke-based techniques provide a broad range of stylizations, as each stroke can be arbitrarily shaped and textured.

A major difficulty in drawing strokes is visibility computation. Conventional, per-fragment depth testing is insufficient for drawing broad strokes, because the strokes are partially occluded by the model itself (Fig. 2). Techniques such as the *item buffer* introduced by Northrup and Markosian [2] can be used to compute visibility of lines prior to rendering strokes, but are much slower than conventional OpenGL rendering and are vulnerable to aliasing artifacts. While techniques exist to reduce these artifacts [3], they induce an even greater loss in the performance.

This paper presents two methods that exploit graphics hardware to draw strokes efficiently and with high-quality visibility testing:

1. **Spine test shader.** This simple method can be used in a conventional line drawing pipeline with minimal modification, but supports a limited range of stylization.
2. **Segment atlas.** This method carries a higher implementation cost that the spine test shader, but provides stored visibility values can be used for stylization, as well as to properly handle curved strokes.

Both methods rely on a conventional depth buffer to determine visibility, but provide support for supersampling in both the depth buffer and the lines themselves (Fig. 5). Both methods provide a similar level of visibility quality and speed.

The major difference between the methods is that the segment atlas method stores visibility information in an intermediate data structure (the *segment atlas*), while the spine test method does not. The spine test method is a single-pass approach that computes stroke visibility at the same time as the final stroke color. The segment atlas method, by contrast, computes and stores the visibility information for all strokes prior to rendering. Computing visibility prior to rendering provides the option to filter or otherwise manipulates the visibility values, allowing effects such as overshoot, haloing, and detail elision. An additional benefit is the ability to properly parameterize strokes with multiple segments, such as curved strokes (e.g., the top of the clevis shape in Fig. 1 (left)).

This paper expands on an earlier paper by the same authors [4] that introduced the segment atlas method. The spine test method is introduced for the first time in this paper and offers a simpler, more conventional alternative to the segment atlas method. This paper also expands upon the description of the segment atlas in [4], adding implementation improvements, further discussion of stylization effects, and a comparison to the spine test method.

Applications for these approaches include any context where interactive rendering of high-quality lines from 3D models is appropriate, including games, design and architectural modeling, medical and scientific visualization, and interactive illustrations.

2 Background and Related Work

The most straightforward way to augment a shaded model with lines using the conventional rendering pipeline is to draw the polygons slightly offset from the camera, and then, to draw line primitives, clipped against the model via

• *The authors are with Princeton University, 35 Olden St., Princeton, NJ 08544. E-mail: {fcole, af}@cs.princeton.edu.*

Manuscript received 25 Feb. 2009; revised 20 May 2009; accepted 11 Aug. 2009; published online 19 Aug. 2009.
Recommended for acceptance by M. McGuire and E. Haines.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2009-02-0045.
Digital Object Identifier no. 10.1109/TVCG.2009.102.

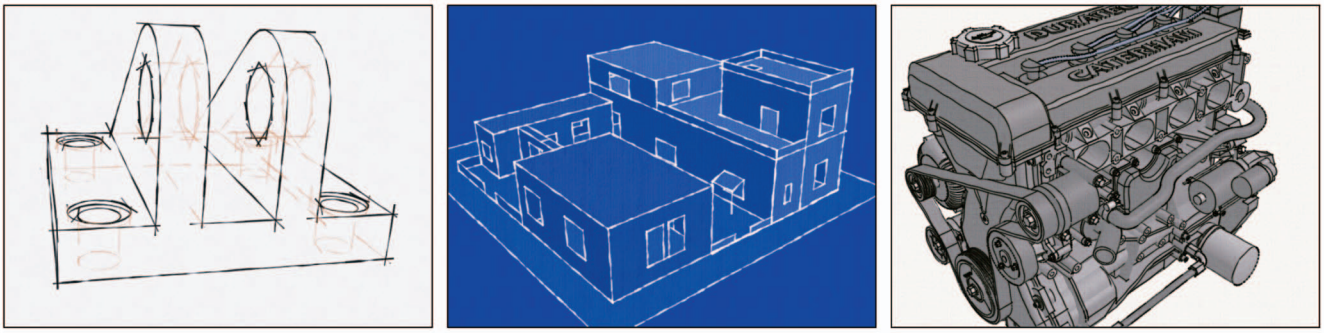

Fig. 1. Examples of models rendered with stylized lines. Stylized lines can provide extra information with texture and shape, and are more aesthetically appealing than conventional solid or stippled lines.

the z-buffer. This is by far the most common approach, used by programs ranging from CAD and architectural modeling to 3D animation software, because it leverages the highly optimized pipeline implemented by graphics cards and imposes little overhead over drawing the shaded polygons alone. Unfortunately, hardware-accelerated line primitives are usually rasterized with a specialized approach such as described by Wu [5], and allow only minimal stylistic control (color, fixed width, and in some implementations screen-space dash patterns).

Another general strategy combines visibility and rendering by simply causing the visible lines to appear in the image buffer. The techniques of Raskar and Cohen [6] and Lee et al. [7] work at interactive frame rates by using hardware rendering. For example, the Raskar and Cohen method draws backfacing polygons in black, slightly displaced toward the camera from the front-facing polygons, so that black borders appear at silhouettes. Such approaches limit stylization because by the time visibility has been calculated, the lines are already drawn.

To depict strokes with obvious character (e.g., texture, wobbles, varying width, deliberate breaks or dash patterns, tapered endcaps, overshoot, or haloes) Northrup and Markosian [2] introduced a simple rendering trick wherein the OpenGL lines are supplanted by textured triangle strips. The naive approach to computing visibility for such strokes would be to apply a z-buffer test to the triangle strips—a strategy that fails where the strokes interpenetrate the model (Fig. 2). Therefore, NPR methods utilizing this type of stylization generally have computed line visibility prior to rendering the lines. Line visibility has been the subject of research since the 1960s. Appel [8] introduced the notion of *quantitative invisibility* and computed it by finding changes in visibility at certain locations such as line junctions. This approach was further improved and adapted to NPR by Markosian et al. [1] who showed it could be performed at interactive frame rates for models of modest complexity.

Appel's algorithm and its variants can be difficult to implement and are somewhat brittle when faced with degenerate segments or overlapping vertices (i.e., when the lines are not in general position). Thus, Northrup and Markosian [2] adapted the use of an *item buffer* (which had previously been used to accelerate ray tracing [9]) for the purpose of line visibility, calling it an "ID reference image" in this context. Several subsequent NPR systems have adopted this approach, e.g., [10], [11], [12]. For an overview of line visibility approaches (especially with regard to silhouettes, which present a particular challenge because they lie at the cusp of visibility), see the survey by Isenberg et al. [13].

Any binary visibility test, including the item buffer approach, will lead to aliasing artifacts, analogous to those that appear for polygons when sampled into a pixel grid. To ameliorate aliasing artifacts, Cole and Finkelstein [3] adapted to lines the supersampling and depth-peeling strategies previous described for polygons, which we will revisit in Section 3.2.

While the item buffer approach can determine line visibility at interactive frame rates for scenes of moderate complexity, it is slow for large models. Moreover, computation of partial visibility—which significantly improves visual quality, especially under animation—imposes a further burden on frame rates. The two algorithms described in Sections 3 and 4 provide high-quality hidden

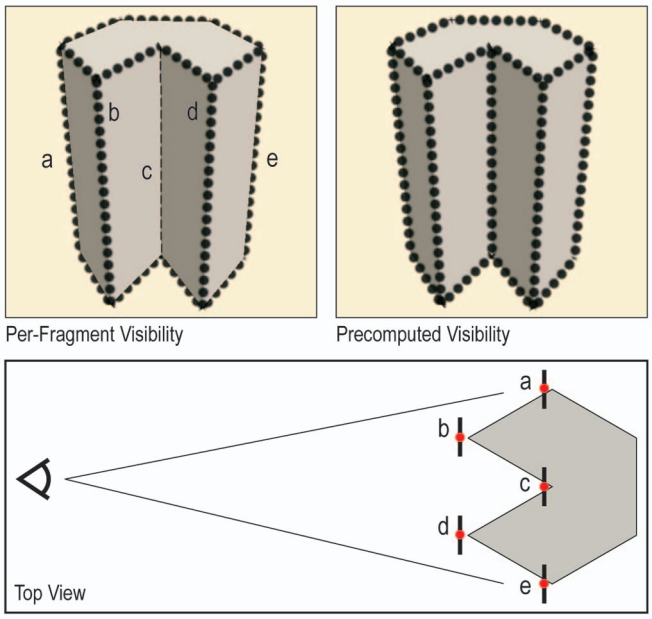

Fig. 2. Per-fragment visibility versus precomputed visibility. When drawing wide lines using a naive per-fragment visibility test, only lines that lie entirely outside the model will be drawn correctly (b and d). Lines a, c, and e are partially occluded by the model, even when some polygon offset is applied. Visibility testing along the spine of the lines (red dots) prior to rendering strokes solves the problem.

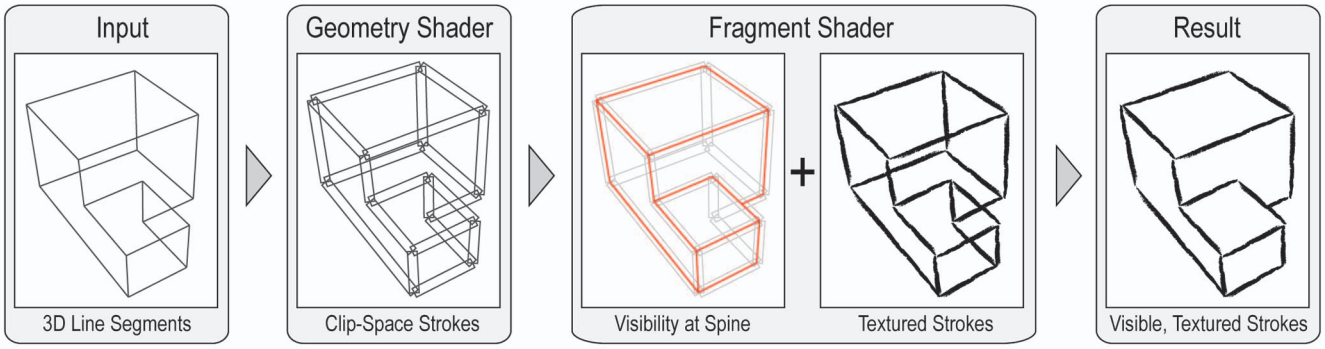

Fig. 3. Steps in the spine test method. The input is a set of 3D line segments. A geometry shader projects the line segments and creates clip-space strokes, preserving the homogenous positions for perspective-correct interpolation. A fragment shader checks visibility at the spine of the stroke and computes a texture color. The visibility and texture are combined to produce the final result.

line removal (with or without partial visibility) at interactive frame rates for complex models.

3 Method 1: Spine Test

Our first method is simple to implement and provides good quality in many cases. The method requires only a single pass to draw the depth buffer and a single pass to draw the lines, so it can be easily added to an existing line rendering implementation. However, the method does not support some important stylization options. In particular, because it generates an independent stroke for each line segment, it cannot properly parameterize stroke paths with multiple segments such as seen in Fig. 4; such paths require a continuous parameterization if they are to be rendered with texture. Nonetheless, many models (such as the Falling Water model in Figs. 6 and 11) have few curved stroke paths, and can thus be effectively rendered with this method.

The algorithm begins with a set of 3D line segments extracted from the model. Most of our experiments have focused on lines that are always drawn no matter the camera angle, for example, creases or texture boundaries. However, our system can also selectively draw edges that lie on silhouettes (e.g., the horizontal lines at the top of the clevis model shown on the left in Fig. 1) by checking the adjacent face normals during stroke generation.

The line segments are passed to the GPU using standard OpenGL drawing calls with the primitive-type GL_LINES. A geometry shader turns each line segment into a rectangular stroke and assigns texture coordinates to each vertex (Section 3.1). After the strokes are positioned and assigned texture coordinates, a fragment shader tests visibility at the nearest point on the spine of the stroke. As explained in Section 3.2, this visibility test can be a single depth probe or an average of many probes. Finally, the alpha value of each fragment is set to the visibility value of the spine. These steps are visualized in Fig. 3.

3.1 Stroke Generation

Newer graphics processors that support OpenGL 3.0 or the GL_EXT_geometry_shader4 extension (for example, NVIDIA's 8800 series) can execute geometry shaders, which are GPU programs that execute between the vertex and fragment stages and have the ability to add or remove vertices from a primitive. Geometry shaders are thus a natural choice for creating stroke geometry on the GPU. On hardware that does not support geometry shaders, it is also possible to generate strokes by creating a degenerate quad for each line segment and assigning the positions and texture coordinates in a vertex shader (similar to the approach of [14]). The vertex shader approach, however, requires additional vertices to be passed from the host to the GPU and additional software support on the CPU side when compared with the geometry shader approach.

In the spine test method, a geometry shader takes as input line segments and produces as output rectangles, represented as triangle strips. The shader also determines the screen-space length of the rectangle and assigns texture coordinates so that the stroke texture is scaled appropriately. The examples in this paper use 2D images of marks in the style of pen, pencil, charcoal, etc., and are parameterized at a constant rate in screen space. Graphics hardware by default uses perspective-correct texture interpolation, which tends to stretch and compress textures on strokes that are not perpendicular to the viewing direction. Uniform parameterization in screen space requires perspective-correct texturing to be disabled. Conveniently, control over perspective-correct interpolation is provided by the GL_EXT_gpu_shader4 extension, and by OpenGL 3.0.

To limit crawling artifacts, we use the simple strategy of fixing the "zero" parameter value at the screen-space center

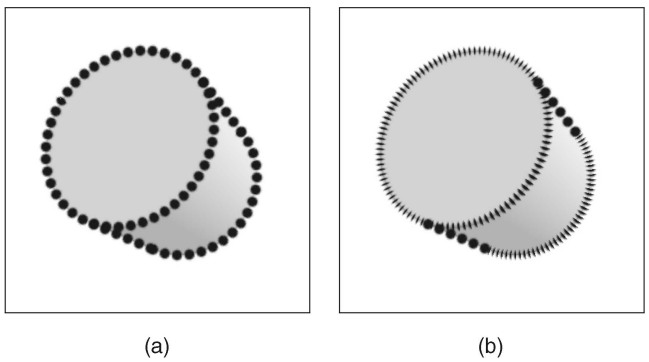

Fig. 4. Curved stroke paths. Strokes such as at the top and bottom of the cylinder consist of multiple segments. (a) The correct approach is to parameterize the entire loop as a single stroke. (b) Texturing each segment independently results in an incorrect result. Single-segment strokes as on the sides of the cylinder are not affected.

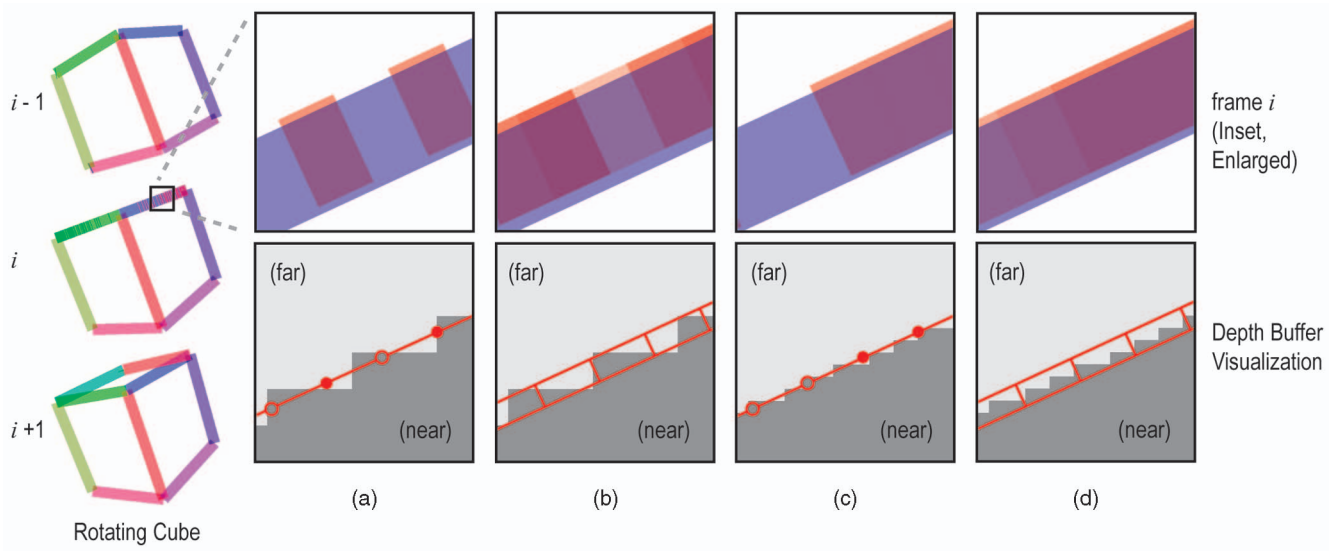

Fig. 5. Visibility aliasing. Aliasing in line visibility usually occurs at changes in occlusion. In this example, the red line is revealed behind the blue line as the cube rotates (left). (a) 1 sample, $1\times$ depth buffer: the artifacts, while transient, can be severe for a single visibility sample with a standard depth buffer. (b) many samples, $1\times$ depth buffer: multiple depth samples soften the artifacts. (c) 1 sample, $3\times$ depth buffer: supersampling the depth buffer without increasing the number of depth samples does not solve the problem, (d) but combining a supersampled depth buffer with multiple samples gives high-quality results: many samples, $3\times$ depth buffer. Top: enlargement showing partially occluded red line with blue line overlaid. Bottom: depth buffer visualization showing visibility samples for red line.

of the stroke. A more sophisticated strategy that seeks temporal coherence from frame to frame was described by Kalnins et al. [11].

While not a novel contribution of our method, we note that generating strokes in this manner makes it very easy to rapidly extract silhouette edges from smooth portions of a mesh, such as the rounded top of the clevis on the left in Fig. 1. The extraction is performed by sending all mesh edges to the GPU, then selecting the edges that lie on a silhouette boundary. To provide the necessary information to the GPU, neighboring face normals are packed into the vertex attributes for an edge prior to rendering the strokes. While generating a stroke for an edge, these face normals are checked for a silhouette condition (one front-facing and one back-facing polygon). If the edge is not a silhouette, it is discarded and no stroke is generated. The edge can be discarded directly by a geometry shader, or indirectly by a vertex shader by sending the vertices behind the camera.

Unfortunately, when drawing stroke paths with many segments, there is no way to know at the geometry shader level the proper parameterization of each segment, since each segment is processed independently and in parallel. It is, therefore, impossible to texture the entire path as one continuous stroke. This drawback is not very noticeable for models with many long, straight strokes, but is objectionable for models with many curving paths and short segments (Fig. 4). In contrast, the segment atlas method described in Section 4 supports computation of arc length and avoids this problem.

3.2 Visibility Testing

In order to perform depth testing at the spine of the stroke, the depth buffer must be drawn in a separate pass and loaded as a texture into the fragment shader. The visibility of a fragment is then computed by comparing the depth value of the closest point on the spine of the stroke with the depth value of the polygon under the spine, much like a conventional z-buffer scheme.

This simple approach commonly suffers from errors due to aliasing. There are two potential sources of aliasing: undersampling of the depth probes and polygon aliasing ("jaggies") in the depth buffer itself (both shown in Fig. 5). Aliasing errors occur at changes in line visibility, such as when a line is revealed by a sliding or rotating object. These errors manifest as broken or dashed lines. Broken lines may or may not be objectionable in still imagery, but under animation, the breaks move, causing popping and sparkling artifacts. Any individual line will only exhibit visibility artifacts from a small set of camera angles. However, complex models (such as shown in this paper) include so many lines that errors are very common (Fig. 6).

As noted by Cole and Finkelstein [3], aliasing can be alleviated by determining a *partial visibility* value for each line fragment. Conceptually, partial visibility can be computed by replacing the line (which has zero width) with a narrow quadrilateral, then computing the conventional α (occlusion) value for that quadrilateral. In our case, partial visibility is determined by making multiple depth probes in a box filter configuration around the line sample (Fig. 5b). Additional depth probes are usually very fast (Section 5), but can become expensive on limited hardware.

Any number of depth probes will not produce an accurate result if the underlying depth buffer has aliasing error (Fig. 5b). While impossible to eliminate entirely, this source of aliasing can be reduced through supersampling of the depth buffer by increasing the viewport resolution. Simply scaling the depth buffer without adding additional depth probes for each sample produces a marginal increase in image quality (Fig. 5c), but combining depth buffer scaling and depth test supersampling largely eliminates aliasing artifacts (Fig. 5d). Since typical applications are seldom fill rate bound for simple operations like drawing

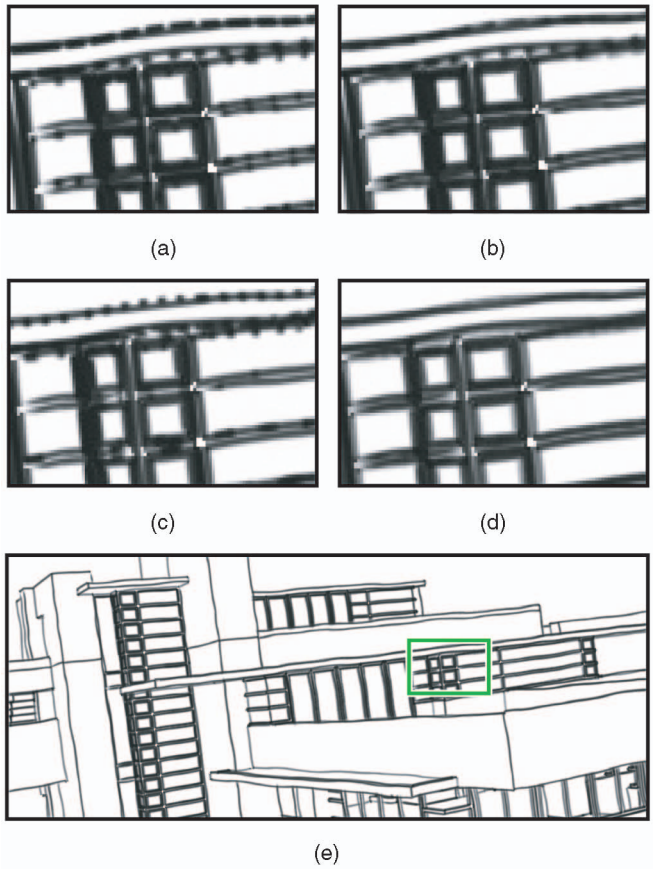

Fig. 6. Aliasing in visibility test. Results for varying number of samples and scale of depth buffer. Green box in (e) indicates location of magnified area. Visibility supersampling is used in both the spine test and segment atlas methods. (a) 1 sample, $1\times$ depth buffer. (b) 16 samples, $1\times$ depth buffer. (c) 1 sample, $3\times$ depth buffer. (d) 16 samples, $3\times$ depth buffer. (e) Falling water.

the depth buffer, increasing the size of the buffer typically has little impact on the performance outside of an increase in memory usage. Results of these techniques for a complex model can be seen in Fig. 6.

4 Method 2: Segment Atlas

Stylization for curved strokes, or even simple effects such as endcaps or haloes, require some nonlocal information. For example, each segment in a curved stroke must have texture coordinates based on the entire arc length of the stroke. This information is costly to compute with a single-pass approach such as the spine test, because much of the computation is redundant across segments. The same observation holds for endcaps or haloes: while in principle, each fragment could check a large neighborhood to determine the closest visibility discontinuity, it is much more efficient to store the visibility in a separate pass. Additional effects that can be achieved by precomputing visibility are explained in Section 4.5.

The segment atlas approach is designed to efficiently compute and store the visibility information for every stroke in the scene. The input includes 3D line segments, as with the spine test method, but also line strips (stroke paths). The output is a segment atlas containing visibility samples for each projected and clipped stroke, spaced by a constant screen-space distance (usually 2 pixels).

The pipeline has four major stages, illustrated in Fig. 7: line projection and clipping, computation of atlas offsets, drawing the segment atlas and testing visibility, and stroke rendering. All stages execute on the GPU, and all data required for execution reside in GPU memory in the form of OpenGL framebuffer objects or vertex buffer objects.

4.1 Projection and Clipping

The first stage of the pipeline begins with a set of candidate line segments, projects them, and clips them to the viewing frustum. Ideally, we would use the GPU's clipping hardware to clip each segment. However, in current graphics hardware, the output of the clipper is not available until the fragment program stage after rasterization has already been performed. We, therefore, must use a fragment program to project and clip the segments, using our own clipping code. The fragment program uses the same camera and projection matrices as the conventional projection and clipping pipeline.

The input to the program is a six-channel framebuffer object packed with the world-space 3D coordinates of the endpoints of each segment $(\mathbf{p}, \mathbf{q})$ (Fig. 7, step 1). In our implementation, this buffer must be updated at each frame with the positions of any moving line segments. However, the fragment program could also be modified to transform the segments with a time-varying matrix. The output of the fragment program is a nine-channel buffer containing the 4D homogeneous clip coordinates $(\mathbf{p}', \mathbf{q}')$ and the number of visibility samples l. The number of visibility samples l is defined as:

$$l = \lceil \|\mathbf{p}'_w - \mathbf{q}'_w\|/k \rceil, \tag{1}$$

where $(\mathbf{p}'_w, \mathbf{q}'_w)$ are the 2D window coordinates of the segment endpoints, and k is a screen-space sampling rate. The factor k trades off positional accuracy in the visibility test against segment atlas size. We usually set $k = 1$ or 2, meaning that visibility is determined every 1 or 2 pixels along each line; there is diminishing benefit in determining with any greater accuracy the exact position at which a line becomes occluded.

A value of $l = 0$ is returned for segments that are entirely outside the viewing frustum. Segments for which $l \leq 1$ (i.e., subpixel-sized segments) are discarded for efficiency if not part of a path, but otherwise must be kept or the path will appear disconnected.

In a separate step, the sample counts l are converted into segment atlas offsets s by computing a running sum (Fig. 7, step 2). The sum is calculated by an exclusive-scan operation on l [15]. Once the atlas offsets s are computed, each segment may be drawn in the atlas independently and without overlap.

If the system must handle multisegment paths, the segment table may also include two extra channels to store the offsets of each segment from the start and end of its path. By comparing these pointers, a stand-alone segment can be distinguished from a segment that is part of a path. This information may be used during the final stroke rendering step to smoothly connect adjacent segments of multisegment paths (Section 4.4).

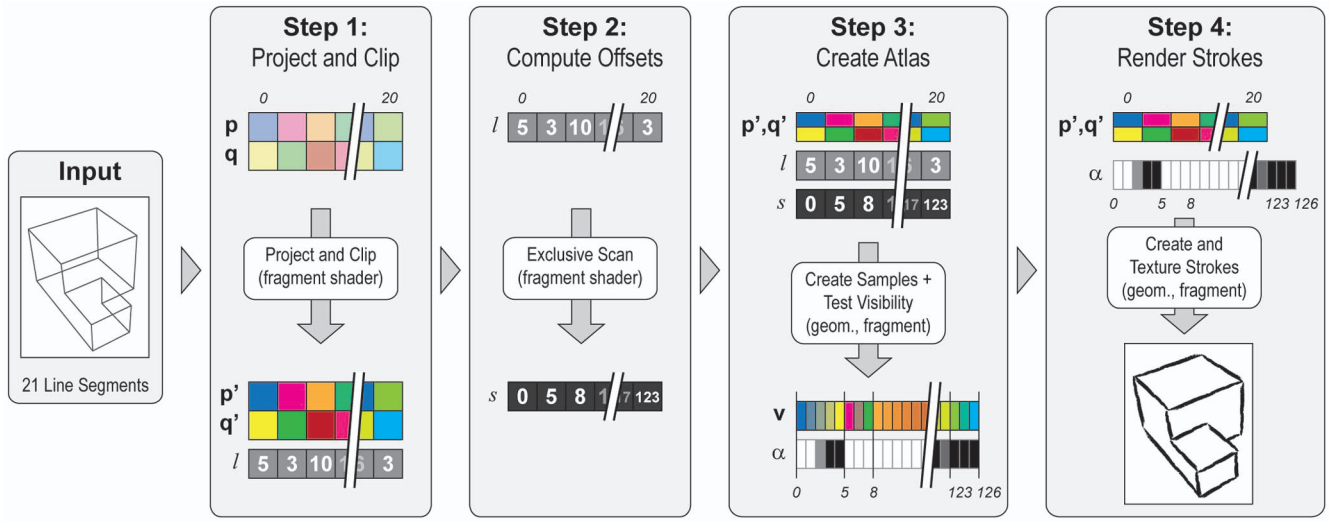

Fig. 7. Segment atlas pipeline. The input 3D line segments $(\mathbf{p}_i, \mathbf{q}_i)$ are stored in a table on the GPU. At each frame, each 3D segment is projected and clipped by a fragment shader, which also determines a number of samples l_i proportional to screen-space length (step 1). Next, a scan operation computes the atlas offsets s from the running sum of l (step 2). The sample positions $\mathbf{v}$ are then created by interpolating $(\mathbf{p}', \mathbf{q}')$ and writing to the segment atlas at offset s (step 3). Visibility values α_j are determined by probing the depth buffer at each $\mathbf{v}_j$ (see Fig. 9). Finally, strokes are created at $(\mathbf{p}', \mathbf{q}')$ and textured with the visibility values α to produce the final rendering. Note the colors used throughout to identify individual segments.

Finally, silhouette edges may also be extracted during the projection and clipping stage by loading face normals alongside the vertex world coordinates and checking for a silhouette edge condition at each segment. If the edge is not a silhouette, it is discarded by setting $l = 0$. This method is similar to the approach of Brabec and Seidel [16] for computing shadow volumes on the GPU. Note, however, that our current method is unable to stitch these silhouette edges into continuous multisegment silhouette paths, e.g., the outline of a sphere. The parameterization of multisegment paths is computed by the exclusive-scan operation, which assumes that the segment indexes are neighboring and constantly increasing. The segments of a silhouette path, by contrast, are in effectively random order in the segment table. In addition, silhouette paths based on polygon edges can include degeneracies (see [17]). Continuous parameterization of silhouette paths on the GPU is, therefore, an area for future work.

4.2 Segment Atlas Creation

The purpose of the segment atlas is to store the visibility samples for every segment in the scene. The ith segment is allocated l_i visibility samples, or entries, in the atlas (for example, segment 2 might be 5 pixels long and be assigned three entries, while segment 3 might be 20 pixels long and be assigned 10 entries). Each set of entries begins at the segment atlas offset s_i. Each entry consists of a 3D screen-space sample position $\mathbf{v}$ and a visibility value α. While storing the sample position $\mathbf{v}$ is unnecessary after visibility has been computed, current GPUs commonly support only four-channel textures, and the visibility values require only a single channel. On future hardware, storing only the visibility values α would save GPU memory.

To compute the screen-space positions $\mathbf{v}$ of the samples, we make use of the rasterization hardware of the GPU. We set up the segment atlas as a rendering target (e.g., an OpenGL framebuffer object) and draw single-pixel wide lines (*proxy lines*) into the atlas, as follows: The host passes one vertex to the GPU, identified by an index i, for each segment. A geometry shader then looks up the ith entry in the projected and clipped segment table, and produces two vertices for a single proxy line segment. If the hardware does not support geometry shaders, the host must pass two vertices to a vertex shader, each identified with index i and a binary "start vertex/end vertex" flag. The shader then positions the vertex at either the beginning or the end of the proxy segment, depending on the flag.

In either case, the proxy segment i begins in the atlas at position s_i and is l_i pixels long. The color of the first vertex of the proxy is set to the clip-space position $\mathbf{p}'_i$, and the color of the second vertex is set to $\mathbf{q}'_i$. When the proxy lines are drawn, the rasterization hardware performs the interpolation of the clip-space positions. A fragment shader then performs the perspective division and viewport transformation steps to produce the screen-space coordinate $\mathbf{v}$ (Fig. 7, step 3). At the same time, the fragment shader checks the visibility of the sample as described in Section 4.3. The final output of the fragment shader is the interpolated position $\mathbf{v}$ and the visibility value α.

The most natural representation for the segment atlas is a very long, 1D texture. Unfortunately, current GPUs do not allow for arbitrarily long 1D textures as targets for rendering. The segment atlas must, therefore, be mapped to two dimensions (Fig. 8). This mapping can be achieved by wrapping the atlas positions at a predetermined width w, usually the maximum texture width W allowed by the GPU ($W = 4{,}096$ or 8,192 texels is common). The 2D atlas positions $\mathbf{s}$ are given by

$$\mathbf{s} = (s \bmod w, \lfloor s/w \rfloor). \tag{2}$$

The issue then becomes how to deal with segments that extend outside the texture, i.e., segments for which $(s \bmod w) + l > w$. One way to address this problem is to draw the segment atlas twice, once normally and once with

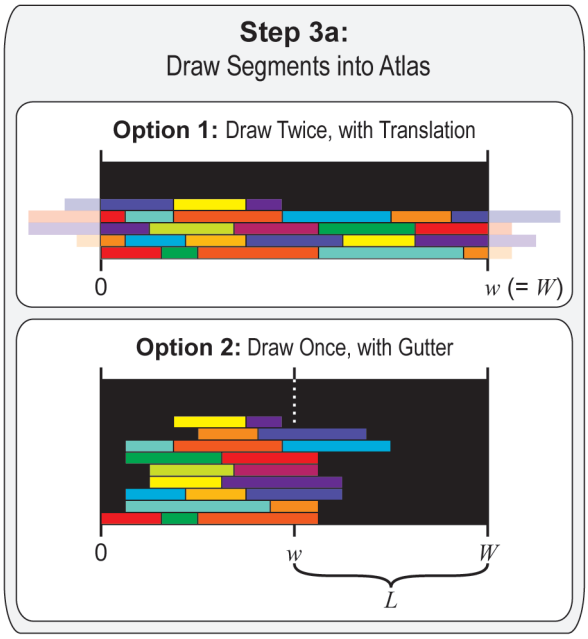

Fig. 8. Segment atlas wrapping. Because current generation GPUs do not support arbitrarily long 1D textures, the segment atlas must be wrapped to fit in a 2D texture. One option is to draw the atlas twice, wrapping segments that fall outside the width w (shown faded). Another option is to establish a gutter of size L to catch segments that fall outside w. Here, W is the maximum texture width and L is the maximum segment length.

the projection matrix translated by $(-w, 1)$. Long segments will thus be wrapped across two consecutive lines in the atlas (Fig. 8 top). Specifically, suppose L is the largest value of l, which can be conservatively capped at the screen diagonal distance divided by k. If $w > L$, drawing the atlas twice is sufficient, because we are guaranteed that each segment requires at most one wrap. Drawing twice incurs a performance penalty, but as the visibility fragment program is usually the bottleneck (and is still run only once per sample), the penalty is usually small.

For some rendering applications, however, it is considerably more convenient if segments do not wrap (Section 4.4). In this case, we establish a gutter in the 2D segment atlas by setting $w = W - L$. The atlas position is then only drawn once (Fig. 8 bottom). This approach is guaranteed to waste $W - L$ texels per atlas line. Moreover, this waste exacerbates the waste due to our need to preallocate a large block of memory for the segment atlas without knowing how full it will become. Nevertheless, the memory usage of the segment atlas (which is limited by the number of lines drawn on the screen) is typically dominated by that of the 3D and 4D segment tables (which must hold all lines in the scene).

4.3 Visibility Computation

As mentioned in Section 4.2, the visibility test for each sample is performed during rasterization of the segments into the segment atlas. While drawing the atlas, a fragment program computes an interpolated homogeneous clip-space coordinate for each sample and performs the perspective division step. The resulting clip-space z value is then compared to a depth buffer (Fig. 9).

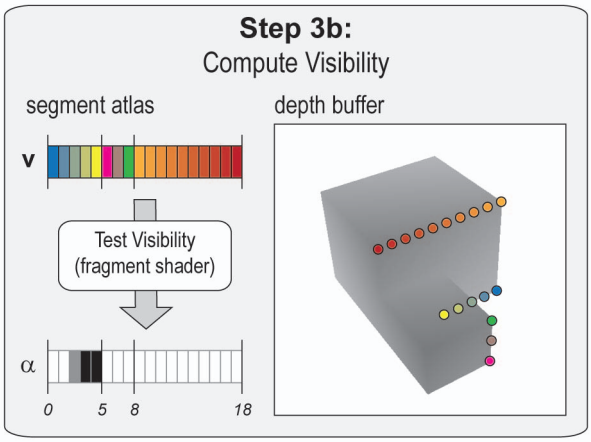

Fig. 9. Visibility testing. The first three segments in Fig. 7 are shown. Each sample in the segment atlas corresponds to a fragment. The fragment shader uses the screen-space position $\mathbf{v}_j$ to test the sample against the depth buffer, recording the result in the visibility value α_j. Colors are the same as in Fig. 7.

The visibility test itself is similar to the test for the spine test approach, with the same configuration of multiple depth probes and supersampled depth buffer. Since visibility is only tested once per spine sample, however, rather than once for every fragment along the width of the stroke, even more depth probes can be efficiently computed.

4.4 Stroke Rendering

After visibility is computed, all the information necessary to draw strokes is available in the projected and clipped segment table and the segment atlas. The most efficient way to render the strokes is to generate, on the host, a single point per segment. A geometry shader then uses the point as an index and looks up the appropriate $(\mathbf{p}', \mathbf{q}')$ in the projected and clipped segment table. The segment endpoints may also be looked up in the segment atlas, if the positions $\mathbf{v}$ are stored in the atlas. However, we find that the original segment table is more convenient since both vertex positions are stored at the same texture offset in different channels. The geometry shader then emits a quad that lies between the segment endpoints, with width determined by the pen style.

As with the spine test method, hardware without geometry shaders can generate the same quads, albeit less efficiently, by generating a degenerate quad on the host and positioning the four vertices in a vertex shader (again, similar to [14]).

Lastly, a fragment shader textures the quad with a 2D pen texture and modulates the texture with the corresponding 1D visibility values from the segment atlas. A range of effects can be achieved by varying the pen texture and color with visibility (Figs. 10 and 13).

4.5 Additional Effects

By storing the visibility and screen-space positions simultaneously for all strokes in the scene, the segment atlas method allows a range of additional rendering effects not possible with the spine test method. Some examples include:

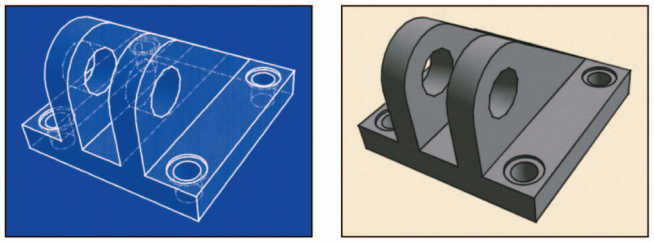

Fig. 10. Variation in style. (a) A different texture may be used for lines that fail the visibility test, allowing visualization of hidden structures. (b) Our method also produces attractive results for solid, simple styles.

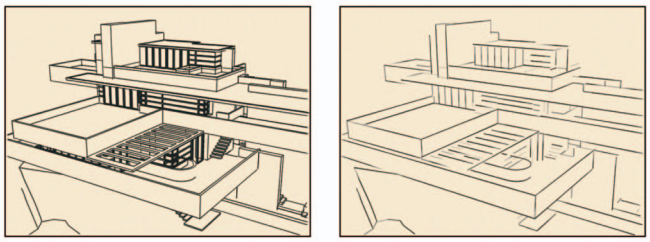

Fig. 11. Line density control. The segment atlas can store information besides visibility, such as local line density. (a) No density control. (b) Line density reduction as described in [12].

4.5.1 Mitering

In order to render multisegment strokes without visible gaps or overlap between segments, the ends of adjacent segments must be smoothly connected (*mitered*). Proper mitering of segment i requires the positions and orientations of segments $i-1$ and $i+1$ (if they exist). This information can be looked up in the projected and clipped segment table (see Section 4.1). Corner mitering (joining with a sharp corner) can be performed in the final rendering step by either a geometry or vertex shader, simply by adjusting the four vertices of each segment quad. While not implemented in this work, smooth mitering (joining with a rounded corner) should also be possible by emitting extra vertices from a geometry shader.

4.5.2 Filtering

The segment atlas also provides the opportunity to filter the visibility information to fill small holes or remove short, spurious sections. Other image processing operations can be performed on the atlas as well. For example, erosion and dilation can produce line overshoot or undershoot (haloing) effects (Fig. 1). A convincing sketchy overshoot effect can be achieved by setting the dilation amount to a constant screen-space length, then modulating this length pseudorandomly with the path index (or index of the starting segment of the path) to vary the size of the overshoot. For operations such as dilation, it is necessary to add padding around each segment in the atlas so that the segment can dilate beyond its normal length. Padding can be added easily by increasing the number of samples when computing the atlas offset (Section 4.2).

4.5.3 Density Control

The segment atlas can also be used to store any type of per-sample information, not just visibility. For example, it can store a measure of the density of lines in the local area, as produced by a stroke-based line density control scheme [18], [12]. Results from the system described in [12], as implemented using a segment atlas, are shown in Fig. 11.

4.6 Readback

For applications that are difficult to implement entirely on the GPU, such as stroke simplification [19] or complex NPR shaders [20], the segment atlas can be read back to the host. Reading back and processing the entire segment atlas is inefficient, however, because for reasonably complex models, the vast majority of line samples in any given frame will have zero visibility. We can reduce this cost by applying a stream compaction operation [21] to the segment atlas visibility values. This operation yields a packed buffer with only visible samples remaining. For models of moderate complexity, compaction and readback adds an additional cost of $\sim$20 ms per frame.

5 Results

We implemented the two methods using OpenGL and GLSL, taking care to manage GPU-side memory operations efficiently. For comparison, we also implemented an optimized conventional OpenGL rendering pipeline using line primitives, the item buffer approach of Northrup and Markosian [2], and the improved item buffer approach of Cole and Finkelstein [3]. We did not use NVIDIA's CUDA architecture, because the segment atlas drawing step uses conventional line rasterization and the rasterization hardware is unavailable from CUDA.

Table 1 shows frame rates for four models ranging from 1k-500k line segments. The clevis, house (Falling Water), ship, and office models are shown in Figs. 10, 11, and 12. The "+s" indicates that silhouettes were extracted and drawn in addition to the fixed lines. Timings for clevis and house are averaged over an orbit of the model, whereas timings for the ship and office are averaged over a walk-through sequence. All frames are rendered at $1{,}024 \times 768$ using a commodity Dell PC running Windows XP with an Intel Core 2 Duo 2.4 GHz CPU and 2 GB RAM, and an NVIDIA 8800GTS GPU with 512 MB RAM.

TABLE 1
Frame Rates (FPS) for Various Models and Methods

Model	clevis	house	ship	office	ship+s	off.+s
# tris	1k	15k	300k	330k	-	-
# seg	1.5k	14k	300k	300k	500k	400k
OGL	1000+	300+	42	32	-	-
IBlo	87	24	9.6	7.0	-	-
IBhi	20	3.4	0.5	0.4	-	-
STlo	900+	146	26	28	19	23
SThi	300+	75	24	25	19	21
SAlo	400+	119	33	29	23	24
SAhi	200+	76	25	24	22	21

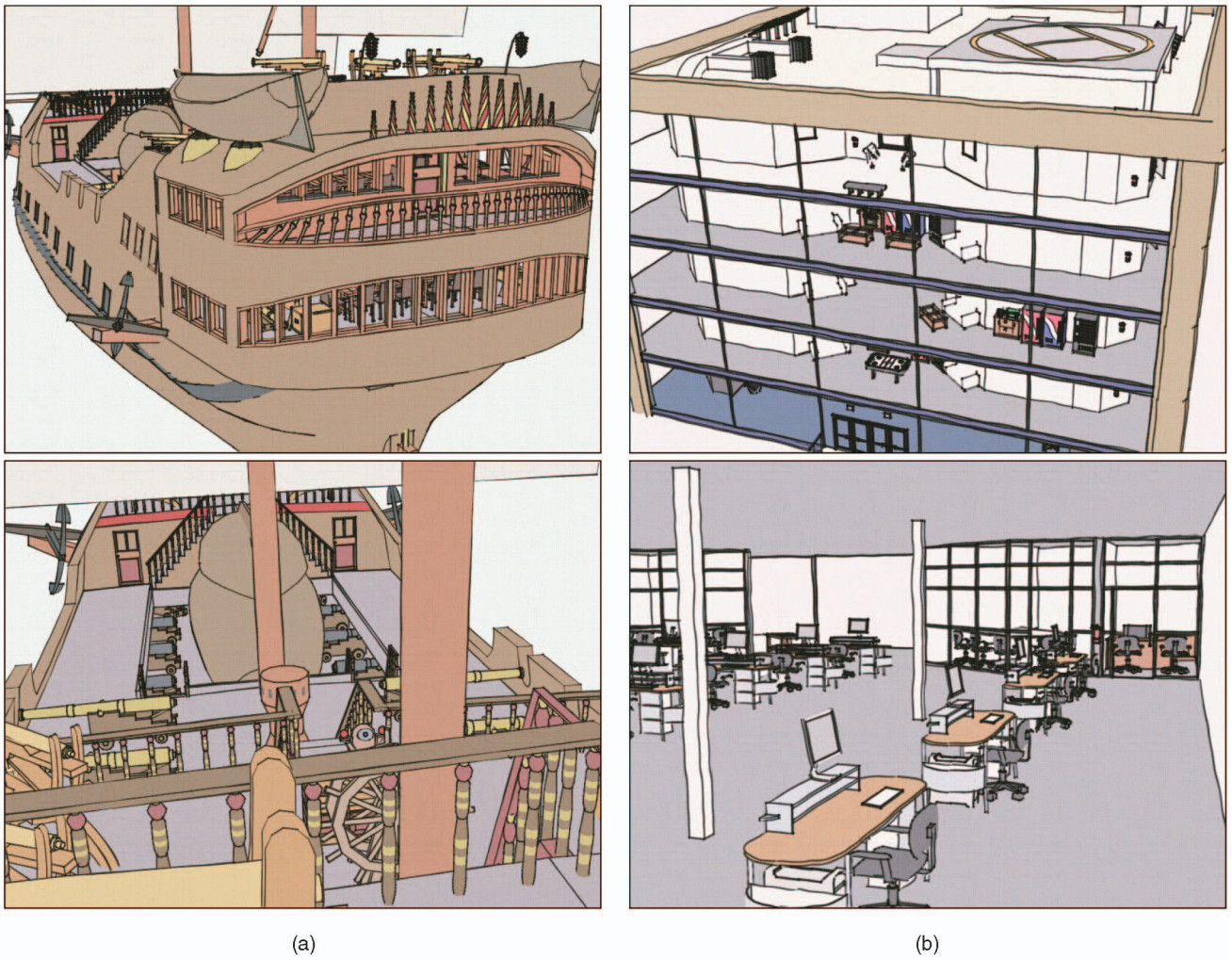

Fig. 12. Complex models. (a) The ship model has 300k triangles and 500k total line segments. (b) The office model has five levels, each with detailed furniture, totaling 330k triangles and 400k line segments. Both models can be rendered at high-quality and interactive frame rates using both the spine test and segment atlas methods.

We tested the following rendering algorithms: (OGL) conventional OpenGL lines; (IBlo) single item buffer [Northrup2000]; (IBhi) $9\times$ supersampled item buffer with three layers [Cole2008]; (STlo/SAlo) spine test shader and segment atlas, respectively, with a single depth probe, which is comparable to IBlo; and (SThi/SAhi) spine test shader and segment atlas, respectively, with nine depth probes and $2\times$ scaled depth buffer, which is comparable to IBhi. For small models (clevis and house), both the spine test and segment atlas methods are slower than conventional OpenGL rendering by factors of 2-4×, though overall speed is still high. Additional samples and depth buffer scaling also incur a noticeable penalty for these models. For the more complex models (ship and office), the penalty for using either method declines. Both methods are within 50 percent of conventional OpenGL in the high-quality modes (SThi/SAhi). The basic segment atlas (SAlo), which suffers from some aliasing artifacts but still provides good quality, is within 75 percent of OpenGL on both the office and ship models.

Both of the new methods are always considerably faster than the item buffer-based approach, but the most striking difference is when comparing the high-quality modes of each method. The item buffer approach with $9\times$ supersampling and three layers, as suggested by [3], gives similar image quality to our methods with nine depth probes and $2\times$ scaled depth buffer. The new methods, however, deliver performance increases of up to $50\times$ for complex models.

As mentioned in Sections 3.1 and 4.1, both methods allow for easy extraction and rendering of silhouette edges on the GPU. The last two rows of Table 1 show the performance impact when extracting and rendering silhouettes. The increase in cost is roughly proportional to the increase in the total number of potential line segments. We did not implement silhouette extraction for the other methods. However, silhouette extraction can be a costly operation when performed on the CPU.

While accurate timing of the stages of our method is difficult due to the deep OpenGL pipeline, the major costs ($\sim$80-90 percent of total) lie in the sample visibility testing stage and depth buffer drawing stage. For small models, the sample visibility testing is dominant, while for large models, the depth buffer creation is the primary single cost. Projection, clipping, and stroke rendering are minor costs.

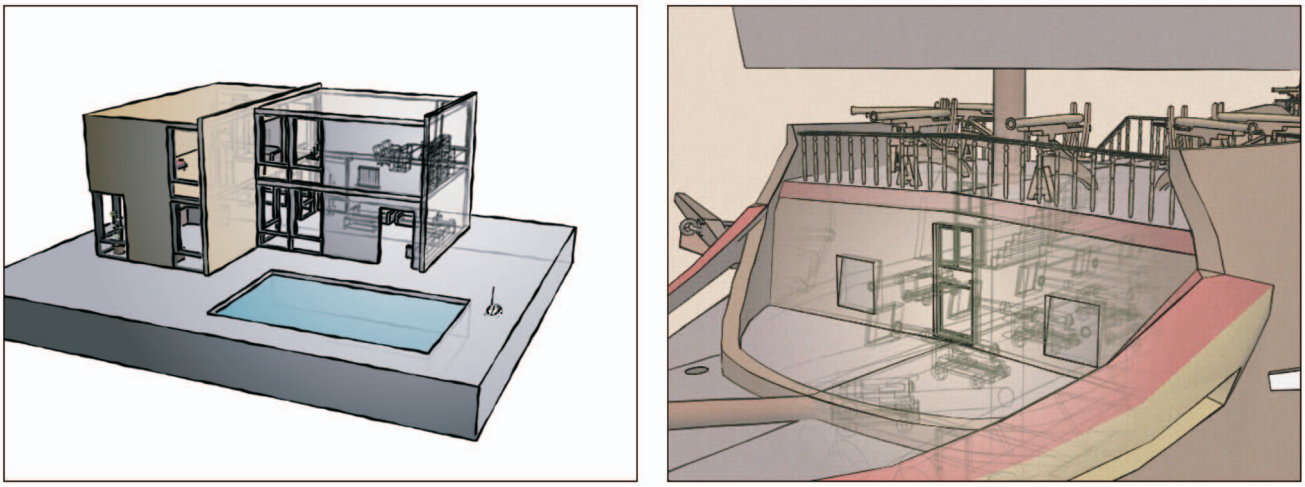

Fig. 13. Drawing hidden lines using the segment atlas. Locally controlling line visibility, using the stylized focus technique of [12], can reveal the internal structure of a model while providing context or hiding unimportant areas. Because the segment atlas stores visibility information for all strokes, hidden and visible lines can be drawn with no extra cost to performance.

6 Conclusion and Future Work

The proposed methods allow rendering of high-quality stylized lines at speeds approaching those of the conventional OpenGL rendering pipeline. They provide improved temporal coherence and less aliasing (sparkle) than previous approaches for drawing stylized lines, making them suitable for animation of complex scenes. The spine test shader (method 1) is particularly simple, and should be easy to include in existing line rendering systems. The segment atlas pipeline (method 2), while more complex, is still fairly easy to implement, and provides a broader range of stylization options. Compared with previous approaches for computing line visibility, both are robust and conceptually simple. We believe that these approaches will be useful for interactive applications such as games and design and modeling software, where previously, the performance penalty for using stylized lines has been prohibitive.

The ability to **store full visibility** information for all lines allows for special rendering of hidden lines (Fig. 13), but also opens several possibilities for future work. Just as Cole et al. [12] introduced "stylized focus" as an artistic effect inspired by photorealistic defocus effects, we can imagine a "stylized motion blur" effect inspired by photorealistic motion blur. By storing the segment atlases from previous frames, we could blur the *visibility values* from consecutive frames rather than the final rendered strokes. Blurring visibility could, for example, allow a disappearing stroke to break up into shrinking splotches of ink, rather than simply fading out.

Storing copies of the segment atlas from previous frames could also allow for performance increases in situations where computing the atlas samples is a significant cost. Rather than recomputing each sample from scratch at each frame, the sample positions could be reprojected from frame to frame and fully refreshed intermittently. Reprojection would distort the sampling rate of each line and introduce errors for clipped lines, but may be worthwhile in some applications.

Other future work in this area may include adapting **line density control** methods such as proposed in [18], [12] to operate more effectively on the GPU. Our current implementation of [12] exhibits some sparkling artifacts under animation and causes a hit in the performance. One challenge is that these approaches do not take into account partial visibility of lines, which is necessary for smooth animation.

While not a direct extension of our method, we would also like it to handle other **view-dependent lines** such as smooth silhouettes [17], suggestive contours [22], and apparent ridges [23]. Including these line types at a reasonable performance cost may require an extraction algorithm that executes on the GPU. In contrast to lines that are fixed on the model, consistent parameterization of such lines from frame to frame presents its own challenge [11].

Acknowledgments

The authors would like to thank the editors and reviewers for their comments and assistance in revising the paper, and Michael Burns and the Google 3D Warehouse for the example models. This work was sponsored in part by the US National Science Foundation (NSF) grant IIS-0511965.

References

[1] L. Markosian, M.A. Kowalski, D. Goldstein, S.J. Trychin, J.F. Hughes, and L.D. Bourdev, "Real-Time Nonphotorealistic Rendering," *Proc. ACM SIGGRAPH '97*, pp. 415-420, 1997.

[2] J.D. Northrup and L. Markosian, "Artistic Silhouettes: A Hybrid Approach," *Proc. Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '00)*, pp. 31-37, June 2000.

[3] F. Cole and A. Finkelstein, "Partial Visibility for Stylized Lines," *Proc. Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '08)*, pp. 9-13, June 2008.

[4] F. Co1e and A. Finkelstein, "Fast High-Quality Line Visibility," *Proc. Symp. Interactive 3D Graphics (I3D '09)*, pp. 115-120, Feb. 2009.

[5] X. Wu, "An Efficient Antialiasing Technique," *Proc. ACM SIGGRAPH '91*, pp. 143-152, 1991.

[6] R. Raskar and M. Cohen, "Image Precision Silhouette Edges," *Proc. Symp. Interactive 3D Graphics (SI3D '99)*, pp. 135-140, 1999.

[7] Y. Lee, L. Markosian, S. Lee, and J.F. Hughes, "Line Drawings via Abstracted Shading," *ACM Trans. Graphics*, vol. 26, no. 3, pp. 18:1-18:5, July 2007.

[8] A. Appel, "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proc. 22nd Nat'l Conf. ACM*, pp. 387-393, 1967.

[9] H. Weghorst, G. Hooper, and D.P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. Graphics,* vol. 3, no. 1 pp. 52-69, Jan. 1984.
[10] R.D. Kalnins, L. Markosian, B.J. Meier, M.A. Kowalski, J.C. Lee, P.L. Davidson, M. Webb, J.F. Hughes, and A. Finkelstein, "WYSIWYG NPR: Drawing Strokes Directly on 3D Models," *Proc. ACM SIGGRAPH '02,* pp. 755-762, 2002.
[11] R.D. Kalnins, P.L. Davidson, L. Markosian, and A. Finkelstein, "Coherent Stylized Silhouettes," *ACM Trans. Graphics,* vol. 22, no. 3 pp. 856-861, July 2003.
[12] F. Cole, D. DeCarlo, A. Finkelstein, K. Kin, K. Morley, and A. Santella, "Directing Gaze in 3D Models with Stylized Focus," *Proc. Eurographics Symp. Rendering '06,* pp. 377-387, June 2006.
[13] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, "A Developer's Guide to Silhouette Algorithms for Polygonal Models," *IEEE Computer Graphics and Applications,* vol. 23, no. 4, pp. 28-37, July/Aug. 2003.
[14] M. McGuire and J.F. Hughes, "Hardware-Determined Feature Edges," *Proc. Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '04),* pp. 35-47, 2004.
[15] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan Primitives for GPU Computing," *Proc. Graphics Hardware '07,* pp. 97-106, 2007.
[16] S. Brabec and H.-P. Seidel, "Shadow Volumes on Programmable Graphics Hardware," *Proc. EUROGRAPHICS '03,* vol. 22, pp. 433-440, Sept. 2003.
[17] A. Hertzmann and D. Zorin, "Illustrating Smooth Surfaces," *Proc. ACM SIGGRAPH '00,* pp. 517-526, 2000.
[18] S. Grabli, F. Durand, and F. Sillion, "Density Measure for Line-Drawing Simplification," *Proc. Pacific Graphics,* pp. 309-318, 2004.
[19] P. Barla, J. Thollot, and F. Sillion, "Geometric Clustering for Line Drawing Simplification," *Proc. Eurographics Symp. Rendering '05,* pp. 183-192, 2005.
[20] S. Grabli, E. Turquin, F. Durand, and F. Sillion, "Programmable Style for NPR Line Drawing," *Proc. Eurographics Symp. Rendering '04,* pp. 33-44, 2004.
[21] D. Horn, "Stream Reduction Operations for GPGPU Applications," *GPU Gems 2,* M. Pharr, ed., ch. 36, pp. 573-589, Addison Wesley, 2005.
[22] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive Contours for Conveying Shape," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 848-855, 2003.
[23] T. Judd, F. Durand, and E.H. Adelson, "Apparent Ridges for Line Drawing," *ACM Trans. Graphics,* vol. 26, no. 3, 2007.

Forrester Cole received the PhD degree in computer science from Princeton University in June 2009. His research interests include perception of abstract imagery and interactive methods for nonphotorealistic rendering. He studied computer science at Harvard College and graduated in 2002. He is currently a postdoctoral researcher in the Computer Graphics Group at the Massachusetts Institute of Technology.

Adam Finkelstein received the PhD degree from the University of Washington in 1996. He is an associate professor of computer science at Princeton University. His research interests in computer graphics include nonphotorealistic rendering, multiresolution techniques, animation, and applications of computer graphics in art. He is also one of the organizers of the Art of Science Exhibition. From 1987 to 1990, he was a software engineer at Tibco, where he wrote software for people who trade stock. He was an undergraduate student at Swarthmore College (class of 1987) where he studied physics and computer science.