Fast High-Quality Line Visibility

Forrester Cole
Princeton University

Adam Finkelstein
Princeton University

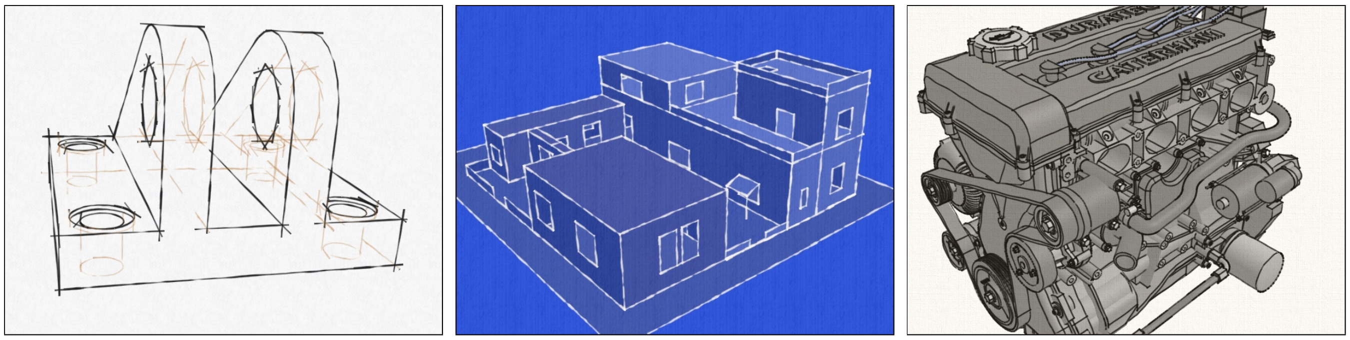

Figure 1: *Examples of models rendered with stylized lines.* Stylized lines can provide extra information with texture and shape, and are more aesthetically appealing than conventional solid or stippled lines.

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

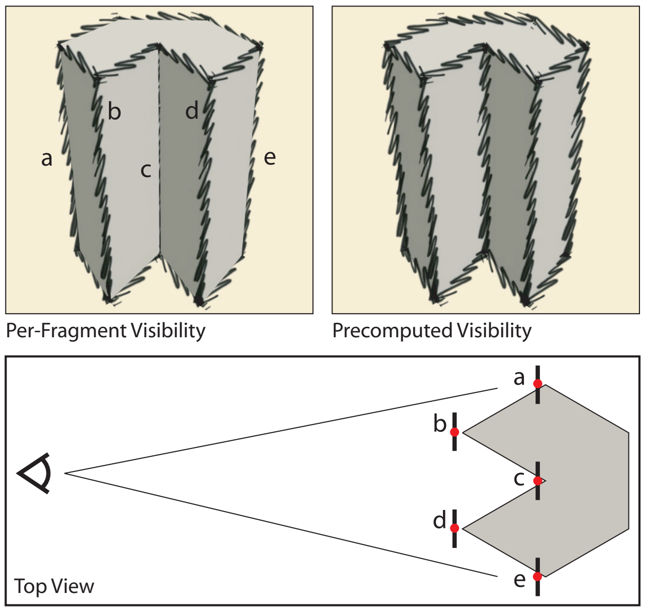

Figure 2: *Naive depth testing per-fragment vs. precomputed visibility.* When drawing wide lines, only lines that lie entirely outside the model will be drawn correctly (b and d). Lines a, c, e are partially occluded by the model, even when some polygon offset is applied. Visibility testing along the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

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

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.

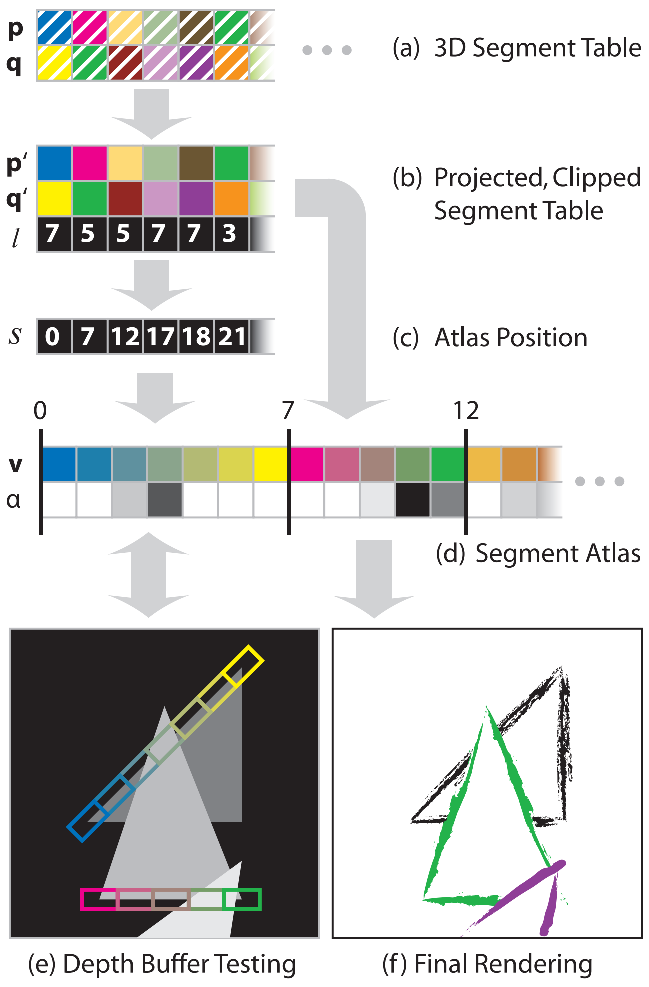

Figure 3: *Pipeline.* (a) The 3D line segments $(\mathbf{p}_i, \mathbf{q}_i)$ are stored in a table. (b) A fragment program projects and clips each segment to produce $(\mathbf{p}'_i, \mathbf{q}'_i)$, and determines a number of samples l_i proportional to its screen space length. (c) A scan operation 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

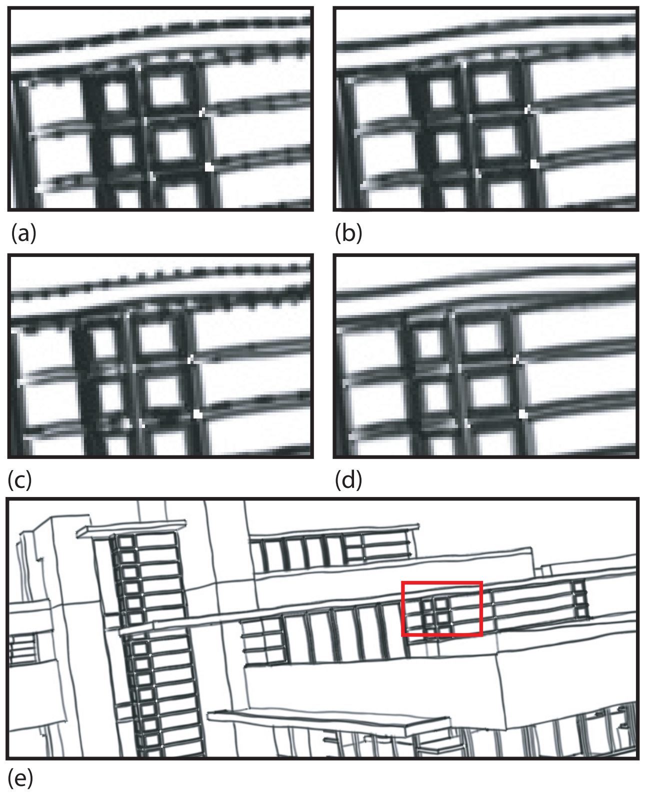

Figure 4: *Aliasing in visibility test.* Results for varying number of samples and scale of depth buffer. (a) 1 sample, 1x depth buffer. (b) 16 samples, 1x depth buffer. (c) 1 sample, 3x depth buffer. (d) 16 samples, 3x depth buffer. Red box in (e) indicates location of magnified area in the Falling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

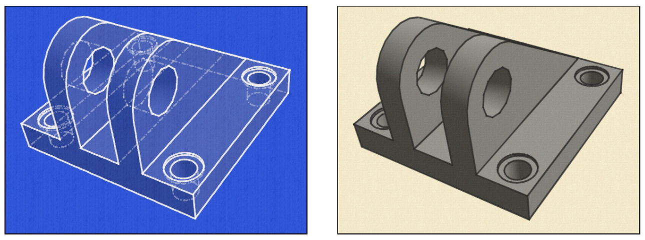

Figure 5: *Variation in style.* A different texture may be used for lines that fail the visibility test (*left*), allowing visualization of hidden structures. Our method also produces attractive results for solid, simple styles (*right*).

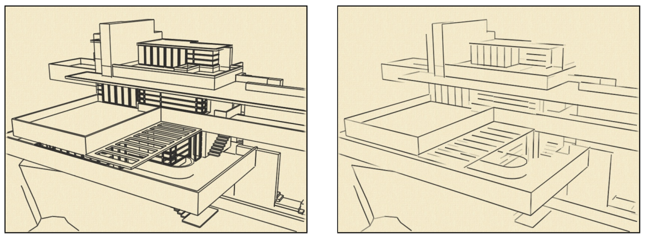

Figure 6: *Line density control.* One reason to read back the segment atlas to the host is to control screen space line density. Left: no density control. Right: line density reduction as in [Cole2006].

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.

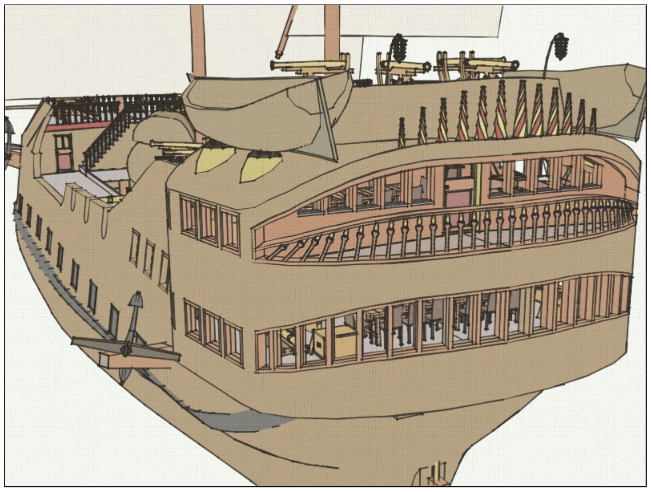

Figure 7: *Ship model.* The ship model has 300k triangles and 500k total line segments, and can be rendered at high-quality and interactive frame rates using our method.

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.

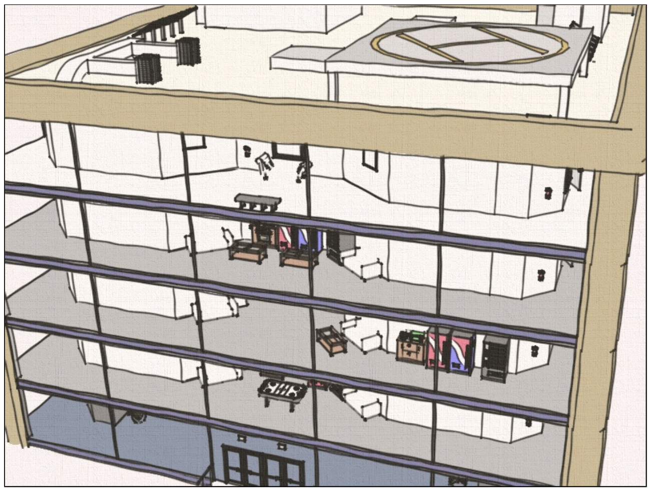
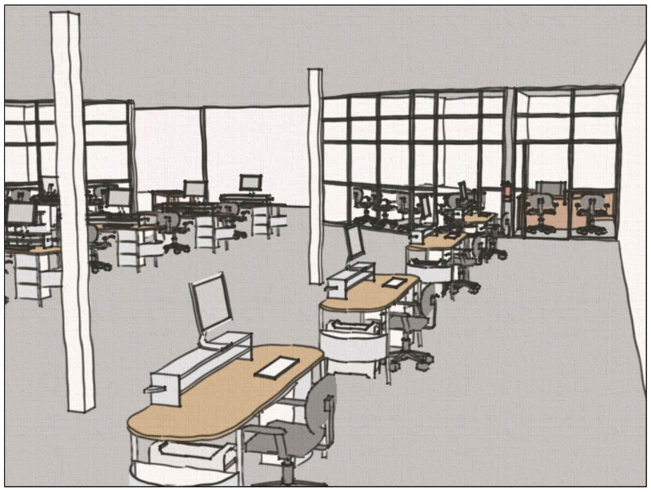

Figure 8: *Office model.* The office model has five levels, each with detailed furniture, totaling 330k triangles and 400k line segments.

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.