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.

1 Introduction

Stylized lines play a role in many applications of non-photorealistic 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 (Figure 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 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 (Figure 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 datastructure (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 manipulate 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 Figure 1 left).

This article 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 article, and offers a simpler, more conventional alternative to the segment atlas method. This article 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 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).

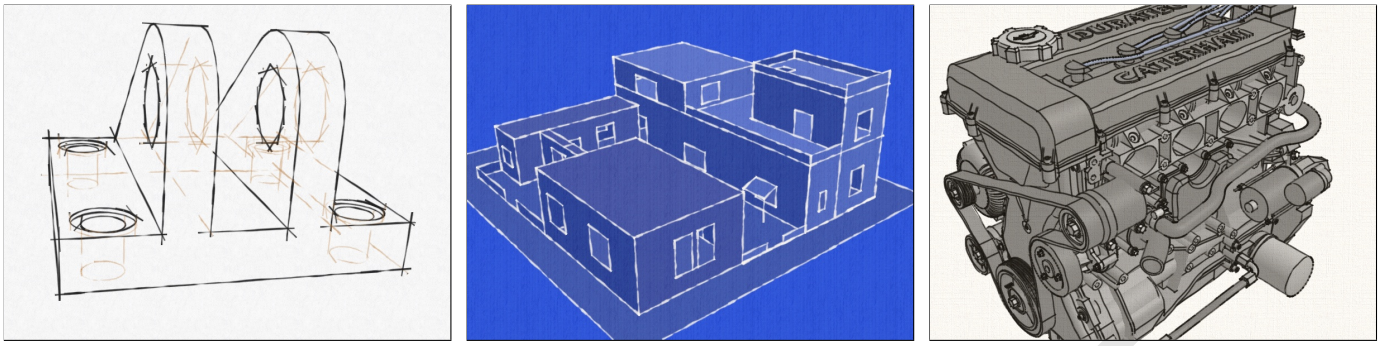

Fig. 1. *Examples of models rendered with stylized lines.* Stylized lines can provide extra information with texture and shape, and are more aesthetically appealing than conventional solid or stippled lines.

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 back-facing polygons in black, slightly displaced towards 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

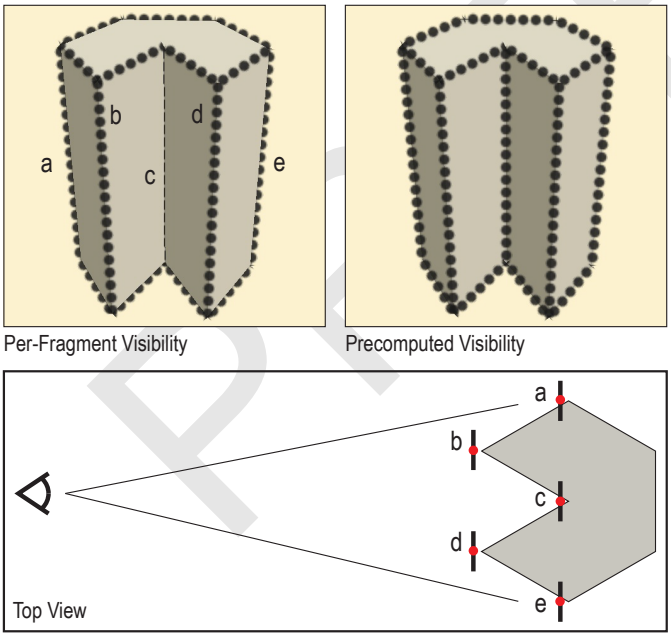

Fig. 2. *Per-fragment visibility vs. precomputed visibility.* When drawing wide lines using a naive per-fragment visibility test, only lines that lie entirely outside the model will be drawn correctly (b and d). Lines a, c, e are partially occluded by the model, even when some polygon offset is applied.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

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 (Figure 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 1960's. 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 line removal (with or without partial visibility) at interactive frame rates for complex models.

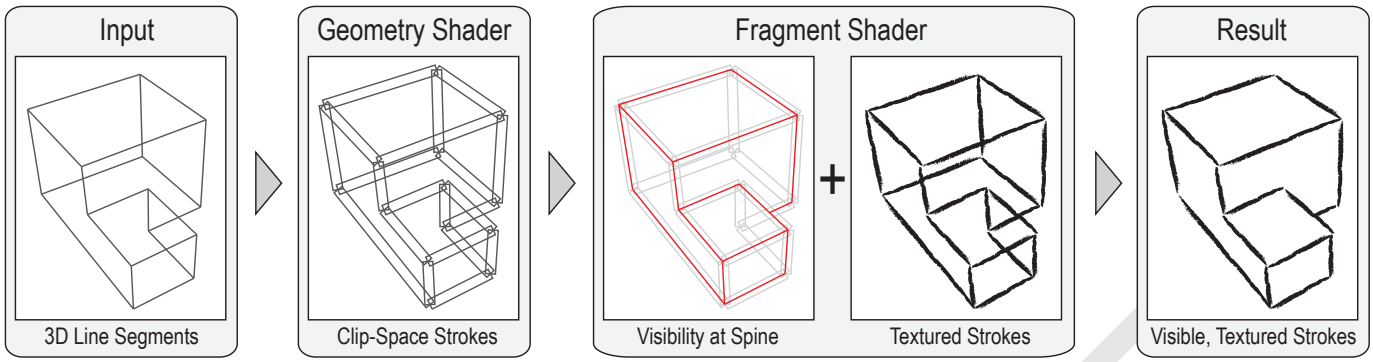

Fig. 3. *Steps in the spine test method.* The input is a set of 3D line segments. A geometry shader projects the line segments and creates clip-space strokes, preserving the homogenous positions for perspective-correct interpolation. A fragment shader checks visibility at the spine of the stroke, and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

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 Figure 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 Figures 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 Figure 1) by checking the adjacent face normals during stroke generation.

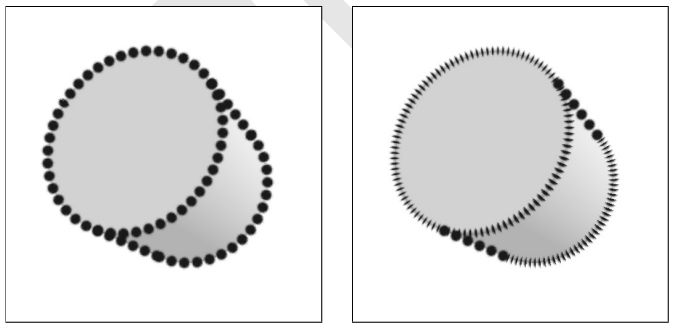

Fig. 4. *Curved stroke paths.* Strokes such as at the top and bottom of the cylinder consist of multiple segments. The correct approach is to parameterize the entire loop as a single stroke (*left*). Texturing each segment independently results in an incorrect result (*right*). Single-segment strokes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

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 Figure 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 requires 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

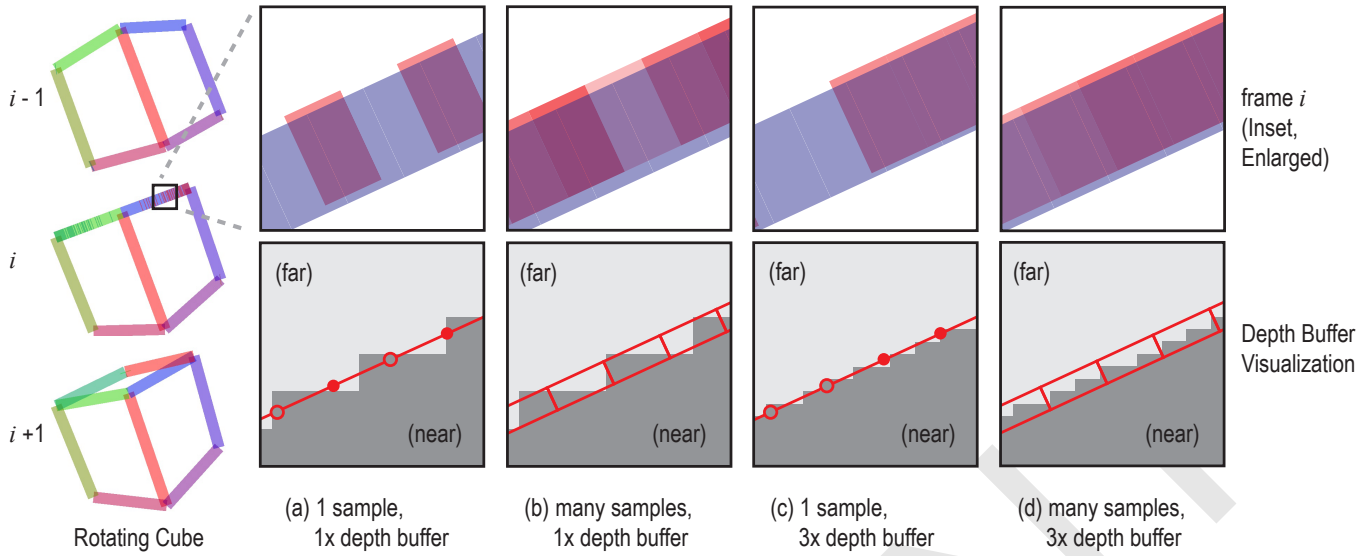

Fig. 5. *Visibility aliasing.* Aliasing in line visibility usually occurs at changes in occlusion. In this example, the red line is revealed behind the blue line as the cube rotates (left). The artifacts, while transient, can be severe for a single visibility sample with a standard depth buffer (a).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

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 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 Figure 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 (Figure 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 Figure 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 (Figure 6).

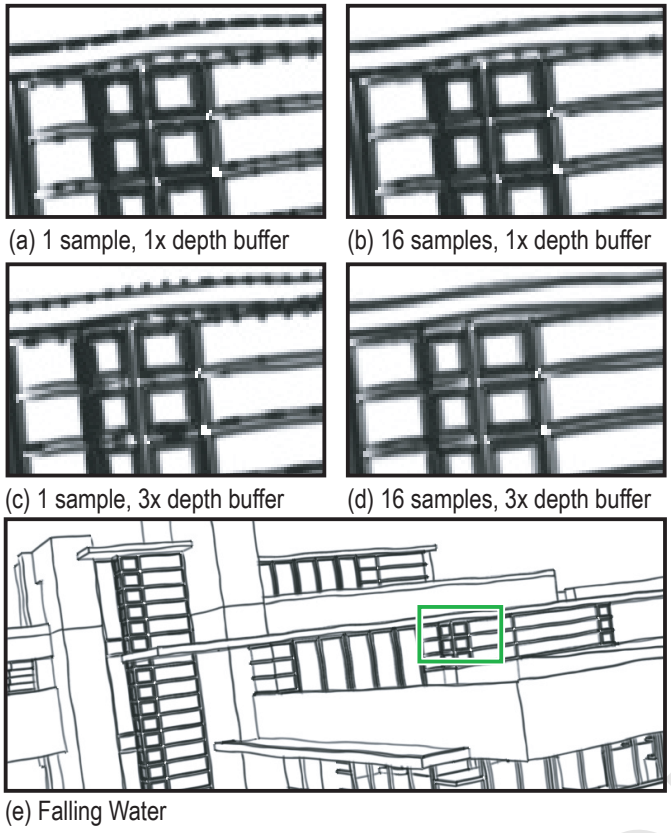

Fig. 6. *Aliasing in visibility test.* Results for varying number of samples and scale of depth buffer. Green box in (e) indicates location of magnified area. Visibility supersampling is used in both the spine test and segment atlas methods.

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 (Figure 5b). Additional depth probes are usually very fast (Table 1), 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 (Figure 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 (Figure 5c), but combining depth buffer scaling and depth test supersampling largely eliminates aliasing artifacts (Figure 5d). Since typical applications are seldom fill rate 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. Results of these techniques for a complex model can be seen in Figure 6.

4 Method 2: Segment Atlas

Stylization for curved strokes, or even simple effects such as endcaps or haloes, require some non-local 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 Figure 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 resides 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 $(\mathbf{p},\mathbf{q})$ (Figure 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:

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

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

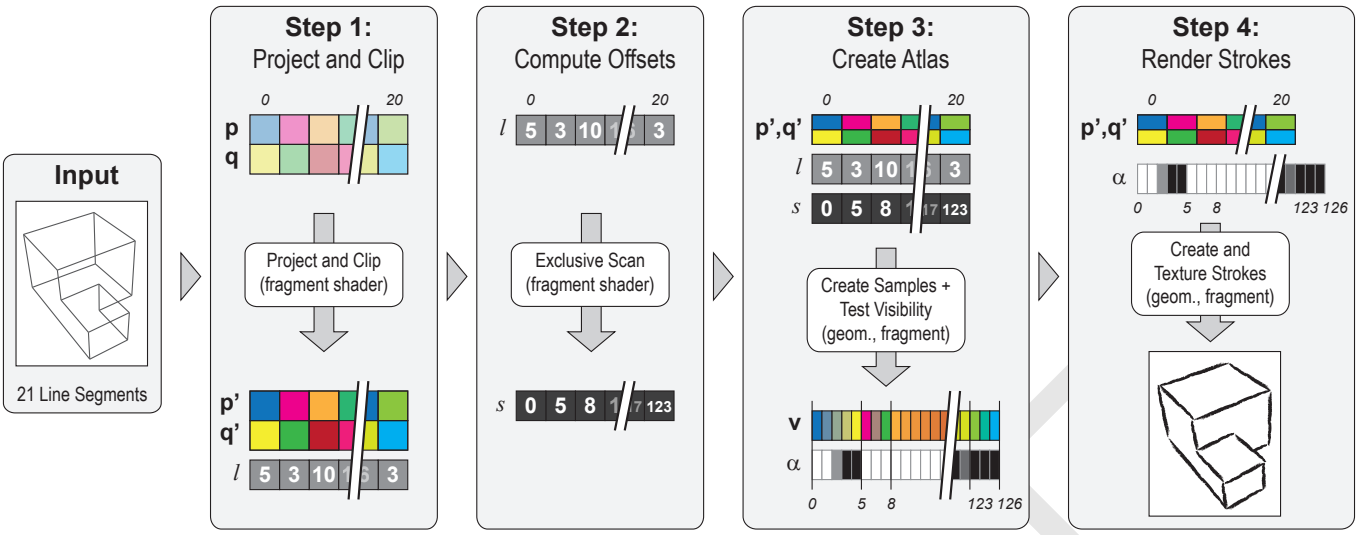

Fig. 7. *Segment Atlas Pipeline.* The input 3D line segments $(\mathbf{p}_i, \mathbf{q}_i)$ are stored in a table on the GPU. At each frame, each 3D segment is projected and clipped by a fragment shader, which also determines a number of samples l_i proportional to screen space length (step 1). Next, 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 Figure 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.

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., 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.

In a separate step, the sample counts l are converted into segment atlas offsets s by computing a running sum (Figure 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 multi-segment 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 standalone 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 multi-segment paths (Section 4.4).

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 multi-segment silhouette paths, e.g., the outline of a sphere. The parameterization of multi-segment paths is computed by the exclusive-scan operation, which assumes that the segment indices 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 3 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 $\mathbf{v}$ (Figure 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 (Figure 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 ($W = 4096$ or 8192 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 the projection matrix translated by $(-w, 1)$. Long segments will thus be wrapped across two consecutive lines in the atlas (Figure 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 (Figure 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).

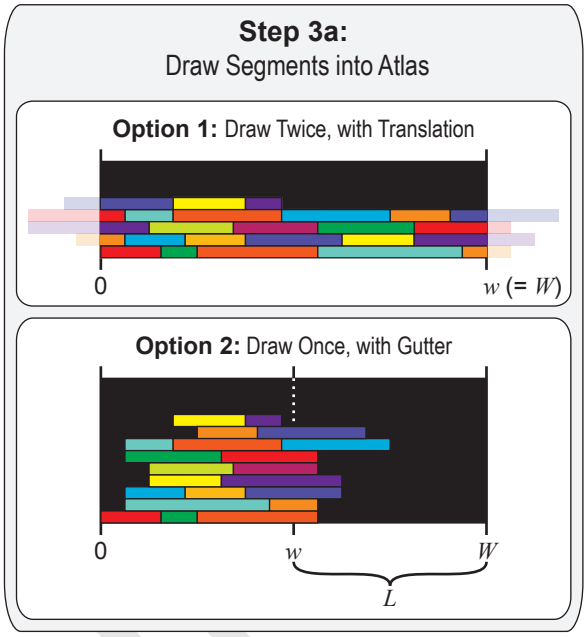

Fig. 8. *Segment atlas wrapping.* Because current generation GPUs do not support arbitrarily long 1D textures, the segment atlas must be wrapped to fit in a 2D texture. One option is to draw the atlas twice, wrapping segments that fall outside the width w (shown faded). Another option is to establish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

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 (Figure 9).

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

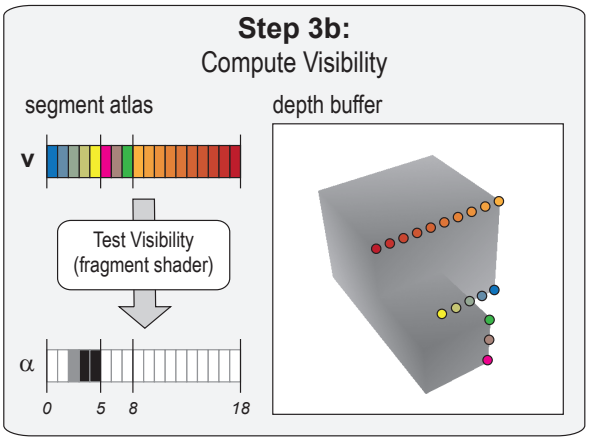

Fig. 9. *Visibility Testing.* The first three segments of Figure 7 are shown. Each sample in the segment atlas corresponds to a fragment. The fragment shader uses the screen space position $\mathbf{v}_j$ to test the sample against the depth buffer, recording the result in the visibility value α_j. Colors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

the atlas. However, we find 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 (Figures 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

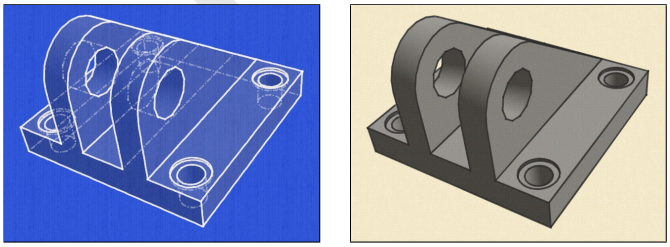

Fig. 10. *Variation in style.* A different texture may be used for lines that fail the visibility test (*left*), allowing visualization of hidden structures. Our method also produces attractive results for solid, simple styles (*right*).

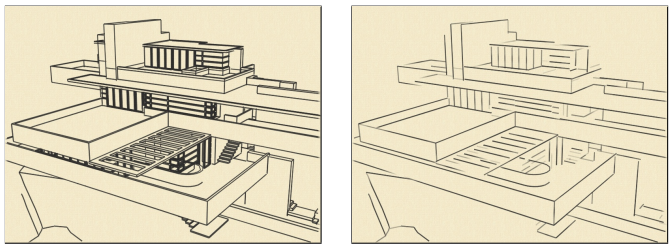

Fig. 11. *Line density control.* The segment atlas can store information besides visibility, such as local line density. Left: no density control. Right: line density reduction as described in [12].

allows a range of additional rendering effects not possible with the spine test method. Some examples include:

Mitering

In order to render multi-segment 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.

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 (Figure 1). A convincing sketchy overshoot effect can be achieved by setting the dilation amount to a constant screen-space length, then modulating this length pseudo-randomly 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).

Density Control

The segment atlas also can 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 Figure 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, since 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 ~ 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 Figures 10-12. The "+s" indicates 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 walkthrough sequence. All frames are rendered at 1024×768 using 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.

We tested the following rendering algorithms: (OGL) conventional OpenGL lines; (IBlo) single item buffer [Northrup2000]; (IBhi) $9\times$ supersampled item buffer with 3 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 9 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 $2-4\times$, 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% 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% 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.

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

The item buffer approach with $9\times$ supersampling and 3 layers, as suggested by [3], gives similar image quality to our methods with 9 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% 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.

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 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 (Figure 13), but also opens several possibilities for future work. Just as [12] introduced "stylized focus" as an artistic effect inspired by

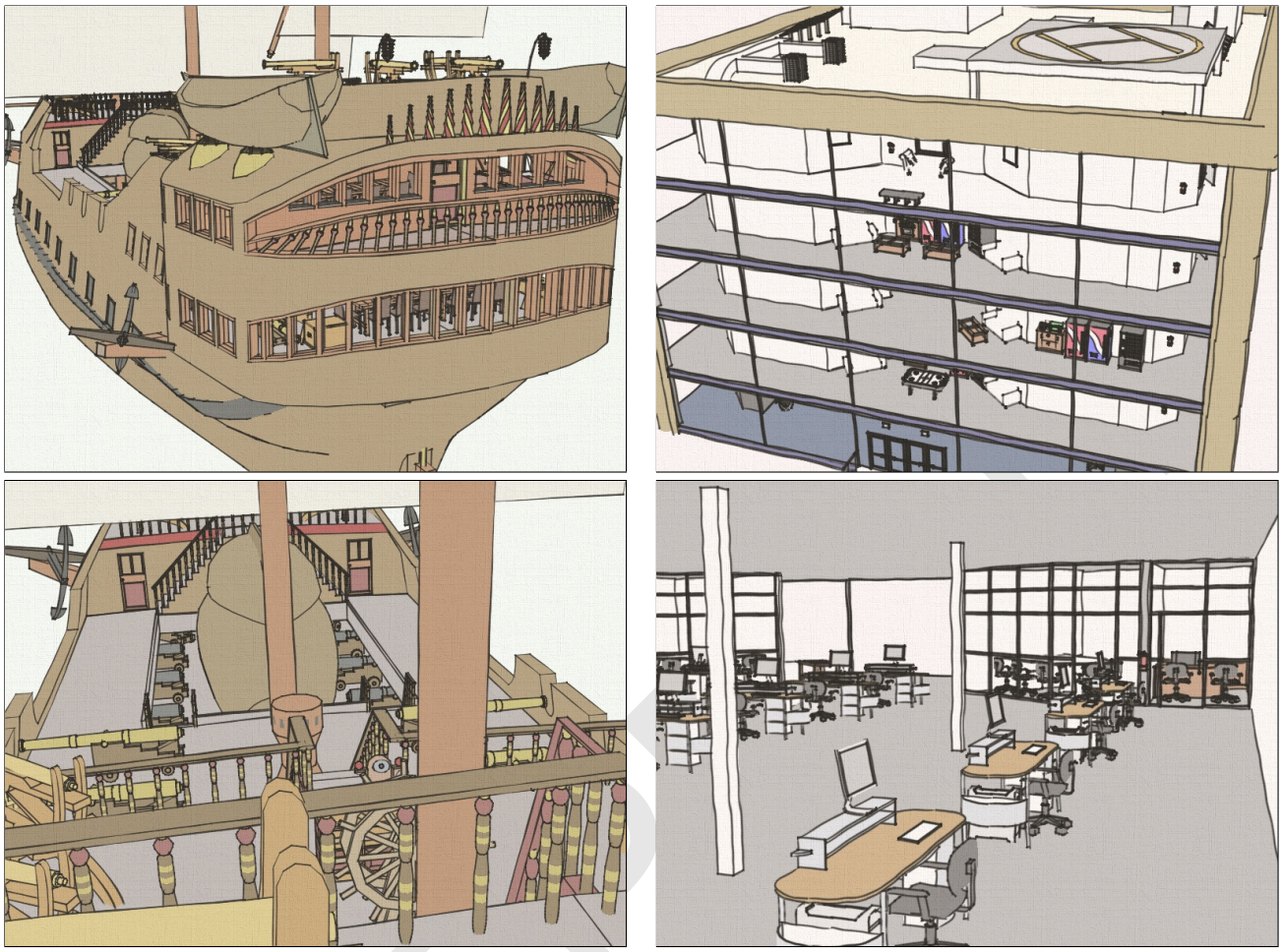

Fig. 12. *Complex models.* The ship model (*left*) has 300k triangles and 500k total line segments. The office model (*right*) has five levels, each with detailed furniture, totaling 330k triangles and 400k line segments. Both models can be rendered at high-quality and interactive frame rates using both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

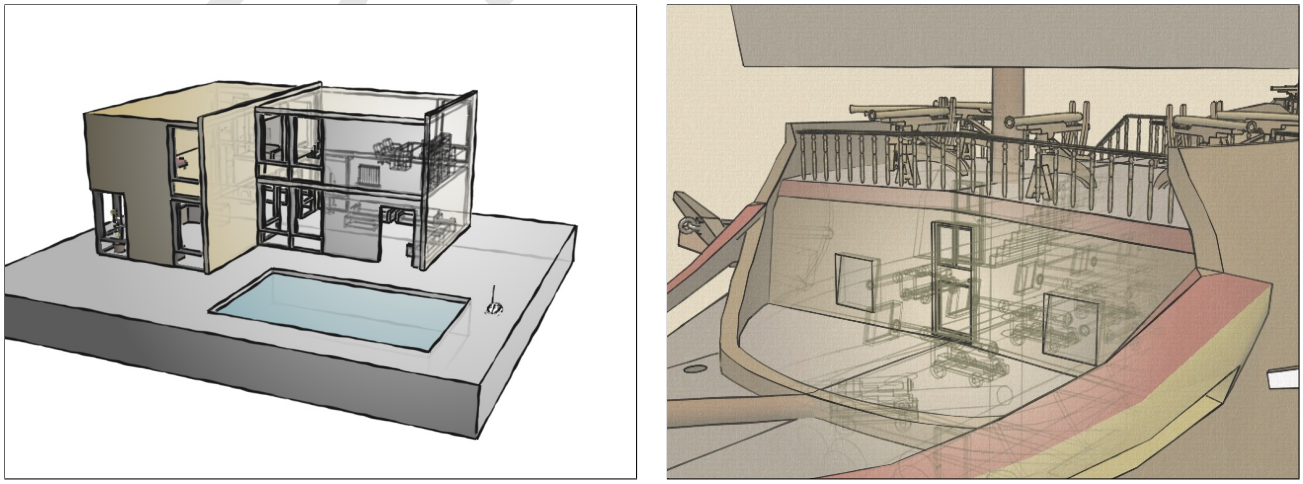

Fig. 13. *Drawing hidden lines using the segment atlas.* Locally controlling line visibility, using the stylized focus technique of [12], can reveal the internal structure of a model while providing context or hiding unimportant areas. Because the segment atlas stores visibility information for all strokes,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

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 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

We 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 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," in *Proceedings of SIGGRAPH 1997*, pp. 415–420, 1997.

[2] J. D. Northrup and L. Markosian, "Artistic silhouettes: a hybrid approach," in *Proceedings of NPAR 2000*, pp. 31–37, Jun. 2000.

[3] F. Cole and A. Finkelstein, "Partial visibility for stylized lines," in *Proceedings of NPAR 2008*, pp. 9–13, Jun. 2008.

[4] F. Co1e and A. Finkelstein, "Fast high-quality line visibility," in *Proceedings of I3D 2009*, pp. 115–120, Feb. 2009.

[5] X. Wu, "An efficient antialiasing technique," in *Proceedings of SIGGRAPH 1991*, pp. 143–152, 1991.

[6] R. Raskar and M. Cohen, "Image precision silhouette edges," in *Proceedings of SI3D 1999*, pp. 135–140, 1999.

[7] Y. Lee, L. Markosian, S. Lee, and J. F. Hughes, "Line drawings via abstracted shading," *ACM Trans. Graph.*, vol. 26, no. 3, pp. 18:1–18:5, Jul. 2007.

[8] A. Appel, "The notion of quantitative invisibility and the machine rendering of solids," in *Proceedings of the 22nd national conference of the ACM*, pp. 387–393, 1967.

[9] H. Weghorst, G. Hooper, and D. P. Greenberg, "Improved computational methods for ray tracing," *ACM Tran. Graph.*, 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," in *Proceedings of SIGGRAPH 2002*, pp. 755–762, 2002.

[11] R. D. Kalnins, P. L. Davidson, L. Markosian, and A. Finkelstein, "Coherent stylized silhouettes," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 856–861, Jul. 2003.

[12] F. Cole, D. DeCarlo, A. Finkelstein, K. Kin, K. Morley, and A. Santella, "Directing gaze in 3D models with stylized focus," *Proceedings of Eurographics Symposium on Rendering 2006*, pp. 377–387, Jun. 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, Jul./Aug. 2003.

[14] M. McGuire and J. F. Hughes, "Hardware-determined feature edges," in *Proceedings of NPAR 2004*, pp. 35–47, 2004.

[15] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*, pp. 97–106, 2007.

[16] S. Brabec and H.-P. Seidel, "Shadow volumes on programmable graphics hardware," in *EUROGRAPHICS 2003*, ser. Computer Graphics Forum, vol. 22, pp. 433–440, no. 3. Eurographics, September 2003.

[17] A. Hertzmann and D. Zorin, "Illustrating smooth surfaces," in *Proceedings of SIGGRAPH 2000*, pp. 517–526, 2000.

[18] S. Grabli, F. Durand, and F. Sillion, "Density measure for line-drawing simplification," in *Proceedings of Pacific Graphics*, pp. 309–318, 2004.

[19] P. Barla, J. Thollot, and F. Sillion, "Geometric Clustering for Line Drawing Simplification," in *Proceedings of Eurographics Symposium on Rendering 2005*, pp. 183–192, 2005.

[20] S. Grabli, E. Turquin, F. Durand, and F. Sillion, "Programmable style for NPR line drawing," in *Proceedings of Eurographics Symposium on Rendering 2004*, pp. 33–44, 407, 2004.

[21] D. Horn, "Stream reduction operations for GPGPU applications," in *GPU Gems 2*, ch. 36, pp. 573–589, M. Pharr, Ed. Addison Wesley, 2005.

[22] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive contours for conveying shape," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 848–855, 2003.

[23] T. Judd, F. Durand, and E. H. Adelson, "Apparent ridges for line drawing," *ACM Trans. Graph.*, vol. 26, no. 3, ar. 19, 2007.