Multi-scale Line Drawings from 3D Meshes

Alex Ni, Univ. of Michigan
Kyuman Jeong, POSTECH
Seungyong Lee, POSTECH
Lee Markosian, Univ. of Michigan

Abstract

We present an effective method for automatically selecting the appropriate scale of shape features that are depicted when rendering a 3D mesh in the style of a line drawing. The method exhibits good temporal coherence when introducing and removing lines as needed under changing viewing conditions, and it is fast because the calculations are carried out entirely on the graphics card. The strategy is to pre-compute a sequence of filtered versions of the mesh that eliminate (via a signal processing technique) shape features at progressively larger scales. Each mesh in the sequence retains the original number of vertices and connectivity, providing a direct correspondence between meshes. The mesh sequence is loaded onto the graphics card, and at run-time a vertex program interpolates positions and curvature values between two of the meshes, depending on distance to the camera. A pixel program then renders silhouettes and suggestive contours to produce the line drawing. In this way, fine shape features are depicted over nearby surfaces, while appropriately coarse-scaled features are depicted over more distant regions.

1. Introduction

Simplified line drawings are often preferred over realistic depictions in a number of contexts. Examples range from illustrations created to convey information about structure (as in repair manuals or medical texts), to story-telling directed at children (who respond to simple cartoon-like images), to the early stages of design, when a realistic rendering gives the false impression that design decisions have been largely finalized. In the context of 3D computer graphics, an additional consideration is that rendering complex shapes realistically requires significant resources, when very often the same shapes can be effectively depicted in a line drawing style that uses less data, modeling effort (e.g. for textures and BRDFs), and computation time.

The most basic non-photorealistic rendering uses little or no shading, and simply draws lines along silhouettes and sharp features. Recently, Decarlo *et al.* introduced *suggestive contours* [3]. These are additional view-dependent lines (like silhouettes) that convey a more complete impression of shape while adding relatively few additional lines. We refer to rendering styles that rely primarily on silhouettes, sharp features, and suggestive contours as *computer-generated line drawings*, or *line drawings* for short.

An important (but largely overlooked) point about computer-generated line drawings is that lines convey information about shape features *at some scale*, and the choice of scale matters. Specifically, features depicted via lines should project to a reasonable size *in image space*. For example, from the viewpoint of a person standing on a mountain slope, nearby pebbles and undulations in the dirt on the scale of a few centimeters could reasonably be depicted via lines in a line drawing. But the same features, seen from a distant viewpoint on a neighboring mountain, would appear in the drawing as a meaningless mass of sub-pixel lines. Rendering them conveys no shape information. Undulations in the terrain on a larger scale (tens of meters, say) would instead be appropriate to convey via lines from that viewpoint.

This issue can be side-stepped by restricting the camera to maintain roughly a constant distance from the surface, and selecting an appropriately filtered mesh (e.g. down-sampled and smoothed as needed for the intended viewing distance). For more general situations where the camera cannot be so constrained, a better solution is needed. We propose a novel solution that has the following benefits:

- It automatically controls the scale of depicted features to meet a target image-space size.
- It is simple and easy to implement.
- It achieves excellent temporal coherence when transitioning between features at different scales.
- It runs entirely on the graphics card, and so is quite fast.
- Our improved fragment program better controls the width of rendered lines.

The main disadvantage of the method is that it requires multiple versions of the original mesh to be stored simultaneously on the graphics card. Addressing this problem is an important challenge for future work.

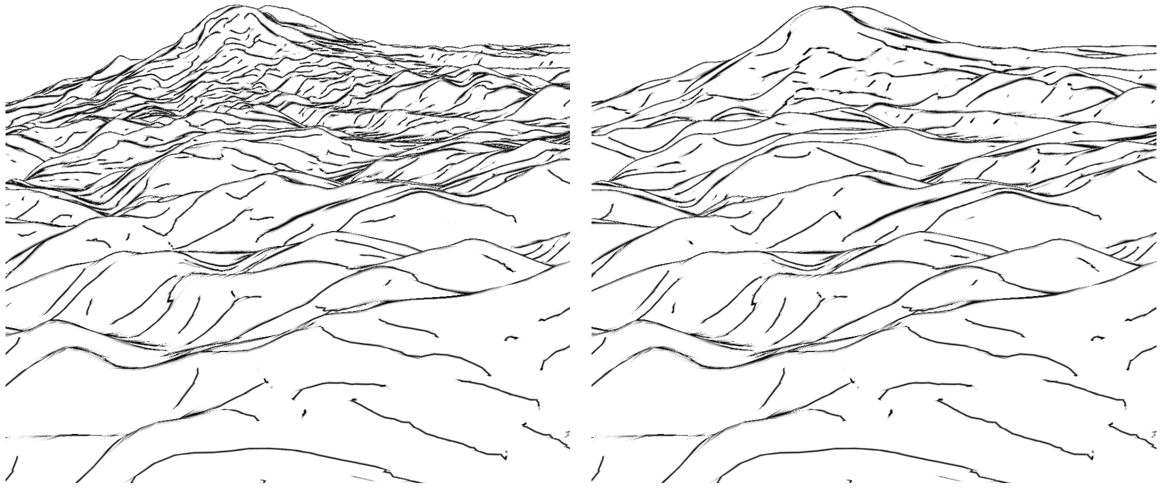

Figure 1. Left: a terrain model rendered with full detail. Right: The same view rendered with reduced detail in distant regions.

The basic idea is as follows. Given an input mesh M_0, we produce a sequence of progressively smoothed meshes M_1 through M_{N-1}. (For all the results demonstrated in this paper and the accompanying video [15], we used $N = 4$.) These meshes retain the same sampling and connectivity as the original mesh. It is thus straightforward to establish a correspondence between them. The smoothing is carried out so that each mesh in the sequence can resolve shape details of approximately half the size of those resolved by the next mesh in the sequence. Also in the pre-process, we compute vertex normals and curvature information for each mesh. This data is loaded onto the graphics card and at run-time a vertex program interpolates positions, normals, and curvature values between two of the meshes, depending on distance to the camera. A pixel program then renders silhouettes and suggestive contours to produce the line drawing. In this way, fine shape features are depicted over nearby surfaces, while appropriately coarse-scaled features are depicted over more distant regions.

2. Related Work

Techniques for producing computer-generated line drawings date back at least to 1967 with the work of Appel [1]. He proposed an *object-space approach* in which silhouettes and sharp features are detected in object space, then projected into the image, processed for visibility, and rendered as strokes. A variety of methods in this category have been proposed [4, 14, 5, 8, 11, 21, 13, 3]. (For a survey, see Isenberg *et al.* [10].) An important advantage of object-space methods is that they can produce stylized strokes – i.e., strokes that wobble, overshoot, or vary in width, color or texture to resemble natural media. They generally require good-quality meshes (in the form of oriented 2-manifolds).

Alternatives to object-space methods include image-space methods [20], and frame-buffer methods [5, 18, 17]. In both cases, feature lines are not detected explicitly, but instead appear in the image due to per-pixel operations. A disadvantage of these approaches is that they do not support stroke stylization. They are generally simple, though, and can work with arbitrary models, including "polygon soup."

Despite the variety of work on computer-generated line drawings, we are not aware of any that addresses the problem of depicting just those shape features that project into the image at an appropriate scale. Somewhat related is the idea of controlling line *density* in image space. Recently, Grabli *et al.* [6] and Wilson and Ma [22] separately describe systems that render complex 3D geometry in the style of pen-and-ink drawings, with special processing to control the resulting stroke density and reduce visual "clutter." Both systems combine 3D operations with 2D image processing to detect regions of visual complexity (e.g. dense silhouettes), and remove detail where it exceeds a threshold. Both systems require significant processing time, and neither addresses temporal coherence for image sequences, or the specific problem of depicting shape features at a desired scale in image space.

The work of Pauly *et al.* [16] has some similarities to our work. While they work with point sampled surfaces, they produce a scale-space representation of the input shape (as we do), consisting of a sequence of shapes with progressively coarser features smoothed away. They extract features at various scales, and apply the results in a

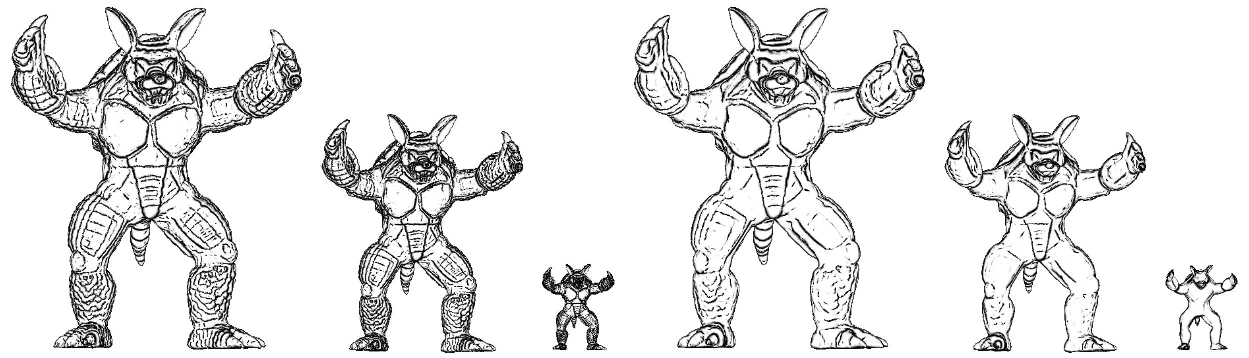

Figure 2. An armadillo model rendered with constant detail from three viewpoints (left), then with controlled detail (right).

non-photorealistic renderer. However, the resulting features (ridges and valleys) are view-independent, and the scale is not chosen according to the image-space size of the projected shape.

Our method is implemented entirely in hardware, via a vertex program that computes positions and curvature values, and a fragment program that draws silhouettes and suggestive contours via per-pixel operations. As a result, the method is extremely fast, but does not produce stylized strokes. However, the same algorithm could be carried out in the CPU, resulting in object-space stroke paths that could be rendered as stylized strokes. Our fragment program is somewhat related to the idea of using environment maps to render silhouettes, described by Gooch *et al.* [5], and also to the strategy of using a specially constructed texture (parametrized by radial curvature in one dimension, and its derivative in the other) to render suggestive contours, described by DeCarlo *et al.* [2]. Our method does not use texture maps or environment maps. Instead it uses screen-space derivatives to better control line width of the resulting silhouettes and suggestive contours. DeCarlo *et al.* alluded to such a strategy as future work in their paper.

3. Pre-process: Smoothing

Our overall goal is to let the user choose, within some reasonable range, an image space “target size” for features depicted in the line drawing. (Following DeCarlo *et al.*[2], we treat edge length as a measure of feature size.) As the camera zooms in, we do not expect to add detail beyond what is present in the original mesh, so the problem is to eliminate details that are too small for the current viewing conditions. One possible approach would be to view-dependently simplify the mesh so that the length of its edges in image space (ignoring foreshortening) approximately equals the target size.

This scheme presents two problems: it requires considerable run-time computation, and achieving temporal coherence of silhouettes and suggestive contours on a dynamically changing mesh is difficult. In fact, prior to developing the method we present here, we investigated such an approach using a view-dependent progressive mesh scheme to perform mesh simplification [12]. The alternative strategy we propose in this paper is more temporally coherent, and much faster. (E.g., the 346,000 polygon armadillo model shown in Figure 2 renders over 30 times faster, at 64 fps.)

Instead of directly manipulating the connectivity of the mesh, we use the signal processing framework of Guskov *et al.*[7] to pre-compute a sequence of meshes (with identical connectivity) in which progressively larger shape features are smoothed away. In brief, Guskov’s method works as follows. In an analysis phase, the input mesh is first simplified to a “base mesh” via a sequence of edge collapse operations. (This is a modified progressive mesh scheme [9]). A vertex is removed with each edge collapse, but first an associated “detail coefficient” is recorded. The detail coefficient is defined as the difference between the actual positions of vertices in a local neighborhood of the collapsed edge and their “predicted” positions – i.e., positions that minimize a discrete fairing functional defined over the local neighborhood. The complete sequence of edge collapses (with associated detail coefficients) is stored.

With this representation, the original mesh can be exactly reconstructed by reversing the process: the edge collapses are undone via “vertex split” operations, applied in reverse order. Each time, the positions of vertices in the local neighborhood are predicted as before, and the actual positions are restored by adding back the detail coefficient. A kind of band-pass filter can be achieved by simply omitting the addition of detail coefficients after a given number of vertex split operations. The resulting mesh has the same connectivity as the original, with fine details removed.

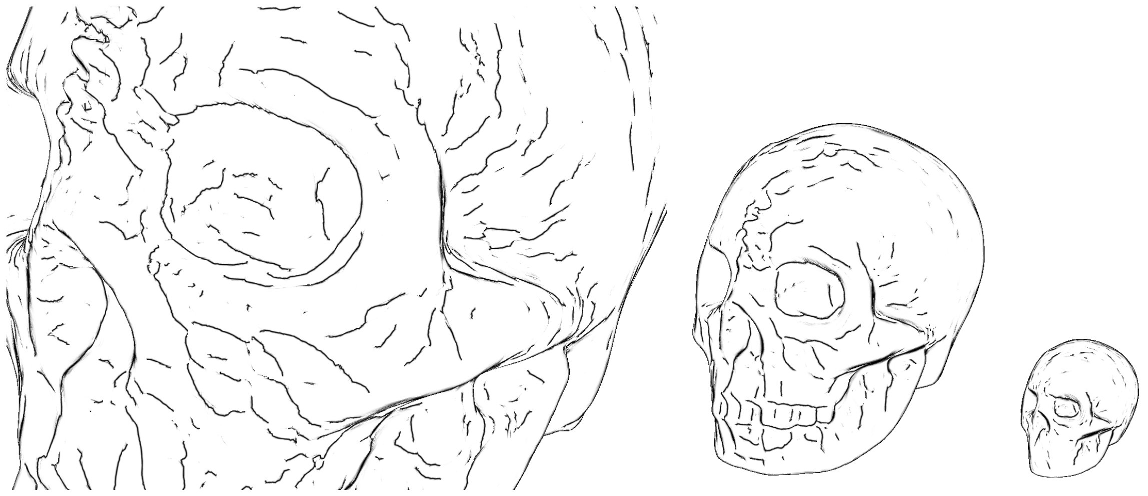
Figure 3. A skull model rendered with controlled detail from three viewpoints.

We can double the feature size resolved by the mesh (eliminating detail) if we stop adding detail coefficients after a certain fraction r of vertex split operations have been applied (assuming the mesh triangles are roughly equal in size). I.e., if n is the total number of vertex split operations, we omit the detail coefficients after the first rn operations. Initially we expected $r = 1/4$ would work to double the feature size of the mesh (since a decimated mesh with $1/4$ the original edges should have edges roughly twice the length of the original). However, the smoothing effect associated with each edge collapse is greater than would occur for an ordinary progressive mesh, since with each operation 7 vertices are moved to "smoother" locations. We thus found empirically that taking $r = 3/4$ produces a smoothed mesh that is (subjectively) closer to our goal of doubling the feature size.

We thus produce a sequence of meshes M_k, where for each k we omit detail coefficients after the first $r^k n$ vertex split operations have been applied. Consequently, M_k contains features half the size of features M_{k+1}. In practice, we stop at M_3, whose features are 8 times coarser than those of M_0. More meshes could be used to achieve a greater range of detail levels, at the cost of additional memory taken up on the graphics card.

The last step in the pre-process stage is to compute, for each vertex of each mesh in the sequence, principal directions and curvatures, and the derivative-of-curvature tensor **C**, as described by DeCarlo *et al.*[2].

Because the majority of our smoothing is accomplished by moving vertices along their normals, the vertex density can change and affect the accuracy of one-ring based normal and curvature calculations. These minor defects can be largely removed by smoothing the normals (used to compute curvature) by averaging them with their neighbors.

4. Run-time

At run-time, the pre-computed data is loaded onto the graphics card. We use a display list to assure the data remains resident on the card.

The number of floats needed per vertex is as follows. Position: 3, principal directions and curvatures: 8, derivative-of-curvature tensor **C**: 4. Total: 15 floats, or 60 bytes per vertex, assuming 4-byte floats. Thus the required memory is $60vN$ bytes, where v is the number of vertices in the mesh and N is the number of meshes in the sequence M_k. E.g., the armadillo model shown in Figure 2 has about 173,000 vertices, and $N = 4$, so roughly 41 MB of memory on the graphics card is needed.

4.1. Vertex Program

The responsibility of the vertex program is (1) to determine the desired feature scale (per vertex), and then (2) interpolate data between the corresponding two levels of the mesh sequence to produce values used by the fragment program (Section 4.2) to render silhouettes and suggestive contours.

To determine the feature scale at a vertex, we start with an image space target size T. In our demonstrations in this paper, we treat T as a constant that is specified by the user via a slider. Given the average edge length L around a vertex in mesh M_0, we can compute the distance d_0 at which mesh features near the vertex appear in the image at the desired size: $d_0 = cL/T$, where c is a constant determined by parameters of the perspective camera and by the window size. By construction, for $k > 0$ the corresponding distance for mesh M_k is $d_k = 2^k d_0$. Consequently, for each vertex processed by the vertex program, we first compute its distance d to the camera, then compute its fractional level ℓ in the mesh sequence as $\ell = \log(\frac{d}{d_0})$. We then interpolate needed values between level $k = \lfloor \ell \rfloor$ and $k+1$, using interpolation parameter $\ell - k$. (In our current implementation, we assume that edges of mesh M_0 are of roughly constant size, and so we use the same value of L for all vertices.)

The interpolated values mentioned above are all scalar quantities: radial curvature, derivative of radial curvature, and $\mathbf{n} \cdot \mathbf{v}$, where $\mathbf{n}$ is the unit surface normal and $\mathbf{v}$ is the unit "view vector" pointing from the vertex to the camera. This dot product is used by the fragment program to compute silhouettes, while radial curvature and its derivative are used to render suggestive contours. The surface normal is not explicitly passed into the program, since it can be computed via the cross product of the principal directions.

4.2. Fragment Program

As hinted by DeCarlo *et al.* [2] as possible future work, we use a fragment shader to control the width of rendered lines. To control line width, we use the screen-space derivative instructions specified in the OpenGL Shading Language standard [19]. With these operations (also known as "shader anti-aliasing") we compute the per-pixel gradient of radial curvature in screen-space. We divide the radial curvature value at a pixel by the magnitude of the gradient to yield approximate distance (in pixels) to the suggestive contour (i.e., the zero-crossing of radial curvature). Pixels sufficiently close to the zero-crossing are filled with the stroke color. An anti-aliased line can be achieved by replacing an abrupt cut-off with a function that falls off smoothly with distance over a short interval. (We use a gaussian.) Suggestive contours are clipped by testing the derivative, scaled appropriately using the feature size of the model [2].

We apply the same idea to silhouettes, computing the gradient of the scalar field $\mathbf{n} \cdot \mathbf{v}$, i.e. the unit surface normal dotted with the unit view vector, as explained in Section 4.1.

This procedure effectively controls line widths and generally produces good quality, anti-aliased lines. Under some conditions it can lead to artifacts, however. When triangles project to sub-pixel sizes in screen space, a small screen distance to a zero-crossing of radial curvature may correspond to a span of many triangles in 3D. In that case the approximate screen distance computed via the screen-space gradient may lack accuracy, resulting in small irregularities in the rendered line width. Also, when the rendered lines are very wide, noticeable discrepancies in line width can occur at triangle boundaries, due to discontinuities in the screen-space derivative functions. This can be seen at one point in the accompanying video when the line widths are made extra wide.

Lastly, because the mesh is used as a canvas for the lines, any line will be clipped to the mesh's boundary in 2D. Silhouettes by definition lie on this boundary, and so are particularly susceptible to being clipped and losing width. Increasing their width can counteract this.

5. Results and Discussion

Model	Faces	Vertices	Frames per second
Bunny	69,473	34,835	140
Feline	99,732	49,864	100
Armadillo	345,944	172,974	64
Skull	393,216	196,612	70
Terrain (1)	465,470	233,704	70
Terrain (2)	803,880	403,209	12

Table 1. Performance data for some of our test models.

In summary, we have presented a solution to the problem of controlling the scale of features depicted in line drawings of 3D models. Although this problem is perhaps obvious, it has received little attention until now. Our solution is easy to implement, effective, fast, and temporally coherent, as demonstrated in the images in this paper and the accompanying video. Figure 1 shows the ability of the method to selectively control the scale of shape features, with greater detail reduction in distant regions. Figure 2 shows that a static mesh leads to an ever-increasing density of lines when the camera zooms out, but our method avoids that problem. The skull in Figure 3 reveals more detail in close-up views and appropriately less detail when the camera pulls back. These transitions in level of detail happen smoothly, as the video demonstrates. (Note that temporal coherence in the video is somewhat reduced by aggressive MPEG-4 encoding.)

Table 1 displays frame rates and model sizes for some of our test models. In all cases the rendering window was 1024x1024. Our test machine was a 2.8 GHz Pentium 4 with 1 GB of RAM, and a 256 MB NVIDIA GeForce 6800 Ultra GPU. We suspect that the steep drop-off in performance observed for the largest model occurred because the data did not all fit in the memory of the GPU, resulting in extra overhead in reading data from main memory each frame.

The large memory requirement of our method is a drawback. One challenge for future work is to develop ways to compress the data stored on the GPU, taking advantage of the high correlation of the values. Another avenue for future work is to develop more general criteria for reducing or increasing the detail depicted in different parts of a line drawing. E.g., the static terrain model shown in Figure 1 on the left is quite cluttered in the distant parts, while the dynamic model on the right exhibits roughly constant detail. However, the increased density in the image on the left does serve as a depth-cue, which is missing in the right image. Letting the "target size" of features vary according to a procedure that takes into account depth, importance, image location, or other factors is an interesting idea for further exploration.

References

[1] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 1967 22nd national conference*, pages 387–393, New York, NY, USA, 1967. ACM Press.

[2] D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *NPAR 2004: Third International Symposium on Non Photorealistic Rendering*, pages 15–24, June 2004.

[3] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics*, 22(3):848–855, July 2003.

[4] D. Dooley and M. Cohen. Automatic illustration of 3D geometric models: Lines. *1990 Symposium on Interactive 3D Graphics*, 24(2):77–82, March 1990.

[5] B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[6] S. Grabli, F. Durand, and F. Sillion. Density measure for line-drawing simplification. In *Proceedings of Pacific Graphics - 2004*, 2004.

[7] I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. *Proceedings of SIGGRAPH 99*, pages 325–334, 1999.

[8] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 517–526, July 2000.

[9] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 99–108, Aug. 1996.

[10] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. A developer's guide to silhouette algorithms for polygonal models. *IEEE Comput. Graph. Appl.*, 23(4):28–37, 2003.

[11] T. Isenberg, N. Halper, and T. Strothotte. Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. *Computer Graphics Forum*, 21(3):249–258, 2002.

[12] K. Jeong, A. Ni, S. Lee, and L. Markosian. Detail Control in Line Drawings of 3D Meshes. *Proceedings of Pacific Graphics 2005*, 2005.

[13] R. D. Kalnins, P. L. Davidson, L. Markosian, and A. Finkelstein. Coherent stylized silhouettes. *ACM Transactions on Graphics*, 22(3):856–861, July 2003.

[14] L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97*, pages 415–420.

[15] A. Ni, K. Jeong, S. Lee, and L. Markosian. Multi-scale Line Drawings from 3D Meshes. http://graphics.eecs.umich.edu/npr/msld/, 2005.

[16] M. Pauly, R. Keiser, and M. Gross. Multi-scale feature extraction on point-sampled models. In *Proceedings of Eurographics*, 2003.

[17] R. Raskar. Hardware support for non-photorealistic rendering. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 41–47, 2001.

[18] R. Raskar and M. Cohen. Image precision silhouette edges. *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[19] R. J. Rost. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[20] T. Saito and T. Takahashi. Comprehensible rendering of 3D shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):197–206, August 1990. Held in Dallas, Texas.

[21] M. Sousa and P. Prusinkiewicz. A few good lines: Suggestive drawing of 3d models. *Computer Graphics Forum (Proc. of EuroGraphics '03)*, 22(3), 2003.

[22] B. Wilson and K.-L. Ma. Representing complexity in computer-generated pen-and-ink illustrations. In *NPAR 2004: Third International Symposium on Non Photorealistic Rendering*, June 2004.