Multi-scale Line Drawings from 3D Meshes

Alex Ni
Univ. of Michigan

Kyuman Jeong
POSTECH

Seungyong Lee
POSTECH

Lee Markosian
Univ. of Michigan

Figure 1: A skull model rendered with controlled detail from three viewpoints.

Abstract

We present a method to view-dependently control the size of shape features depicted in computer-generated line drawings of 3D meshes. Our method exhibits good temporal coherence during level of detail transitions, and is fast because the calculations are carried out entirely on the GPU. The strategy is to pre-compute, via a digital geometry processing technique, a sequence of filtered versions of the mesh that eliminate shape features at progressively larger scales. Each filtered mesh retains the original connectivity, providing a direct correspondence between meshes.

At run-time, the meshes are loaded onto the graphics card and a vertex program interpolates curvatures and positions between corresponding vertices in adjacent meshes of the sequence. A fragment program then renders silhouettes and suggestive contours to produce a line drawing for which the size of depicted shape features follows a user-specified "target size". For example, we can depict fine shape features over nearby surfaces, and appropriately coarse-scaled features in more distant regions. More general level-of-detail policies could be implemented on top of our approach by letting the target size vary with scene attributes such as depth, image location, or annotations provided by the scene designer.

1 Introduction

Simple line drawings are often preferred over realistic depictions in a number of applications. Examples include illustrations (as in repair manuals or medical texts), story-telling directed at children (who respond to simple, colorful images), and the early stages of design, when a realistic rendering falsely gives the impression that design decisions have been finalized. In the context of 3D computer graphics, an added consideration is that rendering complex shapes realistically requires significant resources, when 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* [2003], which are view-dependent lines that, combined with silhouettes, 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 (varying) target image-space size.
- It is simple and easy to implement.
- It achieves excellent temporal coherence when transitioning between features at different scales.

I3D 2006, Redwood City, California, 14–17 March 2006.
© 2006 ACM 1-59593-295-X/06/0003 $5.00

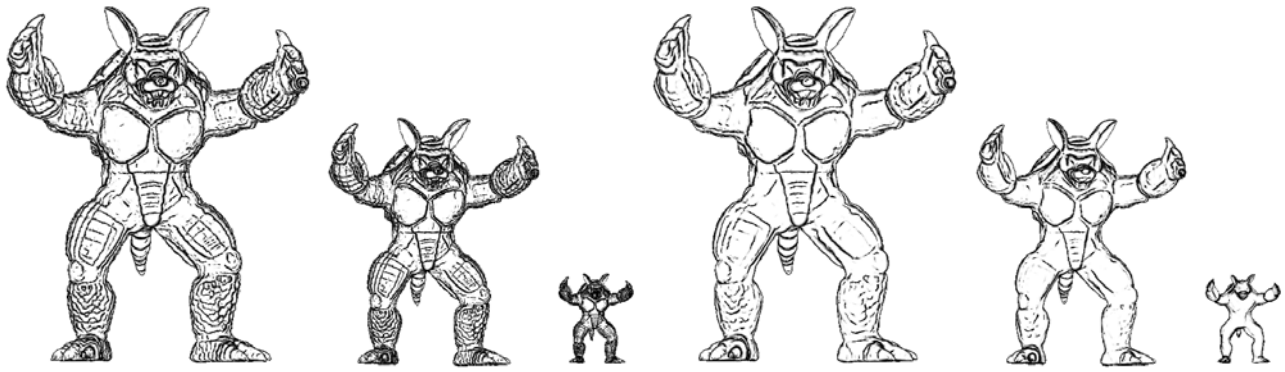

Figure 2: An armadillo model rendered with constant detail from three viewpoints (left), then with controlled detail (right).

- It runs entirely on the graphics card, and so is quite fast.
- Our improved fragment program better controls line width (compared to previous methods, e.g. [DeCarlo et al. 2004]).

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.

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, we used $N = 4$.) These meshes retain the sampling and connectivity of the original mesh, so it is 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. Each mesh is associated with a viewing distance at which its resolvable shape features project to the image at a user-set "target size." The pre-process also computes vertex normals and curvature information for each mesh.

At run-time, the meshes are loaded onto the graphics card and a vertex program interpolates curvatures and positions between the two corresponding vertices in the mesh sequence whose viewing distances bracket the distance to the vertex. A fragment program then renders silhouettes and suggestive contours to produce a line drawing for which the target size controls the size of depicted shape features. For example, we can depict fine shape features over nearby surfaces, and appropriately coarse-scaled features in more distant regions. For the examples in this paper we used a constant target size across the image, chosen by the user via a slider.

With this framework, more specialized control over the size of depicted features can be achieved simply by manipulating the image-space target size. For example, we could depict finer shape features for "important" objects, or decrease feature size somewhat with depth to selectively preserve some degree of depth cuing. In the accompanying video we demonstrate the simple manipulation of target size over the whole image via a slider. Further exploration of such specialized detail control is left as future work.

2 Related Work

Techniques for producing computer-generated line drawings date back at least to 1967 with the work of Appel [1967]. 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 [Dooley and Cohen 1990; Markosian et al. 1997; Gooch et al. 1999; Hertzmann and Zorin 2000; Isenberg et al. 2002; Sousa and Prusinkiewicz 2003; Kalnins et al. 2003; DeCarlo et al. 2003]. (For a survey, see Isenberg *et al.* [2003].) 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 [Saito and Takahashi 1990] and frame-buffer methods [Gooch et al. 1999; Raskar 2001]. 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."

McGuire and Hughes [2004] describe a method for detecting and rendering feature lines (including silhouettes and suggestive contours) entirely in hardware, using an "edge mesh" data structure that makes local connectivity information available within a vertex program. As a result, their method is able to render lines with some stylization, while leveraging the computation power of the GPU. A tradeoff is the significantly increased memory requirements.

None of the methods mentioned above addresses the problem of controlling the size of shape features depicted in the rendered image. Somewhat related is the idea of controlling line *density* in image space. Grabli *et al.* [2004] and Wilson and Ma [2004] 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.* [2003] 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 selected scales, and apply the results in a 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.

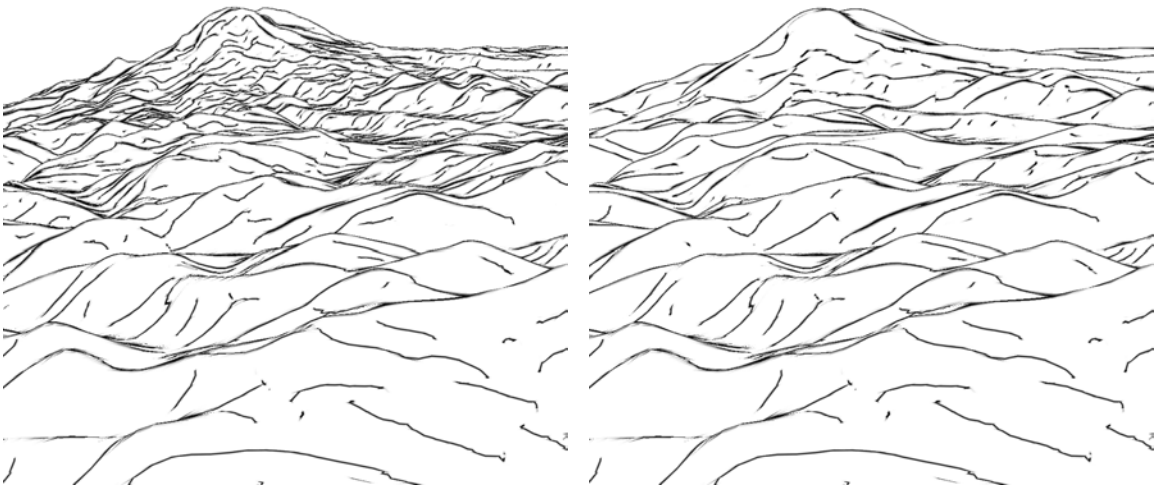

Figure 3: A terrain model rendered with full detail (left), and roughly constant image space detail (right). In practice an effect somewhere between these might be desired. Our method could support that, with an additional slider in the interface.

Our own recent work [Jeong et al. 2005] addresses the same goals as we do in this paper, but takes a different strategy. In that paper, we used a progressive mesh representation to view-dependently simplify the mesh when its resolution exceeds an image space threshold. The strategy we propose in this paper is simpler to implement, more temporally coherent, and much faster for models with fewer than a million polygons. (E.g., the 346,000 polygon armadillo model shown in Figure 2 renders over 30 times faster, at 64 fps.) One advantage of our previous method is that its memory requirements scale better with model size.

3 Pre-process: Smoothing

Our overall goal is to let the user choose an image space "target size" for features depicted in the line drawing. We will also need a value for the 3D length of a "feature." We call this length the "feature size." Often, a good choice is to use edge length, as observed by DeCarlo *et al.* [2004]. This is reasonable for meshes that are constructed "economically," meaning the number of mesh elements is not much more than that needed to resolve the intended shape.

Given these values, we can evaluate whether the mesh is too detailed in the neighborhood of a vertex by projecting the feature size (at that depth) into image space and comparing it to the target size. If it is smaller than the target size, the mesh is too detailed. The choice of feature size is not critical, provided the user has the means (e.g., using a slider) to adjust the target size up or down interactively. For the results presented in this paper, we treated the target size as a constant across the image, which tends to produce a roughly constant line density in the rendered image, as in Figure 3, right. Other policies could be chosen instead, as we discuss in Section 5. Different behaviors result by simply changing the way the target size is specified over the image.

We use the signal processing framework of Guskov *et al.* [1999] 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 [Hoppe 1996]). 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.

We can double the feature size resolved by the mesh (reducing 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 enhanced in Guskov's scheme compared to an ordinary progressive mesh, since with each operation, *seven* 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 those of 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 is to compute, for each vertex of each mesh in the sequence, principal directions and curvatures, and the derivative-of-curvature tensor $\mathbf{C}$ [DeCarlo et al. 2004].

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: 6, derivative-of-curvature tensor **C**: 4. Total: 13 floats, or 52 bytes per vertex, assuming 4-byte floats. Thus the required memory is $52vN$ 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 36 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 detail level (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.

We determine the detail level for each vertex as follows. Given the image space target size T at the vertex and the feature size L associated with the vertex in mesh M_0, we compute the distance d_0 at which L projects to the image with length T; namely: $d_0 = cL/T$. Here, c is a constant determined by camera parameters and 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 implementation, we assume that edges of mesh M_0 are of roughly constant size, 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 suggested by DeCarlo *et al.* [2004] in their discussion of 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 [Rost 2004]. 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 [DeCarlo et al. 2004].

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.

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.

Table 1 displays frame rates and model sizes for some of our test models. In all cases the rendering window was 1024x1024, and each mesh had 4 levels of detail (the original plus 3 smoothed versions). 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 when reading data from main memory each frame.

The large memory requirement is a significant limitation of our method. Each additional level of detail requires storage that is proportional to the number of vertices in the mesh (see Section 4). In practice, this means that on current graphics cards our method can support only a small number of levels of detail for models like those listed in Table 1. This impacts both the quality of the rendered line drawings and the range of viewpoints over which the method can be applied. Addressing this limitation is thus an important challenge for future work. We expect that improvements can be made by exploiting the inherent redundancy in the data, either by using a multi-resolution mesh representation, or by directly compressing the data in some way.

Another topic for future work is the interplay between line drawings and tone. Our method can produce significant changes in perceived tone during level of detail transitions, e.g. when small surface details are eliminated, causing a transition from many lines to relatively few. This effect can be seen in Figures 1 and 2. Failure to regulate tone is also a limitation of ordinary line drawings (without level of detail control). See for example Figure 2, left: the model appears darker when the camera zooms out. Achieving tonal control using just silhouettes and suggestive contours is difficult, as those lines are often sparse (limiting the darkness that can be conveyed), and their proximity in image space is generally not known (without significant additional processing). Still, one way to address this issue for multi-scale line drawings might be to

use a two-pass rendering method: render the line drawing in a back buffer, then estimate tone in the resulting image via image processing operations and render the final image using a modified detail map that increases detail in regions that are deemed too light and reduces it in regions that are too dark.

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. The skull in Figure 1 reveals more detail in close-up views and appropriately less detail when the camera pulls back. Transitions in level of detail happen smoothly, as the video demonstrates. Figure 2 shows that a static mesh produces an ever-increasing density of lines when the camera zooms out, but our method avoids that problem. Figure 3 shows a rendering in which roughly constant detail is achieved in image space by reducing 3D detail as distance to the camera increases.

The main point of our method is not to achieve uniform detail in image space, but to provide the necessary control so that the rendered line drawing can match any provided image space detail map. An important avenue for future work is thus to investigate useful policies for specifying such detail maps (equivalent to specifying a varying target size). Defining the target size to take into account depth, user-assigned importance of objects, image location, image space velocity, tone, or other factors is an interesting avenue for further exploration.

Acknowledgements

We thank Szymon Rusinkiewicz for making the Real-time Suggestive Contours source code available on the web. This research was supported in part by the BK21 program and the ITRC support program in Korea, and the NSF (CCF-0447883).

References

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

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Transactions on Graphics 22*, 3 (July), 848–855.

DECARLO, D., FINKELSTEIN, A., AND RUSINKIEWICZ, S. 2004. Interactive rendering of suggestive contours with temporal coherence. In *NPAR 2004: Third International Symposium on Non-Photorealistic Rendering*, 15–24.

DOOLEY, D., AND COHEN, M. 1990. Automatic illustration of 3D geometric models: Lines. *1990 Symposium on Interactive 3D Graphics 24*, 2 (March), 77–82.

GOOCH, B., SLOAN, P.-P. J., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics* (April), 31–38.

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

GUSKOV, I., SWELDENS, W., AND SCHRÖDER, P. 1999. Multiresolution signal processing for meshes. *Proceedings of SIGGRAPH 99*, 325–334.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 517–526.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 99–108.

ISENBERG, T., HALPER, N., AND STROTHOTTE, T. 2002. Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. *Computer Graphics Forum 21*, 3, 249–258.

ISENBERG, T., FREUDENBERG, B., HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. A developer's guide to silhouette algorithms for polygonal models. *IEEE Comput. Graph. Appl. 23*, 4, 28–37.

JEONG, K., NI, A., LEE, S., AND MARKOSIAN, L. 2005. Detail control in line drawings of 3D meshes. *The Visual Computer 21*, 8-10 (September), 698–706. Special Issue of Pacific Graphics 2005.

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

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-time nonphotorealistic rendering. In *Proceedings of SIGGRAPH 97*, 415–420.

MCGUIRE, M., AND HUGHES, J. F. 2004. Hardware-determined feature edges. In *NPAR 2004: Third International Symposium on Non-Photorealistic Rendering*, 35–44.

PAULY, M., KEISER, R., AND GROSS, M. 2003. Multi-scale feature extraction on point-sampled models. In *Proceedings of Eurographics*.

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

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

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3D shapes. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August), 197–206.

SOUSA, M., AND PRUSINKIEWICZ, P. 2003. A few good lines: Suggestive drawing of 3d models. *Computer Graphics Forum (Proc. of EuroGraphics '03) 22*, 3.

WILSON, B., AND MA., K.-L. 2004. Representing complexity in computer-generated pen-and-ink illustrations. In *NPAR 2004: Third International Symposium on Non Photorealistic Rendering*.