Hardware-Accelerated Interactive Illustrative Stipple Drawing of Polygonal Objects
Vision, Modeling, and Visualization 2002

Aidong Lu† Joe Taylor† Mark Hartner‡ David Ebert†
Charles Hansen‡

†School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47906
{alu, jtaylor, ebertd}@purdue.edu
‡School of Computing, University of Utah, Salt Lake City, UT
{hartner, hansen}@cs.utah.edu

Abstract

Pen and ink rendering techniques provide artistic, illustrative, and informative representations of objects. With recent advances in hardware graphics technology, several researchers have developed interactive non-photorealistic rendering techniques, including hatching, toon rendering, and silhouettes. However, the stippling method of drawing and shading using dots has not received as much focus. In this paper, we present an interactive system for stipple drawing of surface-based objects that provides illustrative stipple renderings of multiple objects and includes adaptation of the stippling to provide a consistent rendering of objects at any scale. We also describe the use of the latest graphics hardware capabilities for accelerating the rendering by performing the silhouette rendering on the GPU and the stipple density enhancement calculations as a vertex program.

1 Introduction

Pen and ink illustration techniques have proven to be very effective in scientific, technical, and medical illustration [4]. Within the computer graphics community, the hatching style has received the most attention (e.g., [3, 7, 9, 10]). An alternative pen and ink technique that uses the specific placement of dots to convey shape, structure, and shading is stippling [1, 5, 7]. While stippling has received less research attention than hatching, it is a very effective illustration technique that has several advantages over hatching. First, it is more effective at conveying subtle form. Second, hatching and line based techniques can improperly suggest nonexistent textures, ribbing, or segmentation of the model [4].

Most previous work in computer generated stippling performs the stippling process as a non-interactive post-process after rendering a grey-scale image [1], similar to dithering techniques. These systems use approaches such as Vornoi diagrams for stipple point placement to approximate a specific grey-level in the resultant image. Praun et al. [7] presented an interactive texture-based approach for real-time hatching and showed an example of a stippled texture as tone art map rendering. However, their system requires the expensive pre-generation of the mip-map textures and concentrates on appropriate orientation and blending of the textures to generate the resultant images. In contrast, our system uses actual points as geometry to achieve stipple rendered images, as can be seen in Figures 1 and 2.

Figure 1: An example stipple rendering of a blobby dataset. The left image has uniform stipple placement. The right image has shading enhancement to convey simple surface shading.

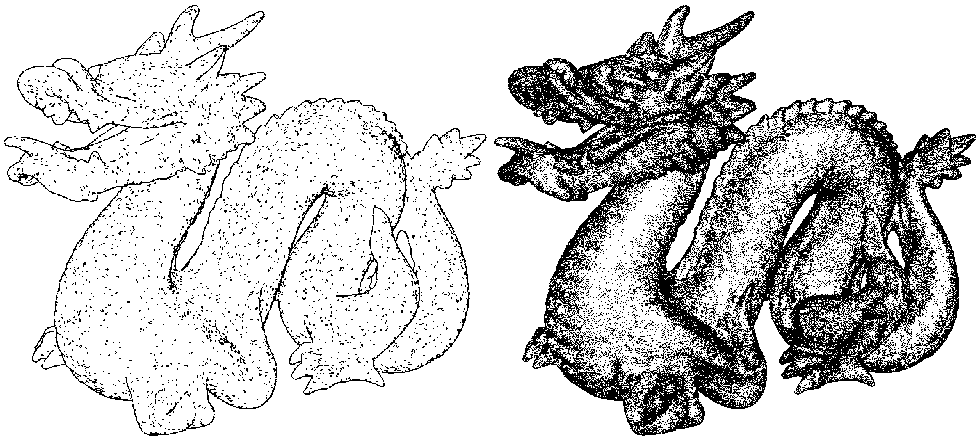

Figure 2: Two example stipple renderings of a dragon with varying levels of stipple density.

Stippling is especially effective when combined with silhouette techniques to define the boundary of an object. We have developed an interactive stipple rendering system for surface-based objects that provides effective, illustrative drawings of multiple objects at interactive rates that incorporates distance based adaptation of the stippling to provide consistent rendering of objects at any scale.

2 The Interactive Stippling System

The interactive point-based system renders both points and polygons to create effective stipple drawings. Points are rendered to produce the stipple patterns and polygons are rendered to create silhouettes, toon shading, and perform hidden surface (point) removal. The system consists of an initial point generation phase, followed by the view-dependent multi-scale rendering phase described below.

2.1 Initial Point Generation

The initial points for each polygon are generated randomly within the polygon and then redistributed to approximate a Poisson-disc distribution [1]. This approximately spaced placement of points simulates traditional stipple placement techniques. Two factors are used to initially adjust the maximum number of stipples for each polygon, N_{max}:

1. **Maximum Point Density,** D_{max}: The user is allowed to adjust a maximum point density parameter to change the overall tone of the image.
2. **Relative Point Density,** D_r: The number of points is automatically adjusted based on the relative size of the initial screen area of the projected polygon so that larger polygons receive more stipples. This ensures that the stipple pattern is independent of the object tessellation. The approximated projected screen area is based on the polygon area and depth and orientation of the polygon as follows:

$$area = area_{orig} * cos(N_z * \pi) * Z_{rel}$$

where N_z is the z component of the image space polygon normal, and Z_{rel} is the relative depth of the polygon in the scene.

Therefore, the maximum stipples for each polygon is calculated as the following:

$$N_{max} = Object_{max} * min(D_{max}, D_r)$$

During interactive rendering, this value is adjusted per frame based on distance and shading enhancements described in the next section.

Illustrative stipple rendering is achieved by rendering the resulting points as geometric primitives. Frame-to-frame coherence of stipples is achieved since their location within a polygon does not vary over time: only the number of stipples per polygon varies based on shading and distance calculations. To eliminate any stipple "popping", the intensity of the last stipple for each polygon is based

on the fractional portion of the density value. Unfortunately, even with back-face culling, geometry which would not have been rendered with hidden surface methods contributes stipples to the final image. Therefore, to achieve correct hidden surface removal, we displace the stipples by a small epsilon along the normal direction. We use the OpenGL polygon offset which takes into effect the orientation of the polygon when calculating the offset.

2.2 Shading

To achieve an effective stipple drawing, the shading of the surface must be conveyed by the density of the stipples. Therefore, a simple modification to the Lambertian diffuse illumination of the Blinn-Phong model, $(N \cdot L)^n$, is used to shade the object, which potentially reduces the number of points drawn per polygon for each frame. The reduction in the number of points provides perceived shading based on the number of stipples. Figure 1 shows the effect of shading on the stipple placement. Note the details which become visible with shading.

2.3 Distance Based Scale Enhancement

A desirable quality of a pen and ink rendering is that the object appears with consistent brightness when rendered at multiple scales. This problem was addressed in hatching by Winkenbach et al. [10], where they solved the sampling and reconstruction problem to produce hatching of a consistent grey-level across multiple scales. For point-rendered stipples, we achieve consistent rendering at multiple scales by modifying the number of points drawn per polygon.[1] There are two issues which impact the number of stipples due to scale: the relative placement of the object in the viewing frustum, and the relative position of each polygon in the object's bounding box. We refer to the first as *extrinsic* distance enhancement and the second as *intrinsic* distance enhancement.

The extrinsic distance enhancement modulates the point density on a per-object basis to provide consistent appearance. We compute the extrinsic distance enhancement differently based on whether a polygon projects larger than or smaller than a user specified minimum screen coverage area for stippling (two to nine pixels).

If the polygon's projection is large enough to generate a stipple pattern, we use the following formula to adjust the point density:

$$pnt\_density = shaded\_pnt\_density / (\frac{Z_{min}}{2} - Z_{curr})^{pr}$$

where Z_{min} is the closest Z value in the scene and all other object Z values are less than this. Z_{curr} is the depth of the center of the polygon, and pr is a user parameter to control the amount of extrinsic distance enhancement. The orientation of the polygon in the view frustum is also used to adjust the point density based on the cosine of the z component of the image space normal vector. The results of this enhancement can be seen in Figure 3.

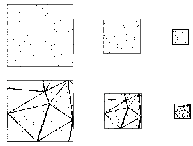

Figure 3: This image shows the extrinsic distance enhancement where the polygon's projection is larger than a pixel. The three images with different sized triangles show how the system automatically modifies the original number of stipples to provide a consistent appearance.

When the projection of a polygon becomes too small, creating a shaded stipple pattern is not practical, and we must select an appropriate percentage of the polygons to achieve a consistent grey-level image as the object decreases in size. When processing polygons in strips or area mode, we use the polygon's current approximate screen area[2] to determine the number of polygons that cover our minimum stipple screen area, N. We use a probability function based on this calculated area to determine if the polygon should be drawn. We, therefore, only generate stipples for every N^{th} polygon. Similarly, for other types of meshes, a random probability value can be assigned to each polygon and used in conjunction with the projected area to determine which surface elements to process. The approximated projected screen area is based on the initial

[1] As described above, this is also adjusted based on the area of the polygon to correctly render models with non-uniform tessellation.

[2] This value is updated per frame based on the changing depth and orientation of the polygon to the camera.

projected area of the polygon and the current view location as follows:

$$area = area_{orig} * cos(N_z * \pi) * Z'_{rel}$$

where N_z is the z component of the image space polygon normal, and Z'_{rel} is the relative depth of the polygon compared to its original position. This area value is then used to determine the probability that a given polygon has its stipples drawn, using one of the two methods above. An example of this enhancement can be seen in Figures 4, 5, and 6.

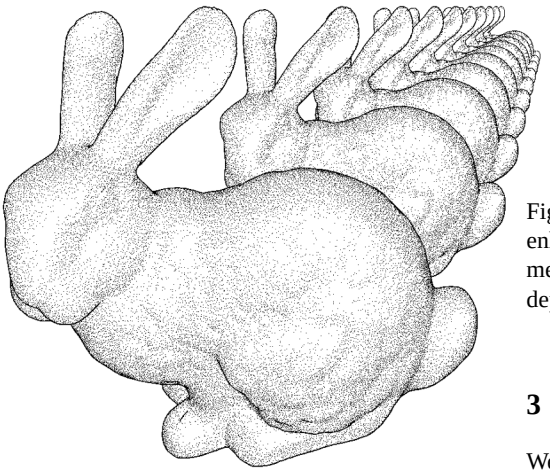

Figure 4: This image shows the multi-scale extrinsic enhancement where polygons become smaller than one pixel. The system automatically adjusts the stipples to provide a consistent appearance.

Our intrinsic distance enhancement also modifies the number of stipples rendered to provide user controllable depth cuing internal to the model. Intrinsic depth enhancement allows the user to focus attention on the closest portion of a polygonal model, using the relative depth of the polygon within the object's bounding box. For this enhancement, we use the following equation:

$$pnt\_density = curr\_pnt\_density * (1 + (Z_{curr}/A)^{de})$$

where Z_{curr} is the depth of the polygon, $(-A, A)$ is the depth range in the object and de is a user controlled parameter that allows the enhancement rate to vary, which exaggerates or lessens the effect. The results of intrinsic distance enhancement can be seen in Figure 7 by comparing the stippling on the left horse and the right horse. The horse on the left has no intrinsic distance enhancement, whereas, the stippling on the body and hind legs of the horse on the right is lighter, exaggerating the depth relationship within the model.

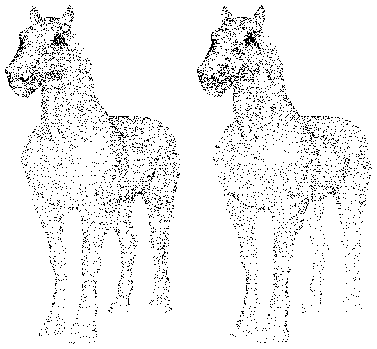

Figure 7: This image shows the effects of distance enhancement. The left image has no depth enhancement, while the image on the right has the relative depth of the horse's components exaggerated.

3 Hardware Acceleration

We have designed our system to balance the computation that is performed on the CPU with that performed on the GPU to achieve the best performance. One main division of the workload is between the stipple density calculation on the CPU and the silhouette calculation on the GPU. We have also implemented GPU-based resolution, distance, and light enhancement for the stipple rendering and will continue to explore moving more of our calculations to the vertex processing unit of the GPU.

3.1 Silhouette Rendering

Silhouettes have become a vital part of many non-photorealistic rendering techniques [2, 8]. Gooch et al. [2] have shown that silhouettes can emphasize structure and detail, as well as be a aesthetically pleasing. We have incorporated two different approaches to silhouette rendering in our stippling system.

The first approach is based on Raskar and Cohen's method for interactive silhouette lines [8]. They describe a simple method for displaying

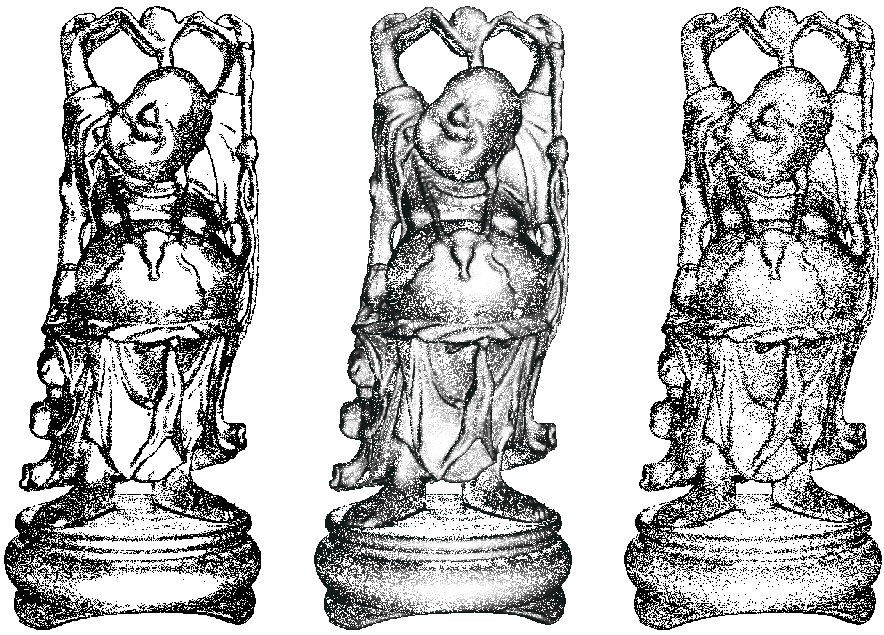

Figure 5: Another example of the multi-scale extrinsic enhancement applied to a statue model. The left image has no enhancement; the middle image has point intensity added for anti-aliasing; and the right image has extrinsic enhancement.

image-based silhouettes using standard OpenGL hardware. Their method involves rendering front-facing polygons into the depth buffer, while back-facing polygons are rendered in wireframe mode with a slight offset and a depth test set to equal, yielding silhouette lines of uniform thickness.

When the object occupies a large portion of the image, thicker silhouette lines give a better result. However, when the object is moved away from the viewpoint, the silhouette lines should become thinner to give a consistent appearance. We use the object's bounding sphere to scale the silhouette line width based on the object's distance from the camera. The radius, which is orthogonal to the viewing direction, is projected onto the screen. The relative size provides a scaling factor for the silhouette thickness. Figure 8 shows an example of the Stanford bunny rendered without silhouettes (left) and with silhouettes (right). The two smaller silhouette bunnies show the effects of thinning the silhouettes. The top-most has the thinning enabled, whereas the bottom smaller bunnies use the same pixel size as the larger object.

Our second approach is to use a "toon" shader silhouette approach and the vertex programmability extension to OpenGL. In a vertex program that is executed for each triangle, we compute the dot product of the eye vector and the surface normal vector and multiply this by an increasing ramp function, clamped between 0.0 and 1.0. This becomes the intensity of the surface. Only surfaces whose normals are nearly perpendicular to the eye vector will have intensity near 1.0, thus producing a silhouette. The sharpness of the silhouette is determined by the steepness of the ramp function. Figure 9 shows an example of the Stanford bunny rendered using this technique with and without stipples. The weakness of this approach is that the silhouette is not uniform when comparing edges of high and low curvature. This can clearly be seen when comparing the silhouette at the feet and ears of the rabbit to the silhouette along its back and chest. To improve the overall rendered quality, these images were generated with an implementation using two separate vertex programs: one for the silhouette polygons and one for the stipple points.

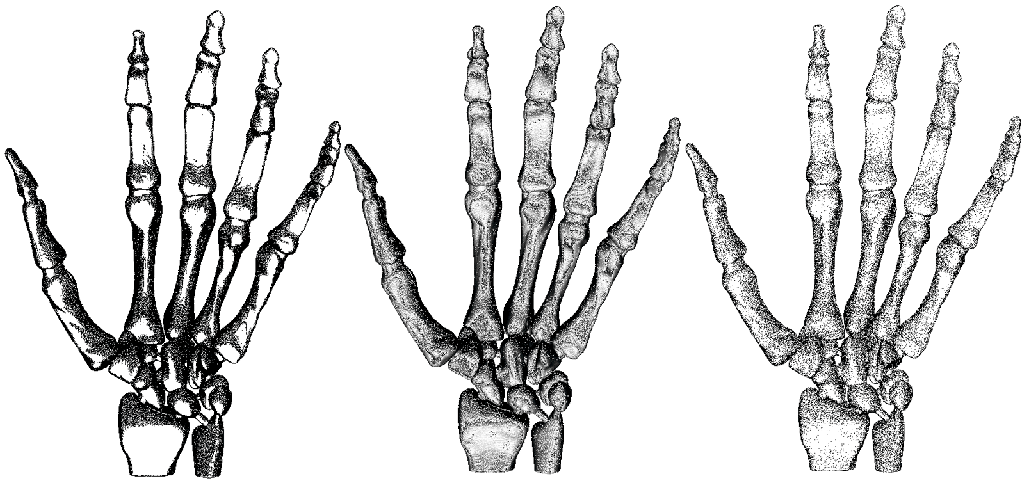

Figure 6: Another example of the multi-scale extrinsic enhancement applied to a hand model. The left hand has no enhancement, the middle image has point intensity added for anti-aliasing, and the right image has extrinsic enhancement.

While the silhouette vertex program implements the technique as described, the stipple vertex program uses a decreasing ramp function causing stipples near the silhouette to fade. The slope of the decreasing function determines the sharpness of this fade. Other approaches to silhouette enhancement include a per-pixel approach using cubemap textures and register combiners, as well as a multi-pass technique using an offset viewport for each pass [6].

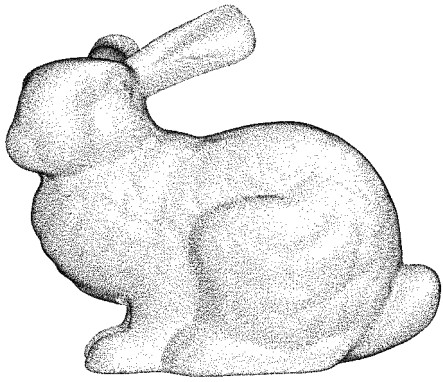

Figure 9: This image shows our toon shading vertex program silhouette method.

3.2 GPU-based Distance and Resolution Enhancement

We have been able to increase the performance of the stipple renderer by implementing all of the stipple enhancements (resolution, distance, lighting) on the GPU using the vertex program extension of OpenGL on the Nvidia GeForce3. The conversion of these functions to the vertex program hardware instruction set is simple, with the standard trick for converting conditionals to vertex programs (compute both paths and select one of the results). For the power function, the lighting coefficients instruction (LIT) was used for a fast, approximate calculation. The vertex program currently uses 36 operations per vertex, which allows for future enhancements for stippling to be computed with vertex programs. Compared to software stipple rendering only, the current performance increase ranges from 0% to almost 100%. The more polygons in the model, the greater the speedup. With silhouette rendering added, depending on the view, the vertex program enhancements can be as much as 50% faster than software stipple rendering (with hardware silhouettes).

When the computations are done in software, the density for a given surface is calculated, and only the number of stipples (vertices) that are to be

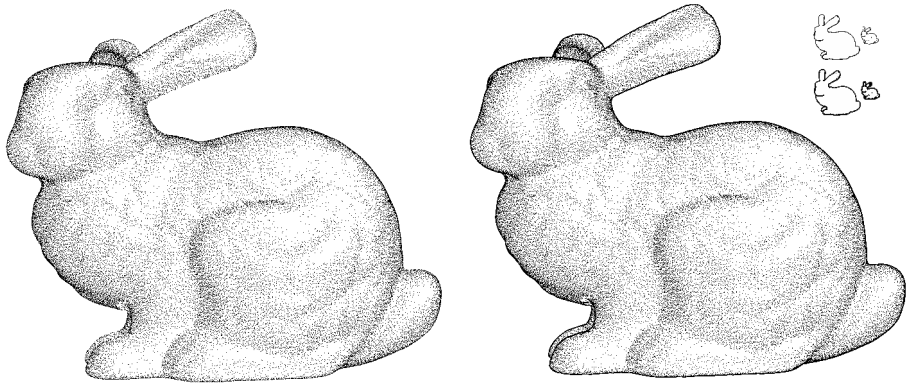

Figure 8: The left image shows stipples without silhouette edges. The right large bunny has image-based silhouettes enabled. To the right and above the lower bunny demonstrates the silhouette thinning method.

drawn are sent to the hardware. When the computations are done in the vertex program, every potential stipple (vertex) for a given surface is sent to the GPU and the density calculation is performed per stipple. Additionally, an index value, ranging from one to the total number of potential vertices, is needed for each stipple. If this index value is greater than the calculated density for that vertex, fast depth culling is used to eliminate further processing of the vertex by adding a large offset to the homogeneous z coordinate of the vertex.

Although this vertex program calculation is faster, it requires redundant calculations compared to the software (CPU) implementation because the density calculations are performed once per stipple (vertex) instead of once per surface (triangle). This redundancy, however, removes the dependency within the set of potential stipples for a given surface on a single calculation (density calculation). Independent stipple calculation can be utilized for even greater performance on the latest, and future, generations of graphics boards that feature multiple parallel vertex processing pipelines (e.g., the Nvidia GeForce4).

4 Results

The results of the stippling system can be seen in Figures 2 through 9 Our performance information is for 1280x1024 rendering on a PC with dual Intel Xeon 2.2 GHz processors and a GeForce3 Ti 500 graphics board. Table 1 shows the performance of the system on several different models, demonstrating that interactive stipple rendering can be performed for useful-sized models on current graphics hardware. This table shows that currently, silhouette rendering occupies a significant portion of the rendering time and also that vertex programs can speed up the stipple rendering by up to 100%, depending on the model. The extrinsic distance enhancements shown in Figures 3 and 4 demonstrate that the system can create appropriate stipple renderings of models at a wide range of scales and can apply these enhancement techniques to different objects within the same scene. As the projection of the object becomes smaller, the extrinsic distance enhancement automatically switches techniques. For a particular scale, the distance enhancement maintains consistent perceived shading by modulating the number of stipples. The intrinsic distance enhancement shown in Figure 7 can create subtle or exaggerated depth effects for illustrative rendering and artistic results. Both of our silhouette techniques enhance the quality of the final images and show a range of artistic styles that can be conveyed with the stipple rendering system.

5 Conclusions

Stippling is one of the most important and effective methods for technical illustration. We have shown that point-based stipple rendering can be

Dataset Name	*Number of Polygons*	*Stipple Only*		*Stipple & Silhouette*	
		Vertex Program (fps)	*No Vertex Program (fps)*	*Vertex Program (fps)*	*No Vertex Program (fps)*
horse	13,352	60.39	60.39	30.19	30.19
low-res dragon	45106	32.05	30.4	15.60	15.24
bunny	69451	30.12	20.07	12.03	11.89
skeleton1	124018	20.66	12.08	10.16	7.53
skeleton2	522567	4.65	2.60	1.94	1.50
hand	654666	4.00	2.07	1.88	1.32
dragon	871414	2.99	1.50	1.32	0.91

Table 1: Running times (frames per second) for different rendering techniques on several example polygonal models.

used to create effective and illustrative renderings of surface-based objects at interactive rates and across multiple scales using the programming capabilities of the latest generation of graphics hardware. The addition of different hardware silhouette techniques enhance the final renderings and create different artistic styles in the renderings. While the current system is very effective, we will further explore optimizing the performance to allow interactive rendering of even more complex models.

6 Acknowledgments

This material is based upon work begun by Chris Morris and is supported by the National Science Foundation under Grants: NSF ACI-0081581, NSF ACI-0121288, NSF IIS-0098443, NSF ACI-9978032, NSF MRI-9977218, NSF ACR-9978099, and the DOE VIEWS program.

References

[1] Oliver Deussen, Stefan Hiller, Cornelius van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3):41–50, August 2000. ISSN 1067-7055.

[2] Bruce Gooch, Peter-Pike Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. Interactive technical illustration. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[3] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 517–526. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000. ISBN 1-58113-208-5.

[4] Elaine R.S. Hodges. *The Guild Handbook of Scientific Illustration*. John Wiley and Sons, Inc., New York, 1989. ISBN 0-471-28896-9.

[5] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report 85-022, University of North Carolina-Chapel Hill Computer Science Department, January 1985.

[6] One-pass silhouette rendering with geforce and geforce2. Technical report, June 2000. http://developer.nvidia.com/.

[7] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 579–584. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[8] Ramesh Raskar and Micheal Cohen. Image precision silhouette edges. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[9] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108, Orlando, Florida, July 1994. ACM SIGGRAPH / ACM Press. ISBN 0-89791-667-0.

[10] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 469–476, New Orleans, Louisiana, August 1996. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1.