EUROGRAPHICS 2002 / I. Navazo Alvaro and Ph. Slusallek
(Guest Editors)

Rendering Artistic Line Drawings Using Off-the-Shelf 3-D Software

J. Loviscach

Fachbereich Elektrotechnik und Informatik, Hochschule Bremen, Bremen, Germany

Abstract

Most commercial 3-D software packages used for animated films and still picture production offer merely rudimentary support for non-photorealistic rendering. However, nearly all of these packages possess interfaces which allow the user to add custom shader programs. We present a solution which uses a custom shader combined with post-processing software in order to draw outlines and creases of 3-D scenes in a way which simulates artistic tools like pencils and ink brushes.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms

1. Introduction

Aside from artistic uses, pen-and-ink renderings of 3-D scenes are often created to increase the comprehensibility of illustrations and to reduce technical demands—for instance in print or electronic media. However, the majority of commercial 3-D software packages—neither low-range applications nor high-end studio solutions—do not include satisfying solutions for non-photorealistic renderings. Even packages which include cartoon-style renderers such as Hash Animation:Master and NewTek Lightwave 3D(R) produce only simple outlines and flat-color fills. Software add-ons to other applications rarely improve on this; most just manage to give other 3-D software packages a basic level of pen-and-ink rendering. There are very few exceptions from this rule, such as the "Sketch Designer" of Curious Labs' Poser(R) 4 and the plug-in solution NPR1 Reyes(R) from Infografica.

Since virtually all 3-D software, commercial or not, allows to define custom shaders, we developed a solution using a such a shader to deliver geometry data to a post-processing application. This shader renders depth data and direction of the normal vector into the picture. It can easily be implemented for different 3-D software packages. Our implementation prototype has been written for Maxon Cinema 4D XL; it consists of merely 200 lines of Maxon's Java™-like scripting language C.O.F.F.E.E.

Most of the work is done by our post-processing software (see figure 1). In an attempt to avoid potential trademark conflicts, it has been named "12point0" for the atomic weight of carbon, the main ingredient of pencil leads. This software receives single pictures or animated picture sequences which

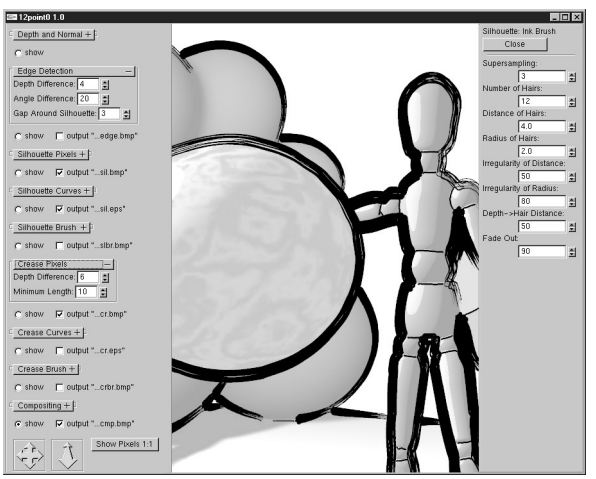

Figure 1: *The post-processing software "12point0". The graphical user interface is built with OpenGL(R) as well as the GLUT and GLUI libraries.*

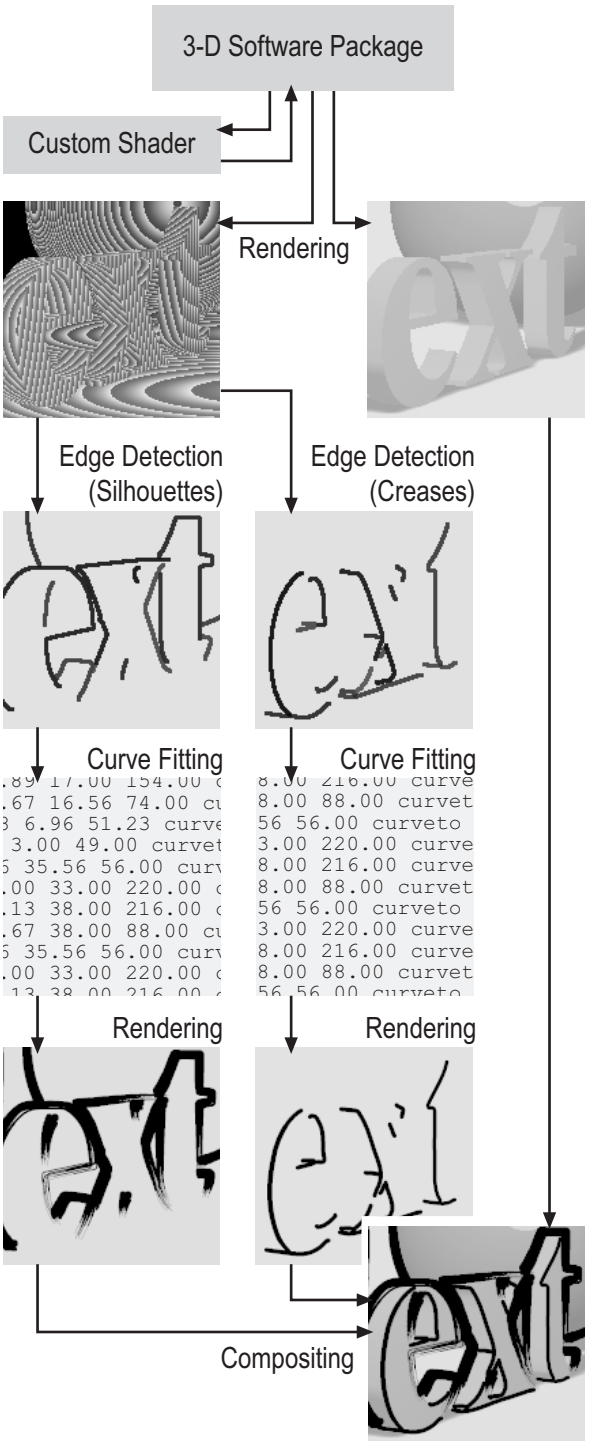

Figure 2: *Data flow and intermediate products of the system.*

have been rendered with help of the custom shader. This data then undergoes four processing steps (see figure 2):

1. extraction of object silhouettes and creases
2. fitting of Bézier curves to the extracted pixel sequences
3. rendering of silhouette and crease curves in artistic styles, optionally with different styles for both types of curves, e. g. creases thinner than the surrounding silhouettes
4. compositing the curves with a regular rendering done by the 3-D software

The final composite is stored as a single image file or as a sequence of image files. The Bézier curves, even though they are an intermediate result, can be written to disk as well. They are exported as EPS vector graphics file so that they can serve as a starting point for usual illustration software.

Our prototype of the post-processing software is implemented in ISO-standard C++ using OpenGL(R), linked to the libraries GLUT and GLUI for the graphical user interface. This should facilitate porting the software from its present platform Microsoft(R) Windows(R) to other systems.

The next section outlines previous work done in this area: section 3 describes how the silhouettes and creases are extracted and approximated by curves; section 4 contains details about artistic style rendering. In section 5 we discuss strengths and issues of the approach. Section 6 closes with ideas of how to improve the presented solution.

2. Previous Work

Saito and Takahashi[20] demonstrate image-precision silhouette generation by using depth data for every pixel, rendered along with RGB data and stored in a "G"-buffer. They detect creases by calculating second-order differences of depth. Storing the surface parameters u, v and the direction of the normal vector in the G-buffer, they can apply curved hatching to shade 3-D objects.

Winkenbach and Salesin[24] convert a 3-D scene into a 2-D mesh representation to construct object outlines. The paths of outline and texture strokes (placed according to surface parametrization) are, however, initially generated as 3-D objects. Gooch et al.[8] propose several algorithms for silhouette extraction, most prominently a hardware solution which uses the OpenGL(R) stencil buffer, and a software solution using a Gauss map, much like Benichou and Elber[1]. Another OpenGL(R)-based solution for drawing silhouettes has been put forward by Raskar and Cohen[17]. Recently, Raskar[18] has extended these ideas to treat creases, too.

Lake et al.[11] present a real-time system which produces flat cartoon colors or pencil sketches with rectilinear hatching textures. Their silhouette extraction detects which edges belong to both a front- and to a back-facing polygon. Applying curvature-driven textures, they render these polygonal lines as rounded curves. Hertzmann and Zorin[9] compute silhouettes of subdivision surfaces using dual surfaces and

hatch them along smoothed versions of the principal curvature directions. Rössl and Kobbelt[19] also perform hatching along these directions. However, they store the two principal curvature directions in a G-buffer, along with the normal vectors they use to detect silhouettes. Deussen et al.[4] apply OpenGL(R) clipping to extract hatching lines with image precision by merely approximating the principal curvature directions.

Northrup and Markosian's[12] randomized silhouette extraction algorithm uses spatial and temporal coherence to speed up the process. The found edges are connected to long paths and drawn like the skeletal strokes of Hsu et al.[10] With the tree illustration system of Deussen and Strothotte[6], stylized lines are possible as well. They use image precision silhouette extraction with a 16 bit depth buffer to render trees. The result is vectorized using the least square fitting method. An object ID buffer can be applied to vectorize each primitive separately.

One of the first realistic simulations of graphic tools is Strassmann's[22] ink painting by moving a linear bristle brush along cubic spline strokes. The bristles leave a trail of color, spread according to pressure and eventually run out of ink along the stroke. A detailed model of pencils and erasers, along with illustration techniques, has been developed by Sousa and Buchanan[21]. A physical fluid model forms the basis of the watercolor simulation developed by Curtis et al.[2]. In another work, Curtis[3] achieves sketchy illustrations by rendering randomly dragged particles as line segments.

Several authors propose system architectures: Mohr and Gleicher[14] describe how to add stylized rendering capabilities by catching and re-interpreting an application's OpenGL(R) calls. The design of the "Sketch" system by Strothotte et al.[23] separates the modeling work from the non-photorealistic renderer.

3. Curve Extraction

The intention to support as many 3-D software packages as possible naturally leads to an image-precision approach where the geometry data is delivered via pixel-based image files[20]. To use a geometry-based format like 3DS, as implemented by e. g. Masuch et al.[13], requires the off-the-shelf 3-D software to export such a file format without alteration of the objects, their motion, and their deformation. However, this condition is unlikely to be met. This is partly a consequence of the complex modeling, texturing, and animation tools provided by the software. Most such effects cannot easily be expressed in a common file format shared by different packages.

For each pixel, our solution stores the depth (more precisely, the ray length) and the direction of the normal vector of the underlying geometry. With this approach, there is no need to evaluate the second derivatives of depth data, which can be very inaccurate if the pixel resolution is not very high in comparison to the sizes of the pictured objects. Instead, we use differences of depth data only to detect silhouettes; creases are found using differences of normal vectors. It also helps to know the normal vector to distinguish silhouettes from the steep depth variation which occurs in strongly foreshortened objects.

Normal vectors help with the extraction and are rather easy to obtain: Typical 3-D software packages make them readily available for custom shaders. Even a hardware-accelerated solution could use them: State-of-the-art consumer graphics cards allow the use of normal vectors to calculate programmer-defined shaders.

Depth and normal vector data could be stored in additional graphics buffers along with the usual RGB data produced by the 3-D software package. However, to keep our solution as flexible as possible, we have chosen the following alternative: The scene is rendered twice in the 3-D software package, once writing regular RGB data, the second time encoding depth and normal vector data into 24 bit pseudo-RGB with the help of a custom shader (see figure 3).

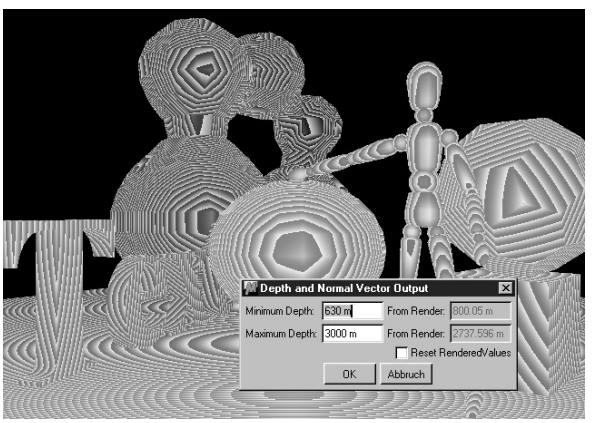

Figure 3: *The custom shader with its minimalist user interface encodes the depth into the 16 bit provided by the red and green picture channels; the direction of the normal vector is encoded in the 8 bits of the blue channel.*

The 16 bits of R and G are used for depth data, while the 8 bits of B contain information about the normal vector. To fully exhaust the 16 bits' range for depth, the user can adjust minimal and maximal depth in the custom shader. The shader reports the actual minimal and maximal depth of the scene to the user, but instead of using these values directly it relies on manual input. This is useful e. g. for animations where an object flies into seemingly infinite depths.

To encode the normal vector in eight bits, it is rounded to the nearest of 253 fixed directions. These are distributed almost uniformly across the half sphere pointing at the viewer. The resulting error is smaller than 8 angular degrees, which means that creases above 16 angular degrees will still be detected, which suffices for this application.

The pixels with discontinuities in depth value and/or normal vector are collected and used as the starting point for the determination of the objects' silhouettes and creases. These collections are subjected to a standard thinning algorithm. The next processing step identifies curves formed by the remaining pixels. It makes use of the known depth data: The depths of neighboring pixels on the same silhouette or crease line cannot be too different.

In the last extraction step, the paths—up to this point collections of pixels—are approximated by Bézier curves. This not only results in a great reduction in data and allows EPS export, but it is also the key feature to draw softly bowed strokes instead of the exact polygonal shapes of the 3-D scene. Fitting Bézier curves is a standard problem in computer graphics[15]. However, for reasons of robustness and speed we employ an new experimental algorithm which does not work iteratively. This will be published separately.

4. Artistic Rendering

The Bézier curves generated for silhouettes and creases can be drawn using a selection of simulated art tools. At the time of writing, our software offers a simple pen, an ink brush, a pencil, and sketchy-looking wiggly lines. These are all first rendered into a finer grid (using up to 4×4 supersampling) and then sampled down to the actual resolution, taking into account the gamma value of a typical display. To draw lines into the fine grid, circular disks are placed along the Bézier curves. By considering the derivative vector of the curve, the dot centers of the circular discs are placed approximately one pixel apart on the finer grid. Since the width of the curve is not necessarily constant along its length, this drawing process cannot easily be sped up by an algorithm like Posch and Fellner's[16].

Figure 4: *The silhouettes have been rendered with a simulated ink brush, the creases with a simple pen. The lines were composited with a pastel colored rendering of the scene created with the 3-D software package.*

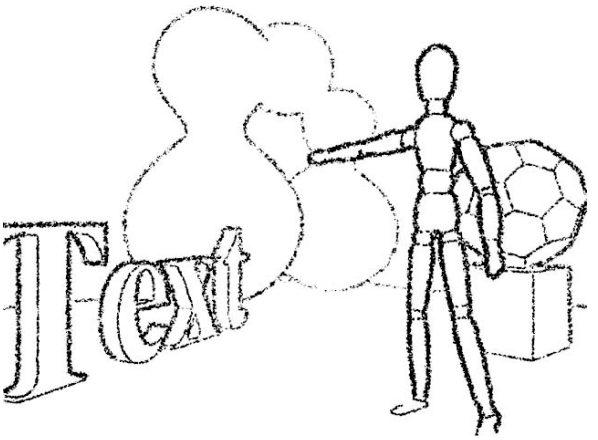

Figure 5: *Silhouettes and creases have been rendered with a pencil. Note how the spatial depth controls both pressure and stroke width.*

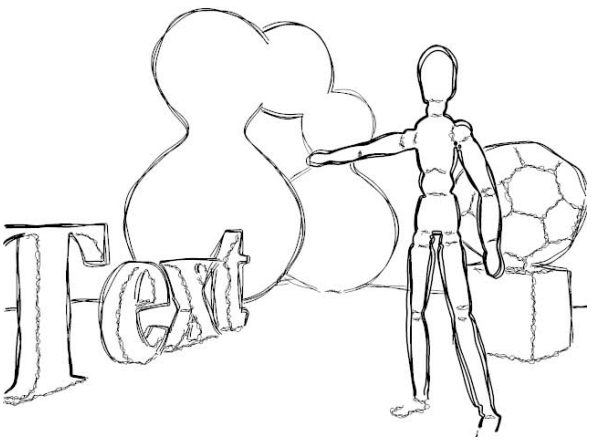

Figure 6: *Silhouettes and creases have been rendered as multiple wiggly lines. The wiggle deformations depend on the screen position and are therefore consistent across image sequences.*

Most of our virtual art tools allow the depth data to influence the size of the dots, so that far-away objects are drawn with thinner lines. The ink brush (see figure 4) uses an adjustable number of dots of irregular but fixed sizes and positions drawn around the position on the curve. Similar to Strassmann's "Hairy Brushes"[22], this creates the look of a brush with several bunches of bristles clinging together. To simulate the ink depletion along a brush stroke, the sizes of the dots are diminished accordingly.

The pencil simulation (see figure 5) can be viewed as a much-simplified version of the model developed by Sousa and Buchanan[21]. It uses a paper texture produced by linearly interpolated lattice noise added over two octaves (see

e. g. Ebert et al.[7]). A pixel is set if the "pressure" exceeds the local value of the paper texture. This pressure depends on the depth—therefore distant objects are drawn in a lighter style—and on the distance to the center line, brightening up the edges of the stroke.

The "wiggly lines" (see figure 6) produce a sketch-style look by drawing a bundle of curves where each is distorted slightly. These curves are rendered as described above. However, each dot is displaced by an pseudo-random offset vector. In order to produce smooth curves, this offset vector is generated by lattice noise interpolated quadratically across the image. Each curve of the bundle is deformed by a noise of its own, to create differing distortions. The wiggling style can by controlled through the noise's grid size—visually, the wavelength of the wiggles—and the noise's influence on the curves.

5. Discussion

Figures 4, 5, and 6 illustrate the performance of our solution with a typical scene. These results show a clear improvement over the cartoon rendering functionality of most of today's 3-D software packages. By using the post-processing application and only developing a small custom shader, this solution can easily be adapted to other packages than our test case, Maxon's Cinema 4D XL. However, the user must deactivate dithering and antialiasing when rendering with the custom shader—otherwise the data hidden in pseudo-RGB would be destroyed.

The resolution of 16 bits for the depth and 8 bits for the normal vector proves to be sufficient for cartoon-like post-processing. Photorealistic rendering and other soft shading effects would need much higher precision. To fulfill the demands of such applications, one could make use of the 48 bit RGB output of some off-the-shelf 3-D software. Another option would be to render two differently encoded sets of 24 bit RGB data in order to output 48 bits of data per pixel.

The silhouette detection is improved by taking the normal vector into account. This helps to solve ambiguities when 3-D faces lie almost along a viewing line, causing a high depth variation from pixel to pixel even though there is no silhouette.

When objects are in close contact, our algorithm doesn't separate them using silhouette lines, but with crease lines instead. This can be seen e. g. in the right foot of the puppet of figure 4. If such an effect is not intended, one could use the same drawing tool for silhouettes and creases, thereby eliminating the difference. Another solution could be to store an object ID for each pixel. Neighboring pixels with different object IDs would then indicate a silhouette.

Even though the source code has been optimized for comprehensibility instead of speed, the post-processing time per picture for the ink brush rendering of the test scene (640×480 pixels) with 4×4 antialiasing amounts to 18 seconds on a 500 MHz Pentium(R)-III PC. By storing and reusing intermediate results, on a Gigahertz-class PC parameter changes can be visualized almost interactively.

When processing image sequences, rendering variations from frame to frame can cause the animation to look fidgety. This is a lesser problem for the pen, pencil, and wiggle line drawing tools. The ink brush tool, however, is of limited use for animation because the thick, inky end of a curve can jump around an object or the brush stroke can even alter its direction.

6. Summary and Future Work

We have demonstrated a solution to create artistically rendered line-drawings which can easily be implemented as an add-on for most 3-D software packages. Work is underway to adapt this software to different operating systems, especially to Apple Mac OS X, and to different 3-D software packages, starting with Alias|Wavefront(R) Maya™ and NewTek Lightwave 3D(R).

It would be rewarding to spend further development work on these extensions:

- Each object could be drawn with an individually styled outline, e. g. by storing an object ID along with geometry data in the G-buffer[6].
- A custom shader could apply hatching[19] or stippling[5] to the 3-D objects instead of soft, continuous shading.

Acknowledgements

The author thanks Gerald Himmelein for helpful discussions.

References

1. F. Benichou, G. Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. Proc. Pacific Graphics 1999, pp. 60–69, 1999 2
2. C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, D. H. Salesin. Computer-Generated Watercolor. *SIGGRAPH '97 Conference Proceedings, Annual Conference Series*, pp. 421–430, 1997. 3
3. C. Curtis. Loose and Sketchy Animation. *SIGGRAPH '98: Conference Abstracts and Applications, Annual Conference Series*, p. 317, 1998. 3
4. O. Deussen, J. Hamel, A. Raab, S. Schlechtweg and Th. Strothotte. An Illustration Technique Using Hardware-Based Intersections and Skeletons. *Proc. GI 1999: Graphics Interface*, pp. 175–182, 1999. 3
5. O. Deussen, S. Hiller, C. v. Overveld and Th. Strothotte. Floating Points: A Method for Computing Stipple Drawings. *Computer Graphics Forum (Eurographics 2000 Proc.)*, **19**(3):40–51, 1999. 5

6. P. Deussen and Th. Strothotte Computer-Generated Pen-and-Ink Illustration of Trees. *SIGGRAPH 2000 Conference Proceedings, Annual Conference Series*, pp. 13–18, 2000. 3, 5

7. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley. *Texturing & Modeling*. Academic Press, second edition 1998. 5

8. B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley and R. Riesenfeld. Interactive Technical Illustration. *Proc. 1999 Symposium on Interactive 3D Graphics*, pp. 31–38, 1999. 2

9. A. Hertzmann and D. Zorin. Illustrating Smooth Surfaces. *SIGGRAPH 2000 Conference Proceedings, Annual Conference Series*, pp. 517–526, 2000. 2

10. S. C. Hsu, I. H. H. Lee and N. E. Wiseman. Skeletal Strokes. *Proc. UIST '93: ACM SIGGRAPH & SIGCHI Symposium on User Interface Software & Technology*, pp. 197–206, 1993. 3

11. A. Lake, C. Marshall, M. Harris and M. Blackstein. Stylized Rendering Techniques For Scalable Real-Time 3D Animation. *Proc. NPAR 2000: First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 13–22, 2000. 2

12. J. D. Northrup, L. Markosian. Artistic Silhouettes: A Hybrid Approach. *Proc. NPAR 2000: First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 31–37, 2000. 3

13. M. Masuch, St. Schlechtweg and B. Schönwälder. daLi!—Drawing Animated Lines! *Proc. Simulation and Animation '97*, pp. 87–96, 1997. 3

14. A. Mohr and M. Gleicher. Non-Invasive, Interactive, Stylized Rendering. *Proc. I3D 2001: ACM Symposium on Interactive 3D Graphics*, pp. 175–178, 2001. 3

15. M. Plass and M. Stone. Curve-Fitting with Piecewise Parametric Cubics. *Computer Graphics*, **17**(3):229–239, 1983. 4

16. K. C. Posch and W. D. Fellner. The Circle-Brush Algorithm. *ACM Transactions on Graphics*, **8**(1):1–24, 1989. 4

17. R. Raskar and M. Cohen. Image Precision Silhouette Edges. *Proc. I3D 1999: ACM Symposium on Interactive 3D Graphics*, pp. 135–140, 1999. 2

18. R. Raskar. Hardware Support for Non-photorealistic Rendering. *Proc. of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 410–415, 2001. 2

19. Ch. Rössl and L. Kobbelt. Line-Art Rendering of 3D-Models. *Proc. Pacific Graphics 2000*, pp. 87–96, 2000. 3, 5

20. T. Saito and T. Takahashi. Comprehensive rendering of 3-d shapes. *ACM Computer Graphics (Proc. SIGGRAPH '90)*, **24**(4):197–206, 1990. 2, 3

21. M. C. Sousa and J. W. Buchanan. Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. *Computer Graphics Forum (Eurographics '99 Proc.)*, **18**(3):195–207, 1999. 3, 4

22. S. Strassmann. Hairy Brushes. *ACM Computer Graphics (Proc. SIGGRAPH '86)*, **20**(4):225–232, 1986. 3, 4

23. Th. Strothotte, B. Preim, A. Raab, J. Schumann and D. R. Forsey. How to Render Frames and Influence People. *Computer Graphics Forum (Eurographics '94 Proc.*, **13**(3):455–466, 1994. 3

24. G. Winkenbach and D. H. Salesin. Rendering Parametric Surfaces in Pen and Ink. *SIGGRAPH '96 Conference Proceedings, Annual Conference Series*, pp. 469–476, 1996 2