Recent Advances in Non-Photorealistic Rendering for Art and Visualization

Course Notes for SIGGRAPH 2002

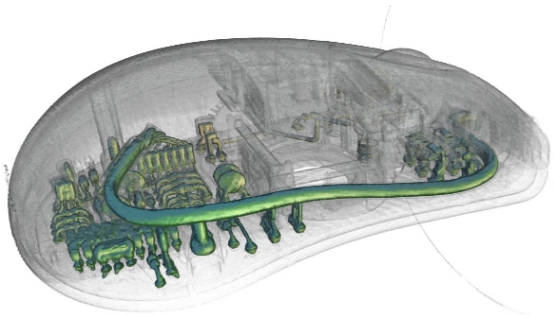
Organizer
Kwan-Liu Ma
University of California, Davis

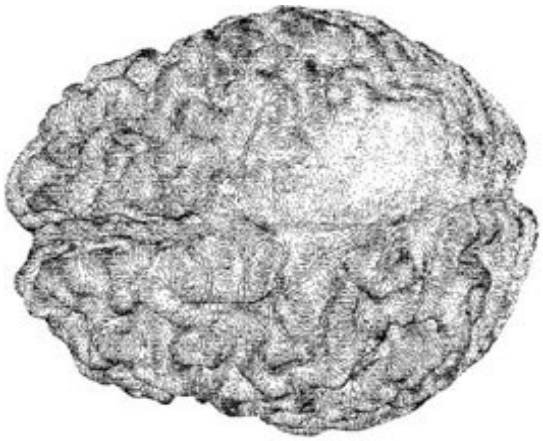
Lectures

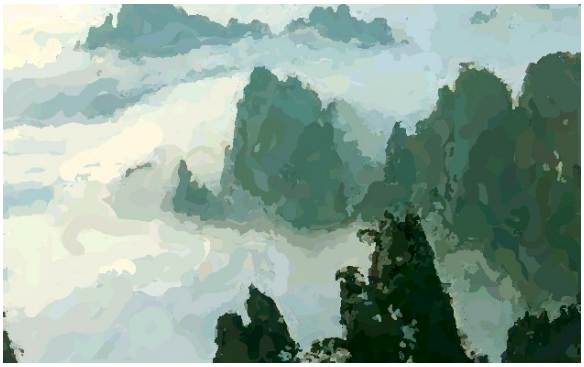
Aaron Hertzmann
University of Washington

Victoria Interrante
University of Minnesota

Eric B. Lum
University of California, Davis







COURSE ORGANIZER

Kwan-Liu Ma
Department of Computer Science
University of California at Davis
One Shields Avenue, Davis, CA 95616
ma@cs.ucdavis.edu
http://www.cs.ucdavis.edu/~ma

LECTURERS

Aaron Hertzmann
GRAIL -Graphics and Imaging Laboratory
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
hertzmann@cs.washington.edu
http://www.cs.washington.edu/home/hertzman

Victoria Interrante
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
interran@cs.umn.edu
http://www.cs.umn.edu/~interran

Eric B. Lum
Department of Computer Science
University of California at Davis
One Shields Avenue, Davis, CA 95616
lume@cs.ucdavis.edu
http://graphics.cs.ucdavis.edu/~lume

BIOGRAPHIES

Aaron Hertzmann received his BA in Computer Science and Art & Art History from Rice University in 1996, and his MS and Ph.D. in Computer Science from New York University, in 1998 and 2001, respectively. In 2003, he will join the faculty of the University of Toronto computer science department. His research interests include non-photorealistic rendering and machine learning for computer graphics.

Aaron has worked at Microsoft Research, Mitsubishi Electric Research Lab, Interval Research Corporation and at NEC Research Institute. He is currently serving on the program committee for NPAR 2002, the second International Symposium on Non-Photorealistic Animation and Rendering.

Aaron has presented four papers at SIGGRAPH, including three on non-photorealistic rendering. He spoke in the SIGGRAPH 1999 course on Non-Photorealistic Rendering.

Victoria Interrante received her PhD in 1996 from the University of North Carolina at Chapel Hill, where she studied under the joint direction of Dr. Henry Fuchs and Dr. Stephen Pizer. From 1996-1998 she worked as a staff scientist at ICASE, a non-profit research center operated by the Universities Space Research Association at NASA Langley.

Her research focuses on the application of insights from perceptual psychophysics, art, and illustration to the design of more effective techniques for visualizing data. Some of her current projects include: the study of texture's effect on shape perception and texture synthesis for shape representation, the study of texture perception and classification for texture synthesis in multivariate data visualization and uncertainty representation, the segmentation and tracking of vortical structures in turbulent boundary layer data, and feature identification for gaze direction determination from uncalibrated photographs. Her work involves collaborations with researchers from a variety of departments at the University of Minnesota, including Electrical Engineering, Aerospace Engineering, Architecture, Mechanical Engineering, and Child Development. In 1999, she received a Presidential Early Career Award for Scientists and Engineers (PECASE) for her work on perceptual issues in data visualization.

She has participated in SIGGRAPH courses in 1997, 1998, 1999, and 2001 with a focus on the fundamental issues of visual perception for image synthesis and visualization.

Eric Lum received a B.S. and an M.S. both in Electrical Engineering from UCLA in 1997 and 1999, respectively. He is currently pursuing a Doctorate in Computer Science at the University of California at Davis. His research interests include non-photorealistic rendering and visualization, interactive visualization, and animation.

Eric has developed several rendering algorithms with the aim to enhance perception of subtle information for better understanding shape, structure and relationship. One algorithm is inspired by watercolor paintings and the other makes extensive use of motion cues. He recently presented his research results on interactive NPR at NPAR 2002. Eric has also participated in a joint effort between UC Davis and two Japanese research institutions for the development of a volume graphics cluster project.

Kwan-Liu Ma received his Ph.D. from the University of Utah in 1993. His research interests includes large-scale data visualization, user interface designs, and high performance computing. One of his current research projects is in NPR with a focus on hardware-assisted techniques. The excellent high performance and quality results he and his students have obtained motivates him to organize this course.

Kwan-Liu organized and taught courses for SIGGRAPH ’97, ’98, and ’99. The SIGGRAPH ’97 course’s contribution to CAL was selected as the most outstanding contribution.

He served as co-chair for the 1997 IEEE Symposium on Parallel Rendering, Case Studies of the IEEE Visualization Conference in both 1998 and 1999, and the first NSF/DOE Workshop on Large-Scale Data Visualization (May 1999). He also served as the guest editor for a theme issue (July 2001) of IEEE CG&A on Large-Scale Data Visualization. Presently, he serves as the secretary of the IEEE Technical Committee on Visualization and Graphics. In October 2000, Kwan-Liu received the PECASE (Presidential Early Career Award for Scientists and Engineers) award. In July 2001, he received the Schlumberger Foundation Technical Award.

Table of Contents

1. Introduction - *Kwan-Liu Ma* **1-1**

2. NPR Techniques for Scientific Visualization - *Victoria Interrante* **2-1**
1 Introduction 2-1
2 Motivation and Background 2-1
3 Case Studies from my Research 2-2
References 2-2
Slides 2-5

3. NPR Techniques for Art - *Aaron Hertzmann* **3-1**
1 Introduction 3-1
2 Stroke-Based Rendering: Stating the Problem 3-3
3 Optimization Algorithms 3-5
3.1 Voronoi algorithms 3-5
3.1.1 Lloyd's method 3-5
3.1.2 Variations on Lloyd's method for SBR 3-7
3.2 Trial-and-error algorithms 3-9
3.2.1 Streamline visualization by trial-and-error 3-10
3.2.2 Painterly rendering by trial-and-error 3-11
4 Greedy Algorithms 3-13
4.1 Single-point strokes 3-13
4.1.1 Single-layer painterly rendering 3-13
4.1.2 Multiple-layer painterly rendering 3-14
4.2 Long, curved strokes 3-16
4.2.1 Painterly rendering with long, curved strokes 3-16
4.2.2 Pen-and-ink and other curve tracing algorithms 3-21
4.3 Globally-greedy 3-22

5 Stroke Models 3-23
5.1 Long, curved strokes with triangle strips 3-23
6 Limitations of the Energy-Minimization Approach 3-24
7 Related Topics 3-25
8 Summary 3-26
References 3-26

4. Interactive NPR - *Eric Lum & Kwan-Liu Ma* **4-1**
1 Introduction 4-1
1.1 Motivation 4-1
1.2 Achieving interactivity 4-2
2 Non-Photorealistic Surface Rendering 4-3
2.1 Silhouette edge rendering 4-3
2.2 Non-photorealistic illumination 4-5
2.3 Non-photorealistic texture 4-5
3 Non-Photorealistic Volume Rendering 4-10
3.1 Volume rendering with graphics hardware 4-10
3.2 Hue varied shading 4-11
3.3 Silhouette edge illustration 4-12
3.4 Surface enhancement 4-13
3.5 Color based on position 4-13
3.6 Multiple rendering parameters 4-14
3.7 A complete example 4-15
3.8 Parallel rendering and performance 4-16
4 Conclusion 4-18
References 4-18

5. Final Remarks 5-1

6. Appendix: Collection of Papers 6-1

1. "*Texture Synthesis for 3D Shape Representation*", G. Gorla, V. Interrante, and G. Sapiro, (to appear) IEEE Transactions on Visualization and Computer Graphics, 2002.

2. "*Line Direction Matters: An Argument for the Use of Principal Direction in 3D Line Drawing*", A. Girshick, V. Interrante, S. Haker, and T. Lemoine, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France,

June 5-7, 2000, pages 43-52.

3. "*Understanding Errors in Approximating Principal Direction Vectors*", J. Goldfcather and V. Interrante, (to appear) ACM Transactions on Graphics, 2003.

4. "*Paint By Numbers: Abstract Image Representations*", P. Haeberli, in SIGGRAPH '90 Conference Proceedings, August 1990, pages 207-214.

5. "*Image Analogies*", A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, in SIGGRAPH 2001 Conference Proceedings, August 12-17, 2001, pages 327-340.

6. "*Illustrating Smooth Surfaces*", A. Hertzmann and D. Zorin, in SIGGRAPH 2000 Conference Proceedings, July 23-28, 2000, pages 517-526.

7. "*Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*", K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, in SIGGRAPH '99 Conference Proceedings, August 8-13, 1999, pages 277-286.

8. "*Harnessing Natural Textures for Multivariate Visualization*", V. Interrante, Visualization Viewpoints, Editors: Theresa-Marie Rhyne and Lloyd Treinish, IEEE Computer Graphics and Applications, November/December 2000, pages 6-11.

9. "*Creating Evenly-Spaced Streamlines of Arbitrary Density*", B. Jobard and W. Lefer, in Proceedings of the 8th Eurographics Workshop on Visualization in Scientific Computing, 1997, pages 45-55.

10. "*Interactive Artistic Rendering*", M. Kaplan, B. Gooch, and E. Cohen, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France, June 5-7, 2002. pages 67-74.

11. "*Art-Based Rendering of Fur, Grass, and Trees*", M. A. Kowalski, L. Markosian, J. D. Northrup, and L. Bourdev, R. Barzel, L. S. Holden, J. F. Hughes, in SIGGRAPH '99 Conference Proceedings, Augst 8-13, 1999, pages 433-438.

12. "*Artistic Silhouettes: A Hybrid Approach*", J. D. Northrup and L. Markosian, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France, June 5-7, 2000, pages 31-37.

13. "*Interactive Pen-and-Ink Illustration*", M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin, in Proceedings of SIGGRAPH '94, July 24-29, 1994, pages 101-108.

14. "*Orientable Textures for Image-Based Pen-and-Ink Illustration*", M. P. Salisbury, M. T. Wong, J. F. Hushes, and D. H. Salesin, in SIGGRAPH '97 Conference Proceedings, August 1997, pages 401-406.

15. "*Weighted Voronoi Stippling*", A. Secord, in Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002), Annecy, France, June 3-5, 2002.

16. "*Fine Tone Control in Hardware Hatching*", M. Webb, E. Praum, A. Finkelstein, H. Hoppe, in Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002), Annecy, France, June 3-5, 2002.

17. "*Computer-Generated Pen-and-Ink Illustration*", G. Winkenbach and D. H. Salesin, in Proceedings of SIGGRAPH '94, July 24-29, 1994, pages 91-100.

Introduction

Kwan-Liu Ma
University of California, Davis

There is a growing interest in Non-Photorealistic Rendering (NPR) methods because of its expressive power for illustrating shapes and spatial relationships as well as for generating artistic drawings and paintings. How we choose to portray a data set can have a significant effect on how accurately and efficiently we communicate with the viewers the information we seek to reveal. In many cases, NPR has been shown to be more effective than photorealistic rendering in communicating subtle information about physical structures or phenomena. We shall see a growing use of NPR for scientific visualization and illustrations as realtime NPR becomes available. Recent advances in painterly rendering have demonstrated that certain artistic styles can be mimicked. The ability to automatically generate arts and illustrations for more effectively communicating with scenes, ideas, or actions would allow content creators for education, films or video games to attain a new level of creativity.

We thus believe a large SIGGRAPH audience will benefit from a course on non-photorealistic approach for scientific visualization and artistic rendering. Courses on NPR related topics have been offered before, two in SIGGRAPH '99 and one in SIGGRAPH 2001. This course is unique since it addresses some of the most relevant aspects of the theoretical basis, software algorithms, hardware-assisted techniques, and applications for NPR. We have designed four concise lectures to motivate the audience, inform them with the state-of-the-art techniques and their applications, and offer them pointers for further research.

In the first lecture, Victoria Interrante will give an overview of the use of NPR rendering techniques in *scientific visualization*, followed by a presentation of the design, implementation and evaluation of several specific NPR methods drawn from her recent research. She will use hand-drawn examples from scientific application areas, such as medical line drawings and archaeological illustrations, as well as some examples from art/illustration to show when and how non-photorealistic representations can be effective in illustrating shapes and structures. She will present mathematical methods for calculating preferred stroke directions over a polygonally-defined mesh, and techniques for synthesizing a high-resolution oriented texture over a surface mesh. She will also discuss results from two controlled observer experiments intended to investigate the effects of surface texture characteristics on 3D shape perception.

In the second lecture, Aaron Hertzmann will describe *stroke-based NPR* methods. This area incorporates a wide variety of methods for pen-and-ink, painterly, and visualization renderings of images, 3D and video, and can be quite challenging to the beginner. In order to help make sense of things, he will present a unified framework for stroke-based NPR and draw attention to the common principles of the area. In particular, stroke-based rendering describes methods where discrete strokes are placed

in order to match some predetermined constraints, such as matching a predetermined image color or intensity. The specific algorithms used depend directly on the form of these constraints.

In the third lecture, Aaron Hertzmann will describe recent advances in *example-based NPR*, where NPR algorithms are designed based on hand-made examples. For example, a painting image filter might be created based on a scanned painting made by a famous artist. This approach is far easier than explicit methods (such as stroke-based rendering), since it is very difficult to write a function that describes the technique of a famous artist; however, only some aspects of a style may be captured. He will survey several recent methods developed in this area, and describe the method of ”Image Analogies” in detail.

Finally, Eric Lum will present a suite of *hardware-accelerated NPR* techniques, making extensive use of the advanced features of commodity graphics cards, for interactive visualization of volume data. In addition to the typical view and rendering parameters, each NPR technique adds its own set of parameters that must be specified. Often the user does not know what type of rendering style is desired, only through experimentation can parameters be found suited for their particular application. He will show how *interactive NPR* facilitate this parameter specification process, and how different rendering styles and NPR techniques may be freely mixed with interactive control. Both the benefit and cost of including each NPR technique will be discussed. He will also describe how to use a cluster of PCs to render large-scale volume while maintaining high interactivity and image quality for large-format display.

For the specific approaches covered in the lectures, examples and previous successes, as well as some of their limitations, are discussed. Where techniques are unavailable, or not yet proven, opportunities and promising research directions are offered. In this course notes, each chapter begins with a summary of the corresponding lecture. We also provide comprehensive bibliographies, as well as including a collection of technical reports and previously published papers relevant to the subjects discussed in this course. We hope you find this course notes helpful.

Kwan-Liu Ma
April 05, 2002

NPR Techniques for Scientific Visualization

Victoria Interrante
University of Minnesota

1 Introduction

Visualization research is concerned with the design, implementation and evaluation of novel methods for effectively communicating information through images. For example: How do we determine how to portray a complicated set of data so that its essential information can be easily and accurately perceived and understood? How can we measure the success of our efforts? Where should we look for insight into the science behind the art of effective visual representation?

For many years, photorealism was the gold standard, and the goal in visualization was to achieve renderings of scientific data that were as nearly indistinguishable as possible from a photograph. However in recent years, there has been an upsurge of interest in using non-photorealistic rendering (NPR) techniques in scientific visualization. What does NPR offer to visualization applications? What are some of the important research issues in developing NPR techniques for data visualization? What are the recent advances in this area? My presentation in this half-day course will attempt to survey these topics, beginning with a discussion of the motivation for using NPR in scientific visualization and continuing through an overview of recent research in the development and assessment of NPR techniques for visualization applications.

2 Motivation and Background

Historical examples of the use of 'non-photorealistic rendering' techniques in scientific visualization can be found in the hand-drawn illustrations from scores of textbooks across multiple fields in the sciences [Loe64][Law71][Zwe61][Rid38]. The universal goal in such illustrations was to clarify the subject by emphasizing its most important or significant features, while de-emphasizing extraneous details [Hod89]. Although a photographic depiction has the potential to capture the exact appearance of an object as we actually see it, with subtle and complex details of coloration and

texture fully represented with perfect accuracy (at least for one exemplar), a drawing offers the possibility to clarify structural or conceptual information that may be difficult to perceive even in a very good photograph [Add86][RS56][HM86][FH78][Por81]. Drawings are also useful for portraying information that does not have a physical correlate, or which cannot be captured or represented photographically [Con83]. Finally, by virtue of their inexact and suggestive (as opposed to explicit) character, some kinds of drawings have the potential to embody a certain amount of ambiguity, allowing flexibility in the interpretation of the object or information portrayed [SPRSF94][SSRL96][Las00]. Recently developed computational methods for applying non-photorealistic rendering techniques to the visualization of scientific data [ER00][GGSC98] [GSGSR99][Hea01][IFP95][IG98][KML99][TC00] seek to satisfy similar objectives.

3 Case Studies from my Research

In the attached papers (see appendix) I describe recent results from my own research in the design, implementation and evaluation of non photorealistic rendering techniques for the visualization of 3D surface shape and multivariate data.

4 References

[1] Lucile R. Addington. Lithic Illustration: drawing flaked stone artifacts for publication, The University of Chicago Press, 1986.

[2] Michael L. Connolly. "Solvent-Accessible Surfaces of Proteins and Nucleic Acids", *Science*, **221**(4612): 709-713, August 19, 1983.

[3] David Ebert and Penny Rheingans. "Volume Illustration: Non-Photorealistic Rendering of Volume Models", *Proceedings of Visualization 2000*, pp. 195-202.

[4] D. Fussel and A. Haaland. "Communicating with Pictures in Nepal: results of practical study used in visual education", *Educational Broadcasting International*, **11**(1): 25-31, 1978.

[5] Ahna Girshick, Victoria Interrante, Steve Haker and Todd LeMoine. Line Direction Matters: An Argument for the Use of Principal Directions in 3D Line Drawings, *First International Symposium on Non Photorealistic Animation and Rendering*, pp. 43-52, 2000.

[6] Jack Goldfeather and Victoria Interrante. "Understanding Errors in Approximating Principal Direction Vectors", *to appear*.

[7] Amy Gooch, Bruce Gooch, Peter Shirley and Elaine Cohen. "A Non-Photorealistic Lighting Model for Automatic Technical Illustration", *Proceedings of Acm SIGGRAPH 98*.

[8] Bruce Gooch, Peter-Pike Sloan, Any Gooch, Peter Shirley and Richard Riesenfeld. "Interactive Technical Illustration", *Proceedings of ACM Symposium on Interactive 3D Graphics, 1999*.

[9] Christopher Healey. "Formalizing Artistic Techniques and Scientific Visualization for Painted Renditions of Complex Information Systems", *Proceedings of IJCAI 2001*.

[10] K. Hirsch and D. A. McConathy, "Picture Preferences of Thoracic Surgeons", *Journal of BioCommunications*, pp. 26-30, Winter 1986.

[11] Elaine R. S. Hodges. The Guild Handbook of Scientific Illustration, Van Nostrand Reinhold, 1989.

[12] Victoria Interrante, Henry Fuchs and Stephen Pizer. "Enhancing Transparent Skin Surfaces with Ridge and Valley Lines", *IEEE Visualization '95*.

[13] Victoria Interrante. "Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution", Proceedings of *SIGGRAPH 97*.

[14] Victoria Interrante and Chester Grosch. "Visualizing 3D Flow", *IEEE Computer Graphics and Applications*, **18**(4): 49-53, 1998.

[15] R.M. Kirby, H. Marmanis and D.H. Laidlaw. "Visualizing Multivalued Data from 2D Incompressible Flows Using Concepts from Painting", *Proceedings of IEEE Visualization '99*.

[16] George R. P. Lawrence. Cartographic Methods, Methuen, 1971.

[17] Paul Laseau, Architectural Representation Handbook, McGraw-Hill, 2000.

[18] William E. Loechel. Medical Illustration; a guide for the doctor-author and exhibitor, Charles C. Thomas, 1964.

[19] Kwan-Liu Ma and Victoria Interrante. "Extracting Feature Lines from 3D Unstructured Grids", *IEEE Visualization'97*.

[20] James B. Porter. "Relief Monuments", in The Student's Guide to Archaeological Illustrating, Brian D. Dillon, ed., Institute of Archaeology, University of California, Los Angeles, 1981.

[21] Gillian Rhodes, Susan Brennan and Susan Carey. "Identification and Ratings of Caricatures: implications for mental representations of faces", *Cognitive Psychology*, **19**, pp. 473-497, 1987.

[22] John L. Ridgway. Scientific Illustration, Stanford University Press, 1938.

[23] T. A. Ryan and Carol B. Schwartz, "Speed of Perception as a Function of Mode of Representation", *American Journal of Psychology*, **69**, pp. 60-69, 1956.

[24] Jutta Schumann, Thomas Strothotte, Andreas Raab, and Stefan Laser. "Assessing the Effect of Non-Photorealistic Rendered Images in CAD", *Proceedings of ACM CHI 96*.

[25] Thomas Strothotte, Bernhard Preim, Andreas Raab, Jutta Schumann and David R. Forsey. "How to Render Frames and Influence People", *Computer Graphics Forum (Proc. EUROGRAPHICS)*, **13**(3): 455-466, 1994.

[26] S.M.F. Treavett and M. Chen, "Pen-and-Ink Rendering in Volume Visualization", *Proceedings of Visualization 2000*, pp. 203-210.

[27] Frances W. Zweifel. A Handbook of Biological Illustration, University of Chicago Press, 1961.

Non-Photorealistic Rendering in Scientific Visualization

Victoria Interrante
University of Minnesota

Outline

- Motivation and Objectives
  - What does non-photorealistic representation offer?
- Overview of Recent Work in the Field
- Examples/Applications from My Own Work
  - Artistic enhancements for conveying 3D shape
  - Texture synthesis for multivariate visualization

Why Non-Photorealistic Rendering?

- Let's look at some applications …
- Drawing vs. photo

Visualizing Anatomy

- A photographic depiction captures the exact appearance of the object as we actually see it
- Subtle, complex details of coloration and texture are fully represented, with great accuracy
- A drawing offers the possibility to clarify structural or conceptual information that may be difficult to perceive in even a very good photo.

Photo vs. Drawing

- Hand-drawn illustrations are routinely used to emphasize important features that are difficult to capture in a photograph, while minimizing secondary detail
- Drawings are also useful to portray information that cannot be captured or represented photographically, such as hidden surfaces
- But are drawings always preferable to photos?

Controlled Experiments in Picture Perception

- Are drawings better than photographs for communicating information effectively?

Speed of imitation of finger position, in seconds (mean):

- 0.039 photo
- 0.044 shaded drawing
- 0.070 line drawing
- 0.046 cartoon

Speed of naming open switch, in seconds (mean)

- 0.690 photo
- 0.719 shaded drawing
- 1.169 line drawing
- 0.288 cartoon

T. A. Ryan and Carol B. Schwartz, "Speed of Perception as a Function of Mode of Representation", American Journal of Psychology, 69, pp. 60-69, 1956.

Their Conclusion:

- Superiority of performance (photograph vs. drawing) varies with the application
- Response times were consistently longest for the basic line drawing images

Pictorial Communication

Results of a study with 400+ adult subjects testing identification rate under different display conditions:

- 59% - black+white photo
- 67% - photo with background removed
- 72% - line drawing w/ detail + shading
- 62% - line drawing wo/ shading, little detail
- 61% - silhouette
- 49% - stylized depiction

”The so-called 'simple' stylized drawings are evidently not simple in anything but appearance, making greater demands on the person trying to interpret them".

D. Fussel and A. Haaland. "Communicating with Pictures in Nepal: results of practical study used in visual education", Educational Broadcasting International, 11(1): 25-31, 1978.

Study of Picture Preferences

- Surgeons rated the ‘schematic’ representation *least* preferable; the ‘semi-schematic’ and ‘realistic’ representations were preferred in equivalent numbers.

K. Hirsch and D. A. McConathy, "Picture Preferences of Thoracic Surgeons", Journal of BioCommunications, Winter 1986, pp. 26-30.

Why use Artistic Enhancement?

- To clarify the pictorial representation:
  - Emphasize important information
  - Minimize visual salience of secondary detail
  - Hierarchically guide the attentional focus
  - Convey the ambiguity of uncertain information

From Photograph to Drawing

- There are some easily definable rules
- There is a lot of room for artistic license
- Obtaining an ideal translation is exceptionally more complicated than would appear at first glance

Why use Non-Photorealistic Representation?

- To allow greater differentiation in the salience of the visual representation
  - Emphasize critical features
  - Minimize the visual salience of secondary details
  - Hierarchically guide the attentional focus

Other Advantages in Non-Photorealistic Representation

- Allows greater differentiation in the specificity of the visual representation
  - To convey information at an early stage of definition
    - to facilitate designer's working with ideas
    - to allow clients to envision multiple possibilities in a design
  - To convey uncertain information
    - to explicitly represent the level of confidence in the data

Other Advantages in Non-Photorealistic Representation

- Allows the expression of multiple styles
  - potentially increasing the 'dynamic range' of information that can be communicated
  - establishing a 'mood' that can influence the subjective context within which the information is perceived and interpreted

Introducing Ambiguity

- specific → universal (one person .. everyone)
- complex -> simple (progressive reduction of detail)
- realistic -> iconic (requiring more translation)

Scott McCloud. Understanding Comics: the invisible art. Harper Perennial, 1994.

Visual Representation in Architectural Design

- Different requirements for different phases of the process, from abstract to concrete
  - Schematic Design
  - Preliminary design
  - Design development
  - Contract documents
  - Shop drawings

Paul Laseau, Architectural Representation Handbook, McGraw-Hill, 2000.

From Art to Scientific Visualization

- Illustrations interpret physical reality; distill the essential components of the scene
- We seek algorithms that can make explicit some of the intuition that artists rely upon to create an effective visual representation

My Fundamental Philosophy

There is a science behind the art of effective data representation

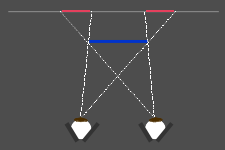
Creating images that facilitate the understanding of a set of data

- Design/Conceptualization
  - Defining an appropriate representational approach
- Implementation
  - Developing new algorithms for image generation
- Evaluation
  - Objectively determining which techniques are more effective when and why

Creating images that facilitate the understanding of a set of data

- Design
  - inspiration from practices in art, illustration
  - insight from research in visual perception
- Implementation
  - computer graphics, computer vision, mathematics
- Evaluation
  - quantitative assessment of impact on task performance

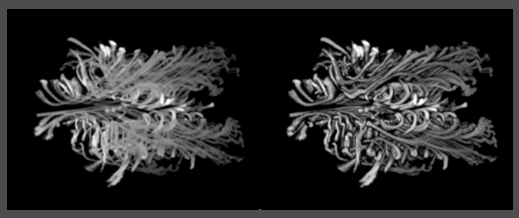
Clarifying Depth Discontinuities: insights from psychology and art



Ken Nakayama and Shinsuke Shimojo (1990) "Da Vinci Stereopsis: Depth and Subjective Contours from Unpaired Image Points", *Vision Research.*

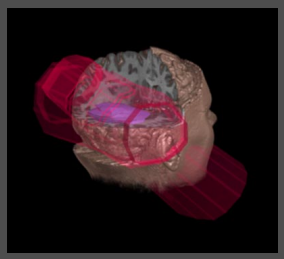
Gaps evoke the impression given by inter-ocularly unpaired regions

Clarifying Depth Discontinuities with Visibility-Impeding Halos



Victoria Interrante and Chester Grosch (1998). "Visualizing 3D Flow", *IEEE Computer Graphics and Applications*, **18**(4): 49-53.

Conveying the 3D Shape and Depth of Transparent Surfaces



Radiation Therapy Treatment Planning

Portraying Overlapping Surfaces

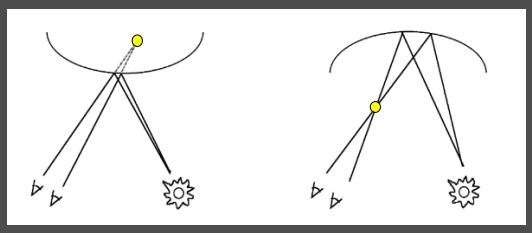
- Transparency offers the best potential solution
- But photorealistic representation isn't sufficient

Why are transparent surfaces difficult to adequately perceive?

- Weak occlusion cues
- Minimal shape-from-diffuse shading
- Unreliable depth information from specular highlights
- Refraction emphasizes silhouettes, but distorts underlying objects

Highlights: Shape but not Depth

In a stereo view, specular highlights will not lie on a curved surface but will appear to float either above or below it

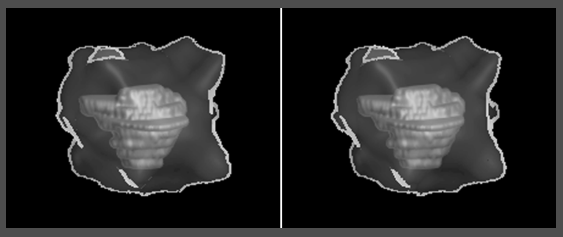


A. Blake and H. Bülthoff (1991) "Shape from Specularities", *Phil. Trans. Royal Soc. of London, B*, 331: 237-252.

Creating a Feature Line Drawing

- Silhouettes
  - separate figure from ground
- Contour lines
  - emphasize discontinuities
    - in depth (*viewpoint dependent*)
    - in curvature (*viewpoint independent*)
- Other Essential Lines
  - Express the underlying form
  - Delineate meaningful features
  - Can be difficult to capture algorithmically

Highlighting Silhouette Edges in Stereo is Problematic



Viewpoint- Independent Feature Lines: ridges and valleys

- Specular highlights tend to cling to ridges
- Valleys tend to remain in shadow
- We perceive objects as subdividing into parts along their valley lines

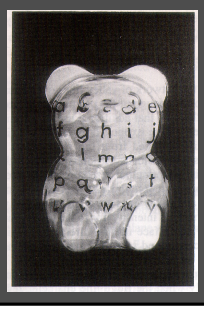
Donald D. Hoffman and Whitman A. Richards. "Parts of Recognition", *Cognition*, vol. 18, pp. 65-96, 1984.

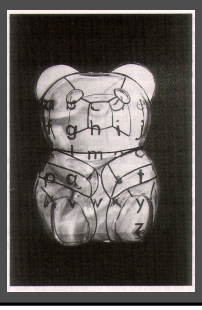
Salient Features of the Contour

- Our eye fills in missing segments along curves of minimum energy
- This process is more robust for contour deletions *between* vertices than for deletions across them

Irving Biederman (1985) "Human Image Understanding: Recent Research and a Theory", Human and Machine Vision, Azriel Rosenfeld, ed., Academic Press, pp. 13-57.

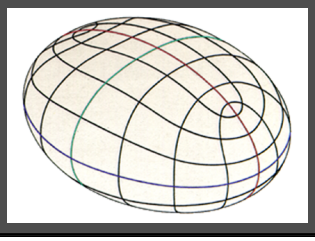
Can Marking the Ridges and Valleys Help?: a "kitchen experiment"





Ridge and Valley Lines: definition

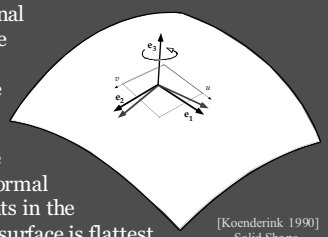
- The locus of points where the principal curvature assumes an extreme value along a line of curvature



Hilbert and Cohn-Vossen (1952)
Geometry and the Imagination

Computing the Principal Directions

- Begin with an orthogonal frame ($\vec{e}_1$, $\vec{e}_2$, $\vec{e}_3$), where $\vec{e}_3 = \vec{n}$, and $\vec{e}_1$ and $\vec{e}_2$ span the tangent plane
- Rotate the frame so that $\vec{e}_1'$ points in the direction of greatest normal curvature, and $\vec{e}_2'$ points in the direction in which the surface is flattest.



[Koenderink 1990]
Solid Shape

Computing the Principal Directions

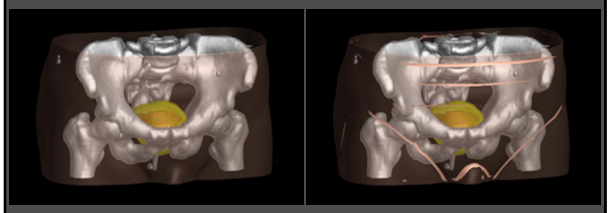
- The frame rotation can be obtained by diagonalizing the second fundamental form matrix:

$$A = \begin{bmatrix} \tilde{\omega}_1^{13} & \tilde{\omega}_1^{23} \\ \tilde{\omega}_2^{13} & \tilde{\omega}_2^{23} \end{bmatrix}$$

  which describes how the surface normal changes as you move very slightly away from the origin of the frame across the surface in the $\vec{e}_1$ and $\vec{e}_2$ directions;
- When $\vec{e}_1$ and $\vec{e}_2$ coincide with the principal directions, there is no sideways "twist" in the surface normal direction as you move in these directions; terms $\tilde{\omega}_2^{13}$ and $\tilde{\omega}_1^{23}$ are zero.

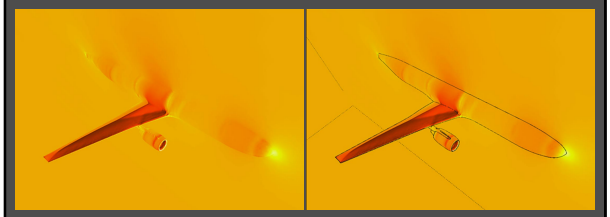
[Koenderink 1990]

Using Ridge and Valley Lines to Emphasize Intrinsic Shape Features



Victoria Interrante, Henry Fuchs and Stephen Pizer (1995)
"Enhancing Transparent Skin Surfaces with Ridge and Valley Lines", *IEEE Visualization'95.*

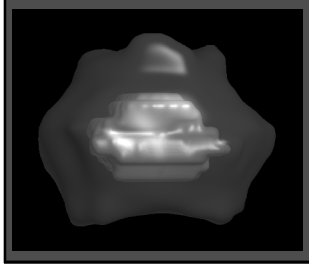
Using Locally Important Edges to Capture the Structure of Faceted Objects

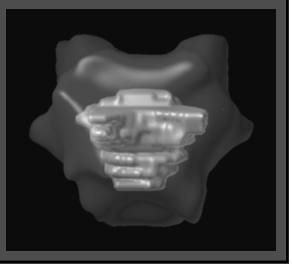


Kwan-Liu Ma and Victoria Interrante (1997)
"Extracting Feature Lines from 3D Unstructured Grids", *IEEE Visualization'97.*

Conveying the 3D Shape of Arbitrary Smoothly Curving Surfaces

Adding texture could help, but what texture will show shape best?





Transparency Representation in an Opaque Medium

Giuseppe Croff, *Veiled Nun*, 1869.

Artistic Inspiration

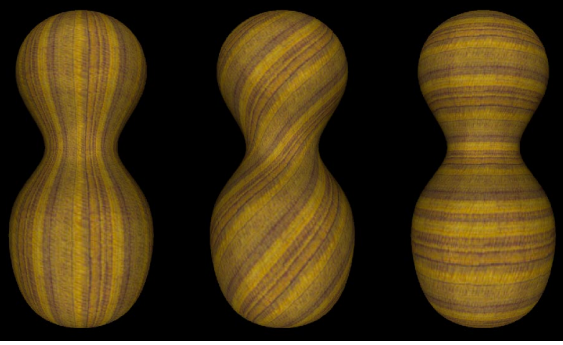
- Russell Drake's "single line system of shading"
  - the flow of the shape is conveyed through the directions of the carefully drawn strokes
  - multiple overlapping surfaces are displayed with clarity
- But not all artists use line in this way.

Lumbosacral and Sacro-iliac fusion . Russell Drake, medical illustrator, Mayo Foundation, 1932.

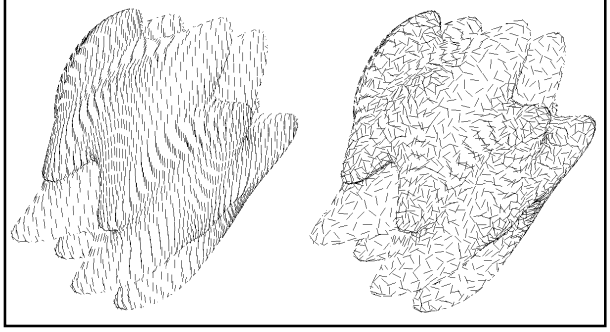
Paul Richer, Artistic Anatomy. Translated and edited by Robert Beverly Hale, Watson-Guptill Publications, 1971.

Does texture orientation matter?

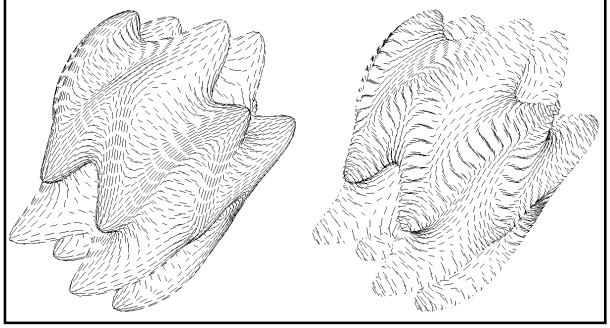
Texture type, and how it is applied, is significant

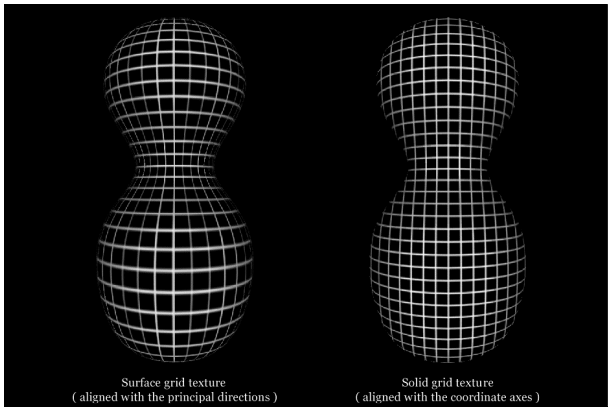


Line Direction Matters



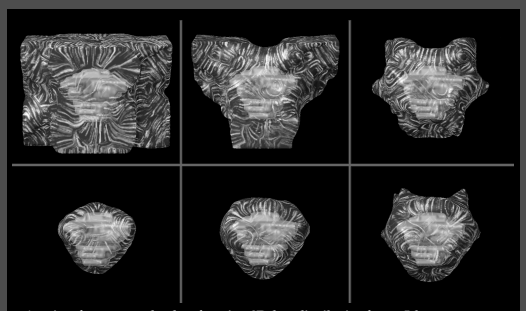
Line Direction Matters





A Principal Direction Texture

- Long, curved strokes following the *flow* of the principal directions over a surface
- A single, solid texture, defined once for all isosurfaces in a smooth 3D distribution



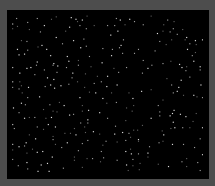
A series of transparent level surfaces in a 3D dose distribution from a 5-beam treatment plan for prostate cancer, each shown in relation to the physician-defined target region.

Line Integral Convolution (a review)

- Introduced by Cabral and Leedom [93]
  - A new method for conveying directional information through patterns of correlation in a texture
- Extended by Stalling and Hege [95]
  - define long, accurate streamlines using an adaptive 4th order Runge-Kutta integration method
  - resample the streamlines via a C^1 continuous cubic spline interpolation, to obtain equally spaced points
  - compute weighted sums of the input texture values over a fixed number of points along the streamlines

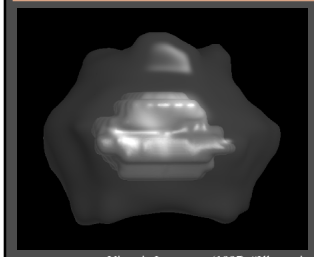
Defining the LIC Stroke Texture

Begin with a sparse set of evenly-distributed points



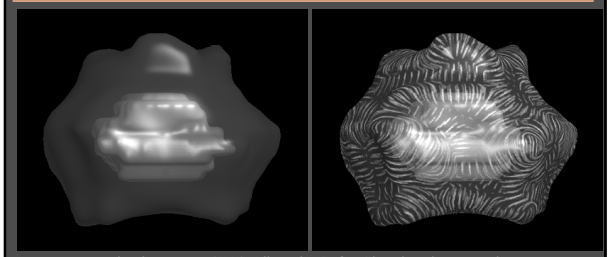
Advect the empty space along with the full to help finesse the problem of aesthetic streamline placement

Conveying the 3D Shape of Arbitrary Smoothly Curving Iso-Surfaces



Victoria Interrante (1997) "Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution", *SIGGRAPH 97*.

Conveying the 3D Shape of Arbitrary Smoothly Curving Iso-Surfaces



Victoria Interrante (1997) "Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution", *SIGGRAPH 97*.

Shape from Texture

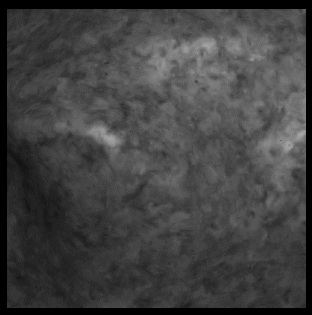
- Showing shape with a line-like texture seems to contradict the shape-from-texture research which asserts that shape perception is impeded by texture pattern anisotropy

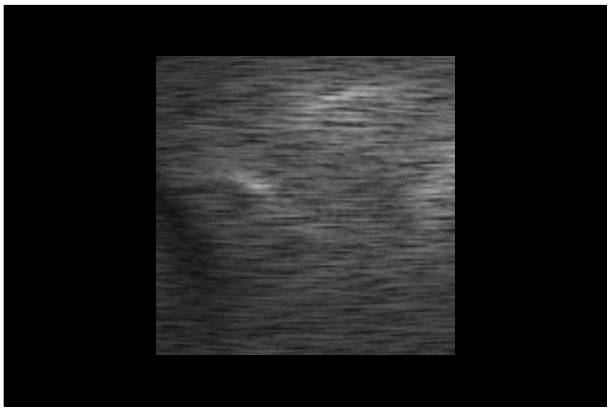
How Does Texture Orientation Affect Surface Shape Perception?

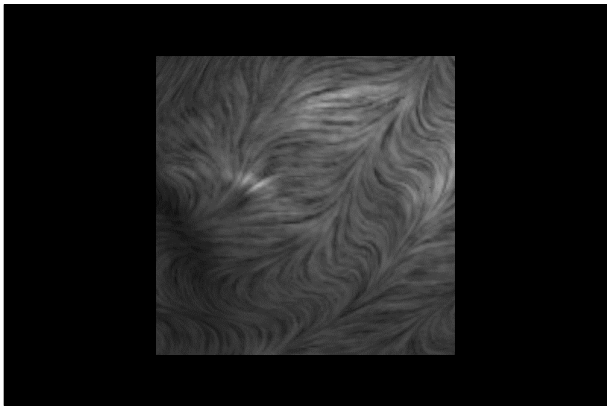
- Can anisotropy in the principal directions help?
- Does anisotropy in non-principal directions hurt?
- Do these effects hold for shaded displacement texture?
- To what extent are these effects mitigated by stereo viewing?

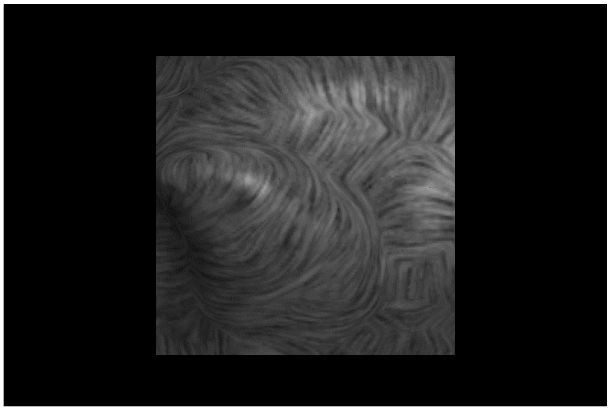
Compared Four Direction Types

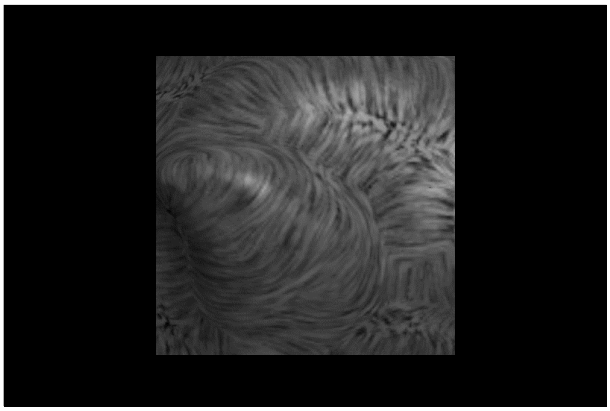
- Principal direction (pdir)
- Uniform direction (udir) = (-ny, nx, 0)
  - zero geodesic curvature
- Random direction (rdir) : rotate udir about $\vec{n}$ by a random angle $\theta \in [-\pi/2 .. \pi/2]$
  - effectively isotropic
- Sinusoidally varying direction (sdir): rotate udir in the tangent plane by a coherently varying angle $\theta = 10\pi(x+y+z/n)$

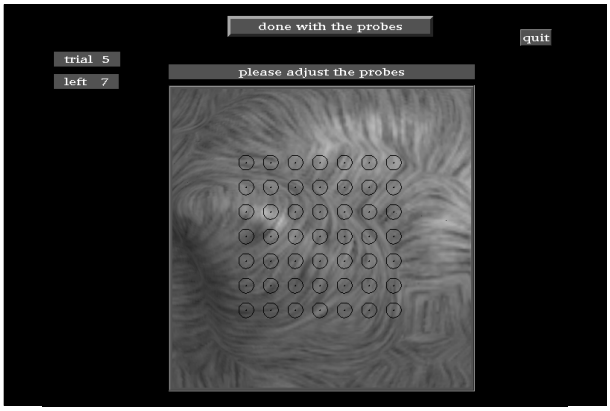






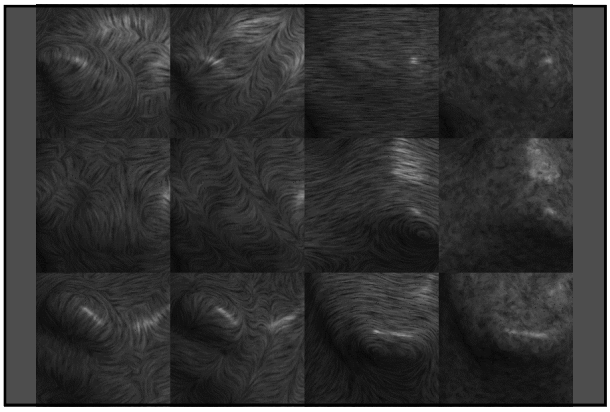


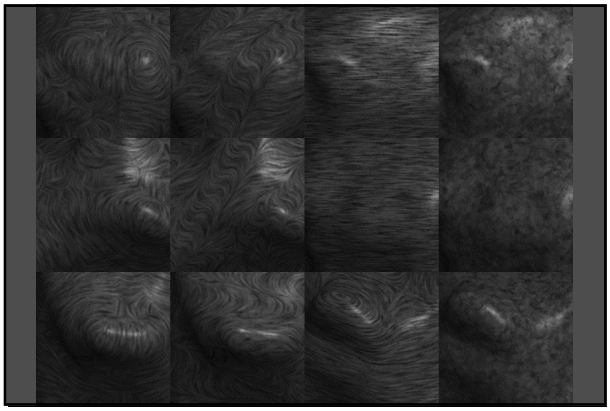




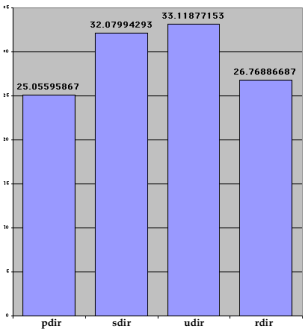
Experiment Details

- 4 different texture patterns: pdir, sdir, udir, rdir
- 6 different surface stimuli
- 49 probes per image, same points for each texture
  - users were asked to reconstruct the surface
- 2 different viewing conditions: flat, stereo
- 5 subjects (naïve to purpose of experiment)
  - Split into two groups; each saw half of the data
  - Four sessions, 6 surfaces each, randomized presentation order, 2 sessions of flat images followed by 2 sessions with stereo images

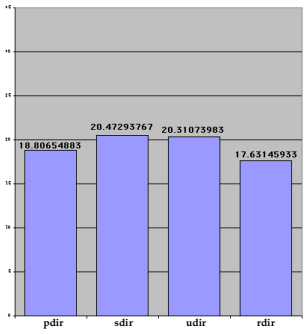




Mean alignment error (3D angle), flat viewing condition



Mean alignment error (3D angle), stereo viewing condition



Experiment's Conclusions

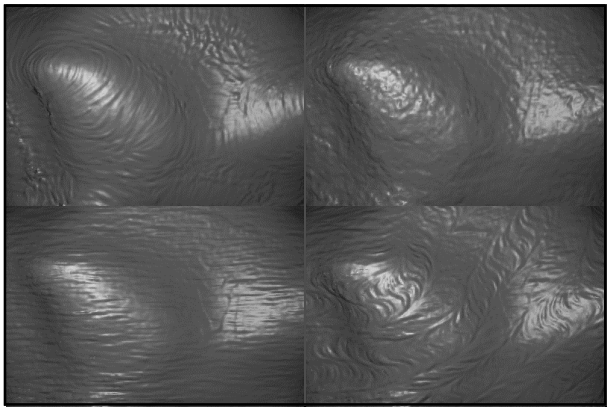
- Texture pattern orientation has a statistically significant effect on surface shape perception
- Shape perception is poorer in the presence of anisotropic textures that have nonzero geodesic curvature
- Shape perception seems equivalently good from the anisotropic texture that is aligned with the first principal direction as it is from the isotropic texture

Displacement Texture

- What is the impact of texture orientation if the texture is defined as a pattern of relief rather than as a pattern of intensity variations?
- Shape-from-texture perception could work differently with relief patterns than with patterns of coloration

Displacement Texture

- What is the impact of texture orientation if the texture is defined as a pattern of relief rather than as a pattern of intensity variations?
- Does shape-from-texture work differently with relief patterns than for patterns of coloration?
- Preliminary results suggest that performance follows a similar pattern



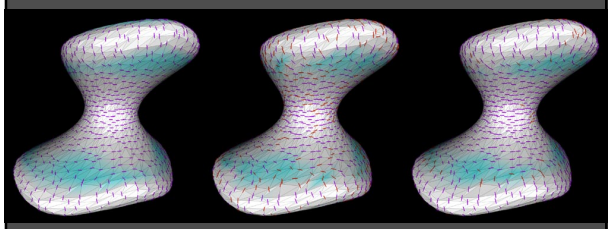
Conclusions

- Texture pattern orientation affects surface shape perception
- Shape perception is poorer in the presence of anisotropic textures that are unaligned with the principal directions
- Shape perception is as good with an anisotropic texture that is aligned with the first principal direction as it is with an isotropic texture

More Challenges

- How to compute accurate estimates of principal directions at the vertices in a polygonally-defined model?
- How to obtain a smooth vector field of principal directions that permits tracing long, smooth strokes that flow gracefully across umbilics and between regions of opposing directional dominance?
- How to apply oriented, image-based textures so that they follow the principal directions?

Estimating Principal Directions on Arbitrary Meshes



Exact Solution | Normal Curvature Method | Adjacent-Normal Cubic Method

J. Goldfeather, V. Interrante (2001) "Understanding Errors in Approximating Principal Direction Vectors".

Error Analysis

- All solutions follow these steps:
  - Find a least squares fit to $Ux = b'$ at p
    - For example: $\kappa'_{yi} = 2[(p-q_i)\bullet N'_p]/[(p-q_i)\bullet(p-q_i)]$

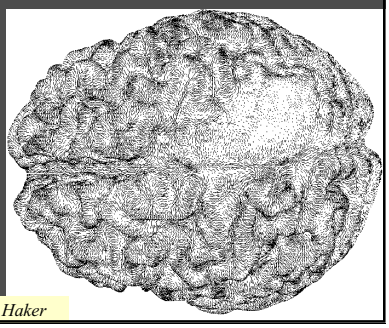
    produces a system of equations $y_i^T W y_i = \kappa'_{yi}$
    where y_i is the unit projection of pq_i on the tangent plane
  - Use the least squares solution x' to define the 2x2 Weingarten matrix W' at p
  - The eigenvectors of W' indicate the principal directions

Sources of Error

- Small errors in estimates of normal curvatures can produce huge errors in the estimates of the principal directions
- The problem is a combination of both the magnitude of the errors in *b*' and the direction in which they are made, which is influenced by the local geometry of the mesh
- The adjacent-normal cubic method does a better job of controlling the magnitude of the errors because it uses third-order approximations rather than second-order

Possible NPR Applications

- Use principal direction strokes to create interactively manipulable computer generated pen-and-ink style representations of arbitrary, polygonally-defined objects



Brain image by Steve Haker

Principal Direction Texture Mapping

- Synthesize an arbitrary, image-based texture pattern over an arbitrary polygonal surface so that the dominant orientation of the texture follows the first or second principal direction over the surface

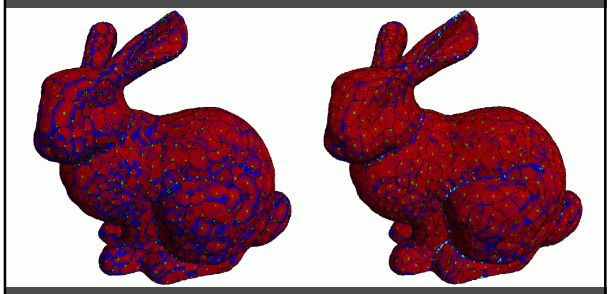
Fitted Textures

- Define a mapping
  - Partition the surface into nearly equally-sized, nearly planar patches
- Synthesize the texture
  - Generate the required texture from a given sample, maintaining continuity across boundaries

[Efros and Leung 99]

Gabriele Gorla, Victoria Interrante and Guillermo Sapiro (2001), "Growing Fitted Textures over Surfaces".

Surface Partitioning



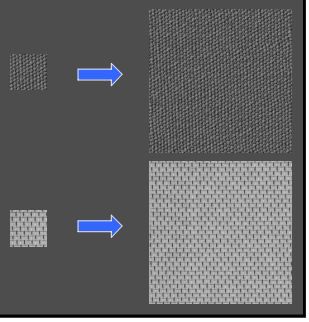
Before and after optimization; blue indicates amount of nonplanarity

Surface Partitioning

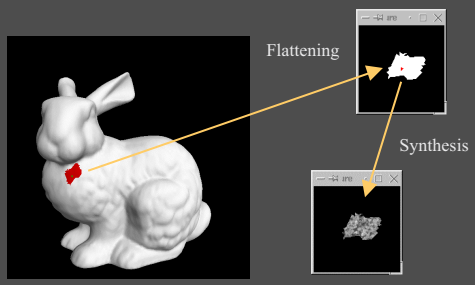
- Pick a random triangle T and assign it to a new patch P
- Add all triangles C connected to T that satisfy:
  - normal(C)•normal(T) > N_threshold
  - distance(C,T) < D_threshold
- Repeat until surface is completely covered
- Optimize with "triangle stealing"

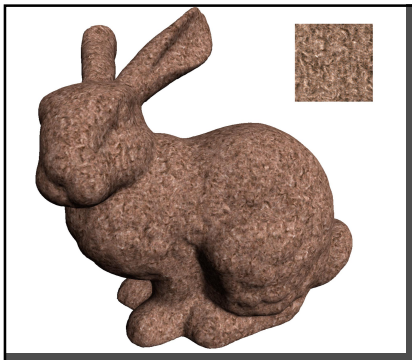
Texture Synthesis

- Two-pass version of Efros and Leung's Markov Random Field texture synthesis method
  - Exhaustive small neighbourhood matching
    - Saves the best matches for further processing
  - Selective processing at the most promising locations using the entire neighbourhood



Putting it all together

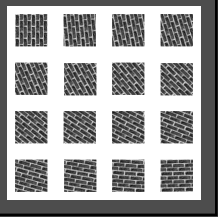




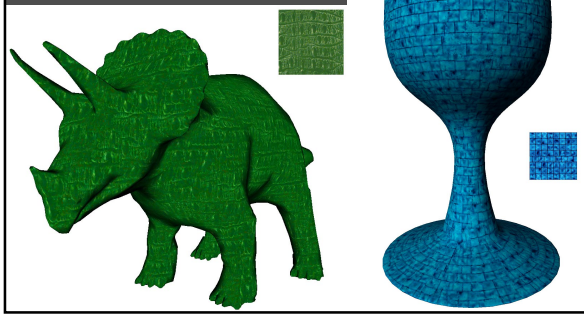
... Unoriented Texture Doesn't Work Well

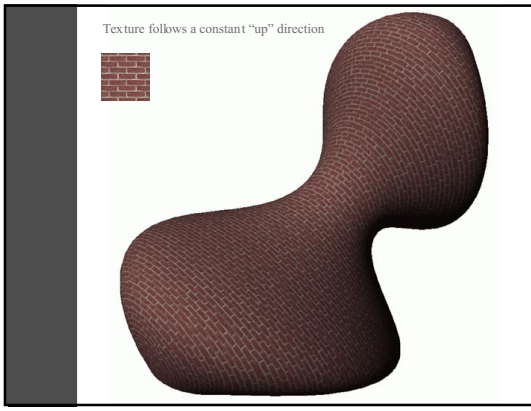
Orienting the Texture

- User should not have to select the directions manually
- The texture direction should follow the surface curvature in a natural way
- The final result should be pleasant
- Change the target of the search a per-pixel basis to follow the specified direction
- Textures are pre-rotated improve performance

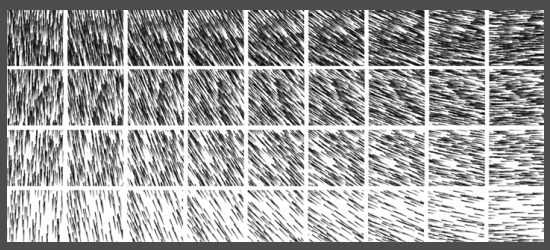


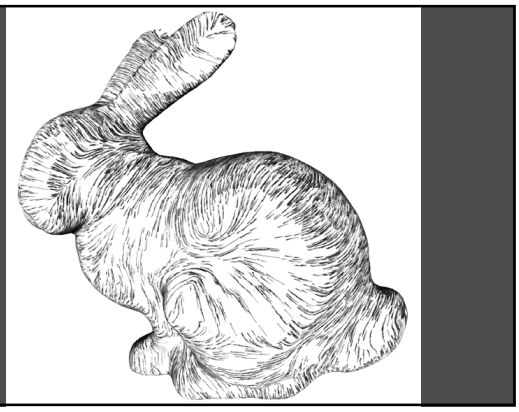
Textures following a constant "up" direction

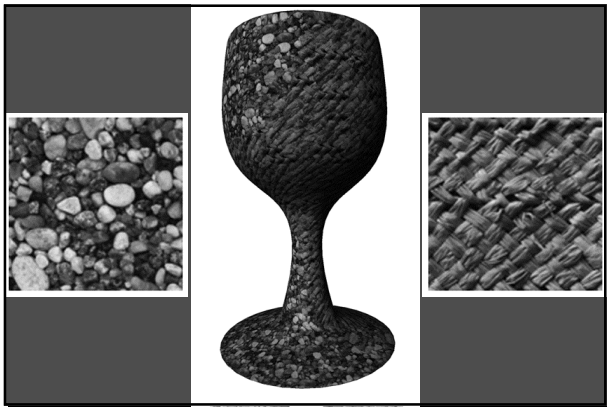




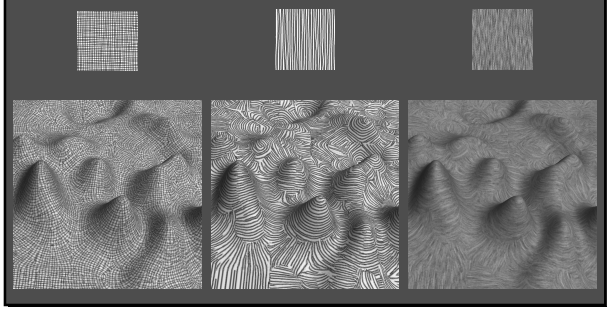
Indexing into an Array of Textures







Texture Design for Shape Representation - Experimental Investigations



Acknowledgments

- Students: Ahna Girshick, Gabriele Gorla, Sunghee Kim, Jeremy Leboy, Todd LeMoine, Jason Gott, Seonho Kim, Tim Urness, Haleh Hagh-Shenas, Cheong-Hee Park
- Collaborators: Jack Goldfeather, Guillermo Sapiro, Steve Haker, Tomas Filsinger
- Funding agencies: University of Minnesota (Grant-in-Aid of Research, Artistry and Scholarship), and NSF (Large Scientific Data Visualization, and Presidential Early Career Award for Scientists and Engineers)

Stroke-Based Rendering

Aaron Hertzmann
University of Washington

1 Introduction

This chapter describes **stroke-based rendering** (SBR), an automatic approach to creating non-photorealistic imagery by placing discrete elements called **strokes**, such as paint strokes or stipples. Many stroke-based rendering algorithms and styles have been proposed, including styles of painting, pen-and-ink drawing, tile mosaics, stippling, streamline visualization, tensor field visualization and jigsaw image mosaics. This tutorial attemps to make sense of the disparate work in this area by creating a unified view of SBR algorithms, which helps us to identify the common elements, as well as the unique ideas of each. Moreover, presenting ideas in this fashion suggests possibilities for future research.

We can introduce SBR algorithms with a painterly rendering [Her98, Her02]:

Source photo

Painted version

Final rendering

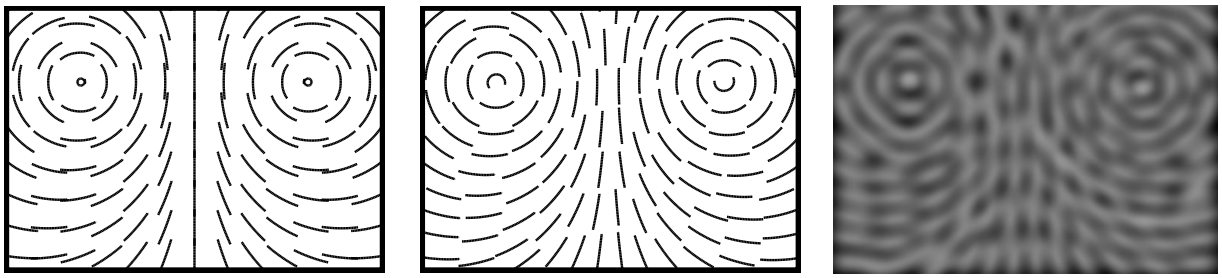
This figure shows an SBR algorithm in action: starting from a photograph, a collection of brush strokes are placed to match the photograph, and then rendered to appear as if created with oil paint.

Although the details vary, all SBR algorithms create images by placing strokes according to some **goals**. The most common goal is that we want the painting to "look like" some other image — in this case, we want to place colored brush strokes to look like the picture of the mountain. Another important goal is to limit the number of strokes in some way. Otherwise, the algorithm can just use many tiny brushstrokes, producing a very good match to the source image without much abstraction.

Finally, once the strokes have been placed, they can be rendered in some other form. Note that we did not add texture until after the brush strokes are placed; we compared the source photo to

some intermediate image with a simplified stroke model. This is both for efficiency and for aesthetic reasons, to be discussed later. The main point is that the final rendering may be different from the way we expressed our goals about the image.

Here is another example of a automatic vector-field visualization [TB96]: Here, streamlines are



Vector field Final rendering Blurred rendering

used to effectively convey the motion of a vector field. In order to clearly illustrate the vector field, the placements should be placed evenly — the middle rendering was created with the goal that the blurry version should be as close to a constant grey value as possible. For comparison, the image on the left shows stroke placements on a regular grid without adjustment. Again, we can see that this streamline visualization algorithm is an SBR algorithm: it places strokes (streamlines) according to specified goals (to follow the vector field and to match a target tone in the blurred image).

It is usually not possible to exactly meet all of the goals; hence, it is useful to have a way of trading-off the goals, and quantifying their importance. We can do this by formalizing an SBR problem as an **objective function** minimization problem. An objective function is a mathematical formula that explains "how good" our rendering is; SBR algorithms can be seen as attempting to minimize objective functions. For example, it isn't possible to place the streamlines in the above visualization to achieve a purely constant tone in the gray image. Hence, instead, we can use as an objective function the deviation of the blurred image from a constant image.

So far, we have described two different SBR problem statements, one for painterly rendering and one for visualization, but said nothing of how to design algorithms for these problems. There are two main approaches to designing SBR algorithms: **greedy algorithms**, in which strokes are greedily placed to match the target goals, and **optimization algorithms**, where the algorithm iteratively places and then adjusts stroke positions to minimize the objective function. A greedy algorithm produced the above painterly rendering, and an optimization algorithm produced the streamline visualization.

Haeberli introduced both a semi-automatic greedy algorithm and an automatical optimization algorithm in a seminal paper [Hae90]. Digital paint systems had previously automated some of the stroke renderings [Smi01], but did not automate any stroke placement choices. Wireframe renderings had previously been common in computer graphics (preceding photorealistic rendering), and Yessios [Yes79] described a system for drafting with strokes based on architectural drafting styles.

Although this tutorial focuses on the technical details of SBR algorithms, it is important to remember that they are useless without human control. Every aspect of the system (including the choice of stroke models, the setting of weight parameters, and the selection of input imagery) requires aesthetic

decisions which can only be made by an artist working towards some goal. Ideally, a human artist using the system should have total control over the decisions being made. For example, a user should be able to specify spatially-varying styles, so that different rendering styles are used in different parts of the image, or to specify positions of individual strokes. However, one of the great advances in art in the age of digital machines is the ability to create complex systems of procedural art, where the artist does not directly create the final work, but rather creates rules according to which the final decisions are made[1]. Hence, an artist may design the energy function, but not necessary edit every individual image produced by the algorithm. In one possible scenario, the artwork may "occur" at a time after the artist's involvement. The main goal of SBR algorithms is to provide procedural tools that automate parts of the image creation process, not to replace the artist (which would be both a futile and an undesirable goal).

In this tutorial, I survey some of the various SBR styles and algorithms that have been created, and discuss the advantages and disadvantages of each. I will first describe the framework in somewhat more rigorous detail, including the use of objective functions to define specific problems. I will then describe specific SBR applications, grouped by algorithm in order to emphasize how, from a computational point of view, styles that look superficially different are often just variations on a theme. Pointers to related research (including extensions to animation) are given in Section 7.

2 Stroke-Based Rendering: Stating the Problem

In this section, I outline a general view of stroke-based rendering algorithms in terms of energy minimization. I begin with some preliminary definitions and examples, followed by the basic problem statement as well as a statistical view of the problem, and conclude with a discussion of the advantages and disadvantages of this approach. SBR algorithms will be surveyed in the remaining sections of this chapter.

We begin with a few definitions. First, we need to define what our strokes can look like.

Definition: A **stroke** is a data structure that can be rendered in the image plane. A **stroke model** is a parametric description of strokes, so that different parameter settings produce different stroke positions and appearances.

For example, one form of stippling uses a very simple stroke model:

Stippling stroke model Individual strokes (stipples)

As illustrated above, a stipple is a stroke that can be described with two parameters: the (x,y) position of the stipple in an image, and the radius R of the stipple. (As we shall see in Section 3.1,

[1]Technically speaking, procedural art does not require modern technology (e.g. see [Aar97, Mur98]).

other definitions of a stipple are possible.)

We create images by combining strokes:

Definition: An **image structure** is a data structure containing

- A canvas, defined by a background color or texture,
- An ordered list of strokes, definined by their parameter settings.

An image structure contains the necessary data to create a rendering. To create an image, the list of strokes is rendered by alpha-compositing over the background. The background is usually just a solid color or a predefined texture image. Examples of images composed of paint strokes and streamlines were shown in Section 1. For example, a PostScript file containing line art can be thought of as an image structure, since it contains a list of stroke definitions; the data in the file can be rendered on the screen or on a printer.

Finally, we need a quantitative way of evaluating how good a rendering is:

Definition: An **SBR energy function** is a function $E : I \rightarrow R$, where I is the set of possible image structures, and R is the set of real numbers.

Intuitively, we can think of the energy function as scoring "how good" the painting or drawing is. An energy function $E(I)$ takes an image structure as input and outputs a number indicating the "quality" of the image — generally, the goal of an SBR algorithm is to produce an image with the smallest possible energy. The energy function is sometimes also called an "objective function," "cost function," or "error function." The term "energy" comes from analogous uses in physics, such as searching for the minimum energy configuration of a set of particles.

SBR algorithms are normally defined in terms of some input data, usually an input image. In most algorithms described in this survey, the energy function measures how closely the rendering matches some input image. Additionally, the energy function encodes tradeoffs. For example, a painterly rendering algorithm takes an input image and produces an image structure containing color paint strokes that matches the source image. However, one could get a perfect match to the source image by placing thousands of tiny brush strokes, which would not look too different from the source image. We can create a more interesting painting by adding an "abstraction" term to the energy function, by assigning higher energies to paintings that use less strokes. For example, the energy function could be

$$\begin{aligned} E(I) &= E_{match}(I) + w_{abs} E_{abs}(I) \\ E_{match}(I) &= \sum_{(x,y)\in I} \|I(x,y) - S(x,y)\|^2 \\ E_{abs}(I) &= \text{the number of strokes in } I \end{aligned}$$

where w_{abs} is a scalar weight parameter, and $E_{match}(I)$ is the sum-of-squared differences between the source image and the rendering. This energy function has one parameter w_{abs}. We can control the level of abstraction in the painting style by adjusting the value of this parameter: setting w_{abs} to be small specifies that we want a very realistic style (reproducing the original image as close as possible);

setting w_{abs} to be large specifies a very abstract style (capturing the image with very few strokes). (Of course, this is only one possible notion of abstraction.) This suggests the following definition of style:

Definition: An **SBR style** is a stroke model and energy function (including parameter settings) over image structures.

In other words, an algorithm creates an image in a specific style by minimizing the corresponding energy function. Note that this is a broadly-inclusive notion of style — in this view, changing the parameters to a gallery effect in an imaging tool constitutes changing the style (although not by very much). The goal of this framework is to provide a structure within which many styles can be created and applied.

3 Optimization algorithms

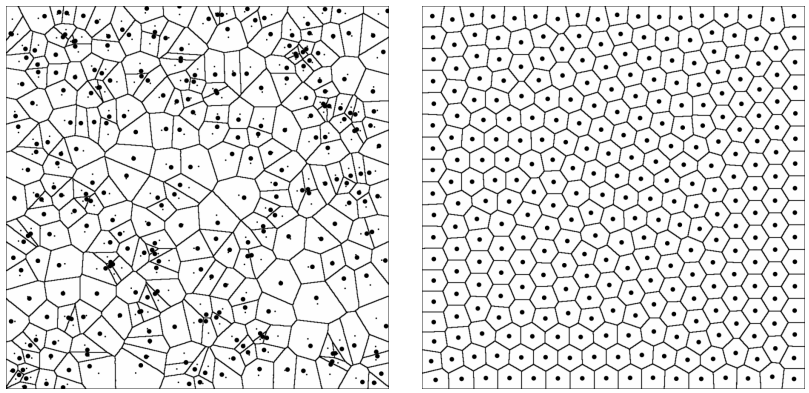
In this section, I describe optimization algorithms for SBR problems. Two kinds of optimization algorithms have been applied to SBR. The first kind, which I will call **Voronoi algorithms**, exploits special properties of the SBR problem to perform efficient global update steps. The second kind, which I will call **trial-and-error algorithms**, assumes no special structure and perform heuristically-chosen tests to try to reduce the energy. In general, the Voronoi algorithms are very effective and fast, but cannot be applied to all problems. The trial-and-error algorithms are very general-purpose, but at the cost of substantial computation times. (It so happens that both of the approaches have also been called "relaxation algorithms.")

3.1 Voronoi algorithms

Voronoi algorithms are useful for SBR problems where the final image will be composed of many identical non-overlapping strokes, and where only the density of the strokes is constrained. The central idea is to use efficient techniques from computational geometry to place evenly-spaced strokes into an image. Moreoever, these techniques can be made very fast using graphics hardware. However, these algorithms do not directly optimize with respect to an image-based metric (e.g. thatthe rendering should match target tones) since the energy function is defined in terms of stroke densities.

3.1.1 LLOYD'S METHOD

How can we create a set of evenly-spaced points within an image? We will use an iterative optimization procedure, which requires defining an appropriate energy function. Let $\mathbf{p} = (x,y)$ be a pixel locations in an image, and let $\mathbf{C}_i$ be special point locations called "centroids;" the strokes will eventually be placed at these locations. Let $L^i_{\mathbf{p}} \in \{0,1\}$ be a binary labeling of pixels: if $L^i_{\mathbf{p}} = 1$, then the pixel $\mathbf{p}$ has been "assigned" to centroid i. Every pixel is assigned to exactly one centroid: $\sum_{\mathbf{p}} L^i_{\mathbf{p}} = 1$. The goal is



(a) (b)

Figure 1: (a) Voronoi diagram of a set of points $\mathbf{C}_i$. The image is partitioned into a set of regions, one region for each point. Each region is designed so that it contains all pixels that are closest to the corresponding point $\mathbf{C}_i$. (b) By applying Lloyd's method, the points are adjusted so that they lie at the centroid of their region of the Voronoi diagram. The resulting point set is evenly spaced and can be used to specify regular stroke placements. Images from [Sec02], used by permission.

to choose both a set of centroids and a labeling that minimizes the energy function[2].

$$E(I) = \sum_{\mathbf{p}\in I} L^i_{\mathbf{p}} \|\mathbf{p}-\mathbf{C}_i\|^2 \quad (1)$$

$$= \sum_{\mathbf{p}\in I} L^i_{\mathbf{p}} ((\mathbf{p}_x-\mathbf{C}_x)^2+(\mathbf{p}_x-\mathbf{C}_y)^2) \quad (2)$$

where the centroids and labeling are implicitly members of I. In short, we want every pixel to be close to its assigned centroid. If we knew the set of centroids $\mathbf{C}_i$ in advance, then computing the optimal labeling would be easy — we just assign every pixel to the nearest centroid. The resulting labeling is known as a Voronoi diagram (Figure 1(a)) — it partitions the plane according to which centroids $\mathbf{C}_i$ are nearest to each point.

From looking at Figure 1(a), it should be clear that picking some randomly-chosen point set and then computing the Voronoi diagram does not give a very good arrangement of centroids. In fact, we can improve upon this set of centroids simply by adjusting the point centers to best fit this partition — in other words, by holding fixed the labeling and optimizing the energy function with respect to the centroids. The new optimal centroids are given by $\mathbf{C}_i = \frac{\sum_{\mathbf{p}} L^i_{\mathbf{p}}\mathbf{p}}{\sum_{\mathbf{p}} L^i_{\mathbf{p}}}$. This is just the mean of the pixel locations that are assigned to $\mathbf{C}_i$; this formula is easily obtained by setting $\frac{\partial E(I)}{\partial \mathbf{C}_i} = 0$ and solving for $\mathbf{C}_i$. We can iterate these two steps. This, in fact, is known as Lloyd's method:

[2]NPR researchers have typically presented the continuous version of this energy, e.g. $\int \|\mathbf{p}-\mathbf{C}_i\|^2 d\mathbf{p}$ where $\mathbf{C}_i$ is the nearest centroid to $\mathbf{p}$, where $\mathbf{p} \in R^2$ is a real-valued point in the plane rather than a discrete pixel location. I prefer the discrete version, because it more closely reflects the problem actually being solved. One benefit is that we can prove convergence of the discrete version of the algorithm, whereas convergence has not been proven for the continuous version.

function LLOYDSMETHOD(n, I):
  initialize the centroids $\mathbf{C}_i$ by randomly sampling n points uniformly in the image I
  while the algorithm has not converged
    reestimate the labeling by $L^i_{\mathbf{p}} \leftarrow \begin{cases} 1 & i = \arg\min_i \|\mathbf{p} - \mathbf{C}_i\|^2 \\ 0 & \text{otherwise} \end{cases}$
    reestimate the centroids by $\mathbf{C}_i \leftarrow \frac{\sum_{\mathbf{p}} L^i_{\mathbf{p}} \mathbf{p}}{\sum_{\mathbf{p}} L^i_{\mathbf{p}}}$
  return the centroids $\mathbf{C}_i$

Figure 1(b) shows the same point after applying Lloyd's method. The algorithm is said to converge when the energy does not change between steps. The algorithm is guaranteed to reduce the energy at every step before convergence, because each step minimizes the energy with respect to some parameters. It is guaranteed to converge, because the energy decreases at every step before convergence, and because there is a finite number of possible labelings L. Lloyd's method was discovered separately by the signal-processing community (where it is known as "vector quantization" [GG92]) and the machine learning community (where it is known as "k-means clustering" [Bis95]).

Running this optimization in software over an entire image can be quite slow. However, it has recently been shown that graphics hardware can be used to make the process very fast [WND97, HCK+99]. The basic idea is to accelerate the labeling step by using Z-buffer hardware — by rendering a cone at each centroid location, one can show that computing the nearest centroid is equivalent to determining which cone is visible in each pixel. The centroids are updated by a single pass over the image. See [HCK+99] for details.

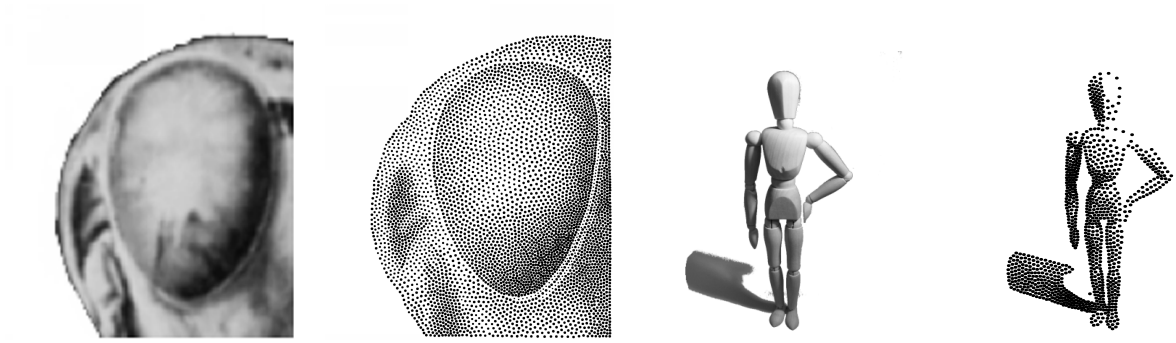
3.1.2 VARIATIONS ON LLOYD'S METHOD FOR SBR

Now that we have a procedure for regular placement of points, it is straightforward to design SBR algorithms on top of it. Perhaps the simplest SBR problem to describe is stippling: placement of many small dots to match some target grayscale image. Deussen et al. [DHvOS00] presented the first such method, using stipples to approximate gray tones in a target image (Figure 2). In their method, the image is manually segmented into distinct regions, and stipples are placed evenly within each region, by applying Lloyd's method to each region separately. The centroids are initialized using a half-toning algorithm. Once the centroid locations are chosen, they are replaced with stipples. The sizes of a stipple is proportional to the gray level of the image underneath it.

Secord [Sec02] presents an alternate stippling style and algorihtm, by varying the dot spacing instead of the dot size (Figure 2). The idea is to define a spatially-varying density function $\rho(\mathbf{p})$ that determines how dense the stippling should be in different parts of the image. This density function is directly derived from the tones of the target image $T(\mathbf{p})$, i.e. $\rho(\mathbf{p}) = 1 - T(\mathbf{p})/m$ where m is the max gray level in T. The new energy function is

$$E(I) = \sum_{\mathbf{p} \in I} L^i_{\mathbf{p}} \rho(\mathbf{p}) \|\mathbf{p} - \mathbf{C}_i\|^2 \quad (3)$$

$$= \sum_{\mathbf{p} \in I} L^i_{\mathbf{p}} \rho(\mathbf{p}) ((\mathbf{p}_x - \mathbf{C}_{ix})^2 + (\mathbf{p}_x - \mathbf{C}_{iy})^2) \quad (4)$$



(a) (b) (c) (d)

Figure 2: Stippling algorithms. In stippling, many small points are placed to match a gray tone image. Results from [DHvOS00] are shown in (b), and from [Sec02] in (d). In order to match the target gray tones, Deussen et al.'s algorithm varies stipple size (keeping stipple density constant), whereas Secord's algorithm varies stipple density (keeping stipple size constant). The eye in the grasshopper image was manually segmented from the rest of the image. Images used by permission.

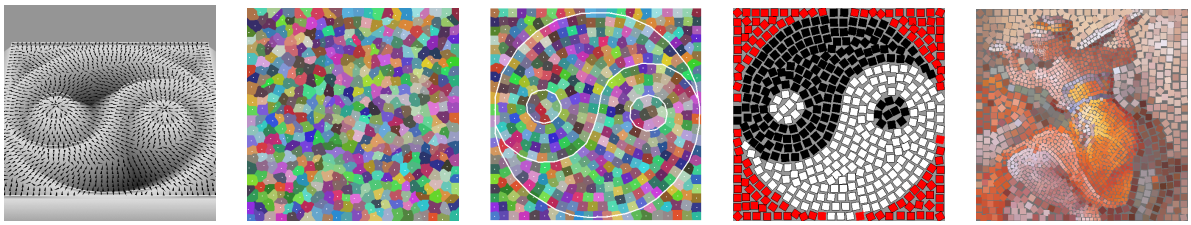
Following the same steps as before directly leads to a slightly different version of Lloyd's method: the labeling step is the same, but the centroids are now reestimated as $\mathbf{C}_i \leftarrow \frac{\sum_{\mathbf{p}} L^i_{\mathbf{p}} \rho(\mathbf{p})\mathbf{p}}{\sum_{\mathbf{p}} L^i_{\mathbf{p}} \rho(\mathbf{p})}$. This summation can be accelerated by precomputing sums of $\rho(\mathbf{p})$. This method gives somewhat sharper image boundaries, since the stipple placement is directly affected by the intensity of the source image. Secord also uses a simpler initialization procedure based on rejection sampling [Mac98]. Specifically, the algorithm samples point locations from a uniform distribution, includes the sampled points in the initialization with probability proportional to $\rho(\mathbf{p})$.

Lloyd's method can also be used to create tile mosaics from color source images. A simple approach is to create a Voronoi diagram of an image, and then color each region of the image by the color from the underlying source image [HCK+99]. However, this produces a mosaic with very irregular tile shapes.

Hausner [Hau01] describes two enhancements to this method (Figure 3). First, square tile shapes can be generated by replacing the L_2 norm with the L_1 norm ($\|\mathbf{v}\|_1 = |\mathbf{v}_x| + |\mathbf{v}_y|$). Second, an orientation field $\phi(\mathbf{p})$ can be specified for the image to create tilings with consistent orientations (Figure 3(a)). The orientation of each tile is constrained to match the vector field: $\phi_i = \phi(\mathbf{C}_i)$ The new energy function is now

$$E(I) \quad = \quad \sum_{\mathbf{p} \in I} L^i_{\mathbf{p}} \|R_{\phi(\mathbf{C}_i)}(\mathbf{p} - \mathbf{C}_i)\|_1^2 \tag{5}$$

where $R_{\phi(\mathbf{C}_i)}$ is a rotation matrix with orientation $\phi(\mathbf{C}_i)$. We can create a new optimization procedure as follows:



(a) (b) (c) (d) (e)

Figure 3: Tile mosaic results from [Hau01]. (a) Perspective view of the vector field used for the yin-yang example. The vector field was generated from the height field shown. (b) Initial Voronoi diagram of randomly-placed points. (c) Final tiling. Edges shown in white are excluded from the optimization. (d) Rendered tiling, using colors from a source image. (e) Tiling of a Lybian Sibyl image. Images used by permission.

function TILEMOSAIC(n, I):
  initialize the centroids $\mathbf{C}_i$ by randomly sampling n points uniformly in the image I
  while the algorithm has not converged
    reestimate the labeling by $L^i_{\mathbf{p}} \leftarrow \begin{cases} 1 & i = \arg\min_i \|R_{\phi(\mathbf{C}_i)}(\mathbf{p} - \mathbf{C}_i)\|_1^2 \\ 0 & \text{otherwise} \end{cases}$
    reestimate the centroids by $\mathbf{C}_i \leftarrow \frac{\sum_{\mathbf{p}} L^i_{\mathbf{p}} \mathbf{p}}{\sum_{\mathbf{p}} L^i_{\mathbf{p}}}$
  return the centroids $\mathbf{C}_i$

Note that this algorithm is no longer optimal, since the centroid update step is not guaranteed to improve the energy function. Furthermore, the algorithm does not take color information into account, until after the tile positions have been chosen. Nonetheless, the algorithm tends to achieve good results in practice.

Tiling can be applied to manually-segmented regions, as before. Edges can be enhanced by removing them from the energy function. Specifically, points $\mathbf{p}$ that lie on image edges are not included in the labeling or centroid computation steps; this discourages Voronoi regions from straddling edges. Tile sizes and shapes can be modified by adjusting the energy function in various ways; the resulting problem is amenable to hardware acceleration [Hau01]. Examples are shown in Figure 3.

3.2 Trial-and-Error algorithms

It is difficult to extend Voronoi methods to take color information into account, and to handle problems where strokes may overlap. So far, the only optimization method applicable to these problems are trial-and-error methods. Trial-and-error methods can be applied to *any* SBR problem. The idea is simple: we propose a change to the image structure. If the proposed change reduces the energy, then the change is incorporated; otherwise, it is discarded. The algorithm then repeats. If the proposal mechanism is well-designed, then the algorithm should eventually converge to a low-energy result. However, there are no guarantees that this will happen, and, even if it does, the computation time

could be substantial.[3] Here is pseudocode for a trial-and-error algorithm:

```
function TRIALANDERROR(I):
    I ← empty image structure
    while not done
        C ← SUGGEST()                    // Suggest change
        if (E(C(I)) < E(I))              // Does the change help?
            I ← C(I)                     // If so, adopt it
    return I
```

Termination conditions are up to the user. For example, the optimization can run for a fixed amount of time, until the user is satisfied with the results, or when only a small portion of the proposals are accepted.

Of critical importance is the design of a good proposal mechanism. Purely random proposal mechanisms can waste substantial time making little progress, whereas hand-tuned mechanisms can quickly get much better results. Trial-and-error algorithms are actually quite closely-related to greedy algorithms, since they both use hand-designed proposals. The main differences are that the trial-and-error includes checks to make sure that the proposal actually improves the image, and that the procedure can iteratively improve previous strokes. Hence, using trial-and-error frees you from the difficult task of designing a mechanism that always makes good strokes.

Haeberli [Hae90] introduced the first trial-and-error algorithm for non-photorealistic rendering; results are shown in Figure 4. In each case, a fixed number of strokes are randomly perturbed, and the perturbations are kept only if the sum-of-squares difference to the source image is reduced.

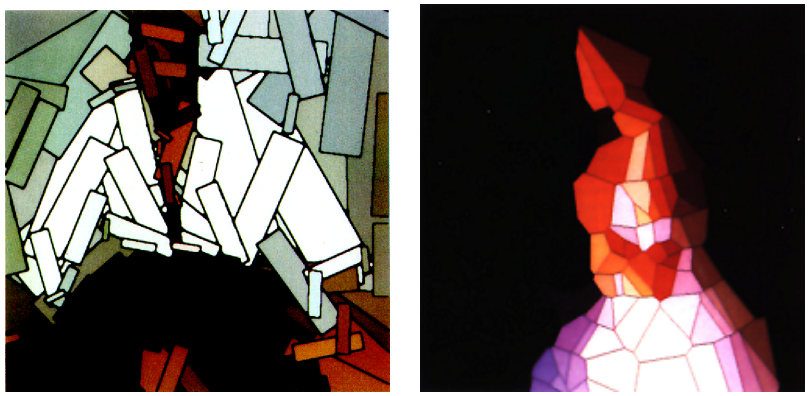
3.2.1 STREAMLINE VISUALIZATION BY TRIAL-AND-ERROR

More recently, Turk and Banks [TB96] demonstrated a trial-and-error algorithm for vector field visualization of streamlines.[4] As described in the introduction, the problem is to illustrate a vector field with streamlines for clear visualization of the vector field. However, a straightforward approach to the problem — simply tracing streamlines from some predetermined starting points — creates irregular streamline spacing that distract from the flow of the vector field. Hence, we need some way of evaulating the quality of the streamline visualization, and then optimizing for that quality measure. Turk and Banks proposed blurring the streamline rendering, and comparing the result to a predefined constant value t. The corresponding energy function is simply

$$E(I) = \sum_{\mathbf{p} \in I} ((G * I)(\mathbf{p}) - t)^2 \tag{6}$$

[3] A closely related technique, called Markov Chain Monte Carlo (MCMC) [Mac98] can give asymptotic quality guarantees. Thus far, MCMC has not been applied to problems in SBR.

[4] Building on the work of Turk and Banks, Jobard and Lefer later described a greedy streamline placement algorithm that is much faster than the trial-and-error method [JL97]. I describe the former method here for completeness, and because it may become useful in future problems.



(a) (b)

Figure 4: Images computed using trial-and-error algorithms [Hae90]. (a) Overlapping rectangular strokes. (b) Voronoi diagram of a set of point centers. For this version of the problem, faster methods were later developed (Section 3.1). Images used by permission.

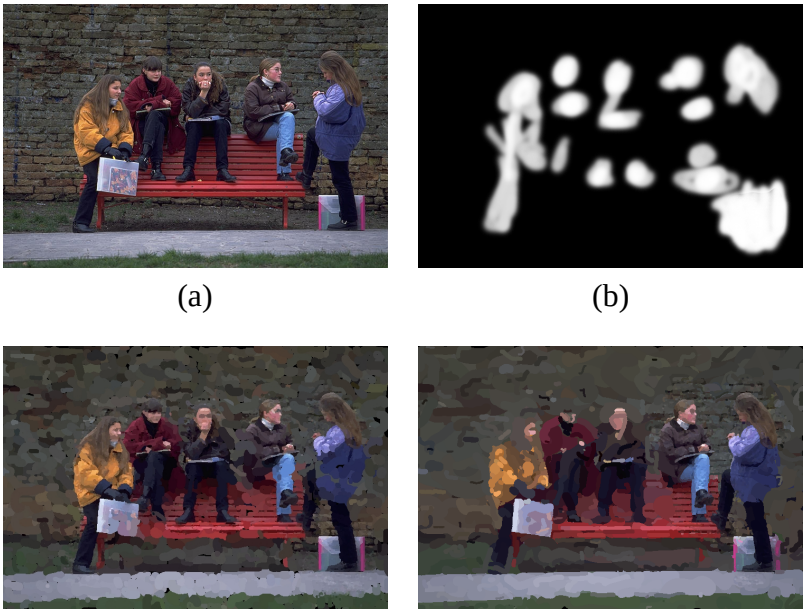
where $(G * I)(\mathbf{p})$ denotes the blurred version of the streamline image. (A similar energy function was used earlier by Salisbury et al. [SABS94].) Note that we could also penalize the deviations of the streamline from the vector field, since our goal is to produce streamlines that exactly follow the vector field. Fortunately, the trial-and-error algorithm is able to enforce this constraint at every step, and thus it is not necessary to include it in the energy function.

In order to apply a trial-and-error algorithm to this problem, we must define the proposal mechanism. While a purely random proposal mechanism may decrease the energy in the long run, it will be far too long to be practical. Hence, Turk and Banks defined many proposal heuristics designed to decrease the energy as much as possible with each step. The first proposals are all streamline placements, in order to quickly fill the image with streamlines. Placing a streamline entails picking a starting point, and then tracing along the vector field some predefined distance. Then, each proposal is one of several possibilities, including adding, deleting, lengthing, and shortening strokes. Each proposal is guaranteed to force the resulting strokes to follow the vector field, and proposals are more likely in image locations that have high energy (i.e. irregular tones). See [TB96] for details.

3.2.2 PAINTERLY RENDERING BY TRIAL-AND-ERROR

I have built a trial-and-error painterly rendering algorithm [Her01]. At a high level, the goal is to seek concise paintings that match a source image closely and cover the image with paint, but use as few strokes as possible. Each brush stroke is defined by a brush radius and a list of control points (Section 5.1). The energy function is

$$\begin{aligned} E(I) &= E_{app}(I) + E_{nstr}(I) + E_{cov}(I) \\ E_{app}(I) &= \sum_{(\mathbf{p}) \in I} w_{app}(\mathbf{p}) \| I(\mathbf{p}) - S(\mathbf{p}) \| \\ E_{nstr}(I) &= w_{nstr} \cdot (\text{number of strokes in } I) \end{aligned}$$



(a) (b) (c) (d)

Figure 5: Spatially-varying style, from [Her01]. (a) Source image. (b) Interactively-painted weight image (w_{app}). The weight image allows the user to specify where emphasis will be assigned. (c) Resulting painting with the given weights. More detail appears near faces and hands. (d) Another choice of weights; detail is concentrated on the rightmost figures.

$$E_{cov}(I) \quad = \quad w_{cov} \cdot (\text{number of empty pixels in } I)$$

This energy is a linear combination of three terms. The first term, E_{app}, measures the pixelwise differences between the painting and a source image S. The number of strokes term E_{nstr} is used to penalize the number of strokes. The coverage term E_{cov} is used to force the canvas to be filled with paint, if desired, by setting w_{cov} to be very large. The weights w are user-defined values. The color distance $\|\cdot\|$ represents Euclidean distance in RGB space. The weights $w_{app}(\mathbf{p})$ are defined by a **weight image** that allows the user to specify spatially-varying weights.

The first two terms of the energy function quantify the trade-off between two competing desires: the desire to closely match the appearance of the source image, and the desire to use as little paint as possible, i.e. to be economical. By adjusting the relative proportion of w_{app} and w_{nstr}, a user can specify the relative importance of these two desires, and thus produce different painting styles.

By default, the value of $w_{app}(\mathbf{p})$ is initialized by a binary edge image, computed with a Sobel filter. This gives extra emphasis to the edges, although a constant weight often gives decent results as well. If we allow the weight to vary over the canvas, then we get an effect that is like having different energy functions in different parts of the image (Figure 5). The weight image $w_{app}(\mathbf{p})$ allows us to specify how much detail is required in each region of the image, and can be generated automatically, or hand-painted by a user. This gives the user high-level control without requiring the user to make every low-level choice.

The trial-and-error algorithm is similar in spirit to Turk and Banks'. However, the problem is much

harder to optimize, because the strokes have many more attributes, there is no predefined vector field, and strokes can overlap. Consequently, the trial-and-error algorithm can take many hours to run, and is not currently very practical. However, the algorithm does give very economical results, an substantial high-level control to a user. See [Her01] for further details of the algorithm.

4 Greedy algorithms

The most common stroke-based rendering algorithms are **greedy**: strokes are added to the image structure in a single pass, and strokes are never modified once they have been created. Greedy algorithms make use of heuristics and carefully-designed placement steps. This means that they can quickly produce high-quality results, but at the cost of flexbility. Greedy algorithms are rarely defined in terms of an energy function, although one is sometimes implicit. In some situations, devising an appropriate energy function may be difficult, but a useful algorithm can be developed without one.

Since strokes are placed once and never modified after, there are really two central questions that define a greedy algorithm:

- Where do we place strokes?
- What shapes will each stroke have?

Each greedy algorithm operates simply by repeatedly placing strokes, making these choices each time.

4.1 Single-point strokes

Haeberli [Hae90] describes a simple, semi-automatic painting algorithm (Figure 6. The user provides a source image. The user sees a rendering of the painting, which is initially blank. Using a mouse or tablet, the user clicks and drags within the painting area. A single brush stroke is placed at the location of each mouse click. The system automatically chooses the color by extracting it from the color of the source image at that point, and orients the stroke in the direction of the gradient of the image. Hence, the user decides where the strokes go, and the algorithm decides what they look like. The user may set other parameters (such as stroke sizes) by adjusting settings or via pressure on a tablet interface. This system provides a fun and easy way to make abstract and attractive versions without requiring the user to possess any drawing skills.[5]

4.1.1 SINGLE-LAYER PAINTERLY RENDERING

Subsequently, a number of commercial software packages (such as Adobe Photoshop, Xaos Paint-Tools, and Microsoft Impressionist) incorporated fully-automatic versions of Haeberli's algorithm; a complete description of such an algorithm by Litwinowicz, along with several enhancements [Lit97].

Litwinowicz's basic algorithm takes a source image and orientation field as input, and generates a painting with a set of oriented, short brush strokes. The brush strokes are placed on a grid in the

[5] Haeberli's system is online at ftp://ftp.sgi.com/sgi/graphics/grafica/impression/index.html.

Figure 6: Interactive painterly rendering system, from [Hae90]. The user clicks on different image locations, and strokes are generated at these locations; the color and gradients of the strokes are taken from a source image. Images used by permission.

image plane, with randomly perturbed positions. Each stroke takes its color from the source image at its location, and its orientation from the orientation field. The strokes are drawn in random order, in order to remove the regularities that would appear otherwise. The orientation field specifies the desired orientation of the strokes and is generated from the source image in a preprocessing step. A simple way to generate this orientation field is to set the orientation $\phi(\mathbf{p})$ at pixel $\mathbf{p}$ to the the normal to the gradient of the image — this gives the direction in which the image is "most constant." However, in constant regions of the image, the gradient will not be well-defined. The orientations in these regions can be filled in using a smoothing algorithm, such as thin-plate spline interpolation. Additionally, strokes may be clipped to image edges to improve the edges of the result. See [Lit97] for details.

Similar short-stroke algorithms have been applied to a number of problems in scientific visualization of tensor-field data [Hea01, Lai01].

4.1.2 MULTIPLE-LAYER PAINTERLY RENDERING

I have developed an extension to these algorithms that can create brush strokes with multiple sizes. We can motivate the algorithm by observing that an artist often will begin a painting as a rough sketch, and go back later over the painting with a smaller brush to add detail. While much of the motivation for this technique does not apply to computer algorithms[6], it also yields desirable visual effects. In this image processing algorithm, fine brush strokes are used only where necessary to refine the painting, and rest of the painting is left coarse. User-controlled emphasis can also be used to define where fine strokes are used. The algorithm is similar to a pyramid algorithm [BA83], in that we start with a coarse approximation to the source image, and add progressive refinements with smaller brushes. In a sense, this algorithm can be viewed as greedily optimizing an energy function that penalizes the difference

[6]For example, one motivation is to establish the composition before committing to fine details, so that the artist may experiment and adjust the composition.

between the painting and the source image, and penalizes the number of strokes.

The algorithm takes as input a source image and a list of brush sizes. The brush sizes are expressed as radii $R_1 \ldots R_n$. The algorithm then proceeds by painting a series of layers, one for each radius, from largest to smallest. Generally, it is most useful to use powers of two: $R_i = R_1 2^{i-1}$, with some user-determined value for R_1. The initial canvas is a constant color image.

A reference image is first created for each layer by blurring the source image. The reference image represents the image we want to approximate by painting with the current brush size. The idea is to use each brush to capture only details which are at least as large as the brush size. We use a layer subroutine to paint a layer with brush R_i, based on the reference image. This procedure locates areas of the image that differ from the reference image and covers them with new brush strokes. Areas that match the source image color to within a threshold (T) are left unchanged. The threshold parameter can be increased to produce rougher paintings, or decreased to produce paintings that closely match the source image.

Blurring may be performed by one of several methods. We normally blur by convolution with a Gaussian kernel of standard deviation $f_\sigma R_i$, where f_σ is some constant factor. Non-linear diffusion [PM90] may be used instead of a Gaussian blur to produce slightly better results near edges, although the improvement is rarely worth the extra computation time. When speed is essential, we use a summed-area table [Cro84].

This entire procedure is repeated for each brush stroke size. A pseudocode summary of the painting algorithm follows.

```
function PAINT(I_s, // source image
               I_p, // canvas
               R_1...R_n) // brush sizes
    Create a summed-area table A from I_s if necessary
    refresh ← true
    foreach brush size R_i, from largest to smallest, do
        Compute a blurred reference image I_{R_i} with blur size f_σ R_i
            from A or by convolution
        grid ← R_i
        Clear depth buffer
        foreach position p on a grid with spacing grid
            M ← the region [p_x − grid/2...p_x + grid/2; p_y − grid/2...p_y + grid/2]
            areaError ← Σ_{p∈M} ||I_p(p) − I_{R_i}(p)||
            if refresh or areaError > T then
                p ← argmax_{p∈M} ||I_p(p) − I_{R_i}(p)||
                PAINTSTROKE(p, I_p, R_i, I_{R_i})
        refresh ← false
```

Each layer is painted using a simple loop over the image canvas. The idea is very similar to Litwinowicz's algorithm. However, we can no longer place samples simply on a jittered grid, since this approach may miss sharp details such as lines and points that pass between grid points. Instead, the algorithm searches each grid point's neighborhood to find the nearby point with the greatest error, and

paint at this location. All strokes for the layer are planned at once before rendering. Then the strokes are rendered in random order to prevent an undesirable appearance of regularity in the brush strokes. In practice, we can avoid the overhead of storing and randomizing a large list of brush strokes by using a Z-buffer. Each stroke is rendered with a random Z value as soon as it is created. The Z-buffer is cleared before each layer. Note that this may produce different results with significant transparency, when transparent objects are not rendered in back-to-front order. The layers of a painting are illustrated in Figure 7.

$\|\cdot\|$, when applied to a color vector, denotes Euclidean distance in RGB space. I also experimented with CIE LUV, a perceptually-based metric [FvDFH90]. Surprisingly, we found it to give slightly worse results — it is not clear why.

PAINTSTROKE in the above code listing is a generic procedure that places a stroke on the canvas beginning at $\mathbf{p}_1$, given a reference image and a brush radius. Following [Hae90], Figure 8(a) shows an image illustrated using a PAINTSTROKE procedure which simply places a circle of the given radius at $\mathbf{p}$, using the color of the source image at location $\mathbf{p}$. Following [Lit97], Figure 8(b) shows an image illustrated with short brush strokes, aligned to the normals of image gradients. Note the regular stroke appearance. In the next section, we will present an algorithm for placing long, curved brush strokes, closer to what one would find in a typical painting.

This technique focuses attention on areas of the image containing the most detail (high-frequency information) by placing many small brush strokes in these regions. Areas with little detail are painted only with very large brush strokes. Thus, strokes are appropriate to the level of detail in the source image.

This choice of emphasis assumes that detail areas contain the most "important" visual information. Other choices of emphasis are also possible — for example, emphasizing foreground elements or human figures. The choice of emphasis can be provided by a human user, as output from a 3D renderer, or from a computational image interpretation.

4.2 Long, curved strokes

4.2.1 PAINTERLY RENDERING WITH LONG, CURVED STROKES

This method can be extended to use long, continuous curves instead of short strokes. In my system, I limit brush strokes to constant color (Section 5.1), and use image gradients to guide stroke placement. The idea is that the strokes will represent isocontours of the image with roughly constant color. Our method is to place control points for the curve by following the normal of the gradient direction. When the color of the stroke is further from the target color in the reference image than the painting, the stroke ends at that control point.

A more detailed explanation of the algorithm follows. The spline placement algorithm begins at a given point in the image $\mathbf{p}_0$, with a given a brush radius R. The stroke is represented as a list of control points, a color, and a brush radius. Points are represented as floating point values in image coordinates. The control point $\mathbf{p}_0$ is added to the spline, and the color of the reference image at $\mathbf{p}_0$ is used as the color of the spline.

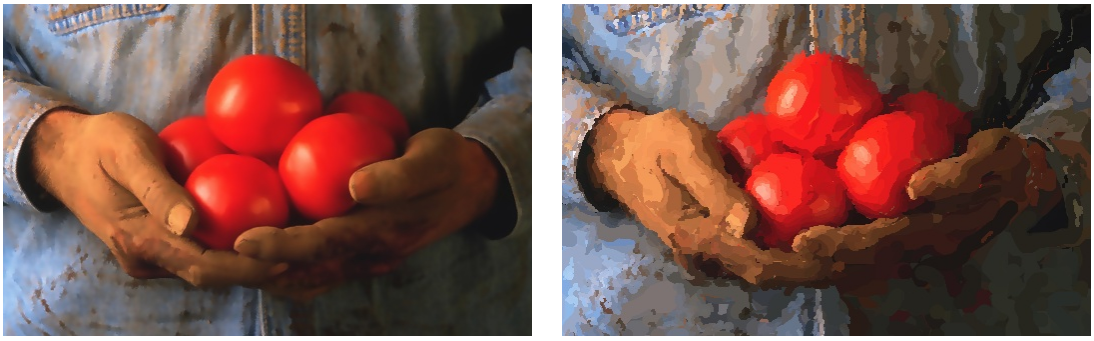
We then need to compute the next point along the curve. The gradient direction θ_0 at this point

Reference images Layers

Figure 7: Painting with three brushes. The left column shows the reference images; the source image is shown in the lower left. The right column shows the painting after the first layer (brush radius 8), the second layer (radius 4), and the final painting (radius 2). Note that brush strokes from earlier layers are still visible in the final painting. (The curved stroke placement is described in the next section).

(a) (b)

Figure 8: Applying the multiscale algorithm to other types of brush strokes. Each of these paintings was created with brush strokes of radius 8, 4, and 2. (a) Brush strokes are circles, following [Hae90]. (b) Brush strokes are short, anti-aliased lines placed normal to image gradients, following [Lit97]. The line length is 4 times the brush radius.



(a) (b)

Figure 9: Using non-linear diffusion to create reference images. (a) Reference image for coarsest level of the pyramid. (Compare to Figure 7(e)). (b) The output image is less noisy, but hard edges are maintained.

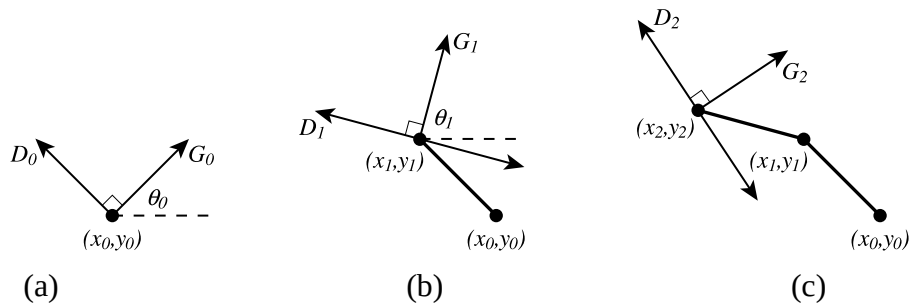


Figure 10: Painting a brush stroke [Her98]. (a) A brush stroke begins at a control point $\mathbf{p}_0$ and continues in direction $\mathbf{D}_0$, normal to the gradient direction $\mathbf{G}_0$. (b) From the second point $\mathbf{p}_1$, there are two normal directions to choose from: $\theta_1 + \pi/2$ and $\theta_1 - \pi/2$. We choose $\mathbf{D}_1$, in order to reduce the stroke curvature. (c) This procedure is repeated to draw the rest of the stroke. The stroke will be rendered as a cubic B-spline, with the $\mathbf{p}_i$ as control points. The distance between control points is equal to the brush radius.

is computed from the Sobel-filtered luminance[7] of the reference image. The next point $\mathbf{p}_1$ is placed in the direction $\theta_0 + \pi/2$ at a distance R from $\mathbf{p}_0$ (Figure 10). Note that we could have also used the direction $\theta_0 - \pi/2$; this choice is arbitrary. We use the brush radius R as the distance between control points because R represents the level of detail we will capture with this brush size; in practice, we find that this size works best. This means that very large brushes create broad sketches of the image, to be later refined with smaller brushes.

The remaining control points are computed by repeating this process of moving along the image and placing control points. For a point $\mathbf{p}_i$, we compute a gradient direction θ_i at that point. There are actually two possible candidates directions for the next direction: $\theta_i + \pi/2$ and $\theta_i - \pi/2$. We choose the next direction that leads to the lesser stroke curvature: we pick the direction $\mathbf{v}_i$ so that the angle between $\mathbf{v}_i$ and $\mathbf{v}_{i-1}$ is less than or equal to $\pi/2$ (Figure 10), where $\mathbf{v}_i$ can be $(R\cos(\theta_i \pm \pi/2), R\sin(\theta_i \pm \pi/2))$. The stroke is terminated when (a) the predetermined maximum stroke length is reached, or (b) the color of the stroke differs from the color under the last control point more than it differs from the current painting at that point. We find that a step size of R works best for capturing the right level of detail for the brush stroke.

We can also exaggerate or reduce the brush stroke curvature by filtering the stroke directions. The filter is controlled by a single predetermined filter constant, f_c. Given the previous stroke direction $\mathbf{v}'_{i-1} = (\Delta x_{i-1}{}', \Delta y_{i-1}{}')$, and a current stroke direction $\mathbf{v}_i = (\Delta x_i, \Delta y_i)$, the filtered stroke direction is $\mathbf{v}'_i = f_c \mathbf{v}_i + (1 - f_c)\mathbf{v}'_{i-1}$.

The entire stroke placement procedure is as follows:

[7] The luminance of a pixel is computed as $Y(r,g,b) = 0.30 * r + 0.59 * g + 0.11 * b$ [FvDFH90].

```
function PAINTSTROKE(p_0, R, I_R, I_p)
    // Arguments: start point p_0, stroke radius (R),
    // reference image (I_R), painting so far (I_p)
    color ← I_R(p_0)
    K ← a new stroke with radius R and color color
    add point p_0 to K
    for i = 1 to maxStrokeLength do
        // compute image derivatives
        g ← (255 * ∂Y_R/∂x (p_{i-1}), 255 * ∂Y_R/∂y (p_{i-1}))

        // detect vanishing gradient
        if R_i ||g|| ≥ 1    // is gradient times length at least a pixel?
            // rotate gradient by 90 degrees
            v_i ← (−g_y, g_x)

            // if necessary, reverse direction
            if i > 1 and v_i • v_{i-1} < 0 then
                v_i ← −v_i

            // filter the stroke direction
            v_i ← f_c v_i + (1 − f_c) v_{i-1}
        else
            if i > 1
                // continue in previous stroke direction
                v_i ← v_{i-1}
            else
                return K

        p_i ← p_{i-1} + R_i v_i / ||v_i||
        if i > minStrokeLength and ||I_R(p_i) − I_p(p_i)|| < ||I_R(p_i) − color|| then
            return K
        add p_i to K
    end for
    return K
```

$Y_R(\mathbf{p})$ is the luminance channel of I_R, scaled from 0 to 1.

A slight speedup may be gained by performing all computations in luminance space, and then restoring the color information, as described in [HJO+01]. The results are very nearly as good as full color processing.

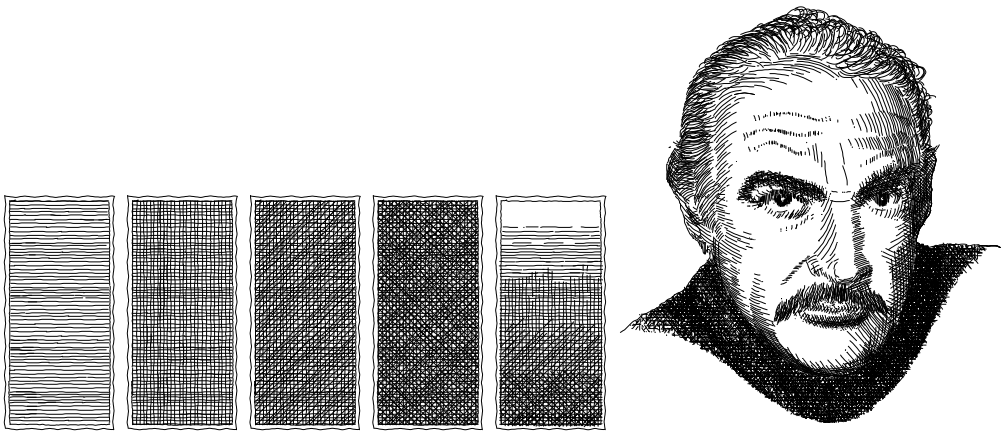


Figure 11: Gray tones generated with a prioritized stroke texture [SABS94]. The strokes in the texture are rendered in a specific order which allows different tones to be generated from a single texture. The right image was generated from a source photo using prioritized stroke textures. Images used by permission.

4.2.2 PEN-AND-INK AND OTHER CURVE TRACING ALGORITHMS

Unfortunately, I ran out of time while writing and this section is brief. Hopefully it will be greatly expanded in a later version of this tutorial. For now, here is a annotated summary of pen-and-ink and other curve tracing algorithms:

- **Interactive pen-and-ink illustration:** [SABS94] introduce an interactive tool for placing pen-and-ink strokes. The user specifies desired tone for a region, and strokes are automatically placed to match these tones. Strokes are stored as **prioritized stroke textures** (Figure 11), which allow the system to render complex hatching patterns. [SALS96] describe an extension that incorporates edge information into the hatching, and [SWHS97] describe a system the allow the user to specify varying orientations for the illustration as well.

- **Streamline visualization:** Jobard and Lefer [JL97] describe a greedy streamline placement algorithm for vector field visualization. Just as with the Voronoi algorithms for stippling, the idea is to penalize stroke density rather than a blurred version of the streamline image.

- **Pen-and-ink illustration of 3D surfaces:** Similar principles can be applied to illustrating surfaces. Typically, the target tones come from a rendering of the surface and the target stroke orientations come from orientation fields defined on the surface. The resulting algorithm is a variation on previous SBR techniques, but with many adjustments as dictated by the pen-and-ink style and the 3D model. [WS94] describe a system for applying prioritized stroke textures to rendering 3D surfaces with texture, and to allowing a user to specify emphasis for different parts of the image; this method is generalized to surfaces with manually-defined orientation fields in [WS96] (Figure 12). In these methods, the hatching attempts to match the orientation, rendered tone, and texture of the image. In [HZ00], we first generate the tone and orientation field for a

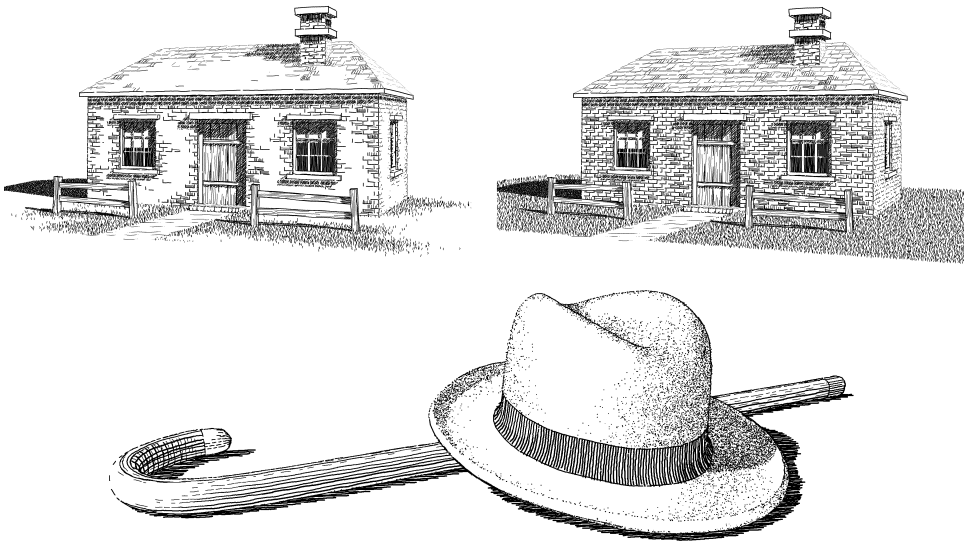


Figure 12: Pen-and-ink illustrations of 3D models, from [WS94] and [WS96], respectively. The left house shows the effect of a user-defined emphasis function; detail is only drawn where specified by the user. The right image shows pen-and-ink illustration of a parametric surface. Images used by permission.

3D model, and extend the method of [JL97] to hatch the surface (Figure 13). A related hatching algorithm is described by [GIHL00].

- **Graftals:** [KMN+99] describe a greedy placement algorithm for stroke placement. The key idea of their technique is to use small illustrations as strokes; for example, a tuft of grass is placed as a unit. This method can render models with an appropriate sense of cartoonish texture.

- **Loose-and-sketchy rendering:** Curtis [Cur98] describes a loose technique for placing pen-strokes to match a black-and-white silhouette rendering. The method can create stiff or loose-and-sketchy rendering styles.

4.3 Globally-greedy

Gooch et al. [GCS02] describes a painterly rendering algorithm that combines elements of optimization and greedy procedures. The algorithm is not defined with respect to a specific energy function, but the overall stroke placement is computed by a global image processing procedure. Individual strokes are then created for their specific regions.

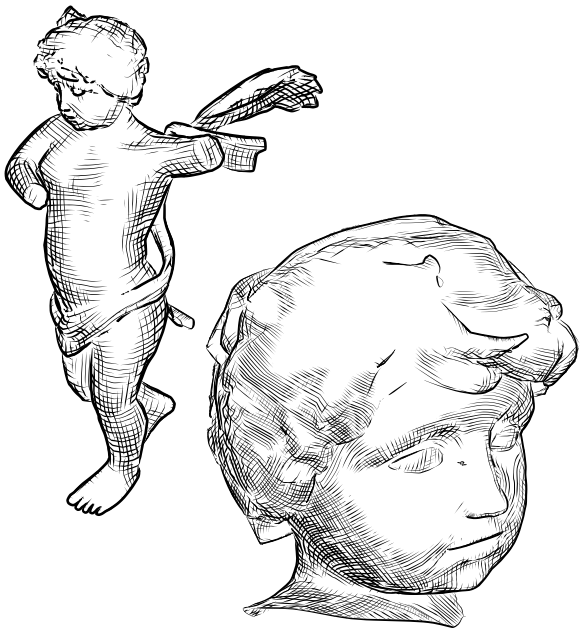


Figure 13: Pen-and-ink illustration of a smooth surface, from [HZ00]. The target orientation field and tones are generated automatically to illustrate the surface.

5 Stroke models

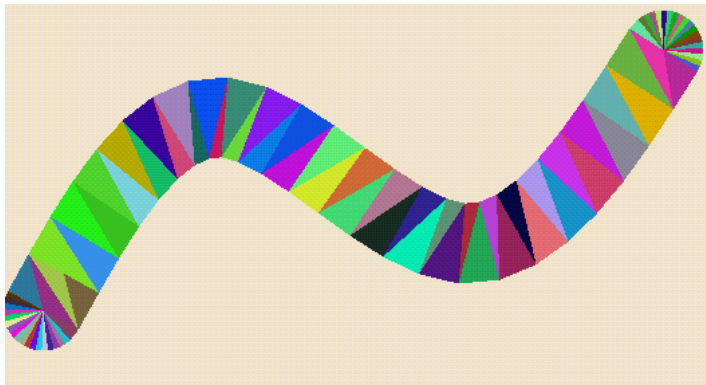
In this section, I describe the curved paint stroke model that I have used. Other stroke models (including stipples, tiles, single-point strokes, and pen strokes) will hopefully appear in a later version of this document. More sophisticated curved stroke models are also possible (e.g. [HLW93, Fra98, NM00]).

5.1 Long, curved strokes with triangle strips

In our system, a basic brush stroke is defined by a curve, a brush thickness R, a stroke color C. The stroke is rendered by placing the stroke color at every image point that is within R pixels of the curve. The curve is an endpoint-interpolating cubic B-spline defined by a set of control points. A dense set of curve points can be computed by recursive subdivision.

Our basic technique for scan-converting a brush stroke is to tessellate the stroke into a triangle

strip:



Given a moderately dense list of control points $\mathbf{p}_i$ and a brush thickness R we can tesselate the stroke by the following steps:

1. Compute curve tangents at each control point. An adequate approximation to the tangent for an interior stroke point $\mathbf{p}_i$ is given by $\mathbf{v}_i = \mathbf{v}_{i+1} - \mathbf{v}_{i-1}$. The first and last tangents are $\mathbf{v}_0 = \mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{v}_{n-1} = \mathbf{p}_{n-1} - \mathbf{p}_{n-2}$.

2. Compute curve normal directions as $\mathbf{n} = (\mathbf{n}_{xi}, \mathbf{n}_{yi}) = (\mathbf{v}_{yi}, -\mathbf{v}_{xi})/\|\mathbf{v}_i\|$

3. Compute points on the boundary of the stroke as points offset by a distance R along the curve normal direction. The offsets for a control point are $\mathbf{a}_i = \mathbf{p}_i + R\mathbf{n}_i$ and $\mathbf{b}_i = \mathbf{p}_i - R\mathbf{n}_i$.

4. Tesselate the stroke as shown above.

5. If desired, add circular "caps" as triangle fans.

This algorithm can also be used with varying brush thicknesses, by specifying a profile curve for the thickness. We do this by assigning a thickness for each control point, and subdividing the thicknesses at the same time as subdiving the control point positions.

This method fails when the stroke has high curvature relative to brush thickness and control point spacing. Such situations can be handled, for example, by repeated subdivision near high curvature points. Generally, we have not found these errors to be of much concern, although they may be problematic for high-quality renderings.

6 Limitations of the energy-minimization approach

At present, it seems unlikely that every desirable SBR style can be formulated in the energy function formulation that I presented in Section 2. There are a number of ways this can be manifested:

- It is difficult to capture "loose and sketchiness" or randomness in an energy function. Moreover, painting and drawing are not deterministic procedures; an artist may produce different images each time. One way to express randomness would be to replace the energy function with the

probability density over renderings. A standard trick for converting an energy to a probability density is to exponentiate and then normalize: $p(I) = e^{-E(I)}/Z$ for some unknown normalization constant Z. Painting is then a process of sampling from this density; the density would usually be conditioned on the input data.

- Several of the greedy approaches described above (such as prioritized stroke textures [SABS94]) and in the next section are difficult to express in terms of energy functions.
- It is sometimes easier to design a direct procedure for a rendering style than to design an energy function, especially since designing styles is a creative process. Often, we design a new algorithm or styles without really understanding what "why they work." Ideally, one would develop additional insight after the fact that allows one to convert the direct procedure to an energy function.

Finally, it bears repeating that direct procedures are much faster than optimization procedures. However, knowing the energy function can often give insight into how the direct procedure works and how to improve it.

7 Related topics

In this section I survey some work related to SBR, but not directly within the scope of this tutorial:

- **Animation.** Several authors have described extensions to the methods for creating animations from input video or 3D animation [Dan99, HP00, Her01, Lit97, MMK+00, Mei96].
- **Image Mosiacs.** Image mosaics, such as "PhotoMosaics" [SH97, FR98] and jigsaw image mosaics [KP02] may also be viewed as SBR problems. A library of images is collected, and the stroke model consists of a reference to one of the images, as well as the image rotation, translation, and optional deformation.
- **Example-based strokes.** A few authors have developed preliminary research in synthesizing SBR imagery by example. Freeman et at. [FTP99] describe a technique for creating new strokes in the style of example strokes. Chen et al. [CXS+01] describe a portraiture system that uses a style learned from examples. Jodoin et al. [JEGPO02] have developed a method for learning hatching styles from examples.
- **Thresholding methods.** Several authors have developed methods for stroke-based rendering where the stroke placements and the stroke tones are decoupled [DOM+01, Fre01, Ost99]. These methods allow greater flexibility by allowing the user to choose stroke placements independent of tones.
- **Rendering from silhouettes.** A number of authors have described SBR algorithms that generate strokes from silhouettes [Cur98, HZ00, MKT+97, NM00]. This process is conceptually straightforward, once one has a procedure for extracting smooth silhouettes and for rendering strokes.

- **Dithering.** The process of placing evenly-spaced stipples is closely related to the problem of halftoning [FvDFH90], which remains an active research area [Ost01]. One can also exploit this observation for painterly rendering [SY00].

- **Hardware-Accelerated Rendering:** Several authors have described SBR systems where strokes are precomputed for texture-maps, and these texture maps are blended in real-time hardware rendering [Fre01, FMS01, KLK+00, PHWF01]. Although these systems, lose the desirable property of rendering strokes in image-space instead of on texture maps, they nonetheless give high-quality, temporally coherent results at very high frame rates.

- **Texture.** A number of methods have been developed to simulate the appearance of realistic media, by approximate simulation [CPE92, CAS+97, Sma90], and/or procedural synthesis [Fra98, GCS02, Her02, Str86].

8 Summary

- Stroke-based rendering is a class of NPR problems where discrete strokes are placed in an image to match some input imagery or other data. Stroke-based rendering encompasses many non-photorealistic rendering problems, including painting, drawing, pen-and-ink, tensor field visualization, and stippling.

- Optimization algorithms explicitly search for an stroke-based rendering I that minimizes some energy function $E(I)$. Voronoi algorithms efficiently optimize stroke densities. Trial-and-error algorithms optimize general energy functions, but can be very inefficient.

- Greedy algorithms place strokes in a single pass. At each step, the algorithm picks a possible stroke location, decides whether to place a stroke, decides the stroke's shape and continues. Many existing greedy algorithms do not have a known explicit energy function formulation.

References

[Aar97] Espen Aarseth. *Cybertext: Perspectives on Ergodic Literature*. Johns Hopkins Univ Press, 1997.

[BA83] P. J. Burt and E. H. Adelson. Laplacian pyramid as a compact image code. *IEEE Trans. Commun.*, 31(4):532–540, 1983.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[CAS+97] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. *Proceedings of SIGGRAPH 97*, pages 421–430, August 1997.

[CPE92] Tunde Cockshott, John Patterson, and David England. Modelling the Texture of Paint. In A. Kilgour and L. Kjelldahl, editors, *Computer Graphics Forum*, volume 11, pages 217–226, 1992.

[Cro84] Franklin C. Crow. Summed-area Tables for Texture Mapping. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):207–212, July 1984.

[Cur98] Cassidy Curtis. Loose and Sketchy Animation. In *SIGGRAPH 98: Conference Abstracts and Applications*, page 317, 1998.

[CXS+01] Hong Chen, Ying-Qing Xu, Heung-Yeung Shum, Song-Chun Zhu, and Nan-Ning Zheng. Example-based Facial Sketch Generation with Non-parametric Sampling. *Proc. International Conference on Computer Vision*, pages 433–438, 2001.

[Dan99] Eric Daniels. Deep Canvas in Disney's Tarzan. In *SIGGRAPH 99: Conference Abstracts and Applications*, page 200, 1999.

[DHvOS00] Oliver Deussen, Stefan Hiller, Cornelius van Overveld, and Thomas Strothotte. Floating Points: A Method for Computing Stipple Drawings. *Computer Graphics Forum*, 19(3), August 2000. ISSN 1067-7055.

[DOM+01] Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling Strokes and High-Level Attributes for Interactive Traditional Drawing. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 71–82. Eurographics, June 2001. ISBN 3-211-83709-4.

[FMS01] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine. *Computer Graphics Forum*, 20(3):184–191, 2001. ISSN 1067-7055.

[FR98] Adam Finkelstein and Marisa Range. Image Mosaics. In *Electronic Publishing, Artistic Imaging and Digital Typography, Proceedings of the EP'98 and RIDT'98 Conferences*, 1998.

[Fra98] Fractal Design. Painter 4.0, 1998. Software package.

[Fre01] Bert Freudenberg. Real-Time Stroke Textures. page 252, 2001.

[FTP99] William T. Freeman, Joshua B. Tenenbaum, and Egon Pasztor. An example-based approach to style translation for line drawings. Technical Report TR99-11, MERL, February 1999.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, 1990.

[GCS02] Bruce Gooch, Greg Coombe, and Peter Shirley. Artistic Vision: Painterly Rendering Using Computer Vision Techniques. *NPAR 2002 : Second International Symposium on Non Photorealistic Animation and Rendering*, June 2002. To appear.

[GG92] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

[GIHL00] Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: An argument for the use of principal directions in 3d line drawings. In *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 43–52. ACM SIGGRAPH / Eurographics, June 2000.

[Hae90] Paul E. Haeberli. Paint By Numbers: Abstract Image Representations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 207–214, August 1990.

[Hau01] Alejo Hausner. Simulating Decorative Mosaics. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 573–578. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[HCK+99] Kenneth Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

[Hea01] C. G. Healey. Formalizing Artistic Techniques and Scientific Visualization for Painted Renditions of Complex Information Spaces. In *Proceedings International Joint Conference on Artifical Intelligence 2001*, pages 371–376, 2001.

[Her98] Aaron Hertzmann. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. In *SIGGRAPH 98 Conference Proceedings*, pages 453–460, July 1998.

[Her01] Aaron Hertzmann. Paint by relaxation. In *Computer Graphics International 2001*, pages 47–54, July 2001. ISBN 0-7695-1007-8.

[Her02] Aaron Hertzmann. Fast Paint Texture. In *NPAR 2002: Proceedings of the Second Annual Symposium on Non-Photorealistic Animation and Rendering*, June 2002.

[HJO+01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image Analogies. *Proceedings of SIGGRAPH 2001*, pages 327–340, August 2001.

[HLW93] S. C. Hsu, I. H. H. Lee, and N. E. Wiseman. Skeletal strokes. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Video, Graphics, and Speech, pages 197–206, 1993.

[HP00] Aaron Hertzmann and Ken Perlin. Painterly Rendering for Video and Interaction. In *Proceedings of the First Annual Symposium on Non-Photorealistic Animation and Rendering*, June 2000.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000.

[JEGPO02] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Pichi, and Victor Ostromoukhov. Hatching by Example: a Statistical Approach. *NPAR 2002 : Second International Symposium on Non Photorealistic Animation and Rendering*, June 2002. To appear.

[JL97] Bruno Jobard and Wilfrid Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proc. of 8th Eurographics Workshop on Visualization in Scientific Computing*, pages 45–55, 1997.

[KLK+00] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Corrêa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000*, pages 527–534, July 2000.

[KMN+99] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-Based Rendering of Fur, Grass, and Trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999.

[KP02] Junhwan Kim and Fabio Pellacini. Jigsaw Image Mosaics. *SIGGRAPH 2002 Conference Proceedings*, 2002.

[Lai01] David H. Laidlaw. Loose, artistic ”textures” for visualization. *IEEE Computer Graphics & Applications*, 21(2):6–9, March / April 2001. ISSN 0272-1716.

[Lit97] Peter Litwinowicz. Processing Images and Video for an Impressionist Effect. In *SIGGRAPH 97 Conference Proceedings*, pages 407–414, August 1997.

[Mac98] David J. C. Mackay. Introduction to Monte Carlo Methods. In Michael I. Jordan, editor, *Learning in Graphical Models*, pages 175–204. MIT Press, 1998.

[Mei96] Barbara J. Meier. Painterly Rendering for Animation. In *SIGGRAPH 96 Conference Proceedings*, pages 477–484, August 1996.

[MKT+97] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, pages 415–420, August 1997.

[MMK+00] Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-based Rendering with Continuous Levels of Detail. In *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 59–66, June 2000.

[Mur98] Janet Murray. *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. MIT Press, 1998.

[NM00] J.D. Northrup and Lee Markosian. Artistic Strokes. In *Proceedings of the First Annual Symposium on Non-Photorealistic Animation and Rendering*, June 2000. To appear.

[Ost99] Victor Ostromoukhov. Digital facial engraving. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 417–424, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

[Ost01] Victor Ostromoukhov. A simple and efficient error-diffusion algorithm. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 567–572. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[PHWF01] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-Time Hatching. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 579–584. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[PM90] Pietro Perona and Jitendra Malik. Scale-Space and Edge Detection using Anisotropic Diffusion. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12:629–639, December 1990.

[SABS94] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen–And–Ink Illustration. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, pages 101–108, July 1994.

[SALS96] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-Dependent Reproduction of Pen-and-Ink Illustrations. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 461–468. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Sec02] Adrian Secord. Weighted Voronoi Stippling. In *NPAR 2002: Proceedings of the Second Annual Symposium on Non-Photorealistic Animation and Rendering*, June 2002.

[SH97] Robert Silvers and Michael Hawley. *Photomosaics*. Henry Holt, 1997.

[Sma90] David Small. Modeling Watercolor by Simulating Diffusion, Pigment, and Paper fibers. In *SPIE Proceedings*, volume 1460, 1990.

[Smi01] Alvy Ray Smith. Digital Paint Systems: An Anecdotal and Historical Overview. *IEEE Annals of the History of Computing*, 23(2):4–30, Apr-Jun 2001.

[Str86] Steve Strassmann. Hairy Brushes. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 225–232, August 1986.

[SWHS97] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image-Based Pen-and-Ink Illustration. In *SIGGRAPH 97 Conference Proceedings*, pages 401–406, August 1997.

[SY00] Michio Shiraishi and Yasushi Yamaguchi. An algorithm for automatic painterly rendering based on local source image approximation. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 53–58, June 2000.

[TB96] Greg Turk and David Banks. Image-Guided Streamline Placement. In *SIGGRAPH 96 Conference Proceedings*, pages 453–460, August 1996.

[WND97] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Developers Press, second edition, 1997.

[WS94] Georges Winkenbach and David H. Salesin. Computer–Generated Pen–And–Ink Illustration. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, pages 91–100, July 1994.

[WS96] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. In *SIGGRAPH 96 Conference Proceedings*, pages 469–476, August 1996.

[Yes79] C. I. Yessios. Computer drafting of stones, wood, plant and ground materials. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, volume 13, pages 190–198, Chicago, Illinois, August 1979.

Interactive NPR

Eric B. Lum and Kwan-Liu Ma
University of California at Davis

1 Introduction

This tutorial surveys the state-of-the-art techniques for interactive non-photorealistic rendering with an emphasis on applications to volume visualization. This first section begins by giving a general motivation behind the need for interactivity in NPR and discusses some of the general approaches toward achieving high-speed rendering. The second section surveys methods that have been developed for interactive non-photorealistic rendering of polygonal surfaces, the area where the vast majority of the research in the field has been devoted. The third section describes an approach for how non-photorealistic rendering can be applied to interactive volume visualization, while the fourth section concludes this tutorial.

1.1 Motivation

Interactivity is often associated with spatial exploration, where parameters such as position, zoom and light direction can be varied over time. The resulting animations controlled by the user allow further insight to be gained in the subject being viewed. Much of the interest in non-photorealistic rendering relates to its use in enhancing spatial structures and clarifying shape. Thus, combining interactive methods with NPR is a natural progression since the combination of the two can be even more effective for illustrating shape.

Non-photorealistic rendering techniques typically have some set of rendering parameters associated with them that vary the style of the resulting images. The required tuning of these parameters does not make these algorithms less desirable, but rather are essential for giving the user the tools necessary for creating the types of images they desire. In the case of scientific visualization, there is rarely any single set of ”correct” rendering parameters as seen in the two images shown in Figure 1. Rendering parameters should be selected that add emphasis and clarity to the aspects of the visualization the user is interested in.

Often, the user of non-photorealistic rendering software is not an artist. They might, for example, be a scientist who would like to generate an image that illustrates a particular structure they are studying. The user might not know which non-photorealistic rendering techniques are appropriate, or might not even have a clear vision of how they would like the resulting visualization to look. If, however, the

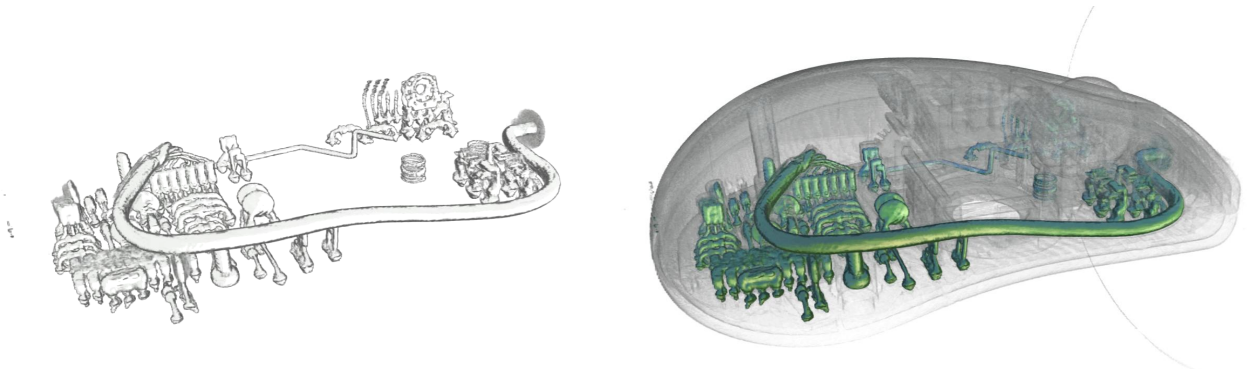


Figure 1: Two non-photorealistic volume renderings of a mouse data set with different sets of rendering parameters. Interactive rendering allows users to select rendering parameters appropriate for their application

tuning of rendering parameters is interactive, the user can quickly explore the parameter space to find an appropriate set of values.

This selection of parameters is an iterative process, where the user changes a parameter, views the result, and then changes parameters further. If the users can see the results of changes in sub-second times they will be much more effective selecting appropriate parameters for their task. Thus, changes in view and object position are only one aspect of interactive rendering. It is equally important to give the user the interactivity necessary to explore the rendering parameter space.

Unfortunately, many factors make interactive non-photorealistic rendering difficult. First, the addition of non-photorealistic rendering techniques typically adds to the amount of calculation required in the rendering process. As a simple example, silhouette edge rendering requires the additional calculations associated with silhouette extraction. Furthermore, the standard technique of rendering data at lower resolution or rendering to a lower resolution window to achieve interactivity is often not suited to the needs of the user when selecting non-photorealistic rendering parameters. Non-photorealistic rendering can be used effectively to clarify fine structures. In order to specify rendering parameters optimized for viewing these structures, objects must be rendered at a high enough resolution for them to be visible.

1.2 Achieving Interactivity

A number of techniques have been developed to facilitate interactive non-photorealistic rendering. Several of these approaches are software based and use data structures that allow for the efficient extraction of features to be illustrated non-photorealistically. To achieve interactivity there has been a trend toward the use of consumer PC graphics cards, many of which have capabilities exceeding those found in high end workstations just a few of years ago. With each product iteration these cards have improved performance, increased memory capacity, and new features supported in hardware. More recent interactive non-photorealistic rendering algorithms load data (geometry and textures) into video memory and then perform rendering calculations on the graphics card to exploit the high performance

memory and processor found on the card, and avoid the I/O costs of sending large amounts of data from main memory across the graphics bus to the graphics card for every frame. Current consumer graphics cards have up to 128 megabytes of memory and support programmable vertex operations as well as the customizable combining of textures.

Another method to achieve interactivity is the use of clusters of commodity PCs for relatively low cost high performance rendering of large amounts of data [LM02] [LMC02]. In particular, by using the aggregated CPU performance, memory capacity, and I/O of a PC cluster with consumer graphics cards, scalable algorithms exist to render large problems at higher frame rates. The use of a PC cluster can still present a number challenges related to how to deal with the relatively slow communication between nodes and how to efficiently distributing the rendering task such that the processing load is well balanced among nodes.

2 Non-Photorealistic Surface Rendering

Most of the research in interactive non-photorealistic rendering has been focussed on the display of polygonal surfaces. These include methods for silhouette edge extraction and rendering, non-photorealistic illumination, and the illustration of surfaces with textured hatching or brush strokes.

2.1 Silhouette Edge Rendering

Silhouettes edges have been utilized for the illustration of surfaces [SABS94, IFP95] since dark lines drawn around an object can be effective in showing an objects structure. These lines can also be used to help indicate an object's spatial relationship with other objects that consist of similar material since dark edges drawn around overlapping objects can provide depth cues with respect to their surfaces.

A number of interactive software techniques have been developed for silhouette edge extraction and rendering. Markosian et al. [MKT+97] describe an interactive probabilistic approach for silhouette extraction using a variation of Appel's hidden-line algorithm. Gooch et al. [GSG+99] present a method which uses an edge structure based on the direction of the neighboring face normals. In a preprocessing step, edges are stored as a hierarchy of arcs on a Gauss map that can be used to extract silhouettes in logarithmic time in the number of edges for smooth models. Since this approach is software based, it also gives them flexibility in changing line styles. An example of an image rendered using their technique is shown in Figure 2. Elber [Elb99] describes a method for line art rendering that uses line art strokes that are generated in a preprocessing step and placed in a data structure for quick extraction during rendering based on view and light source direction. Strokes are used for both hatching and silhouette rendering and are grouped based on surface normal direction. Buchanan and Sousa [BS00] as well as Lake et al. [LMHB00] present software methods for interactive silhouette extraction by setting flags for each edge based on the front/back face properties of the neighboring faces. Hertzmann and Zorin [HZ00] present an approach that uses the concept of dual surfaces for interactive silhouette detection. Their deterministic method works for both orthographic and perspective projection and avoids complete traversal of all edges of the mesh. Tests they conducted suggest performance to be roughly linear to the number of silhouette triangles.

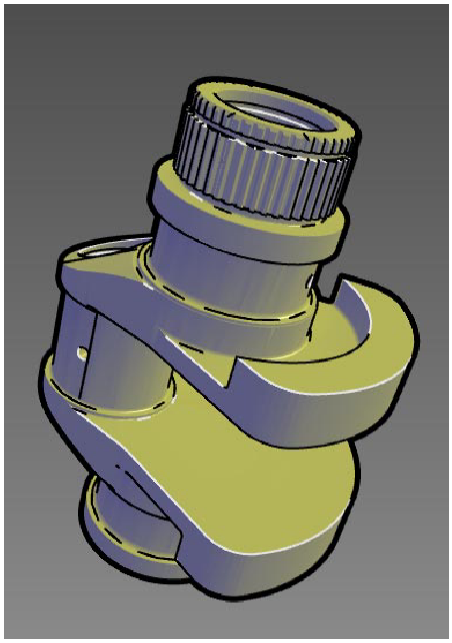


Figure 2: Result from Gooch et al. [GSG+99] showing interactively rendered technical illustration with non-photorealistic silhouette edges and shading. Their approach uses a Gauss map to efficiently extract silhouette edges that are rendered in varying line styles.

Several methods have also been developed that utilize graphics hardware for silhouette edge rendering. Gooch et al. [GSG+99] describe how environment mapping can be used for single pass hardware accelerated silhouette edge rendering. This is accomplished by using an environment map that is dark in regions perpendicular to the view direction. They note that the resulting silhouettes are artistically interesting but can be inappropriate for technical illustration since line thickness is nonuniform. Raskar and Cohen [RC99] render silhouette edges in hardware using multi-pass rendering. They draw all polygons once with z-buffer on, and then draw all back-facing in a second pass as either wireframe with z-buffer equality or as filled polygons that have been moved toward the camera in the second pass. In following work, Raskar [Ras01] presents a method for the rendering of ridges, valleys and silhouettes edges by rendering extra polygons for each edge that are generated on the graphics card using programmable vertex operations.

Northrup and Markosian [NM00] describe a screen/object space hybrid approach for silhouette rendering where silhouette edges are detected in object space, but viability and adjacency is determined in screen space. This allows for the linking of silhouette edge segments into smooth curves that are rendered in various styles interactively controlled by the user. An example that demonstrates the artistic silhouette edges produced by their technique is shown in Figure 3.

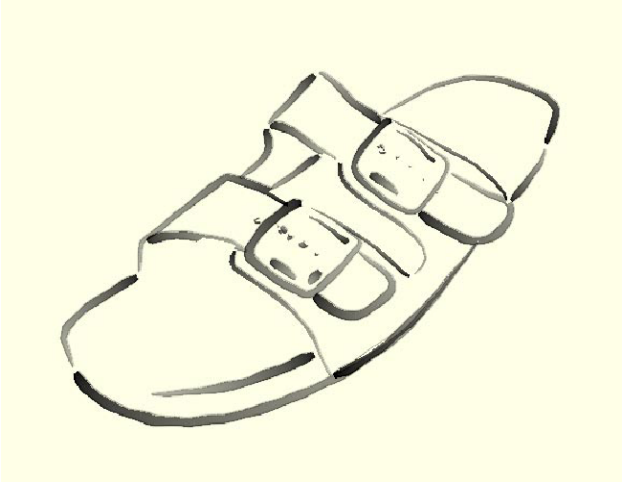


Figure 3: Result from Northrup and Markosian [NM00] that demonstrates their artistic silhouette edge rendering. Their hybrid object/screen space method links silhouette edges into curves that are rendered in various styles.

2.2 Non-Photorealistic Illumination

Lighting can be extremely effective in conveying the shape and structure of an object. Artists often illustrate lighting through not only the varying of pigment value (intensity) but also through the variation of color temperature [Mac99]. Directly illuminated objects are represented with warmer colors which include yellow, orange and red. Gooch et al. [GGSC98] use cool to warm shading for the illumination of surfaces. They present a shading model which allows extreme changes in color value to be reserved for outlines and highlights and describe how an approximation of their model can be implemented using graphics hardware that supports Phong shading. Sloan et al. [SMGG01] present methods that allow users to interactively specify their own non-photorealistic shading models. Their system provides real-time feedback which gives users the ability to explore a wide range of artistic styles.

Lum and Ma [LM01] describe an inverted non-photorealistic illumination model that renders illuminated objects by building up layers of simulated pigment based on the surface darkness. By adding paint to a simulated white canvas where objects are dark, they achieve color coherence similar to that which would appear from the mixing of a limited palette of paints. However, their implementation uses ray-tracing and is therefore not fully interactive at higher screen resolutions.

2.3 Non-Photorealistic Texture

Textured surfaces in the form of either brush strokes or hatching strokes can help to clarify shape and indicate material properties. Klein et al. [KLK$^+$00] describe what they call ”art maps”, which consist of multi-resolution mipmapped textures with constant sized brush strokes. This permits rendering in hardware with brush stroke textures that do not vary in size regardless of a polygon’s projected size in screen space.

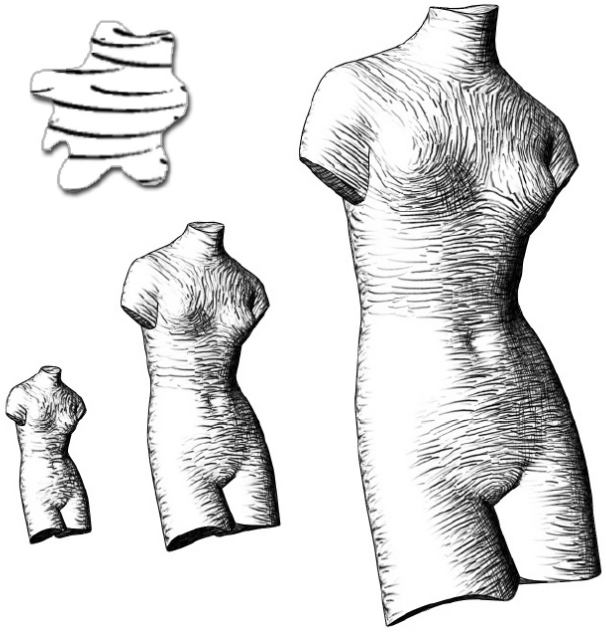


Figure 4: Result from Praun et al. [PHWF01] showing Venus model rendered using tonal art maps(TAM). Notice that hatch strokes do not change in size as the size of the model in projected screen space is reduced.

Praun et al. [PHWF01] extend this technique with mipmapped textures they call ”tonal art maps” (TAM) which contain pre-rendered hatched textures of varying density(tone) and resolution. In order to preserve tone coherence between tone levels, lighter levels contain subsets of darker leveled tones. By utilizing the R,G,B color channels of each texture, they are able to perform real-time hatching with six grayscale tone levels using a single rendering pass with two texture units. Hatching direction is oriented along the principal curvature directions of a surface such that stroke directions follow the shape of an object using principal curvature directions as illustrated in Figure 4. Since the lighter tone levels are subsets or darker levels, and the TAMs are applied in object space, the resulting animation exhibits frame-to-frame coherence and avoids the ”shower-door” effect. In further work by Webb et al. [WPFH02], two methods for finer control of tone using newer features found on consumer PC graphics hardware are presented. The first technique utilizes volumetric textures, where tone density is varied over the third dimension. The second method uses pixel shaders to provide a per-pixel lighting texture thresholding for reduced aliasing. Examples of images rendered using each of these techniques are shown in Figure 5. In some ways similar to TAMs, Lake et al. [LMHB00] describe pencil sketch shading using multiple textures of varying pencil stroke density. The textures can be applied in screen space for a hand-drawn appearance as seen in Figure 6 or the textures can have fixed object space coordinates for better frame-to-frame coherence.

Majumder and Gopi [MG02] present a technique for using graphics hardware accelerated charcoal rendering. They introduce what they call the ”contrast enhancement operator” (CEO) which modifies lighting contrast to account for the limited dynamic range found in charcoal drawings. Charcoal is

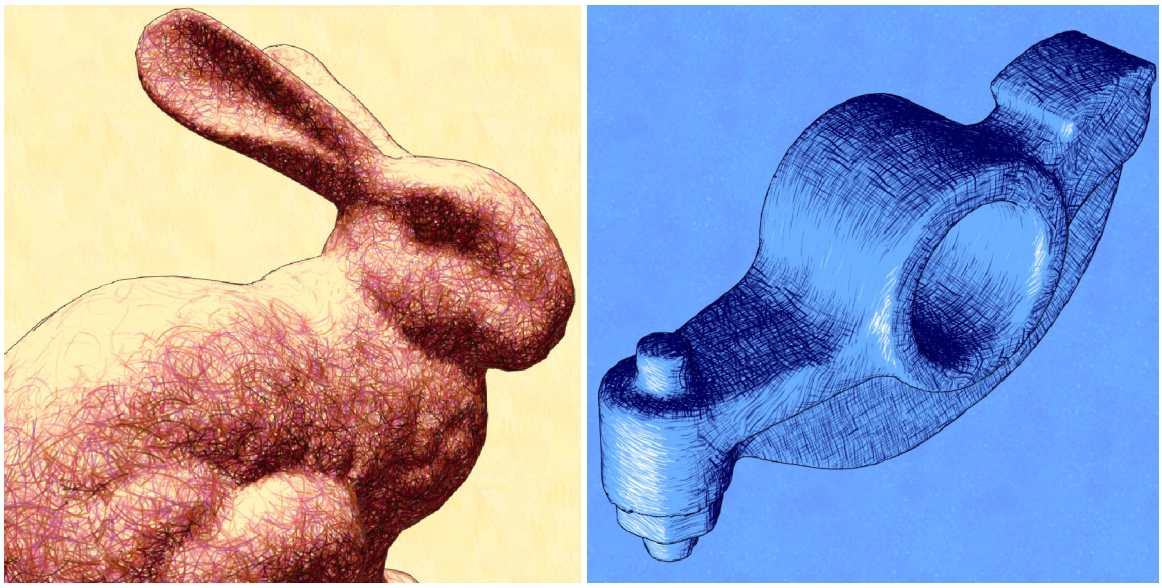


Figure 5: Results from Webb et al. [WPFH02] that show tonal art maps with fine tone control. Left image shows bunny model rendered using their volumetric texture technique, while the right image of a rocker arm uses their pixel shader method.

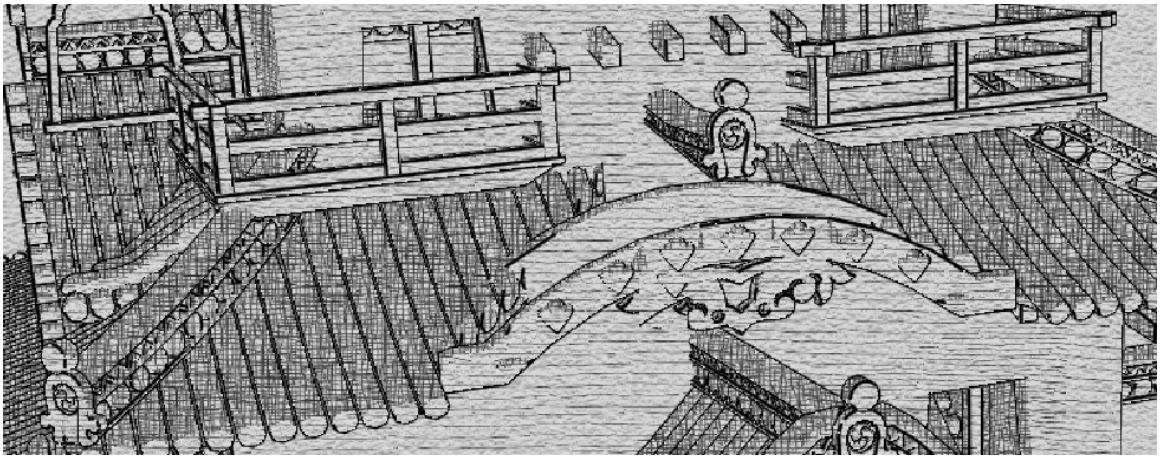


Figure 6: Result from Lake et al. [LMHB00] showing shading with pencil textures of varying density. These textures can be applied projected screen space for a more hand-drawn appearance or can be attached in object space for smooth animation.

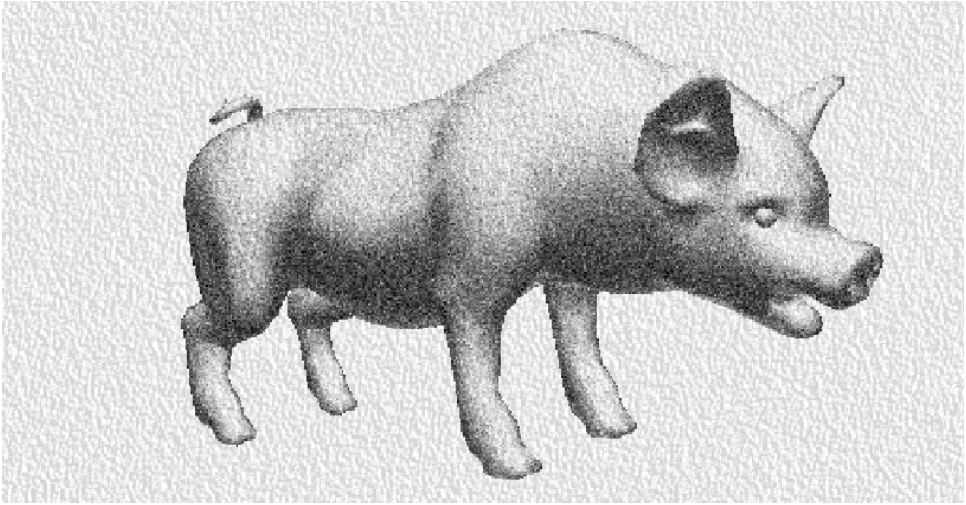


Figure 7: Result from Majumder and Gopi [MG02] shows a pig model rendered in a charcoal style using texture hardware. They use a ”contrast enhancement operator”(CEO) which modifies lighting contrast to account for the limited dynamic range found in charcoal drawings.

represented using grain textures of varying densities and can be displayed at real-time rendering rates using a consumer PC graphics card. Their method also allows the user to interactively vary parameters that influence rendering style, including grain density, paper coarseness and the amount of smudging. An example of an image generated using their technique is shown in Figure 7.

Several authors have described particle-based approaches for interactive storke-based rendering. Kowalski et al. [KMN+99] describe a method for rendering fur, grass, and trees using procedural stroke-based textures. They use graphics hardware to render reference color and ID images. The color image is used to guide the placement of strokes in screen space while the ID image maps pixels to object space to keep track of stroke positions in 3-D space for frame-to-frame coherence. Kaplan et al. [KGC00] use interactive particle systems to render polygonal models in a number of styles. The user is able to control how the particles are rendered and can also change their density and position to mimic a number of hand drawn effects as shown in Figure 8. Like the oriented strokes introduced by Meier [Mei96], the particles exist in 3D object space thus preserving frame-to-frame coherence. Cornish et al. [CRL01] use view-dependent particle systems for stroke-based rendering. By using multi-resolution data structure, they are able to control particle density for efficient interactive rendering.

Mohr and Gleicher [MG01] describe the replacement of the OpenGL runtime graphics library with a NPR version that extracts scene information for non-invasive interactive rendering with alternate styles. Frudenberg et al. [FM01] present a pen-and-ink style game engine that permits interactive walk-throughs with frame-coherent hatch textures.

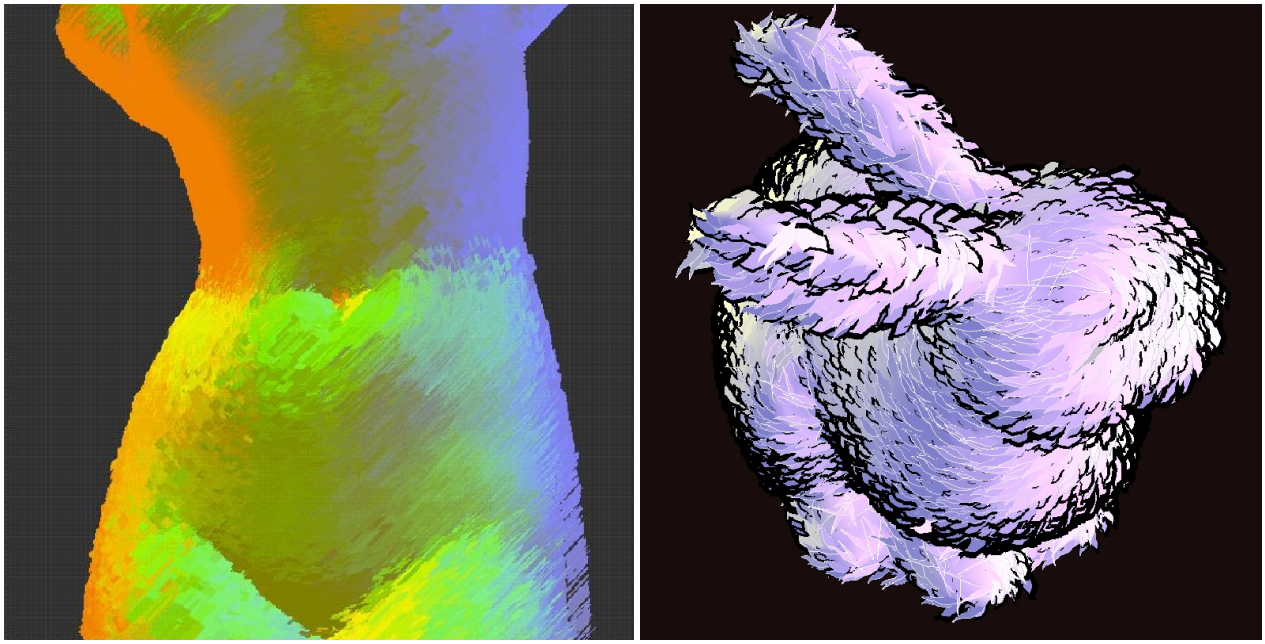


Figure 8: Results from Kaplan et al. [KGC00] shows different artistic styles rendered using their particle approach. The particles exist in 3D object space thus preserving frame-to-frame coherence.

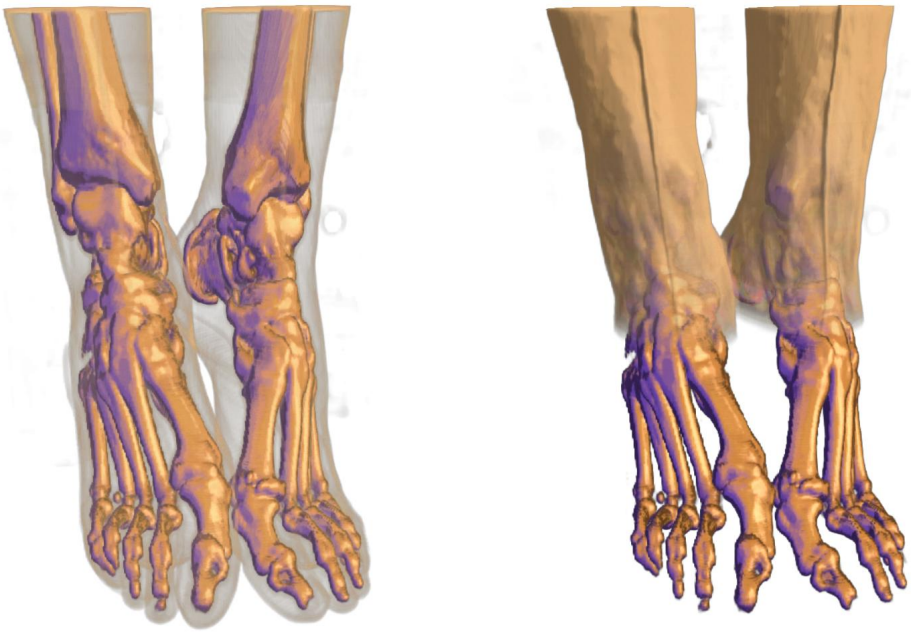


Figure 9: Left: Using gradient based feature enhancement the skin surface is made visible. Right: The skin and flesh are rendered in a more photorealistic style, while the bones are rendered with hue varied shading and silhouettes edges.

3 Non-Photorealistic Volume Rendering

Direct volume rendering is a technique for the display of data that samples a 3D space [Lev90]. It has wide application in science and engineering including the study of data sets from medical MRI and CT scans, weather simulation, structure analysis, air flow surrounding an aircraft, and many other investigations. Volume data sets often consist of regularly spaced scalar samples, called voxels (volume elements), which are mapped to color and opacity values using a lookup table referred to as transfer function. Rendering consists of the display of the accumulation of the semi-transparent material in the volume. By varying the transfer function, the user is able to assign colors to different structures in the volume based on scalar value, and can visualize structures of interest by making them more opaque than surrounding material.

Strictly speaking, traditional volume rendering is not necessarily a form of photorealistic rendering, since it often involves the visualization of values like temperature, pressure, or density that are not directly visible in reality. Never the less, by applying artistically inspired NPR methods to volume rendering, creating more effective visualizations is possible. Treavett and Chen [TC00] present a technique for pen-and-ink style volume rendering. Ebert and Rheingans [ER00] describe a software approach for non-photorealistic volume rendering using techniques that include silhouette edge rendering, cool to warm shading, and depth-cued color variation. Treavett et al. [TCSJ01] describe a software method for non-photorealistic rendering of volumes in painterly and pen-and-ink styles, as well as what they call ”artistic modelling”. This technique involves manipulation of the volume data itself, including the distortion of the volume to give a warped appearance or the addition of solid stroke textures to give the effect of rough carving. In our work [LM02] we show how texture hardware on a consumer PC graphics cards can be used for interactive volume rendering with a number of non-photorealistic techniques. We also describe how large volumes can be interactively rendered non-photorealistically using a cluster of graphics card equipped PCs.

3.1 Volume Rendering With Graphics Hardware

Direct volume rendering can be accomplished by drawing a set of view-aligned polygon slices that sample a 3-D texture containing the volumetric data [VGH96]. By using pixel textures, which allow a texel value to store coordinates into a second texture, scalar and lighting information can be encoded into a single static texel, with transfer function and shading changes occurring through the variation of a single texture [MHS99]. Kniss et al. [KKH01], describe how multi-dimensional transfer functions can be interactively specified and rendered with traditional lighting by using multi-textured/multi-pass rendering. Engel et al. [EKE01] use a technique they call pre-integated volume rendering, which also uses multi-textured/multi-pass techniques for improved rendering accuracy using fewer polygon slices.

Paletted textures store indices into a color palette that samples the RGBA color space. Through the manipulation of the palette over time, the textures can be varied based on transfer function or the viewing parameters without changing the data stored in the textures themselves. It is important to use a representation that avoids any manipulation of data that is stored for every voxel since volumetric data sets tend to be large, and traversing the entire volume in software can severely hamper interactivity. Unlike pixel textures which are filtered prior to their texel lookup, paletted texture are

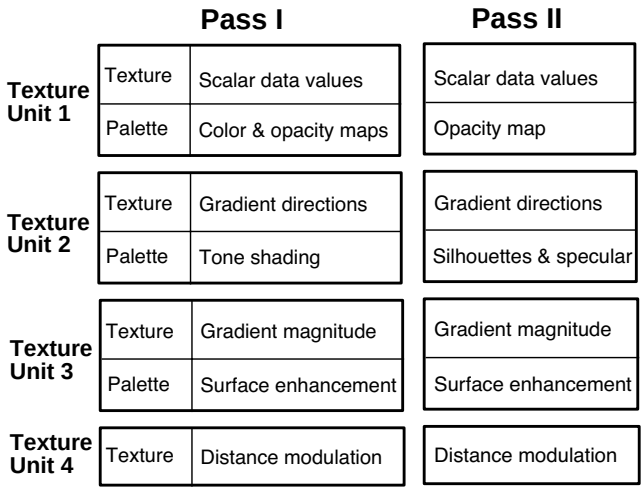


Figure 10: Rendering using two passes each with 4 texture units.

filtered after color lookup in RGBA space, permitting non-linear palette mappings to be used with linear interpolation in hardware.

Multi-texturing allows several textures to be combined on a single polygon during the rendering process [WNDS99]. By utilizing several separate volumetric textures that store scalar data value, gradient magnitude, and gradient direction, and combining them with properly adjusted color palettes, several different non-photorealistic rendering techniques can be rendered in hardware. The Nvidia Geforce3 has four texture units, which means a total of four textures can be blended onto a single polygon. For some effects, additional rendering passes can be added, which consist of blending additional textured polygons on top of the previously rendered polygons in the frame buffer.

An example of how the textures can be allocated in these passes is shown in Figure 10. In this example each of the four texture units is assigned a different texture. The first texture consists of the original scalar values stored in a paletted 3-D texture. The second stores the normalized gradient direction of each voxel. One method for encoding these directions is to use 8-bit paletted textures, with each direction quantized to one of 240 vectors obtained from the faces of a subdivided combination of a dodecahedron and icosahedron [VGH96]. The gradient direction information is useful for lighting and silhouette edge rendering. A third texture contains gradient magnitudes which can be used for enhancing surfaces. The fourth contains a 1-D texture for manipulating color and opacity based on the spatial properties of voxels. The actual number of texture units needed as well as the required number of rendering passes depend on the number of NPR techniques to be applied as will be discussed in the following sections.

3.2 Hue Varied Shading

The cool to warm shading described by Gooch et al. [GGSC98] can be implemented by modulating each voxel color with a lighting texture that manipulates color temperature. As described in the previous section, a paletted normal direction texture can be used, with each texel containing an index into a

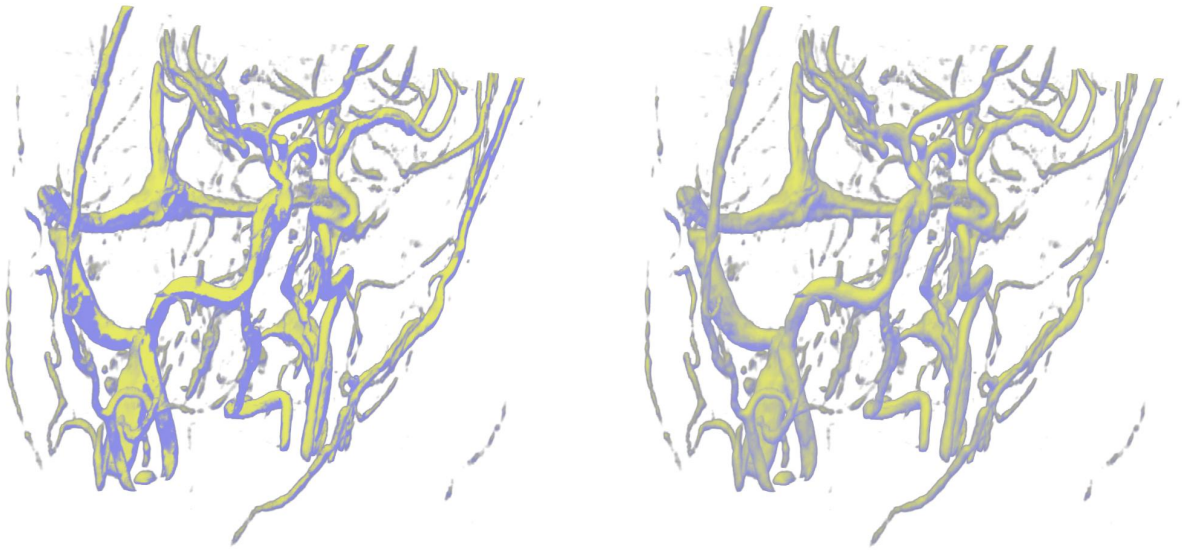


Figure 11: Left: Vessel with abrupt color temperature varied shading. Right: smooth shading.

sampling of a normalized vector space. The palette for this texture is created by first calculating the dot product between each possible normal direction and the light direction. Once the dot product is calculated, the color temperature variation for that product is looked up in a shading colormap specified by the user.

The manipulation of the saturation of the colors used in the shading colormap can control the degree temperature variations in shading are visible. By making the colors more saturated, the effects of this type of shading becomes more subtle permitting more of an objects original color (as specified in the transfer function) to be seen. Furthermore, by changing color value across the shading colormap more traditional lighting with variation in color intensity can be produced.

In addition by making the transition between cool and warm colors relatively short, it is possible to produce the abrupt lighting transitions seen in the image of blood vessels shown in the left image of Figure 11. This can be contrasted with the relatively smooth lighting found in the right image of Figure 11.

3.3 Silhouette Edge Illustration

Silhouette edge rendering can be particularly useful in volume rendering applications since transfer functions are often set such that objects are semi-transparent, sometimes making spatial relationship difficult to determine. Silhouette rendering can be accomplished by using the paletted gradient direction texture with the color palette adjusted such that voxels with gradient perpendicular to the view direction are modulated to black. Alternately this could also be done using a gradient direction texture with per pixel environment mapping similar to the technique described by Gooch et al. [GSG+99].

The image on the top of Figure 12 shows fine blood vessels that are enhanced by rendering silhouette edges. Notice that the spatial relationships between the overlapping vessels near the top of the left

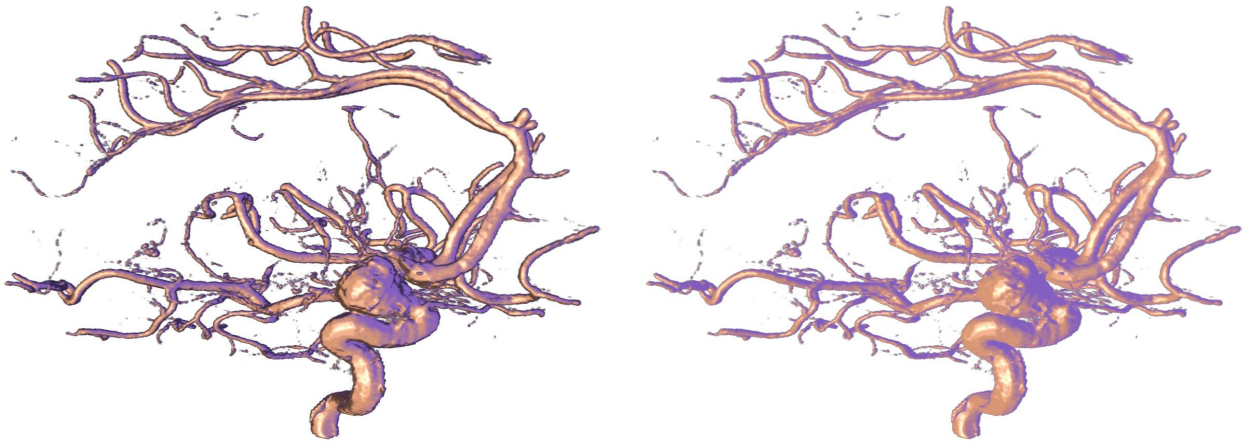


Figure 12: The silhouettes shown on the left image help clarify the spatial relationship between vessels.

image are clearer than those found in the right image without silhouette edge rendering in Figure 12.

3.4 Surface Enhancement

Much of the appeal of direct volume rendering, versus other volume visualization techniques like isosurface rendering, is its usefulness in visualizing continuous regions of semi-transparent material. However, enhancing surfaces can still be effective in clarifying some structures in a volume. Gradients have been used for surfaces enhancement in direct volume rendering applications [Lev90]. Since the transition between features in a volume tend to have the highest gradient magnitude, increasing the opacity in these regions can help to clarify surfaces. The skin surface shown on the left side of Figure 9 is made visible through gradient enhancement, with the underlying material still visible.

Surface enhancement can be implemented by assigning one texture unit a gradient magnitude texture, and allowing the user to specifying a gradient opacity map that modulates the rendered voxel based on gradient. Silhouette edges and specular highlights are typically associated with surfaces, and can be inappropriate for constant valued, semi-transparent regions in a volume. By rendering silhouette edges in a second rendering pass, a separate gradient enhancing functions for the specular and silhouette rendering, resulting in silhouette edges and specular highlights that are only rendered in regions of high gradient.

3.5 Color Based on Position

Color can be manipulated based on distance to improve depth perception [FvFH96]. Aerial perspective has been used by painters to convey depth through the variation of color hue and value based on depth. Typically warmer hues are used for the foreground and become cooler in the background. In addition, color values tend to become lighter and less intense with distance [Kun99].

This can be implemented in hardware using a 1-D texture that modulates the color of the rendered volume along some direction. The depth cues provided by the variation in color are evident in the left

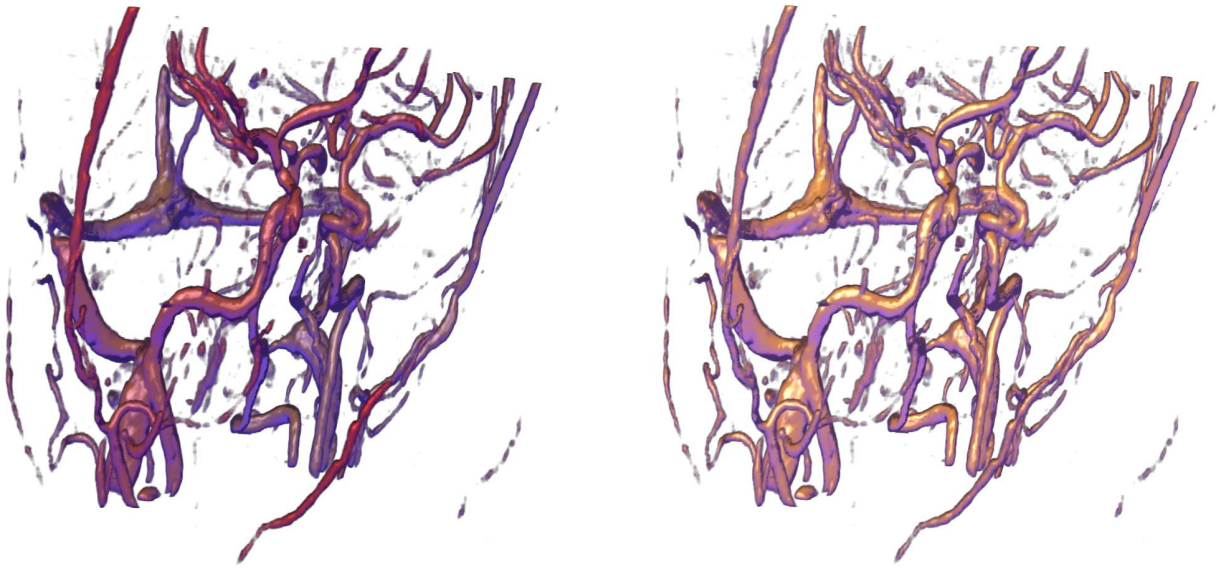


Figure 13: The depth cues provided by the variation in color temperature clarify spatial relationships between vessels. Left: Depth based color manipulation Right: Without depth based color manipulation .

image of Figure 13 where the warmer colored foreground vessels appear closer. This can be contrasted with the right image of Figure 13 where the spatial relationship between vessels is less clear.

Through non-linear fading of the alpha channel along the view direction, closer material can be made more transparent making underlying features more visible, with foreground material still slightly visible to provide context for the features of interest as shown in the right image in Figure 1. Figure 9 on the other hand shows a fading of opacity based on vertical position.

3.6 Multiple Rendering Parameters

Non-photorealistic rendering can be used to add emphasis to specific aspects of a data set. One way this can be accomplished is to use multiple rendering styles for different objects in a volume. For example non-photorealistic and more traditional rendering methods can be combined, making the different materials more distinguishable, or rendering styles can be varied to emphasize or deemphasize the different types of features in a volume. One result of using multiple sets of rendering parameters is that the parameter space is multiplied in complexity, making interactivity all the more important.

Hauser et al. describe how multiple rendering techniques can be combined when visualizing a single volume to better illustrate different types of objects in a volumes [HMBG00]. A similar capability can be accomplished by permitting the user to specify multiple transfer functions, each with its own set of non-photorealistic rendering parameters. Each transfer function can be set to render a different type of object in the volume, with rendering requiring an extra set of rendering passes for each additional transfer function.

The right image in Figure 1 shows the use of this technique, where the internals of the Microsoft

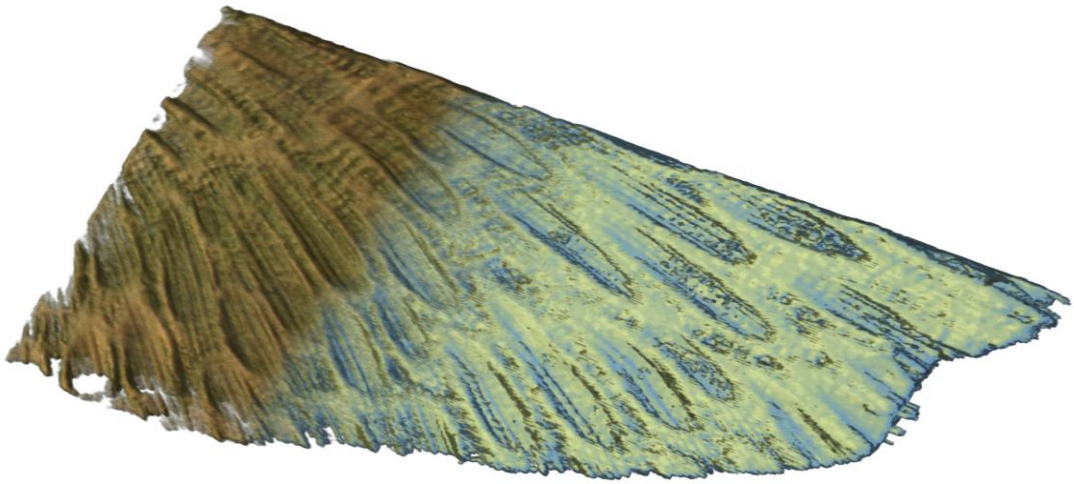


Figure 14: Non-photorealistic and photorealistic rendering styles are mixed in this image of coral.

mouse have been rendered in a non-photorealistic style, while the external plastic parts are rendered more photorealistically. The right image in Figure 9 show bones rendered non-photorealistically using cool to warm shading and silhouettes. The skin on the other hand is rendered in a more photorealistic style, with a fade that allows the several of the bones to be unobstructed by the flesh and skin. In Figure 14 a piece of coral is shown using non-photorealistic and photorealistic rendering parameters. The smooth transition between parameters sets is accomplished using position-based opacity modulation.

3.7 A Complete Example

To summarize the process, we use a CT scan of a Microsoft mouse to illustrate the effect of each non-photorealistic rendering technique. As shown in Figure 15(a), when the data set is rendered using only the transfer function without lighting or NPR enhancements it is very difficult to acquire intuition about the spatial structure of each object, particularly with respect to depth.

With addition of cool to warm shading, shown in Figure 15(b), it becomes easier to determine the surface orientations as seen in Figure 15(c). The addition of warmth and coolness to the volumes does not yield distinctly warm or cool colors, but rather results in a variation in relative temperature to indicate shape information. Notice however, that there is little variation in color intensity across each volume making it still difficult to gain depth clues as to the spatial interactions of the rendered structures. With the addition of the silhouettes edges seen in Figure 15(d) the individual structures become much clearer as seen in Figure 15(e). For example, the separation between the two capacitors near the center of the mouse is much more distinct that when only cool to warm shading is used. Next, Figure 15(f) displays depth color cues for the volume, and Figure 15(g) shows the result after the volume is modulated by these cues. This has the subtle effect of adding warmth to the nearer portion

of the mouse. Finally in Figure 15(h) we see the result when a second set of rendering parameters is used to render the outer shell of the mouse with a more photorealistic rendering style. Using depth based variation in opacity the closer portions of the mouse are more transparent allowing the inside to be seen.

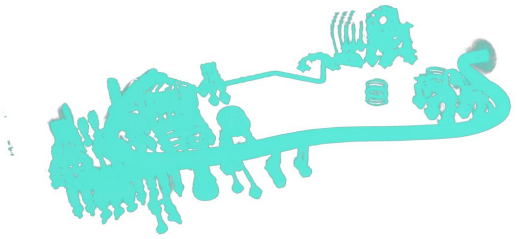
3.8 Parallel Rendering and Performance

One significant limitation of volume rendering using consumer PC graphics hardware is the limited amount of video memory. For example, the Nvidia Geforce3 has between 64 and 128 megabytes of video memory that is shared between the frame buffer and texture memory. It is very desirable to fit the volume being rendered entirely in texture memory to avoid having to swap data into the graphics card from main memory over the relatively slow graphics bus. By subdividing the volume spatially and distributing it across a cluster of PCs equipped with graphics cards it is possible to fit significantly larger volumes into the aggregated video memory of the entire cluster. In addition to the larger amounts of texture memory provided by a PC cluster, performance improvements also result from the combined fill-rate of multiple graphics cards.

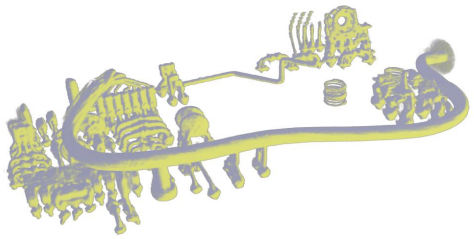
In our implementation we describe in [LM02] we subdivide and distribute the volume to different nodes of a cluster using k-d tree subdivision. During rendering, for each frame, every node on the cluster renders its subvolume and composites the resulting subimage using binary-swap [MPKH94] with the final image being sent to the host for display. We used a PC cluster with nine computers, each with an AMD Athlon 1.3 Ghz processor, one gigabyte of PC133 SDRAM and a Geforce3 with 64 megs of video memory. Eight of the computers use 100-Base-T fast Ethernet. The ninth host computer, used for final display and user interface control, has a gigabit Ethernet connection to the cluster's switch. With this low cost cluster were able to render to a 512×512 window, a $512\times512\times512$ volume at about 2.0 frame per second using two rendering passes. If four rendering passes are used to render the volume using two sets of rendering parameters, the frame rates drops to about 1.2 frame per second. Our experimentation showed this framerate is sufficiently high to make possible interactive exploration of rendering parameter space, in particular the variation of transfer function, viewing direction and non-photorealistic rendering parameters. This permits the tuning of parameters for the creation of meaningful images that illustrates specific structures in a volumetric data set.

There are some limitations to this approach for interactive non-photorealistic volume rendering. First, rendering rates are still not fast enough for the real-time generation of smooth animation. Higher performance can be achieved by rendering fewer axis aligned polygons, or by using fewer NPR techniques requiring fewer texture units and rendering passes. But the time required for reading the frame buffer from each node and the communication time for compositing the resulting sub-images soon becomes a bottleneck.

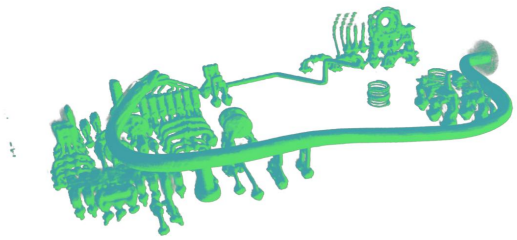
The use of graphics hardware also limits the size of the volume each node can render by the combined texture memory of the cluster. For example, rendering a $1024\times1024\times1024$ volume with eight PCs would require each node to render a $512\times512\times512$ volume which would not fit in texture memory and would make necessary the relatively slow transfer of large amounts of data across the graphics bus for every frame. Despite these limitations, the use of graphics hardware allows for rendering performance that exceeds any software approaches.



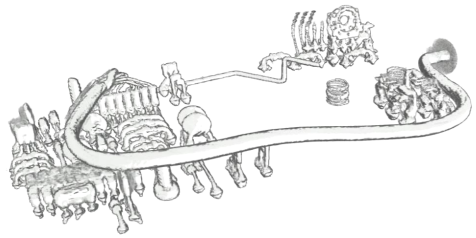
(a) Volume without lighting



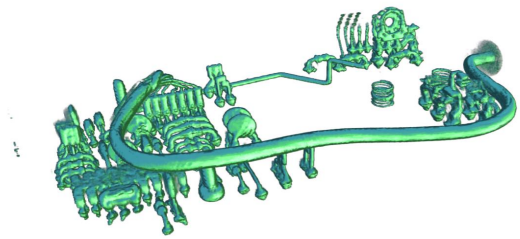
(b) Cool to warm shading contribution



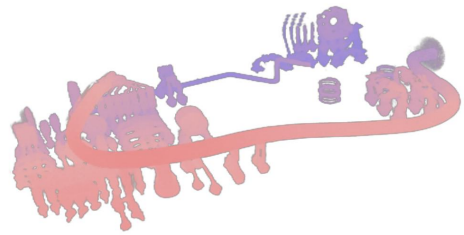
(c) Volume with cool to warm shading



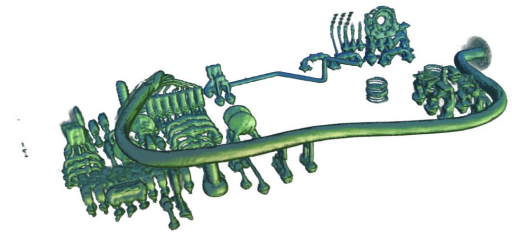
(d) Silhouette contribution



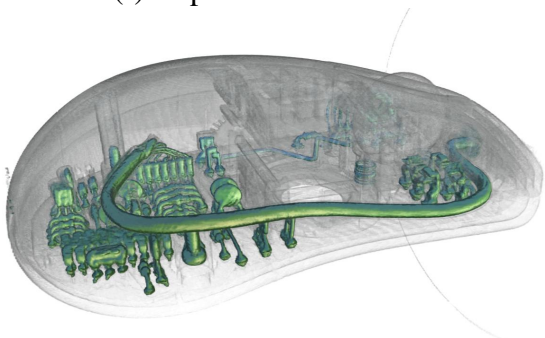
(e) Volume with silhouette



(f) Depth color cue contribution



(g) Volume with depth color cue



(h) Final volume visualization

Figure 15: A complete example of combined techniques using the mouse data set.

4 Conclusion

In recent years, non-photorealistic rendering has received increasing amounts of attention for both art and scientific applications. Non-photorealistic rendering combined with interactivity, can enable spatial exploration by the user so they can gain further insights into the an objects structure by viewing it multiple angles. Exploration of rendering parameter space, on the other hand, allows for the control of how non-photorealistic rendering techniques are applied and can be used to produce illustrations more appropriate for the users task. Consequently, we foresee increasing use of non-photorealistic rendering in scientific data visualization. With the continued advancement of graphics hardware capabilities, there will be ample room for the development of new interactive non-photorealistic rendering techniques that utilize even more rendering styles.

References

[BS00] J. W. Buchanan and M. C. Sousa. The edge buffer: A data structure for easy silhouette rendering. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, June 2000.

[CRL01] D. Cornish, A. Rowan, and D. Luebke. View-dependent particles for interactive non-photorealistic rendering. In *Graphics Interface 2001*, pages 151–158, June 2001.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, pages 9–16, 2001.

[Elb99] G Elber. Interactive line art rendering of freeform surfaces. In *Computer Graphics Forum*, pages 1–12, September 1999.

[ER00] D. Ebert and P Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000 Conference*, pages 195–202, October 2000.

[FM01] B. Freudenberg and T. Masuch, M.and Strothotte. Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine. In *Eurographics 2001*, 2001.

[FvFH96] D. J. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 1996.

[GGSC98] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98 Conference Proceedings*, pages 447–452, July 1998.

[GSG+99] B. Gooch, P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *1999 Symposium on interactive 3D graphics*, April 26-28 1999.

[HMBG00] H. Hauser, L. Mroz, G.-I. Bischi, and M.E. Groller. Two-level volume rendering- fusing mip and dvr. In *IEEE Visualization 2000 Conference Proceedings*, pages 211–218, 2000.

[HZ00] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *SIGGRAPH 2000 Conference Proceedings*, pages 517–526, August 2000.

[IFP95] Victoria Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Proceedings of IEEE Visualization '95 Conference*, pages 52–59, October 1995.

[KGC00] M. Kaplan, B. Gooch, and E. Cohen. Interactive artistic rendering. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, June 2000.

[KKH01] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of IEEE Visualization 2001 Conference*, October 2001.

[KLK+00] A. W. Klein, W. W. Li, M. M. Kazhdan, W. T. Correa, A. Finkelstein, and T. A. Funkhouser. Non-photorealistic virtual environments. In *SIGGRAPH '00 Conference Proceedings*, 2000.

[KMN+99] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. Art-based rending of fur, grass and trees. In *SIGGRAPH '99 Conference Proceedings*, pages 433–438, August 1999.

[Kun99] J. Kunz. *Watercolor Basics: Color*. North Light Books, 1999.

[Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[LM01] E. B. Lum and K.-L. Ma. Non-photorealistic rendering using watercolor inspired textures and illumination. In *Pacific Graphics 2001 Conference Proceedings*, pages 322–330, October 16-18 2001.

[LM02] E. B. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002)*, June 3-5 2002.

[LMC02] E. Lum, K.-L. Ma, and J. Clyne. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *(to appear)IEEE Transactions on Visualization and Computer Graphics*, 2002.

[LMHB00] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rending techniques for scalable real-time 3d animation. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, June 2000.

[Mac99] G. Mackenzie. *The Watercolorist's Essential Notebook*. North Light Books, 1999.

[Mei96] B. J. Meier. Painterly rendering for animation. In *SIGGRAPH '96 Conference Proceedings*, pages 477–484, August 1996.

[MG01] A. Mohr and M. Gleicher. Non-invasive, interactive, stylized rendering. In *2001 ACM Symposium on Interactive 3D Graphics*, 2001.

[MG02] A. Majumder and M. Gopi. Hardware accelerated real time charcoal rendering. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002)*, June 2002.

[MHS99] M. Meissner, U. Hoffmann, and W. Strasser. Enabling classication and shading for 3d texture mapping based volume rending using opengl and extensions. In *IEEE Visualization '99 Conference Proceedings*, 1999.

[MKT+97] L. Markosian, M. A. Kowalski, S. J. Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. Real-time nonphotorealistic rendering. In *SIGGRAPH '97 Conference Proceedings*, pages 415–420, August 1997.

[MPKH94] K.-L. Ma, J. Painter, M. Krogh, and C. Hansen. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics & Applications*, 14(4):59–68, July 1994.

[NM00] J.D. Northrup and L. Markosian. Artistic silhouettes: A hybrid approach. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, June 2000.

[PHWF01] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *SIGGRAPH '01 Conference Proceedings*, August 2001.

[Ras01] R. Raskar. Hardware support for non-photorealistic rendering. In *Siggraph Graphics Hardware '01*, August 2001.

[RC99] R. Raskar and M. Cohen. Image precision silhouette edges. In *1999 Symposium on interactive 3D graphics*, April 1999.

[SABS94] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin. Interactive pen-and-ink illustration. In *SIGGRAPH '94 Conference Proceedings*, pages 101–108, July 1994.

[SMGG01] Peter-Pike Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: A model for capturing npr shading from art. In *Graphics Interface 2001*, pages 143–150, June 2001.

[TC00] S. M. F. Treavett and M Chen. Pen-and-ink rendering in volume visualisation. In *Proceedings of IEEE Visualization 2000 Conference*, pages 203–209, October 2000.

[TCSJ01] S. M. F. Treavett, M. Chen, R. Satherley, and M. W. Jones. Volumes of expression: Artistic modelling and rendering of volume datasets. In *Proceedings of Computer Graphics International 2001*, pages 99–106, July 2001.

[VGH96] A. Van Gelder and U. Hoffman. Direct volume rendering with shading via three-dimension textures. In *ACM Symposium on Volume Visualizatrion '96 Conference Proceedings*, 1996.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *The OpenGL Programming Guide, Third Edition*. Wesley, 1999.

[WPFH02] M. Webb, E. Praun, A. Finkelstein, and H. Hoppe. Fine tone control in hardware hatching. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002)*, June 2002.

Final Remarks

This course provided a concise introduction to non-photorealistic rendering: generation of artistic imagery and perceptually effective scientific visualization. We began with a survey of both historical and contemporary examples of the use of NPR, followed by an explanation of fundamental algorithms. We have also presented a few approaches at the forefront of NPR research including example-based rendering methods for automatic generation of artistic imagery and hardware-accelerated parallel techniques for achieving interactive high-resolution rendering.

Although we could not cover all the topics in detail with 3.5 hours, we believe we have addressed the most relevant issues and trends of applying non-photorealistic rendering to scientific applications and art, and provided pointers to future research. In Friday's Painting and Non-Photorealistic Graphics session you will also learn about some exciting new research results.

A collection of sixteen papers is attached to the end of the course notes which sample the state-of-the-art NPR techniques relevant to the individual lecture topics presented here. In addition, the comprehensive bibliographies provided in each chapter should be a good starting point for searching further information about recent development of related technologies. For those who are just entering this exciting research area, you may want to check out Gooch & Gooch's book "Non-Photorealistic Rendering" and the two NPAR proceedings first.

Thank you for your attendance and enjoy the rest of the conference. Please feel free to contact me for further information.

Kwan-Liu Ma
`ma@cs.ucdavis.edu`

Appendix - Collection of Papers

1. "*Texture Synthesis for 3D Shape Representation*", G. Gorla, V. Interrante, and G. Sapiro, (to appear) IEEE Transactions on Visualization and Computer Graphics, 2002.

2. "*Line Direction Matters: An Argument for the Use of Principal Direction in 3D Line Drawing*", A. Girshick, V. Interrante, S. Haker, and T. Lemoine, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France, June 5-7, 2000, pages 43-52.

3. "*Understanding Errors in Approximating Principal Direction Vectors*", J. Goldfcather and V. Interrante, (to appear) ACM Transactions on Graphics, 2003.

4. "*Paint By Numbers: Abstract Image Representations*", P. Haeberli, in SIGGRAPH '90 Conference Proceedings, August 1990, pages 207-214.

5. "*Image Analogies*", A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, in SIGGRAPH 2001 Conference Proceedings, August 12-17, 2001, pages 327-340.

6. "*Illustrating Smooth Surfaces*", A. Hertzmann and D. Zorin, in SIGGRAPH 2000 Conference Proceedings, July 23-28, 2000, pages 517-526.

7. "*Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*", K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, in SIGGRAPH '99 Conference Proceedings, August 8-13, 1999, pages 277-286.

8. "*Harnessing Natural Textures for Multivariate Visualization*", V. Interrante, Visualization Viewpoints, Editors: Theresa-Marie Rhyne and Lloyd Treinish, IEEE Computer Graphics and Applications, November/December 2000, pages 6-11.

9. "*Creating Evenly-Spaced Streamlines of Arbitrary Density*", B. Jobard and W. Lefer, in Proceedings of the 8th Eurographics Workshop on Visualization in Scientific Computing, 1997, pages 45-55.

10. "*Interactive Artistic Rendering*", M. Kaplan, B. Gooch, and E. Cohen, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France, June 5-7, 2002. pages 67-74.

11. "*Art-Based Rendering of Fur, Grass, and Trees*", M. A. Kowalski, L. Markosian, J. D. Northrup, and L. Bourdev, R. Barzel, L. S. Holden, J. F. Hughes, in SIGGRAPH '99 Conference Proceedings, Augst 8-13, 1999, pages 433-438.

12. "*Artistic Silhouettes: A Hybrid Approach*", J. D. Northrup and L. Markosian, in Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000), Annecy, France, June 5-7, 2000, pages 31-37.

13. "*Interactive Pen-and-Ink Illustration*", M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin, in Proceedings of SIGGRAPH '94, July 24-29, 1994, pages 101-108.

14. "*Orientable Textures for Image-Based Pen-and-Ink Illustration*", M. P. Salisbury, M. T. Wong, J. F. Hushes, and D. H. Salesin, in SIGGRAPH '97 Conference Proceedings, August 1997, pages 401-406.

15. "*Weighted Voronoi Stippling*", A. Secord, in Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002), Annecy, France, June 3-5, 2002.

16. "*Fine Tone Control in Hardware Hatching*", M. Webb, E. Praum, A. Finkelstein, H. Hoppe, in Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002), Annecy, France, June 3-5, 2002.

17. "*Computer-Generated Pen-and-Ink Illustration*", G. Winkenbach and D. H. Salesin, in Proceedings of SIGGRAPH '94, July 24-29, 1994, pages 91-100.

Line Direction Matters: An Argument For The Use Of Principal Directions In 3D Line Drawings

Ahna Girshick *Victoria Interrante** *Steven Haker*** *Todd Lemoine****

Nissan Cambridge Basic Research *University of Minnesota **Yale University ***LambSoft

ABSTRACT

While many factors contribute to shape perception, psychological research indicates that the direction of lines on the surface may have an important influence. This is especially the case when other techniques (shading, silhouetting) do not present sufficient shape information. The psychology literature suggests that lines in the principal directions of curvature may communicate surface shape better than lines in other directions. Moreover, principal directions have the quality of geometric invariance so line directions are based on the surface geometry and are viewpoint and light source independent, and the lines do not move above over the surface during animation unless desired. In this work we describe principal direction line drawings which show the flow of curvature over the surface. The technique is presented for arbitrary surfaces represented by either 3D volume data or a polygonal surface mesh. The latter format is common in the field of computer graphics yet thus far has not been widely used for principal direction estimation. The methods offered in this paper can be used alone or in conjunction with other NPR techniques to improve artistic 3D renderings of arbitrary surfaces.

Keywords: non-photorealistic rendering, principal direction line drawings, line direction, line drawings, geometrically invariant line drawings.

1 INTRODUCTION

Amongst the varied goals of artistic Non-Photorealistic Rendering (NPR) is the pursuit of *perceptually efficient* images. A perceptually efficient visual representation emphasizes important features and minimizes extraneous detail and is essential for making comprehensible artistic images. Computer-generated line drawings are a particularly effective form of NPR since lines' features (length, width, intensity, density, quality, direction, etc.) can be combined to create shaded, textured, and expressive images which capture the essence of the form of an object. In the field of computer-generated line drawing, 3D representations of curved surfaces generally focus on the silhouette edges, disregarding large amounts of interior curvature information. These depictions often rely on either previous knowledge of the surface or the use of motion (movement of the surface, viewpoint, or light source). In this work we explore a 3D line drawing technique which is independent of the surface's orientation, the viewpoint, or the light source. In particular, we examine line direction and use this paper to raise the question: Does line direction matter?

We argue that line direction does matter, and suggest the use of the principal directions of curvature for directing lines to improve the depiction of surface shape in artistic line drawings. The advantages of principal directions (see Appendix A for a mathematical definition) are that they are geometrically-invariant, highlight the most direct path on a surface between two points, indicate the directions of the curvature extrema at any point, and have been suggested by psychologists as the preferred interpretation for making surface shape judgments.

The importance of geometric invariance should not be underestimated. Geometrically-invariant cues are based on properties of the surface geometry and are by definition viewpoint and light source independent. While shading and silhouetting provide substantial shape information, valuable curvature information can be lost in shadows or the interior of the surface. Furthermore, viewpoint dependent lines may move around in a distracting manner during motion or animation. Geometric invariance does not imply that lines must be rigidly "pasted" onto the surface during animation. If line movement is desired, the geometrically-invariant vector field can help guide more fluid movement over the surface. Combining geometrically-invariant cues with shading or silhouetting can be especially powerful. Geometrically-invariant line attributes such as color and density can be manipulated with respect to viewpoint or light source [7].

Despite the promise for principal directions, their full potential in NPR has yet to be realized. The reasons perhaps may be related to the difficulties in estimating an accurate, smoothly continuous vector field of principal directions. The problem is most challenging for polygonal surface meshes, a particularly common data format for arbitrary 3D surfaces. Additionally, principal direction line drawings must address the complex issues of creating uniformly distributed, non-intersecting, long smooth lines which gracefully traverse umbilics, planar regions, and transitions of directional dominance. Here we examine both 3D

Nissan Cambridge Basic Research, 4 Cambridge Center, Cambridge, MA 02139, girshick@cbr.com

*University of Minnesota, Computer Science Department, Minneapolis, MN 55455, interran@cs.umn.edu

**Yale University, Department of Computer Science, New Haven, CT 06520, haker@cs.yale.edu

***LambSoft, Minneapolis, MN 55405, tlemoine@lamb.com

volume datasets and polygonal surface meshes, and suggest some techniques for line tracing.

The main contribution of this work is to show that for a 3D line drawing, line direction can matter and principal direction line drawings can be used to better convey surface shape. In the next section we motivate the importance of line direction with psychological evidence. We follow with related work in computer-generated 3D line drawing. In section four, we provide a brief overview of principal direction estimation techniques. Section five shows the effects of line direction and section six presents techniques for principal direction line drawings. In the final section we draw some conclusions and discuss areas of future work.

2 PSYCHOLOGICAL EVIDENCE FOR THE IMPORTANCE OF LINE DIRECTION

The psychology literature gives us a sense of how the human visual system perceives images and is an essential reference for making perceptually efficient renderings. Early research asserted that humans can use surface markings, or texture, to perceive surface orientation. Gibson [8] was amongst the first to emphasize the significance of texture cues for shape and depth perception. He was able to show convincingly that observers could reliably interpret the slant of the planar surface by the cues provided by the projection distortion of the texture patterns.

Of relevance to this work is the open question of whether anisotropic (directed) textures are as suitable for conveying shape information as isotropic (undirected) textures. Interrante [12] was unable to show an effect of texture type in shape perception under conditions of stereo and motion for various plausible isotropic and anisotropic textures for transparent surfaces, including grids and principal direction textures. Yet Cumming et al. [3] found an indicative effect of texture type for stereoscopic shape perception between a plausible and unlikely texture. While shape-from-texture research often makes assumptions of isotropy or homogeneity, Knill [16] hypothesized that there are different modes to visually perceive isotropic and anisotropic textures.

While the question of effects of isotropic versus anisotropic texture still remains open, it is evident that when anisotropic surface markings are dependent on surface geometry, surface depth and orientation perception is improved. Knill [16] found that in an anisotropic texture processing mode, the curvature of geodesic surface markings determines perception of local surface orientation. The experiments of Johnston et al. [14] showed that stereoscopic depth perception of curved surfaces with texture which provided a good indication of surface geometry was superior to random dot textures. Stevens [24] was among the first to suggest that humans can make surface shape judgments by assuming that surface contours (lines on the surface) are aligned with the principal directions of curvature. In later work Stevens and Brookes [23] demonstrated that principal direction surface contours are also good indications of relative surface slant. More recently, Mamassian and Landy [17] found that surface shape judgments are biased by the assumption that surface contours are aligned with the principal directions. From the above literature, it is reasonable to believe that surface shape and depth perception may be generally aided by textures, and also by anisotropic textures based on surface geometry, particularly lines aligned with the principal directions

3 RELATED WORK

Computer-generated 3D line drawings borrow from centuries of artists' techniques and have recently received significant attention in the NPR community. Winkenbach and Salesin used stroke textures to create depth and shape in line drawings of parametric surfaces [26]. Markosian et al. emphasized the silhouette edges for viewpoint-dependent images of arbitrary 3D surfaces [18]. Curtis used 3D models to generate loose and artistic sketches and animations [4]. Elber rendered geometrically-invariant line drawings and textures of parametric and implicit surfaces [6].

Principal directions have been suggested [11,26] and approached [2,6] in line drawings. In [26,6], lines were traced along the parametric lines of parametric surfaces, which sometimes coincided with the principal directions. Saito and Takahashi [20] rendered line drawings lines of parametric surfaces along geodesic lines. Interrante et al. [11] used 3D principal direction textures to illustrate surface shape in volume data. However, none of these works addressed the challenge of estimating the principal directions from arbitrary surfaces (particularly polygonal surface mesh formats) nor that of tracing long strokes in one direction (rather than cross-hatching) through umbilics, planar regions, and areas of changing directional dominance. This work is based upon a preliminary sketch by Girshick and Interrante [9].

4 PRINCIPAL DIRECTION ESTIMATION

For data of any format, the first step towards a principal direction line drawing is to estimate the principal direction vector field, comprised of the principal directions at a set of points on the surface. There are a variety of methods for estimating principal directions, each with its various strengths and weaknesses, however a full discussion of the computational details is not in the scope of this paper. Do Carmo outlines analytic calculations of principal directions for parametric surfaces in [5]. For iso-intensity surfaces in 3D volume data, Monga et al. used the Hessian of the 3D data to compute the principal directions [19]. Interrante et al. used a similar technique based on Gaussian-

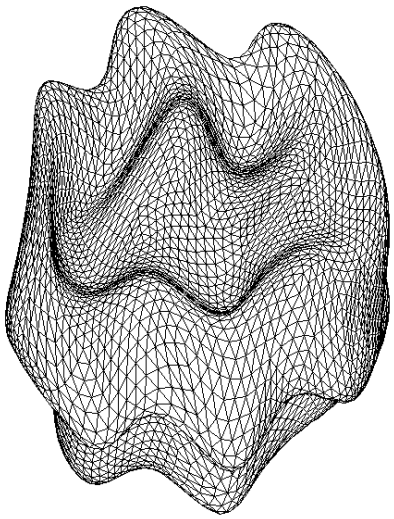


Figure 1 Polygonal surface mesh of arbitrary 3D "blob".

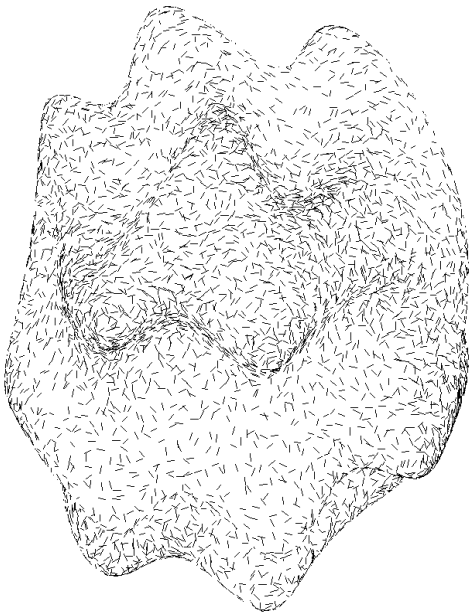


Figure 2 Random vector field of object in figure 1.

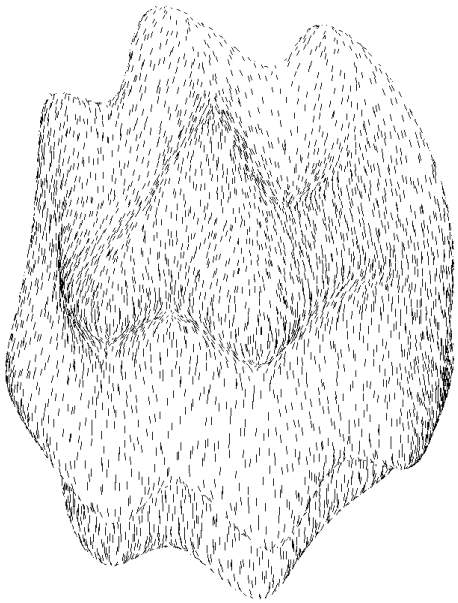


Figure 3 Uniform (vertical) vector field of object in figure 1.

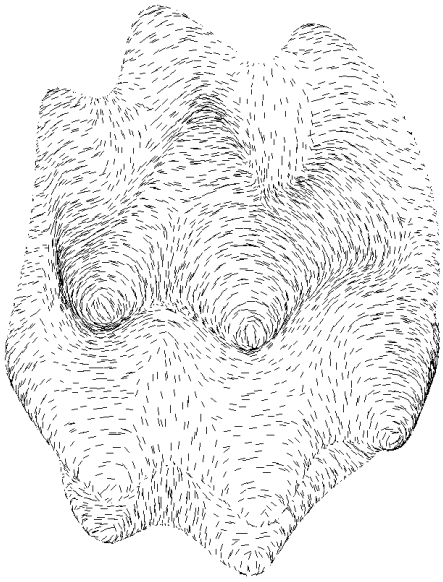


Figure 4 First principal direction vector field of object in figure 1.

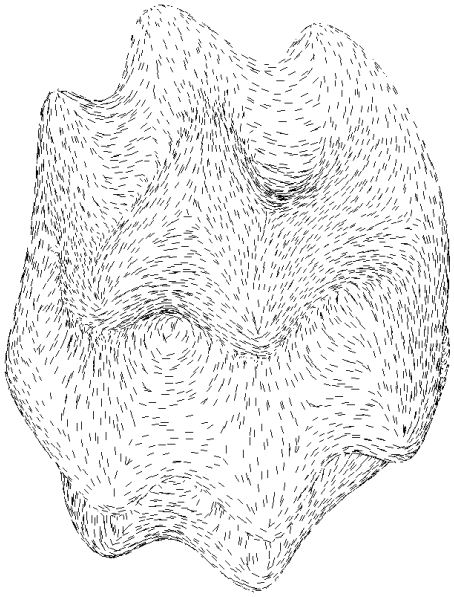
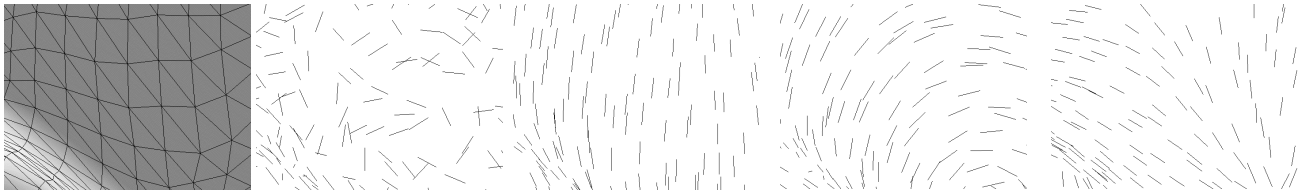


Figure 5 Second principal direction vector field of object in figure 1.



6a Shaded surface mesh **6b** Random vector field **6c** Uniform vector field **6d** First principal direction vector field **6e** Second principle direction vector field

Figure 6 Close-ups of the same region of the object in figure 1.

weighted finite-differencing [12]. We used this approach for the volume datasets in this paper.

As of yet there is no reliable standard technique for locally estimating principal directions from a polygonal surface mesh. Samson and Mallet [21] fit cubic patches to the local neighborhood around a vertex, using the vertex's normal and neighboring normals, and then compute the partial derivatives to obtain principal directions. Hamann [10] employs a similar approach except uses quadratic patches and relies solely on deviation from a vertex's tangent plane without using neighboring vertex normals. Joshi et al. provide good examples of this approach in [15]. Chen and Schmitt [1] and Taubin [25] avoid explicitly describing surface patches but instead construct a quadratic form at each vertex. In [25] the quadratic form represents an orthonormal basis whose eigenvectors are the principal directions. The principal curvatures are the directional curvatures in the principal directions. For the polygonal surface meshes in this work, we use variations of both Hamann's and Taubin's methods, with similar results. The accuracy of both is highly dependent on the sample distribution of the local surface geometry and is an area of current work.

5 EFFECTS OF LINE DIRECTION

The significance of line direction for a line drawing is perhaps best illustrated visually with the underlying vector field. As will be explained in the next section, a line drawing can be rendered by tracing strokes which follow the flow of a vector field [22]. Figures 1–5 show various vector fields on the same arbitrary "blob" dataset, shown as a polygonal surface mesh in figure 1. A 3D volume dataset would produce similar results. The vector field is illustrated by projecting the field direction at each vertex of the underlying mesh onto the tangent plane at that point. The random vector field in figure 2 and the uniform vector field in figure 3 convey surface shape only through texture compression, which provides hints of the silhouette edges, but not through the use of line direction. When the silhouette edges are not visible, as in the close-ups in figures 6b and 6c, the surface shape is largely ambiguous.

Figures 4 and 5 show first and second principal direction vector field respectively. Compared to figures 2 and 3, these vector fields appear to better convey local surface orientation, including ridges and valleys, subtle surface undulations, changes in curvature, and interior silhouette edges. Figure 6 shows the close-ups of the vector fields in the absence of silhouette edges. When comparing the four close-ups in figures 6b through 6e,

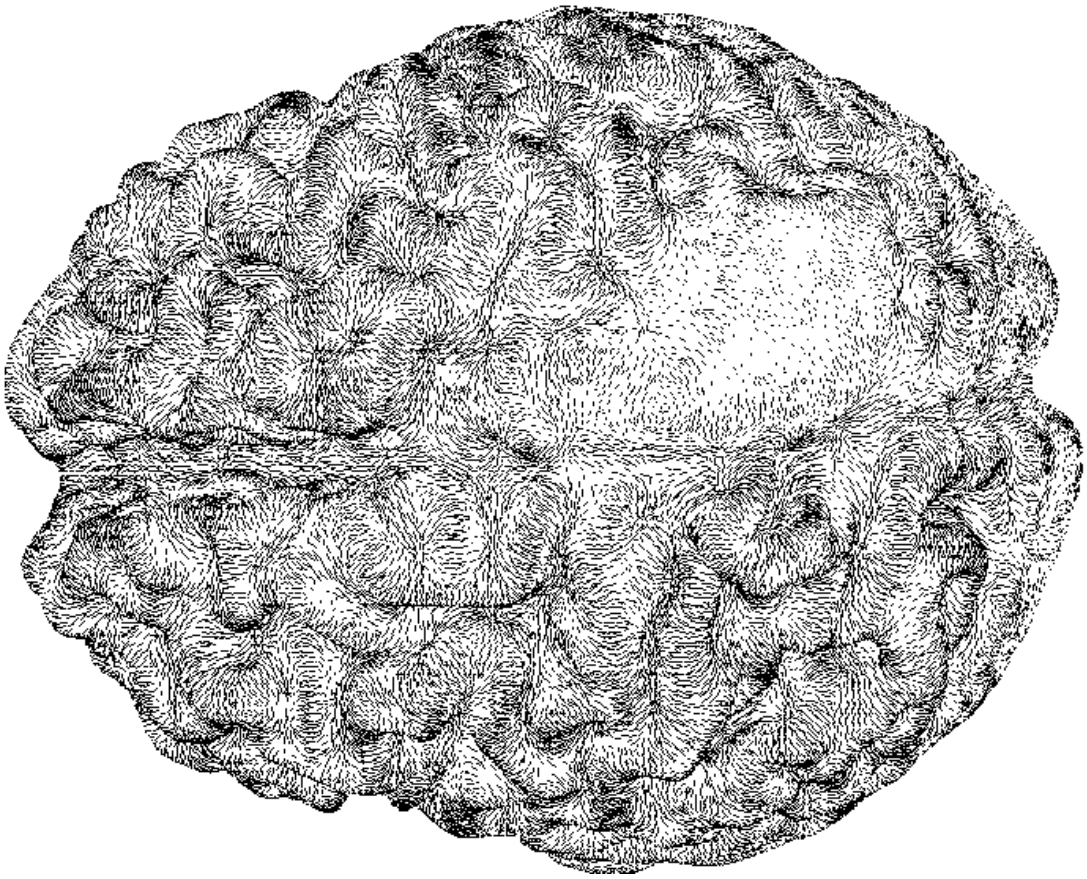


Figure 7 First principal direction vector field of a brain represented by a polygonal surface mesh. Data source: Ron Kikinis, Harvard Medical School.

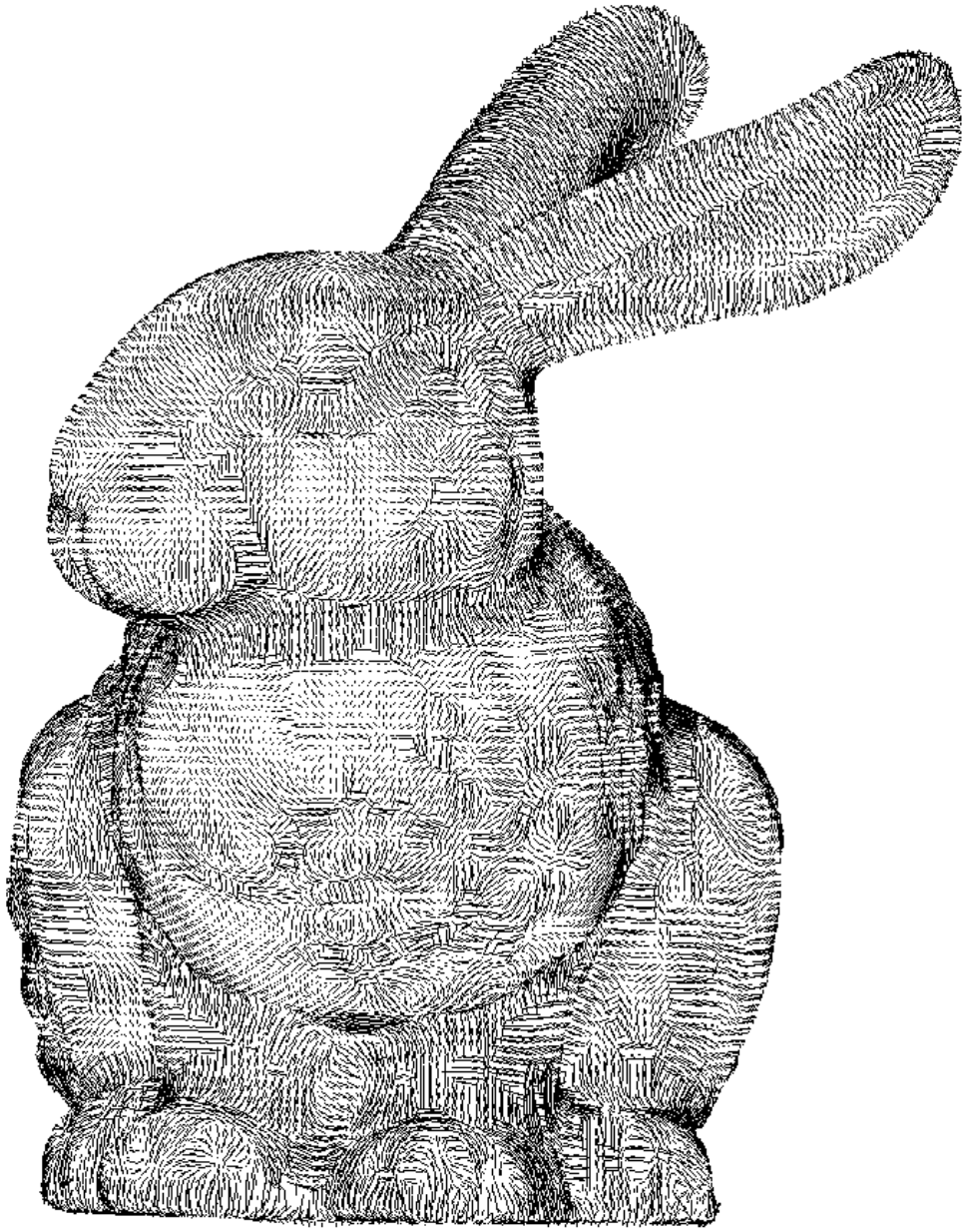


Figure 8 First principal direction vector field of a bunny represented by a polygonal surface mesh. Data source: Stanford University Computer Graphics Lab.

it seems to be easier to judge the surface shape from principal direction vector fields than the random and uniform vector fields. Figures 7 and 8 provide more examples of first principal direction vector fields on more complex surfaces. One can predict the difficulty in perceiving the surface shape if these figure used random or uniform vector fields.

6 PRINCIPAL DIRECTION LINE DRAWINGS

Principal direction line drawings illustrate the flow through the principal direction vector fields described in the previous section. In this section we describe the details for both 3D volume data and polygonal surface meshes. For 3D volume data, the vector field is a 3D volume and the strokes are traced through the volume. For polygonal surface data, the vector field lies on the explicitly defined surface mesh and the strokes must be drawn on the surface.

6.1 Principal Direction Line Drawings of 3D Volume Datasets

Figures 9 and 10 show different styles of principal direction line drawings of the same human pelvis CT volume dataset. Both figures underwent the same preprocessing stage. Initially a first principal direction volume vector field is generated using the technique described in section four. Then a sparse set of strokes

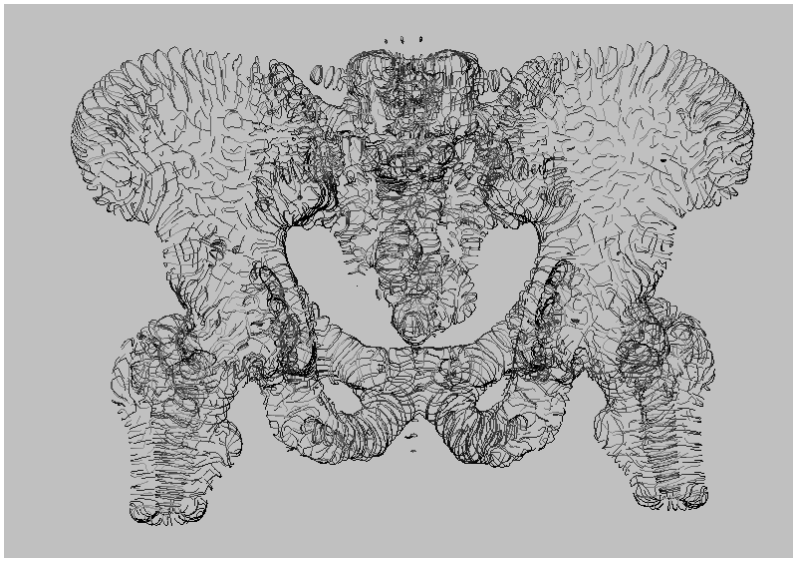


Figure 9 Principal direction line drawing (with shading and without hidden line removal) of a bone/soft tissue boundary iso-intensity surface in a CT 3D volume dataset of a human pelvis.

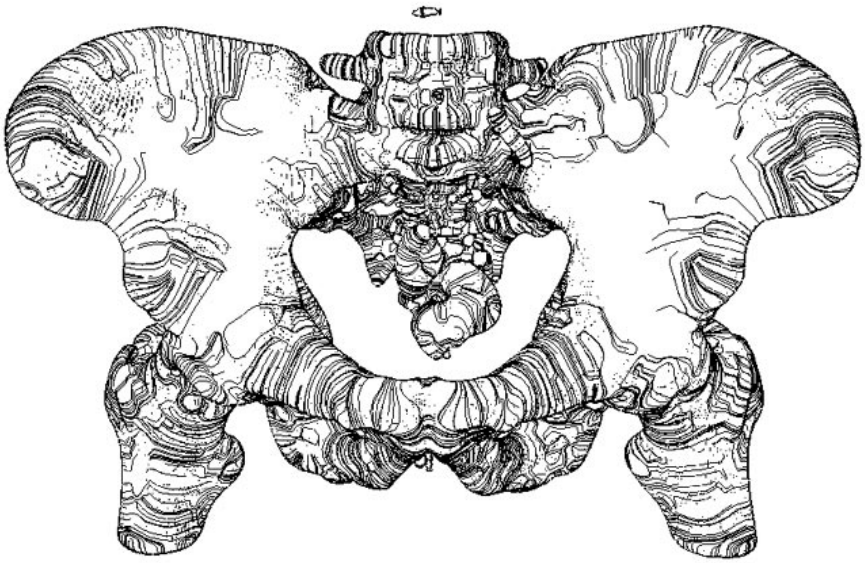


Figure 10 Principal direction line drawing with silhouette edges and hidden line removal of the volume dataset in figure 8.

is traced through the vector field, each stroke originating from a point near the surface which is not too close to neighboring starting points, such that the set has the approximate distribution of a Poisson disk.

In figure 9, the strokes represent individual streamlines [22] through the vector field. The lines are shaded according to the surface normal direction indicated by the gray level gradient in the volume data, but hidden line removal has not been done. The result is especially powerful during animation, when the geometrically-invariant lines "stick" to the surface.

In figure 10 we attempted to create a freer sketch of the volume data set, using hidden line removal, including silhouettes and selecting only a subset of possible strokes. Because it is viewpoint-dependent, by definition it is not geometrically-invariant. However the lines are still directed in the principal directions and defined based on the geometry of the surface, so the static 2D image should provide the same visual cues to the surface shape, at least near the silhouette edges. The subset of lines to render was selected with a preference towards placing lines in areas of higher curvature lines and near silhouette edges. Line length is proportional to the magnitude of the first principal curvature at the start point.

6.2 Principal Direction Line Drawings of Polygonal Surface Meshes

The main steps in creating a principal direction line drawing from a polygonal surface mesh are estimating a smoothly continuous principal direction vector field and tracing evenly spaced strokes which follow the flow of the vector field. The steps are described separately below, but for efficiency they can be done simultaneously.

6.2.1 Creating a continuous principal direction vector field

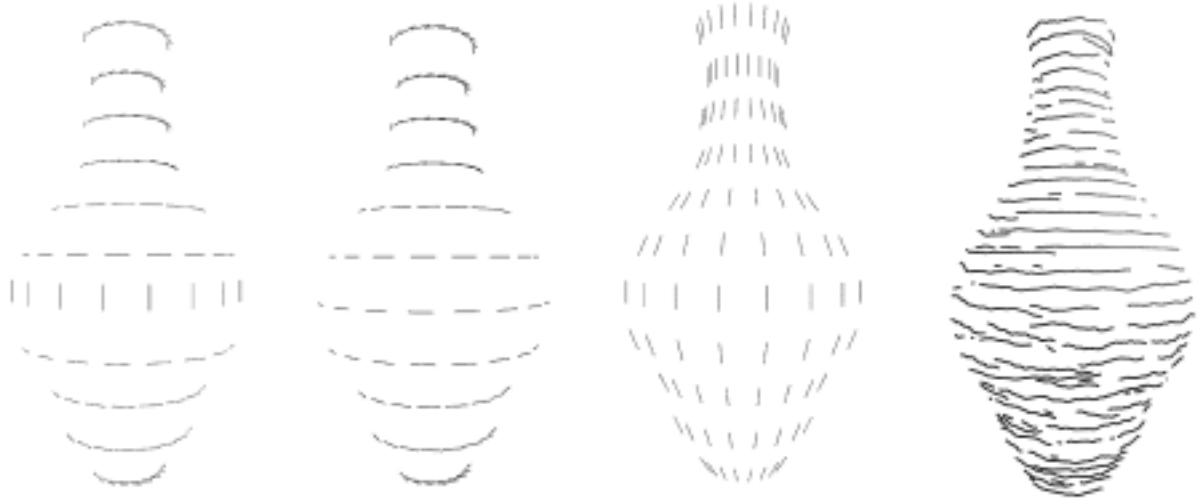
At any point on a 3D surface, each of the orthogonal first and second principal directions have a positive and negative direction. Thus there are four possible directions for the vector field at each point. Ideally we would always choose the first principal direction (either positive or negative). However, in regions close to umbilics and planes, where curvature is almost similar in all directions, the first and second principal directions may suddenly switch places causing a flip of up to 90 degrees, resulting in a sudden disruption of flow. Figure 11a demonstrates this for a simple vase mesh. The first principal direction field is continuous except around the girth of the vase where it is *almost* spherical and the curvature is slightly greater vertically than horizontally. In this case, a *continuous* principal direction line drawing minimizes distracting details and is more aesthetically pleasing than a *first* principal direction line drawing. The continuous vector field is created by first choosing an arbitrary reference vector. In the example of figure 11, the choice of reference vector can lead to only two possible outcomes, but in a more complex dataset it might be advantageous to choose a meaningful starting reference vector. Next, for each vertex, the direction which is closest to the reference vector is chosen. The reference vector is updated to reflect the choice. Figures 11b and 11c show the two possible continuous principal direction vector fields for this dataset. The principal direction line drawing corresponding to 11b is shown in 11d. This approach for creating continuous vector field works well for surface regions with well-defined principal directions. However, at true umbilics, where normal curvature is the same in all directions, and on planes, where normal curvature is zero in all directions, the principal directions are undefined. For these regions, we interpolate between neighboring well-defined regions of the vector field. Even still, for a complex surface such shown in figures 1 and 8, regions may occur where it is necessary to make an abrupt switch in line direction. A possible technique for gracefully transitioning between line directions is to minimally employ cross-hatching using both the first and second principal direction fields combined. However we do not advocate the general use of crosshairs such as in figure 12, as the inelegant crosses can become distracting and muddle the flow of curvature.

6.2.2 Tracing strokes through the vector field on a polygonal surface

The objective of this step is to obtain an approximately uniformly-distributed set of non-intersecting long curved lines, which lie on the surface. The streamline tracing technique of Jobard and Lefer [13] is extended from 2D images to 3D surfaces to generate evenly-spaced non-intersecting lines. The curvature of each line is achieved by continually redirecting it as it traverses the changing vector field.

Each stroke is composed of a set of control points. The criterion for each valid control point is that it lies at a minimum distance threshold from all existing strokes. The first stroke starting point is random, and the remaining stroke starting points are chosen to be as close as possible to existing points without breaking the minimum distance threshold.

The direction of the stroke is updated at frequent distance intervals as well as when a stroke crosses a polygon boundary. The stroke's direction at any given point on a polygon is



11a First principal direction vector field.

11b Continuous vector field of greatest overall curvature.

11c Continuous vector field of less overall curvature.

11d Continuous principal direction line drawing for 11b. Shading and slight randomness added to strokes for artistic effect.

Figure 11 Various principal direction vector fields and principal direction line drawing of a simple vase.

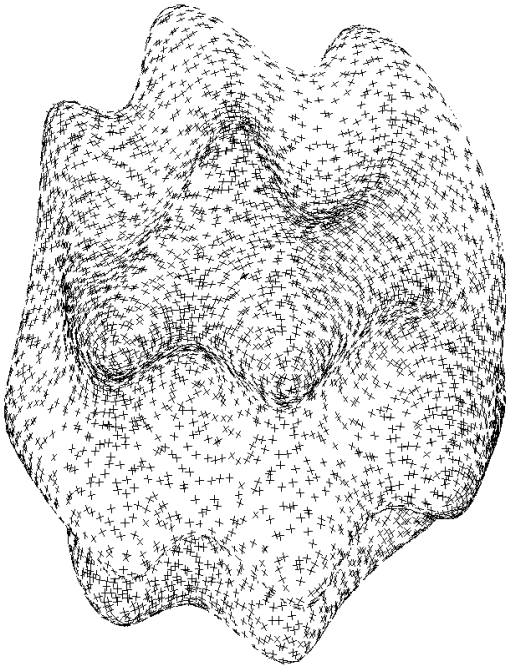


Figure 12 First and second principal direction vector field of the object in figure 1.

determined by trilinearly interpolating the principal directions of the polygon's vertices. Strokes are terminated if they approach the minimum distance threshold. This process is shown in figure 13. To avoid the cost of calculating an implicit surface, each segment of a stroke is projected onto the polygonal surface mesh. Provided a sufficiently fine mesh, this approximation is worth the savings in computation.

Regions of opposing force occur when neighboring principal directions point in opposing directions. These vector field discontinuities crop up near umbilics and planar. The current approach is to terminate strokes when this happens.

The result of this technique is shown in figure 14, with some randomness added for wiggly lines. A more artistic image might be achieved by varying the line density according to the light source, and adding silhouette lines.

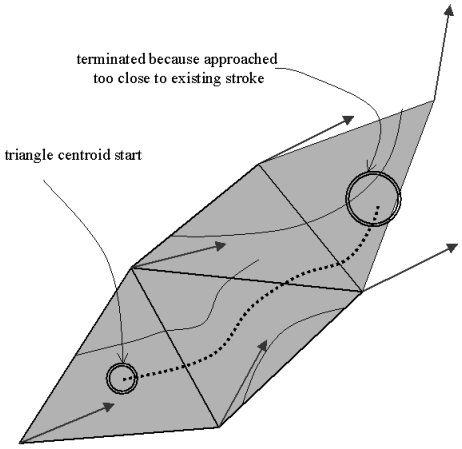


Figure 13 Stroke tracing through a principal direction vector field on a polygonal surface mesh. (image for illustrative purposes only).

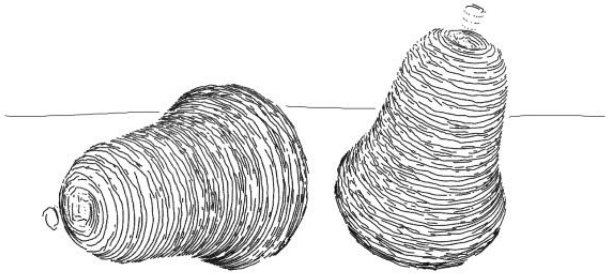


Figure 14 Principal direction line drawing of pears, represented by triangular surface meshes. Hidden line removal was used, and slight random noise was added to the stroke tracing process.

6.2.3 Rendering

The rendering of the line drawing is straightforward. A stroke is a set of control points which can be rendered as either a simple polyline or spline. Our approximations were fine enough to use anti-aliased polylines with no perceivable difference over splines.

7 CONCLUSIONS AND FUTURE WORK

The most troublesome areas in obtaining a continuous principal direction line drawing are those where the principal directions are undefined and in regions of opposing force. In the first case, the current interpolation technique works well if the unknown regions are small and bordered by more well-defined principal directions, but fails for larger areas and is a topic of future work. In the latter case, we would like to eventually gracefully merge strokes from neighboring regions of opposing principal directions, possibly with subtle cross-hatching, instead of terminating them.

This work outlined the approach for both 3D volume datasets and polygonal surface. Principal direction line drawings for parametric surfaces can follow a similar approach. For polygonal surface meshes, we found the existing principal direction estimation techniques to be insufficiently accurate for asymmetric local mesh geometries. We are currently working on their improvement which is of great relevance to principal direction line drawings.

Our exploration of principal direction line drawing techniques has indicated to us that simply showing line direction is not necessarily sufficient for an aesthetically pleasing rendering. Adding silhouette lines (as shown in figure 15), shading, density variations, and longer lines all improve the quality of the image, and are areas of future work.

This work poses the important question of whether line direction matters for creating a perceptually efficient line drawing. We have provided compelling psychological evidence and visual examples to believe that line direction affects surface shape perception. In particular, the principal directions of curvature appear to be more effective than non-principal directions at conveying surface shape. Principal direction lines on a surface have the advantage that they show the path of greatest curvature and are geometrically-invariant, so they appear the same from

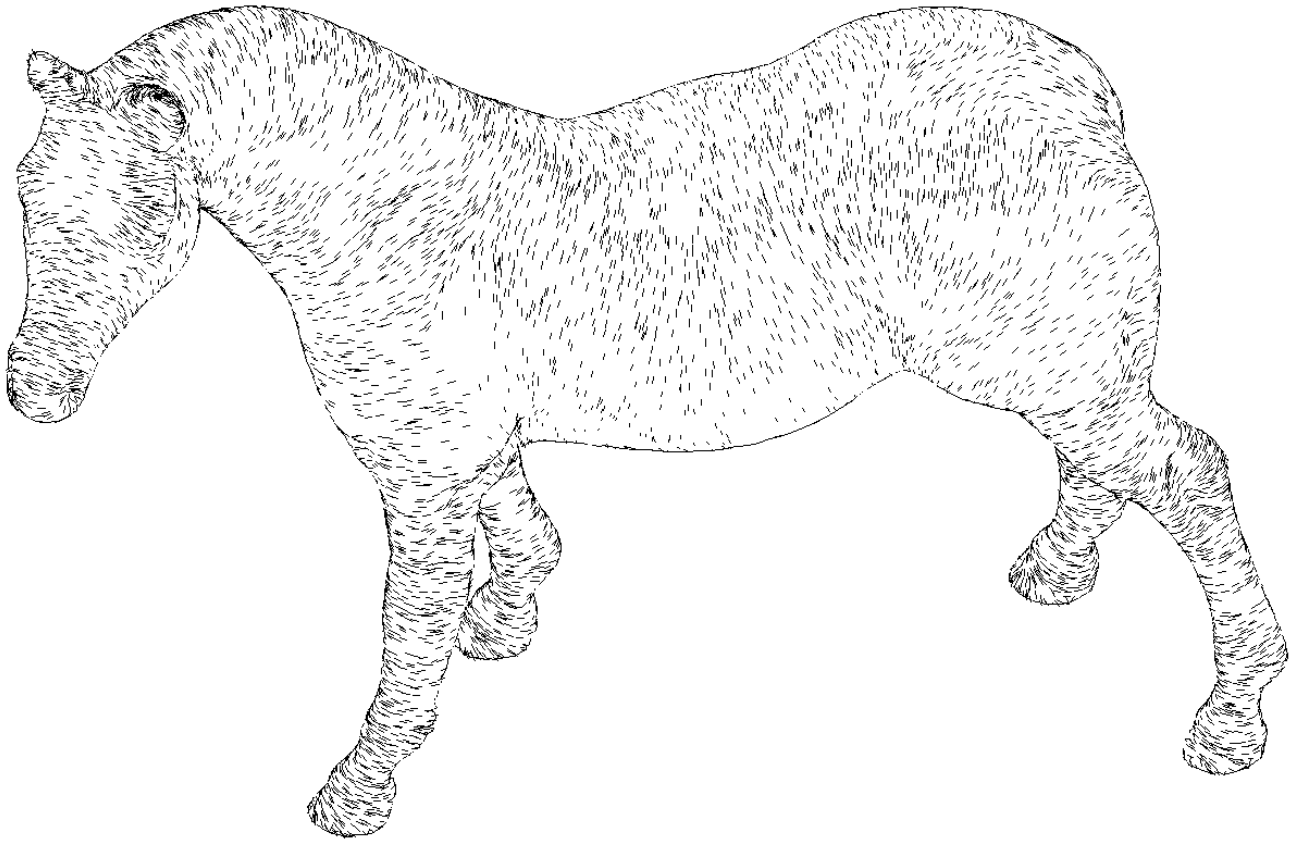


Figure 15 First principal direction vector field and silhouette lines of a horse dataset, courtesy of Cyberware, Inc.

all viewpoints and do not shift during animation. Principal direction line drawings are well-suited for showing the subtle undulations of an arbitrary, smoothly curved surface in 3D, especially when silhouette edges are not visible. They can be used alone or in conjunction with other graphics techniques such as shading and drawing silhouette edges. One intention of this work is to serve as a reminder that perceptually efficient images are an important part of artistic NPR. We also wish to inspire more perceptual studies of the effectiveness of principal direction line drawings.

8 ACKNOWLEDGMENTS

This work was supported by a Grant-in-Aid of Research, Artistry, and Scholarship from the University of Minnesota and NSF Grant CCR-9875368. Special thanks to Detlev Stalling, Mireille Boutin, Ron Kikinis, and Erwin Boer.

9 APPENDIX A: DEFINITION OF PRINCIPAL DIRECTIONS OF CURVATURE

The normal curvature at p in a given direction T will be referred to as the directional curvature $\kappa_p(T)$. The *first principal direction*, T_1, is the direction of the maximum magnitude of normal curvature, called the *first principal curvature* (κ_p^1). The *second principal direction*, T_2, is orthogonal to the first, and is the direction of the other curvature extreme, called the *second principal curvature* (κ_p^2). For elliptic surface patches (with positive Gaussian curvature) the second principal direction is the direction which the surface is most nearly flat. For hyperbolic, (saddle-shaped) patches (with negative Gaussian curvature), the second principal direction is the direction of the lesser of the two extrema. The two principal directions T_1 and T_2 are orthogonal and lie in the tangent plane at the point p, creating an orthonormal basis with the normal vector N at p. Figure 14 shows an example of the orthonormal basis on a hyperbolic surface patch. The product of the two principal curvatures equals the Gaussian curvature, $K=\kappa_p^1 \bullet \kappa_p^2$.

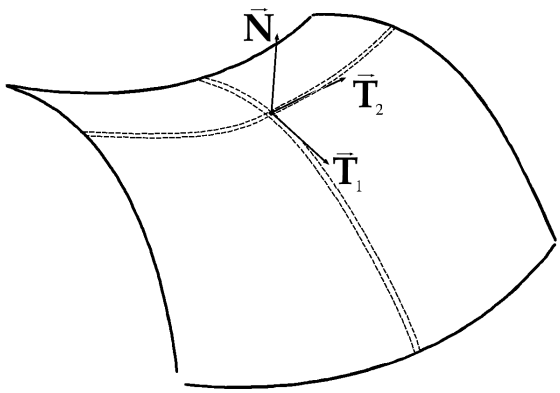


Figure 16 Orthonormal basis formed by normal and two principal directions and curvature strips in the principal directions at a point on a hyperbolic patch.

10 REFERENCES

[1] X.Chen and F.Schmitt. Intrinsic Surface Properties from Surface Triangulation. *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 739-743, 1992.

[2] R.Coutts and D.Greenberg. Rendering with Streamlines. *Visual Proceedings (SIGGRAPH 97)*, p.188, 1997.

[3] B.Cumming, E.Johnston and A.Parker. Effects of Different Texture Cues on Curved Surfaces Viewed Stereoscopically. *Vision Research*, 33(5/6):827-838, 1993.

[4] C. Curtis. Loose and Sketchy Animation. *Visual 999Proceedings (SIGGRAPH 98).*

[5] M.do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, 1976.

[6] G.Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):77-81, January-March 1998.

[7] G.Elber. Interactive Line Art Rendering of Freeform Surfaces. *Proceedings of Eurographics 99*, July 1999.

[8] J.Gibson. *The Perception of the Visual World*, Houghton Mifflin, 1950.

[9] A.Girshick and V.Interrante. Real-time Principal Direction Line Drawings of Arbitrary 3D Surfaces. *Visual Proceedings (SIGGRAPH 99)*, p. 271.

[10] B.Hamann. Curvature Approximation for Triangulated Surfaces. *Geometric Modelling : Dagstuhl*, pp.139-153, Springer-Verlag, 1993.

[11] V.Interrante. Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution. *Computer Graphics (Proceedings of SIGGRAPH 97)*, pp.109-116, 1997.

[12] V.Interrante, H.Fuchs and S.Pizer. Conveying the 3D Shape of Smoothly Curving Transparent Surfaces via Texture. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):98-117, April-June 1997.

[13] B.Jobard and W.Lefer. Creating Evenly-Spaced Streamlines of Arbitrary Density. *Proceedings of 8th Eurographics Workshop on Visualization in Scientific Computing*, pp.45-55, 1997.

[14] E.Johnston, B.Cumming and A.Parker. Integration of Depth Modules: Stereopsis and Texture. *Vision Research*, 33(5/6):813-826, 1993.

[15] S. Joshi, J. Wang, M. Miller, D. Van Essen, U. Grenander. On the Differential Geometry of the Cortical Surface. *Vision Geometry IV (Proc. SPIE's 1995 International Symposium on Optical Science, Engineering, and Instrumentation)*, 2573:304-311, 1995.

[16] D.C.Knill. From Contour to Texture: Static Texture Flow is a Strong Cue to Surface Shape. *Perception* 26 *(Supplement: ECVP '97 Abstracts)*, p.111,1997.

[17] P.Mamassian and M.Landy. Observer Biases in the 3D Interpretation of Line Drawings. *Vision Research*, 38:2817-2832, 1998.

[18] L.Markosian, M.Kowalski, S.Trychin, L.Bourdev, D.Goldstein, and J.Hughes. Real-Time Nonphotorealistic Rendering. *Computer Graphics (Proceedings of SIGGRAPH 97)*, pp.415-420, 1997.

[19] O.Monga, S.Benayoun, and O.D.Faugeras. From Partial Derivatives of 3D Density Images to Ridge Lines. *Proceedings of the 1992 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 354-359, 1992.

[20] T.Saito and T.Takahashi, Comprehensible Rendering of 3D Shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, pp.197-206, 1990.

[21] P.Samson and J.Mallet. Curvature Analysis of Triangulated Surfaces in Structural Geology. *Mathematical Geology*, 29(3):391-412, 1997.

[22] D.Stalling. Personal communication, 1997.

[23] K.Stevens and A.Brookes. Probing Depth in Monocular Images. *Biological Cybernetics*, 56:355-366, 1987.

[24] K.Stevens. The Visual Interpretation of Surface Contours. *Artificial Intelligence*, 17:47-73, 1981.

[25] G.Taubin. Estimating the Tensor of Curvature of a Surface from a Polyhedral Approximation. *Proceedings of the 5th International Conference on Computer Vision (ICCV)*, pp.902-907, 1995.

[26] G.Winkenbach and D.Salesin. Rendering Parametric Surfaces in Pen and Ink. *Computer Graphics (Proceedings of SIGGRAPH 96)*, pp.496-476, 1996.

Understanding Errors in Approximating Principal Direction Vectors

By Jack Goldfeather and Victoria Interrante

Introduction

Suppose we are given only a surface mesh of vertices and polygons approximating some unknown smooth surface. There have been many methods proposed for approximating principal directions of the underlying surface [1,2,4,8,12,13]. In this paper we will examine a few of the known methods, showing how well they can work in some cases and how miserably they can fail in others. In particular we will show how very tiny normal curvature approximation errors can be magnified into large errors in the estimated principal directions. We also introduce a new method that we believe performs significantly better under certain conditions than many other proposed methods. In section I, we briefly describe the motivation for this work and its significance to applications in computer graphics. In section II, we outline the basic mathematics behind computing principal directions, stating the necessary formulas for the Weingarten curvature matrix and for its use in computing normal curvature in a given direction. In section III we describe in detail three methods each of which approximates the Weingarten curvature matrix at a vertex of the mesh. In section IV we apply each method to a test surface using a number of different mesh schemes and then examine the direction errors. In section V we develop some mathematical relationships between surface approximation errors and errors in principal directions and illustrate how similar-appearing approximation errors can lead to vastly different principal direction errors. In particular, we illustrate that while the chance for large errors increases near umbilical points, they can still occur at points on the surface where there is significant difference between the two principal curvatures. We conclude by summarizing the conditions under which principal direction estimation errors are most likely to occur, suggesting some steps that can be taken to improve the situation, and outlining promising directions for future work in this area.

I. Motivation

There are many applications for which it is useful to be able to calculate accurate estimates of the principal directions at points on a given surface. Principal direction vector fields have been widely used in shape analysis and surface interrogation in computer-aided manufacturing and design [7,11]. In these applications the surfaces are analytically defined, so the principal directions can be solved for directly and there are few problems with errors in the estimations. Principal direction vector fields have also been successfully used in conjunction with volumetrically-defined data to enhance the visual representation of surface shape for applications in molecular [3] and medical [10] visualization. In these applications the principal directions can be obtained by diagonalizing the Second Fundamental Form, whose entries can be fairly well approximated using first and second directional derivatives of the sampled 3D distribution.

Recently, there has been increasing interest expressed in the possibility of using principal direction vector fields over polygonal meshes for such purposes as guiding the direction of hatching strokes in pen-and-ink style renderings [5,9] or adaptively guiding the orientation of synthesized anisotropic texture patterns for enhanced surface shape representation [6]. However such efforts have been complicated by the lack of a robust and reliably accurate method for estimating the principal directions at points on an underlying smooth surface that is represented solely by a polygonal mesh. Although a number of methods for principal direction estimation have been previously published [13,8,2,4,1,12], inevitably what we have found in practice is that the computed vector field appears "noisy", requiring smoothing that can result in many of the vectors falling out of alignment with the true principal directions and complicating our efforts to generate surface markings that closely follow the form.

In the research described in this paper we sought to gain insight into the sources of errors in principal direction estimation on surfaces defined by point samples organized in a polygonal mesh, in order to determine what the worst potential pitfalls are and how they might be overcome. Our goal was to enable an approach in which instead of accepting large principal direction estimation errors and then working to hide them, we could strive to obtain more accurate initial principal direction estimates while flagging the points at which the probability of obtaining erroneous estimates is high.

II. A Quick Review of Surface Curvature

Let p be a point on a smooth surface S, let N_p be the unit normal to S at p, and suppose $X(u,v)$ is a local parametrization of S in a neighborhood of p. Then using $X_u(p)$, $X_v(p)$, N_p as a local coordinate system, we can compute principal curvatures and principal directions as follows: Let λ_1 and λ_2 ($\lambda_1 \geq \lambda_2$) be the eigenvalues, and p_1, p_2 the associated unit eigenvectors of the Weingarten curvature matrix

$$W = \begin{pmatrix} \frac{eG - fF}{EG - F^2} & \frac{fE - eF}{EG - F^2} \\ \frac{fG - gF}{EG - F^2} & \frac{gE - fF}{EG - F^2} \end{pmatrix}$$

where

$$e = N_p \circ X_{uu}(p) \qquad E = X_u(p) \circ X_u(p)$$

$$f = N_p \circ X_{uv}(p) \qquad F = X_u(p) \circ X_v(p)$$

$$g = N_p \circ X_{vv}(p) \qquad G = X_v(p) \circ X_v(p)$$

Note that in the special case that X_u and X_v are orthogonal unit vectors, this becomes the symmetric matrix

$$W = \begin{pmatrix} e & f \\ f & g \end{pmatrix}.$$

If u is a unit vector in the tangent plane to S at p, then

$$\kappa_u = u^T W u$$

is the **normal curvature** of the surface in the direction of u.

It follows that λ_1 and λ_2 are the maximum and minimum normal curvatures of the surface at p, and $p_1 = \begin{pmatrix} p_{11} \\ p_{12} \end{pmatrix}$ and $p_2 = \begin{pmatrix} p_{21} \\ p_{22} \end{pmatrix}$ are the principal curvature vectors expressed in local coordinates. That is,

$$v_1 = p_{11} X_u + p_{12} X_v$$

$$v_2 = p_{21} X_u + p_{22} X_v$$

are the principal direction vectors in R^3.

An important observation is that at points where $\lambda_1 = \lambda_2$, the notion of principal direction is not defined, since all vectors are eigenvectors. Such a point is called an *umbilical* point on the surface.

III. Three Principal Direction Approximation Methods

The first step in computing principal directions on a surface mesh is to compute at each vertex p a vector N'_p that approximates the true unit surface normal N_p at p. Most methods compute a "normalized average" - i.e. a set of vectors is summed and the resulting vector is normalized to length 1. As several different surface normal approximation methods have been proposed, we decided to take a closer look at three:

1. The normalized average of the surrounding triangle unit normals.
2. The normalized average of the surrounding triangle unit normals, each weighted by the angle of the triangle at p.
3. The normalized average of the adjacent edges, each weighted by the sum of the cotangents of the angles opposite the edge in the two triangles sharing the edge [2]. This weighting only makes sense if each pair of triangles is acute, so another method must be used at vertices where a triangle is not acute.

We experimented with each of these schemes on a complicated surface, using a variety of local meshes and a large number of random points on the surface. Figure 1 shows our findings.

In all cases, the vertices that we used lay in the analytically-defined surface. Under these conditions, all of the normal approximation methods worked very well. The average error in the approximated normal over all mesh types was 0.90 degrees for the unweighted method, 0.97 degrees for the angle-weighted method, and 1.04 degrees for the edgelength-weighted method. The median error over all mesh types was 0.41 degrees for the unweighted method, 0.50 degrees for the angle-weighted method, and 0.54 degrees for the edgelength weighted method.

Although the normal vector approximation errors can appear from Figure 1 to be superficially slightly smaller for the angle-weighted method in the cases of the global meshes and for the unweighted method in the cases of the local meshes, these differences are not significant either practically or statistically. Of course, using an incorrect surface normal will introduce error into a principal direction calculation. However, as we will show, even if the exact surface normal is used, there are other sources of significant potential error in principal direction estimation. For this reason, we decided to use exact surface normals in the subsequent steps of our investigation.

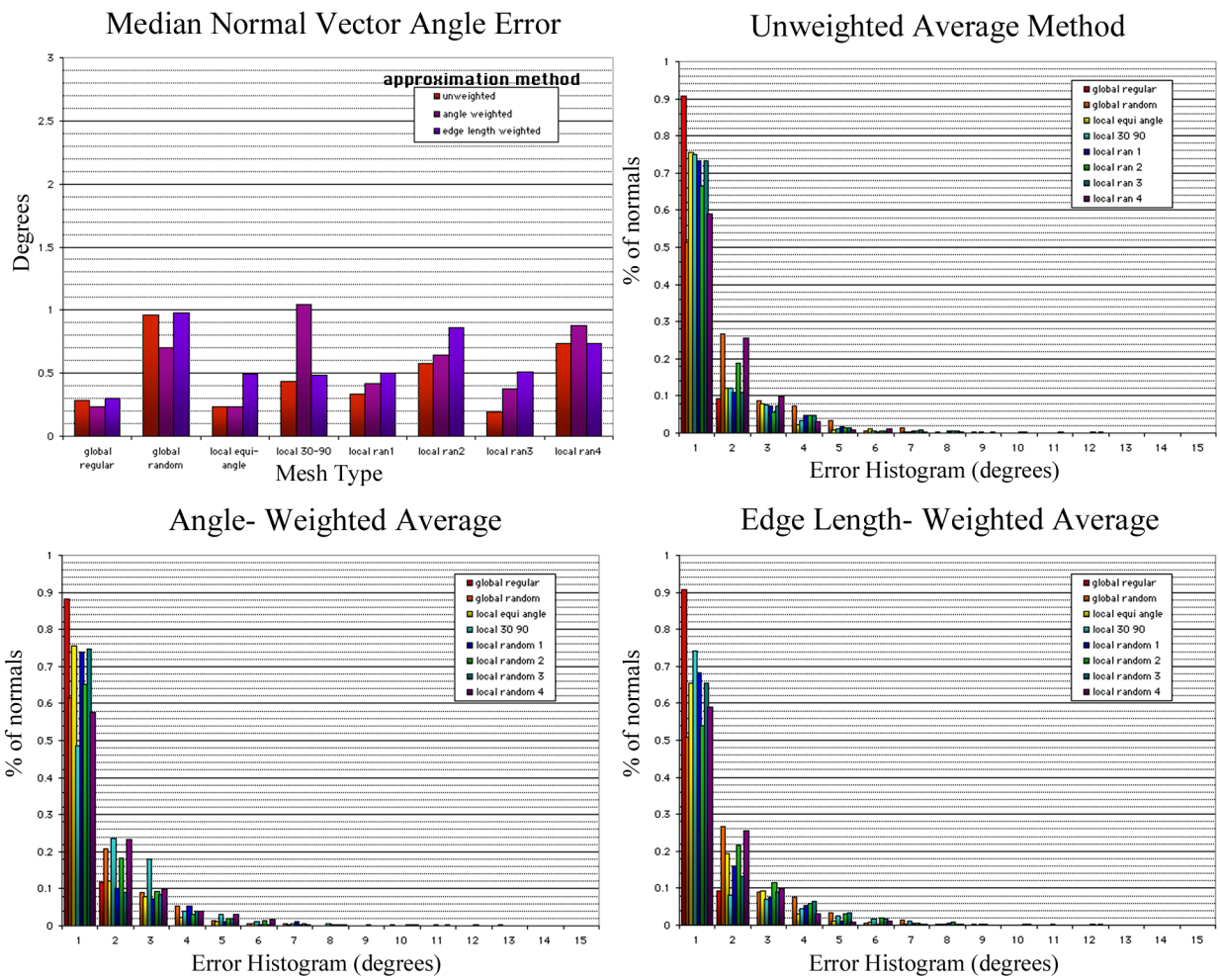


Figure 1. An experimental comparison of three normal vector approximation methods. Upper left: Median errors, per method and mesh type. Upper right: Error histogram, unweighted averaging method. Lower left: Error histogram, angle- weighted averaging method. Lower right: Error histogram, edge length- weighted averaging method.

The next step in estimating the principal directions is to choose a pair of orthonormal vectors x_1 and x_2 in the plane tangent to N'_p to form a local orthonormal coordinate system $L = \{x_1, x_2, N'_p\}$ in R^3. All subsequent calculations are done with respect to this local coordinate system. We examined three principal direction estimation methods, outlined below, each of which approximates the Weingarten curvature matrix expressed in this local coordinate system.

III.a The Normal Curvature Approximation Method

Suppose W is the unknown Weingarten matrix with respect to local coordinates L at vertex p. Suppose there are n vertices adjacent to p and let q_i denote the ith adjacent vertex. We denote by y_i the unit vector obtained by projecting the vector $p\bar{q}_i$ (expressed in local coordinates L) onto the plane tangent to N'_p and normalizing the result. Then using the result from section I, the normal curvature in the direction y_i is given by $\kappa_{y_i} = y_i^T W y_i$. An approximation to this normal curvature is given by:

$$\kappa'_{y_i} = 2\frac{(p - q_i) \circ N'_p}{(p - q_i) \circ (p - q_i)}$$

which is the curvature of the unique osculating circle passing through p and q_i with normal N'_p at p [2]. This produces a system of equations:

$$y_i^T W y_i = \kappa'_{y_i} \qquad i = 1, 2, \ldots, n \tag{1}$$

that we wish to solve for W. In [2], these equations are weighted in the same manner as the weighting in the third normal approximation method mentioned at the beginning of this section, but it is not clear that this improves the result much. The fact is that the osculating circle only produces a second-order approximation to the true normal curvature and second-order approximations can introduce significant error in many cases.

The first step in solving for W is to reorganize (1) as follows. Let $y_i = (u_i, v_i)$ and

$$W = \begin{pmatrix} A & B \\ B & C \end{pmatrix}.$$

Then

$$y_i^T W y_i = \begin{pmatrix} u_i & v_i \end{pmatrix} \begin{pmatrix} A & B \\ B & C \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} u_i^2 & 2u_iv_i & v_i^2 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix}.$$

If we let U be the $n \times 3$ matrix with rows $\begin{pmatrix} u_i^2 & 2u_iv_i & v_i^2 \end{pmatrix}$, $x = \begin{pmatrix} A & B & C \end{pmatrix}^T$, and d be the n-vector whose i^{th} entry is κ'_{y_i} the entire system can be written as the matrix equation

$$Ux = d. \tag{2}$$

If d was the vector of true normal curvatures, this linear system would have an exact solution (assuming the adjacent vertices do not have some degenerate pattern, like all lying on the same line.) In practice, the

best we can hope to find is a least squares fit, i.e., a vector x that minimizes $||Ux - d||$. There are standard numerical methods for finding this least squares solution. The resulting matrix

$$W' = \begin{pmatrix} A' & B' \\ B' & C' \end{pmatrix}$$

is used to approximate the principal directions. It is worth noting that the matrix U depends only on the projection of the adjacent vertices onto the local tangent plane, i.e. it is fixed by the choice of the mesh on the surface. The vector d depends on measurements we make at these adjacent vertices - i.e. it varies according to the amount of error we make. As we will see in a later section, it is possible to separate the mesh from the measurements so that we can understand how measurement error interacts with the local mesh geometry.

III.b The Quadratic Surface Approximation Method

In this method we try to best-fit a quadratic surface to the adjacent vertices. We begin by transforming each adjacent vertex q_i to local coordinates (x_i, y_i, z_i). In these local coordinates, p becomes $(0, 0, 0)$, N'_p lies along the positive z-axis, and the quadratic surface looks like

$$z = f(x, y) = \frac{A}{2}x^2 + Bxy + \frac{C}{2}y^2.$$

It is easy to show that the Weingarten matrix for such a surface is

$$W = \begin{pmatrix} A & B \\ B & C \end{pmatrix}.$$

As in the Normal Curvature method, we plug in the adjacent vertices to get a system of equations

$$\begin{pmatrix} \frac{1}{2}x_i^2 & x_i y_i & \frac{1}{2}y_i^2 \end{pmatrix} x = z_i \qquad i = 1, \ldots, n.$$

As before, we can find a least-squares fit to this system. It order to be able to compare the result to the normal curvature method, we scale the ith equation by $\frac{2}{k_i^2}$, where $k_i = \sqrt{x_i^2 + y_i^2}$. Then $(x_i, y_i) = k_i(u_i, v_i)$ so that

$$\begin{pmatrix} \frac{1}{2}x_i^2 & x_i y_i & \frac{1}{2}y_i^2 \end{pmatrix} = \frac{k_i^2}{2} \begin{pmatrix} u_i^2 & 2u_i v_i & v_i^2 \end{pmatrix}.$$

Then we can write each scaled equation in the system as:

$$\begin{pmatrix} u_i^2 & 2u_iv_i & v_i^2 \end{pmatrix} x = d_i$$

where $d_i = \frac{2}{k_i^2} z_i$. There is a nice geometric interpreatation of d_i. Consider the parabola with equation $y = \frac{d_i}{2} x^2$. Plugging in $x = k_i$ we get

$$y = \frac{d_i}{2} k_i^2 = z_i$$

so that this parabola passes through the origin and the point (k_i, z_i). Since $k_i = \sqrt{x_i^2 + y_i^2}$, we can think of this parabola in three dimensions as passing through the local coordinate origin and the point (x_i, y_i, z_i).

The one variable, planar formula for computing curvature is :

$$\kappa = \frac{|y''|}{(1 + y'^2)^{3/2}}.$$

Using this formula for the parabola, at (0,0) we get $\kappa = |d_i|$. In other words, d_i is the normal curvature of a parabola passing through (x_i, y_i, z_i) and the origin. We can now see that the Quadratic Surface method is identical to the Normal Curvature method, except that curvature approximations are done using parabolas rather than circles. This suggests that these two methods will produce similar results, with one performing slightly better than the other depending on whether circles or parabolas better approximate the surface locally in some direction.

III.c The Adjacent-Normal Cubic Approximation Method

In both of the proceeding methods, we did not use all of the information available to us. Namely, we did not use the known (approximated) normal vectors at adjacent vertices q_i. We can use this information to create a third-order approximation method that we believe has not yet appeared in the literature. As we will see in the next section, this method seems to perform significantly better than the first two in many cases.

As in the Quadratic method we try to fit a surface to the adjacent vertex data. Let

$$f(x, y) = \frac{A}{2} x^2 + Bxy + \frac{C}{2} y^2 + Dx^3 + Ex^2y + Fxy^2 + Gy^3.$$

The Weingarten matrix for this surface is still $W = \begin{pmatrix} A & B \\ B & C \end{pmatrix}$, because in the local coordinate system the curvature only depends on the second degree terms. However, using the third degree terms in the least-squares fit will produce values for (A, B, C) different from the ones we get in the quadratic method. The normal to this surface is given by:

$$N(x, y) = (f_x(x, y), f_y(x, y), -1)$$

$$= (Ax + By + 3Dx^2 + 2Exy + Fy^2, Bx + Cy + Ex^2 + 2Fxy + 3Gy^2, -1).$$

Let (a_i, b_i, c_i) denote the normal at the data point (x_i, y_i, z_i) (both normal and point must be transformed to the local coordinates), and let $x = (\,A \quad B \quad C \quad D \quad E \quad F \quad G\,)^T$. Rewrite the normal as $(-\frac{a_i}{c_i}, -\frac{b_i}{c_i}, -1)$. Then for each point, we have an equation

$$(\,\tfrac{1}{2}x_i^2 \quad x_iy_i \quad \tfrac{1}{2}y_i^2 \quad x_i^3 \quad x_i^2y_i \quad x_iy_i^2 \quad y_i^3\,)\,x = z_i$$

and for each normal we have two equations:

$$(\,x_i \quad y_i \quad 0 \quad 3x_i^2 \quad 2x_iy_i \quad y_i^2 \quad 0\,)\,x = -\frac{a_i}{c_i}$$

$$(\,0 \quad x_i \quad y_i \quad 0 \quad x_i^2 \quad 2x_iy_i \quad 3y_i^2\,)\,x = -\frac{b_i}{c_i}.$$

As in the preceding method, we scale these equations by the same scale factor $\frac{2}{k_i^2}$, to obtain a system

$$Ux = d$$

where U is a $3n \times 7$ matrix and d is a $3n$-vector. Again, we find a least-squares fit, but use only A, B, and C from the result. Note that n has to be at least 3 for there to be at least as many equations as unknowns.

III.d Higher Order Methods

The preceding method can be extended to higher orders:

$$f_4(x,y) = \frac{A}{2}x^2 + Bxy + \frac{C}{2}y^2 + \sum_{n=0}^{n=3} D_n x^{3-n} y^n + \sum_{n=0}^{n=4} E_n x^{4-n} y^n$$

$$f_5(x,y) = \frac{A}{2}x^2 + Bxy + \frac{C}{2}y^2 + \sum_{n=0}^{n=3} D_n x^{3-n} y^n + \sum_{n=0}^{n=4} E_n x^{4-n} y^n + \sum_{n=0}^{n=5} F_n x^{5-n} y^n$$

and so on. However, in order that there be at least as many equations as unknowns, the maximum degree of the approximation that we can compute is limited by the number of adjacent vertices in the mesh. For example, if we use 6 adjacent vertices, then there will be 18 equations. The highest order method that we can use is then degree 5, since such an approximation will have the 18 unknowns $(A, B, C, D_0, \ldots, D_3, E_0, \ldots, E_4, F_0, \ldots, F_5)$.

As we will see in what follows, higher order methods seem to make small errors smaller, but can also make large errors larger.

IV. Testing the Methods

We have noted that principal direction approximation methods are often tested using very simple models, such as surfaces of revolution triangulated using a regular pattern. Most methods seem to work well in these circumstances. However we have found that these previously proposed methods can fail for more complex surfaces with irregular mesh patterns. By fail we mean that the approximated principal directions can be wildly off from the true ones, even at non-umbilical points. We sought to gain deeper insight into the source of these errors, in order to better determine how it might be possible to attempt to overcome them.

We set out to design a test surface that had significant areas where the curvature was "interesting", and we devised a number of different ways to create local triangular meshes on this surface.

Our surface, $S(u,v)$, is defined by:

$$f(u) = -2u^4 + 2u^2 + u/6 + 0.3$$

$$g(u) = \begin{cases} .9 + u + f(-.9), & -.9 - f(-.9) \le u \le -.9; \\ f(u), & -.9 \le u \le 1; \\ 1 - u - f(1), & 1 \le u \le 1 + f(1); \end{cases}$$

$$h(u) = \begin{cases} -.9, & -.9 - f(-.9) \le u \le -.9; \\ u, & -.9 \le u \le 1; \\ 1, & 1 \le u \le 1 + f(1); \end{cases}$$

$$x(u,v) = g(u)\cos(v)$$

$$y(u,v) = g(u)\sin(v)$$

$$z(u,v) = h(u) + 0.2\sin(2x(u,v)) + 0.15\cos(3x(u,v)y(u,v))$$

$$S(u,v) = (x(u,v), y(u,v), z(u,v)) \qquad .9 - f(-.9) \le u \le 1 + f(1), \qquad 0 \le v \le 2\pi.$$

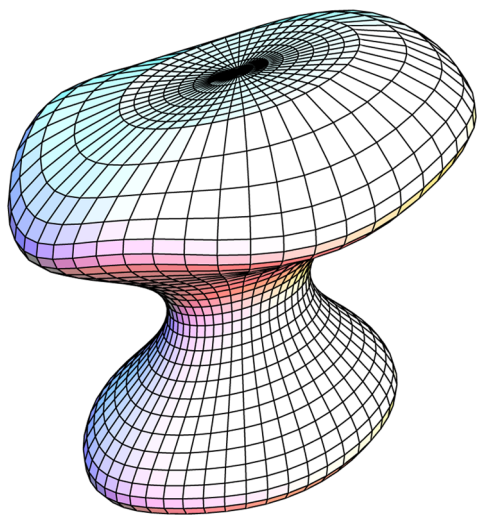


Figure 2. Test Surface

We selected 575 points on this surface, none of which were umbilical points. Table 1 shows the distribution of principal curvature differences for these points.

TABLE 1. Principal Curvature Differences on Test Surface

$\lambda_1 - \lambda_2$	Points with $\lambda_1 - \lambda_2$ in indicated range
0-0.1	17
0.1-0.2	47
0.2-0.3	37
0.3-0.4	29
0.4-0.5	30
> 0.5	415

In devising mesh schemes, we had a number of hypotheses we wished to test:

1. Does regularity of the mesh improve accuracy and randomness reduce it?
2. Does enforcing equidistance to adjacent vertices improve accuracy?
3. Does enforcing equiangularity between adjacent vertices improve accuracy?
4. Does choosing adjacent vertices along the true principal directions matter?

With these in mind, we used two approaches to creating meshes.

IV.a Global Parameter Space Triangulations

First we created a Global Regular triangulation of parameter space. Figure 3a shows a 4 by 4 global regular triangulation. The actual triangulation we used was 50 by 50. Then we created a Global Random triangulation by randomly moving the interior points. Figure 3b shows a 4 by 4 global random triangulation.

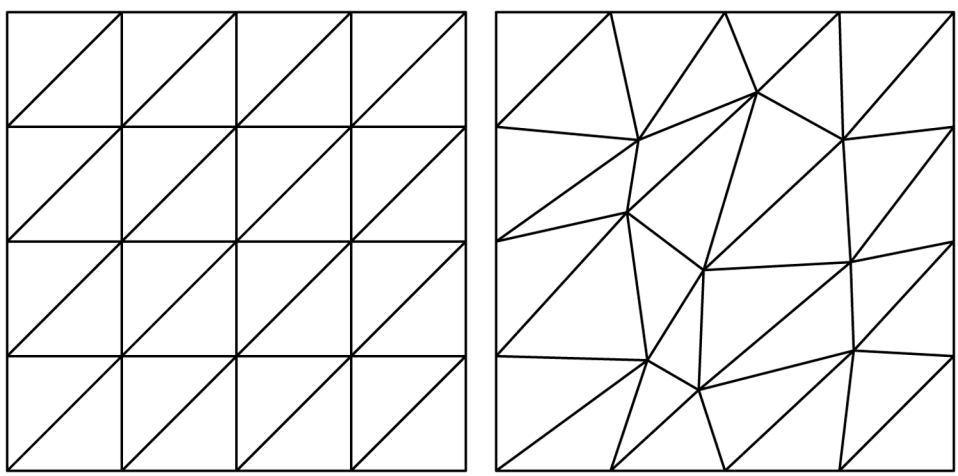


Figure 3. Left: Global Regular Triangulation, Right: Global Random Triangulation.

IV.b Local Tangent Space Projections

Next we created six local meshing patterns, which we defined independently around each of 575 individual vertices from the global random mesh. We used these patterns to investigate the hypotheses listed in the preceding section. We began by defining the six planar mesh patterns shown in Figure 4. These patterns were formed by using various rotations of a single radial edge about a given centerpoint.

In all cases six edges were used, with the first edge defined to be aligned with the analytically-determined first principal direction. In Local Mesh 1, the angle of rotation between all of the edges was defined to be equal (60°). In Local Mesh 2, the angles between the edges were defined in a regular pattern with 3-way symmetry, alternating 30° with 90°. Local Meshes 3-6 were generated by choosing the five successive angles of rotation randomly in [1° .. 90°].

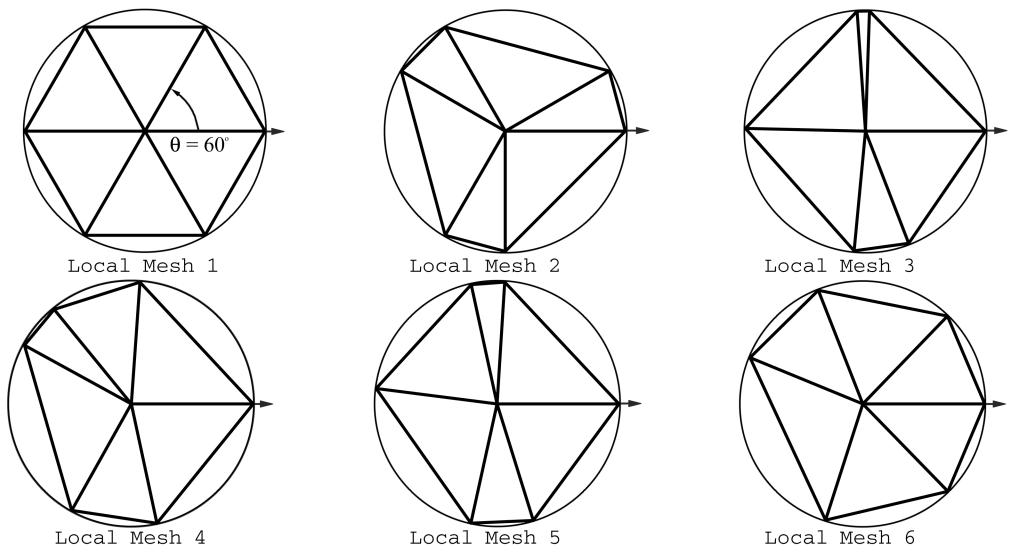


Figure 4. Six Local Mesh patterns.

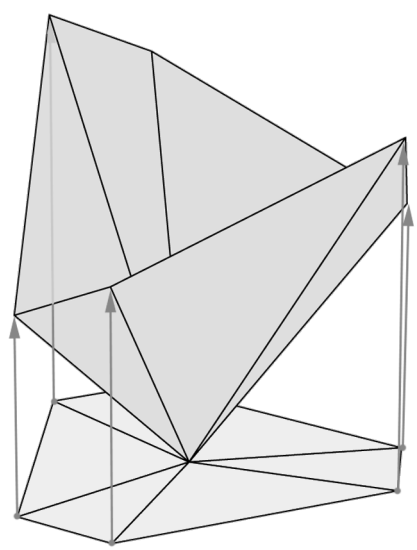


Figure 5. Surface mesh obtained from local mesh in tangent plane.

For each of these patterns and at each of the points on the surface created by the Global Random triangulation we then determined the points on the surface that projected onto the pattern in the tangent plane at that point (Figure 5). Finally, we rotated the patterns in the tangent plane 5 times (72 degrees each time). The initial orientation of each local mesh was configured to have one edge lying along a principal direction. Subsequent rotations moved this edge away from the principal direction. We did this in order to see if having mesh edges coincidentally aligned with or counter-balanced against the principal directions affected the accuracy of the approximation.

Figure 6 shows the estimated principal directions computed using each of the three methods in the case where the data is represented by the Global Regular mesh. All of the methods performed very well under these conditions. Figure 7 shows the estimated principal directions computed using the same three methods at the vertices of the Global Random mesh. Note the increase in the prevalence of large angle errors, especially with the two second order methods; estimated directions more than 10° out of alignment with the true principal directions are highlighted in red.

More results for our surface are summarized in Figures 8-9 and Tables 2-5 in the appendix. Figure 8 shows the median angle error, in degrees, for each of our methods, using each of our meshes for 575 points on the surface, across all rotations. The size and prevalence of errors varied according to the mesh type, with the smallest errors occurring in the local equi-angle case and the greatest errors occurring in the cases of the global and local random meshes and the local 30°-90° pattern. Performance was best, overall, when the Cubic method was used, and was not as good in the cases of the Quadratic and Normal Curvature methods. The performance in the cases of these two second order methods was very similar.

Figure 9 provides a more detailed look at the data, showing the percent of the mesh points having angle errors below certain thresholds, for each method. Figure 7a shows, for example, that for the Cubic method performance was fairly consistent across all mesh types, with approximately 22-27% of the mesh points having angle errors above 1°, and with that number rapidly dropping to below 5% for errors above 10°-15°. Figure 7b shows that performace was more varied by mesh type in the case of the Normal Curvature method, with between 24% to over 40% of the mesh points having angle errors above 1°, and with that number dropping below 5% at thresholds between 7°-22°, depending on mesh type.

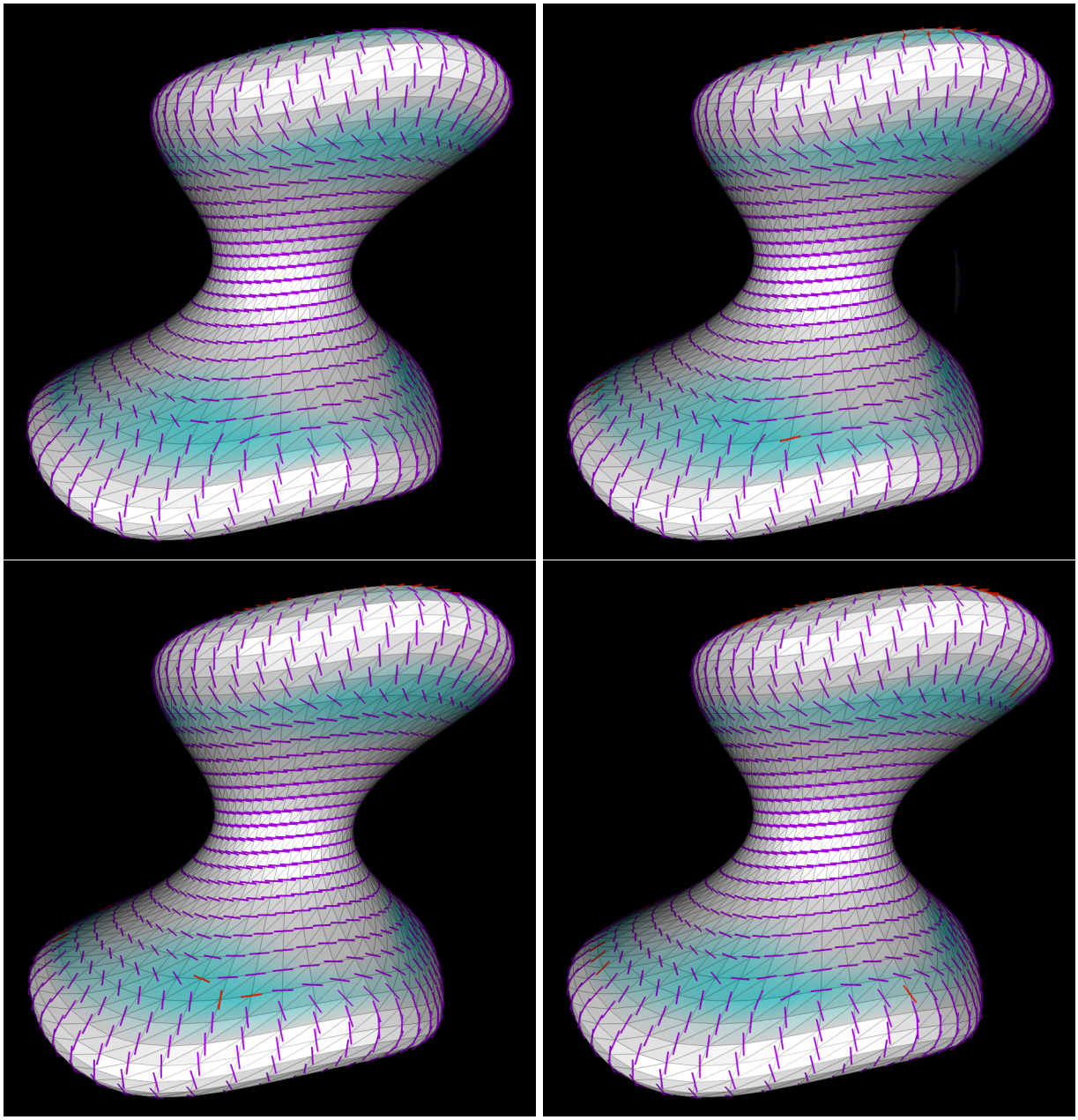


Figure 6. Approximated principal directions, Global Regular Mesh. Directions marked in red have errors above $10°$. Surface coloration highlights areas in which the difference between the two principal curvatures becomes small (near-umbilic points). Upper left: Exact directions. Upper right: Approximated directions, Quadratic Method. Lower left: Approximated directions, Normal Curvature Method. Lower right: Approximated directions, Cubic Method.

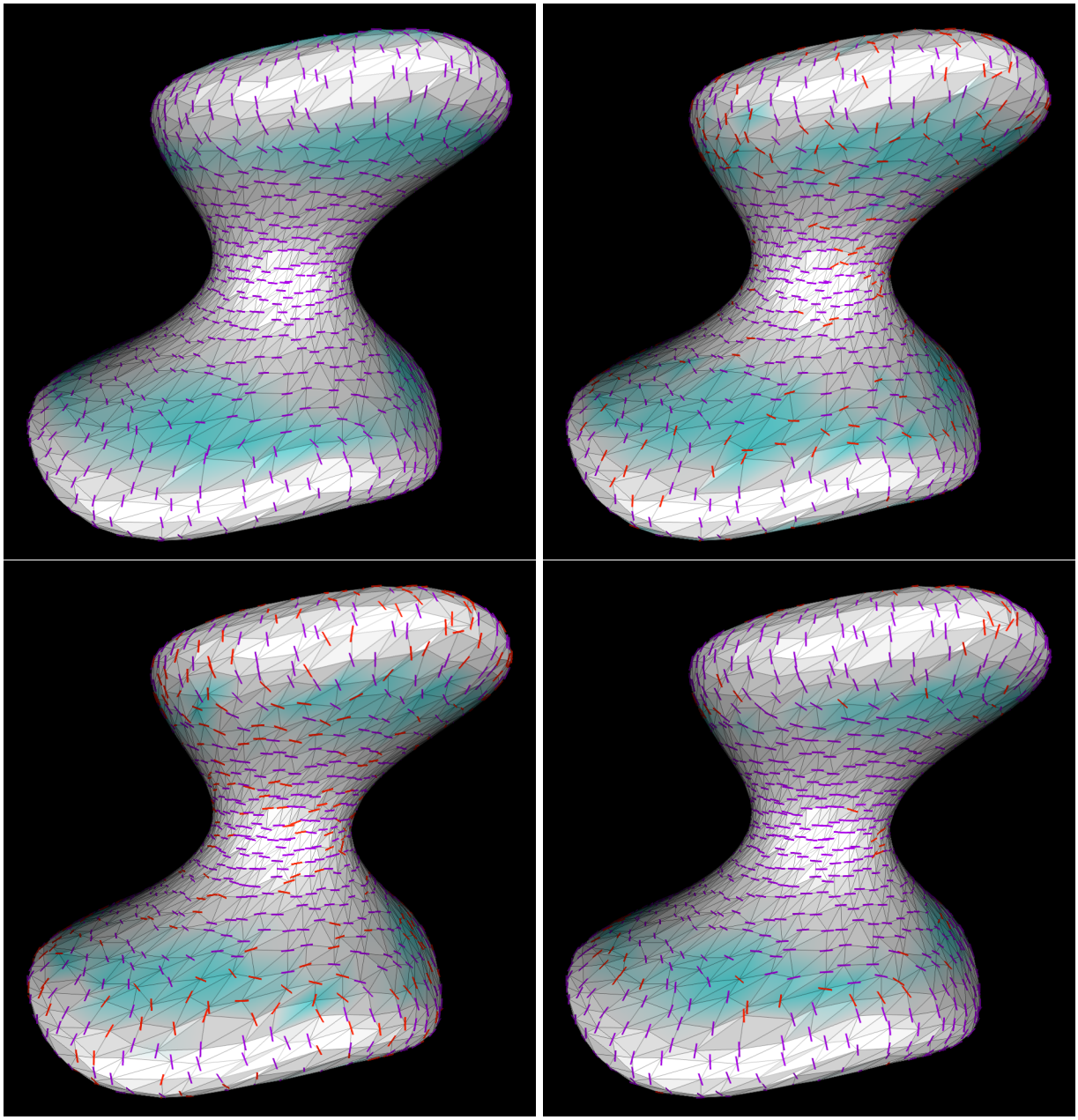


Figure 7. Approximated principal directions, Global Random Mesh. Directions marked in red have errors above $10°$. Surface coloration highlights areas in which the difference between the two principal curvatures becomes small (near-umbilic points). Upper left: Exact directions. Upper right: Approximated directions, Quadratic Method. Lower left: Approximated directions, Normal Curvature Method. Lower right: approximated directions, Cubic Method.

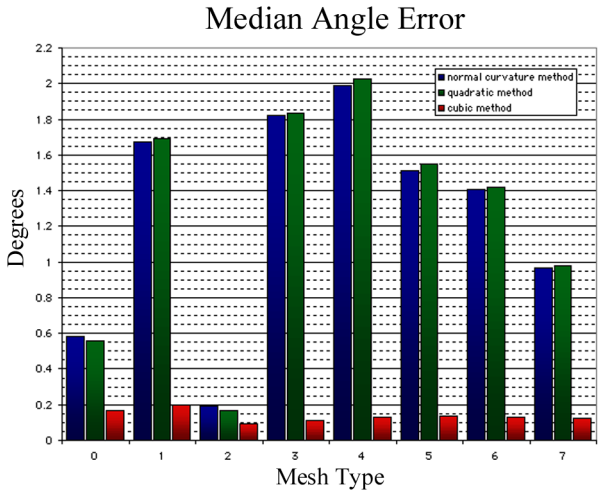


Figure 8. Median error in the approximated principal directions, per mesh type and method.

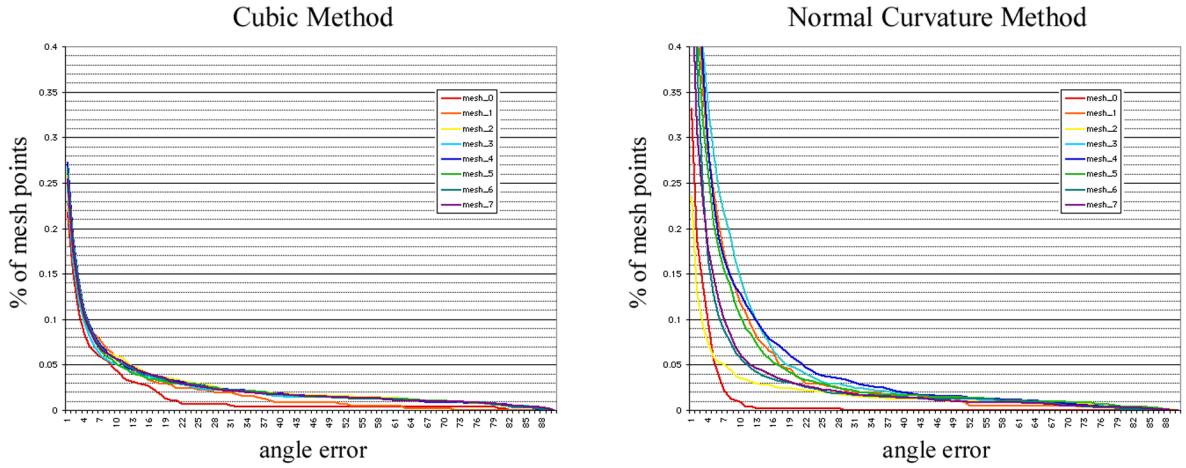


Figure 9. Prevalence of errors in the approximated principal directions, by magnitude.

Table 2, given in the appendix, provides numeric data listing the number of points at which the degree error fell into each indicated range. For example, the first entry in the table says that out of 575 vertices tested using the Global Regular mesh, 498 had a principal direction error of less than 3°, 56 had an error between 3°-6°, and 21 had an error of more than 6°. The rotations only make sense for the six Local Mesh schemes. We found no apparent advantage for the principal-direction aligned rotations (see, for example, the equiangular case).

Table 3 shows that the results are virtually identical when the constraint of equal edge length on the surface (as opposed to on the local parameterization in the tangent plane) is enforced. At each vertex we recomputed the local mesh to have edges of length equal to the average of the 6 local edges from Table 2. In Table 4 we enforced a global equal length constraint. The average edge length in Table 2 is 0.08. Table 4 shows the result for all edges constrained to be of lengths 0.05 and 0.1. We show only the Rotation 1 column, as the other columns are similar.

Table 5 separates out those points with a principal curvature difference of less than 0.2. Each entry of the form n(m) is read as n points with curvature difference greater than 0.2 and m points with curvature difference less than 0.2, so for example, the first entry 464(34) 33(23) 18(3) says that of the 21 points where there was an error of more than 6 degrees, only 3 of them were even close to being umbilical points. Table 5 is perhaps the most revealing. It reveals that large principal direction errors can occur **away from umbilical points**.

The results suggest several conclusions:

1. The degree ≥ 3 methods nearly consistently outperform both of the degree 2 methods.
2. The regular meshes tend to produce better results than the random ones.
3. The equiangular mesh (Local Mesh 1) produces good results for all three methods.
4. Aligning edges with principal directions does not seem to matter.
5. Enforcing equality of adjacent edge length does not seem to matter, but the shorter all of the edges are the better.
6. Large errors can occur away from umbilical points.

V. A Unified Approach to Error Analysis

The previous section illuminates how well or badly the three methods that we examined can work. We saw that some mesh schemes seem to produce fewer large errors than others, but that in nearly every case there are non-umbilical points where the error is large. In this section we will develop the mathematics necessary to understand why this is so.

From a mathematical point of view, all three of the methods outlined in Section III have the same two steps:

(1) Find a least-squares fit to a linear system

$$Ux = d$$

where U is computed from the mesh geometry and d is computed from measurements at the adjacent vertices in the surface mesh.

(2) Use the first three entries, $\begin{pmatrix} A' \\ B' \\ C' \end{pmatrix}$, in the least-squares solution x to form a symmetric matrix

$$W' = \begin{pmatrix} A' & B' \\ B' & C' \end{pmatrix}$$

and then find the eigenvalues and eigenvectors of W'.

We seek to understand how the local mesh geometry and the measurement error interact to create error in principal direction.

We begin with a standard result from linear algebra:

Theorem 1. A least squares solution to $Ux = d$ is the exact solution to $U^T Ux = U^T d$.

In our situation, U is known exactly and is computed from the projection of the adjacent vertices onto the local tangent plane, but d is known only approximately. If we let d_T denote the (unknown) vector that d approximates and x_T the exact solution to $U^T Ux_T = U^T d_T$, then

$$U^T U(x - x_T) = U^T(d - d_T).$$

Then $e = x - x_T = \begin{pmatrix} e_A \\ e_B \\ e_C \end{pmatrix}$ is what creates the error in our approximation to the Weingarten matrix. More precisely,

$$W' = \begin{pmatrix} A' & B' \\ B' & C' \end{pmatrix} = \begin{pmatrix} A & B \\ B & C \end{pmatrix} + \begin{pmatrix} e_A & e_B \\ e_B & e_C \end{pmatrix}.$$

We will see in what follows that it is not so much the magnitude of e that creates principal direction errors, but rather the relationships between these errors that really determines the magnitude of the approximation errors.

The first step in our analysis is to show that direction errors are independent of the choice of local coordinates. It might seem that since the entries in the matrix U depend on the choice of local coordinates

in the tangent plane, the entries in the matrix W' will also depend on the local coordinates and this, in turn, will affect errors in direction. However, we can prove the following theorem:

Theorem 2. Suppose L and L' are two different orthonormal coordinate systems in the tangent plane and U and U' are the coefficient matrices found in any of our approximation methods using L and L', respectively. Then the approximated Weingarten matrices have the same eigenvalues and eigenvectors.

Proof: We prove the theorem for the Normal Curvature method, as the other methods have a similar proof. If (u, v) and (u', v') are the coordinates of a point in L and L', respectively, then

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = R \begin{pmatrix} u \\ v \end{pmatrix}.$$

Using this, it is not hard to show that $U' = US$ where

$$S = \begin{pmatrix} \cos^2\theta & 2\cos\theta\sin\theta & \sin^2\theta \\ -\sin\theta\cos\theta & \cos^2\theta - \sin^2\theta & \sin\theta\cos\theta \\ \sin^2\theta & -2\sin\theta\cos\theta & \cos^2\theta \end{pmatrix}.$$

Let $U^T U x = U^T d$ and $U'^T U' x' = U'^T d$. Replacing U' by US in the second equation produces

$$S^T U^T U S x' = S^T U^T d.$$

Since S is invertible, we get $U^T U S x' = U^T d$. That is, $x = Sx'$. Now if $x = (A, B, C)$ and $x' = (A', B', C')$, it is not hard to show that

$$\begin{pmatrix} A & B \\ B & C \end{pmatrix} = R^{-1} \begin{pmatrix} A & B \\ B & C \end{pmatrix} R.$$

That is, the approximate Weingarten matrices are orthogonally similar, so they have the same eigenvalues and eigenvectors. This completes the proof.

Theorem 2 tells us that our direction errors will not depend on how we chose the local coordinates. As a consequence of this theorem, it suffices to choose the true principal directions as our local coordinates. This will simplify some of our later calculations. In particular, with respect to these local coordinates, $W = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$ where λ_1 and λ_2 are the principal curvatures. Suppose W' is our approximated matrix with respect to these local coordinates. We express W' in two ways:

(1) Since W' approximates W,

$$W' = \begin{pmatrix} \lambda_1 + e_A & e_B \\ e_B & \lambda_2 + e_C \end{pmatrix}.$$

(2) Since W' is a symmetric 2 by 2 matrix, it can be orthogonally diagonalized. That is, it can be written

$$W' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \tau_1 & 0 \\ 0 & \tau_2 \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

where τ_1 and τ_2 are the eigenvalues of W' and $\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$ and $\begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$ are orthonormal eigenvectors. Note also that since we have chosen the true eigenvectors as our local coordinate system, θ is precisely the angle error between the true directions and the approximated ones. We can find θ by equating these two ways of writing W'. First, multiply out the three matrices in (2) to obtain:

$$W' = \begin{pmatrix} \tau_1 \cos^2\theta + \tau_2 \sin^2\theta & \tau_1 \sin\theta\cos\theta - \tau_2 \sin\theta\cos\theta \\ \tau_1 \sin\theta\cos\theta - \tau_2 \sin\theta\cos\theta & \tau_1 \sin^2\theta + \tau_2 \cos^2\theta \end{pmatrix}.$$

Equating corresponding entries in the two ways to express W', we obtain three equations:

(1) $$\tau_1 \cos^2\theta + \tau_2 \sin^2\theta = \lambda_1 + e_A;$$

(2) $$\tau_1 \sin^2\theta + \tau_2 \cos^2\theta = \lambda_2 + e_C;$$

(3) $$\tau_1 \sin\theta\cos\theta - \tau_2 \sin\theta\cos\theta = e_B.$$

Subtracting equation (2) from equation (1), and multiplying equation (3) by 2, we obtain, respectively:

(4) $$(\tau_1 - \tau_2)\cos(2\theta) = e_A - e_C + \lambda_1 - \lambda_2;$$

(5) $$(\tau_1 - \tau_2)\sin(2\theta) = 2e_B.$$

Finally, dividing equation (5) by equation (4) we get the result:

(6) $$\tan(2\theta) = \frac{2e_B}{e_A - e_C + \lambda_1 - \lambda_2}.$$

We summarize this in the following theorem:

Principal Error Theorem

Suppose $W' = W + E$ where

$$W = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \qquad E = \begin{pmatrix} e_A & e_B \\ e_B & e_C \end{pmatrix}$$

Then the angle θ between the first eigenvectors of W and W' satisfies

$$\tan(2\theta) = \frac{2e_B}{e_A - e_C + \lambda_1 - \lambda_2}.$$

Since tan is an increasing function, angle errors will be large when

$$\frac{2e_B}{e_A - e_C + \lambda_1 - \lambda_2}$$

is large.

Many simulations have convinced us that the size of the numerator in this expression is the most significant culprit in producing angle errors. Let's examine more carefully how it is computed.

From our earlier work in this section, the total error vector $e = (e_A, e_B, e_C, \ldots)$ is the solution to

$$U^T U e = U^T \delta d$$

where U is computed from the local mesh projected onto the local tangent plane, and $\delta d = d - d_T$ is the vector of errors we made because we did not compute the exact normal curvatures, (or the surface is not exactly quadratic or cubic, depending which method we are using). Hence

$$e = (U^T U)^{-1} U^T \delta d.$$

If we let $Y = (U^T U)^{-1} U^T$, and let Y_i denote the $i^(th)$ row of Y, then

$$e_B = Y_2 \circ \delta d = ||Y_2|| ||\delta d|| \cos \theta_{dY_2}.$$

where θ_{dY_2} is the angle between Y_2 and δd. Y_2 is fixed by the choice of local mesh, so we see that e_B can vary widely depending on θ_{dY_2}. In the worst case ($\theta_{dY_2} = 0$), $e_B = ||Y_2|| ||\delta d||$ and in the best case ($\theta_{dY_2} = \frac{\pi}{2}$), even if δd has large magnitude, $e_B = 0$.

We illustrate with an example how volatile the situation can be. Suppose we are using Local Mesh 3 at a point where the difference between the principal curvatures is 0.5, that is, away from an umbilical point.

This mesh's local uv coordinates are:

$$\begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \\ u_3 & v_3 \\ u_4 & v_4 \\ u_5 & v_5 \\ u_6 & v_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0.0334296 & 0.999441 \\ -0.0698233 & 0.997559 \\ -0.999741 & 0.0227421 \\ -0.0945938 & -0.995516 \\ 0.360022 & -0.932944 \end{pmatrix}$$

and

$$Y = (U^T U)^{-1} U^T$$

$$= \begin{pmatrix} 0.504502 & 0.00796006 & -0.0165789 & 0.498407 & 0.0275063 & -0.0217962 \\ 0.128266 & 0.427104 & -0.0347749 & 0.0265751 & 0.697892 & -1.24506 \\ -0.00117602 & 0.321524 & 0.263133 & -0.0136473 & 0.352888 & 0.0772777 \end{pmatrix}.$$

First we choose an error vector in the same direction as the second row of Y:

$$\delta d_1 = \begin{pmatrix} 0.0256532 & 0.0854208 & -0.00695498 & 0.00531501 & 0.139578 & -0.249012 \end{pmatrix}$$

To do this, we scaled down the second row to produce errors of a magnitude consistent with using a circle to approximate normal curvatures along 6 adjacent vertices. Using δd_1 produces the error vector:

$$e_1 = Y\delta d_1 = \begin{pmatrix} 0.0256532 & 0.447604 & 0.0555445 \end{pmatrix}.$$

We then apply the Principal Error Theorem:

$$\tan(2\theta) = \frac{2e_B}{e_A - e_C + \lambda_1 - \lambda_2} = \frac{2(0.447604)}{0.0256532 - 0.0555445 + 0.5} = 1.904.$$

producing an error of $\theta = 31.1471$ degrees.

Now we choose an error vector whose entries are close to those in δd_1, but which is perpendicular to the second row of Y:

$$\delta d_2 = \begin{pmatrix} 0.02 & 0.28 & 0.1 & -0.05 & 0.1 & 0.150304 \end{pmatrix}$$

In this case, $e_2 = Y\delta d_2 = \begin{pmatrix} -0.0147848 & 0 & 0.163903 \end{pmatrix}$ which produces an error of 0 degrees!

Figure 10 shows the correlation between the magnitude of the error and the each of the components in (6) for two of the methods examined, at 575 points on our test surface and using six different rotations of each local mesh at each point.

Note that the size of the numerator, $2e_B$, depends upon $||Y_2||$, which is determined by the mesh, $||d||$, which reflects the magnitude of the errors we have made in approximating the normal curvatures, and θ_{dY_2}, the angle between Y_2 and δd. Note also that the error vectors for the degree 3 method have 18 components, not 6, so although the magnitude of Y_2 in these cases is significantly smaller, the magnitude of an 18 component δd is likely to be greater. The size of the denominator in (6) depends both upon $e_A - e_C$ and upon the difference between the two principal curvatures, $\lambda_1 - \lambda_2$. By definition, $\lambda_1 > \lambda_2$. The magnitude of $e_A - e_C$ in turn depends upon $||Y_1 - Y_3||$, $||d||$, and the angle between $Y_1 - Y_3$ and δd. When both $\lambda_1 - \lambda_2$ and $e_A - e_C$ are small, larger errors become more likely.

However, it seems that it is the size of the numerator that has the most significant effect. In the case of the cubic method, the sizes of the errors in the estimates of the normal curvatures seems to be the factor most significantly influencing the size of the error in the principal direction estimates; in the case of the normal curvature method, the direction in which the errors were made seems for some meshes to be nearly as significant.

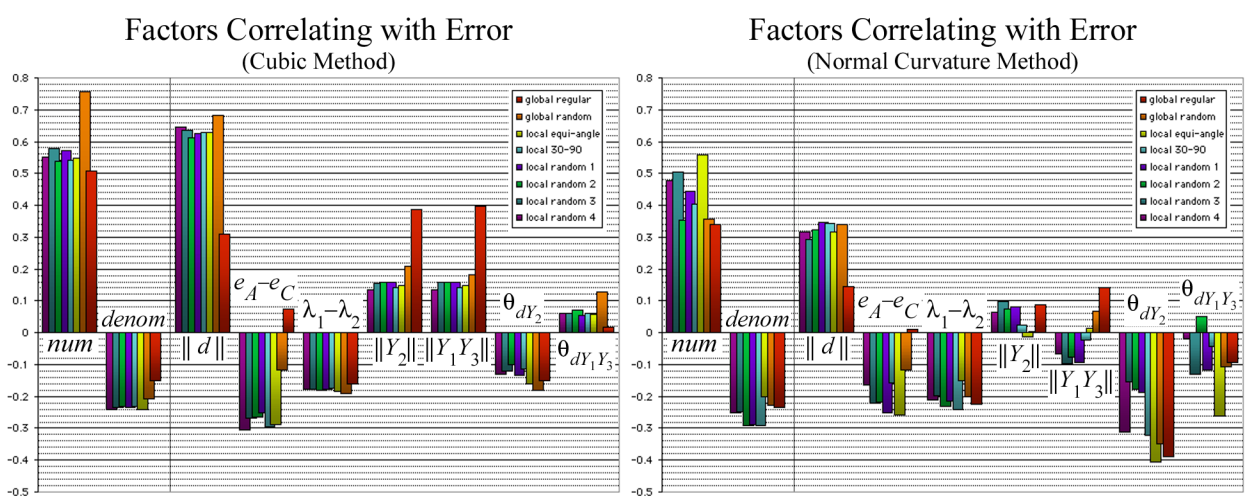


Figure 10. Correlation of the various components of equation (6) with error.

Table 6 in the appendix shows the median values of each of the components of (6) by mesh type and method. From this table it can be seen that the median errors appear to be smallest when the median angle between δd and Y_2 is closest to $90°$, indicating the advantage of situations in which a higher proportion of errors can cancel out.

V. Conclusions

We have shown that approximating principal directions on a surface mesh can be a tricky business. Although it is important to minimize the error in approximating various quantities such as normal curvature, an unfortunate pattern of small errors can produce large angle errors. Using a non-trivial test surface, we have shown that principal direction approximation methods are particularly prone to error when working with irregularly sampled data, and that these errors appear to be more severe for second order methods than for higher order methods that take advantage of more of the known information about the surface.

There are many important directions for future work in this area. A major practical concern that we did not touch upon at all in our work is the problem of surface smoothing. In many cases, approximate meshes are obtained from a sampling process that is prone to a certain amount of intrinsic error. How might we best recover the mesh that describes the continuous smooth surface that is approximated by a collection of noisy samples? If we have only a mesh that is highly irregular as well as noisy, how could we improve our chances of obtaining a smooth principal direction vector field and avoiding the introduction of large principal direction estimation errors? Would resampling to obtain a mesh in which the edge distribution is more equiangular help? Finally, it might be interesting to see how well the findings we have made hold up under further testing with a wider variety of mesh types and surface shapes.

TABLE 2. Degree Errors for 575 Points on Test Surface

Mesh Type	Rotation 1			Rotation 2			Rotation 3			Rotation 4			Rotation 5		
Degree Error	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6
Normal Curvature Method															
Global Regular	498	56	21												
Global Random	374	84	117												
Local Mesh 1	514	28	33	521	18	36	529	22	24	531	20	24	515	26	34
Local Mesh 2	327	101	147	412	65	98	351	112	112	321	88	166	321	82	172
Local Mesh 3	252	128	195	309	174	92	379	96	100	506	25	44	346	118	111
Local Mesh 4	316	124	135	433	75	67	437	66	72	473	44	58	309	91	175
Local Mesh 5	324	153	98	440	76	59	525	19	31	501	44	30	391	109	75
Local Mesh 6	466	58	51	490	46	39	487	39	49	373	78	124	401	92	82
Quadratic Approximation															
Global Regular	501	54	20												
Global Random	372	87	116												
Local Mesh 1	512	30	33	517	21	37	529	22	24	531	20	24	511	30	34
Local Mesh 2	322	104	149	412	65	98	347	114	114	320	87	168	320	79	176
Local Mesh 3	243	134	198	306	178	91	380	93	102	506	25	44	355	115	105
Local Mesh 4	316	130	129	433	73	69	434	69	72	471	45	59	306	89	180
Local Mesh 5	320	157	98	440	74	61	525	18	32	501	44	30	396	101	78
Local Mesh 6	458	64	53	488	48	39	484	43	48	367	79	129	406	90	79
Cubic Approximation															
Global Regular	514	24	37												
Global Random	505	22	48												
Local Mesh 1	500	25	50	500	28	47	505	33	37	501	34	40	492	33	50
Local Mesh 2	507	26	42	503	27	45	503	34	38	512	26	37	509	30	36
Local Mesh 3	522	22	31	497	32	46	483	39	53	490	35	50	481	42	52
Local Mesh 4	504	27	44	506	26	43	490	37	48	507	25	43	503	33	39
Local Mesh 5	519	24	32	494	33	48	482	43	50	499	33	43	485	36	54
Local Mesh 6	491	35	49	492	34	49	502	30	43	514	27	34	504	29	42
Degree 4 Approximation															
Global Regular	511	25	39												
Global Random	495	20	60												
Local Mesh 1	525	12	38	525	11	39	530	6	39	525	10	40	519	12	44
Local Mesh 2	483	28	64	513	8	54	508	13	54	480	24	71	471	36	68
Local Mesh 3	513	13	49	529	11	35	450	43	82	450	35	90	516	16	43
Local Mesh 4	500	17	58	527	15	33	524	8	43	505	17	53	493	22	60
Local Mesh 5	507	25	43	500	21	54	433	32	110	428	36	111	450	35	90
Local Mesh 6	501	24	50	519	7	49	499	20	56	511	18	46	496	15	64
Degree 5 Approximation															
Global Regular	523	4	48												
Global Random	518	3	54												
Local Mesh 1	527	6	42	525	8	42	530	7	38	530	5	40	519	12	44
Local Mesh 2	529	7	39	529	5	41	531	7	37	537	2	36	533	4	38
Local Mesh 3	521	12	42	536	2	37	536	3	36	537	5	33	528	7	40
Local Mesh 4	528	10	37	538	7	30	529	6	40	533	3	39	518	11	46
Local Mesh 5	523	11	41	532	3	40	538	2	35	536	5	34	522	8	45
Local Mesh 6	533	3	39	526	6	43	524	8	43	529	4	42	524	7	44

TABLE 3. Degree Errors for 575 Points on Test Surface — Local Equal Edge Length Enforced

Mesh Type	Rotation 1			Rotation 2			Rotation 3			Rotation 4			Rotation 5		
Degree Error	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6	0-3	3-6	>6
Normal Curvature Method															
Global Regular	498	56	21												
Global Random	376	89	110												
Local Mesh 1	518	25	32	518	26	31	533	18	24	533	18	24	525	20	30
Local Mesh 2	321	103	151	417	55	103	350	112	113	321	86	168	330	74	171
Local Mesh 3	243	135	197	308	181	86	383	90	102	502	35	38	362	94	119
Local Mesh 4	318	117	140	432	68	75	439	70	66	472	48	55	315	89	171
Local Mesh 5	324	163	88	442	73	60	522	26	27	495	44	36	378	119	78
Local Mesh 6	469	52	54	498	39	38	491	38	46	363	92	120	402	85	88
Degree 2 Approximation															
Global Regular	501	54	20												
Global Random	374	88	113												
Local Mesh 1	518	25	32	518	25	32	532	19	24	532	19	24	520	24	31
Local Mesh 2	317	106	152	418	54	103	347	115	113	320	86	169	328	75	172
Local Mesh 3	240	135	200	302	187	86	381	92	102	501	36	38	362	96	117
Local Mesh 4	320	121	134	432	68	75	438	69	68	471	47	57	315	86	174
Local Mesh 5	320	163	92	442	71	62	522	26	27	495	44	36	382	111	82
Local Mesh 6	458	61	56	495	41	39	488	39	48	359	91	125	402	88	85
Degree 3 Approximation															
Global Regular	514	24	37												
Global Random	491	30	54												
Local Mesh 1	501	32	42	498	33	44	506	32	37	502	34	39	501	33	41
Local Mesh 2	503	35	37	508	32	35	501	30	44	509	33	33	510	32	33
Local Mesh 3	517	28	30	496	38	41	485	43	47	489	35	51	484	45	46
Local Mesh 4	500	35	40	506	35	34	496	38	41	507	34	34	506	30	39
Local Mesh 5	513	33	29	496	41	38	485	42	48	492	36	47	485	45	45
Local Mesh 6	492	36	47	491	39	45	498	34	43	513	30	32	504	32	39
Degree 4 Approximation															
Global Regular	511	25	39												
Global Random	479	24	72												
Local Mesh 1	505	14	56	494	23	58	500	19	56	496	22	57	493	22	60
Local Mesh 2	485	32	58	514	23	38	506	18	51	479	30	66	469	41	65
Local Mesh 3	524	9	42	526	16	33	454	41	80	444	39	92	514	18	43
Local Mesh 4	512	15	48	530	16	29	533	10	32	509	18	48	495	25	55
Local Mesh 5	519	18	38	503	20	52	435	34	106	421	33	121	454	39	82
Local Mesh 6	508	19	48	525	7	43	501	18	56	526	13	36	496	30	49
Degree 5 Approximation															
Global Regular	523	4	48												
Global Random	505	9	61												
Local Mesh 1	530	14	31	520	13	42	533	7	35	532	2	41	525	14	36
Local Mesh 2	533	2	40	543	4	28	534	10	31	537	1	37	540	3	32
Local Mesh 3	530	10	35	537	5	33	540	3	32	540	3	32	532	5	38
Local Mesh 4	535	5	35	544	9	22	531	11	33	539	9	27	521	12	42
Local Mesh 5	528	12	35	536	5	34	539	1	35	539	2	34	531	5	39
Local Mesh 6	534	4	37	528	8	39	525	11	39	533	3	39	529	4	42

TABLE 4. Degree Errors for 575 Points on Test Surface — Global Equal Length Edges

	Length = 0.05			Length = 0.10		
Degree Error	0-3	3-6	>6	0-3	3-6	>6
	Normal Curvature Method					
Local Mesh 1	569	5	1	489	29	57
Local Mesh 2	397	111	67	267	105	203
Local Mesh 3	345	98	132	145	177	253
Local Mesh 4	408	119	48	246	124	205
Local Mesh 5	457	92	26	197	223	155
Local Mesh 6	538	24	13	418	71	86
	Degree 2 Approximation					
Local Mesh 1	569	5	1	486	32	57
Local Mesh 2	396	112	67	266	103	206
Local Mesh 3	347	96	132	138	186	251
Local Mesh 4	407	120	48	250	122	203
Local Mesh 5	456	93	26	184	230	161
Local Mesh 6	538	24	13	404	84	87
	Degree 3 Approximation					
Local Mesh 1	570	0	5	473	34	68
Local Mesh 2	571	0	4	475	35	65
Local Mesh 3	571	1	3	497	27	51
Local Mesh 4	571	0	4	470	40	65
Local Mesh 5	571	1	3	488	32	55
Local Mesh 6	568	1	6	462	40	73
	Degree 4 Approximation					
Local Mesh 1	574	0	1	463	27	85
Local Mesh 2	574	0	1	438	47	90
Local Mesh 3	575	0	0	499	21	55
Local Mesh 4	574	0	1	470	28	77
Local Mesh 5	574	0	1	497	27	51
Local Mesh 6	572	0	3	484	29	62
	Degree 5 Approximation					
Local Mesh 1	574	0	1	509	10	56
Local Mesh 2	574	0	1	515	6	54
Local Mesh 3	574	0	1	519	7	49
Local Mesh 4	574	0	1	518	6	51
Local Mesh 5	574	0	1	513	10	52
Local Mesh 6	572	0	3	514	4	57

TABLE 5. Degree Errors for 575 Points Separated by Principal Curvature Difference of 0.2 on Test Surface

Mesh Type	Rotation 1		
Degree Error	0-3	3-6	>6
	Normal Curvature Method		
Global Regular	464(34)	33(23)	18(3)
Global Random	337(37)	71(13)	103(14)
Local Mesh 1	460(54)	25(3)	26(7)
Local Mesh 2	317(10)	64(37)	130(17)
Local Mesh 3	232(20)	116(12)	163(32)
Local Mesh 4	297(19)	98(26)	116(19)
Local Mesh 5	300(24)	130(23)	81(17)
Local Mesh 6	425(41)	46(12)	40(11)
	Degree 2 Approximation		
Global Regular	466(35)	32(22)	17(3)
Global Random	335(37)	74(13)	102(14)
Local Mesh 1	458(54)	27(3)	26(7)
Local Mesh 2	312(10)	67(37)	132(17)
Local Mesh 3	223(20)	122(12)	166(32)
Local Mesh 4	297(19)	104(26)	110(19)
Local Mesh 5	296(24)	135(22)	80(18)
Local Mesh 6	417(41)	52(12)	42(11)
	Degree 3 Approximation		
Global Regular	458(56)	24(0)	33(4)
Global Random	448(57)	19(3)	44(4)
Local Mesh 1	445(55)	23(2)	43(7)
Local Mesh 2	453(54)	22(4)	36(6)
Local Mesh 3	467(55)	18(4)	26(5)
Local Mesh 4	450(54)	23(4)	38(6)
Local Mesh 5	464(55)	20(4)	27(5)
Local Mesh 6	436(55)	35(0)	40(9)
	Degree 4 Approximation		
Global Regular	454(57)	24(1)	37(2)
Global Random	437(58)	18(2)	56(4)
Local Mesh 1	467(58)	10(2)	34(4)
Local Mesh 2	426(57)	27(1)	58(6)
Local Mesh 3	455(58)	12(1)	44(5)
Local Mesh 4	441(59)	16(1)	54(4)
Local Mesh 5	451(56)	21(4)	39(4)
Local Mesh 6	447(54)	20(4)	44(6)
	Degree 5 Approximation		
Global Regular	463(60)	4(0)	48(0)
Global Random	457(61)	3(0)	51(3)
Local Mesh 1	467(60)	6(0)	38(4)
Local Mesh 2	470(59)	7(0)	34(5)
Local Mesh 3	461(60)	11(1)	39(3)
Local Mesh 4	468(60)	10(0)	33(4)
Local Mesh 5	462(61)	11(0)	38(3)
Local Mesh 6	473(60)	3(0)	35(4)

TABLE 6. Components of the Principal Direction Errors - Median Values

Normal Curvature Method

	Error	Num.	Denom.	$\|\|Y_2\|\|$	$\|\|Y_1Y_3\|\|$	$\|\|d\|\|$	θdY_2	θdY_1Y_3
Global Regular	0.5816	0.0242	1.5943	0.8866	1.0961	0.2871	85.7942	85.9385
Global Random	1.6724	0.0826	1.5751	0.7080	1.2248	0.3196	76.7223	74.3489
Local Equi-angle	0.1918	0.0167	1.7288	0.5774	1.1547	0.3040	87.6794	86.1888
Local 30-90	1.8221	0.0837	1.6099	0.5773	1.1547	0.3138	71.1061	69.5400
Local Random 1	1.9897	0.0948	1.6284	1.2098	1.9767	0.2728	76.3860	78.1628
Local Random 2	1.5147	0.0737	1.6646	0.6342	1.2462	0.3057	75.1857	68.6009
Local Random 3	1.4092	0.0798	1.7039	1.0117	1.6519	0.2725	80.3466	81.3653
Local Random 4	0.9644	0.0460	1.6908	0.5838	1.2010	0.3124	80.7168	76.7227

Cubic Method

	Error	Num.	Denom.	$\|\|Y_2\|\|$	$\|\|Y_1Y_3\|\|$	$\|\|d\|\|$	θdY_2	θdY_1Y_3
Global Regular	0.1647	0.0127	1.8344	0.0153	0.0303	2.5361	75.7809	63.8826
Global Random	0.1990	0.0152	1.7970	0.0160	0.0304	2.8937	77.9822	70.3364
Local Equi-angle	0.0906	0.0087	1.7875	0.0149	0.0298	2.8991	84.8990	66.5182
Local 30-90	0.1131	0.0102	1.7881	0.0157	0.0315	2.8181	83.1192	70.1950
Local Random 1	0.1311	0.0133	1.8100	0.0153	0.0306	2.6616	80.1992	67.0239
Local Random 2	0.1340	0.0120	1.7754	0.0156	0.0312	2.7878	81.2036	68.9842
Local Random 3	0.1328	0.0133	1.8103	0.0153	0.0305	2.6916	80.2445	66.7793
Local Random 4	0.1267	0.0115	1.7982	0.0155	0.0310	2.8677	82.4222	68.9718

REFERENCES

1. Xin Chen and Francis Schmitt, *Intrinsic Surface Properties from Surface Triangulation*, Proceedings, European Conference on Computer Vision, May 1992.
2. Mathieu Desbrun, Mark Meyer, Peter Schroder,and Alan H. Barr, *Discrete Differential Geometry Operators in nD*, preprint, July 22, 2000.
3. Bruce A. Duncan and Arthur J. Olsen, *Shape Analysis of Protein Surfaces*, Journal of Molecular Graphics, Vol. 10, No. 1, 1992.
4. Patrick J. Flynn and Anil K. Jahn, *On Reliable Curvature Estimation*, IEEE Computer Vision and Pattern Recognition, pp. 110-116, 1989.
5. Ahna Girshick, Victoria Interrante, Steven Haker and Todd Lemoine, *Line Direction Matters: an argument for the use of principal directions in line drawings*, First International Symposium on Non-Photorealistic Rendering and Animation, Annecy, 2000.
6. Gabriele Gorla, Victoria Interrante and Guillermo Sapiro, *Growing Fitted Textures on Surfaces*, IMA preprint, February 2001.

7. Stefanie Hahmann, *Visualization Techniques for Surface Analysis*, in Advanced Visualization Techniques, Chandrajit Bajaj, ed., John WIley, 1999.

8. Bernd Hamann, *Curvature Approximation of 3D manifolds in 4D space*, Computer Aided Geometric Design, 11, pp. 621-632, 1994.

9. Aaron Hertzmann and Denis Zorin, *Illustrating Smooth Surfaces*, Proceedings of ACM SIGGRAPH 2000, pp. 517-526.

10. Victoria Interrante, *Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution*, Proceedings of ACM SIGGRAPH 97, pp. 109-116.

11. T. Maekawa, F.-E. Wolter and N. M. Patrikalakis, *Umbilics and Lines of Curvature for Shape Interrogation*, Computer-Aided Geometric Design, Vol. 13, No. 2, pp. 133-161, 1996.

12. Phillippe Samson and Jean-Laurent Mallet, *Curvature Analysis of Triangulated Surfaces in Structural Geology*, Mathematical Geology, Vol. 29, No. 3, pp. 391-412, 1997.

13. Gabriel Taubin, *Estimating the Tensor of Curvature of a Surface from a Polyhedral Approximation*, Proc. 5th Intl Conf. on Computer Vision (ICCV'95), pp. 902-907,June 1995.

Texture Synthesis for 3D Shape Representation

Gabriele Gorla[†], Victoria Interrante[‡] and Guillermo Sapiro[†]
[†]*Electrical and Computer Engineering,* [‡]*Computer Science and Engineering*
University of Minnesota

Abstract

Considerable evidence suggests that a viewer's perception of the 3D shape of a polygonally-defined object can be significantly affected (either masked or enhanced) by the presence of a surface texture pattern. However investigations into the specific mechanisms of texture's effect on shape perception are still ongoing and the question of how to design and apply a texture pattern to a surface in order to best facilitate shape perception remains open. Recently, we have suggested that for anisotropic texture patterns, the accuracy of shape judgments may be significantly affected by the orientation of the surface texture pattern anisotropy with respect to the principal directions of curvature over the surface. However it has been difficult, until this time, to conduct controlled studies specifically investigating the effect of texture orientation on shape perception because there has been no simple and reliable method for texturing an arbitrary doubly curved surface with a specified input pattern such that the dominant orientation of the pattern everywhere follows a pre-defined directional vector field over the surface, while seams and projective distortion of the pattern are avoided. In this paper, we present a straightforward and highly efficient method for achieving such a texture and describe how it can potentially be used to enhance shape representation. Specifically, we describe a novel, efficient, automatic algorithm for seamlessly synthesizing, from a sample 2D pattern, a high resolution fitted surface texture in which the dominant orientation of the pattern locally follows a specified vector field over the surface *at a per-pixel level*, and in which seams, projective distortion, and repetition artifacts in the texture pattern are nearly completely avoided. We demonstrate the robustness of our method with a variety of texture swatches applied to standard graphics datasets, and we explain how our method can be used to facilitate research in the perception of shape from texture.

CR categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Texture.
Keywords: Texture synthesis, texture mapping, shape perception, shape representation.

1. Introduction

Adding texture to the surface of a polygonal model can not only profoundly enhance its visual richness, but can also significantly affect our perception of the object's geometry. Certain textures have been shown to impede accurate shape perception, for example by masking faceting artifacts [11]. Others may have the potential to enhance shape perception by emphasizing the lines of curvature of the form [19]. In computer graphics and visualization, where we have the ability to model a textured object in any way that we desire – to both define the texture pattern and define how it is applied over the surface – we have the potential to use texture in highly controlled ways to influence shape perception. Unfortunately, the existing theories on shape perception from texture do not provide sufficient guidance to answer the question of how to best design and apply a texture to a surface in order to facilitate the accurate understanding of its shape.

Researchers in perceptual psychology have been investigating the effects of various texture pattern characteristics on surface shape perception for many years through controlled observer experiments [6, 22, 24, 26, 36]. Unfortunately the scope of these studies has been limited by the capabilities of available rendering and texture-mapping utilities and algorithms. In particular it has not been possible, in general, to map an arbitrary pattern onto an arbitrary doubly curved surface so that the orientation of the pattern everywhere follows a specific predefined vector field at a per-pixel level, while minimizing any distortion of the underlying pattern and avoiding the introduction of pattern discontinuities. Hence most of the studies conducted to date have either not considered the effect of the orientation of the texture pattern with respect to the surface curvature [6, 36], or have been limited to highly restricted synthetic texture

patterns, such as isolated pairs of line segments [26], or have been limited to highly restricted surface geometries, such as the case of singly-curved surfaces [22, 24]. As a result, our ability to gain deeper insight into the specific impact of texture pattern orientation on surface shape perception, and to use this insight to inform theories of shape perception from texture, has been limited. In particular, it is not yet clear to what extent the introduction of texture pattern anisotropy *per se* interferes with surface shape perception [6], or to what extent it is sufficient for unimpeded shape perception to ensure only that the texture pattern does not turn in the surface (i.e. that it does not contain significant geodesic curvature) [22].

In computer graphics and visualization, many feel that the importance to shape perception of the particular alignment over the surface of an anisotropic texture pattern remains open to debate. However we have come to believe, based on informal observations of numerous surfaces under numerous texture conditions, that one's ability to make accurate judgments about the shape of an underlying surface can be significantly influenced both by the characteristics of the texture pattern itself *and* the way in which the pattern is laid down over the surface. Clearly, in order to objectively assess the impact of texture orientation on surface shape perception, it is necessary to conduct further controlled, quantitative experiments. In a recent study using line integral convolution based texture [20], we found indications that observers' shape judgments of a doubly-curved surface are more accurate in the presence of a purely anisotropic texture that follows the first principal directions of curvature over the surface (a special instance of a pattern with zero geodesic curvature) than in the presence of either a purely anisotropic texture that exhibits zero geodesic curvature but does not follow one of the principal directions, or a pattern that contains significant non-zero geodesic curvature (figure 1) [20]. However we found no indications that shape perception is significantly better in the presence of a purely anisotropic principal direction oriented texture pattern than it is in the control case of the purely isotropic texture pattern.

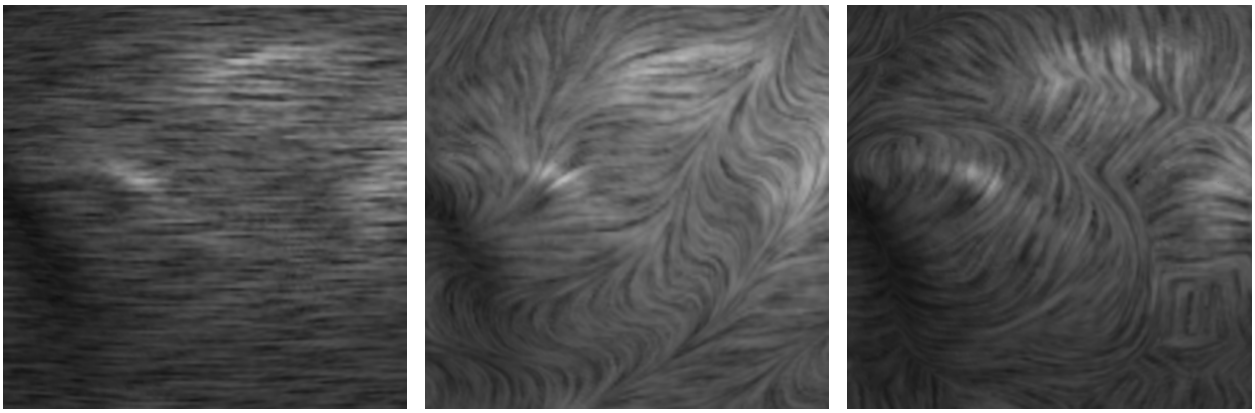


Figure 1: Sample close-up images in an experiment examining the effect on shape perception of differently oriented anisotropic texture patterns synthesized via line integral convolution. Left: a uniformly oriented texture with zero geodesic curvature, Center: a texture with non-zero geodesic curvature; Right: a texture with nearly zero geodesic curvature that follows the first principal direction over the surface. The underlying surface shape is identical in all three cases.

In order to further investigate the key question of how we might best both define and apply a texture pattern to facilitate surface shape perception, we will have to conduct additional studies using a wider variety of surface texture patterns. This requires an algorithm for texturing an arbitrary doubly curved surface with an arbitrary 2D pattern such that the dominant direction of the pattern follows a specified vector field over the surface at a per-pixel level. In this paper we describe such an algorithm that we developed for this purpose. Our algorithm is very straightforward, easy to implement, and highly efficient, and has the potential to be useful for a wide variety of graphics applications that require the aesthetic mapping of a given texture pattern to a given surface, independent of the desire to effectively portray the surface shape.

Given a 2D texture pattern and a polygonal surface model, the historical challenge has been to determine how to apply the pattern to the surface in a manner that minimizies the visual impact of seams and projective distortion while orienting the pattern so that it flows over the shape in a desirable way.

Many different approaches to this basic problem are possible and, concurrently with our work, many similar approaches have been developed. The key distinguishing features of the method that we describe in this paper stem from the fact that it was specifically developed for the purposes of shape representation, in which curvature information is carried by a high resolution texture pattern. Our method has the advantages of being highly efficient for large quantities of texture, very straightforward to implement, and producing high quality results across a wide variety of texture types and models. For the purposes of shape-from-texture investigations, our fitted texture approach is superior to our previous 3D line integral convolution approach [20] because the resulting textured objects can be easily displayed at interactive frame rates using a conventional renderer on a standard PC with texture mapping hardware.

Our technique consists of the following main steps:

- Partition the polygons of the model into contiguous patches, as nearly planar (to prevent distortion) and as nearly similarly sized (to simplify texture map handling) as reasonably possible.
- Compute a vector field over the object, or read a pre-defined field from a file.
- Synthesize the texture pattern over each patch, maintaining pattern continuity across the boundaries with neighboring patches, using an efficient, orientation-adaptive variation of the non-parametric sampling method proposed by Efros and Leung [9].

An example of the results of our algorithm is presented in figure 2.

2. Previous Work

A variety of methods have been previously proposed for texturing polygonal models with patterns that are as free as possible of seams and distortion artifacts.

One method is solid texturing [29, 30, 42], in which the texture pattern is defined over a 3D volume. Particularly good results have been achieved with this method for water, as well as for objects made of wood and stone. However, there are significant challenges in synthesizing 3D textures modeled after sampled materials [17, 8], and current methods for creating custom-fitted 3D textures whose features follow a surface's shape [19] are severely limited in scope and applicability.

Methods for applying 2D image-based texture to arbitrary polygonal models for the most part must balance the inherent trade-off between seams and distortion (one cannot in general apply a 2D image to a non-developable surface without incurring one or the other), employing piecewise flattening in the case of arbitrary parametric [3] or polygonally-defined [25] models, or using careful surface parameterization [23], or pre-distortion of the texture [1, 41] to achieve desired results in other particular cases. Conformal mapping [15] offers a global solution that preserves angles, but not lengths or areas.

Closer to our objectives, Neyret and Cani [28] proposed an excellent technique for achieving seamless and virtually distortion-free mapping of 2D isotropic texture patterns on arbitrary objects via custom-defined triangular texture tiles that are continuous with one another across various of their boundaries. Unfortunately an extension of this method to anisotropic texture patterns is not obvious. Praun et al. [33] subsequently proposed "lapped textures", which provides capabilities that are the most similar to those towards which our method aspires, although the approach that we take is very different. The lapped texture method repeatedly pastes copies of a sample texture swatch onto overlapping patches across a surface after some subtle warping and reorientation to align the pattern with a user-defined vector field. This method produces very good results when used with texture patterns that contain enough high frequency detail and natural irregularity in feature element sizes and relative positions. This is needed to perceptually mask artifacts due to the partial overlap or misalignment of feature elements across patch boundaries. As currently formulated, the lapped texture approach is not particularly well-suited for rigidly structured patterns, such as a checkerboard, or textures such as netting, which are characterized by the global continuity of specific elongated elements. It is also less well-suited for use with vector fields that contain significant high frequency variation. In addition, the lapped texturing process as described in [33] involves considerable amounts of user interaction.

Figure 2: An example of a synthesized surface texture produced by our method. No manual intervention of any kind was employed. This texture was grown from an original 92x92 swatch [5], pre-rotated to 63 orientations each cropped to 64x64 pixels, to cover 291 surface patches at 128x128 resolution following a vector field locally defined by the projection of (0,1,0) onto the tangent plane at each point. The entire process required approximately 12 minutes.

The method that we describe in this paper provides capabilities beyond those offered by these previous methods. It achieves nearly seamless and distortion-free texturing of arbitrary polygonally-defined models with a texture pattern derived from a provided sample. Most importantly, the method is suitable for use with anisotropic patterns. It generally preserves larger scale texture pattern continuity across patch boundaries, does not require manual user intervention, and allows the orientation of the applied pattern to locally follow a specified vector field on a per-pixel basis.

Our method falls into the category of methods that achieve texture pattern continuity without distortion by in effect synthesizing the texture "in place" over the surface of the object. Previous methods in this category include direct painting [16], and reaction-diffusion texture synthesis [37, 41], which yield excellent results for hand-crafted textures and textures modeled after organic processes. Our goal was to achieve similarly good results with automatically synthesized textures that are perceptually equivalent to a given sample swatch.

Our work is perhaps most fundamentally motivated by the impressive advances in texture synthesis methods [17, 7, 32, 44, 9, 39] which have made it possible to create, for an increasingly wide range of patterns, unlimited quantities of a texture that is perceptually equivalent to a small provided sample.

Leveraging research in human texture perception, Heeger and Bergen [17] developed a highly successful method for synthesizing textures that capture the essential perceptual properties of a variety of homogeneous stochastic sample patterns. Their method works by iteratively modifying a random noise image so that its intensity histogram matches the histogram of the sample texture across each of the subbands in a steerable pyramid representation of each image. De Bonet [7] developed a related method based on interchanging elements in the Laplacian pyramid representation of a self-tiling pattern where possible, while preserving the joint-occurrence relationships of features across multiple resolutions. This method yields impressive results for an even wider variety of patterns, though some difficulties remain in preserving larger scale globally significant structure. Several other highly sophisticated texture analysis/synthesis approaches have been subsequently developed [9, 32, 44]. Of these, we chose to follow the texture synthesis approach proposed by Efros and Leung [9] because of its combination of simplicity and quality of output. In this method, a new texture pattern is grown, pixel-by-pixel, by sampling into a provided template pattern and choosing randomly from among the pixels whose neighborhoods are close matches to the yet partially-defined neighborhood of the pixel to be filled in, in the pattern being synthesized.

Subsequent to the appearance of the original Efros and Leung paper, and concurrently with the development of our method, a number of new advancements in 2D and 3D texture synthesis have been achieved. Wei and Levoy [39] proposed a method that addressed one of the most serious concerns with the method of [9] which was speed. Their method used tree-structured vector quantization to improve, by several orders of magnitude, the speed of the search for the pixel with the best matching neighborhood, at the cost of some loss of quality in the resulting synthesized patterns. Ashikmin [2] proposed a method, based on the cutting and pasting of larger areas than a single pixel from the sample texture, that achieved improved results for highly structured textures such as flowers and leaves. Efros and Freeman [10] proposed a new method, called 'texture quilting', that produces even more consistently excellent results across a broad spectrum of texture patterns by specifically maintaining both continuity and coherence across broader local regions of the pattern. Most similar to the objectives of our work, Wei and Levoy [40], Turk [38], and Ying *et al.* [43] all proposed methods for synthesizing a sample 2D texture pattern over an arbitrary mesh in 3D, with the objective of resolving the classical texture mapping problem: to avoid seams and to minimize pattern distortion. The results produced by our method are very similar in many respects to the results produced by these concurrently developed methods, with some subtle differences that will be discussed in a later section. The distinguishing characteristic of our method is that it is optimized for the synthesis of large quantities of *high resolution* texture in which the direction of the texture pattern follows a specified vector field, such as the direction of greatest normal curvature, over the surface at a per *pixel* level, in order to be useful for applications in which one is specifically concerned with the use of texture for facilitating shape representation.

In the remainder of this paper we describe the method that we have developed and the details of its implementation, talk about some of the issues that arise in automatically determining a good way to orient

a texture pattern over a surface, show representative results, and conclude with a discussion of the current limitations of our implementation and directions for future work.

3. Proposed Method

Our proposed method is basically a two step process. First the surface is partitioned into small, almost flat patches. Then, the texture is grown over the planar projection of each individual patch taking into consideration the proper boundary conditions to maintain the continuity of the texture pattern across seams at the patch boundaries. During the synthesis of an anisotropic pattern, the texture is locally constrained into alignment with a specified vector field over the surface. For simplicity, we store the resulting synthesized texture as a collection of separate small images for each patch, although other approaches are certainly possible.

3.1 Partitioning

The goal of the first stage is to partition the mesh into a minimum number of approximately planar patches (collections of triangles). Obviously these are conflicting goals for any closed surface and a suitable tradeoff must be found. To keep our implementation as simple as possible, we restrict patches to be of approximately the same size. Maintaining relatively consistent patch sizes simplifies texture memory management by allowing us to allocate and synthesize texture maps of a consistent fixed resolution for each patch. Please note that mesh partitioning is a task that is common to many computer graphics algorithms and many approaches have been previously described [cf. 34, 27]. In this section we describe for the sake of completeness the details of the particular approach that we used.

Two input parameters define the maximum patch size (which influences the scale at which the texture appears over the surface) and the maximum projective distortion that the user is willing to tolerate. The initial partitioning is done with a greedy algorithm, after which an optimization step is performed to reduce the average projection error.

The process for the initial partitioning can be summarized by the following pseudo code:

```
while (unassigned_triangles > 0) {
    pick an arbitrary unassigned triangle T;
    assign T to a new group G;
    add to group G all connected triangles C that satisfy:
        - Normal(C) • Normal(T) > min_cosine_displacement;
        - distance from the center of C to the farthest vertex of T is less than max_dist ;}
```

The image on the left side of figure 3 shows a representative result after the first stage in the splitting. The green triangles are the reference triangles that define the plane onto which the patch will ultimately be flattened. It is easy to notice that some of the patches obtained at this point are very small and/or contain triangles that are relatively far from being aligned with the reference plane. A refinement pass is used to reduce both the number of patches and the number of triangles that are oriented at a sharp angle to the plane into which they will ultimately be projected. Two simple experimental rules are iteratively applied until no significant improvement is observed:

- remove a patch if it is very small, and its triangles can be added to a neighboring patch without violating the distance constraint;
- reassign a triangle T from patch P1 to a neighboring patch P2 if T borders P2 and is more closely aligned with the reference plane of that patch than with its own.

The image on the right side of figure 3 shows the results after iterative refinement. The simple refinement procedure that we use is not guaranteed to converge to the theoretically optimal result but we have found that the results are consistently good and quite sufficient for our purposes. Since both the splitting and optimization stages are of linear complexity and account, on average, for between 1 and 4% of the total computational time we did not feel the need to improve their speed. The greedy splitting step took about 2 seconds, and the refinement about 10 seconds for the 70,000 triangle data set shown. In the rare event that acceptable results are not achieved in this phase, the splitting process can be repeated using a tighter limit on the acceptable normal error (which will result in more, smaller patches). Increasing the

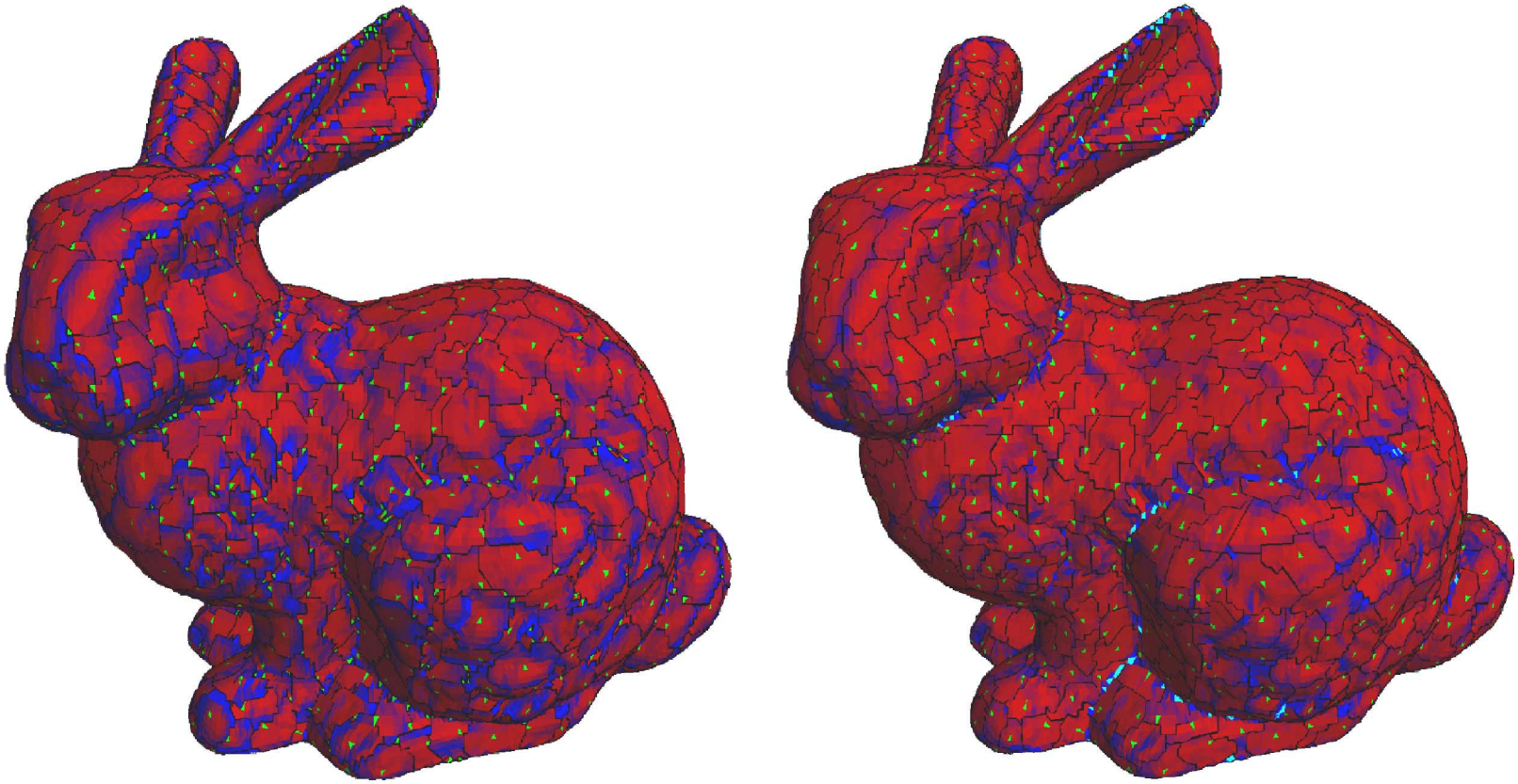


Figure 3: The partitioned bunny, after the first stage of our “greedy” splitting algorithm (left) and after iterative refinement (right), which decreased the number of patches from 988 to 699. Patch boundaries are outlined in black on each model. The green triangles are the seed triangles, which also define the reference plane for each patch. Triangles whose normal directions differ by less than 18 degrees from the direction of the normal to the reference plane, corresponding to a 5% error in the linear projection, are colored red. Triangles up to 25 degrees out of alignment with the reference plane for their patch, corresponding to a 10% error in the projection, are colored blue. Triangles rotated more than 25 degrees from the reference plane (>10% error) are colored cyan.

number of patches does not cause an increase in the computational expense in the subsequent texture synthesis step since the synthesis cost is only dependent on the total number of pixel synthesized. More significant is the issue that with very small patches comes an increased risk that the texture synthesis process will run into difficulties and "grow garbage", either due to the paucity of available contextual information along the shortened boundary, or to the near proximity of mutually incompatible pre-defined boundary conditions.

3.2 Parameterization

After the model has been partitioned into contiguous patches, the triangles comprising each patch are projected onto their common reference plane, and the texture coordinates are defined at each vertex according the coordinates of the projected vertices in the reference plane coordinate system. One major advantage of such a simple parameterization is that there is no need to store the texture coordinates with the output model as they can be easily recomputed at runtime. Adjacent triangles from the neighboring patches, which provide the boundary conditions for maintaining the continuity of the texture pattern during synthesis, are then rotated about their shared edges into the reference plane. We use rotation for these triangles rather than projection to minimize the projective distortion of the texture that we will need to refer to for reference purposes. However, it is necessary to check for the very infrequently encountered cases where it is not possible to rotate each of the adjacent triangles of a particular patch into the projection plane without causing some of these triangles to overlap. This entire process is illustrated in figure 4.

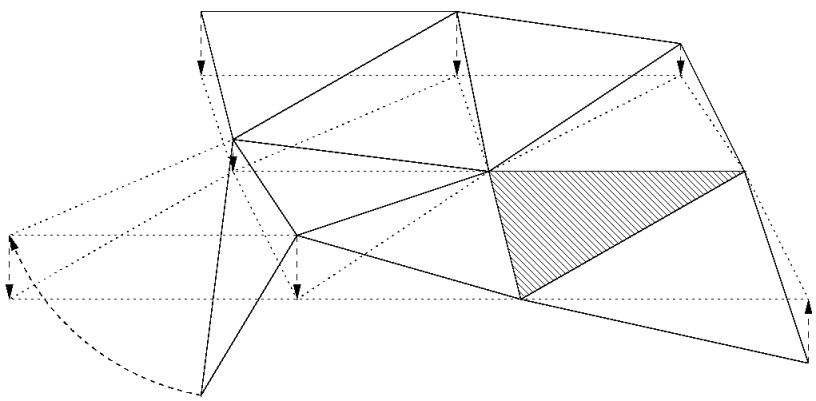


Figure 4: A diagrammatic illustration of the flattening process. The normal of the grey-shaded triangle defines the plane into which the remaining triangles in the patch are projected. The union of these triangles defines the area across which the texture will be synthesized. Neighboring triangles from the adjacent patches are rotated (to minimize projective distortion) into the plane so that any texture already present in these triangles can provide boundary conditions for the texture synthesis, in order to enable the achievement of a seamless final result.

3.3 Synthesis

At this point, we have created a 2D image for each patch containing:

- an area, defined by the projection of the triangles of the patch onto the reference plane, which contains the pixels to be filled by the synthesized texture; and
- an area, defined by the rotation into the reference plane of the neighboring triangles from the adjacent patches, that will hold any previously synthesized texture and provide the boundary conditions necessary to avoid seams due to discontinuities between texture element features in adjacent patches.

It is important to note that the partitioning process described in the immediately previous sections is completely independent from the synthesis algorithm. Any constrained synthesis method that can fill

arbitrary regions with arbitrary boundary conditions could potentially be used, although none of the existing algorithms we reviewed appeared to provide both the image quality and the speed required for this project. The work of Efros and Leung [9] came closest to meeting our needs and thus we elected to follow their general approach, which has been shown to produce good results for a wide range of texture patterns that can be modeled by Markov random fields (i.e. textures whose characteristics are fairly consistent under translation over the image and in which the value at any given point can be fully characterized by the values at its closely neighboring points over a limited range).

In order to make tractable the problem of efficiently synthesizing enough texture to cover a standard model of arbitrary topology at a reasonable resolution, the first objective of our proposed method is to achieve results that are of the same caliber as those demonstrated by Efros and Leung but that require significantly less time, while also preserving the flexible applicability of their approach. To do this, we use a new two-pass search strategy. The first pass, which is exhaustive, is done using a very small unweighted neighborhood (usually between 1/3 to 1/2 of the diameter of the full size neighborhood). The n best matches, where n is a user-definable parameter, are saved in a list to be processed by the second pass. This two pass approach presumes strong locality in the input textures (which holds true for many natural texture patterns) and has the effect of rapidly eliminating most of the uninteresting part of the search space. The size of this preselect list ultimately determines the overall speed of the synthesis algorithm. We found that some textures produced excellent results with preselect lists of as few elements as the number contained in one scanline of the original image; these we considered easy to synthesize. Others required 4 or even 8 times more elements in the preselect list, and these we considered hard to synthesize. Instances of easy and hard textures are shown in figure 2 and figure 9 respectively. In the second pass, each of the pixels in the preselect list is tested against the full size weighted neighborhood and the error metrics are updated. Among the best 10 or 10% of matches (whichever is greater) a random pixel is chosen and used in the synthesized image. Figure 5 shows a sample result of this synthesis algorithm in the 2D case. The speed of our method does not match the speed of Wei and Levoy's tree-structured vector quantization [39] for the synthesis of rectangular swatches of texture, but unlike their method it does not require the use of a fixed size causal neighborhood, and the results it produces are of consistently high quality. The proposed method is still fast enough to make feasible our goal of growing a fitted surface texture via the Markov random field sampling approach. Figure 6 illustrates the complete texture synthesis process. The image on the upper left represents the state of the model after the synthesis of the first two patches. The image in the upper center identifies the triangles comprising the third patch, with the triangle that defines the plane of the patch rendered in green, and the rest of the triangles rendered in red. The image on the upper right shows the results after the patch is filled with the synthesized texture. The image on the lower left shows the projection of the patch onto the plane. The image in the left center of the lower row shows the texture boundary conditions supplied by the neighboring triangles to this patch. The next two images in the lower row show what the patch looks like midway through and at the end of the texture synthesis step.

3.3.1 Isotropic Textures

As basically formulated, the approach we have just described can be used to cover the surface of a model with an isotropic texture pattern in an orientation-insensitive way. In other words, if we assume that our sample texture pattern is perfectly rotationally symmetric, we can directly use this approach, in its most basic form, to seamlessly synthesize the texture pattern across all of the patches in the model without any special considerations apart from the boundary conditions. However, as we quickly found, there are very few acquired textures that can be used with good results without regard to orientation. Even patterns which we initially believed to be isotropic based on inspection of the 2D sample image revealed unexpected orientation dependencies due to the subtle structuring that stems from the illumination process. Figure 7 illustrates this problem. Notice how the wool texture appears flat when applied without regard to orientation. This is due to the disruption of the pattern of shading.

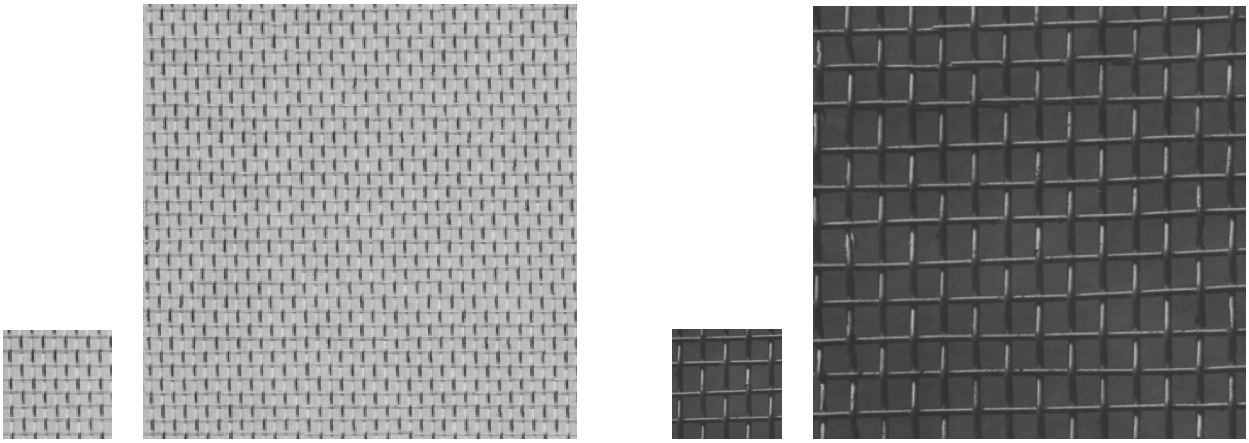


Figure 5: Left: An example of the results of our two-pass texture synthesis method using pattern D06 from the Brodatz album [5]. The speed of the synthesis approach varies from pattern to pattern depending on the sizes of the match-defining neighborhoods and the length of the preselect list. In this case we used a first pass neighborhood of 5x5, a maximum preselect list length of 64, and a second pass neighborhood of 12x12 to synthesize the 256x256 patch on the right from the 64x64 pixel sample on the left in about 73s. Right: an example using D01.

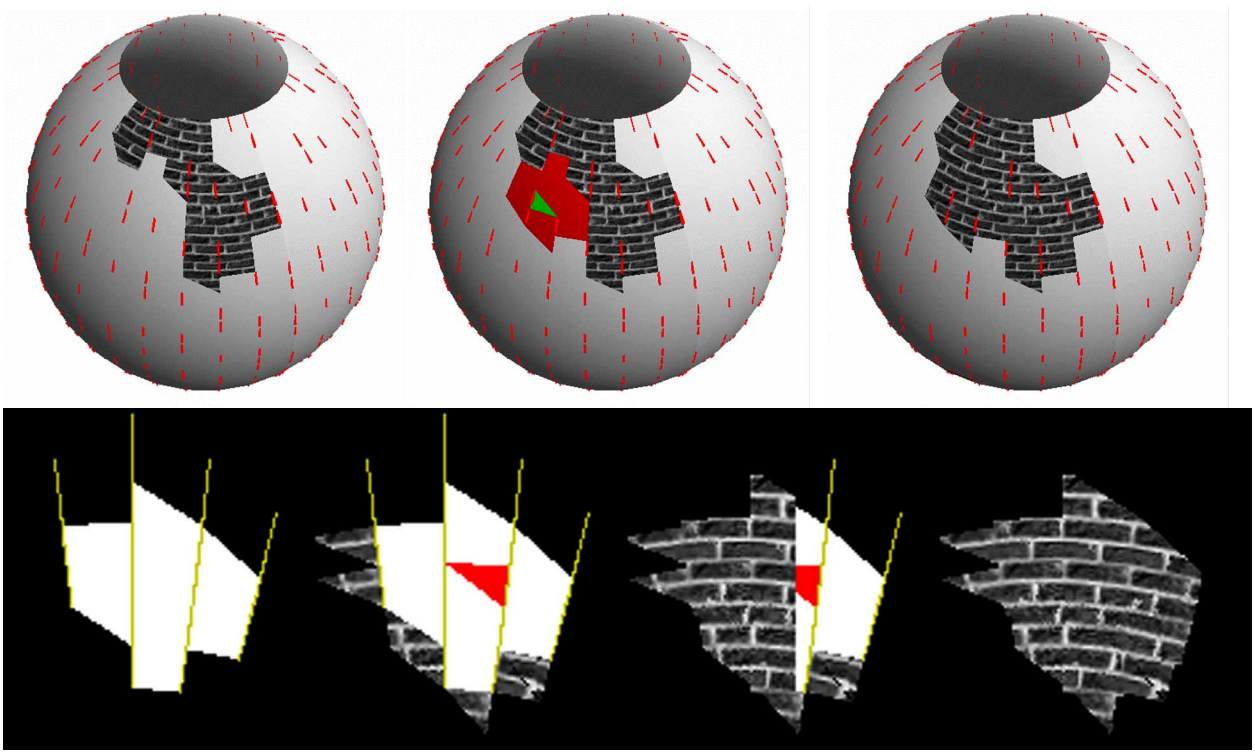


Figure 6: A step-by-step illustration of the basic process of our method. Upper row: the identification of the patch to be synthesized, and the synthesis result. Lower row (from left to right): the planar projection of the patch; the boundary conditions provided by the neighboring textured triangles rotated into the plane of the patch; midway through the texture synthesis process (synthesis is proceeding from left to right); the complete synthesized patch.

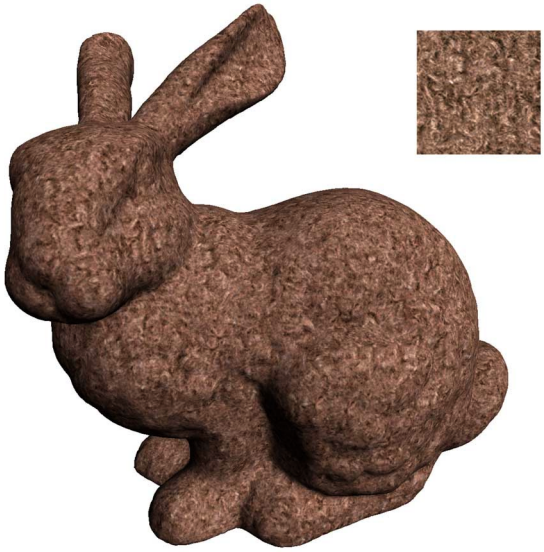


Figure 7: A texture ('wool.bw', from SGI) that originally appeared to be isotropic, reveals its anisotropic nature (due to the effects of shading) when synthesized over the Stanford bunny dataset via an approach in which the texture orientation is allowed to vary arbitrarily between each patch.

3.3.2 Directional Textures

In the vast majority of cases, it is necessary to control the orientation of the texture over the surface. For greatest flexibility, we allow a directional texture to follow any specified direction field. In the next section some examples will be discussed. We note that in the lapped textures method, Praun et al. [33] also align textures on a per patch basis, slightly distorting the parameterization to achieve good local continuity within the patch with the underlying directional specification. In the case of sparse triangulation, they reduce undersampling of the vector field by locally subdividing the mesh.

In our presented method the synthesis algorithm has been enhanced to allow per-pixel texture re-orientation. We pre-rotate the original texture into a quantized number of orientations, and during synthesis perform the search for best-matching neighborhoods in the pre-rotated image that is most closely aligned to the direction locally specified by the vector field. Figure 8 shows a sample of one

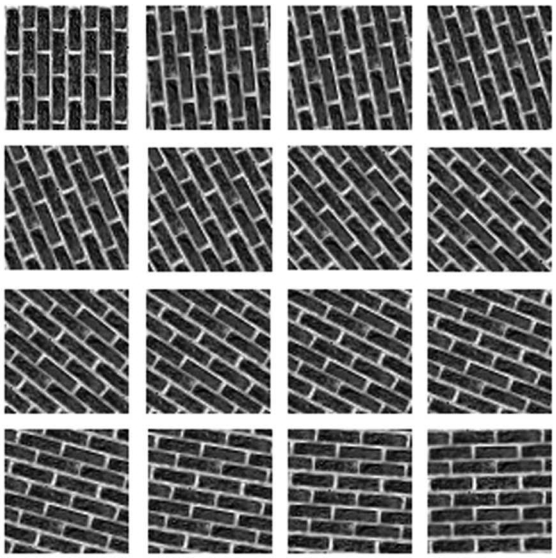


Figure 8: A quadrant of pre-rotated brick texture samples.

quadrant of pre-rotated brick texture. If the number of pre-rotated images is sufficiently high, the synthesized texture will follow the vector field smoothly. For the examples in this paper, we used between 64 to 128 different rotations of the input texture. Although it is of course possible to specify the use of any arbitrary number of pre-rotated images, we did not notice an appreciable increase in the quality of the results when finer quantizations were used.

Searching for matches in pre-rotated texture images allows considerably faster synthesis than would be possible if we had to perform the rotation on-the-fly for each pixel of the texture during synthesis. While it is relatively fast, with modern 3D hardware, to compute any arbitrary rotation of the original image, in most implementations there is a very high cost associated with reading the results from the frame buffer.

3.3.2.1 Constant Direction Fields

We originally began this work with the intent to explore the possibility of applying textures along the principal directions of curvature. Despite the latent potential in that approach, it is not without its difficulties, which we will discuss in greater detail in the following section. We quickly discovered that aesthetic results could be also achieved for a wide range of models using other, much simpler, vector field definitions. Notably, the field of "up" directions, locally projected onto the tangent plane at each point, appears to yield particularly nice results for many textures and datasets, as shown in figures 9 and 10. It is worth mentioning in the context of these two images that we worked hard to challenge our texture synthesis method, testing its performance on difficult texture patterns such as the crocodile skin, which contains potentially problematic sets of features spanning a wide range of spatial frequencies (from 3–21 pixels in diameter), and the square glass blocks pattern, which is a highly structured checkerboard-style design in which irregularities in the size, shape and/or positioning of any of the elements have the potential to stand out especially prominently.

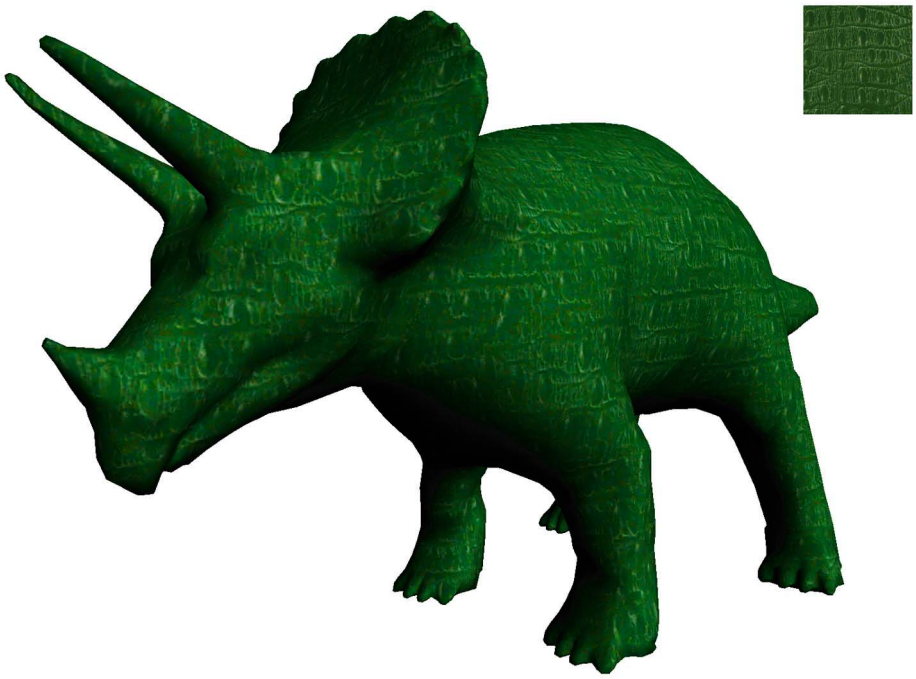


Figure 9: The crocodile skin texture (D10) synthesized over the triceratops model following the direction field (0,1,0) locally projected onto the tangent plane.

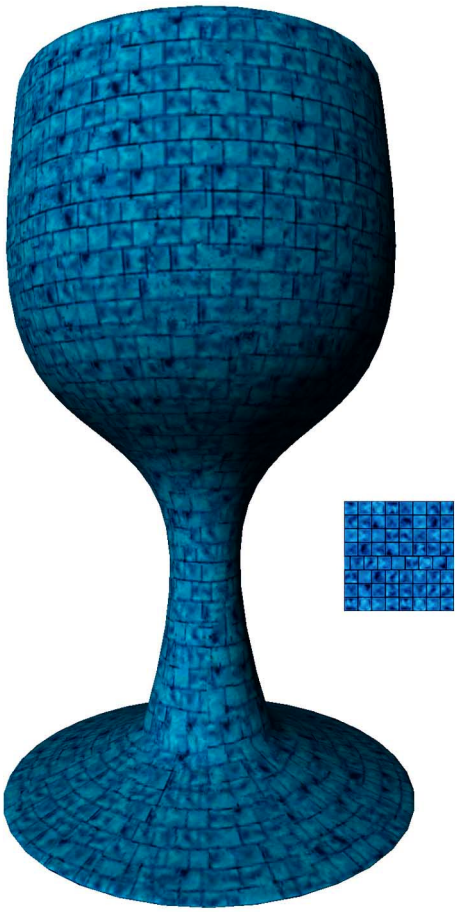


Figure 10: Glass block texture applied to a simple model with a constant directional field. Note the general preservation of continuity in the texture pattern and the relative consistency of the bricks' shapes and sizes, despite scattered artifacts. The direction field is locally given by the projection of the central axis of the object onto the tangent plane of each patch. Some of the patches in this model were resynthesized in a postprocess.

3.3.2.2 Principal Direction Fields

Although the constant direction field produces aesthetic results in many instances, there are also many cases for which it is not well suited. Specifically, it tends to fail for models that do not have a single well-defined intrinsic orientation, and it can not successfully emphasize local shape features. Of greatest intrinsic interest to our ongoing research is the possibility of applying an oriented texture pattern to the surface of an object such that it will be everywhere aligned with the principal directions of curvature.

Recent results in biological vision research support the idea that the principal directions play an important role in surface shape understanding, and we are interested in probing these ideas further through controlled studies of the effects of texture pattern orientation on observers' perception of the 3D shapes of complicated underlying models. Mammassian and Landy [26] have shown that observers' interpretations of line drawings of simple patches are consistent with an inherent bias, among other things, towards interpreting lines on objects as being oriented in the principal directions, supporting an observation made by Stevens [35] nearly 20 years ago. Li and Zaidi [24] examined observers' ability to estimate the relative curvatures of developable surfaces textured with various implicitly or explicitly plaid-like patterns, and concluded that shape perception depends critically upon the observation of changes in oriented energy along lines corresponding to the principal directions. However these ideas remain to be examined in the context of more complicated, arbitrary surfaces, where the first and second

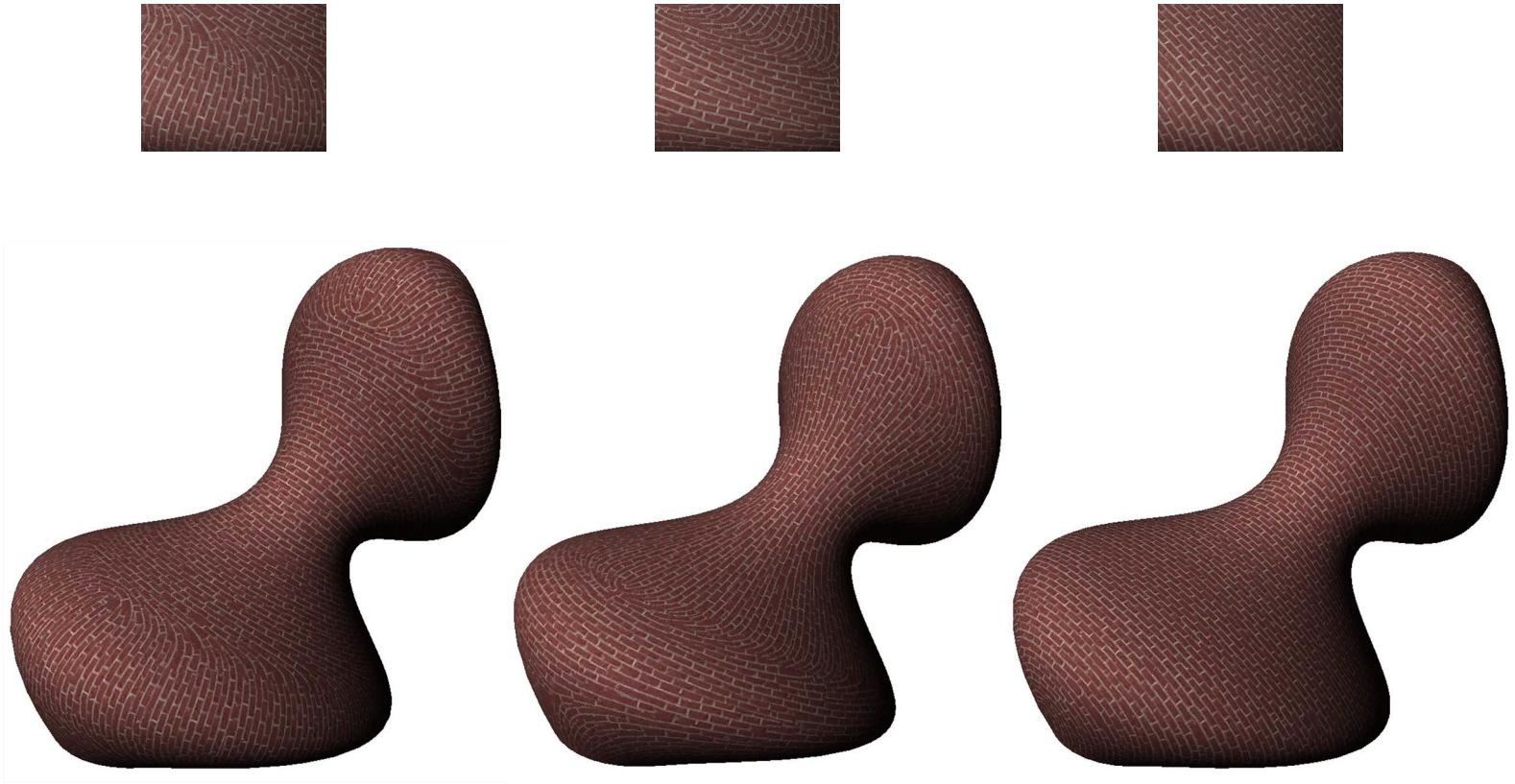


Figure 11: An illustration of the effect that the orientation of a directed pattern over a curved surface can have on our perception of the surface's 3D shape. On the left, the bricks are oriented in the direction of greatest signed normal curvature; in the middle they are oriented in the direction of least signed normal curvature, and on the right they are oriented in the same constant "up" direction used for the models in figures 9-10. Below the entire surface is shown, with the silhouette cues to shape available for reference.

principal directions can switch places numerous times. A significant challenge in this effort is to obtain accurate computations of the principal direction vector fields.

We recently worked with Dr. Jack Goldfeather to develop robust methods for computing smooth, accurate principal direction vector fields across arbitrary polygonally-defined objects [13]. A complementary approach developed by Bertalmio *et al.* [4] has the potential to facilitate the anisotropic smoothing of these fields. Our present results in applying an anisotropic texture over the surface of an object such that its dominant orientation is everywhere aligned with the first and second principal directions are shown in figure 11, and contrasted there with the results obtained using a constant "up" direction. Although it is clearly not possible to prove the benefit of a principal direction oriented texture for shape representation through one or even a handful of representative examples, with the existence of a method for synthesizing a variety of principal direction textures over arbitrary curving forms it becomes feasible to rigorously investigate the impact of various texture orientation strategies on the accuracy of shape perception judgments through controlled observer experiments. We are currently in the midst of carrying out a set of such studies and expect to report the results shortly [21].

4. Implementation

Our system is fully automatic, and does not require user interaction during either the splitting or texture synthesis process. The system has several parameters which can be adjusted by the user to increase the likelihood of obtaining optimal results with different kinds of textures or models. Within the splitting stage, these parameters include: an upper bound on the size of any single patch during splitting (which ultimately affects the scale of the texture on the model) and an upper bound on the angle that the normal of any member triangle can make with the reference direction for a patch (which affects the size and total number of patches). Within the texture synthesis stage, the user may first choose among several possible texture orientation options: to have the texture follow the direction of greatest or least signed or unsigned normal curvature, to follow the projection onto the local tangent plane of a constant specified direction, or to follow no specific direction (in which case the pattern is assumed to be invariant under rotation). With regard to texture synthesis, the user may also control: the sizes of the neighborhoods used in the each pass of the texture synthesis (larger neighborhoods generally increase the computational expense of the synthesis but are sometimes necessary in to preserve features across a range of different scales); the number of first-round preselected locations to be tested for a match on the second pass; and the weighting scheme used over the neighborhood during the matching process.

In most cases, it is sufficient to define the direction of texture synthesis across a patch according to the distribution pattern of previously textured pixels in the boundary region, under the assumption that starting from the side containing the greatest number of previously filled pixels will provide the most stable seed for the synthesis. Unfortunately, this is not always true (see the section on errors below). Certain strongly directional patterns seem to yield better results when the synthesis is performed following the direction of the vector field controlling the texture orientation. This approach was used also by Turk [38]. For quickly varying direction fields, starting the synthesis from an area of the patch in which the direction field is most calm seems to improve the quality of the result.

We use one of two different methods to determine the direction in which the synthesis proceeds from patch to patch. The simple method, which seems to works well on fairly uniform vector fields, is to begin with a randomly chosen patch and proceed to any blank connected patch, filling in any holes at the end. For principal direction vector fields better results can be achieved choosing the blank connected patch in which the difference between the two principal curvatures is greatest. This favors working first in areas over which the principal directions are clearly defined, providing a stable seed for the synthesis of the other patches.

The most computationally expensive part of the algorithm is the texture synthesis, which accounts for 96-99% of the running time. Partitioning and optimizing the patches takes only about 1-3 seconds for simple meshes such as the Venus and triceratops, up to 10-12 seconds for larger meshes such as the 70,000 triangle bunny.

The complexity of the synthesis algorithm is linear with respect to the number of pixels and practically independent from the number of polygons in the input mesh. Growing the weave texture on the Venus model in figure 2 took slightly less than 12 min. The goblet/glass block combination required about 20 minutes. These timings refer to a C++ implementation compiled with gcc[1] running on a standard linux (2.2.16) 933MHz Pentium III PC with a 32Mb GeForce 2 GTS. Table 1 provides a detailed comparison of the speeds of our method and other similar concurrently developed methods. Notice that although some of the other methods report faster execution times, they all involve the synthesis of significantly smaller amounts of texture. The relative efficiency of our method becomes clear when one normalizes for texture quantity.

	Our method	**Wei & Levoy [40]**	**Turk [38]**	**Ying et al. [43]**
Published time for texture synthesis, from a 64x64 sample	10m	82s (TSVQ) 695s (exaustive)	23m	10m (multires.) 3m (coherent)
Texels per vertices synthesized	1,000,000	25,000	256,000	400,000
Machine	P3-933MHz	P2-450MHz	R12K-360MHz	unknown
Estimated machine speed normalization factor	1	2	2.1	unknown
Estimated normalized texture synthesis rate	1600 pixels/s	600 vertices/s (TSVQ) 70 vertices/s (exaustive)	380 vertices/s	unknown

Table 1: A comparison of the speed of our method vs. similar concurrently developed methods.

4.1 Limitations

A major limitation of our method is its inability to capture low frequency texture features across several patches. Since our objective in developing this algorithm was to achieve a method for facilitating shape perception, we assumed that the scale of our desired texture pattern would be substantially finer than the scale of the shape features in our model. Thus our method is not as generic as the methods described in [40], [38] and [43]. However for high resolution textures our method provides substantial speedups over these algorithms.

As described in [8] and [10] all synthesis algorithms based on the "one-pixel-at-a-time" approach are susceptible to "catastrophic failure", in which the algorithm falls into the wrong part of the search space and "grows garbage". Our algorithm is no exception and will occasionally fail and produce undesirable results across all or part of a patch. Depending on the texture, we find that 0-3% of the patches typically contain some synthesis errors. Unfortunately, areas as small as 5x5 pixels (in a 128x128 pixel patch) are easily noticed. The difficulties in automatically detecting such small areas complicate efforts to implement a mechanism for automatic correction. Our current implementation allows the user to interactively select and re-synthesize individual unaesthetic patches after the main automatic synthesis process has finished. Figs 9-11 in this paper show models in which parts of the texture were resynthesized across one or more patches. Figs 2, 5 and 7 show results that were obtained without any such postprocessing

Another limitation that bears mentioning is that we found that the large amounts of texture generated by the synthesis caused problems for certain machine architectures, specifically those that had hard built-in limitations on the amount of texture memory that could be used. When a high level of detail is desired, without any possibility for pattern repetition, the total amount of texture required to cover the mesh can easily exceed the total size of the texture memory. All of the machines that did not allow the storage of

[1] optimization flags -O2 -funroll-loops -fstrict-aliasing -march=i686

textures in main memory failed to display the more complex models (e.g. the crocodile skinned triceratops). Using a texture atlas similar to the one described in [33] might help reduce the texture memory usage, but would incur the cost of incorrect mipmapping and having to store texture coordinates. In our current implementation each patch is stored as a single texture, producing a texture memory waste of up to 25% to 50% depending on the model. However thanks to the OpenGL texture compression extensions, all of the models presented in this paper can maintain interactive frame rates on our standard PC, even in cases where the amount of uncompressed texture exceeds 100Mb.

The method that we have proposed is currently designed to be applied to static models, and we have not thought about how to extend it to the case of deforming animated objects. Modifications to make the texture synthesis process deterministic are an obvious first step toward satisfying this requirement, but it is not immediately clear how one could guarantee that independently synthesized patterns will not differ profoundly from frame to frame as the mesh defining the object is globally deformed.

To achieve good results with the shape-following textures, the direction field must be band-limited: for any given texture there is a maximum spatial frequency that can be followed in the synthesis process and still produce correct results. Nevertheless we found that the method did a good job at singular points on the brick-textured lava lamp object, as can be seen in figure 11. Figures 12a and 12b provide further examples of the behaviour of our method near singularities.

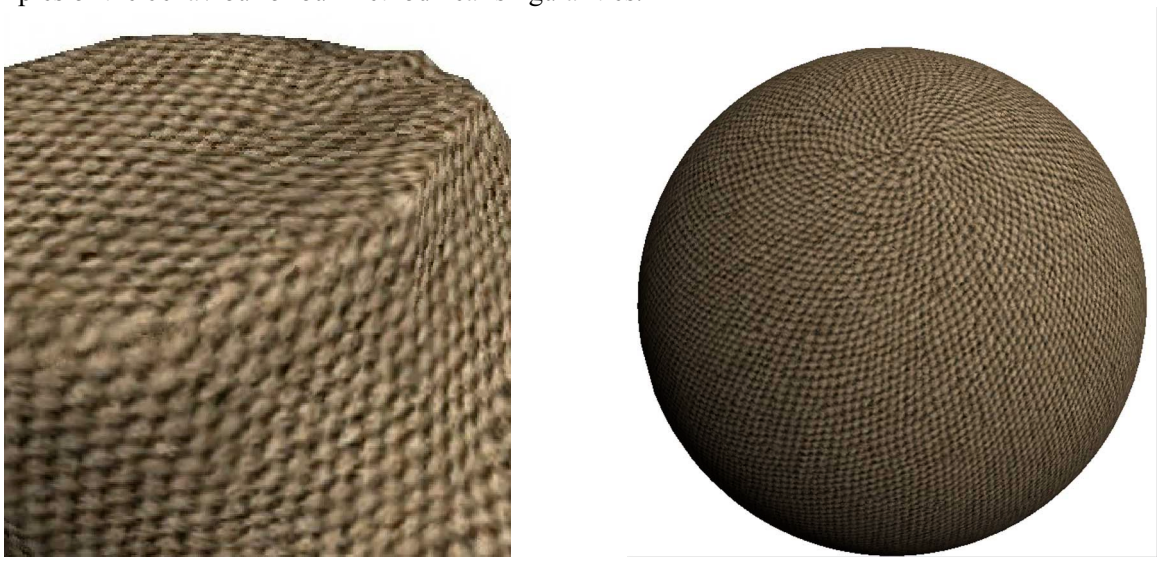


Figure 12: Left: A zoomed-in view of the singular point in the “up” directional field on the Venus model shown in figure 2. Right: A singular point on a sphere.

5. Applications And Future Work

There are many promising applications for this system and many directions for future work. One of the most interesting of these is multi-texturing. On a per-pixel basis it is possible to change not only the direction of the synthesized texture but even the texture itself according to any arbitrary function. Figures 13 and 14 are made using the illumination equation and two and four different textures respectively, each one with 63 rotations. These models, like all of the others in this paper, can be displayed at interactive frames rates on our standard PC.

The multi-texturing methods described in this paper have the potential to be useful for important applications in scientific visualization, for example in encoding a scalar distribution using texture type variations across an arbitrary domain in 2D or 3D. Other direction fields, such as gradient descent, hold promise for different applications, such as non-photorealistic rendering of terrain models (esp. in the case when it is desired to see through the surface). The methods that we have proposed can also be used for the visualization of scientifically computed vector fields over surfaces. An intriguing possible use for an extension of this work is in defining texture mixtures.

Figure 13: A demonstration of multi-texturing, in which the search for matches is performed within an array of different texture types. The texture type index can be defined by any function. In this example, we used the illumination function. In a real application one would want to use something more meaningful, such as soil type over a topographical terrain model.

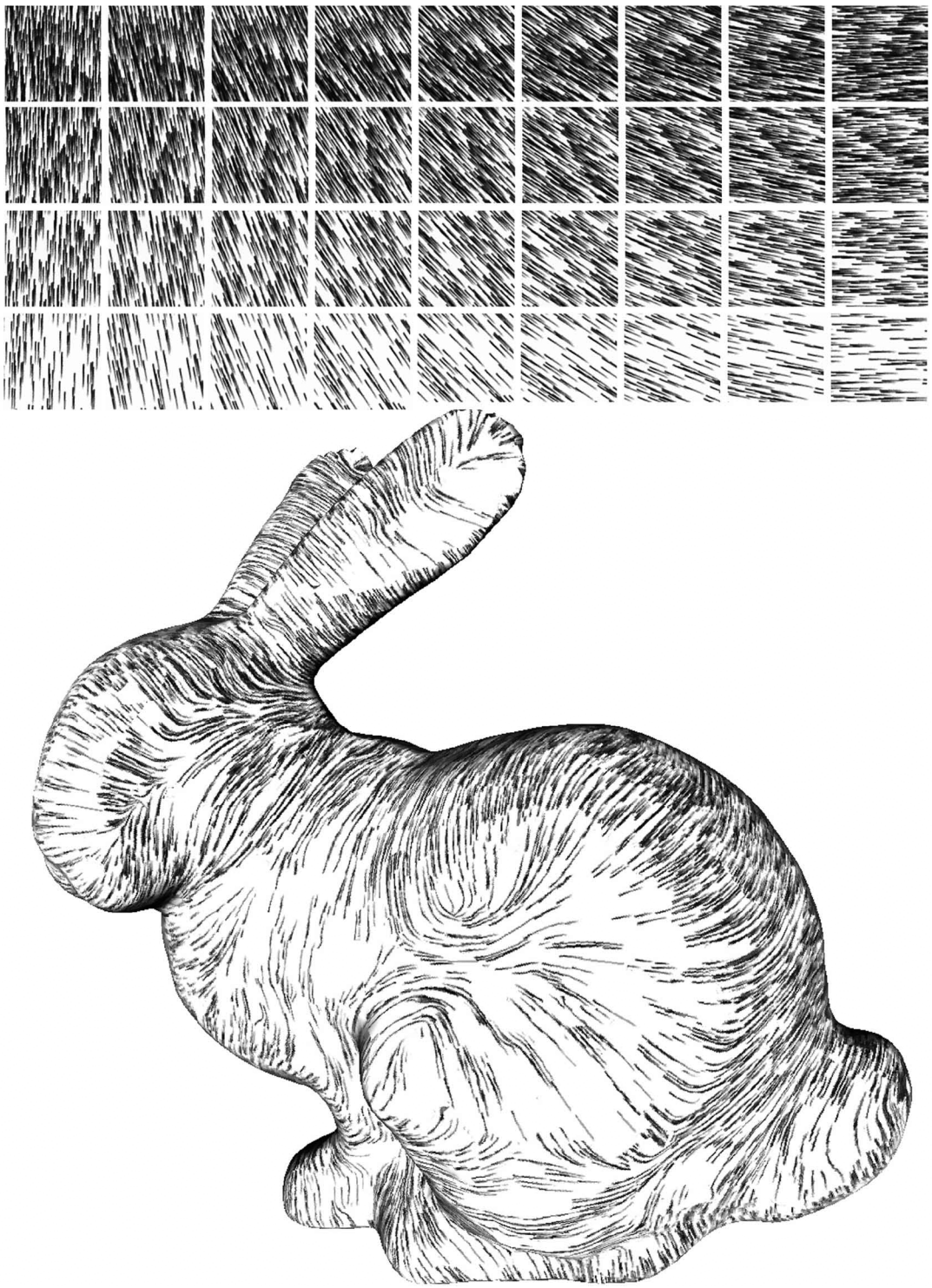


Figure 14: Multiple textures containing lines of different widths applied to an automatically-defined smooth vector field approximating the first principal direction over the Stanford bunny. Indexing along the dimension of varying stroke density was done as a function of the illumination.

6. Conclusions

In this paper we describe a novel surface texturing algorithm with a number of useful applications. Our described method for automatically synthesizing a desired texture pattern in a controlled orientation over the surface of an arbitrary object not only has potential applications in computer graphics, where it provides a simple and efficient solution to the classic problem of fitting a planar pattern to a non-developable surface in a way that minimizes both discontinuity and distortion. More importantly, it also has important potential applications in visualization, where it provides a means for texturing an arbitrary doubly curved surface with an arbitrary anisotropic pattern such that the orientation of the pattern follows a specified directional field over the surface at a per pixel level. With this capability it becomes possible to more thoroughly and directly investigate the effects on shape perception of a broader range of texture characteristics including orientation, and to come closer to answering the question of how best to design and apply a texture that can facilitate the accurate and intuitive perception of a surface's 3D shape.

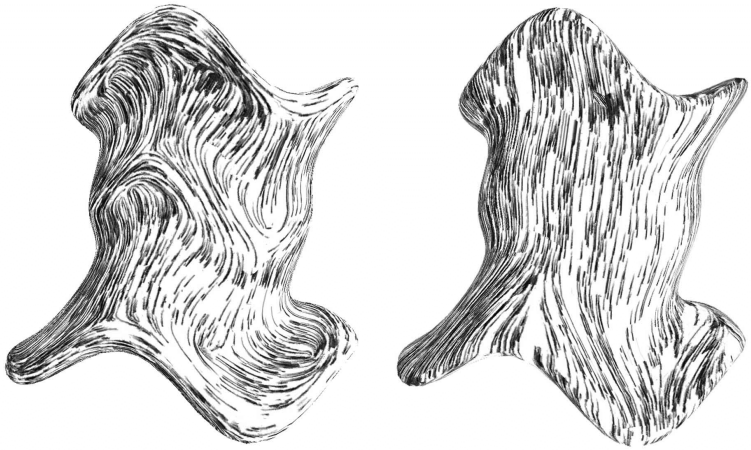


Figure 15: A final demonstration of the effect of texture orientation on shape perception — pen-and-ink style texture synthesized on an unfamiliar surface following the second principal direction (left) vs. a constant 'up' direction (right).

7. Acknowledgments

This work was supported by a University of Minnesota Grant-in-Aid of Research, Artistry and Scholarship. Additional support was provided by an NSF Presidential Early Career Award for Scientists and Engineers (PECASE) 9875368, NSF CAREER award CCR-9873670, ONR-N00014-97-2-0509, an Office of Naval Research Young Investigator Award, Presidential Early Career Award for Scientists and Engineers (PECASE) Navy/N00014-98-1-0631, and the NSF Learning and Intelligent Systems Program.

8. References

[1] Nur Arad and Gershon Elber. Isomeric Texture Mapping for Free-form Surfaces, *Computer Graphics Forum*, *1997*, **16**(5): 247– 256.

[2] Michael Ashikmin. Synthesizing Natural Textures. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, March 2001, pp. 217 – 226.

[3] Chakib Bennis, Jean-Marc Vézien and Gérard Iglésias. Piecewise Surface Flattening For Non-Distorted Texture Mapping, *Proceedings of SIGGRAPH 91*, pp. 237 – 246.

[4] Marcelo Bertalmio, Guillermo Sapiro, Li-Tien Cheng and Stanley Osher. A Framework for Solving Surface Differential Equations for Computer Graphics Applications, UCLA-CAM report 00-43, December 2000 (www.math.ucla.edu).

[5] Phil Brodatz. Textures: A Photographic Album for Artists and Designers, Dover Publications, 1966.

[6] Jeremy S. De Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images, *Proceedings of SIGGRAPH 97*, pp. 361 – 368

[7] Bruce G. Cumming, Elizabeth B. Johnston and Andrew J. Parker. Effects of Different Texture Cues on Curved Surfaces Viewed Stereoscopically, *Vision Research*, 1993, **33**(5/6): 827 – 838.

[8] J.-M. Dischler, D. Ghazanfarpour and R. Freydier. Anisotropic Solid Texture Synthesis Using Orthogonal 2D Views, Computer Graphics Forum, **17**(3), September 1998.

[9] Alexei A. Efros and Thomas K. Leung. Texture Synthesis by Non-Parametric Sampling, *Proceedings of the International Conference on Computer Vision*, vol. 2, 1999, pp. 1033 – 1038.

[10] Alexei A. Efros and William T. Freeman. Image Quilting for Texture Synthesis and Transfer, *Proceedings of ACM SIGGRAPH 2001*, pp. 341-346.

[11] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley and Donald P. Greenberg. A Model of Visual Masking for Computer Graphics, *Proceedings of SIGGRAPH 97*, pp. 143 – 152.

[12] Ahna Girshick, Victoria Interrante, Steve Haker and Todd LeMoine. "Line Direction Matters: an Argument for the use of Principal Directions in Line Drawings", *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 43 – 52, June 2000.

[13] Jack Goldfeather. "Understanding Errors in Approximating Principal Direction Vectors", TR01-006, Dept. of Computer Science and Engineering, Univ. of Minnesota, Jan. 2001.

[14] Gabriele Gorla, Victoria Interrante and Guillermo Sapiro. Growing Fitted Textures, *IMA preprint 1748*, Feb. 2001.

[15] Steven Haker, Sigurd Angenent, Allen Tannenbaum, Ron Kikinis, Guillermo Sapiro and Michael Halle. Conformal Surface Parameterization for Texture Mapping, *IEEE Transactions on Visualization and Computer Graphics*, **6**(2), April/June 2000, pp. 181 – 189.

[16] Pat Hanrahan and Paul Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes, *Proceedings of SIGGRAPH 90*, pp. 215 – 223.

[17] David J. Heeger and James R. Bergen. Pyramid-Based Texture Analysis/Synthesis, *Proceedings of SIGGRAPH 95*, pp. 229 – 238.

[18] Victoria Interrante, Henry Fuchs and Stephen Pizer. Conveying the 3D Shape of Smoothly Curving Transparent Surfaces via Texture, *IEEE Computer Graphics and Applications*, 1997, **3**(2): 98 – 117.

[19] Victoria Interrante. Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution, *Proceedings of SIGGRAPH 97,* pp. 109 – 116.

[20] Victoria Interrante and Sunghee Kim. Investigating the Effect of Texture Orientation on the Perception of 3D Shape, *Human Vision and Electronic Imaging VI*, SPIE **4299**, January 2001, pp. 330 – 339.

[21] Victoria Interrante, Sunghee Kim and Haleh Hagh-Shenas. Conveying 3D Shape with Texture: Recent Advances and Experimental Findings, *Human Vision and Electronic Imaging VII*, SPIE **4662**, January 2002, to appear.

[22] David C. Knill. Contour into Texture: The Information Content of Surface Contours and Texture Flow, *Journal of the Optical Society of America, A*, **18**(1), January 2001, pp. 12-35.

[23] Bruno Lévy and Jean-Laurent Mallet. Non-distorted Texture Mapping for Sheared Triangulated Meshes, *Proceedings of SIGGRAPH 98*, pp. 343 – 352.

[24] Andrea Li and Qasim Zaidi. Perception of Three-Dimensional Shape From Texture is Based on Patterns of Oriented Energy, *Vision Research*, **40**(2), January 2000, pp. 217 – 242.

[25] Jérôme Maillot, Hussein Yahia and Anne Verroust. Interactive texture mapping, *Proceedings of SIGGRAPH 93*, pp. 27 – 34

[26]Pascal Mamassian and Michael S. Landy. Observer Biases in the 3D Interpretation of Line Drawings, *Vision Research*, **38**(18), September 1998, pp. 2817 – 2832.

[27]Alan P. Mangan and Ross T. Whitaker. Partitioning of 3D Surface Meshes using Watershed Segmentation, *IEEE Transactions on Visualization and Computer Graphics,* **5**(4), pp, 308-321.

[28]Fabrice Neyret and Marie-Paul Cani. Pattern-Based Texturing Revisited, *Proceedings of SIGGRAPH 99*, pp. 235 – 242.

[29]Darwyn R. Peachey. Solid texturing of Complex Surfaces, *Proceedings of SIGGRAPH 85*, pp. 279–286.

[30]Ken Perlin. An Image Synthesizer, *Proceedings of SIGGRAPH 85*, pp. 287 – 296.

[31]Dan Piponi and George Borshukov. Seamless Texture Mapping of Subdivision Surfaces by Model Pelting and Texture Blending, *Proceedings of SIGGRAPH 2000*, pp. 471 – 478.

[32]Javier Portilla and Eero P. Simoncelli. A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients, *International Journal of Computer Vision*, **40**(1), October 2000, pp. 49 – 70.

[33]Emil Praun, Adam Finkelstein and Hughes Hoppe. Lapped Textures, *Proceedings of SIGGRAPH 2000*, pp. 465 – 470.

[34]Pedro V. Sander, John Snyder, Steven J. Gortler and Hughes Hoppe. Texture Mapping Progressive Meshes, *Proceedings of ACM SIGGRAPH 2001*, pp. 409-416.

[35]Kent A. Stevens. The Visual Interpretation of Surface Contours, *Artificial Intelligence*, **17**(1-3), August 1981, pp. 47 – 73.

[36]James T. Todd and Robin A. Akerstrom. Perception of Three-Dimensional Form from Patterns of Optical Texture, *Journal of Experimental Psychology; Human Perception and Performance*, **13**(2), 1987, pp. 242 – 255.

[37]Greg Turk. Generating Textures for Arbitrary Surfaces Using Reaction-Diffusion, *Proceedings of SIGGRAPH 91*, pp. 289 – 298.

[38]Greg Turk. Texture Synthesis on Surfaces, *Proceedings of ACM SIGGRAPH 2001*, pp. 347 – 354.

[39]Li-Yi Wei and Marc Levoy. Fast Texture Synthesis Using Tree-structured Vector Quantization, *Proceedings of SIGGRAPH 2000*, pp. 479 – 488.

[40]Li-Yi Wei and Marc Levoy. Texture Synthesis Over Arbitrary Manifold Surfaces, *Proceedings of ACM SIGGRAPH 2001*, pp. 355 – 360.

[41]Andrew Witkin and Michael Kass. Reaction-Diffusion Textures, *Proceedings of SIGGRAPH 91*, pp. 299– 308.

[42]Steven Worley. A Cellular Texture Basis Function, *Proceedings of SIGGRAPH 96*, pp. 291– 294.

[43]Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and Shape Synthesis on Surfaces, *Proceeedings of the 12th Eurographics Workshop on Rendering*, June 2001.

[44]Song Chun Zhu, Ying Nian Wu and David Mumford. Filters, Random Fields and Maximum Entropy (FRAME): Towards a Unified Theory for Texture Modeling, *International Journal of Computer Vision*, **27**(2), March 1998, pp. 107 – 126.

Paint By Numbers: Abstract Image Representations

Paul Haeberli
Silicon Graphics Computer Systems

ABSTRACT

Computer graphics research has concentrated on creating photo-realistic images of synthetic objects. These images communicate surface shading and curvature, as well as the depth relationships of objects in a scene. These renderings are traditionally represented by a rectangular array of pixels that tile the image plane.

As an alternative to photo-realism, it is possible to create abstract images using an ordered collection of brush strokes. These abstract images filter and refine visual information before it is presented to the viewer. By controlling the color, shape, size, and orientation of individual brush strokes, impressionistic paintings of computer generated or photographic images can easily be created.

CR Categories and Subject Descriptors: I.3.2 **[Computer Graphics]:** Picture/Image Generation - Display algorithms; I.3.6 **[Computer Graphics]:** Methodology and Techniques - Interaction techniques;

Additional Key Words and Phrases: Painting, image processing, abstract images.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Introduction

This paper is not about radiosity, anti-aliasing, or motion blur. Its not about making pictures more realistic. It is about creating interesting abstract representations of natural and synthetic scenes.

Graphic designers are experts at visual communication. In their work, graphic designers use photographic images when they are appropriate, but often chose to use more abstract images such as drawings or paintings. In many cases the designer must balance realism and effectiveness. Sometimes a realistic photographic image may be less effective than a stylized image.

In a panel discussion at Siggraph 1988 [Phillips 88] on the design of effective images Margret Hagen described the goal of the visual artist:

> "The goal of effective representational image making, whether you paint in oil or in numbers, is to select and manipulate visual information in order to direct the viewer's attention and determine the viewer's perception."

Impressionist painters use brush strokes to control light to simulate objects without modelling object detail explicitly. Only a few brush strokes are needed to represent a standing figure, a person's face at a distance, or a tree. By carefully selecting the location, color, size and direction of brush strokes, they control visual information to communicate abstract images to the viewer.

A Simple Painting Technique

Our goal is to take a synthetic or natural scene, and convert it into abstract impressionistic image. We want to make it easy for a user to interactively select and manipulate visual information to explore many different representations of a single source image. To do this we will point sample the source image at some set of brush stroke locations, and draw a synthetic brush stroke with the appropriate color.

A simple interactive program allows the user to operate on a source image. The basic interactive technique

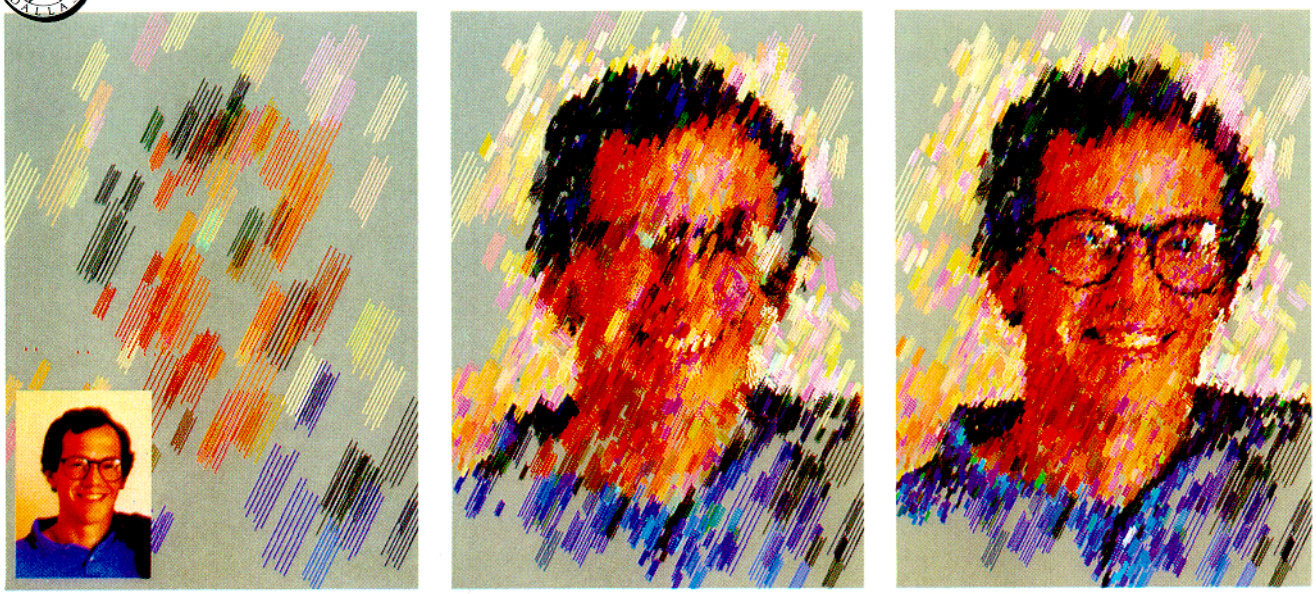


Figure 1. Creating a painting.

is to follow the cursor across the canvas, point sample the color of a stored image at the location of the cursor, and then paint a brush stroke of that color. Figure 1. shows a painting being created. The image that drives the painting is shown as an inset. The final painting is on the right. Using this process a user can easily explore different abstract representations of the source image.

In the work described here, a painting is an ordered list of brush strokes. Each stroke of the brush is described by a collection of attributes as shown in Figure 2.

Location - Position of the brush stroke.

Color - The RGB and Alpha color of the stroke.

Size - How big the stroke is.

Direction - Angle of the stroke in the painting.

Shape - The look of the brush stroke.

Figure 2. Brush stroke attributes

We will use the location, color, size, direction, and shape of brush strokes to communicate visual information to the viewer. The kinds of information we want to communicate are surface color, surface curvature, center of focus, and location of edges.

The program follows the position of the cursor across the canvas and point samples the source image to obtain a color. If a mouse button is down, an image of a brush stroke is drawn at a particular size and direction. In this way, visual information is selectively transferred from the source image to the canvas. By changing the size, direction and shape of brush strokes, many different representations of a single photographic image may easily be created.

One limitation in conventional paint programs is the time needed to pick a new color to paint with. [Lewis 84] described this as the "put-that-color-there" procedure that most paint programs use. Since the overhead in chosing a new color is high, it is very difficult to create a painting with a large number of different colors. The simple technique described escapes this problem by continuously sampling the color of the input image as painting proceeds.

Brush stroke locations are created in a stochastic distribution in the neighborhood of the cursor. This generates a scattering of brush stokes instead of a line of strokes when the cursor is moved in a straight line. This is an example of a simple interactive particle system [Reeves 85].

The size of brush strokes can be controlled in two ways. One technique uses the average speed of the cursor to control the brush size. If the user is painting quickly, the brush strokes will be made larger. When the user slows down, the brush size gets smaller. This makes it easy to first create a rough representation of the image with large bush strokes before adding finer detail with smaller brush strokes. Another technique allows the user to control the size of brush strokes using up and down arrow keys.

The orientation of brush strokes is controlled in several ways. One technique orients brush strokes with respect to the direction that the user moves the cursor. This is the tracking brush described in [Smith 79]. An alternative is to use a gesture of the mouse to set the current direction for brush strokes. To do this the user clicks down with the middle mouse button, moves in the direction brush strokes should flow, and releases the button. Other input devices could be used to interactively change the brush stoke size and orientation, as described in [Smith 79].

The shape of brush stokes can significantly influence the character of the final painting. The geometry of brush

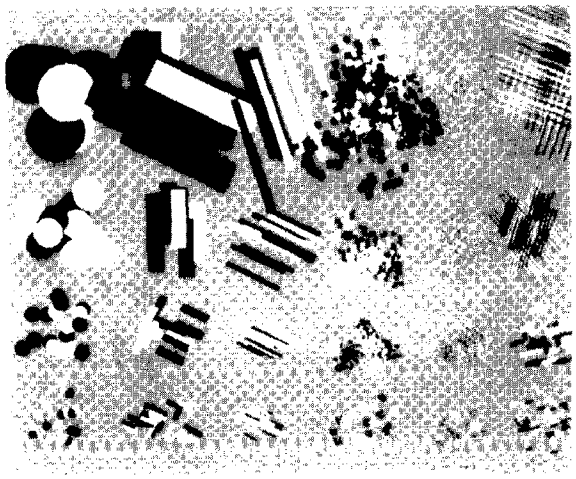


Figure 3. Several brush shapes.

strokes can be selected using a pop-up menu. The user can select among circles, rectangles, lines or scatterings of points and polygons for brush strokes, or they draw their own stroke if they want. In this implementation, all brush strokes are drawn using geometry instead of bitmaps - this makes scaling and rotating the brush very efficient on our hardware. Figure 3 shows several possible brush stroke shapes.

Three paintings of the same image are shown in Figure 4. The original image is shown inset. Diagonal brush strokes were used to create the painting on the left. In these paintings the amount of detail across the painting was modulated to direct the viewer. Noise was added to the original image to create a distribution of paint colors.

One interesting brush stroke geometry is a cone. When a group of cones are z-buffered onto the canvas, they create domains of color that share the image plane. The result is a pattern of color that has a cellular characteristic. These domains of color are Dirichlet domains [Preparata 85] derived from the location of the brush strokes. A painting made with z-buffered cones is shown in the center of Figure 4. By lingering with the cursor more detail can be exposed on any part of the canvas. A pointillist representation is shown on the right of Figure 4.

A description of the painting can be saved by interacting with a menu. A complete painting is an ordered set of brush stokes. A binary file format is used to store a stroke by stoke description of the painting. Figure 5 shows what a textual representation of a painting looks like.

```
painting with 9298 strokes
position        RGBA color            siz dir brush
0.447 0.541  241 128 173 255  attr:  30  89   5
0.444 0.531  220  57  35 255  attr:  30  89   5
0.441 0.524  172  29   1 255  attr:  30  89   5
0.444 0.553  230 100  75 255  attr:  30  89   5
0.447 0.526  220 112 162 255  attr:  30  93   5
0.456 0.554  245 189 137 255  attr:  30  93   5
0.503 0.522  245 183 237 255  attr:  31 107   5
0.479 0.545  228 141  92 255  attr:  31 101   5
0.498 0.517  246 181 230 255  attr:  31 103   5
```

Figure 5. A textual representation of a painting

Operations on Paintings

After a painting is saved, we can transform it using several paint processing programs. A painting renderer transforms a description of a painting into an RGB image. This can be used to create an extremely high resolution image of a painting.

A set of unary operators modify a single painting. All the brush strokes in a stored painting can be scaled up or down. Paintings can be altered by adding noise to brush stroke locations, color, size or direction. If wanted, each

Figure 4. Three paintings of one image.

brush stroke can be rotated by adding a constant to its direction. The order of brush strokes in a Painting can be sorted by color, direction, position, or size.

Binary painting operators use two paintings as input. It is easy to interpolate between or extrapolate beyond two paintings that have the same number of brush strokes. Paintings may be animated in this way. The brush strokes in two paintings can also be concatenated to overlay one painting on top of another.

Another tool lets us use an RGB image to modify the color, size, or direction of brush strokes in a stored painting.

The tools described above can be used to automatically roto-scope live action. To do this, we generate a rectangular array of brush locations, then add noise to their positions. Finally, RGB colors are assigned to each brush stroke in the painting by sampling one frame of live action.

Advanced Painting Techniques

Interesting effects can be created by using arbitrary images to control the brush direction across the canvas. Figure 6. shows a painting that was created by using a second image to control the brush direction. This painting consists of 9298 brush strokes.

When sampling a natural image, we start with only RGB information. Sometimes when painting, we might want to make brush strokes become aligned automatically with edges in the image. This can be done by generating an image that contains information about direction of the luminance gradient in the image [Kass 87]. Figure 7. shows a painting that was created using this technique. First the original image was converted to black and white and low pass filtered. Then the direction of the gradient of this image was used to control the brush stroke direction while painting. Notice how the brush strokes outline the shape of the head and flow along the collar of the shirt.

Another interesting technique is to blend the brush strokes onto the canvas. In Figure 8, a scanned in image of a real brush stroke is used. This brush stroke image was texture mapped onto the canvas to create the final painting.

Spice for images: Video-Sodium Glutamate

Before painting a synthetic or natural image it may be good to use a little spice (VSG) to accentuate the features we consider most important. These enhancements can be applied globally or locally to draw attention to particular parts of the painting.

When traditional artists make illustrations or paintings, many techniques are used. Often important edges in the scene are exaggerated. Where a dark area meets a light area the dark area is drawn slightly darker, and the light area is drawn slightly lighter. Some artists call this tech-

Figure 6. Using a second image to control brush stroke direction.

Figure 7. Using the gradient direction to control brush strokes.

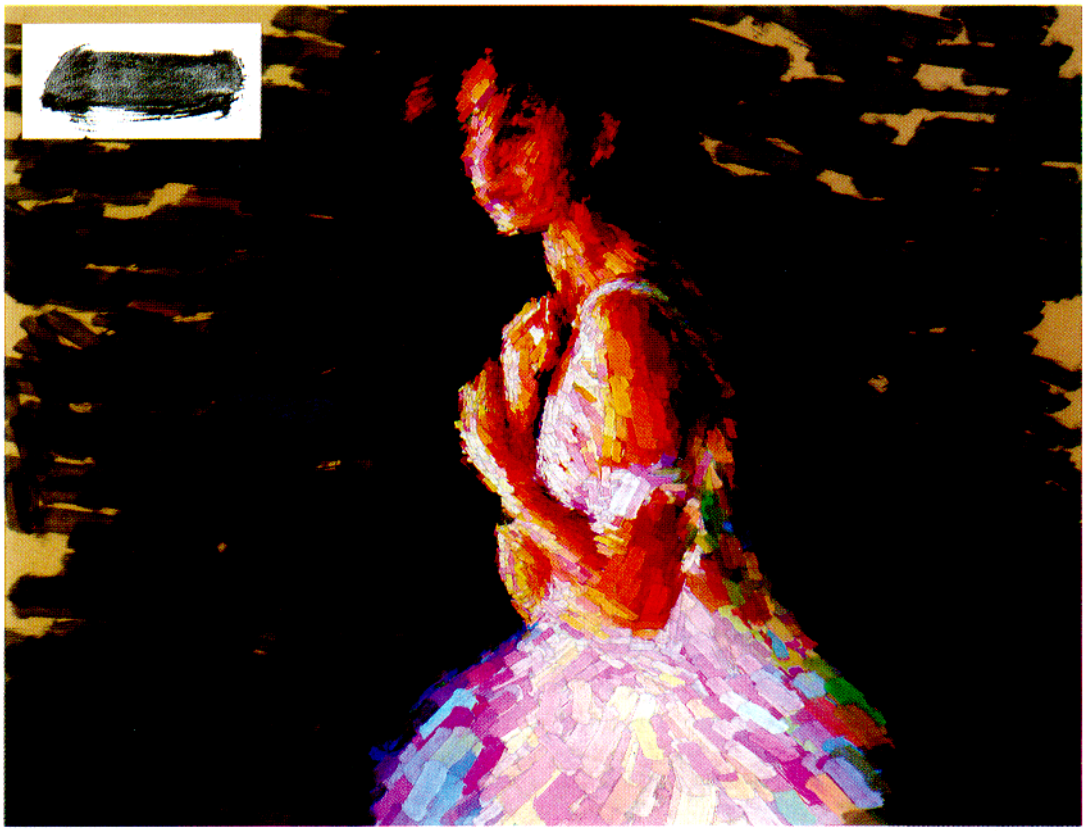


Figure 8. A texture mapped brush stroke.

nique "pushing" an edge. This technique can be used to make depth relationships between objects in the scene more explicit where they overlap [Porter 88].

To simulate "pushing edges" we can high pass filter the original image. High frequencies are enhanced by using unsharp masking. This is done as a two step process. First the original image is blurred by convolution. Next we create an enhanced image by extrapolating from the blurry image out beyond the original image.

To do this we define a linear interpolation operation on these two images such that a parameter of 1.0 gives us the blurred image and 0.0 gives the original image. To create an image with pushed edges, we make the parameter negative. As a result of this flat fields in the original image will remain unchanged, but edges are accentuated.

This process has two important variables; how blurry the image is made in the first step, and how much we extrapolate in the second step. Convolving with a 3 by 3 kernel, and making the parameter -0.5 will accentuate very high frequency detail and change the shading only very near edges. To make the shading change within 10 pixels of edges, a kernel with a width of more than 20 should be used. Some interesting effects can be created when the kernel is made very large - approaching 20% of the image diameter.

An artist may chose to enhance the richness of some colors in a scene. Sometimes the color is uniformly saturated throughout the scene, or only particular parts of the image are enhanced in this way.

If we want to increase the saturation of the image first we create a luminance image using a formula like this:

```
lum = 0.3*r + 0.59*g + 0.11*b.
```

Next, we extrapolate from the luminance image out beyond the original image. Achromatic parts of the image will remain unchanged, but all the colored parts of the image will be even more colorful, while preserving the same luminance. It is important that the original image be properly color balanced before this is done, otherwise improper colors for skin tones will become obvious.

When painting solid colored areas, artists may use a wide range of colors to communicate the color of a surface. This helps the viewer see a range of component colors in a surface that may be a single, flat color.

To add detail to regions of flat color, noise may be added to the image. When this is done, the final painting will have brush stokes with an interesting distribution of colors. This can make the final painting much more lively and interesting.

Most books on painting recommend sticking to a fairly limited palette of colors so as to achieve an overall harmony of color across the painting. The palettes used by the impressionists usually contained fewer than 12 colors [Callen 82]. These raw colors were mixed to create additional intermediate colors. Many beautiful paintings use remarkably few colors. With a restricted set of colors some brush strokes in the sky will closely match the color of brush strokes used to represent water. Restricting the number of different colors in a painting has the effect of unifying the painted image as a whole.

By quantifying colors in the source image, we can reduce the number of different colors in a scene without restricting the color gamut. If a sufficient amount of noise was added as discussed above, then no contouring will be visible.

Artists often cover the entire canvas with a wash of color before painting the image. The color of this background image can affect how the colors are perceived by the viewer if it is left exposed in some areas of the final painting. It has been noted by Michel-Eugene Chevreul [Smith 87] that having some proximity to gray makes all primary colors gain in brilliance and purity. Allowing the background wash color to be exposed throughout an image gives it a kind of unity and integrity.

Colors are sometimes used to provide depth cues. Colors in the range green (grass), cyan, and blue (sky) recede, while yellow, orange, red and magenta move to the foreground.

We can use many of the techniques above to enhance digital images before painting begins.

Sampling Geometry using Ray-Painting

These painting techniques can be used to create painted representations of synthetic 3D scenes as well. When sampling geometry, we have direct access to the color of each surface, its normal, and its depth. We can use the surface normal to control the direction of the brush strokes. This provides the viewer with valuable information about the orientation of surfaces. Figure 9. shows a raytraced scene, and a painting that uses surface normals to control the direction of brush strokes. To make these illustrations, the user interface of the paint program was attached to a raytracer, letting the user reveal the geometry by sampling it in real-time. Notice how the brush strokes appear to wrap around the sphere, the cone and the cylinder.

Approximating Images by Relaxation

An iterative relaxation technique may be used to create interesting paintings with remarkably brush strokes. The left side of Figure 10 shows a painting of 100 rectangular brush strokes that approximates an image of a seated man. The right side of Figure 10 shows a painting

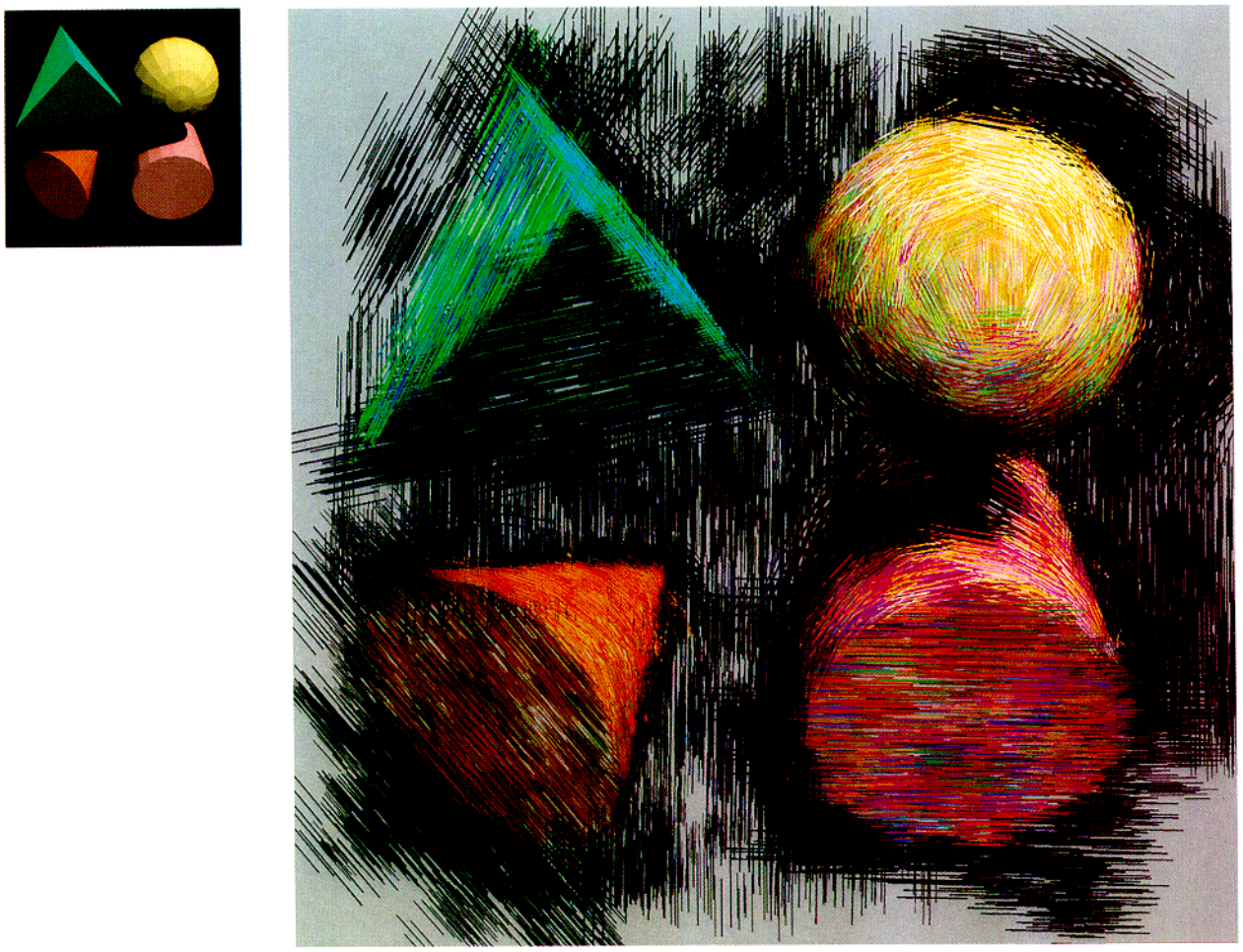


Figure 9. Sampling geometry using raytracing.

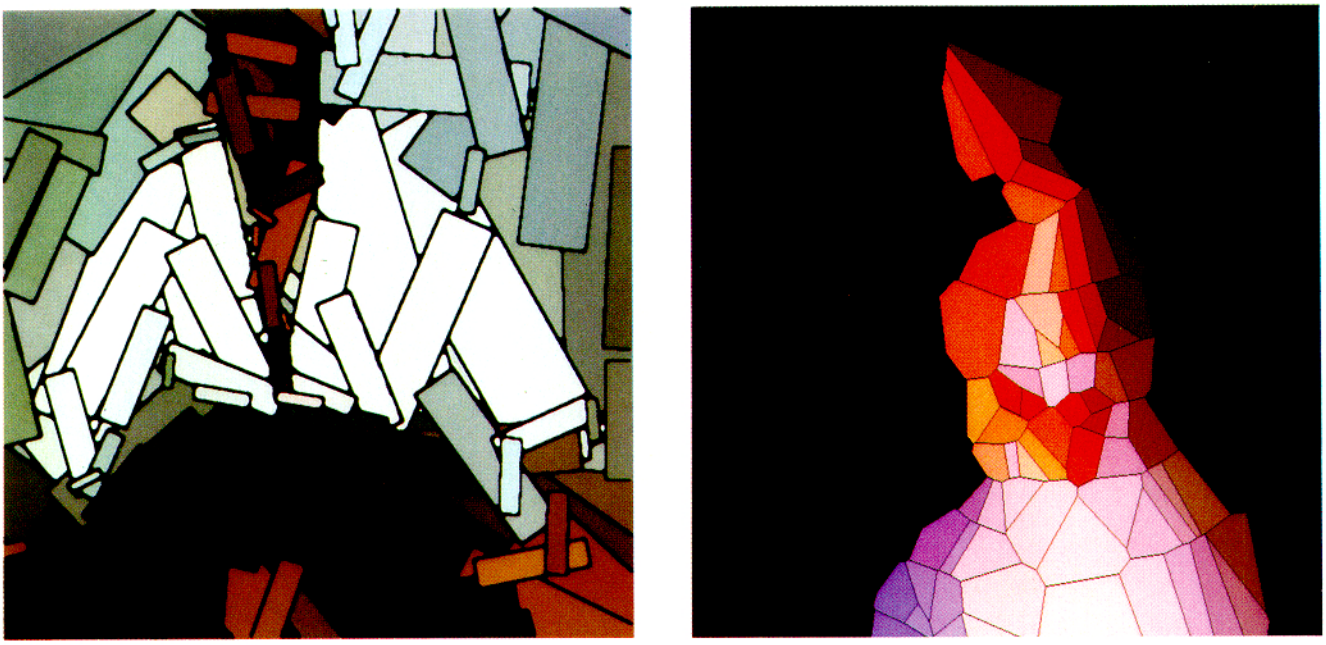


Figure 10. Using relaxation to create paintings automatically.

of 100 Dirichlet domains. These two paintings were created by stochastically perturbing the attributes of the brush strokes, while minimizing the root mean squared difference between the original image and the painted representation. This process ran for several hours before these images were saved.

Conclusions

We present several techniques for creating static and animated abstract images of photographed and synthetic scenes.

In this work, the goal is not to make photo-realistic images, but rather effective, interesting images that communicate. By interactively processing an image we can select and manipulate visual information to eliminate distracting detail, provide cues about surface orientation, and influence the viewer's perception of the subject.

It is natural that we want to continue to explore new painting techniques. A logical extension of this work would incorporate the texture synthesis work of [Lewis 84] and the brush modelling work of [Strassman 86].

Acknowledgements

I would like to thank Seth Teller for creating software to support rendering Dirichlet domains. I also appreciate the many helpful comments provided by the reviewers.

Some aspects of this work are patent pending.

References

[Callen 82] Anthea Callen, "Techniques of the Impressionists", 1982.

[Heckbert 87] Paul S. Heckbert, "Ray Tracing Brand Gelatin", Computer Graphics, 1987.

[Kass 87] Michael Kass and Andrew Witkin, "Analyzing Oriented Patterns", Computer Vision, Graphics, and Image Processing, 37, 1987.

[Lewis 84] John-Peter Lewis, "Texture Synthesis for Digital Painting", Computer Graphics, 1984.

[Phillips 88] Dick Phillips, "Siggraph '88 Panels Proceedings", 1988.

[Porter 88] Tom Porter and Sue Goodman, "Designer Primer", Charlse Scribner's Sons, 1988.

[Preparata 85] Franco P. Preparata and Michael Ian Shamos, "Computational Geometry", Springer-Verlag, 1985.

[Reeves 85] William T. Reeves and Ricki Blau, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", Computer Graphics, 1985.

[Smith 79] Alvy Ray Smith, "Table Paint", SIGGRAPH tutorial notes for "Two Dimensional Computer Animation, 1981.

[Smith 87] Ray Smith, "The Artists Handbook", Alfred A. Knopf, 1987.

[Strassman 86] Steve Strassman, "Hairy Brushes", Computer Graphics, 1986.

Illustrating smooth surfaces

Aaron Hertzmann Denis Zorin

New York University

Abstract

We present a new set of algorithms for line-art rendering of smooth surfaces. We introduce an efficient, deterministic algorithm for finding silhouettes based on geometric duality, and an algorithm for segmenting the silhouette curves into smooth parts with constant visibility. These methods can be used to find all silhouettes in real time in software. We present an automatic method for generating hatch marks in order to convey surface shape. We demonstrate these algorithms with a drawing style inspired by *A Topological Picturebook* by G. Francis.

CR Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation– *Display algorithms.*

Additional Keywords: Non-photorealistic rendering, silhouettes, pen-and-ink illustration, hatching, direction fields.

1 Introduction

Line art is one of the most common illustration styles. Line drawing styles can be found in many contexts, such as cartoons, technical illustration, architectural design and medical atlases. These drawings often communicate information more efficiently and precisely than photographs. Line art is easy to reproduce, compresses well and, if represented in vector form, is resolution-independent.

Many different styles of line art exist; the unifying feature of these styles is that the images are constructed from uniformly colored lines. The simplest is the style of silhouette drawing, which consists only of silhouettes and images of sharp creases and object boundaries. This style is often sufficient in engineering and architectural contexts, where most shapes are constructed out of simple geometric components, such as boxes, spheres and cylinders. This style of rendering captures only geometry and completely ignores texture, lighting and shadows. On the other end of the spectrum is the pen-and-ink illustration style. In pen-and-ink illustrations, variable-density hatching and complex hatch patterns convey information about shape, texture and lighting. While silhouette drawing is sufficient to convey information about simple objects, it is often insufficient for depicting objects that are complex or free-form. From many points of view, a smooth object may have no visible silhouette lines, aside from the outer silhouette (Figure 8), and all the information inside the silhouette is lost. In these cases, can be added to indicate the shape of the surface.

The primary goal of our work was to develop rendering techniques for automatic generation of line-art illustrations of piecewise-smooth free-form surfaces. When using conventional photorealistic rendering techniques (e.g. Z-buffer or ray tracing) one can typically replace a smooth surface with a polygonal approximation, and thus reduce the problem to that of rendering polygonal meshes. This no longer true when our goal is to generate line drawings. Some differential quantities associated with the smooth surface must be recovered in order to generate visually pleasing hatch directions and topologically correct silhouette lines. Some of the problems that occur when a smooth surface is replaced by its polygonal approximation are discussed in greater detail in Section 4.

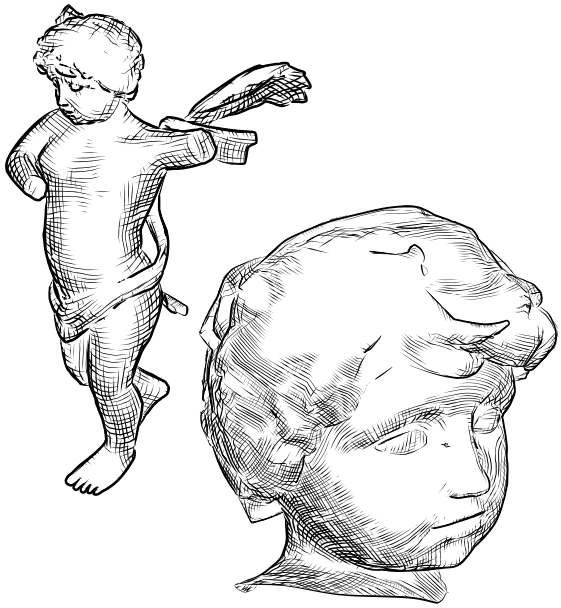


Figure 1: Illustrations of the Cupid mesh.

In this paper we address two general problems: computing silhouette curves of smooth surfaces, and generating smooth direction fields on surfaces that are suitable for hatching. The algorithms that we have developed can be used to implement a number of non-photorealistic rendering techniques. Our main focus is on a particular rendering style, which aims to communicate all essential information about the shape of the surface with a limited amount of hatching.

Contributions. *Algorithms.* To support rendering of smooth surfaces, we have developed a number of novel algorithms including:

- An efficient, deterministic algorithm for detecting silhouettes; (Section 4.3). In addition to non-photorealistic applications, this method can be used to accelerate computation of shadow volumes.
- An algorithm for cusp detection and segmentation of silhouette curves into smooth parts with constant visibility (Section 4.2).
- An algorithm for computing smooth direction fields on surfaces, suitable for use in hatching (Section 5). These fields have a wide range of uses, ranging from high-quality pen-and-ink rendering to interactive illustration and hatching.

An important feature of our approach is that any polygonal mesh can serve as input; the smooth surface that we render is inferred from the mesh. We do not assume an explicitly specified parame-

terization, which make our approach more general than previously developed techniques.

Rendering style. We have developed a new non-photorealistic rendering style based on the techniques of Francis [15], and influenced by the cartoons of Thomas Nast [34] and others.

The rules for drawing in this style are described in Section 6.

2 Previous Work

The methods used in nonphotorealistic rendering can be separated into two groups: image-space and object-space. The image-based approach is general and simple; however, it is not particularly suitable for generating concise line drawings of untextured smooth surfaces. Image-based techniques are presented in [5, 30, 7, 18, 6, 28]; these algorithms exploit graphics hardware to produce image precision silhouette images. Our technique is an object space method; it directly uses the 3D representation of objects, rather than their images. Winkenbach and Salesin [36] describe a method for producing appealing pen-and-ink renderings of smooth surfaces. Parametric lines on NURBS patches were used to determine the hatch directions and silhouette lines were computed using polyhedral approximation to the surface. Their main technical focus is on using the hatch density to render complex texture and lighting effects. Their system relied on a surface parameterization to produce hatch directions; however, such a parameterization does not exist for many types of surfaces, and can often be a poor indicator of shape when it does exist. Elber [12, 13] and Interrante [21] used principal curvature directions for hatching. Curvatures generally provide good hatch directions, but cannot be reliably or uniquely computed at many points on a surface. Our system makes use of the principle curvature directions, and uses an optimization technique to "fill in" the hatching field where it is poorly-defined. Deussen et al. [9] use intersections of the surfaces with planes; while being quite flexible, this approach requires segmentation of the surface into parts, where different groups of planes are used; the plane orientations computed using skeletons relate only indirectly to the local surface properties.

Our work also draws on techniques developed for vector field visualization [8, 22]. It should be noted that relatively little work has been done on generating fields on surfaces as opposed to visualization of existing fields. Elber [12, 13] discuses the relative merits of some commonly-used hatching fields (principle curvature directions, field of tangents to the isoparametric lines, the gradient field of the brightness).

Silhouette detection is an important component of many nonphotorealistic rendering systems. Markosian et al. [25] presented a randomized algorithm for locating silhouettes; this system is fast but does not guarantee that all silhouettes will be found. Gooch et al. [18] and Benichou and Elber [3] proposed the use of a Gauss map to efficiently locate all object silhouettes under orthographic projection. In this paper, we present a new method for silhouette detection that is fast, deterministic, and applicable to both orthographic and perspective projection.

Our method for computing the silhouette lines of free-form surfaces is closely related to the work of [14, 17] in computing silhouettes for NURBS surfaces.

3 Overview

In this section we present a general overview of our algorithms.

Surface representation. The input data for our system is a polygonal mesh that approximates a smooth surface. Polygonal meshes remain the most common and flexible form for approximating surfaces. However, information about differential quantities (normals, curvatures, etc.) associated with the original surface is lost. We need a way to estimate these quantities and compute, if necessary, finer approximations to the original smooth surface. This can be done if we choose a method that allows us to construct a smooth surface from an approximating arbitrary polygonal mesh, and easily compute the associated differential quantities (normals, curvatures, etc.).

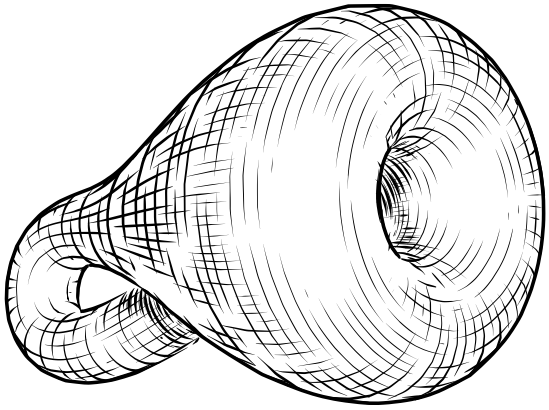


Figure 2: Klein bottle. Lighting and hatch directions are chosen to convey surface shape. Undercuts and Mach bands near the hole and the self-intersection enhance contrast.

We use piecewise-smooth subdivision, similar to the algorithms presented in [20], with an important modification (Appendix A) to make the curvature well-defined and nonzero at extraordinary vertices. However, other ways of defining smooth surfaces based on polygonal meshes can be used, provided that all the necessary quantities can be computed.

Algorithms. Our rendering technique has three main stages: computation of a direction field on the surface, computation of the silhouette lines and generation of hatch lines.

Hatch direction field. This stage defines a view-independent field on the surface that can be used later to generate hatches. Rather than defining two separate directional fields, we define a single *cross field* (Section 5) for hatches and cross-hatches. The main steps of our algorithm are: smooth the surface if necessary; compute an initial approximation to the field in areas of the surface where it is well defined, initialize the directions arbitrarily elsewhere; optimize the directions in places where the cross field was not well defined.

Silhouette curve computation. We compute the curves in several steps (Section 4): compute boundary, self-intersection and crease curves, as well as boundaries of flat areas; compute silhouette curves as zero-crossings of the dot product of the normal with the view direction; find cusps, determine visibility, and segment the silhouette curves into smooth pieces.

Hatch generation. Our hatch generation algorithms follow some of the rules described by Francis [15] (Section 6). The surface is divided into four levels of brightness with corresponding levels of hatching: highlights and Mach bands (no hatching), midtones (single hatching), shadowed regions (cross-hatching), and undercuts (dense cross-hatching). Line thickness varies within each region according to the lighting. Undercuts and Mach bands are used to increase contrast where objects overlap. Lights are placed at the view position or to the side of the object. The hatching algorithm covers all hatch regions with cross-hatches, then removes hatches from the single hatch regions as necessary.

4 Computing Silhouette Drawings

In this section we describe algorithms for generating the simplest line drawings of smooth surfaces, which we call silhouette drawings. A silhouette drawing includes only the images of the most visually important curves on the surface: boundaries, creases, silhouette lines and self-intersection lines. Finding intersections of

smooth surfaces is a complex problem, which we do not address in the paper. We find self-intersections of a mesh approximating the surface and assume that self-intersection lines of the mesh approximate the self-intersection curves of the surface sufficiently well. Boundary curves and creases are explicitly represented in the surface; thus, we focus our attention on the problem of computing the silhouette lines. We will refer to the creases, boundaries and self-intersection curves as *feature curves*.

Before proceeding, we recall several definitions[1]. First, we define more precisely what we mean by a piecewise-smooth surface. A piecewise-smooth surface can be thought of as a finite union of a number of smooth surfaces with boundaries. A smooth embedded surface is a subset M of $\mathbf{R}^3$ such that for any point $\mathbf{p}$ of this subset there is a neighborhood $U(\mathbf{p}) = \text{Ball}_\epsilon(\mathbf{p}) \cap M$ and a C^1-continuous nondegenerate one-to-one map $\mathbf{F}(u, v)$ from a domain D in $\mathbf{R}^2$ onto $U(\mathbf{p})$. The domain D can be taken to be an open disk for interior points, and a half-disk (including the diameter, but excluding the circular boundary) for smooth boundary points. It follows from the definition that the normal $\mathbf{F}_u \times \mathbf{F}_v$ is defined and is nonzero everywhere on the surface. The direction of $\mathbf{F}_u \times \mathbf{F}_v$ at any point of the surface is independent, up to a sign, of the local parameterization $\mathbf{F}$ and is denoted $\mathbf{n}(\mathbf{p})$.

The *silhouette set* for the smooth surface is the set of points $\mathbf{p}$ of the surface such that $(\mathbf{n}(\mathbf{p}) \cdot (\mathbf{p} - \mathbf{c})) = 0$, where $\mathbf{c}$ is the viewpoint. The silhouette is in general a union of flat areas on the surface, curves and points. We isolate flat areas and consider them separately. Isolated silhouette points are unstable, and are not relevant for our purposes. For a surface that does not contain flat areas and is C^2, the silhouette for a general position of the viewpoint can be shown to consist of C^1 non-intersecting curves (silhouette curves).

An important role in our constructions is played by the *curvature* of the surface. More specifically, we are interested in principal curvatures and principal curvature directions. The two principal curvatures at a point $\mathbf{p}$ are maximal and minimal curvatures of the curves obtained by intersecting the surface with a plane passing through $\mathbf{p}$ and containing the normal to the surface. The principal curvature directions are the tangents to the curves for which the maximum and minimum are obtained; these directions are always orthogonal and lie in the tangent plane to the surface. The formulas expressing these quantities in terms of the derivatives of $\mathbf{F}$ are standard and can be found, for example, in [4]. The most important property of the principal curvatures that we use can be formulated as follows: *if a surface has principal curvatures κ_1 and κ_2, and the unit vectors along principal directions and the normal are used to define an orthonormal coordinate system (r, s, t), with r and s parameterizing the tangent plane then locally the surface is the graph of a function over the tangent plane*

$$t = \kappa_1 r^2 + \kappa_2 s^2 + o(r^2 + s^2) \tag{1}$$

It follows that principal curvatures and principal curvature directions locally define the best approximating quadratic surface.

4.1 Silhouettes of Meshes and Smooth Surfaces

The simplest approach to computing the silhouette curves would be to replace the smooth surface with its triangulation and find the silhouette edges of the triangular mesh. However, there are significant differences between the silhouettes of smooth surfaces and their approximating polygonal meshes (Figure 4). For polygonal meshes, complex cusps (Figure 3), where several silhouette chains meet, are stable, that is, do not disappear when the viewpoint is perturbed. Singularities of projections of polyhedra were studied in considerable detail (see a recent paper [2] for pointers); a simple classification in the two-dimensional case, which does not appear to be explicitly described elsewhere, can be found in [1]. For smooth surfaces, the only type of stable singularity is a simple cusp, as it was shown in the classic paper by Whitney [35]. As a consequence, silhouette curves on smooth surfaces are either closed loops, or start and end on feature lines, while on polygonal surfaces they may intersect, and their topology is more complex. Moreover, we observe

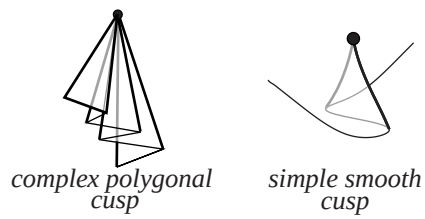


Figure 3: *Left:* Complex cusps are stable on polygonal meshes. *Right:* Only simple cusps are stable on smooth surfaces.

that *no matter how fine the triangulation is, the topology of the silhouette of a polygonal approximation to the surface is likely to be significantly different from that of the smooth surface itself*. This does not present a problem for fixed-resolution images: if the distance between the projected silhouette of the mesh and the projected silhouette of the smooth surface is less than a pixel, the topological details cannot be distinguished. However, if we do want to generate resolution-independent images capturing the essential features of the silhouette of the smooth surface correctly, or apply line styles to the silhouette curves, the polygonal approximation cannot be used. (Figure 4). Similar observations were made in [5].

To preserve the essential topological properties of silhouettes, we compute the silhouette curves using an approach similar to the one used in [14, 17] for spline surfaces. Recall that the silhouette set of a surface is the zero set of the function $g(\mathbf{p}) = (\mathbf{n}(\mathbf{p}) \cdot (\mathbf{p} - \mathbf{c}))$ defined on the surface. The idea is to compute an approximation to this function and find its zero set. For each vertex $\mathbf{p}$ of the polygonal approximation, we compute the true surface normal and $g(\mathbf{p})$ at the vertex. Then the approximation to the function $g(\mathbf{p})$ is defined by linear interpolation of the values of the function. As the resulting function is piecewise-linear, the zero set will consist of line segments inside each triangle of the polygonal approximation. Moreover, we can easily enforce the general position assumption by picking arbitrarily the sign of the function $g(\mathbf{p})$ at vertices where it happens to be exactly zero. As a result, the line segments of the zero set connect points in the interior of the edges of the mesh, and form either closed loops or non-intersecting chains connecting points on the feature lines (Figure 4), similar in structure to the actual silhouette curves. We may miss narrow areas on the surface where the sign is different from surrounding areas. It is easy to see, however, that the silhouette curves we obtain by our method will have the same topology as the silhouette curves of some surface obtained by a small perturbation of the original. This means that we are guaranteed to have a plausible image of a surface, but it may not accurately reflect features of size on the order of the size of a triangle of the approximating mesh in some cases. The silhouette algorithm is described in greater detail in [19].

4.2 Cusp Detection

While the silhouette curves on the surface do not have singularities in a general position, the projected silhouette curves in the image plane do; there is a single stable singularity type, aside from terminating points at feature lines: a simple cusp (Figure 3). The most straightforward way to detect these singularities is to examine the

[1] We do not state rigorous mathematical definitions in complete detail; an interested reader can find them in most standard differential geometry texts.

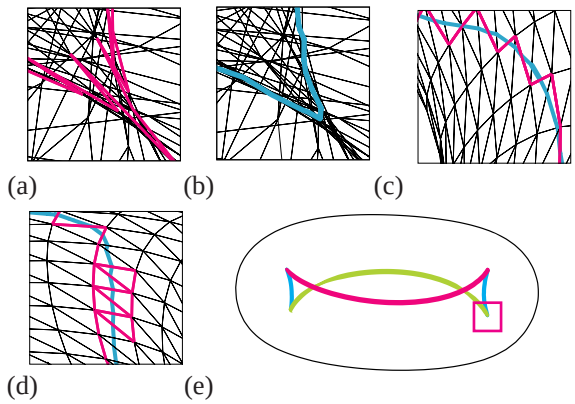


Figure 4: (a) Silhouette edges of a polygonal approximation produce jagged silhouette curves. (b) Our method produces smooth silhouette curves by inferring information about the smooth surface from the polygonal mesh. (c) The same curves shown from another viewpoint and overlayed. (d) A complex cusp occurs in the polygonal approximation when the surface is nearly parallel to the view direction. This does not occur in the smooth silhouette curve. (e) Smooth line drawing of the "smiling torus." The red box shows the location of the curves in (a)-(c).

tangents of the silhouette curves; cusps are the points where the tangent is parallel to the view direction. However, this approach is not numerically reliable, especially if the silhouette curves are approximated by polylines. We propose a new, numerically more robust way to find the cusps, using the following geometric observations.

Consider a silhouette point $\mathbf{p}$ with principal curvature directions $\mathbf{w}_1$ and $\mathbf{w}_2$ and principal curvatures κ_1 and κ_2. Let $\mathbf{c}$ be the viewpoint; since $\mathbf{p}$ is the silhouette point, the viewing direction $\mathbf{v} = \mathbf{c} - \mathbf{p}$ is in the tangent plane. Let $[c_1, c_2, 0]$ be the components of $\mathbf{c}$ with respect to the coordinates (r, s, t) associated with the principal curvature directions, computed by $c_1 = (\mathbf{v} \cdot \mathbf{w}_1)$ and $c_2 = (\mathbf{v} \cdot \mathbf{w}_2)$. As we have observed, $\mathbf{p}$ is a cusp when the tangent to the silhouette at $\mathbf{p}$ is parallel to the viewing direction $\mathbf{v}$. The tangent to the silhouette can easily be expressed in terms of curvature. Approximation (1) yields the following approximation to the normals in a small neighborhood near $\mathbf{p}$: $\mathbf{n}(r, s) = [-2\kappa_1 r, -2\kappa_2 s, 1]$. The equation of the 2nd order approximation to the silhouette curve is an implicit quadratic equation, $g(r, s) = (\mathbf{n}(r, s) \cdot \mathbf{v}(r, s)) = 0$, where $\mathbf{v}(r, s)$ is the viewing direction $\mathbf{c} - \mathbf{p}(r, s) = [c_1 - r, c_2 - s, -\kappa_1 r^2 - \kappa_2 s^2]$. We calculate the vector perpendicular to the silhouette at $\mathbf{p}$ as $\nabla g(0, 0) = [-2\kappa_1 c_1, -2\kappa_2 c_2]$. The resulting condition for the viewing direction to be parallel to the silhouette tangent (or, equivalently, perpendicular to $\nabla g(0, 0)$) to the viewing direction is $\kappa_1 c_1^2 + \kappa_2 c_2^2 = 0$. Therefore, we can define a parameterization-independent scalar function on the surface which we call the *cusp function*:

$$C(\mathbf{p}) = \kappa_1 (\mathbf{v} \cdot \mathbf{w}_1)^2 + \kappa_2 (\mathbf{v} \cdot \mathbf{w}_2)^2$$

where all quantities are evaluated at point $\mathbf{p}$. This function has the following important property: *cusps are contained in the intersection set of the two families of curves: one obtained as the zero set of the function $g(\mathbf{p})$, the other as the zero set of the cusp function $C(\mathbf{p})$* (Figure 5). The zero set of $C(\mathbf{p})$ can be approximated in the same way as the zero set of $g(\mathbf{p})$; each triangle of the polygonal mesh may contain a single line segment approximating the zero set of $C(\mathbf{p})$ and another approximating the zero set of $g(\mathbf{p})$. This allows us to compute approximate cusp locations robustly, without introducing many spurious cusps, and at the same time using relatively coarse polygonal approximations to the smooth surface.

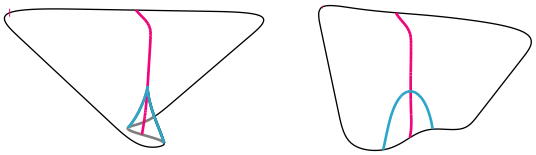


Figure 5: *Left:* Cusps are found as intersections of zero sets of two functions defined on the surface, the dot product of the normal with the viewing direction and the cusp function. The silhouette curve is shown in blue, the cusp zero set in red. *Right:* The same curves; view from a viewpoint different from the one that was used to compute the curves.

4.3 Fast Silhouette Detection

In the previous section, we have presented an algorithm for constructing approximations to the silhouette curves which, when implemented in the simplest way, requires complete traversal of the mesh. Such a traversal is unnecessary; typically, only a small percentage of mesh faces contain silhouettes [25, 23]. For polygonal meshes, a number of fast techniques were developed that allow one to avoid complete traversal. A stochastic algorithm was proposed in [25]. A deterministic algorithm based on the Gauss map was proposed in [3, 18], but is restricted to orthographic projection. We present a new deterministic algorithm for accelerated location of silhouettes, which works for both orthographic and perspective projection. This algorithm is equally suitable for finding silhouettes defined as zero sets, and for finding silhouette edges of polygonal meshes.

Our algorithm is based on the concept of *dual surfaces.* The points of the dual surface M' are the images of the tangent planes to a surface M under a duality map, which maps each plane $Ax + By + Cz + D = 0$ to the homogeneous point $[A, B, C, D]$. More explicitly, M' can be obtained by mapping each point of M to a homogeneous point $\mathbf{N} = [n_1, n_2, n_3, -(\mathbf{p} \cdot \mathbf{n})]$, where $\mathbf{n} = [n_1, n_2, n_3, 0]$ is the unit normal at $\mathbf{p}$. Note that the inverse is also true: each plane in the dual space corresponds to a point in the primal space. Let $\mathbf{C} = [c_1, c_2, c_3, c_4]$ be our viewpoint in the homogeneous form. Then the silhouette of the surface consists of all points $\mathbf{p}$ for which $\mathbf{C}$ is in the tangent plane at that point. For perspective projection, this means that $(\mathbf{C} \cdot \mathbf{N}) = (\mathbf{c} - \mathbf{p}) \cdot \mathbf{n} = 0$. For orthographic projection, the homogeneous formula is the same: $(\mathbf{C} \cdot \mathbf{N}) = (\mathbf{c} \cdot \mathbf{n}) = 0$, where $\mathbf{c}$ is interpreted as the view direction. Our algorithm is based on the following observation: *the image of the silhouette set of the surface with respect to the viewpoint $\mathbf{C}$ under the duality map is the intersection of the plane $(\mathbf{C} \cdot \mathbf{x}) = 0$, with the dual surface.* This fact allows us to reduce the problem of finding the silhouette to the problem of intersecting a plane with a surface (Figure 7), for which many space-partition-based acceleration techniques are available. However, an additional complication is introduced by the fact that some points of the dual surface may be at infinity. This does not allow us to consider only the finite part of the projective space, which can be identified with $\mathbf{R}^3$. However we can identify the whole 3D projective space with points of the unit hypersphere S^3, or, equivalently, of the boundary of a hypercube, in four-dimensional space. As four-dimensional space is somewhat difficult to visualize, we show the idea of the algorithm on a 2D example in Figure 6. In the 2D case, the problem is to compute all *silhouette points* on a curve, that is, the points for which the tangent line contains the viewpoint.

While the geometric background is somewhat abstract, the actual algorithm is quite simple. The input to the algorithm is a polygonal mesh, with normals specified at vertices, if we are computing silhouettes using zero-crossings. The normals are not necessary if we are locating the silhouette edges of the polygonal mesh. There are two parts to the algorithm: initialization of the spatial partition and

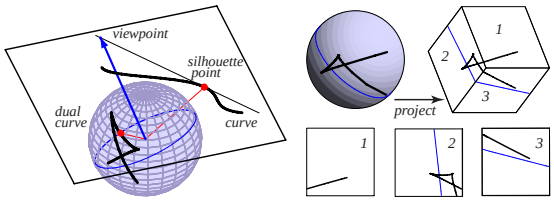


Figure 6: *Left*: Using a dual curve to find silhouette points. The figure shows a curve in the plane $z = 1$ and its dual on a sphere. The blue arrow is the vector c from the origin in 3D to the viewpoint in the plane, the blue circle is the intersection of the plane passing through the origin perpendicular to c with the unit sphere. The red points are a silhouette point and its dual. The silhouette point can be found by intersecting the blue circle with the dual curve and retrieving corresponding point on the original curve. *Right*: Reducing the intersection problem to planar subproblems. The upper hemisphere containing the dual curve is projected on the surface of cube and at most 5 (in this case 3) planar curve-line intersection problems are solved on the faces.

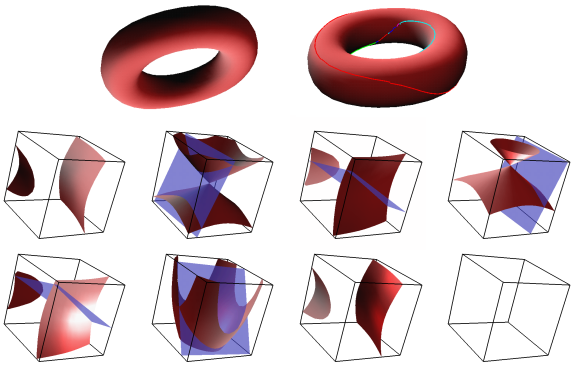


Figure 7: Silhouette lines under the duality map correspond to the intersection curve of a plane with the dual surface. *Top:* Torus shown from camera and side views. *Bottom:* The eight 3D faces of the hypercube, seven of which contain portions of the dual surface. The viewpoint dual is shown as a blue plane. Silhouettes occur at the intersection of the dual plane with the dual surface.

intersection of the dual surface with the plane corresponding to the viewpoint. The second part is fairly standard, so we focus on the first part.

Step 1: For each vertex $\mathbf{p}$ with normal $\mathbf{n}$, we compute the dual position $\mathbf{N} = [n_1, n_2, n_3, -(\mathbf{p} \cdot \mathbf{n})]$. The dual positions define the dual mesh which has different vertex positions but the same connectivity.

Step 2: Normalize each dual position $\mathbf{N}$ using l_∞-norm, that is, divide by $\max(|N_1|, |N_2|, |N_3|, |N_4|)$. After division, at least one of the components N_i, $i = 1..4$, becomes 1 or -1. The resulting four-dimensional point is on the surface of the unit hypercube. The three-dimensional face of the cube on which the vertex is located is determined by the index and sign of the maximal component.

Step 3: Each triangle of the dual mesh is assigned to a list for every three-dimensional face in which it has a vertex.

Step 4: An octtree is constructed for each three-dimensional face, and the triangles assigned to this face are placed into the octtree.

The second step of the algorithm, which is repeated for each frame, uses the octtree to find the silhouette edges for a given camera position by intersecting the dual plane with the dual surface.

We have implemented an interactive silhouette viewer based on the dual space method. In our tests, silhouette tests were performed on twice as many triangles as there were actual triangles containing silhouettes, suggesting that performance is roughly linear in the number of silhouette triangles. This represents a substantial speedup over traversing the entire mesh. Silhouette edge detection and visibility calculations on the three-times subdivided Venus model (∼90,000 triangles) can be performed at approximately 17 frames per second on a 225 MHz SGI Octane, without using graphics hardware, which is similar to the performance of the nondeterministic algorithm of [25].

4.4 Visibility

Before computing visibility, we separate the silhouette curves into segments. Visibility is determined for each segment. The following points are used to separate segments: cusps, silhouette-feature joints, and inverse images of silhouette-feature and silhouette-silhouette intersections in image space. Visibility can change only at these points, thus each segment is either completely visible or invisible.

Determining visibility is fundamentally difficult for smooth surfaces, because it cannot be inferred precisely from visibility of the approximating mesh. Our algorithm can only guarantee that the correct visibility will be produced if the mesh is sufficiently fine, using a theoretically-estimated required degree of refinement. However, the estimate is too conservative and difficult to compute to be practical; in our implementation, we refine the mesh to a fixed subdivision level.

Our visibility algorithm is based on the following observation: at any area on the surface, the rate of change of the normal is bounded by the maximal directional curvature. For a sufficiently fine triangulation, one can guarantee that for any triangle for which $(\mathbf{n} \cdot (\mathbf{p} - \mathbf{c}))$ changes sign, there is a silhouette edge of the polygonal approximation adjacent to a vertex of the triangle. We use the visibility of these edges to compute visibility of the silhouette curves. The visibility of the silhouette edges can be determined using known techniques (e.g. [25]).

For each curve we find visibility of all nearby silhouette edges (which is not necessarily consistent) and use the visibility of the majority of the edges to determine visibility of the chain. It is possible to show that this method will produce correct visibility for sufficiently fine meshes in the following sense: there is a smooth surface for which the precise projection has the same topology as the one computed by our method.

In practice, we have found that the algorithm performs well even without extra refinement near the silhouettes, provided that the original mesh is sufficiently close to the surface. An efficient algorithm with better-defined properties would be useful.

5 Direction Fields on Surfaces

Fields on surfaces. To generate hatches, we need to choose several direction fields on visible parts of the surface. The direction fields are different from the more commonly used vector fields: unlike a vector field, a direction field does not have a magnitude and does not distinguish between the two possible orientations.

The fields can either be defined directly in the image plane as in [31], or defined on the surface and then projected. The advantage of the former method is that the field needs to be defined and continuous only in each separate area of the image. However, it is somewhat more difficult to use the information about the shape of the objects when constructing the field, and the field must be recomputed for each image. We choose to generate the field on the surface first.

A number of different fields on surfaces have been used to define hatching directions. The most commonly-used field is probably the field of isoparametric lines; this method has obvious limitations,

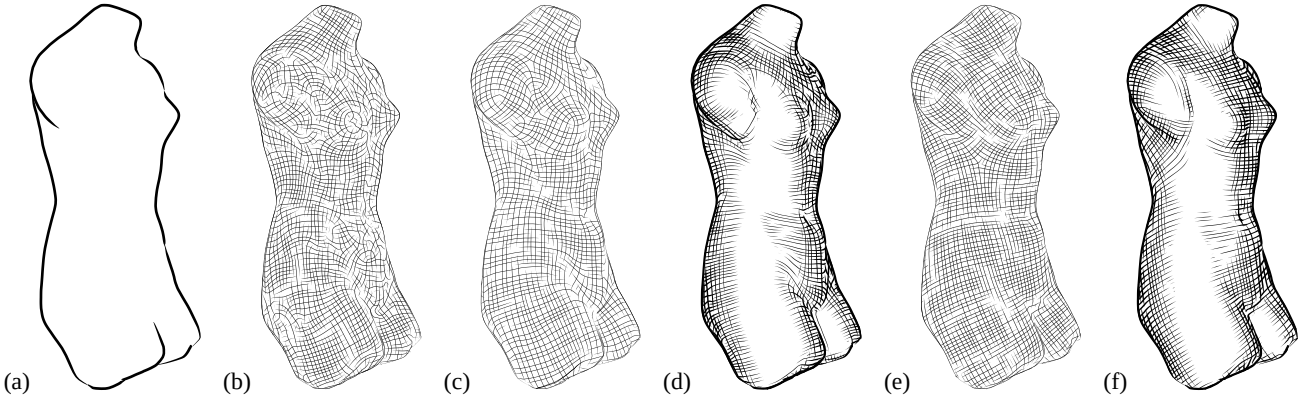


Figure 8: Direction fields on the Venus. (a) Silhouettes alone do not convey the interior shape of the surface. (b) Raw principle curvature directions produce an overly-complex hatching pattern. (c) Smooth cross field produced by optimization. Reliable principal curvature directions are left unchanged. Optimization is initialized by the principal curvatures. (d) Hatching with the smooth cross field. (e) Very smooth cross field produced by optimizing all directions. (f) Hatching from the very smooth field.

as the parameterization may be very far from isometric, and is not appropriate for surfaces lacking a good natural parameterization, such as subdivision surfaces and implicit surfaces. The successes and failures of this approach provide valuable clues for construction of fields for hatching.

The most natural geometric candidate is the pair of principal curvature direction fields [13, 21]. corresponding to the minimal and maximal curvatures[2]. We will refer to the integral lines of these fields as *curvature lines*. These fields do not depend on parameterization, capture important geometric features, and are consistent with the most common two-directional hatching pattern. However, they suffer from a number of disadvantages. All umbilical points (points with coinciding principal curvatures) are singularities, which means that the fields are not defined anywhere on a sphere and have arbitrarily complex structure on surfaces obtained by small perturbations of a sphere. On flat areas (when both curvatures are very small) the fields are likely to result in a far more complex pattern than the one that would be used by a human.

Other candidates include isophotes (lines of constant brightness) and the gradient field of the distance to silhouette or feature lines [25, 12]. Both are suitable for hatching in a narrow band near silhouettes or feature lines, but typically do not adequately capture shape further from silhouettes, nor are they suitable for cross-hatching.

Our approach is based on several observations about successes and failures of existing methods, as well as hatching techniques used by artists.

• *Cylindric surfaces.* Surface geometry is rendered best by principal curvature directions on cylindrical surfaces, that is, surfaces for which one of the principal curvatures is zero (all points of the surface are parabolic). This fact is quite remarkable: psychophysical studies confirm that even a few parallel curves can create a strong impression of a cylindrical surface with curves interpreted as principal curvature lines [32, 24]. Another important observation is that for cylinders the principal curvature lines are also geodesics, which is not necessarily true in general. Hatching following the principal curvature directions fails when the ratio of principal curvatures is close to one.
Deussen et al. [9] uses intersections of the surface with planes to obtain hatch directions; the resulting curves are likely to be locally close to geodesics on slowly varying surfaces.

• *Isometric parameterizations.* Isoparameteric lines work well as curvature directions when a parameterization exists and is close to isometric, i.e. minimizes the metric distortion as described in, for example, [10, 27]. In this case, parametric lines are close to geodesics. Isoparametric lines were used by [36, 11].

• *Artistic examples.* We observe that artists tend to use relatively straight hatch lines, even when the surface has wrinkles. Smaller details are conveyed by varying the density and the number of hatch directions (Figure 9).

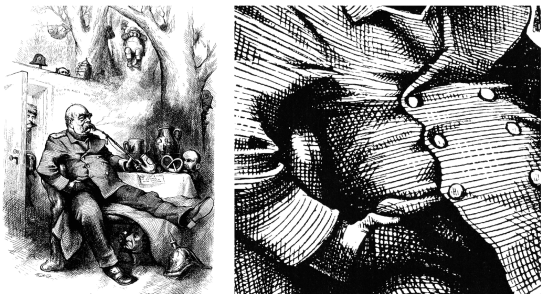


Figure 9: Almost all hatches in this cartoon by Thomas Nast curve only slightly, while capturing the overall shape of the surface. Note that the hatches often appear to follow a cylinder approximating the surface. Small details of the geometry are rendered using variations in hatch density.

These observations lead to the following simple requirements for hatching fields: *in areas where the surface is close to parabolic, the field should be close to principal curvature directions; on the whole surface, the integral curves of the field should be close to geodesic.* In addition, if the surface has small details, the field should be generated using a smoothed version of the surface.

Cross fields. While it is usually possible to generate two global direction fields for the two main hatch directions, we have observed that this is undesirable in general. There are two reasons for this: first, if we would like to illustrate nonorientable surfaces, such fields may not exist. Second, and more importantly, there are natural cross-hatching patterns that cannot be decomposed into two smooth fields even locally (Figure 10). Thus, we consider *cross fields*, that is, maps defined on the surface, assigning an unordered pair of perpendicular directions to each point.

Constructing Hatching Fields. Our algorithm is based on the considerations above and proceeds in steps.

[2] It is possible to show that for a surface in general position, these fields are always globally defined, excluding a set of isolated singularities.

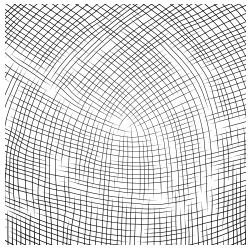


Figure 10: A cross-hatching pattern produced by our system on a smooth corner. This pattern cannot be decomposed into two orthogonal smooth fields near the corner singularity. The analytic expression for a similar field in the plane is $v_1(r,\theta) = [\cos(\theta/4), \sin(\theta/4)]$; $v_2(r,\theta) = [-\sin(\theta/4), \cos(\theta/4)]$. This field is continuous and smooth only if we do not distinguish between v_1 and v_2.

Step 1. Optionally, create a smoothed copy of the original mesh. The copy is used to compute the field. The amount of smoothing is chosen by the user, with regard to the smoothness of the original mesh, and the scale of geometric detail the user wishes to capture in the image. For example, no smoothing might be necessary for a close-up view of a small part of a surface, while substantial smoothing may be necessary to produce good images from a general view; in practice we seldom found this to be necessary.

Step 2. Identify areas of the surface which are sufficiently close to parabolic, that is, the ratio of minimal to maximal curvature is high, and at least one curvature is large enough to be computed reliably. Additionally, we mark as unreliable any vertex for which the average cross field energy of its incident edges exceeds a threshold, in order to allow optimization of vertices that begin singular.

Step 3. Initialize the field over the whole surface by computing principal curvature directions. If there are no quasi-parabolic areas, user input is required to initialize the field.

Step 4. Fix the field in quasi-parabolic areas and optimize the field on the rest of the vertices, which were marked as unreliable. This step is of primary importance and we describe it in greater detail.

Our optimization procedure is based on the observation that we would like the integral lines of our field to be close to geodesics. We use a similar, but not identical, requirement that the field is as close to constant as possible. Minimizing the angles between the world-space directions at adjacent vertices of the mesh is possible, but requires constrained optimization to keep the directions in the tangent planes. We use a different idea, based on establishing a correspondence between the tangent planes at different points of the surface, which, in some sense, corresponds to the minimal possible motion of the tangent plane as we move from one point to another. Then we only need to minimize the change of the field with respect to the corresponding directions in the tangent planes.

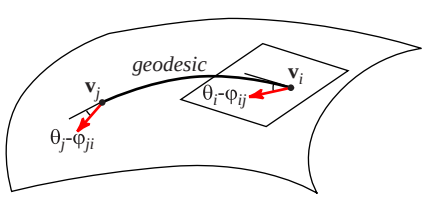


Figure 11: Moving vectors along geodesics.

Given two sufficiently close points $\mathbf{p}_1$ and $\mathbf{p}_2$ on a smooth surface, a natural way to map the tangent plane at $\mathbf{p}_1$ to the tangent plane at $\mathbf{p}_2$ is to transport vectors along the geodesics (Figure 11); for sufficiently close points there is a unique geodesic $\gamma(t)$, $t = 0..1$, connecting these points. This is done by mapping a unit vector $\mathbf{u}_1$ in the tangent plane at $\mathbf{p}_1$ to a unit vector $\mathbf{u}_2$ in the tangent plane at $\mathbf{p}_2$, such that the angle between $\mathbf{u}_1$ and the tangent to the geodesic $\gamma'(0)$ is the same as the angle between $\mathbf{u}_2$ and $\gamma'(1)$. In discrete case, for adjacent vertices of the approximating mesh $\mathbf{v}_i$ and $\mathbf{v}_j$, we approximate the tangents to the geodesic by the projections of the edge $(\mathbf{v}_i, \mathbf{v}_j)$ into the tangent planes at the vertices. Let the directions of these projections be $\mathbf{t}_{ij}$ and $\mathbf{t}_{ji}$. Then a rigid transformation T_{ij} between the tangent planes is uniquely defined if we require that $\mathbf{t}_{ij}$ maps to $\mathbf{t}_{ji}$ and that the transformation preserves orientation. Then for any pair of tangent unit vectors $\mathbf{w}_i$ and $\mathbf{w}_j$ at $\mathbf{v}_i$ and $\mathbf{v}_j$ respectively, we can use $\|T_{ij}\mathbf{w}_i - \mathbf{w}_j\|$ to measure the difference between directions. One can show that the value of this expression is the same as $\|T_{ji}\mathbf{w}_j - \mathbf{w}_i\|$. To measure the difference between the values of the cross field at two points, we choose a unit tangent vector for each point. The vectors are chosen along the directions of the cross field. There are four possible choices at each point. We choose a pair of unit vectors for which the difference is minimal.

We now explicitly specify the energy functional. The cross field is described by a single angle θ_i for each vertex $\mathbf{v}_i$, which is the angle between a fixed tangent direction $\mathbf{t}_i$, and one of the directions of the cross field; we do not impose any limitations on the value of θ_i, and there are infinitely many choices for θ_i differing by $n\pi/2$ that result in the same cross field. Let φ_{ij} be the direction of the projection of the edge $(\mathbf{v}_i, \mathbf{v}_j)$ into the tangent plane at $\mathbf{v}_i$. Using this choice of coordinates, one can show that the quantity $\|T_{ij}\mathbf{w}_i - \mathbf{w}_j\|$ is equal to $\min_k \sqrt{2 - 2\cos((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}) + k\pi/2)}$. Minimization of this quantity is equivalent to minimization of $E(i,j) = \min_k (-\cos((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}) + k\pi/2))$, which is not differentiable. We observe, however, that $E_0(i,j) = -8E(i,j)^4 + 8E(i,j)^2 - 1$ is just $-\cos 4((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}))$, and is a monotonic function of $E(i,j)$ on $[\sqrt{2}/2..1]$, the range of possible values of $E(i,j)$. Thus, instead of minimizing $E(i,j)$, we can minimize $E_0(i,j)$. We arrive at the following simple energy:

$$E_{\text{field}} = -\sum_{\text{all edges } (\mathbf{v}_i, \mathbf{v}_j)} \cos 4((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}))$$

which does not require any constraints on the variables θ_i. Note that the values φ_{ij} are constant. Due to the simple form of the functional, it can be minimized quite quickly. We use a variant of the BFGS conjugate gradient algorithm described in [37] to perform minimization. For irregularly-sampled meshes, the energy may also be weighted in inverse proportion to edge length. We have not found this to be necessary for the meshes used in this paper. The result of the optimization depends on the threshold chosen to determine which vertices are considered unreliable; in the extreme cases, all vertices are marked as unreliable and the whole field is optimized, or all vertices are marked as reliable and the field remains unoptimized. Figure 8 shows the results for several thresholds.

6 Rendering Style

6.1 Style Rules

Our rendering style is based to some extent on the rules described by G. Francis in *A Topological picturebook* [15], which are in turn based on Nikolaïdes' rules for drawing drapes [26]. We have also used our own observations of various illustrations in similar styles. We begin our style description by defining undercuts and folds. A visible projected silhouette curve separates two areas of the image: one containing the image of the part of the surface on which the curve is located, the other empty or containing the image of a different part of the surface. We call the former area a *fold*. If the

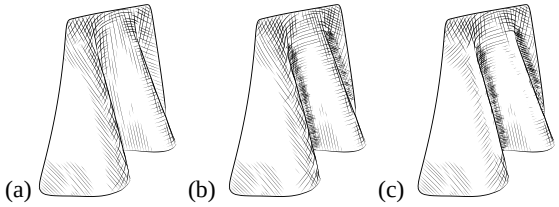


Figure 12: Hatching rules shown on drapes. (a) There are 3 main discrete hatch densities: highlights, midtones, and shadows, corresponding to 0, 1, and 2 directions of hatches. (b) Undercuts. (c) "Mach bands." Undercuts and Mach bands increase contrast where surfaces overlap.

latter area contains the image of a part of the surface, we call it an *undercut*.

We use the following rules, illustrated in Figure 12.

- The surface is separated into four levels of hatching: highlights and Mach bands (no hatching), midtones (single hatching), shadowed regions (cross-hatching), and undercuts (dense cross-hatching). Inside each area, the hatch density stays approximately uniform. The choice of the number of hatch directions used at a particular area of the surface is guided by the lighting and the following rules:
- If there is an undercut, on the other side of the silhouette from a fold, a thin area along the silhouette on the fold side is not hatched ("Mach band effect").
- Undercuts are densely hatched.
- Hatches are approximately straight; a hatch is terminated if its length exceeds a maximum, or if its direction deviates from the original by more than a fixed angle.
- Optionally, hatch thickness within each density level can be made inversely proportional to lighting; the resulting effect is rather subtle, and is visible only when the hatches are relatively thick.

6.2 Hatch Placement

The hatching procedure has several user-tunable parameters: basic hatch density specified in image space; the hatch density for undercuts; the threshold for highlights (the areas which receive no hatching); the threshold that separates single hatch regions from cross hatch regions; the maximum hatch length; the maximum deviation of hatches from the initial direction in world space. Varying these parameters has a considerable effect both on the appearance of the images and on the time required by the algorithm. Threshold values are usually chosen to divide the object more or less evenly between different hatching levels.

Once we have a hatching field, we can illustrate the surface by placing hatches along the field. We first define three intensity regions over the surface: no hatching (highlights and Mach bands), single hatching (midtones), and cross hatching (shadowed regions). Furthermore, some highlight and hatch regions may be marked as undercut regions. The hatching algorithm is as follows:

1. Identify Mach bands and undercuts.
2. Cover the single and cross hatch regions with cross hatches, and add extra hatches to undercut regions.
3. Remove cross-hatches in the single hatch regions, leaving only one direction of hatches.

6.3 Identifying Mach Bands and Undercuts

In order to identify Mach bands and undercuts, we step along each silhouette and boundary curve. A ray test near each curve point is used to determine if the fold overlaps another surface. Undercuts and Mach bands are indicated in a 2D grid, by marking every grid cell within a small distance of the fold on the near side of the surface as a Mach band, and by marking grid cells on the far side of the surface within a larger distance as undercuts. (This is the same 2D grid as used for hatching in the next section.)

6.4 Cross-hatching

We begin by creating evenly-spaced cross-hatches on a surface. We adapt Jobard and Lefers' method for creating evenly-spaced streamlines of a 2D vector field [22]. The hatching algorithm allows us to place evenly-spaced hatches on the surface in a single pass over the surface.

Our algorithm takes two parameters: a desired hatch separation distance d_{sep}, and a test factor d_{test}. The separation distance indicates the desired image-space hatch density; a smaller separation distance is used for undercuts. The algorithm creates a queue of surface curves, initially containing the critical curves (silhouettes, boundaries, creases, and self-intersections). While the queue is not empty, we remove the front curve from the queue and seed new hatches along it at points evenly-spaced in the image. Seeding creates a new hatch on the surface by tracing the directions of the cross-hatching field. Since the cross field is invariant to 90 degree rotations, at each step the hatch follows the one of four possible directions which has the smallest angle with the previous direction. Hatches are seeded perpendicular to all curves. Hatches are also seeded parallel to other hatches, at a distance d_{sep} from the curve. A hatch continues along the surface until it terminates in a critical curve, until the world-space hatch direction deviates from the initial hatch direction by more than a constant, or until it comes near a parallel hatch. This latter condition occurs when the endpoint of the hatch $\mathbf{p}_1$ is near a point $\mathbf{p}_2$ on another hatch, such that the following conditions are met:

- $||\mathbf{p}_1 - \mathbf{p}_2|| < d_{test} d_{sep}$, measured in image space.
- A straight line drawn between the two points in image space does not intersect the projection of any visible critical curves. In other words, hatches do not "interfere" when they are not nearby on the surface.
- The world space tangents of the two hatch curves are parallel, i.e. the angle between them is less than 45 degrees, after projection to the tangent plane at $\mathbf{p}_1$.

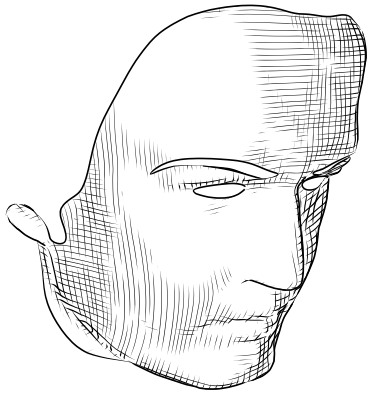
The search for nearby hatches is performed by placing all hatches in a 2D grid with grid spacing equal to d_{sep}. This ensures that at most nine grid cells must be searched to detect if there are hatches nearby the one being traced.

6.5 Hatch Reduction

Once we have cross-hatched all hatch regions, we remove hatches from the single hatch regions until they contain no cross-hatches. By removing hatches instead of directly placing single a hatch direction, we avoid the difficulty inherent in producing a consistent vector field on the surface. Our algorithm implicitly segments the visible single-hatch regions into locally-consistent single hatching fields. This allows us to take advantage of the known view direction and the limited extent of these regions.

The reduction algorithm examines every hatch on the surface and deletes any hatch that is perpendicular to another hatch. In particular, a hatch is deleted if it contains a point $\mathbf{p}_1$ nearby a point $\mathbf{p}_2$ on another hatch such that:

- $\mathbf{p}_1$ and $\mathbf{p}_2$ lie within the single hatch region.
- $||\mathbf{p}_1 - \mathbf{p}_2|| < 2d_{sep}$, measured in image space.
- A straight line drawn between the two points in image space does not intersect any visible critical curve.
- The world space tangents of the two hatch curves are perpendicular, i.e. the angle between them is greater than 45 degrees after projection to the tangent plane at $\mathbf{p}_1$.



Deleting a hatch entails clipping it to the cross-hatch region; the part of the hatch that lies within the cross-hatch region is left untouched.

The order in which hatches are traversed is important; a naïve traversal order will usually leave the single hatch region uneven and inconsistent. We perform a breadth-first traversal to prevent this. A queue is initialized with a hatch curve. While the queue is not empty, the front curve is removed from the queue. If it is perpendicular to another curve in the single hatch region, then the curve is deleted, and all parallel neighbors of the hatch that have not been visited are added to the queue. When the queue is empty, a hatch that has not yet been visited is added to the queue, if any remain. The tests for perpendicular is as described above; the angle condition is reversed for the parallel test.

7 Results and Conclusions

Most of the illustrations in this paper were created using our system. Figures 1, 8 demonstrate the results for relatively fine meshes that define surfaces with complex geometry. Figures 2 and 13 show the results of using our system to illustrate several mathematical surfaces.

The time required to create an illustration varies greatly; while silhouette drawings can be computed interactively, and the field optimization takes very little time, hatching is still time-consuming, and can take from seconds to minutes, depending on hatch density and complexity of the model. Also, for each model the parameters of the algorithms (thresholds for hatching, position of the light sources, hatch density) have to be carefully chosen;

Future work. As we have already mentioned, improvements should be made to the silhouette visibility algorithm. Performance was not our goal for the hatching algorithm. It is clear that substantial speedups are possible. While the quality of fields generated by our algorithms is quite good, it would be desirable to reduce the number of parameters that may be tuned.

A more fundamental problem is the lack of control over the the number, type and placement of singularities of the generated field. As most surfaces of interest have low genus, the number of singularities can be very small for most surfaces.[3] However, the user currently has little control over their placement and additional support must be provided. Furthermore, the hatch reduction algorithm could be made more robust to irregular cross-hatching patterns, and the hatching could be improved reduce hatching artifacts, perhaps by employing the optimization technique of Turk and Banks [33].

[3]The relation between the numbers of singularities of different types is determined by the analogs of Euler formula; such formulas are known for vector and tensor fields; obtaining classification of singularities and a formula of this type for the cross fields described in the paper is an interesting mathematical problem.

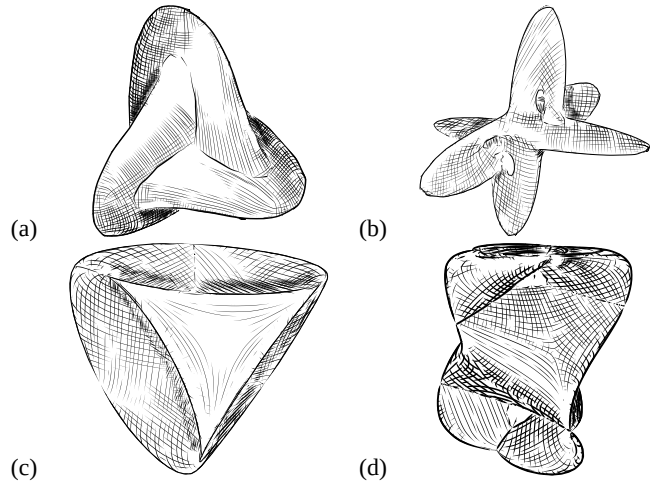


Figure 13: Several surfaces generated using G. Francis' generalization of Apéry's Romboy homotopy [16]. (a) Boy surface; (b) "Ida"; (c) Roman surface; (d) Etruscan Venus.

Acknowledgments

Our special thanks go to Jianbo Peng who implemented the dual surface silhouette detection algorithm. We are grateful to Pat Hanrahan, who suggested this research topic to us. We thank the anonymous reviewers for their comments. Chris Stolte participated in the project in its early stages.

This research was supported in part by the NSF grant DGE-9454173 and the NYU Center for Advanced Technology.

A C^2-surfaces based on subdivision

Commonly used subdivision surfaces, such as variants of Loop subdivision, produce either surfaces with curvatures that do not converge or have zero curvature at extraordinary vertices. There are fundamental reasons for this [29]. This property is rather undesirable, if we would like to compute silhouette curves, as it means either flat points or singular behavior near extraordinary points. We have developed a surface representation based on subdivision that produces surfaces that are everywhere C^2, do not have zero curvature at extraordinary vertices, and agree arbitrarily well with the limit surfaces produced by subdivision. This representation is described elsewhere [38]. However, for our purposes it is sufficient to have a way to compute curvatures for the surface associated with a mesh, and it is not necessary to have a complete surface evaluation algorithm.

The curvature computation that we propose is based on ideas from subdivision and is compatible with the curvature computations for subdivision surfaces in the regular case.

Consider a vertex $\mathbf{v}$ of the initial mesh of valence k. We will regard a part of the smooth surface corresponding to the 1-neighborhood of $\mathbf{v}$ as parameterized over a regular k-gon in the plane. Introduce the polar coordinates (r, φ) in the plane, with $u = r\cos\varphi$ and $v = r\sin\varphi$. then the second-order approximation to the surface can be written as

$$a_0 + (a_{11}\sin\varphi + a_{12}\cos\varphi)r + (a_{20} + a_{21}\sin 2\varphi + a_{22}\cos 2\varphi)r^2$$

A simple calculation shows that the least squares fit to $k+1$ points of the 1-neighborhood $p_0 \dots p_k$ assumed to be values at $(\sin(2\pi i/k), \cos(2\pi i/k))$, $i = 0..k$. with p_0 in the center, leads to

$$a_0 = p_0; \quad a_{20} = -p_0 + \frac{1}{k}\sum_i p_i$$

$$a_{11} = \frac{2}{k}\sum_i p_i \sin\frac{2\pi i}{k}; \quad a_{12} = \frac{2}{k}\sum_i p_i \cos\frac{2\pi i}{k}$$

$$a_{21} = \frac{2}{k}\sum_i p_i \sin\frac{4\pi i}{k}; \quad a_{22} = \frac{2}{k}\sum_i p_i \cos\frac{4\pi i}{k}$$

Note that the formulas for a_{11} and a_{21} coincide with the standard formulas for the tangents to the Loop subdivision surface, and a_{20}, a_{21}, a_{22}, with appropriate variable changes, produce second derivatives in the regular case. To make our calculations compatible with the Loop surface, we replace $a_0 = p_0$ with $a_0 = p_0^{limit}$, the limit position of the control point p_0. As a result, we obtain a set of simple rules for computing the coefficients of an approximating quadratic surface, which, after appropriate change of variables can be used to compute curvatures and is compatible with the Loop subdivision rules. In [38], we show that one can construct a C^2 surface which has precisely these curvatures at the vertices. A similar construction works for the boundary case. We should note that for valences $k = 3, 4$, the coefficients of the quadric are not independent, and thus not all possible local behaviors can be approximated well.

Given known partial derivatives $\mathbf{F}_u, \mathbf{F}_v, \mathbf{F}_{uu}, \mathbf{F}_{uv}, \mathbf{F}_{vv}$ of the local parameterization of the surface, the principal curvature directions and magnitudes can be computed as eigenvalues and eigenvectors of the following matrix:

$$\begin{pmatrix} E & F \\ F & G \end{pmatrix} \begin{pmatrix} L & M \\ M & N \end{pmatrix} \tag{2}$$

where $E = (\mathbf{F}_u \cdot \mathbf{F}_u)$, $F = (\mathbf{F}_v \cdot \mathbf{F}_u)$, $G = (\mathbf{F}_v \cdot \mathbf{F}_v)$, $L = (\mathbf{F}_{uu} \cdot \mathbf{n})$, $M = (\mathbf{F}_{uv} \cdot \mathbf{n})$, $N = (\mathbf{F}_{vv} \cdot \mathbf{n})$.

References

[1] I. A. Babenko. Singularities of the projection of piecewise-linear surfaces in $\mathbf{r}\mathrm{span}3$. *Vestnik Moskov. Univ. Ser. I Mat. Mekh.*, 1991(2):72–75.

[2] Thomas F. Banchoff and Ockle Johnson. The normal Euler class and singularities of projections for polyhedral surfaces in 4-space. *Topology*, 37(2):419–439, 1998.

[3] Fabien Benichou and Gershon Elber. Output sensitive extraction of silhouettes from polygonal geometry. *Pacific Graphics '99*, October 1999. Held in Seoul, Korea.

[4] William M. Boothby. *An Introduction to Differentiable Manifolds and Riemannian Geometry*. Academic Press, 1986.

[5] Wagner Toledo Corrêa, Robert J. Jensen, Craig E. Thayer, and Adam Finkelstein. Texture Mapping for Cel Animation. In *SIGGRAPH 98 Conference Proceedings*, pages 435–446, July 1998.

[6] Cassidy Curtis. Loose and Sketchy Animation. In *SIGGRAPH 98: Conference Abstracts and Applications*, page 317, 1998.

[7] Philippe Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Technical Report 2919, INRIA, June 1996.

[8] Thierry Delmarcelle and Lambertus Hesselink. The topology of symmetric, second-order tensor fields. In *Visualization '94*, pages 140–147, October 1994.

[9] Oliver Deussen, Jörg Harnel, Andreas Raab, Stefan Schlechtweg, and Thomas Strothotte. An Illustration Technique Using Hardware-Based Intersections. *Graphics Interface '99*, pages 175–182, June 1999.

[10] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. In *Computer Graphics Proceedings*, Annual Conference Series, pages 173–182. ACM Siggraph, 1995.

[11] Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, September 1995.

[12] Gershon Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January – March 1998.

[13] Gershon Elber. Interactive line art rendering of freeform surfaces. *Computer Graphics Forum*, 18(3):1–12, September 1999.

[14] Gershon Elber and Elaine Cohen. Hidden Curve Removal for Free Form Surfaces. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 95–104, August 1990.

[15] George K. Francis. *A Topological Picturebook*. Springer-Verlag, New York, 1987.

[16] George K. Francis. The Etruscan Venus. In P. Concus, R. Finn, and D. A. Hoffman, editors, *Geometric Analysis and Computer Graphics*, pages 67–77. 1991.

[17] Amy Gooch. Interactive Non-Photorealistic Technical Illustration. Master's thesis, University of Utah, December 1998.

[18] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[19] Aaron Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In Stuart Green, editor, *Non-Photorealistic Rendering*, SIGGRAPH Course Notes. 1999.

[20] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Huber Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In *Computer Graphics Proceedings*, Annual Conference Series, pages 295–302. ACM Siggraph, 1994.

[21] Victoria L. Interrante. Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution. In *SIGGRAPH 97 Conference Proceedings*, pages 109–116, August 1997.

[22] Bruno Jobard and Wilfrid Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proc. of 8th Eurographics Workshop on Visualization in Scientific Computing*, pages 45–55, 1997.

[23] Lutz Kettner and Emo Welzl. Contour Edge Analysis for Polyhedron Projections. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 379–394. Springer Verlag, 1997.

[24] Pascal Mamassian and Michael S. Landy. Observer biases in the 3D interpretation of line drawings. *Vision Research*, (38):2817—2832, 1998.

[25] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, pages 415–420, August 1997.

[26] Kimon Nikolaïdes. *The Natural Way to Draw*. Houghton Miffin, Boston, 1975.

[27] Hans Køhling Pedersen. A Framework for Interactive Texturing on Curved Surfaces. *Proceedings of SIGGRAPH 96*, pages 295–302, August 1996.

[28] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[29] Ulrich Reif. A degree estimate for polynomial subdivision surfaces of higher regularity. *Proc. Amer. Math. Soc.*, 124:2167–2174, 1996.

[30] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 197–206, August 1990.

[31] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image-Based Pen-and-Ink Illustration. In *SIGGRAPH 97 Conference Proceedings*, pages 401–406, August 1997.

[32] Kent A. Stevens. Inferring shape from contours across surfaces. In Alex P. Pentland, editor, *From Pixels to Predicates*, pages 93–110. 1986.

[33] Greg Turk and David Banks. Image-Guided Streamline Placement. In *SIGGRAPH 96 Conference Proceedings*, pages 453–460, August 1996.

[34] J. Chal Vinson. *Thomas Nast: Political Cartoonist*. University of Georgia Press, Atlanta, 1967.

[35] Hassler Whitney. On singularities of mappings of euclidean spaces. I. Mappings of the plane into the plane. *Ann. of Math. (2)*, 62:374–410, 1955.

[36] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. In *SIGGRAPH 96 Conference Proceedings*, pages 469–476, August 1996.

[37] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Software*, 23(4):550–560, 1997.

[38] D. Zorin. Constructing curvature-continuous surfaces by blending. in preparation.

Image Analogies

Aaron Hertzmann[1,2] *Charles E. Jacobs*[2] *Nuria Oliver*[2] *Brian Curless*[3] *David H. Salesin*[2,3]

[1]New York University [2]Microsoft Research [3]University of Washington

A A' B B'

Figure 1 An image analogy. Our problem is to compute a new "analogous" image B' that relates to B in "the same way" as A' relates to A. Here, A, A', and B are inputs to our algorithm, and B' is the output. The full-size images are shown in Figures 10 and 11.

Abstract

This paper describes a new framework for processing images by example, called "image analogies." The framework involves two stages: a *design phase*, in which a pair of images, with one image purported to be a "filtered" version of the other, is presented as "training data"; and an *application phase*, in which the learned filter is applied to some new target image in order to create an "analogous" filtered result. Image analogies are based on a simple multi-scale autoregression, inspired primarily by recent results in texture synthesis. By choosing different types of source image pairs as input, the framework supports a wide variety of "image filter" effects, including *traditional image filters*, such as blurring or embossing; *improved texture synthesis*, in which some textures are synthesized with higher quality than by previous approaches; *super-resolution*, in which a higher-resolution image is inferred from a low-resolution source; *texture transfer*, in which images are "texturized" with some arbitrary source texture; *artistic filters*, in which various drawing and painting styles are synthesized based on scanned real-world examples; and *texture-by-numbers*, in which realistic scenes, composed of a variety of textures, are created using a simple painting interface.

CR Categories: I.2.6 [Artificial Intelligence]: Learning– Analogies; I.3.3 [Computer Graphics]: Picture/Image Generation– Display algorithms; I.4.10 [Image Processing and Computer Vision]: Image Representation– Statistical; J.5 [Computer Applications]: Arts and Humanities– Fine arts

Additional Keywords: example-based rendering, texture synthesis, non-photorealistic rendering, Markov random fields, autoregression, texture-by-numbers, texture transfer

Please see **http://grail.cs.washington.edu/projects/image-analogies/** for additional information and results.

1 Introduction

a·nal·o·gy n. *A systematic comparison between structures that uses properties of and relations between objects of a source structure to infer properties of and relations between objects of a target structure*. [14]

A native talent for perceiving analogies is . . . the leading fact in genius of every order.

—William James, 1890 [28]

Analogy is a basic reasoning process, one that we as humans employ quite commonly, and often unconsciously, to solve problems, provide explanations, and make predictions [44]. In this paper, we explore the use of analogy as a means for creating complex image filters (Figure 1). In particular, we attempt to solve the following problem:

Problem ("IMAGE ANALOGIES"): Given a pair of images A and A' (the *unfiltered* and *filtered source images*, respectively), along with some additional *unfiltered target image* B, synthesize a new *filtered target image* B' such that

$$A \ : \ A' \ :: \ B \ : \ B'$$

In other words, we want to find an "analogous" image B' that relates to B in "the same way" as A' relates to A. In general, this is a very difficult problem to solve; in this paper, we describe an approach that works well in many cases.

An advantage of image analogies is that they provide a very natural means of specifying image transformations. Rather than selecting from among myriad different filters and their settings, a user can simply supply an appropriate exemplar (along with a corresponding unfiltered source image) and say, in effect: "Make it look like this." Ideally, image analogies should make it possible to learn very complex and non-linear image filters—for instance, filters that can convert a photograph into various types of artistic renderings having the appearance of oil, watercolor, or pen-and-ink, by analogy with actual (real-life) renderings in these styles. In addition, these various types of filters would not need to be invented individually or programmed explicitly; ideally, the same general mechanism could be used instead to provide this very broad variety of effects.

While image analogies are clearly a desirable goal, it is not so clear how they might be achieved.

For one thing, a crucial aspect of the image analogies problem statement is the definition of similarity used to measure not only the relationship between each unfiltered image and its respective filtered version, but also the relationship between the source pair and the target pair when taken as a whole. This issue is tricky, in that we want to use some metric that is able to preserve recognizable features of the original image filter from A to A', while at the same time is broad enough to be applied to some completely different target image B. Moreover, it is not obvious what features of a training pair constitute the "style" of the filter: in principle, an infinite number of different transformations could be inferred from a pair of images. In this paper, we use a similarity metric that is based on an approximation to a Markov random field model, using raw pixel values and, optionally, steerable filter responses [45]. To measure relationships between the source and target image pair, we sample joint statistics of small neighborhoods within the images, as discussion in Section 3.

In addition, we would like the synthesis of the filtered target image B' to proceed at a reasonable rate. Thus, we will need a way to index and efficiently search over the various images A, A', and B, using the similarity metric, to choose the appropriate parts of the transform $A \rightarrow A'$ in synthesizing $B \rightarrow B'$. We use an autoregression algorithm, based primarily on recent work in texture synthesis by Wei and Levoy [49] and Ashikhmin [2]. Indeed, our approach can be thought of as a combination of these two approaches, along with a generalization to the situation of corresponding *pairs* of images, rather than single textures.

Finally, in order to allow statistics from an image A to be applied to an image B with completely different colors, we sometimes operate in a preprocessed luminance space, as described in detail in Sections 3.3 and 3.4.

In actual usage, we envision image analogies involving two stages. In the *design* (or *training*) phase, a designer (possibly an expert) creates a filter by selecting the training images A and A' (for example, from scanned imagery), annotating the images if desired, and (directly or indirectly) selecting parameters that control how various types of image features will be weighted in the image analogy. The filter can then be stored away in a library. Later, in the *application* phase, a user (possibly someone with no expertise at all in creating image filters) applies the filter to some target image B.

Obviously, we cannot expect our image analogies framework to do a perfect job in learning and simulating all possible image filters, especially from just a single training pair. Moreover, many of the filters that we would like our framework to be able to learn are, in fact, *extremely* difficult even for humans to master. Nevertheless, we have found our image analogies framework to work rather surprisingly well in a variety of situations, as demonstrated in Section 4. These include:

- ***traditional image filters***, such as blurring or "embossing" (Section 4.1);
- ***improved texture synthesis***, in which some textures are synthesized with higher quality than previous approaches (Section 4.2);
- ***super-resolution***, in which a higher-resolution image is inferred from a low-resolution source (Section 4.3);
- ***texture transfer***, in which images are "texturized" with some arbitrary source texture (Section 4.4);
- ***artistic filters***, in which various drawing and painting styles, including oil, watercolor, and line art rendering, are synthesized based on either digitally filtered or scanned real-world examples (Section 4.5); and
- ***texture-by-numbers***, in which realistic scenes, composed of a variety of textures, are created using a simple "painting" interface (Section 4.6).

In all of these cases, producing the various different effects is primarily just a matter of supplying different types of source image pairs as input. For example, a blur filter is "learned" by supplying an image and its blur as the (A, A') pair. Similarly, an oil-painting style is learned by supplying an image and its oil-painted equivalent as the input pair. Ordinary texture synthesis can be viewed as a special case of image analogies in which the unfiltered images A and B are null (i.e., considered to match trivially everywhere), and the analysis/synthesis is performed just on A' and B'. Alternatively, texture-by-numbers is achieved by using a realistic image, such as a landscape, as A' and supplying a simplified, hand-segmented version of the landscape as A—for instance, where one solid color in A corresponds to "sky texture" in A', another to "grass texture," and so on. These same colors can then be painted onto B to generate a new realistic landscape B' with similar textures.

Finally, we also describe a real-time, interactive version of our algorithm, which can be used to provide image analogies underneath the footprint of a digital painting tool (Section 5). While texture-by-numbers is a natural application for this tool, it can be used with any type of image analogy.

While successful in many ways, image analogies do not work in every case since they attempt to model only low-level statistics of the image pairs. Thus, higher-level features such as broad, coherent brush strokes are not always captured or transferred very effectively. Section 6 discusses the limitations of our approach and suggests areas of future research.

2 Related work

Image analogies build upon a great deal of previous work in several disparate areas, including machine learning, texture synthesis, non-photorealistic rendering, and image-based rendering. As far as we know, some of the applications supported by image analogies are completely new, such as the ability to learn "artistic filters" from digitized imagery of real artistic renderings. Others, such as super-resolution and texture transfer, have previously been addressed by other works, in some form or another. Although the algorithms we describe compare favorably—and improve upon—much of this previous work, it is the generality and convenience of the image analogies framework that we believe makes it so interesting and useful. Here, we survey some of the most closely related work.

Generalizing from a set of known examples, as we attempt to do in this paper, is a central problem in machine learning. Analogical reasoning is central to problem solving, learning, and creativity [19, 31, 33]. For this reason, a goal from the early days of artificial intelligence has been to build systems able to reason by analogy; early works include Evan's ANALOGY program [15] and Winston's seminal work on finding and exploiting parallels in simple theories [52]. In this paper, we propose a novel statistical approach for finding analogies between images, from the perspective of modeling transformations that are not just shifts, scales, and rotations but rather mappings of one sort of object or relation into another.

Recently, a number of applications of machine learning to problems in computer graphics have been published, including Video Rewrite [7], Voice Puppetry [5], Video Textures [43], and Style Machines [6]. Our paper continues in the two-word-title tradition of these earlier works. In research that is perhaps most closely related to our own, Freeman *et al*. [17] use Markov random fields (MRFs) for scene learning, in which they attempt to learn the transformation from some captured image to the scene interpretation, such as extracting high-resolution data from low-resolution data ("super-resolution") or inferring optical flow from image pairs. Our method differs from previous MRF modeling techniques in that we do not require an iterative algorithm in order to apply the model. More importantly, our work introduces a variety of new applications of these methods for computer graphics.

In the last few years, a great deal of work has been published in both the computer graphics and computer vision communities on the problem of *texture synthesis*: the creation of images that match the texture appearance of a given digitized sample. Heeger and Bergen [23] introduced this problem to the computer graphics community in 1995. More recently, De Bonet [4] and Efros and Leung [11] showed that a nearest-neighbor search can perform high-quality texture synthesis in a single pass, using multiscale and single-scale neighborhoods, respectively. (This search may be viewed as an approximation to sampling from an MRF, an approach used by Zhu et al. [54] and Portilla and Simoncelli [40].) Wei and Levoy [49] unify these approaches, using neighborhoods consisting of pixels both at the same scale and at coarser scales. Vector quantization [20] or other clustering may be used to summarize and accelerate the nearest-neighbors computation [17, 34, 35, 39, 49].

In unpublished work, Eilhauer *et al*. [13] demonstrate a method for synthesizing texture to match a given image, work that we extend in this paper under the name "texture transfer." In these same proceedings, Efros and Freeman [12] describe improved methods for texture synthesis and show how these methods can also be used for texture transfer. Our texture transfer method also bears some resemblance to Veryovka and Buchanan's methods for halftoning one image with the texture of another [47] and for using multiple textures to illustrate 3D meshes [46].

In recently published work, Ashikhmin [2] describes a texture synthesis method that works by greedily extending existing patches whenever possible, rather than by searching the entire example texture. The algorithm is very fast and produces results that often look much better than the output from previous synthesis methods. However, the greedy search seems to have difficulty restarting effectively when a patch being copied ends (e.g., runs off the end of the image). In our tests, this problem produces abrupt discontinuities between the texture patches. As mentioned earlier, our work builds upon this algorithm, combining it with Wei and Levoy's approach and generalizing this combination to image analogies. Ashikhmin also allows a user to draw a color image as a target for texture synthesis, in a manner that has some resemblance to our "texture-by-numbers" application. However, as discussed in detail in Section 4.6, image analogies are able to address a number of drawbacks in Ashikhmin's approach by virtue of using a pair of images, rather than a single image, to control the synthesis procedure.

One important application of image analogies that we explore in this paper is the automatic synthesis of various artistic styles from examples. In the past few years, there has been a great deal of work in creating artistic styles by computer [10, 21, 24, 26, 32, 36, 41, 42, 50, 51], a field that has come to be known as *non-photorealistic rendering* (NPR). One drawback of most of these previous works, however, is that the methods have had to be specifically tailored to a specific rendering style (or space of styles); image analogies, by contrast, although they may not be able to create any single style as well, can be used for a very broad range of effects.

In this paper, we aim to broaden the range of NPR techniques by exploring what could be thought of as *example-based rendering* (EBR), in which scanned-in artistic imagery is reused for NPR synthesis. The earliest examples of this sort of approach were tools like the "clone brush" in Adobe Photoshop, or the "image hose" in Corel (originally, Fractal Design) Painter. Recently, a number of EBR approaches to NPR have been proposed in a variety of research systems [8, 27, 30, 53]. Perhaps most similar in spirit to our own approach (albeit in somewhat different domains) are a method for creating pen strokes from examples [18] and a method for estimating parameters of a 3D line-drawing illustration system from an example rendering made within the same system [22].

3 Image analogies

Here, we describe a set of data structures and algorithms to support image analogies.

3.1 Definitions and data structures

As input, our algorithm takes a set of three images, the *unfiltered source image* A, the *filtered source image* A', and the *unfiltered target image* B. It produces the *filtered target image* B' as output.

Our approach assumes that the two source images are *registered*; that is, the colors at and around any given pixel p in A correspond to the colors at and around that same pixel p in A', through the image filter that we are trying to learn. Thus, we will use the same index p to specify both a pixel in A and its corresponding pixel in A'. We will use a different index q to specify a pixel in the target pair B and B'.

For the purposes of this exposition, we will assume that the various images contain not just an RGB color, but additional channels of information as well, such as luminance and various filter responses. Together, all of these channels (including RGB) comprise the *feature vector* for each pixel p. We use $A(p)$ (or $A'(p)$) to denote the complete feature vector of A (or A') at pixel p and, similarly, $B(q)$ (or $B'(q)$) to specify the feature vector at pixel q. Note that the features used for the A and B images need not be the same as for the A' and B' images. The particular features we use are described in more detail in Section 3.3 below (however, experimenting with alternate or additional features is certainly a rich area for future research). As we shall see, these features will be used to guide the matching process, in order to help select the most suitable pixels from A' to use in the synthesis of B'.

Finally, our algorithm will need to keep track of the position p of the source pixel that was copied to pixel q of the target. Thus, we will store an additional data structure $s(\cdot)$ (for "source"), which is indexed by q, and has the property $s(q) = p$.

In summary, our algorithm maintains the following data structures, of which the RGB channels of $A(p)$, $A'(p)$, and $B(q)$ are inputs, the RGB channels of $B'(q)$ is the output, and the other channels of A, A', B, and B', as well as $s(q)$, are intermediate computed results in the synthesis process:

$A(p)$: **array** $p \in$ *SourcePoint* **of** *Feature*
$A'(p)$: **array** $p \in$ *SourcePoint* **of** *Feature'*
$B(q)$: **array** $q \in$ *TargetPoint* **of** *Feature*
$B'(q)$: **array** $q \in$ *TargetPoint* **of** *Feature'*
$s(q)$: **array** $q \in$ *TargetPoint* **of** *SourcePoint*

where *SourcePoint* and *TargetPoint* are 2D pixel locations in the source and target pairs, respectively.

We will actually use a multiscale representation of all five of these quantities in our algorithm. Thus, we will typically index each of these arrays by their multiscale level ℓ using subscripts. For example, if A_ℓ represents the source image A at a given resolution, then $A_{\ell-1}$ represents a corresponding lower-resolution image at the next coarser level, with half as many pixels in each dimension. We will use L to denote the maximum level, i.e., the level for the highest-resolution versions of the images.

3.2 The algorithm

Given this notation, the image analogies algorithm is easy to describe. First, in an initialization phase, multiscale (Gaussian pyramid) representations of A, A', and B is constructed, along with their feature vectors and some additional indices used for speeding the matching process (e.g., an approximate-nearest-neighbor search (ANN), as described below). The synthesis then proceeds from coarsest resolution to finest, computing a multiscale representation of B', one level at a time. At each level ℓ, statistics pertaining

to each pixel q in the target pair are compared against statistics for every pixel p in the source pair, and the "best" match is found. The feature vector $B'_\ell(q)$ is then set to the feature vector $A'_\ell(p)$ for the closest-matching pixel p, and the pixel that matched best is recorded in $s_\ell(q)$.

The algorithm can be described more precisely in pseudocode as follows:

function CREATEIMAGEANALOGY(A, A', B):
  Compute Gaussian pyramids for A, A', and B
  Compute features for A, A', and B
  Initialize the search structures (e.g., for ANN)
  for each level ℓ, from coarsest to finest, **do**:
    for each pixel $q \in B'_\ell$, in scan-line order, **do**:
      $p \leftarrow$ BESTMATCH(A, A', B, B', s, ℓ, q)
      $B'_\ell(q) \leftarrow A'_\ell(p)$
      $s_\ell(q) \leftarrow p$
  return B'_L

The heart of the image analogies algorithm is the BESTMATCH subroutine. This routine takes as input the three complete images A, A' and B, along with the partially synthesized B', the source information s, the level ℓ, and the pixel q being synthesized in B'. It finds the pixel p in the source pair that best matches the pixel being synthesized, using two different approaches: an *approximate search*, which attempts to efficiently find the closest-matching pixel according to the feature vectors of p, q, and their neighborhoods; and a *coherence search*, based on Ashikhmin's approach [2], which attempts to preserve coherence with the neighboring synthesized pixels. In general, the latter approach will usually not return a pixel that matches as closely with respect to the feature vectors; however, since the L_2-norm is an imperfect measure of perceptual similarity, coherent pixels will often look better than the best match under L_2. We therefore rescale the approximate-search distance according to a *coherence parameter* κ, in order to make it artificially larger when comparing the two choices. Thus, the larger the value of κ, the more coherence is favored over accuracy in the synthesized image. In order to keep the coherence term consistent at different scales, we attenuate it by a factor of $2^{\ell-L}$ since pixel locations at coarser scales are spaced further apart than at finer scales. (In a sense, $2^{\ell-L}\kappa$ represents an estimate of the scale of "textons" [29] at level ℓ.) We typically use $2 \leq \kappa \leq 25$ for color non-photorealistic filters, $\kappa = 1$ for line art filters, and $0.5 \leq \kappa \leq 5$ for texture synthesis.

Here is a more precise statement of this algorithm:

function BESTMATCH(A, A', B, B', s, ℓ, q):
  $p_{\text{app}} \leftarrow$ BESTAPPROXIMATEMATCH(A, A', B, B', ℓ, q)
  $p_{\text{coh}} \leftarrow$ BESTCOHERENCEMATCH(A, A', B, B', s, ℓ, q)
  $d_{\text{app}} \leftarrow \|F_\ell(p_{\text{app}}) - F_\ell(q)\|^2$
  $d_{\text{coh}} \leftarrow \|F_\ell(p_{\text{coh}}) - F_\ell(q)\|^2$
  if $d_{\text{coh}} \leq d_{\text{app}}(1 + 2^{\ell-L}\kappa)$ **then**
    return p_{coh}
  else
    return p_{app}

Here, we use $F_\ell(p)$ to denote the concatenation of *all* the feature vectors within some neighborhood $N(p)$ of both source images A and A' at both the current resolution level ℓ and at the coarser resolution level $\ell - 1$. We have used 5×5 neighborhoods in the fine level and 3×3 neighborhoods in the coarse level (Figure 2). Similarly, we use $F_\ell(q)$ to denote the same concatenation for the target images B and B', although in the case of the filtered target image B' the neighborhood at the finest resolution includes only the portion of the image that has already been synthesized. (Note that $F(\cdot)$ is overloaded in our notation; the index p or q will be used to determine whether a particular $F(\cdot)$ is a source or target neighborhood feature vector.) In each case, the norm $\|F_\ell(p) - F_\ell(q)\|^2$

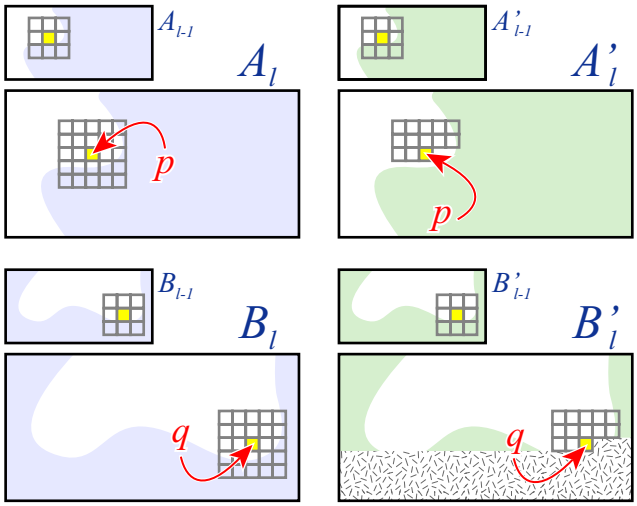


Figure 2 Neighborhood matching. In order to synthesize the pixel value at q in the filtered image B'_ℓ, we consider the set of pixels in B'_ℓ, B_ℓ, $B'_{\ell-1}$, and $B_{\ell-1}$ around q in the four images. We search for the pixel p in the A images that give the closest match. The synthesis proceeds in scan-line ordering in B'_ℓ.

is computed as a weighted distance over the feature vectors $F(p)$ and $F(q)$, using a Gaussian kernel, so that differences in the feature vectors of pixels further from p and q have a smaller weight relative to the differences at p and q. We also normalize the vectors so that each scale of the pyramid has equal weight. Note that some special processing is required at boundaries, as well as at the lowest resolution of the pyramid, since the neighborhoods are a little bit different in these areas. We perform brute-force searches with only the partial neighborhoods in these cases.

For the BESTAPPROXIMATEMATCH procedure, we have tried using both *approximate-nearest-neighbor search* (ANN) [1] and *tree-structured vector quantization* (TSVQ) [20], using the same norm over the feature vectors. In our experience, ANN generally provides more accurate results for the same computation time, although it is also more memory intensive. We used ANN for all of the examples shown in this paper. Principal components analysis (PCA) can be used to reduce the dimensionality of feature vectors leading to a substantial speed-up in the search. We generally keep 99% of the variance, which can lead to a reduction in dimensionality of about an order of magnitude. However, using PCA can degrade the quality of the results on some simple cases; it is most useful in cases with large feature vector sizes (e.g., when steerable filters are used).

The BESTCOHERENCEMATCH procedure simply returns $s(r^\star) + (q - r^\star)$, where

$$r^\star = \arg\min_{r \in N(q)} \|F_\ell(s(r) + (q - r)) - F_\ell(q)\|^2$$

and $N(q)$ is the neighborhood of already synthesized pixels adjacent to q in B'_ℓ. This formula essentially returns the best pixel that is coherent with some already-synthesized portion of B'_ℓ adjacent to q, which is the key insight of Ashikhmin's method.

3.3 Features

Feature selection and representation is a large open problem and an active area of research in machine learning. For now, we have experimented with several different components for the feature vectors. Using the RGB channels themselves is the most obvious choice (and the first thing we tried). However, for some filters, we found that our source pairs did not contain enough data to match the target pair well using RGB color. This is due to the well-known "curse of

dimensionality:" the neighborhood space for RGB images is much larger than for grayscale images, and thus a single image pair provides a correspondingly sparser sampling of the space for RGB than for grayscale. Consequently, the neighborhood histogram of A may still be poorly matched to B, whereas this is less of a problem for grayscale images.

An alternative, which we have used to generate many of the results shown in this paper, is to compute and store the luminance at each pixel and use it in place of RGB in the distance metric. Luminance can be computed in a number of ways; we use the Y channel from the YIQ color space [16], where the I and Q channels are "color difference" components. This approach is motivated by vision science: we are much more sensitive to changes in the luminance channel than to changes in color difference channels [48]. After processing in luminance space, we can recover the color simply by copying the I and Q channels of the input B image into the synthesized B' image, followed by a conversion back to RGB. An added benefit of this approach is the speedup inherent in performing the matching and synthesis with one third as many color channels. The downside, however, is that color dependencies in the analogy filter are lost. In this paper, we worked in luminance space for the blur filter, super-resolution, and artistic filter examples.

Another way to improve the perceptual results of matching is to compute multiple scales of oriented derivative filters [4, 23, 54]. To this end, we can compute a steerable pyramid [45] for the luminance of A and B and concatenate the filter responses to the feature vectors for these images. The distance metric then becomes a weighted combination of similarity in luminance, as well as similarity in orientation among regions of the unfiltered images. We used third-derivative steerable filters comprised of four filter kernels for synthesizing the line art examples (Figure 8); for our other experiments, we found them to make little or no difference.

3.4 Luminance remapping

Converting images to luminance can still give poor overlap between image neighborhood histograms; for example, a light A will be of little use when processing a dark B. As a preconditioning step, we would like to discover a luminance transformation that brings the histograms into correspondence. One standard approach is histogram matching [9]; however, we find that it uses non-smooth mappings with undesirable side-effects.

Our approach, instead, is to apply a linear map that matches the means and variances of the luminance distributions. More concretely, if $Y(p)$ is the luminance of a pixel in image A, then we remap it as

$$Y(p) \leftarrow \frac{\sigma_B}{\sigma_A}(Y(p) - \mu_A) + \mu_B$$

where μ_A and μ_B are the mean luminances, and σ_A and σ_B are the standard deviations of the luminances, both taken with respect to luminance distributions in A and B, respectively. We apply the same linear transform to A', in order to keep the training pair consistent. In this paper, luminance remapping is only used for the color artistic filters (Section 4.5).

This same approach can be extended to matching color distributions in a fairly straightforward way [25]. It is not used in this paper because we found synthesis in luminance to give better results.

4 Applications

By supplying different types of images as input, the image analogies framework can be used for learning filters for many different types of applications. We describe here the applications that we have experimented with so far. Timings for these tests are discussed in the next section. Additional results can be found at **http://grail.cs.washington.edu/projects/image-analogies/**.

4.1 Traditional image filters

As a simple test, we tried learning some traditional image-processing filters, including a "blur" filter (Figure 3) and an "emboss" filter from Adobe Photoshop (Figure 4). While the image-analogies framework gives adequate results, it is nowhere near as efficient as applying the filter directly. Still, these experiments verify that the image analogies framework works for some basic filters.

4.2 Improved texture synthesis

Texture synthesis is a trivial case of image analogies, where the elements of the A and B images are zero-dimensional or constant. The algorithm we have described, when used in this way for texture synthesis, combines the advantages of the weighted L_2 norm and Ashikhmin's search algorithm, although without the speed of Ashikhmin's algorithm. For example, the synthesized textures shown in Figure 5 have a similar high quality to those of Ashikhmin's algorithm, without the edge discontinuities.

4.3 Super-resolution

Image analogies can be used to effectively "hallucinate" more detail in low-resolution images, given some low- and high-resolution pairs (used as A and A') for small portions of the images. (The image analogies framework we have described is easily extended to handle more than a single source image pair, which is what we have done for these examples.) Figure 6 demonstrates this application, using images of a set of maple trees and of a Dobag rug, respectively. An interesting area for future work is to choose the training pairs automatically for image compression, similar to fractal image compression [3].

4.4 Texture transfer

In texture transfer, we filter an image B so that it has the texture of a given example texture A' (Figure 7). Texture transfer is achieved by using the same texture for both A and A'. We can trade off the appearance between that of the unfiltered image B and that of the texture A by introducing a weight w into the distance metric that emphasizes similarity of the (A, B) pair over that of the (A', B') pair. Increasing w causes the input image to be reproduced more faithfully, whereas decreasing w ties the image more closely to the texture. When $w = 0$, texture transfer reduces to ordinary texture synthesis. For somewhat better results, we also modify the neighborhood matching by using single-scale 1×1 neighborhoods in the A and B images. Thus, $F_\ell(p)$ contains the value of the pixel p in A_ℓ (or B_ℓ), as well as the usual neighborhoods around p in A'_ℓ and $A'_{\ell-1}$ (or B'_ℓ and $B'_{\ell-1}$).

This application of image analogies may be somewhat counterintuitive, since an intuitive interpretation of an "analogy" in which A is the same as A' is as the identity filter. However, texture transfer is actually another valid interpretation [25]. Image analogies synthesize images drawn from the statistical distribution of neighborhoods in A'—in texture transfer, this is done while trying to match the B image as closely as possible.

4.5 Artistic filters

Although the problem is in general very difficult, we have had some success in using image analogies to transfer various artistic styles from one image to another, as shown in Figures 8–13 and 15.

For many example images, we do not have a source photograph available; hence, a substitute must be created. We generally view the A' image as providing texture (e.g., pen strokes or paint texture) to an untextured image. To create A from A', we apply an anisotropic diffusion [37] or similar filter to A' (we used the "Smart Blur" filter from Adobe Photoshop), in order to maintain sharp contours but eliminate texture. For line art filters, we blur the image before applying anisotropic diffusion.

For the color artistic filters in this paper, we performed synthesis in luminance space, using the preprocessing described in Sections 3.3 and 3.4. (In general, we find that matching with color gives richer and more appealing results, but can often fail quite dramatically.)

For line art filters, using steerable filter responses in feature vectors leads to significant improvement. We suspect that this is because line art depends significantly on gradient directions in the input images. We use steerable filter responses only for matching in A and B (but not in A'/B').

We have also had some success when A is a known photograph and A' is a painting or drawing made by hand from the photograph. (Unfortunately, these results are not shown here, for copyright reasons.) In these cases, some care must be taken to carefully register the photograph to the painting since our algorithm currently assumes that the images are in approximate pointwise correspondence. For our training pairs of this type, we first aligned the images by manually estimating a global translation, rotation, and scale. We then warped the example source image with a custom image warping program designed for local image adjustments [25].

The scale of the training images determines the fineness of the features in the B' image and may be chosen by the filter designer, or left as a parameter to the user applying the filter.

4.6 Texture-by-numbers

Texture-by-numbers allows new imagery to be synthesized by applying the statistics of a labeled example image to a new labeling image B. For example, given a labeling of the component textures of a realistic image, a new realistic one may be painted just by painting the arrangement of the component textures (Figure 14). An example of this kind of synthesis, performed in real-time as part of a "painting" interface, is demonstrated in the accompanying video.

A major advantage of texture-by-numbers is that it allows us to synthesize from images for which ordinary texture synthesis would produce poor results. Consider the photograph of an oxbow shown in Figure 14. Although the image has textural regions, attempting to create a new version of the river via ordinary texture synthesis would produce very poor results, as the synthesis process would mix unrelated textures. In statistical terms, the problem is that the texture distribution is not stationary. On the other hand, specifying a corresponding A image makes the A' image into a useful texture, in the sense that the *conditional* density of the A' image is stationary (now conditioned on A). Given a new B image, we can generate a new scene to match it. Note that, in addition to filling in textures in a sensible manner, the boundaries between texture regions also match the examples, since they are synthesized from examples in A with similar boundary shapes.

A more sophisticated example is shown in Figure 16. Treating the scenery as a single texture produces poor results because the synthesis mixes foreground texture with background texture. In other words, the texture is stationary horizontally but not vertically. In this case, we provide a gradient in the red channel of the A image, which constrains the synthesis so that near elements will not be mixed with far elements.

Texture-by-numbers requires an appropriate choice of the A image in order to factor out non-stationary components of A'. In our experience, the synthesis is somewhat forgiving, degrading gracefully as the assumptions become less appropriate. In principle, the A image can be of arbitrary dimension and content. For example, it could include additional information about normals, depths, or orientations [42] of the A' image to improve the texture-by-numbers process.

This application bears some resemblance to the user-guided texturing described by Ashikhmin [2]; however, it fixes several of the problems with that method. In Ashikhmin's method, multiple passes are usually required for a good match. In addition, Ashikhmin's greedy search may create poor matches when a very large example texture is used, since the synthesis cannot "restart" until it finishes copying a patch. More significantly, the colors in the target must be distinct: the algorithm would have difficulty, for example, distinguishing between green trees and green grass. In addition, our algorithm also allows for extra channels of information (such as depth, normals, etc.) to be used to control the synthesis.

5 Interactive editing

For many applications, the ability to directly manipulate an image via a user interface is crucial. In the accompanying video, we demonstrate an application in which a user can "paint" a landscape by coarsely specifying locations for the trees, sky, etc. The main difficulty is that a single change to a B image could theoretically affect the rest of the image, and the full synthesis algorithm is currently too slow to run at interactive rates. However, we can exploit the fact that, in practice, user painting operations only affect a small area of the image at a time, and, under the locality assumption, these operations will have exponentially-decaying influence on distant image pixels. Hence, we can maintain an acceptable image by updating only the modified pixels and their neighbors.

The user interface presents a painting interface for placing RGB values into the B or B' images. The B image is initialized with some default value (e.g., a blank image or a predefined image), and a corresponding B' image is synthesized from the initial B'. The initial B' may also be precomputed.

The key to making interactive painting efficient in this context is to provide an immediate update of B' using a coherence search (Section 3.2) as the user paints, and to refine B' with the full search (approximate plus coherence search) progressively, only as processing cycles allow. Our implementation has two threads, an event handler and a synthesis thread. When the user paints into the B image, the event handler queues the painting locations at all scales for updating. The synthesis thread performs a coherence search on the changed pixels in scan-line order, though it uses the full search with causal neighborhoods for every tenth pixel. Pixels that have not been updated are marked as pixels to be ignored when comparing neighborhood distances during coherence search. These pixels are placed in another queue, and, whenever the first queue is empty, the update thread performs full search with non-causal neighborhoods on the contents of the second queue.

6 Discussion and future work

In this paper, we have described a framework for image processing by example, which generalizes texture synthesis for the case of two corresponding image pairs. We have shown how the framework described is applicable to a wide variety of image texturing problems, including image filtering, texture synthesis, super-resolution, texture transfer, artistic filters, and texture-by-numbers. As the framework is very general, we suspect that it (or some related framework, based on the same kinds of analogic reasoning) may eventually be applicable to a much broader domain of problems, perhaps extending well outside of image processing—or even image synthesis—to encompass such disparate varieties of data as motion capture, music synthesis, and so on. Thus, we are intrigued by the many possibilities for future research!

There is still much work to be done, both in improving the methods that we have presented, and in investigating other approaches. Here is a partial list of some of the directions we would like to pursue:

Speeding it up. The performance of our algorithm is logarithmic with respect to the size (in pixels) of the training pair (due to using heuristic search techniques), and linear with respect to the size of the target. Although we have made use of several techniques to enhance performance (e.g., PCA, ANN), our algorithm is still rather slow, taking anywhere from on the order of tens of seconds to do simple texture synthesis, to a few minutes for the texture transfer

examples, to a few hours for the artistic renderings on a 1GHz PC processor. One reason for this fairly poor performance is that our implementation has never been hand-tuned, since we have been interested more in ease of prototyping than in speed. We expect that optimizing the implementation would give about a factor of 5 speed-up. In addition, we would like to explore better regression and search techniques that could speed up the algorithm even more.

Estimating other types of image statistics. Capturing the style of a filter from an example is a fundamentally difficult problem because the solution is not unique and often depends on prior knowledge that may be difficult to implement. Our approach attempts to capture low-level statistical features of the filter. Unfortunately, many artistic styles involve larger-scale features that our approach does not capture so well. More specialized image analogies methods could be tailored to specific applications. For example, it would be interesting to try to learn NPR filters by estimating statistics of stroke shapes and positions; however, this likely entails difficult combinatorial optimization problems, as well as the problem of acquiring training data.

Better features and similarity metrics. None of the neighborhood distance metrics that we use provide perfect measures of perceptual similarity, since each one fails in a variety of situations. Our algorithm, which attempts to combine the best attributes of the L_2 metric with those of Ashikhmin's algorithm, still produces some undesirable artifacts. Finding a better matching function and/or better features is an important open area of research for image analysis and synthesis applications.

Better color processing. In order to achieve good results for small training sets, we have sacrificed the full use of color information by using luminance space for some filters. There are many possible ways to improve color processing. A simple improvement would be to generate the I and Q channels of B' by fitting a linear transformation from the I and Q channels of A to A' (similar to our luminance matching step), and then applying this transform to the I and Q channels of B. Another approach would be to improve preprocessing to enable synthesis in full color, such as by allowing different linear transformations for different image patches, or by fitting a smooth, non-linear histogram matching function.

Learning image correspondence and registration. Another limitation of our technique is that it assumes a pointwise correspondence between the training images, which requires careful registration of the example images. It would be more desirable to learn the filter, while simultaneously learning the correspondence between pixels, and even automatically apply a learned distortion during synthesis.

Combining with automatic texture segmentation. Our work generalizes recent results in texture synthesis. Whereas texture synthesis algorithms can only resynthesize images from stationary statistics, our methods can resynthesize non-stationary statistics based on some user guidance. One possible extension of this work would be to synthesize from non-stationary images in a completely automatic fashion, perhaps using a texture segmentation as an input [35].

Extensions to 3D models and animation. An important extension is to processing images of 3D models, where additional channels of information (e.g., depths, normals, and silhouettes) can be used to improve the filter reproduction. Likewise, there are natural extensions of these ideas to creating video and animation.

Extensions to other domains. We are interested in applying the image analogy algorithm to a number of other filter learning problems, such as scratch removal, alpha estimation, motion synthesis, musical synthesis, and so on. Moreover, it should be possible to generalize the framework we have described to make use of analogies between data in completely different domains. As a whimsical example, A might be an image and A' an interpretive dance based on the image; a new image B could then be used to synthesize a new interpretive dance B'. Several problems would need to be solved to make this possible. For example, the framework we have described requires a a one-to-one correspondence between data in the training pair, and no such obvious correspondence would exist in this case. Still, we are encouraged by our early results, and look forward to improving these methods and exploring new applications of analogic reasoning in computer graphics.

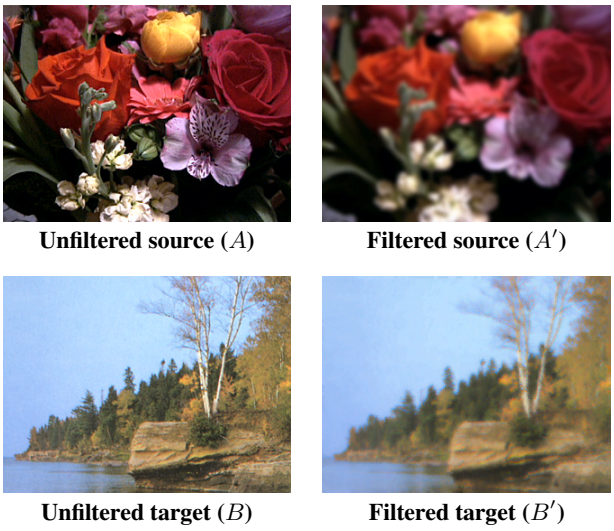


Figure 3 Toy example: Learning a blur filter. The A and A' images comprise the training data. Images were converted to luminance, and then the filter learned from A and A' was applied to B to get B'. (Shore image courtesy John Shaw [38].)

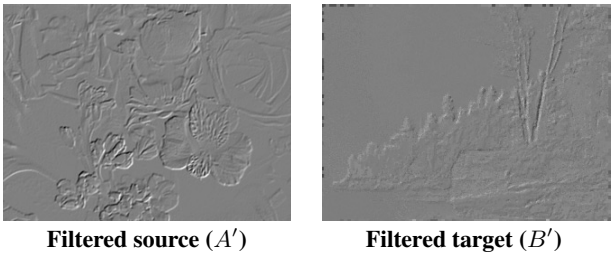


Figure 4 Toy example: Learning an emboss filter. The A and B images are the same as in Figure 3.

Acknowledgements

We are indebted to Douglas Zongker and Wilmot Li for their generous assistance with creating the video and figures for this paper. We are grateful to Matthew Brand for suggesting that we look at histograms, and to Shlomo Gortler for showing us his students' work on texture transfer [13]. Stephen Spencer provided indispensible technical assistance during the final phases of this work. This work was supported in part by NSF grants DGE-9454173 and CCR-9875365, and by an Intel equipment donation.

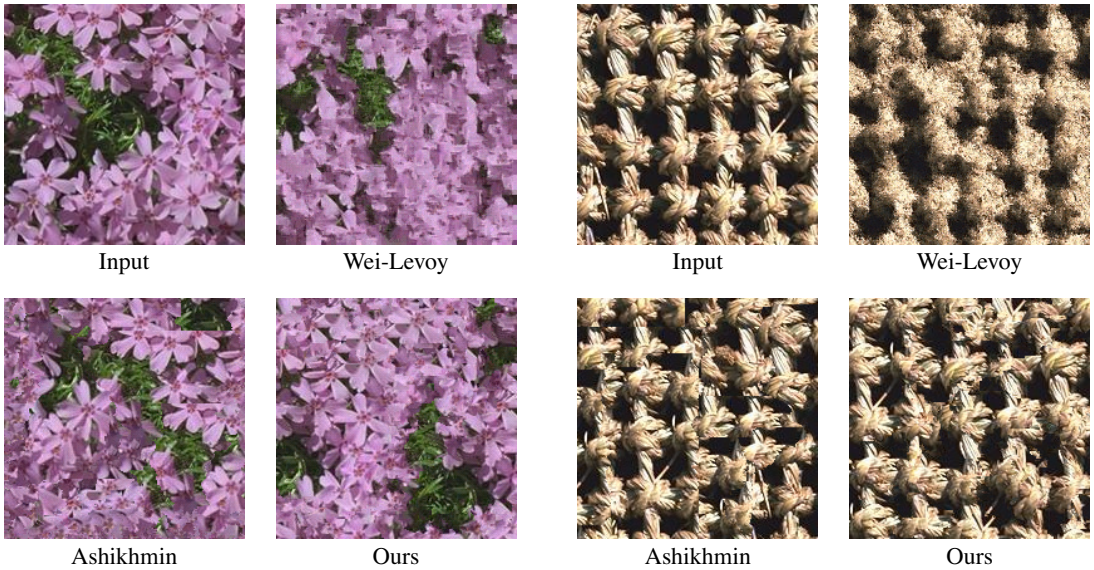


Figure 5 Improved texture synthesis. The textures synthesized with Wei and Levoy's algorithm [49] give blurry results because the L_2 norm is a poor measure of perceptual similarity. Ashikhmin's algorithm [2] gives high-quality coherent patches, but creates horizontal edges when patches reach the end of the source image, such as in the upper right corner of the flower texture. Additionally, Ashikhmin's algorithm does not capture appearance at multiple scales, such as the regular pattern of the weave. Our algorithm combines the advantages of these two previous methods. We used $\kappa = 5$ for both textures in this figure. (The input textures were obtained from the MIT VisTex web page, Copyright © 1995 MIT. All rights reserved).

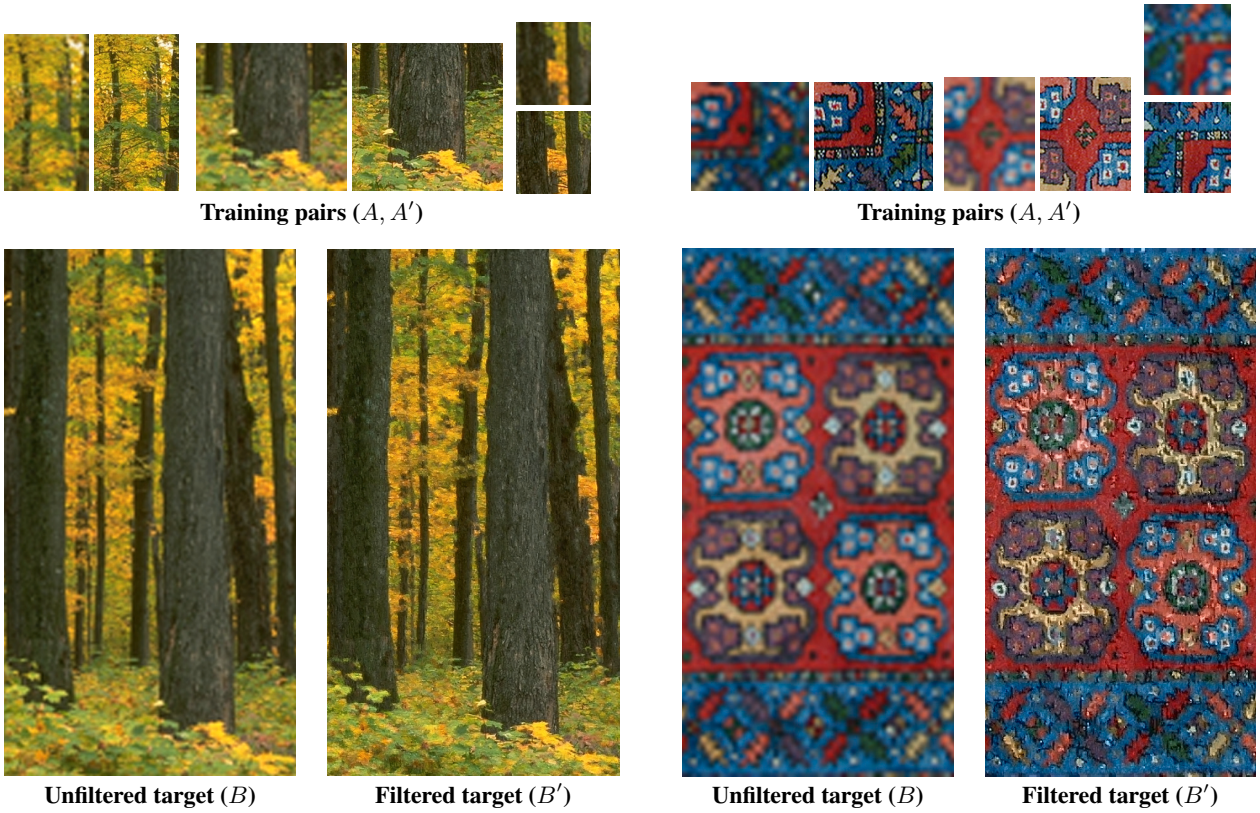


Figure 6 Super-resolution. For each example, the training pairs (above) contain low- and high-resolution versions of a portion of an image. This training data is used to specify a "super-resolution" filter that is applied to a blurred version of the full image (below, left) to recover an approximation to the higher-resolution original (below, right). (Maple trees image courtesy Philip Greenspun, http://philip.greenspun.com).

Figure 7 Texture transfer. A photograph (shown in Figure 8) is processed to have the weave and rug textures shown on the left. In each case, the texture is used for both A and A'. The center and right result images show the effect of trading off fidelity to the source image versus fidelity to the texture. (The weave texture was obtained from the MIT VisTex web page, Copyright © 1995 MIT. All rights reserved).

Figure 8 Line art illustrations by example. Two illustration styles, defined by the hand-drawn A' images, are applied to the two B images shown (at reduced size) in the upper left. Each A image was created by applying a blur and an anisotropic diffusion to the corresponding A'. The resulting four B' images are shown on the right. The upper A' image is from Gustave Doré's illustrations for *Don Quixote*; the lower A' is from an engraving by Francesco Bartolozzi, a Renaissance artist.

Figure 9 Unfiltered target images (B) for the NPR filters and texture transfer. (Leftmost image courtesy John Shaw [38].)

Unfiltered examples (A) **Filtered examples** (A')

Figure 10 Training pairs for the color NPR filters used in this paper. The upper A' image is a detail of *Starry Night above the Rhône* by Vincent Van Gogh; the "unfiltered" source image was generated by processing the painting with Photoshop's "Smart Blur" filter. The lower image pair is a photograph and a watercolor created from it with a semi-automatic digital filter [10].

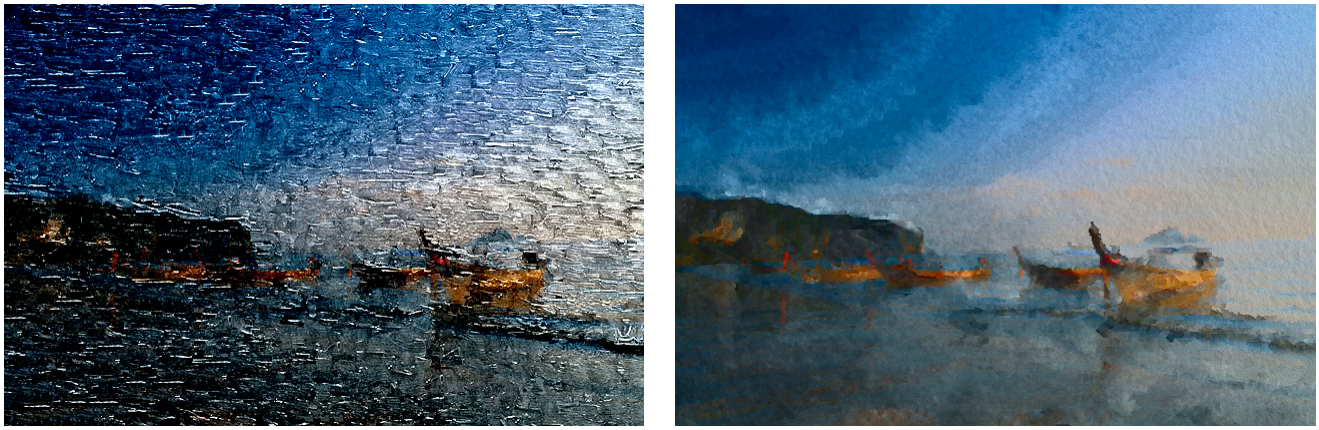
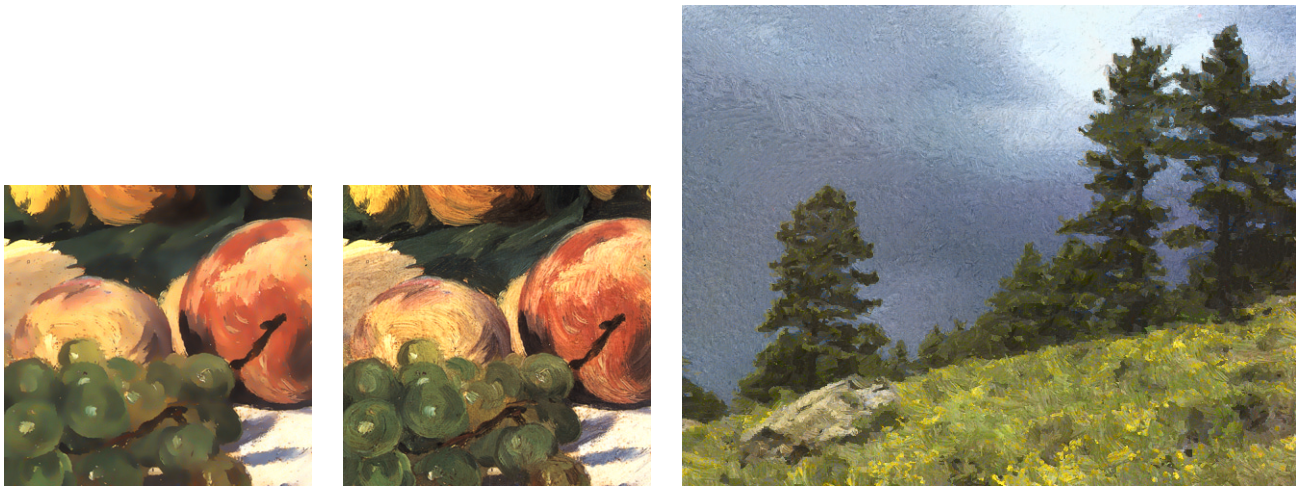


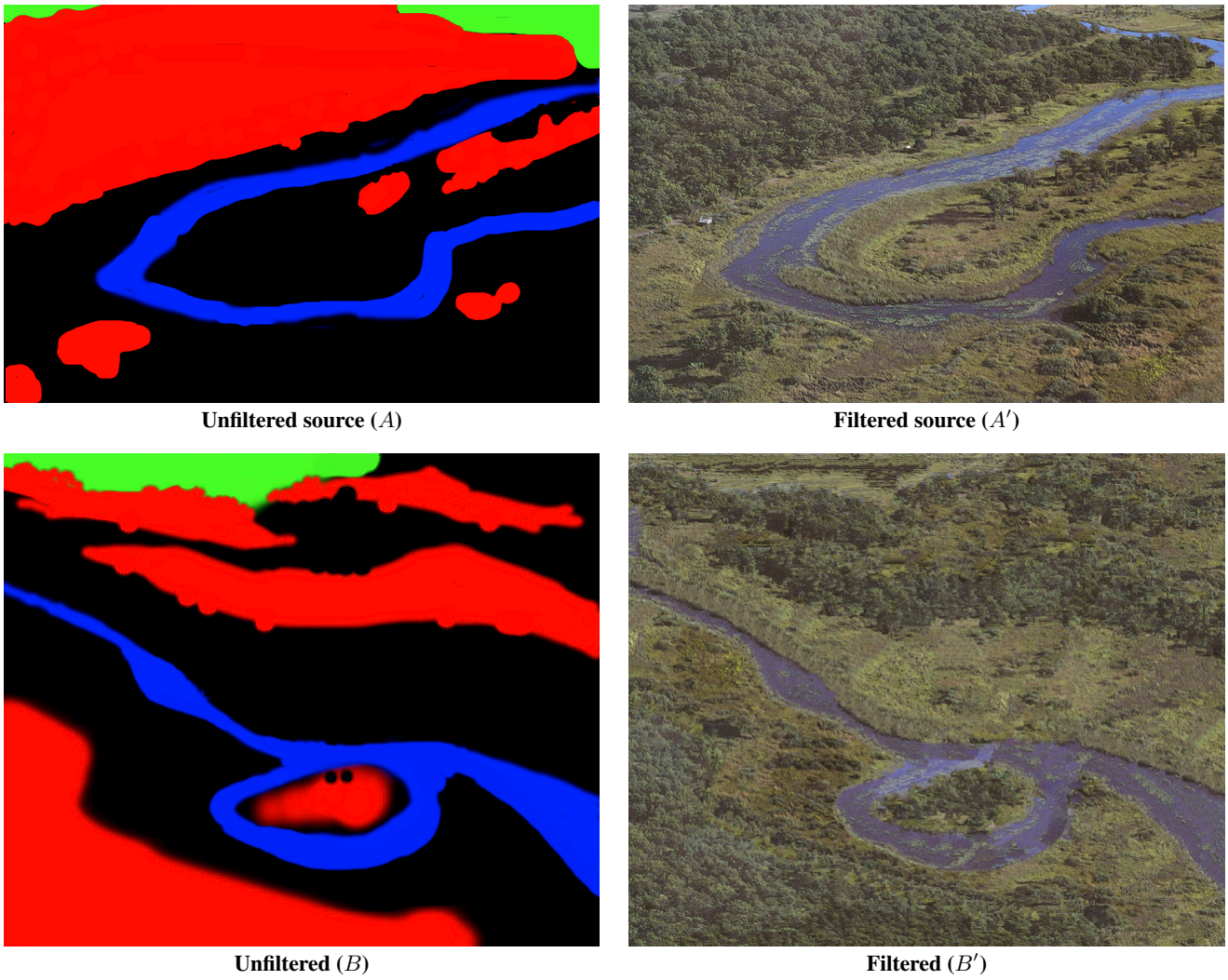
Figure 11 Boat paintings by example. The left image is painted in a style learned from a Van Gogh painting (Figure 10, top row); the right image is in the style of a watercolor filter (Figure 10, bottom row).

Figure 12 Paintings by example. The left images are painted in a style learned from a Van Gogh painting (Figure 10, top row); the right images are in the style of a watercolor filter (Figure 10, bottom row).



Unfiltered source (A) **Filtered source** (A') **Filtered** (B', $\kappa = 5$)

Figure 13 Paintings by example, varying the coherence parameter κ. The filtered target image (A') is a detail of *Still Life with Melon and Peaches* by Edouard Manet. Additional images are shown on the facing page in Figure 15. The source image is shown in Figure 9.

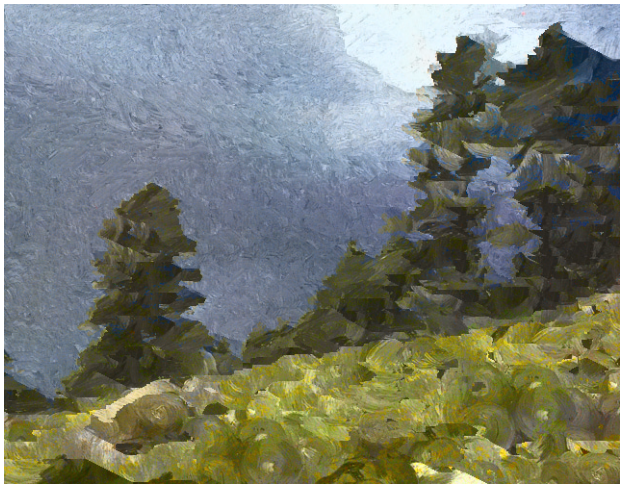


Unfiltered source (A) **Filtered source** (A')

Unfiltered (B) **Filtered** (B')

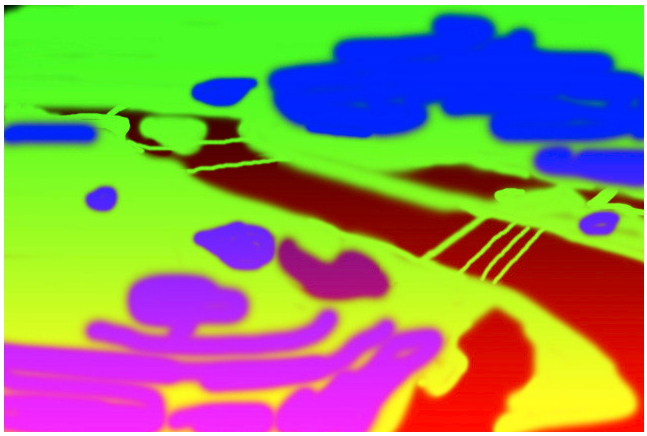
Figure 14 Texture-by-numbers. The unfiltered source image (A) was painted by hand to annotate A'. The unfiltered target image (B) was created in a paint program and refined with our interactive editor; the result is shown in B'. (Oxbow image courtesy John Shaw [38].)

Filtered (B', $\kappa = 10$)



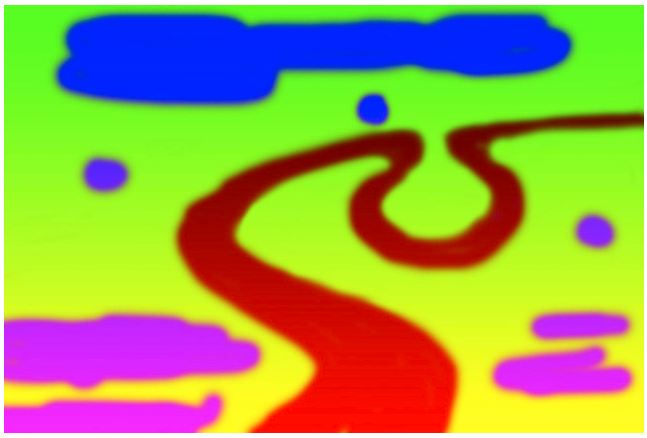
Filtered (B', $\kappa = 15$)

Figure 15 Paintings by example, varying the coherence parameter κ. The training is from a painting by Manet. Additional images are shown on the facing page in Figure 13. The source image is shown in Figure 9.



Unfiltered source (A)

Filtered source (A')



Unfiltered (B)

Filtered (B')

Figure 16 Rerouting the Potomac. Ordinary texture synthesis cannot reproduce the terrain in the photograph because it is not stationary: far elements are different from near elements. The use of the gradient channel in A and B distinguishes near from far, allowing the photograph to be used for texture-by-numbers.

References

[1] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions. *Journal of the ACM*, 45(6):891–923, 1998. Source code available from http://www.cs.umd.edu/~mount/ANN.

[2] Michael Ashikhmin. Synthesizing Natural Textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001.

[3] Michael F. Barnsley, Lyman P. Hurd, and Louisa F. Anson. *Fractal Image Compression*. A.K. Peters Ltd, 1993.

[4] Jeremy S. De Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images. *Proceedings of SIGGRAPH 97*, pages 361–368, August 1997.

[5] Matthew Brand. Voice Puppetry. *Proceedings of SIGGRAPH 99*, pages 21–28, August 1999.

[6] Matthew Brand and Aaron Hertzmann. Style machines. *Proceedings of SIGGRAPH 2000*, pages 183–192, July 2000.

[7] Christoph Bregler, Michele Covell, and Malcolm Slaney. Video Rewrite: Driving Visual Speech with Audio. *Proceedings of SIGGRAPH 97*, pages 353–360, August 1997.

[8] Ian Buck, Adam Finkelstein, Charles Jacobs, Allison Klein, David H. Salesin, Joshua Seims, Richard Szeliski, and Kentaro Toyama. Performance-driven hand-drawn animation. *NPAR 2000: First International Symposium on Non Photorealistic Animation and Rendering*, pages 101–108, June 2000.

[9] Kenneth Castleman. *Digital Image Processing*. Prentice-Hall, 1996.

[10] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. *Proceedings of SIGGRAPH 97*, pages 421–430, August 1997.

[11] Alexei Efros and Thomas Leung. Texture Synthesis by Non-parametric Sampling. *7th IEEE International Conference on Computer Vision*, 1999.

[12] Alexei A. Efros and William T. Freeman. Quilting for Texture Synthesis and Transfer. *Proceedings of SIGGRAPH 2001*, August 2001.

[13] Alex Eilhauer, Alice Pritikin, Dylan Weed, and Steven J. Gortler. Combining Textures and Pictures with Specialized Texture Synthesis, 2000. http://www.people.fas.harvard.edu/~pritikin/cs/graphics/.

[14] Chris Eliasmith. Dictionary of Philosophy of Mind. http://artsci.wustl.edu/~philos/MindDict/.

[15] T.G. Evans. A program for the solution of geometric analogy intelligence test questions. In M. Minsky, editor, *Semantic Information Processing*. MIT Press, 1968.

[16] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, 1990.

[17] W. T. Freeman, E. C. Pasztor, and O. T. Carmichael. Learning Low-Level Vision. *Intl. J. Computer Vision*, 40(1):25–47, 2000. See also http://www.merl.com/reports/TR2000-05/.

[18] William T. Freeman, Joshua B. Tenenbaum, and Egon Pasztor. An example-based approach to style translation for line drawings. Technical Report TR99-11, MERL, February 1999.

[19] D. Gentner. Structure mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[20] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

[21] Paul E. Haeberli. Paint By Numbers: Abstract Image Representations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 207–214, August 1990.

[22] J. Hamel and T. Strothotte. Capturing and re-using rendition styles for non-photorealistic rendering. *Computer Graphics Forum*, 18(3):173–182, September 1999.

[23] David J. Heeger and James R. Bergen. Pyramid-Based Texture Analysis/Synthesis. *Proceedings of SIGGRAPH 95*, pages 229–238, August 1995.

[24] Aaron Hertzmann. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. In *SIGGRAPH 98 Conference Proceedings*, pages 453–460, July 1998.

[25] Aaron Hertzmann. *Algorithms for Rendering in Artistic Styles*. PhD thesis, New York University, May 2001.

[26] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000.

[27] Youichi Horry, Ken ichi Anjyo, and Kiyoshi Arai. Tour Into the Picture: Using a Spidery Mesh Interface to Make Animation from a Single Image. *Proceedings of SIGGRAPH 97*, pages 225–232, August 1997.

[28] William James. *The Principles of Psychology*. 1890.

[29] Bela Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 290:91–97, 1981.

[30] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Corrêa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000*, pages 527–534, July 2000.

[31] Arthur Koestler. *The Act of Creation*. Picador, London, 1964.

[32] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-Based Rendering of Fur, Grass, and Trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999.

[33] G. Lakoff and M. Johnson. *Metaphors we live by*. University of Chicago Press, Chicago, IL, 1980.

[34] Thomas Leung and Jitendra Malik. Recognizing surfaces using three-dimensional textons. *7th IEEE International Conference on Computer Vision*, September 1999.

[35] Jitendra Malik, Serge Belongie, Jianbo Shi, and Thomas Leung. Textons, Contours, and Regions: Cue Integration in Image Segmentation. *7th IEEE International Conference on Computer Vision*, September 1999.

[36] Barbara J. Meier. Painterly Rendering for Animation. In *SIGGRAPH 96 Conference Proceedings*, pages 477–484, August 1996.

[37] Pietro Perona and Jitendra Malik. Scale-Space and Edge Detection using Anisotropic Diffusion. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12:629–639, December 1990.

[38] Ferdinand Petrie and John Shaw. *The Big Book of Painting Nature in Watercolor*. Watson-Guptill Publications, 1990.

[39] Kris Popat and Rosalind W. Picard. Cluster-based probability model and its application to image and texture processing. *IEEE Trans. on Image Processing*, 6(2):268–284, February 1997.

[40] J. Portilla and E. P. Simoncelli. A Parametric Texture Model based on Joint Statistics of Complex Wavelet Coefficients. *International Journal of Computer Vision*, 40(1):49–71, December 2000.

[41] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen–And–Ink Illustration. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, pages 101–108, July 1994.

[42] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image-Based Pen-and-Ink Illustration. In *SIGGRAPH 97 Conference Proceedings*, pages 401–406, August 1997.

[43] Arno Schödl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video Textures. *Proceedings of SIGGRAPH 2000*, pages 489–498, July 2000.

[44] K. Schunn and K. Dunbar. Priming, Analogy and Awareness in complex reasoning. *Memory and Cognition*, 24:271–284, 1996.

[45] Eero P. Simoncelli and William T. Freeman. The Steerable Pyramid: A Flexible Architecture for Multi-Scale Derivative Computation. *Proc. 2nd Int'l Conf on Image Processing*, October 1995.

[46] Oleg Veryovka and John W. Buchanan. Comprehensive Halftoning of 3D Scenes. *Computer Graphics Forum*, 18(3):13–22, September 1999.

[47] Oleg Veryovka and John W. Buchanan. Halftoning With Image-Based Dither Screens. *Graphics Interface '99*, pages 167–174, June 1999.

[48] B. Wandell. *Foundations of Vision*. Sinauer Associates Inc., 1995.

[49] Li-Yi Wei and Marc Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.

[50] Georges Winkenbach and David H. Salesin. Computer–Generated Pen–And–Ink Illustration. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, pages 91–100, July 1994.

[51] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. In *SIGGRAPH 96 Conference Proceedings*, pages 469–476, August 1996.

[52] P.H. Winston. Learning and Reasoning by Analogy. *Communications of the ACM*, (23) 12, December 1980.

[53] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective panoramas for cel animation. *Proceedings of SIGGRAPH 97*, pages 243–250, August 1997.

[54] Song Chun Zhu, Ying Nian Wu, and David Mumford. Filters, Random fields, And Maximum Entropy: Towards a Unified Theory for Texture Modeling. *International Journal of Computer Vision*, 12(2):1–20, March/April 1998.

Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware

Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, Dinesh Manocha

University of North Carolina at Chapel Hill
Department of Computer Science

Abstract: We present a new approach for computing generalized 2D and 3D Voronoi diagrams using interpolation-based polygon rasterization hardware. We compute a discrete Voronoi diagram by rendering a three dimensional distance mesh for each Voronoi site. The polygonal mesh is a bounded-error approximation of a (possibly) non-linear function of the distance between a site and a 2D planar grid of sample points. For each sample point, we compute the closest site and the distance to that site using polygon scan-conversion and the Z-buffer depth comparison. We construct distance meshes for points, line segments, polygons, polyhedra, curves, and curved surfaces in 2D and 3D. We generalize to weighted and farthest-site Voronoi diagrams, and present efficient techniques for computing the Voronoi boundaries, Voronoi neighbors, and the Delaunay triangulation of points. We also show how to adaptively refine the solution through a simple windowing operation. The algorithm has been implemented on SGI workstations and PCs using OpenGL, and applied to complex datasets. We demonstrate the application of our algorithm to fast motion planning in static and dynamic environments, selection in complex user-interfaces, and creation of dynamic mosaic effects.

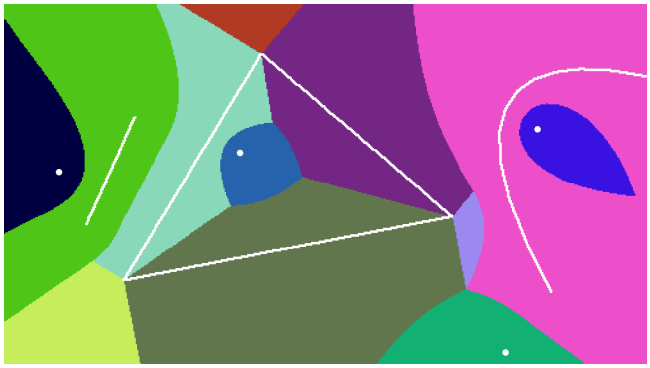
CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.3 [Computer Graphics]: Picture/Image Generation.

Additional Key Words: Voronoi diagrams, graphics hardware, polygon rasterization, interpolation, motion planning, proximity query, medial axis, OpenGL, framebuffer techniques.

1 INTRODUCTION

Given a set of primitives, called Voronoi sites, a Voronoi diagram partitions space into regions, where each region consists of all points that are closer to one site than to any other. Voronoi diagrams have been used in a number of applications including visualization of medical datasets, proximity queries, spatial data manipulation, shape analysis, computer animation, robot motion planning, modeling spatial structures and processes, pattern recognition, and locational optimization. The concept of Voronoi diagrams has been around for at least four centuries, and since the 1970s, algorithms for computing Voronoi diagrams of geometric primitives have been developed in computational geometry and related areas.

e-mail: {hoff,culver,keyser,lin,dm}@cs.unc.edu
WWW: http://www.cs.unc.edu/~geom/voronoi/



Cover Plate: Discrete approximation of the generalized Voronoi diagram of four points, a line, a triangle, and one cubic Bézier curve computed interactively on a PC.

Good theoretical and practical algorithms are known for computing ordinary Voronoi diagrams of points in any dimension. Ordinary Voronoi diagrams can be generalized in many different ways by using different distance functions and site shapes. A common generalization is to compute the diagram for higher-order sites, such as lines and curves. This greatly increases the complexity since the boundaries of the diagram are composed of high-degree algebraic curves and surfaces, and their intersections; the boundaries of an ordinary point Voronoi diagram are linear. No practically efficient and numerically robust algorithms are known for constructing a topologically consistent, continuous representation of generalized Voronoi diagrams.

Given the practical complexity of computing an exact generalized Voronoi diagram, many authors have proposed approximate algorithms. Interesting approaches include computing the Voronoi diagram of a point-sampling of the sites, adaptively subdividing space to locate the Voronoi boundary, and point-sampling the space to form a volumetric representation of the diagram. In practice, these previous algorithms take considerable time and memory on large numbers of input sites, or are restricted in generality.

Main Contributions: In this paper, we present an approach that computes discrete approximations of generalized Voronoi diagrams to an arbitrary resolution using polygon rasterization hardware. Our contributions include:

1. Efficient methods to approximate the distance function, with bounded error, for points, lines, polygons, polyhedra, curves, and curved surfaces using a polygonal mesh that is linearly interpolated by graphics hardware.

2. Efficient algorithms to find Voronoi boundaries and neighbors, and to construct Delaunay triangulations.

3. Techniques to construct weighted and farthest-site generalized Voronoi diagrams in 2D and 3D.
4. Demonstration of the effectiveness of our approach to the following applications:
   - Fast motion planning in static and dynamic environments
   - Selection in complex user-interfaces
   - Generation of dynamic mosaics

The resulting techniques have been effectively implemented on PCs and high-end SGI workstations using the OpenGL graphics library. A 2D example computed in real-time is shown in the cover plate. Our techniques improve upon the state of the art in following ways:

- **Generality**: We make no assumption with respect to input primitives. We only need to mesh the distance function of a site over a grid of point samples.
- **Efficiency**: We show that our approach is quite fast. Its speed arises from using coarse polygonal approximations of the distance functions while still maintaining a specified error bound, using polygon rasterization hardware to reconstruct the distance values, and using the Z-buffer depth comparison to perform distance comparisons. We demonstrate the 2D approach on models composed of nearly 100K triangles in a real-time motion planning application through a complex dynamic scene. We derive efficient meshing strategies for polygonal models in 3D, and show the results of a prototype implementation that demonstrates its potential.
- **Tight Bounds on Accuracy**: Although our approach produces a discretized Voronoi diagram, all sources of error are enumerated and techniques are given to produce output within any specified tolerance.
- **Ease of Implementation**: The approach can be easily implemented on current graphics systems. The special cases are limited and the problem reduces to simply meshing a distance function for any new site.

2 RELATED WORK

The concept of Voronoi diagrams has been around for at least four centuries. In his treatment of cosmic fragmentation in *Le Monde de Mr. Descartes, ou Le Traite de la Lumière*, published in 1644, Descartes uses Voronoi-like diagrams to show the disposition of matter in the solar system and its environment. The first presentations of this concept appeared in the work of [Diric50] and [Voron08]. Algorithms for computing Voronoi diagrams have been appearing since the 1970s. See the surveys by [Auren91] and [Okabe92] on various algorithms, applications, and generalizations of Voronoi diagrams.

2.1 Voronoi Diagrams of Points

Among the algorithms known for computing Voronoi diagrams of points in 2D, 3D, and higher dimensions are the divide-and-conquer algorithm proposed by [Shamo75] and Fortune's sweepline algorithm [Fortu86]. Numerically robust algorithms for constructing topologically consistent Voronoi diagrams have been proposed by [Inaga92, Sugih94]. A number of implementations in exact and floating-point arithmetic are also available.

2.2 Generalized Voronoi Diagrams

Algorithms have been proposed for constructing Voronoi diagrams of higher order sites. Two broad approaches based on incremental and divide-and-conquer techniques have been summarized in [Okabe92]. The set of algorithms includes divide-and-conquer algorithms for polygons [Lee82, Held97], an incremental algorithm for polyhedra [Milen93b], and 3D tracing for polyhedral models [Milen93, Sherb95, Culve99]. Curved sites and CSG objects are handled in [Chian92, Dutta93, Hoffm94]. In all these cases, the computation of generalized Voronoi diagrams involves representing and manipulating high-degree algebraic curves and surfaces and their intersections. As a result, no efficient and numerically robust algorithms are known for computing them.

2.3 Approximate Voronoi Diagrams

Many authors compute approximations of generalized Voronoi diagrams based on the Voronoi diagram of a point-sampling of the sites [e.g. Sheeh95]. However, deriving any error bounds on the output of such an approach is difficult, and the overall complexity is not well understood.

[Vleug95] and [Vleug96] have presented an approach that adaptively subdivides space into regular cells and computes the Voronoi diagram up to a given precision. [Laven92] uses an octree representation of objects and performs spatial decomposition to compute the approximation. [Teich97] computes a polygonal approximation of Voronoi diagrams by subdividing the space into tetrahedral cells. All these algorithms take considerable time and memory for large models composed of tens of thousands of triangles, and cannot easily be extended to directly handle dynamic environments.

The idea of using polygon rasterizing hardware and rendering of cones to construct 2D Voronoi diagrams of points is suggested in [Haebe90] and in the OpenGL 1.1 Programming Guide [Woo97].

2.4 Graphics Hardware

Polygon rasterization graphics hardware has been used for a number of geometric computations, such as visualization of constructive solid geometry models [Rossi86, Goldf89] and interactive inspection of solids, including cross-sections and interferences [Rossi92]. Algorithms for real-time motion planning using raster graphics hardware have been proposed by [Lengy90].

3 OVERVIEW

In this section, we present the basic concepts important to our approach. We give a formal definition of generalized Voronoi diagrams and present a simple brute-force strategy for computing a discrete approximation. We then show how we may greatly accelerate this using graphics hardware.

3.1 Generalized Voronoi Diagrams

The set of input sites is denoted as $A_1, A_2, \ldots, A_k$. For any point p in the space, $dist(p, A_i)$ denotes the distance from the point p to the site A_i. The dominance region of A_i over A_j is defined by

$$Dom(A_i, A_j) = \{ p \mid dist(p, A_i) \leq dist(p, A_j) \}$$

For a site A_i, the Voronoi region for A_i is defined by

$$V(A_i) = \cap_{j \neq i} Dom(A_i, A_j)$$

The partition of space into $V(A_1)$, $V(A_2)$, …, $V(A_k)$ is called the *generalized Voronoi diagram*. The (ordinary) Voronoi diagram corresponds to the case when each A_i is an individual point. The boundaries of the regions $V(A_i)$ are called *Voronoi boundaries*. For primitives such as points, lines, polygons, and splines, the Voronoi boundaries are portions of algebraic curves or surfaces.

3.2 Discrete Voronoi Diagrams

Perhaps the simplest way to compute a discrete Voronoi diagram is to uniformly point-sample the space containing Voronoi sites. For each sample point, we find the closest site and its distance. Associating each point in space with its closest sample point induces a uniform subdivision into rectangular cells. For any point, we know the distance to the closest site to within the maximum distance between a point in space and a sample point, i.e. half the diagonal length of a cell.

A simple brute-force approach to find the closest sites is to iterate through all sample points, computing distances to all sites and recording the closest site and distance. The algorithm can be rearranged to iterate through the sites: for each site, compute distances to all sample points and update the current closest site and distance. The second arrangement is amenable to an implementation in graphics hardware.

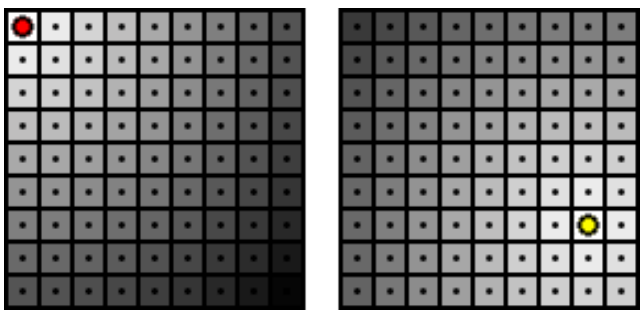


Figure 1: Image of the sampled distance functions for two point sites. Uniform point sampling induces a rectangular cell subdivision of space.

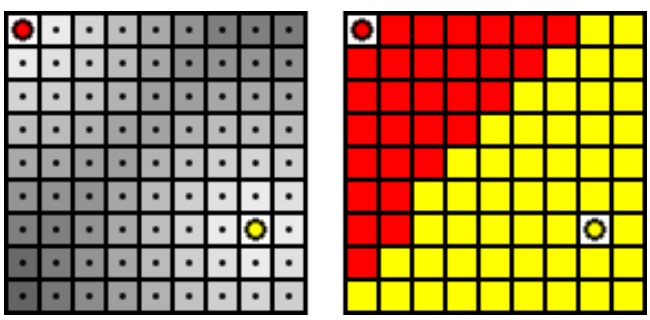


Figure 2: The two distance images are composited through a distance comparison operation. The current closest site and the distance to each site is updated based on the lesser distance value. The resulting Voronoi diagram is composed of a distance image (left) and an closest-site ID image (right).

3.3 Polygon Rasterization Hardware

Our approach makes use of standard Z-buffered raster graphics hardware for rendering polygons. The frame buffer stores the attributes (intensity or shade) of each pixel in the image space; the Z-buffer, or depth buffer, stores the *z*-coordinate, or depth, of every visible pixel. Given only the vertices of a triangle, the rasterization hardware uses linear interpolation to compute depth values across the triangle's surface. All raster samples covered by a triangle have an interpolated *z*-value.

3.4 Our Approach

A key concept for our approach is that of the *distance function* for a site, which gives, for any point, the distance to that site. The main idea of our approach is to render a polygonal mesh approximation to each site's distance function. Each site is assigned a unique color ID, and the corresponding distance mesh is rendered in that color using a parallel projection. We make use of two components of the graphics hardware: linear interpolation across polygons and the Z-buffer depth comparison operation. When rendering a polygonal distance mesh, the polygon rasterization reconstructs all distances across the mesh. The Z-buffer depth test compares the new depth value to the previously stored value. If the new value is less, the Z-buffer records the new distance, and the color buffer records the site's ID. In this way, each pixel in the frame buffer will have a color corresponding to the site to which it is closest, and the depth-buffer will have the distance to that site. In order to maintain an accurate Voronoi diagram, we bound the error of the mesh to be smaller than the distance between two sample points.

Our approach is inspired by an interesting sidenote in the OpenGL 1.1 Programming Guide [Woo97]. In the Section "Now That You Know" on "Dirichlet Domains", the authors briefly discuss a simple method to construct discretized 2D Voronoi diagrams for points using OpenGL graphics hardware. The authors mention the use of cones for Voronoi diagrams of points in 2D, but warn that the technique "might require thousands of polygons." We show that we can render cones using fewer than 100 triangles for a 1K×1K resolution grid and achieve the same level of accuracy. In addition, we generalize this approach to higher-order sites in both two and three dimensions.

4 THE DISTANCE FUNCTIONS

For both 2D and 3D, our discrete Voronoi diagram computation has been reduced to finding a 3D polygonal mesh approximation to the distance function of a Voronoi site over a planar 2D rectangular grid of point samples. The error in the approximation must be bounded so that by rendering this mesh using graphics hardware, we can efficiently and accurately compute the distances between the site and all of the point samples.

In this section, we describe the distance functions associated with various sites, and provide efficient methods for meshing these functions within a specified error tolerance.

4.1 2D Voronoi Diagrams

Denote the distance from a site A to each pixel location (x,y) by $dist(A,(x,y))$. The *distance function* of A is given by $d(x,y)=dist(A,(x,y))$. Meshing this function corresponds to approximating the graph of $d(x,y)$ with a polygonal model.

The three basic types of 2D sites are points, lines, and polygons. Their corresponding distance functions are shown in the table. In this section, we present algorithms for computing distance meshes for each of them.

2D site	Shape of Distance Function	Figure
Point	Right circular cone	3a
Line segment	"Tent"	3b
Polygon	Cones and tents	5

Table 1: Shape of Distance Functions for 2D Sites

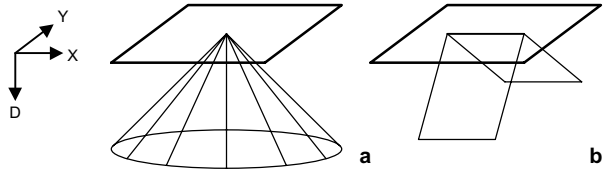


Figure 3: The distance meshes used for a point (left) and a line segment (right). The XY-plane containing the site is shown above each mesh.

4.1.1 Points in 2D

The distance function for a point in the plane is a right circular cone. We approximate cones as a triangle fan proceeding radially outward from the apex (Figures 3a and 4-left). A point's Voronoi region can potentially extend to any portion of the region of interest, and thus the radius at the cone's base must be of size $M\sqrt{2}$ if the scene is contained in an $M\times M$ square.The mesh's radial lines lie on the cone. The maximum error in distance occurs at the cone base between adjacent vertices. Because the cone is right circular, the error in approximating the circular base as viewed from above is equal to the error in distance.

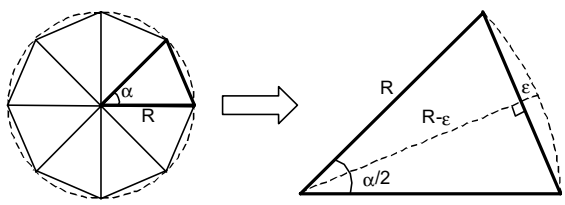


Figure 4: A single triangle of the meshed point distance function cone. α is the angle we wish to maximize, R is the radius of the cone (max dist between site and sample pt), and ε is the max error.

From this formulation (see Figure 4), we compute the maximum angle as:

$$\cos(\alpha/2) = \frac{R-\varepsilon}{R} \;\rightarrow\; \alpha = 2\cos^{-1}\left(\frac{R-\varepsilon}{R}\right)$$

For example, for a maximum distance error of no more than one pixel's width, a cone mesh for a 512×512 grid will require only 60 triangles. A 1024×1024 grid will require 85 triangles.

4.1.2 Line Segments in 2D

The distance function for a line segment is composed of three parts: one for the segment itself and one for each endpoint. The endpoints are treated the same way as points. The distance function for the line segment (excluding the endpoints) is just a "tent" (Figure 3b); its distance mesh is composed of two quadrilaterals. These represent the distance function exactly, so there is no error in the distance mesh representation. The only error for the line segment is in the cone mesh for the endpoint distance functions, as described in the previous section.

4.1.3 Polygons and Per-feature Voronoi Diagrams

It is often useful to consider sites as a collection of features, rather than as a single entity. For example, a line segment would be considered as three features: the two endpoints and the linear edge between them. By rendering the distance meshes for different features in different colors, we obtain a discrete approximation of a *per-feature Voronoi diagram*. Such diagrams are useful in several contexts: for example, the computation of a medial axis of a polygon. A picture of a per-feature Voronoi diagram for a polygon is given in Figure 5-left.

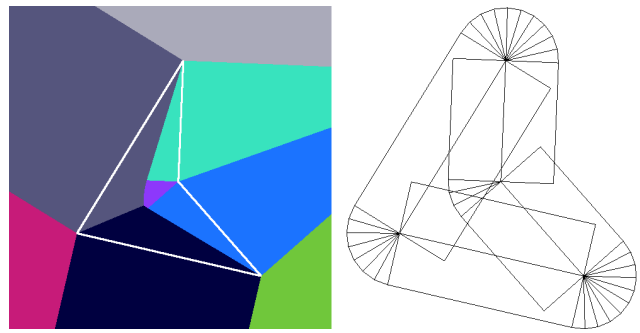


Figure 5: The per-feature Voronoi diagram of a quadrilateral (left). The corresponding distance mesh (right).

Polygons are rendered as a series of linear segments connected at the vertices. Each edge and vertex is a feature. For the vertices, rendering a triangle fan connecting two adjacent edges, rather than a full point distance mesh cone, saves on the total number of triangles computed and ensures that the distance meshes for adjacent features join smoothly. See Figure 5-right for an illustration.

4.2 3D Voronoi Diagrams

Our algorithm computes a 3D discrete Voronoi diagram slice-by-slice. Each slice is parallel to the (x,y)-plane and is computed independently.

Consider the slice $z=z_0$. To construct the intersection of the Voronoi diagram with this slice, consider the distance function for a site A, restricted to the slice. Denote the restricted distance function by $dist(x,y)=dist(A,(x,y,z_0))$. In this section, we describe $dist(x,y)$ for polygon, line segment, and point sites. As in the 2D case, computing the discrete Voronoi diagram is a matter of meshing the distance function $d=dist(x,y)$ for each site and rendering these meshes.

The distance meshes we give for the 3D problem are for a per-feature Voronoi diagram. Thus, a detached triangle site is treated as seven features: a polygon, three line segments, and three points. As in 2D per-feature diagrams, some features have a restricted region of influence.

3D site	Shape of distance function	Figure
Polygon	Plane	6
Line segment	Elliptical cone	7
Point	1 sheet of a hyperboloid of 2 sheets	8

Table 2: Shape of Distance Functions for 3D Sites

4.2.1 Polygons in 3D

The influence of this site in 3D is confined to the region formed by sweeping the polygon orthogonally through space, since points outside this region are considered to be closer to an edge or vertex of the polygon. In the slice, this region is a polygon, and $dist(x,y)$ is linear within this region, as illustrated in Figure 6. The distance to the site is computed at the vertices of the region, and a distance mesh composed of a single polygon is rendered. No meshing error

is incurred. If the polygon intersects the slice, the intersection is computed and the polygon is decomposed into two sub-polygons. Each sub-polygon is treated as above.

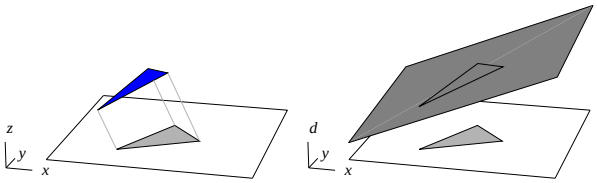


Figure 6: A polygonal site and its region of influence in a slice (left). The corresponding linear distance function (right).

4.2.2 Line Segments in 3D

The graph of the distance function for a line segment site is an elliptical cone (Figure 7). The apex of the cone lies at the intersection of the segment's line with the slice, and the cone's eccentricity is determined by the relative angle of the line and the slice. The 3D region of influence of a line segment lies between two parallel planes through the endpoints, since a point outside these planes is closer to one of the endpoints than to the segment.

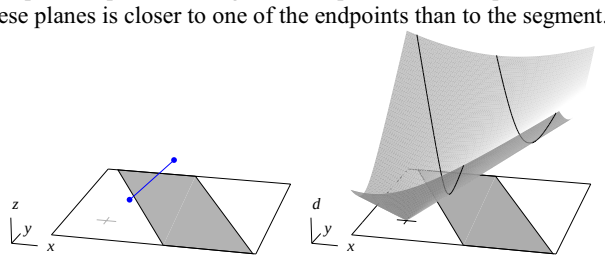


Figure 7: A line-segment site and its region of influence in a slice (left). The corresponding conical distance function (right).

4.2.3 Points in 3D

The distance function for a point site is shown in Figure 8. Its graph is one sheet of a hyperboloid of revolution of two sheets. If the point lies in the slice, the distance function is a cone rather than a hyperboloid. The region of influence for a single point is the entire slice.

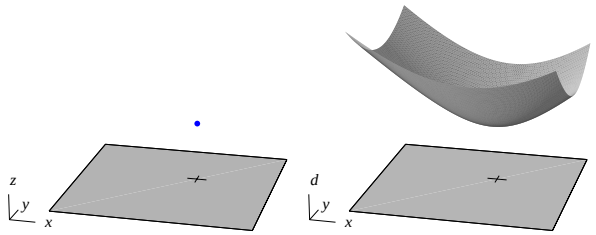


Figure 8: A point site and its region of influence in a slice (left). The corresponding hyperbolic distance function (right).

4.2.4 Meshes for Line Segments and Points in 3D

The construction of bounded-error meshes for the line-segment and point distance functions is detailed in [Hoff99]. The method attempts to minimize the complexity of the mesh by committing the maximum allowable error ε in each mesh cell. The structure of the mesh depends only on the *resolution* of the Voronoi diagram, defined by the ratio of the diameter M of the model to the maximum meshing error ε. The mesh structure is precomputed; during the Voronoi diagram construction, the mesh is constructed using table-lookup. Examples of the meshes produced by this method are shown in Figure 9.

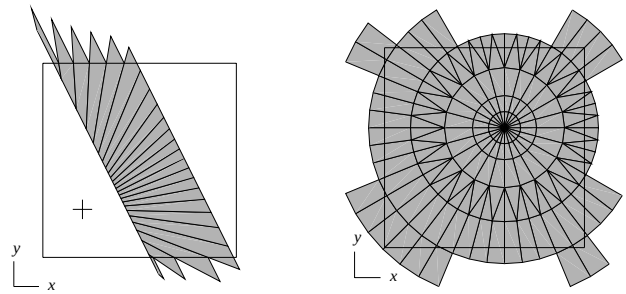


Figure 9: A bounded-error distance mesh for the line-segment site (left) and the point site (right).

4.3 Generalization to Curved Sites

The exact distance function for a curved site can be rather complicated, and for splines or algebraic curves is a high-degree algebraic function. We simplify this by creating a linear tessellation of the curved site, and then meshing the distance function of this approximation. We can use algorithms such as in [Filip87] and [Kumar96] to obtain bounded-error tessellations.

Figure 10 shows the mesh for a Bézier curve. Since the mesh for a linear segment is exact, the distance error for any of the linear segments is just the error in the deviation of the line from the original curve. The endpoints of the curve must be treated as points, just as for the line segment. The distance mesh for the "joints" between linear segments is a portion of the radial mesh of triangles. An overall maximum error bound of ε can be obtained for the entire curve by:

- tessellating the curve into linear segments with maximum error bound of ε;
- rendering the distance mesh for the linear segments; and
- treating the endpoints and joints as points, and rendering each point distance mesh with maximum error bound of ε.

This approach generalizes to 3D surfaces, which can be tesselated into a polygonal mesh. The error is bounded in a similar way.

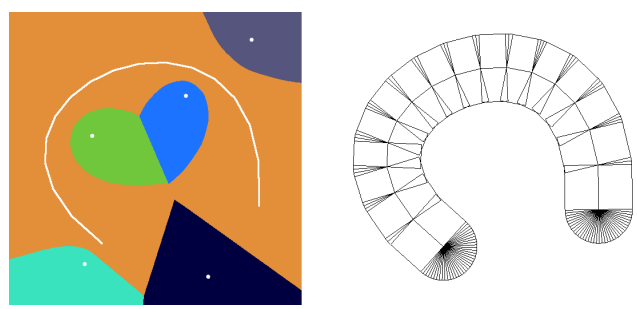


Figure 10: The Voronoi diagram of a Bézier curve and 5 points (left). The distance mesh for the Bézier curve that has been tessellated into 16 segments (right).

4.4 Weighted and Farthest-site Diagrams

In a *weighted* Voronoi diagram, the distance functions are additively or multiplicatively weighted [Okabe92]. Translation of a distance mesh along the distance axis accounts for additive

weights. Linear scaling along the distance axis accounts for multiplicative weights. In 2D, this is equivalent to changing the angle of the cone or tent. Scaling the distance mesh also scales the meshing error.

In a *farthest-site* Voronoi diagram, the farthest site from each point is found. Unlike in the nearest-site diagram, the distance function monotonically decreases as we move away from the site. We obtain the proper distance relationships by negating the distance functions. In practice, however, we need only reverse the depth-test (less-than to greater-than) and change the depth initialization from ∞ to 0.

Figure 11: Standard nearest-site Voronoi regions (left). Farthest regions for the same sites (middle). Weighted regions (right). Weights: line, 2; dark point, 1; light point, 0.5.

5 BOUNDARIES AND NEIGHBORS

A continuous Voronoi diagram representation usually specifies the *Voronoi boundaries* that separate the set of Voronoi regions. In our discrete representation, we must search for the boundaries using approaches similar to iso-surface extraction and root-finding techniques [Bloom97]. However, instead of trying to bracket zero-crossings between sample points where iso-surface functions evaluate to values of opposite sign, we simply find the boundaries in the space between pixel samples of different color. Using the same approaches, we can either point-sample the boundary or compute an approximate mesh representation. In order to increase the precision, we must either use a higher overall resolution or adaptively refine.

One approach is to examine each pair of adjacent cells in 2D or 3D. If the colors are different, the location between the samples is marked as a point on the Voronoi boundary. The operation is very simple and can be accelerated through image operations in graphics hardware.

Another approach is based on a *continuation* method that starts at a point known to be on the boundary and walks along the boundary until all boundary points have been found [Bloom97]. Since we only compare locations near known boundaries, it is output sensitive. The correctness of the continuation method depends on whether the Voronoi boundaries are connected. The boundaries of a generalized Voronoi diagram of a collection of convex sites are always connected, so the method is correct for inputs consisting of point, line-segment, or convex polygonal sites. The method may fail in the presence of curves, curved surfaces, or concave sites where the generalized Voronoi diagram may have isolated components.

In this approach, at least one boundary point must be known as a "seed" value. Assuming convex sites, some Voronoi boundary passes through the edge of the bounded region in which we are computing the diagram, so the method begins by examining every window border pixel. When all Voronoi boundaries are connected only one seed point is needed since all others can be reached from that first point. Starting from a seed point, we recursively check all neighbors that are a different color from the current pixel's. All visited pixels are marked and avoided in the recursion.

This algorithm also finds the *Voronoi neighbors*–pairs of sites that share a Voronoi boundary. This concept is useful in a wide variety of applications, including computing the dual of the ordinary Voronoi diagram–the Delaunay triangulation. The boundary finding algorithms find pairs of adjacent pixels with different colors. The sites corresponding to those two colors are reported to be Voronoi neighbors. Connecting Voronoi neighbors with line segments constructs the Delaunay triangulation.

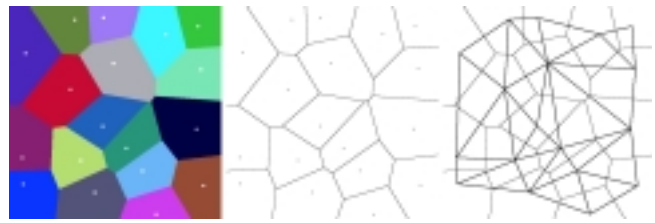


Figure 12: Voronoi diagram of set of 2D points (Left). Boundaries found with continuation-based approach (middle). Delaunay triangulation by connecting neighboring sites (right).

6 SOURCES OF ERROR

In this section we analyze all sources of error in our approach, and discuss how to reduce this error. We consider two broad categories: error in distance approximation and combinatorial error.

6.1 Distance Error

Distance error is the error in the distance computed from a pixel to a site. There are three sources of distance error:

- *Meshing error*, from approximating the true distance function by the distance mesh. We discussed how to bound this error in Section 4.
- *Tessellation error*, from tessellating a curved site into a number of linear sites. The tessellation algorithms presented in [Filip87, Kumar96] give tight bounds. Tessellation error is reduced by using a finer approximation to the site.
- *Hardware precision error,* from the use of fixed-precision arithmetic (integer or floating-point) during rasterization. Hardware precision error cannot be removed without resorting to multiple-precision arithmetic, but hardware error is usually negligible compared to meshing error.

These errors are additive–i.e. the error from one source is not magnified by the other sources. The total distance error is at most the sum of the errors from these three sources.

6.2 Combinatorial Error

Combinatorial error refers to qualitative error as opposed to quantitative. For example, a pixel is assigned the wrong color, or the algorithm reports an incorrect pair of Voronoi neighbors. There are three sources that contribute to combinatorial error:

- *Distance error*, as described in the previous section. With significant distance error, depth comparison at a pixel may make a farther site appear closer, causing the pixel to be colored incorrectly.

- *Resolution error*, a result of discrete sampling. If this sampling is too coarse, we may miss some Voronoi regions or find spurious neighbors. Handling resolution error is described below.
- *Z-buffer precision error*, the limitations of the number of bits of precision provided by the Z-buffer. Current graphics systems have 24 bits or 32 bits of precision for each pixel in the Z-buffer, which is more than the 23 bits provided in standard floating-point. If the distances between two pixels cannot be determined within that precision, the Z-buffer cannot accurately choose the correct color. This effect is small when compared to the other two, but can be significant at very high resolutions with very little distance error. A higher-precision Z-buffer can be simulated in software at a significant loss in efficiency.

Adaptive resolution allows us to "zoom in" on a region of interest, reducing potential resolution error. This involves identifying a window of interest and applying the appropriate linear transformation for zooming into that region. Figure 13 shows an example. Note that when zooming in, sites outside of the viewing region can still have Voronoi regions inside the region. Thus, the "maximum distance to a site" must be adjusted appropriately when computing the distance error bounds.

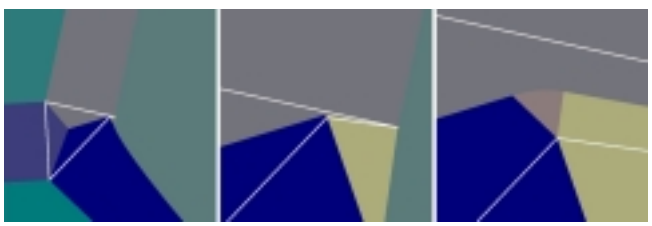


Figure 13: Adaptive resolution allows us to zoom in on features that could otherwise be missed.

Resolution error can cause a number of combinatorial problems, such as missing the entire Voronoi region of a site. One such example is shown in Figure 14 (left two images). When no cell has the color of a particular site, we can separately render the site itself, computing the pixels covering that site. By zooming around those pixels, we will find pixels in the Voronoi region of that site. The same technique can be applied to cells in 3D. Another problem arising from resolution error is incorrectly finding Voronoi neighbors (shown in Figure 14 – right two images). This problem (when due solely to resolution error) can be alleviated by adaptively zooming in on all boundary pixels.

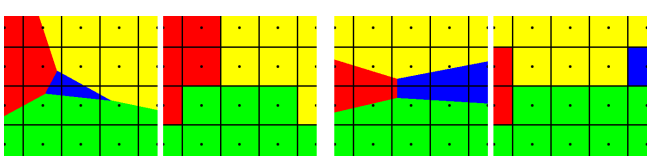


Figure 14: Problems caused by resolution error. An entire region in the center will be missed since it does not hit any pixel centers (left two images). The left and right regions, which should meet, become disconnected after rasterization (right two images).

6.3 Error Bounds

Distance error occasionally causes a pixel to be colored incorrectly. However, in a certain sense, the pixel is "almost" the right color. Assume that there is no Z-buffer precision error, and that we can bound the maximum distance error by ε, as described earlier. For a pixel P colored with the ID of site A and with a computed depth buffer value of D, we know that:

$$D - \varepsilon \le dist(P,A) \le D + \varepsilon$$

Furthermore, we know that for any other site B,

$$D - \varepsilon \le dist(P,B)$$

From this information, we easily determine that

$$dist(P,A) \le dist(P,B) + 2\varepsilon$$

where $dist(X,Y)$ means the distance from the center of pixel X to site Y. That is, if a pixel is colored with the ID of A, then site A is no more than 2ε farther from the pixel center than any other site. The same bound holds in 3D.

7 APPLICATIONS

There are many applications that benefit from fast computation of a discrete Voronoi diagram, an approximation to the distance function, or both. We describe three that we have implemented.

7.1 Motion Planning

Motion planning is a fundamental problem in robotics and computational geometry, with applications to the animation of digital actors, maintainability studies in virtual prototyping, and robot-assisted medical surgery. The classic Piano Mover's problem involves finding a collision-free path for a robot moving from one location (and orientation) to another in an environment filled with obstacles. Numerous approaches to this problem have been proposed, some of which are based on generalized Voronoi diagrams [Latom91]. The underlying idea is to treat the obstacles as sites. The Voronoi boundaries then provide paths of maximal clearance between the obstacles. Due to the practical complexity of computing generalized Voronoi diagrams, the applications of such planners have been limited to environments composed of a few simple obstacles.

Our discrete Voronoi computation algorithm can be applied to motion planning in both static and dynamic environments. The Voronoi algorithm computes the approximate distance to the nearest obstacle. The basic approach we implemented is based on the potential field method, which repels a robot away from the obstacles and towards the goal using a carefully designed artificial potential function. Other Voronoi diagram or distance-based approaches are also possible. The details of our motion planning algorithm are provided in [Hoff99].

We demonstrate our planner's effectiveness in a complex environment: the interior of a house, composed of over 100,000 triangles. We use the x- and y-components of the polygons to give the 2D input primitives for our algorithm. The robot has three degrees of freedom: x- and y-translation along the ground and rotation about the z-axis. Color plate 2 and the video show a sequence of piano motions automatically generated by our motion planner in a static environment. Color plate 2 also shows an image of the distance function for the house. We also apply our planner to environments with moving obstacles. Our video demonstrates the movement of a music stand through a house filled with moving furniture. The entire potential field and the motion planning sequence are computed in real time.

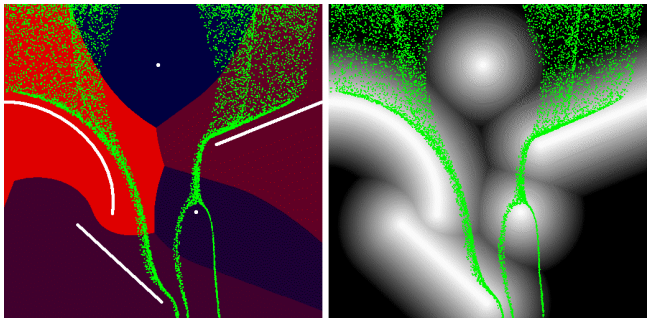


Figure 15: Motion planning of falling particles. Sites are avoided by using the Voronoi diagram's distance buffer (right) to create a potential field. This same principle is used in the rigid-body planner.

7.2 Selection in Complex User Interfaces

Complex 2D user interfaces sometimes require quick determination of the object nearest to the cursor. The Voronoi diagram of the interface can be used as a nearest-object lookup table indexed by sample points. Given the cursor position, it is simple to find the nearest sample point, and thus the nearest object. In some interfaces it may be desirable to know the distance to the selected object as well. We used this technique in our 2D implementation to allow the user to interactively move sites with the mouse.

7.3 Mosaics

We can use our approach for generating Voronoi diagrams to create an interesting artistic effect called mosaicing. A mosaic is a tiled image, where each tile has a single color. The Voronoi diagram of a point set can be used as a tiling [Haebe90]. Each Voronoi tile is colored with a color taken locally from the image. In our implementation, each tile is colored by the image pixel closest to the point site (see color plate 1). Our algorithm can perform this operation very quickly, allowing dynamic mosaics in which the mosaic tiling, the source image, or both may change in real time.

By randomly distributing point sites across an image, we obtain an effect similar to many mosaic filter effects seen in image editing programs. By clustering point sites around areas of higher detail, we obtain a classic tiling seen in many real-life mosaics where smaller tiles are used in areas of greater detail.

8 IMPLEMENTATION

For the 2D case, we implemented a complete interactive system incorporating all of the features and applications described here. Example output is shown throughout the paper. The video demonstrates interactive computation of more complex diagrams. In 3D, we show results from a prototype system that uses a simpler distance meshing strategy (see color plate 3 and the video for example output).

We implemented the 2D and 3D systems in C++ using the OpenGL graphics library and the GLUT toolkit. Any graphics API specification that uses a standard Z-buffered interpolation-based raster graphics system is sufficient to support the Voronoi diagram computation. Motion planning and the basic operations of boundary and neighbor finding require reading back of the color and depth buffers. Our system runs, without source modification, on both an MS-Windows-based PC and a high-end SGI Onyx2 with InfiniteReality Graphics. Surprisingly, the performance on a 400 Mhz Intel Pentium II PC with an Intergraph Intense 3D Pro 3410-T graphics accelerator was comparable to the SGI performance. In fact, in boundary finding, neighbor finding, and particle motion planning applications, the performance exceeded the high-end SGI. This was mainly due to intense buffer readback requirements. Each distance mesh must cover every pixel, so performance is bounded by the graphics hardware's pixel fill-rate. For large numbers of input sites, therefore, the SGI outperforms the PC.

When the distance-error tolerance is relaxed, the amount of geometry rendered for each site can be reduced, slightly improving performance. However, the biggest gains are achieved by reducing the number of pixels filled. In many practical cases, we can increase the performance significantly by bounding the site distance functions to a maximum distance. This allows reduction of the size of the distance meshes drawn so that only a portion of the screen is covered for each site. We exploit this observation to obtain interactive rates in the 1,000-point example shown in color plate 1, in the 10,000-point example shown in the video, and in the general case for the computation of the potential field used in the motion-planner. For closed higher-order primitives, such as polygons, we can further increase performance by restricting the distance function to only the inside or outside regions. This is useful in computing potential fields and medial axes.

9 CONCLUSIONS AND FUTURE WORK

We have presented a method for rapid computation of generalized discrete Voronoi diagrams in two and three dimensions using graphics hardware. We have presented techniques for creating a mesh of the distance function for each site with bounded error, and described how this distance mesh allows us to compute the Voronoi diagram rapidly. We have analyzed various sources of error, as well as how to bound or reduce those errors. Finally, we have demonstrated a few applications using our approach.

In the future, we would like to extend this work in the following ways: generalizations of distance functions and site geometry, further applications, other distance meshing strategies, and more acceleration techniques for the 3D Voronoi volume computation.

ACKNOWLEDGEMENTS

Supported in part by ARO Contract DAAH04-96-1-0257 and DAAG55-98-1-0322, NSF Career Award CCR-9625217, NSF grants EIA-9806027 and DMI-9900157, NIH Research Resource Award 2P41RR02170-13, ONR Young Investigator Award and Intel. We would also like to thank Sarah Hoff for extensive help with editing and color plates, Chris Weigle for suggesting mosaicing using 2D point Voronoi diagrams, the UNC-walkthrough group for the house model, and the reviewers for their helpful comments.

REFERENCES

[Auren91] F. Aurenhammer. *Voronoi Diagrams: A Survey of a Fundamental Geometric Data Structure*. ACM Computing Surveys, 23:345–405, 1991.

[Bloom97] J. Bloomenthal, C. Bajaj, J. Blinn, M-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc. San Francisco, CA. 1997.

[Chian92] C–S. Chiang. *The Euclidean Distance Transform*. Ph. D. thesis, Dept. Comp. Sci., Purdue Univ., West Lafayette, IN, August 1992. Report CSD-TR 92–050.

[Culve99] T. Culver, J. Keyser, and D. Manocha. *Accurate Computation of the Medial Axis of a Polyhedron*. Proc. of the Fifth Symp. on Solid Modeling and Applications. 1999.

[Dutta93] D. Dutta and C.M. Hoffmann. *On the Skeleton of Simple CSG Objects*. Journal of Mechanical Design, ASME Transactions, 115(1):87–94, 1993.

[Diric50] G.L. Dirichlet. *Uber die Reduktion der Positiven Quadratischen Formen mit Drei Unbestimmten Ganzen Zahlen*. J. Reine Angew. Math., 40:209–27, 1850.

[Filip87] D. Filip and R. Goldman. *Conversion from Bézier-rectangles to Bézier-triangles*. CAD, 19:25–27, 1987.

[Fortu86] S. Fortune. *A Sweepline Algorithm for Voronoi Diagrams*. In Proc. 2nd Annual ACM Symp. on Comp. Geom., pages 313–322, 1986.

[Goldf89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning*. IEEE Computer Graphics and Applications, 9(3):20–28, May 1989.

[Haebe90] P. Haeberli. *Paint by Numbers: Abstract Image Representation*. Computer Graphics (SIGGRAPH '90 Proc). vol. 25. pgs 207-214.

[Held97] M. Held. *Voronoi Diagrams and Offset Curves of Curvilinear Polygons*. Computer-Aided Design, 1997.

[Hoff99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. Technical Report TR99-011, Dept. of Comp. Sci., University of North Carolina at Chapel Hill, 1999.

[Hoffm94] C.M. Hoffmann. *How to Construct the Skeleton of CSG Objects*. In A. Bowyer and J. Davenport, editors. Proc. of the Fourth IMA Conference, The Mathematics of Surfaces, University of Bath, UK, Sept. 1990. Oxford University Press, New York, 1994.

[Inaga92] H. Inagaki, K. Sugihara, and N. Sugie. *Numerically Robust Incremental Algorithm for Constructing Three-dimensional Voronoi Diagrams*. In Proc. 4th Canad. Conf. Comp. Geom., pgs 334-339, 1992.

[Kumar96] S. Kumar, D. Manocha, and A. Lastra. *Interactive Display of Large NURBS Models*. IEEE Trans. on Vis. and Computer Graphics. vol 2, no 4, pgs 323-336, Dec 1996.

[Latom91] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Laven92] D. Lavender, A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark. *Voronoi Diagrams of Set-theoretic Solid Models*. IEEE Computer Graphics and Applications, 12(5):69–77, Sept 1992.

[Lee82] D.T. Lee. *Medial Axis Transformation of a Planar Shape*. IEEE Trans. Pattern Anal. Mach. Intell., PAMI–4:363–369, 1982.

[Lengy90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH '90 Proc.), vol. 24, pgs 327–335, Aug 1990.

[Milen93] V. Milenkovic. *Robust Construction of the Voronoi Diagram of a Polyhedron*. In Proc. 5th Canadian. Conference on Comp. Geom., pgs 473-478, 1993.

[Milen93b] V. Milenkovic. *Robust Polygon Modeling*. Computer Aided Design, 25(9), 1993. (special issue on Uncertainties in Geometric Design).

[Okabe92] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[Rossi92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. Computer Graphics (SIGGRAPH '92 Proc.), vol. 26, pgs 353–360, July 1992.

[Rossi86] J.R. Rossignac and A.A.G. Requicha. *Depth-buffering Display Techniques for Constructive Solid Geometry*. IEEE Computer Graphics and Applications, 6(9):29–39, 1986.

[Sheeh95] D.J. Sheehy, C.G. Armstrong, and D.J. Robinson. *Computing the Medial Surface of a Solid from a Domain Delaunay Triangulation*. In Proc. ACM/IEEE Symp. on Solid Modeling and Applications, May 1995.

[Shamo75] M.I. Shamos and D.Hoey. *Closest-point Problems*. In Proc. 16th Annual IEEE Symposium on Foundations of Comp. Sci., pages 151–162, 1975.

[Sugih94] K. Sugihara and M. Iri. *A Robust Topology-oriented Incremental Algorithm for Voronoi Diagrams*. International Journal of Comp. Geom. Appl., 4:179–228, 1994.

[Sherb95] E.C. Sherbrooke, N.M. Patrikalakis, and E. Brisson. *Computation of the Medial Axis Transform of 3D Polyhedra*. In Solid Modeling, pages 187–199. ACM, 1995.

[Teich97] M. Teichmann and S. Teller. *Polygonal Approximation of Voronoi Diagrams of a Set of Triangles in Three Dimensions*. Tech Rep 766, Lab of Comp. Sci., MIT, 1997.

[Vleug95] J. Vleugels and M. Overmars. *Approximating Generalized Voronoi Diagrams in Any Dimension*. Technical Report UU-CS-1995-14, Dept. of Comp. Sci., Utrecht University, 1995.

[Vleug96] J. Vleugels, V. Ferrucci, M. Overmars, and A. Rao. *Hunting Voronoi Vertices*. Comp. Geom. Theory Appl., 6:329–354, 1996.

[Voron08] G.M. Voronoi. *Nouvelles Applications des Paramètres Continus à la Théorie des Formes Quadratiques. Deuxième Mémoire: Recherches sur les Parallélloèdres Primitifs*. J. Reine Angew. Math., 134:198–287, 1908.

[Woo97] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*, Second Edition. Addison Wesley, 1997.

Plate 1: We can use the Voronoi diagram to create real-time mosaic effects. Left: Voronoi diagram computed for 1000 2D point sites at a resolution of 512x512. Center: Same diagram using the point locations to index into a 256x256 face texture to obtain the region colors. Right: The final effect at 512x512 with points removed. In the video, we show this same effect in real-time with 10,000 points.

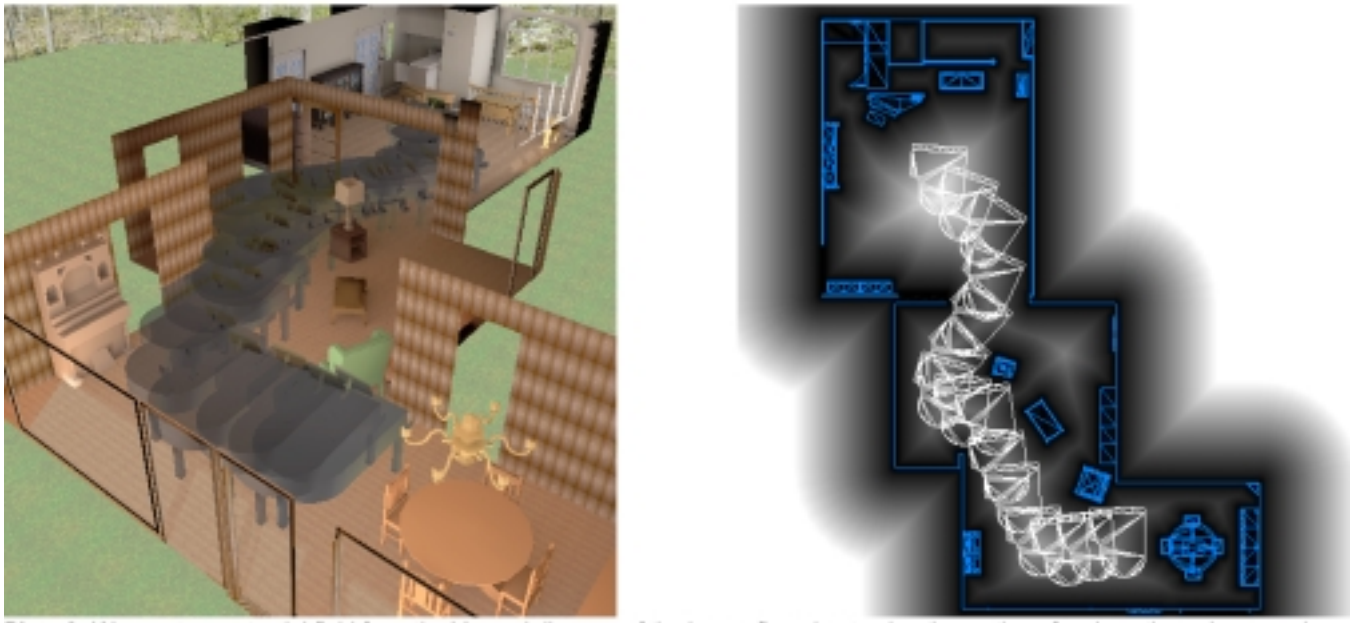


Plate 2: We create a potential field from the Voronoi diagram of the house floorplan to plan the motion of a piano through a complex static scene in real-time. Left: The piano motion sequence. Right: Motion sequence overlaying the floorplan distance function. The video demonstrates navigation through a dynamic scene in real-time.

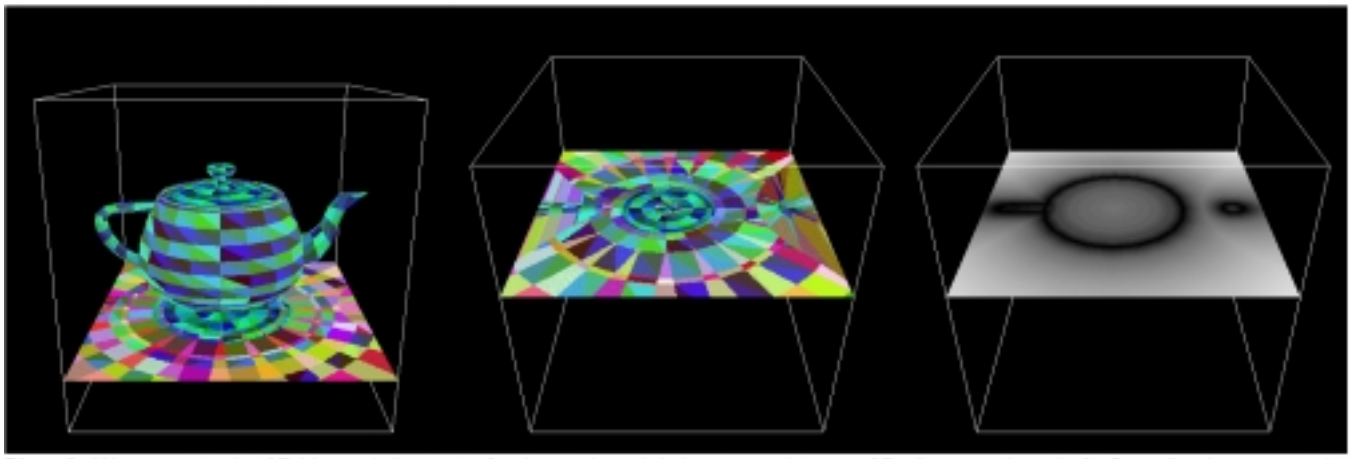


Plate 3: We compute the 3D Voronoi diagram of polygonal models by computing one 2D slice at a time. Left: One slice is computed just below the teapot. Center: Complete slice halfway through. Right: Distance image. The colors indicate the Voronoi regions for face edge, and vertex sites. Only the face site colors are shown on the model.

Editors: Theresa-Marie Rhyne and Lloyd Treinish

Harnessing Natural Textures for Multivariate Visualization

Victoria Interrante

University of Minnesota

In our ongoing quest to convey more information more clearly in a single image, harnessing the full potential of texture for data representation remains an elusive goal. Others have begun excellent work in this area,[1-3] and my efforts are inspired by their example. The grail that I seek is a partially ordered multidimensional palette of richly detailed and varying texture patterns that can be used—in conjunction with lightness and hue—to represent multivariate information. The goal is to facilitate the flexible visual appreciation of the correlations of various quantities across the different dimensions. The approach that I outline here departs a bit from the norm, but is motivated by a desire to proceed more directly from my vision of what I want to achieve, unrestrained by the limitations of the tools I have on hand. In the following discussion, I motivate the adoption of rich, natural textures—resembling those from photographic images[4]—as elemental primitives and sketch some of the approaches that we can take to enhance our understanding of how to effectively harness their properties. My intent here is not to present results, but to expound on the issues and conclude with the questions to which we're still seeking answers.

Why natural textures?

The intricate variety and subtle richness of detail of texture patterns found in nature support possibilities for data representation far more vast and comprehensive than we could ordinarily hope to achieve from standard primitives. Even if we must ultimately rely on synthesized textures for data visualization, by looking to nature for inspiration we have the potential to expand our vision of what to strive for in such a synthesis. The graphic design community has long held that perfectly regular synthetic textures on a flat plane, in particular the infamous hatching patterns that Edward Tufte refers to as "chart junk,"[5] are discomforting to the eye and annoying to look at. Natural textures are not only more aesthetic, but they also put less extraneous stress on the visual system, leaving our eyes freer to observe and attend to the most intrinsically important texture-pattern characteristics.

Understanding human texture perception

To create a perceptually meaningful multidimensional *texture space* that can be indexed in the same fashion as a color space, we must begin by knowing what we're looking for. We need to proceed from a rigorous and experimentally supported understanding of how human observers perceive and interpret texture patterns, under the conditions in which we intend for these patterns to ultimately be viewed. This grounding provides a structure for guiding our search through the complex space of possibilities and formalizes the intuition that a good designer calls upon to create a visualization that works.

A number of researchers[6] have conducted studies to try to elucidate the most significant perceptual dimensions of texture. The results of these experiments will aid us, though some important questions remain. It's beyond the scope of this article to summarize previous findings further than to say that most of the studies used unaltered images from the Brodatz album, subjects were generally asked to cluster the textures into groups, and there appears to be general agreement that a small number (about three) of characteristic dimensions seem sufficient to describe most of the structure underlying this classification. The interpretation of the dimensions varies from study to study, but most often includes aspects of the following:

- periodic (consisting of repeated discrete elements) <-> nonperiodic
- strongly directional <-> rotationally invariant
- coarse <-> fine (spatial frequency of the dominant detail)
- regular (deterministic) <-> random
- high contrast <-> low contrast
- homogeneous (spatially invariant) <-> heterogeneous

Clearly, there is some overlap in these categorizations. Also, it's not evident that we can hope to determine an orthogonal basis that encompasses all members of the texture pattern set. However, the apparent low perceptual dimensionality of the space and the strong agreement between the studies bodes well for our application.

Open questions in visual texture perception

Identifying the features according to which people tend to classify texture patterns gives us important insight into how to structure a perceptually meaningful texture space.

0272-1716/00/$10.00 © 2000 IEEE

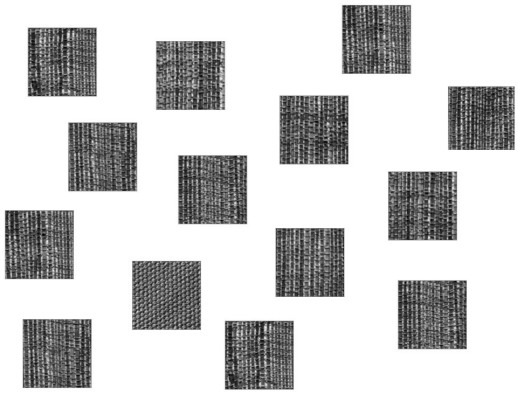
But some important issues have been implicitly overlooked in the studies conducted so far. Foremost are the uncontrolled-for influences of higher level processes.

Payne et al.[7] observed, in studies similar to those referenced above, that material property categories appeared to have a strong influence on a fair number of their observers' clustering decisions. They also noted that their subjects often commented that they felt they were using different criteria to find matches for different texture images. Knowing that people tend to make judgments based on different criteria in different cases, but not knowing who is considering what when or why, weakens the general applicability of experimental results based on these traditional methodologies. I believe that it might be useful to attempt to control for, or at least quantify the impact of, some of these effects by considering alternative paradigms for objectively measuring texture patterns' perceptual similarity.

The question of whether to control for rotation, scale, luminance, and contrast variance among texture samples when seeking insight into the perceptual groupings of texture images is a second important issue that studies using the Brodatz album have frequently overlooked. How do we want to consider apparent texture pattern differences that aren't clearly intrinsic to the pictured material but that could conceivably be attributed to external factors such as viewpoint or lighting?[8]

On the one hand, Ware and Knight[3] have said that orientation, size, and contrast are the primary orderable dimensions of texture. But samples of a texture that are differently oriented, scaled, and lit still intrinsically appear to be the same thing and thus remain good candidates for similarity grouping. As pre-attentive features of individual elements, size, contrast, and orientation differences are undisputedly important in facilitating "pop-out". At the same time, our visual system is remarkably adept at maintaining perceptual constancy across changes in illumination or viewpoint. Many computational methods for classifying texture adopt a rotationally invariant texture recognition approach for this reason. A strong argument exists for equalizing characteristics such as scale, luminance, and contrast before classifying or quantifying texture-pattern differences for the purposes of visualizing multiple distributions across a 2D image. Doing so would let us retain the ability to introduce variations in these feature dimensions universally across all the other texture dimensions and use them to encode additional values. To this end, it might be illuminating to try to separately examine the relative effects of rotation, scale, and contrast differences versus other texture-characterizing differences. Our considerations will also differ in the case of visualizing distributions across surfaces through a 3D domain.

The question of how best to factor out the variations in contrast and luminance—when we choose to do so—must also be carefully considered. Although it's relatively straightforward to equalize the intensity histograms of sample texture images before processing them for similarity, it's not clear that histogram equalization adequately preserves the meaningful qualities of the texture patterns.



1 The salience of differences in various properties of texture patterns might be estimated through psychophysical experiments using briefly presented stimuli such as these.

Quantifying the perceptual similarity of texture patterns

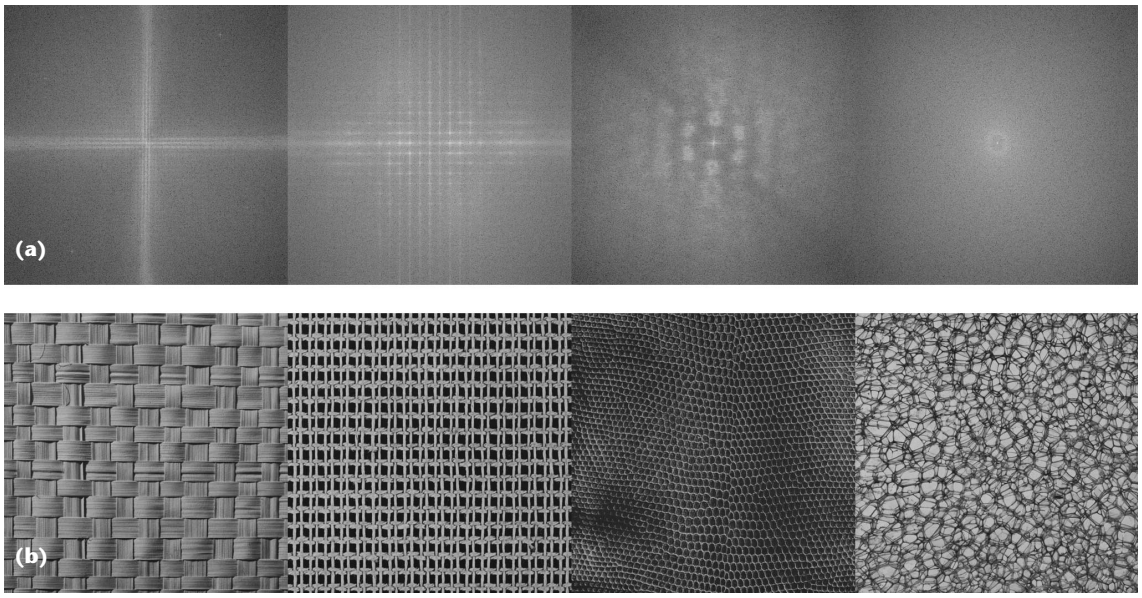
In addition to determining which textures tend to cluster, it's important for creating a perceptually linear texture space to quantify the perceptual distances between individual texture patterns. It's also necessary to estimate the magnitude of the perceived distance due to the differences along each of the feature dimensions.

One possible approach is to estimate the magnitude of the change required to enable a "just noticeable difference" between images along individually selected texture dimensions such as scale, contrast, orientation, regularity, and so on using psychophysical methods.

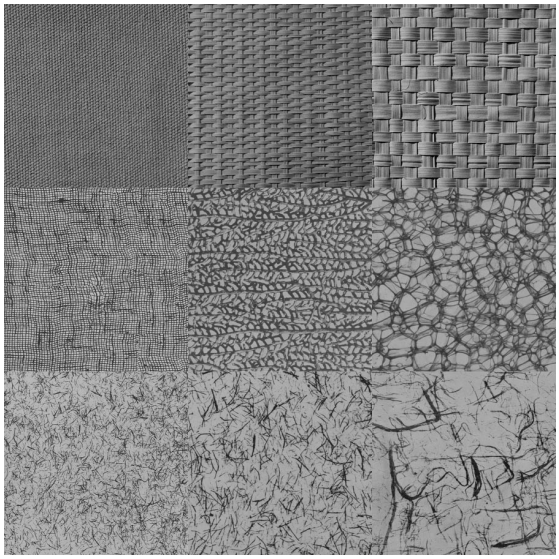
Another possibility, which may be more appropriate for judging the kinds of differences that cannot be easily brought down to threshold levels, is to measure the pre-attentive discriminability or salience of differences in features of individual texture patterns randomly embedded in homogeneous and heterogeneous fields of distracters, as illustrated in Figure 1. The objective in this case then is to determine how large of a difference is required to allow the effortless identification of the "odd man out" in brief, masked stimulus presentations. Studies using individual element arrays have found that salience (or the tendency to "pop-out") tends to increase when the targets are characterized by redundant, unique properties such as luminance and hue or color and orientation.[9] Similarly, the salience of the target tends to decrease as the heterogeneity of the distracter elements increases, even when the heterogeneity occurs along a different perceptual dimension.

It may additionally be of interest to determine how many different texture types people can simultaneously discriminate, using a methodology similar to what Healey employed for studying color.[10]

The long and rich history of research on texture classification algorithms in the image processing community also offers valuable resources for constructing a texture palette. Although the extent and variety of the possible computational approaches for classifying texture patterns is somewhat overwhelming, most of the successful methods work by extracting a finite set of features from the texture patterns (via transforms similar to those shown in Figure 2) and then calculating various statistics across these feature sets. Rubner and



2 (a) Statistics of texture pattern features, such as those evident in this set of power spectra images, form the basis for many computational texture characterization methods. (b) It's easy when looking at the original images to form an intuitive understanding of the types of information carried by the various features of the Fourier coefficients.



3 A small potential texture palette. Scale increases along the horizontal axis, regularity increases along the vertical axis, and intensity increases along the left-to-right descending diagonal.

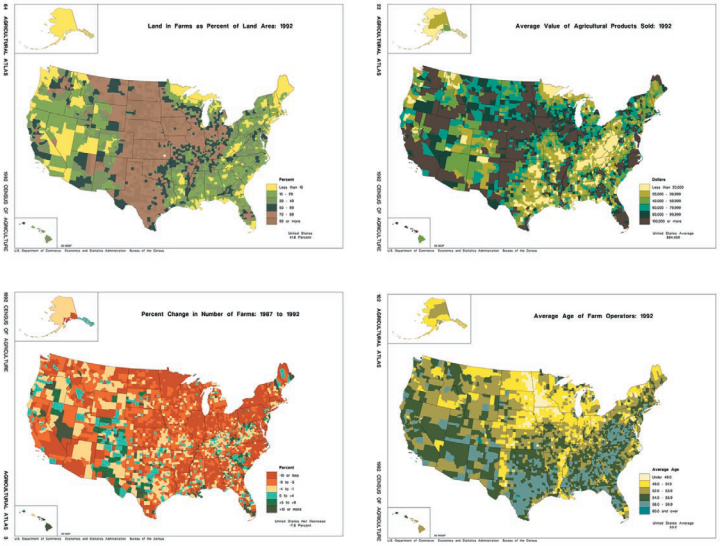
Tomasi[8] proposed quantifying texture similarity by using the Earth Mover's Distance as a metric for the goodness of fit between histograms of these features. A key issue for us is ensuring that the computer vision results and the human observer criteria agree.

Creating a texture palette

One approach to creating a texture palette (Figure 3) is to begin with a collection of well-chosen input images. You can then objectively determine where they lie in the best-fitting multidimensional texture feature space that they span and appropriately fill in the remaining open space with intermediate textures that lie at equal perceptual distances along each of the dimensions.

Some of the difficulties in creating a texture palette are that the texture space may not be orthogonal, there may be interaction among certain dimensions (such as contrast and spatial frequency), and some texture type mixtures may not be meaningful. However, my intuition is that the closer we can get to aesthetically filling out the space with real acquired images, the easier it will be to patch the holes.

Texture synthesis methods such as those that Portilla and Simoncelli[11] and Zhu et al.[12] have proposed may hold the greatest potential for creating a set of sample textures that fill out a multidimensional palette. Working from these frameworks, we may have the possibility to deterministically create natural-looking intermediate textures that interpolate the characteristic properties of their neighboring swatches in texture space or that



4 A set of four agricultural maps,[14] representing distributions of four different variables across the US counties.

conform to a prespecified set of desired criteria. However, these methods currently invoke too many features (roughly 800 plus) and are not quite mature enough to guarantee the realism that we seek.

Efros and Leung[13] suggest a very different synthesis method for seamlessly generating highly realistic samples of "more of the same" textures from a given sample. The success of this method is subject to the assumption of stationarity and a reasonable estimate of the extent of the lowest spatial frequency detail that must be preserved. However the feasibility of extending this approach to texture interpolation has yet to be shown.

Another issue we face is how to properly deal with the lighting consistency problems that will inevitably arise when we try to combine acquired textures that exhibit relief. It may be necessary to solve for the surface relief, allowing the material intensity texture to be handled separately. In the 2D case, the most important consideration is simply to maintain consistency. Additionally, it's desirable to avoid orienting textures so that to preserve convexity, observers must envision the light as coming from below.

Feasibility issues

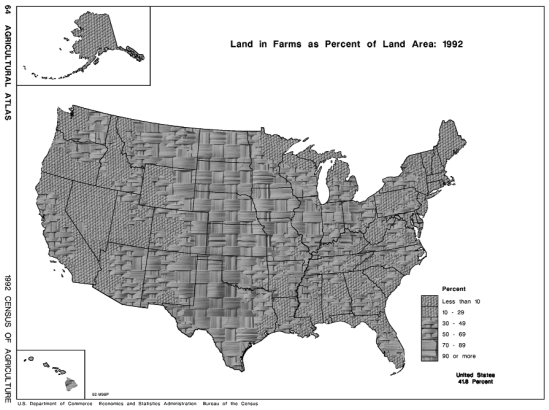
Some of the questions that we need to address are

- What does a reasonable partitioning of a natural texture space look like?
- Would it be feasible to try to choose exemplars at the endpoints of each perceptually relevant texture dimension, characterize them statistically, then interpolate to obtain intermediate textures that fill out the space?
- To what extent do we need to guarantee that different textures will meld continuously into each other at the transitions between level set regions?
- How can we most effectively combine color with texture to convey yet more information in a meaningful way?

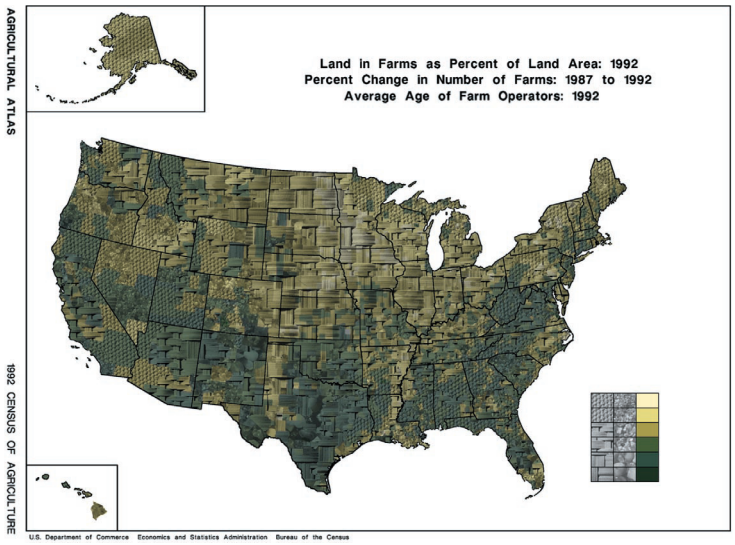
Clearly, texture has the greatest potential to be effective as a tool for visual data representation when it conveys local values of an underlying function across homogeneous cells of sufficient size to allow the characteristic detail of the resident texture to be discriminated.

The philosophy behind my attempt to more effectively harness the potential of texture for multivariate data visualization is that we should begin from a vision of what we want to achieve and work from there to figure out how to accomplish the desired results. To more freely explore the possibilities before becoming bogged down in implementation issues, I mocked up several tests by hand using Adobe Photoshop and scanned images from the Brodatz texture album[4] to represent multivariate agricultural data[14] at the county level (Figures 4 through 7).

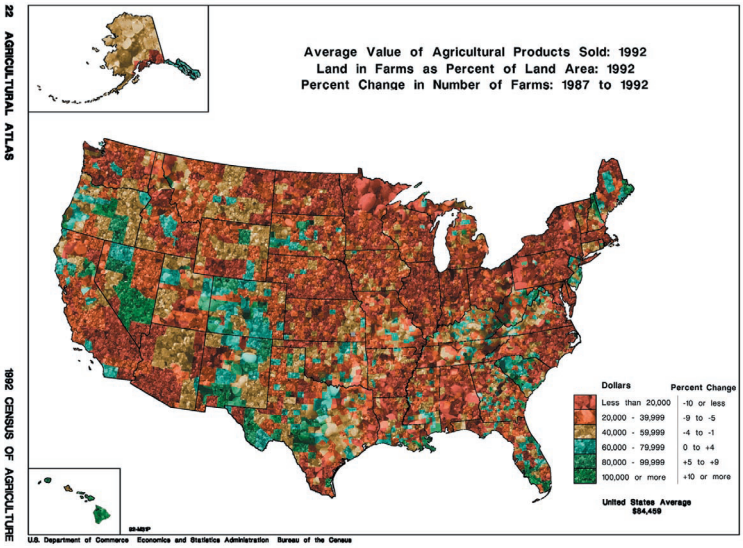
In Figure 7, texture scale represents one of three



5 A simple hand-crafted demonstration of a display in which three natural textures of increasing scale convey three discrete ranges in the value of a variable along a single



6 A hand-crafted example in which one dimension of color and two dimensions of texture overlap to represent three different variables in each cell.



7 A second hand-crafted example in which one dimension of color and two dimensions of texture represent three different variables in each cell.

ranges of the average value of the agricultural products produced in each county (with the largest scale corresponding to the lowest value, evoking the metaphor of inhospitable terrain). The texture type represents the direction of change in the amount of land in farms (with rocks representing areas with an overall loss of farmland and weaves representing areas with an overall increase in the amount of land used for farming). Color labels the percentage of the total land area in each county used for farming (with the green tones indicating the higher percentages and the brown tones representing the lower percentages of land occupied by farms).

Representing uncertainty

Both color and texture admit intriguing possibilities for representing uncertainty in data measurements. Texture regularity has particularly good potential as an intuitive marker for certainty, with texture pattern irregularities increasing in prominence where measurement reliability is lower. Color regularity has somewhat weaker potential as a marker for measurement certainty, as textures that contain more balanced or restricted distributions of hues may appear less distinct or vibrant (implying decreased noteworthiness) than textures in which the hues vary more widely across the spectrum. Figures 8 and 9 provide examples of these two different configuration series. It remains to be seen how easily irregularity of the kind shown in Figure 9 can be incorporated into a computational texture synthesis definition for such patterns.

Conclusions

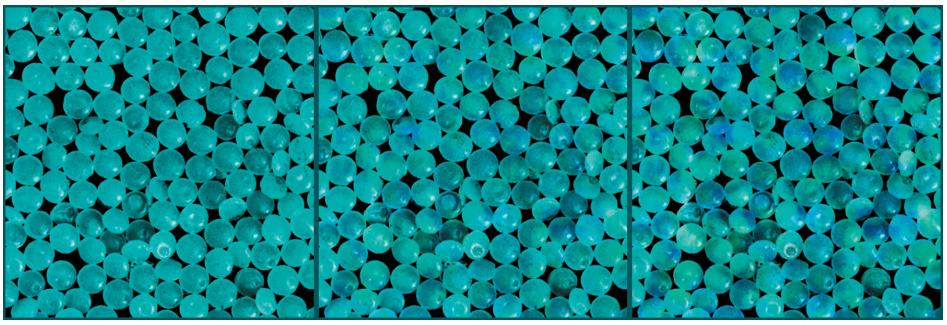
Despite the excellent progress made in recent years,[1-3] I believe that there remains great untapped potential for the effective use of texture in multivariate visualization. I have proposed that we might take important steps towards realizing more of this potential by attempting to harness the power of rich natural textures. I envision that a successful approach will begin from a fundamental understanding of visual texture perception and progress toward an understanding of how to synthesize a multidimensional palette of detailed texture samples whose variations evoke an intrinsic appreciation of the local and global relationships between multiple quantities across a 2D domain. ■

Acknowledgments

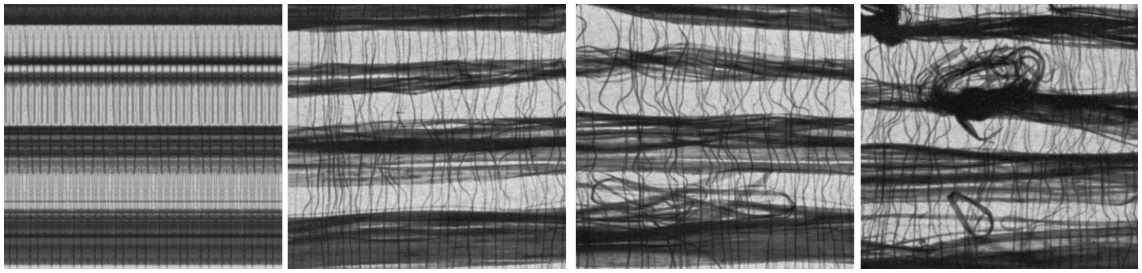
In framing this work, I have benefited from informal discussions with a number of different people. I am particularly grateful to Tomás Filsinger for providing valuable criticism and insights into understanding effective uses of texture from an artist/graphic designer's perspective, and to Guillermo Sapiro and Dan Kersten for pointers into the computer vision and texture synthesis literature. My fledgling efforts are partially supported by a 1999-2000 University of Minnesota Grant-in-Aid of Research, Artistry, and Scholarship and an NSF Presidential Early Career Award for Scientists and Engineers (Pecase) award.

References

1. C.G. Healey and J.T. Enns, "Large Datasets at a Glance: Combining Textures and Colors in Scientific Visualization," *IEEE Trans. Visualization and Computer Graphics*, Vol. 5, No. 2, 1999, pp. 145-167.
2. D.H. Laidlaw et al., "Visualizing Diffusion Tensor Images of the Mouse Spinal Cord," *Proc. IEEE Visualization 98*, ACM Press, New York, 1998, pp. 127-134.
3. C. Ware and W. Knight, "Using Visual Texture for Information Display," *ACM Trans. Graphics*, Vol. 14, No. 1, 1995, pp. 3-20.
4. P. Brodatz, *Textures: A Photographic Album for Artists and Designers*, Dover Publications, New York, 1966.
5. E.R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Conn., 1983, p. 111.
6. A.R. Rao and G.L. Lohse, "Identifying High Level Features of Texture Perception," *CVGIP: Graphical Models and Image Processing*, Vol. 55, No. 3, 1993, pp. 218-233.



8 Variability in hue might be used to convey the range of imprecision or uncertainty in the data value recorded for a region.



9 Increasing irregularity in a texture pattern can intuitively represent increasing amounts of uncertainty.

7. J.S. Payne, L. Hepplewhite, and T.J. Stonham, *Texture Similarity: Using Human Studies to Interpret Retrieval Results*, Tech. Report TR-685, Dept. Electronic and Computer Engineering, Brunel University, UK, 1997.
8. Y. Rubner and C. Tomasi, "Texture Metrics," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics*, IEEE Press, Piscataway, N.J., 1998, pp. 4601-4607.
9. H-C. Nothdurft "Salience from Feature Contrast: Additivity Across Dimensions," *Vision Research*, Vol. 40, 2000, pp. 1183-1201.
10. C.G. Healey, "Choosing Effective Colors for Data Visualization," *Proc. IEEE Visualization 96*, ACM Press, New York, 1996, pp. 263-270.
11. J. Portilla and E.P. Simoncelli, "A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients," *Int'l J. of Computer Vision*, 2000, to appear; currently available on the Web at http://www.cns.nyu.edu/ftp/eero/portilla99.pdf.
12. S.-C. Zhu et al., "Exploring Texture Ensembles by Efficient Markov Chain Monte Carlo—Toward a 'Trichromacy' Theory of Texture," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 22, No. 6, 2000, pp. 554-569.
13. A.A. Efros and T.K. Leung, "Texture Synthesis by Nonparametric Sampling," *IEEE Int'l Conf. Computer Vision, Volume II*, IEEE Press, Piscataway, N.J., 1999, pp. 1033-1038.
14. *1992 Agricultural Atlas of the United States*, U.S. Census Bureau Geography Div. Map Gallery, http://www.census.gov/geo/www/mapGallery.

Readers may contact Interrante at the Dept. of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Bldg., 200 Union St. SE, Minneapolis, MN 55455, e-mail interrante@cs.umn.edu.

Readers may contact department editors Rhyne and Treinish by e-mail at rhyne.theresa@epamail.epa.gov and lloydt@us.ibm.com

Correction

In the September/October issue, the Visualization Viewpoints article "Visualizing Visualizations: User Interfaces for Managing and Exploring Scientific Visualization Data" by Kwan-Liu Ma incorrectly implied that Fritz Hasler (NASA Goddard) alone worked on the Distributed Image Spreadsheet (p. 17). In fact, the work was done by both Hasler and Kannappan Palaniappan (University of Missouri-Columbia).

your **Vision** *our* **Future**

Southwest Research Institute™

has exciting opportunities for

Computer Scientists & Engineers

- **Modeling and Simulation**
- **Software Engineering & Design**

Positions require a BS or MS in Computer Science, Computer Engineering, Electrical Engineering, or equivalent; programming experience in C, C++, and Java on PC- or UNIX-based workstations.

Responsibilities include software prototype development, design, testing, implementation, integration, maintenance, program mgt., and traveling to client sites. **Opportunities available in San Antonio, TX; Huntsville, AL; Warner Robins, GA; and Oklahoma City, OK.**

Please visit:

www.swri.org/jobs/

An Equal Opportunity/Affirmative Action Employer
A Public and Community Services Employer

- *Competitive salaries*
- *Medical, dental, & vision care*
- *Retirement & benefits*
- *Educational advancement*
- *Professional development*
- *Awards for individual research*
- *Paid relocation*
- *Employee assistance programs*
- *On-site fitness facility*
- *Generous vacation*
- *Ten paid holidays*

SwRI™

Southwest Research Institute

Creating Evenly-Spaced Streamlines of Arbitrary Density*

Bruno Jobard and Wilfrid Lefer

Laboratoire d'Informatique du Littoral, Calais, France
{jobard,lefer}@lil.univ-littoral.fr

Abstract. This paper presents a new evenly-spaced streamlines placement algorithm to visualize 2D steady flows. The main technical contribution of this work is to propose a single method to compute a wide variety of flow field images, ranging from texture-like to hand-drawing styles. Indeed the control of the density of the field is very easy since the user only needs to set the separating distance between adjacent streamlines, which is related to the overall density of the image. We show that our method produces images of a quality at least as good as other methods but that it is computationally less expensive and offers a better control on the rendering process.

Introduction

The problem of visualizing vector fields has been widely addressed in the past years because it has numerous applications. The main issue is to visualize properly the direction and magnitude of the flow. Spatial resolution techniques such as arrow plots, streamlines or particles traces suffer from their spatial resolution that limits drastically their usefulness, in particular in the presence of a turbulent flow. Moreover the effectiveness of the streamline and particle methods depends critically on the placement of the forming seed points. Texture-like methods, such as Spot Noise [10] and LIC [1] produce dense field images showing the flow features in fine-grain detail. Another issue is to compute sparse flow fields, laying stress on the visual appearance of the field, which produces hand-drawing style images. Recently, Turk and Banks presented an imaged-guided streamline placement to compute hand-drawing style representations of a flow field [9]. This paper presents another effective algorithm for the placement of evenly-spaced streamlines. The main technical contribution of this work is to propose a single method to compute a wide variety of flow field images, ranging from texture-like to hand-drawing styles. Indeed the control of the density of the field is very easy since the user only needs to set the separating distance between adjacent streamlines, which is related to the overall density of the image.

* This work was supported by the council of the Nord-Pas-de-Calais region.

1 Related work

Visualizing a vector field in a general manner requires high spatial resolution techniques to properly render fine-grain details. Such methods generally yield dense field representations. However there are situations where a sparse density image is needed, by instance to produce an illustration similar to those used to enhance the purpose of text fields in a book. Methods proposed to visualize a flow field falls into these two categories: dense field representations and hand-drawing style.

The first method for representing a flow field with high spatial resolution has been proposed by van Wijk [10]. The *spot noise* method creates a directional texture by superimposing many flow-oriented ellipses. Each ellipse is generated by projecting a spherical spot onto a surface and by advecting the spot with the direction and magnitude of the vector field at the projection point. This amounts to the flow field-controlled generation of a band-limited noise. Initially straight, the spots are now bent along short streamlines to follow the curvature of the vector field [2]. An important feature of this method is the local control on the generated image. More spots gives rise to more accuracy. With spot noise the generation time depends on the number of spots used to generate the texture. Consequently, by setting the number of spots a trade-off can be obtained between image quality and rendering time.

Another interesting method is the *line integral convolution* (LIC) proposed by Cabral and Leedom [1] [4] [8]. A LIC texture is generated by convoluting an input texture with a streamline-oriented one dimensional filter kernel. The images obtained with this technique are very effectives, showing more details than the previously enumerated one. But this is obtained at the expense of computation. This computation cost is mainly due to the number of streamlines that have to be computed, and let us notice that even with the *fastLIC* method [8], several streamlines cover each single pixel of the resulting texture, giving rise to frequent recomputations.

The image-guided streamline placement method proposed in [9] uses a stochastic mechanism to iterately refine the placement of the streamlines. First an initial set of randomly placed streamlines is created. Then this set of streamlines is updated using three valid operations: (1) changing the position and/or length of a streamline, (2) joining streamlines that nearly abut, and (3) creating a new streamline to fill a gap. At each step of the refinement process a small change, i.e. a combination of the three operations mentioned above, is randomly performed. An energy function consults a low-pass-filtered (blurred) version of the image to measure the variation of energy between the current and the updated images and the modification is only accepted if the variation of energy if negative. This method produces high quality images but the convergence is very slow and obtaining a good visual appearance often requires several minutes for each image to be computed. Moreover this method is not suitable for dense field images because of the combinatorial explosion of the possible modifications at each step.

The remainder of this paper is organized as follows. In section 2 we present a method for effective user-controlled evenly-spaced streamlines placement. Section 3 describes the use of the method to produce hand-drawing style images and we compare our approach with the image-guided streamline placement from Turk and Banks. In section 4 we show how texture-like images can be obtained and we discuss the advantages and drawbacks of this method compared to LIC. We conclude and offer directions for future research in section 5.

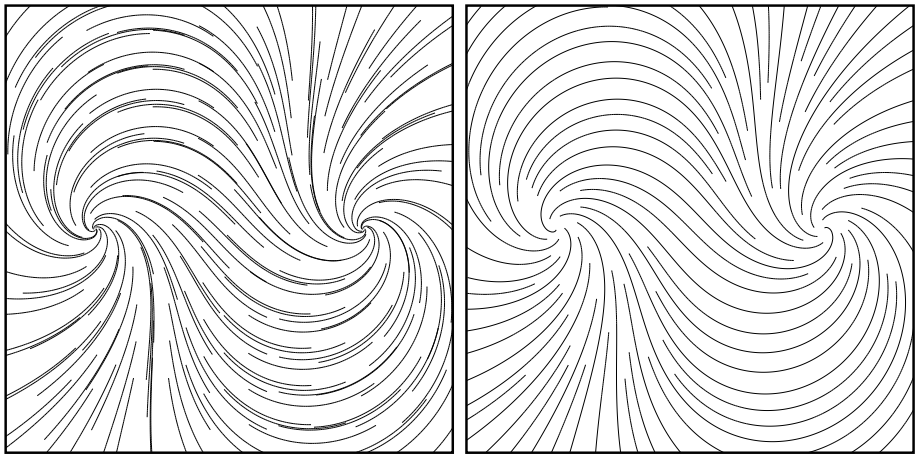


Fig. 1. (a) Long streamlines with seed points placed on a regular grid (left); (b) Same flow field computed using our streamline placement method (right)

2 Algorithm overview

The goal of this work is to produce long and evenly spaced streamlines in a single pass. The basic principle of our algorithm is similar to a method presented by Max et al. in [6]. The goal of their work was to cover a 3D surface (not necessary tangential to the field) with a set of streamlines. Once a seed point has been selected in the field, they make a streamline growing beyond that point backward and forward. The growing process is stopped when the streamline reaches an edge of the surface, a singularity in the field (source or sink) or becomes too close to another streamline. The streamline is then divided into a set of small segments of contrasting color and projected onto the surface. Although Max's method was intended to visualize a flow on a 3D surface, it can be generalized to all kinds of steady 2D field.

We extend this work in the following manner. First we give a number of precisions concerning the implementation of the algorithm together with a couple of optimizations. Second we show how the algorithm can be controlled by the user to produce a wide range of flow fields images, ranging from hand-drawing

to LIC-like style.

An important feature of the algorithm is that it processes in a single pass (as compared to Turk's progressive refinement approach). To compute an image, a number of streamlines are calculated until a user-fixed density level has been obtained. Computing a new streamline is achieved in the following manner. A new seed point is chosen at a minimal distance apart from all existing streamlines. Then a new streamline is integrated beyond the seed point backward and forward until either it goes too close from another streamlines or it leaves the 2D domain over which the computation takes place. The algorithm ends when no more valid seed point can be found. Figure 1b shows an image obtained with our algorithm and figure 1a shows the same flow field visualized using a distribution of the seed points over a regular grid. The three following sections detail three important points of the algorithm: the control of the distance between adjacent streamlines, seed points selection and streamline integration.

2.1 Control of the separating distance

Density is a global feature of the field. However we need to express it as a local feature in order to have a local control on the texture generation. We express the density as the distance between two adjacent streamlines. Let d_{sep} be this distance. Hence the control of the density of the field is achieved by controlling that there is not any pair of streamlines apart from a distance lower than d_{sep}. This control occurs during the construction of each streamline. During the construction, a new sample point is valid only if it is at a separating distance greater than d_{sep}. If it is not the case, the streamline is stopped in this direction (during construction streamlines grow in both directions independently). In order to make the computation of the separating distance faster, rather than computing the exact distance from the new seed point location to the streamline, we compute the distance from the seed point to the sample points along the streamline. To make this approximation acceptable, the sample points on a streamline must be evenly spaced and the distance between them must be smaller than d_{sep}. Thus, a new sample point is valid if the distance between it and all the existing sample points is greater than the separating distance. Since this test has to be computed for all the sample points, it must be as fast as possible. To accelerate the computation of the distance, we use a cartesian grid superposed to the vector field domain, the width and height of a cell being exactly d_{sep}. Each cell contains a list of pointers to the sample points located within the cell. Thus, given a new seed point location, the cell containing the location is easily determined. Let us call this cell the local cell. The distance has to be computed only for the sample points located either within the local cell or within the eight cells surrounding the local cell. In practice, we have noticed that an average of 5-7 distance computations is necessary to determine if a new sample point is valid or not. The denser is the grid, the less comparisons are required.

remark: Practically, we consider two important distances, d_{sep} and d_{test}. d_{sep} is the separating distance given by the user. It represents the minimal distance between seed points and streamlines. d_{test} is a percentage of d_{sep}. It corresponds to the minimal distance under which the integration of the streamline will be stopped in the current direction. We found $d_{test} = 0.5 \times d_{sep}$ gives good visual result by producing long streamlines. For instance, images of Figure 2 have been calculated with two different values of d_{test}.

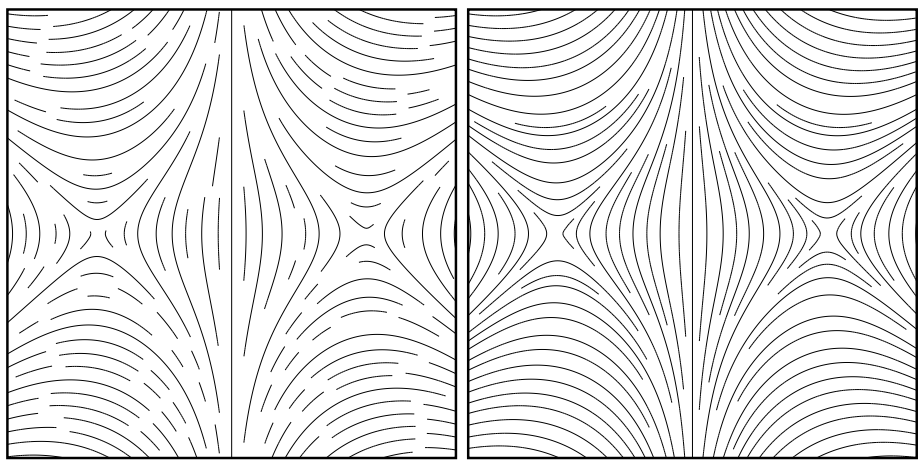


Fig. 2. increasing difference between d_{sep} and d_{test} lengthen streamlines; (left) $d_{1_{test}} = 0.9 \times d_{sep}$; (right) $d_{2_{test}} = 0.5 \times d_{sep}$

2.2 Seed points selection

When using streamlines for vector field visualization, a common problem is to select proper seed points for path tracking. Dovey proposed two approaches to resample non-uniformly spaced grids in order to achieve an uniform density of vector glyphs [3]. A vector field is represented with short segments oriented by the flow. In case of short streamlines or hedgehogs, the resulting image mainly depends on the distribution and density of the seed points over the domain. In case of long streamlines a constant density of seed points do not ensure a good distribution of the streamlines.

In order to obtain a good visual appearance of the flow field, an accurate seed point selection has to be perform. The principle of our algorithm is to derive all the seed points possible to find from an existing streamline before trying with another existing one. The proposed seed points are chosen at a distance $d = d_{sep}$ from the sample points of each streamline. Our algorithm uses a queue to store the newly created streamlines which are processed from the older one to the more recently created one. The algorithm is given below.

```
Compute an initial streamline and put it into the queue
Let this initial streamline be the current streamline
Finished := False
Repeat
    Repeat
        Select a candidate seedpoint at d = d_sep apart from the current streamline
    Until the candidate is valid or there is no more available candidate
    If a valid candidate has been selected Then
       Compute a new streamline and put it into the queue
    Else
        If there is no more available streamline in the queue Then
           Finished := True
        Else
            Let the next streamline in the queue be the current streamline
        EndIf
    EndIf
Until Finished=True
```

Figure 3 shows all the streamlines the algorithm has been able to derive from the first streamline for a given vector field.

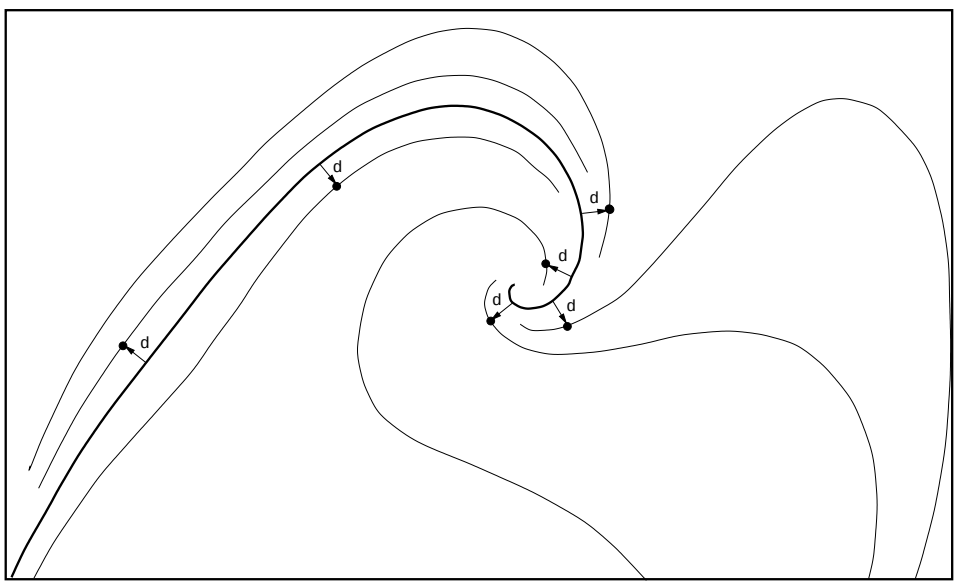


Fig. 3. streamlines are derived from the first (thick) one by choosing seed points (circles) at a distance d = d_{sep} from it

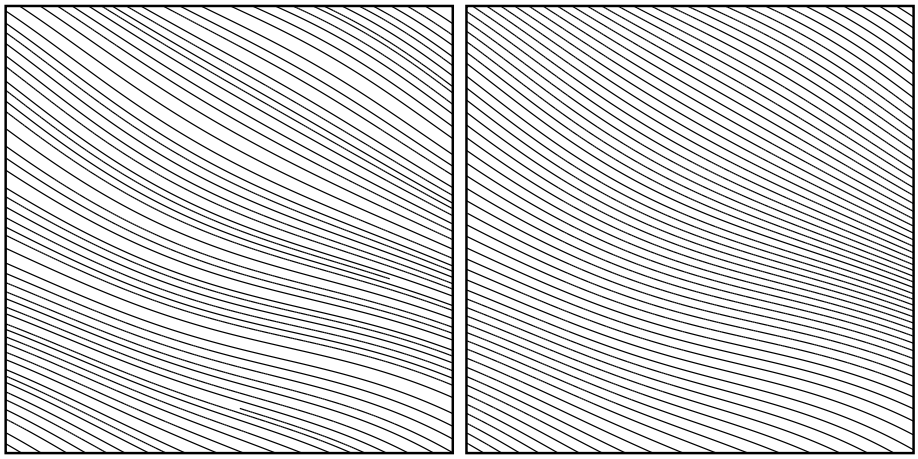


Fig. 4. Two seed points selection method with the same density of streamlines: (left) random selection, (right) our selection method

For sparse illustrations choosing to start streamlines close to existing ones gives better visual results than selecting seed points in a random fashion (see figure 4) but is more time-consuming. For dense texture generation, the quality of images produced with various seed point selection methods is quite similar.

2.3 Streamline integration

In order to measure a consistent separating distance between a point and a streamline, sample points along a streamline must be evenly spaced (see section 2.1). Many integrators are able to produce such evenly spaced sequences of sample points. They can be classified into three categories:

- fixed step size integrators such as Euler, Midpoint or Runge-Kutta methods with normalized vector fields,
- non constant or adaptive step size integrators with a post interpolation phase such as cubic Hermite-interpolation, which deal with large distances between neighboring sample points and curvature of the streamlines [8],
- continuous integration methods such as DOPRI5, which is a fifth order Runge-Kutta integrator with adaptive step size monitoring and fourth order error estimation and produces a dense output directly by using informations gathered at each step of the integration [5].

At present we use the Midpoint integrator but future investigations will concern the choice of a better integration method. In particular using an adaptive step size integrator will decrease the number of integrations required, reducing overall computation time.

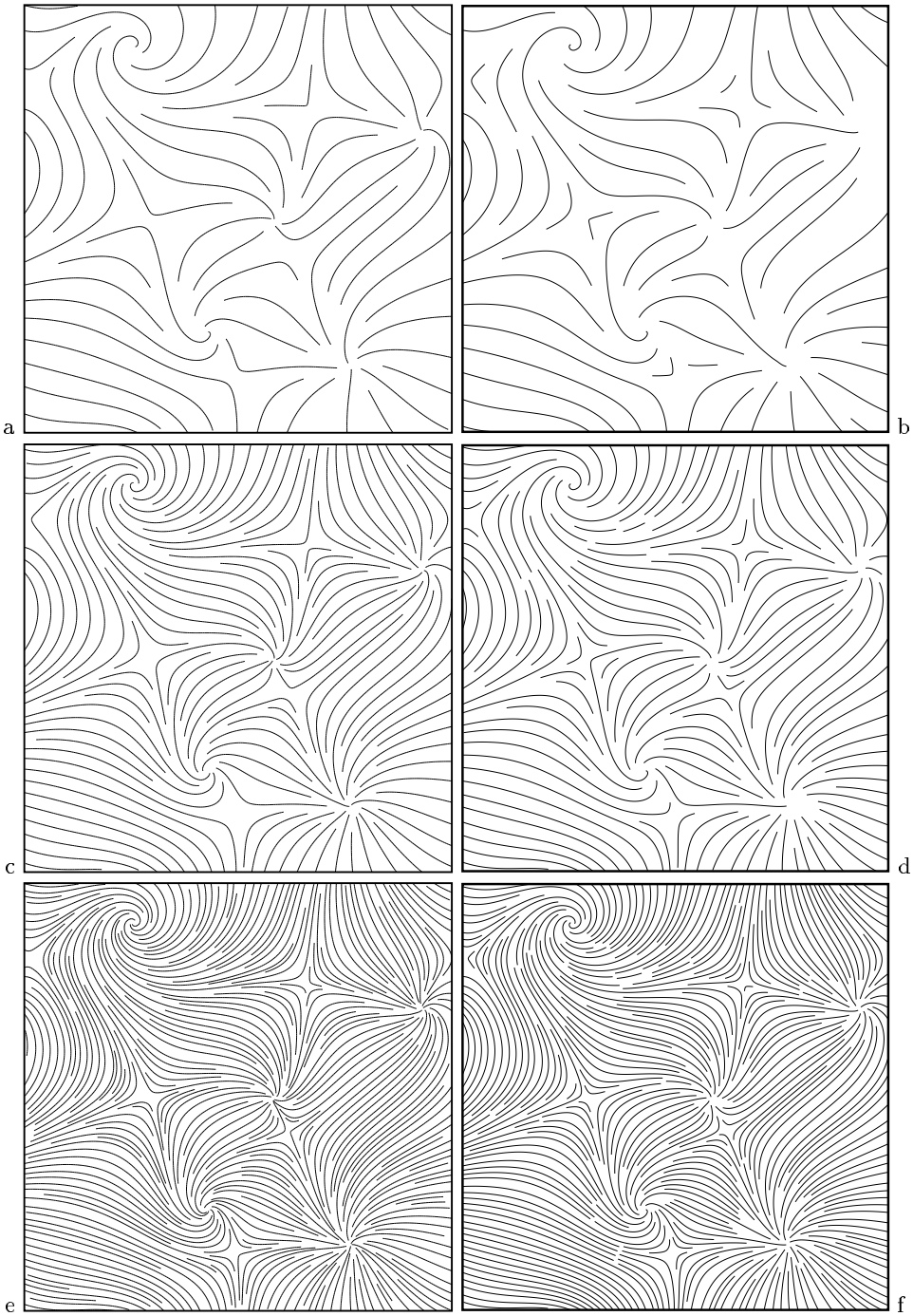


Fig. 5. Image comparisons for separating distances of 6%, 3% and 1.5% of image width; left column: Image-guided placement; right column: Our streamline placement.

3 Hand-drawing style

Sparse illustrations of flows fields are the more interesting application of our method. Turk et al. proposed a method to compute such a representation in [9]. The method computes an initial set of streamlines which is then iterately refined until the global energy of the image falls below a fixed threshold or the user stops the process. The images obtained with this method are of great quality but the convergence of the iterative process is very slow and becomes much slower when desired density increases. Moreover the energy function used to measure the quality of the result is not directly related to the visual appearance of the image, requiring the appreciation of the user to stop the process.
The advantage of our method as compared to Turk's one is that it processes in a single pass, computing the final image directly. Figure 5 shows the same flow field computed by Turk's method with various refinement degrees and by our method and table 1 gives computation times necessary to produce the different images. We see that our method produces images of the same quality as Turk's ones but is less time-consuming.

separating distance	Image-guided placement	Our placement algorithm
6% image width	Fig 5a: stopped at 2 mn	Fig 5b: 4 seconds
3% image width	Fig 5c: stopped at 4 mn	Fig 5d: 9 seconds
1.5% image width	Fig 5e: stopped at 10 mn	Fig 5f: 17 seconds

Table 1. Computation times on a MIPS R4600 Processor, 132Mhz with 32Mo. Image-guided placement images and computation times were obtained with the Greg Turk's original publicly available algorithm.

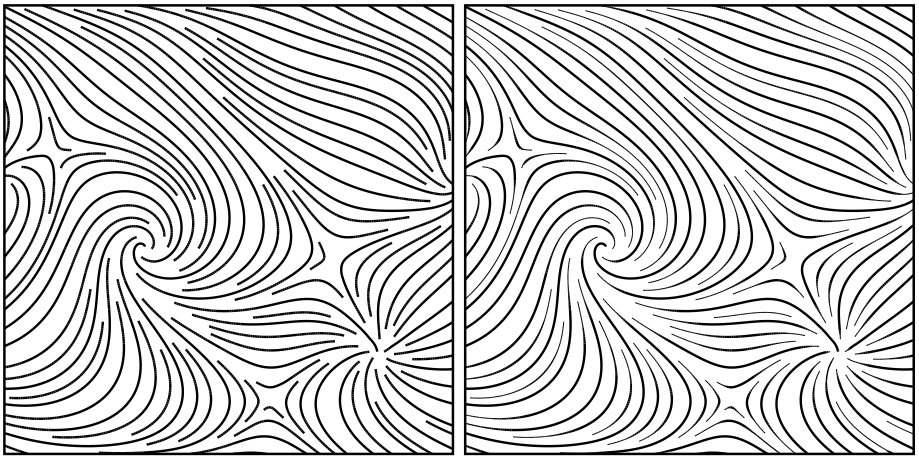


Fig. 6. Hand-drawing style images computed without and with the tapering effect

Tapering effect. As stated in section 2.1 the actual distance between streamlines is not constant. Since the density is related to the distance between streamlines, disparities of density may appear in the resulting image. To reduce this visual artifact, Turk suggested to *taper* the ends of the streamlines by decreasing the thickness of the lines as they go closer to another one. In case of Turk's algorithm, this is achieved in a post-processing step. Our implementation allows to directly include the tapering optimization during the growing process of each streamline. The width of the streamline is computed using the following formula:

$$thicknessCoef = \begin{cases} 1.0 & \forall d \geq d_{sep} \\ \dfrac{d - d_{test}}{d_{sep} - d_{test}} & \forall d < d_{sep} \end{cases} \quad ; \; thicknessCoef \in [0; 1]$$

where d is the distance to the closer streamline (see section 2.1 for the definition of the different distances).
Fig 6 shows the same image computed without and with the tapering effect.

Glyph mapping. Once the streamline placement has been computed, the streamlines can be viewed as *skeletons* on which directional glyphs can be mapped. Figure 7 shows an image obtained by mapping such kind of icons onto the computed streamlines. This enables to add a directional information in the field.

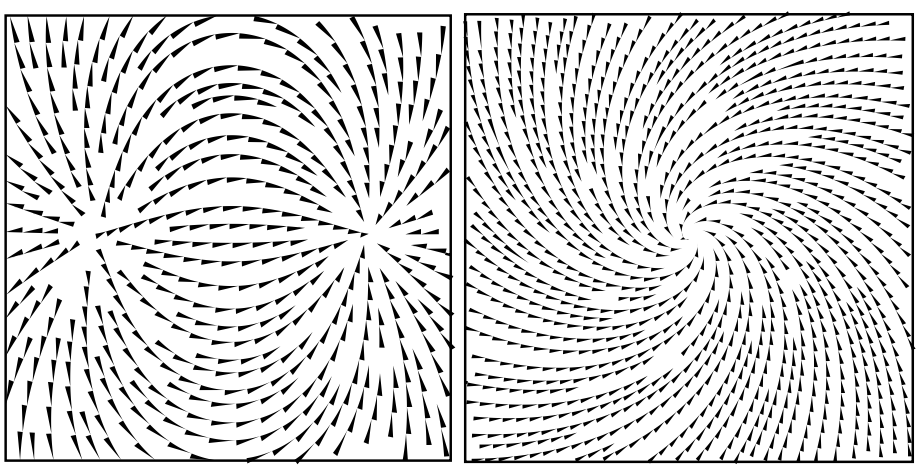


Fig. 7. examples of illustrations with glyph mapping

4 Texture generation

By decreasing the separating distance d_{sep}, the coverage of the streamlines becomes dense over the field. To depict the tangential component of the flow in a dense representation we have to differentiate close streamlines to be able to follow them over the field. This is achieved by mapping a periodic intensity function onto the streamline. Let us consider $f(x)$ a function which associates an intensity to every sample point on a streamline where x is the rank of the sample point within the streamline. f will give the shape of the intensity wave on the streamline. For instance, one may associate the two functions f_1 and f_2 given below:

$$f_1(x) = \frac{1}{2}(1 + sin(\frac{2\pi x}{N})) \quad and \quad f_2(x) = \frac{x \ mod \ N}{N-1}$$

where N is the length of the period as a number of sample points.

f_1 will give a smooth continuously increasing and decreasing intensity while the modulo function f_2 will produce discontinuous segments of increasing intensity in order to remove ambiguities about the orientation of the flow. Figures 8 and 9 show an image obtained using this pair of functions.

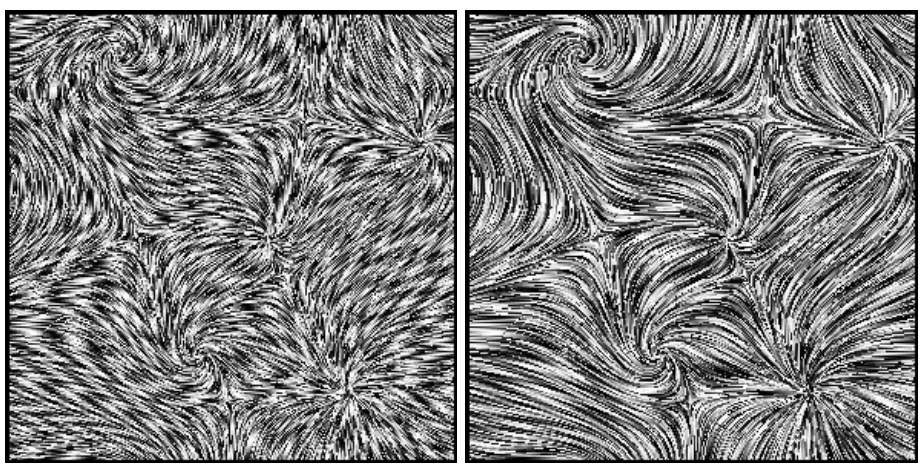


Fig. 8. textures generated with the f_1 function for intensity effects; (left) short and (right) long period of f_1

The images obtained with our method look like LIC images. In fact our algorithm is somehow a dual version of LIC. With LIC, for each pixel p of the output image, one integrates a particle path centered on p forward and backward and then averages intensities of the input texture pixels to get the intensity of the pixel being calculated. In our method, we find an optimal dense coverage of the

field which minimizes the number of integrated streamlines and then associate an intensity to each pixel of all the streamlines.

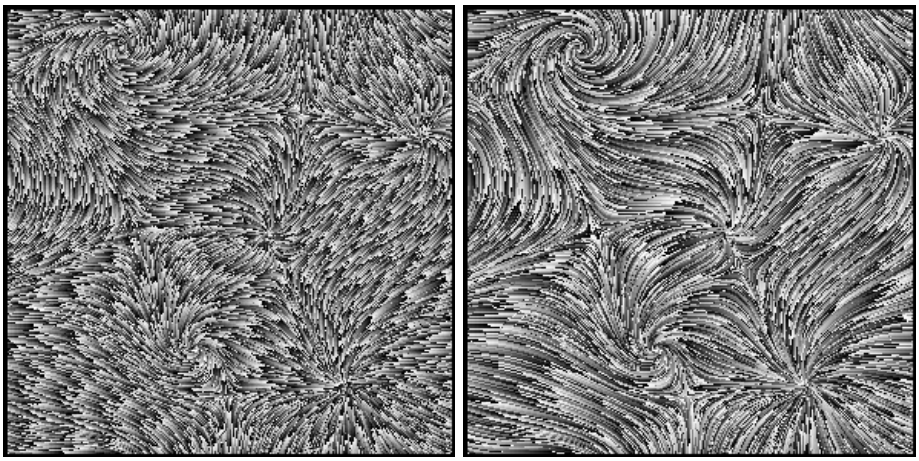


Fig. 9. textures generated with the f_2 function for intensity effects; (left) short and (right) long period of f_2

Now let us point on the advantages of our method over LIC. The first advantage is that our method does not require an input texture to process. With LIC a problem arises when one want to change the length of the apparent streamlines. In case of LIC, it is necessary to change the length of the convolution filter, which gives rise to a computation overhead. It is stated in [1] that doubling the length increases the computation time by a factor of four. Okada and Lane have proposed another solution, which consists in executing the LIC algorithm twice, the resulting image of the first execution being used as the input of the second execution [7]. This method results in lines which are more easily distinguishable but it concentrates the pixels values in a narrow range of intensity, which decreases the global contrast. This effect can be removed by applying post-filters to the final image, but this is done at the expense of a computation overhead (approximately by a factor of two).
With our approach we are able to change the length of the apparent streamlines on demand by simply changing the length of the period of the function as described above, without recomputation of any streamline.

In the traditional LIC algorithm, a streamline integration and a line convolution are computed for each individual pixels, so most of the time is spent in convolution and integration operations. Decreasing the number of streamlines computed would greatly benefit the LIC algorithm. Stalling and Hege proposed the *FastLIC* algorithm which reduces the overall number of streamline computations by sharing the line integral convolution information of each streamline with all the pixels it goes through [8].

With our approach, we compute the minimum number of streamlines necessary to cover entirely the 2D area over which the field is studied.
For instance, the computation of textures images of size 512×512 (such as figures 8 and 9) takes about 25 seconds on a R5000SC-64Mo based system. These results have been obtained using a random seed points selection method and a separating distance of 0.3%. As far as dense texture images are concerned, a drawback of our method is the aliasing effect due to the drawing of adjacent line segments of different colors. To remove this artifact we can smooth the final image by simply applying a blur filter.

5 Conclusion

We have presented an effective method to place long evenly-spaced streamlines with an accurate control on the density of the final image. By changing the separating distance between streamlines we are able to produce from sparse to dense representations of flow fields. We show that our method produces images of a quality at least as good as other methods but that it is computationally less expensive and offers a better control on the rendering process. Future investigations will concern a more efficient integrator, generalization to unsteady flows and real time animation.

References

1. Brian Cabral and L. Leedom. Imaging vector fields using line integral convolution. *Computer Graphics*, 27:263–272, jul 1993.
2. W. C. de Leeuv and Jarke J. van Wijk. Enhanced spot noise for vector field visualization. In *Proc. of Visualization '95*, pages 233–239. IEEE Press, Los Alamitos, CA, oct 1995.
3. Don Dovey. Vector plots for irregular grids. In *Proc. of Visualization '95*, pages 233–239. IEEE Press, Los Alamitos, CA, oct 1995.
4. Lisa K. Forsell. Visualizing flow over curvilinear grid surfaces using line integral convolution. In *Proc. of Visualization '94*, pages 240–247. IEEE Press, Los Alamitos, CA, oct 1994.
5. E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I - Nonstiff Problems*. Springer-Verlag, 1993.
6. Nelson Max, Roger Crawfils, and Charles Grant. Visualizing 3D velocity fields near contour surfaces. In *Proc. of Visualization '94*, pages 248–255. IEEE Press, Los Alamitos, CA, oct 1994.
7. A. Okada and D. Lane. Enhanced line integral convolution with flow feature detection. Technical Report NAS-96-007, NAS, jun 1996.
8. D. Stalling and H-C. Hege. Fast and resolution independent line integral convolution. *Computer Graphics*, 29:249–256, jul 1995.
9. Greg Turk and David Banks. Image-guided streamline placement. *Computer Graphics*, 30:453–460, jul 1996.
10. Jarke J. van Wijk. Spot noise: Texture synthesis for data visualization. *Computer Graphics*, 25(4):309–318, jul 1991.

Interactive Artistic Rendering

Matthew Kaplan **Bruce Gooch** **Elaine Cohen**

Department of Computer Science
University of Utah
http://www.cs.utah.edu/

Abstract

We present an algorithm for rendering subdivision surface models of complex scenes in a variety of artistic styles using an interactively editable particle system. The algorithm is suitable for modeling artistic techniques explicitly by the user, or automatically by the system. Our approach can simulate a large number of artistic effects due to the fact that almost any type of mark made on paper or canvas can be imitated. Any of our artistic effects is customizable by the user through a particle editing interface. The algorithm maintains complete frame-to-frame coherence, a characteristic required for good animation, and runs at interactive rates on current computer graphics workstations.

CR Categories: I.3.0 [Computer Graphics]: General; I.3.6 [Computer Graphics]: Methodology and Techniques.

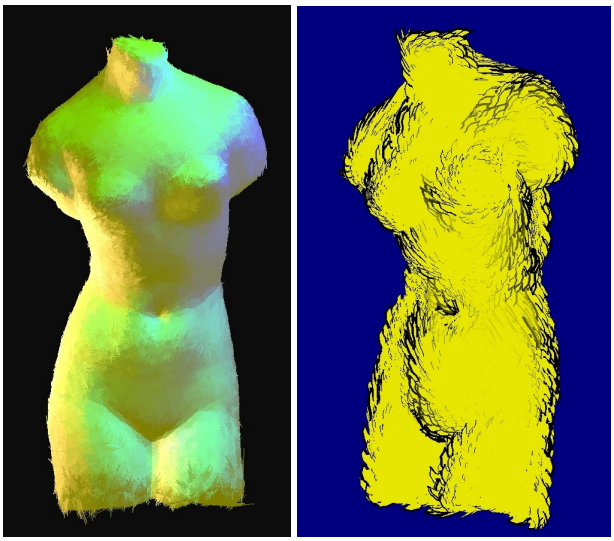
Keywords: interaction, illustration, non-photorealistic rendering, silhouettes, lighting models.

1 Introduction

Both humans and computer programs can take an input image, scene or geometric representation and produce an output image. Computers excel at producing photorealistic images that are difficult for humans to reproduce by hand. On the other hand, artists have developed systems for expressive abstraction to represent complex systems of texture, geometry, tone, and lighting using a few simple pen and ink or paint strokes. This allows artists to concentrate on conveying an idea or feeling instead of being forced to focus on the details that a photorealistic representation entails.

Artists use different abstractions in different circumstances. They represent scenes of staggering visual complexity such as grass, trees, fur or scales using expressive strokes that are indicative of the texture but do not describe it in complete detail. To achieve additional effects, such as dramatic lighting on a smooth surface, an artist may use a cross hatching technique to indicate the relative light on the object at any point. While these methods are only stylized approximations of the actual texture, object geometry, and lighting, they can be more expressive than photorealistic methods.

By expressive, we mean "effectively conveying the meaning or feeling". A motivating question behind our work is : "How do human artists portray the fundamental meaning or feeling of a scene?" A goal has been to find ways in which non-artists can use computers to generate expressive images or scenes. Evidence can be found in the psychophysical research literature to support the idea that the very simplicity of hand-crafted media serves as a means of interaction between the artwork and the viewer [8]. When an artist leaves a scene incomplete, some of the marks on the medium act as "clues" for the viewer. The viewer can use these simple ideas or clues to reconstruct or complete the image by inference [14]. This type of transaction between art and viewer allows the viewer to fill in details for himself and, hence, become more involved in the image. The more detailed the image becomes, the less able the viewer is to "fill in the blanks." When little is left to the imagination, there are fewer possibilities for meaning or feeling to be assigned to a piece of art by the viewer. The simple nature of art allows a greater personification of art on an individual level. Stroke based rendering may enable us to use a computer to emulate handmade art.



(a) (b)

Figure 1: (a) An painterly rendering of the Venus de Milo. (b) Venus de Milo with a fur texture applied.

Goals of this research were an algorithm and related system framework that supports artistic rendering with complete frame-to-frame coherence at interactive rates, to allow a user to completely control particle placement, size, shape and orientation on a per object basis, to generate a variety of artistic effects of significant complexity, and to provide a framework though which other artistic techniques can be rendered with ease. The program renders scenes in a variety of artistic styles based on user defined parameters such as shading models or texture and strokes styles.

Our technique associates particles with the geometry of subdivision surface models. Particles are used to represent hand drawn

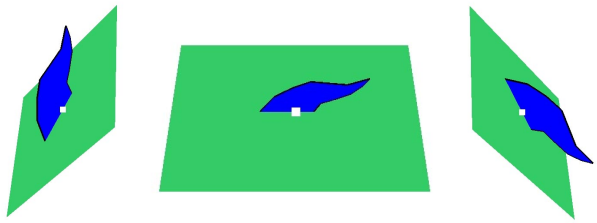


Figure 2: A blue fur/leaf geograftal has been placed on a green regular quadrilateral surface. The white dot illustrates the point at which the geograftal is located on the surface.

strokes or suggestive geometric features not present in the original model. These particles can then be interactivly edited by a user to create artistic effects. The geometry associated with the particles are rendered at interactive speeds on top of the underlying geometry based on information derived from the associated surface's position in screen space.

2 Related Work

Particle systems in general have been used extensively in photorealistic scenes to render complex textures and objects on top of simple geometric structures. Reeves [9] used particle systems to create trees and other complex objects. Fleischer *et al.* [2] used particle systems to render biologically based cellular textures. Meier [7] showed that particle based brush strokes could be used to produce painterly renderings. Her work is a precursor to our technique, since she introduced the method of using particles to locate feature defining strokes in order to maintain complete frame-to-frame coherence.

Smith introduced the idea of graftals as parallel graph grammar languages that were used with a particle system to create complex tree models [12]. An inspiration for this research, Kowalski *et al.* [4] used graftals to render complex textures such as fur and foliage on top of simple models. Their graftal textures demonstrated how different hand drawn effects could be generated by a computer. They also introduced the idea of a graftal texture as an aggregate of individual graftals with a fixed location on the visible part of a model.

The word *graftal* has grown from Smith's original definition to describe a structure that creates surfaces via an implicit model and produces data upon request. We further generalize the definition of graftals to include procedural geometric entities. In this way we can precompute as much information as possible about the graftal, including the normal, position, and color, while waiting until runtime to calculate view or lighting dependent qualities such as size, orientation, or highlight placement and color. For clarity, in the rest of this paper we will refer to graftals which use our augmented definition as *geograftals*. Figure 2 illustrates the concept of a geograftal being applied to a surface.

Our research combines and extends the work of both Meier and Kowalski *et al.* By using geograftal objects, viewpoint invariance is maintained for purposes of animation. Moreover, since each geograftal is statically placed on a model's surface, the creator of a scene can edit the attributes of any geograftal to obtain full control over the look and feel of the hand drawn effects that are produced by our system. We show how this system can be used to automatically create other hand drawn effects such as pointillism, colored pencil, oil paintings, impressionist paintings and pen and ink drawings.

Winkenbach and Salesin [16] showed how pen and ink drawings could be created for parametric surfaces. Our system can create simulated pen and ink drawings that are similar to their results for general polyhedral meshes. Salisbury *et al.* [10] demonstrated how impressive pen and ink drawings could be rendered when the user chose the principal directions for strokes to be rendered. Elber [1] showed that line art for freeform surfaces could be displayed interactively. This paper develops methods that use the principal curvatures of the polyhedral mesh to decide stroke directions automatically, which can produce effects similar to both algorithms. Tone is defined as the relative amount of light at a surface seen by the viewer. Pen and ink renderings have the property of changing tonal values when the same strokes are drawn for a model seen at different scales. Inspired by the method of Salisbury *et al.* [11] for creating scale independent tone for pen and ink drawings, we use scaling functions on our geograftals to recreate the effect of scale independent image tone for pen and ink drawings.

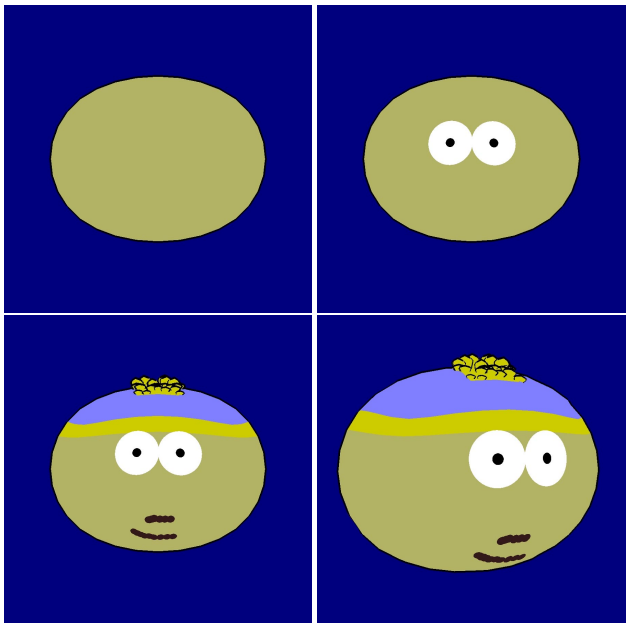


Figure 3: Geograftals are painted onto a ellipsoid model: First two geograftals representing eyes are placed. Then geograftal dots are used to paint facial features on the model. Next, an area of the model is randomly annotated with geograftal a "hair" texture, and finally the model and its associated geograftal textures are rotated. Character likeness courtesy of Comedy Central.

3 Geograftal Implementation

Section 3.1 describes the system implementation. Section 3.2 presents geograftal generation. In Section 3.3 we discuss drawing geograftals, and the effect of stored attributes and scaling functions on how the geograftals are drawn on the screen. Finally, in Section 3.4, we introduce multilayer editing of geograftals in an interactive environment to allow simple creation of more complex effects.

3.1 Software Framework

The system renders a mesh of connected quadrilateral sufaces using OpenGL. Each model within a scene is stored as a list of constituent quadrilateral surfaces which define the mesh. Individual geograftal objects are stored with each surface. Attributes such as location, width, height, type and color are stored with each geograftal object.

(a) (b)

Figure 4: (a) An imitation of Dr. Seuss's truffula trees. (b) A multilayer geograftal texture.

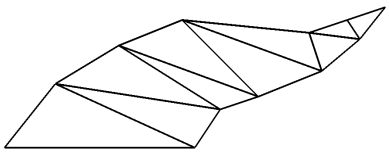


Figure 5: Geograftals are drawn as triangle strips in OpenGL. The basis vectors for this geograftal are simply unit vectors on the X and Y axes.

The type value denotes both the shape of the geograftal and stroke effects such as pen, pencil, or paint brush.

The research we present uses geometry-based procedural objects rather than the graftal texture approach used previously by others. We associate individual geograftal objects with a surface object that represents the geometry of the model. Each geograftal determines at run time how it is to be drawn on the screen. Geograftals also have associated scaling functions to control their size and shape.

3.2 Generating Geograftals

There are two goals when generating geograftal objects. One is to achieve the simulation of random placement over the surface of the object to imitate the hand drawn quality we desire. The other is to have complete control over the placement of specific geograftals for special cases. We can achieve precision placement by allowing the user to select the point on the surface to place the geograftal. We achieve a random placement on a surface by parameterizing the surface quadrilateral and then choosing random U and V values. These two cases are illustrated by Figure 3. Geograftal normals are calculated as a linear combination of the normals at the four corner points. The type, width and height attributes are specified by the user.

We produce effects similar to those in Kowalski *et al.* by choosing random geograftals, based on a user defined density, for all of the surfaces to be textured. The system also allows the user to add more geograftals interactively to create the appearance of non-uniform or denser texture coverage of the surface. Due to the initial random placement of particles, the appearance of a random texture is created. Since geograftals are associated with the surface rather than a texture, we can apply more than one distinct type of geograftal to a surface with this method, resulting in the appearance of multiple textures on the model. An example of geograftal textures is given in Figure 4(a).

3.3 Drawing Geograftals

Each geograftal is displayed as an OpenGL triangle strip and each geograftal silhouette as a OpenGL line strip. A triangle strip for a fur/leaf geograftal is shown in Figure 5. Points that make up these strips are described as a linear combination of two basis vectors. Rather than forming an orthonormal basis with the view vector, we use the geograftal normal and the cross product of the geograftal normal with the view vector as basis vectors. The length of the basis vectors is specified by user defined width and height values. This has the effect of creating realistic foreshortening on the geograftals while maintaining a consistent orientation via the view vector.

Figure 6: Leaf geograftals on this sphere are shown at even intervals along the X axis. Towards the interior, the geograftals disappear completely.

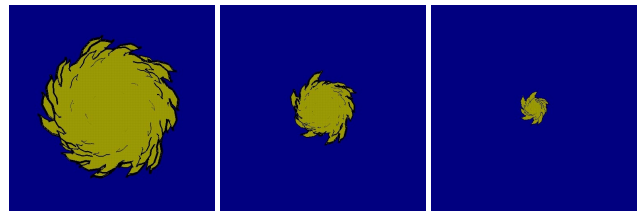


Figure 7: As objects move away from the viewer some of the geograftals shrink in size while a randomly selected few grow in size.

The hand drawn effect that we are trying to attain has the property of being most evident near silhouette edges. Therefore, geograftal objects are drawn as flush to the screen and as large as possible when the geograftal's normal nears perpendicular with the view vector. The property of foreshortening is desirable because it maximizes the screen space of geograftal objects near silhouettes and interior cusps.

Frame-to-frame coherence is maintained by drawing all of the geograftals for every frame. While it isn't necessary to render back-facing surfaces, all of the geograftals must be drawn since they may poke out from behind the object. In any case, they are more likely to be visible after the surface with which they are associated disappears or before the surface appears. If only geograftals located on front facing surfaces are drawn, jarring "popping" effects occur when a geograftal's associated surface switches between front or back facing. Popping occurs when a geograftal suddenly appears or disappears. The same disturbing popping occurs if geograftals are no longer drawn in the interior of an object. In order to create the illusion that geograftals gradually appear and disappear, we use scaling functions that affect the size of a geograftal based on the geograftal's position and orientation in relation to the view vector. Because all geograftals are drawn during every frame and because we now change the attributes of the geograftals to create the desired effects, popping no longer occurs. We define the general scaling function:

$$F_G = 1 - |\textit{Geograftal Normal} \cdot \textit{View Vector}| \qquad (1)$$

where F_G is the scaling factor. This scaling function produces a value between 0 and 1, where 0 means the geograftal normal is directly opposite the view vector and 1 means that the geograftal's normal is perpendicular to the view vector. Then, the basis vectors that define each geograftal are multiplied by F_G. This has the effect

of changing the geograftal's size and shape continuously over the surface of the object. The popping effect is eliminated while maintaining the style of drawing only geograftals near silhouettes. This creates large geograftals near silhouettes and tiny geograftals in the interior.

Unfortunately, the black edge lines of the small geograftals still disrupt the effect of displaying information only near silhouette edges. We solve this problem by creating additional scaling functions. The first scales the geograftal edge width, making the edges drawn near silhouettes more prominent and those drawn in the interior less visible. This is an effect that artists strive for, making the outline of an object bold to create a cohesive effect for the whole object while giving less importance to individual details [6]. The second scaling function changes the color of the geograftal edges to match the color of the object. This creates the illusion that the geograftal has disappeared near the interior but smoothly appears as it approaches an area where it needs to be drawn. We have found that constraining F_G between 0 and 0.7 works well. This enforces color scaling only in the interior of the object and draws geograftal silhouettes near model silhouettes in black. An example of how geograftals scale their width, height and line color is shown in Figure 6.

When an object is at a greater distance from the viewer, many artists draw a smaller number of disproportionately large strokes [13]. To avoid removing and adding geograftals as they pass a distance threshold, we apply scaling functions to the geograftals with distance. We can define a smooth scaling factor for distance:

$$F_D = \textit{Actual Distance}/\textit{Maximum Distance} \tag{2}$$

where *Actual Distance* is the actual geograftal distance from the viewer and *Maximum Distance* is the maximum distance from the viewer possible in the viewing frustum. Each geograftal is assigned a random scaling function of a predetermined type that allows for some of them to grow smaller and some to grow larger with distance. To minimize the geograftal's features with distance, we multiply their feature values by F_D. To make geograftal features larger with distance, their feature values are multiplied by $1 + (1 - F_D)$. These are just two of the many scaling functions we have tested that yield pleasing results.

Scaling by F_D has the desired effect as shown in Figure 7. We found, however, that leaving the size and number of the geograftals the same while using a scaling function to decrease the width of the silhouette edges with distance may have a more pleasing effect. This interactively simulates the effect of line weight depth cueing. Line weight depth cueing establishes the distance from the viewer to the model based on the boldness of the silhouette. Artists use this method to portray three dimensions in a two-dimensional medium [6].

Quite different effects are created when non-linear functions are used to scale the geograftals. We have experimented with these types of scaling functions but none seem to be as visually pleasing as the linear examples we have presented.

3.4 Editing Geograftals

Because we specify geograftals as individual objects and geograftal textures as the sum of geograftals placed on the surfaces of the models, we are able to edit the geograftal's attributes individually or all at once and save the results. To edit an individual geograftal, the user chooses which geograftal to affect and then changes that geograftal's parameters. The edits update interactively, so the changes can be seen as they are made. This allows individual geograftals to be "sculpted" on an object as in Figure 3. With this method, distinguishing features can be designed. Since the attributes of every object are fully modifiable, this gives the user the ability to precisely model an entire scene.

Global edits are also possible with our system. By changing global attribute values, we can change the attributes of all of the geograftals or just a subset of them. Since we may want to build a geograftal texture with more than a single type of geograftal, we allow for multilayer editing. Each geograftal texture applied to a model is represented as a layer. A layer is a set of geograftals composed of a single geograftal type which are associated as a group. By selecting a layer on a multilayer texture, we can apply changes just to the geograftals that belong to that specific texture layer. This allows us to composite multiple texture layers on a single model, achieving intricate multilayer textures as shown in Figure 4(b).

4 Shading

Artists often use lighting effects to provide more information about the shape of an object. Geograftals can be used to convey the same shape information that is conveyed by traditional lighting and shading as described in Section 5.4. It is convenient to calculate and save lighting information to scale and color geograftals rather than use an automated lighting system, such as the one provided by OpenGL. The particular lighting we present is occlusion free and simulates shadows with sources of negative light. We calculate the light values for every point on the mesh and then apply smooth shading to the surfaces to obtain the appearance of a lit object. The value of the light at a surface point is only sensitive to the direction of the normal at that point relative to the light source. This describes the shape of the object uniformly over the surface without regard to shadows. We use the following non-standard equation to describe the light at any point:

$$C = O_c + \Sigma\left(L_c N \cdot L_D\right) \tag{3}$$

where O_c is object color, L_c is light color, N is the point normal, L_D is the light direction and C is the resultant color. This is a simple additive light shader that maintains the color of the object.

After clamping the value to a valid color range, we apply smooth shading to the surfaces using the light values for the points to obtain a smoothly lit model. We can obtain interactive rates for this calculation in complex scenes as reported in Table 1. Geograftals are shaded by taking their color value from the nearest point on the surface. See Figures 10(a) and 12 for examples of these shading techniques.

5 Line Drawing

A number of line drawing effects can be created by using geograftals to represent strokes on a surface. In Section 5.1 we describe a method of calculating silhouettes and internal cusps. Next, in Section 5.2 an algorithm for interactive line art is presented. Finally, in Section 5.5 this algorithm is extended to include other painting techniques.

5.1 Interactive silhouette and cusp calculation

Silhouettes are lines that describe the shape of the object near outside edges along the polyhedral mesh. Here they are defined be an edge which joins both a front and a back facing surface. We define a cusp as an edge that joins any surfaces whose normals have an angle of greater than 90 degrees between them.

Cusps can be identified in a preprocess, since they are view-independent. The cusp edges can be put in a list and displayed quickly as a set of GL_LINES.

Silhouettes, however, are defined only in relation to the view vector. One of the goals of this work is to provide scene invariance. Markosian *et al.* [5] showed how to calculate silhouettes quickly,

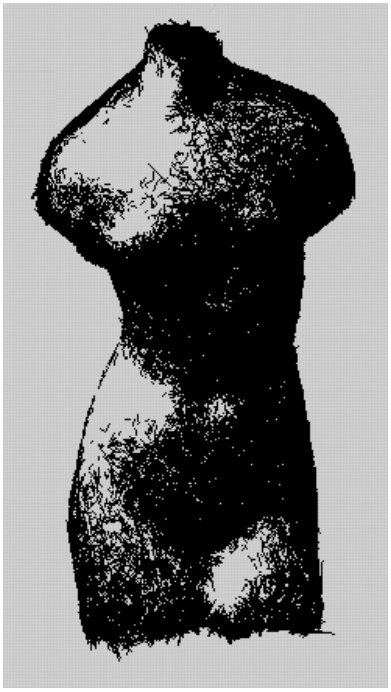


Figure 8: Venus model shaded with randomly placed geograftal strokes.

Model	Points	Polygons	0 g/p	1 g/p	100 g/p
Sphere	98	96	66	65.1	33
3 Spheres	294	288	66	61	16.5
Refined Sphere	386	384	65.8	58.1	13.2
Venus De Milo	4257	4254	62	11.9	1.44

Table 1: Model information and timings, in frames per second, on a 195Mhz MIPS R10K SGI Indigo 2 with an Extreme graphics card, for the number of geograftals per polygon. 100 g/p is presented as a pathological case.

but did not guarantee that every silhouette edge would be found. Gooch *et al.* [3] showed how silhouettes could be rendered using environment maps. The shape of such silhouettes is intriguing but does not provide predictable effects. For our purposes, a brute force search of all front-facing polygons proved fast enough. As an example, we were able to achieve 62 frames per second using a model with 4257 surfaces, displaying both surfaces and silhouettes as shown in Table 1. After calculating the silhouettes for the new viewpoint, they are put in a list and can be displayed quickly in the same fashion as the cusps. Silhouettes calculated by this method are demonstrated in Figure 9(a). The method presented by Kowalski *et al.* obtained a hand drawn look by perturbing the silhouette edges. It is clear that this method is viable here since points in the edge list can be perturbed to obtain this effect, but this may not be practical for preserving scene invariance since random perturbations of silhouettes edges would lead to inconsistent renderings of a scene.

Artists often draw silhouettes with varying widths to indicate the lighting, curvature, or distance from the viewer of the object [13]. We emulate these effects by changing silhouette line widths with a scaling function associated with the light on the surface along silhouette edges.

$$S_w = \left(1 - \left(RGB_{\text{Total}}/RGB_{\text{Max}}\right)\right) MSW \qquad (4)$$

Here, S_w is the silhouette width, RGB_{Total} is the sum of the RGB color values at the corner points of the surface, RGB_{Max} is the maximum RGB color values for those points and MSW is the maximum silhouette width. The inner term, $1 - \left(RGB_{\text{Total}}/RGB_{\text{Max}}\right)$ indicates the relative amount of light on an edge. We next multiply this by the largest pixel width value for the given edge line to obtain the actual line width for a silhouette edge S_w. This results in a silhouette width that changes smoothly with the lighting on an object as in Figure 9(b).

5.2 Automatic Line Shading

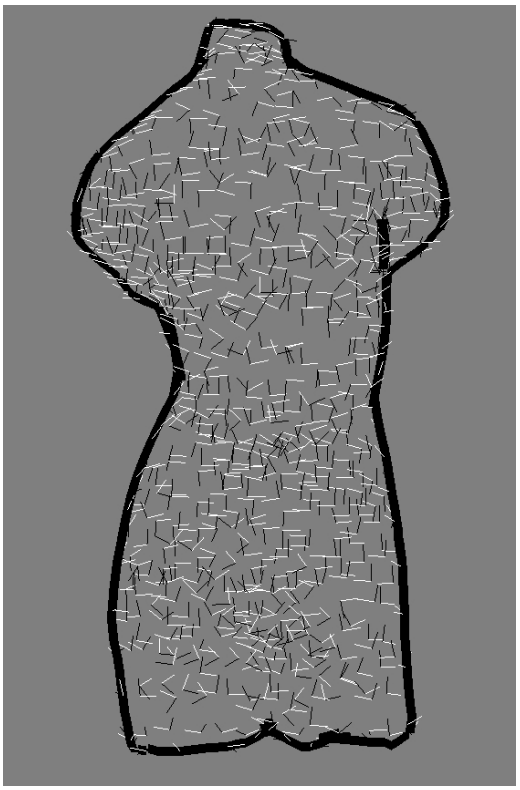
Artists often use loose and sketchy cross-hatching techniques to illustrate lighting effects on an object with black and white pen and ink strokes [13] as in Figure 8. We use graftals as stroke objects to define a texture that responds to light rather than view position. Stroke computation methods, then interactive display methods are presented.

5.3 Determining Stroke Attributes

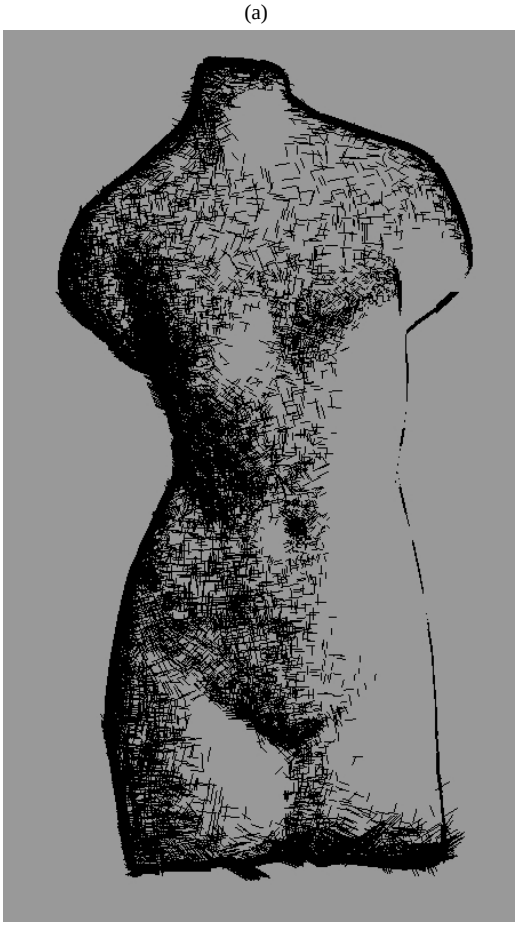
We would like strokes to appear to be randomly placed lines on the surface of the object. Represented as geograftals, strokes are randomly placed on the surface using the method described in Section 3.2. The geograftal position is used to locate the first point that defines the stroke line. There are several strategies which may be employed to find the second point that defines the stroke line segment. To give the effect of random crosshatching, we choose a random unit vector on the surface plane. Then this vector is multiplied by the stroke length and the result is added to the original position yielding a second point which defines the end of the stroke. We define the length of the stroke to equal the average of the lengths of the u, v basis vectors for the local surface quadrilateral. This length is convenient because it prevents the stroke from protruding past adjacent surfaces and affecting the perceived light of surfaces that are not immediate neighbors. This constrains the stroke to provide information about local light only. Perceived light on the model is blurred when strokes from a large number of non-local surfaces overlap.

Winkenbach *et al.* [16] previously showed that allowing a user to choose vector directions in which to draw strokes for a two dimensional scene yields pleasing results. (While this may be effective in an image-based rendering approach, we found it impractical to have the user specify directions in our system, since after rotation and translation, the vectors may not describe relevant strokes because the objects they were associated with may be in a different screen position.) It is noticeable that users of image-based approaches often choose vectors that seem to closely correspond to the principal curvatures of the surface. We automatically generate this effect by deriving an estimate of the two principal curvatures at each point in the polyhedral mesh as presented by Taubin [15]. These principal curvatures can be used to calculate the second point to represent the stroke. This method is similar in spirit to a method presented by Elber [1] and yields strokes which appear consistent with model's curvature. The principal curvature vectors for the Venus de Milo mesh are shown in Figure 9(a). The line art result of such strokes is shown in Figure 9(b).

Since strokes are represented in this system as single lines, they may overlap the edge of the mesh surface with which they are associated. Indeed, it is desirable for them to do so. If strokes were drawn entirely within each polygon, the viewer would have a clear sense of the mesh structure since no strokes would cross any edges. However, strokes rendered in this manner hang off the edges of silhouettes, leading to a loose and sketchy feel, which is often a natural side effect of hand drawn art.



(a)



(b)

Figure 9: (a) shows the principal curvatures of the mesh at each surface point. (b) shows the model shaded with curvature oriented geograftal strokes.

5.4 Stroke Shading Model

The stroke shading algorithm requires a potentially large number of geograftals for each surface. Calculating these strokes at run-time might be too costly to achieve interactive rates. In order to achieve interactive rates, we need to precompute a list of the strokes that might be needed to render tone for each object. Tone (as defined in Section 2) on a surface is represented by the number of strokes drawn on each surface quadrilateral.

We define an array of N stroke geograftals for each surface. When objects are initialized, we precompute N strokes and enter them in this array. We use the lighting values that we calculated in Section 4 to determine the number of strokes to draw to define relative tones.

$$T = \left(1 - \left(\mathrm{RGB}_{Total}/\mathrm{RGB}_{Max}\right)\right) N \tag{5}$$

Here, T is the number of strokes to draw to define the tone, RGB_{Total} is the sum of the RGB color values at the corner points of the surface, RGB_{Max} is the maximum RGB color values for those points that make up the surface. The inner term $\mathrm{RGB}_{Total}/\mathrm{RGB}_{Max}$ defines a smooth scale S, between 0 and 1 that represents the relative amount of light at a surface. T is an index into the array of the strokes. When we display the strokes, we use this index to determine how many strokes to draw. We draw each stroke whose array index is less than our index bound T. This calculation is quick to update since we never change the strokes.

This algorithm maintains interframe coherence because the same strokes are drawn under identical lighting conditions. Since only the lighting model affects the stroke index, viewpoint changes have no effect on the strokes drawn. When the strokes are drawn there is a recurrence of the popping effect. We have found that for sufficiently large N (100 seems to work well in practice), this effect is minimal since the popping occurs as a part of a smooth transition between line drawn tones within a larger texture.

Salisbury pointed out that the tone of a line drawing is changed when the drawing is scaled [11]. We use a scaling function that increases the stroke index as distance from the viewer increases. This effectively compensates for the relative tone of an object as its distance from the viewer is changed. Our simulations have shown that the results of the scaling function depends on factors such as the surface area of the polygons in the system, and the size of the viewing frustum relative to the stroke widths and the objects. We achieve our best results using scaling functions that are tailored by the user to the specific object and viewing volume. An example of this is shown in Figure 10(f). Since a precise solution to the tone matching problem may require significant run time computation for every viewpoint change, and because we attempt to maintain interactive rates, it is currently beyond the scope of this work.

5.5 Colored Line Art

A variety of colored pencil or oil painting effects are achieved by coloring the stroke lines. Since there is no inherent light or color in the black strokes applied in the previous section, the strokes were sufficient to imply the concept of light. When the strokes are colored, they no longer imply lighting. They seem to reveal holes in the model because lighting is now shown by the color of the strokes rather than by stroke placement. We present two solutions to this problem. The first solution is to shade the underlying surfaces. This is analogous to a painter laying down an underpainting to establish tone before applying detail. In this case, the strokes are used conceptually as with the geograftal objects described in Section 3. The strokes are a “hint” of a texture that allow the user to complete the rest of the texture by inference. The second solution is to increase the coverage of the strokes on each surface in an equal way. Here we allow the user to set the line index into the stroke array for every

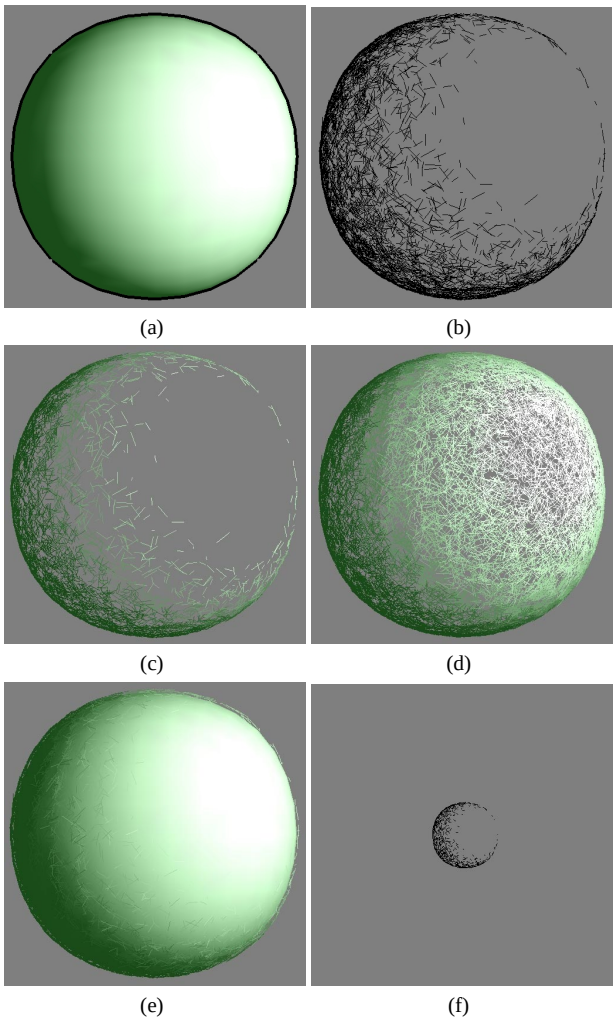


Figure 10: The line art strokes that imply lighting in picture (b) imply holes in the sphere in (c). To correct this we either draw enough strokes to obtain an adequate surface coverage as in (d), or shade the surfaces on which the strokes lie as in (e). In (f), the tone of (b) has been maintained under translation by applying a scaling function.

surface to a value that is large enough to obtain a sufficient surface coverage. We can simulate different painting effects by changing global attributes such as brush type, size and orientation. Using this technique we produce effects similar to Meier's painterly rendering [7]. These techniques are illustrated in Figure 10. Other artistic techniques produced with this system such as impressionist and pointillist are shown in Figure 11.

6 A Graftal Painting System

A geograftal system can be used to paint strokes onto the surface of an object. After choosing a brush shape and size, it is possible to paint on the object by choosing the position for the brush stroke to be applied. We seed a geograftal object of the appropriate brush type and attributes at that position and apply a stroke. Unlike painting systems based on standard texture mapping, geograftals sidestep the problems of surface distortion and parametric overlapping because only the position of the geograftal is tied to the object. Because of this, geograftal features such as size and shape are not vulnerable to distortion or cropping where the shape of the model is irregular or where the parametric bounds of the texture map lie. Moreover, the strokes of the geograftal paint system are fully editable. An example of this is shown in Figure 3 where the mouth and chin of the character have been painted on by a user.

7 Future Work and Conclusion

We have presented a new technique for rendering artistic effects within a coherent framework. We have shown a stable modeling approach that encompasses and extends the artistic effects previously obtainable by various non-photorealistic rendering methods. Our system provides an automated method for rendering complex scenes with expressive artistic techniques from simple models. We have proposed a method that maintains interframe coherence by introducing the idea of geometric graftal objects to represent graftal textures and have shown that such a system can generalize to many other artistic effects. Moreover, the user of our system has complete control over each geograftal object and we provide multilayer editing techniques. The system renders at rates from 2 to 60 frames per second on a low end workstation, depending on the complexity of the scene and the effects desired. Our test scenes had meshes containing between 512 to 16,000 polygons.

Extending the range of styles currently available for geograftal rendering is one of the immediate goals of future work. It would also be valuable to formalize notions of view dependent object behavior in ways that allow for easy geograftal design by users. Since each model is represented as a subdivision surface mesh, it might be easy to achieve higher frame rates by using lower resolution meshes for rendering objects at a distance. It might also be useful to examine other intrinsic properties of the subdivision mesh to generate line strokes. Our silhouettes currently only vary line width based on lighting. Other types of data such as curvature might be useful metrics for silhouette line width variation. Furthermore, our system currently only handles straight line strokes. Strokes represented by user defined curves or bitmaps would be a valuable addition.

References

[1] Gershon Elber. Interactive line art rendering of freeform surfaces. *Computer Graphics Forum*, 18(3):1–12, September 1999. ISSN 1067-7055.

[2] Kurt Fleischer, David Laidlaw, Bena Currin, and Alan Barr. Cellular texture generation. *Proceedings of SIGGRAPH 95*, pages 239–248, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[3] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Rich Riesenfeld. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999. ISBN 1-58113-082-1.

[4] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-based rendering of fur, grass, and trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[5] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97*, pages 415–420, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[6] Judy Martin. *Technical Illustration: Materials, Methods, and Techniques*, volume 1. Macdonald and Co. Publishers, 1989.

[7] Barbara J. Meier. Painterly rendering for animation. *Proceedings of SIGGRAPH 96*, pages 477–484, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[8] V.S. Ramachandran and William Hirstein. The science of art a neurological theory of aesthetic experience. *Journal of Consciousness Studies*, 6(6-7):15–51, 1999.

[9] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):313–322, July 1985. Held in San Francisco, California.

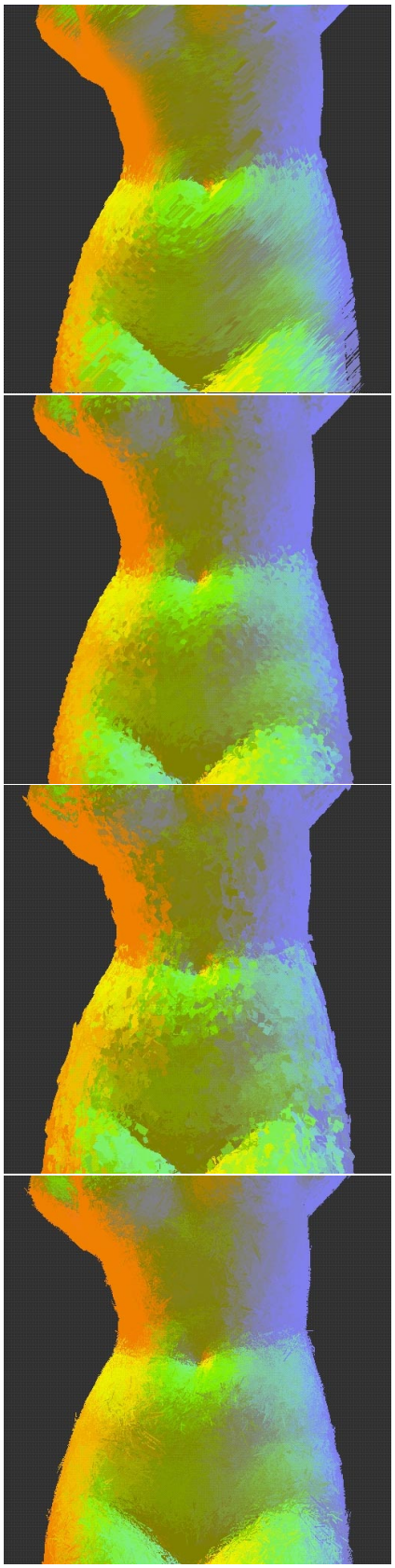


Figure 11: By interactively varying input parameters, we achieve different painterly effects. Examples shown here are, from top to bottom, oriented impressionist, pointillist, and on the bottom two pictures, random impressionist with different input parameters.

[10] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. *Proceedings of SIGGRAPH 97*, pages 401–406, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[11] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. *Proceedings of SIGGRAPH 96*, pages 461–468, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[12] Alvy Ray Smith. Plants, fractals and formal languages. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):1–10, July 1984. Held in Minneapolis, Minnesota.

[13] Alvy Ray Smith, Michael Wright, and James Horton. *An Introduction to Art Techniques*. DK Publishing Inc., 1995.

[14] Robert L. Solso. *Cognition and the Visual Arts*. MIT Press/Bradford Books Series in Cognitive Psychology, 1999.

[15] Gabriel Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. *ICCV95*, pages 902–907, 1995.

[16] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

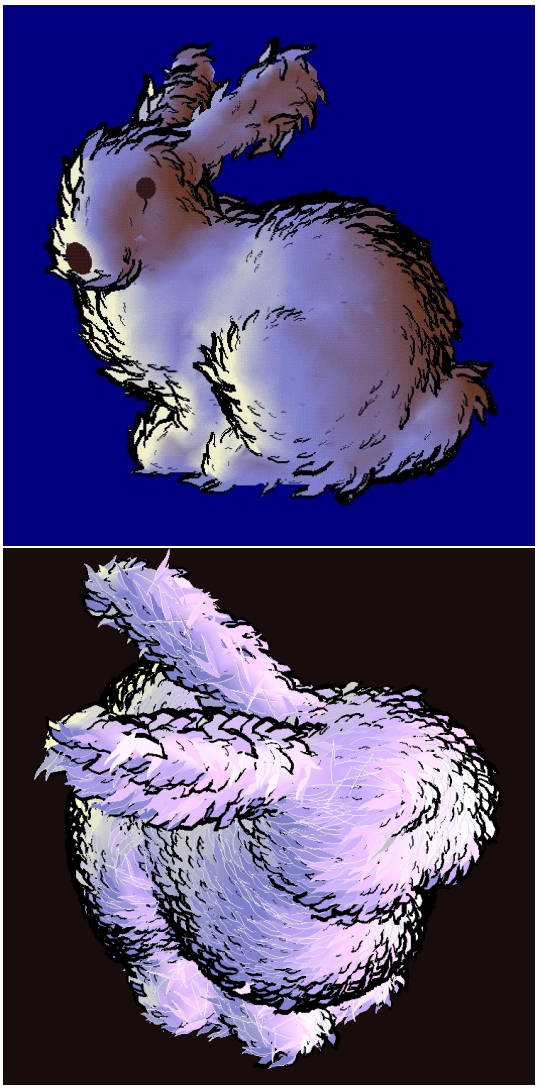


Figure 12: Bunnies with varied graftal fur textures applied. Model courtesy of Stanford University.

To appear in the SIGGRAPH 99 conference proceedings

Art-Based Rendering of Fur, Grass, and Trees

Michael A. Kowalski[◇*] *Lee Markosian*[◇] *J.D. Northrup*[◇] *Lubomir Bourdev*[†]
Ronen Barzel[‡] *Loring S. Holden*[◇] *John F. Hughes*[◇]

[◇]Department of Computer Science
Brown University
{mak,lem,jdn,lsh,jfh}@cs.brown.edu

[†]Advanced Technology Group
Adobe Systems
lbourdev@adobe.com

[‡]Pixar
ronen@pixar.com

Abstract

Artists and illustrators can evoke the complexity of fur or vegetation with relatively few well-placed strokes. We present an algorithm that uses strokes to render 3D computer graphics scenes in a stylized manner suggesting the complexity of the scene without representing it explicitly. The basic algorithm is customizable to produce a range of effects including fur, grass and trees, as we demonstrate in this paper and accompanying video. The algorithm is implemented within a broader framework that supports procedural stroke-based textures on polyhedral models. It renders moderately complex scenes at multiple frames per second on current graphics workstations, and provides some interframe coherence.

CR Categories and Subject Descriptors: I.3.3: Computer Graphics: Picture/Image Generation; line and curve generation; bitmap and frambuffer operations; I.3.5 Computer Graphics: Computational Geometry and Object Modeling: curve, surface, solid, and object representations; I.3.7: Three-Dimensional Graphics and Realism: Color, shading, shadowing, and texture. **Additional Key Words:** Non-photorealistic rendering, graftals, procedural textures.

1 Introduction

Any art student can rapidly draw a teddy bear or a grassy field. But for computer graphics, fur and grass are complex and time-consuming. Even so, the artist's few-stroke rendering may have greater persuasive or evocative power than the usual computer-graphics rendering.

How does the artist effectively communicate the teddy bear or grass? By rapidly creating an impression of free-form shape – difficult to do with conventional 3D modeling systems – and then drawing a few well-chosen strokes. This paper describes some of our efforts to expand the expressive power of 3D graphics by adopting techniques for depicting complexity from the centuries-old disciplines of art and illustration.

Three goals in our work on art-based graphics are to give the designer of a scene control over the style of rendering; to ease the burden of modeling complex scenes by treating the rendering strategy as an aspect of modeling; and to provide interframe coherence for the kinds of stylized renderings we've developed. Other goals, less directly related to the work in this paper, include the development of systems for rapidly creating free-form shapes [9], and the control of scene composition.

Our approach to creating complex expressive renderings is to target the kinds of images made by artists and illustrators and reproduce the effects and techniques we observe in those images. Much 2D art and illustration is created by making strokes on a flat surface (paper, canvas), so we have based our work on what we call "stroke-based textures." We started from the drawings of Theodore Geisel ("Dr. Seuss") [4, 5], in part because they are such an extreme departure from the domain of conventional computer graphics.

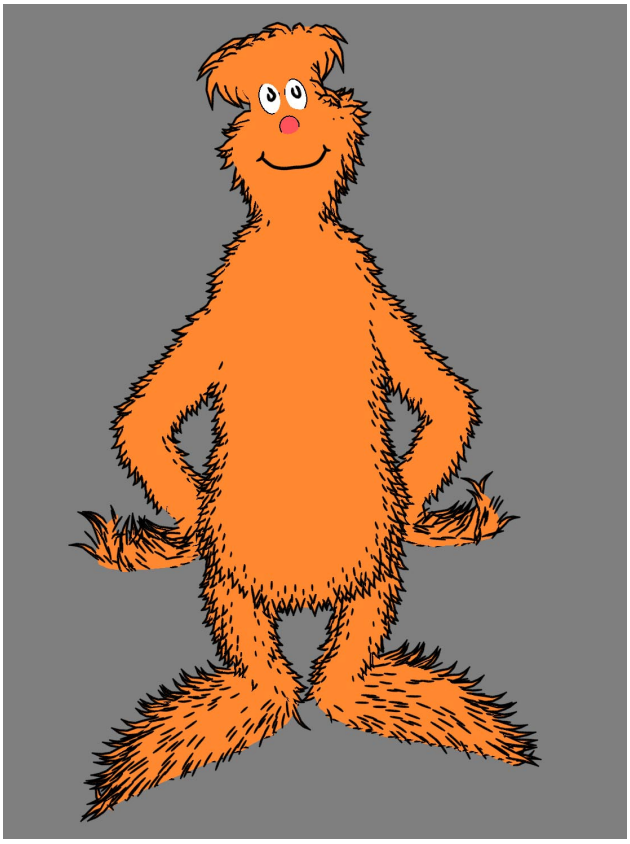


Figure 1 A furry creature, after Dr. Seuss. The fur is generated in a view-dependent way by a procedural stroke-based texture that places it near silhouettes, varying the style of the tufts according to how much the underlying surface faces the viewer.

Currently at ATR Media Integration & Communications Research Laboratories, Kyoto, 619-0288, Japan.

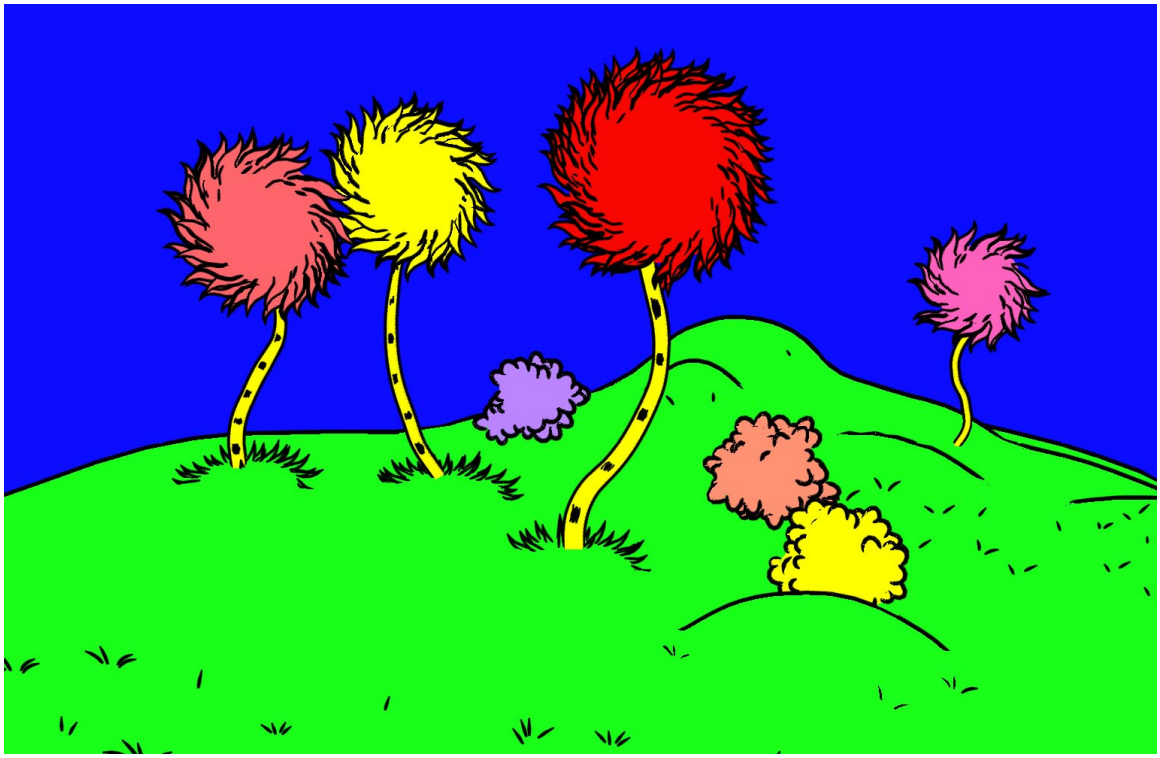


Figure 2 A more complex scene, again based on the style of Dr. Seuss. The grass, bushes, and truffula treetops are implemented with graftal textures that use the same basic algorithm to place graftals with a variety of shapes and drawing rules. The truffula tree trunks are drawn by stroke textures (not graftal textures) assigned to ribbon-like surfaces that always face the viewer. The treetops use the same type of graftal as seen in the previous figure, but with a different orientation rule: they always circulate clockwise around the treetop, no matter what the point of view. This cannot be modeled with any fixed geometry, of course.

There are two main research challenges: the development of algorithms and a software framework in which procedural stroke-based textures can be rendered, and the development of a user interface (within the context of a free-form modeling system) that allows a designer to assign and customize such procedural textures. In this paper, we describe our approach to the first of these challenges.

We have developed a system to generate and render stroke-based textures that mimic the styles of two artists, and a framework for generalizing this to other techniques. The system renders the images shown in this paper at several frames per second on a Sun Sparc Ultra 2 model 2/300 with Creator 3D graphics, and even faster on a high-end PC. Our main contributions are the system architecture, the (partial) temporal coherence of the texture elements, and the particular methods used to mimic Dr. Seuss's and Geoffrey Hayes's [7] styles.

2 Prior Work

Using art as a motivation for computer graphics techniques is not new, and our work builds on the efforts of many others. Fundamental to our ideas are the particle systems of Reeves [12, 13], which he used to create trees, fireworks, and other complex imagery from relatively simple geometry. Alvy Ray Smith's later use of particles, together with recursively defined L-systems that he called "graftals," extended this to more biologically accurate tree and plant models [15]. His "Cartoon Tree" is a direct precursor to the work in this paper. Graftals have since come to be described more generally: according to Badler and Glassner [1], "Fractals and graftals create surfaces via an implicit model that produces data when requested." We use the word "graftal" in this much more general sense.

We use a modified version of the "difference image" stroke-placing algorithm of Salisbury *et al.* [14] to place procedural texture elements at specific areas of the surface. Winkenbach and Salesin [17] described the use of "indication" (showing a texture on part of an object) in pen-and-ink rendering. And Strothotte *et al.* [16] extensively discuss the use of artistic styles to evoke particular effects or perceptions.

At a more mechanical level, Meier's work on particle-based brush strokes [11] was a major inspiration in two ways: first, her use of particles to govern strokes that suggest complexity in her Monet-like renderings showed that not all complexity need be geometric; second, the fixed spacing of the particles on the objects, which limited how closely one could zoom into the scene, inspired us to seek a similar but hybrid screen/object space technique.

The present work builds on our earlier efforts [10] to produce nonphotorealistic effects at interactive frame rates. One limitation of our earlier system was that it supported just one "style" at any given time – applying the style equally to every object in the scene. A more flexible system would allow the designer of a virtual scene to assign different nonphotorealistic "textures" to different surfaces within the scene. The framework we describe in the next section makes this possible. For instance, the fur texture on the creature in figure 1 is applied over most of the body but not the face, even in profile. Our other images show more examples of the selective use of distinct nonphotorealistic textures applied to objects in a scene according to what each represents.

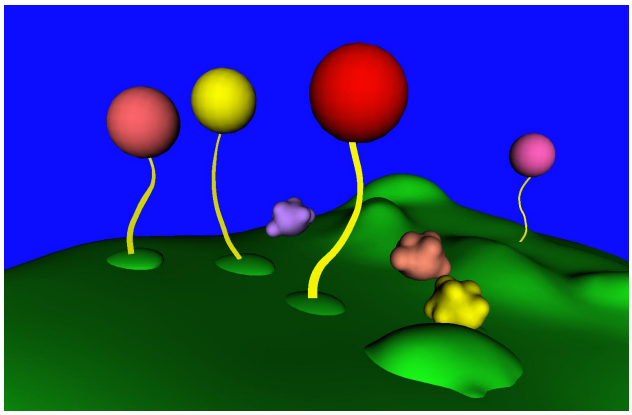


Figure 3 The same scene as in figure 2 rendered without graftal textures or the stroke-based textures on the truffula trunks.

3 Software Framework

Our procedural stroke-based textures are implemented within a general system for rendering polyhedral models using OpenGL [2]. In our system models are divided into one or more surface regions (called *patches*), to each of which the user can assign one or more procedural textures (called *textures*) – although just one is active at a time. The "procedure" that defines a texture needn't be complicated – many simply draw their patch in some conventional style (e.g., smooth-shaded or wireframe). One of our textures performs Floyd-Steinberg dithering [3]. Others perform a variety of hatching effects.

An important component of the system is the provision of *reference images*. These are off-screen renderings of the scene, subsequently read from frame-buffer memory to main memory and made available to the procedural textures. We currently use two kinds of reference images: a *color reference image* and an *ID reference image*.

To prepare the color reference image, the active texture of each patch is asked to render into it in some appropriate way, depending on how the texture will use the image. For example, the graftal textures described in the next section use the color reference image in a special way to decide where to draw tufts of fur, grass, or leaves. For the ID reference image, triangles (or edges) are each rendered with a color that uniquely identifies that triangle or edge. Lighting and blending are disabled so that the colors are preserved exactly.

After the ID reference image is prepared, all of its pixels are checked in one pass: when a pixel contains the ID of a triangle or edge, that pixel location is stored in a list on the patch that contains the triangle or edge. Later, the active texture of the patch can access the list of pixel locations in its main rendering loop. For example, the dithering texture simply runs the Floyd-Steinberg algorithm on the pixels of its patch.

The ID reference image can be used to determine the visibility of a point on a known triangle – the details of how to make this work robustly, even for triangles whose screen dimensions are less than a pixel, are beyond the scope of this paper (see [8]). If the triangle belongs to a patch of a convex surface, a simple test can be used: If the point (in screen space) is more than one pixel from the boundary of the patch, and also from any visible silhouette curve in the scene, then it is visible if and only if its triangle is front-facing and the value in the ID reference image at the point's screen position identifies a triangle of the same patch.

4 Graftal Textures

The textures described in this section place fur, leaves, grass or other geometric elements into the scene procedurally, usually to achieve a particular aesthetic effect (e.g., indicating fur at silhouettes but tending to omit it in interior surface regions). We'll call this class of textures *graftal textures*. They all share the same basic procedure for placing tufts, leaves, grass, etc., all of which we call *graftals*. The key requirements are that graftals be placed with controlled screen-space density in a manner matching the aesthetic requirements of the particular textures, but at the same time seem to "stick" to surfaces in the scene, providing interframe coherence and a sense of depth through parallax.

4.1 Placing graftals with the difference image algorithm

To meet these requirements, we have adapted the "difference image" algorithm (*DIA*) used by Salisbury *et al.* [14] to produce pen-and-ink-style drawing from grayscale images. Their algorithm controls the density of hatching strokes in order to match the gray tones of the target image. For each output stroke drawn, a blurred image of the stroke is subtracted from a "difference" image (initially the input image). The next output stroke is placed by searching in the difference image for that pixel most (proportionally) in need of darkening, and initiating a stroke there. The resulting image consists of marks whose density conveys the gray tones of the original.

The DIA meets our first requirement of placing marks (or in our case, graftals) with a controlled screen-space density. To control graftal placement according to a particular aesthetic requirement, each graftal texture simply draws its patch into the color reference image so that darker tones correspond to regions requiring a denser distribution of graftals. We call the result the *desire image*, and the value at a pixel in that image measures the *desire* that graftals be placed there. For example, to render the furry creature in figure 1, the reference image is drawn darker near silhouettes – easily done by placing a point light near the camera position. Also, some regions (e.g., the feet) can be explicitly darkened by the designer to promote a greater density of graftals there.[1]

To meet the requirement that graftals appear to stick to surfaces in the scene, we must convert the 2D screen position of a graftal (assigned to it by the DIA) to a 3D position on some surface. This is achieved in $O(1)$ time (per graftal) by using the ID reference image to find the triangle (and the exact point on the triangle with a ray-test) corresponding to a given screen position.

This now allows graftals to be distributed over surfaces in the scene to achieve a desired screen-space density – for a single frame. To create some interframe coherence, we modify the algorithm:

- In the first frame, graftals are placed according to the DIA.
- In each successive frame, the graftal texture first attempts to place the graftals from the preceding frame.
- Then, when all the "old" graftals have been considered for placement and accepted or rejected, the graftal texture executes the DIA to place new graftals into the scene as needed.

An existing graftal may fail to be placed in a frame for two reasons: (i) the graftal is not visible (it is occluded or off-screen); (ii) there is insufficient desire in the desire image at the graftal's screen position. This can happen if the original desire value at the graftal's screen position was small (e.g. the graftal is far from a silhouette).

[1] To further encourage drawing near silhouettes, we filter the desire image, replacing each desire value d with $2d - d^2$, where d ranges from 0 (no desire) to 1 (maximum desire).

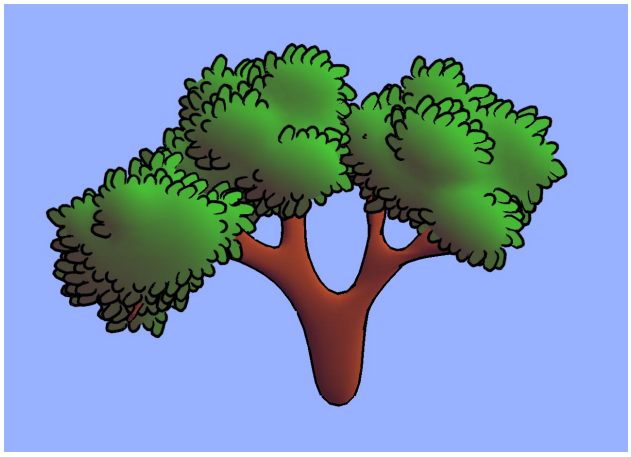


Figure 4 A tree rendered in the style of Geoffrey Hayes [7]. The leaves are drawn with graftals based on OpenGL triangle fans, rather than the triangle strip-based type of graftal shown in figure 5. The interior is shaded with the technical-illustrator shader of Gooch *et al.*[6].

It can also happen if the camera has zoomed out and the graftal's neighbors, now closer to it in screen space, have already subtracted the available "desire" in the vicinity. When a graftal fails to be placed in a frame, it is discarded. Otherwise it updates its attributes (as described below) and is drawn.

We make a final modification to the DIA: we use a bucket-sort data structure to find the pixel with the greatest desire. This key step can be completed in $O(1)$ time rather than the $O(\log(n))$ quad-tree method of Salisbury *et al.*, where n is the number of pixels.[2] This lets the algorithm run at interactive speeds on simple scenes.

4.2 Subtracting the blurred image

When a graftal is placed in the scene (either initially or in subsequent frames) it subtracts a blurred "image" of itself from the difference image. For this, graftals are treated as points with a given (variable) screen size, so the blurred image is just a Gaussian dot.

Pixels in the desire image are encoded with values ranging from zero (no desire) to one (maximum desire). Each graftal has an associated "volume" that determines how much total "desire" it subtracts from the desire image. This volume is proportional to the graftal's approximate screen space area. Intuitively, a visually large graftal subtracts a large volume, corresponding to a wide blurred dot: this eliminates desire in a wide region near the graftal, preventing others from being placed there.

Graftals can scale their geometry and volume so that they tend to maintain a desired screen-space size and relative density. For example, strictly adhering to the laws of perspective when zooming away from the model could result in graftals being drawn too small to be individually discernible. An artist might choose to draw them larger than they would realistically appear in this case. In any case, graftals that appear smaller in screen space should scale their volume accordingly in the DIA , or they will be placed too sparsely.

To perform such compensatory scaling, each graftal must keep track of its approximate screen space size. It does so by first converting its object-space length L to a screen-space measurement s in every frame (ignoring foreshortening). Then it chooses a scale factor r by which to multiply L as follows. As part of its definition, the graftal is given a desired screen space length d and corresponding volume v_0 (chosen by the user). At one extreme the graftal could take $r = d/s$, so that it always appears the same size on the screen regardless of distance. At the other extreme it could take $r = 1$, which would be strictly realistic. In our examples, we have taken a weighted average between the two extremes:

$$r = w(d/s) + (1 - w),$$

with weight $w = 0.25$. This approach moderates the degree to which the graftal scales with distance, providing a measure of resistance to change from its ideal size. Finally, the volume in each frame is calculated as $v = v_0(rs/d)^2$ to keep it proportional to the graftal's current screen size.

Now let d_0 be the value in the desire image at the graftal's screen position $\mathbf{x}_0$ (which has been verified as visible). Let $v \quad 0$ be the volume of the graftal. We seek a 2D Gaussian function g such that

$$g(\mathbf{0}) = d_0 \quad \text{and} \quad \int_{\infty}^{\infty}\int_{\infty}^{\infty} g(\mathbf{x})\,dxdy = v.$$

This is given by $g(\mathbf{x}) = d_0 e^{\quad d_0 \ \mathbf{x}^{\ 2} \ \ v}$. Thus, to subtract the "blurred image" at $\mathbf{x}_0$, we subtract $g(\mathbf{x} - \mathbf{x}_0)$ from each pixel location $\mathbf{x}$ in the desire image. Of course, outside of some radius values of g are negligible. If we take m to be the minimum value we can represent in the 8 bits we use to store desire, then this radius is $(v\log(2d_0/m)/(\pi d_0))^{1\ \ 2}$. We only subtract g from pixel locations within this radius of the graftal's screen position.

As the graftal subtracts this Gaussian from the desire image, it records the total desire subtracted. (It can't subtract more from a pixel than is stored there.) When all goes well, this quantity should equal the volume v (ignoring discretization errors and the small portion of the Gaussian outside the maximum radius above). If the total is less than v, the graftal may draw itself with a reduced level of detail. If the total is too low (below 0.5 in all our examples), the graftal reports failure to its texture and is removed from the scene. To avoid "popping" when graftals appear and disappear, they may initially be drawn with reduced detail, quickly increasing to full detail over a short time, and reversing the process when they are removed. This has its limitations, though, as we discuss below.

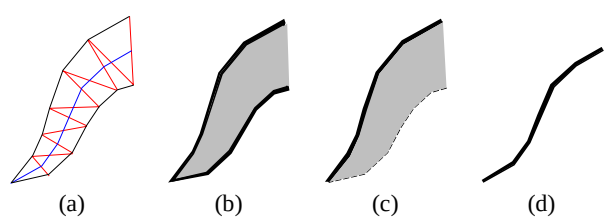


Figure 5 A fur graftal is based on a planar polyline and table of widths, used to construct a GL triangle strip (a). The graftal can render itself in three ways: It can draw a set of filled polygons with strokes along both borders (b) or just one (c); or it can draw just the spine (d).

4.3 Details of fur graftals

A fur graftal – the kind used for the furry creature in figure 1, and for the truffula tufts and grassy mounds at the base of the trees in figure 2 – is not particularly complex. It is based on a flat tapering shape by a gradually reducing width about a central spine (see figure 5). The central spine is a planar polyline, and the taper widths are recorded in an array. For the model in figure 1, just a few taper

[2] We thank Ken Lao for suggesting this idea.

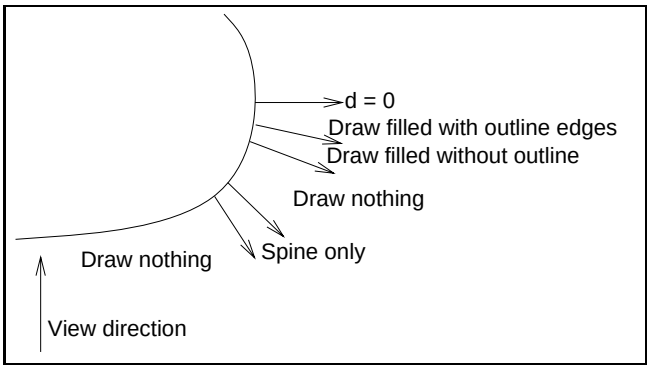


Figure 6 The dot product d of the view vector and surface normal determines a graftal's drawing style. In figure 1 the "Draw filled without outline" region is empty: we transition directly from "filled with outline" to "draw nothing."

widths were assigned; for the truffula tufts there were about seven. The shape of the central spine was drawn on graph paper and entered by hand.

After being placed with the DIA, each fur graftal determines how to orient and draw itself by computing the dot product d of the unit surface normal n at the graftal's position with the unit view vector v from the camera to its position (see figure 6). For varying values of d, the tuft may be drawn filled, filled with one edge, filled with both edges, as a spine only, or not at all. For the fur in figure 1, we draw just the spine for $-0.75 < d < -0.6$. We draw the filled tuft with both edges for $-0.55 < d < 0$. Other schemes are possible, and we believe that adjusting the thresholds and drawing styles during "fade-in" and "fade-out" might help smooth these transitions.

Finally, the fur graftals are oriented to face the camera – that is, to lie in the plane containing the underlying surface normal and most nearly orthogonal to the view vector. They're placed so that in general they bend down. This behavior can be modified (as in the truffula tufts) so that they point clockwise, or so that they follow directions that have been "painted" onto the graftal texture's patch, as in the feet in figure 1.

5 Results and Future Work

Our system can produce scenes that evoke a remarkable sense of complexity, in a style that's new to 3D graphics, and at interactive rates. Figures 2, 4 and 8 show the kinds of results that can be achieved with graftal textures. In each case the underlying geometry was simple to produce, yet the renderings have an expressiveness often lacking in computer graphics imagery. With our system, even the truffula scene can be rendered at several frames per second on a high-end PC.

The accompanying video[3] shows our system in action. It includes sequences captured in real-time and animation sequences rendered off-line and played back at significantly higher frame rates. The problem of poor frame-to-frame coherence stands out most noticeably in the latter case. Graftals that persist from frame to frame maintain geometric coherence (if we simply redistributed graftals at every frame, the flicker would be overwhelming); unfortunately, the DIA has no inherent interframe consistency, so it's easy for a graftal to be "crowded out" in one frame, replaced in the next, and so on, causing the flickering artifacts that are so noticeable in the video.

We have recently begun experimenting with some strategies to address this problem. One possibility is to make much greater use of fading and alpha blending when introducing graftals into the scene and taking them out. The degree to which this approach is usable depends quite a lot on the specific style being targeted. Fading in a large yellow truffula tuft outlined in black against a blue sky may be just as jarring as introducing it suddenly; but fading in semi-transparent blades of grass rendered with a watercolor-like effect over green terrain might seem perfectly acceptable.

One problem we have encountered in our early experiments with fading in tufts of fur like those in figure 1 occurs when all the tufts along a silhouette are newly introduced, and thus nearly transparent. The model then appears (briefly) to be missing its fur along that silhouette. A possible solution that we have not yet implemented is to maintain a separate population of tufts drawn on back-facing surfaces. This requires an auxiliary ID reference image prepared with front-facing triangles culled. It also requires two separate calls to our modified DIA each frame – one to place front-facing tufts, another to place back-facing ones. The point is that tufts emerging into view from behind a silhouette (as the object turns) would already be drawn and thus would not "pop in." Each frame might take twice as long to render – possibly a worthwhile trade-off if the resulting animations are significantly more watchable.

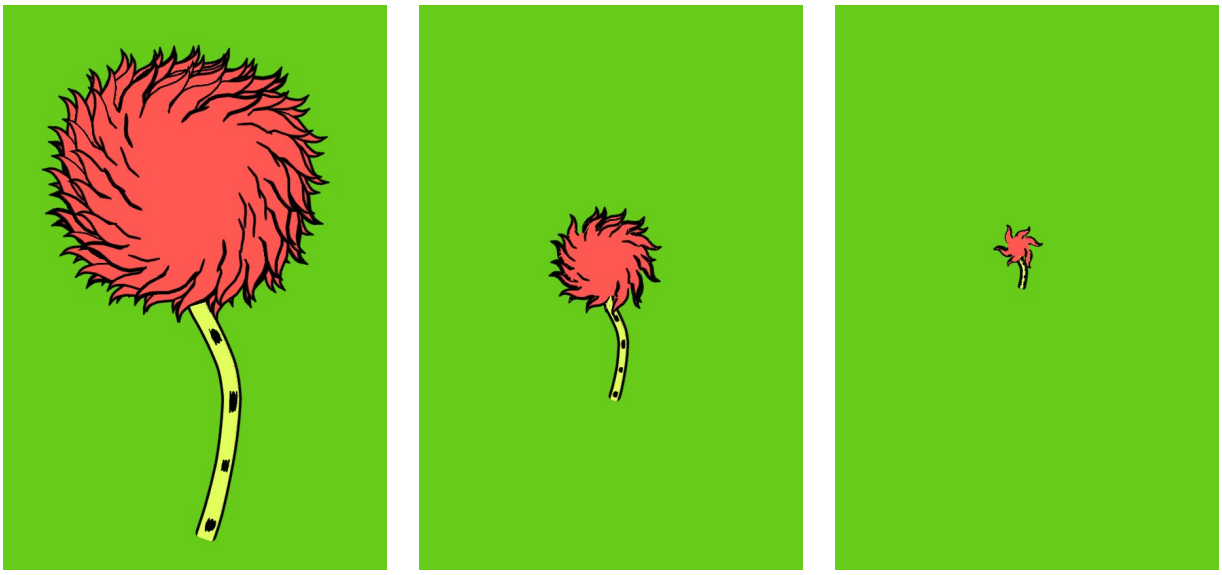
Another strategy we have experimented with is to use *static* graftals (see figure 7). With this approach, graftals are assigned fixed positions on the surface, rather than being generated each frame as needed. They still draw in a view-dependent way – those far from a silhouette, say, may not draw at all. This works quite well as long as the camera does not zoom out too far: in that case the graftals are drawn too densely in screen space. We can overcome this by assigning graftals several levels of priority – say numbered 0 through 2. Each level is distributed evenly over the surface, with those in a given level outnumbering those in the next lower level by a factor of about four. In a given frame, every graftal at level 0 "draws" itself view-dependently (possibly not at all if far from a silhouette). Each also subtracts its blurred image from the desire image, as in the DIA , and measures its success rate. If, collectively, this rate is high enough, the next level is given the chance to draw, and goes through the same procedure to decide whether the last level should also have the chance to draw. This strategy is suitable for localized objects - – we have tested it on versions of the truffula treetops – but not for landscapes where the choice of what level graftal to draw must vary over the surface according to distance from the camera. Our preliminary results indicate the effectiveness of this strategy. We demonstrate this in the accompanying videotape. Figure 7 shows a truffula tree top with static graftals drawn at three levels of detail.

A separate area for future work is to explore a range of new styles. For example, we have begun to explore rendering fur using two layers, one light and one dark, on a neutral background (see figure 8). This is a common technique in traditional drawing that can suggest more complex lighting effects (such as glossiness) while omitting unnecessary detail: nothing is drawn in midtone regions, which are represented by the unaltered background color.

6 Acknowledgments

We thank Michael LeGrand for creating the "furry creature" model, and Cynthia B. Rubin and Scott Klemmer for advice on preparing the images. Thanks also to Andy van Dam and the Graphics Group, and to our sponsors: the NSF Graphics and Visualization Center, Advanced Network and Services, Alias/Wavefront, Autodesk, IBM, Intel, Microsoft, National Tele-Immersion Initiative, Sun Microsystems, and TACO. Lee Markosian received support for his graduate education from Intel through the Intel Foundation Ph.D. Fellowship program.

[3] See the Siggraph 99 Conference Proceedings Video Tape.



(a) (b) (c)

Figure 7 A truffula treetop with static graftals organized in a three-level hierarchy. When the camera is close, all three levels are drawn (a). As the camera zooms out, only two levels are drawn (b), and finally just the base level is drawn (c).

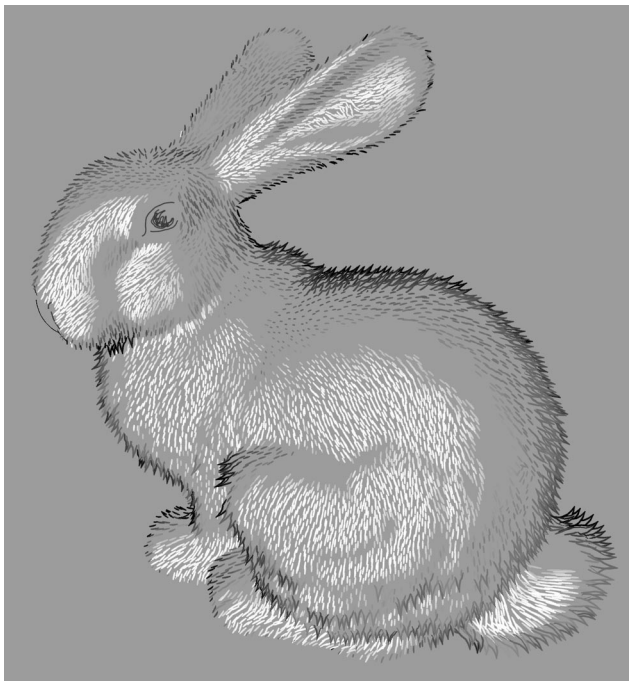


Figure 8 Fur rendered in light and dark layers on a neutral background. Model courtesy of Stanford University.

References

[1] Norman I. Badler and Andrew S. Glassner. 3D object modeling. In *SIGGRAPH 97 Introduction to Computer Graphics Course Notes*. ACM SIGGRAPH, August 1997.

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual, 2nd Edition*. Addison-Wesley Developers Press, 1996.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1992.

[4] Dr. Seuss (Theodor Geisel). *The Lorax*. Random House, New York, 1971.

[5] Dr. Seuss (Theodor Geisel). *The Foot Book*. Random House, New York, 1988.

[6] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH 98 Conference Proceedings*, pp. 447–452. ACM SIGGRAPH, July 1998.

[7] Geoffrey Hayes. *Patrick and Ted*. Scholastic, Inc., New York, 1984.

[8] Lee Markosian. *Art-based Modeling and Rendering for Computer Graphics*. PhD thesis, Brown University, November 1999 (expected completion).

[9] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John Hughes. Skin: A constructive approach to modeling free-form shapes. In *SIGGRAPH 99 Conference Proceedings*. ACM SIGGRAPH, August 1999.

[10] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. In *SIGGRAPH 97 Conference Proceedings*, pp. 415–420. ACM SIGGRAPH, August 1997.

[11] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH 96 Conference Proceedings*, pp. 477–484. ACM SIGGRAPH, August 1996.

[12] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[13] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH 85 Conference Proceedings*, pp. 313–322. ACM SIGGRAPH, July 1985.

[14] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *SIGGRAPH 97 Conference Proceedings*, pp. 401–406. ACM SIGGRAPH, August 1997.

[15] Alvy Ray Smith. Plants, fractals and formal languages. In *SIGGRAPH 84 Conference Proceedings*, pp. 1–10. ACM SIGGRAPH, July 1984.

[16] T. Strothotte, B. Preim, A. Raab, J. Schumann, and D. R. Forsey. How to render frames and influence people. In *Computer Graphics Forum*, volume 13, pp. 455–466. Eurographics, Basil Blackwell Ltd, 1994. Eurographics '94 Conference issue.

[17] Georges Winkenbach and David H. Salesin. Computer–generated pen–and–ink illustration. In *SIGGRAPH 94 Conference Proceedings*, pp. 91–100. ACM SIGGRAPH, July 1994.

To appear in the proceedings of NPAR 2000

Artistic Silhouettes: A Hybrid Approach

J.D. Northrup and Lee Markosian

Brown University, Providence, RI 02912

Abstract

We present a new algorithm for rendering silhouette outlines of 3D polygonal meshes with stylized strokes. Rather than use silhouette edges of the model directly as the basis for drawing strokes, we first process the edges in image space to create long, connected paths corresponding to visible portions of silhouettes. The resulting paths have the precision of object-space edges, but avoid the unwanted zig-zagging and inconsistent visibility of raw silhouette edges. Our hybrid screen/object space approach thus allows us to apply stylizations to strokes that follow the visual silhouettes of an object. We describe details of our OpenGL-based stylized strokes that can resemble natural media, but render at interactive rates. We demonstrate our technique with the accompanying still images and animations rendered with our technique.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms

Additional Key Words: Strokes, non-photorealistic rendering

1 Introduction

The outline, or silhouette, of a shape is often one of its most striking features. Our work attempts to render attractive silhouette outlines for 3D geometry in real-time, creating brush-strokes resembling natural media along well-chosen paths around each object. This breaks down into three distinct phases. First, we must determine where the silhouette edges are. Because silhouettes are inherently view-dependent, we have to find them every time the scene or camera changes. Second, we need to choose where we want to place each stroke. We want to pick paths that will look good and remain as consistent as possible from one frame to the next. Third, we draw each stroke in a style defined by the user. This last step should be flexible enough to enable the user to achieve almost any artistic goals she may have in mind.

In Section 2, we review several known silhouette detection algorithms to address the first phase. The remainder of this paper focuses on the second two phases. Section 5 describes our algorithm for determining stroke paths, and Section 6 explains our artistic stroke-rendering framework for producing final images.

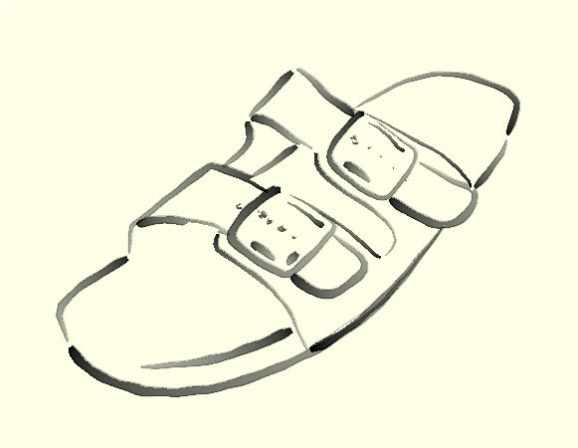


Figure 1 An example of an ink-wash style rendered with our algorithm.

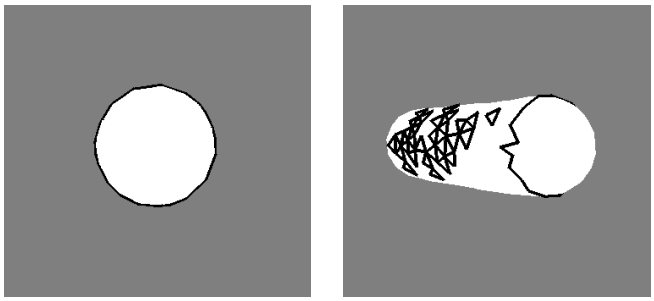
2 Silhouette Edge Detection

Our system begins with 3D scenes consisting of standard triangle meshes. We need to analyze the structure of these models to determine which edges form silhouette outlines. We define a **silhouette edge** to be an edge that connects a front-facing triangle to a back-facing triangle. Because this condition depends both on the camera viewpoint and the state of the model, we must compute these silhouette edges every frame that the world changes.

The brute-force approach to finding the silhouette edges simply checks every edge of the mesh, every frame. This may suffice for high-quality, non-interactive animations which can afford to sacrifice speed for guaranteed results, but causes a major bottleneck in real-time applications like ours.

Several algorithms for rapidly finding the silhouette edges already exist. For a model with n edges and k silhouette edges, the method used by Gooch *et al.* [5] performs the detection of all silhouette edges in $O(k \log n)$ by precomputing a spherical hierarchy data structure, but their algorithm is rather complicated and difficult to implement. Furthermore, reliance on expensive pre-computation makes this algorithm inappropriate for use with changing meshes.

We use the randomized algorithm presented by Markosian *et al.* [11]. Taking advantage of temporal coherence, this algorithm uses the silhouettes found in the previous frame as a starting point for a search of the current frame. For a mesh with n edges, we randomly select a small fraction of edges to test. When a new silhouette is found, its neighbors are also checked for local continuation of the silhouette contour, leveraging the spatial coherence of silhouettes. Further details are supplied in [10]. In our experience, this algorithm has proven efficient, simple to code, and robust enough for our real-time applications.



(a) (b)

Figure 2 Current algorithms render the innocent-looking silhouette of (a) by drawing the complicated mess of overlapping edges shown in (b).

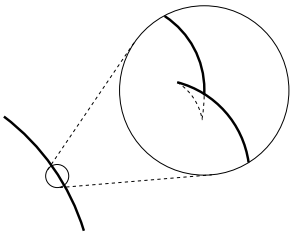


Figure 3 Small zig-zags called swallowtails often occur along silhouette profiles.

3 Problems Rendering Silhouette Edges

The second phase should render the raw silhouette edges found in the previous step in some pleasing way. One straightforward approach for implementations using a traditional API such as OpenGL [1] is to turn on *z*-buffering and render the edges as line strips. This is simple and fast, but does not allow stylization along the silhouettes. When using the *z*-buffer, anything drawn at the silhouette edges will be clipped against the body of the mesh. Ideally, we would disable the *z*-buffer and somehow draw strokes only along edges which are already known to be visible.

Furthermore, a few well-placed strokes often express the shape of a figure much more elegantly than a crowd of shorter marks. With this in mind, we seek to simplify the silhouette edges extracted from our models into paths that we can use to draw long, smooth strokes. Working purely in object space, it's hard to determine where these paths overlap, so we end up over-drawing the silhouette, as shown in Figure 2.

Also, the silhouette edges are connected to each other in ways that often reflect the small-scale structure of the mesh—not the overall shape of the object's outline. Paths created by joining up silhouette edges will contain small but frequent zig-zags that can cause unwanted artifacts when rendering them as strokes. These zig-zags occur at intersections called swallowtails where the silhouette temporarily reverses direction to connect two overlapping edges. This phenomenon is illustrated in Figure 3.

4 Image-Based Solutions

One class of algorithms addresses the problems inherent to working with silhouette edges by ignoring them altogether. Image-based silhouette-rendering algorithms avoid explicitly finding the 3D silhouette edges and instead opt to use 2D image-processing techniques. Notably, the work of Saito and Takahashi [16] renders the outlines of 3D objects by applying edge-detection filters to specially prepared depth and normal maps, and compositing the results with the rest of the scene. However, such approaches suffer from aliasing as the silhouette positions jump from pixel to pixel in the image, because the silhouette positions are not accurately tied to the underlying geometry. Furthermore, these algorithms do not easily allow the use of stylized strokes. Curtis [4] has introduced a technique for generating strokes along these pixel outlines, but using only pixel data sacrifices precision, especially when trying to decide how to join several intersecting curves. Also, techniques requiring multipass filtering and compositing are often too slow for real-time applications. Raskar and Cohen [15] present a geometry-based approach suitable for real-time use which also avoids explicitly finding silhouette edges, but their methods do not allow for stylized strokes.

These image-based techniques inherently focus on depicting only the portions of the silhouettes that contribute to the final 2D image. Like any good artist, these algorithms never even *consider* explicitly depicting every single silhouette edge, so they avoid the problems due to overlap and sub-pixel swallowtails. Furthermore, they do not need *z*-buffering because the visibility is already known, allowing systems like Curtis's to apply stylizations without worrying about strokes being clipped.

5 A Hybrid Algorithm

Our work combines the benefits of the image-based approach with the accuracy of a geometry-based approach. As in the latter, we begin by detecting the silhouette edges of the model, but then we compute visibility and adjacency using a 2D projection of the silhouette edges. This lets us maintain the precision of object coordinates while still working in 2D where that makes most sense. At a high level, the algorithm proceeds as follows:

1. Find silhouette edges.
2. Determine visible segments of each edge.
3. Apply correction for overlaps in segments.
4. Link segments into smooth paths.
5. Render stylized strokes along these paths.

In what follows, the term **edge** refers to a silhouette edge of the mesh; a single visible portion of an edge (such as found in step 2 above) is called a **segment**; and a collection of segments that form a continuous sequence in image space is called a **path**.

Step 1 above is discussed in Section 2. In the remainder of this section we describe steps 2 through 4. The final step is described in Section 6.

5.1 Extracting Visible Segments

Once we have detected the silhouette edges in the current frame, the next step is to determine which portions of the silhouette edges are visible. Our method makes use of the "ID reference image" discussed in a previous paper [9]. Briefly, we create the ID reference image by first rendering the scene with each silhouette edge and each mesh triangle drawn in a color that uniquely identifies it, then reading this image from the framebuffer into memory. Details on how to use the ID reference image to determine visibility can be found in [10].

We next iterate over all the pixels in the reference image and build a list L of edges that contributed at least one pixel. We thus remove from consideration silhouette edges that make no contribution to the pixels of the current frame (such as many of the silhouette edges seen in Figure 2 (b)).

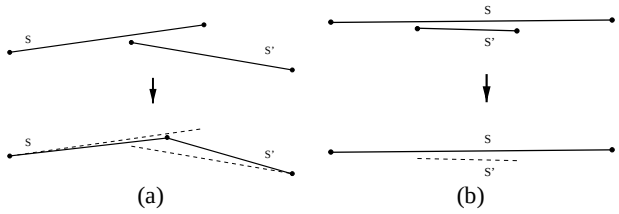


Figure 4 In (a), segments S and S' are corrected by redefining their overlapped endpoints to midpoint between them. The angle between the segments is exaggerated for clarity. An undesired segment is eliminated in (b).

Next we scan-convert along each edge in L to determine which portions of it show up in the ID reference image, and hence are visible.[1] We record each such visible portion, or *segment*. A segment consists of its two image-space endpoints and a pointer to the associated edge. The endpoints need not project exactly onto the edge; for example, a segment may extend across a pixel even when its associated edge occupies just a small fraction of the pixel in the current view.

To determine whether an edge e "shows up" in the ID reference image at a given image space point x on e, we check the ID reference image at x, and also at nearby points (within two pixels in practice) along the image-space line perpendicular to e and passing through x. If e shows up anywhere along this line, we consider e to be visible at x. In addition, we record a list of neighboring edges encountered along this line for use in the next step of the algorithm.

5.2 Correcting For Overlaps

The segments are intended to be linked together to form long, connected image-space paths that will serve as the basis for stylized strokes. Before we perform this linking step, we first carry out two correction steps that promote longer and smoother paths. These are shown in Figure 4.

We "merge" segments that overlap and are nearly parallel; and we eliminate a segment if it is adjacent and nearly parallel to another segment, and the first segment is shorter. In both cases, we consider two edges nearly parallel if the angle between them is less than 1 degree.

5.3 Linking Segments into Paths

At this point we have a collection of segments that together closely approximate the visible silhouettes of the scene. The next step is to link these segments into long chains, or paths, that will form the basis for the strokes. To do this, we first search near each segment's endpoints for potential neighbors. The search is a $n \times n$-pixel local search in the reference image.[2] We perform a series of tests to compute the suitability of each potential match between segment s and neighbor n, outlined in Figure 5.

Whenever we link a pair together, we keep a list of "divorced" segments, i.e., any segments that the new pair had previously been linked to. Once we have tried to find neighbors for each edge, we allow these divorced segments another chance at linking up.

[1] It can happen that a visible portion of an edge does *not* show up in the ID reference image. This might happen for example if the edge is too small to contribute to the rasterization of the image. For our purposes, it is sufficient to find just those visible portions that appear in the ID reference image.

[2] In our system, $n = 3$.

```
CHECK-MATCH(s, n)
    θ      angle between s and n
    θmax   max angle allowed to link
    D      distance between endpoints of s and n
    Dmax   max distance allowed to link

    if s is already linked to n
        reject n
    if θ    θmax
        reject n
    if endpoints of s and n don't overlap
    and D    Dmax
    and θ    angle of s's current neighbor (if any)
    and θ    angle of n's current neighbor (if any)
        link s and n
```

Figure 5 The CHECK-MATCH function determines whether segments s and n are suitable for being linked to each other. We use $\theta_{max} = 45$ and $D_{max} = 2$ pixels.

5.4 Rendering Paths

The final phase of our algorithm renders each newly-created path using an "artistic stroke." These strokes are defined in image space, and the visibility of the silhouettes they represent is already assured, so we can disable depth testing and safely draw the strokes with various image-space stylizations. Enabling depth testing would preclude the use of such stylizations, since we cannot reliably assign depth values to parts of the stroke affected by the stylization. We now go into more detail about how these strokes are built and displayed.

6 Fast Artistic Strokes for 3D Scenes

Ultimately the success of any art-based rendering system depends on producing appealing images. Thus the final step of our algorithm takes great care to allow for a wide range of expressive strokes. To convincingly mimic traditional 2D illustration, our strokes should appear foremost as marks on a flat drawing surface—not objects floating in 3D space. From this vantage point, we may choose to selectively reintroduce hints of depth and distance, which we discuss below in Section 6.2.3.

On a practical level, this goal necessitates using a coordinate system in which stroke proportions will reflect screen distance, rather than 3D world distance. We use a variant of what the OpenGL reference manual labels "device coordinates" [1], normalized to preserve the aspect ratio of the drawing area. In other words, our coordinates range from -1 to 1 across the smaller screen dimension, and from $-d$ to d across the larger dimension, where d is the aspect ratio of the window. Unlike a pure screen-space coordinate system, our points retain their z component, allowing us to render strokes using traditional depth buffering if desired. This z-buffered approach is used to render the strokes outlining the graftals of Kowalski *et al.* [9, 12], and the strokes of Cohen, Zeleznik and Hughes' user-drawn world [3]. For the algorithm described in this paper, we only place strokes where we already know they will be visible, so we can draw without z-buffering.

6.1 Creating Basic Strokes

Given a list of vertex/width pairs, we would like to render a stroke that passes through each vertex, smoothly transitions between the given widths, and joins corners to create a continuous path. Unfortunately OpenGL does not support lines of varying width and leaves large gaps between corners of thick line strips. These problems led

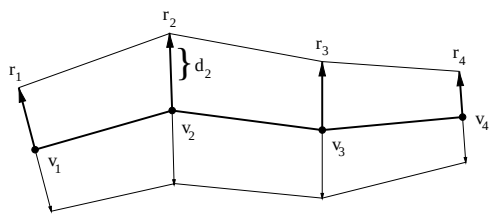


Figure 6 Constructing rib vectors $\vec{r_i}$ to add width to a four-vertex stroke.

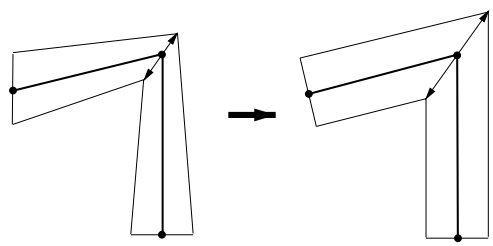


Figure 7 Effect of scaling rib vector to maintain constant path width.

us to create a variable-width line primitive by leveraging OpenGL's speed and flexibility at drawing long triangle strips. We proceed with a method inspired by the work of Hsu *et al.* [7, 8].

For each path vertex v_i and its corresponding pixel width w_i, we generate a vector "rib" r_i along the angle bisector which allows us to give breadth to the stroke. This is illustrated in Figure 6. Figure 7 demonstrates the need to scale the length of these ribs to maintain a desired path width. The scale factor for each rib is computed as

$$f = \left|\frac{|r_i|}{r_i \cdot n_i}\right|$$

where n_i is the normalized vector perpendicular to the the path direction between v_i and v_{i+1}. Intuitively, this means that the ribs are scaled wider at sharper corners, and left alone at straight segments. We limit the amount of scaling to a factor of 2, since otherwise the miter for a very sharp corner would be too large.

Once we have a suitable set of ribs, we render the stroke as a series of triangle strips. Point pairs for the strips are generated by offsetting each path vertex v_i by its corresponding rib vector r_i, and its opposite $-r_i$. The end result of this phase is a variable-width line strip with nicely joined corners. Now we need to liven up the strokes by adding artistic effects.

6.2 Stylistic Variations

In our system, the designer decides on a combination of stylization operations to apply to the strokes for a given object. All of the style options can be freely mixed and matched, allowing for a wide range of expressiveness. Figure 8 shows how each operation affects a stroke. Since each stroke must be rebuilt every frame, we have tried to make sure that these operations add little overhead to the overall stroke building and rendering burden. We will now describe three categories of stylizations possible with our system.

6.2.1 Resolution-Dependent Stylizations

For the first type of stylization, we perturb the appearance of the stroke along its length. To do this, we need to ensure that the screen distance between adjacent vertices does not exceed some specified maximum (in practice, 2-3 pixels). Ideally, we would resample the stroke to have perfectly even segments using an interpolating spline curve. In practice, we linearly divide each individual path segment,

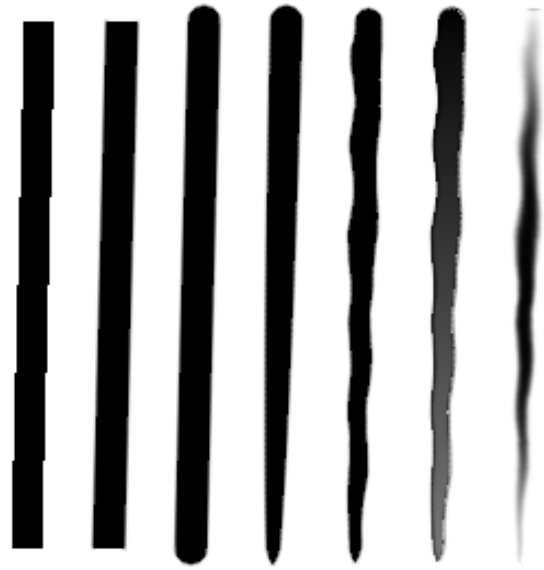


Figure 8 The cumulative effects of adding stroke operations, from left to right: raw stroke, antialiasing, taper, flare, wiggle, alpha fade, and texture-mapping.

adding evenly-spaced vertices to approximate the desired overall spacing. This leaves the original shape of the path undisturbed, and keeps this phase quick.

Once we have a sufficiently fine sampling of stroke vertices, we can perturb their locations and widths to achieve an uneven, hand-drawn look. Our current system applies offsets created by the user with the help of a separate tool. This allows our strokes to reflect the individual character of the user's lines. These strokes are similar to the ones described in [11].

Another style adds an alpha fade in which we linearly increase the transparency along the length of each stroke, creating a simple watercolor or ink-wash feel, as seen in Figure 1. One other operation flares the overall width of the stroke from end-to-end to create a brush-stroke shape. The flare function we use is

$$f = \sqrt{1 - t^2} \qquad \text{where} \qquad t = \frac{\text{current vertex index}}{\text{max vertex index}}$$

6.2.2 Other Stylizations

The second category of operations does not require fixed spacing. These include antialiasing, tapering the ends of each stroke to a point, and applying texture maps along the length of the stroke. Our implementations of OpenGL only support one antialiased primitive: the 1-pixel-wide line. Luckily, we can use this to simulate smoothed triangle strips by placing an antialiased line-strip around the boundary of the stroke. This adds virtually no overhead, and we can know that the antialiasing will register correctly because the line-strips use the exact same coordinates as the body of the stroke.

Tapering the ends of strokes is a bit more complex, but this is an important effect to simulate. First, we ensure a sufficient distribution of vertices to achieve a gradual thinning near the ends of each stroke. We insert vertices along the beginning and ending segments, linearly interpolated in the manner mentioned above. If we are using any of the operations from the previous section, we have already performed an overall interpolation, and may skip this step. Next we scale the rib size of these taper vertices by a function similar to the

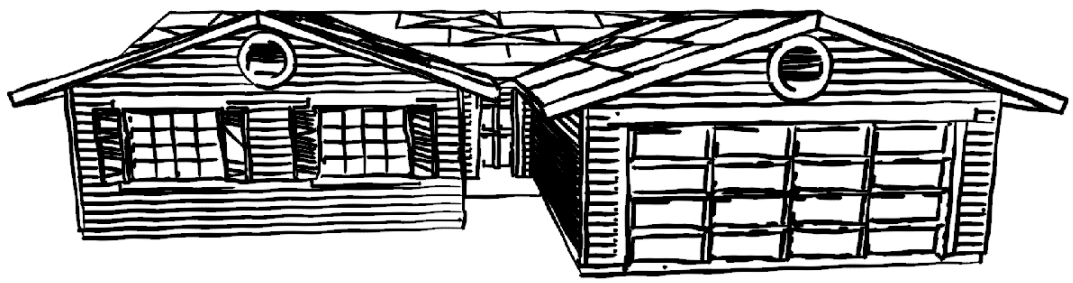


Figure 9 A simple architectural rendering that uses wiggle, flare, and tapering.

flare function given above. The result is a smoothly rounded ending for each stroke.

A second style of tapering adds vertices *beyond* the endpoints of the stroke using the direction of the final segment. This creates a rounded extension to the stroke that can be used to hide seams at stroke boundaries. This second taper style is used to render all of the examples in this paper, and addresses the same problem treated by Gooch *et al.* [6], who used fat dots to hide the problem.

One final effect is to stretch a texture map over the length of the stroke. Smoothly faded strokes are easy to draw using this method. We simply apply textures with large regions of transparency (see Figure 12).

6.2.3 Depth and Distance Cue

Often traditional illustration styles give perspective cues by decreasing line weight for distant objects. We mimic this effect by modifying line widths based on two scaling factors: distance cue and depth cue.

The first modifies the overall stroke width based on the distance to an object. As the object recedes, the silhouettes gradually thin; as the object approaches the camera, the silhouettes widen. When the silhouette-rendering algorithm is first applied to a mesh, we compute the value D_i, which is the initial ratio of object-space length to screen-space length at the origin of the mesh. In successive frames, we compute the current scaling factor D_c, and multiply the scaling factor

$$f_D = \overline{\frac{D_c}{D_i}}$$

with the width of each stroke used to render that mesh. This is a non-linear scale in order to soften the effects of the distance cue.

The second pass varies the width of the strokes as the depth *within an object* varies. This provides a simple cue to the foreshortening of different parts of the mesh, similar to an effect demonstrated by Gooch [6]. Every frame, we compute the frame-buffer z-value bounds of each mesh, z_{min} and z_{max}. We would like to scale the width of each stroke vertex v based on its depth, z_v, so that the foremost vertices are scaled by a factor $1 + S$, and the rearmost vertices by $1 - S$. We compute the scale f_v for vertex v as

$$f_v = \max(0, 1 + S\frac{z_{max} + z_{min} - 2z_v}{z_{max} - z_{min}}) \qquad \text{where} \qquad 0 \quad S \quad 1$$

and multiply the width of each vertex by it. Unlike f_D, this f_v varies linearly with depth to enhance the localized, intra-object perspective hint.

7 Discussion

In the end, the success of any non-photorealistic rendering system rests on the quality of its final rendered images. We find our view-dependent stroke framework robust, fast, and very flexible for creating effective real-time illustrations. Similar uses of tapering artistic strokes can be found in the pen-and-ink work of Salisbury, Winkenbach, Salesin and others [17, 18], and previous work here at Brown University [11, 9, 3].

The results of using our artistic stroke renderer to depict the paths found by our silhouette-extraction algorithm are shown in Figures 1 and 9 - 12. Figures 11 and 12 can be found in the Color Plates section. These examples range from simple outlines to highly stylized brushwork, giving an idea of the flexibility offered.

Our system is fast enough for interactive use. On a high end Sun workstation, our frame rate varies from about 2 fps for scenes like the house in Figure 9 with many silhouettes, to 10 fps for simpler models like the frog hand or the sandal (Figures 10 and 1). Antialiasing our stroke primitives greatly enhances image quality, especially noticeable in frame-by-frame renderings (see the accompanying animations for examples). However, animations of the scene in Figure 12 reveal small, unwanted strokes that sometimes appear in regions of negative curvature. In these cases, our visibility and placement algorithms have decided to render edges that we find unattractive. It is possible that we could create additional tests to filter out these edges, but this is left as future work.

Another limitation of our algorithm is that it does not make any attempt to remember where it drew silhouettes in previous frames, much less how it parameterized the stylizations used in those strokes. This was a conscious decision in order to simplify and speed our algorithm. We gain some degree of coherence from our reliance on mesh structure, and provide line styles which hide inconsistencies between frames. For styles with uniform line character, such as in Figure 10, we can produce smooth, temporally-coherent animations, but for animations of styles in which each stroke is more noticeably varied, such as in Figure 12, lack of inter-frame coherence is often distracting. For high frame-rates, this randomness becomes much more noticeable, but in some cases it can become a desirable aesthetic effect [4].

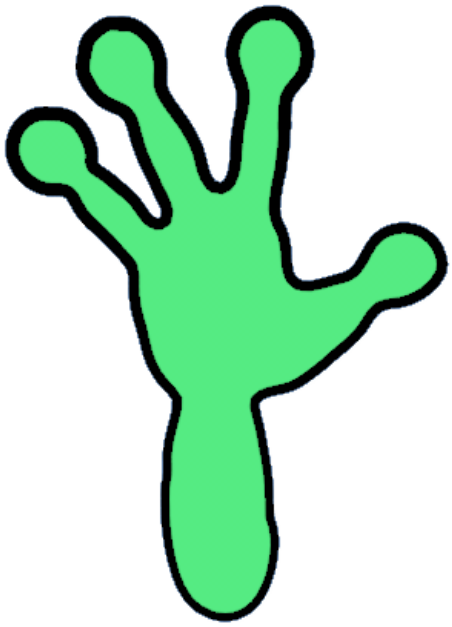


Figure 10 A frog hand. Stroke width varies as a depth cue.

8 Future Work

We know of related algorithms that have attempted to provide control over inter-frame coherence, notably the work of Bourdev [2], and Masuch *et al.* [13, 14]. One possible direction for future work would integrate the control afforded by these algorithms with the aesthetic details of our present work.

One of the advantages of our algorithm is that it works for arbitrary closed meshes, but often we would like to include detail that has been previously annotated by a user. We cannot expect an algorithm to choose the "right" strokes to draw a scene because there is no way to decide what those strokes would be. For a realistic renderer, there *is* a verifiable standard of what the right answer should look like: the real world.

However, a non-photorealistic/art-based rendering system should not impose the standards of the "real world" on our images. Like traditional artistic media, such a system should allow the user the possibility of expressing anything she wants, in any way she wants. It is our hope that the work presented in this paper provides tools that will help realize this goal, but much work still needs to be done before an artist will feel at home using such a system.

9 Acknowledgements

We thank Jun Ohya and Ryohei Nakatsu of ATR Labs for invaluable advice and support. This work is supported in part by the NSF STC for Computer Graphics and Scientific Visualization, Adobe, Advanced Network and Services, Alias/Wavefront, Department of Energy, IBM, Intel, Microsoft, National Tele-Immersion Initiative, Sun Microsystems, and TACO.

References

[1] OpenGL Architecture Review Board. *OpenGL Reference Manual, 2nd Edition*. Addison-Wesley Developers Press, 1996.

[2] Lubomir D. Bourdev. Rendering Nonphotorealistic Strokes with Temporal and Arc-Length Coherence. Master's thesis, Brown University, May 1998.

[3] Jonathan M. Cohen, John F. Hughes, and Robert C. Zeleznik. Harold: A World Made of Drawings. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. Held in Annecy, France.

[4] Cassidy Curtis. Loose and Sketchy Animation. Siggraph '98 Technical Sketch, 1998.

[5] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. *Proceedings of SIGGRAPH 98*, pages 447–452, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[6] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Rich Riesenfeld. Interactive Technical Illustration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999. ISBN 1-58113-082-1.

[7] S. C. Hsu, I. H. H. Lee, and H. E. Wiseman. Skeletal Strokes. In *Proceedings of UIST '93*, pages 197–206, November 1993.

[8] Siu Chi Hsu and Irene H. H. Lee. Drawing and Animation Using Skeletal Strokes. *Proceedings of SIGGRAPH 94*, pages 109–118, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[9] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-Based Rendering of Fur, Grass, and Trees. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[10] Lee Markosian. *Art-based Modeling and Rendering for Computer Graphics*. PhD thesis, Brown University, May 2000.

[11] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. *Proceedings of SIGGRAPH 97*, pages 415–420, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[12] Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-based Rendering with Continuous Levels of Detail. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. Held in Annecy, France.

[13] Maic Masuch, Stefan Schlechtweg, and Bert Schönwälder. daLi! - Drawing Animated Lines! In *Proceedings of Simulation und Animation '97*, pages 87–96. SCS Europe, 1997.

[14] Maic Masuch, Lars Schuhmann, and Stefan Schlechtweg. Frame-To-Frame-Coherent Line Drawings for Illustrated Purposes. In *Proceedings of Simulation und Visualisierung '98*, pages 101–112. SCS Europe, 1998.

[15] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999. ISBN 1-58113-082-1.

[16] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):197–206, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[17] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen-And-Ink Illustration. *Proceedings of SIGGRAPH 94*, pages 101–108, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[18] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

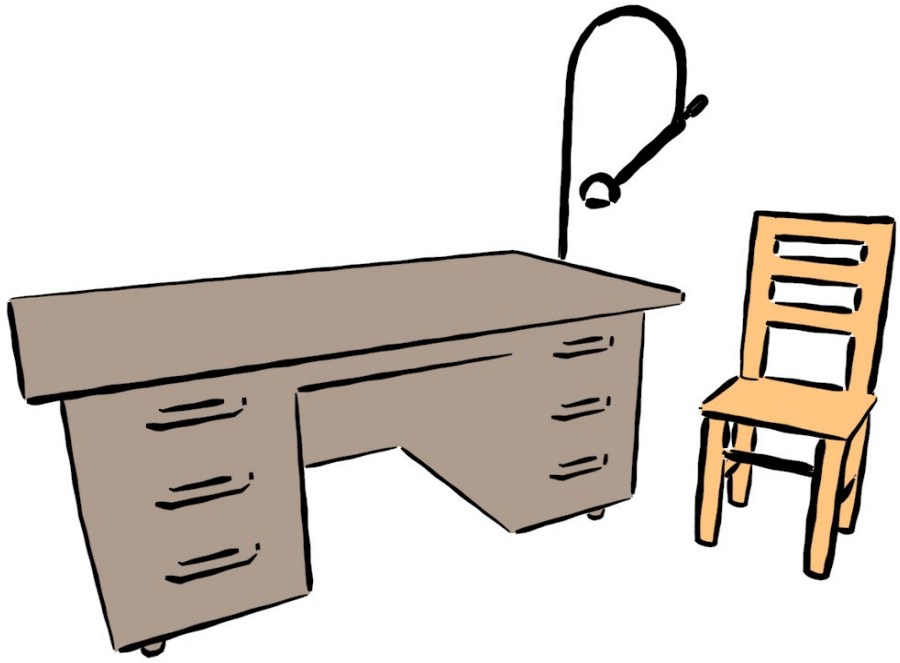


Figure 11 A simple office scene.

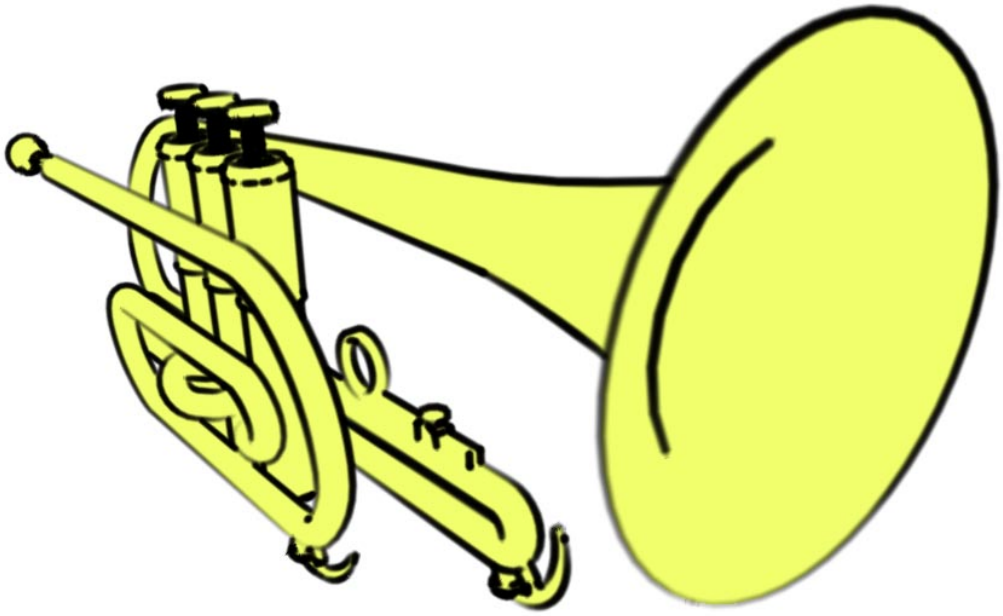


Figure 12 This trumpet demonstrates the use of texture-mapping to create soft strokes.

Interactive Pen-and-Ink Illustration

Mike Salisbury, Sean Anderson, Ronen Barzel*, David Salesin
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

*Pixar
1001 West Cutting Boulevard
Richmond, California 94804

Technical Report 94-01-07b
April 1994

Interactive Pen-and-Ink Illustration

Michael P. Salisbury *Sean E. Anderson* *Ronen Barzel*[*] *David H. Salesin*

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

[*] Pixar
1001 West Cutting Blvd
Richmond, California 94804

Abstract

We present an interactive system for creating pen-and-ink illustrations. The system uses *stroke textures*—collections of strokes arranged in different patterns—to generate texture and tone. The user "paints" with a desired stroke texture to achieve a desired tone, and the computer draws all of the individual strokes.

The system includes support for using scanned or rendered images for reference to provide the user with guides for outline and tone. By following these guides closely, the illustration system can be used for interactive digital halftoning, in which stroke textures are applied to convey details that would otherwise be lost in this black-and-white medium.

By removing the burden of placing individual strokes from the user, the illustration system makes it possible to create fine stroke work with a purely mouse-based interface. Thus, this approach holds promise for bringing high-quality black-and-white illustration to the world of personal computing and desktop publishing.

CR Categories and Subject Descriptors: I.3.2 [Computer Graphics]: Picture/Image Generation - Display algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques - Interaction techniques; I.4.3 [Image Processing]: Enhancement.

Additional Key Words: Comprehensible rendering, non-photorealistic rendering, prioritized stroke textures.

1 Introduction

Pen-and-ink is an extremely limited medium, allowing only individual monochromatic strokes of the pen. However, despite the limitations of the medium, beautiful pen-and-ink illustrations incorporating a wealth of textures, tones, and styles can be created by skilled artists. Indeed, partly because of their simplicity and economy, pen-and-ink illustrations are widely used in textbooks, repair manuals, advertising, and many other forms of printed media.

Part of the appeal and utility of pen-and-ink illustrations is that they can be easily printed alongside text, using the same ink on the same paper, without any degradation. For the same reasons, pen-and-ink-style illustrations could also be useful in the domain of desktop publishing and laser printers—especially if the illustrations were generated and manipulated directly on a computer.

While the problem of painting full-color images on a computer has received considerable attention in the computer graphics community, the requirements of an interactive pen-and-ink-style illustration system are different enough to merit special study. Pen-and-ink illustrations have two major properties that distinguish them from other art media:

1. *Every stroke contributes both tone (darkness) and texture.* Since tone and texture are not independent parameters, the pen artist must take care to convey both of these qualities simultaneously.

2. *Strokes work collectively.* In general, no single stroke is of critical importance; instead, strokes work together to express tone and texture.

This paper describes an interactive pen-and-ink-style illustration system. The overall goal of the system is to enable a user to easily generate effective and attractive illustrations directly on a computer. In this work, we are not concerned with creating purely computer-generated images; rather, the computer is utilized as a tool to enhance the speed and ease with which a user can create illustrations.

The interactive illustration system allows a variety of texturing in order to achieve the same range of style and expressive ability that is possible with a physical pen and ink. We do not want to limit the user to any specific algorithmic "look."

The system places a particular emphasis on using continuous-tone images as a reference for the user, and thus provides a form of "interactive digital halftoning" in which the user can introduce texture as an integral part of the resulting illustration. In this sense, the visual artifacts that are necessarily produced in quantizing a greyscale image can be given an artistic or expressive nature. Also, of practical significance, photocopying does not degrade pen-and-ink-style images to the same extent as conventionally-halftoned images.

1.1 Background: Pen-and-ink illustration

We give here a brief description of some of the salient features and terminology of hand-drawn pen illustration, relevant to the design of an interactive system. For further discussion and instruction, interested readers should consult Guptill [6], a comprehensive text on pen and ink illustration. In addition, Simmons [16] provides instruction on illustrating using a "technical pen," which draws strokes of constant width. Both books contain dozens of stunning examples. A discussion of pen-and-ink principles as they relate to purely computer-generated imagery can be found in Winkenbach et al. [20].

Because texture in an illustration is the collective result of many pen strokes, each individual stroke is not critical and need not be drawn precisely. Indeed, a certain amount of irregularity in each stroke is

desirable to keep the resulting texture from appearing too rigid or mechanical.

The most commonly used textures include: *hatching*, formed by roughly parallel lines; *cross-hatching*, formed by overlapped hatching in several directions; and *stippling*, formed by small dots or very short lines. Textures can also be wavy, scribbly, or geometric and can appear hard or soft, mechanical or organic.

The perceived grey level or *tone* in an illustration depends largely on how dense the strokes are in a region (just like the dots in a dithered halftone image). Although grey-level ramps can be achieved by judiciously increasing stroke density, fine-art illustrations typically emphasize contrast between adjacent regions, and often employ a very limited number of distinct grey levels.

Shapes in an illustration can be defined by *outline* strokes. These strokes are exceptional in that they may be long and individually significant. Often the outline is left implicit by a change in tone or texture. The choice of whether or not to use outlines is largely an aesthetic one, made by the artist, and used to achieve a particular effect. For example, explicit outlines are typically used for hard surfaces, while implied outlines generally convey a softer or more organic object.

Producing fine-art-quality, hand-drawn pen-and-ink illustrations requires a great deal of creativity and artistic ability. In addition, it requires a great deal of technical skill and patience. A real pen and ink have no undo!

1.2 Related work

Most of the published work on "digital painting" is concerned with the problem of emulating traditional artists' tools. Only a few of these works take an approach similar to ours of creating higher-level interactive tools that can produce the same results as their predecessors: Lewis [10] describes brushes that lay down textured paint; Haeberli [7] shows how scanned or rendered image information can be used as a starting point for "painting by numbers;" and Haeberli and Segal [8] use hardware texture-mapping for painting and also mention 3D halftoning effects.

Considerable work has also been done for creating black-and-white illustrations, generally for engineering or graphical design work. The earliest such system was Sutherland's "Sketchpad" [18]. Gangnet et al. [5] use planar decomposition to manipulate and clip geometric objects. Pavlidis [11] provides a method for "cleaning up" schematic drawings by removing hand-drawn irregularities. Quite the opposite (and more along the lines of our work), the Premisys Corporation markets a commercial product, "Squiggle," [13] that adds waviness and irregularities to CAD output to augment lines with extra information and to make the results appear more hand-drawn. Saito and Takahashi [14] produce automated black-and-white illustrations of 3D objects.

Our research group is exploring several different aspects of the pen-and-ink illustration problem. This paper discusses the issues of interactively creating pen-and-ink illustrations, with an emphasis on using 2D greyscale images as a starting point. A second paper shows how principles of illustration can be incorporated into an automated system for rendering 3D models [20]. A third paper examines the issues involved in representing, editing, and rendering the individual strokes that are the building blocks of any line illustration system [4].

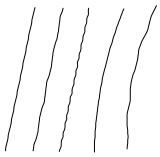


Figure 1: *A closeup view of several individual pen strokes, with various amounts of curve and waviness.*

1.3 Overview

The next section discusses the overall design of our system, as well as its individual capabilities and features. Section 3 presents some example illustrations and describes our experience with using the system. Section 4 suggests directions for future research. The primary data structures and algorithms of our prototype implementation are outlined in appendix A.

2 The Illustration System

Full-color paint systems often support direct simulations of traditional artist tools, such as brushes and paint [3, 17]. However, for our application, there is little purpose in providing the user with a simulated "ink pen" to draw the pen strokes, for several reasons:

- A mouse-based interface does not support the fine control needed for detailed stroke work.
- The strokes of an illustration are not of great individual importance.
- Drawing individual strokes is tedious, and we would like our system to reduce much of that tedium.

Thus, rather than focus on the individual strokes, the system tries to directly support the higher-level cumulative effect that the strokes can achieve: texture, tone, and shape. The user "paints" using textures and tones, and the computer draws the individual strokes.

The illustration system cannot completely ignore individual strokes, however. Outlines are the most notable example of strokes that have individual significance; in addition, an artist might occasionally need to touch up fine details of textured work. Therefore, the system also allows users to draw individual strokes and provides controls for modifying stroke character through smoothing and through the substitution of various stroke styles [4].

To further aid users in creating illustrations, the system allows scanned, rendered, or painted images to be used as a reference for tone and shape. The system also supports edge extraction from images, which is useful for outlining. Finally, a range of editing capabilities is supported so that users are free to experiment or make mistakes.

The following sections discuss the capabilities and workings of the system in greater detail.

2.1 Strokes

It is important that the strokes automatically generated by the system be irregular. Uneven strokes make an illustration look softer, more natural, and hand-drawn, whereas regular strokes introduce mechanical-looking texture. The use of irregular strokes can be compared to the introduction of randomness in image dithering [19].

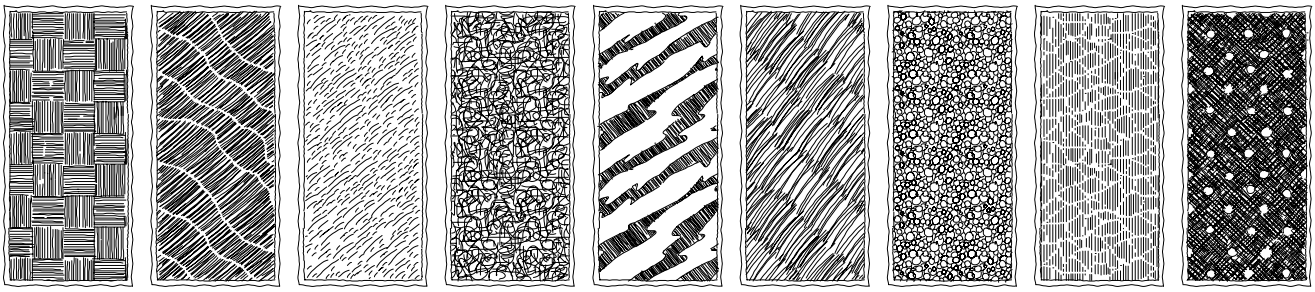


Figure 2: *Assorted stored stroke textures.*

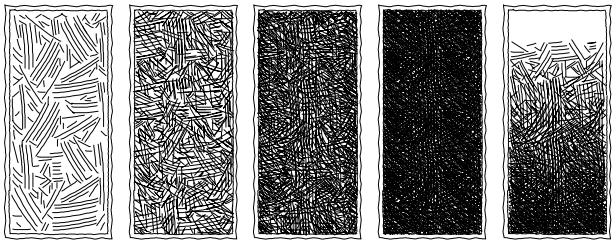


Figure 3: *A single texture drawn with several tone values.*

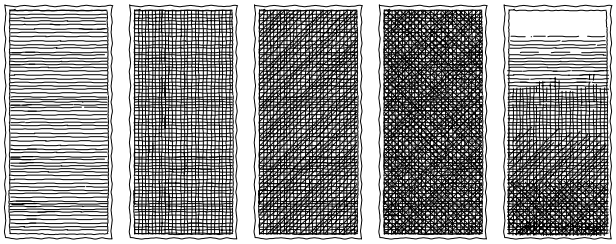


Figure 4: *A prioritized texture. Only the most significant strokes are drawn for light tone values; less important strokes are brought in to darken the texture.*

We cannot simply draw strokes in completely random directions, however—the stroke direction is one of the key elements in defining a texture. Instead, the system perturbs the strokes in a variety of small ways (see Figure 1): strokes can be drawn with a slight wiggle (a wave with slightly randomized frequency and phase); straight strokes can be given a slight overall curvature; and stroke length and direction can be jiggled slightly. Section A.3 describes the stroke-drawing algorithm in greater detail. Currently, strokes in our system are each of constant width, as per a "technical pen" [16].

2.2 Textures

The user paints by rubbing a "brush" over the illustration; the strokes that appear in the region under the brush are generated based on a user-selected *stroke texture* (see Figure 2). The system supports a library of user-defined *stored* stroke textures, as well as several built-in *procedural* stroke textures. In this way, a wide variety of illustration styles can be achieved. These two types of stroke textures are described in more detail below.

Stored stroke textures

A stored texture is simply a collection of strokes. Drawing a texture at a given darkness is a matter of choosing from the collection a subset that has enough strokes to reach the desired tone. (Some textures may be inherently too light—they may not have enough strokes to make dark tones.)

For textures such as stipples and scribbles, the choice of strokes to draw for a given tonality is not critical. In these cases, the system simply selects strokes from the texture in a random sequence, generating candidate strokes and testing the tonal effect of candidate strokes as described in Section A.3. Candidate strokes that pass the tests are drawn, and those that fail are discarded (see Figure 3).

For other textures, however, the system supports a predefined *priority* for each stroke, which specifies an order to use in generating and testing candidate strokes. For example, Figure 4 illustrates a texture in which only horizontal hatches are drawn for light tones, while cross-hatching strokes are used for darker tones. Another example would be a texture in which larger scribbles are drawn before smaller ones.

Creating a good prioritized stroke texture is not always easy—some design iteration may be required before the strokes and their priorities work well together. Once a texture has been created and perfected, however, it can be archived for repeated use. The system lets the user draw textures interactively and can also support textures that are computed programmatically or that are taken from edges extracted from scanned images.

Procedural stroke textures

Many interesting texture effects can be computed procedurally. The system currently supports three types of procedural texturing: stippling (randomly distributed points or short strokes), parallel hatching, and curved strokes. The latter two textures can follow along or against the gradient of a reference image. Since these are the only textures truly built into the system, they are the basic building blocks from which user-drawn stored textures are formed.

To draw procedural stroke textures, the system simply generates appropriate candidate strokes under the region of the brush and tests them, as discussed in detail in Section A.3. More intricate prioritized procedural stroke textures, such as "brick," "wood," or "shingle" textures, can also be defined [20], although they are not currently implemented in our interactive system.

2.3 Reference images

A scanned, rendered, or digitally painted continuous-tone image can be underlaid "beneath" the illustration being drawn, and displayed faintly. This reference image can be used in several ways (see Figure 5):

Figure 5: *Using a grey scale image for reference. Left to right: Original grey scale image; extracted edges; curved hatching across the gradient.*

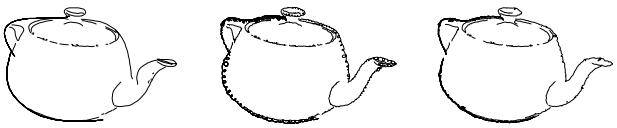


Figure 6: *Manipulating curve detail. Left to right: Teapot edges from Figure 5, with detail removed; alternate details applied to the curves.*

- As a visual reference for the artist.
- As a tone reference for painting, in which case the texture darkness will match that of the image.
- As a source image from which edges are extracted to use for outlining and clipping. The user can select edges corresponding to versions of the image at various resolutions.
- As a progenitor of *stencils*. The user can interactively define stencils by specifying ranges of intensities in the reference image; strokes are drawn only where the reference image value is within the specified ranges.
- As a reference for determining stroke and texture orientation. Textures that follow the reference gradient can be particularly useful for conveying curved surfaces.

Note that its extensive support for reference images makes the illustration system a particularly effective tool for interactive digital halftoning. However, it does not provide automatic halftoning—it is up to the user to choose which stroke textures to apply, where to apply them, and how dark to make them, based on the user's intent and aesthetic sense for the final illustration. One could imagine an automated system to extract texture from an image, but there is not always enough information in the image to achieve the desired effect. For example, the original reference photograph for the goose in Figure 9 does not show feathers in any great detail; the artist must choose textures and introduce tone variation to convey the sense of feathering.

2.4 Detail manipulation

The illustration system supports multiresolution curves [4], allowing users to add or remove detail from strokes and edges. For example, an illustration can be initially made using smooth strokes, which can later be adjusted in subtle or not so subtle ways, using a variety of wiggly or scribbly detail. Alternatively, detail can be removed from an edge extracted from the tone reference in order to yield smoother outlines (see Figure 6).

2.5 Clipping

The user can specify outlines, which may or may not be drawn in the final illustration, but against which strokes (and stroke textures) are clipped. Outlines can be drawn by hand or can be taken from edges in reference images.

Just as individual strokes should not be too regular, the clipped ends of textures should in general be slightly ragged. The system introduces a small amount of random variation by clipping strokes too soon or allowing them to spill beyond the edge of the clipping region (see Figure 7).

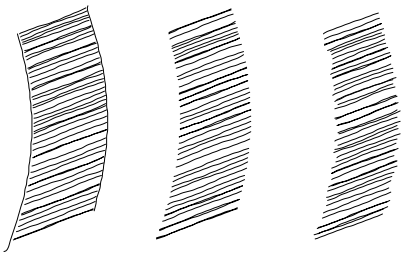


Figure 7: *Strokes clipped to an outline. Left: The outline is drawn. Center: The outline has been removed; notice the hard edge caused by exact clipping. Right: A small amount of random sloppiness creates a softer edge.*

2.6 Individual strokes

Sometimes individual strokes are important enough to be drawn by hand; for example, the hairs in Figure 11 were individually created. The user can draw individual strokes with a mouse or with a tablet. These strokes can be given waviness and clipped in the same manner as automatically-generated strokes. To overcome the mouse's lack of smoothness, unwanted detail can be removed via the multiresolution curve mechanism, or prestored "artistic" irregularities can be introduced, as described in Section 2.4.

2.7 Editing collections of strokes

In addition to modifying individual strokes, the user can edit collections of strokes. Editing operations can be applied to all strokes, to those generated from a given texture, or to strokes selected interactively.

Perhaps the most interesting editing operation is the "lighten" operation. Rather than simply erasing all strokes under the brush, "lighten" incrementally removes strokes. Thus, a textured region that is too dark can be made lighter without destroying the integrity of the texture, instilling pen-and-ink with qualities of a subtractive medium. For example, in the lower left-hand drawing of Figure 8, the mottled effect in the background was created by painting a crosshatch texture to a uniform darkness, then slightly lightening in a few places with touches of the brush.

3 Results

The pen-and-ink illustration system is implemented in C++ and runs at interactive speed on an SGI Indigo2 workstation, without any additional hardware assistance. The system has proven quite successful at assisting users in easily producing a variety of illustrations. All figures in this paper were drawn using the illustration system; only a few minutes were required for the simplest figures, and a few hours were required for the goose in Figure 9. All figures were out-

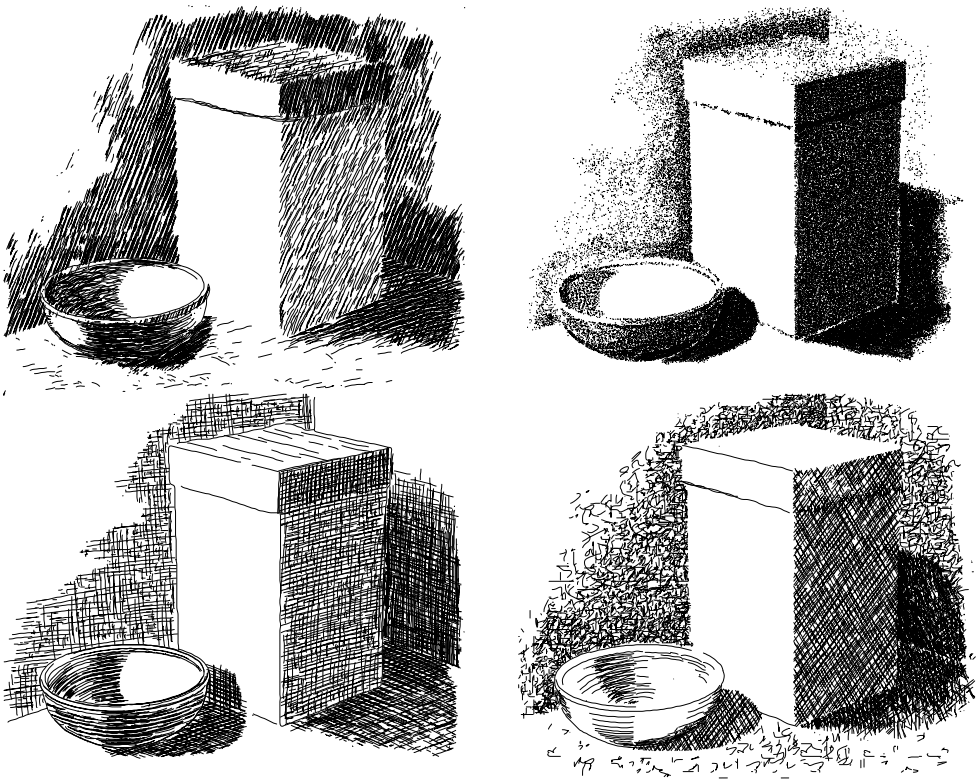


Figure 8: *A single scene, drawn in a variety of styles. Pitz [12] suggests drawing this scene with varying styles, as an exercise for student illustrators. The three drawings on top and left are attempts to closely follow examples given in the book, while the lower right is our own stylistic expression. The illustrations were created using an image of a simple 3D model as a tone reference.*

put in PostScript by our system and printed with the text on a GCC SelectPress 1200dpi printer.

To test the range and quality of the system, we chose to tackle exercises and mimic drawings from illustration texts. Figures 8, 9 and 10 show some of the results. We must admit that the target pen-and-ink drawings in the textbooks are generally finer than ours. However, when we consider that our illustrations were made on a bitmap display, using only a mouse, by programmers who are not trained illustrators, and in a matter of minutes for the simpler drawings, we find our results very encouraging.

4 Future work

The illustration system we have built suggests a number of areas for future research:

- *Experimenting with better interaction techniques.* The control panel of our prototype system has a button or slider for nearly every low-level operation and parameter in the program and hence is somewhat cumbersome to use. A better interface would provide support for common illustrator techniques, such as haloed outlines and stippled edges. In addition, we would like to explore adding much higher-level controls for producing illustrations, including commands to "increase contrast" or "focus attention" on certain regions of the illustration.

- *More sophisticated strokes and stroke textures.* Our simple procedural and stored textures do not yet provide all of the subtlety and variety available to the pen. For example, we would like to include the ability to vary the thickness along a stroke, which is supported in other pen-and-ink work [4, 20].

- *Resolution-independence.* The user should be able to work at a convenient screen resolution, while the final output should have strokes drawn with the highest resolution the printer can support. However, changing resolution in a naive fashion may change the appearance of strokes and stroke textures in undesirable ways. We would like to explore methods of storing illustrations not as collections of strokes, but as higher-level descriptions of tone, texture, and clipping information that could be used to generate the image appropriately at any arbitrary resolution.

- *Combining with 3D.* We would like to interface our interactive system with an automatic renderer for creating pen-and-ink illustrations from 3D models [20] to create an integrated interactive 2D and 3D illustration system.

Acknowledgements

We would like to thank Adam Finkelstein for helping us incorporate his stroke detail research into our editor, and Georges Winkenbach and Tony DeRose for their useful discussion of illustration principles. We would also like to thank SGI for loaning us several machines in order to meet our deadlines.

This work was supported by an NSF National Young Investigator award (CCR-9357790), by the University of Washington Graduate Research and Royalty Research Funds (75-1721 and 65-9731), and by industrial gifts from Adobe, Aldus, and Xerox.

References

[1] Brian Cabral and Leith (Casey) Leedom. Imaging Vector Fields Using Line Integral Convolution. Proceedings of SIGGRAPH 93 (Anaheim,

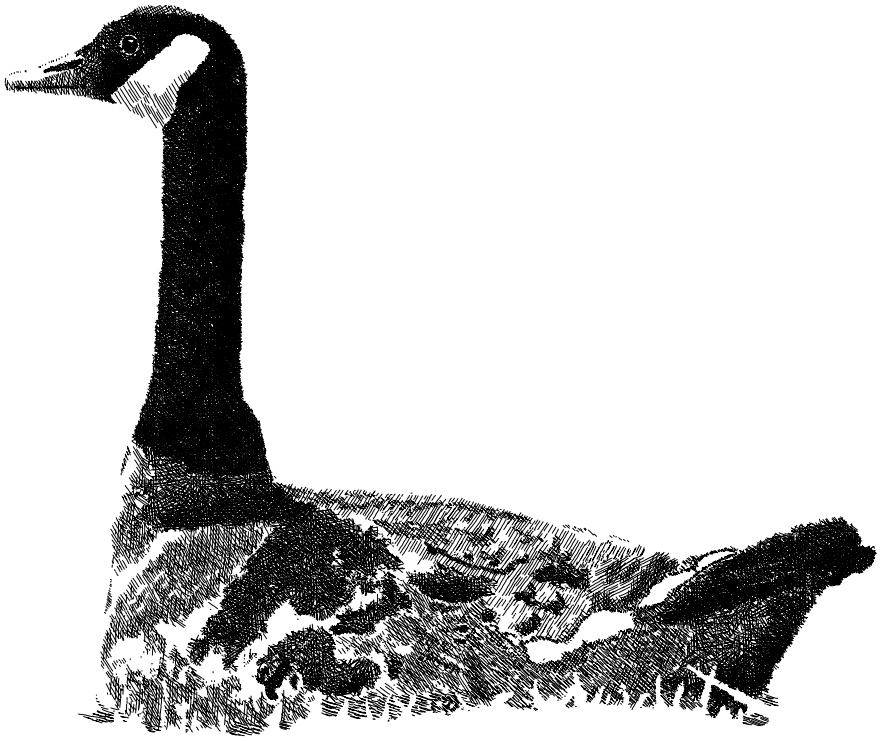


Figure 9: *A Canada goose. Simmons [16] uses the goose as a worked example of tight technical pen drawing from a reference photograph; we attempted to mimic the author's final result with our system, using only a mouse. The book includes a reproduction of the author's own reference photograph, which we scanned to use as a tone reference.*

California, August 1–6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, pages 263–272.

[2] John Canny. A Computational Approach To Edge Detection. In Rangachar Kasturi and Ramesh C. Jain, editors, *Computer Vision: Principles*, pages 112–131. IEEE Computer Society Press, Los Alamitos, California, 1991. Reprinted from *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[3] Tunde Cockshott, John Patterson, and David England. Modelling the Texture of Paint. In *Proceedings of EUROGRAPHICS '92*, pages C–217 to C–226, September 1992.

[4] Adam Finkelstein and David H. Salesin. Multiresolution Curves. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

[5] Michel Gangnet, Jean-Claude Herve, Thierry Pudet, and Jean-Manuel Van Thong. Incremental Computation of Planar Maps. Proceedings of SIGGRAPH '89 (Boston, Massachusetts, July 31–August 4, 1989). In *Computer Graphics* 23, 3 (August 1989), pages 345–354.

[6] Arthur L. Guptill. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York, 1976.

[7] Paul Haeberli. Paint by Numbers: Abstract Image Representations. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6–10, 1990). In *Computer Graphics* 24, 4 (August 1990), pages 207–214.

[8] Paul Haeberli and Mark Segal. Texture Mapping as a Fundamental Drawing Primitive. In *Proceedings of the Fourth Annual EUROGRAPHICS Workshop on Rendering*, pages 259–266, Paris, June 1993. Ecole Normale Superieure.

[9] Douglas Kirkland. *Icons*. Collins Publishers San Francisco, San Francisco, California, 1993.

[10] John-Peter Lewis. Texture Synthesis for Digital Painting. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 23–27, 1984). In *Computer Graphics* 18, 3 (July 1984), pages 245–252.

[11] Theo Pavlidis. An Automatic Beautifier for Drawings and Illustrations. Proceedings of SIGGRAPH '85 (San Francisco, California, July 22–26, 1985). In *Computer Graphics* 19, 3 (July 1985), pages 225–230.

[12] Henry C. Pitz. *Ink Drawing Techniques*. Watson-Guptill Publications, New York, 1957.

[13] The Premisys Corporation, Chicago. *Squiggle*, 1993.

[14] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6–10, 1990). In *Computer Graphics* 24, 4 (August 1990), pages 197–206.

[15] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[16] Gary Simmons. *The Technical Pen*. Watson-Guptill Publications, New York, 1992.

[17] Steve Strassman. Hairy Brushes. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18–22, 1986). In *Computer Graphics* 20, 4 (August 1986), pages 225–232.

[18] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphics Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, 1963.

[19] Robert Ulichney. *Digital Halftoning*. The MIT Press, Cambridge, 1987.

[20] Georges Winkenbach and David H. Salesin. Computer-Generated Pen-and-Ink Illustration. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

A Implementation details

This appendix outlines the implementation of our prototype pen-and-ink illustration system. We will focus on the most significant features: the data structures allowing quick updating and editing of the illustration, and the stroke generation and testing algorithms.

Section A.1 describes the data types used in the system. Section A.2 presents the global data items maintained. The process of generating and using strokes is discussed in Section A.3.

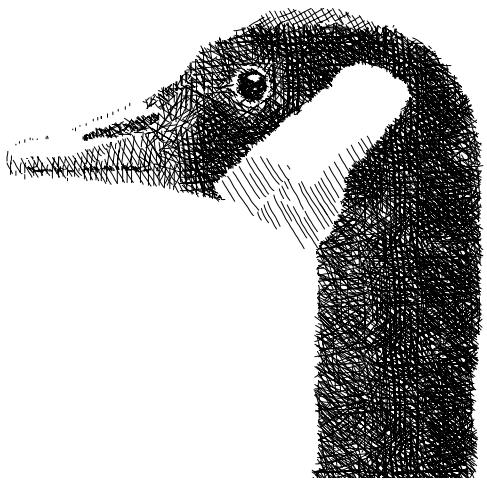


Figure 10: *Close-up of the goose head.*

A.1 Data Types

The two basic data structures used by the illustration system are the *stroke* and the *stroke database*.

Stroke

The data type at the heart of the system is the $Stroke$. Each stroke includes the following fields:

- $pixels$: An arbitrary-size array of *(x,y)* pixel coordinate pairs.
- $length$: The size of the $pixels$ array.
- $width$: The width of the stroke, in pixels.
- $bbox$: The rectangular bounding box of the stroke's pixels.
- id: The texture from which the stroke was derived.
- $priority$: The ranking of a stroke, if in a prioritized texture.

The entries of the $pixels$ array contiguously trace the path of the stroke: *x* and *y* never change by more than ± 1 from one entry to the next.

The operations supported by the $Stroke$ type include: testing to see if a stroke intersects a given rectangular region, circular region, or other stroke; decreasing $length$ by trimming entries off the ends of $pixels$; merging two contiguous strokes into a single stroke; and returning the point in the stroke that is closest to a given pixel.

A stroke can be manipulated as a multiresolution curve [4]. Each entry in $pixels$ is used as a control point of an interpolating spline, which is subject to multiresolution analysis and can have its detail edited or replaced. The resulting curve is scan-converted to recover the contiguous $pixels$ entries required by the $Stroke$ type.

Stroke database

A stroke database maintains a collection of $Stroke$ instances, supporting addition and deletion of strokes, and various queries. It is important that the database operations and queries be quick enough to allow painting and editing at interactive speed.

We implement the stroke database using a modified k-D tree (see Sedgewick [15]). Each node of the tree corresponds to a region of the image; the children of the node partition that region. The partition is always horizontal or vertical and is chosen so as to distribute as evenly as possible the strokes of a region between the two children. Each leaf node contains a list of the strokes that intersect its region. For performance purposes, a limit of 10 strokes per leaf and a minimum size of 5×5 pixels per leaf are maintained (with the area restriction having precedence).

In the modified k-D tree, a given stroke may be referenced by several leaves since a stroke can cross partitions of the tree. The structure allows us to quickly find all strokes that may overlap a specified region by the usual recursion along the limbs of the tree that include the region. Minor extra bookkeeping is required when iterating through the strokes in the leaves to ensure that a stroke is not visited multiple times.

The queries supported by a stroke database include: finding all strokes within a given rectangular or circular region; finding all strokes that overlap a given stroke; and finding the stroke nearest a given pixel. Each query may specify criteria such as a particular id value. These queries allow the system to perform operations such as deleting a stroke and updating the screen as follows: first, find the stroke nearest the cursor; next, delete the stroke from the database and erase it from the screen; finally, find all strokes that overlap the deleted stroke and redraw them.

A.2 Global data objects

The system maintains several global data objects to support the interactive illustration processes:

- *Main stroke database and image bitmap.* The illustration is maintained in a dual representation: a stroke database maintains the collection of $Stroke$ instances that make up the illustration; and an image bitmap allows the system to quickly determine if a pixel has been drawn by one or more strokes. When storing to disk, only the stroke database needs to be saved; the image bitmap can be recreated by traversing the database and drawing all the strokes.
- *Clip-edge database and clip-edge bitmap.* To allow fast clipping of drawn strokes to outline edges, the system maintains a global bitmap into which all clipping edges are drawn (clipping is discussed in Section A.3). The clip edges can come from edge detection of the reference image or from freehand drawing. To allow the user to activate and deactivate edges, the edges are stored as $Stroke$ instances in a stroke database.
- *Stored stroke textures.* The system loads stored stroke textures on demand from a library on disk. A stored texture is defined as a rectangular region with toroidal wrap-around, so that the texture can seamlessly tile the illustration plane. Each texture is maintained in the system as a stroke database. For a prioritized texture, each stroke has an associated priority value. The stroke database of a stored stroke texture is queried but is not modified when the texture is used.
- *Reference image.* The system stores the reference image in memory, allowing quick pixel-by-pixel tone reference and stenciling. Unlike the image bitmap of the illustration, the reference image is an 8-bit greyscale. When a reference image is loaded from disk, the detected edges in the image are added to a clip-edge database and bitmap. We use a Canny edge extractor [2] to detect edges at several image resolutions. This potentially time-consuming processing is only done the first time a given reference image is used; the resulting edges are saved on disk along with the image, so that they can be loaded quickly in the future.

A.3 Drawing strokes

The process to "paint" with strokes is similar for the supported procedural textures—stippling, straight hatching, and curved hatching—and for stored stroke textures. The following pseudocode outlines this process:

```
Paint:
  for each brush position P
    while S ← GenerateCandidateStroke(P)
      ClipStroke(S)
      if TestStrokeTone(S) then
        DrawStroke(S)
      end if
    end while
  end for
```

The steps of this process are described below.

☞ $GenerateCandidateStroke(P)$: At each brush position P, the system may in general try to draw many strokes. Each invocation of $GenerateCandidateStroke$ returns the next stroke instance from a set of candidates. The next stroke returned may be generated dynamically based on the success of the previous strokes. The generation of candidate strokes depends on the texture:

- *Stippling.* There is only a single candidate: a stipple dot at a random location under the brush (chosen with uniform distribution in the brush's polar coordinates). The stipple dot is generated as a length 1 stroke.
- *Straight hatching.* The system tries a sequence of line segments with decreasing length, until a segment is drawn or a minimum length is reached. The midpoint of each stroke is a random location under the brush, and the direction and initial length are specified by the user. The direction may be fixed or aligned relative to the gradient of the reference image. The user may request a small randomization of the direction and length. The user may also specify that only full-length strokes be used, in which case if the initial candidate is not drawn, no further strokes are attempted. Each candidate stroke is a perturbed line segment, generated by the following pseudocode:

```
PerturbedLineSegment(x1, y1, x2, y2, a, ω, c):
    ; (x1, y1) and (x2, y2) are the endpoints of the line segment.
      ; a is the magnitude and ω the base frequency of waviness.
      ; c is the magnitude of curviness.
      ; random() value has uniform distribution on [0, 1].
      ; gaussian() value has normal distribution on [−1, 1].
        dx ← x2 − x1
        dy ← y2 − y1
        s ← √(dx² + dy²)
        δ ← 2πω (1 + (1/4) gaussian())
      γ ← (1/2) δ gaussian()
      i ← 0,  j ← 0,  φ ← 2π random()
  for α ← 0 to 1 step 1/max(|dx|, |dy|)
        ; perturb line with sine waviness and quarter-wave curve.
        b ← a sin(φ)/s + c (cos((π/2)α − (π/4)) − 1)
        pixels[i] ← (x1 + α dx + b dy,  y1 + α dy + b dx)
          ; occasionally shift the sine wave frequency.
          if jδ > (π/2) and gaussian() > (1/3) then
              γ ← (1/2) δ gaussian()
              j ← 0
          end if
          ; update for next pixel.
          φ ← φ + δ + γ
        i ++,  j ++
    end for
```

When needed, intermediate pixels are inserted in order to maintain the contiguity requirement of the *Strokes* type.

- *Curved hatching.* Similar to straight hatching, the system tries strokes of decreasing length until one is accepted. The user specifies the initial length and direction relative to the reference image gradient. A curved stroke is generated by following the image gradient as a vector field (much as was done by Cabral and Leedom [1]) forward and backward for the given length.
- *Stored Strokes.* The system queries the texture's database for a list of strokes that lie under the brush, modulo tiling of the image plane with the texture. The strokes of the resulting list are tried in priority order for prioritized textures, or random order for non-prioritized textures. A prioritized texture may be flagged as *strictly prioritized*, in which case if a candidate stroke fails the tone test, the remaining lower-priority strokes are not considered. Each candidate stroke is generated by translating the stored stroke's *pixels* to the proper tile in the image. Our system does not currently add any randomness to the strokes beyond that which was used when the texture was originally defined. Tiling artifacts are typically not objectionable if the illustration feature size is smaller than the tile size, but could be alleviated through random stroke perturbations.

☞ *ClipStroke(S)*: The candidate stroke S is subjected to a series of clipping operations:

1. *To the bounds of the overall image.*
2. *To the brush.* Clip the strokes to the brush for stored stroke textures to give the user a traditional "textured paint." This clipping step is not performed for procedural textures; in this case, the candidate strokes are generated starting under the brush but may extend beyond its bounds.
3. *To clip-edges.* Trace from the center of the stroke out to each end, examining the corresponding pixels of the global clip-edge bitmap, stopping when an edge is met.
4. *To a reference-image stencil.* Trace from the center of the stroke out to each end, examining the corresponding pixels of the reference image. Can stop at black, white, or any of a number of user-defined ranges of image intensities.

The clipping operations return a "first" and a "last" index into the stroke's *pixels* array, but before actually trimming the stroke, these indices are perturbed up or down by a small random amount to achieve ragged clipping as described in Section 2.5. The magnitude of the perturbation is adjustable by the user. If the stroke is clipped to zero length, it can be trivially rejected at this point.

☞ *TestStrokeTone(S)*: Two tests are performed to see how stroke S affects the image. First, the stroke's pixels in the image buffer are tested: if all the pixels are already drawn, the stroke has no effect on the image and is trivially rejected. Next, the effect of the stroke on the image tone is determined: the stroke is temporarily drawn into the image bitmap and the resulting tone is computed pixel-by-pixel along its length, by low-pass filtering each pixel's neighborhood. Depending on the user's specification, the desired tone may be determined from the reference image's value (via similar low-pass filtering along the stroke), or may simply be a constant value. The stroke fails if it makes the image tone darker than the desired tone anywhere along its length.

☞ *DrawStroke(S)*: To draw stroke S, its pixels in the image bitmap are set, the display is updated, and an instance of S is added to the main stroke database. For stored stroke textures, the system checks to see if the new stroke S overlays an existing instance of the same stroke—such an occurrence could happen, for example, if the earlier stroke was clipped to the brush and the user has now moved the brush slightly. Rather than adding the new stroke, the previously-drawn stroke is extended to include the new stroke's pixels in order to avoid overwhelming the data structures. Note that for a new instance of a stroke to align with a previous instance, any extra randomness should be exactly repeatable; the values for the stroke perturbations should be derived from a pseudorandom function over the illustration plane.

Figure 11: *An illustrated portrait. The reference image was a photograph by Douglas Kirkland [9].*

Orientable Textures for Image-Based Pen-and-Ink Illustration

Michael P. Salisbury *Michael T. Wong* *John F. Hughes** *David H. Salesin*

University of Washington *GVSTC

Abstract

We present an interactive system for creating pen-and-ink-style line drawings from greyscale images in which the strokes of the rendered illustration follow the features of the original image. The user, via new interaction techniques for editing a direction field, specifies an orientation for each region of the image; the computer draws oriented strokes, based on a user-specified set of example strokes, that achieve the same tone as the image via a new algorithm that compares an adaptively-blurred version of the current illustration to the target tone image. By aligning the direction field with surface orientations of the objects in the image, the user can create textures that appear attached to those objects instead of merely conveying their darkness. The result is a more compelling pen-and-ink illustration than was previously possible from 2D reference imagery.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation — Display algorithms. I.4.3 [Image Processing] Enhancement — Filtering

Additional Key Words: Controlled-density hatching, direction field, image-based rendering, non-photorealistic rendering, scale-dependent rendering, stroke textures.

1 Introduction

Illustrations offer many advantages over photorealism, including their ability to abstract away detail, clarify shapes, and focus attention. In recent years, a number of systems have been built to produce illustrations in a pen-and-ink style. These systems can be classified into two broad categories, depending on their input: *geometry-based systems* [1, 2, 7, 12, 16, 17, 18], which take 3D scene descriptions as input; and *image-based systems* [10, 13], which produce their illustrations directly from greyscale images. The main advantage of geometry-based systems is that—because they have full access to the 3D geometry and viewing information—they can produce illustrations whose strokes not only convey the tone and texture of the surfaces in the scene, but—by placing strokes along the natural contours of surfaces—they can also convey the 3D forms of the surfaces. Existing image-based systems, on the other hand, have no knowledge of the underlying geometry or viewing transformations behind the images they are rendering, and until now have been able to convey 3D information only by having a user draw individual strokes or specify directions for orienting particular collections of strokes across the image.

University of Washington, Box 352350, Seattle, WA 98195-2350
{ salisbur | mtwong | salesin }@cs.washington.edu
*NSF STC for Computer Graphics and Scientific Visualization,
Brown University Site, PO Box 1910, Providence, RI 02912
jfh@cs.brown.edu

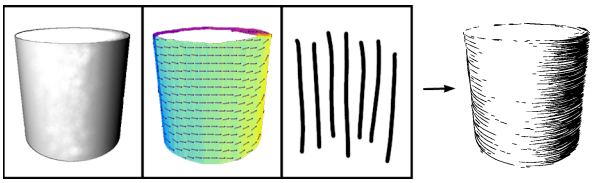


Figure 1 The three components of a layer are from left to right tone, direction, and a stroke example set. An illustration (far right) is rendered based upon one or more such layers.

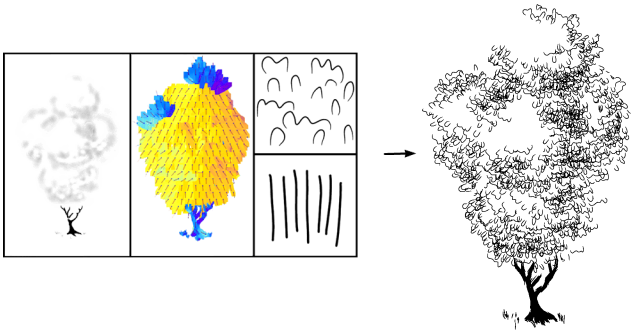


Figure 2 A tree with curved strokes for leaves and straight strokes for branches and trunk.

In this paper, we introduce the notion of "orientable textures" and show how they can be used to readily convey 3D information in an image-based system for pen-and-ink illustration. In our interactive system, a user creates an illustration from a reference image by specifying three components: a greyscale *target image* that defines the desired tone at every point in the illustration, a *direction field* that defines the desired orientation of texture at every point, and a *stroke example set*, or set of strokes, to fill in the tone areas (see figures 1 and 2). Given these three components and a scale for the final illustration, the system creates an *orientable texture*—generated procedurally—that conveys the tone, texture, and forms of the surfaces in the scene. An illustration is composed of one or more such layers of orientable textures, allowing an illustration to be rendered with several, potentially overlapping, types of strokes.

The ability to generate comparable illustrations with an image-based system rather than a geometry-based system offers several advantages. First, using an image-based system greatly reduces the tasks of geometric modeling and of specifying surface reflectance properties, allowing much more complicated models (such as furry creatures and human faces) to be illustrated. Second, an image-based system provides the flexibility of using *any* type of physical photograph, computer-generated image, or arbitrary scalar, vector, or tensor field as input, allowing visualization of data that is not necessarily even physical in nature. Finally, image-based systems offer more direct user control: the ability to much more easily modify tone, texture, or stroke orientation with an interactive digital-paint-style interface.

Although this paper is, to our knowledge, the first to use orientable textures for image-based pen-and-ink illustration (in which the strokes must convey not only orientation, but texture and tone), the idea of orienting strokes for illustration dates back at least as far

as the seminal papers by Saito and Takahashi [11] and Haeberli [6] in SIGGRAPH 90. Winkenbach and Salesin [17] and Meier [9] also make use of oriented strokes for geometry-based illustration.

Supporting orientable textures for image-based pen-and-ink illustration requires solutions to several new subproblems, which we discuss in this paper. These problems include: creating interactive techniques that facilitate the specification of the kind of piecewise-continuous vector fields required for illustration; rendering strokes and stroke textures according to a vector field in such a way that they also produce the proper texture and tone; and efficiently estimating tone as new oriented strokes are progressively applied.

The next section describes the user interface for specifying the components of an illustration. Section 3 discusses the rendering of illustrations with oriented textures. Section 4 presents our results.

2 The interactive system

We provide an editor, similar to a conventional paint program, that allows the user to interactively alter the tone and direction components of a layer.[1] The user can view and edit arbitrary portions of a component at varying levels of zoom, superimpose multiple components, and paint directions directly on top of the target image. For an example of the high-level control afforded by our system, refer to figure 3.

Editing tone. Our tone editor is similar to existing paint programs. It supports lightening, darkening, and other image-processing operations, as well as painting. The user can load a reference image and designate it as a "cloning source." Selected portions of this reference may then be painted into a given layer's tone component. Tone may also be transferred between layers by painting. A negative cloning brush allows the user to freely and creatively reverse tonal relationships in a reference image.

Editing direction. Since we represent a direction field as a grid of direction values, much like an image of pixels, the direction-field editor is similar to the tone editor.[2]

The user "paints" directions on the image with a collection of tools, a few of which we describe here. The basic tool is the *comb*, which changes the directions of pixels beneath the cursor to match the direction of motion of the cursor. If a user wishes to smooth out discontinuities in the direction field, there is a *blending tool* that smooths a region of directions by convolving each point under the brush with a 3×3 filter.[3] There are also various region-filling tools. One tool lets the user fill a region with a constant direction. Another provides *interpolated fill*: the user draws two curves, after which the region between them is filled with directions that are tangents of linear interpolants of the curves. A third provides *source fill*, which orients directions away from a selected point.

The current state of the direction field is shown in two ways: first, a grid of line segment indicators covers the image and everywhere points in the direction of the field; second, a color-coded direction image is superimposed on the tone image

Applying the stroke example set. A *stroke* is a mark to be placed on the page. Each stroke is *oriented*, in the sense that it can be rotated to any angle to follow the direction field where it is placed. The *stroke example set* is a collection of strokes, all drawn with respect to the vertical orientation, that serve as prototypes for the strokes in the final image. Each such stroke is represented as a cubic

[1] The stroke example set is created in a separate program and can be loaded by name.

[2] We represent directions as values from 0 to 255, with 0 down, 128 up, and values increasing counter-clockwise. The resolution of the direction grid is the same as that of the tone image.

[3] We filter directions by first converting them into unit vectors, then performing a weighted sum of those vectors with the weights $(1, 2, 1; 2, 4, 2; 1, 2, 1)$, and then converting the resulting vector back into a direction.

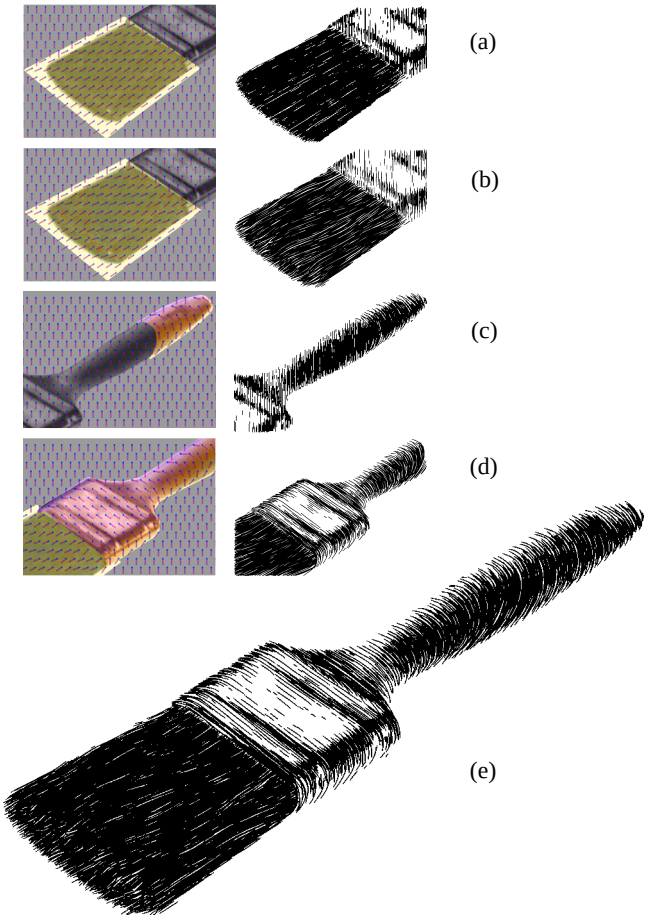


Figure 3 The steps in specifying the direction field for a paintbrush illustration. Shown in inset at various stages during the development of the illustration are, on the left, the user interface, and on the right, the corresponding rendered illustration. By default, the direction field is oriented downward. In (a) we see the effect of an interpolated fill between two lines on either side of the brush bristles. Panel (b) shows the state of the direction field and illustration after some irregularities were introduced to the bristles by nine coarse strokes of the direction comb along the length of the bristles, and thirty fine strokes at the bristle tips. Panel (c) shows the state of the brush handle after interpolating fills between four curves drawn to reflect its surface orientation. In (d), the last section of the direction field covering the metal ferrule has been defined with three interpolating fills. Panel (e) shows the completed brush illustration.

B-spline with knot sequence $(0, 0, 0, 1, 2, \ldots, n-1, n, n, n)$, making it endpoint-interpolating. Thus a stroke example set for "parallel hatching" would contain many nearly vertical line segments, as shown in the third panel of figure 1, while for the leaves in figure 2, the strokes are wavy to suggest the edges of masses of foliage. When a stroke is drawn at a point in the illustration, it is rotated so that the vertical vector in the stroke texture aligns with the direction vector at that point; it is further warped so that this relation is true all along the stroke (see Section 3.1).

The repeated use of strokes from the example set to achieve tone with a specified orientation is a kind of procedural stroke texture. Non-procedural stroke textures were used by Salisbury *et al.* [13, 14]. In this previous work, the textures tiled the plane, and the stroke selected for drawing at a point was the one that happened to pass through that point. By contrast, in this new system the placement of strokes on the final illustration is independent of their relative position in the texture. Spacing between strokes is instead maintained indirectly by the rendering system (see Section 3). Dynamic placement of strokes is an important feature, for if we have

Figure 4 Magnifying a low-resolution direction field using (left) a standard symmetric resampling kernel, and (right) the modified kernel used by Salisbury *et al.* [14]. The same sharp tone component was used for both illustrations.

a direction field that diverges (say, for drawing the water spraying outwards from a fountain) and a stroke texture of parallel straight-line strokes that we wish to have follow the diverging field, a simple plane-tiling will not follow the field, and an embedding of the stroke texture that *does* follow the field will be stretched at the divergent end, necessarily causing the strokes to become more sparse. By contrast, our new method will insert additional strokes as the field widens, thus maintaining the density. In trade for this, we lose the texture-wide coherence that was available in our previous work.

3 Rendering

Once the user has specified the three components of a layer (tone, direction, and texture) our pen-and-ink renderer combines all of the components of each layer to generate the pen strokes of the final illustration. The user need only be concerned with the overall high-level aspects of the illustration such as tone and stroke direction; the system does the tedious work of placing all the strokes. Besides providing easy control over essential elements of an illustration, this separation of components until rendering allows us to produce illustrations at any size by first rescaling the components and then rendering, as described by Salisbury *et al.* [14]. Figure 4 demonstrates magnification of the direction field that respects edge discontinuities.

The rendering process is driven by a notion of "importance." We define the *importance* of a point as the fraction of its intended darkness that has not yet been accumulated at that point. By drawing in order of importance, we make all areas approach their target darkness at the same rate. Rendering therefore consists, roughly, of looking for the location with greatest importance, placing a stroke there, updating an image that records the importance, and repeating, until the importance everywhere is below a termination threshold. Each step of the process has subtleties, which are discussed below.

Matching the illustration to the target. We aim to place strokes in the illustration so that the tone of the illustration "matches" that of the tone image. Matching is necessarily approximate, because the illustration is purely black and white, whereas the tone image is greyscale. To facilitate this approximate matching, we think of each stroke as adding darkness to a *region* of the illustration. Moreover, since strokes in dark areas will be closely spaced and those in light areas will be sparse, the size of each region must be inversely proportional to the darkness. One way of spreading the darkness of a stroke over a region is to blur the image of the stroke when considering the effect of its darkness. To measure the progress of our illustration towards the target image, we therefore compare a blurred version of the illustration with the tone image, where the blurring consists of applying averaging filters of variable size across the illustration, with the size increasing with the target lightness in a region. The diameter of the blurring filter is the same as the average inter-stroke distance required to achieve the target lightness.

We record our success at matching the illustration to the tone image by maintaining a *difference image*, updated after each stroke is drawn, whose value at each pixel is the difference between the tone image and a blurred version of the illustration. The *importance image* is derived from the difference image; its value at each point is the current difference divided by the initial value of the difference.[4]

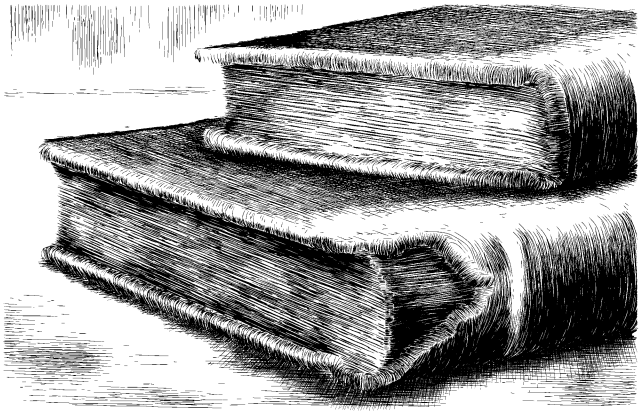


Figure 5 Stacked books (after illustration by Frank Lohan [8].)

Drawing strokes in the right place. One of the basic rules of pen-and-ink illustration is that strokes should be placed evenly: close together in dark areas, widely spaced in light areas [8]. In the computation of the difference image, the importance-image values at points within some distance of a stroke are lowered when the stroke is drawn, with points near the stroke being lowered most; the size of the region affected is determined by the target tone (see Section 3.2). This algorithm tends to maintain stroke separation.

To help determine where to draw the next stroke, i.e., the location with greatest importance, we maintain a quadtree on the importance image, updated locally whenever a stroke is drawn.

Deciding when to stop. We do not actually try to drive the importance image to zero: even our filtered version of the strokes cannot hope to match the values in the tone image exactly. Instead, we try to drive the importance image to within a narrow tolerance around zero.[5] When the maximum value in the importance image is below a termination threshold, the renderer declares the illustration complete and stops drawing strokes.

3.1 Drawing a Stroke

The lowest-level activity is the actual drawing of a stroke, in itself a complex task. Once the algorithm knows where to place it, the stroke must be oriented, bent, and drawn. It must also be clipped if extending it further would make the illustration too dark. We discuss these processes in turn.

Orienting and bending. To start, the algorithm randomly selects a prototype stroke from the stroke example set. We would like to map this stroke into the direction field so that, at every point along its length, the stroke's new angle relative to the direction field is the same as the prototype stroke's angle with respect to the vertical direction. Since this mapped stroke is not easy to find, we approximate it by mapping the control hull of the prototype stroke into the direction field in an angle-preserving way, as described below. This process produces a mapped stroke that is close to our ideal stroke and is easy to compute, although it is the *control hull* of the stroke that passes through the target point rather than the stroke itself. The errors thus introduced are small as long as the control hull fits the stroke closely and the direction field does not change too fast.

To map the control hull into the direction field, we first pin a random control point P_i of the stroke onto the target location X in the

[4] If the initial difference is zero (i.e., if the target tone is white), the importance is set to zero.

[5] The storage values 0 to 255 correspond to importance values of -0.14 to 1.0. This range is a compromise between providing enough resolution in the positive values to distinguish differences in importance, and allowing negative values so that slightly overdarkened areas can be accommodated.

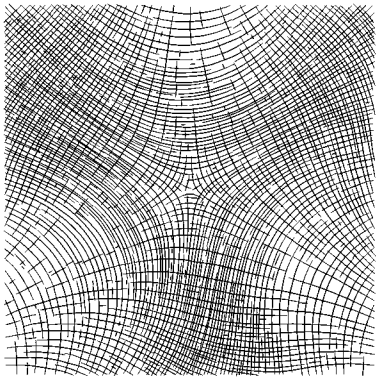


Figure 6 A visualization of four quantities from a symmetric tensor field. The integral curves of the principle-direction field are shown by strokes; the density of the strokes in each direction is related to the magnitude of the principle value associated with that direction.

Figure 7 Hair and face (after untitled photograph by Ralph Gibson [3].)

illustration. To find the location of P_{i+1}, we need to map the points along the segment P_iP_{i+1} to locations $\gamma_i(s)$ in the illustration, for $0 \leq s \leq 1$. To define γ_i, let θ_i denote the angle between the vector $v_i = P_{i+1} - P_i$ and the vertical; for each s, we want the angle between the tangent $\gamma_i'(s)$ and the direction field at $\gamma_i(s)$, called $d(\gamma_i(s))$, to be θ_i as well. In addition, we want the arclength of $\gamma_i(s)$ between $s = 0$ and $s = 1$ to be the length of v_i. In summary, we want

$$\begin{aligned} \gamma_i(0) &= X \\ \text{angle}(\gamma_i'(s),\ d(\gamma_i(s))) &= \theta_i \\ \|\ \gamma_i'(s)\ \| &= \|\ v_i\ \| \end{aligned}$$

We solve this set of differential equations numerically, using Euler integration, and record $\gamma_i(1)$ as the place to map P_{i+1}. We repeat this process to place the remaining points of the hull. Because our strokes have many control points, this approach effectively warps the stroke so that at every point its angle to the direction field in the illustration is very similar to its angle to the vertical in the stroke example set.

Clipping. Pen-and-ink artists have various rules for clipping strokes. One widely-accepted convention is that strokes do not cross object boundaries or boundaries between semantically different portions of objects, such as the edges of hard shadows [15]. We adhere to this convention by clipping strokes when they reach places where the direction field turns rapidly.[6] Strokes are also clipped when continuing to draw them would over-darken some region of the image. If a stroke is sufficiently short and has been clipped for this latter reason, it is removed altogether—pen-and-ink artists do not generally use short strokes to fill in every little bit of a dark area—and the importance value there is set to "below threshold" so that no further strokes will be draw into that area.

After the stroke is followed as far as possible in each direction from the pinned location, it is added to the illustration, and the difference and importance images are updated.

3.2 Updating the difference image

To quickly update the difference image with each added stroke, we sacrifice accuracy for efficiency through two approximations that seem to work well in practice.

The first approximation is that instead of blurring the current illustration after adding each stroke and subtracting the result from the tone image, we subtract a blurred version of the stroke from the difference image. This assumption amounts to presuming that the blurred version of multiple strokes will be the same as the sum of blurred versions of the individual strokes, which is fine when strokes do not overlap; when they do, we lighten the blurred version of the stroke as described below.

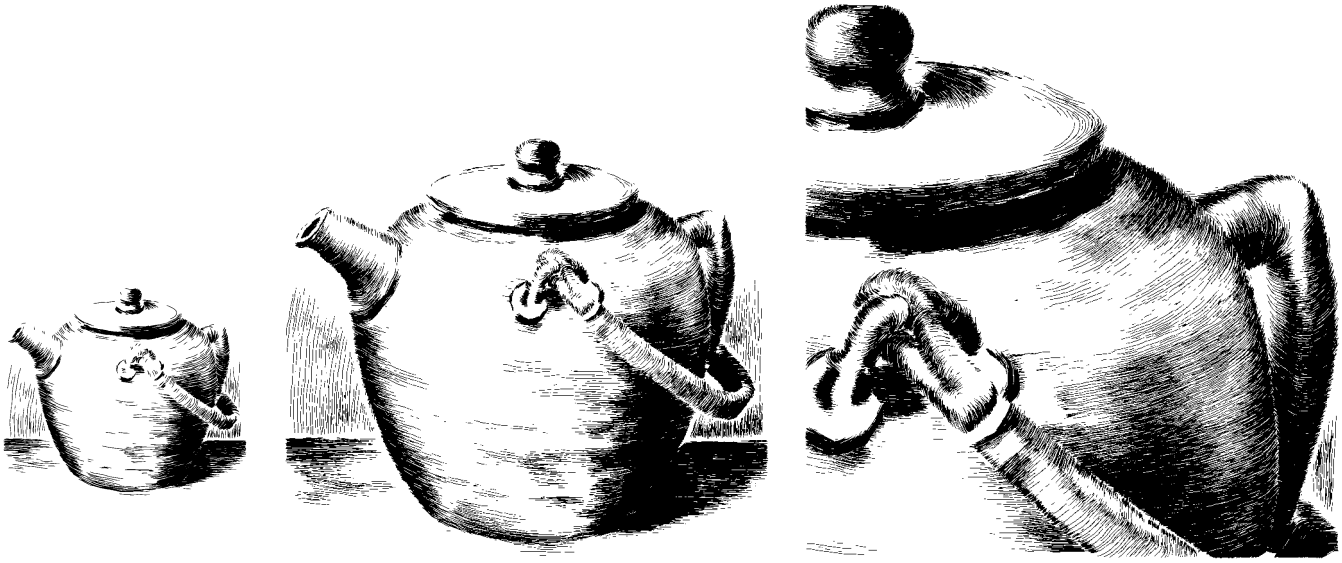
The second approximation is in our computation of the filtered image of a stroke. Instead of rendering the stroke itself, we render its control hull as a wide blurry line. The width w is computed as $2h/t$ mm, where h is the stroke thickness (in mm) and t is the desired tone value between 0.0 (white) and 1.0 (black), and then clamped to the range 1–10 mm. We use Gupta-Sproull antialiased line drawing [4], but we supply the algorithm with a modified "darkness look-up table," whose width is as specified above, and whose height is twice the reciprocal of the width.[7] If the strokes are drawn with even spacing w, a nearly-constant blurred tone of average value t results. In our Gupta-Sproull computation, we treat neither the endpoints nor major-axis-direction changes as exceptional cases. In practice, these simplifications seem to have had no discernible effect.

Overlapping strokes and darkness adjustment. For light areas in the final illustration, strokes rarely overlap, whereas in dark areas they will often overlap. If each stroke in a dark region is counted as contributing as much darkness as a comparable stroke in a light area, the dark-area strokes will be overcounted: points where strokes cross will count as having been darkened twice or more. We therefore compute a *lightening factor*, which is a function of tone and the stroke example set. These lightening factors are computed in a preprocessing step: we draw many strokes into a buffer and record the buffer's darkness after each stroke. When we finish, we will know that, for instance, in an area of 50% grey, only 90% of the pixels drawn end up being visible; the rest overlap with other black pixels. In that case, when filling a region with a target tone of 50% grey, we would reduce the darkness of the filtered strokes to 90% before adding them to the blurred image, assuming that on average only 90% of their area does not overlap with other strokes in that region and will therefore actually contribute darkness to the illustration.

This approximation is not only faster than drawing-then-blurring, it also allows us to render a new stroke directly into the difference image without using a separate buffer. The lightening factor described above is incorporated into the "darkness look-up table" so that each stroke is drawn by looking at the underlying target tones. These tones determine which portion of the darkness look-up table

[6] Some automated assistance in detecting object boundaries would be valuable. We also intend to let the user draw into an "outline image," which would be used for both drawing outlines and truncating hatching strokes.

[7] For width w, height h and distance from stroke center x, the look-up value is $(0.884/h)e^{-2.3(x/w)^2}$, which is simply a bump function that tapers to nearly zero.



(a) (b) (c)

Figure 8 A teapot at three different scales (after illustration by Arthur Guptill [5].)

to use, and the values found there are directly incorporated into the difference image.

3.3 Output enhancements

The strokes to be drawn are deposited in a PostScript file, along with an interpreter that converts B-splines into drawable PostScript Bézier segments. We can also add two "stroke character" enhancements to the B-splines before printing (see the stroke detail inset of Figure 9).

The first enhancement is to render strokes with variable width.[8] Each stroke has three widths associated with it—one at each end and one in the middle. These widths are adjustable on a per-layer basis from the editing interface, and impart subtle expressive effects. Tapering the ends of strokes is ideal for rendering hair, but inappropriate for rendering hard shadows, for example.

The second enhancement is the addition of small "wiggles" to strokes more than 5mm long, to simulate a hand-drawn appearance. This effect is achieved by first resampling the control hull (except for the endpoints, which we copy), placing points with random spacing of about 4mm $\pm$1mm. We then randomly perturb each interior control point slightly along the angle bisector of its two adjacent sides, and perturb the two end control points both along and orthogonal to the control hull segments that they terminate. In the current system, the perturbations are uniformly distributed between $-$0.15mm and 0.15mm.

4 Results

The pen-and-ink illustration system was written in two linked parts: the user interface was written in C++, and the rendering engine was written in Modula-3. The interface runs at interactive speed, and the pen-and-ink renderer takes a few minutes to render the illustrations presented here (see Table 1).

We have produced several illustrations to test the capabilities of our system. Figures 5 and 8 are attempts to closely follow examples of real pen-and-ink drawings from illustration texts. Figure 8 also shows that our system can rescale illustrations while maintaining the character of their texture.

[8]The adjustments that are made are ignored in the computation of darkness—they are to be thought of as merely embellishments.

Fig	Content	% Reduction	# Strokes	Time (sec)
5	Books	58	16722	258
6	Vectors	35	665	25
7	Hair/Face	79	37618	788
8a	Teapot small	65	2924	50
8b	Teapot	65	8361	77
8c	Teapot closeup	65	13617	200
9	Raccoon	62	55893	960

Table 1 Illustration statistics and rendering timings measured on a Silicon Graphics workstation with a 180MHz R5000 processor.

Figure 6 shows a way of visualizing measured or computed vector fields using our system. It was created by bypassing the interactive stage of the system and feeding directions and tones directly into the renderer. Figures 7 and 9 show our ability to render non-smooth, difficult-to-model surfaces such as hair and fur. Our stroke lengths are approximately 1–10cm in the original PostScript rendering. This scale is similar to that at which pen-and-ink artists typically work. These artists often reduce their work for final presentation to achieve a finer, more delicate feel. We have done the same with our illustrations; the reductions are reported in Table 1.

5 Future work

Our current system suggests two principle areas for future research.

Interactive illustrations. Currently the user interacts with the components of the underlying representation of the illustration. It would be nice for the user to have the option of interacting instead with the pen-and-ink illustration itself. Modifications to the illustration would be immediately reflected by corresponding changes in the tone or direction. While previous interactive systems [13] have allowed the user to directly manipulate the illustration, they do not—as does our system—allow the user to specify abstract high-level attributes of the illustration, and thus are not required to make a large number of changes as the result of a simple user action. With our system, changing the directions underneath the cursor can easily require removing and reapplying hundreds of strokes. Much of the incremental update mechanism needed for such behavior is already supported by our system, but we currently would require a considerable increase in rendering speed to make such an interface responsive enough to be usable.

Figure 9 Raccoon with detail inset showing stroke character.

Coherent textures. Many pen-and-ink drawings make use of textures such as bricks or shingles or fabrics that require strokes to appear in locally coherent patterns. Many artists also draw small groups of parallel hatches together in coherent clusters when filling in large areas of tone. We would like to support these kinds of coherent textures in our illustrations. The biggest difficulty is in dealing with diverging direction fields, since it is not obvious how to maintain local coherence and scale while following such a field without tearing the texture at some point.

Acknowledgments

This work was supported by an Alfred P. Sloan Research Fellowship (BR-3495), an NSF Presidential Faculty Fellow award (CCR-9553199), an ONR Young Investigator award (N00014-95-1-0728) and Augmentation award (N00014-90-J-P00002), and an industrial gift from Microsoft.

References

[1] Debra Dooley and Michael Cohen. Automatic illustration of 3D geometric models: Lines. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, pp. 77–82, March 1990.

[2] Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, September 1995.

[3] Ralph Gibson. *Tropism: photographs*. Aperture, New York, 1987.

[4] S. Gupta and R. F. Sproull. Filtering edges for gray-scale displays. *Computer Graphics (SIGGRAPH '81 Proceedings)*, 15(3):1–5, August 1981.

[5] Arthur L. Guptill. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York, 1976.

[6] Paul Haeberli. Paint by numbers: Abstract image representations. *Computer Graphics*, 24(4):207–214, August 1990.

[7] John Lansdown and Simon Schofield. Expressive rendering: A review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

[8] Frank Lohan. *Pen and Ink Techniques*. Contemporary Books, Inc., Chicago, 1978.

[9] Barbara J. Meier. Painterly rendering for animation. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pp. 477–484. Addison Wesley, August 1996.

[10] Yachin Pnueli and Alfred M. Bruckstein. **Dig**$^{\mathbf{i}}{}_{D}$*ürer* — a digital engraving system. *The Visual Computer*, 10(5):277–292, 1994.

[11] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. *Computer Graphics*, 24(4):197–206, August 1990.

[12] Takafumi Saito and Tokiichiro Takahashi. NC machining with G-buffer method. *Computer Graphics*, 25(4):207–216, July 1991.

[13] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, pp. 101–108. ACM Press, July 1994.

[14] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pp. 461–468. Addison Wesley, August 1996.

[15] Gary Simmons. *The Technical Pen*. Watson-Guptill Publications, New York, 1992.

[16] Thomas Strothotte, Bernhard Preim, Andreas Raab, Jutta Schumann, and David R. Forsey. How to render frames and influence people. *Computer Graphics Forum*, 13(3):455–466, 1994. Eurographics '94 Conference issue.

[17] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, pp. 91–100. ACM Press, July 1994.

[18] Georges Winkenbach and David H. Salesin. Rendering free-form surfaces in pen and ink. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pp. 469–476. Addison Wesley, August 1996.

To appear, NPAR 2002, Annecy, France

Weighted Voronoi Stippling

Adrian Secord*
Department of Computer Science
University of British Columbia, Vancouver, BC, Canada

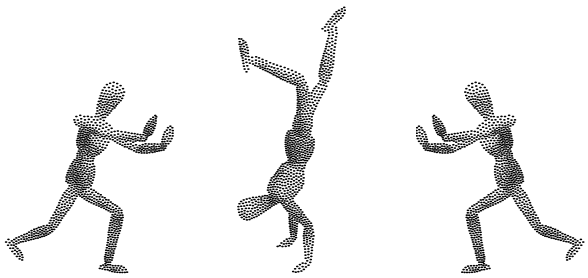


Figure 1: Artist's posable figures with approximately 1000 stipples each

Abstract

The traditional artistic technique of *stippling* places small dots of ink onto paper such that their density give the impression of tone. The artist tightly controls the relative placement of the stipples on the paper to produce even tones and avoid artifacts, leading to long creation times for the drawings.

We present two non-interactive techniques for generating stipple drawings from grayscale images using weighted centroidal Voronoi diagrams. An iterative technique acts on input images directly to produce high-quality stipple drawings and a real-time approach uses precomputed dot distributions to stipple images quickly.

CR Categories: I.3.3 [Picture/Image Generation]: Display algorithms I.3.4 [Graphics Utilities]: Paint systems I.3.5 [Computational Geometry and Object Modelling]: Geometric algorithms, languages and systems

Keywords: Non-photorealistic rendering, stippling, Voronoi diagrams

1 Introduction

Stippling as a technique came into existence to give artists control over the half-toning processes used when printing images in books was a new and difficult task [Jastrzebski 1985]. The technique consists of carefully placing many small dots of ink on paper to approximate different tones. Stipples are placed closer together to form dark regions and further apart to form lighter regions. The stipples must be placed evenly yet randomly so that the human eye does not see spurious patterns that are not a part of the intended impression. The stipples may vary in size and occasionally shape to convey subtle details.

The original advantage of stippling was its ease of reproduction. The half-toning used to print images in books was of highly variable quality and often drawings were drastically resized to meet space requirements. While normal drawings suffered from such treatment stipple drawings retained their attributes more faithfully. In addition, printing a stippled drawing requires only the ability to produce dots of a single colour, making it an inexpensive technique [Wood 1994].

However, stippling has significant artistic merit independent of its utility. The stipples can represent fine detail and texture with little cost in complexity. Stippling is particularly good at clearly representing smooth, rounded objects without sharp edges and so is often used in medical and archaeological texts.

We wish to generate stipple drawings from images with as little user input as possible. The goal is to develop a tool which can generate high-quality stipple drawings from any source whatsoever, which implies that we use images as input and not 3D models. While this limits the amount of information we have to work with, it allows us a greater variety of input sources. For example, a user could start from a scanned pencil sketch, a photograph, the output of a 3D interactive application, frames of an animation, etc.

One of the features of a good stipple drawing is that the stipples are *well-spaced*, that is, the stipples do not clump together, leave uneven voids, or form unwanted patterns. The artist achieves this by carefully placing each stipple onto the page, explaining why stipple drawings often take weeks to create by hand.

Central to our approach is the use of centroidal Voronoi diagrams to produce good distributions of points, as explained in Section 2.1. These distributions can be pre-computed for various different constant tonal values and accessed at run-time to generate stipple drawings rapidly, as covered in Section 4. Alternatively, the input image can be used directly as a weighting function to create a distribution of points that approximate its tones. This method produces images of higher quality but takes more processing time, as explained in Section 3.

*ajsecord@cs.ubc.ca, http://www.cs.ubc.ca/~ajsecord

1.1 Related Work

Our iterative method is a direct descendant of the one described in Deussen et al. [2000]. Their method generates stipple drawings by first placing stipples roughly using a dithering algorithm on the input image and then relaxing them using Lloyd's algorithm until they are well-spaced. Lloyd's algorithm was first introduced to computer graphics by McCool and Fiume in [1992] for the generation of sampling point sets. Lloyd's algorithm and the resulting centroidal Voronoi diagrams are explained in section 2.2. However, Deussen et al.'s relaxation step does not take into account the underlying image, resulting in a blurring of image boundaries. The blurring occurs because the relaxation process attempts to space out closely-packed stipples and compress widely-spaced stipples. Since their tool was designed for interactive use with an artist, they solved this problem by having the user confine sets of stipples to fixed regions, which are aligned to important image boundaries. Since the stipples are not allowed to cross the region boundaries during relaxation, the most important edges are preserved. We wish to find an algorithm that will maintain image boundaries without human interaction.

Hausner [2001] uses an approach similar to our iterative technique outlined in Section 3 for aligning the rectangular tiles of a decorative mosaic. Their approach differs significantly from ours in that they must align the tiles' orientation in addition to their position. However, edges that are to be preserved by the algorithm must be entered separately by the user. This makes the algorithm less suitable for a non-interactive application.

2 Voronoi Diagrams

An ordinary Voronoi diagram is formed by a set of points in the plane called the *generators* or *generating points*. Every point in the plane is identified with the generator which is closest to it by some metric. The common choice is to use the Euclidean L_2 distance metric

$$|\mathbf{x}_1 - \mathbf{x}_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

where $\mathbf{x}_1 = (x_1, y_1)$ and $\mathbf{x}_2 = (x_2, y_2)$ are any two points in the plane. The set of points in the plane identified with a particular generator form that generator's Voronoi region, and the set of Voronoi regions covers the entire plane. Figure 2(a) illustrates a set of generating points and their associated Voronoi regions.

We implemented the fast 3D graphics hardware-based algorithm in Hoff [1999] and originally in [Woo et al. 1997] to compute our Voronoi diagrams. The algorithm draws a set of right cones with their apexes at each generator. The cones all have the same height and are viewed from above the apexes with an orthogonal projection. In addition, each cone is given a unique colour which acts as the generator's identity. Since the cones must intersect if there is more than one generator, the z-buffer determines for each pixel which cone is closer to the viewer and assigns that pixel the appropriate colour value. We can then scan the resulting image and determine which generator is closest to each pixel by using the unique colours. This technique allows us to compute discrete Voronoi diagrams extremely quickly and perform computations on the resulting regions.

2.1 Centroidal Voronoi Diagrams

A *centroidal* Voronoi diagram has the interesting property that each generating point lies exactly on the centroid of its Voronoi region. The centroid of a region is defined as

$$\mathbf{C}_i = \frac{\int_A \mathbf{x}\rho(\mathbf{x})dA}{\int_A \rho(\mathbf{x})dA} \tag{1}$$

where A is the region, $\mathbf{x}$ is the position and $\rho(\mathbf{x})$ is the density function. For a region of constant density ρ, the centroid can be considered as the centre of mass. Figure 2(a) has the centroids of each region marked with small circles.

A centroidal Voronoi diagram is a minimum-energy configuration in the sense that it minimizes $\int_A \rho(\mathbf{x})|\mathbf{C}_i - \mathbf{x}|^2$ [Du et al. 1999]. Practically speaking, a centroidal distribution of points is useful because the points are *well-spaced* in a definite sense. Figure 2(b) shows a centroidal Voronoi diagram.

2.2 Generating Centroidal Voronoi Diagrams

Lloyd's method [Okabe et al. 1992] is an iterative algorithm to generate a centroidal Voronoi diagram from any set of generating points. The algorithm can simply be stated:

Algorithm 1 Lloyd's method

while generating points $\mathbf{x}_i$ not converged to centroids **do**
  Compute the Voronoi diagram of $\mathbf{x}_i$
  Compute the centroids $\mathbf{C}_i$ using equation (1)
  Move each generating point $\mathbf{x}_i$ to its centroid $\mathbf{C}_i$
end while

Figure 2(a) relaxes under Lloyd's algorithm to become Figure 2(b). The convergence of Lloyd's algorithm to a centroidal Voronoi diagram has been proven for the one-dimensional case. The higher dimensional cases seem to act similarly in practice, though no proof is known [Du et al. 1999]. There are several different convergence criteria which should be equivalent in the limiting case as the algorithm runs forever. The obvious criterion to use would be that the computed centroids are numerically equal to the generating points. However, for most applications this criterion is far too stringent and it would be perhaps better to look at the average distance moved by all generating points. Since we are interested in generating well-spaced sets of points, we look at the average change in inter-point distance, or equivalently, the average change in Voronoi region area.

2.2.1 Efficient Computation of Centroids

Calculating the centroids requires efficiently evaluating the integrals in equation (1). Since the integrals are over arbitrary Voronoi regions, we convert to iterated integrals and integrate the region row by row. In this manner we can precompute much of the integral.

The denominator of the centroid is transformed as follows:

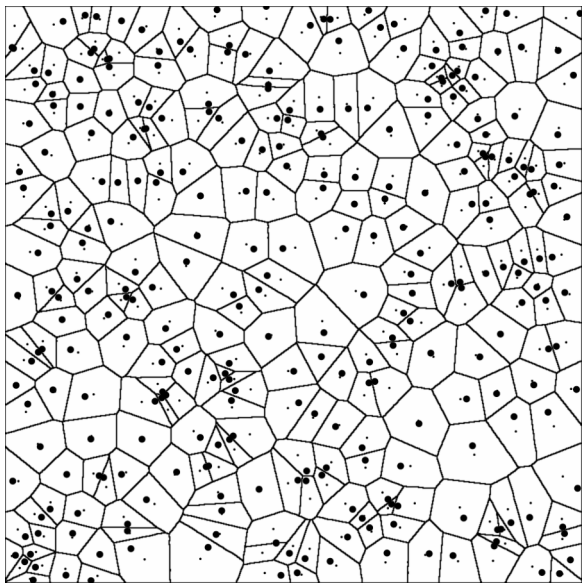
$$\begin{aligned}\int_A \rho(\mathbf{x})dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} \rho(x,y)dxdy \\ &= \int_{y_1}^{y_2} [P]_{x_1}^{x_2}\, dy\end{aligned}$$

where $P \equiv P(x,y) \equiv \int_0^x \rho(s,y)ds$ can be precomputed from the density function[1]. Note that we cannot precompute the entire integral because we do not know the boundaries of the Voronoi regions beforehand.

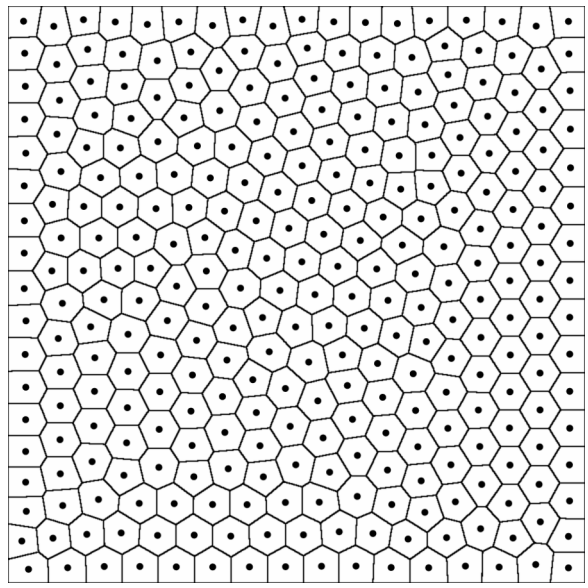
The numerator of the y-coordinate of the centroid is transformed similarly:

$$\begin{aligned}\int_A y\rho(x,y)dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} y\rho(x,y)dxdy \\ &= \int_{y_1}^{y_2} y\,[P]_{x_1}^{x_2}\, dy\end{aligned}$$

[1] Recall that $[\int_0^x f(s)ds]_a^b = \int_0^b f(s)ds - \int_0^a f(s)ds = \int_a^b f(s)ds$



(a) Voronoi diagram generated by the set of generators (large dots). Centroids of each Voronoi region are marked by the small dots.



(b) Centroidal Voronoi diagram

Figure 2: General and centroidal Voronoi diagrams.

The numerator of the x-coordinate of the centroid involves integration by parts:

$$\begin{aligned}\int_A x\rho(x,y)dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} x\rho(x,y)dxdy \\ &= \int_{y_1}^{y_2}\left\{[xP]_{x_1}^{x_2} - \int_{x_1}^{x_2} Pdx\right\}dy \\ &= \int_{y_1}^{y_2}[xP-Q]_{x_1}^{x_2}\,dy\end{aligned}$$

where $Q \equiv Q(x,y) \equiv \int_0^x P(s,y)ds$ can also be precomputed from the density function.

Note that the final expressions require numerical integration only in the y-direction and otherwise involve expressions only at the region boundaries x_1 and x_2. P and Q are precomputed once from the density function and then evaluated at the horizontal end points x_1 and x_2 as needed. This allows us to compute the integrands only at region boundaries and not at every pixel. Otherwise we would have to compute the integrands $x\rho$ and $y\rho$ for every span of pixels across a region and numerically integrated. The above integrand computation is particularly simple – at worst two look-ups for P and Q, a multiplication and a subtraction. In addition, if the Voronoi region is non-convex for numerical reasons, the scan conversion effectively decomposes the region into convex sub-regions, that is, single spans of pixels.

2.3 Resolution of Voronoi Calculation

One disadvantage of using a discrete calculation of the Voronoi regions is the calculation of the centroids is affected by the resolution of the diagram. The relative error of the calculated centroid location will increase as the number of pixels per Voronoi region decreases. A related problem is that if the resolution is low enough, two generating points can effectively overlap and one of the regions will disappear. The solution, as described in [Hoff III et al. 1999], is to split the diagram into tiles and compute each tile at the full resolution available and then stitch the full diagram back together at a higher virtual resolution. The virtual resolution can be increased arbitrarily to meet a lower bound on Voronoi region pixel area.

3 Stippling with Weighted CVDs

The centroidal Voronoi diagrams in Section 2.1 incorporate the idea of a density function $\rho(x,y)$ which weights the centroid calculation. Regions with higher values of ρ will pack generating points closer than regions with lower values. During the iteration of Algorithm 1, the darker regions of the image appear to "attract" more points. We can use Algorithm 1 directly to generate high-quality stippling images by treating a grayscale image as a discrete two-dimensional function $f(x,y)$ where $x,y \in [0,1]$ and $0 \le f(x,y) \le 1$ is the range from a black pixel to a white pixel. Define a density function $\rho(x,y) = 1 - f(x,y)$. We can then stipple a given image by first distributing n points in the image and using algorithm (1). Although any distribution of initial points will eventually converge, it is useful to start with a distribution that approximates the final form. Deussen et al. [2000] use a dithering algorithm and we use simple rejection sampling to generate an initial distribution.

3.1 Results

We expect that at the limit of large numbers of very small stipples, the stipple drawing will approximate the grayscale image. Centroidal Voronoi diagrams produce distributions of points that approximate a blue noise distribution, that is, a random distribution with a constraint on the minimum distance between points. Blue noise distributions are useful because they do not introduce spurious patterns such as lines or grids. They can also approximate a constant tone because of the minimum distance constraint. Blue

Figure 3: Close-up of large Peperomia leaves with 20000 stipples of radius 2×10^{-3}

Figure 4: Small Peperomia plant, lit brightly from the right, with 20000 stipples of radius 1.0×10^{-3}

Figure 5: Large Peperomia plant with 20000 stipples of radius 2×10^{-3}

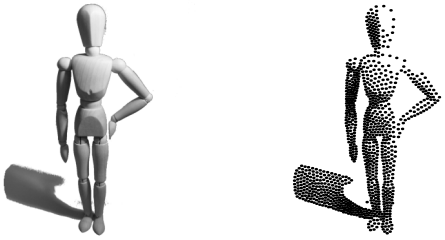


Figure 6: Figure with 1000 stipples of radius 5×10^{-3}

noise distributions have been used to create very high-quality dither patterns for colour reduction [Ulichney 1988]. Figure 3 shows a grey-scale close-up image of some Peperomia leaves with a drawing of 20000 stipples. The fine stippling approximates the tones of the image very well, including the textures inside the leaves.

Figure 4 shows a different small Peperomia plant, lit from the side, with 20000 stipples. Although the number of stipples per square inch is less than in Figure 3, the large number of stipples still renders a faithful image. In particular, note the hard edges maintained by the stipple drawing. Figure 5 shows the full Peperomia plant from Figure 3 with 20000 stipples. Observe the colouration of the centre of the leaf facing the viewer. While the method of Deussen et al. can easily produce sharp edges through user interaction, producing the gradual change in tone visible on the leaf would be difficult. The even spacing of points along the edges of the leaves is the result of the interaction of the centroidal Voronoi diagram, which attempts to space all points evenly, and the density function ρ, which restricts points to the essentially one-dimensional edge.

However, a more interesting test is to apply the method with low stipple counts. Smaller numbers of stipples mean that we cannot rely upon the eye to fuse the tiny size and spacing of the dots into a continuous tone. Figure 6 shows an image of an artist's mannequin and the stippled version with 1000 stipples. Figure 7 shows a climbing shoe in the same format. Note that both the stipple drawings are quite recognizable, especially in comparison to Figure 8, where the source images have been reduced in resolution until they contain approximately 1000 pixels each[2].

[2]This comparison is not quite fair, as the 1000 pixels are forced to be equally spread across the image whereas the stipples are free to move. The point is that the stippling maintains edges and silhouettes even at very low resolutions.

Figure 7: Climbing shoe with 1000 stipples of radius 5×10^{-3}

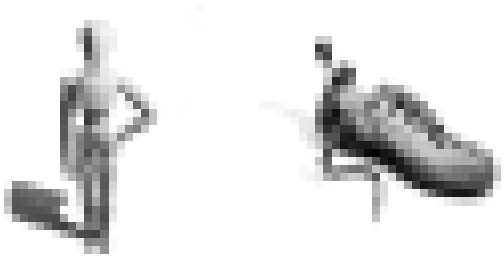


Figure 8: Source images of Figures 6 and 7 rendered with approximately 1000 pixels instead of stipples

Figure 9: Climbing shoe with 5000 stipples of radius 3×10^{-3}

Finally, we note that the most striking drawings come from neither very-high nor very-low numbers of stipples, but medium ranges. Figure 9 shows the climbing shoe of Figure 7 rendered with 5000 stipples. This drawing seems to both reproduce the range of tones from the original and have the "feel" of a real stipple drawing. Figure 10 shows a corn plant rendered with 20000 stipples and displaying both colouration on the leaves and sharp boundaries on the edges. We feel that this image begins to live up to the quote by Hodges in [1989], page 111, in which he attests to the vibrancy of stippled images: "Like a pointallist painting, the drawing will appear to vibrate slightly."

3.2 Parameters and Timings

We computed all the stipple drawings of Section 3 on an Intel Pentium III 1000 MHz machine with 256 Mb of RAM and a NVIDIA GeForce2 MX graphics accelerator. As discussed in Section 2.3, we require the Voronoi regions to have an average area of at least 500 pixels, which forces a virtual resolution of up to 3600 by 3600 pixels for the 20000 stipple drawings. Since we precompute the integrals P and Q from Section 2.2.1 at full virtual resolution, this requires upwards of 100 Mb of memory. The memory requirement could be reduced by an order of magnitude by computing the integrals in tiles in the same way that the Voronoi diagrams are computed, but this did not seem necessary.

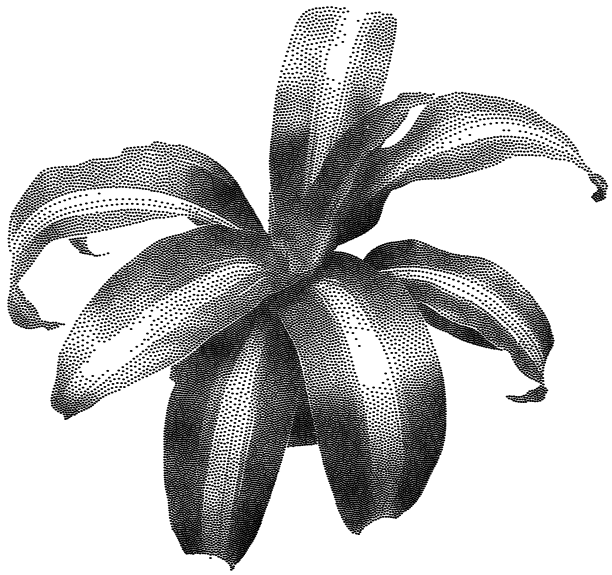


Figure 10: Corn plant with 20000 stipples of radius 1.5×10^{-3}

The iterations were stopped and the stipple drawing output when the difference in the standard deviation of the area of the Voronoi regions was less than 1×10^{-4}. Because the background of the input images was not always pure white, stipples were only output if the input image value at that location was greater than 99% of pure white.

On the system used, the stipple drawings with up to 5000 stipples completed in under a minute and the drawings with 40000 stipples complete in about 20 minutes on an otherwise unloaded machine. The 1×10^{-4} stopping limit was arbitrarily chosen and different values will lead to different runtimes.

4 Precomputing Stipple Levels

The method presented in Section 3 can produce excellent stipple drawings given enough time for algorithm (1) to converge. Clearly a faster algorithm is needed to compute stipple drawings at interactive rates. We can accomplish this, albeit at a cost in quality, by precomputing sets of stipples and stitching them together at runtime.

4.1 Stipple Levels

We will call the result of stippling an image of constant tone t the *t stipple level*, $0 \leq t \leq 1$. To generate the t stipple level, we simply use the method of Section 3 with $\rho = 1$ and $\frac{N}{1-t}$ stipples, where N is the number of stipples required in a pure black image[3,4]. Figure 11 shows nine stipple levels from black to white of a distribution of 1000 stipples, each differing by 125 stipples. Typically we would

[3] We can actually set ρ to any constant value at all, since equation (1) is insensitive to scalings of ρ.

[4] The number of stipples required for a pure black image can be computed by considering the optimal hexagonal packing of discs in the plane and expanding their radii until they completely overlap.

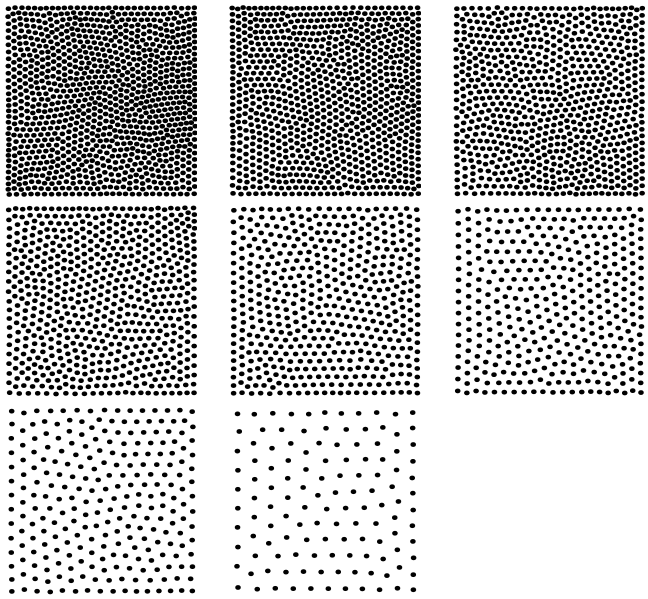


Figure 11: Nine discrete stipple levels with a maximum stipple count of 1000, differing by 125 stipples. The radii of the centre stipple level has been calibrated to represent a 50% gray.

generate a greater number of levels, say 256. We discretise the normally continuous range of tones achievable into a limited number of fixed stipple distributions.

4.2 Fast Stipplings

Using the precomputed stipple levels, we can quickly stipple an image using the following algorithm:

Algorithm 2 Discrete Stippling

for all pixel positions $(x,y) \in [0,1] \times [0,1]$ **do**
  Map image value at (x,y) to stipple level l
  Copy stipples on level l inside $(x-\frac{1}{2}, y-\frac{1}{2}) \times (x+\frac{1}{2}, y+\frac{1}{2})$ to output
end for

Algorithm 2 examines the value of each pixel, determines which stipple level is appropriate, and copies all the stipples that fall inside the area covered by the pixel to the output. Since the input image is processed in scan-line order, we sort the stipples in a particular level into bins that cover a single row of pixels, and then sort the stipples in each bin from low x values to high. Given a particular pixel and a particular level this allows us to quickly find the appropriate stipples.

Conceptually, we split the image into a number of regions to be represented by a single stipple level, then stipple each region individually and recompose the stippled regions into a final drawing. The algorithm is quite fast, but is limited by the amount of memory which must be scanned to produce a stipple drawing. Table 1 shows approximate timings on the system described in Section 3.2 for a simple animation loop. The animation in this case was rendered by OpenGL and read back from its buffers, illustrating the flexibility of using images as input. While increasing the numbers of stipples rendered does have a negative effect on the speed, the greatest factor is the image resolution.

Figure 12 compares the results of the fast algorithm to the high-quality algorithm. On the left are the fast stipple drawings of a black-to-white ramp and a lit sphere and on the right are the high-quality versions. Note on the left the many voids and overlapping stipples on the left that introduce spurious detail. They are the result of two regions of the image being stippled with two different stipple levels. The stipple levels cannot merge smoothly since they have different densities of stipples. The result is a pattern that is not as smooth as it should be.

	5000	10000	20000	40000
100×100	350 fps	300	200	150
300×300	150 fps	120	100	80
600×600	60 fps	45	40	35
900×900	20 fps	20	20	18

Table 1: Frames per second at various numbers of stipples and resolutions

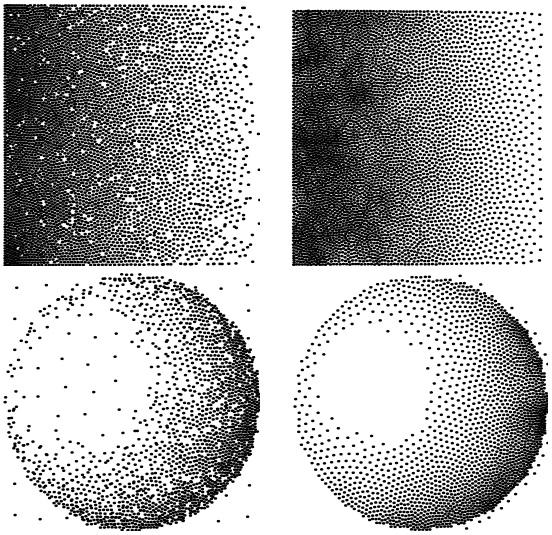


Figure 12: A black-to-white ramp and a lit sphere stippled with the fast algorithm of Section 4.2 on the left and the high-quality algorithm of Section 3 on the right.

In addition, what can not be seen from Figure 12 is the temporal discontinuities that arise when the method is used to stipple an animation. In areas of the image where the tonal value is changing quickly, the pixels get stippled by many different stipple levels in a short time. Even if great care is taken to minimize the differences between one stipple level and the next, the rate at which the pixels change cause them to "shimmer." These problems are minimized by using greater numbers of smaller stipples in the animation.

5 Conclusions and Future Work

We have extended the work on stippling algorithms by introducing a scheme based on weighted centroidal Voronoi diagrams. The presented algorithm has very few user-specified parameters and requires no user interaction. In addition, the input data are grayscale images which can be produced by a wide variety of sources. Apart from simply requiring less work to generate a given stipple drawing, this independence allows cheap stippling to be used in a wider variety of situations than before.

The extension to precomputed stipple levels and thus real-time performance exacted a heavy cost in terms of visual quality, both in still images and in terms of inter-frame coherence of animations. The stipple levels are similar in concept to Praun et al.'s Tonal Art Maps (TAMs) used to hatch 3D objects in [2001]. It should be

straight-forward to generate high-quality stippling TAMs and use their approach to investigate frame-coherent animation. However, this would require abandoning our general image-based approach for 3D models.

The tone generated by a set of stipples is problematic and should be investigated further. All the results in this paper use a rational yet ad-hoc method to set the constant radii of the stipples in a particular image. A better understanding of the relationship between stipple radius, spacing and perhaps colour and the resulting perceived tone is required.

In addition, several interesting extensions to the current algorithm could be investigated, including varying the size of the stipples in a single drawing, and the use of colour stipples. Partially transparent blended stipples could be used to alleviate some of the problems with stippling animations.

6 Acknowledgements

The author would like to acknowledge the extremely helpful nature of the reviewers' comments.

References

DEUSSEN, O., HILLER, S., VAN OVERVELD, C., AND STROTHOTTE, T. 2000. Floating Points: A Method for Computing Stipple Drawings. *Computer Graphics Forum 19*, 3 (August).

DU, Q., FABER, V., AND GUNZBURGER, M. 1999. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Review 41*, 4 (Dec.), 637–676.

HAUSNER, A. 2001. Simulating Decorative Mosaics. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 573–578.

HODGES, E., Ed. 1989. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold.

HOFF III, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 1999. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of SIGGRAPH 99*, ACM Press / ACM SIGGRAPH, New York, A. Rockwood, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 277–286.

JASTRZEBSKI, Z. 1985. *Scientific Illustration*. Prentice-Hall.

MCCOOL, M., AND FIUME, E. 1992. Hierarchical poisson disk sampling distributions. In *Proc. of the Graphics Interface '92*, 94–105.

OKABE, A., BOOTS, B., AND SUGIHARA, K. 1992. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley & Sons.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-Time Hatching. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 579–584.

ULICHNEY, R. A. 1988. Dithering with blue noise. *Proceedings of the IEEE 76*, 1, 56–79.

WOO, M., NEIDER, J., AND DAVIS, T. 1997. *OpenGL®Programming Guide*, second ed. Addison-Wesley.

WOOD, P. 1994. *Scientific Illustration*, second ed. Van Nostrand Reinhold.

Fine Tone Control in Hardware Hatching

Matthew Webb
Princeton University

Emil Praun
Princeton University

Adam Finkelstein
Princeton University

Hugues Hoppe
Microsoft Research

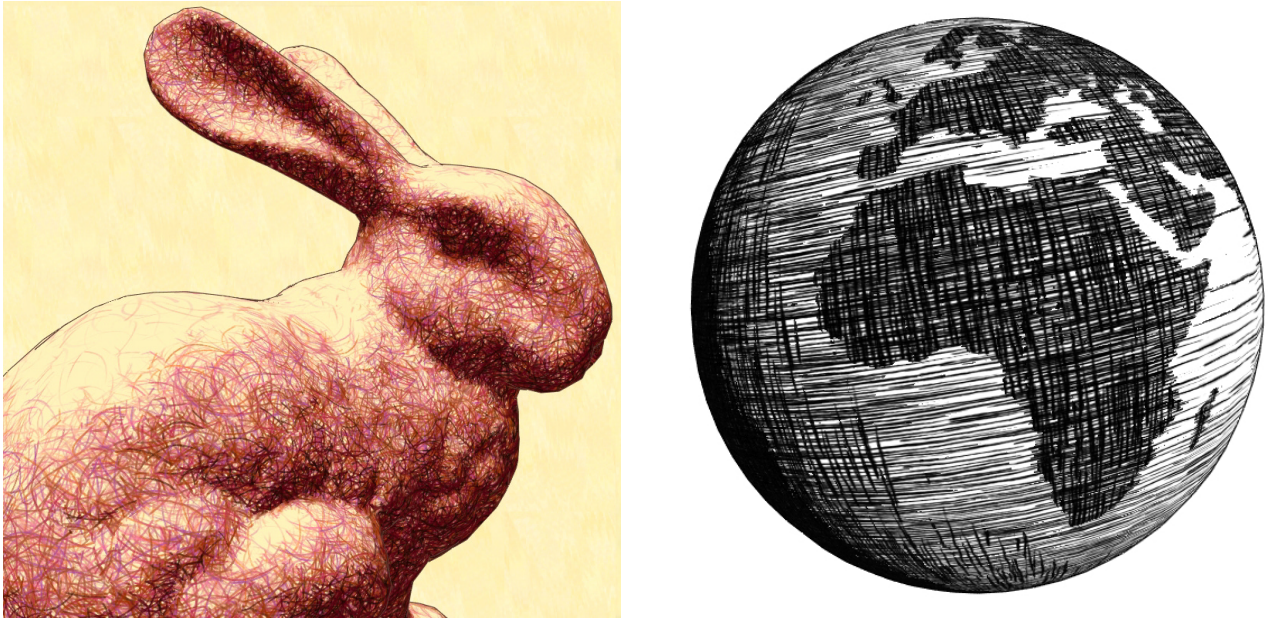


Figure 1: Bunny rendered using color volume texture; globe rendered using threshold textures (modulated per-pixel).

Abstract

Recent advances in NPR have enabled real-time rendering of 3D models shaded with hatching strokes for use in interactive applications. The key challenges in real-time hatching are to convey tone by dynamically adjusting stroke density, while controlling stroke size and maintaining frame-to-frame coherence. In this paper, we introduce two new real-time hatching schemes that leverage recent advances in texture mapping hardware. Both schemes provide enhanced control of tone, thereby avoiding blending or aliasing artifacts present in previous systems. The first scheme, which relies on volume rendering hardware, admits the use of color. The second scheme, which uses pixel shaders, allows per-pixel lighting operations such as texture modulation. Both schemes run at interactive rates on inexpensive PC graphics cards.

Keywords: non-photorealistic rendering, line art, multitexturing

1. Introduction

A variety of non-photorealistic rendering styles use hatching strokes to convey tone (through stroke density), suggest material (through stroke arrangement), and reveal shape (through stroke orientation). Interactivity presents a number of challenges for applications using non-photorealistic rendering: (1) limited run-time computation, (2) frame-to-frame coherence among strokes, (3) control of stroke size and density under dynamic viewing conditions. Two recent algorithms have leveraged advances in hardware texturing capabilities to enable the use of hatching strokes in interactive applications [Freudenberg 2001; Praun et al. 2001]. However, to achieve fine tone control, these systems have suffered from a tradeoff between temporal aliasing and blending artifacts.

In this paper we present two new real-time hatching schemes that extend our previous work on *tonal art maps* (TAMs) [Praun et al. 2001]. By providing greater control over the introduction and removal of strokes in the image plane, both schemes offer finer control over tone. In addition, each new scheme exploits features of modern texture mapping hardware to enable stroke-based rendering effects that were unavailable with previous methods (Figure 1):

- The first scheme exploits *volume texturing* hardware to permit finer tone control, as well as use of **color** hatching strokes.
- The second scheme extends the *texture thresholding* method of Freudenberg [2001], by using multiple thresholds to reduce aliasing artifacts while permitting **per-pixel lighting** operations.

The remainder of this paper is organized as follows. Section 2 offers a brief survey of related work. Section 3 describes in detail the implementation of the two new schemes, and offers some comparisons. Section 4 describes a new method for creating TAMs with color as well as more tonal and character variation than in our previous method. Finally, Section 5 presents results and Section 6 concludes with areas of future work.

2. Related work

There have been a number of systems that use hatching for NPR.

Off-line hatching. Several systems address the problem of generating high-quality hatching for static scenes in an off-line process. Saito and Takahashi [1990] describe a method for post-processing the framebuffer to overlay image-space strokes. Winkenbach and Salesin [1994], and Salisbury et al. [1997] introduce prioritized stroke textures, which map tone values to arrangements of strokes, and present impressive examples of computer-generated hatching. Sousa and Buchanan [1999; 1999] concentrate on the technical aspects of physically simulating real media such as pencil, crayon, blenders, and erasers. Hertzmann and Zorin [2000] create high-quality silhouettes, and describe an image-space stroke placement scheme for cross-hatching.

Real-time hatching. Durand et al. [2001] create hatched images from photographs in real-time using hardware acceleration to perform anti-aliased thresholding. A few recent systems have addressed real-time hatching of 3D models. Markosian et al. [1997] introduce a simple hatching style indicative of a light source near the camera, by scattering a few strokes on the surface near (and parallel to) silhouettes. Elber [1999] shows how to render line art for parametric surfaces in real time; he renders objects by choosing a fixed density of strokes on the surface. Lake et al. [2000] describe an interactive hatching system with stroke coherence in image space (rather than in object space). Freudenberg's approach [2001] consists of coding a stroke texture as a halftone pattern. To shade a pixel, the "height" of the corresponding location in the pattern is compared to the pixel's target tone, using a "soft" threshold function (a clamped linear function with high slope, instead of a step function). This approach inspired our own thresholding scheme in Section 3.2, which encodes multiple thresholds per texel for anti-aliasing.

Real-time hatching with TAMs. In previous work [Praun et al. 2001], we described how prioritized stroke textures could be rendered efficiently using texture hardware by precomputing a *tonal art map* (TAM). The images in a TAM capture hatching at various tones and scales. For visual continuity in an interactive system, we used multitexturing to blend the TAM images over each triangle. Due to hardware limitations, our system could support TAMs with only 6 different tone textures, and these textures were constrained to be grayscale. In this paper we propose two new rendering schemes that are able to utilize TAMs with finer resolution in the tone dimension, and one of the schemes naturally supports colored hatching.

3. New rendering schemes

We now present our two new rendering schemes, and compare their benefits and drawbacks.

3.1 Volume texture scheme

Recent graphics cards support volume textures, whereby a third texture coordinate r is added to the traditional (s,t) to perform lookup within a 3D texture space. Our first rendering scheme uses this third dimension r to encode tone. At load time, TAM images are simply stacked up along the tone axis of the texture volume.

On polygons with large tone variation, our previous scheme would only do linear blending between the 2D textures corresponding to the extreme tone values to be represented, producing many gray strokes. The volume texture method, however, more effectively reproduces all the intermediate tones, since the 3D texture lookup can access all tone levels of a dense TAM. If the set of TAM images is sufficiently dense, the resulting rendering will give the illusion that strokes are added independently, rather than added in blended groups as in [Praun et al. 2001]. For this versatility however, we pay the price of larger texture memory consumption.

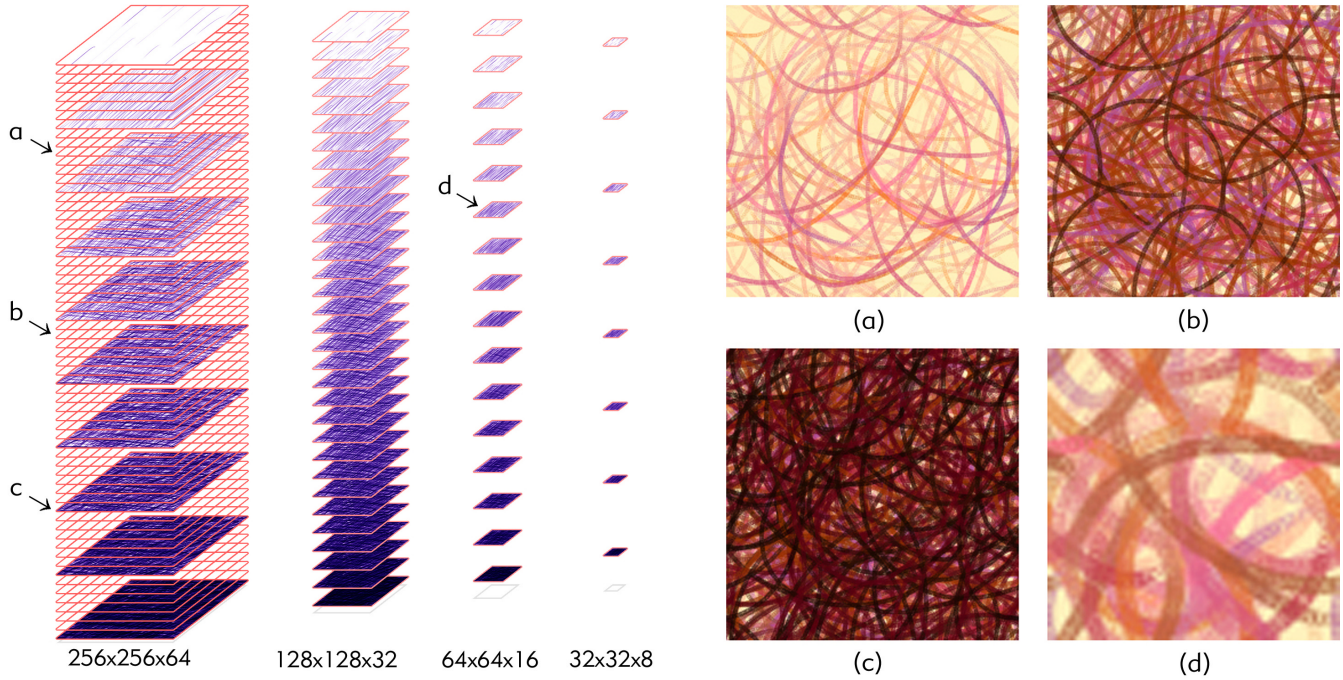


Figure 2. Left: tonal art map (TAM) pyramid. Right: example textures used on Figure 1 bunny (tones and scales indicated at left).

Another advantage of volume texturing is the support of color. Both our original scheme [Praun et al. 2001] and our texture threshold scheme (Section 3.2) maximize the number of reference tone images by packing them into the R,G,B,A channels of 2D textures. This packing limits the one-pass version of the schemes to grayscale strokes, requiring multi-pass implementations to render color.

Since we are using volume textures in a non-standard way, we need to take into account several aspects related to mipmapping. For our application, the ideal filtering behavior would treat the spatial dimensions separately from the tone dimension, i.e. maintain full tonal resolution even as spatial resolution decreases. Unfortunately, current hardware does not offer this behavior. (Disabling filtering altogether is not acceptable since it leads to aliasing.) There are two effects of letting the mipmapping in the tone dimension be influenced by spatial resolution: at coarse mipmap levels we lose both tone resolution and tone range.

The loss of tone resolution is not necessarily detrimental, as long as we start with enough resolution at the finest level. As the object takes up less screen space, it is harder to notice tone variation, so reducing the tone resolution is quite natural. In our examples, we used a 256×256×64 volume. The coarsest spatial level that we generate in our TAM is 32×32 (as strokes are not discernible in coarser levels), corresponding to a resolution of 8 in the tone dimension.

The loss of tone range is caused by the relationship between texture coordinates and texture samples: the first and last samples in a dimension do not correspond to coordinates 0 and 1 respectively, but to $1/2^{\ell+1}$ and $1 - 1/2^{\ell+1}$, for a mipmap level of resolution 2^{ℓ}. For coordinates 0 and 1, texturing returns a 50% interpolation between the first (and respectively, last) sample and the border. Therefore, the range of tones that we can represent without border interpolation is different for each mipmap level. Using a texture with border compresses the overall range of tones available to us, and forces software rendering in our graphics driver. Instead, we perform a border interpolation correction using register combiners. This interpolation uses 100% border contribution for the texture coordinates extremes (0 and 1), rather than the default 50%.

When filling up the volume we need to follow the standard mipmapping sampling pattern. Consider the example where we want 64 levels at the finest resolution. We generate a TAM as in [Praun et al. 2001] with 128 columns. The finest level planes are assigned the 256×256 images with tones 1/128, 3/128, 5/128, … 127/128. The next resolution level are assigned the 128×128 images with tones 2/128=1/64, 3/64, 5/64, …63/64, and so on. Thus, each tone level appears in exactly one mipmap level.

3.2 Texture threshold scheme

Our second rendering scheme extends the method presented by Freudenberg [2001] to produce better anti-aliasing. His method uses a halftoning approach: it stores a texture containing, for each texel, the threshold tone at which a screen pixel should change from white to black. To provide some antialiasing, the method uses a "soft" threshold function (clamp(1-4(1-(I+T))) for an input intensity I and a threshold value T). This soft threshold function works well when the change in tone is achieved by varying the width of the stroke. However, when modulating tone by adding or removing strokes, aliasing artifacts become visible, particularly with thin or overlapping strokes, and in animations. In [Praun et al. 2001], we experimented with thresholding the framebuffer to generate a traditional pen-and-ink style, but encountered similar aliasing artifacts.

When drawing correctly antialiased strokes, most of the stroke pixels should be black, but the few pixels at the stroke boundary that receive only partial coverage should be drawn in gray. (Only when a subsequent hatching stroke covers these pixels might they change to full black.) To capture this behavior, we propose to represent for each texel a piecewise constant function that maps input tones into gray values for the texel. This function therefore has several transitions, rather than a single transition as in conventional halftoning. (See Figure 3.) To render a surface, for each screen pixel we compare its tone (obtained from Gouraud interpolation) with each of the transition X values (obtained from texture lookup). We then take the sum of the heights of all the transitions that pass their comparison tests: [1]

$$\text{Pixel value} = 1 - \sum_i \left(\begin{cases} \Delta y_i & \text{if } (1 - I) > x_i \\ 0 & \text{otherwise} \end{cases} \right)$$

This scheme introduces strokes one by one, much like the volumetric method. In fact, one can view these mapping functions as run-length encodings of rows of texels parallel to the tone dimension in the volume texture from the previous section. The volume texture is rather coherent: a texel keeps its shade for large tone intervals, between the events when different strokes touch the same texel. Since one goal of TAM generation is spatial uniformity, such events are placed as far apart as possible in the tone dimension, leading to large spans of constant values in the volume.

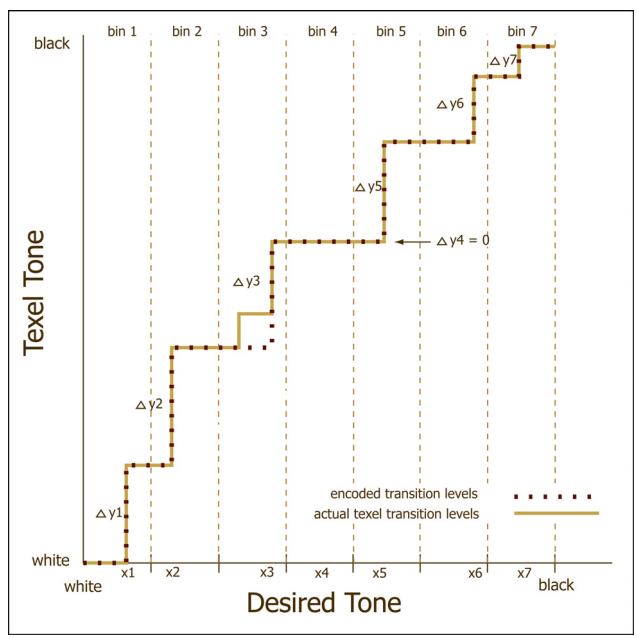


Figure 3: Transition diagram for a single texel.

[1] This function can be implemented on a GeForce3 using register combiners. Simultaneous thresholding of several values can be done by multiplying the 8-bit fixed point colors with 256 (by chaining scale_by_four()'s); while sum of products can be implemented efficiently using dot products. The double inversion (1- intensity; 1-sum) is needed because frame buffers represent "amount of white" while we want to be adding "blackness" (corresponding to black strokes on white paper). Without the inversions, our highlights (light regions on the models) will appear drawn with many overlapping white strokes on a dark canvas, rather than as a white canvas with no strokes.

Since the value of I does not influence which x_i and Δy_i texture locations to address, I can in fact be a more complicated function. For instance, we can modulate I per-pixel with a texture, to produce effects such as the hatched earth globe shown in Figure 1, without affecting the triangulation of the model (as would be necessary in a scheme that could only compute I at vertices).

One problem to consider when representing such transfer functions using textures is (tri-)linear interpolation. If two neighboring texels have the same set of x_i's, interpolating the corresponding Δy_i's yields the correct result. Unfortunately, this doesn't hold for interpolating x_i values. To reduce artifacts, we try to only interpolate between close x_i values: we divide the intensity interval into several bins (not necessarily of equal length), and for each texel only allow a single transition in each bin. Consequently, when different x_i's from adjacent texels are blended together, they can differ only by at most the bin width. In our implementation we used 7 bins corresponding to at most 7 transitions, which we packed in the RGBA channels of 4 textures (we reserve $4\times4 - 7\times2 = 2$ channels for the modulate mask and the splotch mask for a lapped parameterization [Praun et al. 2000]).

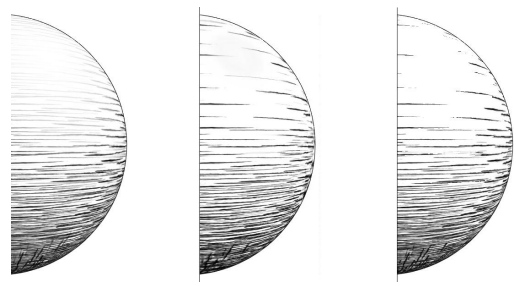
Since strokes are placed uniformly across the TAM textures, it is infrequent that a texel undergoes more than one transition in the same bin. When that happens, we store Δy to be the sum of the transition heights, and randomly pick x_i from among those in the bin. Since more strokes are placed at the dark end of a TAM, we make the bins smaller at the dark end of the spectrum than at the light end. While this binning scheme tends to work well under the assumptions stated, it can lead to banding artifacts when these don't hold. When the TAM is made of many small, thin strokes, the average number of strokes touching a texel increases, and therefore, the number of transitions increases as well. In the limit, when trying to represent a continuous mapping function (no strokes — just 256 gray levels), forcing a representation using only 7 discrete transitions, spaced according to our bin distribution, produces models shaded with only 7 levels of gray, appearing in 7 bands. Choosing x_i's at random within each texel's bins helps make the band boundaries more rough than choosing the mean or average.

3.3 Comparison of the two methods

Both methods presented in this paper offer an improvement over our previous scheme [Praun et al. 2001], by allowing finer control over tone reproduction. Since we have many more TAM columns (samples in the tone dimension), and since each pixel, rather than each vertex, determines the samples to be blended, we can give the illusion of adding (or growing) each stroke individually, in addition to fading in large waves of strokes, as was previously possible. Preference for one effect over another is an aesthetic decision we can now offer the designer.

Figure 4 shows a comparison of the three methods, using different representations of the same 255-column TAM. Our previous method (Figure 4a) has large areas of gray strokes. In the volume rendering approach (Figure 4b) there are a few gray strokes caused by tri-linear interpolation (due primarily to mipmapping rather than tone interpolation). The threshold scheme (Figure 4c) has no gray strokes, only gray pixels around strokes for anti-aliasing. This is due to the fact that interpolation happens on the thresholds, before being compared to the tone. An artifact of this is the presence of a few thin strokes that don't get anti-aliased, since their boundary pixels do not pass the test using the interpolated thresholds.

While the threshold scheme uses less memory, it is actually slightly slower to render than the volume approach, since it involves accessing more textures per pixel. However, it gives us the opportunity for interesting per-pixel effects, such as modulating tone using a different texture. With fewer threshold bins or additional texture accesses on future hardware, one could integrate more complicated effects such as bump mapping and Phong shading. Using these effects with the volume texture rendering approach may be possible in the near future, on graphics cards that allow more complicated dependent texture accesses. Another feature that is likely to be available soon is anisotropic filtering of volume textures; its absence causes the slightly blurrier regions near the silhouettes in Figure 4b.



(a) [Praun et al. 2001] (b) volume (c) 7 thresholds

Figure 4: Comparison of rendering schemes.

One of the advantages of the volume rendering approach is the ease of integrating color. This opportunity raises an interesting artistic question: what can we convey with color that we cannot convey with tone alone? While we do not offer a substantial answer to this question, we have experimented with choosing a path through the color cube, parameterized by luminosity (tone). We have chosen the hues for the colors along this path from compatible color palettes.

4. Fine-level TAM generation

Both rendering schemes require the construction of a TAM that is much denser along the tone axis than in [Praun et al. 2001] (as many as 255 tones[2] instead of 6 tones). This can be constructed using the algorithm described in our previous paper. Obtaining more tone levels does not require any more pre-processing time, since the same number of candidate strokes are still added; we simply "snapshot" more TAM images during the process.

In the remainder of this section, we present an alternative method for TAM generation that allows the user more control and more expressive power. We generate the finest levels of the TAM using a high-quality drawing package, by placing strokes in an image to achieve gradually darker tones. An automated process then replays the sequence of strokes, and selects images corresponding to the tones we want to represent at the finest TAM level. From this, we then construct the coarser levels of the TAM. This scheme works particularly well as it leverages the strengths of the artist and computer to compensate for the other's weakness. The artist need not be overly concerned with the mechanics of TAM generation; he or she simply works on a single texture, drawing a sequence of strokes until satisfied with the range of tones. The computer then handles the task of selecting subsets of strokes to form each image in the volumetric TAM (a task that is prohibitively tedious for a user to undertake).

[2] The number 255 is due to the precision currently available in commodity graphics hardware.

Figure 5: Results. Top two rows: volume texture rendering. Bottom row: thresholds rendering.

In order to maintain coherence and tone at each level of the mipmap volume, it is important to select correlated sets of strokes. The image at a given level and tone (ℓ, t) should consist of the set of strokes in the next lightest tone at the same resolution (ℓ, t-1) plus some subset of the strokes used in the same tone at the next highest resolution (ℓ+1, t). Since the resolution decreased for the new level, the strokes are relatively larger, so fewer of them will be needed for the same coverage, or tone difference. For grayscale TAMs, we can simply select a prefix of the stroke sequence. However, for color TAMs representing a path parameterized by tone through the color space, taking the prefix that produces the desired tone difference will very likely give us the wrong hue. In this case, we first decimate the stroke sequence (throw out a constant fraction of randomly selected strokes), and then take the prefix. In theory, one could do a binary search to find the right fraction for each TAM or even for each image (this fraction depends on stroke properties such as aspect ratio), in practice though we have found that choosing a constant fraction works well, given that we are sampling small tone steps.

5. Results

Figure 5 shows several stills produced with our system. The accompanying video shows short animations of these models.

The hand image is drawn using a style reminiscent of chalk and charcoal. Following artistic conventions, the highlight strokes are hatched in a single direction whereas the shading also employs crosshatched strokes. The fruit bowl image uses an ink texture in which overlapping strokes combine to increase darkness. This differs from the hand image in which overlapping strokes do not darken the surface. The color stipple pattern used on the gargoyle model was an interesting artistic experiment, since it produced the widest range of reactions from the people we have shown it to. It lessens the illusion of a growing front of strokes, since the length of the stroke is short enough that new strokes are distinct from existing strokes. Finally, for the rocker arm, we tried to achieve a look evocative of mechanical sketch.

The bottom row of Figure 5 shows two examples of objects rendered using threshold textures. When these objects rotate, the strokes give the appearance of growing into the highlight regions. The crisp black and white aspect of the strokes is reminiscent of a hand drawn pen-and-ink style.

The Earth image in Figure 1 shows the integration of threshold textures and per-pixel modulation with a map texture. The bunny of Figure 1 is drawn using short arcs with random orientations. When animated, these strokes provide a different impression from the other models: since they grow in different directions, there is no illusion of an advancing front of strokes.

All these models render at around 30-40 frames per second on our GeForce3 card. This includes time spent extracting the silhouettes and drawing the background. The original models have between 7,500 and 15,000 faces. For all models except the Earth globe and the fruit bowl, we created a lapped texture parametrization. The objects in the fruit bowl were created using spline patches, and we used their intrinsic u,v parametrization.

The 6-column TAM used in [Praun et al. 2001] required 800KB of texture memory. By comparison, the volume texture requires 15MB (or 20MB when keeping alpha for border correction), and the threshold textures take up 1.8MB.

6. Conclusions and future work

We have presented two methods to improve the quality of interactive hatch renderings. Both methods provide fine tone control. Volumetric textures allow for greater user expression by adding the ability to render color hatchings. Threshold textures store a discrete set of tone transitions per texel, supporting hatch rendering with fine tone control and anisotropic filtering with far less memory consumption. This is at the cost of restricting the hatched models to grayscale images.

We have found that harsh polygonal silhouettes are often the largest factor in associating the rendering with a 3D model. We would like to investigate methods draw smooth stroke-based silhouettes that complement the volumes textures.

The current implementation of volume TAMs utilizes a large amount of texture memory. Since volume TAMs have an extremely high degree of coherence by definition, it may be possible to greatly reduce the amount of memory consumption.

We would also like to investigate methods that provide a provable error bound on tone and hue among the different mipmap levels.

Furthermore, we are interested in rendering entire scenes instead of single objects. This introduces new opportunities to explore other artistic techniques as haloing and shadowing.

References

DURAND, F., OSTROMOUKHOV, V., MILLER, M., DURANLEAU, F., AND DORSEY, J. Decoupling Strokes and High Level Attributes for Interactive Traditional Drawing. *Proceedings of Eurographics Rendering Workshop* 2001, pp. 71-82.

ELBER, G. Interactive Line Art Rendering of Freeform Surfaces. *Computer Graphics Forum* 18, 3 (September 1999), pp. 1–12.

FREUDENBERG, B. Real-Time Stroke Textures. (Technical Sketch) *SIGGRAPH 2001 Conference Abstracts and Applications*, p. 252.

HERTZMANN, A., AND ZORIN, D. Illustrating Smooth Surfaces. *Proceedings of SIGGRAPH 2000*, Computer Graphics, Annual Conference Series, pp. 517–526.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. Stylized Rendering Techniques for Scalable Real-Time 3d Animation. *Proceedings of NPAR2000*, pp.13–20.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. Real-time Nonphotorealistic Rendering. *Proceedings of SIGGRAPH 97*, pp. 415–420.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. Lapped Textures. *Proceedings of SIGGRAPH 2000*, Computer Graphics, Annual Conference Series, pp. 465-470.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. Real-Time Hatching. *Proceedings of SIGGRAPH 2001*, Computer Graphics, Annual Conference Series, pp. 579-584.

SAITO, T., AND TAKAHASHI, T. Comprehensible Rendering of 3D Shapes. *Proceedings of SIGGRAPH 90*, pp. 197–206.

SALISBURY, M. P., WONG, M. T., HUGHES, J. F., AND SALESIN, D. H. Orientable Textures for Image-Based Pen-and-Ink Illustration. *Proceedings of SIGGRAPH 97*, pp. 401–406.

SOUSA, M. C., AND BUCHANAN, J. W. Observational Model of Blenders and Erasers in Computer-Generated Pencil Rendering. *Proceedings of Graphics Interface '99*, pp. 157–166.

SOUSA, M. C., AND BUCHANAN, J. W. Computer-Generated Graphite Pencil Rendering of 3d Polygonal Models. *Computer Graphics Forum* 18, 3 (September 1999), pp. 195–208.

WINKENBACH, G., AND SALESIN, D. Computer-Generated Pen-and-Ink Illustration. *Proceedings of SIGGRAPH 94*, Computer Graphics, Annual Conference Series, pp. 91–100.

Color Plate for “Fine Tone Control in Hardware Hatching”. Top row: Figure 1 and four sample textures from the corresponding volume texture (a,b,c highest detail, d lower detail). Middle and bottom rows: volume texture renderings from Figure 5.

Computer-Generated Pen-and-Ink Illustration

Georges Winkenbach and David H. Salesin
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

23 June 1994

Technical Report 94-01-08b

Abstract

This paper describes the principles of traditional pen-and-ink illustration, and shows how a great number of them can be implemented as part of an automated rendering system. It introduces "stroke textures," which can be used for achieving both texture and tone with line drawing. Stroke textures also allow resolution-dependent rendering, in which the choice of strokes used in an illustration is appropriately tied to the resolution of the target medium. We demonstrate these techniques using complex architectural models, including Frank Lloyd Wright's "Robie House."

Computer-Generated Pen-and-Ink Illustration

Georges Winkenbach *David H. Salesin*

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Abstract

This paper describes the principles of traditional pen-and-ink illustration, and shows how a great number of them can be implemented as part of an automated rendering system. It introduces "stroke textures," which can be used for achieving both texture and tone with line drawing. Stroke textures also allow resolution-dependent rendering, in which the choice of strokes used in an illustration is appropriately tied to the resolution of the target medium. We demonstrate these techniques using complex architectural models, including Frank Lloyd Wright's "Robie House."

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.5 [Computer Graphics]: Three-Dimensional Graphics and Realism — Color, Shading, Shadowing, and Texture.

Additional Key Words: architectural rendering, comprehensible rendering, non-photorealistic rendering, prioritized stroke textures, resolution-dependent rendering, texture indication.

1 Introduction

Most of the research in computer graphics rendering over the last twenty years has been devoted to the problem of creating images of physical scenes with ever-increasing complexity and realism. The success of this research has been a well-heralded achievement in graphics.

However, the computer's ability to display images of ever-increasing complexity gives rise to a new problem: communicating this complex information in a comprehensible and effective manner. In order to communicate truly complex information effectively, some form of visual abstraction is required. This type of abstraction has been studied most comprehensively in the fields of graphic design and traditional illustration.

In this paper, we therefore examine algorithms for the "non-photorealistic" rendering of complex forms. While photorealistic images certainly have their place, in many applications, such as architectural and industrial design, a stylized illustration is often more effective.

The advantages of illustration are numerous. Illustrations can convey information better by omitting extraneous detail, by focusing attention on relevant features, by clarifying and simplifying shapes, or by exposing parts that are hidden. In addition, illustrations often consume less storage than realistic images, and are more easily reproduced and transmitted. Illustrations also provide a more natural vehicle for conveying information at different levels of detail. Finally, in many applications, illustrations can add a sense of vitality difficult to capture with photorealism.

The benefits of illustrations over photographs are well-recognized in many practical contexts. For example, medical texts almost always employ hand-drawn illustrations in place of (or in addition to) photographs, since they allow tiny and hidden structures to be much better described. In addition, most assembly, maintenance, and repair manuals of mechanical hardware employ illustrations rather than photographs because of their clarity. For example, at Boeing, even when CAD databases of airplane parts exist, all high-quality manuals are still illustrated by hand in order to provide more effective diagrams than can be achieved with either photorealistic rendering or simple hidden line drawings [16].

To explore the use of abstraction as a means for conveying information effectively, it makes sense to begin with an area with well established conventions. For this reason, we are beginning our investigation using the domain of pen-and-ink illustrations of architectural forms, for which a great number of well-documented conventions already exist [5, 11, 13, 14, 17, 20]. Restricting the domain to "pen and ink" also has the advantage that no exotic display technology is required to view the algorithms' output: conventional laser printers, even the inexpensive 300 dots-per-inch variety, give quite reasonable results.

In the rest of this paper, we describe a number of principles of traditional pen-and-ink illustration, and we show how a great number of them can be implemented as part of an automated rendering system.

1.1 Related work

The area of "non-photorealistic rendering" has received relatively little attention in the computer graphics community. We survey most of the related work here.

Seligmann and Feiner have described methods for automatically constructing illustrations to achieve a particular communicative goal [24]. Their system is primarily concerned with the high-level goal of composing the best model for communicating a particular intent, whereas the system we describe is more concerned with the low-level details of rendering the model once it is built. Thus, our system could serve as a "back-end" for theirs.

With respect to the rendering of architectural forms, Yessios described a prototype "computer drafting" system for common materials in architectural designs, including stones, wood, plant, and ground materials [26], which, like our work, attempts to provide a warmer, hand-drawn appearance as opposed to a mechanical one. Miyata also gave a nice algorithm for automatically generating stone wall patterns [19]; these patterns would make a good starting point for some of the pen-and-ink techniques described in this paper.

With respect to line-drawing techniques, Appel et al. were the first to discuss how a line could be "haloed" automatically to give the appearance of one line passing behind another [2]. Kamada and Kawai generalized this work by showing how different line attributes, such as dashed and dotted line, could be used to give a more informative treatment of hidden lines [12]. Dooley and Cohen later introduced more line qualities, such as thickness, and discussed how the treatment of outline and surface shading could be customized by a user to create more effective illustrations [6, 7]. In the commercial realm, the Premisys Corporation markets a product called "Squiggle" that adds waviness and irregularities to CAD output as a post-process, lending a hand-drawn appearance to the drawings [21]. The Adobe Dimensions program allows PostScript stroke textures to be mapped onto surfaces in three dimensions [1].

The research described in this paper was most directly inspired by the work of Saito and Takahashi, who introduced the concept of a "G-buffer" for creating comprehensible renderings of 3D scenes [22]. Our work takes a somewhat different approach, in that it integrates aspects of 2D and 3D rendering, whereas their method essentially uses image processing techniques once the set of G-buffers are created. In addition, by introducing methods for texturing surfaces with strokes, the work in this paper extends the repertoire of the types of renderings that can be produced in a purely automated way.

In related works, our group is exploring several different aspects of the pen-and-ink illustration problem. This paper describes the overall vision of computer-generated illustration, surveys principles from traditional illustration, and shows how they can be incorporated into an automated system for rendering 3D models. A second paper discusses the issues of creating pen-and-ink illustrations interactively, with an emphasis on using 2D greyscale images as a starting point [23]; in this interactive work, the responsibility of producing an effective illustration is primarily the artist's. A third paper examines the issues involved in representing, editing, and rendering the individual strokes that are the building blocks of any line illustration system [8].

1.2 Overview

The rest of this paper is organized as follows. Section 2 surveys the principles of traditional pen-and-ink illustration. Section 3 discusses how these principles can be used to guide the design of an automated system for producing this type of imagery. Section 4 introduces "strokes" and "stroke textures," the building blocks of our system, and describes how they can be used to implement many of the traditional illustration principles. Section 5 discusses some of our results, and Section 6 lays out an agenda for future research in the area. Finally, the appendix gives details about the implementation.

2 Principles of pen-and-ink illustration

While pen-and-ink drawing has a long history, dating back to the illuminated manuscripts of the Middle Ages, it is only relatively "recently" — that is, since the end of the 19th century — that pen-and-ink illustration has been developed as an art form in and of itself.

Pen-and-ink illustration is a limiting medium. The pen gives off no color or tone, so both color and shading must be suggested by combinations of individual strokes. Furthermore, when rendered manually, it is very difficult and time-consuming with pen and ink to cover a large area with tone, and it is practically impossible to lighten a tone once it is drawn.

However, pen-and-ink illustrations have some particular qualities that make them especially attractive. First, they are ideal for outlines: each individual pen-and-ink stroke can be made expressive by employing small irregularities in its path and pressure. Second, pen and ink provide a real economy of expression in representing tones and texture: the character of a few small strokes can clearly indicate the difference between textures like smooth glass and old knotted wood.

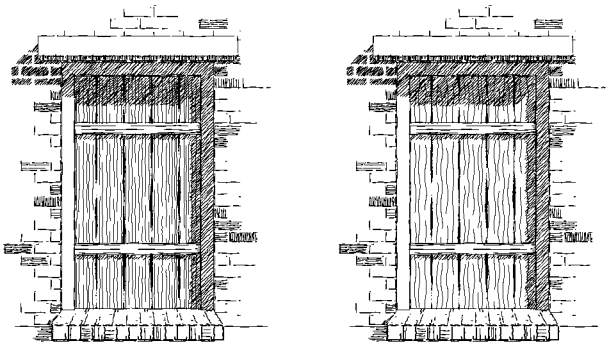


Figure 1: *Two doors. The lines of wood grain are drawn with an even pressure, while the lines between the planks use varying pressure. For wood grain, we typically use the waviness function of the left door, while that of the right door has been exaggerated.*

In addition to these concrete advantages, pen-and-ink drawings by their very nature possess some special qualities that are difficult to capture in other media. Their simplicity provides an appealing crispness and directness. Finally, pen-and-ink illustrations blend nicely with text, due to their linear quality and their use of the same ink on the same paper, making them ideal for printed publications.

In the rest of this section, we survey some of the fundamental principles of illustrating in pen and ink. These principles are distilled primarily from Guptill's classic text, *Rendering in Pen and Ink* [11], and also from Lohan's *Pen&Ink Techniques* [17] and several other sources [5, 13, 14, 20]. While the field of pen-and-ink is too vast to allow a comprehensive treatment within the scope of this paper, the principles described here should be sufficient to motivate many of the design choices for a computer-graphics system. We organize our treatment into three parts: Strokes, Tone and texture, and Outline.

2.1 Strokes

In classical pen-and-ink illustration, a "stroke" is produced by placing the point, or "nib," of a pen in contact with the paper, and allowing the nib to trace out a path. The thickness of the stroke can be varied by varying the pressure on the nib.

Some principles of stroke-drawing are summarized below:

- *Too thin a stroke can give a washed-out appearance; too coarse can detract from the delicate details.*
- *It is frequently necessary to vary the pen position, with the nib sometimes turning as the stroke is drawn.*
- *Strokes must look natural, not mechanical. Even-weight line drawings appear lifeless; instead, the thickness of a line should vary along its length.*
- *Wavy lines are a good way to indicate that a drawing is schematic and not yet completely resolved.*

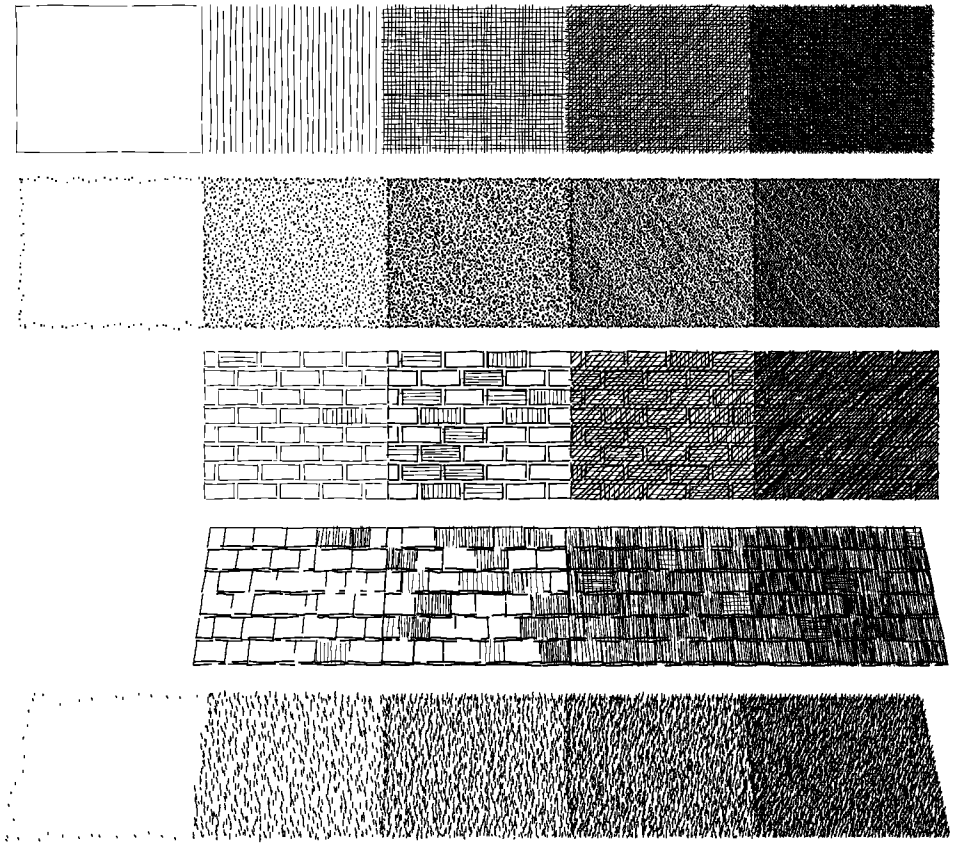


Figure 2: *Using strokes to indicate both texture and tone. The stroke textures used, from top to bottom, are: "cross-hatching," "stippling," "bricks," "shingles," and "grass." Notice how the outline style of the white areas is also particular to each texture.*

2.2 Tones and texture

The terms "value" and "tone" are used interchangeably to refer to the amount of visible light reflected toward the observer from a point on a surface. In traditional pen-and-ink illustration, it is impossible to portray the value of each surface precisely; instead, combinations of strokes are used to create an overall impression of the desired tone.

The tone achieved by a combination of strokes is a function of the ratio of black ink to white paper over a given region of the illustration. If the character of the strokes is varied, then the same strokes that are used to achieve a particular tone can also be used simultaneously to indicate the "texture" of the subject being rendered. This dual role of individual strokes to convey both tone and texture is part of the economy of pen-and-ink illustration.

Here are some of the principles of drawing tones and textures with pen-and-ink strokes:

- *Tones should be created from lines of roughly equal weight and spacing.*
- *It is not necessary to depict each individual tone accurately; however, presenting the correct arrangement of tones among adjacent regions is essential.*
- *To disambiguate objects, it is sometimes important to "force tone" by enhancing contrast or inventing shadows.*
- *The character of strokes is important for conveying texture, as well as geometry and lighting. For example:*
  - *Crisp, straight lines are good for "glass."*
  - *Horizontal surfaces should be hatched with predominantly horizontal lines.*
  - *Absence of detail altogether indicates glare.*
  - *A sketchy kind of line is good for "old" materials, while careful stippling is good for "new" materials.*
- *To lend economy to the illustration, it is important to utilize some form of "indication" for conveying the impression of a texture without drawing every single stroke. The method of indication should also be varied across the drawing to avoid monotony.*

2.3 Outlines

Realistic scenes contain no real outlines; instead, forms are defined by variations in texture and tone. However, outline is nevertheless a very natural means for portraying objects — for example, most children's drawings utilize outline almost exclusively.

The medium of pen and ink is ideal for creating outlines with an incredible range of expressiveness. The pen allows for outlines that change thickness, sometimes disappearing altogether. In addition, the character of the outline stroke can be a very powerful indicator of texture.

Outline strokes are used not only for the contours of an object, but also for delineating the essentials of its interior. For example, in an illustration of a leaf, the veins are typically rendered in outline, in addition to the contour.

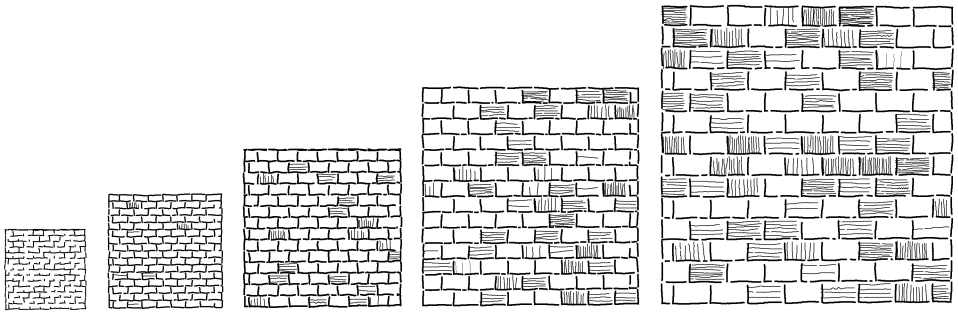


Figure 3: *Creating the same texture and tone at different scales. At the smallest scale, the brick outline strokes are sufficient to build the tone. As the scale increases, the prioritized stroke texture automatically introduces shading inside the bricks to maintain the tone. The same technique applies to generating illustrations at the same scale but for different printer resolutions.*

Different styles of rendering use various combinations of outline and tone; all combinations are possible. Moreover, there exists an entire spectrum between outline and tone illustrations: as outline drawings become increasingly complex, they begin to take on more and more aspects of an illustration with tone.

Here are some of the important classical principles for drawing expressive outlines:

- *The quality of the outline stroke is important for conveying texture. For example, crisp straight lines are good for hard objects, while a greater variety of line quality is better for soft objects.*
- *Thick outlines are used to suggest shadows, or to bring one object or part of an object forward in the scene. Thick line junctions are used to suggest darkness where objects overlap and to add "snappiness" to the illustration.*
- *Outlines should become "haloed" and fade away where one object passes behind another object.*
- *Outlines must be introduced where tones are omitted to convey shape.*
- *Using "indication" for drawing outlines is just as important as for drawing tones.*

3 Computer-generated pen-and-ink illustration

Implementing these principles of pen-and-ink illustration as part of an automated system presents an interesting challenge. A reasonable starting point is to take the traditional "graphics rendering pipeline" for photorealistic imagery and see which parts, if any, need to be altered in order to support this style of non-photorealism.

We identified two fundamental differences:

1. *The dual nature of strokes.* In the traditional graphics pipeline, the renderings of texture and tone are completely independent. A texture is typically defined as a set of images assigned to each surface, which affect the shading parameters. Tone is produced by dimming or brightening the rendered shades, while leaving the texture invariant. However, for pen-and-ink illustration, the very same strokes that produce tone must also be used to convey texture. Thus, tone and texture must become more tightly linked in a system for producing this type of imagery.

2. *The need to combine 2D and 3D information.* In the traditional graphics pipeline, the information used for rendering is entirely three-dimensional, with the final projection to two dimensions largely a matter of sampling the rendered shades. For pen-and-ink illustration, the 2D aspects of the particular projection used are every bit as essential as the 3D information for creating a proper rendering. The necessary 2D information takes a number of forms. First, the size of the projected areas must be used to compute the proper stroke density, in order to accommodate the dual nature of strokes described above. In addition, the 2D adjacencies of the projected geometry must also be used, since outlining depends on such issues as the type of junction between 2D boundaries (whether two adjacent regions in 2D are adjoining in 3D or passing one behind the other), and the level of contrast between tones of adjacent 2D regions.

Thus, our rendering system is a basic graphics pipeline with a few notable changes. The standard aspects of the pipeline include:

- *The model.* Any standard polygonal 3D model will do.
- *The assignment of texture.* Textures are assigned to 3D surfaces in the usual way. However, the textures are no longer described by images, but by "stroke textures," as discussed in the next section.
- *The lighting model.* Any standard illumination model can be employed to compute a "reference solution," which is then used as a target for tone production with strokes. We use the Phong model, which, although not physically-based, appears to be quite adequate for most non-photorealistic rendering.
- *The visible surface algorithm.* Any object-space or list-priority visible surface algorithm will do; we use BSP trees in our implementation.
- *Shadow algorithm.* The shadow algorithm must also use an object-space or list-priority method; we use Chin and Feiner's BSP tree shadow volumes [4].

Here are the notable differences from the standard pipeline:

- *Maintaining a 2D spatial subdivision.* The need to consider 2D adjacency information in rendering suggests the use of some form of spatial subdivision of the visible surfaces. We use a half-edge data structure for maintaining this planar map [18].
- *The rendering of texture and tone.* Polygons are no longer scan converted; instead, both texture and tone must be conveyed with some form of hatching. The stroke textures we define in the next section achieve this effect.
- *Clipping.* The strokes must be clipped to the regions they are texturing. Since so many strokes are drawn, the clipping must be extremely fast. In addition, in order to simulate a hand-drawn

effect, the clipping should not be pixel-based — that is, it should not remove just those pixels of the stroke that are outside the clipping region — since this gives an unnatural, mechanical appearance. Instead, the clipping should be stroke-based, allowing a wavy stroke to sometimes stray slightly outside of the clipping region. To achieve this effect, we clip the straight-line paths of our strokes prior to adding in the function for waviness (see Section 4.1). For fast clipping, we use set operations on a 2D BSP tree representation of the planar map [25].

- *Outlining.* Outlines play a significant role in pen-and-ink illustration. Outlines come in two varieties. The "boundary outlines," which surround visible regions, must be drawn in a way that takes into account both the textures of the surrounded regions, and the adjacency information stored in the planar map. In addition, "interior outlines" are used within polygons to suggest shadow directions or give view-dependent accents to the stroke texture.

A brief description of the rendering process follows; more details about the rendering algorithm can be found in the appendix. To render a scene, the system begins by computing the visible surfaces and the shadow polygons. It then uses these polygons, projected to Normalized Device Coordinate (NDC) space, to build the 2D BSP tree and the planar map. Each visible surface is then rendered. The procedural texture attached to each surface is invoked to generate the strokes that convey the correct texture and tone for the surface. All the strokes are clipped to the visible portions of the surface using set operations on the 2D BSP tree. Finally, the outline strokes are drawn by extracting from the planar map all of the outline edges necessary for the illustration, as described in Section 4.3.

4 Strokes and stroke textures

In this section we discuss strokes and stroke textures, the essential building blocks of our system.

4.1 Strokes

In our system, all strokes are generated by moving a nib along a basic straight path. Character is added to the stroke by perturbing the path with a *waviness function* and by varying the pressure on the nib with a *pressure function*. Figure 1 demonstrates some of the effects that can be achieved with different waviness and pressure functions. A more detailed explanation of our strokes can be found in Appendix A.3.1.

4.2 Stroke textures

A *stroke texture* is a collection of strokes used to produce both texture and tone. We define a *prioritized stroke texture* as a set of strokes each with an associated priority. When rendering a prioritized stroke texture, all of the strokes of highest priority are drawn first; if the rendered tone is still too light, the next highest priority strokes are added, and so on, until the proper tone is achieved.

For our stroke textures, we assign different aspects of the texture different priority. For example, for a "brick" texture, the outlines of the individual brick elements have highest priority, the strokes for shading individual bricks have medium priority, and the hatching strokes that go over the entire surface have lowest priority. In the cross-hatching texture, vertical strokes have priority over horizontal strokes, which have priority over the various diagonal stroke directions. Figure 2 demonstrates several greyscales of tone produced using different procedural prioritized stroke textures, including "cross-hatching," "stipple," "brick," "shingle," and "grass." For each texture, the relative priorities of the strokes can be seen from the collection of strokes used to achieve a particular value of grey. More details about the procedural methods for our stroke textures are given in Appendices A.3.2 and A.3.3.

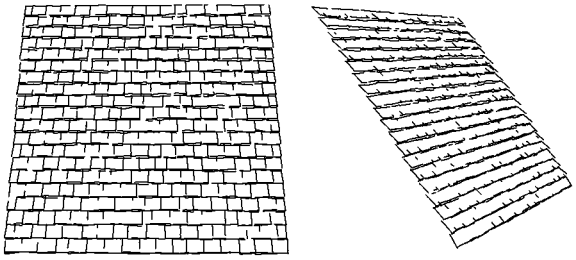


Figure 4: *The effect of changing view direction on outline strokes of a shingle texture. Notice how the vertical edges begin to disappear as the texture is viewed from a more edge-on direction.*

Although not explored in this paper, the idea of prioritized stroke textures is general enough to support many kinds of non-procedurally generated textures as well, such as textures drawn directly by an artist, or strokes produced through edge extraction from a greyscale image. These kinds of non-procedural stroke textures are explored in more detail by Salisbury et al. [23].

4.2.1 Resolution dependence

A common problem with the figures created by existing computer drawing programs is that they do not scale well when printed at different sizes or resolutions. Enlargement is typically performed either by pixel replication, which yields ugly aliasing artifacts, or by drawing the same strokes at higher resolution, which yields thinner strokes and an overall lighter illustration. Reduction is almost always performed by scan-converting the same curves at a lower resolution, often yielding a large black mass of overlapping strokes. Printing speed is also a common problem with illustration reduction, since the same number of strokes needs to be transmitted to and rendered by the printer, even when a smaller number of strokes would have sufficed (and actually have been preferable from an aesthetic standpoint, as well).

The prioritized stroke textures described here do not suffer from these problems. Strokes are chosen to provide the proper texture and tone for a given illustration size and printer resolution, as demonstrated in Figure 3. Note that for smaller images or coarser resolutions, fewer strokes are required, improving printing efficiency. Efficiency can be improved still further by rendering a simplified approximate version of each stroke, accurate to within one printer pixel [8].

4.2.2 Indication

As discussed in the principles of texture generation, it is important to suggest texture without drawing every last stroke. This principle of "indication" lends economy to an illustration. It also makes an illustration more powerful by engaging the imagination of the viewer rather than revealing everything.

Indication is one of the most notoriously difficult techniques for the pen-and-ink student to master. It requires putting just enough detail in just the right places, and also fading the detail out into the unornamented parts of the surface in a subtle and unobtrusive way.

Clearly, a purely automated method for artistically placing indication is a challenging research project.

We therefore decided to compromise and implement a semi-automated method, whereby the user specifies at a very high level where detail should appear in the drawing, and indication is used everywhere else. For easy specification of the areas of detail, we borrowed the idea of using "fields" generated by line segments from the morphing paper of Beier and Neely [3]. The user interactively places "detail segments" on the image to indicate where detail should appear. Each segment is projected and attached to the texture of the 3D surface for which indication is being designed.

A field $w(x, y)$ is generated by the detail segment ℓ at a point (x, y) in texture space according to

$$w(x, y) \quad = \quad (a + b * \textit{distance}((x, y),\, \ell))^{-c}$$

where a, b, and c are non-negative constants that can be used to change the effect of the field. When several detail segments are present, we define the field at a point (x, y) to be that of the closest segment. So as not to create patterns that are too regular, the field $w(x, y)$ is perturbed by a small random value. Textures such as "bricks" and "shingles" evaluate the strength of the field of indication at the center of each brick or shingle element. The set of strokes for that element is generated only if the indication field is above some preset threshold.

This approach seems to give reasonable results, as demonstrated in Figures 6 and 9. Figure 5 shows the detail segments that were used to generate Figure 6.

4.3 Outline

As described in Section 3, outlines come in two varieties: *boundary* and *interior* outlines. The *boundary outlines* surround the visible polygons of the image, and must be drawn in a way that takes into account both the textures of the surrounded regions, and the adjacency information stored in the planar map. The *interior outlines* are used within polygons to suggest shadow directions or to give view-dependent accents to the stroke texture.

In our implementation we have tried to address many of the principles for the effective use of these two types of outline, as described below.

Expressing texture with outline. Each stroke texture $\mathcal{T}$ has associated with it a *boundary outline texture*, which is used whenever the outline of a polygon textured with $\mathcal{T}$ is rendered. The boundary outline textures for some of our procedural textures are demonstrated in the white squares of Figure 2. These boundary outline textures are also displayed with and without their accompanying stroke textures in the illustrations of Figure 7.

Minimizing outline. Let E be an edge that is shared by two faces F and G of a planar subdivision. Our rendering algorithm draws E only if the tones of face F and G are not sufficiently different for the two faces to be easily disambiguated by their shading alone. In this sense, we minimize the use of boundary outline strokes. When a boundary outline stroke *is* drawn, it must be rendered according to the boundary outline texture for one of the two faces F or G. We choose the texture of the face of the planar subdivision that represents a polygon closer to the viewer. Figure 8 demonstrates how outline is omitted in the presence of sharp changes in tone, and added in the absence of tone changes.

Accented outlines for shadowing and relief. "Accenting," or thickening, outline edges is a technique for providing subtle but important cues about the three-dimensional aspects of an illustrated scene. In our implementation, the interior outlines of each brick in the "brick" stroke texture are drawn according to their relationship with the direction of the light source: brick edges that cast shadows are rendered with thickened edges, while illuminated brick edges are not drawn at all. Figure 10 demonstrates this effect.

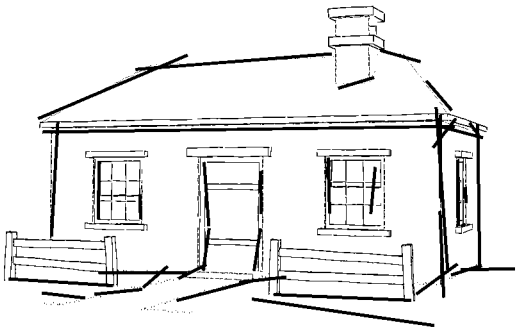


Figure 5: *Detail segments. The user interactively attaches "detail segments" to the surfaces to indicate roughly where details should appear.*

Dependence of viewing direction. In addition to the light source direction, the viewing direction is another important parameter that should be taken into account when drawing outline strokes. For example, consider a roof of shingles. Viewed from above, all edges between individual shingles are clearly visible; viewed from more to the side, however, the shingles tend to blend together, and vertical edges begin to disappear, leaving the horizontal edges predominant. This effect is demonstrated in Figure 4. To implement this effect, each stroke texture is outfitted with a very simplified "anisotropic bidirectional reflectance distribution function" (BRDF), to borrow a term from radiometry, which describes its outline features in terms of both the lighting and viewing directions.

5 Results

Our computer-generated pen-and-ink illustration system was used to create all the figures in this paper. The system was developed on a Macintosh Quadra 700 using ThinkC.

The only input to the program is the scene geometry, including texture assignments for each surface, and some field lines for specifying the "indication."

We also used the system to generate an image of the top two floors of Frank Lloyd Wright's "Robie House," as shown in Figure 9. The model consist of 1043 polygons. It took 30 minutes to compute and print the image. Of this time, 22 minutes were devoted to computing the planar map from the input geometry, and 8 minutes were required for actually rendering the image at 600 dots per inch.

6 Summary and future work

This paper does not propose any radically new algorithms or present any complex mathematics. However, we feel it nevertheless provides a number of contributions to the computer graphics community. These contributions include:

- Surveying established principles from traditional illustration that can be used for communicating visual information effectively.
- Showing that a large number of these principles can be incorporated as part of an automated rendering system, and that the

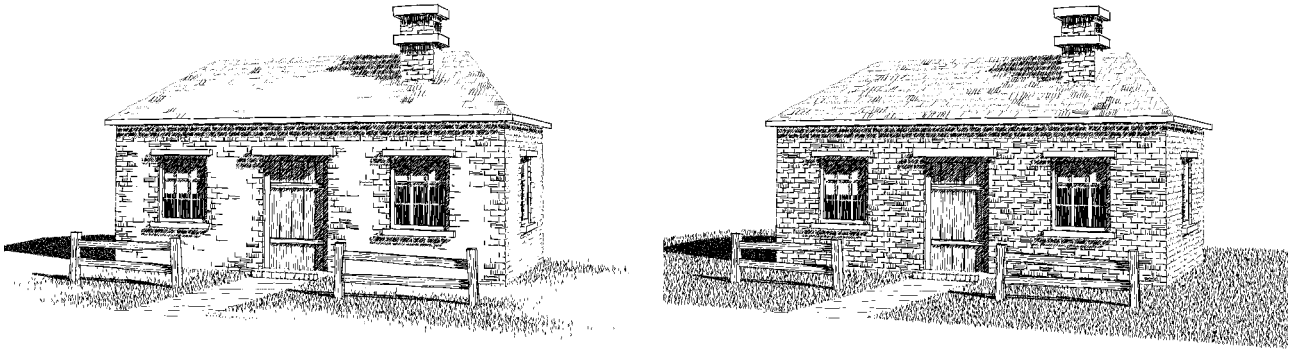


Figure 6: *Indicating texture. The left house is drawn using "indication"; the right house is not.*

information present for driving the ordinary graphics pipeline is in many respects also sufficient for achieving important non-photorealistic effects.

- Introducing the concept of a "prioritized stroke texture," a general framework for creating textures from strokes, and providing a methodology for building procedural versions of these textures.
- Allowing a form of resolution-dependent rendering, in which the choice of strokes used in an illustration is appropriately tied to the resolution of the target medium.

However, the work described in this paper is just one early step in the exploration of automated non-photorealistic rendering algorithms. There are many ways to extend this work, including:

- Improving the procedural stroke textures, and automating further our methods for creating them.
- Incorporating other illustration effects, such as exploded, cut-away, and peel-back views, for showing parts that are hidden.
- Adding more interactive controls to help in designing 3D illustrations. Also, experimenting with very high-level controls—for example, a control to add emphasis to parts of an illustration, which would work by automatically accentuating and suppressing detail over different parts of the image.
- Rendering other natural forms that appear in architectural drawings (and for which established conventions also exist), such as trees, grass, water, human figures, etc.
- Rendering other types of databases besides architectural models, such as databases of mechanical parts. Also, applying traditional illustration techniques and principles to databases that are not inherently visual in nature, such as flow simulations or higher-dimensional datasets. (This variety of rendering could be thought of as a form of scientific visualization.)
- Creating animations. Because our system uses randomness profusely, issues in frame-to-frame coherence arise. For instance, large features that are random, such as the selection of bricks that are shaded, should not vary from frame to frame. However, more subtle features, such as the waviness of strokes used to give the hand-drawn appearance, should be allowed to waver [15].
- Exploring other forms of illustration besides pen-and-ink, including traditional forms like watercolor and air brushing, as well as new methods of conveying information visually that may not necessarily mimic traditional forms.

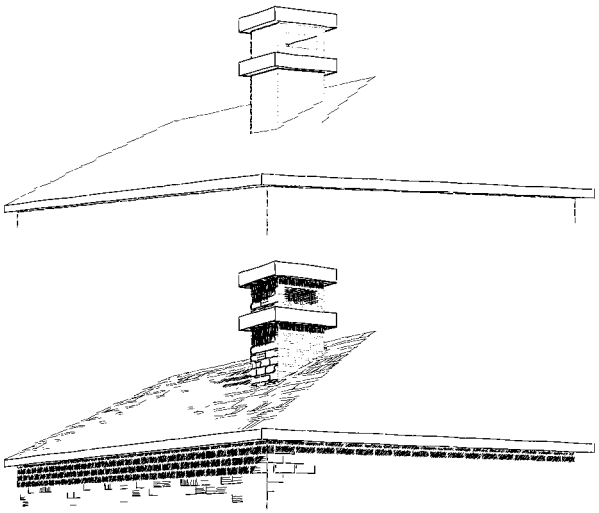


Figure 7: *Indicating texture through outline. Notice how different textures are delineated with different styles of boundary outlines. The upper and lower illustrations are the same, except that all but the boundary outline textures have been removed in the upper illustration to present the different styles more clearly.*

Acknowledgments

We wish to thank Tony DeRose for many helpful ideas along the way. Thanks also to Dan Ambrosi who advised us on illustration techniques and who also suggested the "Robie House" as a model. We are also grateful to Taweewan Winkenbach who spent many hours building the model for the "Robie House".

This work was supported by NSF Presidential and National Young Investigator awards (CCR-8957323 and CCR-9357790), by the University of Washington Graduate Research and Royalty Research Funds (75-1721 and 65-9731), and by industrial gifts from Adobe, Aldus, and Xerox.

References

[1] Adobe Systems Incorporated, Mountain View. *Adobe Dimensions*, 1992.

[2] Arthur Appel, F. James Rohlf, and Arthur J. Stein. The haloed line effect for hidden line elimination. Proceedings of SIGGRAPH '79 (Chicago, Illinois, August 8-10, 1979. In *Computer Graphics* 13, 2 (August 1979), 151–157.

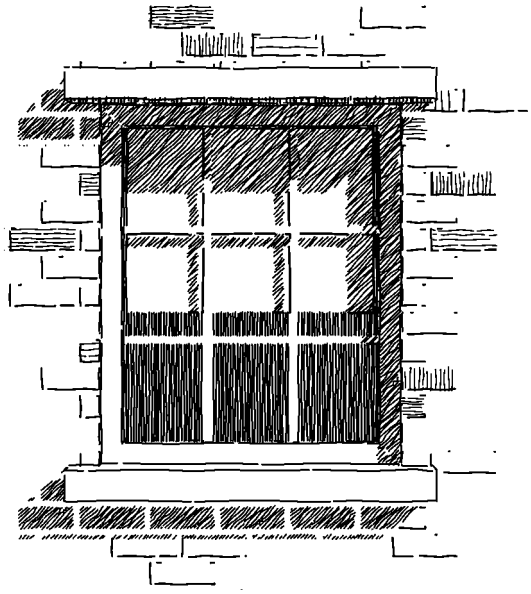


Figure 8: *Outline minimization. Notice how the boundary edges on the vertical and horizontal dividers between the panes appear only where contrast with the adjacent surface is low. (These boundary outlines are also omitted when they face the light source.)*

[3] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics* 26, 2 (July 1992), 35–42.

[4] Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. Proceedings of SIGGRAPH '89 (Boston, Massachusetts, July 31 - August 4, 1989). In *Computer Graphics* 23, 3 (July 1989), 99–106.

[5] Frank Ching. *Architectural Graphics*. Van Nostrand Reinhold Company, New York, 1975.

[6] Debra Dooley and Michael F. Cohen. Automatic illustration of 3D geometric models: Lines. *Computer Graphics*, 24(2):77–82, March 1990.

[7] Debra Dooley and Michael F. Cohen. Automatic illustration of 3D geometric models: Surfaces. In *Proceedings of Visualization '90*, pages 307–314, October 1990.

[8] Adam Finkelstein and David H. Salesin. Multiresolution curves. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24-29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

[9] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. Proceedings of SIGGRAPH '80 (Seattle, Washington, July 14-18, 1980). In *Computer Graphics* 14, 3 (July 1980), 124–133.

[10] Leonidas Guibas, Lyle Ramshaw, and Jorge Stolfi. A kinetic framework for computational geometry. In *Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science*, pages 100–111, 1983.

[11] Arthur Leighton Guptill. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York, 1976.

[12] Tomihisa Kamada and Saturo Kawai. An enhanced treatment of hidden lines. *ACM Transaction on Graphics*, 6(4):308–323, October 1987.

[13] Stephen Klitment. *Architectural Sketching and Rendering: Techniques for Designers and Artists*. Whitney Library of Design, New York, 1984.

[14] Paul Laseau. *Architectural Drawing: Options for Design*. Design Press, New York, 1991.

[15] John Lasseter. Personal communication, January 1994. Pixar, Richmond, California.

[16] John Lewis. Personal communication, November 1993. Boeing Computer Services, Seattle, Washington.

[17] Frank Lohan. *Pen and Ink Techniques*. Contemporary Books, Inc., Chicago, 1978.

[18] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland 20850, 1988.

[19] Kazunori Miyata. A method of generating stone wall patterns. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 387–394.

[20] Tom Porter and Sue Goodman. *Manual of Graphic Techniques 4*. Charles Scribner's Sons, New York, 1985.

[21] The Premisys Corporation, Chicago. *Squiggle*, 1993.

[22] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3D shapes. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 197–206.

[23] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24-29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

[24] Dorée Duncan Seligmann and Steven Feiner. Automated generation of intent-based 3D illustration. Proceedings of SIGGRAPH '91 (Las Vegas, Nevada, July 28 - August 2, 1991). In *Computer Graphics* 25, 4 (July 1991), 123–132.

[25] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. Proceedings of SIGGRAPH '87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics* 21, 4 (July 1987), 153–162.

[26] Chris I. Yessios. Computer drafting of stones, wood, plant, and ground materials. Proceedings of SIGGRAPH '79 (Chicago, Illinois, August 8-10, 1979. In *Computer Graphics* 13, 2 (August 1979), 190–198.

A Implementation Details

A.1 Overview

Three main global data structures are used by our system:

- *The model* M. The model is stored as a collection of polygons in three-space. For convenience, concave polygons and polygons with holes are decomposed into convex polygons.
- BspTree. The 2D BSP tree [25] is a representation of the visible polygons projected to Normalized Device Coordinates (NDC) space. It is used for fast clipping of strokes.
- PlanarMap. The planar map [18] is a partition of the NDC plane into *vertices*, *edges*, and *faces*, according to the NDC projections of the visible polygons. It is used to generate the outline strokes of the surfaces.

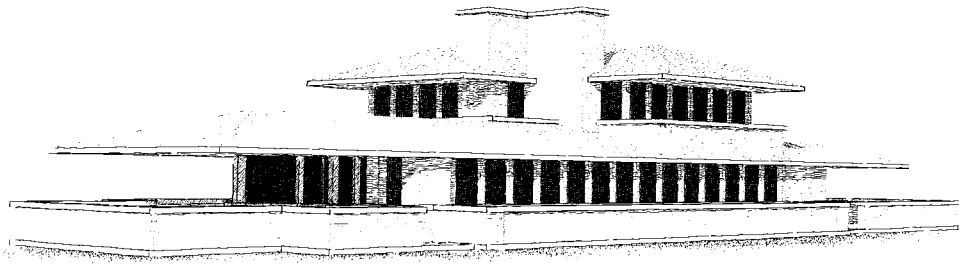


Figure 9: *Frank Lloyd Wright's "Robie House."*

The rendering process is structured as follows:

procedure *RenderScene*(M):
  $(BspTree, PlanarMap) \leftarrow VisibleSurfaces(M)$
  for each visible surface $S \in M$ **do**
    $Strokes \leftarrow Texture(S, Tone(S))$
    for each stroke $s \in Strokes$ **do**
      $Render(ClippedStroke(s, BspTree))$
    end for
  $Render(ConstructMinimalOutline(S, PlanarMap))$
end procedure

The following sections describe the individual stages of the rendering process in more detail.

A.2 Computing the visible surfaces

We use a 3D BSP tree to compute visibilities [9], and Chin and Feiner's shadow volumes [4] to compute the shadow polygons. The result is a set of convex polygons that can easily be ordered in depth with respect to the view point. To build the 2D BSP tree, the visible polygons are examined in front-to-back order. Each polygon is first projected to NDC space, and then inserted into the 2D BSP tree. The insertion into the 2D BSP tree is equivalent to the set union operation described by Thibault and Naylor [25], except that "in" leaf nodes carry an additional pointer back to the 3D polygon from which they originate. As such, the 2D BSP tree forms a partition of NDC space, with each cell in the partition corresponding either to a unique frontmost polygon in the 3D scene, or to the background.

The planar map data structure is computed with the help of the 2D BSP tree. We begin by inserting a single rectangular region, representing the entire drawing surface in NDC space, into the tree. As each node of the tree is traversed, the original region is partitioned into smaller and smaller faces in each branch of the tree. Faces reaching an "out" leaf node are tagged as background faces. Faces reaching an "in" leaf node receive a pointer to the corresponding 3D polygon in M. The BSP tree node also receives a pointer to the planar map face. Because of numerical inaccuracies, it is possible that some leaf nodes in the BSP tree never receive a matching face in the planar map. During clipping, a segment that falls in a leaf node having no planar map pointer is simply discarded. Because such nodes correspond to extremely thin regions, no visible artifacts result.

Geometrically, the planar map and the BSP tree are redundant: they encode the same 2D partition. However, the two data structures are amenable to different tasks. The BSP tree is efficient for clipping strokes through set operations, but does not readily allow searching among neighboring polygons. By contrast, the planar map encodes polygon adjacencies, but does not lend itself as well to clipping.

A.3 Rendering the textures

A.3.1 Individual strokes

A *stroke* $\mathcal{S}$ consists of three parts:

- a *path* $P(u) : [0,1] \rightarrow \mathbb{R}^2$, giving the overall "sweep" of the stroke, as a function of the parameter u.
- a *nib* $\mathcal{N}(p)$, defining the cross-sectional "footprint" of the stroke, as a function of the pressure p on the nib.
- a *character function* $C(u) = (C_w(u), C_p(u))$, describing the *waviness* of the curve $C_w(u)$ (how the curve departs from its path) and the pressure $C_p(u)$ on the nib.

The stroke $\mathcal{S}$ is defined as all pixels in the region

$$\mathcal{S} \quad = \quad (P(u) + C_w(u)) * \mathcal{N}(C_p(u))$$

where $*$ denotes the convolution of two parameterized point sets $A(u)$ and $B(u)$ of the Cartesian plane $\mathbb{R}^2$. This convolution is defined as [10]:

$$A(u) * B(u) \quad = \quad \bigcup_{u \in [0,1]} \{a + b \,|\, a \in A(u) \,\wedge\, b \in B(u)\}.$$

A stroke $\mathcal{S}$ is rendered by scan-converting the path (after waviness is added) and stamping a copy of the nib, scaled by the pressure value, in place of drawing each pixel. Note that more efficient scan-conversion methods undoubtedly exist. Indeed, the investigation of a good representation for individual strokes, including their overall sweep and character functions, is a sizable research topic in and of itself [8].

All strokes are drawn by a C++ object named *InkPen*. An *InkPen* is in turn composed of three objects: a *Nib*, a *WavinessFunction*, and a *PressureFunction*. Different pens can be created by assembling various combinations of these components. So far, we have only used circular nibs of variable radius, and a sine-wave waviness function with randomly perturbed amplitude and wavelength. Two kinds of pressure functions are used throughout the images in this paper: a simple "broken-line" function that lifts the pen off the paper with some randomness, and a random sine wave function that creates strokes of varying thickness. Although our implementation does not allow for all the generality of real pen and ink as described in Section 2.1, the limited set of functions we have implemented still allows for a fairly wide range of expressiveness.

An *InkPen* supports methods to: scale the nib size; query the amount of ink deposited between two points when using a particular nib size; and draw a stroke between two points to achieve a particular darkness, in which case the darkness of the stroke will be appropriately modulated by the *PressureFunction* of the pen.

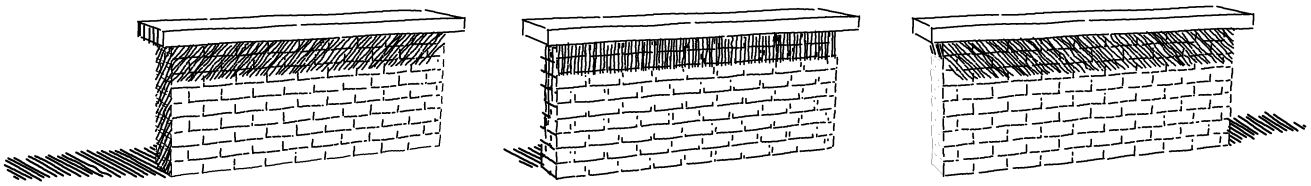


Figure 10: *Illuminated bricks. Notice how the thickened edges change to follow the shadow direction.*

A.3.2 Building tone from strokes

The first step in building tone is to compute a reference shade $v \in [0 = white, 1 = black]$; we use a simple Phong model in our implementation. A procedural stroke texture is then used to produce a set of strokes that together achieve the target grey value v. As a simple example, to achieve a tone v over an area A using a nib of width w with non-overlapping strokes requires drawing strokes of total length vA/w.

To compute tone accurately, it is also important to take into account the overlap between strokes that cross each other. Assuming uncorrelated overlap between the strokes of each non-overlapping set seems to work well. For example, for bidirectional hatching, suppose that the strokes in each hatching direction deposit a quantity of ink x on the paper. Then the total ink in the cross-hatched area is given by the sum of the ink from the strokes in each direction $2x$, minus the quantity of ink that is deposited in the same place twice, which, assuming uncorrelated strokes, can be approximated by x^2. Thus, in order to achieve a value v, we use enough strokes in each direction to achieve a value x that satisfies the quadratic equation $2x - x^2 = v$. The correct level for each set is therefore given by $x = 1 - \sqrt{1 - v}$. This expression is used, for example, by the "cross-hatching" texture to achieve the right tone when two different directions of hatching are crossed (Figure 2).

A.3.3 Stroke textures

To draw more complex textures, we use procedural prioritized stroke textures, as described in Section 4. In this section, we examine the "brick" texture in some detail. The description in this section can readily be extended to other types of stroke textures.

The "brick" texture builds tone out of three sets of strokes: the brick outlines; shading strokes within the bricks; and hatching strokes layered over the whole surface. Each set of strokes is associated with a different *InkPen*.

The rendering process for "brick" textures is summarized below:

procedure *RenderBrickTexture*(*TargetTone*, *Polygon3D*)
  Layout $\leftarrow$ *GenerateBricks*(*Polygon3D*, *BrickSizes*)
  for each brick $B \in$ *Layout* **do**
    DrawBrickOutline(B, *TargetTone*, *ViewPoint*, *Lights*)
    if the tone of B is too light **then**
      ShadeWithinBrick(B, *TargetTone*)
    end if
  end for
  if the overall tone is still too light **then**
    HatchOver(*Layout*, *TargetTone*)
  end if
end procedure

The brick outlines are generated from a simple layout that is computed on the fly in texture space and then projected into device space before drawing takes place.

The width of the nib used to draw the outline strokes is scaled according to the tone being rendered: darker tones use the default nib size; for lighter tones, the nib size is scaled down. The same nib size is used for all the bricks in order to achieve a consistent result over the entire surface. The darkness of each stroke is then adjusted so as to take the BRDF and shadow edge effects into account, as described in Section 4.3. The darkness of the strokes may be further reduced, for example, when the area of the bricks becomes smaller due to perspective forshortening.

Let T_o be the tone created by the outline strokes for a given brick B. We estimate T_o by taking the sum of the amount of ink deposited by each outline stroke and dividing it by the area covered by the brick on the paper, as discussed in Appendix A.3.2. If T_o is not dark enough to achieve the desired tone, then the interior of the brick is shaded. The darkness of each brick is limited by a constant T_s. Rather than shading every brick with the same tone $T = \textit{TargetTone} - T_o$, we instead shade each brick with probability $\min\{1, T/T_s\}$. If the brick is shaded, enough strokes are used to achieve a tone T_s, with some randomness added.

If the shaded bricks still do not achieve the target tone, that is, if $T_o + T_s < \textit{TargetTone}$, then additional hatching is used over the top of all the bricks. For these hatch lines, we use the method described in Appendix A.3.2 to take into account the overlap between strokes.

A.3.4 Clipping strokes

The strokes must be clipped to the visible regions they texture. The 2D BSP tree data structure is used for this purpose. The path of each stroke is "pushed" down the BSP tree until it reaches one or more leaf nodes. Only the portions of the path that reach nodes belonging to the region being textured are rendered. Other clipping conditions can also be used. For instance, hatching shadow areas is handled by generating strokes over the entire surface and clipping them to the regions in shadow.

A.3.5 Constructing the outline strokes

The outlines of visible surfaces are extracted by traversing the set of edges stored in the planar map. As described in Section 4.3, these outline edges are rendered only when the tones on either side of the edge are very similar, and when the edges themselves are not directly illuminated by the light source. Testing the tones of adjacent faces is easily accomplished by searching in the planar map.