Computer-Generated Pen-and-Ink Illustration

by

Georges A. Winkenbach

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by_____
Chairperson of Supervisory Committee

Program Authorized
to Offer Degree _____

Date _____

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Request for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature _____

Date _____

University of Washington

Abstract

Computer-Generated Pen-and-Ink Illustration

by Georges A. Winkenbach

Chairperson of Supervisory Committee: Professor David H. Salesin
Department of
Computer Science and Engineering

This dissertation describes the principles of pen-and-ink illustration, and shows how a great number of them can be implemented as part of an automated rendering system. Illustration techniques in general, and pen-and-ink rendering in particular, offer great potential for creating effective images from CAD models. And with the computer's ability to manipulate increasingly large models, communicating complex information in an effective and comprehensible manner is becoming an important problem. However, this potential remains relatively untapped in the field of computer graphics.

After discussing principles of traditional pen-and-ink rendering, this dissertation shows how the traditional graphics pipeline must be modified to support pen-and-ink rendering. Then, it introduces the new concept of prioritized stroke textures. Prioritized stroke textures form the central mechanism by which strokes are generated so as to both convey a certain texture, such as “bricks”, and achieve a target tone simultaneously. Prioritized stroke textures also have the advantages of being resolution dependent; that is, they take into account both the resolution of the target device, and the size of the image when generating the strokes.

A mathematical framework, and algorithms derived from it, for mapping stroke textures on parametric free-form surfaces are also introduced. Rendering strokes

on parametric surfaces is not a simple problem, because the orientation of the strokes must indicate the shape of the surface, in addition to accurately reproducing tone and conveying texture. The solution proposed in this dissertation generalizes the concept of prioritized stroke textures and allows the use of traditional, image-based, texture mapping techniques. Thus, it extends considerably the range of effects that can be achieved with stroke textures.

Finally, this dissertation describes two methods for building two-dimensional spacial subdivisions of the visible surfaces from the 3D geometry. These “planar maps” are needed during the rendering process for generating the outlines of the visible surfaces, while taking into account adjacency, tone, and texture information. The first method is relatively simple to implement, but is best adapted to polygonal models. The second method is more appropriate for models containing free-form surfaces.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my thesis advisor, David Salesin–as well as Tony DeRose, with whom I've had the privilege to work for several years–for teaching me everything I know about computer graphics and about doing research. They also created a research environment bestowed with an incredible dynamism, where excellence is a hallmark. I'm especially grateful to David Salesin for introducing me to the field of illustration, and for his continual and helpful feedback during my work on pen-and-ink rendering.

My thanks also go to my reading committee, David Salesin, Tony DeRose, and Chris Ozubko for their feedback on the first draft of my dissertation, and for helping me put the final version together. I also thank Jorge Stolfi, who, during a brief visit to our group, inspired the framework for hatching parametric surfaces in pen and ink.

This dissertation would not have been possible without the unweathering moral support I got from my wife Taweewan. She knew how to be a companion, a friend, and a confident, each when I needed it the most. And she always believed in me, even when it was going to take "just a bit longer than I thought." Finally, I express my deepest gratitude to my father, mother and siblings, whose support is unflinching and timeless, despite the great distances that separate us nowadays.

In memory of my brother Vincent

Chapter 1

Introduction

For the past twenty-five years, the field of computer graphics has been largely dominated by photorealistic rendering: the goal of rendering images indistinguishable from photographs. Recent results in global illumination have made considerable progress toward that goal. As a result, image synthesis can today create images of striking realism.

Photorealism can be extremely appealing, but it is not always the best medium for communicating information clearly. A photograph records everything that the camera sees, including the smallest detail and the faintest variation in shading. However, to communicate truly complex information clearly, some form of visual abstraction is required. This type of abstraction has been studied most comprehensively in the field of graphic design and traditional illustration.

The fields of graphic design and illustration have been around, under one form or another, for centuries. Many techniques and "tricks" have been developed to create effective illustrations. For instance, illustrators can remove extraneous details to create simpler and more effective illustrations. They also often simplify or stylize complex shapes to make them more readily comprehensible. In the same vein, they vary the amount of detail present in the illustration to create regions of emphasis; areas of high detail tend to attract the viewer's attention, while areas of low detail tend to fall out of focus. Other techniques include cut-aways and ghosting to show otherwise hidden parts and structures, and exploded views to illustrate assemblies. All these illustration techniques work toward a common goal, to simplify and summarize the subject so as to better capture its essence.

Focus + context
detail + abstract

The advantages of illustrations over photographs are recognized in many practical contexts. For instance, medical texts often use illustrations in addition to, or even in place of photographs, because they allow complex shapes and tiny structures to be clearly described. Likewise, most assembly and repair manuals use illustrations rather than photographs because of their clarity. Architects also often favor illustrations and sketches because they communicate the essence of their design better.

1.1 Goal of thesis

To explore the use of abstraction as a means for conveying information effectively, it makes sense to start with an area with well-established conventions. For this reason, in the course of this dissertation, we'll explore a software architecture and algorithms for implementing a pen-and-ink rendering system. Restricting the domain to "pen and ink" also has the advantage that no "exotic" display technology is required. Conventional, inexpensive laser printers are able to reproduce pen-and-ink renderings very well, as exemplified by many of the figures presented in this dissertation.

Pen and ink is a widely used medium in architectural and scientific illustration. Pen-and-ink drawings, by their very nature, possess some special qualities that are difficult to reproduce in other media. Their simplicity provides an appealing crispness and directness. Pen-and-ink illustrations also blend well with text, due to their linear quality and their use of the same ink on the same paper. Finally, pen-and-ink images are also easy and inexpensive to reproduce, and retain their integrity even when reduced.

In this dissertation, we'll focus our attention on the low-level details of rendering in pen and ink from a 3D model. Higher-level issues, such as composing an effective illustration, will not be addressed here. This research is related to the work of Strothotte et al. [39], in that it attempts to create images with a hand-drawn appearance in a specific style. However, we'll pay close attention to issues specific to pen-and-ink rendering, including outlining, texturing, and conveying tone accurately. We'll introduce a system architecture specifically for pen-and-ink rendering, but which should find uses for rendering in other illustration styles. We'll also introduce new algorithms for mapping pen-and-ink textures on polygonal and parametric surfaces.

1.2 Related work

Despite its advantages, the field of illustration has received relatively little attention in the computer graphics community. We survey the related work here.

One of the earliest results having to do with creating illustrations from 3D models is due to Appel et al. [2]. These authors presented an algorithm for improving a simple wireframe rendering with haloed-line effects. A haloed line has a small white gap where it crosses another line that is closer to the viewer, to enhance the illusion that it is passing behind, and to more clearly convey the illusion of depth. Their solution greatly improves upon wireframe rendering, by making the image easier to comprehend. It also has some advantages over hidden-line elimination, because the resulting images present more information.

Kamada and Kawai generalized the work of Appel et al. by showing how line attributes, such as dashes or dotted lines, could be use to improve the treatment of hidden-lines [20]. For instance, lines hidden behind one surface could be drawn with dashes, lines behind two surfaces with dots, and lines hidden deeper could be eliminated. The choice of attributes and the rules for using them is under user control.

Dooley and Cohen further improved the treatment of line drawing [8]. In their approach, the user can attach “importance tags” to the objects forming the model. At rendering time, these tags are used to direct the choice of line attributes used in rendering a specific object. For instance, an object of high importance hidden behind an object of low importance could be rendered with thick dashed lines. In addition to varying the style and thickness of lines, they also use end conditions, such as haloing, and tapering. For example, a line fading at a distance would be tapered to enhance the illusion of “fading away”. Dooley and Cohen's system uses illustration rules, either specified by the user or inferred from illustration principles, to generate the image from the tagged model.

The same authors took the idea of using the relative importance of objects to the treatment of shaded renderings [9]. In that work, a “high-importance” object might cast a shadow, while a “low-importance” object would not. Likewise, a “high-importance” object would be partially visible through a “low-importance” object. In many respects, Dooley and Cohen were among the first to consider illustration-related issues rigorously, and to introduce the idea of using high-level control, such as importance tags, to influence the rendering.

Saito and Takahashi also proposed a system that enhances a shaded image with features borrowed from the realm of illustration [34]. In their approach, they produce both an image and a "G-buffer" using a modified ray tracer. The G-buffer contains information such as z-depth, surface normal, and object ID for each pixel in the image. They then use image processing techniques that operate on both the image and the G-buffer to augment the shaded rendering with features such as outline edges, contour lines, and crosshatching.

Another system that mixes information from both object space and image space is Piranesi, proposed by Lansdown and Schofield [22]. The Piranesi system looks like a paint system to the user. Textures that imitate natural drawing media are applied automatically or semi-automatically to regions of the image. These regions are selected from a buffer resembling a G-buffer, using selection criteria based on material types, surface normal, and planarity. In part because it uses a semiautomatic approach, the Piranesi system is able to create highly artistic images in a wide variety of styles.

Salisbury et al. also used a semiautomatic approach to create illustrations in their interactive pen-and-ink illustration system [36]. These authors designed a 2D illustration system that provides the user with high-level tools to create pen-and-ink illustrations. Notably, the user can brush a selected area with predefined stroke textures that share many of the characteristics of the stroke textures presented in this dissertation. Most commonly, the area to paint and the tone to achieve in that area are specified by an underlying grey-scale image. By first rendering a grey-scale image from a 3D model, their system can be used to create pen-and-ink illustrations from 3D data.

Another classical form of illustration that has received some attention is line-art rendering. Leister presented a system to render images in a copperplate style [24]. Copper plate rendering provides a highly stylized type of imagery that resembles most engraving. Leister's approach uses a ray-tracer to render the surface curves resulting from the intersection of closely spaced parallel laminae with the objects composing the scene. The resulting images have a certain aesthetic appeal. However, Leister's method produces very stylized illustrations, is not able to adequately convey subtle variation in shading, and does not allow texturing.

Along a similar vein, Elber presented an algorithm to render NURBS surfaces with a coverage of isoparametric curves [10]. Shading is conveyed by adjusting the density of the isoparametric curves as a function of the illumination on the surface. One drawback of Elber's method is that it does not

reproduce smooth variations in shading well. In addition, Elber did not address more fundamental issues in pen-and-ink rendering, such as stroke qualities, outlining, and texturing.

The work of Strothotte et al. also dealt with line-art rendering [39]. These authors presented the design of a "sketch-renderer" that produces images that appear hand-drawn. Their system allows the user to adjust the style of the rendering, from very sketchy to more mature hand-drawn images. The different styles are achieved by varying the line styles and the shadow styles. These authors also addressed the issue of emphasizing a certain area of the rendering by drawing more details there. Finally, they also presented techniques for rendering plant outlines without foliage, and for conveying the illusion of movement using metasymbols derived from comic strips.

With respect to the rendering of architectural forms, Yessios described a prototype "computer drafting" system for common materials in architectural design, such as stone walls, wood, plants, and ground materials [45]. Like the work presented in this dissertation, Yessio's prototype attempts to provide warmer, hand-drawn appearance as opposed to a mechanical one. Miyata also gave a nice algorithm for automatically generating stone-wall patterns [27]. These patterns would make a good starting point for some of the pen-and-ink techniques described in this dissertation.

In the commercial realm, the Premisys Corporation markets a product called "Squiggle" that adds waviness and irregularities to 2D CAD output as a post-process, lending a hand-drawn appearance to the drawings [33]. Also worth noting is the Adobe Dimensions program that allows Postscript stroke textures to be mapped onto surfaces in three dimensions [1]. Adobe Dimensions is able to save the rendered image as a 2D vector-graphics file, thus allowing the resulting objects and texture outlines to be edited in a conventional draw program.

All the research presented so far deals mainly with the rendering problem. Another aspect of creating illustrations, which Seligman and Feiner addressed in their work [37], is that of composing the model to achieve a specific communicative goal. For instance, a communicative goal might be to show the location of a specific object with respect to its context. Their system uses a knowledge base of the physical object being described, together with design and style rules to generate illustrations. The images that it creates includes features such as highlighting, cut-aways, and insets. Seligman and Feiner's concern is more with the high-level goal of composing a model that best communicates a particular intent, and they used a traditional shaded renderer to create the final image.

1.3 Overview of dissertation

We will begin, in Chapter 2, with a review of the pen-and-ink illustration principles that inspired the algorithms described in this dissertation. These principles were gathered from a variety of sources on pen-and-ink rendering. They form only a small subset of all the techniques that an illustrator might use. However, as we will see, they constitute a solid ground upon which to design a pen-and-ink rendering system.

Chapter 3 gives the overall architecture of the pen-and-ink rendering system, which should be useful when reading the subsequent chapters.

Chapter 4 describes the concept and implementation of strokes, the most fundamental building block for pen-and-ink illustrations.

Chapter 5 introduces the concept of a *prioritized stroke texture*. Prioritized stroke textures form one of the most important components of the system. They constitute the fundamental mechanism by which strokes are used to convey both tone and texture simultaneously. Chapter 5 is restricted to textures mapped on polygonal models. That restriction still allows many concepts to be introduced, without the difficulty inherent in dealing with curved surfaces.

Chapter 6 generalizes the stroke textures introduced in Chapter 5 to parametric free-form surfaces. It gives a method to draw strokes that is able to convey the shapes of curved surfaces and smoothly varying tones simultaneously. It also introduces the use of traditional, image-based texture-mapping techniques to extend the range of effects that stroke textures can handle.

Chapter 7 proposes two algorithms for building planar maps. The construction of a planar map is an essential step of the rendering process, as we will discover in Chapter 3. The planar map is used during the rendering phase to outline the visible surfaces. The first planar map algorithm is restricted to polygonal models, while the second can handle models with curved surfaces.

Finally, in Chapter 8, we'll summarize the work accomplished, and explore some areas for future research.

Chapter 2

Principles of Pen-and-Ink Illustration

In this chapter, we survey some of the fundamental principles of pen-and-ink illustration. These principles are distilled from a number of sources, including Guptill's classic text *Rendering in Pen and Ink* [18], Lohan's *Pen&Ink Techniques* [25], Wood's *Scientific Illustration* [44], and several other sources [7, 21, 23, 32, 38].

Pen-and-ink illustration uses only black ink strokes over a (usually) white surface. It is a limiting medium, since both shading and textures must be suggested by a combination of individual strokes. Furthermore, at least when drawn manually, it is very difficult and time consuming to cover large areas with strokes. However, drawing in pen and ink has some attractive characteristics. First, it is ideal for outlining: each individual stroke can be made expressive by giving it some irregularities in its path and pressure. Second, it can provide an economy of expression: often just a few strokes can clearly indicate the difference between textures like smooth glass and rough stone.

The field of pen-and-ink illustration is vast, and it is beyond the scope of this dissertation to provide a comprehensive treatment. However, the principles that we will describe next are sufficient to motivate the design choices of a pen-and-ink rendering system. The treatment of these principles is organized into three parts: Strokes, Tones and textures, and Outlines.

2.1 Strokes

In traditional pen-and-ink rendering, a stroke is produced by placing the nib of a pen in contact with the paper and allowing it to trace out a path. Character is given to the stroke by varying the pressure applied to the pen, hence the thickness of the stroke, by wiggling the pen along the path, and by changing the orientation of the pen. The shape of the nib also plays a role in the quality of the stroke; asymmetric nibs, such as those used in calligraphy, result in the thickness varying in relation to the direction of the stroke.

Stroke-drawing is guided by a number of principles:

Thickness

1. Too thin a stroke can give a washed-out appearance; too coarse can detract from the delicate details.

褪色 粗糙

Pen position

2. It is frequently necessary to vary the pen position, with the nib sometimes turning as the stroke is drawn.

笔尖

Natural

3. Strokes must look natural, not mechanical. Even-weight lines drawing appear lifeless; instead, the thickness of the stroke should vary along its path.
4. Wavy lines are a good way to indicate that a drawing is schematic and not yet completely resolved.
5. Smooth strokes that vary slowly are a good way to indicate smooth material, such as glass. Rough strokes that vary more rapidly indicate rough materials such as stone.

In addition to the character of the strokes, a pen-and-ink rendering system must also manage the collection of strokes as a whole, adjusting their spacing and thickness so as to create the desired tones. In such a system, having to consider all the principles introduced above can be a very complex task. To make the problem more tractable, we'll make use of a reduced number of stroke parameters. Most notably, we'll use nibs with a circular cross-section only, so that the orientation of the pen and the direction of the paths have no bearing on the character of the strokes.

2.2 Tones and textures

The term “tone” is used to refer to the apparent amount of light reflected toward the viewer from a point on a surface. In pen-and-ink illustration, it is impossible to portray the tone of each surface accurately; instead, collections of strokes are used to give an overall impression of the desired tones.

The tone achieved by a combination of strokes is a function of the ratio of black ink over the area of a region of the illustration.

In addition to building tones, strokes are also used to evoke textures. The character of the strokes themselves play a role in conveying different surface qualities, such as smoothness or roughness. Strokes can also be arranged in different ways to indicate complex textures such as bricks or stone walls. The dual role that strokes play in conveying both tones and textures is part of the economy of pen-and-ink illustration.

Here are some principles of drawing tones and textures in pen-and-ink:

1. Tones should be created from lines of roughly equal weight and spacing.
2. It is not essential to depict each individual tone accurately. However, it is important to present the correct arrangement of tones among adjacent regions.
3. To disambiguate objects, it is sometimes useful to "force a tone" by enhancing the contrast or inventing shadows.
4. To lend economy to an illustration, it is important to use some form of "indication" for suggesting a texture without drawing every last stroke. The method of indication should be varied across the illustration to avoid monotony.
5. The character and shape of the strokes play a role in conveying the tone, the texture, as well as the geometry of the surface they are covering. For example, horizontal surfaces should be hatched with predominantly horizontal strokes.
6. Absence of detail altogether indicates glare.
7. A sketchy kind of line is good for "old" materials, while a crisp straight line is good for "glass".

2.3 Outlines

Real scenes contain no outlines; instead, object boundaries are defined by variations in illumination and texture. However, in pen-and-ink illustration, as well as many other drawing media, tones and textures do not always clearly delimit objects. Outlining becomes a very natural way of delineating distinct surfaces.

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 strokes can be a powerful indicator of texture.

A number of principles for drawing expressive outlines follow:

1. The quality of the outline stroke is important for conveying texture. For instance, a crisp line is good for hard objects, while a wavy line with more variation is good for soft objects.
2. Outlines must be introduced where tones are omitted to convey shapes. Yet, outlines should not be used indiscriminately; doing so would result in a childish-looking drawing.
3. The character of the outline should vary with the tone; the outline stroke may even disappear completely in regions of glare.
4. Using indication for outlines is just as important as for drawing tone.

Chapter 3

Architecture of a Pen-and-Ink Renderer

In this chapter, we concern ourselves with the overall architecture of the pen-and-ink renderer. More details about specific components of the system will appear in the following chapters. However, to understand some of the design decisions, as well as how these components fit together, it is useful to have a picture of the system as a whole in mind.

3.1 Pen-and-ink versus photorealistic rendering

To implement a pen-and-ink renderer, a reasonable starting point is to first consider the traditional graphics pipeline, used in most photorealistic renderers, to see which parts, if any, need to be modified. To this end, we begin by examining the differences between photorealistic rendering and pen-and-ink rendering.

3.1.1 Dual nature of strokes

In the traditional graphics pipeline, rendering textures and conveying tones are completely independent tasks. A texture is typically defined as an image assigned to a surface, which affects the shading parameters. The desired tone is produced by dimming or brightening the rendered shades, while leaving the texture unchanged. In contrast, in a pen-and-ink renderer, the same strokes that are used to convey the texture are also used to create the tone. Thus, tone and texture are more tightly linked in a system used to produce this type of imagery.

3.1.2 Stroke density

In the traditional graphics pipeline, the information used for rendering is entirely three-dimensional, with the final projection to image space largely a matter of sampling the rendered shades at each pixel. In pen-and-ink illustration, the image-space area of a particular projection must be taken into account to compute the proper stroke density, in order to accommodate the dual nature of strokes described above.

3.1.3 Image-space adjacencies

In the traditional graphics pipeline, once occlusions are resolved, each surface is shaded independently. The image-space adjacencies between the projected surfaces do not matter. However, a pen-and-ink renderer must take the image-space adjacencies of the projected geometry into account, since outlining depends on issues such as the type of junctions between 2D boundaries (whether two adjacent regions in 2D are adjoining or passing one behind another in 3D), and the contrast between tones of adjacent 2D regions.

3.2 The pen-and-ink graphics pipeline

Many aspects of the standard graphics pipeline are well suited for a pen-and-ink renderer:

- *The model*. Any standard 3D geometry representation will do. This dissertation presents two methods; the first method handles polygonal models only, while the second method also handles parametric free-form surfaces. (In some cases, it will be necessary to distinguish between the two approaches. We'll refer to the first method as the *polygonal renderer*, and to the second method as the *curved-surface renderer*.)
- *The assignment of textures*. Textures are assigned to the 3D surfaces in the usual way. However, textures are no longer described by images, but by the *stroke textures* introduced in Chapter 5.
- *The lighting model*. Any standard illumination model can be used to compute a "reference solution" that is used as a target for tone production with strokes. All the pen-and-ink images presented in this dissertation were rendered using the Phong model, which appears quite adequate for this type of imagery.

- *The visible surface algorithm*. Any object-space or list-priority visible surface algorithm will do. The prototype polygonal renderer uses BSP trees, while the curved-surface renderer uses a method akin to Weiler's algorithm [42].
- *Shadow algorithm*. The shadow algorithm must also use an object-space or a list-priority method. The polygonal renderer uses Chin and Feiner's BSP tree shadow volumes [6], while the curved-surface renderer uses a list-priority method inspired by the shadow map algorithm described by Williams [43].

Next, we consider those aspects of the pen-and-ink renderer that are new or different from the standard pipeline. This description should be considered in conjunction with the pen-and-ink rendering pipeline shown in Figure 3-2.

3.2.1 Image-space planar subdivision

The need to consider 2D adjacency information during rendering suggests that we use some form of planar subdivision of the visible surfaces. We'll use a half-edge data structure [28] to maintain this planar map. The construction of the planar map from the 3D geometry is described in Chapter 7.

3.2.2 Stroke textures

Both tone and texture must be conveyed with some form of hatching. The size and target resolution of the output affect the hatching. Additionally, as we will see in the next chapter, parameters such as the direction of viewing and the position of the light source(s) also affect certain textures. All these parameters must be taken into account at rendering time. The procedural *stroke textures* described in Chapters 5 and 6 fulfill these requirements.

3.2.3 Stroke clipping

The strokes generated by the procedural stroke textures must be clipped to the visible portions of the surface they are texturing. Furthermore, to simulate a hand-drawn appearance, the clipping cannot be pixel-based; that is, it should not just remove those stroke pixels that fall outside of the clipping region. Instead, clipping must be stroke-based, allowing the stroke to sometimes stray slightly outside of the clipping region.

To achieve this effect, the strokes are represented with a path (a polyline), and a *character function* stored as a *waviness function* and a *pressure function*. The path of each stroke is clipped before adding in the character function. The image-space planar map provides a convenient data structure to perform the clipping.

More details about the stroke representation are found in Chapter 4. The use of the planar map data structure for clipping is described in Chapter 7.

3.2.4 Outlining

Outlining plays an important role in pen-and-ink illustration.The outline strokes must be drawn in a way that takes into account both the textures stored in the surrounding regions, as well as the adjacency information stored in the planar map. The procedural stroke textures are ultimately responsible for generating the outline strokes. They rely on the planar map not only to extract the boundary paths, but also to obtain information about the tone difference and the type of boundary (whether the two surfaces are abutting or one is passing in front of the other) of the adjacent surfaces.

3.2.5 Shadows

Rendering shadows is in many ways similar to rendering tone and texture. In particular, shadow strokes must be clipped to the regions in shadow. We'll investigate two methods for clipping shadow strokes in Chapter 7. The first method, better suited for polygonal models, uses an object-space approach to compute shadow polygons before constructing the planar map. The resulting spatial subdivision distinguishes between regions "in" and "not-in" shadow, and can conveniently be used to clip shadow strokes. The second method, more appropriate for models containing free-form geometry, builds an auxiliary planar map, called a *shadow planar map*, that stores the regions illuminated by the shadow-casting light. With the shadow planar map, clipping shadow strokes requires two passes: once against the image-space planar map, and once against the shadow planar map.

3.3 The rendering process

The pen-and-ink renderer uses two main data structures:

- The model M, stored as a collection of polygons and/or parametric surfaces.

```
procedure RenderScene (M)
   Pmap := VisibleSurfaces (M)
   for each visible surface S in M do
      Strokes := Texture (S, Tone(S))
      for each stroke s in Strokes do
         Render (ClippedStroke (s, Pmap))
      end for
      Render (ConstructOutline (S, Pmap))
   end for
end procedure
```

Figure 3-1. Pseudocode for the rendering process.

- The planar map *Pmap*, which partitions the image-space into vertices, edges and faces, according to the 2D projections of the visible surfaces.

The rendering process starts by constructing the planar map *Pmap* from the model *M*. Occlusions are also resolved during this step, so that only visible surfaces are represented in the planar map. Next, each visible surface *S* in *M* is considered in turn. The procedural stroke texture attached to *S* is invoked to generate all the strokes needed to convey the tone, shadows, and texture on that surface. The tone that must be conveyed on *S* is evaluated using the Phong illumination model, either as a pre-computation step, yielding a reference gray-scale image, or "lazily" as the strokes are generated. The texture generates the strokes on the 3D surface without regard for occlusions; therefore, each stroke must be clipped to the visible portions of the surface before being rendered. To this end, the path of each stroke is projected to image-space and clipped using the planar map. Strokes are then rendered along the remaining visible portions of the path. To complete the rendering of surface *S*, one or more paths forming its boundary outline are extracted from the planar map. The adjacency information available in the planar map is used to tag these paths, so that they can be rendered according to the principles of outlining. Finally, the procedural stroke texture is once again invoked to generate the outline strokes along these paths.

The rendering process is summarized in Figure 3-1.

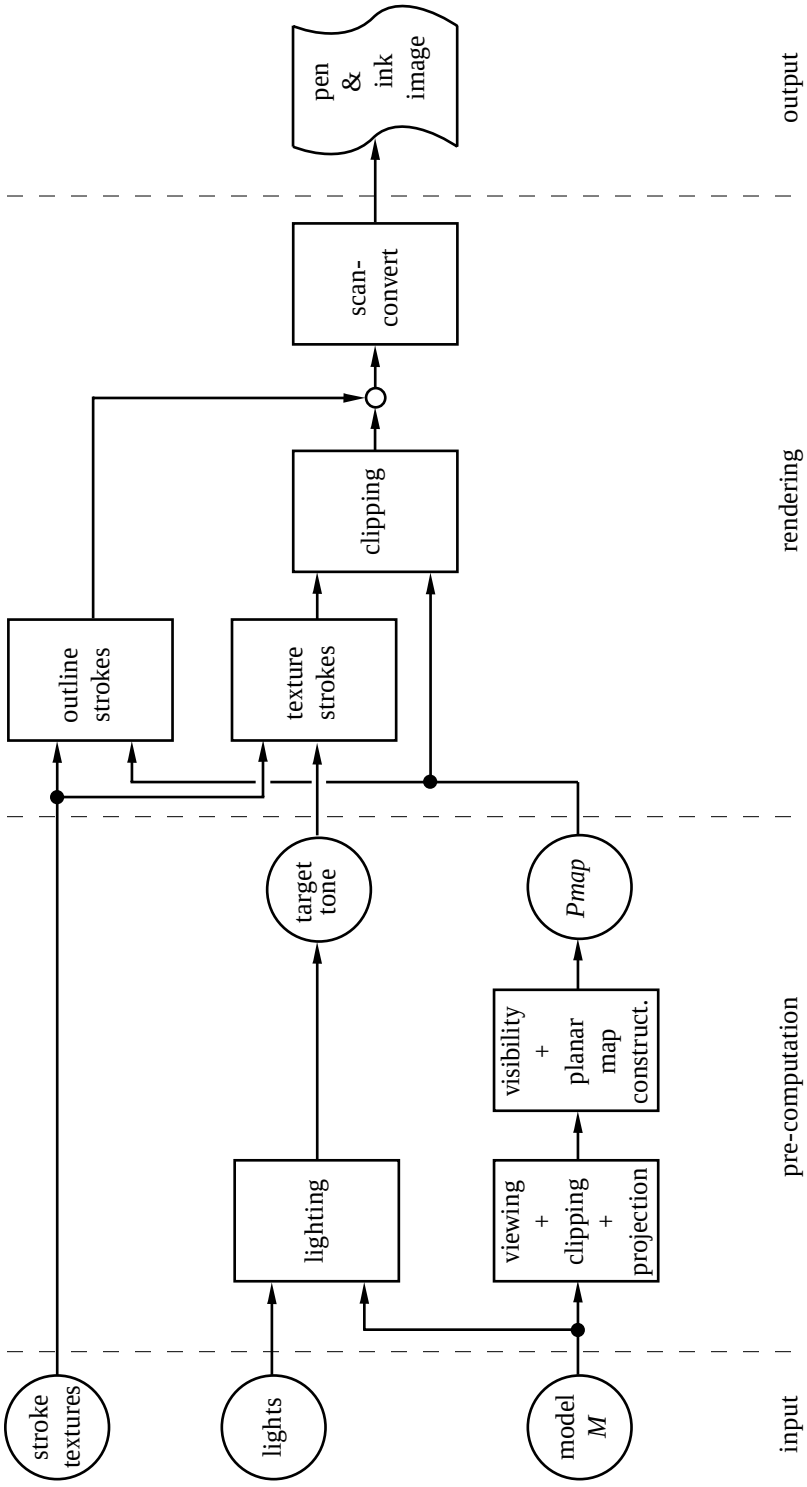


Figure 3-2. Pen-and-ink rendering architecture.

Chapter 4

Strokes

All drawing operations are ultimately carried out by drawing a stroke along a specified path. Initially, the illustration starts with a completely white surface. Darker tones are achieved by covering, or hatching, a specific area with strokes. However, strokes are used for more than just building tone. The character of the strokes, achieved by varying their waviness and thickness, plays an important role in the expressiveness of a pen-and-ink rendering.

Figure 4-1 shows various stroke qualities. Notice the two types of strokes used to render the wood boards on the door: thin strokes with an even thickness and some waviness depict the wood grain, while thicker strokes with a varying thickness and relatively little waviness delineate the gaps between the planks.

4.1 Formal definition

A stroke S is a three-tuple $\left(P(t), N(p), C(t)\right)$ where:

- $P(t):[0,1]\rightarrow\Re^2$ is a *path* giving the overall "sweep" of the stroke, as a function of the scalar distance t along the path.
- $N(p)$ is a *nib* defining the cross-sectional "footprint" of the stroke, as a function of the pressure p on the nib.
- $C(t)=(C_w(t), C_p(t))$ is a character function describing the waviness of the curve $C_w(t)$ (how the curve departs from its path), and the pressure $C_p(t)$ on the nib.

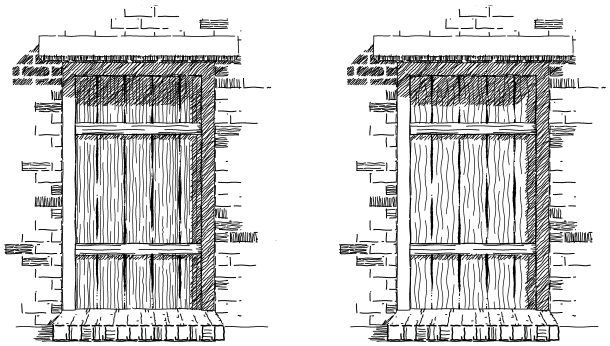


Figure 4-1. Drawing expressive strokes. The waviness of the "wood grain" strokes on the right has been exaggerated.

The stroke *S* is defined as all the pixels in the region

$$S = (P(t) + C_w(t)) * N(C_p(t))$$

where $*$ denotes the convolution of the two parameterized point sets $A(t) = P(t) + C_w(t)$ and $B(t) = N(C_p(t))$ of the Cartesian plane $\Re^2$. This convolution is defined as [16]:

$$A(t) * B(t) = \bigcup_t \left\{ a + b \mid a \in A(t) \text{ and } b \in B(t) \right\}$$

A stroke *S* is rendered by scan-converting the path, after adding the waviness, and stamping a copy of the nib scaled by the pressure value, in place of drawing each pixel.

To control the tone achieved by drawing a series of strokes, we must be able to determine how much ink each stroke is depositing along its path. To this end, we introduce the notion of *thickness* θ of a stroke *S*, which is defined as the width of a constant-thickness curve drawn along the path $P(t)$ that deposits the same amount of ink as the stroke *S*. Given θ, we can evaluate the amount of ink *I* that *S* deposits, assuming that its path does not self-intersect, with the expression

$$I = \theta \cdot \text{length}\left(P(t)\right)$$

4.2 Implementation

In the prototype pen-and-ink rendering system, 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 the user by assembling various combinations of these three components. Each component also has parameters that the user can adjust, such as the minimum size of the nib, and the amplitude and wavelength of the waviness and pressure functions.

Only circular nibs with a variable radius were used to generate all the images presented in this dissertation. This restriction greatly simplifies the building of tones with strokes, since the orientations of the strokes have no influence on their thickness. A unique waviness function was also used. It is a sine wave with a randomly perturbed amplitude and wavelength. Finally, two different pressure functions were used: a simple "broken line" function that lifts the pen off the paper with some randomness, and a random "sine wave" pressure function that adds varying thickness to the simple "broken line" pattern.

Each *InkPen* object supports a way of querying the maximum thickness $\bar{\theta}$ of the strokes that it draws, with the function *QueryMaxThickness*. The maximum thickness of a stroke is dictated by the components and parameter settings that the user has chosen for the given *InkPen*, as well as the resolution of the target device. It is evaluated dynamically, at run-time.

To draw a stroke, the function *DrawStroke* is invoked with the path $P(t)$ and an additional parameter $\rho(t) \in [0,1]$. The parameter $\rho(t)$ instructs the *InkPen* to adjust the stroke so that its thickness varies as $\theta(t) = \rho(t)\,\bar{\theta}$. The pressure function is ultimately responsible for achieving the target thickness $\theta(t)$. It does so in two ways:

- by adjusting the pressure *p* on the nib, and
- by using a "broken line" pattern, once the minimum nib size allowed either by the user or by the resolution of the target device is reached.

The stroke textures that we'll introduce in the next chapter make use of the two functions *QueryMaxThickness* and *DrawStroke* in tandem. By first determining the maximum thickness of strokes, they can adjust the spacing between hatching strokes so as to be able to achieve the darkest tone present anywhere on a given surface. While drawing strokes, they make use of the parameter $\rho(t)$

to adjust the thickness of the strokes, for instance to achieve lighter tones in specific areas of the surface.

To understand more clearly how the pressure on the nib is used to adjust the thickness of the strokes, we consider the "sine wave" pressure function in more detail. With this function, the pressure takes the form

$$p(t) = \tilde{p} + \tilde{A}\sin(2\pi t / \tilde{\omega})$$

where $\tilde{p}$ is the mean pressure, $\tilde{A}$ is the mean amplitude, and $\tilde{\omega}$ is the mean wavelength. To avoid mechanical-looking strokes, $\tilde{A}$ and $\tilde{\omega}$ are perturbed by some random amounts at the start of each wave. To create a speckled appearance when the pressure on the nib is very low, characteristic of an unsteady hand, a small amount of noise is superimposed. The resulting effect is most noticeable on the wood-grain strokes in Figure 6-7, on page 47.

Given the pressure $p(t)$, the radius $r(t)$ of the nib used to draw the stroke takes the form

$$r(t) = \begin{cases} R(1 + p(t)) & \text{when } p(t) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where R is the nib radius, either chosen by the user, or imposed by the finite resolution of the target device, whichever is the greatest. The thickness θ of the stroke is computed by integrating $r(t)$ over an entire mean wavelength and dividing by the length of the wave:

$$\theta = \frac{1}{\tilde{\omega}} \int_{t=0}^{\tilde{\omega}} r(t)dt \qquad (4.1)$$

Achieving the target stroke thickness $\theta(t) = \rho(t)\,\bar{\theta}$ is accomplished by adjusting the mean pressure and the wave amplitude as functions $\tilde{p}(\rho)$ and $\tilde{A}(\rho)$ of $\rho(t)$. The stroke reaches its maximum thickness $\bar{\theta}$ when $\tilde{p}(1) = \bar{p}$ and $\tilde{A}(1) = \bar{A}$, where $\bar{p}$ and $\bar{A}$ are chosen by the user. Thinner strokes are obtained by decreasing the mean pressure $\tilde{p}$ and dampening the wave amplitude $\tilde{A}$ in concert. Because the thickness of the nib is given by the integral in Equation 4.1, closed-form expressions for $\tilde{p}(\rho)$ and $\tilde{A}(\rho)$ are not readily available. Instead, each InkPen builds a table that relates ρ (or θ) to $\tilde{p}$ and $\tilde{A}$ using Equation 4.1. To set $\tilde{p}$ and $\tilde{A}$ given $\rho(t)$, the InkPen uses an "inverse" lookup in that table.

Chapter 5

Stroke Textures

Stroke textures lie at the heart of the renderer. They are ultimately responsible for generating strokes so as to convey tone and texture simultaneously. In this chapter, we'll consider stroke textures in the context of polygonal models only. In addition, we'll also assume flat-shaded surfaces. (The generalization to smoothly-shaded curved surfaces is found in the next chapter.) These two restrictions simplify the design of stroke textures considerably. In particular, all stroke paths become line segments.

We'll begin by studying one of the simplest stroke texture, crosshatching. Crosshatching will allow us to understand how tone is built with strokes, without having to worry too much about how the strokes are generated. Then, we'll be ready to examine slightly more complicated textures, such as bricks and shingles.

5.1 Crosshatching

Hatching uses a series of closely spaced parallel strokes to build tone. Crosshatching overlays strokes with a few distinct hatching orientations to create darker tones. Figure 5-1 (a) shows crosshatching for various tone values.

Creating a tone $T \in [0 = white, 1 = black]$ over a given area A requires that a fraction T of A is covered with ink. In the case of hatching with strokes of thickness θ, that ink coverage is achieved if the spacing between the strokes is $d = \theta / T$.

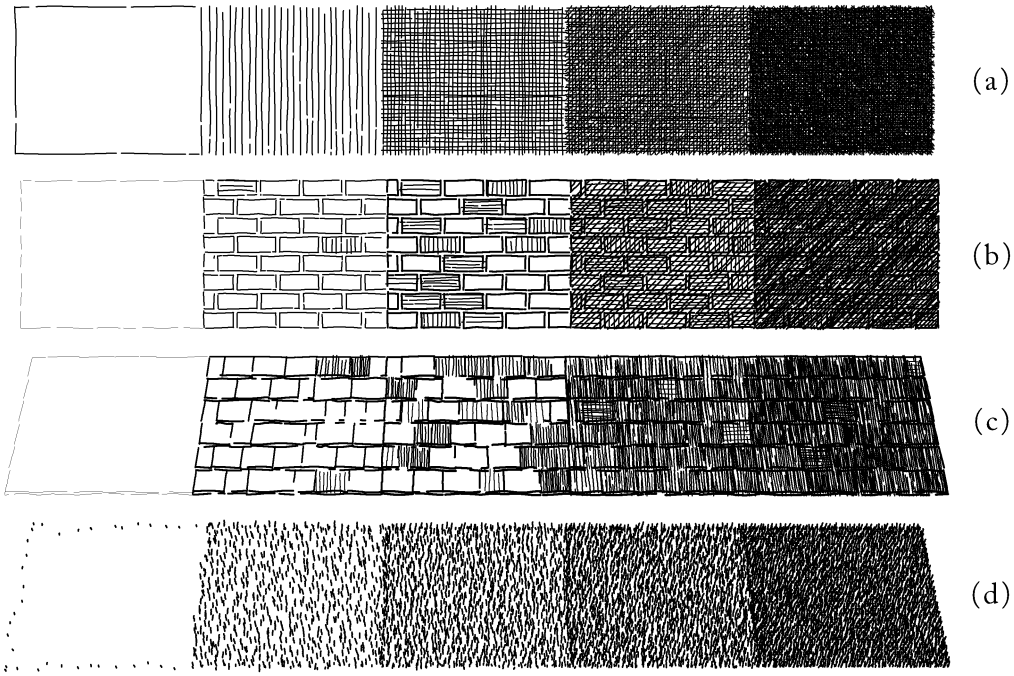


Figure 5-1. A sample of stroke textures. Crosshatching (a), bricks (b), shingles (c), and grass (d).

When more than one hatching direction is used, computing the spacing between the strokes to achieve the target tone T is slightly more complicated, as the overlap between the crossing strokes must be taken into account. As a simplification, we will assume that there is no correlation between each non-overlapping set of strokes. Let us first consider the case of bidirectional crosshatching. If the first hatching direction achieves tone t_1, then a fraction $1-t_1$ of the paper is left exposed. Thus, if the second hatching direction is set independently to achieve tone t_2, the overall tone obtained is

$$T = t_1 + (1-t_1)t_2 \tag{5.1}$$

Imposing $t_1 = t_2 = t$ insures that the hatching directions look identical. Then, the tone t that each hatching direction must achieve independently is easily derived from Equation 5.1:

$$t = 1 - \sqrt{1-T}$$

In general, if n non-overlapping sets of strokes are used for crosshatching, then the tone to be achieved by each set is given by

$$t = 1 - \sqrt[n]{1-T}$$

To see this, we note that the overall tone T_i attained after overlapping $i \le n$ sets of strokes is given by the recurrence relation $T_i = T_{i-1} + (1 - T_{i-1})t$, with $T_0 = 0$. If we let $X = \sqrt[n]{1-T}$, and $t = 1 - X$, it is easy to verify that $T_i = 1 - X^i$ and $T_n = T$.

5.2 Brick and shingle textures

The brick and shingle textures shown in Figure 5-1 (b) and (c) pose the next challenge: the same strokes must be used for conveying both tone and texture simultaneously. Students learning to draw in pen and ink face the same problem, and textbooks on that subject inspired a solution. The idea is to lay strokes on the paper hierarchically. For example, consider the brick texture: the individual bricks are delineated first; then a few bricks are hatched in dark areas; finally, even darker areas, such as shadows, are hatched over the bricks (see Figure 5-1 (b)).

This idea translates to the concept of *prioritized stroke texture*. A prioritized stroke texture is defined as a set of strokes, each with an assigned priority. When rendering a prioritized stroke texture, 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 correct tone is achieved.

For textures such as bricks and shingles, the different aspects of the texture are assigned to different priorities. Again, we consider the brick texture: the outlines of the individual brick elements have the highest priority; the strokes for shading individual bricks have medium priority; and the shading strokes that go over the entire surface have the lowest priority. The same idea applies to other textures, including crosshatching, shingles, and grass. The relative priorities of the strokes used to render these textures can be seen from the collection of strokes used to achieve a particular tone in Figure 5-1.

5.2.1 Accented strokes for shadowing

加重，变浓 “Accenting” or “thickening” feature edges is a technique for providing subtle but important cues about the three-dimensional aspect of an illustrated scene. In the brick texture example, the edges

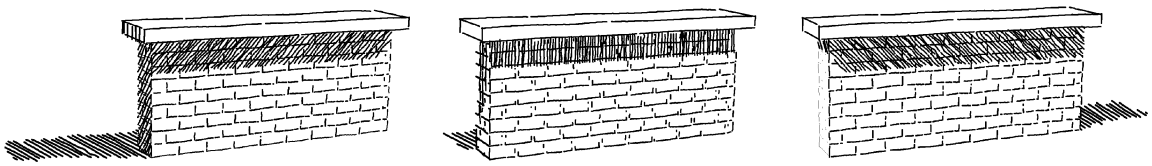


Figure 5-2. Accented strokes for shadowing. The strokes used to delineate each individual brick are accented to reinforce the edges projecting a shadow.

of each brick are drawn according to their relationship with the direction of the light source: edges that cast a shadow are rendered with a thickened stroke, while illuminated edges are drawn with a thin stroke, or even omitted altogether. Figure 5-2 demonstrates this effect.

5.2.2 Dependence of viewing direction

Another important parameter that affects the rendering of certain textures is the viewing direction. For instance, consider the roof of shingles in Figure 5-3. Viewed from above, all edges between the individual shingles are clearly visible; however, when viewed from more to the side, the shingles begin to blend together, and the vertical edges begin to disappear, leaving the horizontal edges predominant. The same principle also affects the brick texture, as can be seen on the two brick walls in Figure 5-6.

5.2.3 Rendering algorithm for bricks

To make the ideas introduced above more concrete, we examine the rendering algorithm for the brick texture in more detail. This description can readily be extended to other types of textures.

Figure 5-4 gives the pseudocode for rendering the “brick” texture, and should be considered in conjunction with the text that follows. Recall that the “brick” texture builds tone out of three sets of strokes: the brick outlines, shading within the bricks, and hatching strokes over the whole surface. Each set of strokes is associated with a different *InkPen*, allowing the user to use strokes of different character for each set. Other user-adjustable parameters associated with the “brick” texture include:

- The width, height, and joint thickness of the bricks.

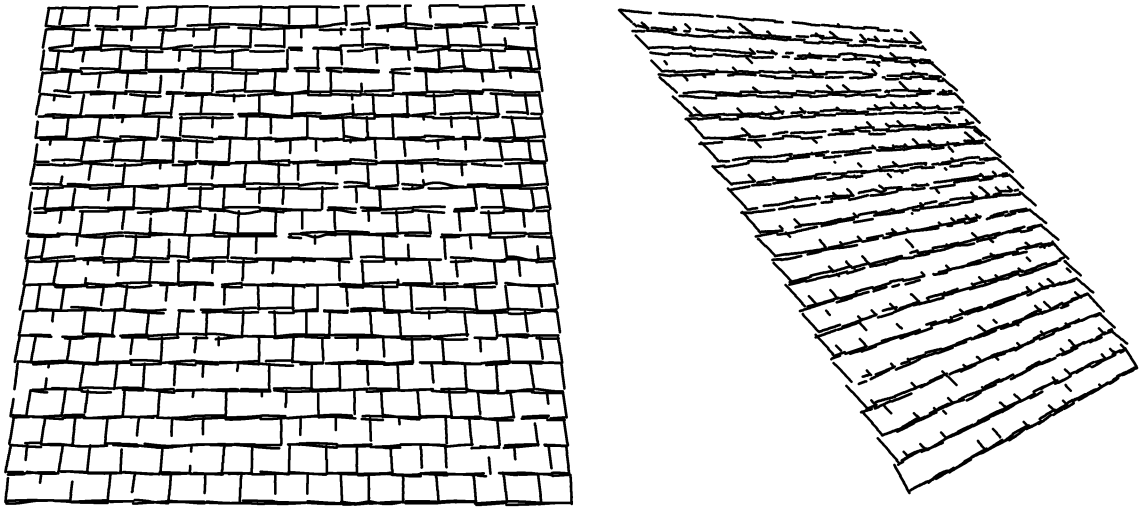


Figure 5-3. Dependence of viewing direction. In the left image, the vertical edges between the shingles are clearly visible. In the right image, as the view moves to a more edge-on direction, these edges begin to disappear.

- Real-valued parameters $k_s \in [0,1]$, and $k_d \in [0,1]$, which modulate how much "shadowing" and "view dependence" affect the outline strokes.
- A tone $T_{shading}$ which controls the darkness of the bricks.

The texture starts by generating a brick pattern. It then proceeds by detailing each brick in the pattern in turn, first by outlining the brick, and then by adding hatching inside the brick. The thickness of the outline stroke is affected by the "shadowing" effect, the "view dependence" effect, and the target tone. Let e_i, $i \in 1,\ldots,4$ denote each of the four brick outline edges. The "shadowing" effect is taken into account by scaling the thickness of the stroke drawn along each edge e_i by the factor

$$\rho_s^i = 1 - k_s \min(1, 1 + \mathrm{w}_i \cdot \mathrm{n}_i)$$

where w_i is a unit vector pointing from the edge mid-point toward the light source and n_i is a unit vector normal to the brick edge under consideration, as shown in Figure 5-5. The value of ρ_s^i reaches its maximum when the light source is behind the edge, and decreases to zero when the light source moves right above the edge. It remains zero when the edge is illuminated.

```
procedure RenderBrickTexture (T:target tone, Q:3D polygon)
   Layout := GenerateBricks (Q, BrickSize)
   for each brick B ∈ Layout do
      DrawBrickOutline (B, T, ViewPoint, Lights)
      if the tone of B is too light then
         ShadeWithinBrick (B, T)
      end if
   end for
   if the overall tone is still too light then
      HatchOver (T, Q)
   end if
end procedure
```

Figure 5-4. Pseudocode for rendering the brick texture.

Similarly, the "view dependence" effect is taken into account by scaling the thickness of the stroke along each edge e_i by the factor

$$\rho_d^i = k_d \left\| \mathrm{v}_i - (\mathrm{v}_i \cdot \mathrm{n}_i)\mathrm{n}_i \right\|$$

where v_i is a unit vector pointing from the edge mid-point toward the view point, as shown in Figure 5-5. The value of ρ_d^i is maximized when the view point is within the plane embedding the surface normal and the edge, and decreases as the view point moves away from that plane.

The tone T_o resulting from the brick outline strokes is evaluated by summing the amount of ink deposited along the outline edges and dividing by the brick area, using the expression

$$T_o = \frac{1}{A} \sum_{i=1}^{4} \rho_s^i \, \rho_d^i \, \mathrm{length}(e_i) \, \bar{\theta}$$

where $\bar{\theta}$ is the maximum thickness of the outline strokes (see Chapter 4), and A is the image-space area covered by the brick. If T_o exceed the target tone T, then the thickness of each stroke is further scaled by the factor $\rho = T / T_o$ before rendering the outline.

Once the outline is drawn, the interior of the brick is shaded. The mean value of the tone for shading the bricks is constant and equal to $T_{shading}$. However, to avoid monotony, the tone of the

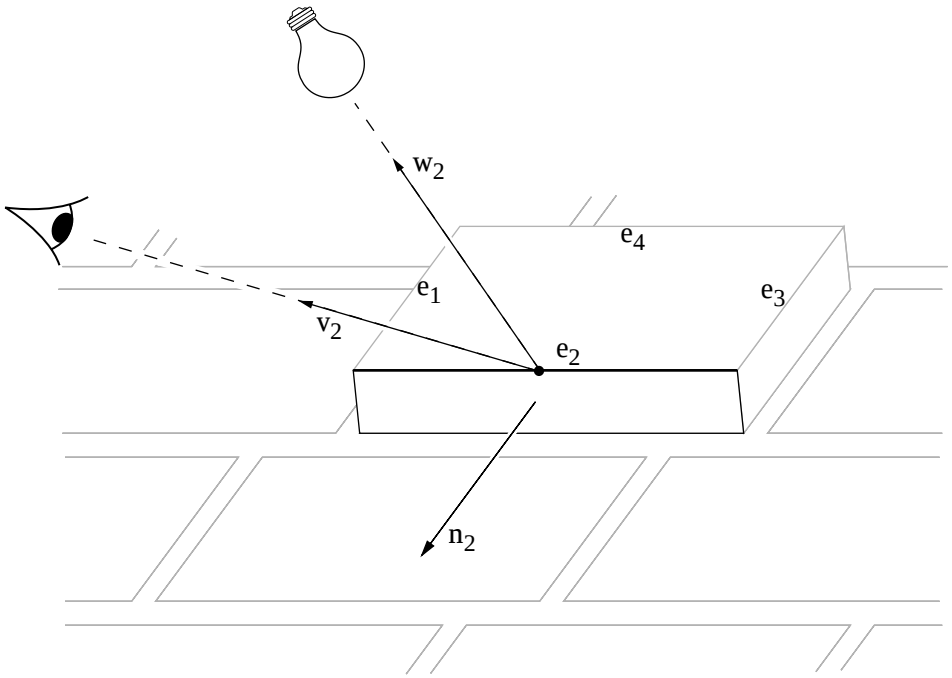


Figure 5-5. Evaluating the “shadow” and “view dependence” effects.

shading varies randomly from one brick to another. To insure that the overall tone does not exceed T, only a few randomly selected bricks are shaded. More specifically, a brick is shaded only with probability $P = (T - T_o)/T_{shading}$.

Finally, if the target tone is still not achieved, the entire surface is hatched. The tone of the hatching is computed by substracting T_o, averaged over all the bricks, and $T_{shading}$ from the target tone.

5.3 Indication

An important principle of pen-and-ink rendering is to suggest a texture without drawing every stroke. This principle of “indication” lends economy to an illustration. It also makes the 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 drawing 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, coming up with a

purely automated method for artistically placing indication is a challenging endeavor. Therefore, the approach that we consider is semiautomatic, whereby the user specifies, at a very high level, where details should appear in the drawing.

For specifying the areas of detail, we borrow the idea of using "fields" generated by line segments from the morphing paper by 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 surfaces 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

$$w(x, y) = \left(k_1 + k_2 \cdot \text{distance}\left((x, y), L\right)\right)^{-k_3}$$

where k_1, k_2, and k_3 are non-negative constants used to adjust the effect of the field. When several detail segments are present, the field at the point (x, y) is defined to be that of the closest segment. In order to create patterns that are not too regular, the field is perturbed by a small random value. Textures such as "brick" and "shingle" evaluate the strength of the field at the center of each brick or shingle. The set of strokes for that element are generated only if the field is above some preset threshold.

Figure 5-6 shows an example of indication achieved with this method. Figure 5-7 shows the detail segments that were used to generate Figure 5-6.

5.4 Resolution dependence

A common problem with figures created by existing computer drawing programs is that they do not scale well when printed at different scales or resolutions. Enlargement is typically performed either by pixel replication, which yields unsightly 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, usually 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 even been preferable from an aesthetic standpoint).

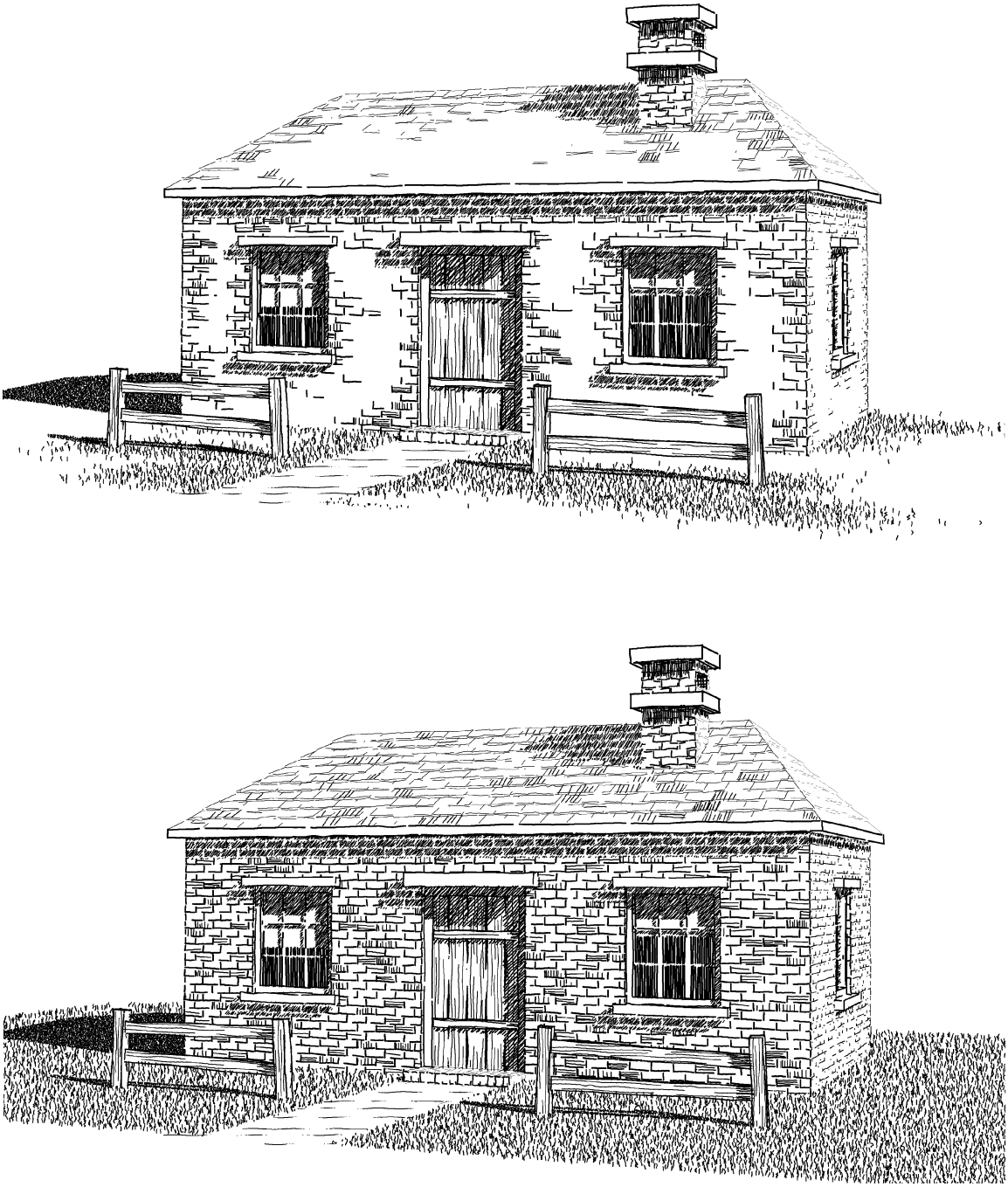


Figure 5-6. Indicating texture. The image at the top uses indication, the bottom image does not.

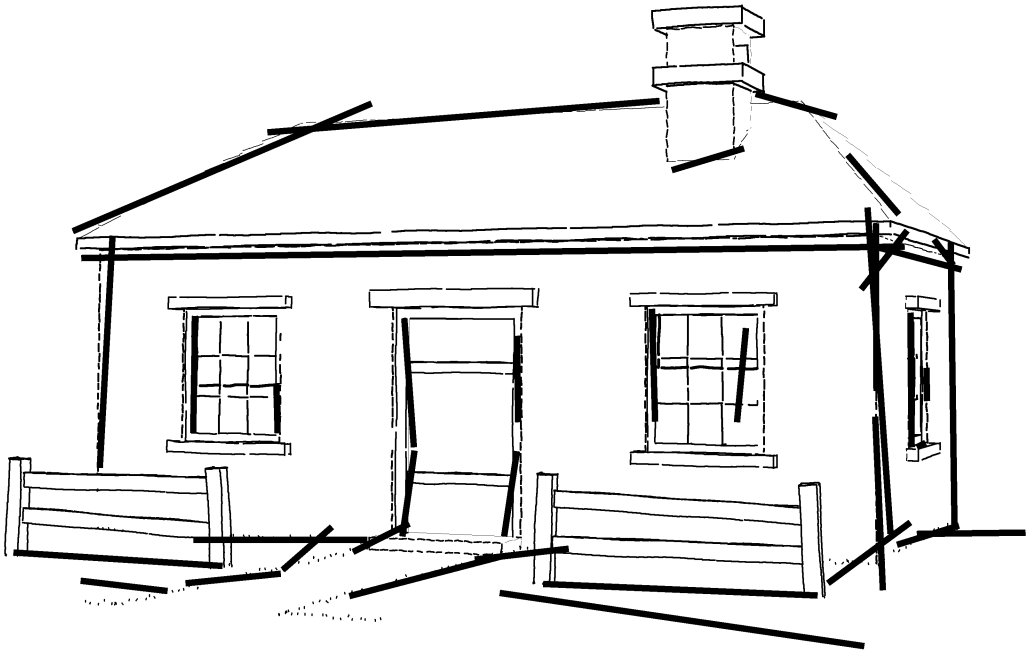


Figure 5-7. Detail segments. The user interactively attached these segments to the surfaces to achieve the indication shown in Figure 5-6.

The prioritized stroke textures described in this chapter do not suffer from this problem. Strokes are generated so as to provide the proper tone and texture for a given image size and resolution. Figure 5-8 demonstrates this feature. It shows the same brick texture at different scales. Note how the smaller image requires fewer strokes to achieve the same tone.

5.5 Outlining

Delineating the *boundary outlines* is another important task handled by the procedural stroke textures. The boundary outlines surround the visible regions 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 outline path for a given visible surface S is assembled from the planar map by visiting as many consecutive edges adjacent to S as possible. An edge E adjacent to S is a candidate for outlining only if S is the frontmost surface, with respect to the view point, of the two surfaces adjacent to E

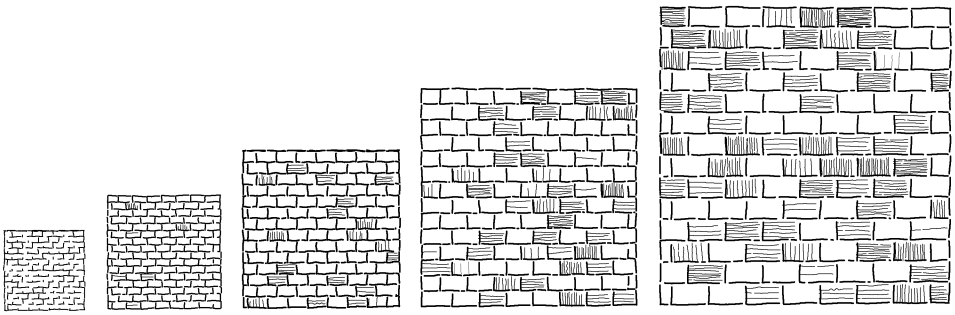


Figure 5-8. Resolution dependence. At the smallest scale, the brick outlines are sufficient to build the tone. As the scale increases, the prioritized stroke texture automatically introduces shading within the bricks to maintain the tone.

(when the two surfaces are abutting, the choice is arbitrary). All the edges are appended into one or more disjoint polyline paths.

Each outline path is then rendered to delineate the surface. However, the principles of pen-and-ink illustration require that two additional factors are taken into account. We describe them next.

5.5.1 Minimizing outlines

Recall from the pen-and-ink principles in Chapter 2 that an edge between two adjacent surfaces S_1 and S_2 in the planar map should be drawn only if the tone difference, or contrast, between S_1 and S_2 is insufficient for disambiguating them. Figure 5-9 illustrates how an outline is omitted in the presence of sharp changes in tone, and added in the absence of tone change. Note how the boundary edges on the vertical and horizontal dividers between the window panes appear only where the contrast with the adjacent surface is low. (These boundary outlines are also omitted when they face the light source.)

The planar map contains the information required to decide whether an outline edge should be drawn or not. Thus, it is a relatively simple matter to mark the edges appropriately, as the outline paths are constructed.

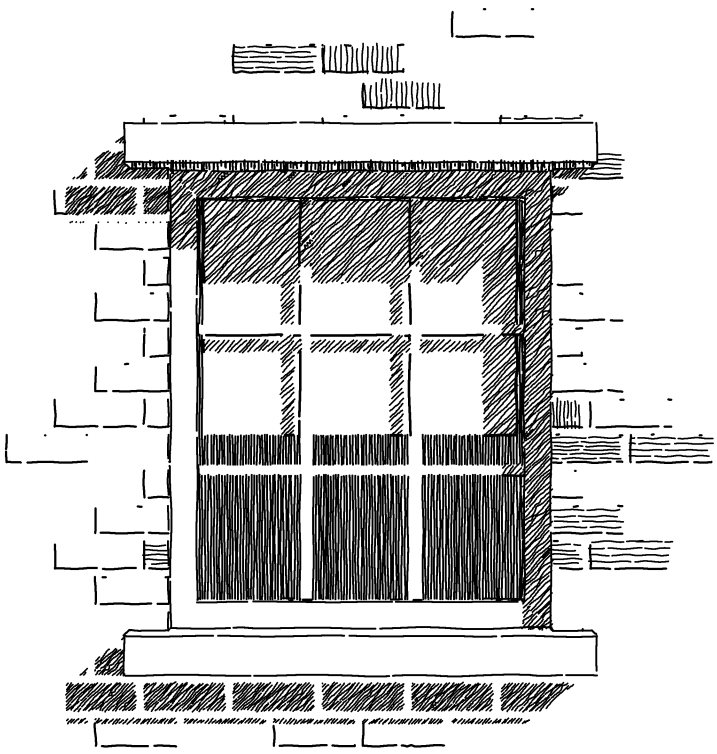


Figure 5-9. Outline minimization. The boundary edges on the vertical and horizontal dividers between the panes appear only where the contrast with the adjacent surface is low. (These boundary outlines are also omitted when they face the light source.)

5.5.2 Expressing texture with outline

To create an effective illustration, it is important that the boundary outline also convey the texture on the surface. The textured boundary outlines for some of the stroke textures are shown in the white squares of Figure 5-1. Figure 5-10 also illustrates the use of textured outlines. The two images are the same, except that all but the boundary outline strokes have been removed from the upper image to present the difference more clearly. Note how the different textures are delineated with distinctive styles.

The procedural stroke textures are responsible for drawing the boundary outlines. Once an outline path has been assembled, it is mapped to texture space and overlaid with the texture pattern. Then, the stroke texture generates the appropriate strokes to render the outline in the proper style.

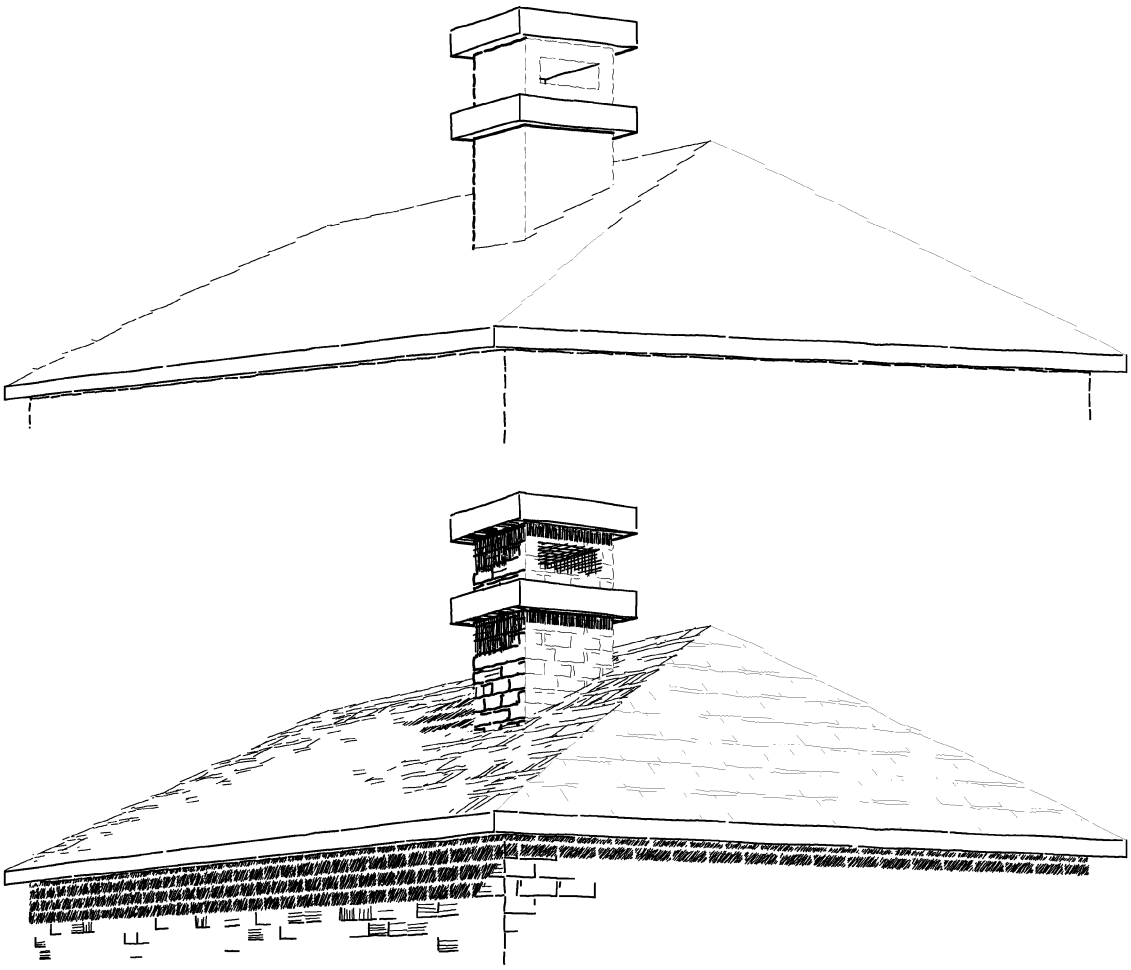


Figure 5-10. Indicating texture with outline. The upper and lower images are the same, except that all but the outline strokes have been removed in the upper image, to clearly show the different outline styles.

5.6 A complex example

To conclude this chapter, we look at a more concrete example. Figure 5-11 shows a pen-and-ink rendering of a simplified model of Frank Lloyd Wright's "Robie House." The 3D model consists of 1043 polygons and six different textures.

Indication was used to lighten the textures at the extremities of the house. In contrast, the center of the house is rendered with very little indication. This use of indication puts more details near the center of the house, and creates an emphasis that draws the viewer's attention there. (Devising a more automated way to create emphasis is one of the many future research directions.)

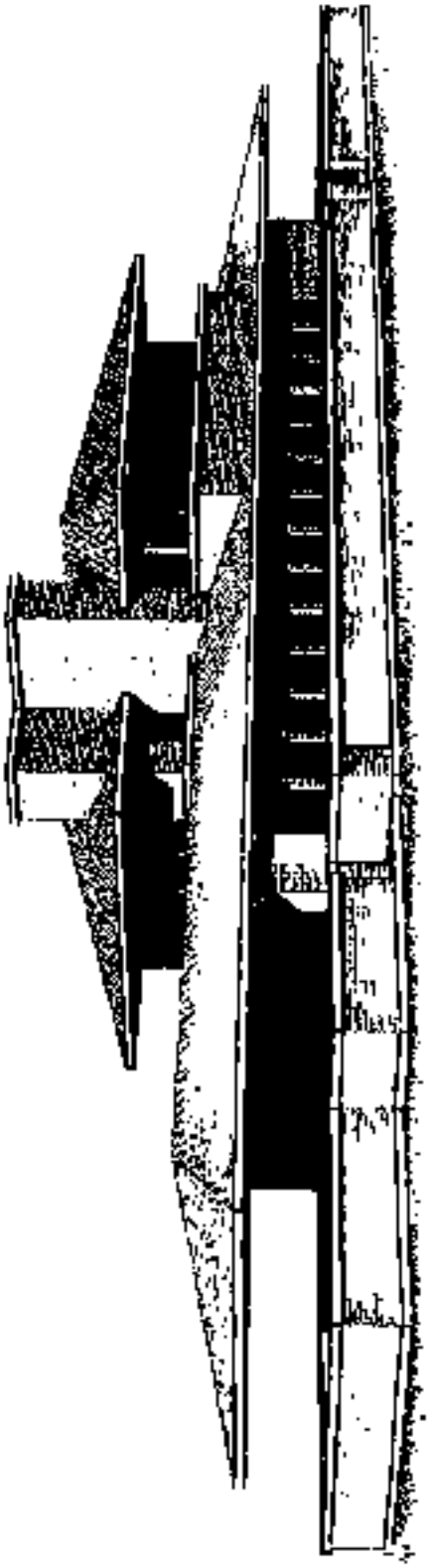


Figure 5-11. Frank Lloyd Wright's "Robie House."

Chapter 6

Stroke Textures on Parametric Free-Form Surfaces

The stroke textures introduced in the previous chapter were developed in the context of a polygonal renderer. This restriction allowed two important simplifications: all strokes paths were line segments, and all surfaces were flat shaded. However, many real-life objects and structures cannot be described adequately with polygons. Therefore, if pen-and-ink rendering is to be a complete framework for generating illustrations, it is essential to extend the range of models that it can handle. With this goal in mind, we'll generalize the concept of stroke texture to parametrically defined free-form surfaces.

6.1 Controlled-density hatching

Curved surfaces require a much more sophisticated approach for generating strokes than flat-shaded polygonal surfaces, which can be hatched in a uniform fashion. First, we'll need a way of orienting the hatching strokes along a surface. In addition, we'll need some mechanism for allowing strokes to gradually disappear in light areas of a surface or in areas where too many strokes converge together. Conversely, we'll also need to allow new strokes to gradually appear in dark areas or in areas in which the existing strokes begin to diverge too much. We'll call such a mechanism *controlled-density hatching*.

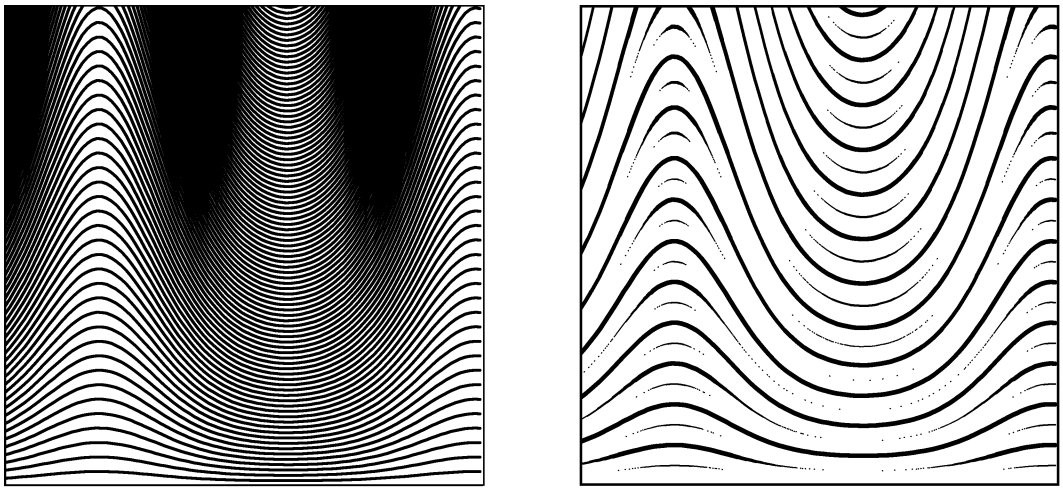


Figure 6-1. Controlled-density hatching for a simple two-dimensional transformation. Rendering isoparametric curves with constant thickness results in an image with varying tone (a). Adjusting the thickness of the strokes yields an image with a constant "apparent tone" (b).

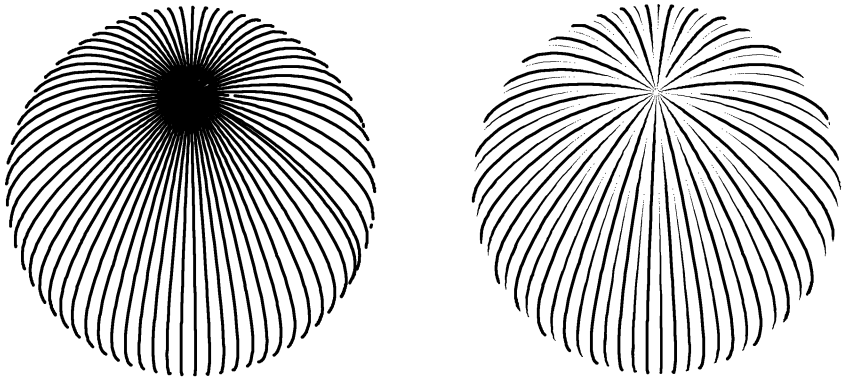
6.1.1 Stroke orientation

The first problem to solve is that of choosing an orientation for the hatching strokes. Ideally, the shape and direction of the strokes should help describe the shape of the surface they hatch. With parametrically-defined surface, we'll use grid lines in the parameter domain (u, v). The grid consists of parallel lines running in one or more user-defined directions. In most cases, we'll simply use isoparametric lines, which run parallel to u and/or v.

Using parametric grid lines is not ideal; in some cases, it is possible for the surface parameterization to deviate significantly from its geometry. Mapping grid lines to strokes in this way also requires a global parameterization across the entire surface. However, this approach works adequately for many surfaces, and it is easy to implement.

6.1.2 Stretching factor

We now turn to the second problem, that of achieving controlled-density hatching. Achieving a given tone by hatching an arbitrary parametric surface is a not a trivial problem. Figure 6-1 illus-



(a) (b)

Figure 6-2. Controlled-density hatching for a perspective view of a sphere. Again, using constant-thickness strokes results in an image with varying tone (a). Using varying stroke thickness keeps the tone constant (b).

trates the difficulty for the case of a simple two-dimensional transformation $M:(u,v)\mapsto(u,v+v\cdot\exp(\sin u))$. In this case, rendering isoparametric curves with constant thickness results in an image with varying tones (a). Our solution is to adjust the thickness of the strokes in order to keep the "apparent tone" constant (b). Figure 6-2 illustrates the same concept, but in this case for a perspective view of a sphere.

In order to solve this problem formally, let's begin by defining a *stroke S* as a pair of functions $(\lambda(t),\theta(t))$, where $\lambda(t)$ is a line in the *parameter domain* (u,v), and $\theta(t)$ is the *thickness of* the stroke at every t. Furthermore, we define the *apparent tone* of an image in the neighborhood of a given point (x, y) in *image space* as the ratio of the amount of ink deposited in that neighborhood to the area of the neighborhood. If the point (x, y) happens to lie on a stroke, the apparent tone can also be expressed as the ratio θ/δ, where θ is the thickness of the stroke at (x, y) and δ is the spacing distance to adjacent strokes.

With these definitions, the controlled-density hatching problem can be formally stated as follows:

Given:

- A parametric surface $S:(u,v)\mapsto(x_w, y_w, z_w)$, which maps points in the parameter domain (u, v) to points in *world-space* (x_w, y_w, z_w);

- a perspective viewing transformation $V:(x_w, y_w, z_w)\mapsto(x, y)$, which maps (visible) points in world space to points in image space (x, y);
- a hatching grid line direction $\mathrm{h}=(h_u, h_v)$ in the parameter domain; and
- a target tone function $T(x, y)$.

Find:

- A set of strokes $\gamma_i=(\lambda_i, \theta_i)$, with lines $\lambda_i(t)$ in the parameter domain running parallel to h, such that the apparent tone of mapping the strokes through $M = V \circ S$ in the neighborhood of (x, y) is $T(x, y)$.

The key step in solving this problem is to determine exactly how the images of two parallel lines in the parameter domain converge and diverge when seen in screen space. In particular, let $\lambda_i(t)$ and $\lambda_{i+1}(s)$ be two parallel lines that are d units apart in the parameter domain. Their images under M, in image-space, are two curves denoted by $\lambda_i'(t)$ and $\lambda_{i+1}'(s)$ respectively. We would like to know the distance d' between the two image-space curves as a function of t. Once we have this distance function $d'(t)$, we can use it to adjust the thickness and spacing of the strokes to compensate for any spreading or compression.

In general, a simple closed-form expression for $d'(t)$ does not exist. Instead, we seek an approximation. To this end, we begin by writing $\mathcal{M}$ as two scalar-valued functions:

$$\mathcal{M}(u,v)\equiv\Big(X(u,v), Y(u,v)\Big)$$

To approximate the behavior of $\mathcal{M}$ in a small neighborhood about $(u,v)=\lambda_i(t)$, we use the Jacobian matrix $J(\mathcal{M})$ [40] evaluated at (u,v)

$$\mathrm{M}=J(\mathcal{M})(u,v)=\begin{bmatrix} X_u & X_v \\ Y_u & Y_v \end{bmatrix}$$

where X_u, X_v and Y_u, Y_v are the partial derivatives of X and Y with respect to u and v respectively. The matrix M is a linear transformation, and can be thought of as the Taylor expansion of $\mathcal{M}$ at (u,v) truncated to the first-order terms. The images of the two grid lines λ_i and λ_{i+1} under M are two parallel lines in image space, denoted by $\mathrm{M}(\lambda_i)$ and $\mathrm{M}(\lambda_{i+1})$. We'll use the distance between these two lines as an approximation for $d'(t)$.

To derive a closed-form expression for the approximated distance function, we first note that the linear transformation $\mathbf{M}$ maps vectors from parameter space to vectors in image space as $[x \quad y]^{\mathrm{T}} = \mathbf{M}\,[u \quad v]^{\mathrm{T}}$. Next, we write the implicit equation $au + bv + c = 0$ for line λ_i in matrix form

$$\begin{bmatrix} a & b \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} + c = 0\,. \tag{6.1}$$

Doing the same for the implicit equation $a'x + b'y + c' = 0$ of $\mathbf{M}(\lambda_i)$ yields

$$\begin{bmatrix} a' & b' \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + c' = 0 = \begin{bmatrix} a' & b' \end{bmatrix} \cdot \mathbf{M} \cdot \begin{bmatrix} u \\ v \end{bmatrix} + c' \tag{6.2}$$

Setting the left-hand side of Equations 6.1 equal to the right-hand side of Equation 6.2, allows us to relate the implicit-form coefficients $\langle a', b', c' \rangle$ and $\langle a, b, c \rangle$. By inverting $\mathbf{M}$, we get

$$\begin{aligned} a' &= (aY_v - bY_u)/\det(\mathbf{M}) \\ b' &= (bX_u - aX_v)/\det(\mathbf{M}) \\ c' &= c \end{aligned} \tag{6.3}$$

In a similar fashion, we can transform the line λ_{i+1} through the linear map $\mathbf{M}$ to get the line $\mathbf{M}(\lambda_{i+1})$ in image space.

Our ultimate goal is to measure how much the curves λ_i' and λ_{i+1}' diverge. We can get an estimate of that measure by looking at the ratio $\eta_i(t)$ of the distance $d_i'(t)$ between the lines $\mathbf{M}(\lambda_i)$ and $\mathbf{M}(\lambda_{i+1})$ in image space, and the distance d_i between the line λ_i and λ_{i+1} in parameter space. To this end, we first note that two parallel lines with implicit-form coefficients $\langle a, b, c \rangle$ and $\langle a, b, c + \delta \rangle$ are offset by the distance $d_i = \delta / \sqrt{a^2 + b^2}$. Using this fact, and Equation 6.3, we can easily derive an expression for $\eta_i(t)$:

$$\eta_i(t) = \frac{d_i'(t)}{d_i} = \frac{\delta / \sqrt{(a')^2 + (b')^2}}{\delta / \sqrt{a^2 + b^2}} = \sqrt{\frac{(X_u Y_v - X_v Y_u)^2 \,(a^2 + b^2)}{(aY_v - bY_u)^2 + (bXu - aX_v)^2}}$$

where X_u, X_v, Y_u, and Y_v are functions of *t*, and, given the hatching direction h, $a = h_v$ and $b = -h_u$.

The scalar-valued function $\eta_i(t)$, called the *stretching factor* of *M*, measures the degree of convergence or divergence that *M* imposes on grid lines near λ_i. Given two grid lines λ_i and λ_{i+1} offset by a small distance δ, the corresponding image-space curves λ_i' and λ_{i+1}' are approximately offset by the distance $\delta'(t) = \eta_i(t)\delta$.

We also define the *maximum stretching factor* $\overline{\eta}_i = \sup_t(\eta_i(t))$. With $\overline{\eta}_i$, we can evaluate the maximum offset distance $\overline{d}_i' = \overline{\eta}_i \delta$ between the two curves λ_i' and λ_{i+1}'.

6.1.3 Simple hatching algorithm

Endowed with the stretching factor $\eta_i(t)$, we are now ready to devise an algorithm to hatch a surface with target tone $T(x, y)$. We'll do so in three steps.

First, we must note that $\eta_i(t)$, because it is derived from a first-order approximation of *M*, is accurate only for very small steps in parameter space. In general, however, the strokes must be offset by a comparatively large distance. To work around this problem, we use an iterative technique. To spread two strokes by a distance $\overline{d}'$, we take a series of small steps of size ϵ in parameter space, updating the maximum stretching factor after each step. Stepping starts from the line λ_i, and proceeds until the accumulated image-space distance, given by $\sum_j \overline{\eta}_j \epsilon$, equals or exceeds $\overline{d}'$.

Second, we must decide what the maximum offset distance between adjacent strokes should be. This value is dictated by the maximum (or darkest) tone $\overline{T}$ that must be achieved anywhere on the surface, as well as the maximum thickness $\overline{\theta}$ of the hatching strokes. To guarantee that enough ink is deposited on the paper to achieve $\overline{T}$, the strokes must be offset by no more than $\overline{d}' = \overline{\theta} / \overline{T}$ units in image space.

Third, we must modulate the thickness of the strokes to accurately render the tone $T(x, y)$. Two factors influence the thickness of the strokes: the actual image-space offset distance $d'(t)$ between the strokes, and the tone to achieve. We can use the iterative algorithm described above to insure that any two adjacent strokes are offset by at most $\overline{d}'$. However, the actual offset distance $d'(t)$ can vary anywhere between 0 and $\overline{d}'$. To compensate for this variation, the thickness of the stroke λ_i' must be scaled by the ratio $d'(t) / \overline{d}' \approx \eta_i(t) / \overline{\eta}_i$. To take the varying tone into account,

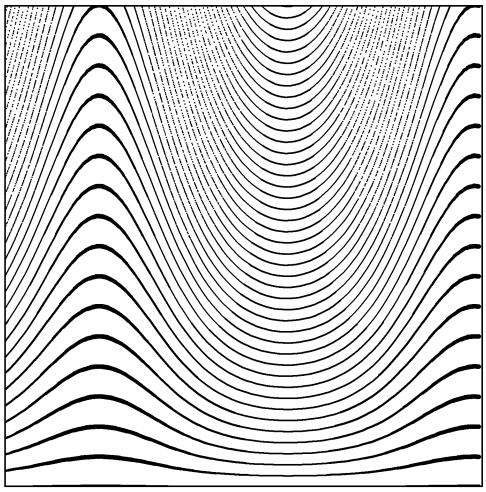


Figure 6-3. Controlled-density hatching. The simple hatching algorithm adjust the thickness of all the strokes simultaneously.

we also scale the stroke thickness by the ratio $T(t)/\overline{T}$. In summary, the thickness of the stroke λ_i' is given by

$$\theta_i(t) = \frac{T(t)}{\overline{T}} \frac{\eta_i(t)}{\overline{\eta_i}} \overline{\theta}$$

Figure 6-3 was produced using this simple hatching algorithm to achieve a constant target tone.

6.1.4 Better hatching with recursion

Although the algorithm described above is able to accurately reproduce tones, it is possible to get "better looking" hatching. The problem with the above algorithm stems from the thicknesses of all the strokes being allowed to vary simultaneously. A much more interesting effect is achieved when short and long strokes are interspersed, as depicted in Figures 6-1 (b) and 6-2 (b). That effect is created by introducing an additional *spreading factor* σ, adjusted by the user. The initial strokes are spread by the distance $\sigma\overline{d}'$ instead of $\overline{d}'$, and the extra space so created is recursively filled with additional *filler strokes*.

To fill the extra space, the offset distance between pairs of strokes is recursively halved, and new fillers strokes are drawn in the middle. The expression for the thickness $\theta_i(t)$ of a filler stroke

$\lambda_i(t)$ is slightly more complicated. To derive it, we first note that the distance between the stroke λ_i and its neighbors, at level ℓ of recursion, is given by

$$d'_\ell(t) = \frac{\sigma}{2^\ell}\frac{\eta_i(t)}{\overline{\eta}_i}\overline{d}' = \frac{\sigma}{2^\ell}\frac{\eta_i(t)}{\overline{\eta}_i}\frac{\overline{\theta}}{\overline{T}} \tag{6.4}$$

With this style of hatching, we would like to achieve the target tone by using the thickest possible strokes before introducing a filler stroke at recursion level ℓ. Thus, if the thickness $\theta_i(t)$ of λ_i is greater than 0 at a given value of *t*, then the thickness of the enclosing strokes is $\overline{\theta}$, and their contribution to the tone in the neighborhood of $(x, y) = \lambda_i'(t)$ is $T_{\ell-1}(t) = \overline{\theta} / d'_{\ell-1}(t)$. Stroke λ_i contributes an additional $T_\ell = \theta_i(t) / d'_\ell(t)$ to the tone. The overall tone that the entire collection of strokes achieves is given by

$$T(t) = T_\ell(t) + T_{\ell-1}(t) = \frac{\theta_i(t)}{d'_\ell(t)} + \frac{\overline{\theta}}{d'_{\ell-1}(t)} \tag{6.5}$$

where $T(t)$ is the target tone on the surface.

Using Equation 6.4 to derive an expression for $d'_\ell(t)$ and $d'_{\ell-1}(t)$, substituting in Equation 6.5, and noting that $\theta_i(t)$ cannot exceed the maximum thickness $\overline{\theta}$, we get

$$\theta_i(t) = \min\left\{\overline{\theta}, \left(\frac{\sigma}{2^\ell}\frac{T(t)}{\overline{T}}\frac{\eta_i(t)}{\overline{\eta}_i} - \frac{1}{2}\right)\overline{\theta}\right\}$$

The recursion stops when $\theta_i(t) \le 0$ everywhere along the stroke.

Typically, σ ranges between 2 and 8. The hatching texture shown in Figure 6-1 (b) was generated using this recursive algorithm and $\sigma = 4$. Figure 6-4 shows the same hatching technique used on a “real” model. Figure 6-4 (a) shows constant-density hatching; Figure 6-4 (b) shows the same model, but with a directional light turned on to create smoothly varying tones. The image in Figure 6-4 (c) uses environment and texture mapping techniques introduced in the next section.

6.2 Controlling tone with texture mapping

The ability to exert “fine-grain” control over the tone brings a new range of possibilities. Notably, it is now possible to use traditional image-based texture mapping techniques to control the tone of

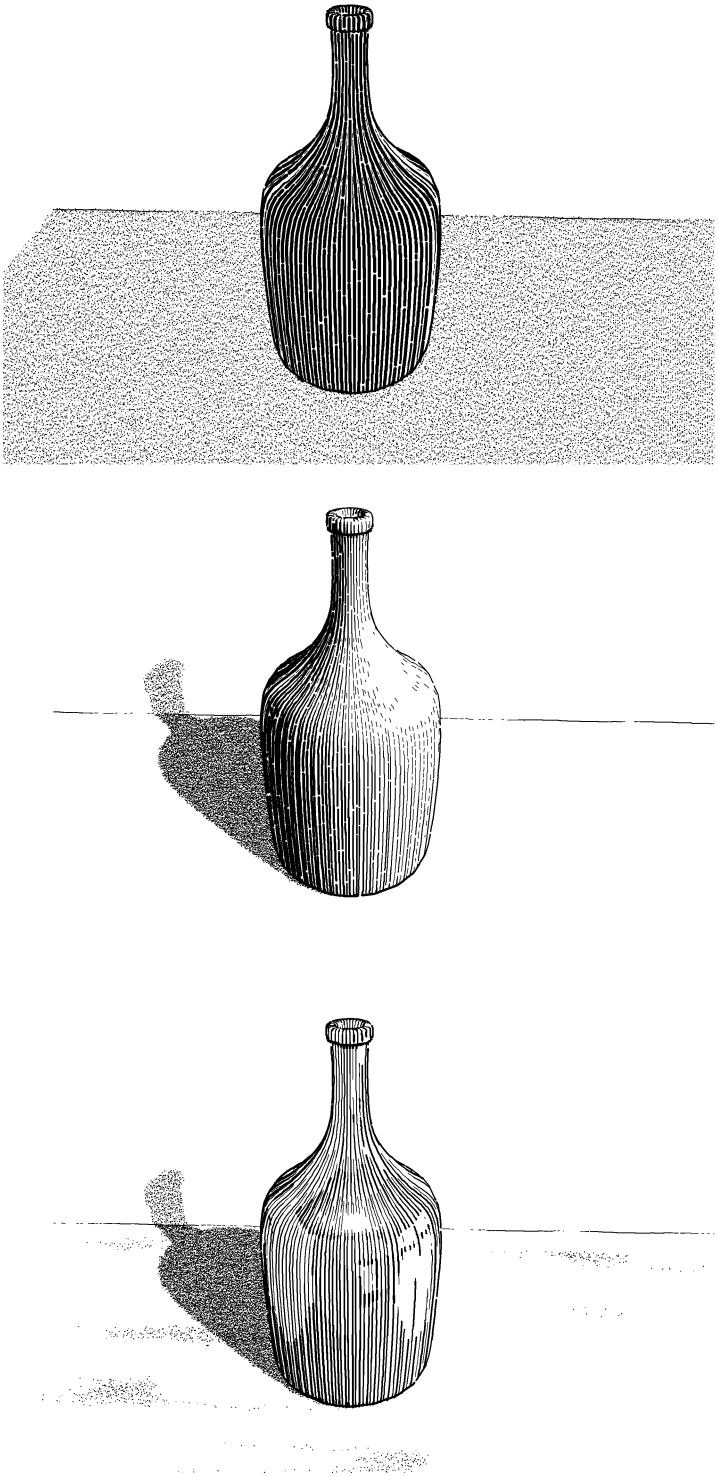


Figure 6-4. Glass bottle. The scene is shown without lighting in the top image, with one directional light in the middle, and finally with texture and environment maps in the bottom image.

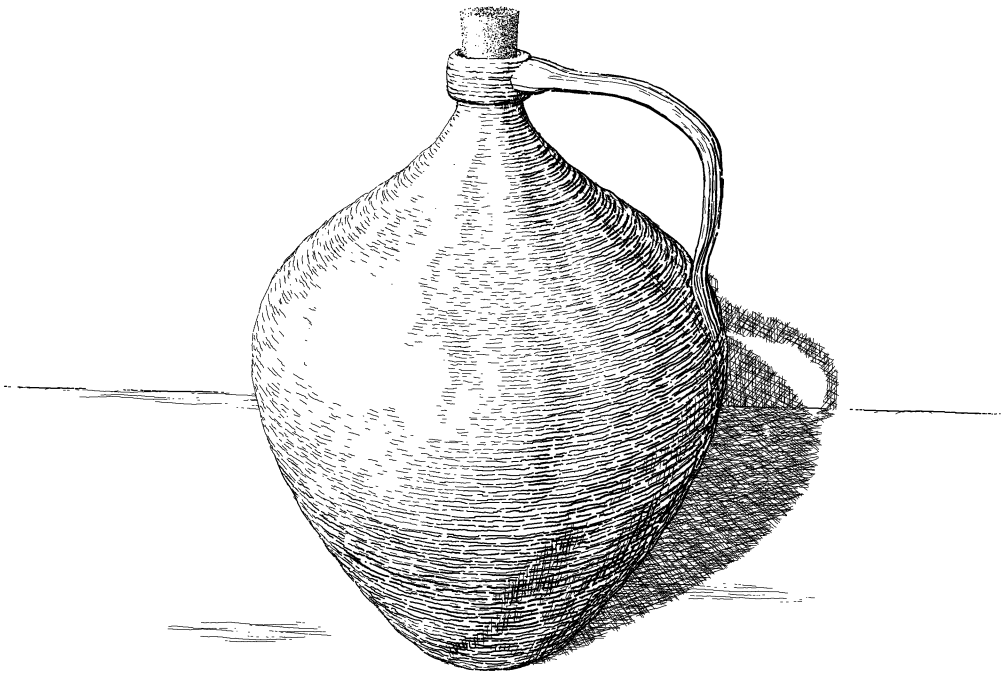


Figure 6-5. Ceramic jug. A bump map is used to give an irregular appearance to the jug's surface, and a texture map is responsible for the stains on the floor.

surfaces. For instance, Figure 6-4 makes use of environment mapping to create a reflection pattern on the bottle. A texture map was used to create a pattern on the bowl in Figure 6-6. In the same figure, a bump map was used to emboss the word “milk” on the jug. Bump mapping was also used on the jug in Figure 6-5, to give an irregular hand-made appearance to its surface.

6.3 Other texture styles

So far, we have focused our attention on hatching — using strokes oriented in a single direction to create tone. However, the concept of prioritized stroke textures introduced in the previous chapter can be used to create more interesting images. We consider a few examples:

Crosshatching. Figures 6-6 and 6-5 show how crosshatching can be used to create dark shadow tones. Crosshatching is also used, in a more subdued style, on the jug in Figure 6-5 and on the cane in Figure 6-8.

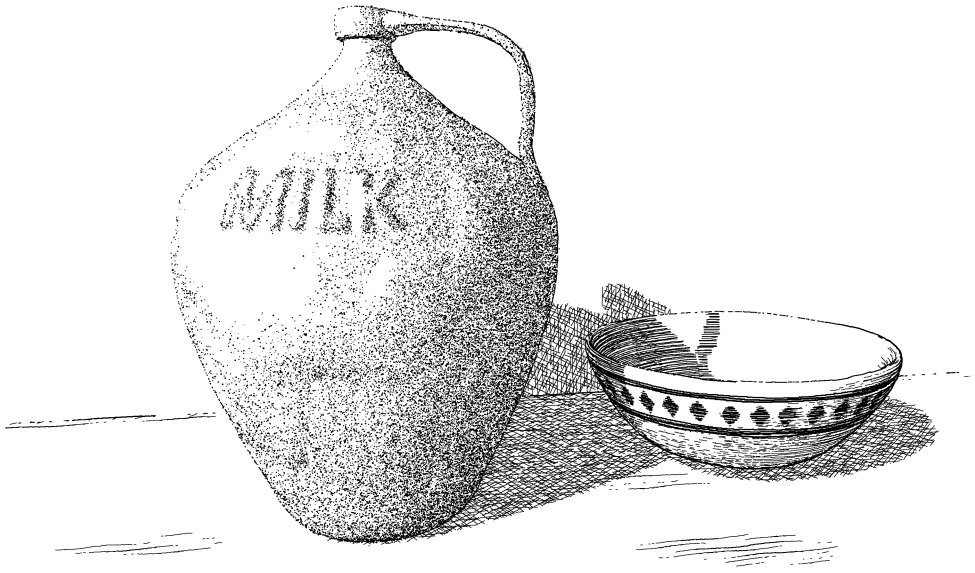


Figure 6-6. Ceramic jug and bowl. Texture maps are responsible for the pattern on the bowl, and the stains on the floor. A bump map is used to emboss the word “milk” on the jug, and also gives the jug some irregularities to make it look hand-made. Also notice the use of stippling on the jug, and crosshatching for the shadows on the floor and background.

Wood texture. The wood texture shown in Figure 6-7 uses a variety of strokes to draw the staves. Thin wavy strokes are used to convey the illusion of wood grain, while thick strokes of varying thickness delineate the gaps between the staves.

Stippling. Stippling is commonly used in pen-and-ink rendering to draw soft and/or grainy materials. It is also useful for rendering subtle tonal variations that would be difficult, or even impossible, to convey with crosshatching. Figures 6-4, 6-6 and 6-8 use stippling.

6.3.1 Stippling texture algorithm

A few words about the “stippling” texture algorithm are in order. The stippling algorithm starts by generating a grid of hatching curves, using the controlled-density hatching algorithm described

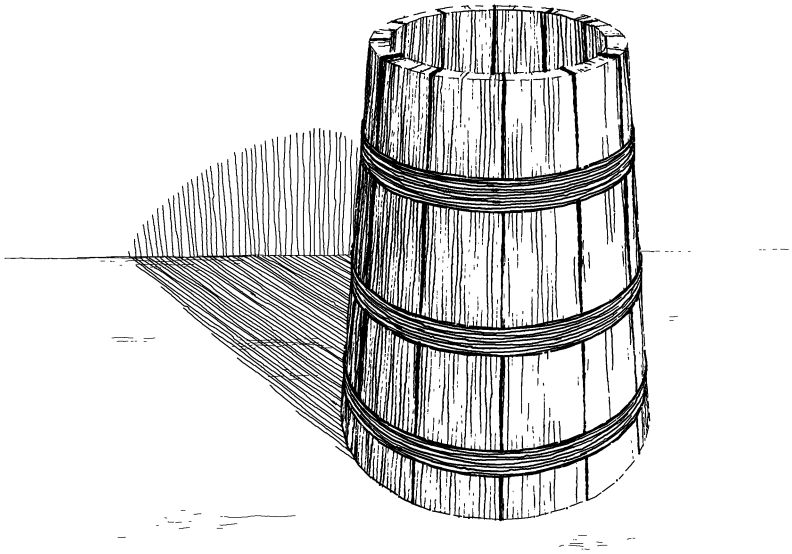


Figure 6-7. Wood bucket. The bucket is modeled with a single surface of revolution. A stroke texture conveys the illusion of wood by using a variety of strokes. A bump maps is used to create some irregularities on the bucket, while a texture map is responsible for the stains on the floor.

earlier. However, the hatching curves are not rendered directly; instead, they are passed on to a stippling routine that lays stipple dots along the curves. The positions of the stipples are perturbed with a uniform random perturbation function. Likewise, the size of the dots is varied slightly.

The maximum density of the stipple dots, and therefore the spacing between the hatching curves, is set so as to achieve the darkest tone $\overline{T}$ on the surface. The actual density of the stipples is adjusted as a function of the actual tone, as well as the actual spacing between the hatching curves. For a given hatch curve $\lambda'(t)$, that is accomplished by drawing a stipple dot only with uniform probability

$$P(t) = \frac{T(t)}{\overline{T}} \frac{\eta(t)}{\overline{\eta}}$$

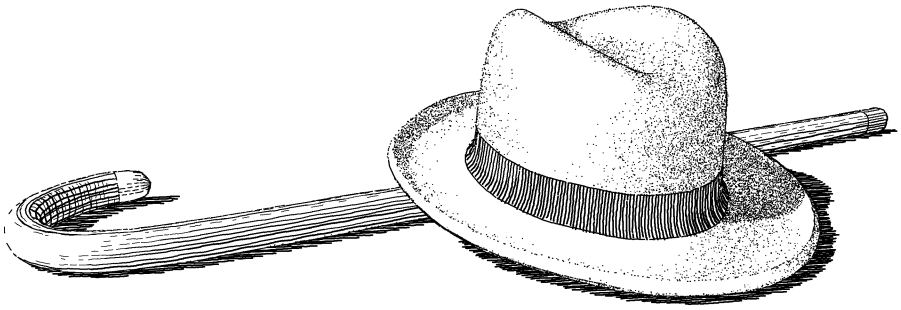


Figure 6-8. Hat and cane. Both the hat and the cane are modeled with B-spline surfaces. The stipple texture on the hat conveys the feeling of a soft velvety material. It also reproduce the smooth tonal variations well. Also note the use of crosshatching on the cane.

Chapter 7

Planar Map Algorithms

One of the essential data structures of the pen-and-ink rendering system is an image-space planar map representing the visible surfaces. It is used for three different tasks during the rendering process: generating the outline strokes for delineating the visible surfaces, clipping strokes to the visible portions of a surface, and clipping strokes to shadow regions.

In this chapter, we'll describe two different algorithms for constructing the planar map. The first algorithm is relatively simple to implement, but is best suited for polygonal models. The second algorithm is more appropriate for scenes containing free-form surfaces. For each algorithm, we'll also describe how the resulting planar map is used for outlining and stroke clipping.

7.1 Planar map from polygonal models

7.1.1 Construction algorithm

The algorithm that constructs the planar map from a polygonal model consists of the following steps:

1. Convert the model geometry to a 3D BSP tree representation.
2. Compute the shadows using Chin and Feiner's shadow volumes.
3. Build an image-space 2D BSP tree from the visible polygons.
4. Build the planar map data structure from the 2D BSP tree.

We now describe these steps in more detail.

Steps 1 and 2 have been well-documented elsewhere [6, 13, 14]. The outcome of these two steps is a set of convex polygons that are split between "shadow" and "non-shadow" regions. (If several light sources cast shadows, the polygons are split and distributed among the different shadow configurations.) We note that, because the algorithm starts by building a 3D BSP tree, the polygons can easily be depth-ordered with respect to any view point.

Step 3 uses a 2D BSP tree to compute a partition of the image plane, such that each region in the partition corresponds to a single visible polygon. To this end, the 3D polygons are examined in front-to-back order. Each polygon is clipped to the view volume, projected to image-space, and then inserted into the 2D BSP tree. Insertion into the BSP tree is virtually identical to the set union operation described by Thibault and Naylor [41]: at each internal node V with binary partitioner π, the polygon is split by π, and the resulting polygon fragments are recursively inserted in the subtrees according to which side of π they fall on, until they reach leaf nodes. Once a fragment q of the polygon reaches a leaf node L of the BSP tree, two outcomes are possible:

1. *Leaf L is empty*: replace L with a BSP tree representation for the 2D region enclosed by q.
2. *Leaf L is not empty*: the region corresponding to L is already covered by a polygon inserted earlier. Since the polygons are inserted in front-to-back order, q is further away from the view point; therefore, q is discarded.

The resulting 2D BSP tree forms a partition of the 2D image space, with each cell in the partition (leaf node) corresponding either to a unique front most polygon in the 3D scene, or to the background.

Step 4 constructs the planar map with the help of the 2D BSP tree built in the previous step. The planar map is maintained using a half-edge data structure [28]. Initially, it consists of a single rectangular face F_0 representing the entire image-space viewport. To build the planar map, F_0 is inserted into the BSP tree. As a planar map face F_i is inserted into an internal tree node V with binary partitioner π, two outcomes are possible:

1. F_i *falls entirely on one side or the other of* π: recursively insert F_i into the corresponding subtree.

2. F_i *intersects the partitioner* π: subdivide F_i into smaller faces F_{i+1} and F_{i+2} by creating a new edge aligned with π, and recursively insert F_{i+1} and F_{i+2} into the corresponding subtrees.

When face F_i reaches a leaf node, it is a duplicate of the 2D region corresponding to that node. At that point, the leaf node receives a pointer to F_i, and F_i receives a pointer to the 3D polygon that gave rise to that leaf node. If the leaf node is empty, F_i is tagged as a background face. These pointers give access to the 3D polygons from both the BSP tree and the planar map, allowing information such as tone and z-depth to be computed "lazily" rather than stored in either data structure.

Because of numerical inaccuracies, it is possible that a BSP tree leaf node will never receive a matching face in the planar map. Fortunately, such leaf nodes correspond to very thin regions in the partition, and they can be ignored without resulting in any noticeable artifact in the rendering.

Geometrically, the planar map and the 2D BSP tree are redundant: they encode the same 2D partition. However, the two data structures are amenable to different tasks. The planar map is useful for generating the outline edges because it explicitly encodes the adjacencies between the different 2D regions. In contrast, the BSP tree is efficient for clipping strokes through set operations, but it does not readily allow searching among neighboring polygons.

7.1.2 Using the planar map for outlining

The outline path for a given 3D polygon Q_1 is assembled by visiting as many consecutive planar map edges adjacent to Q_1 and any other polygon Q_2 as possible. In addition to being adjacent to Q_1, each edge E must meet other outline-minimization criteria to be considered for outlining. These criteria includes:

- *Contrast* — the contrast is computed by evaluating the tone on both Q_1 and Q_2 and taking the difference.
- *Depth* — a ray-tracing technique is used to compute the distance from the view point to both Q_1 and Q_2. Edge E is considered for outlining Q_1 only if Q_1 is the front-most polygon.

The edges are then handed over to the stroke texture attached to Q_1 for generating the outline strokes.

7.1.3 Using the planar map for stroke clipping

All the strokes that a texture generates must be clipped against the visible portions of the textured 3D polygon. To this end, the path of each stroke is clipped using the 2D BSP tree. This process is identical to the polygon insertion algorithm described above: the path is recursively "pushed down" the tree, and split into segments each time it intersects a partitioner. Once a segment μ of the path reaches a leaf node L, it is marked *visible* if L arose from the same 3D polygon. In addition, if the stroke is used for rendering a shadow, μ is visible only if the 3D polygon is a shadow polygon. All adjacent visible strokes segments are then reassembled into one or more contiguous stroke paths before being passed on to the *InkPen* associated with the texture for rendering.

7.2 Planar map from free-form surfaces

One simple approach for constructing the planar map from free-form surfaces would be to approximate the surfaces with "smooth" polygon meshes [13], and then to use the algorithm described in the previous section. However, BSP trees are not well-adapted to handle polygonal models derived from free-form surfaces. The very large number of facets, together with the "smoothness" of the resulting polygonal model, tend to give rise to very large and unbalanced trees, causing the BSP tree algorithms to exhibit their worst-case behavior.

Naylor and Rogers have proposed a way to work around this problem by including Bézier curves into 2D BSP trees [29]. They also suggested some means by which their work can be extended to 3D BSP trees. However, extending their method to handle a large class of parametrically defined surfaces and shadow volumes is not a simple matter. Therefore, we seek a planar map construction algorithm that avoids using BSP trees altogether. The approach we'll use does begin by tessellating the surfaces, yielding polygon meshes. However, we'll build the planar map directly from the meshes rather than using intermediate BSP trees. The resulting algorithm has three main steps:

1. Tessellate the free-form surfaces, yielding a polygon mesh for each surface in the model.
2. Compute a higher-resolution approximation of the silhouette curves.
3. Build the planar map from the resulting polygon meshes.

The first step converts every object in the scene into a polygon mesh representation. The resolution of the tessellation is chosen so as to provide a reasonably accurate approximation of the object's geometry. The simplest approach, used for the prototype pen-and-ink renderer, uses a tessellation

level set by the user. Better tessellation algorithms for parametric surfaces, based on flatness criteria, have been discussed elsewhere [5, 12, 13].

We consider the next two steps in more detail.

7.2.1 Refining the silhouette curves

The second step computes higher-resolution approximations for the silhouette curves of the free-form surfaces. This step is required to obtain smooth and accurate silhouettes without relying on unduly fine tessellations. The algorithm we're about to describe is inspired by the "curve extraction" algorithm developed by Elber and Cohen [11], except that it operates directly on the polygon mesh data structure, instead of the parametric surface.

The algorithm begins by identifying all the mesh edges that span a silhouette by examining the surface normal vectors at each edge endpoint. Assuming that the original tessellation is fine enough to insure that no edge spans silhouettes more than once, then an edge *E* spans a silhouette whenever the projections of the normal vectors onto the viewing direction point in opposite directions. More formally, let V_1 and V_2 be the endpoints of *E*, and let N_1 and N_2 be the surface normal vectors at V_1 and V_2 respectively. Furthermore, let *W* be the location of the view point. For each endpoint, the algorithm evaluates

$$\nu_i = \text{sign}\Big((V_i - W)\cdot N_i\Big),\ i = 1,2$$

where the function sign(*x*) return -1, 0, or 1 when *x* is negative, zero, or positive, respectively. If $\nu_1 \nu_2 = -1$ then *E* spans a silhouette and is tagged accordingly.

Next, the algorithm refines the tessellation in the neighborhood of the silhouette curves. To this end, every mesh face *F* that is adjacent to a tagged edge is subdivided. In the case of a tessellation along isoparametric lines, the parameter intervals for *F* are halved, resulting in four smaller faces.

This process of "detection-refinement" is repeated a number of times to better capture the silhouette curves with the tessellation. In its simplest (and current) form, the algorithm simply repeats this process a fixed number of times.

Finally, to improve the smoothness of the silhouette curves, a numerical technique is used to increase its accuracy. All the edges that span a silhouette are once again identified. For each edge *E*

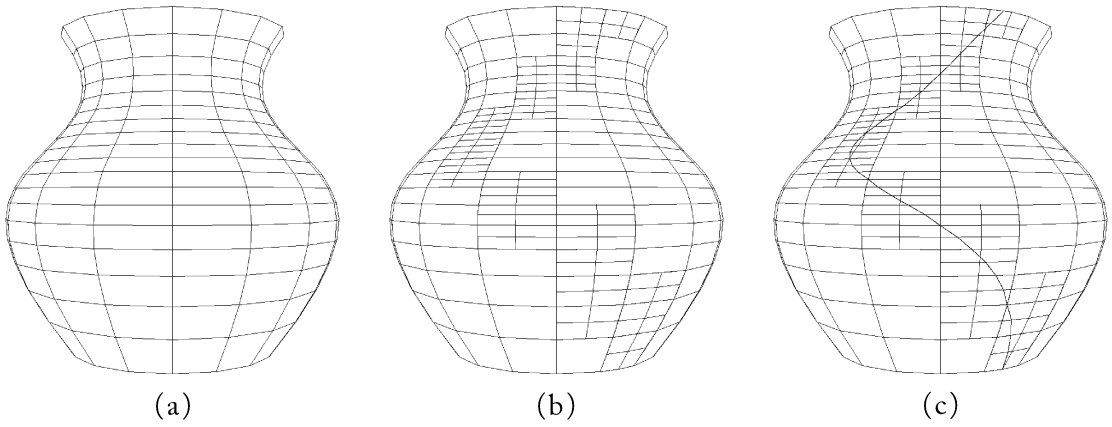


Figure 7-1. Silhouette curve refinement. The original tessellation (a) is refined in the neighborhood of the silhouette curve (b), and a numerical technique is used to enhance the approximation (c).

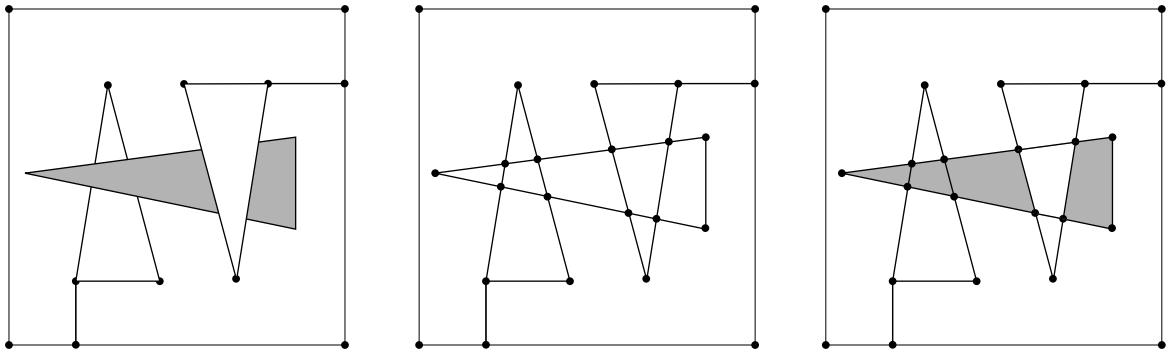
that spans a silhouette, a secant method is used to find the point, along *E*, that lies on the silhouette curve. The edge is then split by introducing a new vertex at that point. Finally, all the silhouette points are joined into a continuous piecewise-linear silhouette curve.

Figure 7-1 illustrates the silhouette curve refinement process graphically.

7.2.2 Building the planar map

The third and last step of the algorithm builds the planar map from the polygon meshes. Occlusions are also resolved during this step. The following algorithm solves this dual problem, and is best thought of as the Weiler-Atherton algorithm [42] modified for storing its results in a planar map data structure. In the development that follows, we'll assume non-intersecting surfaces. This assumption simplifies the algorithm significantly. However, the algorithm could in principle be extended to handle intersecting surfaces.

Just as for the polygonal model algorithm described in section 7.1, a half-edge data structure is used to store the planar map [28]. To begin, the planar map is initialized to a single rectangular face representing the entire viewport. Then, each face of each polygon mesh is inserted in turn. Each face is first clipped against the view volume, then it is projected onto the image plane, and finally, the resulting 2D polygon is inserted into the planar map.



(a) (b) (c)

Figure 7-2. Inserting a polygon into the planar map. The polygon is shown before merging in (a). The polygon is first merged with the topology of the planar map (b), then visibilities are resolved for each planar map face surrounded by the polygon (c).

To insert a polygon *Q* into the planar map, its topology is first merged with that of the planar map (Figure 7-2 (a) and (b)). The merging process is straightforward: a vertex of *Q* is chosen arbitrarily, and its location in the planar map is determined. From there, the algorithm follows the polygon boundary, splitting planar map edges where they intersect with *Q*, and introducing new edges to represent the boundary of *Q*. Once the topologies are merged, occlusions must be solved for each planar map face that is surrounded by *Q*. For this purpose, every planar map face keeps a link to the 3D polygon mesh face it originated from. A raytracing approach is used to determine whether the currently inserted polygon or the polygon linked to the face is closest to the view point. More precisely, a ray is fired through the middle of each edge bounding the planar map face. The intercept distances to the two 3D faces are summed, and the face yielding the smallest sum is proclaimed closest to the view point. If the polygon *Q* is the closest, the information in the planar map face is updated to reflect the new frontmost polygon (Figure 7-2 (c)).

7.2.3 Robustness issues

A common problem in geometric algorithms is that of maintaining consistency between the topological and geometric information when imprecise computations like floating-point arithmetic are used [17, 19, 35]. The planar map algorithm described above is certainly not immune to this problem.

To address this issue, we describe a method inspired by the work of Gangnet et al. [15]. The method relies on infinite-precision rational arithmetic to represent intersections exactly. To begin, we express the point of intersection I of two 2D line segments (V_1,V_2) and (V_3,V_4) as $I = V_1 + \kappa(V_2 - V_1)$. The scalar κ is the affine coordinate of I relative to (V_1,V_2); it is computed with the expression

$$\kappa = \frac{(V_1 - V_3)\times(V_2 - V_1)}{(V_4 - V_3)\times(V_2 - V_1)} \tag{7.1}$$

where $\times$ denotes the vector cross-product. If the coordinates of the segment endpoints are taken on an integer lattice, then the numerator and denominator in Equation 7.1 are also integers, and κ can be stored as an exact rational number.

To support this scheme, the traditional half-edge data structure used to store the planar map has to be modified slightly; in addition to the *vertex*, *edge*, and *face* entities, we introduce the *support segment*. A support segment is stored as a pair of points (V_1,V_2) with integer coordinates. Every edge is associated with a support segment. That is, each edge maintains a pointer to the support segment it originated from, and a pair of rational numbers (κ_t, κ_h) representing the affine coordinates of the tail and head vertex, respectively, relative to the support segment. This mapping between edges and support segments is many-to-one: several colinear edges may share a single support segment. In this modified planar map data structure, the vertices only serve as a placeholder for the topology and do not store any geometric information.

We now turn our attention to the problem of intersecting two edges reliably. Let E_p be an edge with support segment $\Sigma_p = (V_1, V_2)$, and with endpoints having affine coordinates (κ_t^p, κ_h^p) relative to Σ_p. Likewise, let E_q be an edge with support segment $\Sigma_q = (V_3, V_4)$, and with endpoints having affine coordinates (κ_t^q, κ_h^q) relative to Σ_q. To determine if the two edges intersect, we first compute the point of intersection of Σ_p and Σ_q as affine coordinates κ_i^p relative to Σ_p, and κ_i^q relative to Σ_q, using Equation 7.1. Edges E_p and E_q intersect if and only if $\kappa_i^p \in [\kappa_t^p, \kappa_h^p]$, and $\kappa_i^q \in [\kappa_t^q, \kappa_h^q]$.

In addition to computing intersections reliably, another task essential to the half-edge data structure is arranging edges incident to a vertex in counterclockwise (ccw) order. Fortunately, the rational arithmetic framework described above can help us here as well. We begin by dividing the space

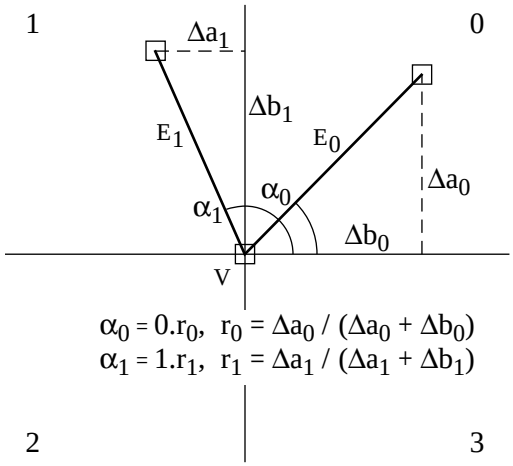


Figure 7-3. Sorting edges incident to a vertex. The incidence angle is represented with a tuple $k_i.r_i$.

about the vertex into four equal quadrants aligned with the coordinate axis (Figure 7-3). The quadrants are number from 0 to 3 in ccw order, starting from the top-right quadrant. Furthermore, given an edge E_i incident upon vertex *V*, let Δa_i be the length of the projection of E_i on the next quadrant boundary in ccw order, and let Δb_i be the length of the projection of E_i on the previous quadrant boundary. The angle of incidence of an edge E_i upon vertex *V* is defined as the tuple $k_i.r_i$ where $k_i \in \{0,1,2,3\}$ is a quadrant number, and r_i is the ratio $\Delta a_i / (\Delta a_i + \Delta b_i)$.

THEOREM 7-1. *The ratio r_i takes its value in the range [0, 1], and increases monotonically with the "true" angle β between the edge E_i and the nearest clockwise quadrant boundary.*

PROOF. We have $1/r_i = (\Delta a_i + \Delta b_i)/\Delta a_i = 1 + \cot\beta$. Thus $1/r_i \in (\infty, 1]$ and decreases monotonically with β. Conversely, $r_i \in [0, 1]$ and increases monotonically with β.

The ratio r_i can be computed as an exact rational number, using congruent triangles, from the endpoints of the support segment of E_i. Thus, the tuple $k_i.r_i$ can be used to impose a total ordering on the edges incident upon *V*.

In the implementation of the prototype pen-and-ink renderer, the discrete lattice into which the support segments take their endpoints is stored with 14-bit integer numbers. This representation

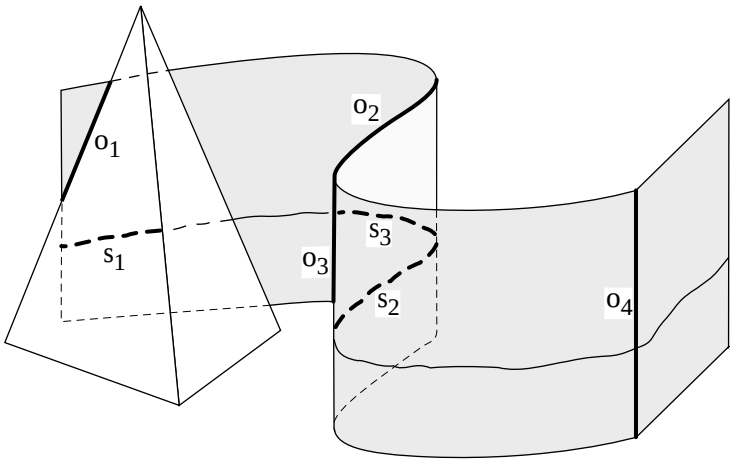


Figure 7-4. Outlining and stroke clipping.

allows all intersections and all vertex-incidence angles to be represented using 32-bit rational numbers (although intermediate computations sometimes require 64-bit numbers). This choice also imposes a resolution of about 800 dot per inch over a 10- by 10-inch image area.

7.2.4 Using the planar map for outlining

Just as for the polygonal renderer, outline paths are assembled by visiting edges of the planar map. However, in the presence of curved surfaces, there are more cases to consider. Four types of edges can give rise to an outline edge (see Figure 7-4):

- An edge such as o_1 that bounds two regions belonging to different objects. This outline edge is associated with the object closest to the view point. If that determination cannot be made, the choice is arbitrary.
- An edge such as o_2 that bounds two regions that belong to the same object, but whose corresponding 3D mesh faces have opposite orientations — backfacing or frontfacing — with respect to the view point.
- An edge such as o_3 that bounds two regions that belong to the same object and have the same orientation, but different depths (distance from the view point).
- An edge such as o_4, that comes from a break (C^1 discontinuity) along the surface.

An outline path is assembled by appending as many adjacent outline edges as possible. The thickness of the stroke along an outline edge is affected by two factors:

- *The tone value on the surface* — allowing an outline edge to fade away in regions of highlight reinforces the quality of the shading.
- *The contrast between two adjoining regions* — if the tone difference between two adjacent surfaces is small, a darker outline is required to mark the boundary.

The degree to which these two criteria affect the outline is selectable by the user.

7.2.5 Using the planar map for stroke clipping

As for outlining, clipping strokes is slightly more complicated for curved surfaces. The free-form surface hatching algorithm creates 3D polyline stroke paths. Each of these paths must be clipped so as to keep only those segments of the paths that fall on visible portions of the surface. To this end, the path is first split at the silhouette points, yielding segments that face either toward or away from the view point on their entire length. Each resulting segment is then projected to image space, and clipped with the help of the planar map. (If the surface is one-sided, the backfacing segments can be rejected immediately.)

The clipping process starts by locating the planar map face in which the first segment vertex lies. From there, it follows the segment edges, updating the visibility as it moves from one planar map face to another. When the visibility changes, a root-finding method is used to evaluate the point of intersection between the path and the planar map edge, and the resulting vertex is added to the path.

Several criteria must be used to determine the visibility of a stroke segment. These criteria ensure that the planar map region under consideration does indeed correspond to the portion of the surface that the stroke is intended to cover. They are, in order of application (see Figure 7-4):

- *Same surface* — the planar map face must originate from the same 3D surface that the stroke is covering (eliminates s_1).
- *Same orientation* — the planar map face must point to a 3D mesh faces that has the same orientation with respect to the view point, back-facing or front-facing, as the stroke segment (eliminates s_2).

- *Same depth* — the 3D location of the stroke segment must be close to the corresponding 3D location on the 3D mesh face (eliminates s_3). This last test is required because, with free-form surfaces, several different points on the same surface can project to the same 2D point.

If all three tests succeed, the segment is visible.

7.2.6 Rendering shadows

The polygonal planar map algorithm described in section 7.1 uses an object-space method to compute shadows before building the planar map. Unfortunately, with curved surfaces, shadow boundaries are much more difficult to generate in object space. So instead, we'll use an algorithm inspired from the two-pass z-buffer shadow algorithm proposed by Williams [43]. In the development that follows, for the sake of clarity, we'll assume that only one light source casts shadows.

The shadow algorithm starts by building a planar map with respect to the light source, called a *shadow planar map,* in addition to the *view planar map* built with respect to the view point. Assuming that the light source is located at some distance from the boundaries of the model, building the shadow planar map is in every respect identical to building a planar map with respect to the view point. Once the shadow planar map is available, clipping a shadow stroke is carried out by first clipping it against the view planar map, just like all other non-shadow strokes. In a second pass, the remaining visible segments of the stroke are clipped against the shadow planar map. However, in this pass, the portions of the stroke that are not visible from the light, and therefore in shadow, are preserved and rendered. (Note that the two planar maps lie in distinct 2D spaces. Therefore, each visible stroke segment left after the first clipping pass must be re-mapped to the shadow planar map plane before the second clipping pass can take place.)

This two-pass approach to shadow stroke clipping has the advantage of being very simple to implement; most of the required code is already available from the algorithm for building the view planar map. However, we must note that this representation of shadow regions has one important disadvantage: the shadow regions are not explicitly represented in the view planar map. Hence, the contrast between two adjacent surfaces, which affects the thickness of outline strokes, can not readily take shadows into account. As a consequence, the current implementation of the curved-surface renderer does not take shadows into account when evaluating the contrast between two adjacent surfaces.

Chapter 8

Results and Future Work

In the course of this dissertation, we've described the software architecture and algorithms for a pen-and-ink renderer. In this final chapter, we'll examine in some more detail how a "high quality" pen-and-ink image is created with the prototype curved-surface renderer. We'll also briefly consider the performance of the system. Finally, we'll summarize the contributions of the thesis work presented in this dissertation, and contemplate areas for future work.

8.1 Design methodology

The kind and quality of images that the prototype renderers are able to produce are demonstrated by many of the figures shown in Chapters 5 and 6. Achieving such images requires adjusting a relatively large number of parameters. Figure 8-1 illustrates the typical steps needed to create a good pen-and-ink image. These steps are described below:

(a) The first image shows the initial model with textures, chosen from a library of pre-designed stroke textures, already assigned. Only an ambient illumination light is set, resulting in all surfaces having a uniform tone.

(b) A light source is introduced. The position of the light source is chosen so as to create interesting shadows, as well as highlights on the objects, which will help define their shapes. In this case, a single light source fulfills these two roles adequately.

(c) The intensity of the light source and ambient light, as well as the illumination parameters associated with the different surfaces, are adjusted. They are set so as to create washed-out

highlights on the jug and bowl, as well as to provide a good contrast between highlights and dark areas.

(d) The characters and thicknesses of the different strokes are adjusted. In particular, the strokes used to hatch the bowl and the shadows are made wavier. The pressure function of the pen used to draw the jug outline is also adjusted to increase the frequency of its dashed pattern.

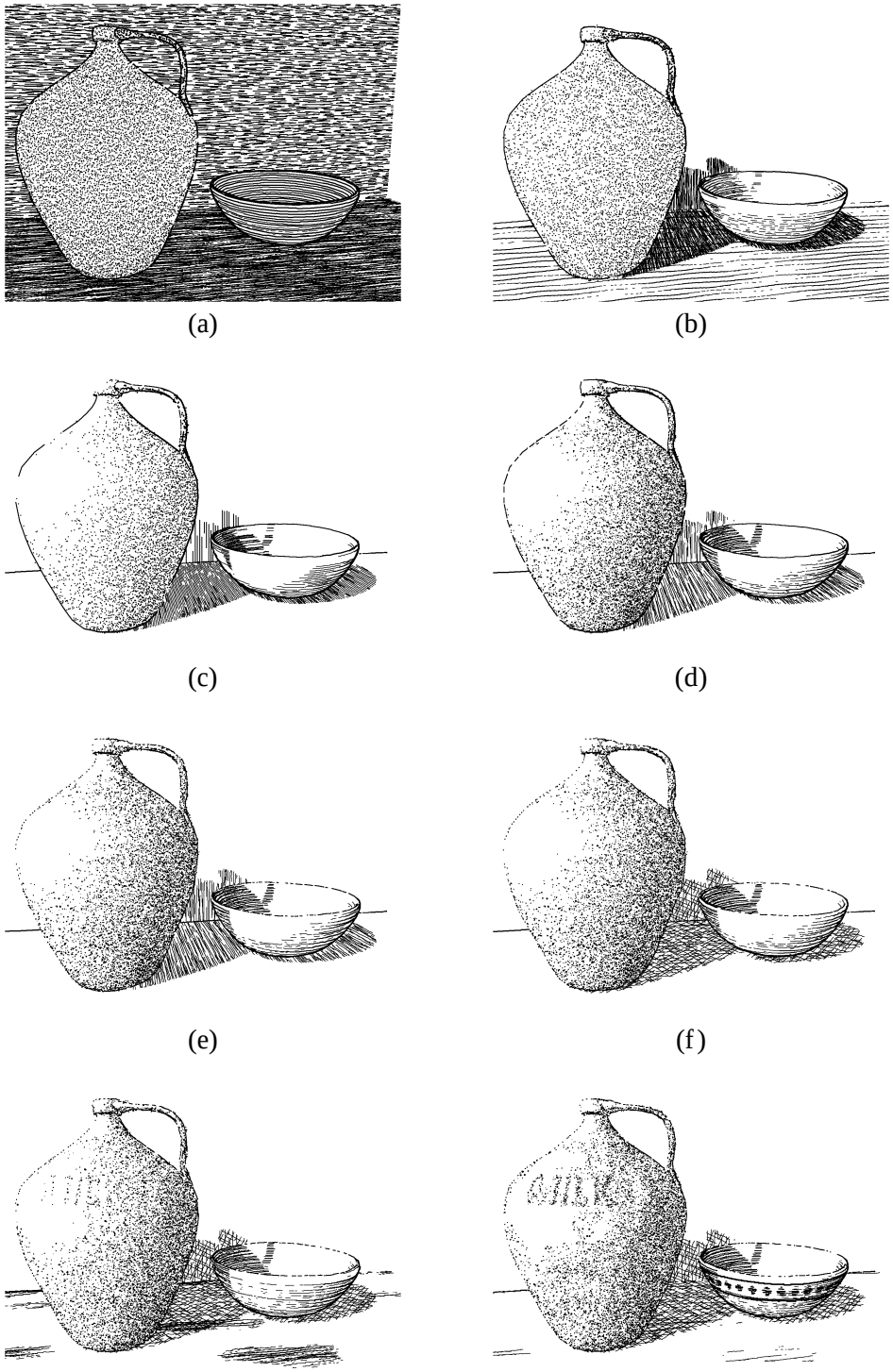
(e) The outline parameters for the pens used to draw the outlines of the jug and bowl are adjusted. For instance, the jug-outline pen is set in such a way that the stroke almost disappears along the highlight area. The pressure function for the same pen is also fine-tuned so that the stroke looks like stippling when it is light.

(f) The single-direction hatching for the shadows on the floor and background is replaced by crosshatching. In this case, crosshatching seems to work better because it creates a uniform, "direction-less" tone that does not fight for attention with the objects in the scene.

(g) Texture maps are introduced: an image map on the floor to create stains, a bump map on the jug to emboss the word "milk" and add some "wobbliness" to convey a hand-made appearance to its surface, and an image map on the bowl to add an interesting pattern.

(h) The influences of the different texture maps are adjusted. The stains on the floor are made barely visible; they're intended to make the scene less monotonous, without drawing too much attention. The bump map on the jug and the image-map on the bowl are given more weight to make their effects more prominent. This final image can also be seen at full-size in Figure 6-6 on page 46.

In practice, the design process is not so linear, and requires going back-and-forth between the different parameters to adjust them. Figure 8-2 shows a screen dump of the dialog presented to the user for designing strokes. An important part of that dialog is the feedback area at the top of the window, which shows a scaled-up version of the stroke given the current parameters. The next level of visual feedback is provided in the main window, where the model can be rendered at different scales. However, the resolution of the screen can not adequately convey the subtle variations in shading and the actual quality of the strokes that is required to fine-tune them, and, during the last few design iterations, the image must often be printed. Through time, the author has built a library



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 8-1. Design methodology. Images (a) through (h) show the typical sequence of adjustments required to create a good pen-and-ink illustration with the system described in this dissertation.

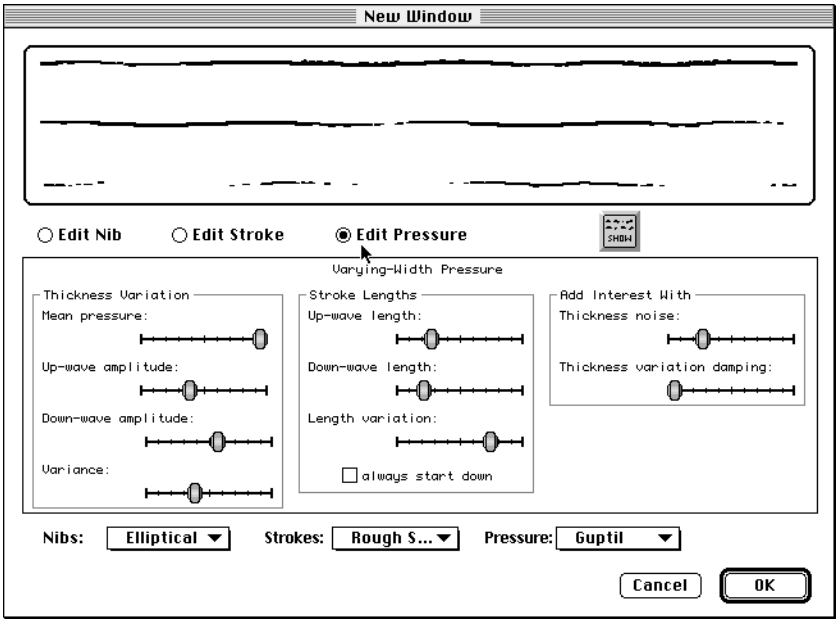


Figure 8-2. Designing strokes. The dialog window for designing strokes is divided into three main parts: a visual feeback area at the top, a parameter area in the middle (showing sliders for adjusting the pressure function parameters in this case), and popup menus to choose different nib, waviness, and pressure components at the bottom.

of pens and stroke textures that can conveniently be re-used. In many cases, these textures require only few adjustments before yielding good results.

8.2 Performance

Table 8-1 gives the rendering times for some of the images shown in Chapters 5 and 6, measured with a prototype renderer coded in C++. The total rendering time is between one and six minutes for simple to moderately complex models. The memory usage for the same models ranges between about two and five megabytes. (The polygonal renderer usually needs more memory than the curved-surface renderer because it uses three different BSP trees simultaneously.) These numbers compare favorably with those typical of a photorealistic renderer, making the results presented in this dissertation not just theoretically interesting, but also potentially useful in practice.

Table 8-1. Rendering times for various models in seconds. These timings were measured on a PowerMac 7100/80. All images were generated at the size printed in this dissertation and at a resolution of 600 dots per inch, except for Figure 5-11, which was generated at 300 dots per inch.

Model	planar map construction	rendering	total
Robie House (Figure 5-11 on page 35)	154	56	210[a]
Glass Bottle (Figure 6-4 on page 44)	74	46	120
Jug and Bowl (Figure 6-6 on page 46)	126	128	254
Ceramic Jug (Figure 6-5 on page 45)	107	179	286
Wooden Bucket (Figure 6-7 on page 47)	21	44	65
Hat and Cane (Figure 6-8 on page 48)	230	120	350

a. The numbers in this row were extrapolated from performance measurements taken with a Quadra 700, and from the typical speedup observed when moving code from the Quadra 700 to a PowerMac 7100 computer. (The polygonal renderer implementation is not compatible with the PowerMac.)

8.3 Contributions

The main contributions of this dissertation are:

Software architecture for pen-and-ink rendering. Although the software architecture introduced in Chapter 3 is described in the context of pen-and-ink rendering, many of the principles on which it is based are applicable to other illustration media. For instance, outlining is relevant to almost all styles of illustration. Therefore, the software architecture proposed in this dissertation should find other uses.

Prioritized stroke textures. The concept of prioritized stroke texture introduced in Chapter 5 provides a framework for creating textures from strokes. It also provides a methodology to build procedural versions of these textures. One important advantage of this framework is that it takes the resolution of the target device and the size of the image into account.

The same concept of prioritized stroke texture was used in the interactive pen-and-ink illustration system designed by Salisbury et al. [36]. In their system, the textures are explicitly stored as a collection of strokes with assigned priorities. As the user “brushes” an area with a chosen texture, strokes are introduced in priority order until the target tone is achieved.

Stroke textures on parametric surfaces. The mathematical framework and algorithms presented in Chapter 6 allow stroke textures to be mapped onto parametric free-form surfaces. They extend considerably the range of models that can be rendered in pen-and-ink. Furthermore, by permitting the use of traditional, image-based, texture mapping techniques, it also extends the ranges of effects that can be achieved with stroke textures.

Planar map algorithms. No fundamentally new algorithm or data structure for building planar maps were introduced. However, the two algorithms described in Chapter 7 show how to build planar maps from 3D geometric data. Furthermore, the use of these planar maps for stroke clipping and outlining is also described. Because the use of the planar map is based on basic illustration principles, these algorithms should find uses elsewhere.

8.4 Future work

This dissertation also leaves several directions open for research. The focus of this work has been on the rendering problem. Almost no attention has been paid to the problem of composing an effective illustration. Selligman and Feiner [37], and Dooley and Cohen [9] have addressed those issues in their work. But much remains to be done to design a truly effective and flexible illustration system. For instance, it would be nice to incorporate other illustration effects, such as exploded views, cut-aways, and peel-back views. Ideally, these effects should be accessible through high-level interactive tools that give the user enough flexibility to compose a good illustration. Very high-level controls to adjust parameters such as emphasis, indication, or lighting would also be useful. For example, an “emphasis” tool would work by automatically accentuating or suppressing details over different parts of the image.

This work started as a study of the visual abstractions that illustration can offer to the rendering of complex shapes. The stroke textures provide some form of visual abstraction. But much work remains to be done in that direction as well. For instance, being able to render natural forms such as trees, grass, water, and human figures, for which established conventions also exist, would greatly enhance the capability of pen-and-ink rendering.

Even within the confines of the pen-and-ink rendering problem, there is much room for improvement. For instance, it would be nice to find ways to automate and improve the creation of stroke

textures, perhaps by using an approach similar to the RenderMan shading language [31]. In this vein, the works of Yessios [45] and Miyata [27] provide a good starting point.

One of the most fundamental limitations of the ~~curved surface renderer~~ is that it only handles geometries with a global parameterization, such as B-splines, NURBS, or surfaces of revolutions. Many commonly used free-form representations, such as patch-based Bézier surfaces, or smoothly shaded polygonal meshes, do not meet this requirement. Therefore, generalizing the free-form hatching algorithm to handle a broader class of curved surfaces is very desirable. There are several avenues worth exploring to fulfill this goal. For instance, recent work due to Maillot et al. [26] and Pedersen [30] describe methods to parameterize geometries that do not already possess an intrinsic parameterization. Another approach is to use information that is more intrinsic to the surface geometry, such as drawing strokes along directions of principal curvature [4], and to do away with any kind of parameterization. Maillot et al. showed how surface curvature can be approximated from a polygonal mesh representation [26], suggesting that such an approach would also work for this type of geometry.

Finally, we conclude this dissertation by bringing the reader's attention to the many other illustration media, besides pen-and-ink, such as watercolor, pencil drawing, air brushing, etc. All illustration styles tend to share a few common basic principles; but each medium also has its own set of methods and "tricks". Devising algorithms and techniques to handle these styles of illustration presents an interesting, albeit formidable, challenge. Finally, with fundamental understanding of illustration principles, we might even consider computer-generated imagery as a style of its own, and find new effective ways of conveying visual information that do not necessarily mimic traditional illustration forms.

Bibliography

1. Adobe. *Adobe Dimensions.* Adobe Systems Incorporated, Mountain View, 1992.
2. Arthur Appel, F. James Rohlf and Arthur J. Stein. The haloed line effect for hidden line elimination. *Computer Graphics 13*, 2 (August 1979), 151–157.
3. Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *Computer Graphics 26*, 2 (1992), 35–42.
4. Manfredo P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.
5. Fuhua Cheng. Estimating subdivision depths for rational curves and surfaces. *ACM Transaction on Graphics 11*, 2 (1992), 140–151.
6. Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. *Computer Graphics 23*, 3 (1989), 99–106.
7. Frank Ching. *Architectural Graphics*. Van Nostrand Reinhold Company, New York, 1975.
8. Debra Dooley and Michael F. Cohen. Automatic illustration of 3D geometric models: Lines. *Computer Graphics 24*, 2 (March 1990), 77–82.
9. Debra Dooley and Michael F. Cohen. Automatic Illustration of 3D Geometric Models: Surfaces. Proceedings of Visualization '90, October, 1990, 307–314.
10. Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transaction on Visualization and Computer Graphics 1*, 3 (September 1995), 231–239.
11. Gershon Elber and Elaine Cohen. Hidden curve removal for free form surfaces. *Computer Graphics 24*, 4 (August 1990), 95–104.
12. Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 1990.
13. James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 1990.
14. H. Fuchs, Z. M. Kedem and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics 14*, 3 (July 1980), 124–133.
15. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet and Jean-Manuel Van Thong. Incremental computation of planar maps. *Computer Graphics 23*, 3 (July 1989), 345–354.

16. Leonidas Guibas, Lyle Ramshaw and Jorg Stolfi. A Kinetic Framework for Computational Geometry. Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science, 1983, 100–111.

17. Leonidas Guibas, David Salesin and Jorge Stolfi. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. Proceedings of the 5th Annual Symposium on Computational Geometry, Saarbruchen, West Germany, 1989, 208–217.

18. Arthur Leighton Guptill. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York, 1976.

19. Christoph Hoffman. The problems of accuracy and robustness in geometric computation. *Computer 22* (1989), 31–42.

20. Tomihisa Kamada and Saturo Kawai. An enhanced treatment of hidden lines. *ACM Transaction on Graphics 6*, 4 (October 1987), 308–323.

21. Stephen Klitment. *Architectural Sketching and Rendering: Techniques for Designers and Artists*. Whitney Library of Design, New York, 1984.

22. John Lansdown and Simon Schofield. Expressive rendering: a review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications 15*, 3 (May 1995), 29–37.

23. Paul Laseau. *Architectural Drawing: Options for Design*. Design Press, New York, 1991.

24. Wolfgang Leister. Computer generated copper plates. *Computer Graphics Forum 13*, 1 (1994), 69–77.

25. Frank Lohan. *Pen and Ink Techniques*. Contemporary Books, Inc., Chicago, 1978.

26. Jérôme Maillot, Hussein Yahia and Anne Verroust. Interactive texture mapping. *Computer Graphics Proceedings, Annual Conference Series* (August 1993), 27–34.

27. Kazunori Miyata. A method of generating stone wall patterns. *Computer Graphics 24*, 4 (August 1990), 387–394.

28. Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1988.

29. Bruce Naylor and Lois Rogers. Constructing Partitioning Trees from Bézier-Curves for Efficient Intersection and Visibility. Proceedings of Graphics Interface '95, 1995, 44–55.

30. Hans Køhling Pedersen. Decorating implicit surfaces. *Computer Graphics Proceedings, Annual Conference Series* (August 1995), 287–290.

31. Pixar. *The RenderMan Interface: Version 3.1*. Pixar, San Rafael, California, 1989.

32. Tom Porter and Sue Goodman. *Manual of Graphic Techniques 4*. Charles Scribner's Sons, New York, 1985.

33. Premisys. *Squiggle*. The Premisys Corporation, Chicago, 1993.

34. Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3D shapes. *Computer Graphics 24*, 4 (August 1990), 197–206.

35. David H. Salesin. *Epsilon Geometry: Building Robust Algorithms from Imprecise Computations.* Ph.D. thesis, Stanford University, March 1991. Available as Stanford Report number STAN-CS-91-1398.

36. Michael P. Salisbury, Sean E. Anderson, Ronen Barzel and David H. Salesin. Interactive pen-and-ink illustration. *In Computer Graphics, Annual Conference Series* (July 1994), 101–108.

37. Dorée Duncan Seligmann and Steven Feiner. Automated generation of intent-based 3D illustration. *Computer Graphics 25*, 4 (July 1991), 123–132.

38. Reggie Stanton. *Drawing & Painting Buildings*. North Light Publishers, Wesport, Connecticut, 1978.

39. Thomas Strothotte, Bernhard Preim, Andreas Raab, Jytta Shumann and David R. Forsey. How to Render Frames and Influence People. Proceedings of Eurographics 94, September, 1994.

40. Angus E. Taylor and W. Robert Mann. *Advanced Calculus*. John Wiley & Sons, Inc., New York, 1983.

41. William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics 21*, 4 (July 1987), 153–162.

42. K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics 11*, 3 (July 1977), 214–222.

43. Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics 12*, 3 (August 1978), 270–274.

44. Phyllis Wood. *Scientific Illustration*. Van Nostrand Reinhold, New York, 1994.

45. Chris I. Yessios. Computer drafting of stones, wood, plant and ground material. *Computer Graphics 13*, 2 (August 1979), 190–198.

Vita

Georges Winkenbach

M.S., Computer Science, University of Washington, Seattle, Washington, June 1991.

B.S., Computer Science, University of Washington, Seattle, Washington, December 1987.

Diploma of Engineer in Electronics, Ecole d'Ingénieurs du Canton de Neuchâtel, Le Locle, Switzerland, March 1983.