Interactive Pen-and-Ink Illustration

Michael P. Salisbury *Sean E. Anderson* *Ronen Barzel*[*] *David H. Salesin*

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

[*]Pixar
1001 West Cutting Blvd
Richmond, California 94804

Abstract

We present an interactive system for creating pen-and-ink illustrations. The system uses *stroke textures*—collections of strokes arranged in different patterns—to generate texture and tone. The user "paints" with a desired stroke texture to achieve a desired tone, and the computer draws all of the individual strokes.

The system includes support for using scanned or rendered images for reference to provide the user with guides for outline and tone. By following these guides closely, the illustration system can be used for interactive digital halftoning, in which stroke textures are applied to convey details that would otherwise be lost in this black-and-white medium.

By removing the burden of placing individual strokes from the user, the illustration system makes it possible to create fine stroke work with a purely mouse-based interface. Thus, this approach holds promise for bringing high-quality black-and-white illustration to the world of personal computing and desktop publishing.

CR Categories and Subject Descriptors: I.3.2 [Computer Graphics]: Picture/Image Generation - Display algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques - Interaction techniques; I.4.3 [Image Processing]: Enhancement.

Additional Key Words: Comprehensible rendering, non-photorealistic rendering, prioritized stroke textures.

1 Introduction

Pen-and-ink is an extremely limited medium, allowing only individual monochromatic strokes of the pen. However, despite the limitations of the medium, beautiful pen-and-ink illustrations incorporating a wealth of textures, tones, and styles can be created by skilled artists. Indeed, partly because of their simplicity and economy, pen-and-ink illustrations are widely used in textbooks, repair manuals, advertising, and many other forms of printed media.

Part of the appeal and utility of pen-and-ink illustrations is that they can be easily printed alongside text, using the same ink on the same paper, without any degradation. For the same reasons, pen-and-ink-style illustrations could also be useful in the domain of desktop publishing and laser printers—especially if the illustrations were generated and manipulated directly on a computer.

While the problem of painting full-color images on a computer has received considerable attention in the computer graphics community, the requirements of an interactive pen-and-ink-style illustration system are different enough to merit special study. Pen-and-ink illustrations have two major properties that distinguish them from other art media:

1. *Every stroke contributes both tone (darkness) and texture.* Since tone and texture are not independent parameters, the pen artist must take care to convey both of these qualities simultaneously.

2. *Strokes work collectively.* In general, no single stroke is of critical importance; instead, strokes work together to express tone and texture.

This paper describes an interactive pen-and-ink-style illustration system. The overall goal of the system is to enable a user to easily generate effective and attractive illustrations directly on a computer. In this work, we are not concerned with creating purely computer-generated images; rather, the computer is utilized as a tool to enhance the speed and ease with which a user can create illustrations.

The interactive illustration system allows a variety of texturing in order to achieve the same range of style and expressive ability that is possible with a physical pen and ink. We do not want to limit the user to any specific algorithmic "look."

The system places a particular emphasis on using continuous-tone images as a reference for the user, and thus provides a form of "interactive digital halftoning" in which the user can introduce texture as an integral part of the resulting illustration. In this sense, the visual artifacts that are necessarily produced in quantizing a greyscale image can be given an artistic or expressive nature. Also, of practical significance, photocopying does not degrade pen-and-ink-style images to the same extent as conventionally-halftoned images.

1.1 Background: Pen-and-ink illustration

We give here a brief description of some of the salient features and terminology of hand-drawn pen illustration, relevant to the design of an interactive system. For further discussion and instruction, interested readers should consult Guptill [6], a comprehensive text on pen and ink illustration. In addition, Simmons [16] provides instruction on illustrating using a "technical pen," which draws strokes of constant width. Both books contain dozens of stunning examples. A discussion of pen-and-ink principles as they relate to purely computer-generated imagery can be found in Winkenbach et al. [20].

Because texture in an illustration is the collective result of many pen strokes, each individual stroke is not critical and need not be drawn precisely. Indeed, a certain amount of irregularity in each stroke is desirable to keep the resulting texture from appearing too rigid or mechanical.

The most commonly used textures include: *hatching*, formed by roughly parallel lines; *cross-hatching*, formed by overlapped hatching in several directions; and *stippling*, formed by small dots or very short lines. Textures can also be wavy, scribbly, or geometric and can appear hard or soft, mechanical or organic.

The perceived grey level or *tone* in an illustration depends largely on how dense the strokes are in a region (just like the dots in a dithered halftone image). Although grey-level ramps can be achieved by judiciously increasing stroke density, fine-art illustrations typically emphasize contrast between adjacent regions, and often employ a very limited number of distinct grey levels.

Shapes in an illustration can be defined by *outline* strokes. These strokes are exceptional in that they may be long and individually significant. Often the outline is left implicit by a change in tone or texture. The choice of whether or not to use outlines is largely an aesthetic one, made by the artist, and used to achieve a particular effect. For example, explicit outlines are typically used for hard surfaces, while implied outlines generally convey a softer or more organic object.

Producing fine-art-quality, hand-drawn pen-and-ink illustrations requires a great deal of creativity and artistic ability. In addition, it requires a great deal of technical skill and patience. A real pen and ink have no undo!

1.2 Related work

Most of the published work on "digital painting" is concerned with the problem of emulating traditional artists' tools. Only a few of these works take an approach similar to ours of creating higher-level interactive tools that can produce the same results as their predecessors: Lewis [10] describes brushes that lay down textured paint; Haeberli [7] shows how scanned or rendered image information can be used as a starting point for "painting by numbers;" and Haeberli and Segal [8] use hardware texture-mapping for painting and also mention 3D halftoning effects.

Considerable work has also been done for creating black-and-white illustrations, generally for engineering or graphical design work. The earliest such system was Sutherland's "Sketchpad" [18]. Gangnet et al. [5] use planar decomposition to manipulate and clip geometric objects. Pavlidis [11] provides a method for "cleaning up" schematic drawings by removing hand-drawn irregularities. Quite the opposite (and more along the lines of our work), the Premisys Corporation markets a commercial product, "Squiggle," [13] that adds waviness and irregularities to CAD output to augment lines with extra information and to make the results appear more hand-drawn. Saito and Takahashi [14] produce automated black-and-white illustrations of 3D objects.

Our research group is exploring several different aspects of the pen-and-ink illustration problem. This paper discusses the issues of interactively creating pen-and-ink illustrations, with an emphasis on using 2D greyscale images as a starting point. A second paper shows how principles of illustration can be incorporated into an automated system for rendering 3D models [20]. A third paper examines the issues involved in representing, editing, and rendering the individual strokes that are the building blocks of any line illustration system [4].

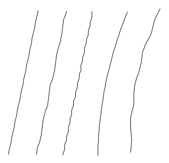

Figure 1: *A closeup view of several individual pen strokes, with various amounts of curve and waviness.*

1.3 Overview

The next section discusses the overall design of our system, as well as its individual capabilities and features. Section 3 presents some example illustrations and describes our experience with using the system. Section 4 suggests directions for future research. The primary data structures and algorithms of our prototype implementation are outlined in appendix A.

2 The Illustration System

Full-color paint systems often support direct simulations of traditional artist tools, such as brushes and paint [3, 17]. However, for our application, there is little purpose in providing the user with a simulated "ink pen" to draw the pen strokes, for several reasons:

- A mouse-based interface does not support the fine control needed for detailed stroke work.
- The strokes of an illustration are not of great individual importance.
- Drawing individual strokes is tedious, and we would like our system to reduce much of that tedium.

Thus, rather than focus on the individual strokes, the system tries to directly support the higher-level cumulative effect that the strokes can achieve: texture, tone, and shape. The user "paints" using textures and tones, and the computer draws the individual strokes.

The illustration system cannot completely ignore individual strokes, however. Outlines are the most notable example of strokes that have individual significance; in addition, an artist might occasionally need to touch up fine details of textured work. Therefore, the system also allows users to draw individual strokes and provides controls for modifying stroke character through smoothing and through the substitution of various stroke styles [4].

To further aid users in creating illustrations, the system allows scanned, rendered, or painted images to be used as a reference for tone and shape. The system also supports edge extraction from images, which is useful for outlining. Finally, a range of editing capabilities is supported so that users are free to experiment or make mistakes.

The following sections discuss the capabilities and workings of the system in greater detail.

2.1 Strokes

It is important that the strokes automatically generated by the system be irregular. Uneven strokes make an illustration look softer, more natural, and hand-drawn, whereas regular strokes introduce mechanical-looking texture. The use of irregular strokes can be compared to the introduction of randomness in image dithering [19].

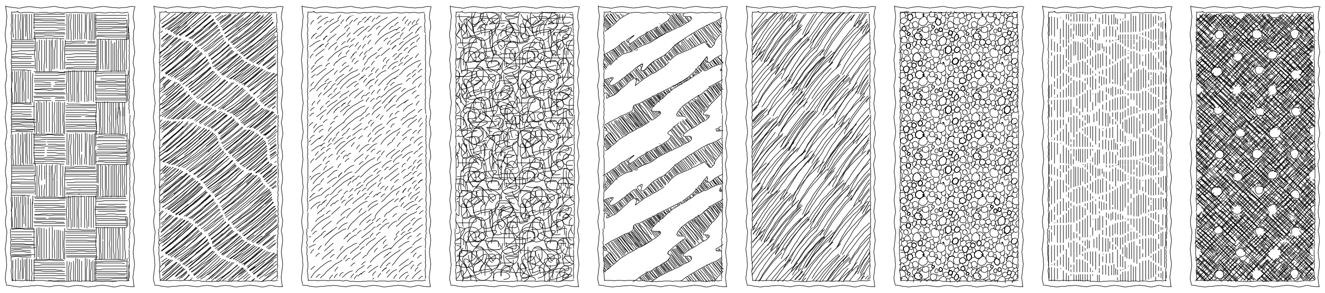
Figure 2: *Assorted stored stroke textures.*

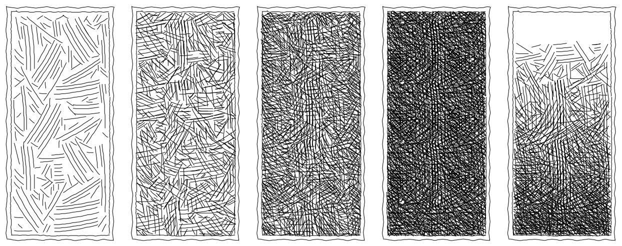
Figure 3: *A single texture drawn with several tone values.*

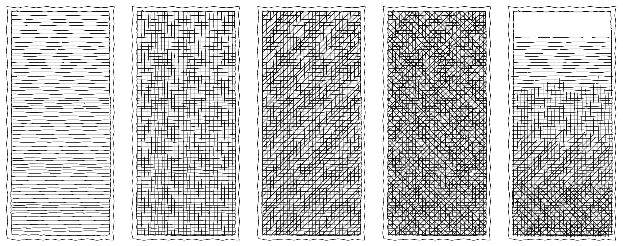
Figure 4: *A prioritized texture. Only the most significant strokes are drawn for light tone values; less important strokes are brought in to darken the texture.*

We cannot simply draw strokes in completely random directions, however—the stroke direction is one of the key elements in defining a texture. Instead, the system perturbs the strokes in a variety of small ways (see Figure 1): strokes can be drawn with a slight wiggle (a wave with slightly randomized frequency and phase); straight strokes can be given a slight overall curvature; and stroke length and direction can be jiggled slightly. Section A.3 describes the stroke-drawing algorithm in greater detail. Currently, strokes in our system are each of constant width, as per a "technical pen" [16].

2.2 Textures

The user paints by rubbing a "brush" over the illustration; the strokes that appear in the region under the brush are generated based on a user-selected *stroke texture* (see Figure 2). The system supports a library of user-defined *stored* stroke textures, as well as several built-in *procedural* stroke textures. In this way, a wide variety of illustration styles can be achieved. These two types of stroke textures are described in more detail below.

Stored stroke textures

A stored texture is simply a collection of strokes. Drawing a texture at a given darkness is a matter of choosing from the collection a subset that has enough strokes to reach the desired tone. (Some textures may be inherently too light—they may not have enough strokes to make dark tones.)

For textures such as stipples and scribbles, the choice of strokes to draw for a given tonality is not critical. In these cases, the system simply selects strokes from the texture in a random sequence, generating candidate strokes and testing the tonal effect of candidate strokes as described in Section A.3. Candidate strokes that pass the tests are drawn, and those that fail are discarded (see Figure 3).

For other textures, however, the system supports a predefined *priority* for each stroke, which specifies an order to use in generating and testing candidate strokes. For example, Figure 4 illustrates a texture in which only horizontal hatches are drawn for light tones, while cross-hatching strokes are used for darker tones. Another example would be a texture in which larger scribbles are drawn before smaller ones.

Creating a good prioritized stroke texture is not always easy—some design iteration may be required before the strokes and their priorities work well together. Once a texture has been created and perfected, however, it can be archived for repeated use. The system lets the user draw textures interactively and can also support textures that are computed programmatically or that are taken from edges extracted from scanned images.

Procedural stroke textures

Many interesting texture effects can be computed procedurally. The system currently supports three types of procedural texturing: stippling (randomly distributed points or short strokes), parallel hatching, and curved strokes. The latter two textures can follow along or against the gradient of a reference image. Since these are the only textures truly built into the system, they are the basic building blocks from which user-drawn stored textures are formed.

To draw procedural stroke textures, the system simply generates appropriate candidate strokes under the region of the brush and tests them, as discussed in detail in Section A.3. More intricate prioritized procedural stroke textures, such as "brick," "wood," or "shingle" textures, can also be defined [20], although they are not currently implemented in our interactive system.

2.3 Reference images

A scanned, rendered, or digitally painted continuous-tone image can be underlaid "beneath" the illustration being drawn, and displayed faintly. This reference image can be used in several ways (see Figure 5):

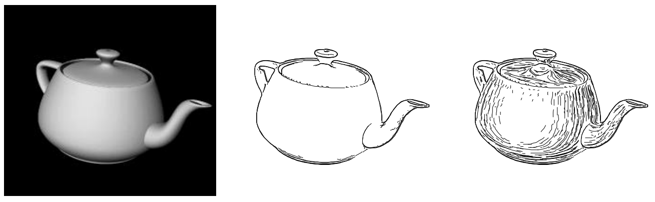

Figure 5: *Using a grey scale image for reference. Left to right: Original grey scale image; extracted edges; curved hatching across the gradient.*

- As a visual reference for the artist.
- As a tone reference for painting, in which case the texture darkness will match that of the image.
- As a source image from which edges are extracted to use for outlining and clipping. The user can select edges corresponding to versions of the image at various resolutions.
- As a progenitor of *stencils*. The user can interactively define stencils by specifying ranges of intensities in the reference image; strokes are drawn only where the reference image value is within the specified ranges.
- As a reference for determining stroke and texture orientation. Textures that follow the reference gradient can be particularly useful for conveying curved surfaces.

Note that its extensive support for reference images makes the illustration system a particularly effective tool for interactive digital halftoning. However, it does not provide automatic halftoning—it is up to the user to choose which stroke textures to apply, where to apply them, and how dark to make them, based on the user's intent and aesthetic sense for the final illustration. One could imagine an automated system to extract texture from an image, but there is not always enough information in the image to achieve the desired effect. For example, the original reference photograph for the goose in Figure 9 does not show feathers in any great detail; the artist must choose textures and introduce tone variation to convey the sense of feathering.

2.4 Detail manipulation

The illustration system supports multiresolution curves [4], allowing users to add or remove detail from strokes and edges. For example, an illustration can be initially made using smooth strokes, which can later be adjusted in subtle or not so subtle ways, using a variety of wiggly or scribbly detail. Alternatively, detail can be removed from an edge extracted from the tone reference in order to yield smoother outlines (see Figure 6).

2.5 Clipping

The user can specify outlines, which may or may not be drawn in the final illustration, but against which strokes (and stroke textures) are clipped. Outlines can be drawn by hand or can be taken from edges in reference images.

Just as individual strokes should not be too regular, the clipped ends of textures should in general be slightly ragged. The system introduces a small amount of random variation by clipping strokes too soon or allowing them to spill beyond the edge of the clipping region (see Figure 7).

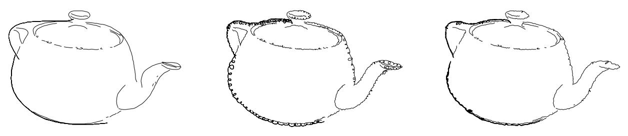

Figure 6: *Manipulating curve detail. Left to right: Teapot edges from Figure 5, with detail removed; alternate details applied to the curves.*

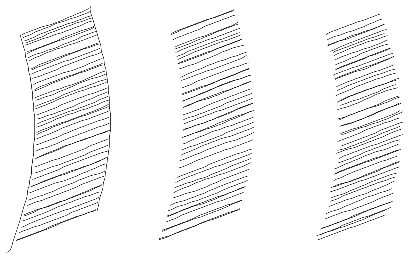

Figure 7: *Strokes clipped to an outline. Left: The outline is drawn. Center: The outline has been removed; notice the hard edge caused by exact clipping. Right: A small amount of random sloppiness creates a softer edge.*

2.6 Individual strokes

Sometimes individual strokes are important enough to be drawn by hand; for example, the hairs in Figure 11 were individually created. The user can draw individual strokes with a mouse or with a tablet. These strokes can be given waviness and clipped in the same manner as automatically-generated strokes. To overcome the mouse's lack of smoothness, unwanted detail can be removed via the multiresolution curve mechanism, or prestored "artistic" irregularities can be introduced, as described in Section 2.4.

2.7 Editing collections of strokes

In addition to modifying individual strokes, the user can edit collections of strokes. Editing operations can be applied to all strokes, to those generated from a given texture, or to strokes selected interactively.

Perhaps the most interesting editing operation is the "lighten" operation. Rather than simply erasing all strokes under the brush, "lighten" incrementally removes strokes. Thus, a textured region that is too dark can be made lighter without destroying the integrity of the texture, instilling pen-and-ink with qualities of a subtractive medium. For example, in the lower left-hand drawing of Figure 8, the mottled effect in the background was created by painting a cross-hatch texture to a uniform darkness, then slightly lightening in a few places with touches of the brush.

3 Results

The pen-and-ink illustration system is implemented in C++ and runs at interactive speed on an SGI Indigo2 workstation, without any additional hardware assistance. The system has proven quite successful at assisting users in easily producing a variety of illustrations. All figures in this paper were drawn using the illustration system; only a few minutes were required for the simplest figures, and a few hours were required for the goose in Figure 9. All figures were out-

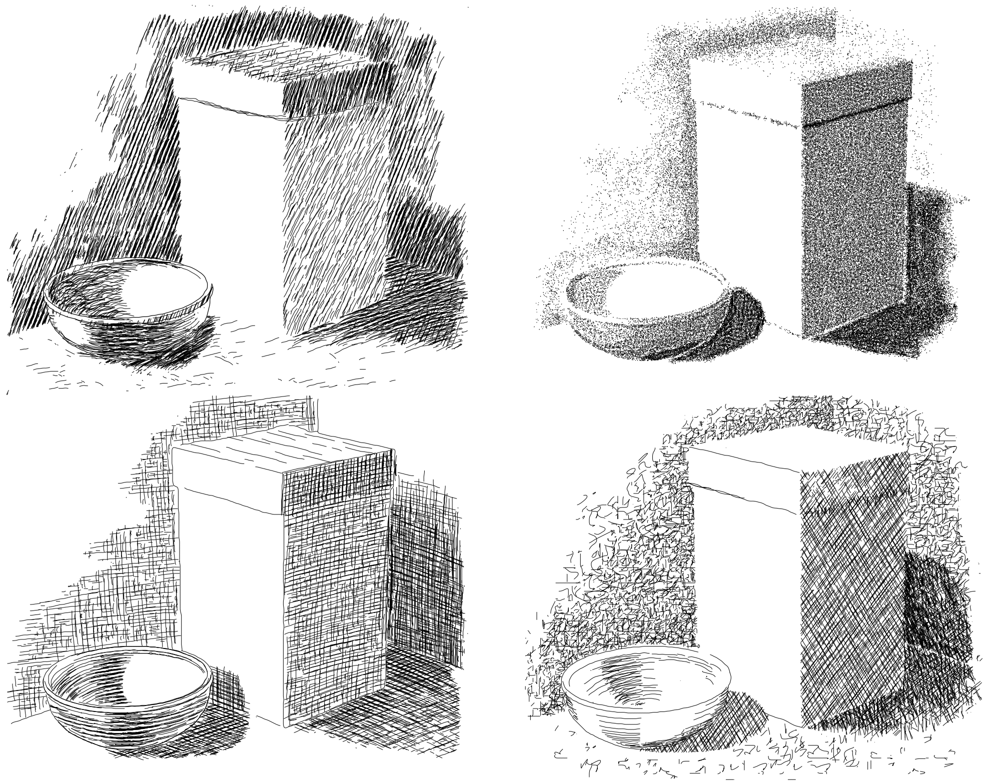

Figure 8: *A single scene, drawn in a variety of styles. Pitz [12] suggests drawing this scene with varying styles, as an exercise for student illustrators. The three drawings on top and left are attempts to closely follow examples given in the book, while the lower right is our own stylistic expression. The illustrations were created using an image of a simple 3D model as a tone reference.*

put in PostScript by our system and printed with the text on a GCC SelectPress 1200dpi printer.

To test the range and quality of the system, we chose to tackle exercises and mimic drawings from illustration texts. Figures 8, 9 and 10 show some of the results. We must admit that the target pen-and-ink drawings in the textbooks are generally finer than ours. However, when we consider that our illustrations were made on a bitmap display, using only a mouse, by programmers who are not trained illustrators, and in a matter of minutes for the simpler drawings, we find our results very encouraging.

4 Future work

The illustration system we have built suggests a number of areas for future research:

- *Experimenting with better interaction techniques.* The control panel of our prototype system has a button or slider for nearly every low-level operation and parameter in the program and hence is somewhat cumbersome to use. A better interface would provide support for common illustrator techniques, such as haloed outlines and stippled edges. In addition, we would like to explore adding much higher-level controls for producing illustrations, including commands to "increase contrast" or "focus attention" on certain regions of the illustration.

- *More sophisticated strokes and stroke textures.* Our simple procedural and stored textures do not yet provide all of the subtlety and variety available to the pen. For example, we would like to include the ability to vary the thickness along a stroke, which is supported in other pen-and-ink work [4, 20].

- *Resolution-independence.* The user should be able to work at a convenient screen resolution, while the final output should have strokes drawn with the highest resolution the printer can support. However, changing resolution in a naive fashion may change the appearance of strokes and stroke textures in undesirable ways. We would like to explore methods of storing illustrations not as collections of strokes, but as higher-level descriptions of tone, texture, and clipping information that could be used to generate the image appropriately at any arbitrary resolution.

- *Combining with 3D.* We would like to interface our interactive system with an automatic renderer for creating pen-and-ink illustrations from 3D models [20] to create an integrated interactive 2D and 3D illustration system.

Acknowledgements

We would like to thank Adam Finkelstein for helping us incorporate his stroke detail research into our editor, and Georges Winkenbach and Tony DeRose for their useful discussion of illustration principles. We would also like to thank SGI for loaning us several machines in order to meet our deadlines.

This work was supported by an NSF National Young Investigator award (CCR-9357790), by the University of Washington Graduate Research and Royalty Research Funds (75-1721 and 65-9731), and by industrial gifts from Adobe, Aldus, and Xerox.

References

[1] Brian Cabral and Leith (Casey) Leedom. Imaging Vector Fields Using Line Integral Convolution. Proceedings of SIGGRAPH 93 (Anaheim,

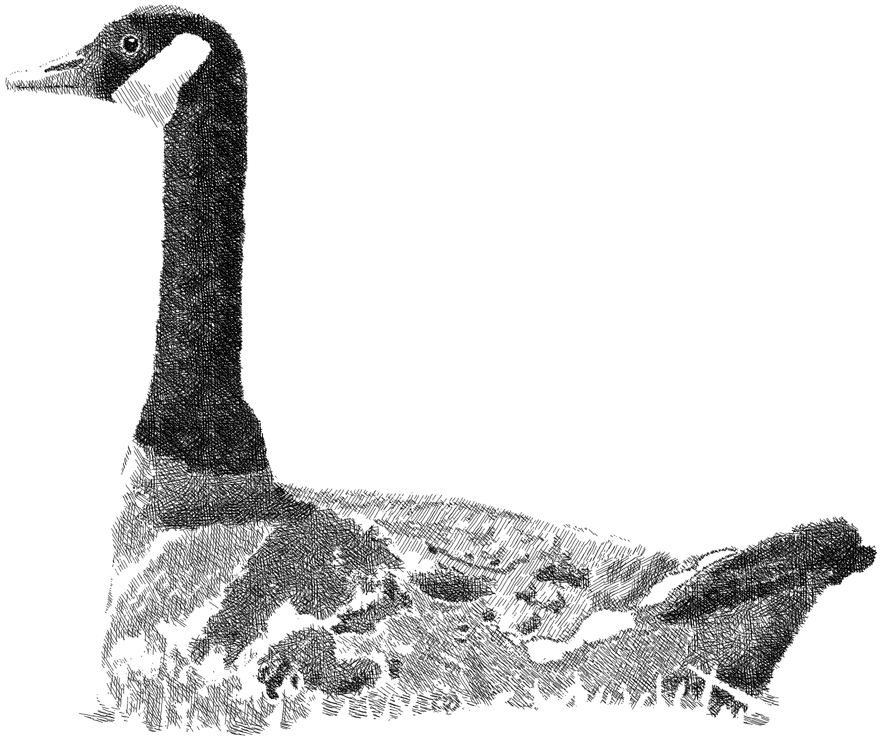

Figure 9: *A Canada goose. Simmons [16] uses the goose as a worked example of tight technical pen drawing from a reference photograph; we attempted to mimic the author's final result with our system, using only a mouse. The book includes a reproduction of the author's own reference photograph, which we scanned to use as a tone reference.*

California, August 1–6, 1993). In *Computer Graphics*, Annual Conference Series, 1993, pages 263–272.

[2] John Canny. A Computational Approach To Edge Detection. In Rangachar Kasturi and Ramesh C. Jain, editors, *Computer Vision: Principles*, pages 112–131. IEEE Computer Society Press, Los Alamitos, California, 1991. Reprinted from *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[3] Tunde Cockshott, John Patterson, and David England. Modelling the Texture of Paint. In *Proceedings of EUROGRAPHICS '92*, pages C–217 to C–226, September 1992.

[4] Adam Finkelstein and David H. Salesin. Multiresolution Curves. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

[5] Michel Gangnet, Jean-Claude Herve, Thierry Pudet, and Jean-Manuel Van Thong. Incremental Computation of Planar Maps. Proceedings of SIGGRAPH '89 (Boston, Massachusetts, July 31–August 4, 1989). In *Computer Graphics* 23, 3 (August 1989), pages 345–354.

[6] Arthur L. Guptill. *Rendering in Pen and Ink.* Watson-Guptill Publications, New York, 1976.

[7] Paul Haeberli. Paint by Numbers: Abstract Image Representations. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6–10, 1990). In *Computer Graphics* 24, 4 (August 1990), pages 207–214.

[8] Paul Haeberli and Mark Segal. Texture Mapping as a Fundamental Drawing Primitive. In *Proceedings of the Fourth Annual EUROGRAPHICS Workshop on Rendering*, pages 259–266, Paris, June 1993. Ecole Normale Superieure.

[9] Douglas Kirkland. *Icons*. Collins Publishers San Francisco, San Francisco, California, 1993.

[10] John-Peter Lewis. Texture Synthesis for Digital Painting. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 23–27, 1984). In *Computer Graphics* 18, 3 (July 1984), pages 245–252.

[11] Theo Pavlidis. An Automatic Beautifier for Drawings and Illustrations. Proceedings of SIGGRAPH '85 (San Francisco, California, July 22–26, 1985). In *Computer Graphics* 19, 3 (July 1985), pages 225–230.

[12] Henry C. Pitz. *Ink Drawing Techniques.* Watson-Guptill Publications, New York, 1957.

[13] The Premisys Corporation, Chicago. *Squiggle*, 1993.

[14] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6–10, 1990). In *Computer Graphics* 24, 4 (August 1990), pages 197–206.

[15] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[16] Gary Simmons. *The Technical Pen.* Watson-Guptill Publications, New York, 1992.

[17] Steve Strassman. Hairy Brushes. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18–22, 1986). In *Computer Graphics* 20, 4 (August 1986), pages 225–232.

[18] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphics Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, 1963.

[19] Robert Ulichney. *Digital Halftoning*. The MIT Press, Cambridge, 1987.

[20] Georges Winkenbach and David H. Salesin. Computer-Generated Pen-and-Ink Illustration. Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics*, Annual Conference Series, 1994.

A Implementation details

This appendix outlines the implementation of our prototype pen-and-ink illustration system. We will focus on the most significant features: the data structures allowing quick updating and editing of the illustration, and the stroke generation and testing algorithms.

Section A.1 describes the data types used in the system. Section A.2 presents the global data items maintained. The process of generating and using strokes is discussed in Section A.3.

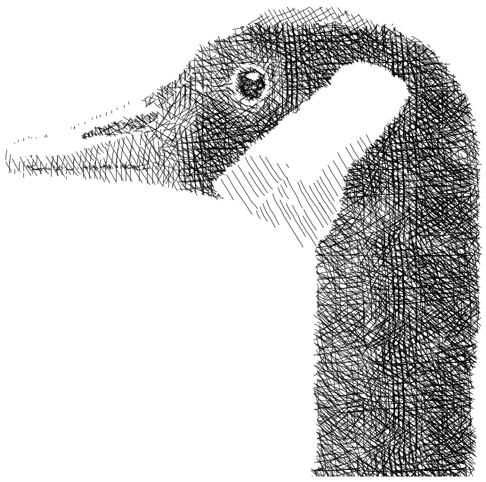

Figure 10: *Close-up of the goose head.*

A.1 Data Types

The two basic data structures used by the illustration system are the *stroke* and the *stroke database*.

Stroke

The data type at the heart of the system is the $Stroke$. Each stroke includes the following fields:

- $pixels$: An arbitrary-size array of *(x,y)* pixel coordinate pairs.
- $length$: The size of the $pixels$ array.
- $width$: The width of the stroke, in pixels.
- $bbox$: The rectangular bounding box of the stroke's pixels.
- id: The texture from which the stroke was derived.
- $priority$: The ranking of a stroke, if in a prioritized texture.

The entries of the $pixels$ array contiguously trace the path of the stroke: *x* and *y* never change by more than ± 1 from one entry to the next.

The operations supported by the $Stroke$ type include: testing to see if a stroke intersects a given rectangular region, circular region, or other stroke; decreasing $length$ by trimming entries off the ends of $pixels$; merging two contiguous strokes into a single stroke; and returning the point in the stroke that is closest to a given pixel.

A stroke can be manipulated as a multiresolution curve [4]. Each entry in $pixels$ is used as a control point of an interpolating spline, which is subject to multiresolution analysis and can have its detail edited or replaced. The resulting curve is scan-converted to recover the contiguous $pixels$ entries required by the $Stroke$ type.

Stroke database

A stroke database maintains a collection of $Stroke$ instances, supporting addition and deletion of strokes, and various queries. It is important that the database operations and queries be quick enough to allow painting and editing at interactive speed.

We implement the stroke database using a modified k-D tree (see Sedgewick [15]). Each node of the tree corresponds to a region of the image; the children of the node partition that region. The partition is always horizontal or vertical and is chosen so as to distribute as evenly as possible the strokes of a region between the two children. Each leaf node contains a list of the strokes that intersect its region. For performance purposes, a limit of 10 strokes per leaf and a minimum size of 5×5 pixels per leaf are maintained (with the area restriction having precedence).

In the modified k-D tree, a given stroke may be referenced by several leaves since a stroke can cross partitions of the tree. The structure allows us to quickly find all strokes that may overlap a specified region by the usual recursion along the limbs of the tree that include the region. Minor extra bookkeeping is required when iterating through the strokes in the leaves to ensure that a stroke is not visited multiple times.

The queries supported by a stroke database include: finding all strokes within a given rectangular or circular region; finding all strokes that overlap a given stroke; and finding the stroke nearest a given pixel. Each query may specify criteria such as a particular id value. These queries allow the system to perform operations such as deleting a stroke and updating the screen as follows: first, find the stroke nearest the cursor; next, delete the stroke from the database and erase it from the screen; finally, find all strokes that overlap the deleted stroke and redraw them.

A.2 Global data objects

The system maintains several global data objects to support the interactive illustration processes:

- *Main stroke database and image bitmap.* The illustration is maintained in a dual representation: a stroke database maintains the collection of $Stroke$ instances that make up the illustration; and an image bitmap allows the system to quickly determine if a pixel has been drawn by one or more strokes. When storing to disk, only the stroke database needs to be saved; the image bitmap can be recreated by traversing the database and drawing all the strokes.
- *Clip-edge database and clip-edge bitmap.* To allow fast clipping of drawn strokes to outline edges, the system maintains a global bitmap into which all clipping edges are drawn (clipping is discussed in Section A.3). The clip edges can come from edge detection of the reference image or from freehand drawing. To allow the user to activate and deactivate edges, the edges are stored as $Stroke$ instances in a stroke database.
- *Stored stroke textures.* The system loads stored stroke textures on demand from a library on disk. A stored texture is defined as a rectangular region with toroidal wrap-around, so that the texture can seamlessly tile the illustration plane. Each texture is maintained in the system as a stroke database. For a prioritized texture, each stroke has an associated priority value. The stroke database of a stored stroke texture is queried but is not modified when the texture is used.
- *Reference image.* The system stores the reference image in memory, allowing quick pixel-by-pixel tone reference and stenciling. Unlike the image bitmap of the illustration, the reference image is an 8-bit greyscale. When a reference image is loaded from disk, the detected edges in the image are added to a clip-edge database and bitmap. We use a Canny edge extractor [2] to detect edges at several image resolutions. This potentially time-consuming processing is only done the first time a given reference image is used; the resulting edges are saved on disk along with the image, so that they can be loaded quickly in the future.

A.3 Drawing strokes

The process to "paint" with strokes is similar for the supported procedural textures—stippling, straight hatching, and curved hatching—and for stored stroke textures. The following pseudocode outlines this process:

```
Paint:
    for each brush position $P$
        while $S \leftarrow GenerateCandidateStroke(P)$
            $ClipStroke(S)$
            if $TestStrokeTone(S)$ then
                $DrawStroke(S)$
            end if
        end while
    end for
```

The steps of this process are described below.

☞ $\underline{GenerateCandidateStroke(P)}$: At each brush position P, the system may in general try to draw many strokes. Each invocation of $GenerateCandidateStroke$ returns the next stroke instance from a set of candidates. The next stroke returned may be generated dynamically based on the success of the previous strokes. The generation of candidate strokes depends on the texture:

- *Stippling.* There is only a single candidate: a stipple dot at a random location under the brush (chosen with uniform distribution in the brush's polar coordinates). The stipple dot is generated as a length 1 stroke.
- *Straight hatching.* The system tries a sequence of line segments with decreasing length, until a segment is drawn or a minimum length is reached. The midpoint of each stroke is a random location under the brush, and the direction and initial length are specified by the user. The direction may be fixed or aligned relative to the gradient of the reference image. The user may request a small randomization of the direction and length. The user may also specify that only full-length strokes be used, in which case if the initial candidate is not drawn, no further strokes are attempted. Each candidate stroke is a perturbed line segment, generated by the following pseudocode:

```
$\underline{PerturbedLineSegment(x_1, y_1, x_2, y_2, a, \omega, c)}$:
    ; $(x_1, y_1)$ and $(x_2, y_2)$ are the endpoints of the line segment.
    ; $a$ is the magnitude and $\omega$ the base frequency of waviness.
    ; $c$ is the magnitude of curviness.
    ; $random()$ value has uniform distribution on $[0, 1]$.
    ; $gaussian()$ value has normal distribution on $[-1, 1]$.
    $dx \leftarrow x_2 - x_1$
    $dy \leftarrow y_2 - y_1$
    $s \leftarrow \sqrt{dx^2 + dy^2}$
    $\delta \leftarrow 2\pi\omega\,(1 + \frac{1}{4} gaussian())$
    $\gamma \leftarrow \frac{1}{2}\delta gaussian()$
    $i \leftarrow 0,\ j \leftarrow 0,\ \phi \leftarrow 2\pi random()$
    for $\alpha \leftarrow 0$ to $1$ step $1/\max(|dx|, |dy|)$
        ; perturb line with sine waviness and quarter-wave curve.
        $b \leftarrow a\sin(\phi)/s + c\,(\cos(\frac{\pi}{2}\alpha - \frac{\pi}{4}) - 1)$
        $pixels[i] \leftarrow (x_1 + \alpha\,dx + b\,dy,\ y_1 + \alpha\,dy + b\,dx)$
        ; occasionally shift the sine wave frequency.
        if $j\delta > \frac{\pi}{2}$ and $gaussian() > \frac{1}{3}$ then
            $\gamma \leftarrow \frac{1}{2}\delta gaussian()$
            $j \leftarrow 0$
        end if
        ; update for next pixel.
        $\phi \leftarrow \phi + \delta + \gamma$
        $i$++, $j$++
    end for
```

  When needed, intermediate pixels are inserted in order to maintain the contiguity requirement of the $Strokes$ type.
- *Curved hatching.* Similar to straight hatching, the system tries strokes of decreasing length until one is accepted. The user specifies the initial length and direction relative to the reference image gradient. A curved stroke is generated by following the image gradient as a vector field (much as was done by Cabral and Leedom [1]) forward and backward for the given length.
- *Stored Strokes.* The system queries the texture's database for a list of strokes that lie under the brush, modulo tiling of the image plane with the texture. The strokes of the resulting list are tried in priority order for prioritized textures, or random order for non-prioritized textures. A prioritized texture may be flagged as *strictly prioritized,* in which case if a candidate stroke fails the tone test, the remaining lower-priority strokes are not considered. Each candidate stroke is generated by translating the stored stroke's $pixels$ to the proper tile in the image. Our system does not currently add any randomness to the strokes beyond that which was used when the texture was originally defined. Tiling artifacts are typically not objectionable if the illustration feature size is smaller than the tile size, but could be alleviated through random stroke perturbations.

☞ $\underline{ClipStroke(S)}$: The candidate stroke S is subjected to a series of clipping operations:

1. *To the bounds of the overall image.*
2. *To the brush.* Clip the strokes to the brush for stored stroke textures to give the user a traditional "textured paint." This clipping step is not performed for procedural textures; in this case, the candidate strokes are generated starting under the brush but may extend beyond its bounds.
3. *To clip-edges.* Trace from the center of the stroke out to each end, examining the corresponding pixels of the global clip-edge bitmap, stopping when an edge is met.
4. *To a reference-image stencil.* Trace from the center of the stroke out to each end, examining the corresponding pixels of the reference image. Can stop at black, white, or any of a number of user-defined ranges of image intensities.

The clipping operations return a "first" and a "last" index into the stroke's $pixels$ array, but before actually trimming the stroke, these indices are perturbed up or down by a small random amount to achieve ragged clipping as described in Section 2.5. The magnitude of the perturbation is adjustable by the user. If the stroke is clipped to zero length, it can be trivially rejected at this point.

☞ $\underline{TestStrokeTone(S)}$: Two tests are performed to see how stroke S affects the image. First, the stroke's pixels in the image buffer are tested: if all the pixels are already drawn, the stroke has no effect on the image and is trivially rejected. Next, the effect of the stroke on the image tone is determined: the stroke is temporarily drawn into the image bitmap and the resulting tone is computed pixel-by-pixel along its length, by low-pass filtering each pixel's neighborhood. Depending on the user's specification, the desired tone may be determined from the reference image's value (via similar low-pass filtering along the stroke), or may simply be a constant value. The stroke fails if it makes the image tone darker than the desired tone anywhere along its length.

☞ $\underline{DrawStroke(S)}$: To draw stroke S, its pixels in the image bitmap are set, the display is updated, and an instance of S is added to the main stroke database. For stored stroke textures, the system checks to see if the new stroke S overlays an existing instance of the same stroke—such an occurrence could happen, for example, if the earlier stroke was clipped to the brush and the user has now moved the brush slightly. Rather than adding the new stroke, the previously-drawn stroke is extended to include the new stroke's pixels in order to avoid overwhelming the data structures. Note that for a new instance of a stroke to align with a previous instance, any extra randomness should be exactly repeatable; the values for the stroke perturbations should be derived from a pseudorandom function over the illustration plane.

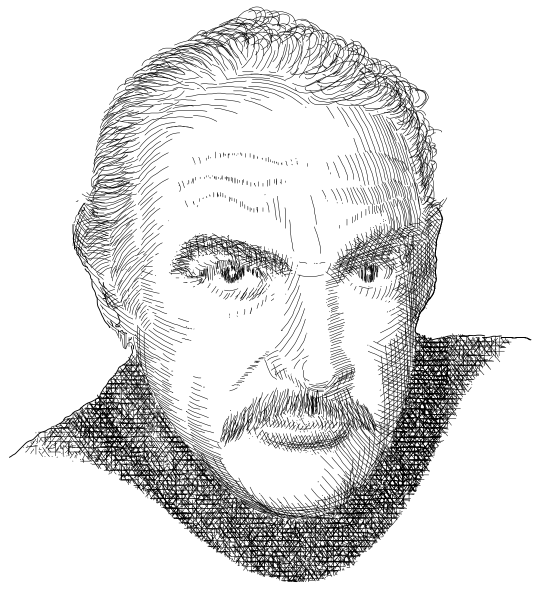
Figure 11: *An illustrated portrait. The reference image was a photograph by Douglas Kirkland [9].*