Hairy Brushes in Computer-Generated Images

by

Steve Strassmann

S.B., Computer Science
Massachusetts Institute of Technology
Cambridge, Mass.
1984

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
OF

MASTER OF SCIENCE

AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1986

Signature of the Author

....

Steve Strassmann
Department of Architecture
May 16, 1986

Certified by

..........................[illegible]..........................

David Zeltzer
Assistant Professor of Computer Graphics

Accepted by

....

Nicholas Negroponte
Chairman, Departmental Committee on Graduate Students

Hairy Brushes in Computer-Generated Images

by

Steve Strassmann

Submitted to the Department of Architecture on May 16, 1986 in partial fulfillment of the requirements of the degree of Master of Science.

Abstract

Paint brushes are modeled as a collection of bristles which evolve over the course of the stroke, leaving a realistic image of a *sumi* (Japanese watercolor) brush stroke. The major representational units are (1) Brush: a compound object composed of bristles, (2) Stroke: a trajectory of position and pressure, (3) Dip: a description of the initial condition of a class of brushes, and (4) Paper: a mapping onto the display device. A modular system allows experimentation with various models of ink flow and color change. By selecting from a library of brushes, dips, and papers, the stroke can then take on a wide variety of expressive textures.

Thesis Supervisor: David Zeltzer
Title: Assistant Professor

Contents

List of Figures

Chapter 1

Acknowledgements

The author is grateful to Karl Sims for the suggestion and use of the generalized polygon interpolation algorithm, and for being an all-around swell kind of guy.

I'd also like to thank the following people:

Valerie Chen for friendship, David Chen for Unix help, Bill Coderre for Zippyisms, Muriel Cooper for paint programs, Brian Croll for commiseration, Jim Davis for healthy criticism, Carl Feynman for fantasy, Alan Kay for believing, David Levitt for jazz, Ron MacNeil for tablet info, Margaret Minsky for bountiful suggestions, Marvin Minsky for inspiration, Kenji Taima for being Marvelous, Dave Zeltzer for advice and support, and all the Renaissance Men and Women at the MIT Media Laboratory.

Brownies supplied by Rosie's of Somerville and Cuisine Chez Vous of Cambridge. Incidental chocolate by Serenade, Lindt, Tobler, and others too numerous to mention.

This work was supported in part by a grant from Apple Computer, Inc., and an equipment loan from Symbolics, Inc.

Chapter 2

Introduction

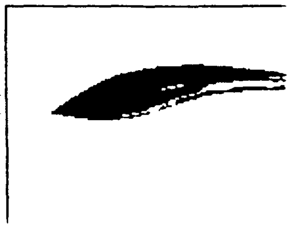
The "brushes" used in conventional computer painting systems are far simpler than real paint brushes. Usually no more than automated rubber stamps, they build up images by placing repeated copies of some static or simply derived pattern. Some systems offer "airbrushes," which simulate a spray of ink by painting pixels in a circular region around the brush.

This thesis describes an investigation into a far more realistic model of painting. The image left by a sopping wet brush or crumbly crayon dragged erratically across a sheet of textured paper can be generated by a representation which keeps track of the physical properties of the materials. This work is useful not only to artists who want to paint intractively, but also for automated rendering of natural (or non-realistic) scenes. As techniques like ray-tracing extend the ability of computers to render scenes with photographic exactitide, there will be a complementary advancement of techniques which allow computers to *suggest* scenes with artistic abstraction.

2.1 Previous Work

In Whitted [12], an unchanging anti-aliased image is dragged to draw a smooth glossy tube. Paint systems using input devices with three or more degrees of freedom (say, position and pressure) allow the user to vary some parameter of the brush pattern (say, radius or hue of a solid circle) as they paint. Lewis [5] describes stochastic and frequency-domain representations of texture, but these techniques do not adequately render the effects at the boundaries of discrete strokes.

Greene [4] describes an input device called the "drawing prism" which digitizes the image of a real brush (or other object) making optical contact with a transparent prism. Although the resulting images are realistic, the system has no representational abstraction higher than the pixel level. The system described in this thesis simulates a brush stroke using a hierarchy of representation, allowing repeatability and experimentation at many levels of control.



2.2 Sumi-e Painting

This research was inspired by the traditional Japanese art known as *sumi-e.*

Pronounced *soo-me-ay,* it comes from the Japanese words "sumi," the black ink used in calligraphy, and "e," meaning picture. Sato's work [8] has many examples and discusses the history, symbolism, and techniques of traditional and modern *sumi-e.* It includes the famous *Mustard Seed Garden Manual of Painting*, a compendium of 1000 years of *sumi-e* experience and technique first published in China in 1679.

Although there are a wide variety of *sumi-e* painting styles, one seems a particularly good candidate for computer simulation. Paintings in the *bokkotsu* style are characterized by a few well-placed strokes on a light background. Pictures with hundreds or more strokes may become practical some day, but for now *bokkotsu sumi-e* is appealing as a model because evocative pictures may be made in black and white, and with only a few strokes. The *bokkotsu* style emphasizes the quality of each stroke; this focuses the attention on the processes and materials involved in the construction of each stroke.

An example of computer-generated *sumi-e* can be seen in Fig. 2.1. The picture comprises 17 strokes, each defined as a spline with between 3 and

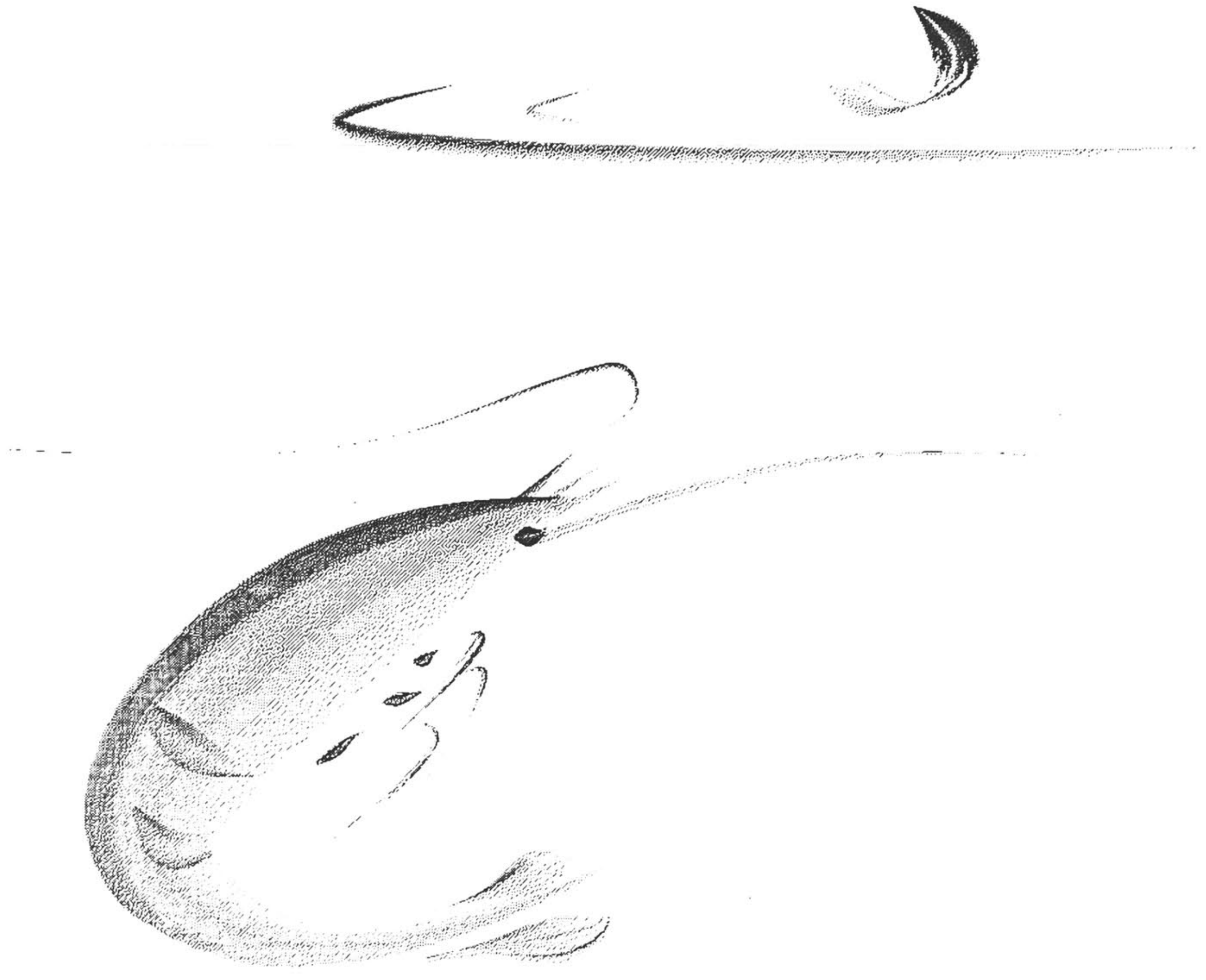
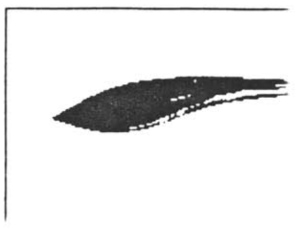


Figure 2.1: An example of *Sumi-e*: "Shrimp and Leaf"



8 control points. It is anti-aliased, and was generated on an 8-bit deep 640×480 pixel frame buffer. The design was drawn free-hand, interactively, using a mouse, after looking at some examples of similar paintings.

Chapter 3

The Representation

The key to a successful implementation is choosing the right representation. In attempting to simulate brushes, there is a broad spectrum of possible representations, ranging from the simple to the complex, which would have a corresponding degree of realism and computational expense. In this chapter, I discuss the basic representational units and my reasons for choosing them.

3.1 The Objects

All of the code is written in Zetalisp using *flavors*, an object-oriented programming style. The basic representational units are therefore flavors, or classes of objects. This creates a useful modular abstraction which allows

the user to deal with the many parameters of drawing in a managable and structured hierarchy.

They are:

1. Brush - a compound object composed of bristles
2. Stroke - a trajectory of position and pressure
3. Dip - a description of the initial state of a class of brushes
4. Paper - a mapping onto the display device

Bristles, which do a lot of the work of the brush, are also objects in their own right, but their description and definition is intimately connected with that of the brush.

3.1.1 The Brush

A brush can be thought of as a collection of bristles, each of which has its own ink supply and position relative to the brush handle. In the simple case, each bristle is a simple shape (dot or rectangle) in a regular one or two-dimensional lattice. In more complex brushes, the bristles may move relative to each other, so each must explicitly store its position and orientation relative to the brush's center. As the brush is moved through the trajectory specified by the stroke, two periodic computations are performed:

- The state of each bristle is updated.

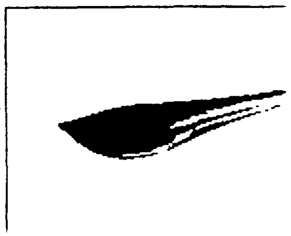
  Updating the bristles consists of evaluating one or more code fragments ("rules") which modify the color, ink quantity, relative position, or other property for each bristle.

- An image computed from the bristles is transferred to the paper.

  Typically, each bristle independently contributes an image (usually a one-pixel dot) to the patch. A single bristle contributes to the image only if two conditions both hold: it is applied to the paper with sufficient pressure, and it has ink remaining. A droplet of ink corresponding to that bristle's color at that point in time is added to the paper by sending a message to a paper object (see Sect. 3.1.4).

  Each bristle has a color: for *sumi* the color is simply a shade of gray, represented as a fraction between 0 and 1. It is assumed that all the ink on a given bristle is of the same color; however, neighboring bristles may be of different colors.

Since the details of how I implemented bristles have changed three times, the actual arrangement of the bristles for each implementation is discussed separately in Chapter 4.



3.1.2 The Stroke

A stroke is a set of parameters (e.g. position and pressure) which evolve as a function of an independent variable. This may be thought of as elapsed time, or the distance along the stroke; any monotonically increasing variable will do. I call this variable "time", and represent it with the symbol S (since T already has a special meaning in LISP). Its value is an approximation of the distance along the stroke.

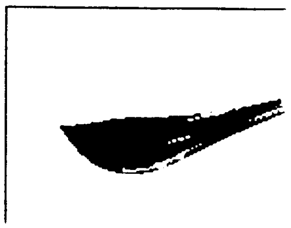
Since there is no special input hardware (other than a keyboard and a mouse) currently attached to our Lisp machines, the shape of the stroke is determined by a spline of 2D coordinates specifed using the mouse, clicking once to specify each control point. For each control point, the user can specify the pressure manually with the keyboard. The spline itself is a connected series of line segments sufficiently small to give the illusion of a smooth curve.

Position and pressure samples and the splines derived from them are stored in the "stroke" object. The user can edit an incorrect stroke, or select a different brush or dip for the same trajectory.

3.1.3 The Dip

In traditional Oriental painting and calligraphy, a complex texture of color and uneven distribution of ink can be applied to the brush. This can set up the patterns of light and darkness which can make a simple straight stroke look like a cylindrical segment of smooth bamboo, or make a cliff rising out of the ocean seem to be covered with moss on top. By separating the abstraction of dip from brush, one can use the same brush for a wide variety of strokes and effects, just as in real *sumi*. If one selects a particular brush and stroke, one can experiment with different dips to achieve exactly the desired effect.

Since moving a brush through a stroke uses up the ink and can change the position and color of the bristles, the dip must carry enough information to restore the brush to its initial state (or a sufficiently similar state), so that strokes can be repeated. This can be anything from using a simple rule to storing an explict snapshot of the state of each bristle. Such a rule is a procedure which has access to parameters such as the position of each bristle within the brush, or user-specified parameters like blotchiness or smoothness. Dipping a brush executes the procedure and/or copies the stored bristle parameters. Randomness can be introduced at the time of



creating the dip, and/or at each act of dipping.

3.1.4 The Paper

The paper object is responsible for rendering the ink as it comes off the brush. As each bristle decides to imprint itself, it sends a message to the paper indicating its position and other relevant parameters. The paper then reacts, usually by rendering a single dot of appropriate color at the appropriate point.

The paper concept is useful because it presents an abstraction which allows the system to run on frame buffers of various resolutions and depths. An arbitrary texture can be mapped over the stroke in several ways to simulate textured papers (see Sect. 5.5), using an algorithm similar to conventional texture-mapping. The paper abstraction also has the potential of modelling such effects as the wetness or absorptive properties of real paper, but I have not yet implemented such behavior. I discuss some of the possibilities in Sect. 7.5.

Currently, the user can draw on frame buffers with either 1, 8, or 24 bits per pixel, at either NTSC (640×480) or high (1280×1024) resolution. Papers of arbitrarily higher resolution can be simulated because of the supersampling patch provided for anti-aliasing (see Sect. 4.3.3).

3.2 Why This Representation?

Although I have identified four major abstractions and a host of effects (described in Chapter 5) which can be created with them, it is still too early to pin down their exact specifications. Rather than design a system which exactly emulates, say, a camel hair brush dipped in a particular brand of india ink, I chose to design a framework in which many categories of paint-like media can be expressed.

Anyone who has ever walked into an art store can attest to the fact that – to the novice – there seems to be a bewildering number of degrees of freedom to control in artistic media. It is important to choose a representation that is modular for the following reasons:

- The user can become familiar with a small repertoire of familiar tools. For example, different brushes can be used and re-used over the same stroke to explore various effects. A certain dip or paper, once perfected, can be saved for later use.
- Since the simulation is based on a modular and hierarchically organized set of effects, aspects of the simulation can be replaced or augmented with more sophisticated algorithmic models as they are developed.
- The same picture can be rendered at many levels of complexity, from

quick drafts to final images, by selectively "turning on" different effects independently of each other.

Chapter 4

The Three Implementations

Although the basic idea hasn't changed, I went through two different rendering implementations before settling down on the third and current one. In this chapter, I describe each of them and discuss their relative advantages and disadvantages.

I think it is useful to describe the first implementation, since the notion of a two-dimensional footprint is potentially more realistic than a one-dimensional one. I include the description of the second implementation because it aids understanding of the third implementation.

4.1 The 1st Implementation: The 2D Footprint

When I first considered simulating the effect of a brush moving across a page, I chose an extension of conventional "paint" systems which move a shape across the page. If you think of this shape as a "footprint", the idea is to move the shape through a trajectory (the stroke) a bit at a time, and after each motion, copy the brush pattern to the frame buffer. To get a more dynamic brush, one computes the evolution of the bristles periodically through the stroke, changing the footprint image before each stamp.

4.1.1 Anti-aliasing the Dynamic Brush

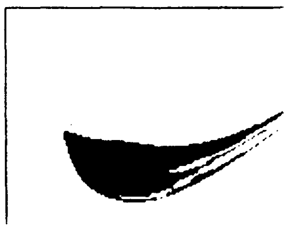
In the case of drawing an anti-aliased image, I assumed that the bristles are much smaller than an image pixel. I use the super-sampled patch idea suggested in Whitted [12] for anti-aliased brush drawing, adapted for a dynamic brush. One can consider the brush to be moved across a virtual screen whose resolution is some multiple (usually 1, 4 or 16) of the resolution of the display screen. This virtual screen never needs to be created, since the brush only affects a small patch of it at any time. At the beginning of a stroke, an array just large enough to enclose the brush throughout the stroke is allocated. At each point on the brush's path along the stroke, the image left by the brush is drawn onto the patch at high resolution. If the brush tries to leave the region cached in the patch, the patch is moved. Values from the trailing edge(s) are merged (by taking the average of the n×n-pixel region) and written out onto the screen, and values from the leading edge(s) are copied to the corresponding multiple locations in the patch. These locations are in fact the ones vacated by the old values, hence the image "wraps around" on the patch.

The actual details of the algorithm are described in Appendix B.

4.1.2 Problems with the 2D Brush

- *Rotation.* One thing that was immediately apparent was the difficulty of specifying the rotation of the brush image as the brush moves through the stroke. This was partially a limitation of my input hardware (a mouse), which made it very difficult to specify rotation, and partially a question of style. I felt that the shape of the image should rotate naturally and automatically to follow the path of the stroke, but I couldn't think of a simple way to compute this. In practice, most of my early images were drawn with brushes which did not rotate over the stroke.

- *Bristle Spreading.* Although the implementation allowed the bristles



to have an arbitrary position relative to the brush center, it was most natural to arrange them in a regular square lattice. This raised the problem of how to simulate the phenomenon of bristle spreading under pressure. If the brush were wet, there's the question of how to fill the gaps between bristles. If the brush were partially wet or dry, there's the even more difficult question of how to ***partially*** fill the gaps.

- *Predictability of Visual Effect.* The visual effect of a two-dimensional shape was hard to predict, since the trailing edge of the brush has the "final say" on what image would be left on the paper. The computation of complicated or interesting behavior in the leading edge or middle regions seemed to be "wasted".

- $O\left(N^2\right)$ *Computation.* This algorithm grows as $O\left(N^2\right)$ where N is the diameter of the brush. This causes the rendering to be very slow for brushes larger than about 30 bristles in diameter.

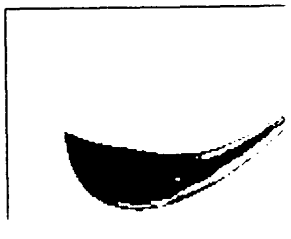
4.2 The 2nd Implementation: Trapezoids and Elbows

It seemed pretty obvious I should consider representing the brush as having a one-dimensional footprint. Early on, I assumed the brush to be like a windshield wiper that stays oriented perpendicular to the path. Since the path is represented as a series of nodes connected by line segments, a first approximation is to draw a trapezoid over each line segment and a transitional "elbow" at each node. The width at each node is defined by the pressure at that node (see Sect. 5.4), and the width is constant at each elbow, and linearly interpolated over each segment.

The brush is now changed to a one-dimensional array of bristles. The nature of the computations performed for things like color and quantity evolution is pretty much unaffected by the change in geometry, except that the neighborhood of each bristle is now reduced from 4 to 2 neighbors, and it is more practical to specify the dip using a mouse on a two-dimensional graph.

4.2.1 Advantages over the 2D Brush

- *Rotation.* Rotation no longer has to be (nor can be) specified by the



user instead, each segment of the brush's path is rendered by rotating the brush so it stays perpendicular to the segment.

- *Spreading.* Spreading is accomplished by defining the width of the brush at any point to be a function of the pressure at that point. Since the pressure is linearly interpolated between nodes, the result is a trapezoid connecting each pair of nodes.

- *Predictability.* Inasmuch as one knows the state of the bristles as they evolve, one can predict the image they would leave behind, since it is not complicated by many bristles writing over the same region of the paper.

- *$O(N)$ vs. $O(N^2)$ Computation.* The rendering for brushes of all sizes is significantly faster, and using very large brushes is practical.

4.2.2 The Algorithm

For an explanation of the notation not described here, see Appendix B.

I still keep the two-step process of moving and evolving the brush, then transferring an image from the brush to the paper. The complexity now lies in the problem of moving the brush. It is important to note that conventional polygon-filling algorithms are not useful to me here. This is because I need to draw each pixel *in chronological order* in order to capture the evolution of the bristles as they moved along the stroke. The path of the brush is computed as follows:

1. Specify the stroke path in the same way as previously. This is represented as the n points $(x, y, p, s)_i$ for $i = 0, \ldots, (n-1)$.

2. The brush's center moves along the line segments connecting consecutive points $(x, y)_i$. Let's call three such points A, B, and C.

3. The brush sweeps over the trapezoid over $\overline{AB}$, drawing a slice at a time, then (if there is a C) it pivots through the elbow over B. Before each move, the bristles are allowed to evolve.

4. The brush's image on the paper is a line segment, swept through a line segment. These lines are computed by a version of Bresenham's algorithm [7, p. 40] which I call "dense" Bresenham's [Fig. 4.1]. It's effectively the same, except that, in traversing the line, only one coordinate at a time is allowed to change from one pixel to the next. This helps ensure the brush covers more of the paper.

Anti-aliasing is handled the same way as described above. The bounding box of the brush is a square whose side is the width of the brush at $p = p_{max}$.

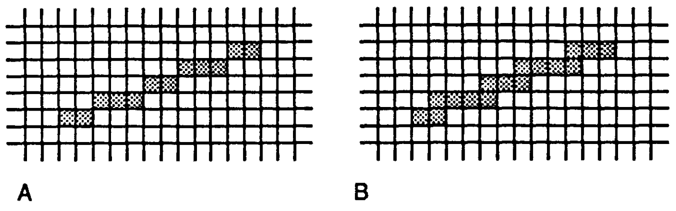


Figure 4.1: Rasterizing a line using (A) Bresenham's algorithm (B) "Dense" Bresenham's.

The brush is always centered in the bounding box.

The trapezoid doesn't actually extend from A to B, because that would overlap with the elbows. Instead, auxiliary points are determined (as shown in [Fig. 4.2]). These points must satisfy the following constraints:

- If $\overline{AB}$ is the first segment of the stroke, $A = M$.
- If $\overline{AB}$ is the last segment of the stroke, $N = B$, and there is no elbow.
- $\overline{EH} \perp \overline{AB}$
- $\overline{FG} \perp \overline{AB}$
- $\overline{IG} \perp \overline{BC}$
- M bisects $\overline{EH}$
- N bisects $\overline{FG}$

- $\left|\overline{BG}\right|$ is the width computed from the pressure at B
- $\left|\overline{NB}\right| = \left|\overline{BO}\right|$

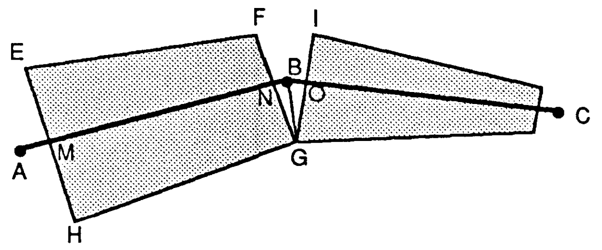


Figure 4.2: Construction of the trapezoid $EFGH$ and elbow FGI.

Care must be taken to detect and handle the case in which the elbow turns the other way.

4.2.3 Problems with Trapezoids and Elbows

The main problem was coverage. No matter how hard I tried, there were pixels on the page that were not covered by the brush.

- If the region covered by a brush is a line (determined by Bresenham's algorithm), and it sweeps out a path which is a line, then there are missing pixels in positions which are function of the orientation of the brush. This results in bizarre artifacts. For example, moving a brush along a horizontal path leaves an image twice as dark as a 45° path

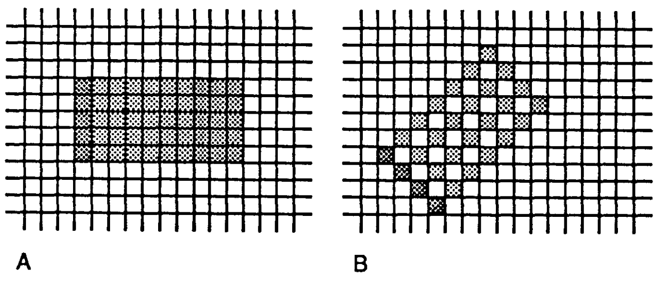


Figure 4.3: A brush swept horizontally gets all the pixels. Swept diagonally, it misses half of them.

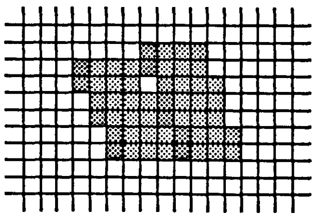


Figure 4.4: A brush whose width changes misses pixels.

[Fig. 4.3]. I tried to solve this by defining the path, then the brush, as a line drawn using “dense” Bresenham's.

- Even using a dense Bresenham's algorithm, there are occasionally missed pixels as the brush changed width. These artifacts arise when the jaggies of consecutive brush images don't coincide neatly, as shown in [Fig. 4.4].

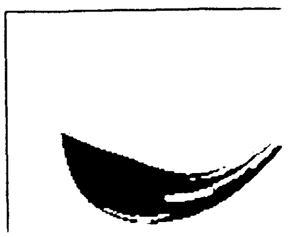
- Similar artifacts arise when the elbow is drawn. One solution is to actually double the thickness of the brush's image (e.g. going back to a 2D image). Then, if it is rotated, the liklihood of missing a pixel is lessened.

- The computation of the trapezoid corners and the elbow is rather awkward, especially for paths of high curvature. It is very difficult to satisfy the constraints mentioned in Sect. 4.2.2 with a simple algorithm. Instead, the code that computes the corners is a messy labyrinth of special cases and approximations that have not been proven valid for all possible strokes.

4.3 The 3rd Implementation: Polygons

The idea for the final implementation was suggested by Karl Sims, a fellow graduate student at the Media Lab. Karl had implemented a generalized polygon interpolation algorithm [10] for use with such rendering algorithms as Phong and Gouraud shading. Given a polygon with V vertices, and a N-dimensional vector value at each vertex, it generates the following:

- A list of all the pixels contained by that polygon.
- For each pixel, an N-dimensional vector which is the linear interpolation of the values at each of the V vertices.



In addition, the algorithm had the desirable property that if two such polygons shared a common edge, every pixel along that edge belonged to one and only one polygon.

4.3.1 Advantages over Trapezoids

There is a subtle reason why this algorithm is profoundly better than the previous two. In the previous implementations, the question I would ask was

> *Here's the brush. Now, what pixels does it draw onto?*

My frustration arose from not being able to always identify all the pixels which should have gotten drawn. This was manifest as various kinds of aliasing artifacts and wasted computation. The correct way to phrase the problem is:

> *Here are the pixels. Now, what part of the brush did they get drawn with?*

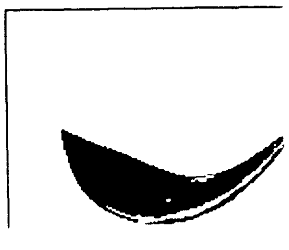
The rephrasing is similar to the idea in ray-tracing of working back from the eye to the light source, not the other way round. Here are a few more advantages of the third algorithm:

- In addition to eliminating the artifacts of missing pixels, this implementation provides for a better level of abstraction. Since the pixels "look up" the brush, it becomes easy to separate the concept of brush size in bristles from brush size in pixels.

- Every pixel along the brush path is sorted and drawn chronologically. This makes the rendering seem more intuitive to the casual observer, and if it were accellerated (using faster hardware) enough for an interactive system, it would give the user a real 'feel' of the ink flowing from a brush.

- Drawing the pixels in chronological order also guarantees that effects which depend on the order in which the ink is drawn can be used.

- Every pixel is drawn on exactly once (unless the stroke doubles back over itself), and exactly one bristle is responsible for each pixel. Although this may not be true to reality, it is a sufficiently good approximation that realistic images can be generated. Since the essence of the trailing edge of a brush is compiled into the state of a single row of bristles, it is easier to understand and predict the resulting image.



4.3.2 Algorithm

The idea is very similar to the previous one. The path is split up into nodes specified by a sample of position and pressure. Between each pair of nodes (called A and B), a segment ($\overline{AB}$) is created. If $\overline{AB}$ is not the last segment, the next point is called C.

For each segment, a quadrilateral ($EFGH$) is constructed which has the following properties [Fig. 4.5]:

- A bisects $\overline{EH}$
- B bisects $\overline{FG}$
- $\left|\overline{EH}\right|$ is the width computed from the pressure at A
- $\left|\overline{FG}\right|$ is the width computed from the pressure at B
- $\overline{FG}$ bisects $\angle ABC$.

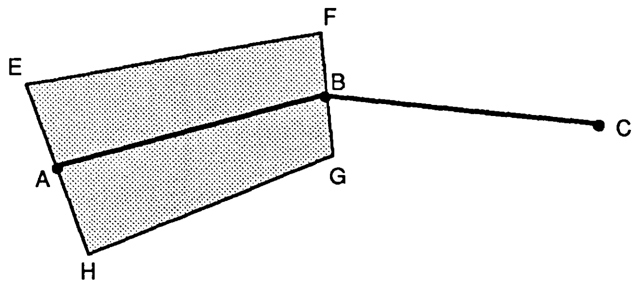


Figure 4.5: Construction of the polygon for the 3rd algorithm.

One exceptional case is when the lines $\overline{EH}$ and $\overline{FG}$ actually intersect [Fig. 4.6]. This can happen if $\overline{AB}$ is relatively short compared to the width $\overline{EH}$. Since the polygon interpolation algorithm insists on being handed vertices in clockwise order, one cannot simply pass on the quadrilateral $EFGH$. I call this the "bow-tie" case, and I handle it by partitioning the bow tie into two triangles, and rendering each one independently. Note that one cannot simply swap the offending vertices, since the chronological order of the vertices must be preserved.

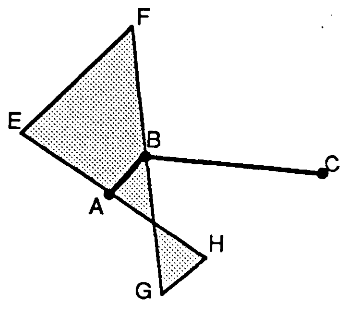


Figure 4.6: The annoying bowtie case.

Once the polygon's vertices are found, three properties are generated for each pixel using 2D interpolation algorithm.

1. Its position on the frame buffer (X, Y). This is generated in the course of the interpolation.

2. Its position along the stroke (S). This is done by interpolating (S_A, S_B, S_B, S_A) on polygon $EFGH$.

3. Its position across the brush (B). This is done by interpolating $(1,1,0,0)$ on polygon $EFGH$.

As the pixels are generated, they are sorted chronologically (by S) into a temporary array of length $(S_B - S_A)$. Then, as the brush moves and updates, all pixels belonging to that portion of the stroke can be drawn. For each pixel, its abstract brush position B (where $0 \leq B \leq 1$) is used to determine the nearest responsible bristle(s).

4.3.3 Anti-aliasing

As in the previous two implementations, anti-aliasing must be done with supersampling, since the brush could theoretically change anywhere, at any time; i.e. every pixel could be an edge. Instead of using a scrolling patch, though, a much simpler patch is used. Since it is assumed that the areas covered by the bounding boxes of consecutive polygons do not overlap much, the complexities of scrolling are omitted.

- For each polygon, a patch the size of the polygon's bounding box, scaled by R, is dynamically allocated.
- The corresponding region is scaled and copied from the screen to the patch.

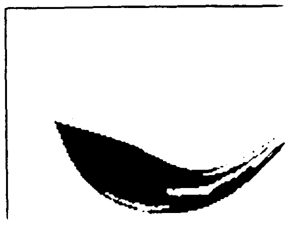
- The polygon is drawn onto the patch at high resolution.
- The patch's contents are scaled down (by local averaging) and copied back to the frame buffer.

4.3.4 Efficiency

The computational time consumed by the algorithm can be separated into two parts:

- The serial part; this is the computation of the stroke geometry, e.g. computation of the polygon vertices and edges.
- The parallel part; this can be broken into two parts:
  - Each bristle executes the evolution rules to determine its next state.
  - Each pixel consults the brush to determine what color it should become.

From running several informal timing benchmarks, it seems that about 90% to 99% of the computation on a serial machine is occupied by the parallel part of the algorithm, except for pathologically small strokes and brushes. Thus, although the polygon-vertex computation seems complex, it occupies an insignificant amount of time compared to the rendering.



Of the parallel part, the ratio of time between the two parts is very much a function of how big the brush is, as measured both in bristles and in pixels. Another important factor is the complexity of the evolution rules.

These results are encouraging for an implementation on a parallel processor that will take advantage of the inherently parallel computation being performed, and make real-time computer brush painting a reality.

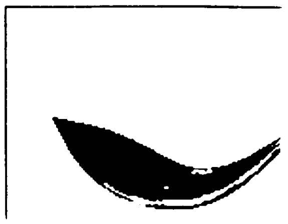
Chapter 5

Effects

In this chapter, I will describe some of the effects one can control by changing different parameters of the simulation. Although there may seem to be a bewildering myriad of parameters to control, it's important to recognize that each parameter has an intuitively recognizable function, and its effect on the image can be appreciated with a minimum of experimentation.

5.1 Ink Quantity

The ink supply on each bristle is assumed to be a reservoir of a finite quantity of fluid, which gets replenished each time the brush is dipped. The quantity is decreased as the brush moves through the stroke, and eventually the



bristle runs out. When the quantity drops to zero, that bristle no longer contributes to the image on the paper.

If a scratchy breakup at the tail of each stroke is desired, the dip should put just the right amount of ink on the brush, including selecting a few bristles to be short-changed so they run out early [Fig. 5.1]. If the stroke is known at the time of the act of dipping, its length is used to help determine the quantity of ink deposited on the bristles. There are parameters which control how many bristles get short-changed, and by how much, either as a fraction of the total stroke length or in units of absolute distance.

5.2 Ink Color

Each bristle has a color: for *sumi* the color is simply a shade of gray, represented as a fraction between 0 and 1. It is assumed that all the ink on a given bristle is of the same color; however, neighboring bristles may be of different colors.

The distribution of color across the brush may be specified as constant, or a linear ramp from one value to another, or as an explicit list of arbitrary values [Fig. 5.2]. Although this distribution must be specified for the beginning of the stroke, there are several ways of thinking about how the color

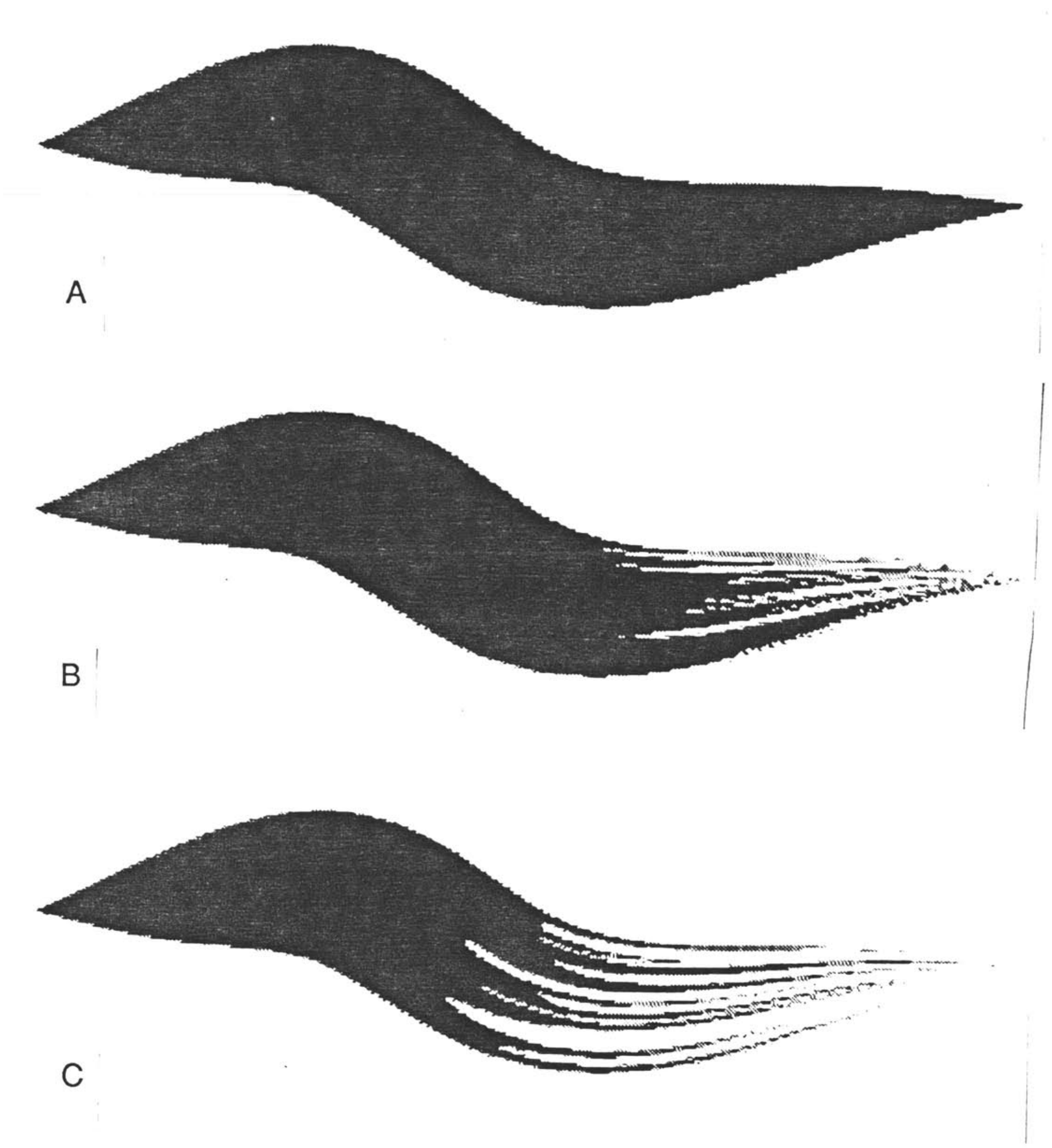
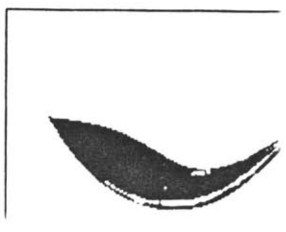


Figure 5.1: Different quantities: (A) A wet brush (B) 50% of the bristles are approx. 33% dry. (C) 75% of the bristles are approx. 50% dry.



evolves over the stroke:

- A distribution is specified for both the start and end of the stroke. At any point in the middle of the stroke, the color of a given bristle is linearly interpolated between the starting and ending values specified for it [Fig. 5.3]. This idea may be extended to generalized distribution samples at arbitrary points in the stroke.

- From the starting distribution, diffusion may be simulated by smoothing the colors of neighboring bristles [Fig. 5.4]. Each bristle updates its color according to a partial interpolation. For example, if

  - C_{i_t} is the color on the ith bristle at time t,
  - D is a speed-of-diffusion parameter between 0 and 1 (1 is rapid diffusion),
  - and the bristles are assumed to be regularly spaced,

  Then $C_{i_{t+1}} = C_{i_t}(1 - D) + \left(\frac{C_{i-1_t} + C_{i+1_t}}{2}\right) D$.

- A generalized evolution algorithm can be supplied [Fig. 5.5]. The color on a bristle may be a function of brush pressure, distance from the origin, or even the quantity of the ink remaining (see Sect. 5.3).

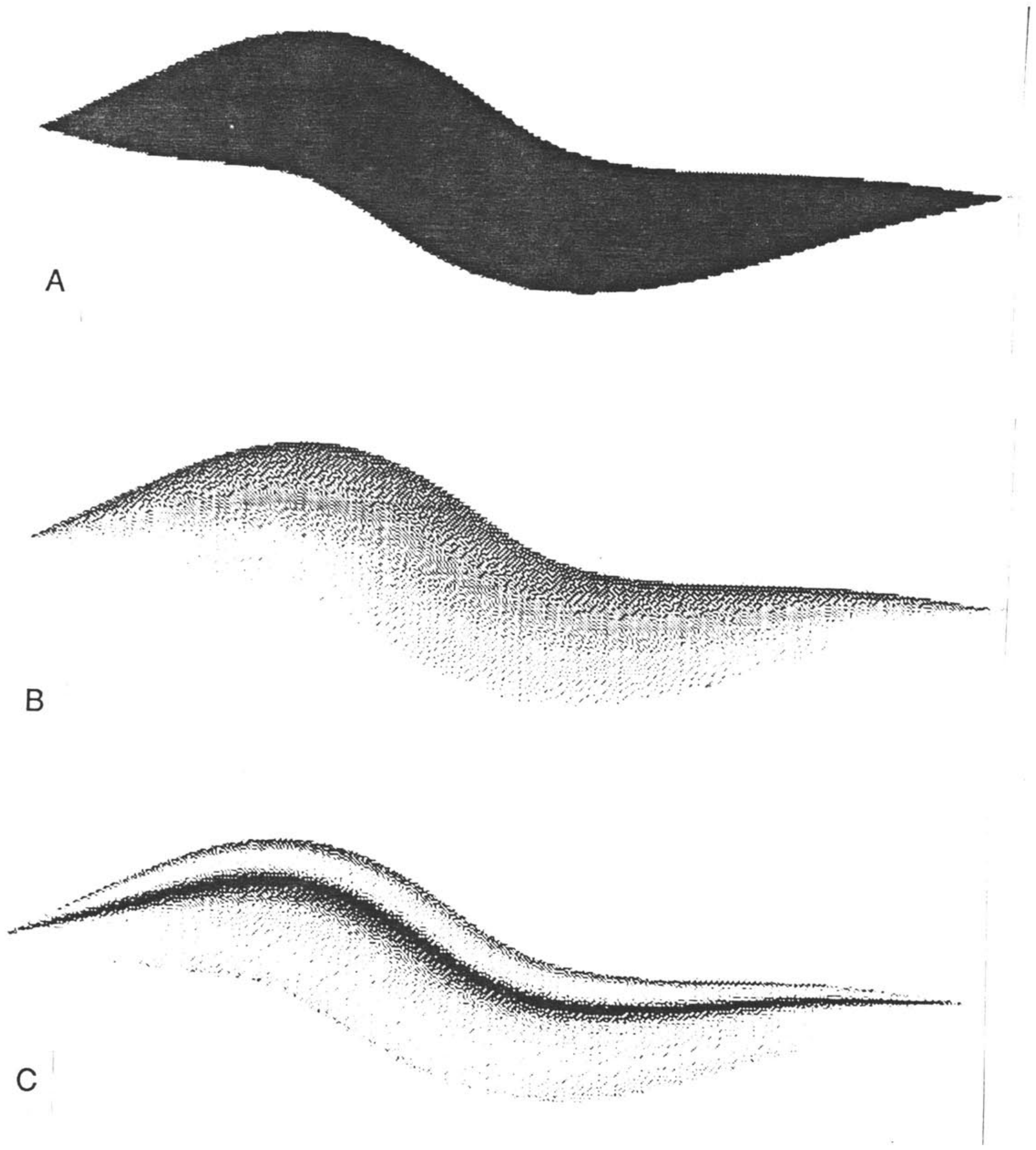


Figure 5.2: Different colors: (A) Constant (B) Linear (C) User-specified

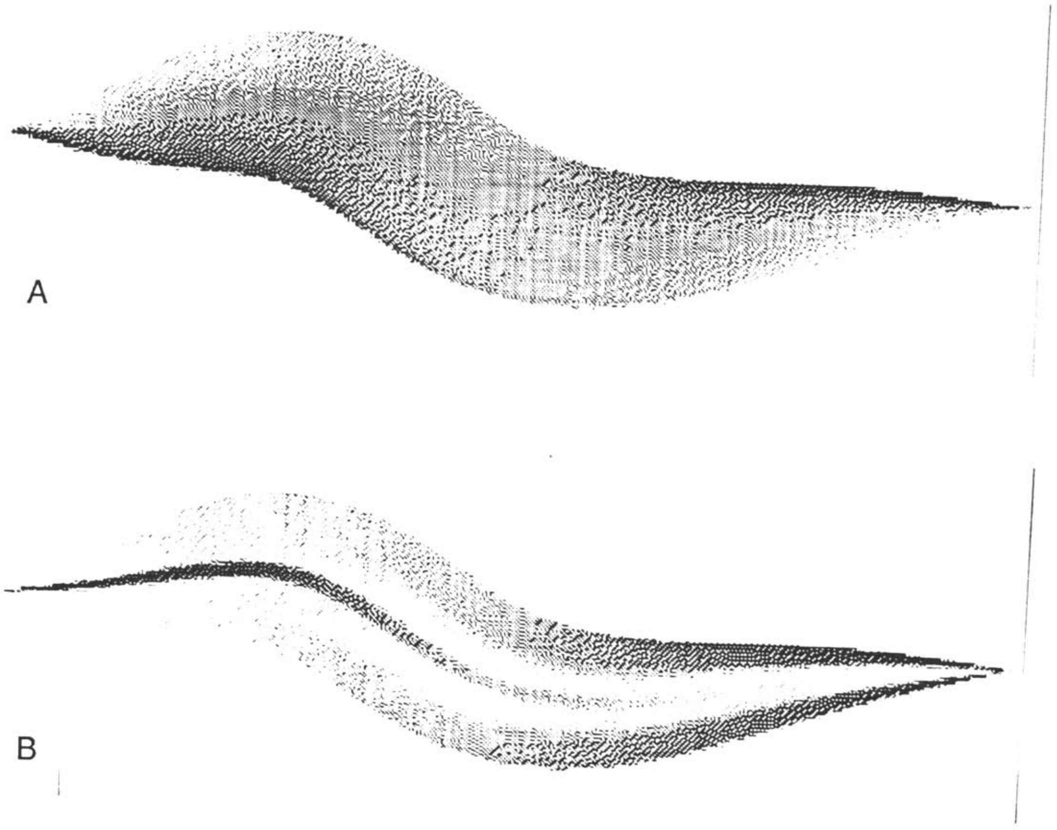


Figure 5.3: Color Interpolation: (A) Start/end interpolation from one ramp to another (B) Interpolation from spike to notch.

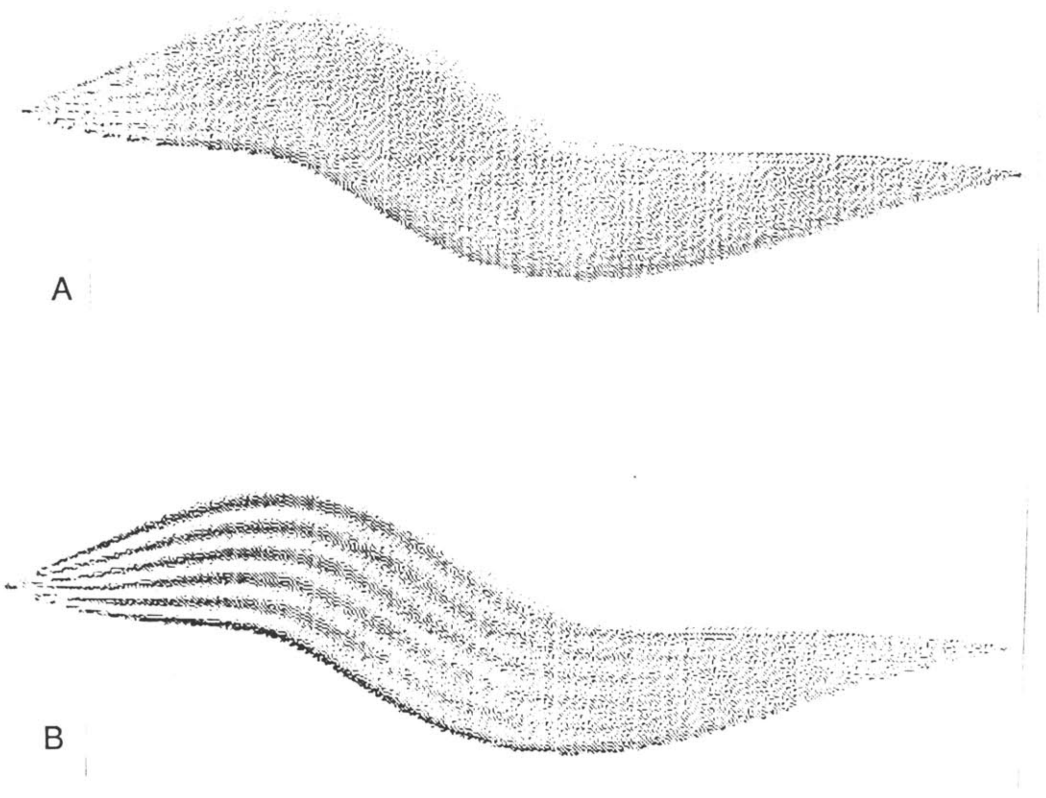


Figure 5.4: Color Diffusion: (A) Fast diffusion ($D = .5$) (B) Slow diffusion ($D = .1$)

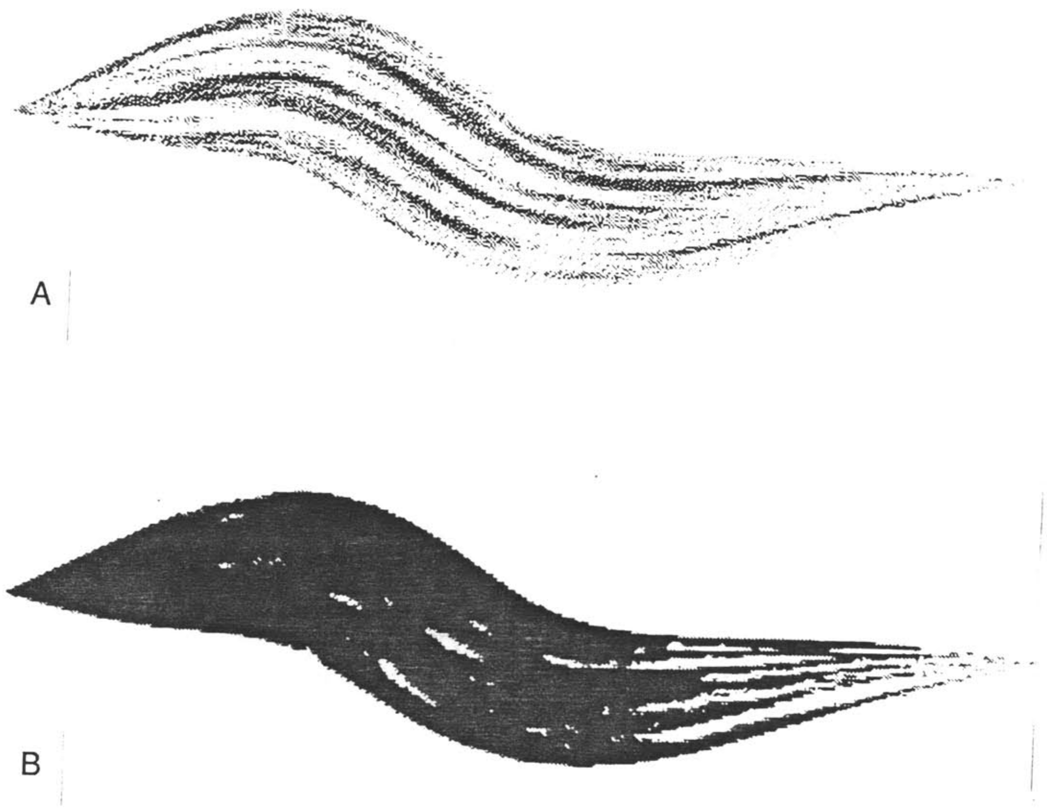


Figure 5.5: Evolution Rules: (A) Random evolution of color (Brownian) (B) "Ink stealing" evolution of quantity

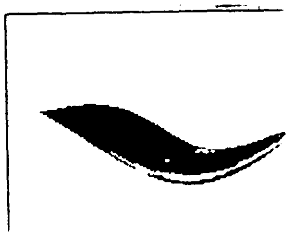
In addition to or instead of the above phenomena, a pattern can be texture-mapped onto the stroke (see Sect. 5.5).

Once the color of the ink on the bristle is decided, the color to place on the paper must be computed. The paper may already be colored due to either the paper's natural texture or previously deposited ink. The user may supply a *color combination function* of two or three inputs to be evaluated each time a given bristle attempts to draw on a particular pixel of the paper; the inputs are the ink color (C_i), the color of the paper at the point to be drawn upon (C_p), and an optional value derived from the texture-mapping array, if there is one.

For *sumi* the default function used is a very simple one: the darkness at the intersection of several strokes is assumed to equal the darkness of the darkest stroke (e.g. $C_{p_{t+1}} = \max(C_{i_t}, C_{p_t})$)

5.3 Evolution of Quantity and Color

Jostling of neighboring bristles sometimes transfers ink among them; this affects both the quantity and color of the bristles concerned. This is modeled by thinking of the brush as a cellular automaton [13] with a small procedure (rule) for quantity and color transfer. As the brush moves across the page a



bit at a time, all bristles repeatedly execute the same rule, which can refer to the parameters of each bristle and its immediate neighbors. It may compute a new value for any of that bristle's parameters and modify it accordingly.

For example, one rule can allow a near-dry bristle to run out of ink, then temporarily "discover" a new supply (either by stealing from a neighbor, or just conjuring it out of nowhere) to create islands of ink and whitespace in the middle of the stroke [Fig. 5.5, (B)]. Incorporating an element of randomness into the rules can give rise to rich textures. On the other hand, avoiding randomness may be necessary in applications like some kinds of animation, where the user wants the complex texture of stroke to be consistent from frame to frame (see Chapter 6).

5.4 Pressure

The pressure on a particular bristle is a function of the geometry of the brush and the overall pressure on the brush at a certain point in the stroke.

Changing the applied pressure during the stroke can have two different kinds of effects:

- *Spreading.* Pressing harder can spread the bristles further apart.

- *Contact.* Pressing harder can bring more bristles into contact with the paper.

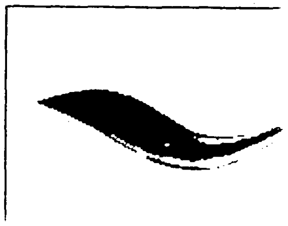
Under spreading, each bristle's distance from the brush center is an arbitrary function of the applied pressure. By default, distance is linearly proportional to pressure, but some interesting effects can be demonstrated by exploring other relationships [Fig. 5.6].

One can also consider that greater overall pressure brings more bristles into contact with the paper [Fig. 5.7]. A value is assigned to each bristle which represents the minimum brush pressure necessary to bring it into contact with the paper. For example, to simulate a round brush of radius 1, each bristle gets a pressure-threshold proportional to the arcsine of its distance from the center of the brush.

Intermittent contact with the paper near the pressure threshold is simulated by adding a rule which causes perturbations (either overall, or for individual bristles) in two parameters:

- Changing the brush's pressure implies one's hand is oscillating.
- Changing the the pressure-threshold of a bristle implies the geometry of the brush is changing.

Although these parameters have different meanings, the image ultimately



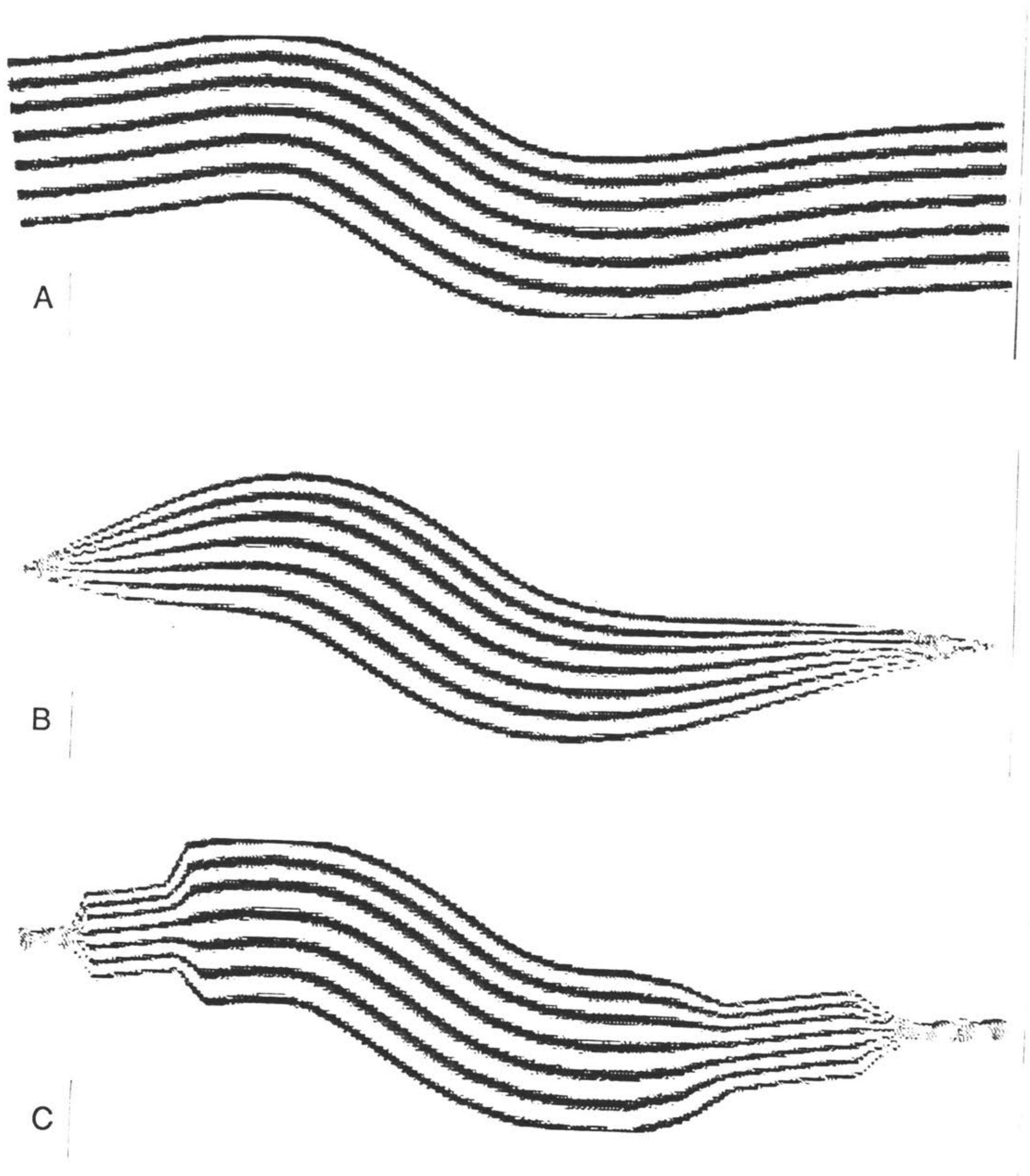


Figure 5.6: Spreading under pressure: (A) Constant width (B) Width $\propto$ pressure (C) Width quantized by user-supplied function

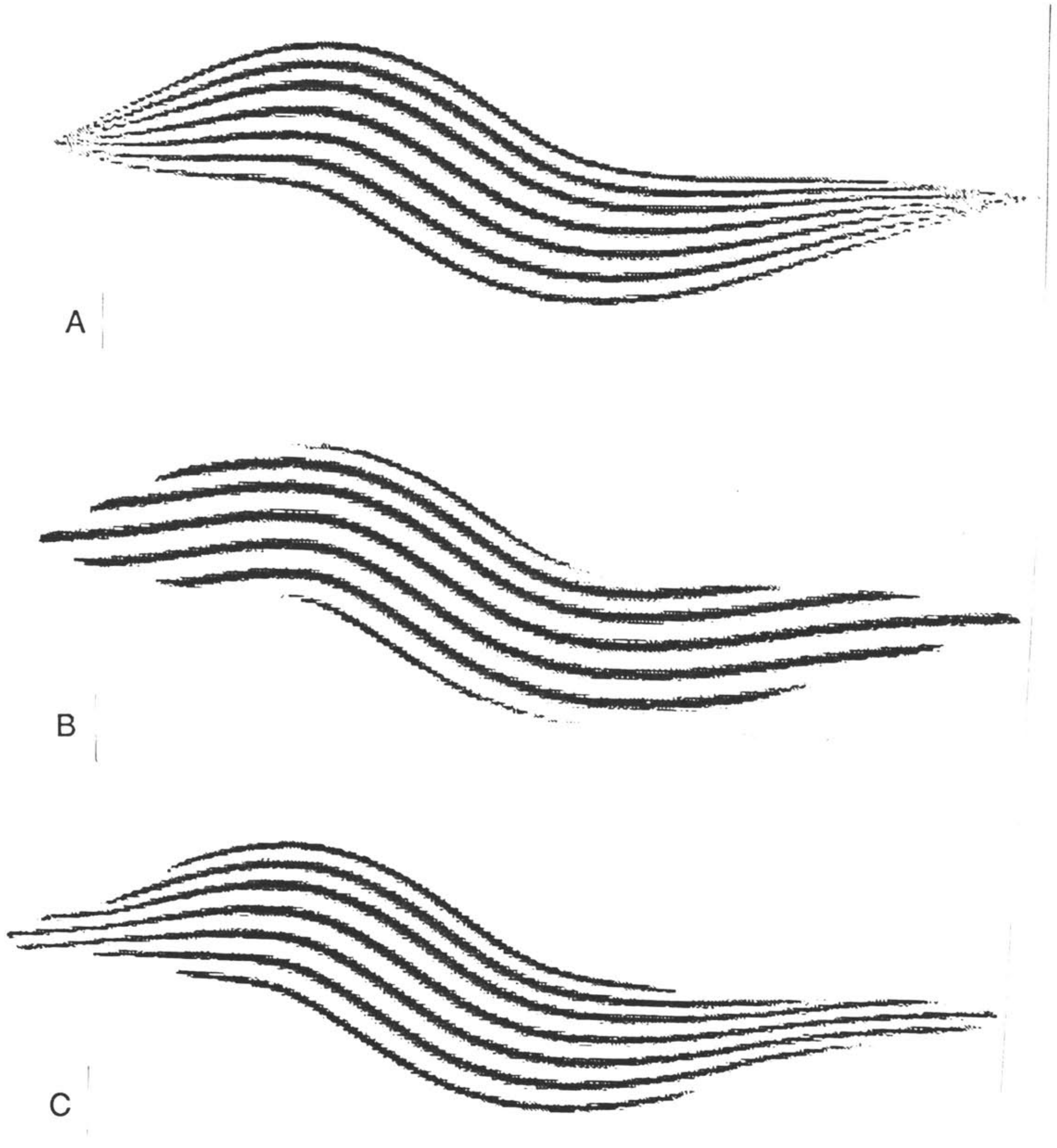
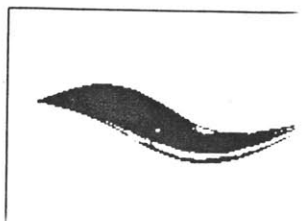


Figure 5.7: Two interpretations of pressure: (A) More pressure spreads bristles (B) More pressure brings more bristles into contact (C) A combination of these two effects.



depends on the difference between them; so it doesn't matter which modified as long as one is consistent. A more realistic test for determining contact might take into account the orientation of the brush and hysteresis (stickiness).

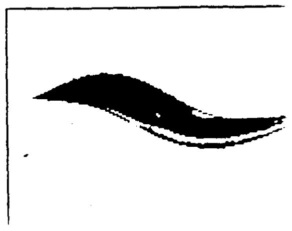
5.5 Texture Mapping

Some interesting effects can be realized by mapping a texture onto the image of the stroke [Fig. 5.8]. There are at least two ways of computing the mapping:

- A rectangular array representing the texture of the paper is mapped by a straightforward flat tiling. When a bristle attempts to draw ink of a certain color on a given pixel, the array element corresponding to that pixel is used.
- A one or two dimensional array is mapped along the long axis of the stroke (this is only used in the implementations where the brush has a one-dimensional footprint). The array element used corresponds to how far along the stroke the brush has travelled. For example, a simple 1D texture (say, a sine wave) mapped onto a curvy stroke gives the impression of banding similar to a raccoon's tail. If the texture map

is two-dimensional, the bristle's radial distance from the brush center is used to compute the array index in the second dimension.

Once a value is supplied by the texture array, it is used in the user-supplied color combination function (see Sect. 5.2). For *sumi* the texture value is a number (usually a fraction between 0 and 1) which is multiplied by the ink color to selectively attenuate it before applying it to the paper.



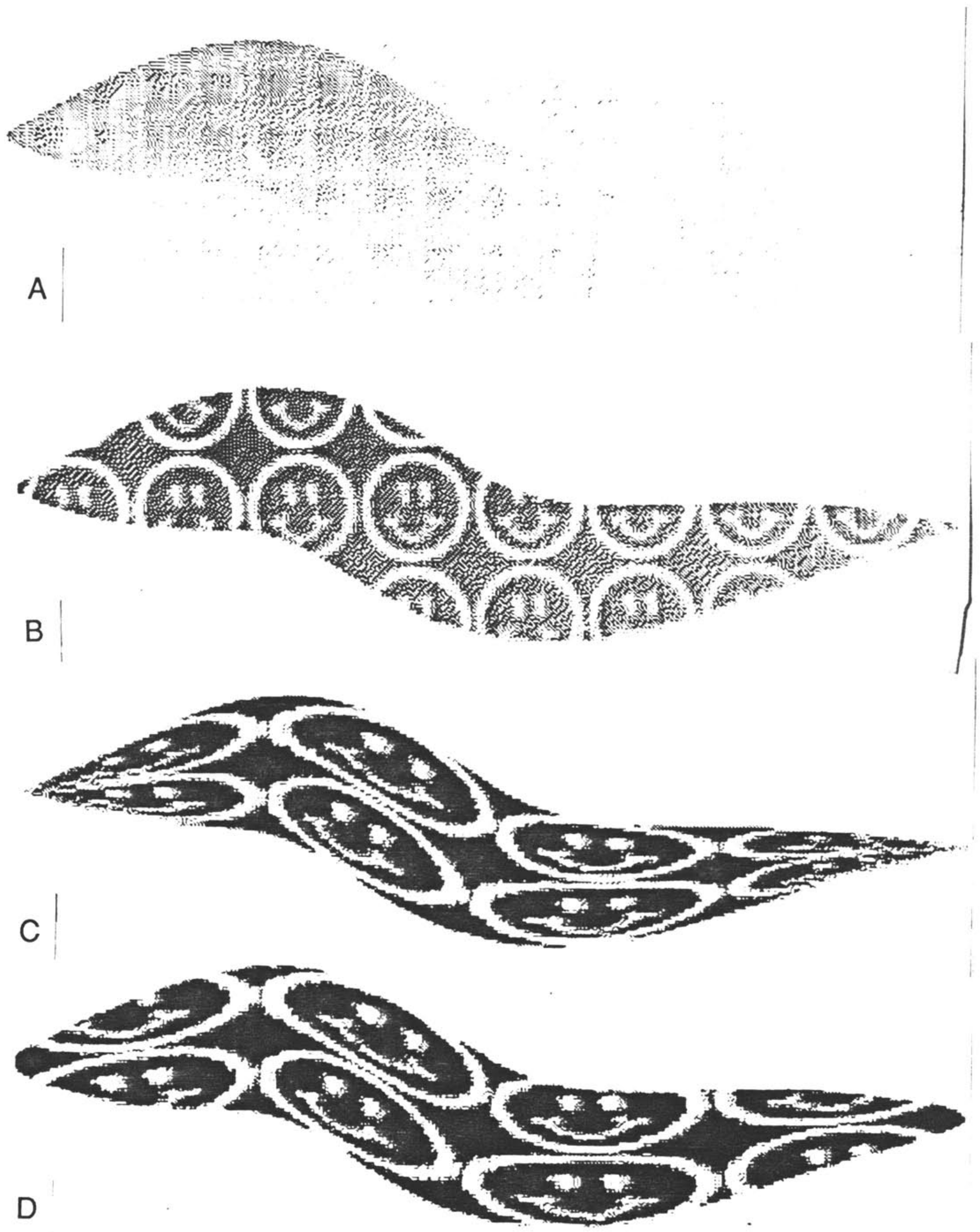


Figure 5.8: Texture mapping: (A) Textured paper (B) Textured by smiley-face paper (C) Texture mapping with spreading bristles (D) Texture mapping with pressure-threshold bristles.

Chapter 6

Animation

One of the most important motivations for this work is the hope that creating reproducible brush strokes will allow paintings to be animated. I have made some preliminary experiments, animating a few single strokes and a scene comprised of 17 strokes. The reader can see one of these animations (admittedly in a very small scale) by flipping the corners of this thesis.

6.1 Terminology

Due to the possibility of confusion, I try to use the following terminology consistently when discussing animation:

- *Motion.* In drawing a single stroke, the brush moves across the page. This can sometimes be confused with the motion of an animated figure.
  - *Brush Motion.* This is the act of drawing a single stroke. After several brush motions are completed you have a still painting.
  - *Stroke Motion.* This is the animation of a stroke as it changes during a movie. If one stroke represents a person's moustache, then stroke motion would be the motion of the moustache as the characters chews some food.
- *Structure.* In animating paintings, some new terms must be introduced to resolve the ambiguity in the concept of an "object".
  - *Elements.* Several strokes form an *element.* Higher order elements can be created by combining other elements.
  - *Configuration.* A stroke or an element retains its identity over several frames. In any given frame, it has a specific *configuration.* A stroke's configuration is the location and pressure of each control point. An element's configuration is a set of operations (usually a linear transformation) to be performed on all components of that element.

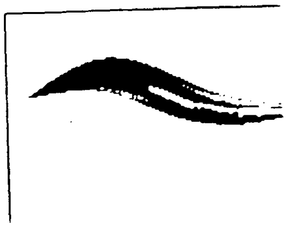
6.2 2D Keyframing

The code supporting animation right now is rather crude. It is basically a two-dimensional keyframing system written by myself which allows the user to specify the key configuations of a stroke. The position and pressure of each control point of the stroke is interpolated between the key frames, using a spline for non-periodic motion and a generalized sinusoid for periodic motion. The same brush and dip is used for any given animated stroke.

6.3 Test Animations

The tests consist of three 8-second (240 frame) animations recorded on an Ampex 1" videotape recorder. Each took a little under 1.5 minutes per frame (or 5-6 hours per test) to render. For each test, the same stroke was used, moving through the same configurations. All brushes were 140 bristles wide, covering a region approximately 500 by 100 pixels in area, and were rendered without anti-aliasing to save time. The following brush/dip pairs were used:

- A smoothly interpolated set of gray values (see Fig. 5.2 C). This was especially pretty, although it didn't look like it was drawn with a brush. Instead, it looked like a leaping salmon or excited flatworm.



- A partly dry brush with stochastic quantity-sharing rules (see Fig. 5.5 B). This didn't turn out so well, since the random number generator controlling the sharing of ink did not use the same seed from frame to frame. Hence, the raggedy trailing edge of the stroke flickers badly as the animation is played back.

  This was particularly disappointing, since I'm particularly fond of just such artifacts in hand-drawn animations using charcoal, chalk and other such media. An earlier very low-resolution animation, played back at about 4 frames per second actually looked far better, since its "chunkiness" supported the perception of dynamic charcoal. In the future, it would be best to try to ensure more frame-to-frame consistency of each element, and avoid rapid motion of elements with high spatial frequencies.

- A texture mapped brush (see Fig. 5.8 C). This resulted in a very appealing animation, partly since the pattern chosen (a smiley face) was humorously deformed as the stroke underwent squash and stretch.

- A randomly colored brush (see Fig. 5.5 B). The seed of the random number generator used to determine the evolution of the brush color along the stroke was reset at the beginning of each configuration, so that the color was consistent from frame to frame. This turned out to help make this one of the prettier strokes, although there was still some flickering moiré patterns due to the fact that the frames were not anti-aliased.

All of the tests demonstrate a plasticity markedly missing from conventional computer animation. Although the test strokes are non-representational, their smoothly flowing motions remind some viewers of shifting eyebrows, worms, or flexing muscles. These results are very encouraging for using brushes for rendering animals and other natural subjects.

6.4 Flip Animation

The reader can view an animated brush stroke by flipping through the images in the right hand margin of this thesis. The animation is one complete cycle of a 40 frame periodic motion similar to that used in the test animations. The brush is partly dry, and uses the stochastic quantity-sharing rules with the same random seed at the beginning of rendering each frame. This results in good frame-to frame consistency. The stroke has four control points sinusoidally interpolated between two configurations, where the phase of each control point along the stroke lags by $\frac{\pi}{8}$ behind the point to

its left.

6.5 The Animated Shrimp

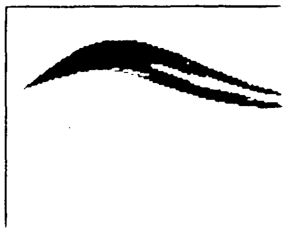
The shrimp shown in Fig. 2.1 was animated using a Sony write-once video disc recorder. The scene was derived from four key frames, spline interpolated over the total 92 frame sequence. Each frame took about one minute to render. With the antennae and legs waving around, the tail kicking, and the ripples flowing away from the leaf, the resulting animation is very lifelike.

Chapter 7

Further work:

7.1 Better Input Methods

Without an input device as expressive as a real brush, the current environment isn't very user friendly. There are many kinds of input devices offering three or more degrees of freedom which might be adapted for manually entering strokes. At the MIT Media Lab, we are exploring force sensitive touch-screens [6], LED-based body trackers [3], and magnetic pointing devices [9]. Other possible input devices include touch-sensitive tablets [2] and the drawing prism [4].



7.2 Better Rendering Hardware

With the advent of parallel computers, the drawing of the most sophisticated of strokes should be possible in real time. This is because almost all of the computation in the algorithms described here are local, that is, dependent only on an immediate neighborhood of bristles or pixels, and thus is well-suited to implementation on machines using parallel architectures.

7.3 Exploring Rules

More experimentation is needed to build a good-sized library of rules. Hopefully, subjective properties like "blotchiness", "dryness", or "clumpiness" can be controlled by adding a rule and setting a parameter or two. New kinds of rules will result in innovative brushes, as well as realistic models of traditional watercolor brushes.

7.4 Real Color

For simplicity, I stayed with monochrome ink even though the frame buffer I used has full 24 bit color. A useful extension would be to allow the user to experiment with a virtual brush laden with various colors. A more complicated rule would describe the behavior of paint mixing. Real electronic paint can change color as a function of thickness of application or chemical reaction with the brush, paper, or other strokes. Gooey paint drips and mounds up in ridges left by clumps of bristles.

7.5 Paper Effects

The wetness and absorptive properties of the ink or paper can be described by specifying the area of the paper covered by each bristle, and an ink redistribution function associated with the paper. The former corresponds to the pre-filtering and the latter to post-filtering steps in anti-aliasing. In addition to the usual blurring (low-pass filter) operations, one could use a simple asymmetrical fractal to simulate the little forked bleeding that capillary action sometimes causes on dry papers.

7.6 Splatter

A bit of splatter from a heavily-laden brush with stiff bristles pulled briskly around a corner might be represented as a rule which gets activated when the brush velocity or accelleration surpasses a certain threshold. It then places a fractal distribution of splattered, fuzzy dots on the paper as a function of the ink supply, trajectory, and pressure on the brush.

7.7 Music and Painting

An appealing analogy to the stroke is the contour of a musical note over time. Each stroke is a set of time-varying parameters (like position and pressure, or loudness and timbre). A cluster of strokes can evoke a recognizable image, much as a collection of notes create a chord or arpeggio.

Occasionally, when I am asked about the limits of realism in my simulation, I am reminded of similar questions asked of builders of electronic instruments. The answer, of course, is that there is room in electronic media for both accurate reproduction of physical phenomena, and for creative exploration with totally new forms of expression which take advantage of the differences inherent in the new media.

7.8 3D Strokes

Perhaps the strokes themselves can be liberated from the 2D quality of paper, and a technology of 3D paintbrushes can be realized. Non-computer techniques come to mind, including sweeping a lit taper through a room to leave a trail of smoke, or "drawing" in an aquarium filled with a viscous gel using a long hypodermic filled with ink. All the issues of describing the evolution of texture through the stroke remain. With stereoscopic displays

[9] or computer-generated holograms [1], one will be able to create tenuous sculptures far more delicate than currently possible. One could even imagine folding translucent paper into origami shapes which define plane fragments on which these brush strokes lie.

Chapter 8

Conclusion

To get realistic brush strokes, one must simulate the phenomena which gives rise to them. The tough part is modeling the behavior of individual bristles, which we do with rules which execute each time the brush moves. To produce an anti-aliased image, the brush's image is drawn onto a small patch which is sampled and incrementally copied to the page.

When an animal, plant, or river can be represented by a few deft strokes, perhaps under some circumstances a brush representation can replace a polyhedral one. Whereas polyhedra are good representations of analytic objects, and polygonal fractals are good for largely amorphous ones, there is a middle classification of things too rich for one and too structured for the other. Given the computational complexity and storage expense of representing a warm, fuzzy bunny as a skeleton of faceted polyhedra covered with skin polyhedra and particle generated hair, perhaps representing it instead as a collection of brush strokes would result in faster rendering, more compact storage, and a more aesthetically appealing image.

Bibliography

[1] Benton, Stephen A., "Holographic Displays — A Review," *Optical Engineering* **14,** 5, Sept-Oct 1975.

[2] Buxton, W., Hill, R., and Rowley, P., "Issues and Techniques in Touch-Sensitive Tablet Input," *Computer Graphics* **19,** 3, July 1985, pp. 215-224.

[3] Ginsberg, C., and Maxwell, D., "Graphical Marionette," *Proc. ACM SIGGRAPH/SIGART Workshop on Motion*, April 1983, pp. 172-179.

[4] Greene, Richard, "The Drawing Prism: A Versatile Graphic Input Device," *Computer Graphics* **19,** 3, July 1985, pp. 103-110.

[5] Lewis, John-Peter, "Texture Synthesis for Digital Painting," *Computer Graphics* **18,** 3, July 1984, pp. 245-251.

[6] Minsky, Margaret, "Manipulating Simulated Objects with Real-world

Gestures using a Force and Position Sensitive Screen," *Computer Graphics* **18,** 3, July 1984, pp. 195-203.

[7] Rogers, David F., "Procedural Elements for Computer Graphics," McGraw-Hill, New York, 1985.

[8] Sato, Shozo, "The Art of *Sumi-e*" (in English), Kodansha International, Tokyo, 1984.

[9] Schmandt, Chris, "Spatial Input/Display Correspondence in a Stereoscopic Computer Graphic Workstation," *Computer Graphics* **17,** 3, July 1983, pp. 253-257.

[10] Karl Sims, personal communication.

[11] Strassmann, Steve "Hairy Brushes," To appear in *Computer Graphics* **20,** 3, July 1986.

[12] Whitted, Turner, "Anti-aliased Line Drawing Using Brush Extrusion," *Computer Graphics* **17,** 3, July 1983, pp. 151-156.

[13] Wolfram, Stephen, "Cellular Automata as Models of Complexity," *Nature* **311,** 4, 1984, pp. 419-424.

Appendix A

How to Use the System

The prototype system is not intended to be a general-purpose paint program. As such, it lacks most of the useful functionality and well-designed user interface users have come to expect from paint programs. Instead, effort was concentrated on exploring new rendering algorithms and effects. I expect the functionality of painting with a hairy brush to be incorporated in well-designed paint programs, an issue which is beyond the scope of this thesis.

It is important to realize that the rendering algorithm is useful not only to users of interactive systems, but also to artists who wish to let the computer derive strokes algorithmically. For example, one could animate a stand of bamboo swaying in the breeze by appropriately perturbing the location of the strokes composing a painting of such a stand.

A.1 System specifics

The system runs on a Symbolics 3600 Lisp Machine with a $1280 \times 1024 \times 24$ bit frame buffer, configured for 8-bit per pixel grayscale. All of the code is written in Zetalisp using *flavors*, using an object-oriented programming style. The rules which govern the evolution of the brush are pieces of Lisp code ("methods" in Zetalisp) associated with a particular flavor of brush. These rules are executed as the brush moves along its path, and the code in them can freely refer to and modify any parameters.

For machines with color hardware, one can choose one of two different resolutions, and either 8 or 24 bits per pixel. 8 is preferable, since most drawing operations are faster. Since the program uses only 256 shades of gray, there is no advantage to using 24 bits per pixel.

A.2 Starting up

To start up the program, log into a Lisp Machine at the MIT Media Laboratory, and execute

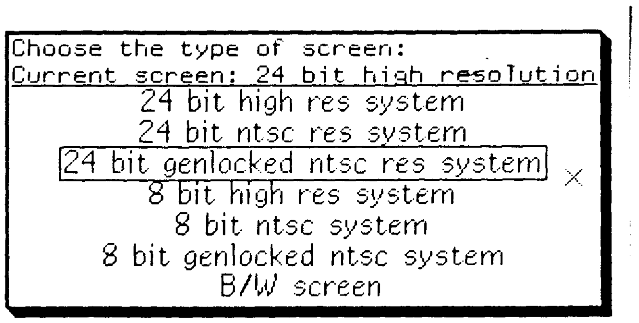
```
(make-system 'sumi)
```

This will load all the necessary files. To get to the top-level menu, type:

```
(sumi)
```

You will be prompted to specify what kind of window you want to use the first time you do this. Afterwards, it will re-use the same window. If you change your mind later and want select a new kind of window, you can get back this menu by typing:

```
 (sumi t)
```



```
Choose the type of screen:
Current screen: 24 bit high resolution
        24 bit high res system
        24 bit ntsc res system
     24 bit genlocked ntsc res system
        8 bit high res system
          8 bit ntsc system
       8 bit genlocked ntsc system
             B/W screen
```

Figure A.1: Specifying what kind of window to use.

For machines with no color hardware, the *sumi* system will work fine on the black and white console, except all grey values get rounded to white or black (sort of like working with high-contrast film).

A.3 The command menu

Once the type of window has been specified, the command menu is displayed [Fig. A.2].

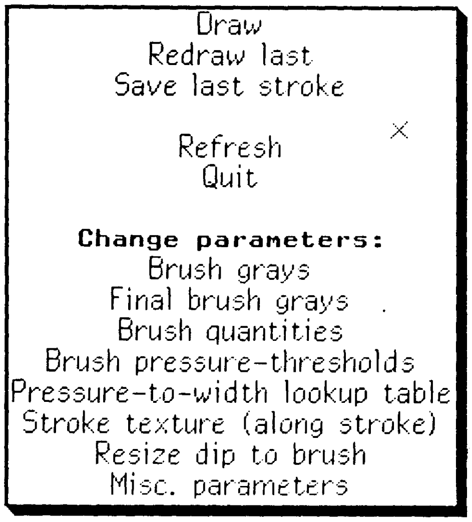


Figure A.2: The command menu.

A.3.1 Drawing commands

- `Draw`. This allows the user to specify a new stroke. See Sect. A.4.

- `Redraw last`. After changing the brush, the most recently drawn stroke is erased and redrawn using the new parameters.

- `Save last stroke`. The stroke and the brush used to draw it are stored in a buffer in user-readable form.

- `Refresh`. Erases the screen. If a paper texture is specified, it is drawn on the screen.

- `Quit`. Quits the program.

A.3.2 Vector parameter editing

The rest of the commands allow the user to change attributes of the system. Some of them use a histogram-editor window, since they edit parameters whose values are vectors.

By drawing with the mouse, an arbitrary mapping of, say, color to the brush can be defined. For example, the graph of brush color in [Fig. A.3] would result in a roughly dark stroke with a skunk-like white stripe down the middle. One useful feature is that the number of samples in the histogram need not match the number of bristles in the brush. Many-to-one and one-to-many mappings are defined so that any bristle which falls in between two samples gets a value which is the linear interpolation of those two samples. This is important, so that any one dip will work with a brush of any size.

Histogram-based parameters controllable by the user are

- `Brush grays`. Specifies the color distribution on the brush at the start of the stroke.

- `Final brush grays`. If path interpolation (Sect. 5.2) is activated, this is the "destination" color distribution.

- `Brush quantities`. Values less than 1 "short-change" a given bristle, causing it to run out of ink early in the stroke.

- `Brush pressure-thresholds`. Sets the minimum pressure necessary for a bristle to be in contact with the paper.

- `Pressure-to-width lookup table`. Defines an arbitrary mapping of pressure to stroke width.

- `Stroke texture`. Defines a periodic one-dimensional texture to map onto the stroke.

- `Resize dip to brush`. Forces the current dip to have as many data samples as the current brush has bristles.

A.3.3 Misc. parameter editing

Finally, miscellaneous other parameters can be changed by selecting the `Misc. parameters.` command [Fig. A.4].

A.4 Drawing

The prototype system takes up to a minute or two to render a stroke, depending on brush complexity and the super-sampling ratio for anti-aliased strokes. Although this is too slow for real-time interactive drawing, the user enters and edits the strokes' paths interactively using a mouse. The input consists of discrete samples of position and pressure, which are then smoothed using a cubic spline by the rendering algorithm.

After clicking on the `Draw` menu item, the user uses the mouse to position the points of the stroke with one hand, while the other hand specifies the pressure with the keyboard.

A.4.1 Pressure specification

The Symbolics 3600 keyboard has two redundant sets of 4 modifier keys (known as "bucky keys" or "buckies"): `CONTROL`, `META`, `SUPER`, and `HYPER`. On the left side of the keyboard, these are ordered right to left; this is reversed on the right side. The boolean values determined by the up- or downness of these keys are called "bucky bits".

The user can dynamically specify 16 different levels of pressure by selectively pressing combinations of the buckies. The cursor responds to this in real time by growing or shrinking accordingly, depending on the pressure-to-width algorithm selected by the user.

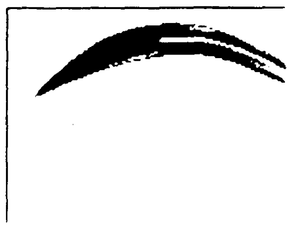
A.4.2 Position specification

Position is specified with the mouse. The Symbolics mouse has 3 buttons, identified as `left`, `middle`, and `right`. To specify control points, the user moves the mouse to the desired position (while choosing the desired pressure with the buckies), and then selects that point by clicking `left` and releasing. Clicking `right` completes input and starts rendering the stroke. Clicking `middle` aborts the input so far and returns the user to the command menu.

There are two different modes of quick user feedback. The stroke drawn by both is identical, but users sometime prefer one style over the other.

- `Dots`. For each data point, a solid dot of appropriate radius for the pressure is drawn on the screen at that point. The user can visualize the final stroke as "connecting the dots."

- `Polygons`. As the user specifies more and more of the stroke, a series of solid black polygons approximating the stroke are drawn on the page. The last two polygons are "rubber-band" objects; that is, they follow the mouse around on the screen, and get fatter or slimmer



according to the specified pressure.

Both the dots and the polygons are drawn in *exclusive-or* mode so they can be rapidly erased when input is terminated.

Both modes were chosen for their speed of drawing and depiction of the input specified so far. Hopefully, as faster rendering hardware becomes available, such coarse feedback can be replaced by real-time rendering of the final stroke.

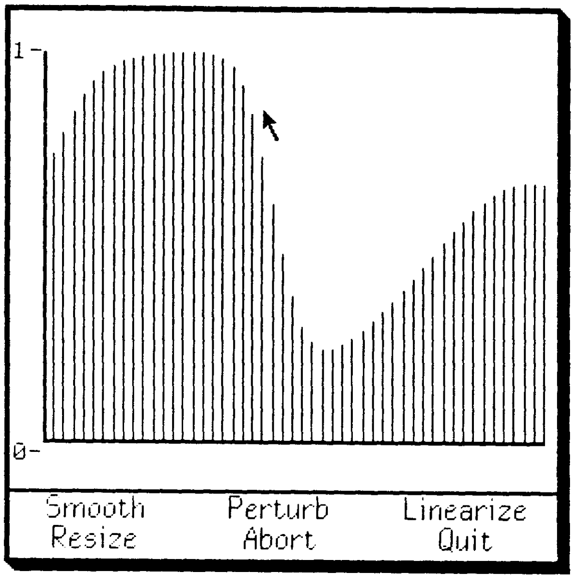
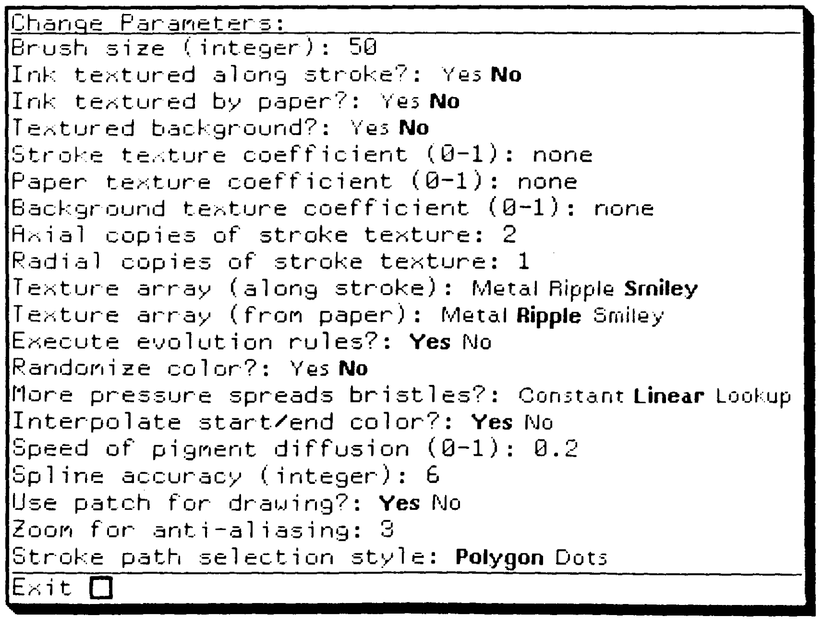


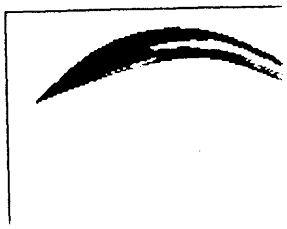
Figure A.3: A histogram-editing window. On the **Y** axis, Black=1 and White=0. The **X** axis refers to the various bristles' positions.



```
Change Parameters:
Brush size (integer): 50
Ink textured along stroke?: Yes No
Ink textured by paper?: Yes No
Textured background?: Yes No
Stroke texture coefficient (0-1): none
Paper texture coefficient (0-1): none
Background texture coefficient (0-1): none
Axial copies of stroke texture: 2
Radial copies of stroke texture: 1
Texture array (along stroke): Metal Ripple Smiley
Texture array (from paper): Metal Ripple Smiley
Execute evolution rules?: Yes No
Randomize color?: Yes No
More pressure spreads bristles?: Constant Linear Lookup
Interpolate start/end color?: Yes No
Speed of pigment diffusion (0-1): 0.2
Spline accuracy (integer): 6
Use patch for drawing?: Yes No
Zoom for anti-aliasing: 3
Stroke path selection style: Polygon Dots
Exit □
```

Selected options (shown in bold): Ink textured along stroke: **No**; Ink textured by paper: **No**; Textured background: **No**; Texture array (along stroke): **Smiley**; Texture array (from paper): **Ripple**; Execute evolution rules: **Yes**; Randomize color: **No**; More pressure spreads bristles: **Linear**; Interpolate start/end color: **Yes**; Use patch for drawing: **Yes**; Stroke path selection style: **Polygon**.

Figure A.4: Editing miscellaneous parameters.



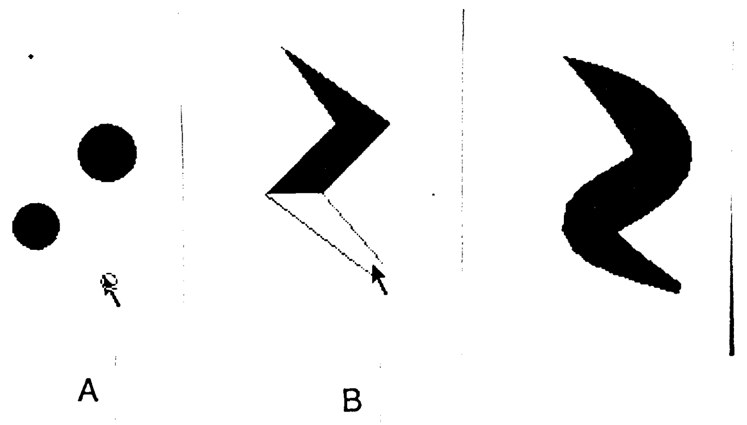


Figure A.5: Two modes of input resulting in the same splined stroke: (A) Dots (B) Polygons.

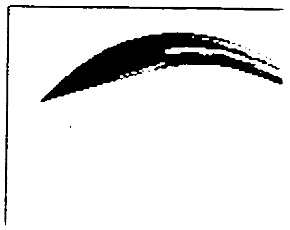
Appendix B

Details of the 2D Brush

This is a description of the algorithm used in the first implementation.

For anti-aliased images, the brush is drawn on a virtual screen with a resolution R times higher than the frame buffer in both axes. All dimensions given below are in virtual coordinates unless otherwise specified. Frame buffer coordinates are referred to as "real" coordinates. The origin is in the upper-left corner of the screen, with **x** increasing toward the right, and **y** increasing toward the bottom. For simplicity, it is assumed that the real and virtual origins coincide.

1. Specify the stroke path. This is represented as the n points $(x, y, p, s)_i$ for $i = 0 \rightarrow (n - 1)$. The values x, y, and p represent position and



pressure. s is an approximation of the distance traveled along the curve, where $s_0 = 0$. The brush's center moves along the line segments connecting consecutive points $(x, y)_i$, as computed by Bresenham's algorithm for drawing line segments. This guarantees that the brush doesn't skip over any locations.

- On a system with no continuous pressure-sensitive input device, the path may be derived from a cubic spline with N control points. Each point is a triple $(X, Y, P)_j$ specifying location and pressure at the j^{th} point. A fourth value approximating the distance along the curve, S_j, is computed for that point.

$$S_0 = 0, S_j = \sum_{k=1}^{j} \sqrt{(X_k - X_{k-1})^2 + (Y_k - Y_{k-1})^2}$$

- From this, two 2D cubic splines are created. One comes from $(X, Y)_j$ and generates $(x, y)_i$, the other from $(P, S)_j$ and generates $(p, s)_i$.

2. Select a brush. Determine the brush's bounding box $(b_w \times b_h)$; this is the maximum width and height of the brush's image as it moves through the stroke. For example, if the size of the brush's image is proportional to the pressure, compute the bounding box at the pres-

sure $p = \max(p_i)$. If the brush is asymmetrical and rotates during the stroke, the box must contain the outermost bristles at all angles of rotation.

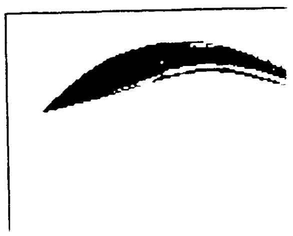
3. Dip the brush into a dip. The dip restores the brush and its bristles to their normal configuration.

   - If the dip maintains any or all of the brush's parameters explicitly, these are copied over to the brush. A list of bristle parameters, if any, is used to set the corresponding values in the bristles.
   - If the dip has any rules governing setting or modifying any of the brush's or its bristles' parameters, they execute.

4. Allocate the patch onto which the brush will draw the high-resolution image. The patch is a 2D array whose dimensions ($p_w \times p_h$) satisfy two conditions:

   (a) Both p_w and p_h are multiples of the super-sampling ratio R. Thus the patch will always map onto a $\frac{p_w}{R} \times \frac{p_h}{R}$ region of the frame buffer.

   (b) The patch must be large enough to contain the bounding box of the brush, with room to spare if the box's edges lie on a fraction



of a real coordinate.

A typical example of this problem is shown in [Fig. B.1]. For this example, $R = 4, b_w = 6$, and $b_h = 8$. To completely contain the brush, the patch's dimensions must be $p_w = 12, p_h = 12$.

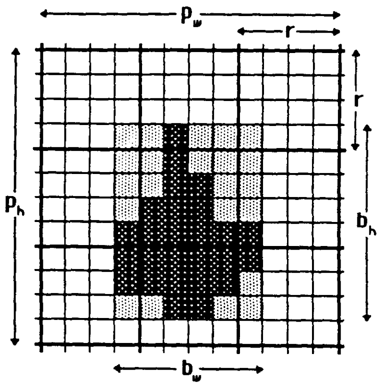


Figure B.1: A 6×8 pixel brush requires a 12×12 patch.

To satisfy these two requirements, the dimensions of the patch are:

$$p_w = \left(\left\lceil \frac{b_w}{R} \right\rceil + 1\right) R, \quad p_h = \left(\left\lceil \frac{b_h}{R} \right\rceil + 1\right) R$$

As the brush moves across the virtual sheet of paper, the upper-left corner of the bounding box remains fixed relative to the brush's center, no matter how small or large the image gets. The following notation is used:

b_pos: The center of the brush. This moves through the trajectory spec-

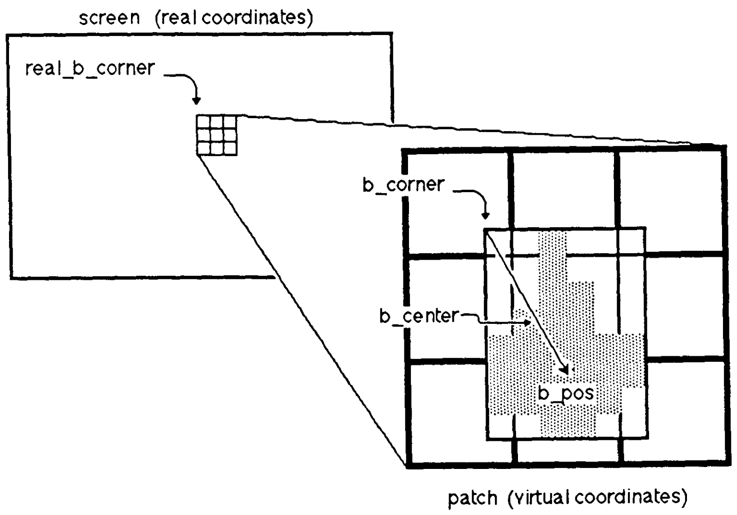


Figure B.2: Positioning the brush from [Fig. B.1].

ified in step 1 above.

b_corner: The upper-left corner of the bounding box of the brush's image.

b_center: This stays constant throughout the stroke. Equals $(b\_pos - b\_corner)$.

real_b_corner: The pixel (in real coordinates) which corresponds to *b_corner*.

$$real\_b\_corner_x = \left\lfloor \frac{b\_corner_x}{R} \right\rfloor, \quad real\_b\_corner_y = \left\lfloor \frac{b\_corner_y}{R} \right\rfloor$$

5. For each position b_center_i, compute the displacement

$$\Delta real\_b\_corner = real\_b\_corner_i - real\_b\_corner_{i-1}$$

- If $\Delta real\_b\_corner = (0,0)$, the brush's image has not crossed a real pixel boundary. No special action is necessary.
- If $\Delta real\_b\_corner \neq (0,0)$, the brush's image has crossed a real pixel boundary. Since the path was generated using Bresenham's

algorithm, the magnitude of the displacement is guaranteed to be at most one pixel in the **x** and/or **y** direction. The information contained in one real row and/or column (i.e. R virtual rows or columns) in the patch must be written to the screen. For each $R \times R$ region, the average gray shade is computed and written to the corresponding screen pixel. The pixels of the newly entered row and/or column may now be copied into the patch, by repeating each value to cover the corresponding $R \times R$ area of the patch.

Rather than move the contents of the patch in the appropriate direction by copying all the values, the pointer *patch_origin* is updated to reflect the fact that the "upper-left corner" of the patch may actually lie somewhere inside the patch array [Fig. B.3]. *patch_origin* is a vector in real (screen) units which describes how much the image is wrapped around on the patch. Therefore, to access an element of the patch array which corresponds to the virtual location (X, Y), use

$$\begin{aligned} patch\_array_x &= (X + (R)(patch\_origin_x - real\_b\_corner_x)) \bmod p_w \\ patch\_array_y &= (Y + (R)(patch\_origin_y - real\_b\_corner_y)) \bmod p_h. \end{aligned}$$

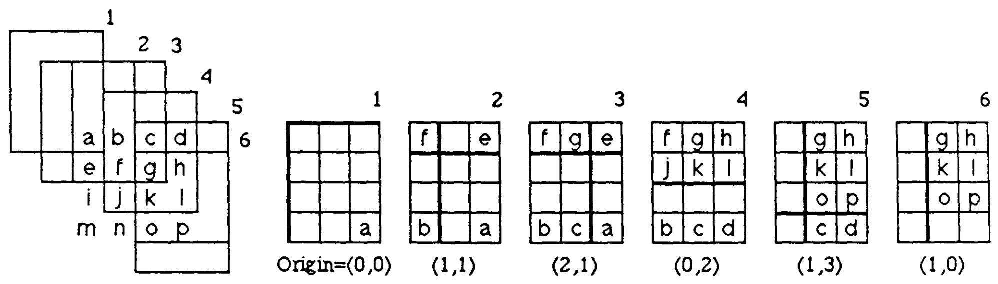


Figure B.3: Scrolling the patch as it crosses the screen.

Before the very first part of the stroke can be written, the patch array is initialized with a portion of the image on the screen. For each pixel of the $\frac{p_w}{R} \times \frac{p_h}{R}$ region of the screen, write its value onto the corresponding $R \times R$ region of the patch array.

6. The brush is informed of its new location, pressure, and an estimate of how far it has travelled so far, i.e. $(x, y, p, s)_i$. It may now execute any rules it has, which in turn may affect the position, color, or any other property of the bristles.

7. Each bristle does the following:

   - If it has any special rules, they get a chance to run. This includes perturbing its pressure, ink color, ink supply, etc.

   - If the ink supply is not empty, and the pressure on this particular bristle is greater than the bristle's pressure-threshold, it places its image onto the patch at the point corresponding to the bristle's virtual location (X, Y). This is the point $(patch\_array_x, patch\_array_y)$ on the patch which we derived above. The bristle's image is a single pixel whose color is computed as follows:

The color of the bristle's ink, the color already on the corresponding patch cell, and the value from the texture array (if any) are passed to the color combination function. By default, if the ink (scaled by the texture value) is darker that what was there before, then that is the new value. Otherwise the patch is left alone. The user may supply an arbitrary color combination function.

8. Although the brush and bristles can freely refer to the paper's texture and other properties while executing rules in the above steps, the paper is consulted one last time. Things like blurring, clumping or growing hairlines to simulate seepage may now take place. If the brush has been moving at a high velocity, a spattered texture may be drawn.

9. Steps 5-8 are repeated through the stroke. After the last iteration, the entire patch-array must be written to the frame buffer, using the same algorithm as in step 5 above.