A Survey of Stroke-Based Rendering

Aaron Hertzmann
University of Toronto

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

This tutorial describes several stroke-based rendering (SBR) algorithms. SBR is an automatic approach to creating nonphotorealistic imagery by placing discrete elements such as paint strokes or stipples.

Figure 1 shows an SBR algorithm in action. Starting from a photograph, a collection of brush strokes are placed in a manner that matches the original photograph, and then rendered to have the appearance of an oil painting.[1,2]

Although the details vary, all SBR algorithms create images by placing strokes according to some goals. The most common goal is making the painting look like some other image—for example, in Figure 1, I wanted to place colored brush strokes to look like the picture of the mountain. Another important goal is to limit the number of strokes in some way so that the result will look like a painting. Otherwise, the algorithm could use many tiny brushstrokes, producing a good match to the source image without much abstraction, but the result won't look like a painting.

Finally, once the algorithm places the strokes, it can render them in some other form. In Figure 1, the algorithm didn't add texture until after placing the brush strokes. It compared the source photo to some intermediate image with a simplified stroke model. This is both for efficiency and aesthetic reasons. The main point is that the final rendering may differ from the way we expressed our goals about the image.

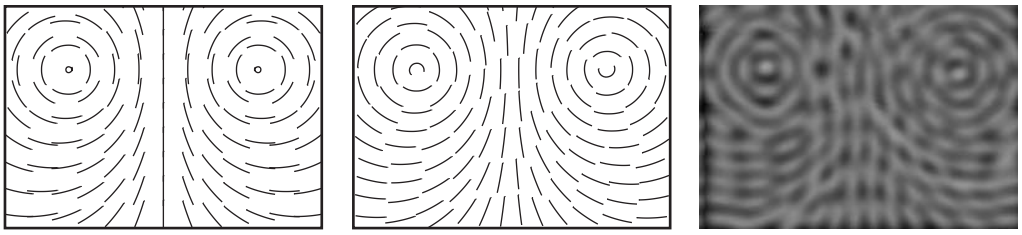
Figure 2 shows another example of an automatic vector field visualization.[3] Here, streamlines effectively convey the vector field's motion. To clearly illustrate the vector field, the placements should be placed evenly—Figure 2b was created with the goal of making the blurry version as close to a constant gray value as possible. For comparison, Figure 2a shows stroke placements on a regular grid without adjustment. Again, we see that this streamline visualization algorithm is an SBR algorithm: it places strokes (streamlines) according to specified goals (to follow the vector field and to match a target tone in the blurred image).

Usually it's not possible to exactly meet all the goals. Hence, it's useful to have a way of trading off the goals and quantifying their importance. We can do this by formalizing an SBR problem as an *objective function* minimization problem. An objective function is a mathematical formula that measures how good a rendering is; SBR algorithms attempt to minimize objective functions. For example, it isn't possible to place the streamlines in Figure 2 to achieve a purely constant tone in the gray image. Instead, we can use as an objective function the deviation of the blurred image from a constant image.

So far, I've described two different SBR problem statements—one for painterly rendering and one for visualization—but said nothing of how to design algorithms

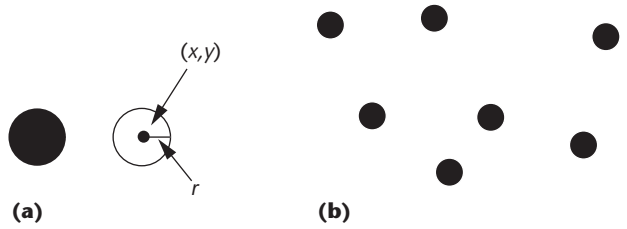
1 Results of a stroke-based algorithm: (a) source photo, (b) painted version, and (c) final rendering.

Published by the IEEE Computer Society



Courtesy of Greg Turk and David Banks

2 Example of an automatic vector-field visualization: (a) vector field, (b) final rendering, and (c) blurred rendering.



3 (a) Detail of a stippling stroke model. (b) Individual strokes (stipples).

for these problems. Two main approaches to designing SBR algorithms exist. *Greedy algorithms* greedily place the strokes to match the target goals. *Optimization algorithms* iteratively place and then adjust stroke positions to minimize the objective function. A greedy algorithm produced Figure 1, and an optimization algorithm produced Figure 2. This is a somewhat unusual view of these algorithms. I have sacrificed chronological ordering in this tutorial in order to present algorithms in this way.

Haeberli introduced both a semiautomatic greedy algorithm and an automatic optimization algorithm in a seminal paper.[4] The subsequent pen-and-ink algorithms developed together by Michael Salisbury, David Salesin, Georges Winkenbach, and others demonstrated the potential of highly automated SBR to create beautiful and expressive imagery.[5-8] Digital paint systems had previously automated some aspects of the stroke renderings, but didn't automate any stroke placement choices.

Although this tutorial focuses on the technical details of SBR algorithms, it's important to remember that they're useless without human control. Every aspect of the system (including the choice of stroke models, the setting of weight parameters, and the selection of input imagery) requires aesthetic decisions that only an artist, working toward some goal, can make. Ideally, a human artist using the system should have total control over the decisions made. For example, a user should be able to specify spatially varying styles to use, so that different rendering styles are used in different parts of the image, or to specify positions of individual strokes. However, one of the great advances in art in the age of digital machines is the ability to create complex systems of procedural art, where the artist doesn't directly create the final work, but rather creates rules according to which final decisions are made (although simpler procedural works—such as those of John Cage—exist without computers). Hence, an artist might design the energy function, but not necessarily edit every individual image produced by the algorithm. In one possible scenario, the artwork's creation might occur at a time after the artist's involvement. The main goal of SBR algorithms is to provide procedural tools that automate parts of the image creation process, not to replace the artist.

Stroke-based rendering

Let's begin with a few definitions. First, we need to define what our *strokes* can look like. A stroke is a data structure that can be rendered in the image plane. A *stroke model* is a parametric description of strokes, so that different parameter settings produce different stroke positions and appearances.

For example, one form of stippling uses a simple stroke model (see Figure 3). A stipple is a stroke that can be described with two parameters: the (x, y) position of the stipple in an image and the radius r of the stipple. (As we shall see in the "Voronoi algorithms" section, other definitions of a stipple are possible.)

We create images by combining strokes into an *image structure*. An image structure is a data structure containing

- a canvas, defined by a background color or texture, and
- an ordered list of strokes, defined by their parameter settings.

To create an image, the list of strokes is rendered by alpha-compositing over the background. The background is usually just a solid color or a predefined texture image. For example, a PostScript file containing only line art is an image structure because it contains a list of stroke definitions; the data in the file can be rendered on the screen or on a printer.

Finally, we use an *SBR energy function* to quantitatively evaluate how good a rendering is. An SBR energy function is a function $E : I \rightarrow R$, where I is the set of possible image structures and R is the set of real numbers. An energy function $E(I)$ takes an image structure as input and outputs a number indicating the quality of the image—generally, the goal of an SBR algorithm is to produce an image with the smallest possible energy. The energy function is sometimes also called an *objective*, *cost*, or *error function*. The term *energy* comes from analogous uses in physics, such as searching for the minimum energy configuration of a set of particles.

SBR algorithms are normally defined in terms of some input data, usually an input image. In most algorithms described here, the energy function measures how

closely the rendering matches some input image. Additionally, the energy function encodes trade-offs. For example, a painterly rendering algorithm takes an input image and produces an image structure containing color paint strokes that matches the source image. However, you could get a perfect match to the source image by placing thousands of tiny brush strokes, which would look nearly identical to the source image. You can create a more interesting painting by adding an abstraction term to the energy function—that is, by assigning higher energies to paintings that use less strokes. For example, the energy function could be

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

where w_{abs} is a scalar weight parameter, and $E_{\text{match}}(I)$ is the sum-of-squared differences between the source image and the rendering. This energy function has one parameter, w_{abs}. You can control the level of abstraction in the painting style by adjusting parameter values: setting w_{abs} to be small specifies that we want a realistic style (reproducing the original image as close as possible); setting w_{abs} to be large specifies an abstract style (capturing the image with few strokes). We can then define an *SBR style* as a stroke model and energy function (including parameter settings) for image structures.

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

Optimization algorithms

Two kinds of optimization algorithms have been applied to SBR. The first kind, which I'll call *Voronoi algorithms*, exploits special properties of the SBR problem to perform efficient global update steps. The second kind, which I'll call *trial-and-error algorithms*, assumes no special structure and performs heuristically chosen tests to try to reduce the energy. Generally the Voronoi algorithms are effective and fast, but we can't apply them to all problems. The trial-and-error algorithms are general-purpose, but at the cost of substantial computation times. (Both of these approaches have also been called *relaxation algorithms*.)

Voronoi algorithms

Voronoi algorithms are useful for SBR problems where the final image will contain many identical nonoverlapping strokes and where only stroke density is constrained. The central idea is to use efficient techniques from computational geometry to place evenly spaced strokes into an image. Moreover, we can make these techniques fast using graphics hardware. However, these algorithms don't directly optimize with respect to an image-based metric (that is, where the rendering should match target tones) since the energy function is defined in terms of stroke densities.

Lloyd's method. How can you create a set of evenly spaced points within an image? By using an iterative optimization procedure, which requires defining an appropriate energy function. Let $\mathrm{p} = (x, y)$ be pixel locations in an image, and let C_i be special point locations called centroids; the strokes will eventually be placed at these locations. Let $L_{\mathrm{p}}^i \in \{0,1\}$ be a binary labeling of pixels: if $L_{\mathrm{p}}^i = 1$, then the pixel p has been assigned to centroid i. Every pixel is assigned to exactly one centroid:

$$\sum_{\mathrm{p}} L_{\mathrm{p}}^i = 1$$

The goal is to choose both a set of centroids and a labeling that minimizes the energy function:

$$E(I) = \sum_{\mathrm{p}\in I} L_{\mathrm{p}}^i \left\| \mathrm{p} - \mathrm{C}_i \right\|^2$$
$$= \sum L_{\mathrm{p}}^i ((\mathrm{p}_x - \mathrm{C}_x)^2 + (\mathrm{p}_y - \mathrm{C}_y)^2)$$

where the centroids and labeling are implicitly members of I. (Nonphotorealistic rendering researchers have typically presented the continuous version of this energy. I prefer the discrete version because it more closely reflects the problem actually being solved. One benefit is that we can prove convergence of the discrete version of the algorithm, whereas convergence hasn't been proven for the continuous version.) In short, we want every pixel to be close to its assigned centroid. If we knew the set of centroids C_i in advance, then computing the optimal labeling would be easy—we just assign every pixel to the nearest centroid. The resulting labeling is known as a Voronoi diagram (see Figure 4a)—it partitions the plane according to which centroids C_i are nearest to each point.

From looking at Figure 4a, it should be clear that picking some randomly chosen point set and then computing the Voronoi diagram doesn't give a good arrangement of centroids. In fact, we can improve upon this set of centroids by adjusting the point centers to best fit this partition—in other words, by holding fixed the labeling and optimizing the energy function with respect to the centroids. The new optimal centroids are given by

$$\mathrm{C}_i = \sum_{\mathrm{p}} L_{\mathrm{p}}^i \mathrm{p} \,/\, \sum_{\mathrm{p}} L_{\mathrm{p}}^i$$

This is just the mean of the pixel locations that are assigned to C_i; this formula is easily obtained by setting $\partial E(I)/\partial \mathrm{C}_i = 0$ and solving for C_i. We can iterate these two steps, which is known as Lloyd's method:

```
function LLOYDSMETHOD(n, I):
    initialize the centroids C_i by randomly sampling n
        points uniformly in the image I
    while the algorithm has not converged
        reestimate the labeling by
```

$$L_p^i \leftarrow \begin{cases} 1 & i = \arg\min_i \|p - C_i\|^2 \\ 0 & \text{otherwise} \end{cases}$$

reestimate the centroids by

$$C_i \leftarrow \sum_p L_p^i p \Big/ \sum_p L_p^i$$

return the centroids C_i

Figure 4b shows the same points after applying Lloyd's method. The algorithm converges when the energy doesn't change between steps. However, the algorithm is guaranteed to reduce the energy at every step before convergence because each step minimizes the energy with respect to some parameters. As a result, the algorithm is guaranteed to converge because the energy decreases at every step before convergence and because there's a finite number of possible labelings L. Lloyd's method was discovered separately by the signal-processing community (where it is known as *vector quantization*) and the machine learning community (where it is known as *k-means clustering*).

Running this optimization in software over an entire image can be quite slow. However, some researchers[9,10] have used graphics hardware to make the process fast.

Variations on Lloyd's method. Now that we have a procedure for regular placement of points, we can easily design SBR algorithms on top of it. Perhaps the simplest SBR problem to describe is stippling. Deussen et al.[11] presented the first such method, using stipples to approximate gray tones in a target image (see Figure 5). In their method, they manually segment the image into distinct regions. They place stipples evenly within each region, by applying Lloyd's method to each region separately. The centroids are initialized using a half-toning algorithm. Once the centroid locations are chosen by Lloyd's method, they're replaced with stipples. The size of each stipple is set proportional to the gray level of the image underneath it.

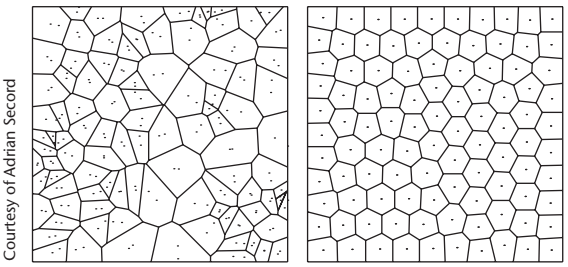
Secord[12] presents an alternate stippling style and algorithm, by varying the dot spacing instead of the dot size (see Figure 5d). The idea is to define a spatially varying density function $\kappa(p)$ that determines how dense the stippling should be in different parts of the image. This density function is directly derived from the tones of the target image $T(p)$, that is, $\kappa(p) = 1 - T(p)/m$, where m is the max gray level in T. The new energy function is

$$E(I) = \sum_{p \in I} L_p^i \kappa(p) \left\| p - C_i \right\|^2$$

$$= \sum_{p \in I} L_p^i \kappa(p)((p_x - C_{ix})^2 + (p_x - C_{iy})^2$$

Following the same steps presented previously directly leads to a slightly different version of Lloyd's method. The labeling step is the same, but the centroids are now reestimated as

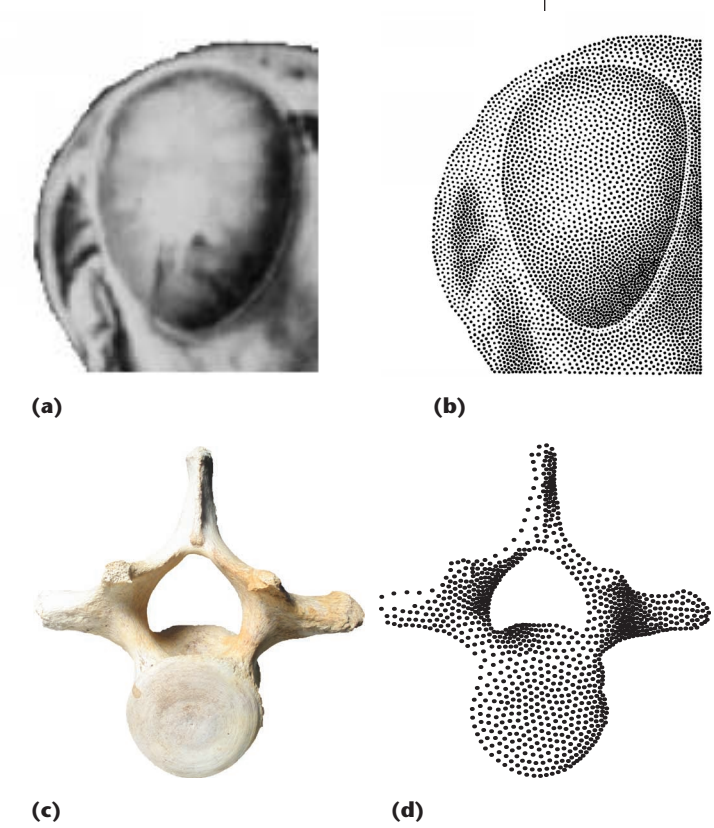
$$C_i \leftarrow \sum_p L_p^i \kappa(p) p \Big/ \sum_p L_p^i \kappa(p)$$



Courtesy of Adrian Secord

(a) (b)

4 (a) Voronoi diagram of a set of points C_i. The image is partitioned into a set of regions, one region for each point. Each region contains all pixels that are closest to the corresponding point C_i. (b) By applying Lloyd's method, the points are adjusted so that they lie at the centroid of their region of the Voronoi diagram. The resulting point set is evenly spaced and can be used to specify regular stroke placements.



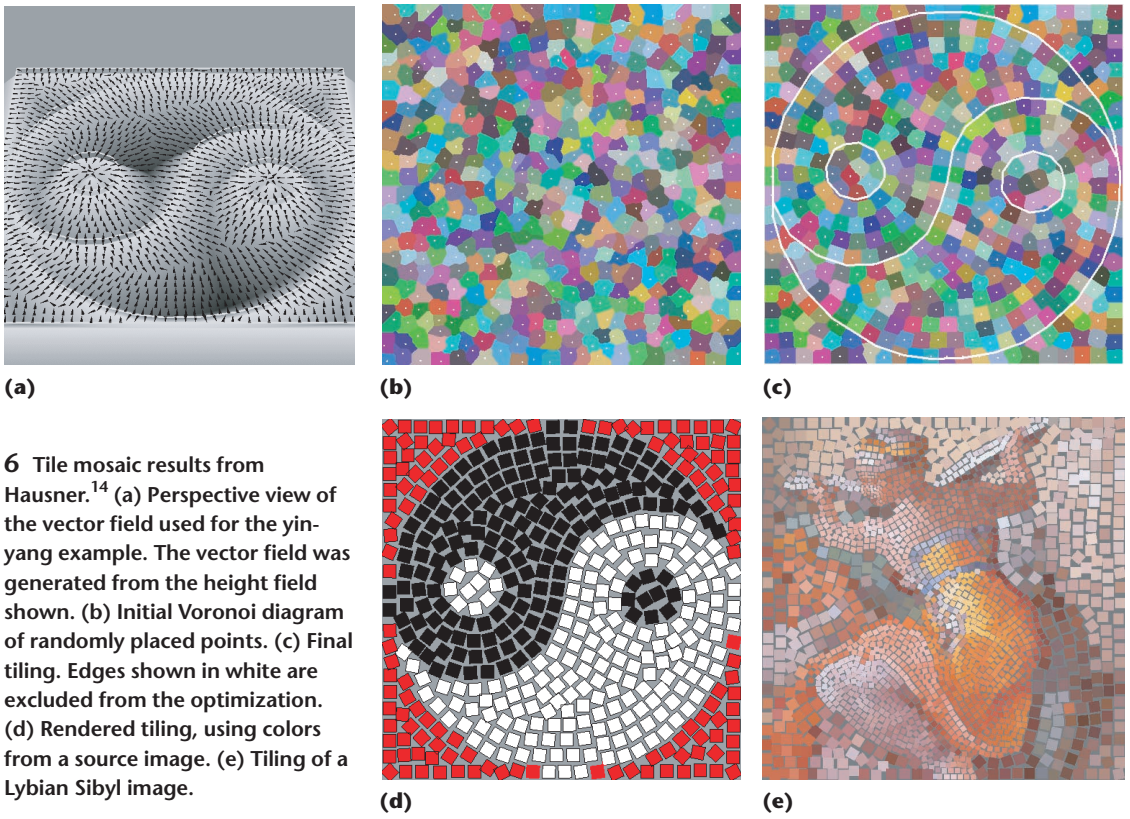
Courtesy of Oliver Deussen et al.

Courtesy of Adrian Secord

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

5 Stippling algorithms. Source images are shown in (a) and (c), results from Deussen et al.[11] are shown in (b), and from Secord[13] in (d). To match the target gray tones, Deussen et al.'s algorithm varies stipple size (keeping stipple density constant), whereas Secord's algorithm varies stipple density. Stipple size is also adjusted as a postprocess. The eye in the grasshopper image was manually segmented from the rest of the image.

This summation can be accelerated by precomputing sums of $\kappa(p)$. This method gives somewhat sharper image boundaries, since the stipple placement is directly affected by the source image's intensity. Secord also uses a simpler initialization procedure based on rejection sampling.



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

Courtesy of Alejo Haunser

6 Tile mosaic results from Hausner.[14] (a) Perspective view of the vector field used for the yin-yang example. The vector field was generated from the height field shown. (b) Initial Voronoi diagram of randomly placed points. (c) Final tiling. Edges shown in white are excluded from the optimization. (d) Rendered tiling, using colors from a source image. (e) Tiling of a Lybian Sibyl image.

Specifically, the algorithm samples point locations from a uniform distribution and includes the sampled points in the initialization with probability proportional to $\rho(p)$.

Lloyd's method can also create tile mosaics from color source images. A simple approach is to create a Voronoi diagram of an image and then color each region of the image by the color from the underlying source image.[10] However, this produces a mosaic with irregular tile shapes.

Hausner[14] describes two enhancements to this method (see Figure 6). First, to generate square tile shapes, replace the L_2 norm with the L_1 norm ($\|v\|_1=|v_x|+|v_y|$). Second, to create tilings with consistent orientations (see Figure 6a), specify an orientation field $\phi(p)$ for the image. The orientation of each tile is constrained to match the vector field: $\phi_i=\phi(C_i)$. The new energy function is now

$$E(I)=\sum_{p\in I} L^i_p \left\| R_{\phi(Ci)}(p-C_i) \right\|_1^2$$

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

function TILEMOSAIC(n, I):
  initialize the centroids C_i by randomly sampling n points uniformly in the image I
  while the algorithm has not converged
    reestimate the labeling by

$$L^i_p \leftarrow \begin{cases} 1 & i=\text{argmin}_i \left\| R_{\phi(Ci)}(p-C_i) \right\|_1^2 \\ 0 & \text{otherwise} \end{cases}$$

    reestimate the centroids by

$$C_i \leftarrow \sum_p L^i_p p \,/\, \sum_p L^i_p$$

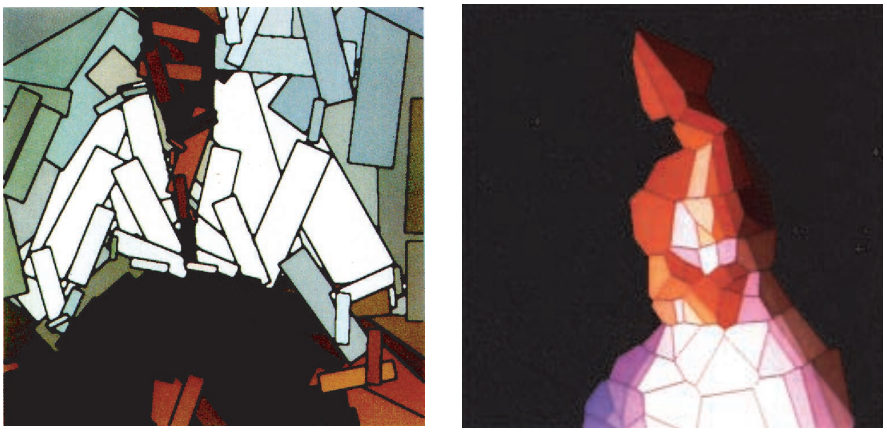
  return the centroids C_i

Note that Hausner's algorithm is no longer optimal since the centroid update step isn't guaranteed to improve the energy function. The algorithm first selects tile positions and orientations, and then colors the tiles. The tile positions and orientations don't take color information into account. Nonetheless, the algorithm tends to achieve good results in practice.

With Hausner's algorithm, we can apply tiling to manually segmented regions, as before. We can also enhance edges by removing them from the energy function. Specifically, points p that lie on image edges aren't included in the labeling or centroid computation steps; this discourages Voronoi regions from straddling edges. Adjusting the energy function in various ways can modify tile sizes and shapes; the resulting problem is amenable to hardware acceleration.[14]

Trial-and-error algorithms

It's difficult to extend Voronoi methods to take color information into account and handle problems where strokes might overlap. So far, the only optimization methods applicable to these problems are trial-and-error approaches, which you can apply to any SBR problem. The idea is simple. The algorithm proposes a change to the image structure. If the proposed change reduces the energy, then the change is incorporated;



Courtesy of Paul Haeberli

(a) (b)

7 Images computed using trial-and-error algorithms.[4] (a) Overlapping rectangular strokes. (b) Voronoi diagram of a set of point centers.

otherwise, it's discarded. The algorithm then repeats.

If the proposal mechanism is well designed, then the algorithm should eventually converge to a low-energy result. However, there are no guarantees that this will happen, and, even if it does, the computation time could be substantial. Here's the pseudocode for a trial-and-error algorithm:

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

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

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

Haeberli[4] introduced the first trial-and-error algorithm for nonphotorealistic rendering; Figure 7 shows the results. In each case, a fixed number of strokes are randomly perturbed, and the perturbations are kept only if the sum-of-squares difference to the source image is reduced.

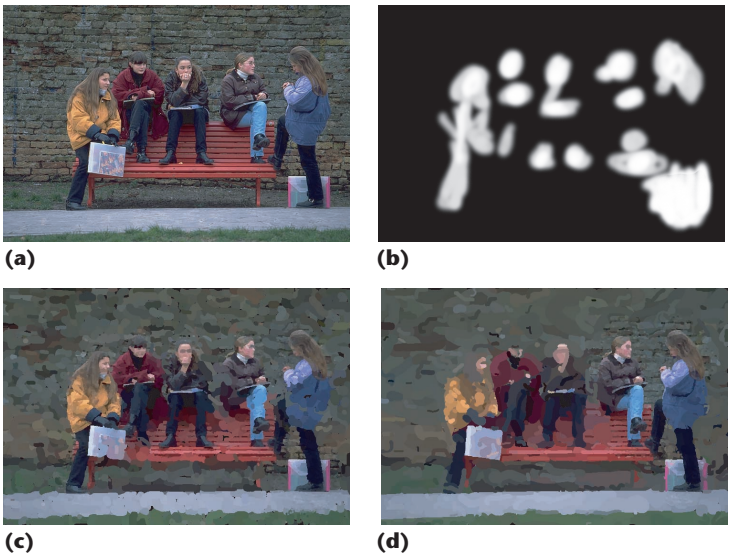
Streamline visualization by trial and error. More recently, Turk and Banks[3] demonstrated a trial-and-error algorithm for vector field visualization of streamlines (see Figure 2). (Jobard and Lefer later described a greedy streamline placement algorithm that's much faster than the trial-and-error method.[15] I describe Turk and Banks' method here for completeness.) As mentioned in the introduction, the problem is to illustrate a vector field with streamlines for clear visualization of the vector field. However, a straightforward approach to the problem—simply tracing streamlines from some predetermined starting points—creates irregular streamline spacing that distracts from the vector field's flow. Hence, you need some way of evaluating the streamline visualization's quality and then optimizing for that quality measure. Turk and Banks proposed blurring the streamline rendering, comparing the result to a predefined constant value t. The corresponding energy function is

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

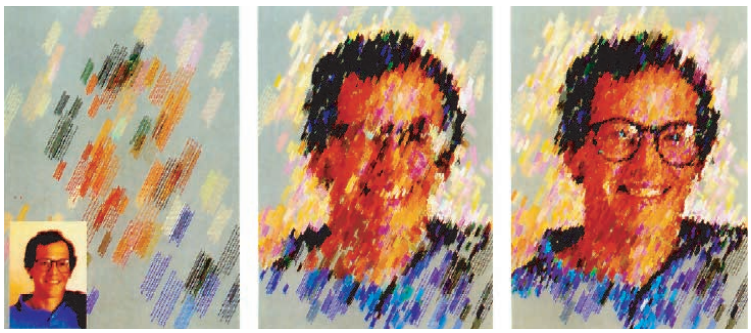
where $(G * I)(\mathrm{p})$ denotes the blurred version of the streamline image. (Salisbury et al. used a similar energy function in an earlier study.[5]) You could also penalize the deviations of the streamline from the vector field, since the goal is to produce streamlines that exactly follow the vector field. Fortunately, the trial-and-error algorithm enforces this constraint at every step, and thus it isn't necessary to include it in the energy function.

To apply a trial-and-error algorithm to this problem, we must define the proposal mechanism. While a purely random proposal mechanism may decrease the energy in the long run, it will be far too slow to be practical. Hence, Turk and Banks defined many proposal heuristics designed to decrease the energy as much as possible with each step.

Painterly rendering by trial and error. I've built a trial-and-error painterly rendering algorithm, which I'll briefly describe here.[16] At a high level, the goal is to seek concise paintings that match a source image closely and cover the image with paint, but use as few strokes as possible. A brush radius and a list of control points define each brush stroke. The energy function is



8 Spatially varying style, from Hertzmann.[16] (a) Source image (courtesy of Philip Greenspun; http://philip.greenspun.com). (b) Interactively painted weight image (w_{app}). (c) Resulting painting with the given weights. More detail appears near faces and hands. (d) Another choice of weights; detail is concentrated on the rightmost figures.



9 Interactive painterly rendering process from Haeberli.[4]

$$E(I) = E_{app}(I) + E_{nstr}(I) + E_{cov}(I)$$

$$E_{app}(I) = \sum_{(p)\in I} w_{app}(p)\left\|I(p) - S(p)\right\|$$

$$E_{nstr}(I) = w_{nstr} \cdot (\text{number of strokes in } I)$$

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

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

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

By default, the value of $w_{app}(p)$ is initialized by a binary edge image. If you let the weight vary over the canvas, then you get an effect similar to having different energy functions in different parts of the image (see Figure 8). The weight image $w_{app}(p)$ lets you specify how much detail is required in each region of the image. You can generate the weight image automatically or hand paint it. This gives users a high level of control without requiring them to make every low-level choice.

This problem is difficult to optimize. The search—even with carefully designed proposal heuristics—can take many hours to run and currently isn't practical. However, the algorithm does give economical results and substantial high-level control to users.

Greedy algorithms

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

Each greedy algorithm operates by repeatedly placing strokes and never modifying them. Consequently, we define a greedy algorithm by how it makes the following two choices in the inner loop:

- Where do we place the next stroke?
- What shape will the next stroke have?

Single-point strokes

Haeberli[4] describes a simple, semiautomatic painting algorithm (see Figure 9). First, the user provides a source image. Then the user sees a rendering of the painting, which is initially blank. Using a mouse or tablet, the user clicks and drags within the painting area and places a single brush stroke at the location of each mouse click. The system automatically chooses the color by extracting it from the color of the source image at that point and orients the stroke in the direction of the image's gradient. Hence, the user decides where the strokes go and the algorithm decides what they look like. The user may set other parameters (such as stroke sizes) by adjusting settings or via pressure on a tablet interface. This system provides a fun and easy way to make abstract and attractive versions without requiring the user to possess any drawing skills.

Single-layer painterly rendering. Numerous commercial software packages have incorporated fully automatic versions of Haeberli's algorithm. Litwinowicz[17] provides a complete description of such an algorithm, along with several enhancements.

Litwinowicz's basic algorithm takes a source image

and orientation field as input and generates a painting with a set of oriented, short, brush strokes. The algorithm places the brush strokes on a grid in the image plane, with randomly perturbed positions. Each stroke's color comes from the source image and each stroke's orientation comes from the orientation field. The system draws the strokes in random order, removing regularities that would appear otherwise. The orientation field specifies the strokes' desired orientation and is generated from the source image in a preprocessing step. A simple way to generate this orientation field is to set the orientation $\phi(p)$ at pixel p to the normal of the image's gradient—this gives the direction in which the image is most constant. However, in constant regions of the image, the gradient won't be well defined. The orientations in these regions can be filled in using a smoothing algorithm, such as thin-plate spline interpolation. Additionally, strokes can be clipped to edges extracted from the original images. This helps the painting preserve the edges of the original image more faithfully.

Multiple-layer painterly rendering. I have developed an extension to these algorithms that can create brush strokes with multiple sizes. We can motivate the algorithm by observing that an artist often will begin a painting as a rough sketch and go back later over the painting with a smaller brush to add detail. While much of the motivation for this technique doesn't apply to computer algorithms, it does yield desirable visual effects. This image-processing algorithm uses fine brush strokes only where necessary to refine the painting and leaves the rest of the painting coarse. Users can also define where fine strokes are used. The algorithm is similar to a pyramid algorithm, in that you start with a coarse approximation to the source image and add progressive refinements with smaller brushes. In a sense, this algorithm greedily optimizes an energy function that penalizes the difference between the painting and the source image and penalizes the number of strokes.

The algorithm takes as input a source image and a list of brush sizes, which are expressed as radii $r_1 \ldots r_n$. The algorithm then proceeds by painting a series of layers, one for each radius, from largest to smallest. Generally, it's most useful to use powers of two: $r_i = r_1 2^{i-1}$, with some user-determined value for r_1. The initial canvas is a constant color image.

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

Blurring may be performed by one of several methods. I normally blur by convolution with a Gaussian kernel of standard deviation $f_\sigma r_i$, where f_σ is some constant factor. Nonlinear diffusion[18] can be used instead of a Gaussian blur to produce slightly better results near edges, although the improvement is rarely worth the extra computation time. When speed is essential, I use a summed-area table.

This entire procedure repeats for each brush stroke size. Here's a pseudocode summary of the painting algorithm:

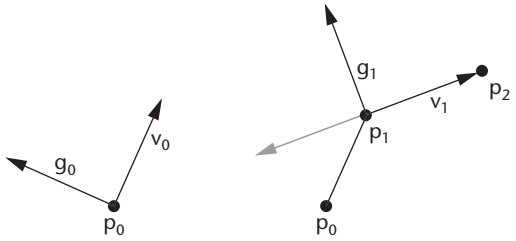
```
function PAINT(Is, //source image
      Ip, //canvas
      r1 … rn) //brush sizes
  Create a summed-area table A from Is if necessary
  refresh ← true
  foreach brush size ri, from largest to smallest, do
    Compute a blurred reference image Iri with blur
      size fσri
    grid ← ri
    Clear depth buffer
    foreach position p on a grid with spacing grid
      M ← the region [px – grid/2 … px + grid/2;
        py – grid/2 … py + grid/2]
        areaError ← ∑p∈M ||Ip (p)–Iri (p)||
        if refresh or areaError > T then
          p ← arg maxp∈M ||Ip (p)–Iri (p)||
          PAINTSTROKE(p, Ip,ri, Iri )
  refresh ← false
```

Each layer is painted using a simple loop over the image canvas. The idea is similar to Litwinowicz's algorithm. However, we can no longer place samples simply on a jittered grid, since this approach might miss sharp details such as lines and points that pass between grid points. Instead, the algorithm searches each grid point's neighborhood to find the nearby point with the greatest error and paint at this location. All strokes for the layer are planned at once and before rendering. Then the strokes are rendered in random order to prevent an undesirable appearance of regularity in the brush strokes. In practice, we can avoid the overhead of storing and randomizing a large list of brush strokes by using a z-buffer. Each stroke is rendered with a random z value as soon as it's created. The z-buffer is cleared before each layer. Note that this might produce different results with significant transparency, when transparent objects aren't rendered in back-to-front order. Figure 10 (next page) shows the layers of a painting using this algorithm.

When applied to a color vector, $|\cdot|$ denotes Euclidean distance in RGB space. I also experimented with the Commission Internationale de l'Eclairage (CIE, or International Commission on Illumination) LUV, a perceptually based color space. Surprisingly, it gave slightly worse results, but I'm not sure why.

PAINTSTROKE in the previous code listing is a generic procedure that places a stroke on the canvas beginning at p_1, given a reference image and a brush radius. This technique focuses attention on areas of the image containing the most detail (high-frequency information) by placing many small brush strokes in these regions. Areas with little detail are painted only with large brush strokes. Thus, strokes are appropriate to the level of detail in the

10 Painting with three brushes. (a) The input image. The remaining images show the painting after (b) the first layer (brush radius 8), (c) the second layer (radius 4), (d) the final painting (radius 2), and (e) the painting with paint texture added. (Brush strokes from earlier layers are still visible in the final painting.)



11 Painting a brush stroke.[1] (a) A brush stroke begins at a control point p_0 and continues in direction v_0, normal to the gradient direction g_0. (b) From the second point p_1, there are two normal directions to choose from: $\theta_1 + \pi/2$ and $\theta_1 - \pi/2$. I choose v_1 to reduce the stroke curvature. This procedure is repeated to draw the rest of the stroke. The stroke will be rendered as a cubic B-spline, with the p_i as control points. The distance between control points is equal to the brush radius.

source image. This choice of emphasis assumes that detail areas contain the most important visual information, although other choices of emphasis are also possible, such as those specified by a user or an eye tracker.

Long, curved strokes

Most real paintings and drawings use long, curved strokes, instead of the short strokes that I've discussed until now.

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

My spline placement algorithm begins at a given point in the image p_0, with a given brush radius r. The stroke is represented as a list of control points, a color, and a brush radius. Points are represented as floating-point values in image coordinates. I add the control point p_0 to the spline, and use the color of the reference image at p_0 as the color of the spline.

I then compute the next point along the curve. The gradient direction θ_0 at this point is computed from the Sobel-filtered luminance of the reference image. The next point, p_1, is placed in the direction $\theta_0 + \pi/2$ at a distance r from p_0 (see Figure 11). You can also use the direction $\theta_0 - \pi/2$; this choice is arbitrary. I use the brush radius r as the distance between control points because r represents the level of detail I'll capture with this brush size; in practice, I find that this size works best. This means that large brushes create broad sketches of the image, which can be later refined with smaller brushes.

The remaining control points are computed by repeat-

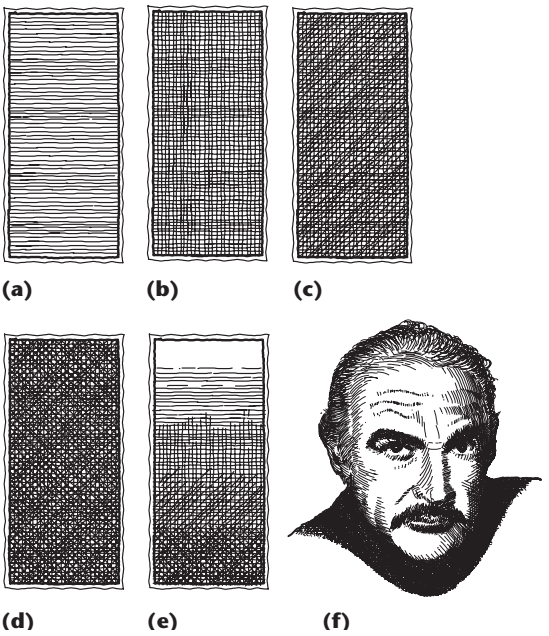
ing this process of moving along the image and placing control points. For a point p_i, we compute a gradient direction θ_i at that point. There are actually two possible candidate directions for the next direction: $\theta_i + \pi/2$ and $\theta_i - \pi/2$. I choose the next direction that leads to the lesser stroke curvature: I pick the direction v_i so that the angle between v_i and $v_i - 1$ is less than or equal to $\pi/2$ (see Figure 11), where v_i can be $(r\cos(\theta_i \pm \pi/2), r\sin(\theta_i \pm \pi/2))$. The stroke terminates when

- it reaches the predetermined maximum stroke length, or
- the reference image color at the current control point differs from the current stroke color more than it differs from the current painting at that point.

I find that a step size of r works best for capturing the right level of detail for the brush stroke. We can also exaggerate or reduce the brush stroke curvature by filtering the stroke directions.

The entire stroke placement procedure is as follows (note that $Y_r(p)$ is the luminance channel of I_r, scaled from 0 to 1):

```
function PAINTSTROKE(p0, r, Ir, Ip)
    //Arguments: start point p0, stroke radius (r),
    //reference image (Ir), painting so far (Ip)
    color ← Ir(p0)
    K ← a new stroke with radius r and color color
    add point p0 to K
    for i = 1 to maxStrokeLength do
     //compute image derivatives
     g ← (255 * ∂Yr/∂x (pi−1),
       255 * ∂Yr/∂y (pi−1))

     //detect vanishing gradient
     if ri ||g|| ≥ 1
      //is gradient times length at least a pixel?
     //rotate gradient by 90 degrees
     vi ← (−gy, gx)

     // if necessary, reverse direction
     if i > 1 and vi · vi−1 < 0 then
      vi ← −vi

     // filter the stroke direction
     vi ← fc vi + (1 − fc)vi−1
    else
     if i > 1
      //continue in previous stroke direction
      vi ← vi−1
     else
      return K

     pi ← pi−1 + ri vi/||vi||
     if i > minStrokeLength and
       ||Ir(pi) − Ip(pi)|| < ||Ir(pi) − color||  then

    return K
    add pi to K
end for
return K
```



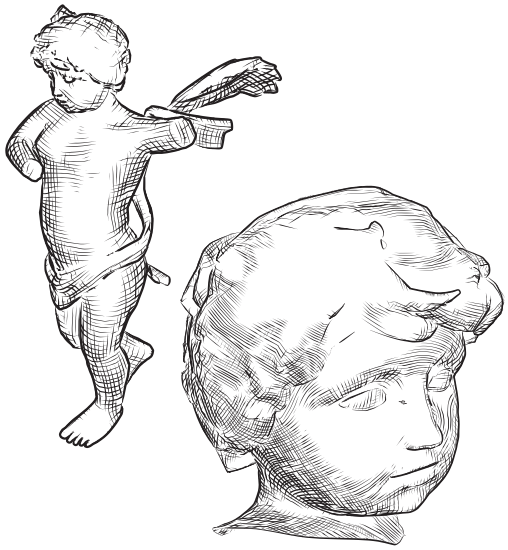
12 Gray tones generated with a prioritized stroke texture.[5] (a-e) The strokes in the texture are rendered in a specific order that lets different tones be generated from a single texture. (f) Image generated from a source photo using prioritized stroke textures. (Images courtesy of Michael Salisbury et al.)

Pen-and-ink and other curve tracing algorithms. Many types of pen-and-ink illustration greedily optimize stroke placement. In the simplest case, the goal is to place pen strokes to achieve a desired stroke density (thus achieving a target tone) with strokes that trace specified orientations. Jobard and Lefer[15] describe an efficient greedy approach to this problem. They define a target density by a desired distance d between strokes. In a nutshell, their algorithm consists of

- identifying seed points in the image with a distance of at least d from all existing curves, and
- tracing a curve from a seed point along the vector field, until that curve comes within d of another curve.

The algorithm continues until the image is densely covered with strokes. This procedure is guaranteed to produce curves that trace the vector field and maintain a distance of at least d from all other curves. This algorithm depends on intelligent heuristics for guiding the choice of seed points.

A more sophisticated problem is to use pen strokes to illustrate tone, orientation, and texture. Salisbury et al.[5] introduced an interactive tool for placing pen-and-ink strokes that convey tone and orientation. The user specifies the desired tone for a region and the system automatically places strokes to match these tones. Strokes are stored as *prioritized stroke textures* (see Figure 12), which let the system render complex hatching patterns while matching a desired tone. Salisbury et al.[6] describe another system that lets the user specify varying orientations for the illustration as well, thus



Courtesy of Aaron Hertzmann and Denis Zorin

13 Pen-and-ink illustration of a smooth surface, from Hertzmann and Zorin.[19] The target orientation field and tones are generated automatically to illustrate the surface.

matching three separate quantities (tone, texture, and orientation) with pen-and-ink strokes.

You can apply similar principles to illustrating surfaces. Typically, the target tones come from a rendering of the surface and the target stroke orientations come from orientation fields defined on the surface. The resulting algorithm is a variation on previous SBR techniques, but with many adjustments as dictated by the pen-and-ink style and 3D model.[19] Denis Zorin and I first generated a tone and orientation field from a 3D model and extended Jobard and Lefer's[15] method to hatch an image of the model (see Figure 13). Winkenbach and Salesin[7,8] describe a system that applies prioritized stroke textures to renderings of 3D surfaces with texture, optionally user-supplied emphasis for different parts of the image, and user-defined orientation fields (see Figure 14). In these methods, the hatching attempts to match the image's orientation, rendered tone, and texture.

Limitations of energy minimization

Currently, it seems unlikely that every desirable SBR style can be formulated in the energy function formulation that I presented in the "Stroke-based rendering" section. There are several reasons for this. It's difficult to capture looseness and sketchiness or randomness in an energy function. Moreover, painting and drawing aren't deterministic procedures; an artist might produce different images each time. One way to express randomness would be to replace the energy function with the probability density over renderings. Painting is then a process of sampling from this density; the density would usually be conditioned on the input data.

Several of the greedy approaches previously described (such as prioritized stroke textures[5]) and mentioned in the "Related Topics" sidebar are difficult to express in terms of energy functions.

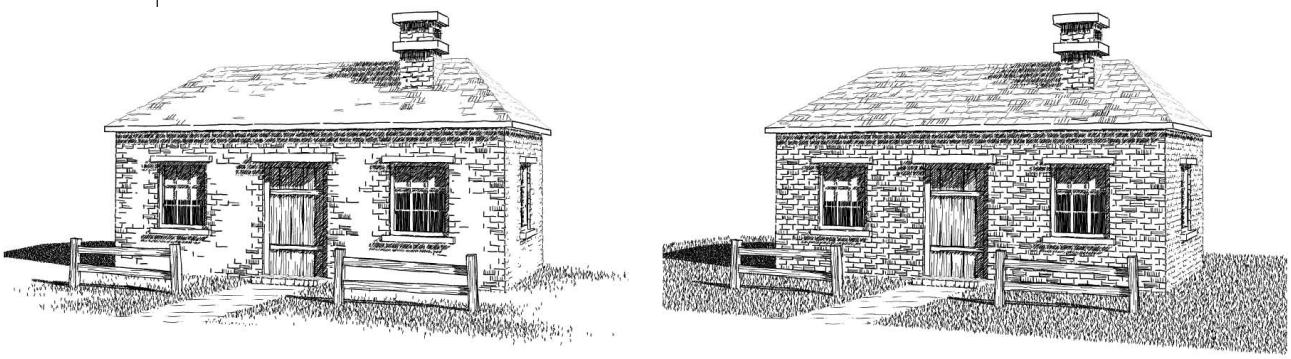
It's sometimes easier to design a direct procedure for a rendering style than to design an energy function, especially since designing styles is a creative process. Often, we design a new algorithm or styles without really understanding why they work. Ideally, we should develop additional insight after the fact that lets us convert the direct procedure to an energy function. Knowing the energy function can often give insight into how the direct procedure works and how to improve it. However, it bears repeating that direct procedures are much faster than optimization procedures.

Conclusion

Many challenges remain in SBR algorithms. First, most of these algorithms are too slow to be useful in an interactive application, although faster computers will lessen the problem. Second, for most applications, artists and end users need better tools for controlling styles, in between setting parameters of a painting algorithm (which may give too little control) and painting all strokes manually (which is very labor intensive). The work of Kalnins et al.[20] gives an excellent example of artistic control over a specific type of SBR.

More importantly, we need to dramatically expand the range of styles that SBR algorithms can create. Techniques up to now have shown the power of SBR algorithms on relatively simple styles—such as a simple version of impressionism. Now, the task is to discover deeper patterns in artistic styles amenable to implementation—beyond simple paint-scattering effects.

A final and significant challenge is to create compelling SBR animation. Because so few examples exist in traditional animation to look to for guidance, this



Courtesy of Georges Winkenbach and David Salesin

14 Pen-and-ink illustrations of 3D models, from Winkenbach and Salesin.[7] The left house shows the effect of a user-defined emphasis function; detail is only drawn where specified by the user. The right house shows a rendering with uniform emphasis.

requires not just developing new algorithms, but also developing new artistic styles. If we as a field are successful, we'll have created a new art form that couldn't have existed without computers. ■

References

1. A. Hertzmann, "Painterly Rendering with Curved Brush Strokes of Multiple Sizes," *Proc. Siggraph 98*, ACM Press, 1998, pp. 453-460.
2. A. Hertzmann, "Fast Paint Texture," *Proc. 2nd Ann. Symp. Non-Photorealistic Animation and Rendering* (NPAR 2002), ACM Press, 2002, pp. 91-96, 161.
3. G. Turk and D. Banks, "Image-Guided Streamline Placement," *Proc. Siggraph 96*, ACM Press, 1996, pp. 453-460.
4. P.E. Haeberli, "Paint By Numbers: Abstract Image Representations," *Computer Graphics* (Proc. Siggraph 90), vol. 24, ACM Press, 1990, pp. 207-214.
5. M.P. Salisbury et al., "Interactive Pen–And–Ink Illustration," *Proc. Siggraph 94,* ACM Press, 1994, pp. 101-108.
6. M.P. Salisbury et al., "Orientable Textures for Image-Based Pen-and-Ink Illustration," *Proc. Siggraph 97*, ACM Press, 1997, pp. 401-406.
7. G. Winkenbach and D.H. Salesin, "Computer-Generated Pen-And-Ink Illustration," *Proc. Siggraph 94,* ACM Press, 1994, pp. 91-100.
8. G. Winkenbach and D.H. Salesin, "Rendering Parametric Surfaces in Pen and Ink," *Proc. Siggraph 96*, ACM Press, 1996, pp. 469-476.
9. M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL,* 2nd ed., Addison-Wesley Developers Press, 1997.
10. K. Hoff III et al., "Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware," *Proc. Siggraph 99*, ACM Press, 1999, pp. 277-286.
11. O. Deussen et al., "Floating Points: A Method for Computing Stipple Drawings," *Computer Graphics Forum*, vol. 19, no. 3, Aug. 2000.
12. A. Secord, "Weighted Voronoi Stippling," *Proc. 2nd Ann. Symp. Non-Photorealistic Animation and Rendering* (NPAR 2002), ACM Press, 2002, pp. 27-43.
13. A. Secord, *Random Marks on Paper: Non-Photorealistic Rendering with Small Primitives*, master's thesis, Dept. of Computer Science, Univ. of British Columbia, Oct. 2002.
14. A. Hausner, "Simulating Decorative Mosaic," *Proc. Siggraph 2001*, ACM Press, 2001, pp. 573-578.
15. B. Jobard and W. Lefer, "Creating Evenly-Spaced Streamlines of Arbitrary Density," *Proc. 8th Eurographics Workshop on Visualization in Scientific Computing*, Eurographics, 1997, pp. 45-55.
16. A. Hertzmann, "Paint by Relaxation," *Computer Graphics Int'l 2001*, IEEE CS Press, 2001, pp. 47-54.
17. P. Litwinowicz, "Processing Images and Video for an Impressionist Effect," *Proc. Siggraph 97*, ACM Press, 1997, pp. 407-414.

Related Topics

Space limitations precluded a complete survey of stroke-based rendering. You can find a detailed list of references at http://www.dgp.toronto.edu/~hertzman/sbr02. A few of the major related topics are

- animation and real-time rendering for creating stroke-based animations, virtual environments, and interfaces;
- example-based strokes for creating illustration styles from hand-drawn examples;
- thresholding algorithms, which decouple stroke placement and stroke tones;
- tensor field visualization, where stroke-like elements are used for scientific visualization of complex data;
- photomosaics and jigsaw image mosaics, in which an image is approximated by a collection of smaller images; and
- simulating stroke texture by numerical simulation and/or procedural synthesis.

18. P. Perona and J. Malik, "Scale-Space and Edge Detection Using Anisotropic Diffusion," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, July 1990, pp. 629-639.
19. A. Hertzmann and D. Zorin, "Illustrating Smooth Surfaces," *Proc. Siggraph 2000*, ACM Press, 2000, pp. 517-526.
20. R.D. Kalnins et al., "WYSIWYG NPR: Drawing Strokes Directly on 3D Models," *ACM Trans. Graphics*, vol. 21, no. 3, 2002, pp. 755-762.

Aaron Hertzmann *is an assistant professor in the Department of Computer Science at the University of Toronto. His research interests are 3D shape reconstruction, nonphotorealistic rendering, and applications of machine learning to computer graphics. Hertzmann received a PhD in computer science from New York University. He is a member of the IEEE.*

Readers may contact Aaron Hertzmann at the Dept. of Computer Science, University of Toronto, 10 King's College Rd., Room 3303, Toronto, ON, M5S 3G4; hertzman@dgp.toronto.edu.