Stroke-Based Rendering

Aaron Hertzmann
University of Washington

1 Introduction

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

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

Source photo

Painted version

Final rendering

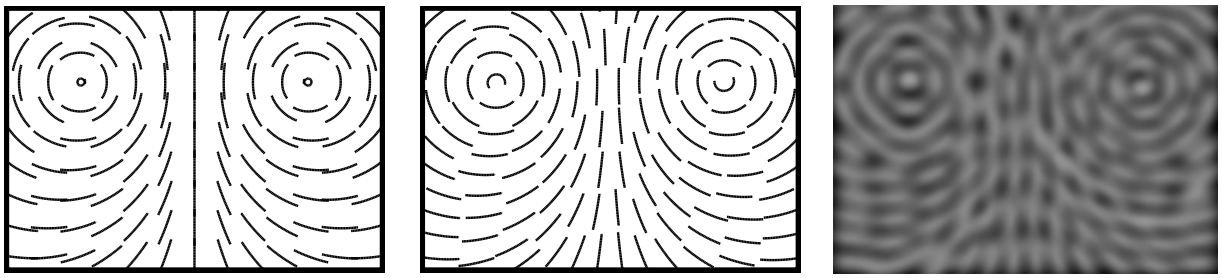
This figure shows an SBR algorithm in action: starting from a photograph, a collection of brush strokes are placed to match the photograph, and then rendered to appear as if created with oil paint.

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

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

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

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



Vector field Final rendering Blurred rendering

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

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

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

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

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

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

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

2 Stroke-Based Rendering: Stating the Problem

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

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

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

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

Stippling stroke model Individual strokes (stipples)

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

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

other definitions of a stipple are possible.)

We create images by combining strokes:

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

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

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

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

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

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

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

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

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

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

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

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

3 Optimization algorithms

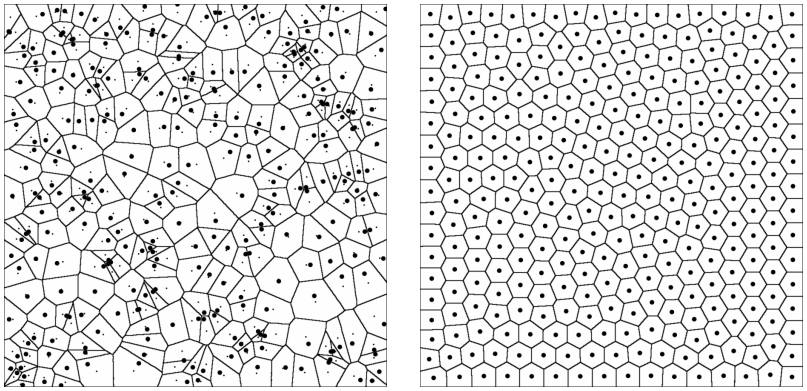
In this section, I describe optimization algorithms for SBR problems. Two kinds of optimization algorithms have been applied to SBR. The first kind, which I will call **Voronoi algorithms**, exploits special properties of the SBR problem to perform efficient global update steps. The second kind, which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

3.1 Voronoi algorithms

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

3.1.1 LLOYD'S METHOD

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



(a) (b)

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

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

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

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

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

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

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

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

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

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

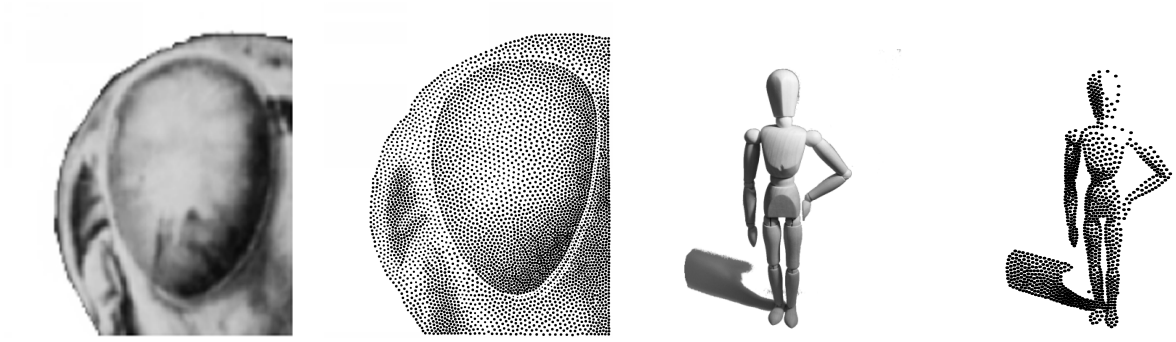
3.1.2 VARIATIONS ON LLOYD'S METHOD FOR SBR

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

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

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

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



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

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

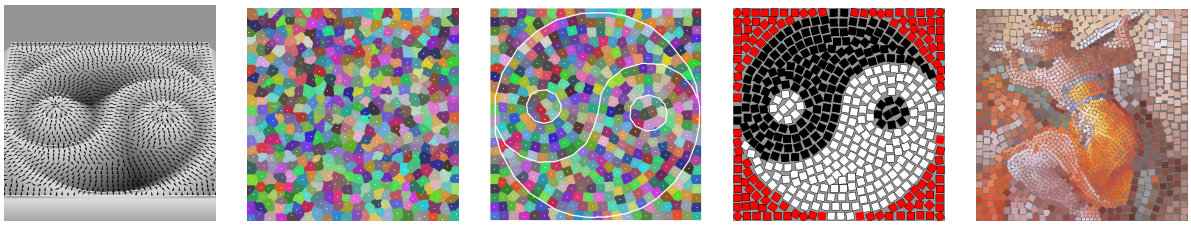
Following the same steps as before directly leads to a slightly different version of Lloyd's method: the labeling step is the same, but the centroids are now reestimated as $\mathbf{C}_i \leftarrow \frac{\sum_{\mathbf{p}} L^i_{\mathbf{p}} \rho(\mathbf{p}) \mathbf{p}}{\sum_{\mathbf{p}} L^i_{\mathbf{p}}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

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

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

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

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



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

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

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

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

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

3.2 Trial-and-Error algorithms

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

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

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

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

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

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

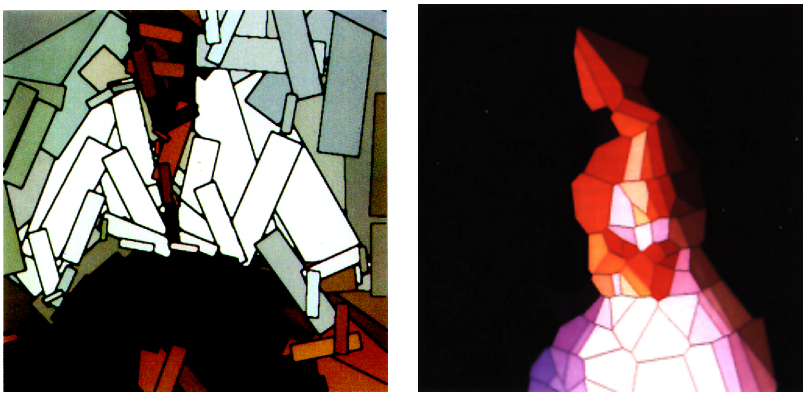
3.2.1 STREAMLINE VISUALIZATION BY TRIAL-AND-ERROR

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

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

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

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



(a) (b)

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

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

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

3.2.2 PAINTERLY RENDERING BY TRIAL-AND-ERROR

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

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

(a) (b)

(c) (d)

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

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

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

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

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

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

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

4 Greedy algorithms

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

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

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

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

4.1 Single-point strokes

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

4.1.1 SINGLE-LAYER PAINTERLY RENDERING

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

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

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

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

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

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

4.1.2 MULTIPLE-LAYER PAINTERLY RENDERING

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Long, curved strokes

4.2.1 PAINTERLY RENDERING WITH LONG, CURVED STROKES

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

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

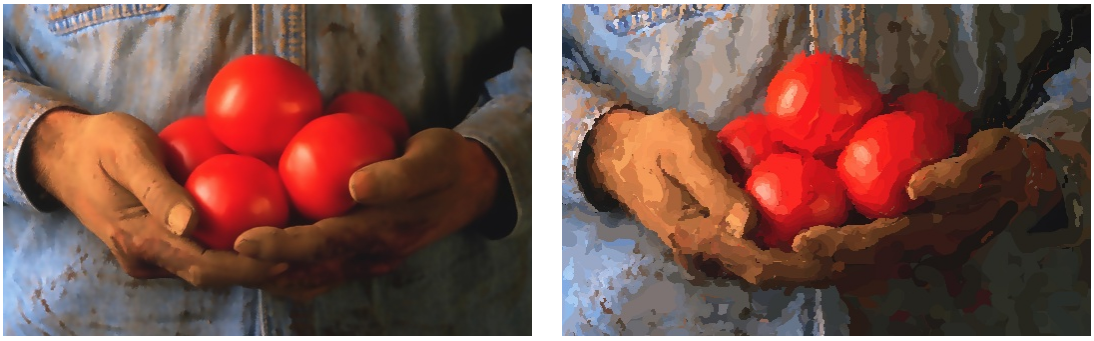
We then need to compute the next point along the curve. The gradient direction θ_0 at this point

Reference images Layers

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

(a) (b)

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



(a) (b)

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

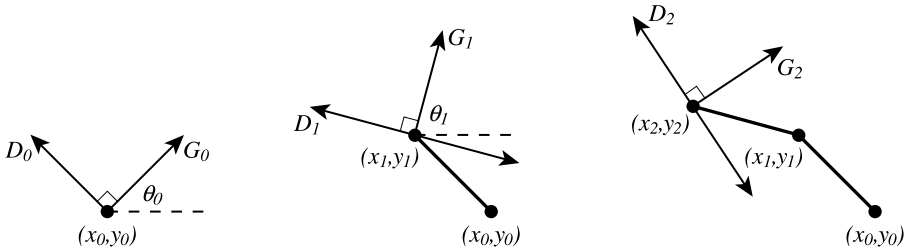


Figure 10: Painting a brush stroke [Her98]. (a) A brush stroke begins at a control point $\mathbf{p}_0$ and continues in direction $\mathbf{D}_0$, normal to the gradient direction $\mathbf{G}_0$. (b) From the second point $\mathbf{p}_1$, there are two normal directions to choose from: $\theta_1 + \pi/2$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

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

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

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

The entire stroke placement procedure is as follows:

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

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

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

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

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

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

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

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

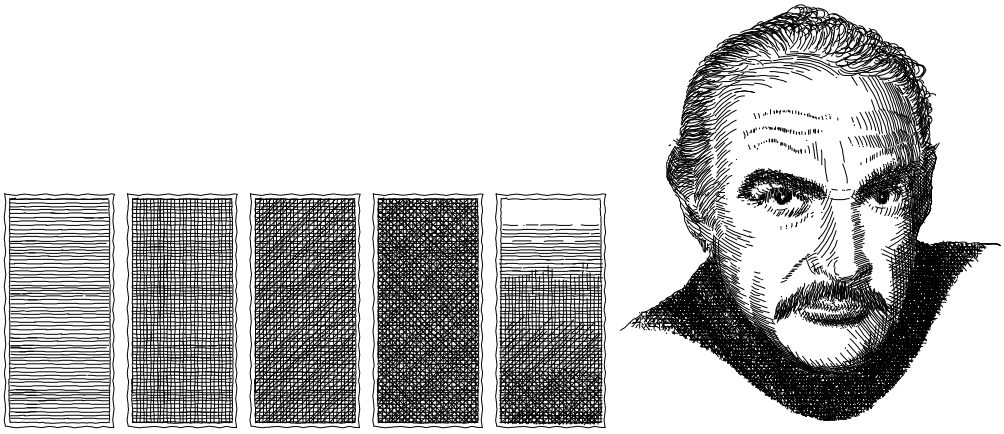


Figure 11: Gray tones generated with a prioritized stroke texture [SABS94]. The strokes in the texture are rendered in a specific order which allows different tones to be generated from a single texture. The right image was generated from a source photo using prioritized stroke textures. Images used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

4.2.2 PEN-AND-INK AND OTHER CURVE TRACING ALGORITHMS

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

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

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

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

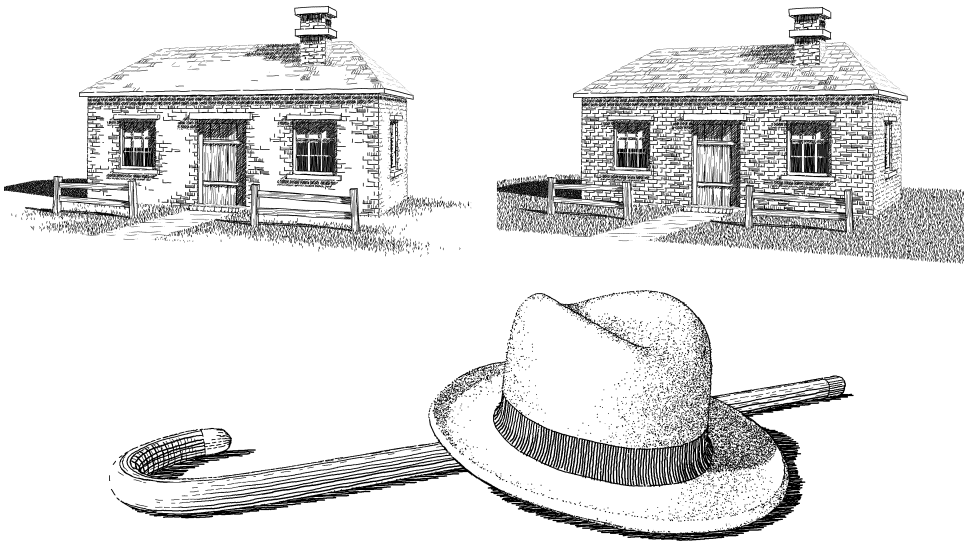


Figure 12: Pen-and-ink illustrations of 3D models, from [WS94] and [WS96], respectively. The left house shows the effect of a user-defined emphasis function; detail is only drawn where specified by the user. The right image shows pen-and-ink illustration of a parametric surface. Images used by permission.

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

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

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

4.3 Globally-greedy

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

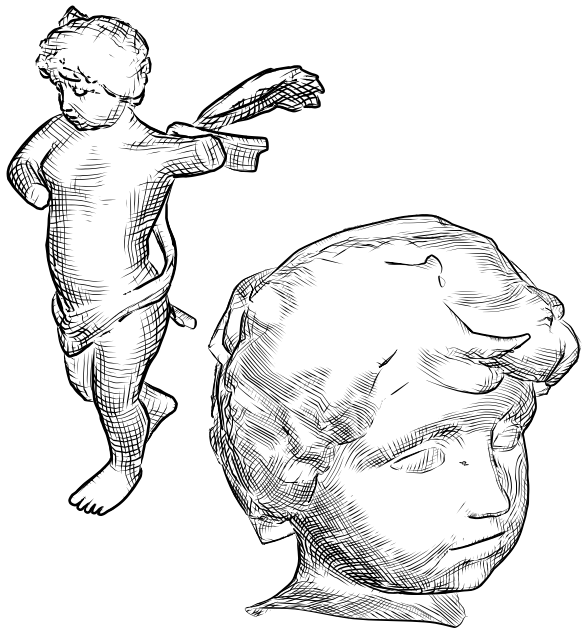


Figure 13: Pen-and-ink illustration of a smooth surface, from [HZ00]. The target orientation field and tones are generated automatically to illustrate the surface.

5 Stroke models

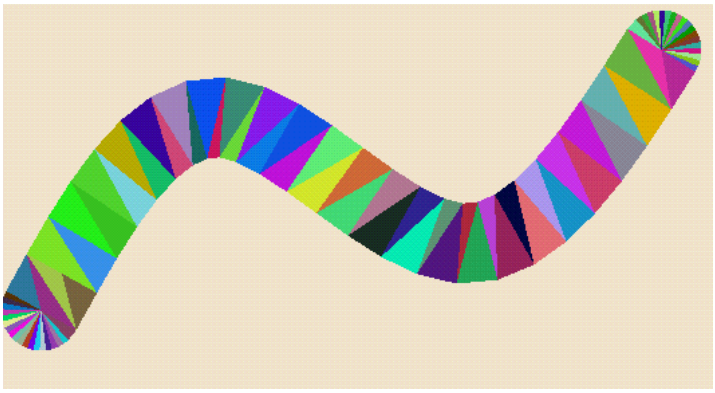
In this section, I describe the curved paint stroke model that I have used. Other stroke models (including stipples, tiles, single-point strokes, and pen strokes) will hopefully appear in a later version of this document. More sophisticated curved stroke models are also possible (e.g. [HLW93, Fra98,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

5.1 Long, curved strokes with triangle strips

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

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

strip:



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

1. Compute curve tangents at each control point. An adequate approximation to the tangent for an interior stroke point $\mathbf{p}_i$ is given by $\mathbf{v}_i = \mathbf{v}_{i+1} - \mathbf{v}_{i-1}$. The first and last tangents are $\mathbf{v}_0 = \mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{v}_{n-1} = \mathbf{p}_{n-1} - \mathbf{p}_{n-2}$.
2. Compute curve normal directions as $\mathbf{n} = (\mathbf{n}_{xi}, \mathbf{n}_{yi}) = (\mathbf{v}_{yi}, -\mathbf{v}_{xi})/\|\mathbf{v}_i\|$
3. Compute points on the boundary of the stroke as points offset by a distance R along the curve normal direction. The offsets for a control point are $\mathbf{a}_i = \mathbf{p}_i + R\mathbf{n}_i$ and $\mathbf{b}_i = \mathbf{p}_i - R\mathbf{n}_i$.
4. Tesselate the stroke as shown above.
5. If desired, add circular "caps" as triangle fans.

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

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

6 Limitations of the energy-minimization approach

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

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

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

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

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

7 Related topics

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

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

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

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

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

8 Summary

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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