Algorithms for Rendering in Artistic Styles

by

Aaron Hertzmann

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2001

Ken Perlin

Denis Zorin

“The camera cannot compete with painting so long as it cannot be used in heaven or in hell.”
Edvard Munch

“Very few cartoons are broadcast live—it’s a terrible strain on the animators’ wrists.”
The Simpsons

Acknowledgments

This thesis would not have been possible without the contributions and support of many people. I consider myself fortunate to have worked with a number of truly remarkable people.

First and foremost, I am indebted to my advisors Ken Perlin and Denis Zorin for their guidance, compassion, encouragement, and collaboration. Ken's boundless creativity is an inspiration, and has had a huge impact on how I approach computer science. Denis' serious but thoughtful influence has been of enormous benefit.

I have learned much from my other collaborators: Henning Biermann, Matthew Brand, Christoph Bregler, Brian Curless, Trevor Darrell, Chuck Jacobs, Nuria Oliver, David Salesin, and Lexing Ying, each of whom I hope to work with again someday.

I wish to thank my New York-area friends for the good times: Ted Bergman, Henning Biermann, Jason Boyd, Rich Radke, and Will Kenton, each of whom I hope to share caipirinhas with again someday.

I am very grateful to the CAT/MRL staff for making it a very cool and very supportive place to work and play. It is a special place.

Finally, I am grateful to my family for the love and encouragment that made it all possible.

Abstract

We describe new algorithms and tools for generating paintings, illustrations, and animation on a computer. These algorithms are designed to produce visually appealing and expressive images that look hand-painted or hand-drawn. In many contexts, painting and illustration have many advantages over photorealistic computer graphics, in aspects such as aesthetics, expression, and computational requirements. We explore three general strategies for non-photorealistic rendering:

First, we describe explicit procedures for placing brush strokes. We begin with a painterly image processing algorithm inspired by painting with real physical media. This method produces images with a much greater subjective impression of looking hand-made than do earlier methods. By adjusting algorithm parameters, a variety of styles can be generated, such as styles inspired by the Impressionists and the Expressionists. This method is then extended to processing video, as demonstrated by painterly animations and an interactive installation. We then present a new style of line art illustration for smooth 3D surfaces. This style is designed to clearly convey surface shape, even for surfaces without predefined material properties or hatching directions.

Next, we describe a new relaxation-based algorithm, in which we search for the painting that minimizes some energy function. In contrast to the first approach, we ideally only need to specify what we want, not how to directly compute it. The system allows as fine user control as desired: the user may interactively change the painting style, specify variations of style over an image, and/or add specific strokes to the painting.

Finally, we describe a new framework for processing images by example, called “image analogies.” Given an example of a painting or drawing (e.g. scanned from a hand-painted source), we can process new images with some approximation to the style of the painting. In contrast to the first two approaches, this allows us to design styles without requiring an explicit technical definition of the style. The image analogies framework supports many other novel image processing operations.

Contents

List of Figures

List of Appendices

Chapter 1

Introduction

New technologies enable new modes of communication and new art forms. Movies and cartoons originally incorporated ideas from their predecessors — including photography, drawing, and theatre — but eventually evolved their own unique languages. Many approaches to image-making have appeared over the centuries, but only a select few have been successfully translated to the moving image; almost every feature-length movie is either live action, cel animation, or rendered with 3D computer graphics. Many styles have been explored in experimental animation, but the techniques used are impractical for widespread and feature-length use. To our knowledge, there has never been a feature-length film drawn or painted as in the style of, say, Vincent Van Gogh or Thomas Nast — this is not because such a thing would be undesirable, but because it has simply not been possible, digitally or otherwise.

The research described in this thesis aims to make such a thing possible. More generally, we want to **combine the expressivity and beauty of natural media with the flexibility and speed of computer graphics.** We argue that this goal is worthwhile and can be achieved by automating some steps of the image-making process, such as brush stroke placement. The motivation can be provided by looking at a few of the applications we envision:

- An animator working on a feature-length movie is asked to create a scene in a specific painting style. She first designs the painting style via a user interface and creates the animation. The animation is then rendered in this style. She specifies different styles for each character in the scene and for the background, and adjusts the parameters to improve the animation. In addition, she "touches up" the rendered animation in a few critical areas where a specific effect is desired; these touch-ups are incorporated into subsequent iterations of the animation.

- The author of a automobile maintenance manual generates technical illustrations from CAD models of the parts, using a pen-and-ink rendering style. In a digital version of the manual, the reader may view the illustrations from different viewpoints and in different rendering styles such as a pen-and-ink, shaded 3D, etc.

- A user at home makes paintings of his friends and family from family portraits and vacation photos, by applying the automatic rendering styles provided by a commercial product. Because he does not want to spend a lot of time and effort to make the pictures, he just picks from one of the automatic filters provided by the product.

- Young children explore and learn about gardening and agriculture in a "hand-drawn style" virtual garden. The simple, expressive rendering style makes the environment enjoyable and accessible for kids.

- Gamers play video games in sketchy or cartoony virtual worlds. The dramatic use of colors and brush strokes enhances the emotional and narrative content of the game.

In general, rendering styles based on physical media are desirable for many reasons. Most importantly, they provide new ways for an artist or designer to create. This means, for example, that the visual appeal and emotional expressiveness of physical media may be translated to animation. Second, informal and compelling user interfaces change the way a user interacts with a system; for example, sketchy user interfaces are probably more appropriate for rapid prototyping and experimentation than are traditional CAD packages. Third, artistic rendering styles have reduced requirements for geometrical modeling, storage, and bandwidth; for example, a multi-user line-art world may require less bandwidth than streaming photorealistic views or models of the world. Finally, artistic rendering styles allow use of cultural references and existing styles; for example, one might wish to make an animated biography of Van Gogh's life in the visual styles of his paintings.

1.1 Overview of the thesis

This thesis describes several novel algorithms for non-photorealistic rendering (NPR). The contributions of this work are:

- Explicitly-defined stroke-based rendering algorithms for 2D and 3D that improve upon previous research:
  - A painterly image processing algorithm is described (Chapter 3), and extended to processing video sequences and for interaction (Chapter 4). These methods produce images with a much greater subjective impression of looking hand-made than do earlier methods. By adjusting algorithm parameters, a variety of styles can be generated, such as styles inspired by the Impressionists and the Expressionists.
  - A pen-and-ink algorithm for illustrating the shape of a smooth 3D surface, in situations when predefined hatching directions and material properties are not available (Chapter 5).
- An energy-minimization approach (Chapter 6). In contrast to explicit methods, we give only a specification for the desired style, and then use a generic optimization

procedure to create the painting. We demonstrate this technique with an economical painting style. Furthermore, we can enable finer user control of the painting style, such as by specifying different styles for different parts of the image.

- It can be quite difficult to create specific, sophisticated styles by these methods, or, indeed, with any hand-coded approach. In order to address this, we describe an example-based image processing algorithm in Chapter 7. Given an example of a painting or drawing (e.g. scanned from a hand-painted source), we can process new images with some approximation to the style of the original, without requiring any explicit definition of the style. This example-based framework supports many other novel image processing operations, such as "texture-by-numbers," where we synthesize new realistic imagery from example photographs.

Our work focuses on the substrate required to make high-quality NPR tools, although some user interface will also be necessary for artists to make use of the styles.

1.2 On research in non-photorealistic rendering

One challenge of NPR research is that difficult to perform meaningful scientific tests in a field where judgments are ultimately subjective. For example, one could run user studies and find out whether people enjoy some computer-generated paintings or think they might be hand-made, but these studies would be of relatively little use in predicting whether there exists any artist that would want to use the tools, and, if so, how many.[1] Hence, it is doubtful that a rigorous scientific methodology could be applied to non-photorealistic rendering research at a high level. Of course, some methodology can be applied to low-level questions, for example, comparing the speeds of various algorithms.

The difficulty of defining a scientific methodology does not make the work any less

[1]Such tests would be more appropriate if our research goal was to fully model painting, in order to pass a "painting Turing test," but this is far from our goal.

valuable, although the evaluation is ultimately somewhat subjective. Like all research, NPR requires a great deal of guesswork and stumbling around in the dark; only after experience does the community gain some feeling for which ideas are valuable. One can easily find examples of useful reasoning and research areas where exact quantitative measurements are impossible (e.g. in political science), and, conversely, quackery justified by plausible scientific methodology (e.g. in political science).

Another challenge for NPR research is to demonstrate that a method works well in a few specific cases, but is yet somehow general. Much of the best research in NPR creates some useful framework that can be applied to many cases. For example, the "graftals" work [KMN$^+$99] is demonstrated with a few cases that work very well, but the concept of discrete procedural illustration elements is a very powerful one. Although our stroke-based rendering work focuses on specific styles, we expect that it will give insight useful for other stroke-based rendering methods. The image analogies method is general in that it does not make assumptions about the stroke media (among other things), but, instead, makes assumptions about the statistical properties of the style.

As NPR research becomes more refined, it may become much more difficult to conduct. Current NPR research tends to take large steps by doing novel things. For example, most of the research described in this thesis represents the first attempts at solving several specific problems, such as automatic placement of curved brush strokes or learning painting filters. In these cases, it is generally easy to argue the benefits of the research because even the problem statements are novel. In the future, however, more and more work may appear that improves on existing results rather than exploring new territory; this work may be more difficult to evaluate.

Our work appropriates visual media styles that traditionally signify the notion of the artist as producing an object of ineffable genius (e.g. a painting) and automates it. In fact, automating these processes offers several "improvements" over real paint from a commercial standpoint: the painting is individualized and simple to create. Hence, one might wonder if the artist has been made obsolete. Fortunately, this is not the case.

Although this work implements a formalization of some artistic technique, any idea it expresses, visually or conceptually, is a product of those who create and use the painting tools. Purely technical skill with a brush does not make an artist. Digital tools reduce the effort and cost of producing imagery by automating otherwise tedious tasks. Moreover, since the tools are different, they are used differently, and the digital art form really becomes a distinct medium. For example, when the camera was invented, some worried that it would replace painters and make them obsolete. In fact, only portrait painting became obsolete (and, to this day, it still has a place); photography is now viewed as an art form, performed by an artist using the camera as an artistic tool. Many of these artistic technologies can be viewed as collaborations — just as a movie is a collaboration of sorts between a director and actors and stage crew, a digital painting may be seen as a collaboration between an artist and a programmer, even if they did not work directly together.

Chapter 2

Survey of Non-Photorealistic Rendering

In this chapter, we survey some of the relevant literature in non-photorealistic rendering. In general, NPR may be seen as any attempt to create images to convey a scene without directly rendering a physical simulation. In this view, computer-generated imagery is an open area to explore for art (e.g. the work of John Maeda and collaborators [MIT] and Snibbe and Levin [SL00]), and scientific visualization (e.g. Saito and Takahashi [ST90] and Ebert and Rheingans [ER00]). We focus here on work that takes a narrower view, namely, attempts to mimic traditional artistic methods and media, at least as a starting point.

Earlier surveys of non-photorealistic rendering include Lansdown and Schofield [LS95], Robertson [Rob97], Gooch [Goo98], the SIGGRAPH 1999 course on NPR [CGG+99], and Craig Reynolds' comprehensive web page of NPR links [Rey].

We begin by surveying "low-level" problems, namely, silhouette detection brush models, and paint models. We then discuss interactive systems, and, finally, various automatic and semi-automatic NPR methods.

2.1 Silhouette and Hidden-Line Algorithms

In this section, we survey the purely technological problem of finding visible silhouettes of 3D surfaces. The problem of extracting silhouettes and related curves from 3D surfaces dates back to the beginnings of computer graphics: the first 3D renderings were silhouette illustrations (e.g. [Wei66]). Silhouettes are important both perceptually and aesthetically. Efficient silhouette detection is an interesting technical problem, because silhouettes typically occur only in a few places on a 3D shape at any given moment [KW97, MKT+97].

Intuitively, silhouette curves lie at the intersection of front-facing (locally visible) and back-facing (invisible) regions of a surface. For polygonal meshes, the silhouette consists of all edges that connect back-facing polygons to front-facing polygons. For smooth surfaces, the silhouettes can be found as those surface points $\mathbf{x}$ with a surface normal $\mathbf{n}$ perpendicular to the view vector (Figure 2.1). Under perspective projection, this is written as:

$$\mathbf{n} \cdot (\mathbf{x} - \mathbf{C}) = 0$$

where $\mathbf{C}$ is the camera center. Note that this definition makes no mention of visibility; a point that is occluded by another object is still considered a silhouette point by this definition. In almost every case, we are only interested in rendering the visible segments of silhouette curves, or in rendering the invisible sections with a different line quality.

For orthographic projection, all view vectors are parallel, and the equation can be written: $\mathbf{n} \cdot \mathbf{v} = 0$, where $\mathbf{v}$ is the view vector.

With one exception [NM00], existing silhouette algorithms can be divided into two classes: methods that operate on 3D surfaces, and methods that operate on the image plane.

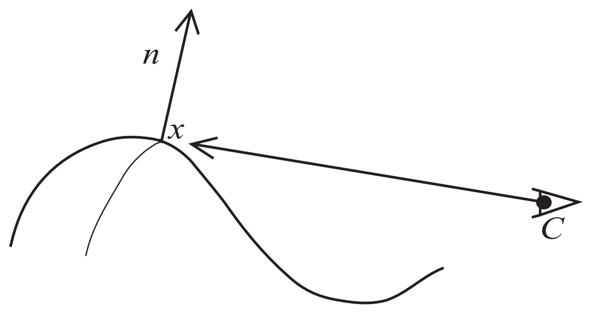


Figure 2.1: For smooth surfaces, the silhouette is the set of points for which the surface normal is perpendicular to the view vector.

2.1.1 Object Space Methods

For polygonal meshes, the brute force method for silhouette detection entails looping over all edges, and testing the condition described above. For orthographic projection of static meshes, this process may be accelerated by precomputing a Gauss map for the surface [GSG$^+$99, BE99]. The Gauss map is a function that maps each point on the surface to its normal direction; it may be thought of as a surface representation over S^2 (the sphere of directions), where each point in the sphere corresponds to the direction from the origin to that point. In practice, the Gauss map is usually represented as a cube. During a pre-processing phase, every edge in the mesh is inserted into the map, as an arc between the normals of the adjacent faces.

Two different hierarchies have been proposed for accelerating silhouette detection under perspective projection. First, one may use the projective dual of a surface [HZ00, BDG$^+$99]; the silhouettes of a surface correspond exactly to a plane-surface intersection in the dual space. Hence, one can represent the projective dual of a surface and then detect silhouettes via efficient surface/plane intersection tests. Second, Sander et al. [SGG$^+$00] describe a surface hierarchy that geometrically clusters polygons; the normals for each cluster in the hierarchy are represented by an approximating cone. Both methods appear to provide similar performance, although the dual surface method appears to be simpler to implement and to require an order of magnitude less preprocessing time.

Markosian et al. [MKT$^+$97] observe that silhouettes represent only a very small fraction of the total number of edges, and use a non-deterministic algorithm to detect silhouettes. Silhouettes are then traced along the surface until the curve becomes invisible. Although this method does not guarantee that all silhouettes will be detected, it may be more suitable for dynamically-changing meshes than a method that requires preprocessing. Bremer and Hughes [BH98] adapted this method to implicit surfaces. Northrup and Markosian [NM00] clean up the detected silhouettes by combining nearby edges and discarding invisible silhouettes, with the help of reference images rendered by graphics hardware.

Other silhouette detection algorithms operate on parameterized patches. Elber and Cohen [EC90] adaptively subdivide patches to locate piecewise linear approximations to silhouette curves, and then refine the curves. Gooch [Goo98] provides a simpler variation on this, using linear interpolation on edges of the control mesh.

A variety of algorithms have been devised for hidden-line elimination. Most methods are based on ray casting, together with the notion of Quantitative Invisibility devised by Appel [App67] to exploit the continuity of visibility: the visibility of a curve can only change at certain locations, such as when it passes under another curve. The visibility of a few points is established and visibility is propagated along the curves. However, for most smooth surface representations, efficient and correct visibility of silhouettes remains an open problem, since precise ray tests can be very expensive.

2.1.2 Image Space Methods

Image space methods are based on rendering an image from the target viewpoint, and performing some processing to generate an approximation to the silhouettes. Visibility is handled by the hardware rendering. Most of these methods are fast, and make use of graphics hardware for acceleration. The simplest of these methods was presented by Gooch et al. [GSG$^+$99] and Raskar and Cohen [RC99]. Front faces are rendered as wireframes into the stencil buffer. Then back faces are rendered into the image as wireframes, using the stencil. This produces an image containing only the mesh silhouettes, which consist of all mesh edges that connect a front-facing polygon to a back-facing polygon. This method is simple, but not very flexible, although line thicknesses can be controlled.

Other image space methods are based on rendering the surface with a smoothly varying function over the surface. These functions include depth [Cur98, Dec96, ST90], surface normals [Dec96], and texture coordinates [CJTF98a]; although, in principle, any smooth function could be used. Edges in this image can then be detected and rendered (Figure 2.2). These methods often give good results, but require some manual adjustment of parameters. Furthermore, it is fundamentally difficult to quantify when an edge in an image corresponds to an actual silhouette; at the corner of a flat folded piece of paper, none of the above functionals will yield an edge.

2.2 Stroke and Paint Models

The simplest stroke models are based on stroke textures. This method is often used in commercial paint products [Sys, Fra] and in [Mei96, Lit97, Hae90]. A single stroke texture is created in advance, and repeatedly copied to the target image, with various orientations and colors. Despite the simplicity of this method, it can be used for a wide variety of stroke types and materials (ink, paint, crayon, charcoal, etc.) The repetition of a single texture is usually visible in the output image; in this case, multiple textures should be used.

Curved strokes are usually represented as parametric curves with offsets to represent stroke thicknesses [HL94, NM00]. Strassmann [Str86] simulates brush hairs as a collection of smooth curves traced around the path of a curve; this model is abstracted

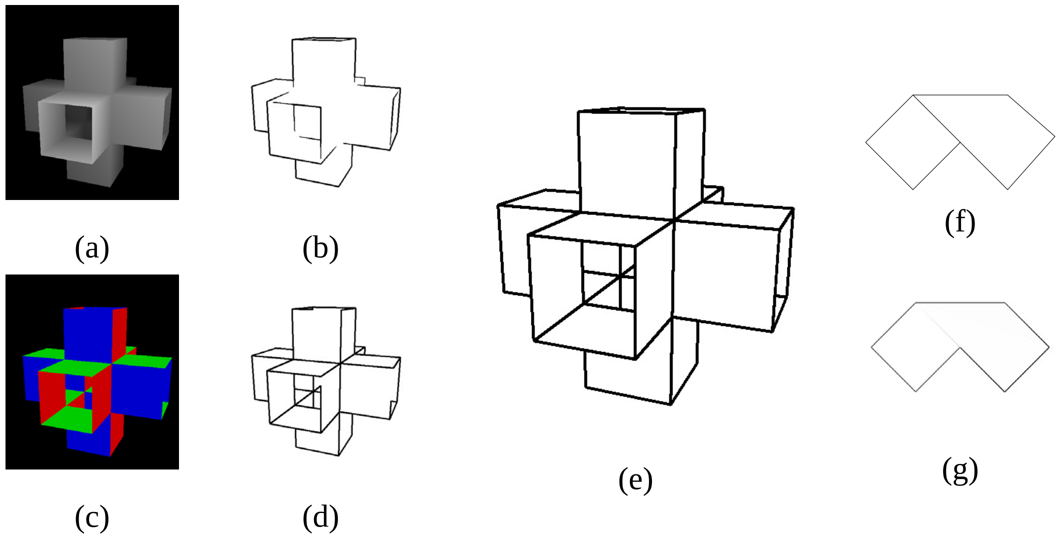


Figure 2.2: Outline drawing with image processing. (a) Depth map. (b) Edges of the depth map. (c) Normal map. (d) Edges of the normal map. (e) The combined edge images. (f) A difficult case: a strip of paper, folded flat. (g) Depth edges. The interior silhouette disappears at the crease.

by Pham [Pha91] and Guo [Guo95]. The product (formerly known as) Fractal Design Painter [Fra] achieves many beautiful stroke texture effects; unfortunately, there do not appear to be any published details. A very stylized appearance is achieved by Hsu et al. by the use of skeletal strokes [HL94], in which an arbitrary image is warped to a stroke shape.

Impressive effects may be achieved by simulating the physics of paint. Cockshott et al. [CPE92] simulate the build-up of oil paint on a canvas. Small [Sma90] describes a simple system for simulating watercolor. Curtis et al. [CAS$^+$97] simulate the flow of watercolor on a surface, modeling the interaction of water pigment and paper, and produce a wide variety of realistic effects. Sousa and Buchanan [SB00] simulate the appearance of graphite pencils.

2.3 Interactive Painting and Drawing

Perhaps the most prominent and widely-used non-photorealistic rendering systems are digital paint systems [Smi82, Smi97, Sys, Fra, Str86, Pha91, Guo95, HL94, CPE92, CAS$^+$97, Mic]. These systems automate few or no painting decisions, giving the artist direct control over the production of images. Multiresolution painting schemes are presented in [BBS94, PV92, PV95]. Painting methods have been adapted for painting 3D images with depth buffers in the Piranesi system [LS95, Lim], and by Teece [Tee98] to allow a user to paint a 3D scene to generate still images or animations. Hanrahan and Haeberli [HH90] provide a tool for painting textures directly onto 3D objects; this idea has been implemented in several commercial products [Rob98]. Haeberli [Hae90] presented a variety of techniques to aid painting, including the use of an underlying source image to determine stroke properties during painting. Salisbury et al. [SABS94] describe an interactive drawing system using stroke textures. Salisbury et al. [SWHS97] later introduced orientable stroke textures, as part of an interactive pen-and-ink image processing system, in which the user specifies stroke types and orientations. As in Haeberli [Hae90], a reference image can be used to specify stroke attributes. Ostromoukhov [Ost99] describes an engraving system in which engraving directions are chosen interactively; the engraving is created by modeling the printing plate as a height field, and filling it with ink. Similarly, Mizuno et al. [MOiT98] create woodcuts interactively.

2.4 Automatic Painting Algorithms

Most automatic painting algorithms process images to produce paintings. The simplest of these apply continuous image processing techniques to produce the appearance of paint, watercolor, etc. Examples of this include diffusion operators [ZM98] and line-integral convolution [CL93]; many similar filters are implemented in commercial software [Sys].

A somewhat more sophisticated approach is to generate a grid of brush stroke textures [Sys, Fra, Too, Mic, TC97]. Stroke colors are determined by sampling the source image, and stroke orientations are chosen to be normal to local image gradients, as in [Hae90]. In order to reduce the blur in these images, the strokes may be clipped to edges of the source image [Lit97]. Shiraishi and Yamaguchi [SY00] describe a stroke placement algorithm that uses local image moments instead of gradients to determine local stroke orientation, and a dithering algorithm to plan stroke placements. Haeberli [Hae90] also introduced a relaxation method for NPR.

A central problem of painterly animation is to balance temporal coherence with fidelity to the painting style and the video input. A simple method is to render a movie, and then apply a painterly filter to each video frame independently. This method produces severe flickering artifacts, making it essentially unusable. Another approach is to apply painted texture-maps to each scene object (e.g. [601]). This gives objects a "shrink-wrapped" look, and appearance quickly degrades as the camera position moves away from the initial position. Klein et al. [KLK$^+$00] improve on this by creating multiple scales of NPR texture maps, so that non-photorealistic features (e.g. strokes) keep consistent scales.

To compensate for these problems, Meier [Mei96] describes a method for painterly animation from 3D models, by placing particles on the 3D objects, and then rendering brush strokes at each particle. Brush stroke attributes, such as color, orientation, and depth, are taken from the 3D surfaces themselves. To some extent, this method gives the sense of 3D brush strokes moving in space, thus defeating the sense of looking at a 2D illustration of a 3D scene—we are now looking at a modified 3D scene.

Litwinowicz [Lit97] processes existing video sequences by computing optical flow, and displacing brush strokes along optical flow vectors. Strokes are added and deleted in order to keep the stroke density roughly constant. This use of optical flow is similar to Meier's use of particles. However, this method does not have the quasi-3D quality of Meier's animations, because operations are perfomed in the image plane, and brush strokes are not locked to the 3D motion. This method is limited by the quality of the optical flow that is used, which leads to some shimmering artifacts in the output. The true motion field can be used for synthetic video sequences.

Another approach is taken by Curtis [Cur98], in which an edge image is approximated with loose and sketchy pen strokes. Temporal coherence is not a significant problem in this case, because the algorithm is only drawing outlines, and not filling the image plane.

Many of these ideas were combined for *What Dreams May Come* [Lit99]. Daniels [Dan99] describes Deep Canvas, a system for interactively painting 3D models with brush strokes that automatically update for new camera views; this system was used for the 1999 Disney animated feature *Tarzan*.

2.5 Automatic Pen-and-Ink Algorithms

Pen-and-ink illustration uses thin pen strokes to convey images. In order to express tone, direction, and texture, pen strokes can be thickened, oriented, and hatched. Winkenbach and Salesin [WS94, WS96] demonstrate a powerful system for generating line drawings of polygonal and parametric surfaces. Visibility is computed by rendering in depth-first order and partitioning the image plane. Each visible surface is hatched according to texture, shading and orientation, using the hatching methods of Salisbury et al. [SABS94]. Similar methods have been described to handle non-parametric surfaces [Elb98, CG97, MKT$^+$97].

Several different methods for choosing hatch directions on surfaces have been proposed; each of these methods has limitations. Winkenbach and Salesin [WS96] and Elber [Elb98] use an intrinsic surface parameterization. However, this method only works when the surface has been designed with a meaningful parameterization; for arbitrary surfaces, there is no guarantee that a surface parameterization will produce good hatch directions, or that one exists at all. Several authors have have also experimented with principal curvature directions [Elb98, Int97, GIHL00, RK00]. These fields work very well when a surface has two distinct curvature magnitudes; for regions with identical curvature magnitudes (i.e. umbilical points), the principle curvature directions are not uniquely defined. Rössl and Kobbelt [RK00] interactively partition the surface into regions with valid principal direction fields, and then skeletonize and hatch each region separately. Deussen et al. [DHR+99] compute more global structures based on a skeleton computed for the mesh; however, this method does not show fine-scale detail in the hatching. Elber has also experimented with fields parallel to a single vector; however, they do not represent shape very effectively. Markosian et al. [MKT+97] hatch in the direction of the cross product of the surface normal to the tangent vector. This produces useful hatches in a narrow region near the silhouette, but produces a poor cross hatch when rotated 90 degress. Veryovka and Buchanan [VB99] modify halftoning algorithms for use with 3D surfaces, producing results similar to surface illustration, although the hatch directions are quantized. Deussen et al. [DS00] describe a technique specially adapted for pen-and-ink illustration of trees.

2.6 Technical Illustration and Cartoons

Gooch et al. [GGSC98, Goo98, GSG+99] have developed several methods for rendering objects for technical illustration. The primary contributions are a new lighting model that mimics artistic shading, and conventions for line drawing, surface shadowing, and interactive rendering.

Computer-assisted cartoon animation is widely used in television and feature films; nowadays, nearly all cartoons use digital ink-and-paint and compositing instead of traditional ink-and-paint and multiplaning. Some systems, such as TicTacToon [FBC+95], model and animate 2D line art, whereas a method used by Disney [Gou97] generates cel animation-style images of 3D models. Real-time 'toon shading can be implemented on graphics hardware (e.g. [LMHB00]). Corrêa et al. [CJTF98b] describe a technique and user interface for applying textures to cel animations. Petrovic et al. [PFWF00] describe a technique and user interface for specifying a ground-plane for cel-animation shadows.

Kowalski et al. [KMN$^+$99, MMK$^+$00] and Kaplan et al. [KGC00] describe a procedural NPR method called graftals. Graftals are non-photorealistic textures that consist of a random placement of discrete elements, e.g. strokes.

2.7 Example-Based Rendering

Very little work has been done on capturing and reusing non-photorealistic rendering styles from examples. Hamel and Strothotte [HS99] estimate line art rendering styles as histogram densities from drawings created with a line art renderer. Freeman et al. [FTP99] synthesize stroke shape from example strokes.

2.8 User Interfaces

A few 3D modeling systems have been described that allow users to create objects and scenes by sketching in a manner similar to how one might with a pen. These systems provide fast, informal interfaces while being fun to use. Here we survey some of the most prominent systems from the graphics literature.

“SKETCH” [ZHH96] provides a gestural language for quickly sketching and editing 3D shapes, while “Teddy” [IMT99] provides a simple method for sketching and editing freeform shapes. “Harold” [CHZ00] provides a simple interactive scene sketcher, where one draws objects as “billboards.” in space. Meyer and Bederson [MB98] peformed user studies that show that users draw differently with a sketchy user interface than with a standard line drawing interface. Sketchy interfaces are an active area of work in the user interface community.

2.9 Shape and Cameras for NPR

Most 3D NPR work has focused on rendering a given scene and camera model; few methods have been developed to model the distortions often seen in paintings and drawings. Rademacher [Rad99] non-rigidly interpolates between view-dependent shapes; Martín et al. [MGT00] provide more structured deformation controls. Agrawala et al. [AZM00] and Levene [Lev98] combine different camera projections for different objects into a single image. Similarly, Wood et al. [WFH$^+$97] combine many views into an image that can be used to simulate 3D camera motion by a sequence of views of a single 2D image.

Chapter 3

Fast and Loose Painterly Image Processing

We now present a new method for creating an image with a hand-painted appearance from a photograph, and a new approach to designing styles of illustration.[1] We “paint” an image with a series of spline brush strokes. Brush strokes are chosen to match colors in a source image. A painting is built up in a series of layers, starting with a rough sketch drawn with a large brush. The sketch is painted over with progressively smaller brushes, but only in areas where the sketch differs from the blurred source image. Thus, visual emphasis in the painting corresponds roughly to the spatial energy (e.g. edges) present in the source image. We paint with long, curved brush strokes, along image contours. Thus we begin to explore the expressive quality of complex brush strokes. The style of the painting may be changed by modifying parameters to the painting algorithm.

3.1 Layered Painting

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

[1]An earlier version of this material was presented in [Her98].

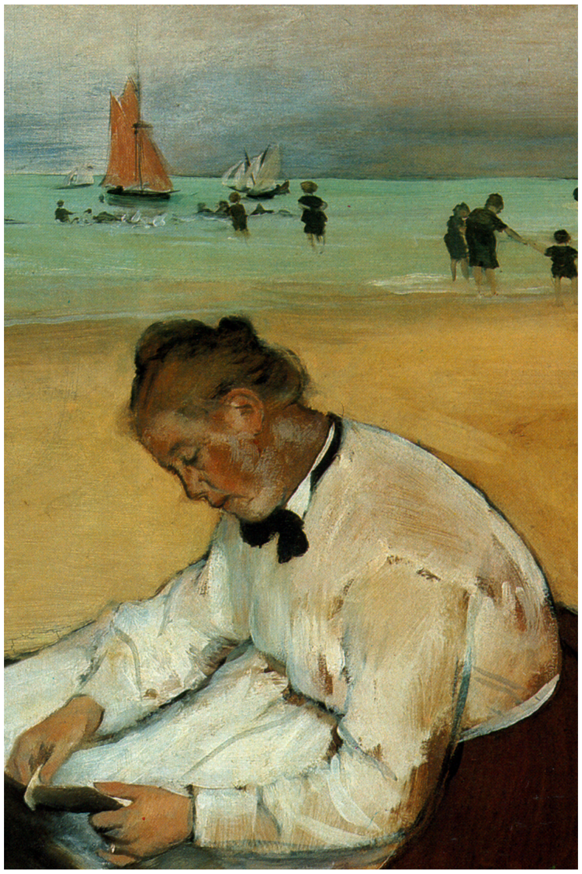


Figure 3.1: Detail of *At The Seashore, Young Woman having her Hair Combed by her Maid*, Edgar Degas, 1876-7. Note that the small brush strokes are used only in regions of fine detail (such as the children in the background), and that they draw attention to these regions.

not apply to computer algorithms[2], it also yields desirable visual effects. In Figure 3.1, note the different character of painting used to depict the blouse, sand, children, boat and sky. Each has been painted with different brush sizes as well as different stroke styles. This variation in stroke quality helps to draw the most attention to the woman and figures, and very little to the ground; in other words, the artist has used fine strokes to draw our attention to fine detail. (Other compositional devices such as shape, contrast and color are also used. These are not addressed in this chapter.) To use strokes of the same size for each region would "flatten" the painting; here the artist has chosen to emphasize the figure over the background. In our image processing algorithm, we use fine brush strokes only where necessary to refine the painting, and leave the rest of the painting alone, although we also allow user-controlled emphasis. Our 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.

Our painting 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. The initial canvas is a constant color image.

For each layer, we first create a reference image 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.

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

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, such as when processing video (as described in the next chapter), 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; initially blank for still images
        $R_1 ... R_n$, // brush sizes
        firstFrame) // boolean; true for still images
    Create a summed-area table $A$ from $I_s$ if necessary
    refresh ← firstFrame
    foreach brush size $R_i$, from largest to smallest, do
        Compute a blurred reference image $I_{R_i}$ with blur size $f_\sigma R_i$
            from $A$ or by convolution
        grid ← $R_i$
        Clear depth buffer
        foreach position $(x, y)$ on a grid with spacing grid
            M ← the region $[x - grid/2 ... x + grid/2,$
                $y - grid/2 ... y + grid/2]$
            areaError ← $\sum_{(i,j) \in M} \|I_p(i,j) - I_{R_i}(i,j)\|$
            if refresh or areaError $> T$ then
                $(x_1, y_1) \leftarrow \arg\max_{(i,j) \in M} \|I_p(i,j) - I_{R_i}(i,j)\|$
                PAINTSTROKE($x_1, y_1, I_p, R_i, I_{R_i}$)
        refresh ← false
```

Each layer is painted using a simple loop over the image canvas. The approach is adapted from the algorithm described in [Lit97], which placed strokes on a jittered grid. That approach may miss sharp details such as lines and points that pass between grid points. Instead, we search 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 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 3.2.

$\|\cdot\|$, when applied to a color vector, denotes Euclidean distance in RGB space: $\|(r_1, g_1, b_1) - (r_2, g_2, b_2)\| = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$. We have 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 (x_1, y_1), given a reference image and a brush radius. Following [Hae90], Figure 3.3(a) shows an image illustrated using a PAINTSTROKE procedure which simply places a circle of the given radius at (x, y), using the color of the source image at location (x, y). Following [Lit97], Figure 3.3(b) shows an image illustrated with short brush strokes, aligned to the normals of image gradients.[3] 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.

Our 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. We explore variations of emphasis further in Section 3.3.3 and Chapter 6.

[3]Note that no stroke clipping is used as in [Lit97]. Instead, small scale refinements of later layers automatically "fix" the edges of earlier layers.

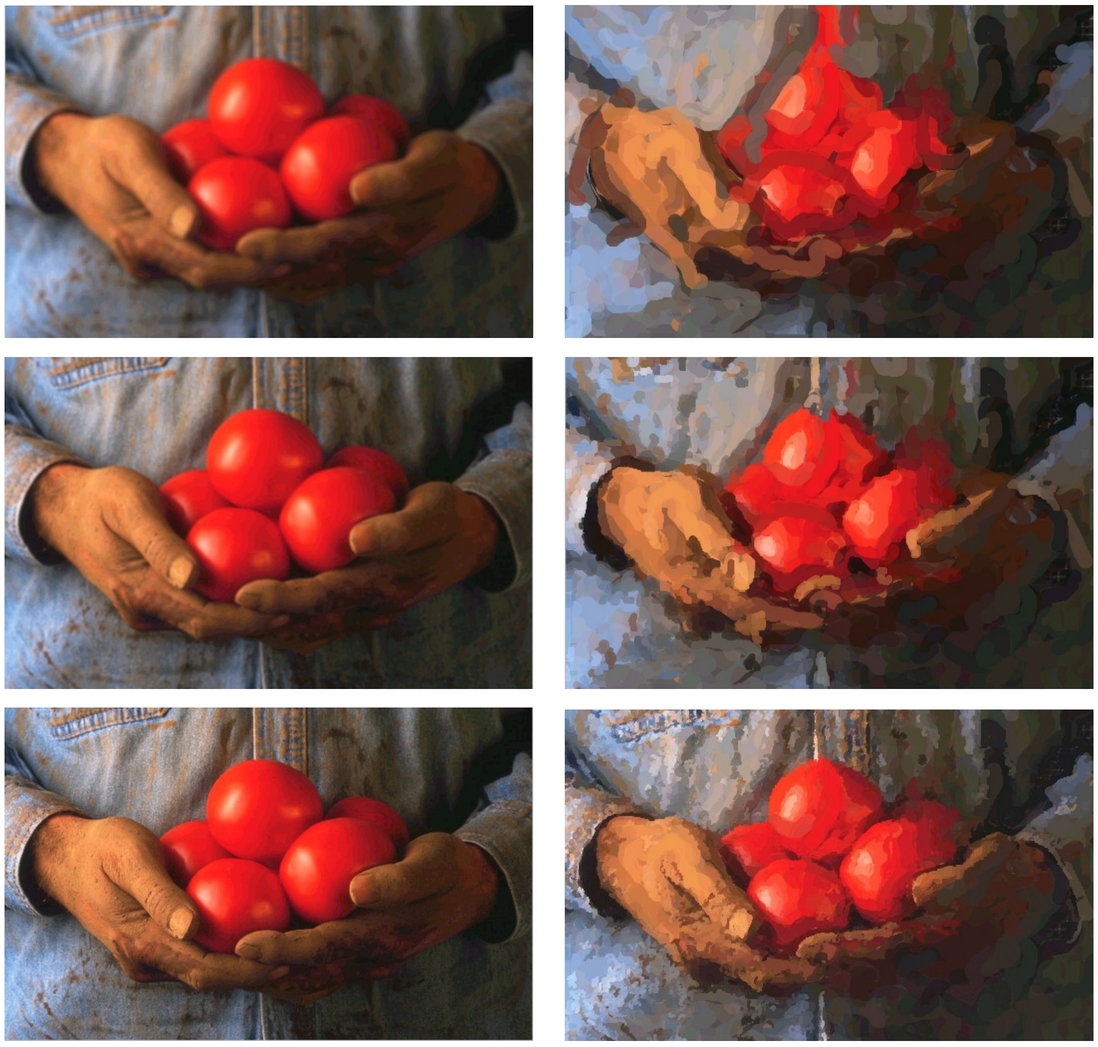
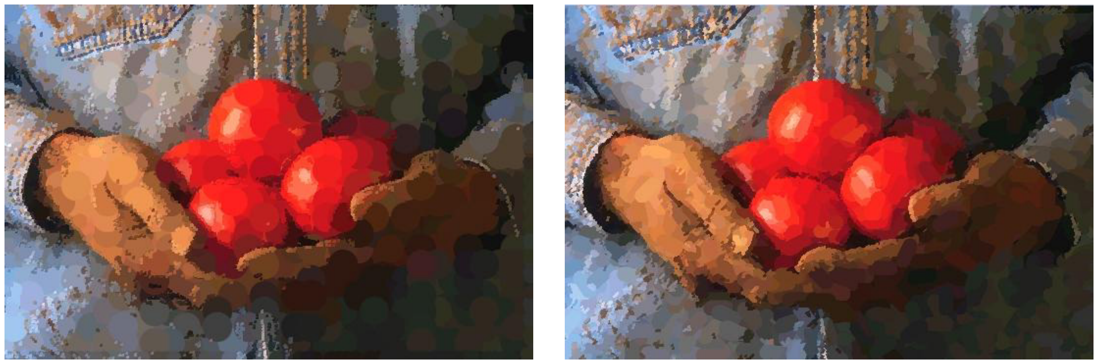
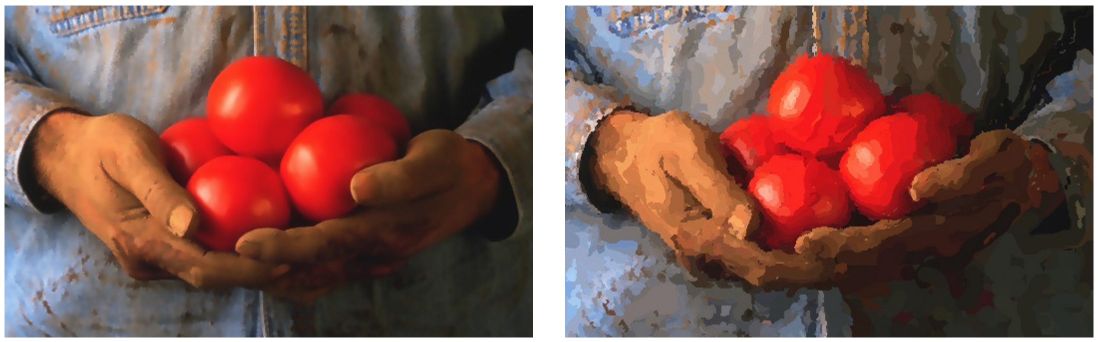


Figure 3.2: Painting with three brushes. The left column shows the reference images; the source image is shown in the lower left. The right column shows the painting after the first layer (brush radius 8), the second layer (radius 4), and the final painting (radius 2). Note that brush strokes from earlier layers are still visible in the final painting. (The curved stroke placement is described in the next section).



(a) (b)

Figure 3.3: 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 3.4: Using non-linear diffusion to create reference images. (a) Reference image for coarsest level of the pyramid. (Compare to Figure 3.2(e)). (b) The output image is less noisy, but hard edges are maintained.

3.2 Creating curved brush strokes

Individual brush strokes in a painting can convey shape, texture, overlap, and a variety of other image features. There is often something quite beautiful about a long, curved brush stroke that succinctly expresses a gesture, the curve of an object, or the play of light on a surface. To our knowledge, all previous automatic painting systems that explicitly use strokes use small strokes, identical aside from color and orientation. We present a method for painting long, continuous curves. For now, we focus on painting solid strokes of constant thickness to approximate the colors of the reference image. We model brush strokes as anti-aliased cubic B-splines, each with a given color and thickness (Appendix 8.3).

In our system, we 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. 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 (x_0, y_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 (x_0, y_0) is added to the spline, and the color of the reference image at (x_0, y_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 is computed from the Sobel-filtered luminance[4] of the reference image. The next point (x_1, y_1) is placed in the direction $\theta_0 + \pi/2$ at a distance R from (x_0, y_0) (Figure 3.5). Note that we could have also used the direction $\theta_0 - \pi/2$; this choice is

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

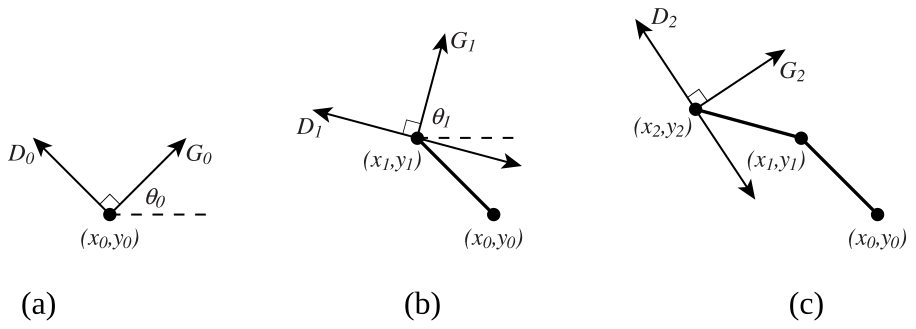


Figure 3.5: Painting a brush stroke. (a) A brush stroke begins at a control point (x_0, y_0) and continues in direction $\mathbf{D}_0$, normal to the gradient direction $\mathbf{G}_0$. (b) From the second point (x_1, y_1), there are two normal directions to choose from: $\theta_1 + \pi/2$ and $\theta_1 - \pi/2$. 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 (x_i, y_i) as control points. The distance between control points is equal to the brush radius.

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 (x_i, y_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 3.5), 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 applying an infinite impulse response filter to 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:

```
**function** PAINTSTROKE($(x_0, y_0)$, $R$, $I_R$, $I_p$)
    // Arguments: start point $(x_0, y_0)$, stroke radius ($R$),
    // reference image ($I_R$), painting so far ($I_p$)
    color $\leftarrow I_R(x_0, y_0)$
    K $\leftarrow$ a new stroke with radius $R$ and color color
    add point $(x_0, y_0)$ to K
    **for** $i = 1$ **to** maxStrokeLength **do**
        // compute image derivatives
        $(g_x, g_y) \leftarrow (255 * \frac{\partial Y_R}{\partial x}(x_{i-1}, y_{i-1}), 255 * \frac{\partial Y_R}{\partial y}(x_{i-1}, y_{i-1}))$

        // detect vanishing gradient
        **if** $R_i\sqrt{{g_x}^2 + {g_y}^2} \geq 1$    // is gradient times length at least a pixel?
            // rotate gradient by $90$ degrees
            $(\Delta x_i, \Delta y_i) \leftarrow (-g_y, g_x)$

            // if necessary, reverse direction
            **if** $i > 1$ **and** $\Delta x_{i-1} * \Delta x_i + \Delta y_{i-1} * \Delta y_i < 0$ **then**
                $(\Delta x_i, \Delta y_i) \leftarrow (-\Delta x_i, -\Delta y_i)$

            // filter the stroke direction
            $(\Delta x_i, \Delta y_i) \leftarrow f_c * (\Delta x_i, \Delta y_i) + (1 - f_c) * (\Delta x_{i-1}, \Delta y_{i-1})$
        **else**
            **if** $i > 1$
                // continue in previous stroke direction
                $(\Delta x_i, \Delta y_i) \leftarrow (\Delta x_{i-1}, \Delta y_{i-1})$
            **else**
                **return** K

        $(x_i, y_i) \leftarrow (x_{i-1}, y_{i-1}) + R_i * (\Delta x_i, \Delta y_i)/\sqrt{{\Delta x_i}^2 + {\Delta y_i}^2}$
        **if** $i >$ minStrokeLength **and**
            $\|I_R(x_i, y_i) - I_p(x_i, y_i)\| < \|I_R(x_i, y_i) - color\|$ **then**
            **return** K
        add $(x_i, y_i)$ to K
    **end for**
    **return** K
```

$Y_R(x, y)$ is the luminance channel of I_R, scaled from 0 to 1.

For painting styles without random color perturbations, a slight speedup may be gained by performing all computations in luminance space, and then restoring the color information. The results are very nearly as good as full color processing. See Section 7.1.5 for more information.

3.3 Rendering Styles

By changing parameters to the painting algorithm, we can create new painting styles. A specific set of values for parameters may be encapsulated as a "painting style." If the parameters are well-behaved, styles may be interpolated and extrapolated by interpolating and extrapolating the corresponding parameters. A group of style parameters describes a space of styles; a set of specific values can be encapsulated in a style. Styles can be collected into libraries, for later use by designers. Some commercial painterly rendering products [Too, Mic] provide the ability to vary rendering parameters and to save sets of parameters as distinct styles.

3.3.1 Some style parameters

In the experiments that follow, we have used the following style parameters.

- Approximation threshold (T): How closely the painting must approximate the source image. Higher values of this threshold produce "rougher" paintings. This may be replaced with a spatially-varying weight image $T(x, y)$ (see next section).
- Brush sizes: Rather than requiring the user to provide a list of brush sizes ($R_1 ... R_n$), we have found it more useful to use two parameters to specify brush sizes: Smallest brush radius (R_1), and Number of Brushes (n). We then set $R_i = 2R_{i-1}$ for $i \in [2...n]$. We have found that a limited range of brush sizes often works best, e.g. $n \leq 3$.
- Curvature Filter (f_c): Used to limit or exaggerate stroke curvature.
- Blur Factor (f_σ): Controls the size of the blurring kernel. A small blur factor allows more noise in the image, and thus produces a more "impressionistic" image.

- Minimum and maximum stroke lengths ($minStrokeLength, maxStrokeLength$): Used to restrict the possible stroke lengths. Very short strokes would be used in a "pointillist" image; long strokes would be used in a more "expressionistic" image.

- Opacity (α): Specifies the paint opacity, between 0 and 1. Lower opacity produces a wash-like effect.

- Color Jitter: Factors to add random perturbations to hue (j_h), saturation (j_s), value (j_v), red (j_r), green (j_g) or blue (j_b) color components. 0 means no random jitter; larger values increase the factor.

The threshold (T) is defined in units of distance in color space. Brush sizes are defined in units of distance; we specify sizes in pixel units, although resolution-independent measures (such as inches or millimeters) would work equally well. Brush length is measured in the number of control points. The remaining parameters are dimensionless.

3.3.2 Experiments

In this section, we demonstrate five painting styles: "Impressionist," "Expressionist," "Colorist Wash," "Pointillist," and "Psychedelic." Figures 3.6, Figure 3.7, and Figure 3.8 shows the application of the first three of these styles to two different images.

Figure 3.9 shows a continuous transition between the "Pointillist" style and the "Colorist Wash" style. By interpolating style parameter values, we can "interpolate" the visual character of rendering styles.

The styles are defined as follows.

- "Impressionist": a normal painting style, with no curvature filter, and no random color. $T = 100, R = (8, 4, 2), f_c = 1, f_\sigma = .5, \alpha = 1, f_g = 1, minStrokeLength = 4,$
$maxStrokeLength = 16$

- “Expressionist”: elongated brush strokes. Jitter is added to color value. $T = 50, R = (8,4,2), f_c = .25, f_\sigma = .5, \alpha = .7, f_g = 1, minStrokeLength = 10, maxStrokeLength = 16, j_v = .5$

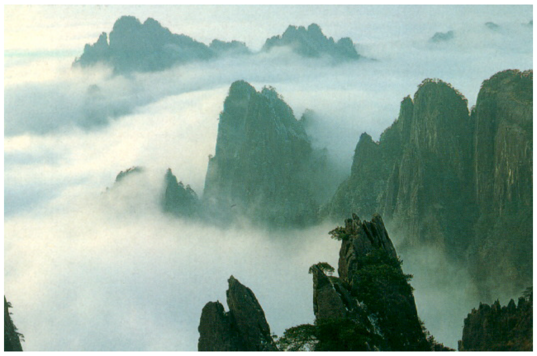
- “Colorist Wash”: loose, semi-transparent brush strokes. Random jitter is added to R, G, and B color components. $T = 200, R = (8,4,2), f_c = 1, f_\sigma = .5, \alpha = .5, f_g = 1, minStrokeLength = 4, maxStrokeLength = 16, j_r = j_g = j_b = .3$

- “Pointillist”: densely-placed circles with random hue and saturation. $T = 100, R = (4,2), f_c = 1, f_\sigma = .5, \alpha = 1, f_g = .5, minStrokeLength = 0, maxStrokeLength = 0, j_v = 1, j_h = .3$.

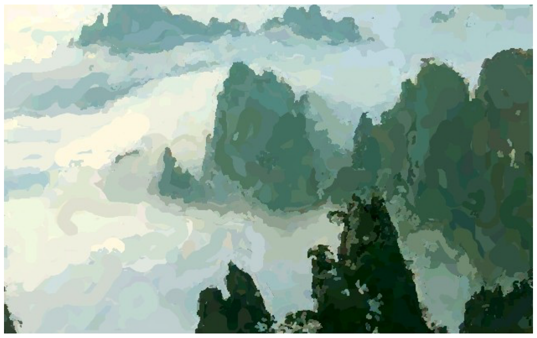
- “Psychedelic”: expressionistic and colorful. $T = 50, R = (8,4,2), f_c = .5, f_\sigma = .5, \alpha = .7, f_g = 1, minStrokeLength = 10, maxStrokeLength = 16, j_h = .5, j_s = .25$

3.3.3 Weight Images

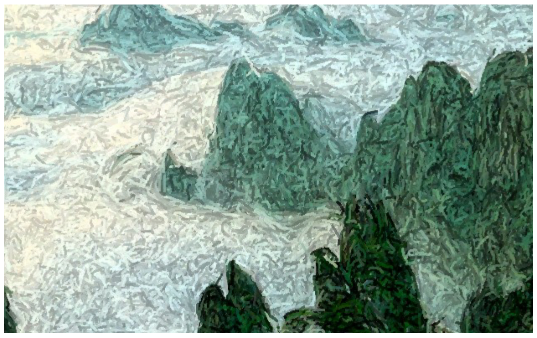
Different styles may be defined for different parts of the image, by a user or some other procedure. We have explored one version of this, where the stroke placement threshold T is replaced with a spatially-varying weight image $T(x, y)$ (Figure 3.10). The PAINT function is modified so that the decision of whether to place a stroke or not is determined by comparing *areaError* to $T(x_1, y_1)$ instead of to T. Due to the approximate nature of the stroke algorithm, the style mixing is rather loose; higher-quality variations are explored in Chapter 6.



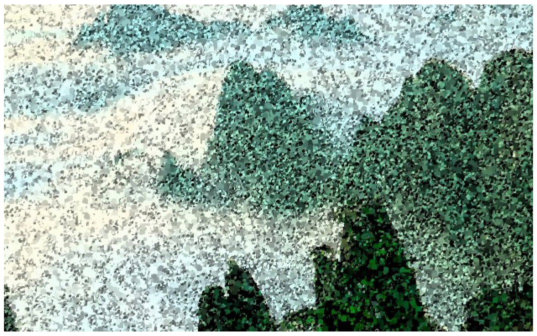
Source image



“Impressionist”

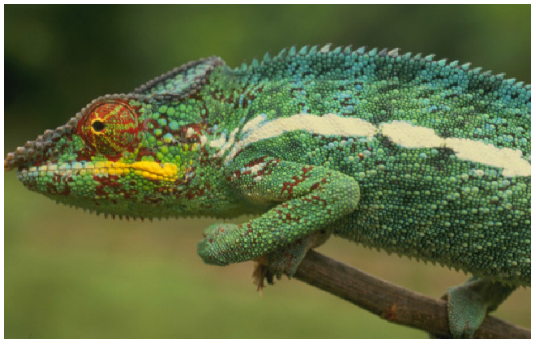


“Expressionist”

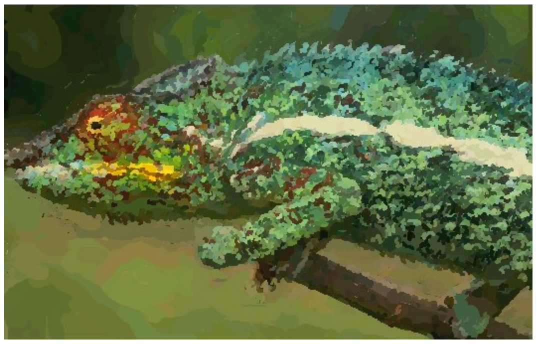


“Pointillist”

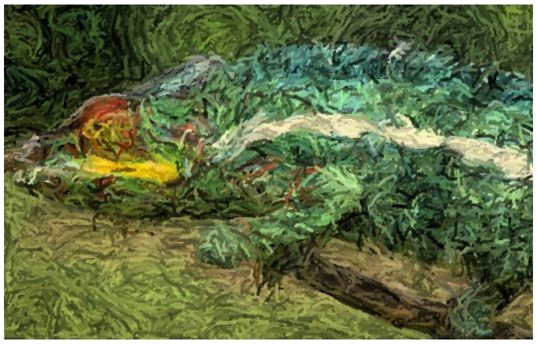
Figure 3.6: Three painting styles



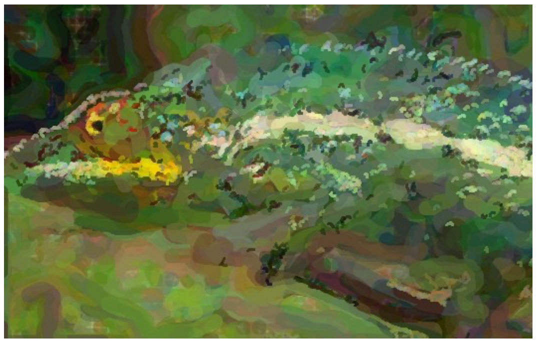
Source image



“Impressionist”

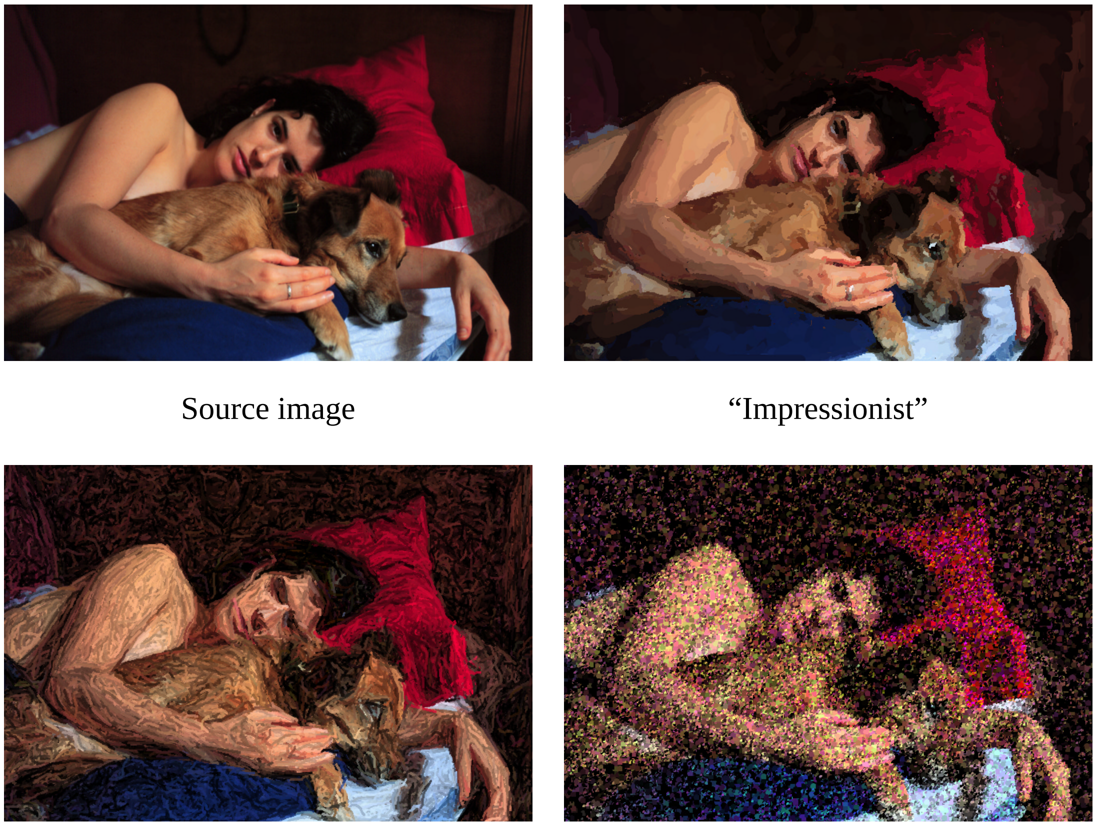


“Expressionist”



“Colorist Wash”

Figure 3.7: Three painting styles.



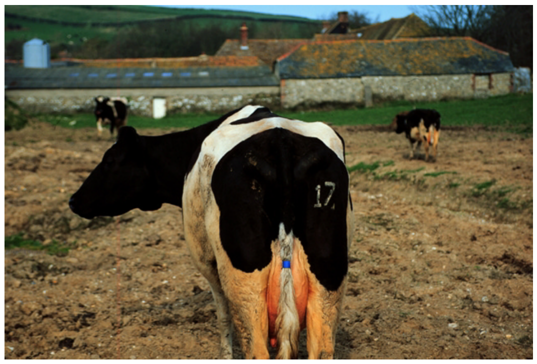
Source image

“Impressionist”

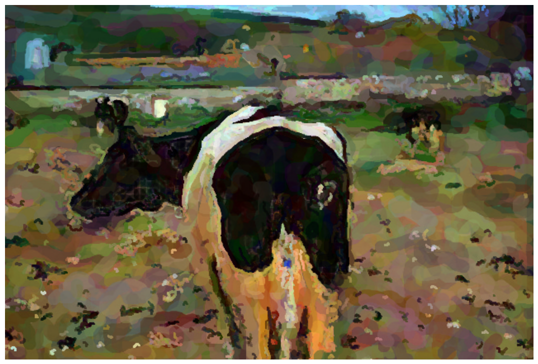
“Expressionist”

“Pointillist”

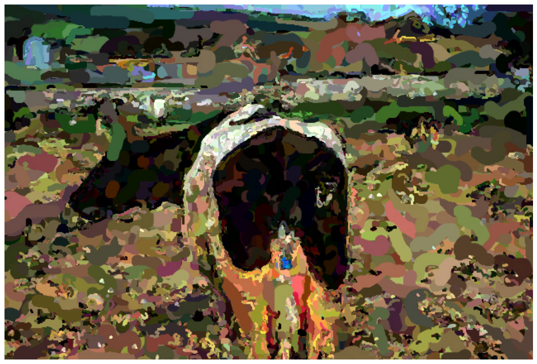
Figure 3.8: Three painting styles



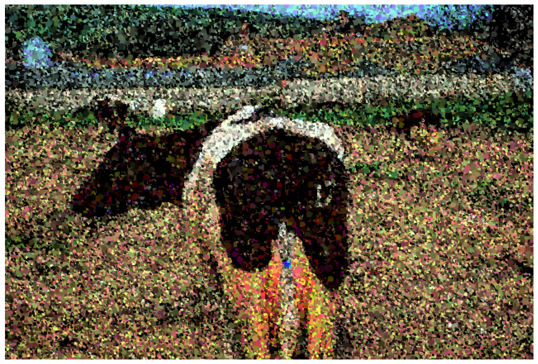
Source image



“Colorist Wash”



Average style



“Pointillist”

Figure 3.9: Style interpolation. The style of the painting on the lower left is the average of the two styles on the right.

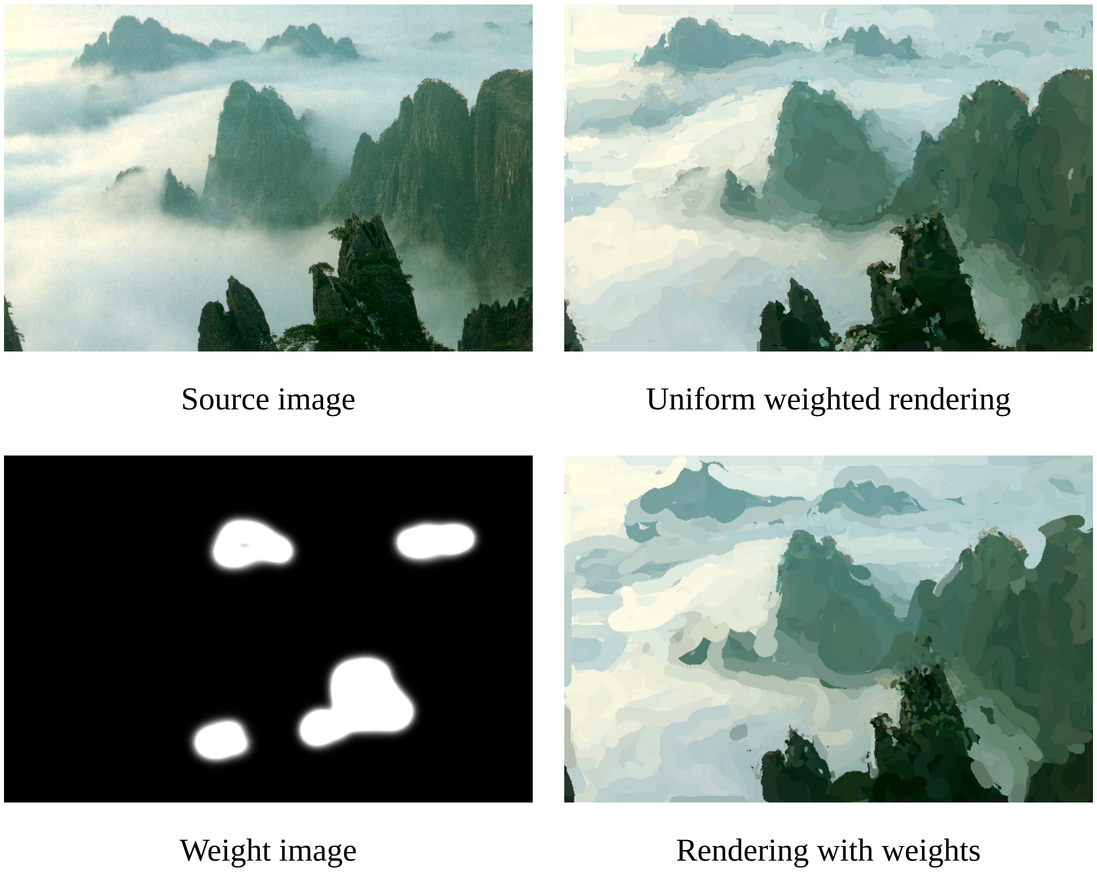


Figure 3.10: Painterly rendering with a user-defined weight image. The weight image allows a user to specify variations in style across the image; the weight image used here varies the looseness of the style. In this case, more detail has been specified for the mountain peaks. (A different version of the software was used to produce these images than was used for the other images in this chapter).

Chapter 4

Painterly Rendering for Video and Interaction

We now present methods for painterly video processing.[1] Based on the techniques from the previous chapter, we "paint over" successive frames of animation, applying paint only in regions where the source video is changing. Image regions with minimal changes, such as due to video noise, are also left alone, using a simple difference masking technique. Optionally, brush strokes may be warped between frames using computed or procedural optical flow.

These methods produce video with a novel visual style distinct from previously demonstrated algorithms. Without optical flow, the video gives the effect of a painting that has been repeatedly updated and photographed, similar to paint-on-glass animation. We feel that this gives a subjective impression of the work of a human hand. With optical flow, the painting surface flows and deforms to follow the shape of the world.

We have constructed an interactive painting exhibit, in which a painting is continually updated. Viewers have found this to be a compelling experience, suggesting the promise of non-photorealistic rendering for creating compelling interactive visual experiences.

[1]This chapter describes joint work with Ken Perlin, first presented in [HP00].

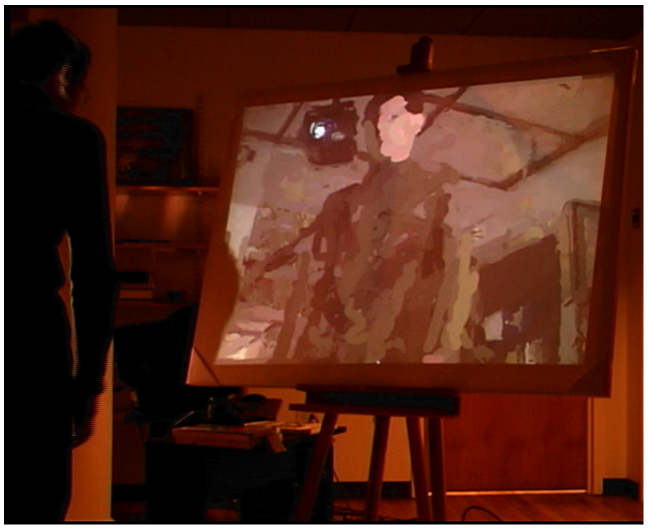


Figure 4.1: A viewer interacting with a “living” painting.

4.1 Painting Over

The simplest method for generating painterly video is to apply a still image filter to each frame independently. As has been previously observed in the literature, subtle changes in the input can cause dramatic changes in the output, creating severe flickering in the output video. The simplest case of flickering can be characterized as static areas of the scene that are painted differently in each frame. This flickering changes the character of the motion and distracts from the action in ways that are usually undesirable.

The algorithm presented in the previous chapter leads to a natural approach to improving temporal coherence. The first frame of the video sequence is painted normally. For each successive frame, we “paint over” the previous frame. In other words, the painting of the first frame is used as the initial canvas for the second frame. The PAINT procedure (Section 3.1) is called with this initial canvas and *firstFrame* is set to **false**. Consequently, unchanging regions of the video frame will be left unchanged when the painting style is reasonably faithful to the source image.

This method produces video with the appearance of a painting that has been repeatedly painted over and photographed. This style is similar to the paint-on-glass style in experimental animation, such as the work of Alexander Petrov or Georges Schwizgebel.

Though painting-over does improve temporal coherence, flickering remains a problem, when a static region of a video frame differs from the corresponding region of the previous painted frame. This discrepancy can occur in several cases. A significant source of error is video noise, which can cause static areas of the scene to be repainted each frame. Another source of difference is stylization and "sloppiness" in the painting: many painting styles produce images that radically diverge from the input. We address this problem by *difference masking*: we paint only in portions of the video which contain significant motion. We measure the sum of the differences between an image region M for the current frame and the corresponding region for the previous frame:

$$\mathit{frameDiff} = \frac{1}{|M|} \sum_{(i,j)\in M} \| I_{t+1}(i,j) - I_t(i,j) \|$$

where I_t and I_{t+1} are successive video frames. If this sum is below a threshold T_V, then no stroke may be placed for the region M. Hence, the conditional in the PAINT procedure in Section 3.1 becomes:

if *refresh* **or** (*frameDiff* $> T_V$ **and** *areaError* $> T$)

For interactive applications, we use a faster test that averages over an image region and compares this average to the previous frame's average:

$$\mathit{frameDiff} = \frac{1}{|M|} \left\| \sum_{(i,j)\in M} I_{t+1}(i,j) - \sum_{(i,j)\in M} I_t(i,j) \right\|$$

This test is computed in constant time using a summed-area table.

An example sequence is shown in Figure 4.3.

This difference masking method will not detect gradual changes over time, such as a fade-in or fade-out. We use a variation on this method for such input sequences, *cumulative difference masking*. A running cumulative sum of the region image difference is kept for each region M at each scale. The cumulative sum is updated at each frame. When the cumulative sum exceeds T_V, it is reset to zero and a stroke is painted.

4.2 Frame Rate

Video frame rate is an important consideration when producing painterly animation. In 30 Hz video, even minor flickering can become highly objectionable, and the paint-over method typically produces severe flickering in noisy or moving image regions. A more fundamental issue is that 30 Hz video can look “too real:” the underlying motion and shape is integrated so well by the human visual system that the video begins to take on the character of ordinary video with bad artifacts, rather than that of a moving painting.

We find that 10-15 frames per second works well: enough of the underlying motion is preserved without too much realism, and flickering is much less objectionable. These frame rates lend a stronger subjective impression of being hand-painted, especially at 10 Hz, and especially for animations with very little motion. It is not entirely obvious why; perhaps we can mentally interpolate the human action of placing each stroke between frames, or perhaps we are used to seeing hand-made animations at lower frame rates (animation shot on “twos” is typically 12 fps or 15 fps). However, reducing the frame rate is not always an option, especially for live-action footage where a change in frame rate significantly alters the character of the movie.

4.3 Optical Flow

In the above paint-over algorithm, brush strokes stay fixed on the image plane, even when the underlying object is moving. This gives the unusual, stylized effect of a continually-smudged image plane, and is sometimes undesirable. Alternatively, we can move the brush strokes with the objects, as suggested by Meier [Mei96]. Litwinowicz [Lit97, Lit99] proposed using optical flow to move brush strokes along the directions of scene motion. We use this idea by warping the image and painting-over. (Other features of Litwinowicz’ method are more difficult to adapt.) Warping strokes also reduces flickering, since the image being painted over should be a closer match to the new video frame.

Optical flow is a measurement of object movement in a video sequence; it is defined as the projection of world motion vectors onto the image plane. Given an object that projects to a pixel (x, y) in frame t of a video sequence, let (x', y') be the projection of that same pixel in frame $t + 1$. If the object is visible in each frame, then the optical flow for that pixel is given by $F(x, y, t) = (x' - x, y' - y)$. We use Simoncelli et al.'s [SAH91] probabilistic variant of coarse-to-fine differential estimation to compute optical flow from an image sequence.

The flow-based painting algorithm is as follows:

```
Paint the first frame
for each successive frame:
    Warp previous source frame to current for difference mask
    Warp all brush stroke control points to the current frame
    Paint the current frame over the warped painting
```

As with previous methods, we warp the painting by warping the stroke control points, not the bitmap. Warping the image can distort it in unpaintlike ways — given an ideal flow field, it would give the look of texture-mapping the painting onto the world.

Since we warp strokes by their control points and rerender, we must store all brush strokes in memory. Consequently, many brush strokes build up over time. We periodically cull strokes that are completely hidden, by rendering every stroke with a unique color and determining which colors are not shown (Figure 4.2).

Processing a sequence with optical flow produces a noticeably different effect than without. Instead of being continually repainted each frame, regions of the image move and shift to follow the surfaces in the scene. In our experiments, the look is of a wet, viscous canvas where brush strokes warp to match the video. The appearance is in large part due to the quality of the computed flow field — the flow algorithm we have used does not detect discontinuities in the flow, and therefore produces a smooth flow field.

This painting style is somewhat similar to experimental animation with wet paint on glass (e.g. the work of Caroline Leaf), pinscreen, or sand animation. However,

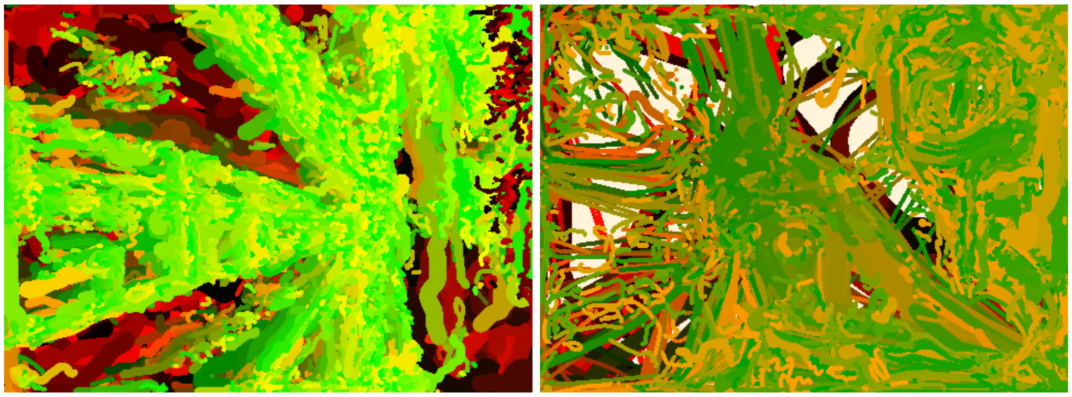


Figure 4.2: Typical pseudocolor images used for culling hidden brush strokes. Brush stroke ID is encoded in the stroke color. When a stroke's ID color does not appear in the pseudocolor image, it is completely hidden and will be deleted. The left image is from the "subway" sequence; the right image is from the end of the "tower" sequence, following a zoom out.

these techniques are limited by the media in their use of color, and thus are mostly monochromatic.

Using a flow field other than the computed optical flow produces interesting effects. For example, processing video or even a still image with a nonzero constant flow field gives a sense of motion in a still world (Figure 4.5). Painting with user-defined flow fields is an intriguing avenue for artistic exploration. Note that the paint-over algorithm without optical flow is a special case of the optical flow algorithm, with $F(x, y, t) \equiv (0, 0)$.

4.4 Experiments

4.4.1 Music Video

We applied our techniques to footage of a jazz recording session. We broke the input footage into large segments, and processed each segment separately. Painting styles

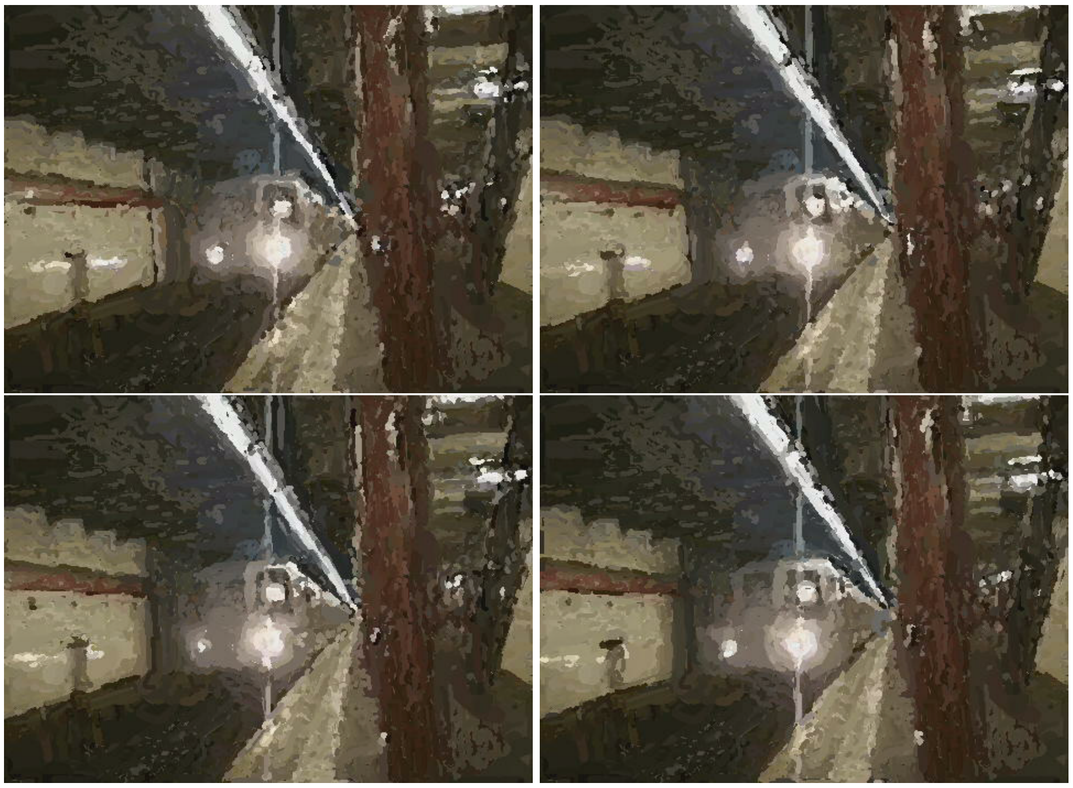


Figure 4.3: Consecutive frames of a video using paint-over and difference masking. The figures are in left-to-right, top-to-bottom order.

were chosen for visual appeal and to enhance the changing mood and intensity of the music. Although no specific formula was used, we found ourselves using some general design patterns: larger brush strokes (more abstract) were used during intense passages and meditative passages; smaller (less abstract) strokes were used in transitional passages; expressionistic styles (more active and abstract) were used mostly during the most intense passages and during solos. We processed almost all of the video at 15 frames per second, and did not use optical flow. The camera was moving at all times, so difference masking had relatively little effect. Dissolves in the source video produced the serendipitous effect of one moving shot being painted over another moving shot. When combined with a dramatic camera motion, a dissolve often appeared no longer as a dissolve, but as a smooth motion from one view to another. Video stills are shown in

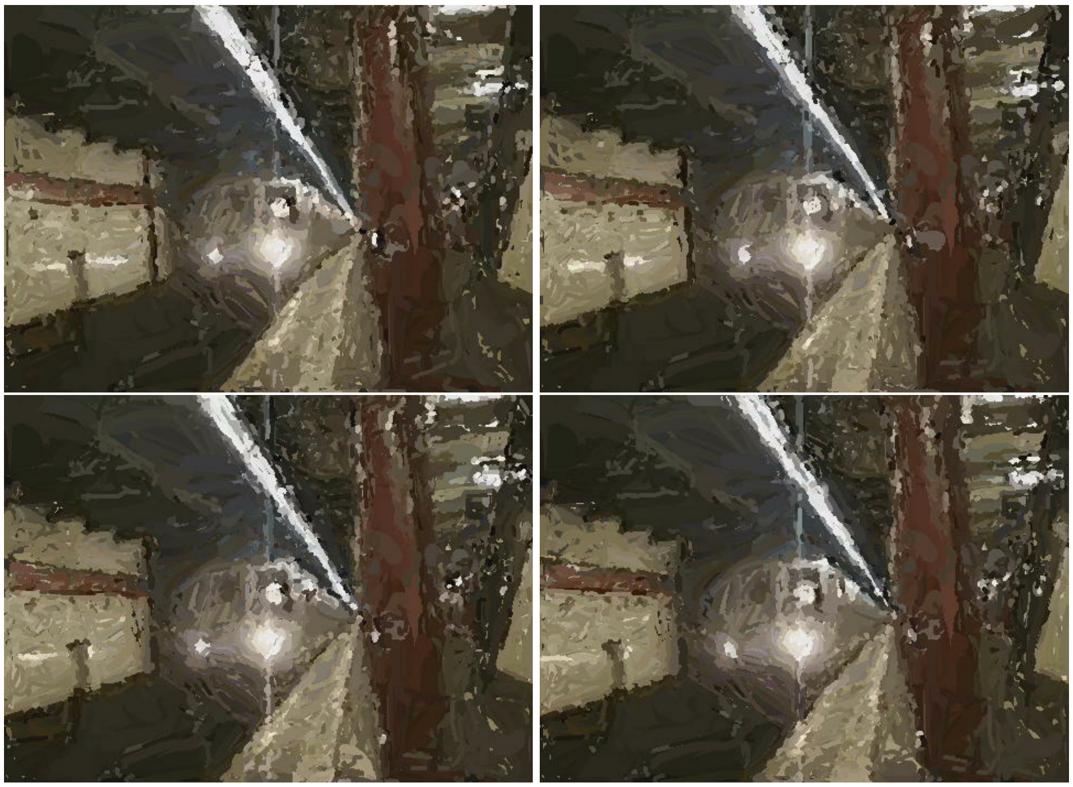


Figure 4.4: Consecutive frames from a painting using optical flow, paint-over and difference masking.

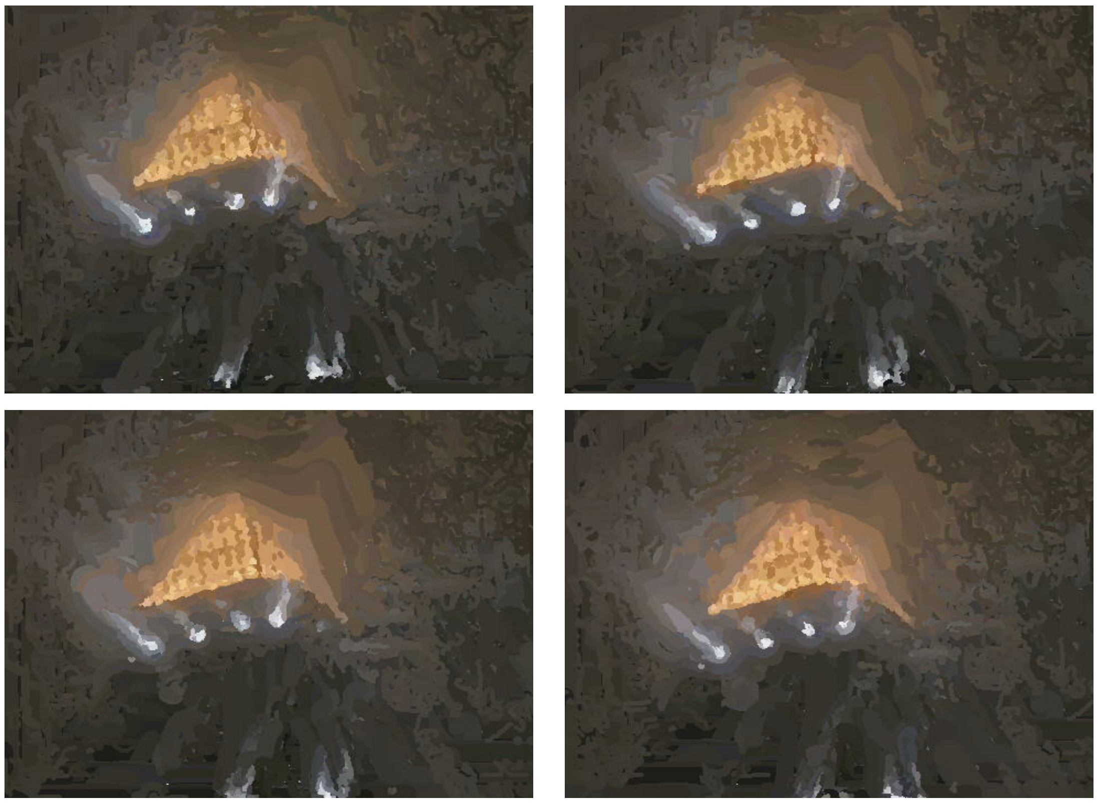


Figure 4.5: Consecutive frames of a video using a constant optical flow field ($F(x, y, t) \equiv (20 \text{ pixels}, 20 \text{ pixels})$). The video shows the top of a building on a foggy night, with camera motion. The flow field causes brush strokes in empty areas to rise toward the upper right corner of the image.

Figure 4.6.

4.4.2 A "Living" Painting

We wanted to demonstrate our methods in an experience that would be immediately accessible to an outside visitor. We fed the output from a video camera into a PC, painted it, and projected it onto a large stretched canvas (Figure 4.1). The result appears on the canvas as a continually-updated painting that visitors can interact with and be a part of. The system runs on a 350 MHz Pentium II, with a Matrox video card and an

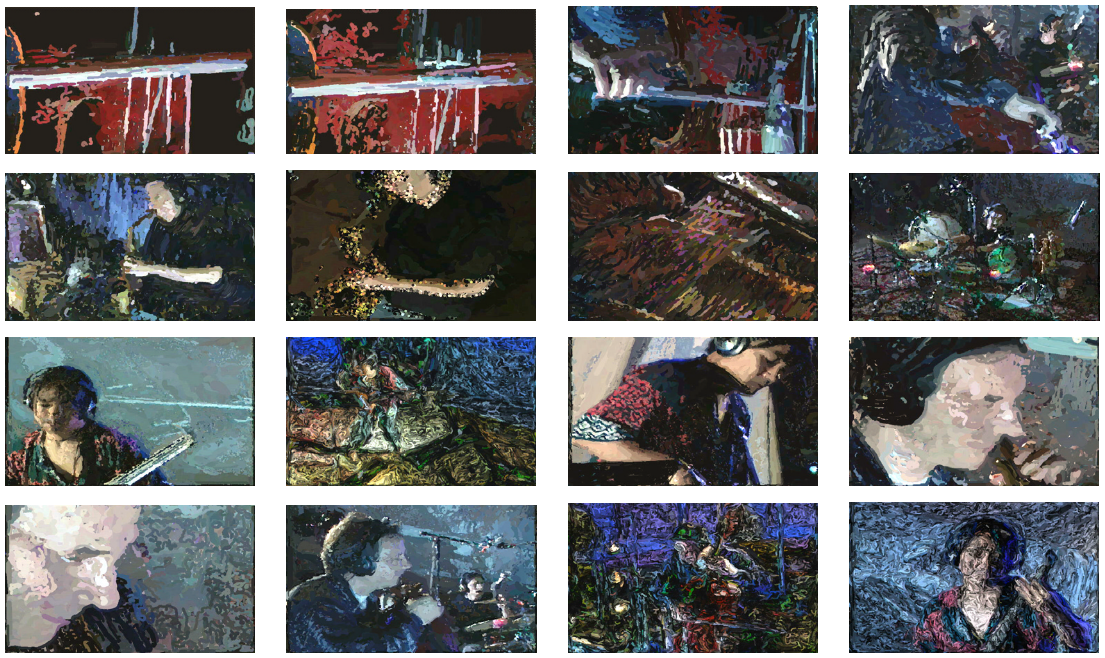


Figure 4.6: Frames from a music video, illustrating various painting styles and resulting effects.

Intergraph graphics board.

The system achieves a frame rate of 1-4 frames per second depending on the amount of motion in the scene; our system is not currently fast enough to include optical flow in the loop. However, at this frame rate, flickering cannot occur. We do not double buffer, allowing the visitor to watch the strokes appear on the canvas as they are created. Difference masking ensures that the painting only changes where there is motion in the scene. This is especially important for the more abstract painting styles in which the entire image would otherwise refresh each frame. Sample images are shown in Figure 4.8.

Our system has been demonstrated for many visitors. We found that users tended to spend a long time in front of the canvas creating various painterly renderings of their own faces and bodies. People also enjoyed watching other people create paintings. Participants seem to immediately understand and accept the process.

We have observed many serendipitous effects in the interactive interface, when mixing painting styles, feeding unusual or abstract video input (floors, closeups, manipulated color levels), and so on. An example is shown in Figure 4.7: certainly, not the painting effect we were going for, but an interesting combination of different effects (out of focus camera, moving viewer, changing style, etc.). The system seems to be able to produce a wide variety of fascinating variations.

The system also suggests a sense of ephemerality, of moments in time captured, painted, and then forgetten. It is an interesting effect to see brief freeze-frame paintings of passersby caught in a specific moment.

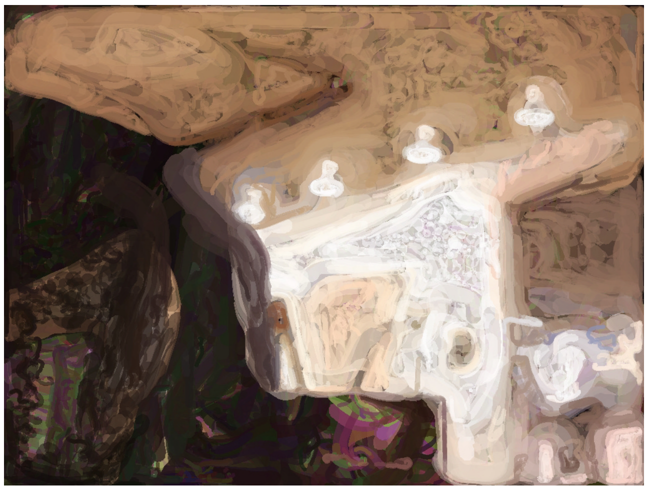


Figure 4.7: Serendipity.

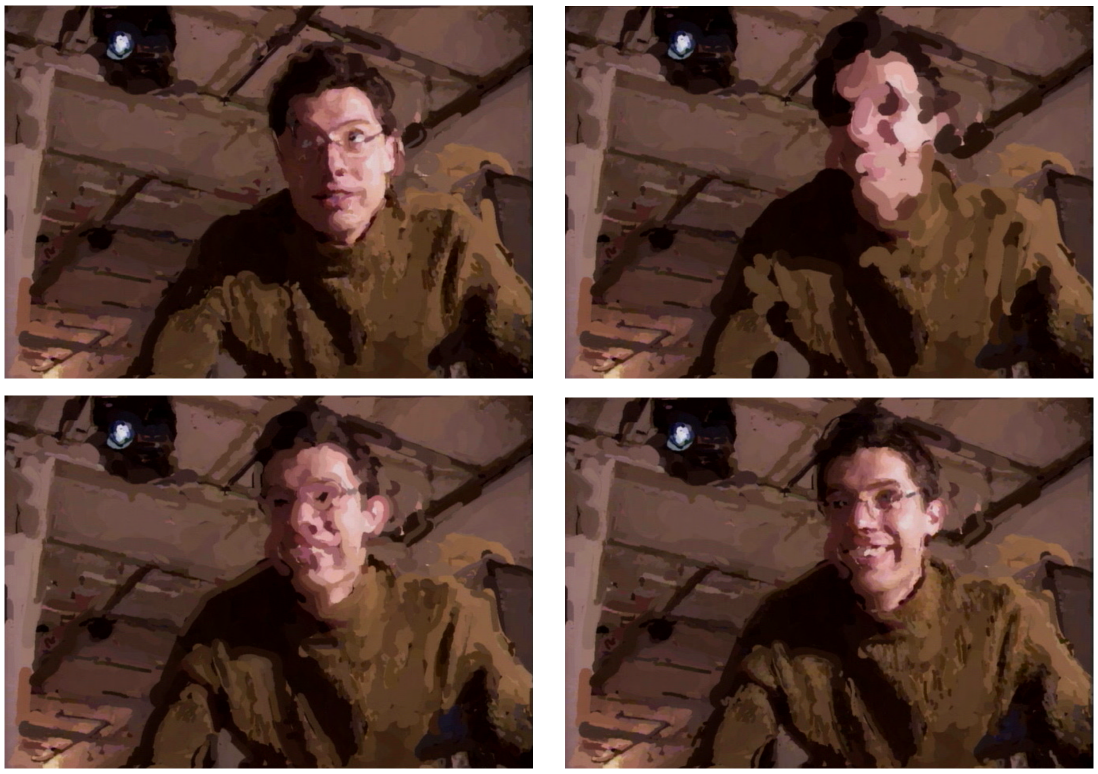


Figure 4.8: A sequence of paint layers from the live painting shown in Figure 4.1. The upper-right image shows strokes painted over the upper-left image; the upper-left and lower-right images show two consecutive “completed paintings.”

Chapter 5

Illustrating Smooth Surfaces

We now present a new set of algorithms for line-art rendering of smooth surfaces.[1] We present an automatic hatching method for conveying surface shape. We demonstrate these algorithms with a drawing style inspired by *A Topological Picturebook* by George Francis [Fra87] and the illustrations of Thomas Nast.

Line art is one of the most common illustration styles. Line drawing styles can be found in many contexts, such as cartoons, technical illustrations, architectural designs and medical atlases. These drawings often communicate information more efficiently and precisely than do photographs. Line art is easy to reproduce, compresses well and, if represented in vector form, is resolution-independent.

Many different styles of line art exist; the unifying feature of these styles is that the images are constructed from uniformly colored lines. The simplest is the style of silhouette drawing, which consists only of silhouettes, images of sharp creases, and object boundaries. This style is often sufficient in engineering or architectural contexts, where most shapes are constructed out of simple geometric components, such as boxes, spheres and cylinders. This style of rendering captures only geometry and completely ignores texture, lighting and shadows. On the other end of the spectrum is the pen-and-ink illustration style. In pen-and-ink illustrations, variable-density hatching and complex hatch

[1]This chapter describes joint work with Denis Zorin, first presented in [HZ00].

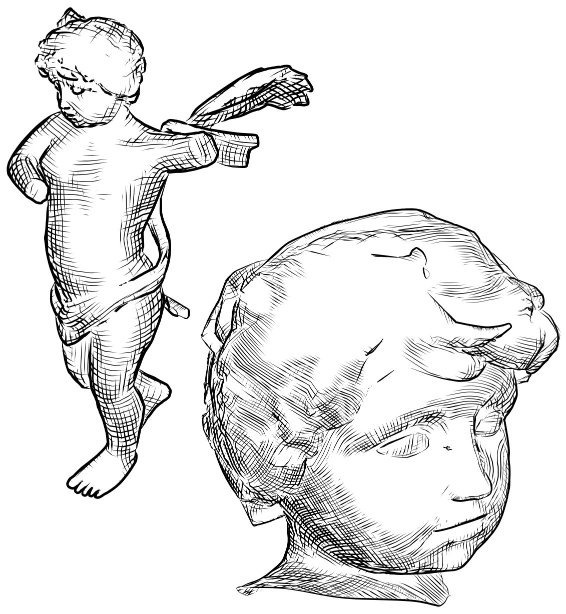


Figure 5.1: Illustrations of the Cupid mesh.

patterns convey information about shape, texture and lighting. While silhouette drawing is sufficient to convey information about simple objects, it is often insufficient for depicting objects that are complex or free-form. From many points of view, a smooth object may have no visible silhouette lines, aside from the outer silhouette (Figure 5.9), and all the information inside the silhouette is lost. In these cases, hatching can be added to indicate the shape of the surface.

This chapter describes a novel pen-and-ink illustration method. We first define our target style, which requires specification of desired hatch densities and of hatch directions. We then describe a hatch placement algorithm designed to create hatches matching these specifications.

5.1 Rendering Style

5.1.1 Hatching Density

Our rendering style is based to some extent on the rules described by G. Francis in *A Topological Picturebook* [Fra87], which are in turn based on Nikolaïdes' rules for drawing drapes [Nik75]. We have also used our own observations of various illustrations in similar styles. We begin our style description by defining undercuts and folds. A visible projected silhouette curve separates two areas of the image: one containing the image of the part of the surface on which the curve is located, the other empty or containing the image of a different part of the surface. We call the former area a *fold*. If the latter area contains the image of a part of the surface, we call it an *undercut*.

We use the following rules, illustrated in Figure 5.2.

- The surface is separated into four levels of hatching: highlights and Mach bands (no hatching), midtones (single hatching), shadowed regions (cross-hatching), and undercuts (dense cross-hatching). Inside each area, the hatch density stays approximately uniform. The choice of the number of hatch directions used at a particular area of the surface is guided by the lighting and the following rules:

- If there is an undercut on the other side of the silhouette from a fold, a thin area along the silhouette on the fold side is not hatched ("Mach band effect").

- Undercuts are densely hatched.

- Hatches are approximately straight; a hatch is terminated if its length exceeds a maximum, or if its direction deviates from the original by more than a fixed angle.

- Hatch thickness within each density level is inversely proportional to lighting.

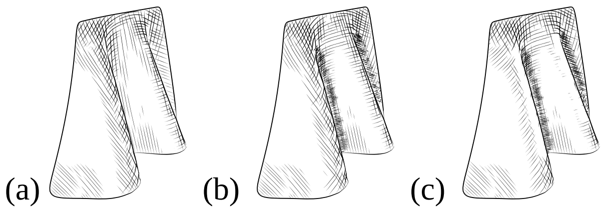


Figure 5.2: Hatching rules shown on drapes. (a) There are 3 main discrete hatch densities: highlights, midtones, and shadows, corresponding to 0, 1, and 2 directions of hatches. (b) Undercuts. (c) “Mach bands.” Undercuts and Mach bands increase contrast where surfaces overlap.

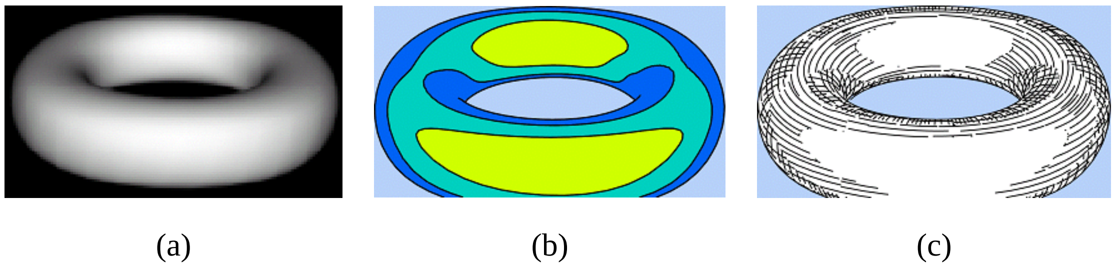


Figure 5.3: Hatching on the torus. (a) Model lit with Lambertian shading. The light source is at the headlight. (b) Hatch regions correspond to quantized lighting; the region boundaries are computed as isocontours of the lighting. Yellow/light-gray indicates highlight regions, cyan/mid-gray indicates single hatch regions, and blue/dark-gray indicates cross-hatch regions. (c) Hatching.

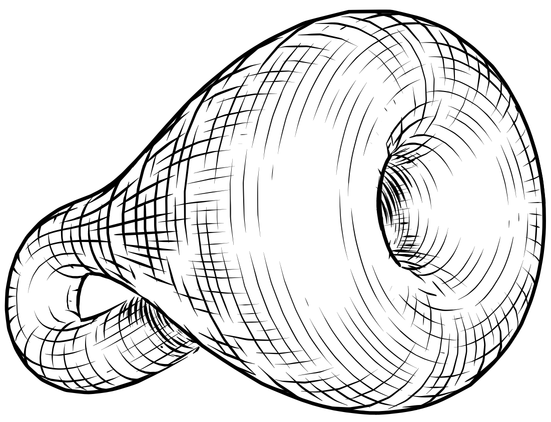


Figure 5.4: Klein bottle. Lighting and hatch directions are chosen to convey surface shape. Undercuts and Mach bands near the hole and the self-intersection enhance contrast.

5.1.2 Hatching Direction Fields

Fields on surfaces. To generate hatches, we need to choose hatch direction fields on visible parts of the surface. The direction fields we use are different from more commonly-used vector fields: unlike a vector field, the fields do not have a magnitude and do not distinguish between four orthogonal orientations.

The fields can be either defined directly in the image plane as in [SWHS97], or defined first on the surface, and then projected. The advantage of the former method is that the field needs to be defined and continuous only in each separate area of the image. However, it is somewhat more difficult to capture the information about the shape of the objects in the streamlines of the field, the field must be recomputed for each image, and regions that are nearby on the surface but not connected in the image might not be hatched coherently. We choose to generate the field on the surface first.

A number of different fields on surfaces have been used to define hatching directions. The most commonly-used field is probably the field of isoparametric lines; this method has obvious limitations, as the parameterization may be very far from isometric, and is not appropriate for surfaces lacking a natural parameterization, such as subdivision surfaces and implicit surfaces.

The most natural geometric candidate is the pair of principal curvature direction fields [Elb99, Int97] corresponding to the minimal and maximal curvature directions[2]. We will refer to the integral lines of these fields as *curvature lines.* These fields are independent of parameterization, capture important geometric features and are also consistent with the most common two-directional hatching pattern. However, they suffer from a number of disadvantages. All umbilical points (points with coinciding principal curvatures) are singularities, which means that the fields are not defined anywhere on a sphere and have arbitrarily complex structure on surfaces obtained by small perturbations of a sphere. On flat areas (when both curvatures are very small) the fields are likely to result in a far more complex pattern than the one that would be used by a human.

Other candidates include isophotes (lines of constant brightness) and the gradient field of the distance to silhouette or feature lines [MKT$^+$97, Elb98]. Both are suitable for hatching in a narrow band near silhouettes or feature lines, but typically do not adequately capture shape further from silhouettes, nor are they suitable for cross-hatching.

Our approach is based on several observations about successes and failures of existing methods, as well as hatching techniques used by artists.

- *Cylindrical surfaces.* Surface geometry is rendered best by principal curvature directions on cylindrical surfaces, that is, surfaces for which one of the principal curvatures is zero (all points of the surface are parabolic). This fact is quite remarkable: psychophysical studies confirm that even a few parallel curves can create a strong impression of a cylindrical surface with curves interpreted as principal curvature lines [Ste86, ML98]. Another important observation is that for cylinders the principal curvature lines are also geodesics, which is not necessarily true in general. Hatching following the principal curvature directions fails when the ratio of principal curvatures is close to one.

[2]It is possible to show that for a surface in general position, these fields are always globally defined, excluding a set of isolated singularities.

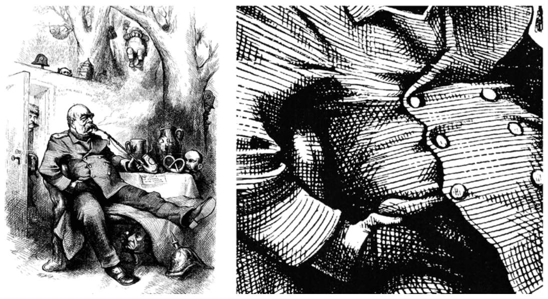


Figure 5.5: Almost all hatches in this cartoon by Thomas Nast curve only slightly, while capturing the overall shape of the surface. Note that the hatches often appear to follow a cylinder approximating the surface. Small details of the geometry are rendered using variations in hatch density.

- *Isometric parameterizations.* Isoparameteric lines work well as curvature directions when a parameterization exists and is close to isometric, i.e. minimizes the metric distortion as described in, for example, [EDD$^+$95, Ped96]. In this case, parametric lines are close to geodesics. Isoparametric lines were used by [WS96, Elb95].

- *Artistic examples.* We observe that artists tend to use relatively straight hatch lines, even when the surface has wrinkles. Smaller details are conveyed by varying the density and the number of hatch directions (Figure 5.5).

These observations lead to the following simple requirements for hatching fields: *in areas where the surface is close to parabolic, the field should be close to the principal curvature directions; on the whole surface, the integral curves of the field should be close to geodesic*. In addition, if the surface has small details, the field should be generated using a smoothed version of the surface.

Cross fields. While it is usually possible to generate two global direction fields for the two main hatch directions, we have observed that this is often undesirable. There are two reasons for this: first, if we would like to illustrate nonorientable surfaces, such fields

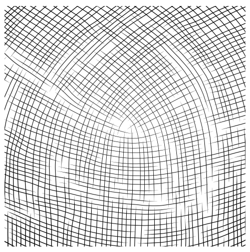


Figure 5.6: A cross-hatching pattern produced by our system on a smooth corner. This pattern cannot be decomposed into two orthogonal smooth fields near the corner singularity. The analytic expression for a similar field in the plane is $v_1(r, \theta) = [\cos(\theta/4), \sin(\theta/4)]$; $v_2(r, \theta) = [-\sin(\theta/4), \cos(\theta/4)]$. This field is continuous and smooth only if we do not distinguish between v_1 and v_2.

may not exist. Second, and more importantly, there are natural cross-hatching patterns that cannot be decomposed into two smooth fields even locally (Figure 5.6). Thus, we consider *cross fields*, that is, maps defined on the surface that assign an unordered pair of perpendicular directions to each point.

Constructing Hatching Fields. Our algorithm is based on the considerations above and proceeds in steps.

Step 1. Optionally, create a smoothed copy of the original mesh. The copy is used to compute the field. The amount of smoothing is chosen by the user, with regard to the smoothness of the original mesh, and the scale of geometric detail the user wishes to capture in the image. For example, no smoothing might be necessary for a close-up view of a small part of a surface, while substantial smoothing may be necessary to produce good images from a general view. Smoothing was not necessary for any of the meshes shown here.

Step 2. Identify areas of the surface which are sufficiently close to parabolic, that is, for which the ratio of minimal to maximal curvature is high, and for which at least one curvature is large enough to be computed reliably. Additionally, we mark as unreliable any vertex for which the average cross field energy (defined below) of its incident edges exceeds a threshold, in order to allow optimization of vertices that begin singular.

Step 3. Initialize the field over the whole surface by computing principal curvature directions in quasi-parabolic areas and propagating them to the rest of the mesh. If there are no quasi-parabolic areas, user input is required to initialize the field.

Step 4. Optimize the field. This step is of primary importance and we describe it in greater detail.

Our optimization procedure is based on the observation that we would like the integral lines of our field to be close to geodesics. We use a similar, but not identical, requirement that the field is as close to "constant" as possible. Minimizing the angles between the world-space directions at adjacent vertices of the mesh is possible, but requires constrained optimization to keep the directions in the tangent planes. We use a different idea, based on establishing a correspondence between the tangent planes at different points of the surface, which, in some sense, corresponds to the minimal possible motion of the tangent plane as we move from one point to another. Then we only need to minimize the change of the field with respect to corresponding directions in the tangent planes.

Given two sufficiently close points $\mathbf{p}_1$ and $\mathbf{p}_2$ on a smooth surface, a natural way to map the tangent plane at $\mathbf{p}_1$ to the tangent plane at $\mathbf{p}_2$ is to transport vectors along the geodesics (Figure 5.7(b)); for sufficiently close points there is a unique geodesic $\gamma(t)$, $t = 0 \ldots 1$, connecting these points. This is done by mapping a unit vector $\mathbf{u}_1$ in the tangent plane at $\mathbf{p}_1$ to a unit vector $\mathbf{u}_2$ in the tangent plane at $\mathbf{p}_2$, such that the angle between $\mathbf{u}_1$ and the tangent to the geodesic $\gamma'(0)$ is the same as the angle between $\mathbf{u}_2$ and $\gamma'(1)$.

In the discrete case, for adjacent vertices of the approximating mesh $\mathbf{v}_i$ and $\mathbf{v}_j$, we approximate the tangents to the geodesic by the projections of the edge $(\mathbf{v}_i, \mathbf{v}_j)$ into the tangent planes at the vertices. Let the directions of these projections be $\mathbf{t}_{ij}$ and $\mathbf{t}_{ji}$ (Figure 5.7(b)). Then a rigid rotation T_{ij} between the tangent planes is uniquely defined

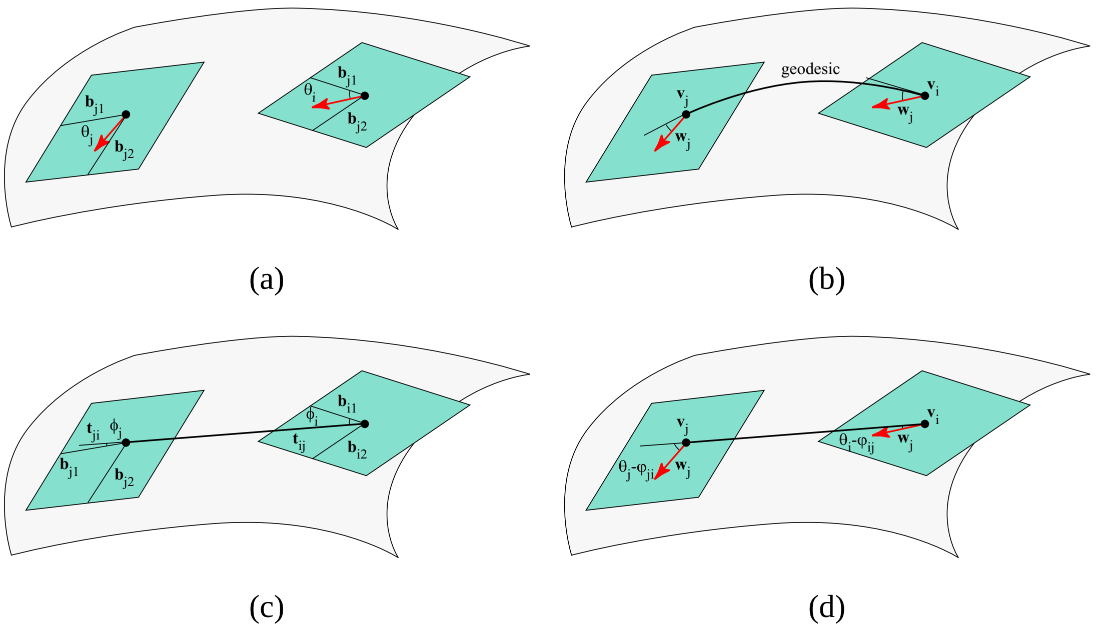


Figure 5.7: Moving vectors along geodesics. (a) A hatch direction is defined for each vertex. This direction may be represented as an angle in terms of the tangent plane basis vectors. (b) Directions may be compared with respect to a geodesic curve on the surface. (c) We compare mesh directions with respect to a mesh edge, as an approximation to a geodesic. This edge is projected to the tangent plane at $\mathbf{v}_i$ to get the direction $\mathbf{t}_{ij}$, and may be represented as an angle ϕ_{ij}. (d) The hatch direction with respect to the mesh edge projection is given by $\theta_i - \phi_{ij}$.

if we require that $\mathbf{t}_{ij}$ maps to $\mathbf{t}_{ji}$ and the transformation preserves orientation.

The directions can also be represented in the form of angles. Let vectors $\mathbf{b}_i^1$ and $\mathbf{b}_i^2$ be an orthonormal basis of the tangent plane at a vertex $\mathbf{v}_i$. A direction in the tangent plane can be uniquely defined as a linear combination of these vectors. In particular, we can represent the direction of $\mathbf{t}_{ij}$ as an angle (Figure 5.7(c): $\phi_{ij} = \tan^{-1}(\mathbf{b}_i^2 \cdot \mathbf{t}_{ij})/(\mathbf{b}_i^1 \cdot \mathbf{t}_{ij})$; conversely $\mathbf{t}_{ij} = \mathbf{b}_i^1 \cos\phi_{ij} + \mathbf{b}_i^2 \sin\phi_{ij}$. The direction ϕ_{ji} is defined similarly from $\mathbf{t}_{ji}$ with respect to $\mathbf{v}_j$. Consequently, the transformation T_{ij} described above consists of a global rotation to align the basis vectors $\mathbf{b}_i^1, \mathbf{b}_i^2$ with $\mathbf{b}_j^1, \mathbf{b}_j^2$, followed by a rotation in the tangent plane by an angle of $\phi_{ji} - \phi_{ij}$.

For any pair of tangent unit vectors $\mathbf{w}_i$ and $\mathbf{w}_j$ at $\mathbf{v}_i$ and $\mathbf{v}_j$ respectively, we can use $\|T_{ij}\mathbf{w}_i - \mathbf{w}_j\|^2$ to measure the difference between directions. Alternatively, we could transform $\mathbf{w}_j$ to $\mathbf{v}_j$ by the inverse transform $T_{ji} = T_{ij}^{-1}$; because T_{ij} is a rotation and thus orthonormal, this gives the same difference $\|\mathbf{w}_i - T_{ji}\mathbf{w}_j\|^2 = \|T_{ij}\mathbf{w}_i - \mathbf{w}_j\|^2$.

To measure the difference between the values of the cross field at two points, we choose a unit tangent vector for each point, each along the directions of the cross field. There are four possible choices at each point; we choose a pair of unit vectors for which the difference is minimal. In particular, the difference between adjacent field directions is:

$$E(i,j) = \min_k \|R(k\pi/2)T_{ij}\mathbf{w}_i - \mathbf{w}_j\|^2$$

where k is an integer and $R(k\pi/2)$ is a matrix corresponding to a $k\pi/2$ rotation in the tangent plane at $\mathbf{v}_i$. This formulation is somewhat cumbersome to optimize over; hence, we formulate it in terms of measuring angle differences as follows:

The cross field is described by a single angle θ_i for each vertex $\mathbf{v}_i$ (Figure 5.7(a)), which is the angle between the tangent direction $\mathbf{b}_i^1$ as above, and one of the directions of the cross field $\mathbf{w}_i$; we do not impose any limitations on the value of θ_i, and there are infinitely many choices for θ_i differing by $k\pi/2$ that result in the same cross field. One can visualize the field as four orthogonal vectors lying in the tangent plane. The directions for a vertex may be written as $\mathbf{w}_i = \mathbf{b}_1 \cos(\theta_i + k\pi/2) + \mathbf{b}_2 \sin(\theta_i + k\pi/2)$.

Using this choice of coordinates, one can use the Law of Cosines to show that

$$\begin{aligned} E(i,j) &= \min_k \|R(k\pi/2)T_{ij}\mathbf{w}_i - \mathbf{w}_j\|^2 && (5.1) \\ &= \min_k (2 - 2\cos((\theta_i - \phi_{ij}) - (\theta_j - \phi_{ji}) + k\pi/2)) && (5.2) \end{aligned}$$

since $(\theta_i - \phi_{ij}) - (\theta_j - \phi_{ji}) + k\pi/2$ is the angle between the unit vectors $R(k\pi/2)T_{ij}\mathbf{w}_i$ and $\mathbf{w}_j$. This form is not not differentiable. We observe, however, that

$$\begin{aligned} E_0(i,j) &= -8E(i,j)^4 + 8E(i,j)^2 - 1 && (5.3) \\ &= -\cos 4((\theta_i - \phi_{ij}) - (\theta_j - \phi_{ji})) && (5.4) \end{aligned}$$

and $E_0(i,j)$ is a monotonic function of $E(i,j)$ on $[\sqrt{2}/2 \ldots 1]$, the range of possible values of $E(i,j)$. Thus, instead of minimizing $E(i,j)$, we can minimize $E_0(i,j)$. We sum $E_0(i,j)$ over all edges of the surface in order to arrive at the following simple energy:

$$E_{\text{field}} = -\sum_{\text{all edges } (\mathbf{v}_i, \mathbf{v}_j)} \cos 4((\theta_i - \phi_{ij}) - (\theta_j - \phi_{ji}))$$

which does not require any constraints on the variables θ_i. Note that the values ϕ_{ij} are constant. Due to the simple form of the functional, it can be minimized quite quickly. We use a variation of the BFGS conjugate gradient algorithm described in [ZBLN97] to perform minimization.

This optimization can be modified for different kinds of fields. To optimize direction field smoothness, simply replace the "4" above with a "1"; for an orientation field (invariant to 180-degree rotations), replace the "4" with a "2".

5.2 Hatch Placement Algorithm

We now describe a hatching algorithm that attempts to satisfy the style rules defined above. The style/hatching procedure has several user-tunable parameters: basic hatch density specified in image space; the hatch density for undercuts; the threshold for highlights (the areas which receive no hatching); the threshold that separates single hatch

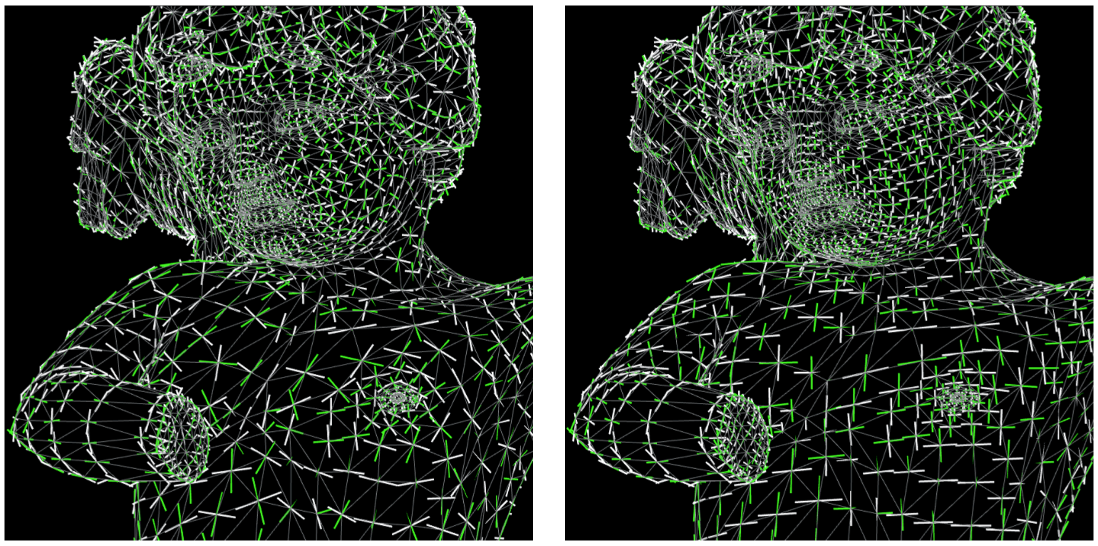

(a) (b)

Figure 5.8: Cross field on the cupid surface. The surface is represented as a coarse approximating mesh; the smooth surface is the limit of subdivision steps. Four orthonormal directions are defined at each surface point. (a) Computed principal directions. Near umbilic points, the results are ill-posed. (b) Result after optimizing all directions. This field is much more suitable for use with hatching.

regions from cross hatch regions; the maximum hatch length; the maximum deviation of hatches from the initial direction in world space. Varying these parameters has a considerable effect both on the appearance of the images and on the time required by the algorithm. Density threshold values are usually chosen to divide the object approximately evenly between different hatching levels.

Once we have a hatching field, we can illustrate the surface by placing hatches along the field. We first define three intensity regions over the surface: no hatching (highlights and Mach bands), single hatching (midtones), and cross hatching (shadowed regions). Furthermore, some highlight and hatch regions may be marked as undercut regions. The hatching algorithm operates as follows:

1. Identify Mach bands and undercuts.

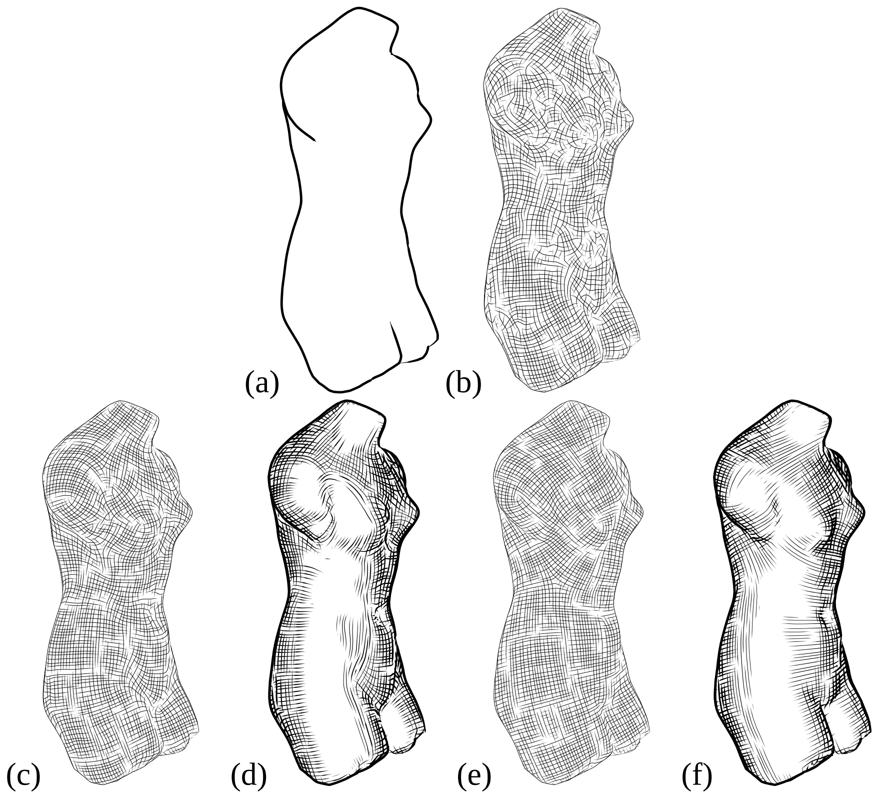


Figure 5.9: Direction fields on the Venus. (a) Silhouettes alone do not convey the interior shape of the surface. (b) Raw principal curvature directions produce an overly-complex hatching pattern. (c) Smooth cross field produced by optimization. Reliable principal curvature directions are left unchanged. Optimization is initialized by the principal curvatures. (d) Hatching with the smooth cross field. (e) Very smooth cross field produced by optimizing all directions. (f) Hatching from the very smooth field.

2. Cover the single and cross hatch regions with cross hatches, and add extra hatches to undercut regions.

3. Remove cross-hatches in the single hatch regions, leaving only one direction of hatches.

5.2.1 Identifying Mach Bands and Undercuts

In order to identify Mach bands and undercuts, we step along each silhouette and boundary curve. A ray test near each curve point is used to determine whether the fold overlaps another surface. Undercuts and Mach bands are indicated in a 2D grid, by marking every grid cell within a small distance of the fold on the near side of the surface (as used for hatching in the next section).

5.2.2 Cross-hatching

We begin by creating evenly-spaced cross-hatches on a surface. We adapt Jobard and Lefers' method for creating evenly-spaced streamlines of a 2D vector field [JL97]. The hatching algorithm allows us to place evenly-spaced hatches in a single pass over the surface.

Our approach is to greedily trace curves along the surface. We "seed" hatch points near existing curves, and trace curves along the cross field until they come to close to another curve, curves too far from its original direction, or reaches a critical curve (e.g. a silhouette). Our algorithm takes two parameters: a desired hatch separation distance d_{sep}, and a test factor d_{test}. The separation distance indicates the desired image-space hatch density; a smaller separation distance is used for undercuts.

We now describe the algorithm in more detail. The algorithm creates a queue of surface curves, initially containing the critical curves (silhouettes, boundaries, creases, and self-intersections). While the queue is not empty, we remove the front curve from the queue and seed new hatches along it at points evenly-spaced in the image. Seeding creates a new hatch on the surface by tracing the directions of the cross-hatching field. Since the cross field is invariant to 90 degree rotations, at each step the hatch follows the one of four possible directions which has the smallest angle with the previous direction. Within a mesh face, the hatch direction is linearly interpolated from the directions at the faces. Hatches are seeded perpendicular to all curves. Hatches are also seeded parallel to other hatches, at a distance d_{sep} from the curve. A hatch continues along the surface until it terminates in a critical curve, until the world-space hatch direction deviates from the initial hatch direction by more than a constant, or until it comes near a parallel hatch. This latter condition occurs when the endpoint of the hatch $\mathbf{p}_1$ is near a point $\mathbf{p}_2$ on another hatch, such that the following conditions are met:

- $\|\mathbf{p}_1 - \mathbf{p}_2\| < d_{test} d_{sep}$, measured in image space.
- A straight line drawn between the two points in image space does not intersect the projection of any visible critical curves. In other words, hatches do not "interfere" when they are not near on the surface.
- The world space tangents of the two hatch curves are parallel, i.e. the angle between them is less than 45 degrees, after projection to the tangent plane at $\mathbf{p}_1$.

The search for nearby hatches is performed by placing all hatches in a 2D grid with grid spacing equal to d_{sep}. This ensures that at most nine grid cells must be searched to detect whether there are hatches nearby the one being traced.

5.2.3 Hatch Reduction

Once we have cross-hatched all hatch regions, we remove hatches from the single hatch regions until they contain no cross-hatches. By removing hatches instead of directly placing single a hatch direction, we avoid the difficulty inherent in producing a consistent vector field on the surface. Our algorithm implicitly segments the visible single-hatch regions into locally-consistent single hatching fields. This way, we can take advantage of the known view direction and the limited extent of these regions, making the task substantially more tractable.

The reduction algorithm examines every hatch on the surface and deletes any hatch that is perpendicular to another hatch. In particular, a hatch is deleted if it contains a point $\mathbf{p}_1$ nearby a point $\mathbf{p}_2$ on another hatch such that:

- $\mathbf{p}_1$ and $\mathbf{p}_2$ lie within the single hatch region.
- $\|\mathbf{p}_1 - \mathbf{p}_2\| < 2d_{sep}$, measured in image space.
- A straight line drawn between the two points in image space does not intersect any visible critical curve.
- The world space tangents of the two hatch curves are nearly perpendicular, i.e. the angle between them is greater than 45 degrees after projection to the tangent plane at $\mathbf{p}_1$.

Deleting a hatch entails clipping it to the cross-hatch region; the part of the hatch that lies within the cross-hatch region is left untouched.

The order in which hatches are traversed is important; a naïve traversal order will usually leave the single hatch region uneven and inconsistent. For example, consider a planar surface with cross-hatching (two orthogonal sets of hatches). To get single hatching, we would like to delete all of one set of parallel hatches, or delete the other (perpendicular) set of parallel hatches. However, when we begin traversal, all hatches are perpendicular to others, and could be deleted. One can easily construct cases where all hatches but one are deleted under a poor traversal order.

We perform a breadth-first traversal to prevent this. A queue is initialized with a hatch curve. While the queue is not empty, the front curve is removed from the queue. If it is perpendicular to another curve in the single hatch region, then the curve is deleted, and all parallel neighbors of the hatch that have not been visited are added to the queue. When the queue is empty, a hatch that has not yet been visited is added to the queue, if any remain. The test of whether curves are perpendicular is as described above; the angle condition is reversed for the parallel test.

The hatch traversal can be summarized by the following pseudocode.

```
$Q$ ← an empty queue
for each hatch $H$
    if $H$ has not been visited already
        add $H$ to $Q$ and mark $H$ visited
        while $Q$ is not empty
            pop the front $J$ from $Q$
            if $J$ can be removed
                clip $J$
                for each parallel neighbor $L$ of $J$
                    if $L$ has not been visited
                        mark $L$ visited
                        add $L$ to $Q$
```

5.3 Results and Conclusions

Most of the illustrations in this chapter were created using our system. Figures 5.1 and 5.9 demonstrate the results for relatively fine meshes that define surfaces with complex geometry. Figures 5.4 and 5.11 show the results of using our system to illustrate several mathematical surfaces.

The time required to create an illustration varies greatly; while silhouette drawings can be computed interactively, and the field optimization takes very little time, hatching is still time-consuming, and can take from seconds to minutes, depending on hatch density and complexity of the model.

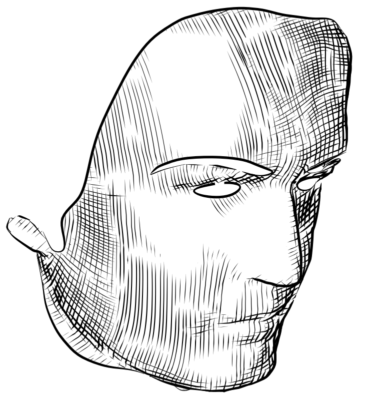


Figure 5.10: One piece from a Beethoven surface.

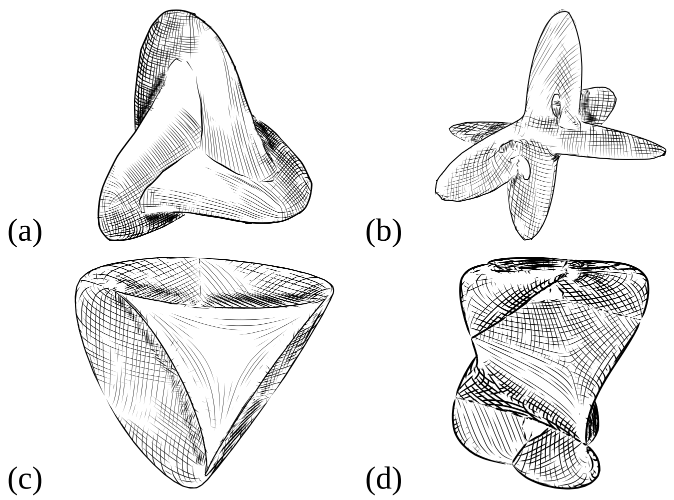


Figure 5.11: Several surfaces generated using G. Francis' generalization of Apéry's Romboy homotopy [Fra91]. (a) Boy surface; (b) "Ida"; (c) Roman surface; (d) Etruscan Venus.

Chapter 6

Paint By Relaxation

Creating a wider space of styles from the painterly rendering algorithm described in the previous two chapters can be difficult. In part, this is because it is not entirely clear what mathematical problem we wish to solve: there is no clear problem statement. Typically, to create a new style, one must write a explicit procedure for it, which can be quite difficult. In this chapter, we pose "painterly rendering" as an energy minimization problem.[1] This provides a general framework: to change the painting style, we change the energy function; to improve the algorithm, we try to find better local minima. The appeal of this strategy is that, ideally, we need only specify what we want, not how to directly compute it. Because the energy function is very difficult to optimize, we use a relaxation algorithm combined with various search heuristics.

This formulation allows us to specify painting style by varying the relative weights of energy terms. The basic energy function we have used yields an economical painting that effectively conveys an image with few strokes. This economical style produces good temporal coherence when processing video, while maintaining the essential 2D quality of the painting.

This gives us the ability to produce paintings that closely match the input imagery. This is particularly important in processing video sequences, where imprecision can

[1]A condensed version of this chapter will appear in [Her01].

lead to unacceptable problems with temporal coherence. The algorithm described in the previous chapter requires many small strokes, and still produces a degree of flickering that may be objectionable for some applications, and this is the algorithm for painting with long, curved strokes that we are aware of. We have attempted to recreate a somewhat "generic," modern painting style: realistic, but with visible brush strokes, inspired by artists such as Richard Diebenkorn, John Singer Sargent, and Lucian Freud.

The energy minimization approach also allows us to bridge the gap between automatic painterly filters and user-driven paint systems. Users may specify requirements for a painting at a high level (painting style), middle level (variations of style over the image), and low level (specific brush strokes). The user may work back and forth between these different levels of abstraction as desired. This feature is of critical importance for taking the most advantage of the skills of artists. This also allows us to accommodate a wide range of user requirements and skill levels, from desktop publishing to feature film production.

Note that the energy function has no *a priori* meaning: it simply represents some quantification of the user's aesthetics, and experimentation plays an important role in determining the function and weights.

6.1 Energy Function

The central idea of this chapter is to formulate painting as an energy relaxation problem. Given an energy function $E(P)$, we seek the painting P^* with the least energy, from among the space of paintings $\mathcal{P}$:

$$P^* = \arg\min_{P \in \mathcal{P}} E(P)$$

For this chapter, a painting is defined as an ordered collection of colored brush strokes, together with a fixed canvas color or texture. A brush stroke is a thick curve defined by a list of control points, a color, and a thickness/brush width. A painting is rendered by compositing the brush strokes in order onto the canvas. Brush strokes are rendered as thick cubic B-splines. Strokes are drawn in the order that they were created.[2]

An energy function can be created as a combination of different functions, allowing us to express a variety of desires about the painting, and their relative importance. Each energy function corresponds to a different painting style.

We use the following energy function for a painting:

$$\begin{aligned} E(P) &= E_{app}(P) + E_{area}(P) + E_{nstr}(P) + E_{cov}(P) \\ E_{app}(P) &= \sum_{(x,y)\in\mathcal{I}} w_{app}(x,y)\|P(x,y) - G(x,y)\| \\ E_{area}(P) &= w_{area} \sum_{S\in P} \text{Area}(S) \\ E_{nstr}(P) &= w_{nstr} \cdot (\text{the number of strokes in } P) \\ E_{cov}(P) &= w_{cov} \cdot (\text{the number of empty pixels in } P) \end{aligned}$$

This energy is a linear combination of four terms. The first term, E_{app}, measures the pixelwise differences between the painting and a source image G. The differences are summed over each pixel (the set $\mathcal{I}$). Each painting will be created with respect to a source image; this is the image that we are painting a picture of.[3] The second term, E_{area}, measures the sum of the surface areas of all of the brush strokes in the painting. In some sense, this is a measure of the total amount of paint that was used by the artist in making the image. The number of strokes term E_{nstr} can be used to bias the painting towards larger brush strokes. The coverage term E_{cov} is used to force the canvas to be filled with paint, when desired, by setting w_{cov} to be very large. All weights w are user-defined values. The color distance $\|\cdot\|$ represents Euclidean distance in RGB space.

[2]We also experimented with ordering by brush radius, and found that, due to the nature of the relaxation, large strokes were never placed after small strokes in any ordering. This occurs because a smaller stroke is almost always a better approximation to the source image than is a large stroke; hence, adding a large stroke on top of many small strokes almost always increases the painting energy in the short term.

[3]One could imagine synthesizing abstract paintings without any source images. In order to allow multiple paintings in the same style, the painting style could be modeled as a probability density, from which paintings are sampled.

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_{area}, a user can specify the relative importance of these two desires, and thus produce different painting styles. Likewise, adjusting w_{nstr} allows us to discourage the use of smaller strokes.

By default, the value of $w_{app}(x, y)$ is initialized by a binary edge image, computed with a Sobel filter. This gives extra emphasis to edge quality. A constant weight usually 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 6.7). Similar to the weight image in Section 3.3.3, the weight image $w_{app}(x, y)$ 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 (Section 6.6).

The particular form of these equations has no intrinsic meaning (that we are aware of); they are simply one possible formulation of the desires expressed above. When the user chooses a set of weights (usually by experimentation), the complete energy function encapsulates some aesthetic sensibility of the user.

Some important visual constraints are difficult to write mathematically. For example, the desire that the painting appear to be composed of brush strokes would be difficult to write as an energy function over the space of bitmaps; hence, we parameterize a painting in terms of strokes, defining some aspects of a painting style as parameters to the relaxation procedure, rather than as weights in the energy function.

6.2 Relaxation

The energy function presented in the previous section is very difficult to optimize: it is discontinuous, it has a very high dimensionality and there does not appear to be an analytic solution.

Following Turk and Banks [TB96], we use a relaxation algorithm. This is a trial-and-error approach, illustrated by the following pseudocode:

```
$P \leftarrow$ empty painting

**while not** *done*
    $C \leftarrow \textsc{Suggest}()$               // Suggest change
    **if** $(E(C(P)) < E(P))$         // Does the change help?
        $P \leftarrow C(P)$                // If so, adopt it
```

There is no guarantee that the algorithm will converge to a result that is globally optimal or even locally optimal. Nevertheless, this method is still useful for producing pleasing results.

The main question of interest is how to make the suggestions. In our initial experiments, we used highly random suggestions, such as adding a disc stroke in a random location with a random size. Most of the computation time was spent on suggestions that were rejected, resulting in paintings poorly-matched to the energy. Because the space of paintings has very high dimensionality, it is possible to make many suggestions that do not substantially reduce the energy. Hence, it is necessary to devise more sophisticated ways of generating suggestions.

6.3 Stroke Relaxation

A basic strategy that we use for generating suggestions is to use a *stroke relaxation* procedure. This is a variation on snakes [KWT87], adapted to the painting energy function. This method refines an approximate stroke placement choice to find a local maximum for a given stroke. This is done by an exhaustive local search to find the best control points for the stroke, while holding fixed the stroke's radius and color, and the appearance of the rest of the painting. We also modify the snake algorithm to include the number of stroke control points as a search variable.

The high-level algorithm for modifying a stroke is:

1. Measure the painting energy
2. Select an existing stroke
3. Optionally, modify some stroke attribute
4. Relax the stroke
5. Measure the new painting energy

Here we omit many implementation details; this algorithm is described in detail in Section 6.7.1. However, we mention two important aspects of the implementation here: First, the stroke search procedure evaluates the painting energy with the stroke deleted (for reasons described later). Hence, the modification suggestion may become a stroke deletion suggestion, at no extra cost. Second, the relaxation procedure would be very expensive and difficult to implement if done precisely. Instead, we use an approximation for the energy inside the relaxation loop, and measure the exact energy only after generating the suggestion.

6.3.1 Single Relaxation Steps

In this section, we describe individual procedures for generating suggestions.

Add Stroke: A new stroke is created, starting from a given point on the image. The new stroke is always added in front of all existing strokes. Control points are added to the stroke, using the PAINTSTROKE procedure from Chapter 3. The stroke is then relaxed. The result of this search is then suggested as a change to the painting.

Reactivate Stroke: A given stroke is relaxed, and the modification is suggested as a change to the painting. However, if deleting the stroke reduces the painting energy more than does the modification, then the stroke will be deleted instead. Note that this procedure does not modify the ordering of strokes in the image. Thus, it will delete any stroke that is entirely hidden, so long as there is some penalty for including strokes in the painting (i.e. $w_{area} > 0$ or $w_{nstr} > 0$).

Enlarge Stroke: If the stroke radius is below the minimum, it is incremented and the stroke is reactivated. The resulting stroke becomes a suggestion.

Shrink Stroke: If the stroke radius is above the maximum radius, the stroke radius is decremented and the stroke is reactivated.

Recolor: The color for a stroke is set to the average of the source image colors over all of the visible pixels of the stroke, and the stroke is reactivated.

6.3.2 Combining Steps

Individual suggestions are combined into loops over the brush strokes in order to guarantee that every stroke is visited by a relaxation step.

Place Layer: Loop over image, *Add Stroke*s of a specified radius, with randomly perturbed starting points.

Reactivate All: Iterate over the strokes in order of placement, and *Reactivate* them.

Enlarge/Shrink All: For each stroke, *Enlarge* until the change is rejected or the stroke is deleted. If the stroke is unchanged, then *Shrink* the stroke until the change is rejected or the stroke deleted.

Recolor All: Iterate over the strokes, and *Recolor* each one.

Script: Execute a sequence of relaxation loops. To make a painting from scratch, we use the following script:

```
foreach brush size $R_i$, from largest to smallest,
    do $N$ times:
        Reactivate all strokes
        Place Layer $R_i$
        Enlarge/Shrink All
        Recolor All
```

We normally use $N = 2$. This script usually brings the painting sufficiently close to convergence. Note that the reactivation loops apply to all brush strokes, not just those of the current size. For processing video, we reduce processing time by setting $N = 1$ and omitting the recolor and enlarge steps.

Creating a final image can take several hours, depending on the image size and the painting style. Most of the images in this chapter were generated as overnight jobs on a 225 MHz SGI Octane R10000. We have processed video at low resolution (320x240) with the reduced script above to save time, yielding a processing rate of about 1 hour per frame.

6.4 Stroke Color, Texture, and Opacity

Our system provides a variety of different ways to render an intermediate or final painting. Brush strokes can be rendered with random color and intensity perturbations (Section 3.3.1), with transparency, and/or with procedural textures. Texturing operations are omitted from the main relaxation procedure solely for the sake of efficiency. Furthermore, separating the rendering step allows us to experiment with different stroke colors and textures without needing to perform relaxation anew. Procedural stroke texturing is described in Appendix 8.3.

All of our stroke scan-conversion is performed in software, using the triangle-strip method described in Appendix 8.3. We have not used hardware scan-conversion because we need to be able to reliably traverse every pixel in a brush stroke without rendering, which is not easily afforded by hardware scan-conversion. Scan-converting in software allows us to guarantee that every loop over the pixels of a stroke (for adding, removing, or summing image statistics) visits every pixel exactly once. For example, when we delete a stroke, we need to visit every pixel that was covered by the stroke to measure its change; software scan-conversion allows us to guarantee that we do indeed visit every pixel that was covered.

6.5 Video

The relaxation framework extends naturally to processing video. A variety of energy formulations can be employed for video. In the simplest formulation, the energy of the video sequence is the sum of the energies for each frame. The target image for each video frame is the corresponding image in the source sequence. For efficiency and temporal coherence, we can use the painting for one frame as the initial painting for the next frame. For speed, we currently process each frame sequentially; a more general extension would be to perform relaxation over the entire sequence in arbitrary order.

This method can be improved if optical flow information is available [Lit97]. Optical flow is a measure of the motion of scene objects projected onto the image plane. Warping brush strokes by optical flow gives the impression of brush strokes moving to follow the motions of objects in the scene. We compute flow using a variant of coarse-to-fine differential motion estimation [SAH91]. In order to generate the initial painting for a frame, we warp the stroke positions by the optical flow before relaxation. This gives a good initialization for the next frame.

This yields a relatively clear video sequence, with much less flickering than in previous algorithms. In particular, it is possible to paint a video sequence with much larger strokes than before. However, the temporal coherence is far from perfect, and more work remains to be done in this area. For example, we experimented with an energy term that penalized the difference between painted frame at time t_i and the painted frame at t_{i-1} warped to t_i. This produced very poor results, because the brush strokes are placed in frame t_{i-1} without regard to the scene geometry. For example, a stroke that is placed over the silhouette of a white surface against a white background will not work well with a change of viewpoint.

6.6 Interaction and Metapainting

The energy formulation is well-suited for an interactive painting application. In this view, the software functions as a “high-level paintbox,” allowing the user to make artistic decisions at any stage and at any level of abstraction in the painting process. In particular, the user may:

- Select overall painting styles.
- Select different styles for different image regions.
- Place individual brush strokes, as suggestions or by decree.

We refer to this approach as *metapainting*.

In our implementation, the user has complete freedom to make these choices before, during, and after relaxation. For example, the user might select an overall style for an image and perform a relaxation. After seeing the result, the user decides that a particular part of the image should be emphasized, paints a new appearance weight function, and then performs relaxation again. The user then realizes that she desires specific brush strokes in one image region, and then paints them in.

Our user interface allows several different methods for visualizing the painting and editing styles, including output painting, difference image, source image, weight image, pseudocolor (each stroke assigned a random color), etc. The user may paint directly into the output image, and may paint weights into the weight image. Painting styles and parameters are controlled by various widgets. Specific strokes may be reactivated by clicking on them. During relaxation, changes to the painting are displayed as they are computed, to give the user a view of the work in progress and to allow modification to the style during the computation.

We believe that interactive metapainting can be a very powerful tool for image-making, combining the ease of an automatic filter with the fine control of a digital paint system. The artist holds absolute control over the appearance of the final painting, without being required to specify every detail. The appeal is even greater for video processing, where one can automatically propagate artistic decisions between video frames.

With faster hardware, the user feedback loop can be made tighter. We look forward to the day when it is possible to visualize changes in painting styles at interactive rates.

6.7 Implementation Details

In this section, we describe some of the algorithms and data structures used in our implementation to avoid redundant computation.

Lazy evaluation and partial result caching are used throughout our code. The values of each subterm in the energy function are cached, and are only updated when they change. For example, one image encodes the per-pixel value of the weighted color difference (WCD) between the painting and source image (i.e. the summand of E_{app}). Whenever a pixel value changes, the corresponding value in the WCD image is updated. The sum of the WCD image is also updated by subtracting the old value and adding the new value. Similar methods are used for the other energy terms.

6.7.1 Stroke Relaxation

We now describe the details of the stroke relaxation procedure introduced in Section 6.3.

Performing a search for a locally optimal set of stroke control points would be prohibitively expensive; the cost is a product of the number of per-pixel operations, the cost of scan-converting a curve section, and an exponential in the size of the search neighborhood. A key idea of the relaxation procedure is to perform relaxation over a simpler energy function that approximates the true energy, and has roughly the same minima as the true energy. The true stroke energy will only be evaluated once, for the resulting suggestion.

Our goal is to replace initial stroke S with a stroke T, where T has the same color and thickness as S, and T minimizes the new energy $E(P - S + T)$. This is equivalent to minimizing $E(P - S + T) - E(P - S)$, which can be evaluated more quickly, since we only need to determine where the painting is changing.[4] We build an approximate energy function $I(T) \approx E(P - S + T) - E(P - S)$ over which to search. In this new energy function, a brush stroke is modeled roughly as the union of discs centered at each control point. The search is restricted in order to ensure that adjacent control points maintain a roughly constant distance from each other. This is done by adding an extra term to the painting energy function: $E_{mem}(P) = w_{mem} \sum(\|v_i - v_{i+1}\| - R)^2$, for a stroke of radius R. This is a variant of membrane energy [KWT87].

All of the terms of the true energy function $E(P)$ have corresponding terms in the approximation $I(T)$. The number of strokes and spacing terms are measured exactly: $I_{str}(T) = w_{nstr}$, $I_{mem}(T) = E_{mem}(T)$. The area energy I_{area} is approximated as the product of the stroke's radius and its control polygon arc length.

The appearance (I_{app}) and coverage improvements (I_{empty}) are approximated by making use of auxiliary functions $I_R(x, y)$ and $I_p(x, y)$ (Figure 6.1). This formulation allows us to use lazy evaluation to avoid unnecessary extra computation of the auxiliary functions. These functions are given by:

$$\begin{aligned}
I_R(x, y) &= \sum_{\|(u,v)-(x,y)\| \le R} I_p(u, v) \\
I_p(x, y) &= h(x, y) w_{app}(x, y) I_C(x, y) - c(x, y) w_{empty} \\
I_C(x, y) &= \|C - G(x, y)\| - \|(P - S)(x, y) - G(x, y)\|
\end{aligned}$$

C is the color of the brush stroke. $c(x, y)$ and $h(x, y)$ are indicator functions computed from the fragment buffer: $c(x, y)$ is 0 if (x, y) is covered by any paint, 1 otherwise; $h(x, y)$ is 0 if (x, y) is covered by a stroke that was created after S. (The new stroke

[4]We use $+$ and $-$ in the set theoretic sense when applied to paintings and strokes, e.g. $P + T = P \cup \{T\}$ = painting P with stroke T added. $E(P)$ is a scalar, and thus $E(P_1) - E(P_2)$ is a scalar subtraction.

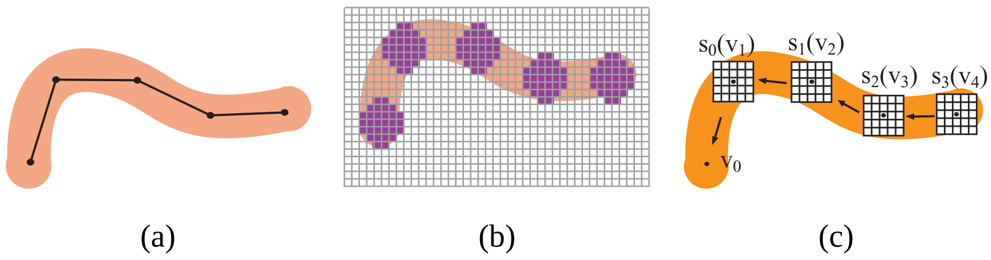


Figure 6.1: Approximating the improvement of a stroke. (b) A stroke is created as a cubic B-spline; the stroke shape is specified by a few control points. (b) The improvement of a stroke is approximated by the sum of the improvements of discs centered at each control point. Each disc has radius R. The improvement of placing a disc of radius R at location (x, y) is stored as an image $I_R(x, y)$. Although not illustrated in the figure, control points are roughly R pixels apart from each other. (b) The image I_p stores improvement to the painting due to each pixel. $I_R(x, y)$ is computed by summing I_p over a disc of radius R. The images I_R and I_p are computed lazily, in order to avoid redundant computation. (c) Dynamic programming for locating a locally optimal stroke shape. A $w \times w$ table is constructed around every control point, except the first. $s_{i-1}(v_i)$ is a table that contains the energy of the optimal stroke ending with vertex v_i. The energy of the optimal stroke of length n can be found by searching the n-th table: $e_{min} = \min_{v_n} s_{n-1}(v_n)$.

will take the same place as the old stroke in the painting; e.g. if S is completely hidden, then T may be as well.) Summing $I_R(x,y)$ over the control points for a stroke gives an approximation to the appearance and coverage improvement terms due to placing the stroke.

With these functions, we can modify a stroke as follows:

```
Measure $E(P)$
Select an existing stroke $S \in P$
Remove the stroke, and measure $E(P - S)$
Optionally, modify some attribute of the stroke
Relax the stroke:
    $S' \leftarrow S$
    **do** $lastImp \leftarrow I(S')$
        $S' \leftarrow \arg\min_{T \in neighborhood(S')} I(T)$
    **while** $I(S') < \min\{lastImp, H\}$
Measure the new painting energy $E(P + S' - S)$.
$P \leftarrow \arg\min_{Q \in \{P, P-S, P+S'-S\}} E(Q)$
```

The **do** loop in the above text represents the actual stroke relaxation, described in the next section.

The constant H is used to prevent the algorithm from pursuing strokes that appear to be unprofitable. We normally use $H = 0$; H could also be determined experimentally.

Locally Optimal Stroke Search

In this section, we describe a dynamic programming (DP) algorithm for searching for a locally optimal stroke, with a fixed stroke color and brush radius. This is a slightly modified version of an algorithm developed by Amini et al. [AWJ90] for use in finding image contours, based on snakes [KWT87].

We search for the locally optimal control polygon, and fix all other stroke and painting parameters. We first define an energy function $e(T)$ of a stroke T consisting of 2D control points $v_0, v_1, ..., v_{n-1}$:

$$e(T) = \sum_{i=0}^{n-1} e_0(v_i) + \sum_{i=0}^{n-2} e_1(v_i, v_{i+1}) \tag{6.1}$$

For our purposes, the energy $e(T)$ is the improvement $I(P,T,S)$ defined in the previous section. Note that n is also variable; we are searching for the optimal control polygon of any length. For now we will assume that the stroke length is kept constant, and later describe how to modify it. Higher order terms, such as bending energy, may also be added to optimization (e.g. $\sum_{i=0}^{n-3} e_1(v_i, v_{i+1,v_{i+2}})$) as described by Amini et al. [AWJ90]; however, the complexity is exponential in the order of the energy, so we have used only the above terms.

The main loop of the search is as follows: given a current stroke, we want to find new control points in the neighborhood of the current control points (Figure 6.1(c)). Each control point has a window of $w \times w$ pixels in which we search for the best nearby control point. We normally use $w = 5$, since $w = 3$ did not seem to explore enough of the space, and $w = 7$ is too slow.

We start by building the DP tables as:

$$\begin{aligned}
s_0(v_1) &= e_0(v_1) + \min_{v_0} e_0(v_0) + e_1(v_0, v_1) \\
s_1(v_2) &= e_0(v_2) + \min_{v_1} s_0(v_1) + e_1(v_1, v_2) \\
&\vdots \\
s_{i-1}(v_i) &= e_0(v_i) + \min_{v_{i-1}} s_{i-2}(v_{i-1}) + e_1(v_{i-1}, v_i) \\
&\vdots \\
s_{n-1}(v_n) &= e_0(v_n) + \min_{v_{n-1}} s_{n-2}(v_{n-1}) + e_1(v_{n-1}, v_n)
\end{aligned}$$

It can easily be shown by induction that these tables have the property that:

$$s_{i-1}(v_i) = \min_{v_0,\ldots,v_{i-1}} e(v_0, \ldots, v_i)$$

In other words, given a fixed value for control point v_i, the optimal stroke energy for a stroke with this control point can be looked up as $s_{i-1}(v_i)$. In addition to these tables, we also create a table of the value of v_{i-1} that produces the corresponding stroke:

$$P_{i-1}(v_i) = \arg\min_{v_{i-1}} s_{i-2}(v_{i-1}) + e_1(v_{i-1}, v_i)$$

Using these tables, we can see that the optimal choice for the last control point, v_n, is given by

$$v_n = \arg\min_{v_n} s_{n-1}(v_n) \tag{6.2}$$

The rest of the stroke is found recursively:

$$v_{i-1} = P_{i-1}(v_i) \tag{6.3}$$

Furthermore, we can find the length m of the locally optimal curve of *any* length (up to n) simply by searching the tables for the smallest value:

$$m = \arg \min_{minStrokeLength \le m \le maxStrokeLength} \min_{v} s_{m-1}(v) \tag{6.4}$$

This operation is does not require extra overhead, because we can keep track of the minimum entry while building the tables.

When $n < maxStrokeLength$, we also add a control point $v_{n+1} = 2v_n - v_{n-1}$ to the stroke before each relaxation step, and search near this longer stroke. This allows the stroke length to increase as well as decrease during the search.

The computational complexity of a single search step is linear in the length of the stroke, but exponential in the order of the energy: $\Theta(n(w^2)^k)$ energy evaluations are required for a stroke of order k and length n For $k = 2$ and $w = 5$ this means 625 computations per control point per iteration, where each computation may be quite expensive.

For stroke evolution, we chose dynamic programming for the advantage of generality. Gradient-based methods such as used by Kass et al. [KWT87] require the energy function to be differentiable, and it appears that a graph-theory formulation [MB95] of this problem would be NP-complete, although heuristics could be applied.

6.7.2 Fragment Buffer

When we delete a brush stroke, we need to find out how this operation modifies the appearance of the painting. A simple, but exceedingly slow, method would be to rerender the revised painting from scratch. Because brush stroke deletion is a very common operation in the relaxation process, we need a faster way to delete brush strokes.

We choose an approach similar to an A-buffer [Car84], called a *fragment buffer*. Each pixel in the image is associated with a list of the fragments that cover that pixel, as well as their associated stroke indices. A fragment is defined as a single RGBA value, and each list is sorted by depth. In our current implementation, this is equivalent to sorting by the order of stroke creation.

To compute the color of a pixel, we composite its fragment list from bottom to top, unless the top fragment is opaque. Creating or deleting a stroke requires generating or deleting one fragment for each pixel covered by the stroke. In our experiments, the length of a typical fragment list rarely exceeds 5 for automatically-generated images.

The fragment buffer is also useful for other operations, such as picking a stroke based from a mouse click, and rerendering strokes with pseudocolors.

There are a variety of alternatives to this method, based on saving partial rendering results; we chose this as a good trade-off between simplicity and efficiency.

6.7.3 Texture Buffer

Our system allows the user to experiment with modifying the stroke rendering styles. Because stroke positions are not modified during this process, it would be wasteful to repeatedly scan-convert strokes while changing colors. We use a texture buffer to avoid this effort. The texture buffer is a fragment buffer with additional texture coordinates provided with each fragment. The fragment also contains partial derivatives for use in the procedural texture; the complete texture fragment data structure contains:

- (u, v) texture coordinates
- Jacobian matrix of partials: $\begin{bmatrix} \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} \end{bmatrix}$, where $s(x, y)$ and $t(x, y)$ define the mapping from screen space to stroke texture space. The partials are computed during

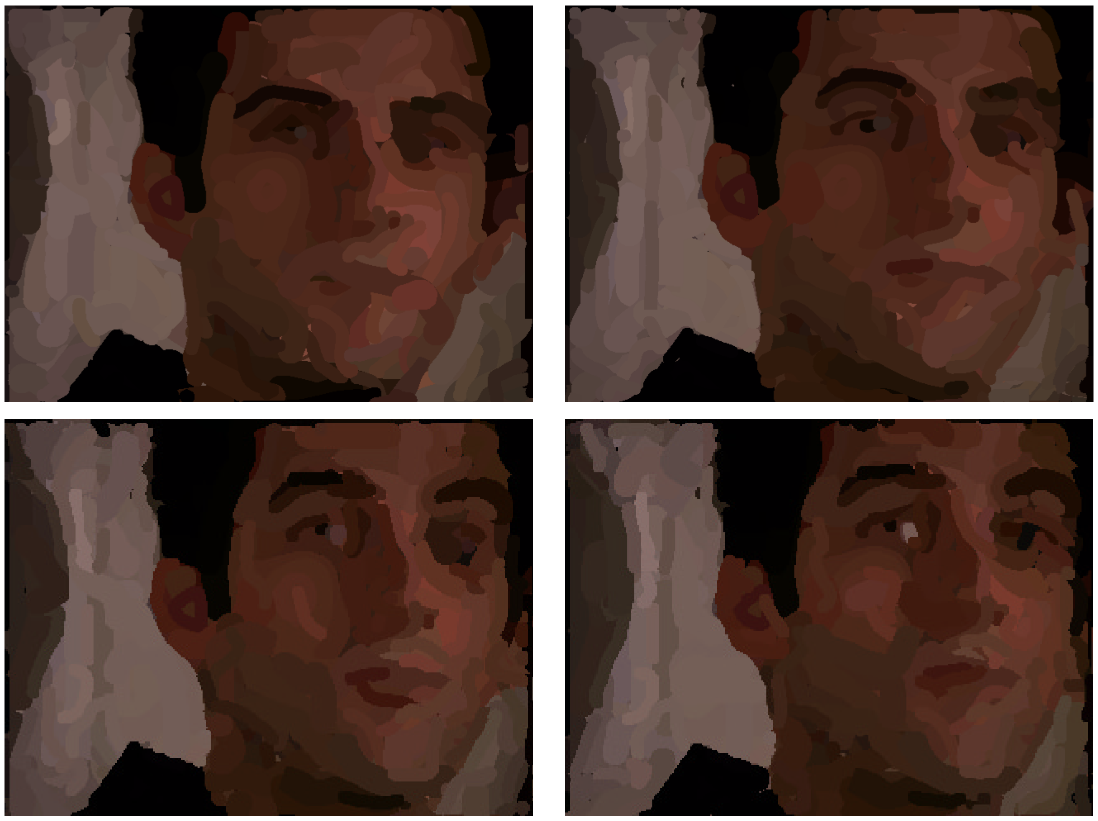


Figure 6.2: Consecutive frames from a painterly animation.

stroke scan-conversion.

- Stroke index, used to look up the stroke color and texturing function.

Before experimenting, we first scan convert all brush strokes into the texture buffer.

Rendering with new textures and colors is then a matter of traversing the texture buffer, calling the appropriate procedure for each texture coordinate, and compositing. The texture buffer could also be used to incorporate texture into relaxation; we currently do not do this for efficiency concerns.

6.7.4 Parallel Implementation

We have implemented a parallel version of the relaxation algorithm for better performance. A server system contains the main copy of the painting, and client machines provide suggestions to the server. One connection is used between each client and the server; in our implementation, this takes the form of a single UNIX socket on the server. Whenever a client generates a suggestion, it is sent to the server over the socket. These suggestions are kept in a queue on the server, and are tested in the order they were received. Whenever the server commits a change to the painting, it is announced over the socket, and adopted by all clients in their local copies of the painting. Although there are many ways of generating suggestions, all suggestions can be written as one of two types of messages: **Delete** stroke number n, or **Create/modify** stroke number n to have radius R, color C, and control polygon P. (There is no distinction between "create" and "modify" commands.) The same protocol is used for sending suggestions to the server as for announcing changes to the clients. Two other message types are used: one message tells clients to advance to the next frame in a video sequence, and another announces changes to the style parameters made by the user.

This system is limited by the main processor's speed in processing suggestions. We have also devised (but not implemented) a decentralized version in which any processor may commit changes to a painting after acquiring a lock on the appropriate image region. In this setup, a main copy of the painting resides on shared storage. Whenever a processor generates a suggestion, it is placed on a global suggestion queue. Before computing a suggestion, a processor must first test every suggestion in the queue and clear it out if the queue is not already in use, to keep the painting up-to-date. The image is broken into blocks, and before testing a suggestion, the processor acquires a write lock on the affected image blocks.

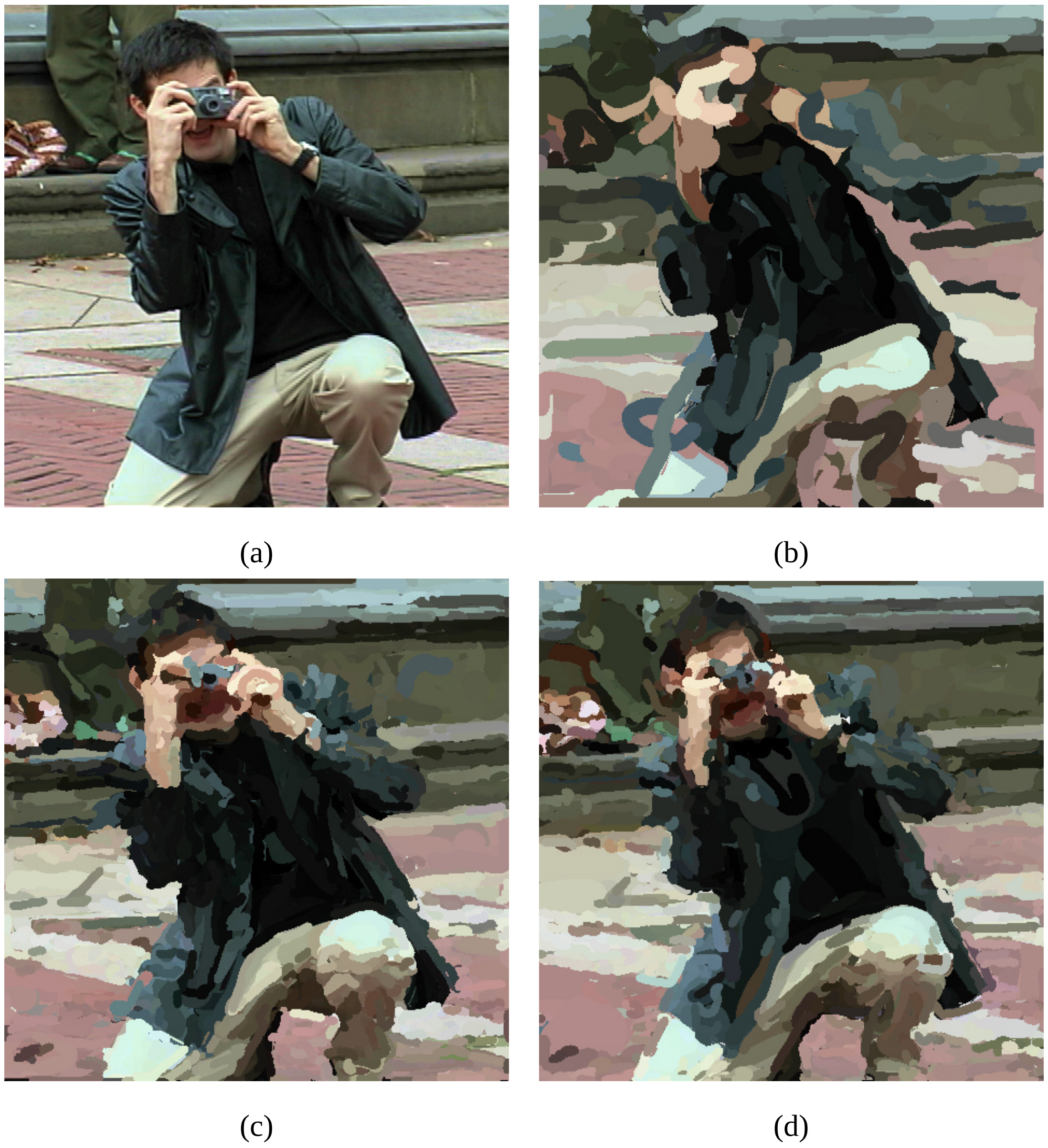


Figure 6.3: A source image, and painterly renderings with the method of [Her98] with 1, 2, and 3 layers, respectively. The algorithm has difficulty capturing detail below the stroke size.

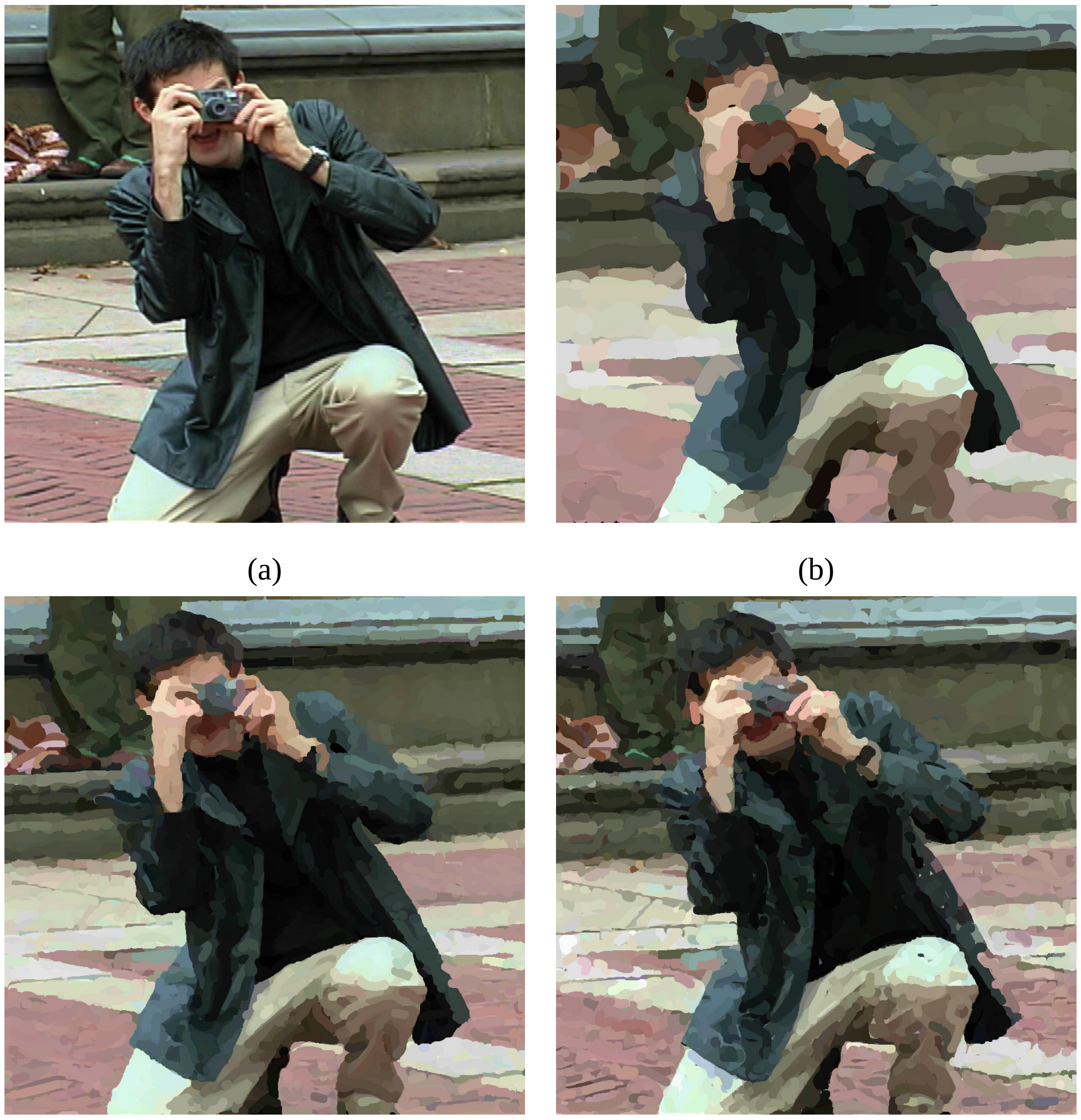

(a) (b) (c) (d)

Figure 6.4: Source image and paintings generated with relaxation, with one and two layers, for comparison to Figure 6.3. (d) A looser painting style.

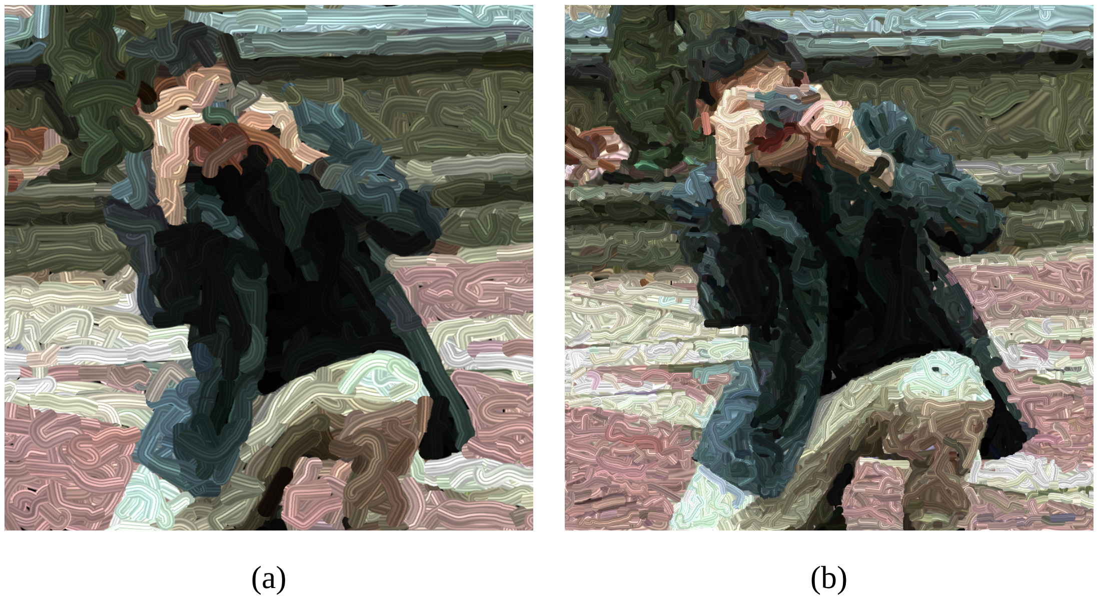


Figure 6.5: Procedural brush textures applied to Figure 6.4(a) and Figure 6.4(c).

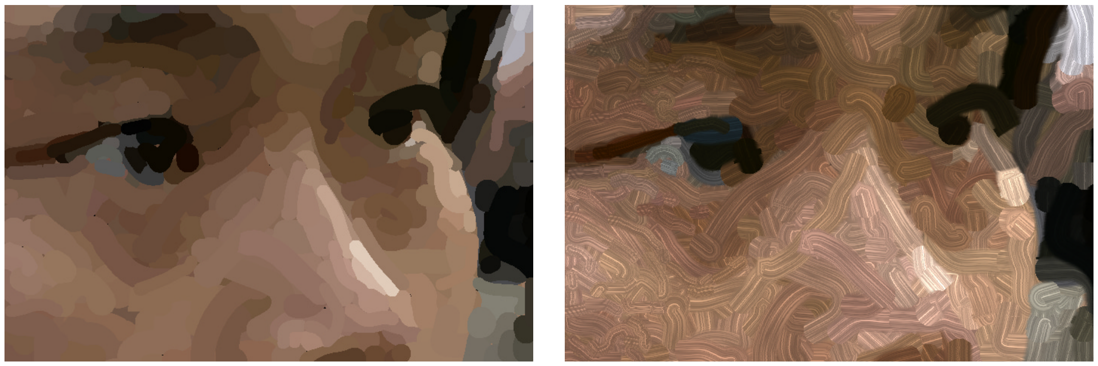
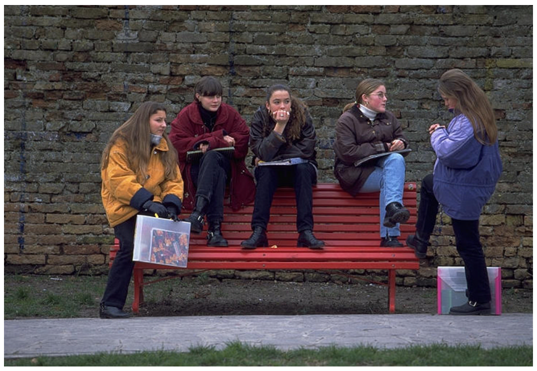
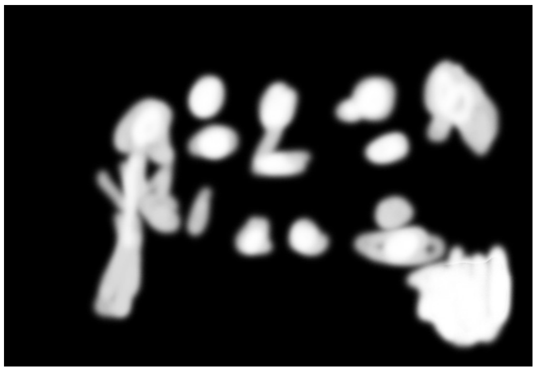


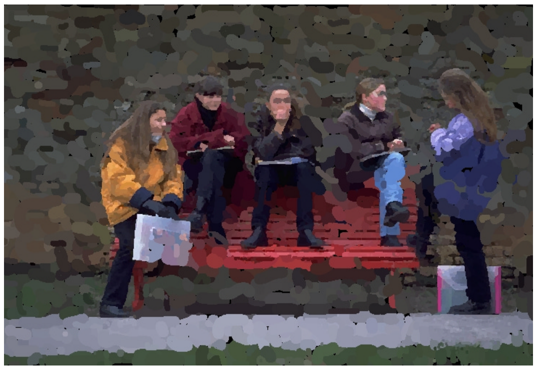
Figure 6.6: A looser painting style, with and without texture



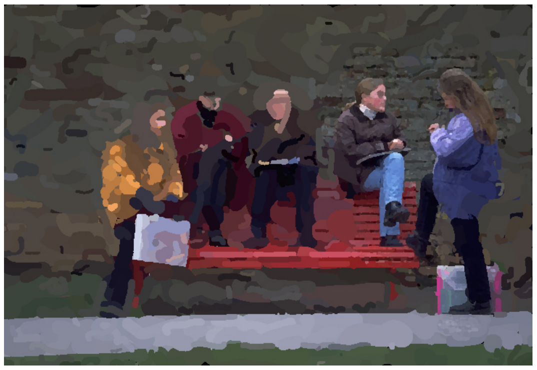
(a)



(b)



(c)



(d)

Figure 6.7: Spatially-varying style provides a useful compositional tool for directing attention. (a) Source image (courtesy of Philip Greenspun, http://photo.net/philg). (b) Interactively painted weight image (w_{app}). (c) Painting generated with the given weights. More detail appears near faces and hands, as specified in the weight image. (d) A more extreme choice of weights and focus. Detail is concentrated on the rightmost figures.

Chapter 7

Image Analogies

a·nal·o·gy n. *A systematic comparison between structures that uses properties of and relations between objects of a source structure to infer properties of and relations between objects of a target structure.* [Eli]

A native talent for perceiving analogies is . . . the leading fact in genius of every order.

—William James, 1890 [Jam90]

Analogy is a basic reasoning process, one that we as humans employ quite commonly, and often unconsciously, to solve problems, provide explanations, and make predictions [SD96a].In this chapter, we explore the use of analogy as a means for creating complex image filters (Figure 1). In particular, our algorithm solves the following problem[1]:

Problem ("IMAGE ANALOGIES"): Given a pair of images A and A' (the *unfiltered* and *filtered source images*, respectively), such that A' is generated a Markov random field (MRF) process from A, along with some additional *unfiltered target*

[1]This chapter describes joint work with Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin [HJO$^+$01].

image B, synthesize a new *filtered target image* B' such that

$$A : A' :: B : B'$$

In other words, we want to find an "analogous" image B' that relates to B in "the same way" as A' relates to A. The MRF assumption on A' states that pixels in the image are governed by statistics that are *local* and *stationary*, that is, that the value of a pixel in A' may be predicted knowing only the values of the pixels in its local neighborhood in A and A'. See [ZWM98, FPC] and Section 7.1.3 for more precise definitions.

While few real-world processes fit the MRF assumptions precisely, we have nonetheless found the image analogies algorithms useful as part of a process that involves two stages. In the *design* (or *training*) phase, a designer (possibly an expert) creates a filter by selecting the training images A and A' (for example, from scanned imagery), annotating the images if desired, and (directly or indirectly) selecting parameters that control how various types of image features will be weighted (and traded off) in the image analogy. The filter can then be stored away in a library. Later, in the *application* phase, a user (possibly someone with no expertise at all in creating image filters) applies the filter to some target image B. For example, an artistic filter could be created by an expert and then provided in a commercial image processing package. Alternately, the filter may be designed in a production studio for use by professional artists and animators. In the latter case, the filter will probably allow/require more input from the artist.

An advantage of image analogies is that they provide a very natural means of specifying image transformations. Rather than selecting from a myriad of different types of filters and parameters, a user can simply supply an appropriate exemplar (along with a corresponding unfiltered source image) and say, in effect: "Make it look like this." Ideally, image analogies should make it possible to learn very complex and non-linear image filters—for instance, filters that can convert a photograph into various types of artistic renderings having the appearance of oil, pastel, or pen-and-ink, by analogy with actual (real-life) renderings in these styles. In addition, these various types of filters would not need to be invented individually or programmed explicitly into a rendering system; ideally, the same general mechanism could be used instead to provide this very broad variety of effects.

While image analogies are clearly a desirable goal, it is not so clear how they might be achieved.

For one thing, a crucial aspect of the "Image Analogies" problem statement is the definition of "similarity" used to measure not only the relationship between each unfiltered image and its respective filtered version, but also the relationship between the source pair and the target pair when taken as a whole. This issue is tricky, in that we want to use some metric that is able to preserve recognizable features of the original image filter from A to A', while at the same time is broad enough to be applied to some completely different target image B. Moreover, it is not obvious what features of a training pair constitute the "style" of the filter: in principle, an infinite number of different transformations could be inferred from a pair of images. In this chapter, we use a similarity metric that is based on an approximation to a Markov random field model, using raw pixel values and, optionally, steerable filter responses [SF95]. To measure relationships between the source and target image pair, we model joint statistics of small neighborhoods within the images.

In addition, for whatever similarity metric we choose, we would like the synthesis of the filtered target image B' to proceed at a reasonable rate. Thus, we will need a way to index and efficiently search over the various images A, A', and B, using the similarity metric, to choose the appropriate parts of the transform $A \to A'$ in synthesizing $B \to B'$. We use an autoregression algorithm, based primarily on recent work in texture synthesis by Wei and Levoy [WL00] and Ashikhmin [Ash01]. Indeed, our approach can be thought of as a combination of these two approaches, along with a generalization to the situation of corresponding *pairs* of images, rather than single textures.

Finally, in order to allow statistics from an image A to be applied to an image B with completely different colors, we sometimes use an additional luminance- or color-matching preprocessing step.

Obviously, we cannot expect our image analogies framework to do a perfect job in learning and simulating all possible image filters, especially from just a single training pair. Moreover, many of the filters that we would like our framework to be able to learn, such as a watercolor painting style, are in fact *extremely* difficult even for humans to master. Nevertheless, we have found image analogies to work rather surprisingly well in a variety of situations, as demonstrated in Section 7.2. These include:

- ***traditional image filters***, such as blurring or "embossing" (Section 7.2.1);
- ***super-resolution***, in which a higher-resolution image is inferred from a low-resolution source (Section 7.2.2);
- ***improved texture synthesis***, in which some textures are synthesized with higher quality than previous approaches (Section 7.2.3);
- ***texture transfer***, in which images are "texturized" with some arbitrary source texture (Section 7.2.4);
- ***artistic filters***, in which various drawing and painting styles, including oil, pastel, and pen-and-ink rendering, are synthesized based on scanned real-world examples (Section 7.2.5); and
- ***texture-by-numbers***, in which realistic scenes, composed of a variety of textures, are created using a simple "painting" interface (Section 7.2.6).

In all of these cases, producing the various different effects is primarily just a matter of supplying different types of source image pairs as input. For example, a blur filter is "learned" by supplying an image and its blur as the (A, A') pair. Similarly, an oil-painting style is learned by supplying an image and its oil-painted equivalent as the input pair. Ordinary texture synthesis can be viewed as a special case of image analogies in which the unfiltered images A and B are null (i.e., considered to match trivially everywhere), and the analysis/synthesis is performed just on A' and B'. Alternatively, texture-by-numbers is achieved by using a realistic image, such as a landscape, as A' and supplying a simplified, hand-segmented version of the landscape as A—for instance, where one solid color in A corresponds to “sky texture” in A', another to “grass texture,” and so on. These same colors can then be painted onto B to generate a new realistic landscape B' with similar textures.

Finally, we also describe a real-time, interactive version of our algorithm, which can be used to provide image analogies underneath the footprint of a digital painting tool (Section 7.3). While texture-by-numbers is a natural application for this tool, it can be used with any type of image analogy.

While successful in many ways, image analogies do not work in every case since they attempt to model only low-level statistics of the image pairs. Thus, higher-level features such as broad, coherent brush strokes are not always captured or transferred very effectively. Chapter 8 discusses the limitations of our approach and suggests areas of future research.

7.1 Image analogies

Here, we describe a set of data structures and algorithms to support image analogies.

7.1.1 Definitions and data structures

As input, our algorithm takes a set of three images, the *unfiltered source image* A, the *filtered source image* A', and the *unfiltered target image* B. It produces the *filtered target image* B' as output.

Our approach assumes that the two source images are *registered*; that is, the colors at and around any given pixel p in A correspond to the colors at and around that same pixel p in A', through the image filter that we are trying to learn. Thus, we will use the same index p to specify both a pixel in A and its corresponding pixel in A'. We will use a different index q to specify a pixel in the target pair B and B'.

For the purposes of this exposition, we will assume that the various images contain not just an RGB color, but additional channels of information as well, such as luminance and various filter responses. Together, all of these channels (including RGB) comprise the *feature vector* for each pixel p. We use $A(p)$ (or $A'(p)$) to denote the complete feature vector of A (or A') at pixel p and, similarly, $B(q)$ or $B'(q)$ to specify their feature vectors at pixel q. Note that the features used for the A and B images need not be the same as for the A' and B' images. The particular features we use are described in more detail in Section 7.1.4 below (however, experimenting with alternate or additional features is certainly a rich area for future research). As we shall see, these features will be used to guide the matching process, in order to help select the most suitable pixels from A' to use in the synthesis of B'.

Finally, our algorithm will need to keep track of the position p of the source from which a pixel was selected in synthesizing a pixel q of the target. Thus, we will store an additional data structure $s(\cdot)$ (for "source"), which is indexed by q, and has the property $s(q) = p$.

In summary, our algorithm maintains the following data structures, of which the RGB channels of $A(p)$, $A'(p)$, and $B(q)$ are inputs, the RGB channels of $B'(q)$ is the output, and the other channels of A, A', B, and B', as well as $s(q)$, are intermediate computed results in the synthesis process:

$A(p)$:	**array** $p \in [1..\textit{SourceWidth}, 1..\textit{SourceHeight}]$ **of** *Feature*
$A'(p)$:	**array** $p \in [1..\textit{SourceWidth}, 1..\textit{SourceHeight}]$ **of** $\textit{Feature}'$
$B(q)$:	**array** $q \in [1..\textit{TargetWidth}, 1..\textit{TargetHeight}]$ **of** *Feature*
$B'(q)$:	**array** $q \in [1..\textit{TargetWidth}, 1..\textit{TargetHeight}]$ **of** $\textit{Feature}'$
$s(q)$:	**array** $q \in [1..\textit{TargetWidth}, 1..\textit{TargetHeight}]$ **of** *Index*

We will actually use a multiscale representation of all five of these quantities in our algorithm. Thus, we will typically index each of these arrays by their multiscale level ℓ using subscripts. For example, if A_ℓ represents the source image A at a given resolution, then $A_{\ell-1}$ represents a filtered and subsampled version of the image at the next coarser level, with half as many pixels in each dimension. We will use L to denote the maximum level, i.e., the level for the highest-resolution version of the images.

7.1.2 The algorithm

Given this notation, the image analogies algorithm is easy to describe. First, in an initialization phase, a multiscale (Gaussian pyramid) representation of A, A', and B is constructed, along with their feature vectors and some additional indices used for speeding the matching process (e.g., an approximate-nearest-neighbor search (ANN), as described below). The synthesis then proceeds from coarsest resolution to finest, computing a multiscale representation of B', one level at a time. At each level ℓ, statistics pertaining to each pixel q in the target pair are compared against statistics for every pixel p in the source pair, and the "best" match is found. The feature vector $B'_\ell(q)$ is then set to the feature vector $A'_\ell(p)$ for the closest-matching pixel p, and the pixel that matched best is recorded in $s_\ell(q)$.

The algorithm can be described more precisely in pseudocode as follows:

```
function CreateImageAnalogy($A$, $A'$, $B$):
    Compute Gaussian pyramids for $A$, $A'$, and $B$
    Compute features for $A$, $A'$, and $B$
    Initialize the search structures (e.g., for ANN)
    for each level $\ell$, from coarsest to finest, do:
        for each pixel $q \in B'_\ell$ do:
            $p \leftarrow$ BestMatch($A$, $A'$, $B$, $B'$, $s$, $\ell$, $q$)
            $B'_\ell(q) \leftarrow A'_\ell(p)$
            $s_\ell(q) \leftarrow p$
    return $B'_L$
```

The heart of the image analogies algorithm is the BestMatch subroutine. This routine takes as input the three complete images A, A' and B, along with the partially synthesized B', the source information s, the level ℓ, and the pixel q being synthesized in B'. It finds the pixel p in the source pair that best matches the pixel being synthesized, using two different approaches: an *approximate search*, which attempts to efficiently find the closest-matching pixel according to the feature vectors of p, q, and their neighborhoods; and a *coherence search*, based on Ashikhmin's approach [Ash01], which attempts to preserve coherence with the neighboring synthesized pixels. In general, the latter approach will usually not return a pixel that matches as closely with respect to the feature vectors; however, for many applications (for example, whenever the impression of coherent brush strokes is desired) we may want to give some extra weight to the choice returned by the coherence search. We therefore rescale the approximate-search distance according to a *coherence parameter* κ, in order to make it artificially larger when comparing the two choices. Thus, the larger the value of κ, the more coherence is favored over accuracy in the synthesized image. We typically use values of κ between 2 and 15 for non-photorealistic filters, and values between 0.5 and 2 for texture synthesis.

Here is a more precise statement of this algorithm:

```
function BESTMATCH($A$, $A'$, $B$, $B'$, $s$, $\ell$, $q$):
    $p_{\text{app}} \leftarrow$ BESTAPPROXIMATEMATCH($A$, $A'$, $B$, $B'$, $\ell$, $q$)
    $p_{\text{coh}} \leftarrow$ BESTCOHERENCEMATCH($A$, $A'$, $B$, $B'$, $s$, $\ell$, $q$)
    $d_{\text{app}} \leftarrow \|F_\ell(p_{\text{app}}) - F_\ell(q)\|^2$
    $d_{\text{coh}} \leftarrow \|F_\ell(p_{\text{coh}}) - F_\ell(q)\|^2$
    if $d_{\text{coh}} \leq d_{\text{app}}(1 + 2^{\ell-L}\kappa)$ then
        return $p_{\text{coh}}$
    else
        return $p_{\text{app}}$
```

Here, we use $F_\ell(p)$ to denote the concatenation of *all* the feature vectors within some neighborhood $N(p)$ of both source images A and A' at both the current resolution level ℓ and at the coarser resolution level $\ell - 1$. We have used 5×5 neighborhoods in the fine level and 3×3 neighborhoods in the coarse level. (See Figure 7.1.) Similarly, we use $F_\ell(q)$ to denote the same concatenation for the target images B and B', although in the case of the filtered target image B' the neighborhood at the finest resolution includes only the portion of the image that has already been synthesized. (Note that $F(\cdot)$ is overloaded in our notation; the index p or q will be used to determine whether a particular $F(\cdot)$ is a source or target feature vector.) In each case, the norm $\|F_\ell(p) - F_\ell(q)\|^2$ is computed as a weighted distance over the feature vectors $F(p)$ and $F(q)$, as described in Section 7.1.4. Note that some special processing is required at boundaries, as well as at the lowest resolution of the pyramid, since the neighborhoods are a little bit different in these areas. We perform brute-force searches with only the partial neighborhoods in these cases.

For the BESTAPPROXIMATEMATCH procedure, we have tried using both *approximate-nearest-neighbor search* (ANN) [AMN$^+$98] and *tree-structured vector quantization* (TSVQ) [GG92], using the same norm over the feature vectors. In our experience, ANN generally provides more accurate results for the same computation time, although it is also more memory intensive. We used ANN for all of the examples shown in this chapter. Principal components analysis (PCA) can be used to reduce the dimensionality of feature vectors leading to a substantial speed-up in the search. We generally keep 99% of the variance, which can lead to a reduction in dimensionality of about an order of magnitude. However, using PCA can degrade the quality of the results on some simple cases; it is most useful in cases with large feature vector sizes (e.g., when steerable filters are used).

The BESTCOHERENCEMATCH procedure simply returns $s(r^\star) + q - r^\star$, where

$$r^\star \;=\; \arg\min_{r \in N(q)} \|F_\ell(s(r) + q - r) - F_\ell(q)\|^2$$

and $N(q)$ is the neighborhood of already synthesized pixels adjacent to q in B'_ℓ. This formula essentially returns the best pixel that is coherent with some already-synthesized portion of B'_ℓ adjacent to q, which is the key insight of Ashikhmin's method.

7.1.3 Energy Function

Although the process described here makes only a single pass over the coarsest output image, it is actually an approximation to a Markov random field optimization, such as in [ZWM98, FPC]. In particular, this is defined by minimizing the following energy function:

$$\begin{aligned} E(S) &= \sum_q \|F(q) - F(S(q))\|^2 H(S,q) \\ &= \sum_q (\|N_B(q) - N_A(S(q))\|^2 + \|N_{B'}(q) - N_{A'}(S(q))\|^2) H(S,q) \end{aligned}$$

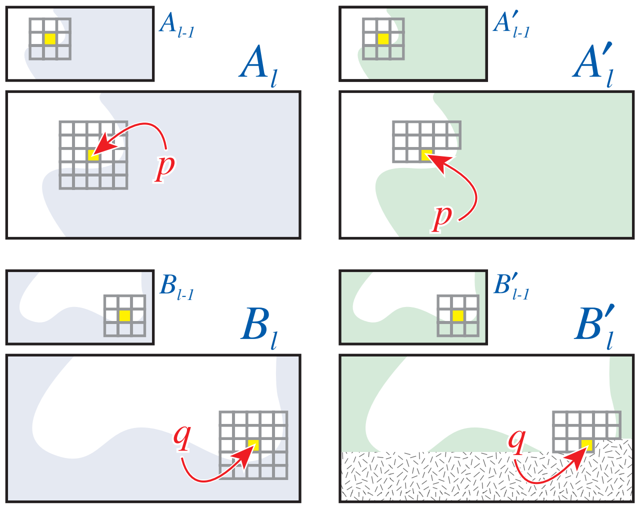


Figure 7.1: Neighborhood matching. In order to synthesize the pixel value at q in the filtered image B'_ℓ, we consider the set of pixels in B'_ℓ, B_ℓ, $B'_{\ell-1}$, and $B_{\ell-1}$ around q in the four images. We search for the pixel p in the A images that give the closest match. The synthesis proceeds in scan-line ordering in B'_ℓ.

$$H(S,q) = \begin{cases} 1 & \text{if } \exists q' \text{ s.t. } \|q-q'\| < c \wedge q-q' = S(q)-S(q') \\ 1+\kappa & \text{otherwise} \end{cases}$$

where N_x are square neighborhoods within single images, and F are the concatenation of unfiltered and filtered square neighborhoods. The $H(S,q)$ term penalizes pixels that are not "coherent;" when $\kappa = 0$, this term has no effect. The c term is a window-size constant. In other words, we wish to arrange the pixels from the example A' image so that, for each pixel in the output, we penalize the difference between its two-image neighborhood in the source and in the target. In this definition, we desire neighborhoods of large size, with weights falling to zero as pixels get further from the neighborhood center.

Note that for pure texture synthesis, this simply reduces to:

$$E_{TS}(S) = \|N_{B'}(q) - N_{A'}(S(q))\|^2 H(S,q)$$

since the unfiltered images are empty, and, thus $\|N_B(q) - N_A(S(q))\| = 0$ for all q.

In order to accomodate small window sizes, we can replace this energy with one that produces an output pyramid by example:

$$E_P(S) \;=\; \sum_{\ell \in [0..L]} \sum_q \|F_\ell(q) - F_\ell(S_\ell(q))\|^2 H(S_\ell, q)$$

This represents an energy over the entire pyramid, not just the last level of the pyramid. The pyramid may be of any type (e.g. Gaussian, Laplacian, steerable, etc.).

The autoregression texture synthesis algorithm that we used is justified by observing that, for any output pixel, the optimal choice for a given pixel is found by searching for the pixel with the best-matching neighborhood. Choosing randomly from among good matches is justified by viewing this problem as random sampling from a space of outputs, e.g. images distributed as $P(S|A, A', B) = \exp(E_P(S))/Z$, where Z is a normalizing constant.

7.1.4 Similarity metric

Feature selection and representation is a large open problem and an active area of research in machine learning. For now, we have experimented with several different components for the feature vectors. Using the RGB channels themselves is the most obvious choice (and the first thing we tried). However, we generally found that our source pairs did not contain enough data to match the target pair well based on color information alone.

An alternative, which we have used to generate many of the results shown in this paper, is to compute and store the luminance at each pixel, and use luminance instead of RGB in the distance metric. The luminance can be computed in a number of ways; we use the Y-channel from the YIQ color space [FvDFH90], where the I- and Q-channels are "color difference" components. This approach is actually well-motivated by vision science: we are much more sensitive to changes in the luminance channel than to changes in color difference channels [Wan95]. After processing in luminance space, we can recover the color simply by copying the I- and Q-channels of the input B image into the synthesized B' image, followed by a conversion back to RGB. (Alternatively, we could use the color preprocessing step described in Section 7.1.5 to approximate the color transform from A to A' and then apply this transform to the I and Q channels of B when copying them to B'.) A shortcoming of this luminance transfer technique is the fact that color dependencies in the analogy filter are lost.

Another way to improve the perceptual results of matching is to compute multiple scales of oriented derivative filters [Bon97, HB95, ZWM98]. We compute a first-order steerable pyramid [SF95] for the luminance of A and B and concatenate these four filter responses to the feature vectors for these images. The distance metric then becomes a weighted combination of similarity in luminance, as well as similarity in orientation among regions of the unfiltered images. We used steerable filters for synthesizing the pen-and-ink examples (Figure 7.10); for our other experiments, we found them to make little or no difference.

For whatever feature vectors we use, we always weight the differences between the feature vectors over the neighborhood about p and q using a Gaussian kernel, so that differences in the feature vectors of pixels further from p and q have a smaller weight relative to these differences at p and q. We also normalize the vectors so that each scale of the pyramid is equally weighted.

7.1.5 Luminance pre-processing

The algorithm as described works best when image A contains neighborhoods that are locally similar to neighborhoods in B. The algorithm will perform poorly when the pixel histograms of A and B are substantially different; for example, a light A will be of little use when processing a dark B. As a pre-conditioning step, we would like to discover a smooth color transformation that brings the histograms into correspondence. Unfortunately, we have yet to find a full solution to this problem, and it remains for us an area of future work. For now, we propose a simple approach to address this problem whenever A and B have sufficiently different color histograms: convert the images to luminance space and compute a linear mapping that matches the first- and second-order statistics of the luminance distributions.

The first step, converting the images to luminance, actually offers two advantages: matching 1D histograms is easier than matching 3D histograms, and (as mentioned before) fewer color channels means faster computations when searching for best feature matches. For the second step, matching the luminance distributions, we can apply histogram matching [Cas96]. In practice, however, we find that histogram matching leads to non-smooth mappings that have undesirable side-effects.

Our approach, instead, is to apply a linear map that matches the means and variances of the luminance distributions. More concretely, if $Y(p)$ is the luminance of a pixel in image A, then we remap it as

$$Y(p) \leftarrow \frac{\sigma_B}{\sigma_A}(Y(p) - \mu_A) + \mu_B$$

where μ_A and μ_B are the mean luminances, and σ_A and σ_B are the standard deviations of the luminances, both taken with respect to luminance distributions in A and B, respectively. We apply the same linear transform to A', in order to keep the training pair consistent.

This same approach can be extended to matching color distributions in a straightforward way (Appendix 8.3). However, we found that luminance matching worked better in our experiments.

7.2 Applications

By supplying different types of images as input, the image analogies framework can be used for learning filters for many different types of applications. We describe here the applications that we have experimented with so far. Timings for these tests are discussed in the next section.

7.2.1 Traditional image filters

As a proof of concept, we first tried learning some simple image-processing filters, including a “blur” filter (Figure 7.2) and an “emboss” filter from Adobe Photoshop (Figure 7.3). While the image-analogies framework gives adequate results, it is nowhere near as efficient as applying the filter directly. Still, these experiments verify that the image analogies framework works for some basic filters.

7.2.2 Super-resolution

Image analogies can be used to effectively “hallucinate” more detail in low-resolution images, given some low- and high-resolution pairs (used as A and A') for small portions of the images. (The image analogies framework we have described is easily extended to handle more than a single source image pair, which is what we have done for these examples.) Figure 7.6 demonstrates this application, using images of a Dobag rug and a set of maple trees, respectively. An interesting area for future work is to choose the training pairs automatically for image compression, similar to fractal image compression [BHA93].

7.2.3 Improved texture synthesis

Texture synthesis is a trivial case of image analogies, where the A and B images are zero-dimensional or constant. The algorithm we have described, when used in this way for texture synthesis, combines the advantages of the weighted L_2 norm and Ashikhmin’s search algorithm, although without the speed of Ashikhmin’s algorithm. For some textures, the synthesized textures have similar high-quality to Ashikhmin’s algorithm, without the edge discontinuities. Figure 7.4 demonstrates some results of this improved algorithm and compares them to previous approaches.

7.2.4 Texture transfer

In texture transfer, we filter an image B so that it has the texture of a given example texture A' (Figure 7.8). Texture transfer is achieved by using the same texture for both A and A'. We can trade off the appearance between that of the unfiltered image B and that of the texture A by introducing a weight w into the distance metric that emphasizes similarity of the (A, B) pair over that of the (A', B') pair. Increasing w ties the image more closely to the texture, while smaller values of w reproduce the input image more faithfully. We typically use values between 0.25 and 1.

This application of image analogies may be somewhat counterintuitive, since an intuitive interpretation of an "analogy" in which A is the same as A' is as the identity filter. This interpretation of the training data is not unique, however, and texture transfer is actually a valid interpretation. Image analogies synthesizes images drawn from the statistical distribution of neighborhoods in A' — in texture transfer, this is done while trying to match the B image as closely as possible. (Image analogies can does learn an identity filter when given training data that adequately samples the space of image neighborhoods.)

Simpler derivations of texture transfer outside of the context of Image Analogies are also possible. In particular, if we modify a texture synthesis algorithm to penalize possible pixels depending on how close its neighborhood matches the corresponding B image, then we get exactly the image analogies texture transfer algorithm:

$$w^2 \|N_{A'}(p) - N_{B'}(q)\|^2 + \|N_{A'}(p) - N_B(q)\|^2 = \|F(p) - F(q)\|^2$$

when $A = A'$, and N_x denotes the single-image neighborhoods that comprise the feature vectors F. Note that the $\|N_{A'}(p) - N_{B'}(q)\|^2$ is the ordinary texture synthesis search term.

7.2.5 Artistic filters

Although the problem is in general very difficult, we have had some success in using image analogies to transfer various artistic styles from one image to another, as shown in Figures 7.10, 7.13 and 7.12.

For many example images, we do not have a source photograph available; hence, a substitute must be created. We generally view the A' image as providing texture (e.g., pen strokes or paint texture) to an untextured image. To create A from A', we apply an anisotropic diffusion [PM90] or similar filter to A' (we used the "Smart Blur" filter from Adobe Photoshop), in order to maintain sharp contours but eliminate texture.

Color and dynamic range are extremely important. If color is used in the similarity metric, then, at the very least, the color histograms of the A and B images should overlap. The situation can be improved by the luminance processing described in Section 7.1.5 (or its color processing variant). In general, we find that matching with color gives richer and more appealing results, but can often fail quite poorly. This is due to the well-known "curse of dimensionality:" the neighborhood-space for RGB images is much larger than for luminance images, and thus a single image pair provides a correspondingly sparser sampling of the space for RGB than for luminance. Consequently, the neighborhood histogram of A may still be poorly matched to B even after linear matching, whereas this is less of a problem for grayscale images.

In many cases, especially pen-and-ink illustrations, using the L_2 norms of raw pixel values gives a perceptually poor result. In these cases, steerable filters generally make a significant improvement. We suspect that this is because pen-and-ink illustrations depend significantly on gradient directions in the input images. In our current implementation, the steerable filter responses are only used for matching in A and B, and not in A'/B'. Steerable filters were not used for the other examples.

When the corresponding unfiltered source image A is known (for example, if a painting was created from a reference photograph), then some care must be taken to register it carefully with the painting, since our algorithm currently assumes that the images are in approximate pointwise correspondence. For our training pairs of this type, we first aligned the images by manually estimating a global translation, rotation, and scale. We then warped the example source image with a custom image warping program designed for local image adjustments (Appendix 8.3).

The scale of the training images determines the fineness of the features in the B' image and may be chosen by the designer.

7.2.6 Texture-by-numbers

Texture-by-numbers allows new imagery to be synthesized by applying the statistics of existing images to a labeling image A. For example, given a labeling of the component textures of a realistic image, a new realistic one may be painted just by painting the arrangement of the component textures (Figure 7.14).

A major advantage of texture-by-numbers is that it allows us to synthesize from images for which ordinary texture synthesis would produce poor results. Consider the photograph of an oxbow shown in Figure 7.14. Although the image has textural regions, attempting to create a new version of the river via ordinary texture synthesis would produce very poor results, as the synthesis process would try to mix unrelated textures. In statistical terms, the problem is that the texture distribution is not stationary. On the other hand, specifying a corresponding A image makes the A' image into a useful texture, in the sense that the *conditional* density of the A' image is stationary (now conditioned on A). Given a new B image, we can generate new scenes. Note that, in addition to filling in textures in a sensible manner, the boundaries between texture regions also match the examples, since they are synthesized from examples in A with similar boundary shapes.

A more sophisticated example is shown in Figure 7.16. Treating the greenery as a single texture produces poor results because the synthesis mixes foreground texture with background texture. In other words, the texture is stationary horizontally but not vertically. In this case, we provide a gradient in the red channel of the A image, which constrains the synthesis so that near elements will not be mixed with far elements.

Texture-by-numbers requires that an appropriate choice of the A image be provided in order to factor out non-stationary components of A'. In our experience, the synthesis is somewhat forgiving, degrading gracefully as the assumptions become less appropriate. In principle, the A image can be of arbitrary dimension and content. For example, it could include additional information about normals, depths, or orientations [SWHS97] of the A' image to improve the texture-by-numbers process.

This application bears some resemblance to the user-guided texturing described by Ashikhmin [Ash01]; however, it fixes several of the problems with that method. (In Ashikhmin's method, multiple passes are usually required for a good match. In addition, the greedy search may create poor matches when a very large example texture is used, since the synthesis cannot "restart" until it finishes copying a patch. More significantly, the colors used in the target must be distinct: the algorithm would have difficulty, for example, distinguishing between green trees and green grass.) Our algorithm also allows for additional channels of information (such as depth, normals, etc.) to be used to control the synthesis.

7.3 Interactive editing

For many applications, the ability to directly manipulate an image via a user interface is crucial. We have developed an application in which a user can "paint" a landscape by coarsely specifying locations for the trees, sky, etc. The main difficulty is that a single change to a B image could theoretically affect the rest of the image, and the full synthesis algorithm is currently too slow to run at interactive rates. However, we can exploit the fact that, in practice, user painting operations only affect a small area of the image at a time, and, under the locality assumption, these operations will have exponentially-decaying influence on distant image pixels. Hence, we can maintain an acceptable image by updating only the modified pixels and their neighbors.

The user interface presents a painting interface for placing RGB values into the B or B' images. The B image is initialized with some default value (e.g., a blank image or a predefined image), and a corresponding B' image is synthesized from the initial B'. The initial B' may also be precomputed.

The key to making interactive painting efficient in this context is to provide an immediate update of B' using a coherence search (Section 7.1.2) as the user paints, and to refine B' with the full search (approximate plus coherence search) progressively, only as processing cycles allow. Our implementation has two threads, an event handler and a synthesis thread. When the user paints into the B image, the event handler queues the painting locations at all scales for updating. The synthesis thread performs a coherence search on the changed pixels in scan-line order, though it uses the full search with causal neighborhoods for every tenth pixel. Pixels that have not been updated are marked ignored when comparing neighborhood distances during coherence search. These pixels are placed in another queue, and, whenever the first queue is empty, the update thread performs full search with non-causal neighborhoods on the contents of the second queue.

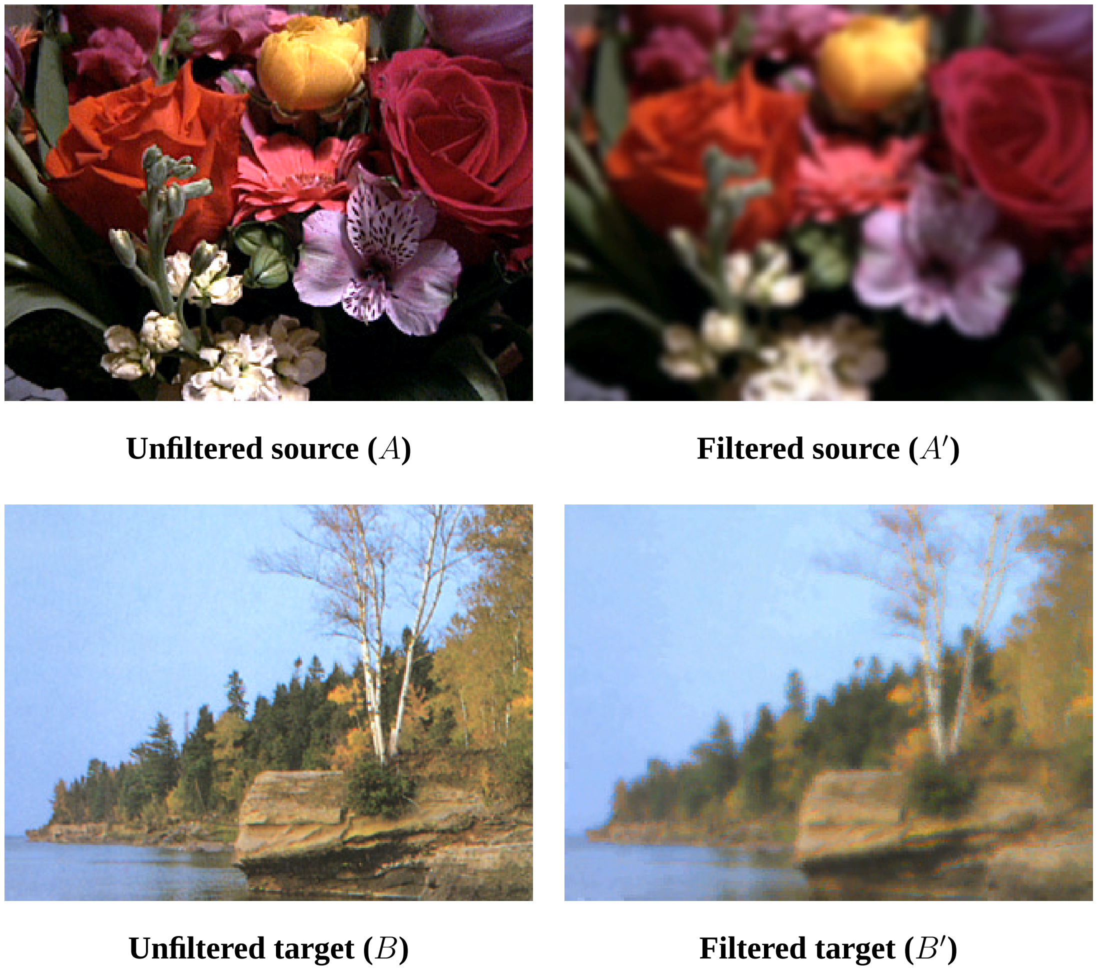
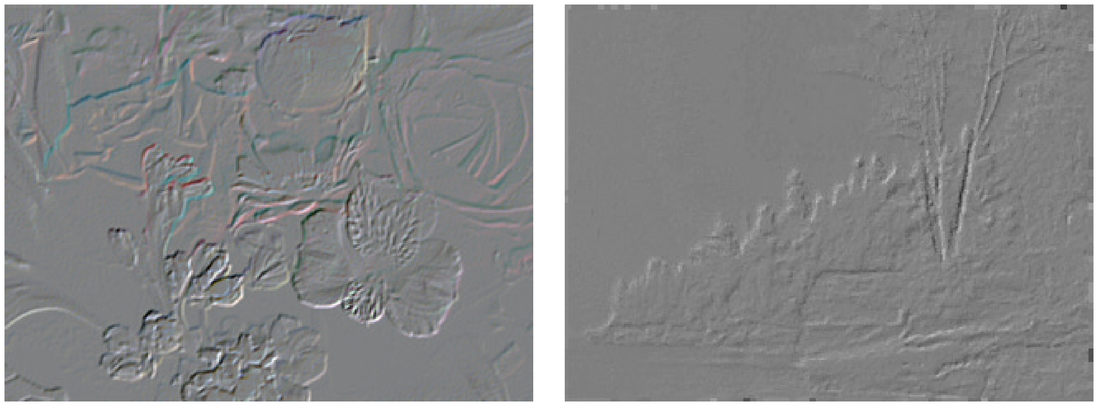

Unfiltered source (A)
Filtered source (A')
Unfiltered target (B)
Filtered target (B')

Figure 7.2: Toy example: Learning a blur filter. The A and A' images comprise the training data. Images were converted to luminance, and then the filter learned from A and A' was applied to B to get B'.



Filtered source (A') **Filtered target (B')**

Figure 7.3: Toy example: Learning an emboss filter. The A and B images are the same as in Figure 7.2.

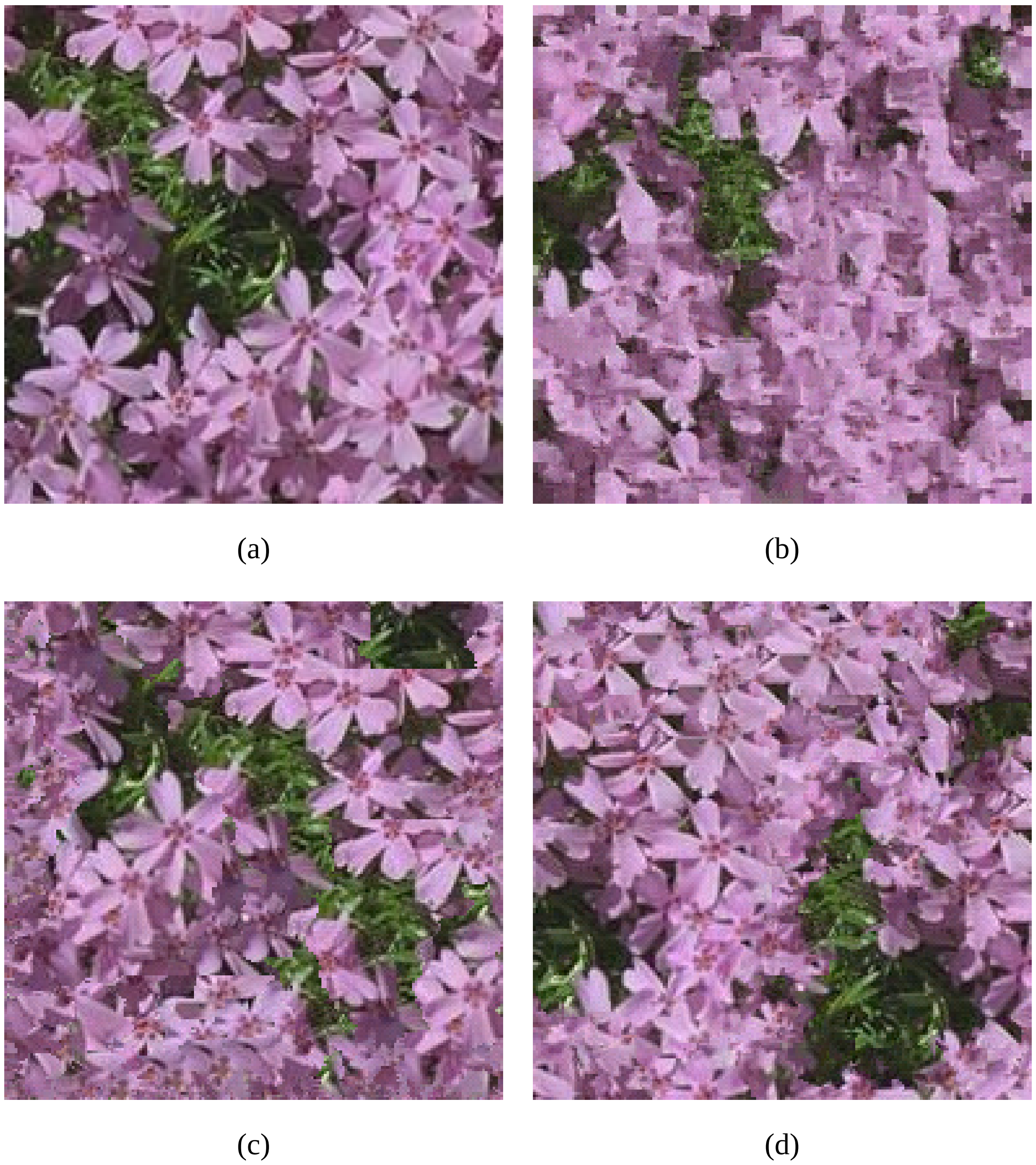


Figure 7.4: Improved texture synthesis. (a) Source texture, obtained from the MIT VisTex web page (Copyright © 1995 MIT). (b) Texture synthesized with Wei and Levoy's algorithm [WL00]. The L_2 norm is a poor measure of perceptual similarity. (c) Ashikhmin's algorithm [Ash01] gives high quality coherent patches, but creates horizontal edges when patches reach the end of the source image. (d) Texture synthesized with our algorithm, which combines the advantages of these two previous methods. We used $\kappa = 5$ for this figure.

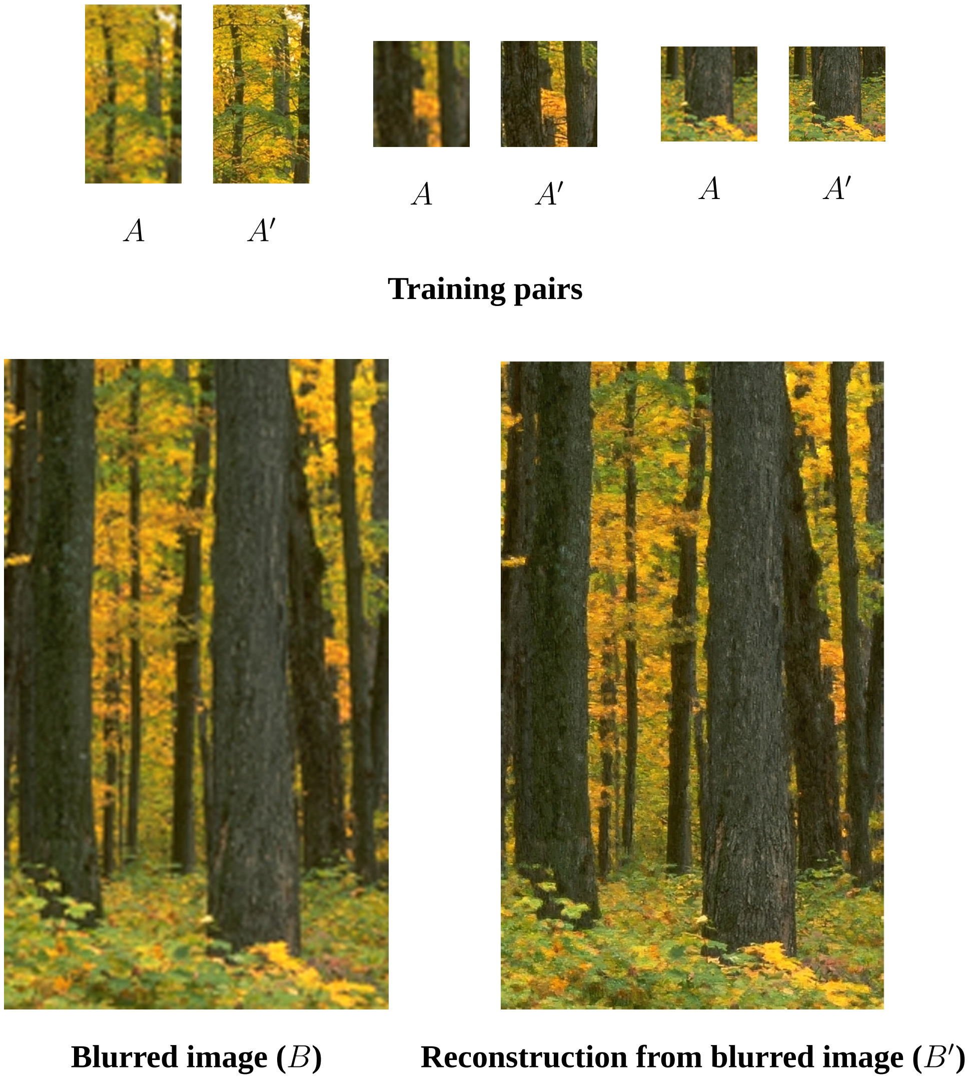


Figure 7.5: Super-resolution. Example training pairs are shown on top. These pairs specify a super-resolution filter that is applied to the blurred image to get the B' image.

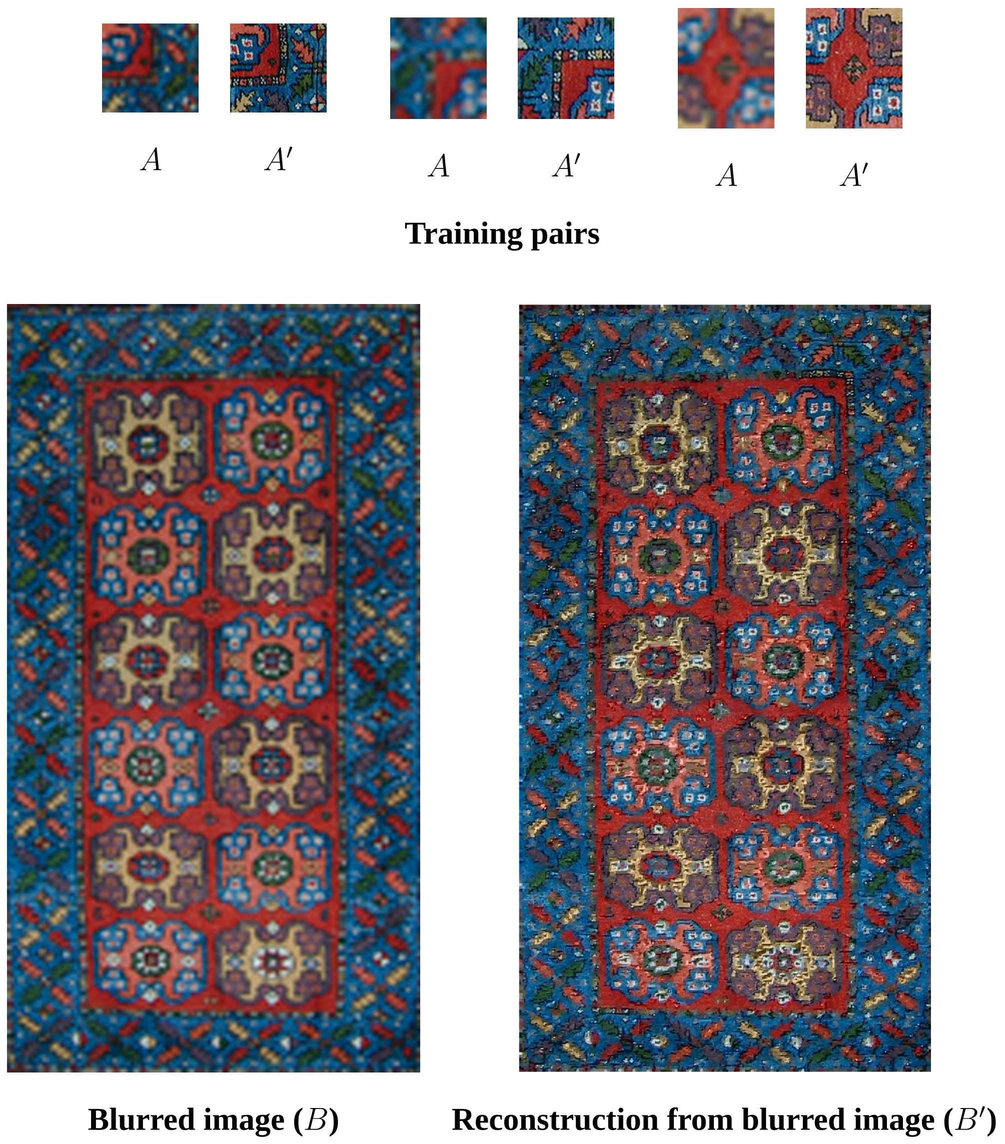


Figure 7.6: Super-resolution. Example training pairs are shown on top. These pairs specify a super-resolution filter that is applied to the blurred image to get the B' image.

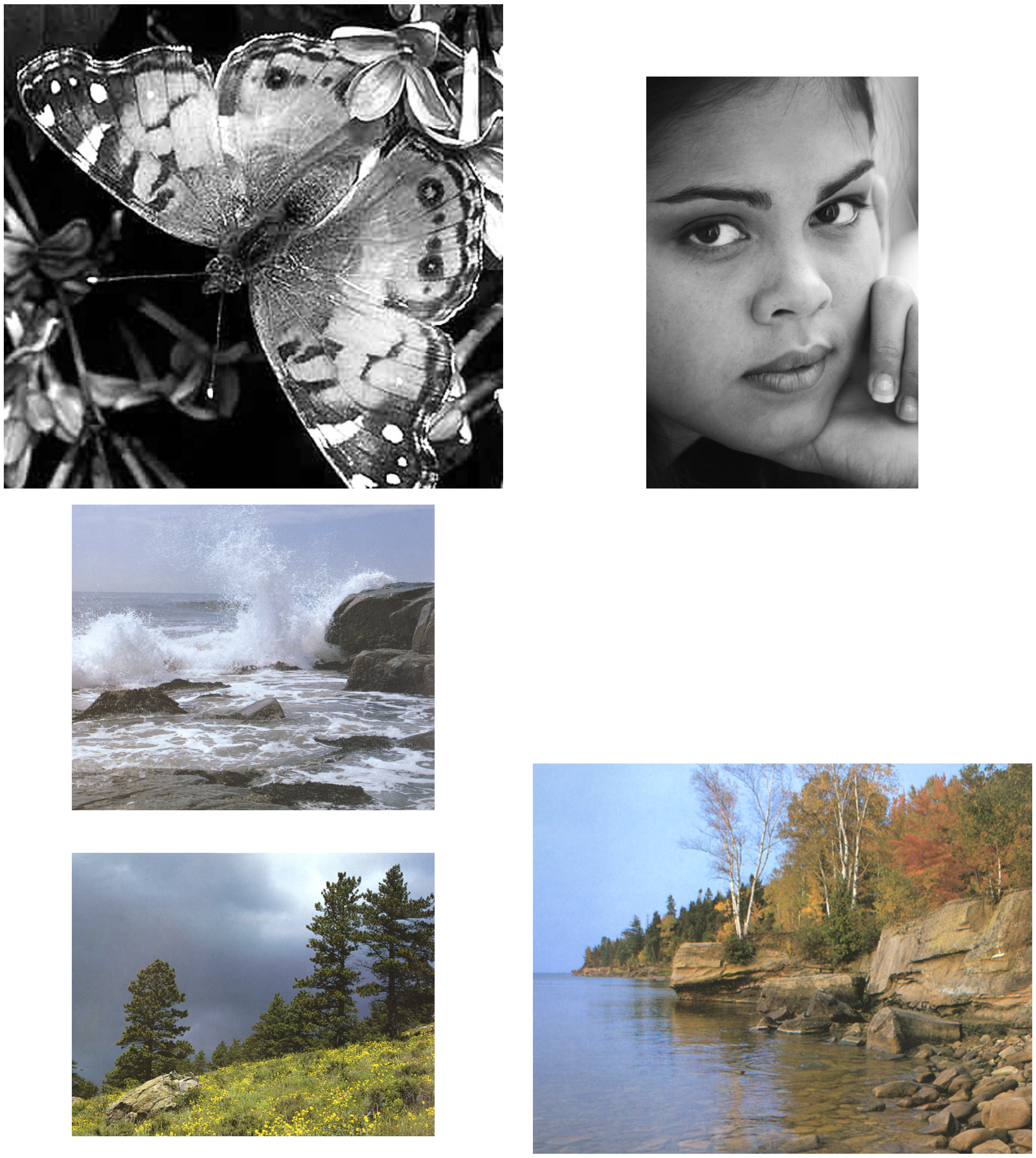


Figure 7.7: Unfiltered target images (B) for the NPR filters and texture transfer.

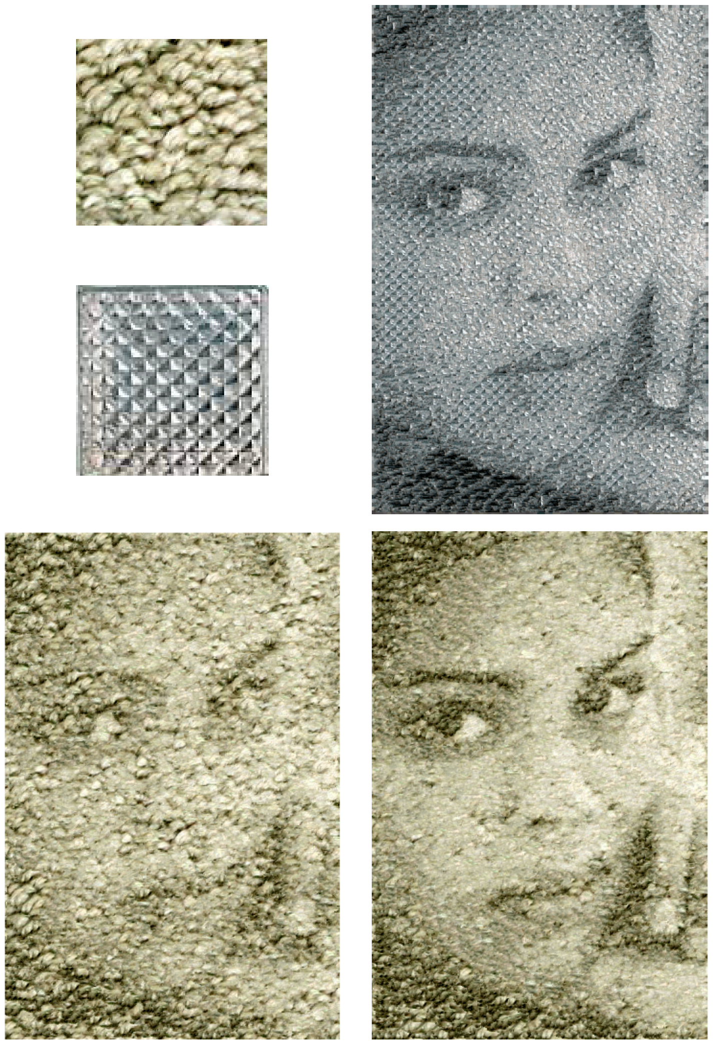
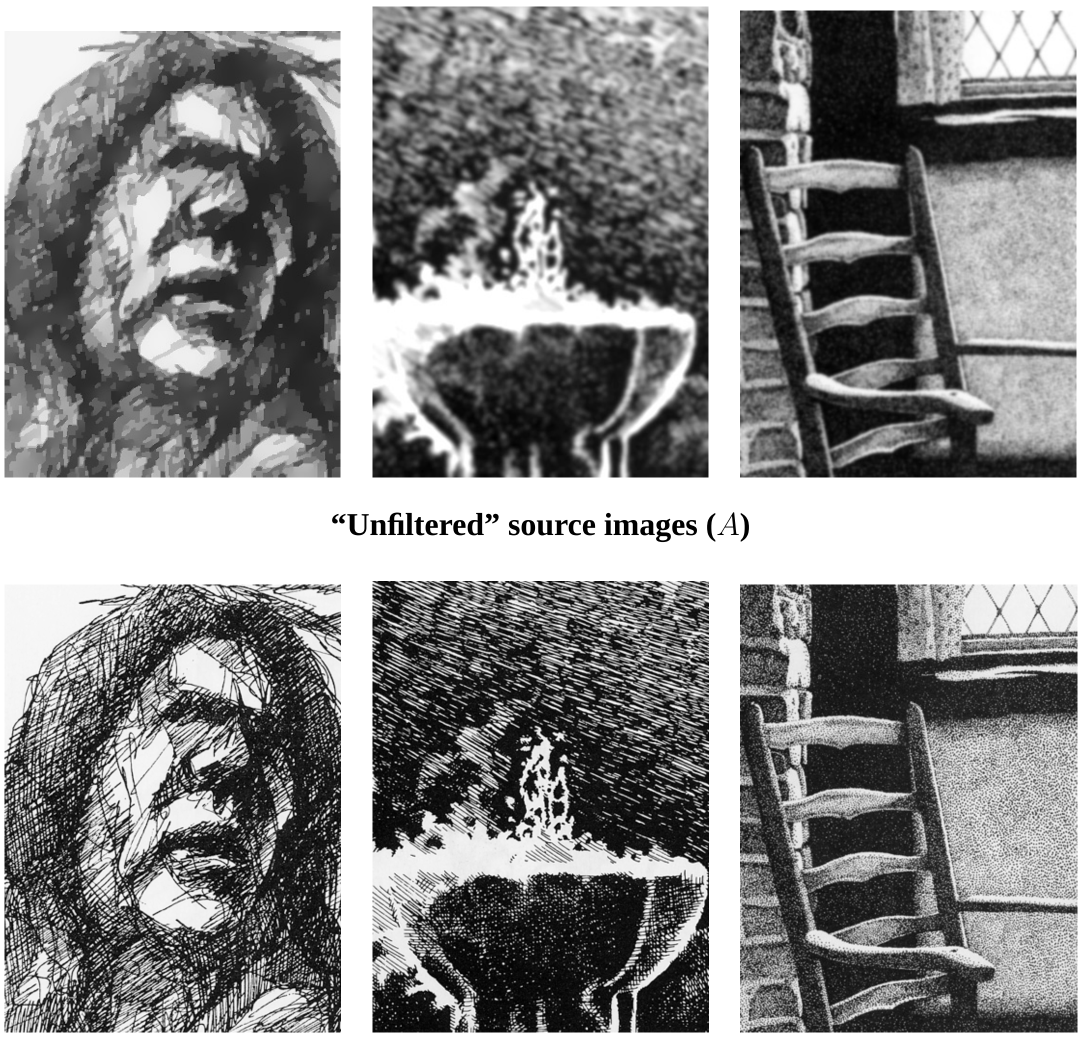


Figure 7.8: Texture transfer. A photograph is processed to have the textures shown in the upper left; each of the other images is an output image B'. The weighting between source image and source texture is used to trade off fidelity to the texture versus fidelity to the source image. The texture is used for both A and A'.


“Unfiltered” source images (A)

“Filtered” images (A')

Figure 7.9: Training pairs for for the pen-and-ink illustrations shown in Figure 7.10. These images were generated by processing the source images with Photoshop’s “Smart Blur” filter.

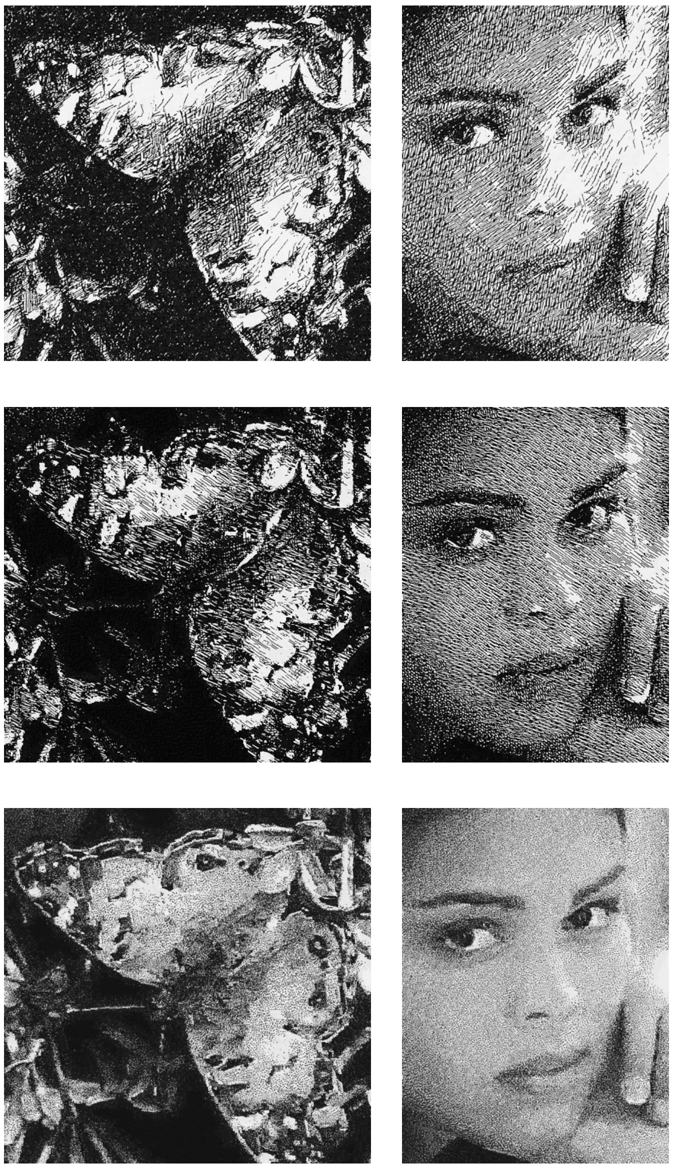


Figure 7.10: Pen-and-ink illustrations by example. The input images are shown in Figures 7.7 and 7.9.

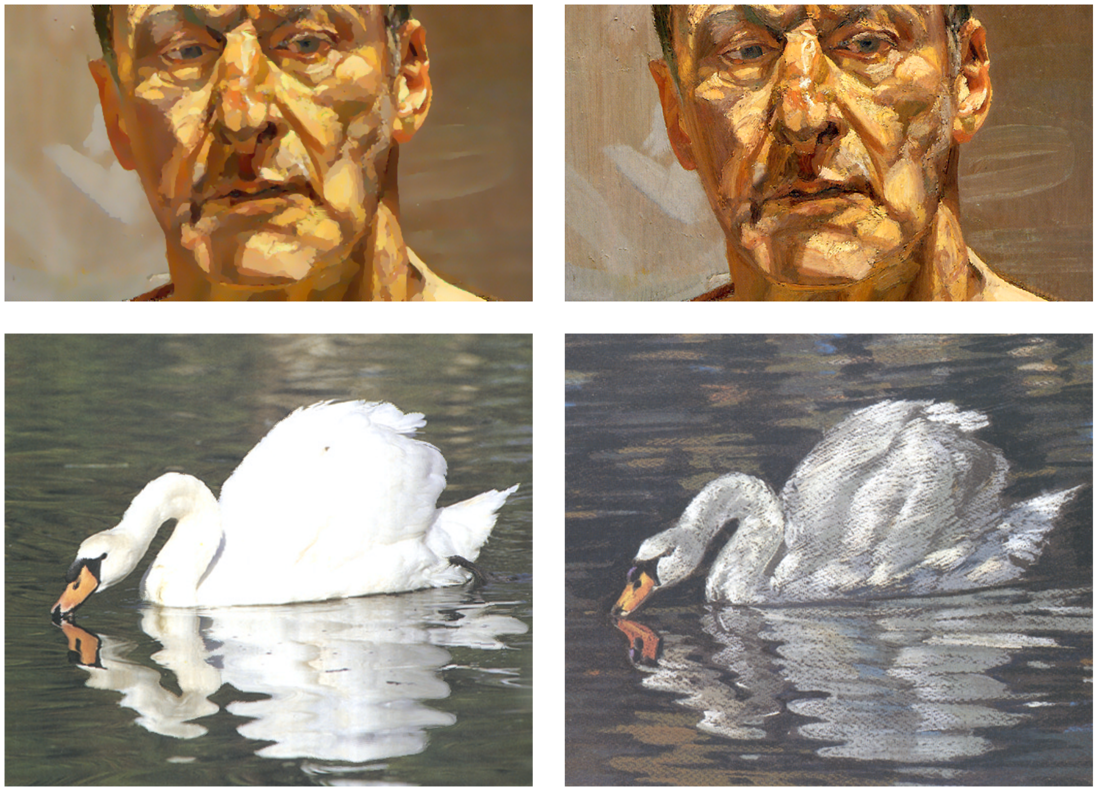


Figure 7.11: Training pairs for the color NPR filters used in this thesis. The upper A' image is a detail of a self-portrait by Lucian Freud in oil paint; the “unfiltered” source image was generated by processing the painting with Photoshop’s “Smart Blur” filter. The bottom image pair is from *The Big Book of Drawing Nature in Pastel* by Schaeffer and Shaw [SS93], an instructional book that shows photographs along with pastel illustrations of them.

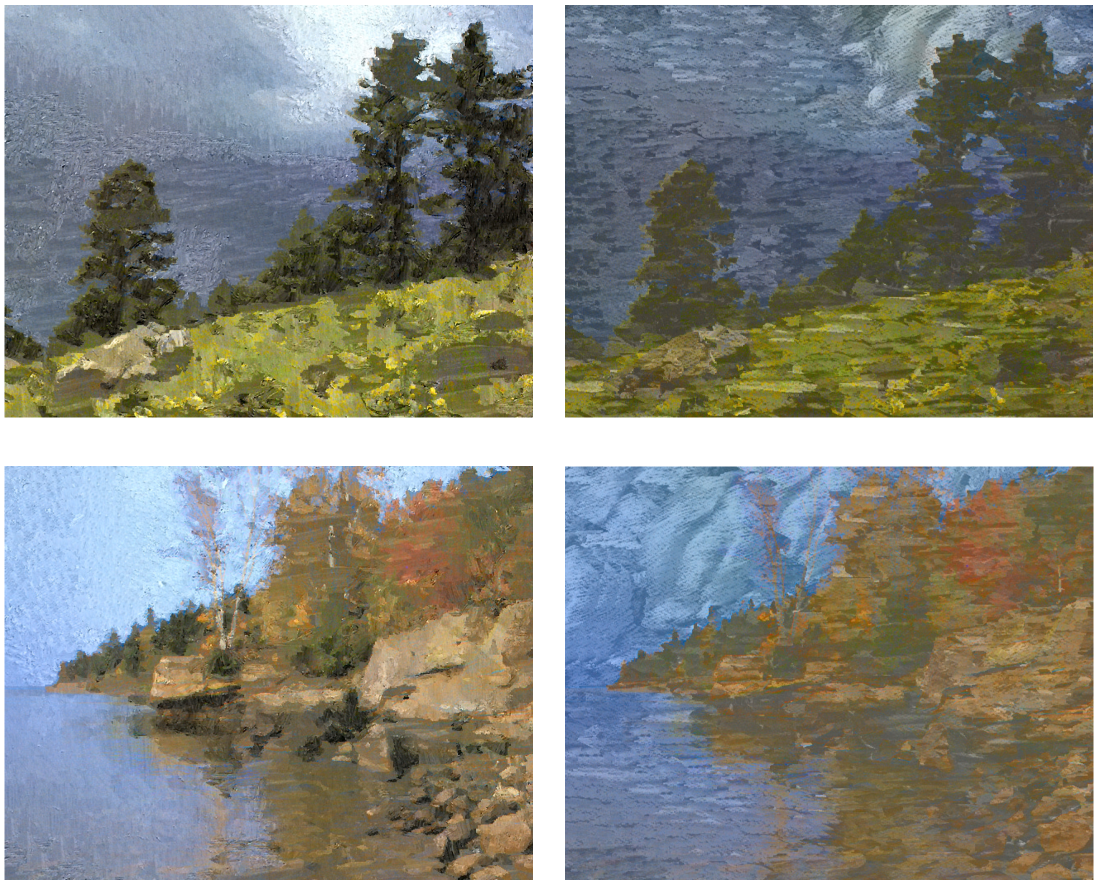


Figure 7.12: Paintings and drawings by example The left images are painted in a style learned from a Lucian Freud painting (Figure 7.11, top row); the right images are drawn in the style of a pastel drawing (Figure 7.11, bottom row).

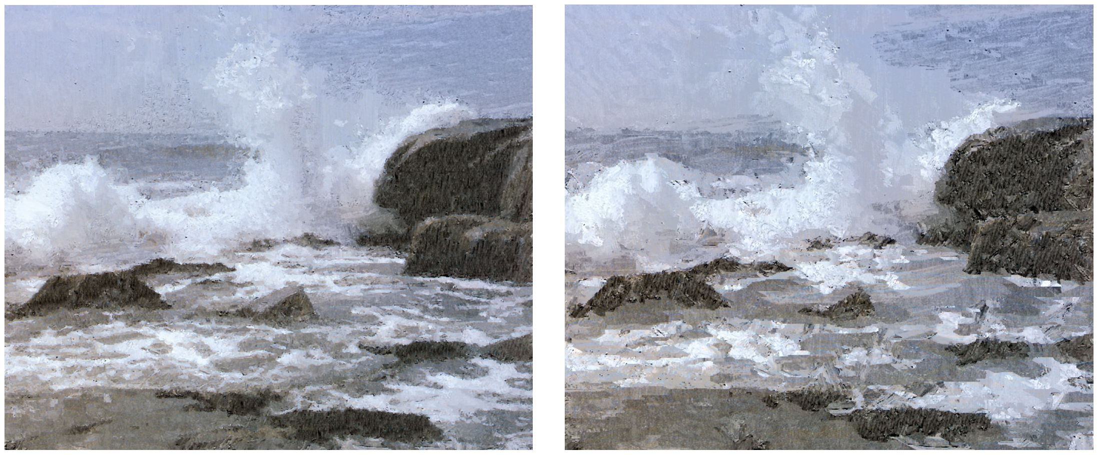


Figure 7.13: Paintings by example, varying the coherence parameter κ. The training from the Lucian Freud self-portrait (upper row of Figure 7.11). The source image is shown in Figure 7.7. In the left image, $\kappa = 5$; in the right, $\kappa = 15$.

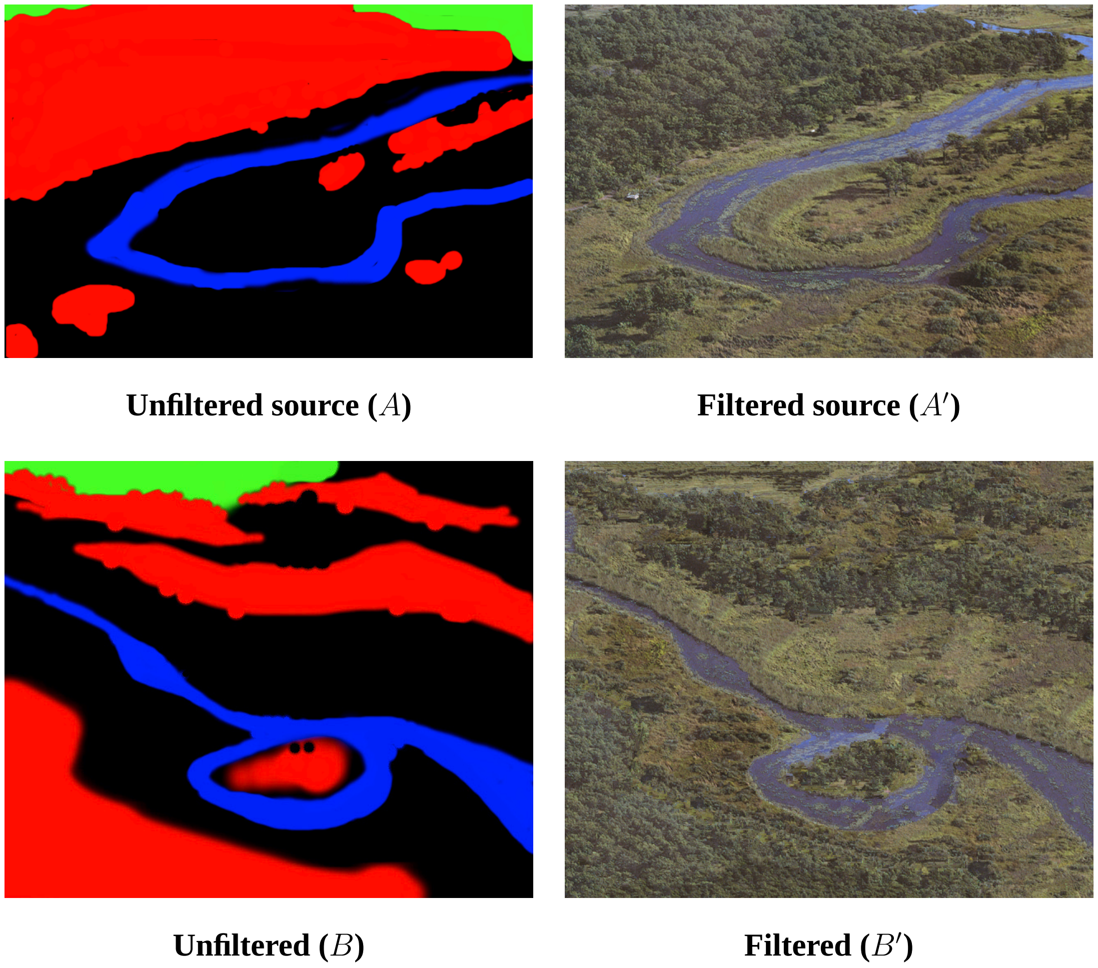
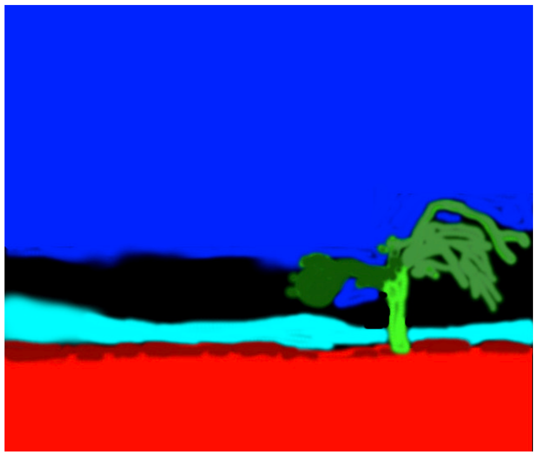
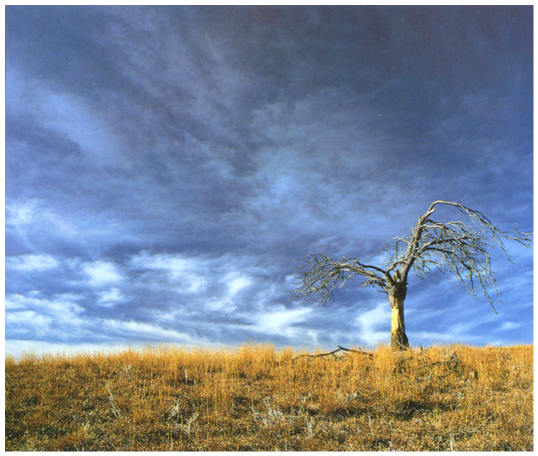


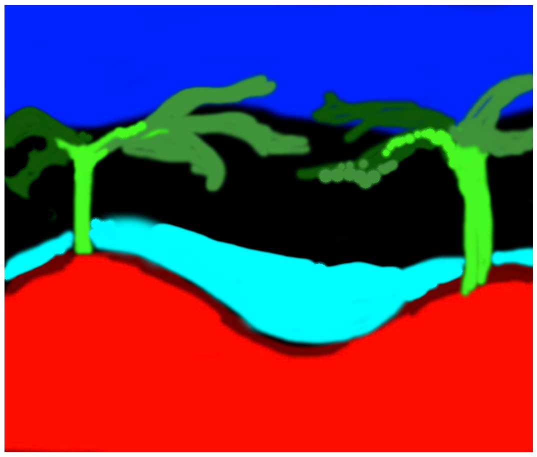
Figure 7.14: Texture-by-numbers. The unfiltered source image (A) was painted by hand to annotate A'. The unfiltered target image (B) was created in a paint program and refined with our interactive editor; the result is shown in B'.



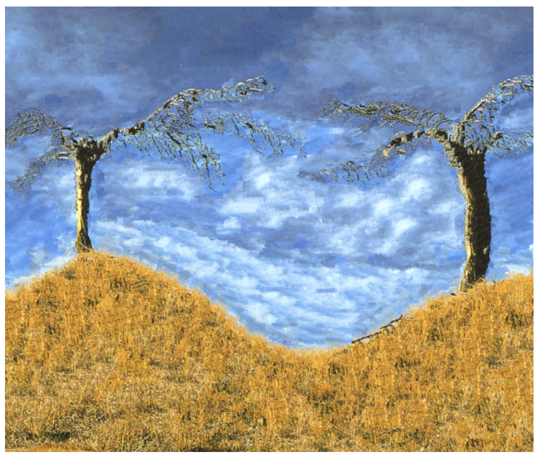
Unfiltered source (A)



Filtered source (A')

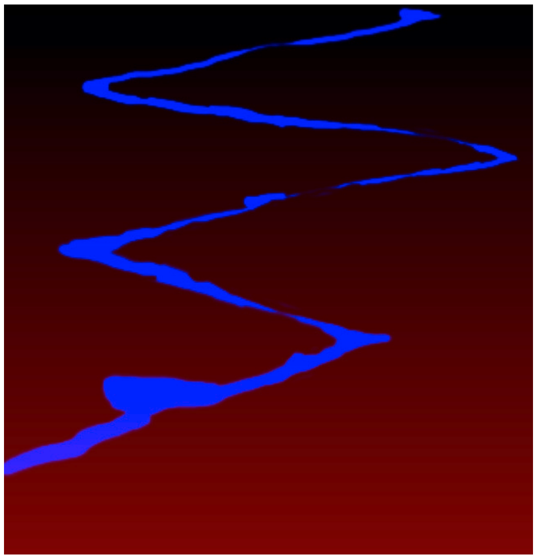


Unfiltered (B)

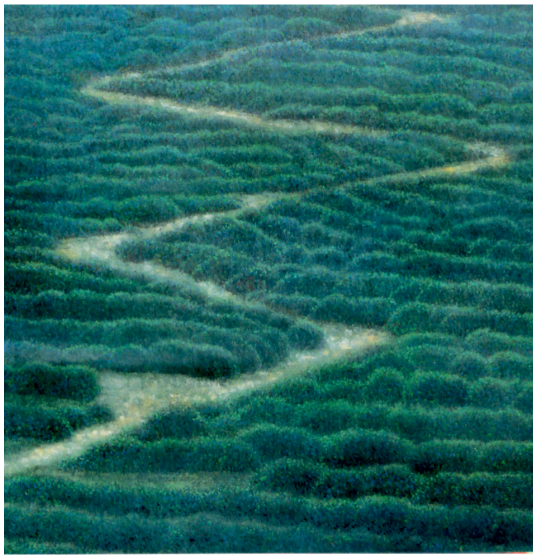


Filtered (B')

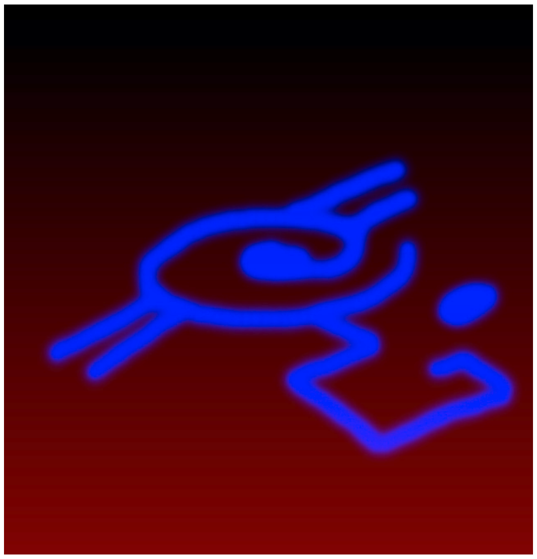
Figure 7.15: Texture-by-numbers. The unfiltered source image (A) was painted by hand to annotate A'. The unfiltered target image (B) was created in a paint program and refined with our interactive editor; the result is shown in B'.



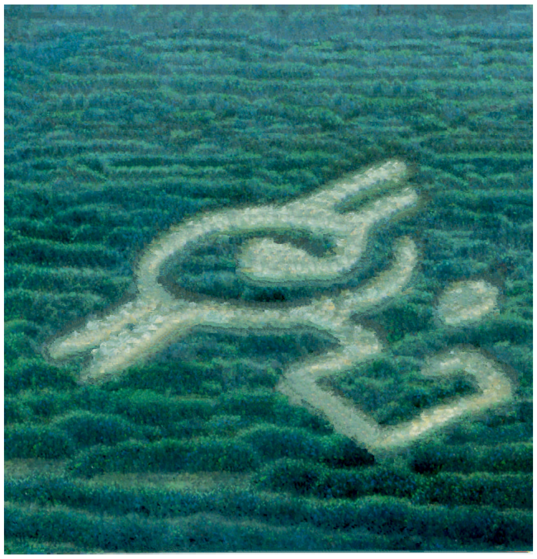
Unfiltered source (A)



Filtered source (A')



Unfiltered (B)



Filtered (B')

Figure 7.16: Crop circles. Ordinary texture synthesis cannot reproduce the terrain shown in the painting because it is not stationary: far elements are different from near elements. The use of the gradient channel in A and B distinguishes near from far, allowing the painting to be used for texture-by-numbers.

Chapter 8

Conclusion

In this thesis, we have described several new approaches to non-photorealistic rendering and rendering. These include stroke-based methods, in which we design strategies for placing brush strokes, and an example-based method, in which we try to capture "style" from examples.

The future of non-photorealistic rendering is wide open, both as a research topic and as an artistic medium. The work in this thesis addresses a small part of the goals articulated in the introduction; much more remains to be done. In the future, we expect the development of tools that: support high-quality rendering in many different styles, model brush strokes explicity or implicitly, automatically capture high-quality styles from example, incorporate an artist's decisions interactively, and render at interactive rates for virtual environments. In the rest of this section, we outline some areas for future work in detail.

8.1 Grand challenges

At its heart — and at the heart of much of computer graphics in general — is the question: how do we make good tools for artists? In much of NPR, there is a chicken-and-egg problem, because it is difficult to design tools for an unknown aesthetic, but difficult to create an aesthetic without having the right tools. Hence, most work involves a continuous dialogue between building technology and exploring aesthetics. Often, the two practices are indistinguishable, since writing code can be a way to explore visual ideas and aesthetics.

Even if we assume that the desired aesthetic is known, determining the appropriate level of automation is quite difficult. In one sense, the goal of NPR work is to create tools that automate just the right parts of a task, i.e. those that can be automated well; some decisions need to be made by the artist. For example, much of this thesis is developed under the assumption that, for some applications, it is not necessary for the artist to individually specify every brush stroke in an image (or still frame). What strategies can we use to divide the work between the user and the system? For those decisions that must be made by an artist, how does one efficiently specify choices, and interactively edit and update them? For example, weight masks (used in this thesis to specify different painting styles for different parts of an image) is one example of painting interfaces used as a strategy along these lines.

The work in this thesis has been motivated by two parallel goals. The first goal is to enable feature-length movies that tell stories using traditional media styles. The second is to enable video games that use NPR techniques to tell stories interactively. In both cases, we hope to enable new art forms that blend traditional techniques with digital media.

8.2 Stroke-Based Rendering

The majority of this work in this thesis attempts to define algorithms that can be used for automatic placement of brush and pen strokes. The algorithms described should all be viewed as preliminary: there is a severe limitation in the actual styles available; it is difficult to control and to modify these styles; and they are not fast enough for real-time interaction.

In the future, we hope to develop new systems that address these problems, by making stroke placement highly controllable. Moreover, we hope to build systems that capture many styles within a compact framework, so that, for example, switching from 2D to 3D processing, or from painting to pen-and-ink, will entail little more than adjusting a few high-level parameters to the same algorithm.

8.3 Image Analogies and Example-Based NPR

There is still much work on example-based rendering be done, both in improving the methods that we have presented in Chapter 7, and in investigating other approaches. In particular, the quality of the synthesis should be improved, perhaps by exploring other regression algorithms. Another promising avenue is the exploration of other applications for these ideas; there are straightforward applications to processing video, 3D models, alpha mattes, environment mattes, curves, and brush strokes, to name a few. As above, the speed of the process and the level of user control should also be improved.

Appendix A

Brush Stroke Rendering

In this chapter, we describe methods we have used for scan-converting and texturing brush strokes. However, methods more sophisticated than these are available (e.g. [HLW93, Fra]).

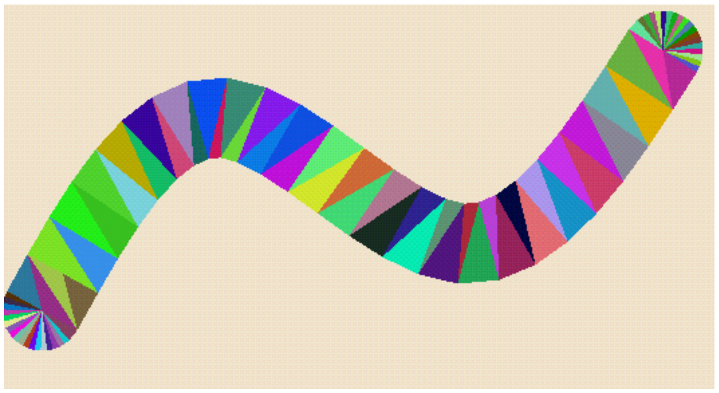
A.1 Brush Stroke model

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 original implementation, used for the figures and video in [Her98], (Chapter 3), was chosen for ease of implementation, and simply swept an arbitrary brush profile along a curve, with some fancy bookkeeping to correctly handle stroke opacity. In the remaining implementation and images, we used the faster method described in the next section.

A.2 Rendering with Triangle Strips

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 thickness (Figure A.1), by specifying a profile curve for the thickness. We do this by assigning a thickness for each

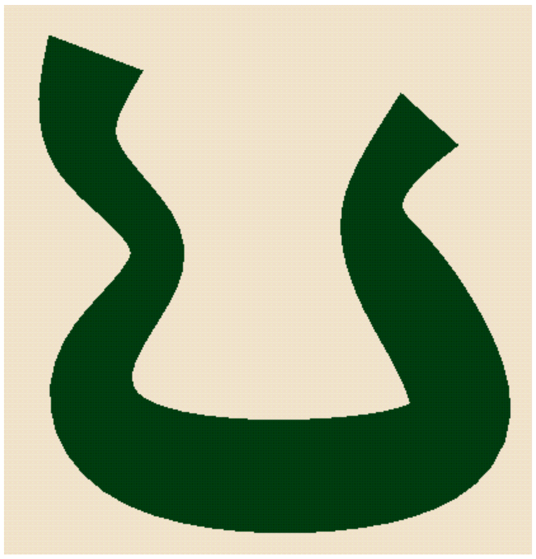


Figure A.1: Variable-thickness brush stroke

control point, and subdividing the thicknesses at the same time as subdiving the control point positions. A curve with random thicknesses at each control point is shown in Figure A.1.

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.

Computation of offset curves is discussed in more detail, in, for example, [Ost93].

A.3 Procedural Brush Textures

Paintings with the texture and appearance of real media can add substantial appeal to a painting. We have explored the use of procedural textures for providing brush stroke textures. Procedural textures are typically faster than paint simulation approaches based on cellular automata [Sma90, CAS$^+$97] and provide finer control over appearance; on the other hand, using procedural textures make it more difficult to achieve the complex, naturalistic appearance produced by simulation. Paper texture and canvas can also be synthesized procedurally [Wor96, Cur99].

The first step in creating stroke textures is to define a parameterization for a stroke [HLW93]. A stroke defines a mapping from (x, y) screen coordinates to (s, t) texture coordinates. The parameter s varies along the spine of the stroke in proportion to the curve's arc length, and the parameter t varies normal to the curve from -1 to 1, as illustrated in Figure A.2. Multiplying t by the brush radius gives a parameterization proportional to distance in image space.

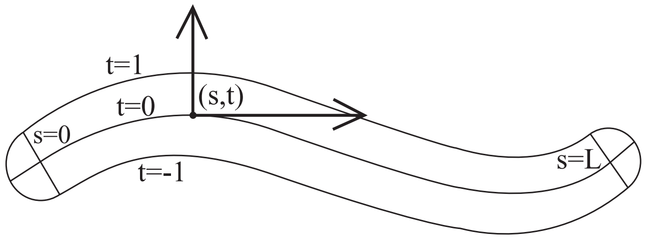


Figure A.2: Parameterization of a brush stroke

We currently employ a simple bristle texture given by $bristle(s,t) = noise(c_1 s, c_2 t)$, where $noise(x, y) \in [-1, 1]$ is a noise function [Per85], and $c \approx 1000$. This function can be used for the stroke opacity, or as a height field for paint lighting; we compute lighting as the dot product between the view direction and the slope of the height field: $I(x, y) = L^T \nabla H(x, y)$, where $H(x, y)$ is the height field at (x, y). The stroke texture itself provides the height field parameterized in texture space: $H(s, t)$. The stroke parameterization also defines a mapping from screen space to texture space as $(s(x, y), t(x, y))$; the derivative of this mapping can be computed during stroke scan-conversion and summarized by the Jacobian matrix $J = \begin{bmatrix} \frac{\partial s}{\partial x} & \frac{\partial t}{\partial x} \\ \frac{\partial s}{\partial y} & \frac{\partial t}{\partial y} \end{bmatrix}$. The intensity can be computed as:

$$H(x, y) \quad = \quad h(s(x, y), t(x, y))$$

$$\begin{aligned}
\nabla H(x,y) &= \begin{bmatrix} \frac{\partial h(s,t)}{\partial x} \\ \frac{\partial h(s,t)}{\partial y} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial h}{\partial s}\frac{\partial s}{\partial x} + \frac{\partial h}{\partial t}\frac{\partial t}{\partial x} \\ \frac{\partial h}{\partial s}\frac{\partial s}{\partial y} + \frac{\partial h}{\partial t}\frac{\partial t}{\partial y} \end{bmatrix} \\
&= J\begin{bmatrix} \frac{\partial h}{\partial s} \\ \frac{\partial h}{\partial t} \end{bmatrix} \\
&= J\nabla h(s,t) \\
I(x,y) &= L^T\nabla H(x,y) = L^T J\nabla h(s,t)
\end{aligned}$$

A soft edge can be added to the stroke by multiplying the opacity by a falloff curve parameterized by t.

Appendix B

Linear Histogram Matching

We now derive the linear histogram matching used in Chapter 7. Our goal is to process the pixel values in an image so that they will have a similar histogram to another image, while preserving the qualitative appearance of the original image. The standard histogram matching operation [Cas96] matches the target histogram exactly, but at the expense of dramatically changing the appearance of the image (e.g. adding or removing edges).

Instead, we use a linear transformation of the source pixel values. We choose the transformation to match the mean and covariance of the target distribution.

The general problem statement is as follows: given a dataset with n vectors $\vec{x}_i$ (i.e. pixel values), we want to find a linear transformation to a new space

$$\vec{y}_i = \beta \vec{x}_i + \vec{\alpha} \tag{B.1}$$

such that the new data points $\vec{y}_i$ have a desired mean and covariance:

$$\begin{aligned} \vec{\mu}_y &= \frac{1}{n} \sum_i \vec{y}_i && \text{(B.2)} \\ \mathbf{K}_y &= \frac{1}{n} \sum_i (\vec{y}_i - \vec{\mu}_y)(\vec{y}_i - \vec{\mu}_y)^T && \text{(B.3)} \end{aligned}$$

For an RGB image, each vector is 3x1, and the matrix β is 3x3. For a luminance image, each value is a scalar, including β. Substituting equation B.1 into B.2, we get:

$$\begin{aligned}
\vec{\mu}_y &= \frac{1}{n}\sum_i \beta\vec{x}_i + \vec{\alpha} \\
&= \beta\frac{1}{n}\sum_i \vec{x}_i + \vec{\alpha} \\
&= \beta\vec{\mu}_x + \vec{\alpha} \\
\vec{\alpha} &= \vec{\mu}_y - \beta\vec{\mu}_x
\end{aligned}$$

where $\vec{\mu}_x$ is the mean of the $\vec{x}$'s. We then substitute into B.3 to get:

$$\begin{aligned}
\mathbf{K}_y &= \frac{1}{n}\sum_i (\beta\vec{x}_i + \vec{\alpha} - (\beta\vec{\mu}_x + \vec{\alpha}))(\beta\vec{x}_i + \vec{\alpha} - (\beta\vec{\mu}_x + \vec{\alpha}))^T \\
&= \frac{1}{n}\sum_i \beta(\vec{x}_i - \vec{\mu}_x)(\vec{x}_i - \vec{\mu}_x)^T\beta^T \\
&= \beta\mathbf{K}_x\beta^T \\
\sqrt{\mathbf{K}_y}\sqrt{\mathbf{K}_y}^T &= (\beta\sqrt{\mathbf{K}_x})(\beta\sqrt{\mathbf{K}_x})^T \\
\beta\sqrt{\mathbf{K}_x} &= \sqrt{\mathbf{K}_y} \\
\beta &= \mathbf{K}_y^{1/2}\mathbf{K}_x^{-1/2}
\end{aligned}$$

Hence, the transform is:

$$\vec{y}_i = \mathbf{K}_y^{1/2}\mathbf{K}_x^{-1/2}(\vec{x}_i - \vec{\mu}_x) + \vec{\mu}_y$$

where $\mathbf{K}_y$ is the covariance of the $\vec{x}$'s, and $\sqrt{}$ is defined so that $\sqrt{A}\sqrt{A}^T = A$.

For scalar image types, this reduces to:

$$y_i = \frac{\sigma_y}{\sigma_x}(x_i - \mu_x) + \mu_y$$

where σ represents variance.

C.1 Matrix Square-Root

To compute $B = \sqrt{A}$, we take the eigenvalue decomposition

$$A = U\Sigma U^T = U\,diag(\lambda_i)U^T$$

and take the square roots of the eigenvalues:

$$B = U \, diag \sqrt{\lambda_i} U^T \tag{B.4}$$

In other words, we just take the square roots of the eigenvalues and recompose the matrix. It can easily be seen that $BB^T = A$. Since we only take the square-roots of covariance matrices, we are guaranteed to have non-negative eigenvalues.

C.2 Uniqueness

The square-root is not unique. For example, if $BB^T = A$, then, for any orthonormal matrix X, $BX(BX)^T = A$. We now sketch an argument for why the matrix square-root used above is a good choice.

An easy case to examine is for scalar pixel values (i.e. in luminance images): in this case, we use $\beta = \sigma_y / \sigma_x$. The other choice of square-root yields $\beta = -\sigma_y / \sigma_x$. This corresponds to rotation by $X = -1$. (Actually, this choice corresponds to a *different* choice of square root for $\sqrt{\sigma_y^2}$ than for $\sqrt{\sigma_x^2}$.) Using $\beta < 0$ reverses the pixel order around the mean; i.e. it inverts the image. This is clearly undesirable for histogram matching: we want the matching function to be monotonic.

Appendix C

Nudge: Freeform Image Deformation and Matching

We now describe *Nudge*, the freeform image editing tool used to register artistic images for Image Analogies training (Chapter 7). As a first step, we used Adobe Photoshop to manually align images with respect to global translation, rotation, and scaling (although this step could be performed automatically or semi-automatically). At this point, local deformations are required to closely align the images (Figure C.3). The deformations are always applied to the A image, in order to not change the textural qualities of the A' image.

Nudge treats the image as viscous: the user "pushes" pixels around, to put them into a desired shape. The behavior is akin to texture-mapping a fluid simulation [Sta99], but without dynamic simulation or animation. A single-image version can be found on the web at http://www.mrl.nyu.edu/~hertzman/nudge.

This alignment is a correspondence problem that appears in image morphing and view interpolation applications [BN92, SD96b], as well as in many other applications. Most freely-available morphing tools do not allow very fine-grain interaction to edit the morph — typically, one tugs at a grid of control points or places line features, which

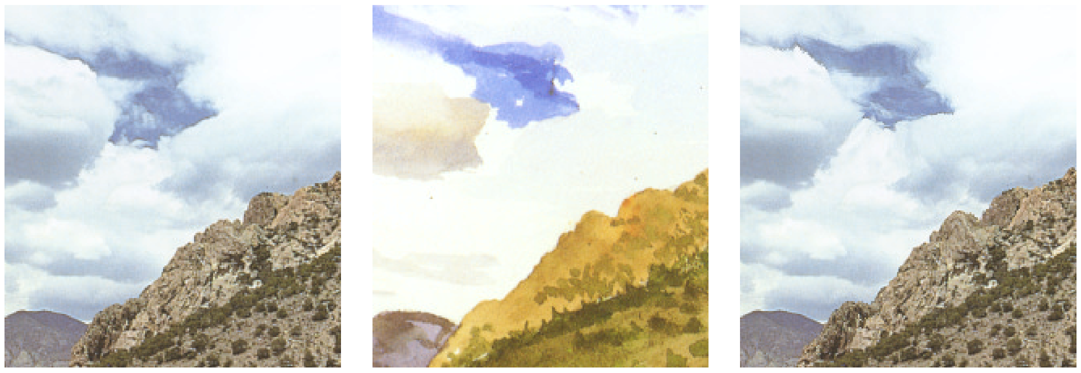


Figure C.3: In order to get images into close correspondence, we interactively deform the input A image in order to match the shape of A'. Note that, after editing, the mountain silhouette in A matches A' much more closely.

does not allow sufficient detail in editing for our purposes.

Nudge uses four images, indexed by pixel locations p:

- The *Source image* $S(p)$ that we wish to distort (usually A).
- The *Target image* $T(p)$ that we wish to match (usually A').
- The *Displacement map* $d(p)$, that contains a vector from a target pixel in the warped image to its source.
- The *Warped image* $W(p)$, defined as

$$W(p) = S(p + d(p))$$

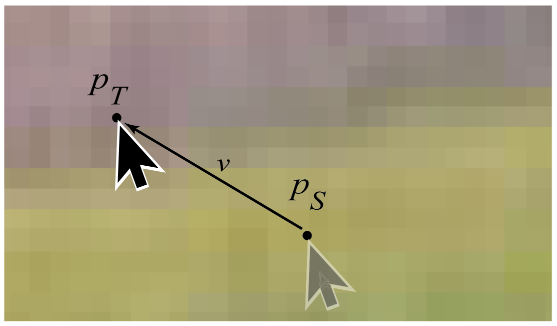
Every pixel in the warped image is directly copied from the target — pixel colors are never blurred. The only exception is that slight blurring will occur if we use bilinear sampling to look up texture values. Our implementation currently uses point sampling for speed.

Keyboard shortcuts allow the user to quickly switch between display modes: Each of the source, target, or warped image may be displayed. For comparison, the warped image may be displayed transparently over the target image in a “light table” mode. The displacement vectors may also be rendered over another image.

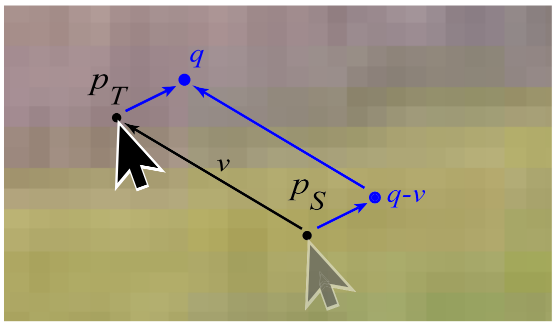
The user modifies the warped image by dragging the mouse (Figure C.4). Each mouse drag even modifies the displacement map, and the warped image is recomputed. In each mouse drag event, we get the current mouse position p_T and the previous mouse position p_S. We then blend the displacement values around p_S into the displacement values around p_T. This gives the effect of “pushing” the warped image pixels in the direction of mouse movement. The blending is parameterized by the brush “strength” f and the brush width w. This blending is performed as follows:

```
function MOUSEDRAG($p_S, p_T$):
    $v \leftarrow p_T - p_S$
    for each pixel $q$ in the neighborhood of $p_T$, do:
        $\alpha \leftarrow f \exp(\|q - p_T\|^2 / w^2)$
        $d(q) \leftarrow (1 - \alpha) * d(q) + \alpha * (d(q - v) - v)$
        $W(q) \leftarrow S(q + d(q))$
```

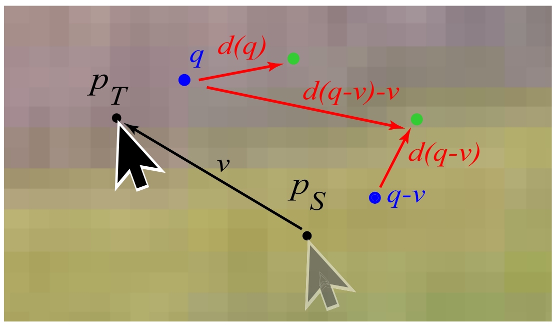
When $\alpha = 0$, q will be unchanged. When $\alpha = 1$, $W(q)$ will have the same color as $W(q - v)$. This is achieved by setting $d(q)$ to the value $d(q - v) - v$, and hence $W(q) = S(q + d(q)) = S((q - v) + d(q - v)) = W(q - v)$. Intermediate values of α interpolate these two behaviors.



(a)



(b)



(c)

Figure C.4: Nudging. (a) Each mouse drag event has a starting point p_S and a target point p_T, connected by the vector $v = p_T - p_S$. (b) Every point q in the neighborhood of p_T is affected by the drag. The point q has a corresponding point $q - v$ near the source point. (c) For the new displacement at q, we blend two values: $d(q)$, the old displacements at q; and the displacement to $q - v$. This latter displacement is $d(q-v)-v$, since it can be expressed as the sum of $-v$ and $d(q - v)$.

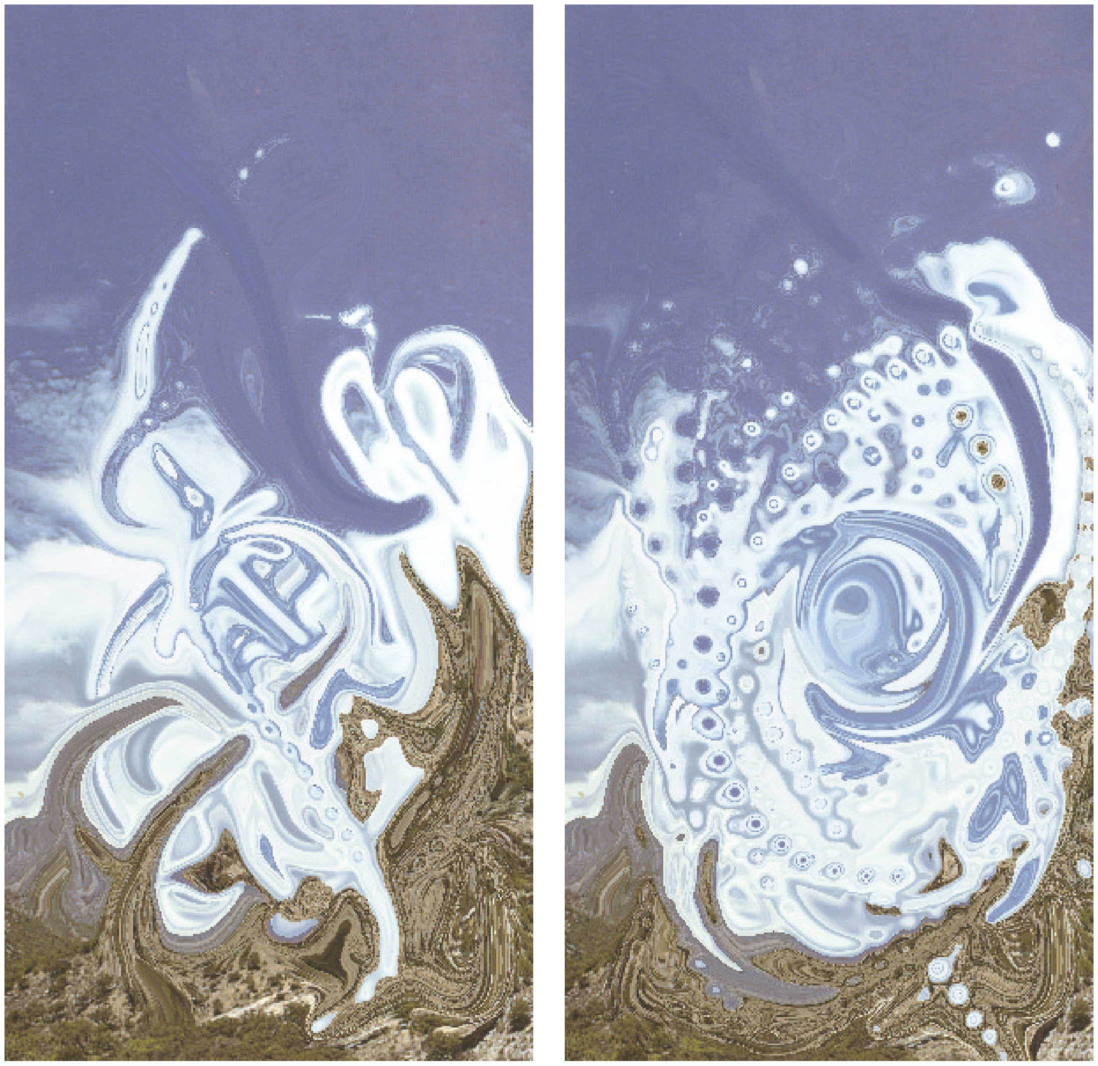


Figure C.5: Whimsical images created with *Nudge*.

Bibliography

[601] 601FX. Martell – The Art of Cognac. In SIGGRAPH Video Review 110: SIGGRAPH 95 Computer Graphics Showcase.

[AMN$^+$98] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions. *Journal of the ACM*, 45(6):891–923, 1998. Source code available from http://www.cs.umd.edu/~mount/ANN.

[App67] Arthur Appel. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. In *Proc. ACM National Conference*, pages 387–393, 1967.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001.

[AWJ90] Amir A. Amini, Terry E. Weymouth, and Ramesh C. Jain. Using Dynamic Programming for Solving Variational Problems in Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):855–867, September 1990.

[AZM00] Maneesh Agrawala, Denis Zorin, and Tamara Munzner. Artistic Multiprojection Rendering. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 125–136, June 2000.

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

[BBS94] Deborah F. Berman, Jason T. Bartell, and David H. Salesin. Multiresolution Painting and Compositing. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 85–90. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[BDG$^+$99] G. Barequet, C.A. Duncan, M.T. Goodrich, S. Kumar, and M. Pop. Efficient perspective-accurate silhouette computation. pages 417–418, June 1999.

[BE99] Fabien Benichou and Gershon Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. *Pacific Graphics '99*, October 1999. Held in Seoul, Korea.

[BH98] David J. Bremer and John F. Hughes. Rapid Approximate Silhouette Rendering of Implicit Surfaces. In *Proc. The Third International Workshop on Implicit Surfaces*, June 1998.

[BHA93] Michael F. Barnsley, Lyman P. Hurd, and Louisa F. Anson. *Fractal Image Compression*. A.K. Peters Ltd, 1993.

[BN92] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):35–42, July 1992.

[Bon97] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. *Proceedings of SIGGRAPH 97*, pages 361–368, August 1997.

[Car84] Loren Carpenter. The A-buffer, an Antialiased Hidden Surface Method. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):103–108, July 1984.

[Cas96] Kenneth Castleman. *Digital Image Processing*. Prentice-Hall, 1996.

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

[CG97] Richard Coutts and Donald P. Greenberg. Rendering with Streamlines. In *SIGGRAPH 97: Visual Proceedings*, page 188, 1997.

[CGG$^+$99] Cassidy Curtis, Amy Gooch, Bruce Gooch, Stuart Green, Aaron Hertzmann, Peter Litwinowicz, David Salesin, and Simon Schofield. *Non-Photorealistic Rendering*. SIGGRAPH 99 Course Notes, 1999.

[CHZ00] Jonathan M. Cohen, John F. Hughes, and Robert C. Zeleznik. Harold: A world made of drawings. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 83–90, June 2000.

[CJTF98a] Wagner Toledo Corrêa, Robert J. Jensen, Craig E. Thayer, and Adam Finkelstein. Texture Mapping for Cel Animation. In *SIGGRAPH 98 Conference Proceedings*, pages 435–446, July 1998.

[CJTF98b] Wagner Toledo Corrêa, Robert J. Jensen, Craig E. Thayer, and Adam Finkelstein. Texture mapping for cel animation. *Proceedings of SIGGRAPH 98*, pages 435–446, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[CL93] Brian Cabral and Leith (Casey) Leedom. Imaging Vector Fields Using Line Integral Convolution. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 263–272, August 1993.

[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.

[Cur99] Cassidy Curtis. Non-photorealistic animation. In Stuart Green, editor, *Non-Photorealistic Rendering*, SIGGRAPH Course Notes, chapter 6. 1999.

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

[Dec96] Philippe Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Technical Report 2919, INRIA, June 1996.

[DHR$^+$99] Oliver Deussen, Jörg Hamel, Andreas Raab, Stefan Schlechtweg, and Thomas Strothotte. An Illustration Technique Using Hardware-Based Intersections. *Graphics Interface '99*, pages 175–182, June 1999.

[DS00] Oliver Deussen and Thomas Strothotte. Computer-generated pen-and-ink illustration of trees. *Proceedings of SIGGRAPH 2000*, pages 13–18, July 2000. ISBN 1-58113-208-5.

[EC90] Gershon Elber and Elaine Cohen. Hidden Curve Removal for Free Form Surfaces. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 95–104, August 1990.

[EDD^{+}95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics Proceedings*, Annual Conference Series, pages 173–182. ACM Siggraph, 1995.

[Elb95] Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, September 1995.

[Elb98] Gershon Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January – March 1998.

[Elb99] Gershon Elber. Interactive line art rendering of freeform surfaces. *Computer Graphics Forum*, 18(3):1–12, September 1999.

[Eli] Chris Eliasmith. Dictionary of Philosophy of Mind. http://artsci.wustl.edu/$\sim$philos/MindDict/.

[ER00] David Ebert and Penny Rheingans. Volume Illustration: Non-Photorealistic Rendering of Volume Models. *Proc. IEEE Visualization '00*, October 2000.

[FBC^{+}95] Jean-Daniel Fekete, Érick Bizouarn, Éric Cournarie, Thierry Galas, and Frédéric Taillefer. TicTacToon: A Paperless System for Professional 2-D animation. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 79–90. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[FPC] William T. Freeman, Egon C. Pasztor, and Owen T. Carmichael. Learning Low-Level Vision. *International Journal of Computer Vision*. To appear. MERL Technical Report TR2000-05a.

[Fra] Fractal Design. Painter. Software Package.

[Fra87] George K. Francis. *A Topological Picturebook*. Springer-Verlag, New York, 1987.

[Fra91] George. K. Francis. The etruscan venus. In P. Concus, R. Finn, and D. A. Hoffman, editors, *Geometric Analysis and Computer Graphics*, pages 67–77. 1991.

[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.

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

[GGSC98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 447–452. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[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. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 43–52, June 2000.

[Goo98] Amy Gooch. Interactive Non-Photorealistic Technical Illustration. Master's thesis, University of Utah, December 1998.

[Gou97] Roger L. Gould. "Hercules:" The 30-Headed Hydra. In *SIGGRAPH 97: Visual Proceedings*, page 213, 1997.

[GSG$^+$99] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[Guo95] Qinglian Guo. Generating Realistic Calligraphy Words. *IEEE Trans. Fundamentals*, E78-A(11):1556–1558, November 1995.

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

[HB95] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. *Proceedings of SIGGRAPH 95*, pages 229–238, August 1995.

[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. *Proc. Computer Graphics International 2001*, 2001. To appear.

[HH90] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[HJO$^+$01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image Analogies. *Proceedings of SIGGRAPH 2001*, August 2001. To appear.

[HL94] Siu Chi Hsu and Irene H. H. Lee. Drawing and Animation Using Skeletal Strokes. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 109–118. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[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. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 7–12, June 2000.

[HS99] J. Hamel and T. Strothotte. Capturing and re-using rendition styles for non-photorealistic rendering. *Computer Graphics Forum*, 18(3):173–182, September 1999.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000. ISBN 1-58113-208-5.

[IMT99] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *Proceedings of SIGGRAPH 99*, pages 409–416, August 1999.

[Int97] Victoria L. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *SIGGRAPH 97 Conference Proceedings*, pages 109–116, August 1997.

[Jam90] William James. *The Principles of Psychology*. 1890.

[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.

[KGC00] Matthew Kaplan, Bruce Gooch, and Elaine Cohen. Interactive artistic rendering. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 67–74, June 2000.

[KLK$^+$00] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Correa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000*, pages 527–534, July 2000. ISBN 1-58113-208-5.

[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.

[KW97] Lutz Kettner and Emo Welzl. Contour Edge Analysis for Polyhedron Projections. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 379–394. Springer Verlag, 1997.

[KWT87] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active Contour Models. *International Journal of Computer Vision*, 1(4), 1987.

[Lev98] Jonathan Levene. A Framework for Non-Realistic Projections. Master's thesis, MIT, 1998.

[Lim] Informatix Software International Limited. Piranesi.

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

[Lit99] Peter Litwinowicz. Image-Based Rendering and Non-Photorealistic Rendering. In Stuart Green, editor, *Non-Photorealistic Rendering*, SIGGRAPH Course Notes. 1999.

[LMHB00] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 13–20, June 2000.

[LS95] John Lansdown and Simon Schofield. Expressive Rendering: A Review of Nonphotorealistic Techniques. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

[MB95] Eric N. Mortensen and William A. Barrett. Intelligent Scissors for Image Composition. *Proceedings of SIGGRAPH 95*, pages 191–198, August 1995.

[MB98] Jon Meyer and Benjamin B. Bederson. Does a sketchy appearance influence drawing behavior? Technical Report HCIL-98-12, Computer Science Department, University of Maryland, College Park, 1998.

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

[MGT00] D. Martín, S. García, and J. C. Torres. Observer dependent deformations in illustration. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 75–82, June 2000.

[Mic] Microsoft Corporation. Impressionist. Software Package.

[MIT] MIT Aesthetics and Computation Group. the aesthetics and computation group. http://acg.media.mit.edu/.

[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.

[ML98] Pascal Mamassian and Michael S. Landy. Observer biases in the 3D interpretation of line drawings. *Vision Research*, (38):2817—2832, 1998.

[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. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 59–66, June 2000.

[MOiT98] Shinji Mizuno, Minoru Okada, and Jun ichiro Toriwaki. Virtual sculpting and virtual woodcut printing. *The Visual Computer*, 14(2):39–51, 1998. ISSN 0178-2789.

[Nik75] Kimon Nikolaïdes. *The Natural Way to Draw*. Houghton Miffin, Boston, 1975.

[NM00] J. D. Northrup and Lee Markosian. Artistic silhouettes: A hybrid approach. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 31–38, June 2000.

[Ost93] Victor Ostromoukhov. Hermite approximation for offset curve computation. In S.P. Mudur and S.N. Pattanaik, editors, *IFIP Transactions B: Graphics, Design and Visualisation*, pages 189–196, 1993.

[Ost99] Victor Ostromoukhov. Digital facial engraving. *Proceedings of SIGGRAPH 99*, pages 417–424, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[Ped96] Hans Køhling Pedersen. A framework for interactive texturing operations on curved surfa ces. *Proceedings of SIGGRAPH 96*, pages 295–302, August 1996.

[Per85] Ken Perlin. An Image Synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985.

[PFWF00] Lena Petrovic, Brian Fujito, Lance Williams, and Adam Finkelstein. Shadows for cel animation. *Proceedings of SIGGRAPH 2000*, pages 511–516, July 2000. ISBN 1-58113-208-5.

[Pha91] Binh Pham. Expressive Brush Strokes. *CVGIP*, 53(1):1–6, January 1991.

[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.

[PV92] Ken Perlin and Luiz Velho. A wavelet representation for unbounded resolution painting. Technical report, New York University, New York, 1992.

[PV95] Ken Perlin and Luiz Velho. Live Paint: Painting With Procedural Multiscale Textures. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 153–160. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[Rad99] Paul Rademacher. View-dependent geometry. *Proceedings of SIGGRAPH 99*, pages 439–446, August 1999.

[RC99] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, April 1999.

[Rey] Craig Reynolds. Stylized Depiction in Computer Graphics: Non-Photorealistic, Painterly and ’Toon Rendering. http://www.red3d.com/cwr/npr/.

[RK00] Christian Rössl and Leif Kobbelt. Line-Art Rendering of 3D-Models. 2000.

[Rob97] Barbara Robertson. Different Strokes. *Computer Graphics World,* December 1997.

[Rob98] Barbara Robertson. Brushing Up on 3D Paint. *Computer Graphics World,* April 1998.

[SABS94] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen–And–Ink Illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH ’94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[SAH91] Eero P. Simoncelli, Edward H. Adelson, and David J. Heeger. Probability Distributions of Optical Flow. In *Proc. IEEE Conference of Computer Vision and Pattern Recognition*, June 1991.

[SB00] Mario Costa Sousa and John W. Buchanan. Observational models of graphite pencil materials. *Computer Graphics Forum*, 19(1):27–49, March 2000.

[SD96a] K. Schunn and K. Dunbar. Priming, Analogy and Awareness in complex reasoning. *Memory and Cognition*, 24, 271-284, 1996.

[SD96b] Steven M. Seitz and Charles R. Dyer. View Morphing: Synthesizing 3D Metamorphoses Using Image Transforms. *Proceedings of SIGGRAPH 96*, pages 21–30, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[SF95] Eero P. Simoncelli and William T. Freeman. The steerable pyramid: A flexible architecture for multi-scale derivative computation. *Proc. 2nd Int'l Conf on Image Processing*, October 1995.

[SGG$^+$00] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. *Proceedings of SIGGRAPH 2000*, pages 327–334, July 2000. ISBN 1-58113-208-5.

[SL00] Scott Sona Snibbe and Golan Levin. Interactive dynamic abstraction. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 21–30, June 2000.

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

[Smi82] Alvy Ray Smith. Paint. In Beatty and Booth, editors, *IEEE Tutorial on Computer Graphics*, pages 501–515. IEEE Computer Society Press, second edition, 1982.

[Smi97] Alvy Ray Smith. Digital Paint Systems Historical Overview. Microsoft Corporation, May 1997.

[SS93] S. Allyn Schaeffer and John Shaw. *The Big Book of Painting Nature in Pastel*. Watson-Guptill Publications, 1993.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 197–206, August 1990.

[Sta99] Jos Stam. Stable fluids. *Proceedings of SIGGRAPH 99*, pages 121–128, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[Ste86] Kent A. Stevens. Inferring shape from contours across surfaces. In Alex P. Pentland, editor, *From Pixels to Predicates*, pages 93–110. 1986.

[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.

[Sys] Adobe Systems. Adobe Photoshop.

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

[TC97] S. M. F. Treavett and M. Chen. Statistical Techniques for the Automated Synthesis of Non-Photorealistic Images. In *Proc. 15th Eurographics UK Conference*, March 1997.

[Tee98] Daniel Teece. 3D Painting for Non-Photorealistic Rendering. In *SIGGRAPH 98: Conference Abstracts and Applications*, page 248, 1998.

[Too] Xaos Tools. Paint Alchemy.

[VB99] O. Veryovka and J. Buchanan. Comprehensive halftoning of 3d scenes. *Computer Graphics Forum*, 18(3):13–22, September 1999. ISSN 1067-7055.

[Wan95] B. Wandell. *Foundations of Vision*. Sinauer Associates Inc., 1995.

[Wei66] Ruth E. Weiss. BE VISION, a Package of IBM 7090 FORTRAN Programs to Drive Views of Combinations of Plane and Quadric Surfaces. *Journal of the ACM*, 13(4):194–204, April 1966.

[WFH$^+$97] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective panoramas for cel animation. *Proceedings of SIGGRAPH 97*, pages 243–250, August 1997.

[WL00] Li-Yi Wei and Marc Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.

[Wor96] Steven P. Worley. A Cellular Texture Basis Function. *Proceedings of SIGGRAPH 96*, pages 291–294, August 1996.

[WS94] Georges Winkenbach and David H. Salesin. Computer–Generated Pen–And–Ink Illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH ’94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

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

[ZBLN97] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Software*, 23(4):550–560, 1997.

[ZHH96] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In *SIGGRAPH 96 Conference Proceedings*, pages 163–170, August 1996.

[ZM98] Song Chun Zhu and David Mumford. GRADE: Gibbs Reaction And Diffusion Equations — a framework for pattern synthesis, image denoising, and removing clutter. In *Proc. ICCV 98*, 1998.

[ZWM98] Song Chun Zhu, Ying Nian Wu, and David Mumford. Filters, Random fields, And Maximum Entropy: Towards a Unified Theory for Texture Modeling. *International Journal of Computer Vision*, 12(2):1–20, March/April 1998.