EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Estimating Color and Texture Parameters for Vector Graphics

S. Jeschke[1] and D. Cline[2] and P. Wonka[3]

[1] Vienna University of Technology [2] Oklahoma State University [3] Arizona State University

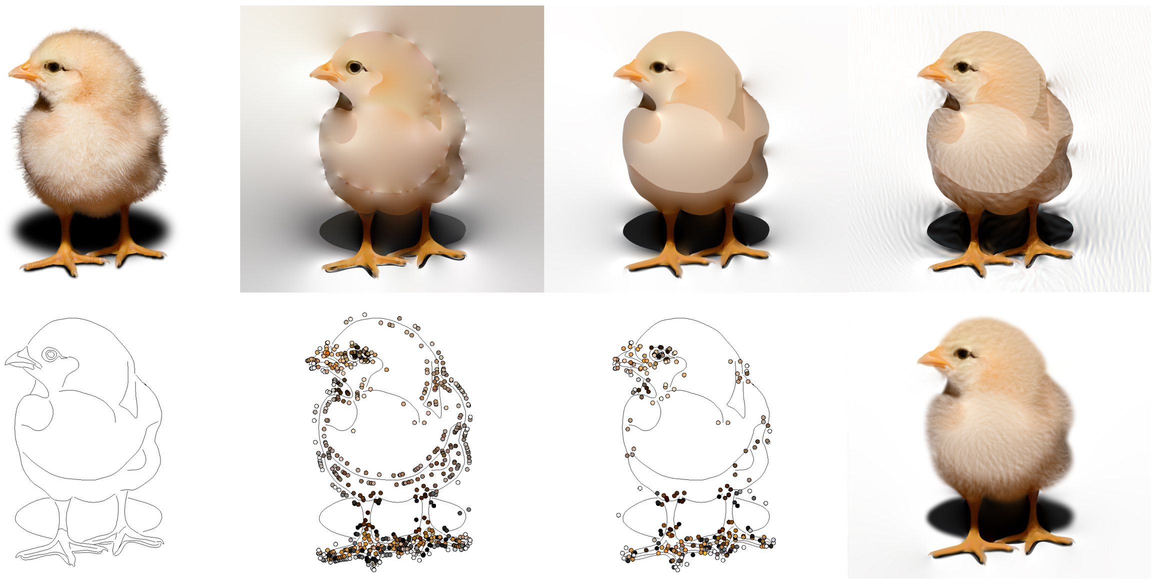

Figure 1: *Top row from left to right: original image, coloring result by [OBW*08], new coloring result, automatic noise fitting result. Bottom row: input curves, the 665 color points by [OBW*08], the 283 color points by the new algorithm, final result after manual editing.*

Abstract

Diffusion curves *are a powerful vector graphic representation that stores an image as a set of 2D Bezier curves with colors defined on either side. These colors are diffused over the image plane, resulting in smooth color regions as well as sharp boundaries. In this paper, we introduce a new automatic diffusion curve coloring algorithm. We start by defining a geometric heuristic for the maximum density of color control points along the image curves. Following this, we present a new algorithm to set the colors of these points so that the resulting diffused image is as close as possible to a source image in a least squares sense. We compare our coloring solution to the existing one which fails for textured regions, small features, and inaccurately placed curves. The second contribution of the paper is to extend the diffusion curve representation to include texture details based on Gabor noise. Like the curves themselves, the defined texture is resolution independent, and represented compactly. We define methods to automatically make an initial guess for the noise texure, and we provide intuitive manual controls to edit the parameters of the Gabor noise. Finally, we show that the diffusion curve representation itself extends to storing any number of attributes in an image, and we demonstrate this functionality with image stippling an hatching applications.*

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Display algorithms—
Three Dimensional Graphics and Realism [I.3.7]: Color, shading, shadowing and texture—

1. Introduction

Diffusion Curve Images (DCI) are a powerful representation for vector graphics containing complex color gradients. A DCI represents an image as a set of Bezier curves with colors defined on either side. The colors diffuse over the image plane resolving to smooth colored regions with sharp, well defined borders along the curves. Curve colors are modeled along both sides of the curves using *color control points*. The intuitive and compact nature of diffusion curves make them a versatile tool for image encoding and manipulation.

In the original work introducing diffusion curves, Orzan et al. [OBW*08] present a number of methods to semi- and fully automatically create Bezier curves and associated colors for a source image. In our experiments we noticed that this initial solution works well for simpler images, but the method for automatically assigning the colors shows severe problems in more difficult cases (see Fig. 1). In particular, the method fails when there are small image features near the curves, when the curves are mis-aligned with image edges by more than a few pixels, and when the image has substantial texture.

A main contribution of the paper is to define a method to robustly solve the diffusion curve coloring problem. That is, given an image and a set of Bezier curves (manually drawn or automatically derived), automatically assign color values to each curve so that the resulting diffused image closely matches the original one. Our method is based on a least squares formulation that finds the "best" colors and naturally handles all of the problematic cases listed above. We also show how to optimize the least squares setup by capturing the influence of each color control point on each image pixel in a single rendering step.

The second contribution of the paper extends the diffusion curve specification to include procedural texture. This is important, because while many natural images and paintings contain interesting and varied textural information, the original diffusion curve representation only provides one distinct look: image edges with very smooth regions in between. Our technique adds procedural texture based on Gabor noise [LLDD09] that mimics the character of a source image texture, rather than trying to exactly reproduce it. Because we leverage the DCI idea of curves with attributes, our textural representation is quite compact. The Gabor noise parameters are defined only for a few points in the image (the color points), and are diffused over the image exactly as image colors. The Gabor noise parameters can be locally estimated in the input image, fitted to the curves just like DCI color information and then used to steer the noise synthesis. The resulting DCIs resemble textured input images more closely than common DCIs.

Besides colors and noise parameters, diffusion curves can be used for any number of attributes that can be represented by a diffusion process over the image plane. Based on this idea, we create stippling and hatching applications that use diffusion curves to represent the stipple density, hatch density and hatch direction. Once the diffusion has taken place, we employ recursive Wang tiles [KCODL06] to place the stipples and hatch strokes.

The main contributions of this paper are to:

- Formulate the problem of assigning colors to diffusion curves so that the resulting image matches a given one as closely as possible in a least squares sense.
- Present an optimized setup for the least squares solution that avoids excessive memory requirements and computations.
- Propose Gabor noise as a means to add textural information to DCIs.
- Automatically derive Gabor noise parameters to match an input image, and allow efficient editing of DCI parameters.
- Demonstrate the DCI representation in other contexts such as NPR stippling and hatching.

2. Related Work

Most vector graphics systems in use today focus on outlines, providing only a few "fill options" for drawing primitives. The individual representations vary mainly by the complexity of the involved primitives, starting from triangular patches [LL06], moving to higher order parametric meshes [PB06], gradient meshes [SLWS07], and diffusion curves [OBW*08] to name a few. However, the required number of primitives for representing an input image is usually directly related to the image frequencies. This makes it hard to efficiently represent and conveniently edit high-frequency image features like textures or noise. A few researchers have attempted to superimpose edge information on a raster image to form a hybrid of vector and raster images. Edge-aware textures [PZ08] are a good example of this. While a successful compromise for rendering, such hybrid methods lose the compact storage and semantic qualities of other vector graphics formats.

Diffusion curves. The *Diffusion Curve Image* (DCI) concept introduced by Orzan et al. [OBW*08] extends the work of Elder et al. [EG01] and expands the expressive range of vector graphics by defining image color as a diffusion process. A DCI is made up of a set of Bezier curves with colors specified on either side. These colors diffuse over the image plane without crossing the curves, creating smooth color gradients contained by sharp, well-defined borders. The colors on the curves are specified at a number of points called *color control points* (or, more simply, *color points*), which are separate from the spatial control points of the curves. Colors between these points along the curves are found by linear interpolation. The diffusion process itself can be cast as a solution to a Laplacian equation with Dirichlet boundary conditions (e.g. fixed colors on the curves and a zero Laplacian everywhere else).

Despite the seeming straightforward setup of the problem, robustly solving the Laplacian equation when the Dirichlet boundaries are not pixel aligned has proven problematic. Jeschke et al. [JCW09a] define a *variable stencil size* Laplacian solver that works well for the diffusion curve problem, and fixes many of the problems associated with earlier Multigrid solutions. A standard Jacobi solver iterates towards the solution of the Laplacian equation in 2D by replacing each pixel $I_{x,y}$ by the average of its 4 neighbors:

$$I_{x,y} = (I_{x+1,y} + I_{x-1,y} + I_{x,y+1} + I_{x,y-1})/4$$

The convergence of the Jacobi solver on this problem is slow, however. A variable stencil size solver improves the convergence by expanding the neighborhood to radius r, so that the update becomes:

$$I_{x,y} = (I_{x+r,y} + I_{x-r,y} + I_{x,y+r} + I_{x,y-r})/4$$

Initially the r value used at each pixel is set to the distance to the nearest fixed point (nearest point on a curve), and this slowly decreases over successive iterations.

Another important question is how to automatically specify the position and color of the color points when trying to reproduce a source image. We dub this problem the *diffusion curve coloring problem*. Orzan et al. [OBW*08] solve this problem as follows: for each Bezier curve they sample the image colors densely on both sides, 3 pixels away from the curve. Color outliers are eliminated, and then unneeded samples are removed using the well known Douglas-Puecker polyline simplification algorithm [DP73], with a threshold of 30 units in L*a*b* space. The resulting colors and parametric locations define the color points, along the curve. Because the colors are assigned using only local information, this method often produces poor results. For example, when attempting to color a textured region, color outliers will be common, leading to numerous color changes along a curve. Visually, this problem manifests itself as a "wrinkled" or "pinched" look near the curves (see Fig. 1). In addition, since colors are only sampled at a fixed distance to a curve, inaccurate curve placement and small features (i.e., smaller than three pixels) result in wrong colors on the curve sides. Since colors are diffused over the whole image, this can lead to large wrongly-colored regions in the final diffused image.

In addition to rendering refinements, several functional extensions to DCIs have been proposed. One of these is a feature embedding algorithm that retains crisp, antialiased curve edges when DCIs are used as texture maps [JCW09b]. DCIs have also been used to model heightfields [JCW09b] and very recently with noise driven by diffused parameters [HGA*10]. Another extension adds controls for the diffusion process itself, allowing the algorithm to specify how colors spread out from the curves across the image plane [BEDT10]. Winnemöller et al. [WOBT09] present a set of tools to design (near) regular textures from diffusion curves and realistically drape them onto objects in images.

Gabor noise. In this paper, we argue that much of the character of a source image texture can be captured using spatially varying noise of some kind. Lagae et al. [LLC*10] provide a comprehensive state-of-the-art survey of procedural noise functions. In this paper, we extend the DCI format to include texture using Gabor noise, which was recently introduced by Lagae et al. [LLDD09] and extented to spatially varying noise [LLD10]. Essentially, Gabor noise works by splatting a set of Gabor kernels onto texture space. The kernel size, direction and its variation define the noise and its spectral properties. Gabor noise has a number of desirable characteristics: It is easily controllable with a few parameters, it is naturally anisotropic, and can be rendered quickly. This has recently been demonstrated in the context of non photorealistic rendering [BLV*10].

Another important quality of Gabor noise is that its parameters can be estimated *locally* in an input image. Gabor space analysis has been extensively studied in the computer vision community, mostly for texture discrimination [FS89, BCG90, MM96], but also for texture synthesis [PZ89]. More recently, Gabor space analysis for texture synthesis has again gained interest in the computer graphics community [LVLD10, GDS10].

3. Robust Diffusion Curve Colorization

Given an image and a set of Bezier curves for a DCI, our goal is to define the color points and the corresponding color values. Keeping the number of colors reasonably low while still faithfully reproducing the image is essential for efficiently storing and editing diffusion curves. Thus, we provide both geometric and color based mechanisms to reduce the number of color points. The overall workflow for our algorithm is as follows:

1. Generate a dense set of color points.
2. Sparsify these points based on a geometric constraint.
3. Solve for the best fit color values.
4. Remove unneeded points based on a color error estimate.

The last two steps are alternated until convergence as will be described in the following subsections.

3.1. Color Point Placement

Appropriate spacing for color points. One approach to spacing color points along the curves would be to attempt to fit the image colors as closely as possible. This seems like a worthwhile goal, but in practice too much color fidelity along the curves often leads to unpleasing high frequency artifacts. The "color wrinkles" seen in the densely-spaced reconstructions in Fig. 2 are a good example of such color overfitting. This observation leads us to a purely geometric heuristic for the initial spacing of color points, namely that the density of the color points usually should not greatly exceed the curve spacing, i.e., the distance to the neighboring curve. This is also approximately the highest frequency that

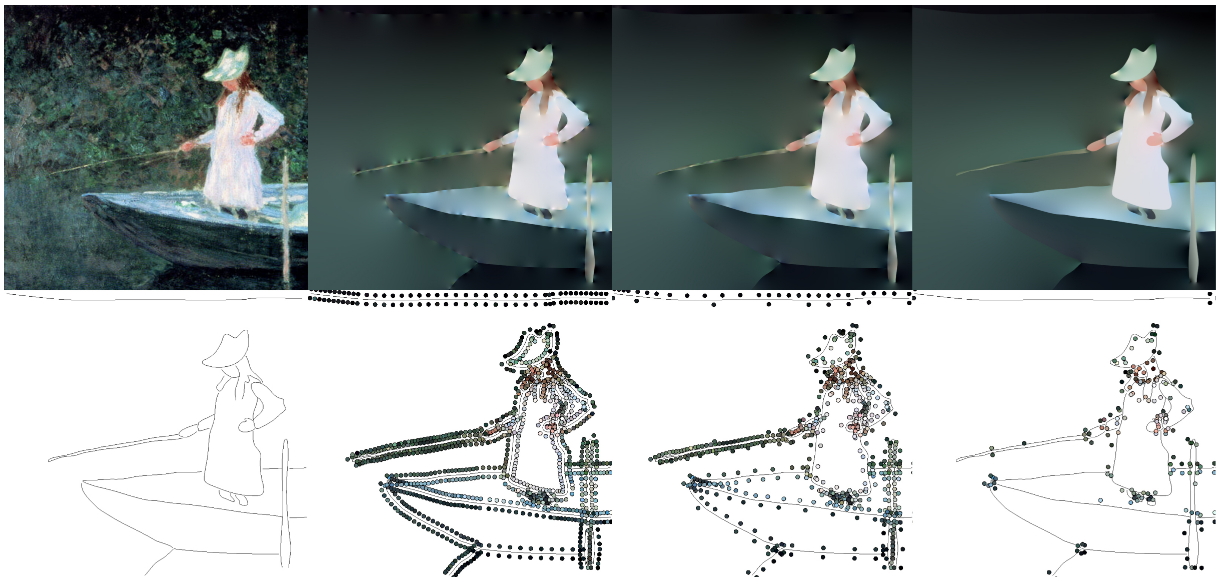

Figure 2: *Color points derivation. Top row (from left to right): original image, 1130 dense color points, 675 sparse color points, 176 simplified color points. The bottom row shows the curves and color points according to the image above each.*

can be faithfully reproduced away from the curves. Finer color changes along the curves should be abstracted away, as they cannot be satisfactorily reproduced. Of course, the user can manually add back color points at arbitrary curve locations to bring out particular details.

Geometry-based placement. In order to meet the above criterion, we need to compute the curve spacing (distance to the closest neighbor curve) along each curve. Based on these distances, the color points can then be placed. In practice our algorithm starts with a dense set of color points, and then eliminates points until the proper density is reached. Initially color points are added every 10 pixels along both sides of the curves. We then render a Voronoi diagram of the points (similar to Jeschke et al. [JCW09a]).

Let x_i be a color point. We can approximate the major axis of Voronoi region i as the furthest distance, d_i, from point x_i to any pixel in the region. To make the region square (to equalize the representable frequencies both along and perpendicular to the curves) the major and minor axes of the region should be the same, so the distance between x_i and its neighbors should also be d_i. Based on this idea, we remove color points as follows: For each curve, start with the first color point and retain it. Next, walk along the curve deleting color points until the walked length is greater than the maximum distance encountered since the last retained point, d_j. Retain this color point (call it x_k) and repeat until the second-to-last color point on the curve. Finally, retain the last point.

Color simplification. After the initial placement based on curve geometry, we remove any remaining superfluous color points that are not needed to define the curve colors, as follows: First, compute initial colors for all color points as will be described in Sec. 3.2. Then, remove unneeded colors using the same curve simplification procedure described by Orzan et al. [OBW*08]. We use an error threshold of 25 units in CIE L*a*b* space to determine whether to remove color points in this procedure. Afterwards, colors are computed again for the new (sparser) set of color points and simplified. The process terminates when no more color points are removed during the simplification step. Figure 2 shows the process of deriving the color points, starting with a dense set of points, removing some of them based on geometric spacing, and arriving at a final set through iterative point removal guided by the source image colors.

3.2. Automatic Color Value Assignment

Problem statement. In this section we assume that the color points have already been identified (see Sec. 3.1), and we are interested in solving for the according color values. Given an image with m pixels and a number of diffusion curves with n color points, a diffusion curve image can be defined in matrix form as follows:

$$\mathbf{Ax} = \mathbf{b}, \qquad (1)$$

where $\mathbf{b}$ is a vector of length m defining the diffused image and $\mathbf{x}$ is a vector of length n that contains the color values of the color points. n is on the order of 20 to 2,000 for typical diffusion curve images and m is 262,144 for a 512^2 pixel image, for example. $\mathbf{A}$ is an $n \times m$ matrix containing in each column n weights that define the *amount of influence* each color point has on that particular pixel. With this definition, the problem we have to solve is: given $\mathbf{b}$ (the image) and

A (the influence matrix), compute **x** (the colors of the color points) so that the resulting image $\mathbf{b}'$ is "as close as possible" to **b**. We solve this equation in the least squares sense, multiplying both sides of Equ. 1 by $\mathbf{A^T}$ and solve the resulting system using a standard preconditioned conjugate gradient solver. $\mathbf{A^T A}$ is a symmetric positive semi definite matrix which is small enough so that it is quite easy to work with in practice. By contrast, **A** is a very large matrix that is difficult to capture and work with. While solving least squares optimization problems via the normal equations is a standard approach, the technical contribution of our work is an elegant and efficient algorithm to obtain the products $\mathbf{A^T A}$ and $\mathbf{A^T b}$ without the need to construct **A** first.

Efficient acquisition of $\mathbf{A^T A}$ and $\mathbf{A^T b}$. First we describe how to form **A** by acquiring the according weights. Afterwards, we extend our discussion to the efficient acquisition of $\mathbf{A^T A}$ and $\mathbf{A^T b}$.

A simple process to obtain the influence of a particular color point (call it x_i) on each pixel of the output image is as follows: set all color points in **x** to 0 except for x_i that is set to 1. Now diffuse the curve colors using for example the variable stencil diffusion described in [JCW09a]. The resulting image contains exactly the weights of x_i for all pixels (column i of matrix **A**). Fig. 3 (top, left) shows the resulting weights for 3 color points. The process is repeated for all color points, thus successively filling the rows of **A**. However, for a larger number n of color points, this approach quickly causes computation and memory problems. Diffusing the whole image for each x_i will be slow for a large number of colors ($n > 100$). For example, for the 283 colors of the chick in Fig. 1 this approach required 78.4 s compared to 0.43 s with the new approach. Even worse, each x_i generates a row with the same number of entries as there are pixels in the image, which quickly exceeds the available memory.

In order to overcome this problem, instead of diffusing each x_i separately to form **A**, we diffuse *all weights simultaneously* in a single diffusion and form $\mathbf{A^T A}$ and $\mathbf{A^T b}$ from the resulting image. For this we modified the solver of Jeschke et al. [JCW09a]. The main innovation is to store the l (in our case 4) most influential color point IDs at each pixel. In each Jacobi iteration we (1) collect the color point IDs and weights from four neighbours (potentially $4l$). (2) Then we merge identical color points (by adding their weights) and (3) store from this set the four with the highest weights. These four weights are (4) normalized to sum to one for each pixel and finally stored for the next Jacobi iteration. This process is repreated for the desired number of iterations (typically 8). Let W denote this *combined weight image*. Note that every pixel in W contains precisely the l most influential color points and according weights, so that all information required to form $\mathbf{A^T A}$ and $\mathbf{A^T b}$ is present after a single diffusion process. Of course, some nonzero coefficients are lost since we only propagate the l most influencial color points per pixel. This can be observed as the difference between the

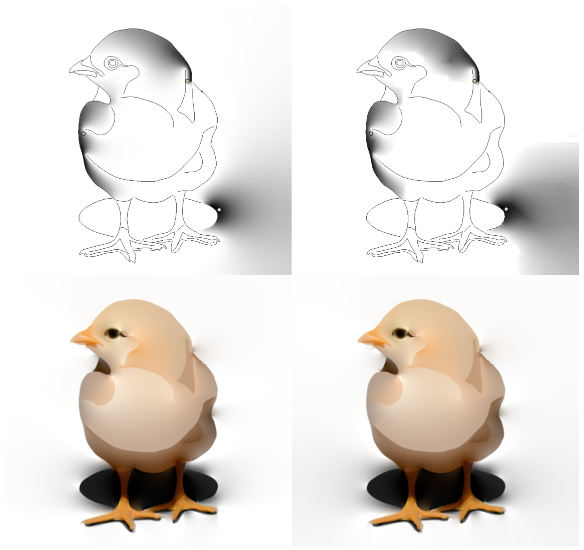

Figure 3: *Top row: weights for 3 color points, either individually diffused (left) or diffused in parallel (right). Bottom row: the according results of the solver.*

left and right top image in Fig. 3. However, note that near the color points where pixels have high weights, the weights are fairly similar. In numerous tests we confirmed that the differences to a solution built with an individual diffusion pass per color point are hardly noticeable, as shown in the bottom row in Fig. 3. We have also experimented with using fewer than four significant colors. The result with three significant colors is similar to the four color case, but when just one or two significant colors are used, the reconstructed image tends to appear flat and dull (see additional images, Fig. 11).

In practice, the diffusion process runs entirely in graphics hardware, taking advantage of the highly parallel architecture. In order to make best use of a single common 4 channel floating point texture, we encode each color point ID and according weight into a single float value with the integer part defining the ID and the residual defining the weight. As a result, $l = 4$ different color points can be stored per pixel. Of course, l could trivially be increased by diffusing multiple textures. However, as stated above, we did not see any need for this in our experiments. Diffusing in this manner has the added benefit of making **A** and $\mathbf{A^T A}$ sparser.

The most striking aspect of W is that it implicitly encodes the $\mathbf{A^T A}$ product. Thus, $\mathbf{A^T A}$ and $\mathbf{A^T b}$ can directly be generated from W without having to create the potentially huge matrix **A** first. To see how this is so, consider the diagrams in Fig. 4. Each pixel in the image defines a (sparse) row in **A** and a corresponding column in $\mathbf{A^T}$. This may seem a mundane observation, but it becomes important when coupled with the fact that row i in **A** interacts **only** with column i of $\mathbf{A^T}$ in the product. These correspond to the weights associated with pixel i in the **A** matrix. Consequently, to form

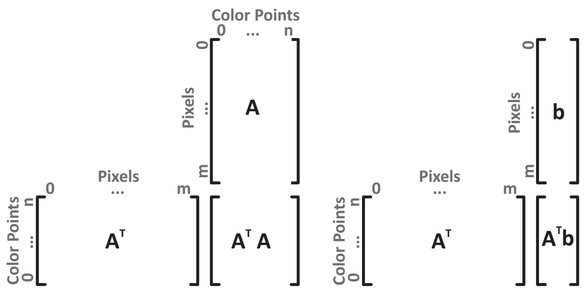

Figure 4: $\mathbf{A}^\mathbf{T}\mathbf{A}$ *and* $\mathbf{A}^\mathbf{T}\mathbf{b}$ *computation from a single image.*

$\mathbf{A}^\mathbf{T}\mathbf{A}$, only the pairwise products of the l weights in every pixel have to be multiplied and tallied in the according positions of $\mathbf{A}^\mathbf{T}\mathbf{A}$, as Fig. 4 (left) shows. $\mathbf{A}^\mathbf{T}\mathbf{b}$ is computed analogously (see Fig. 4 (right)). In a sense, we can say that $\mathbf{A}$ encodes the interactions between the image pixels and the color points, and $\mathbf{A}^\mathbf{T}\mathbf{A}$ encodes the interactions of the color point colors with each other.

Solving $\mathbf{A}^\mathbf{T}\mathbf{A}x = \mathbf{A}^\mathbf{T}\mathbf{b}$. We use the preconditioned conjugate gradient solver included with the boost UBlas library for solving the above equation. We tried several preconditioners and found the diagonal one to work best for our problem. $\mathbf{A}^\mathbf{T}\mathbf{A}$ as computed from W typically has a sparsity of less than 5%, which makes computations quite efficient.

3.3. Manual Attribute Adjustment

The combined weight image W can additionally be used as the basis for an intuitive user interface. Since W encodes the l most influencial color points, by simply clicking on any pixel the user is implicitly making a selection of l color points (4 in our case). Based on this selection, the user can change colors or other attributes to give the selected pixel a desired value. For example, suppose the user selects pixel p with color $c = (r,g,b)$, and then specifies a new color $c' = (r',g',b')$. We need to calculate a change to the corresponding color points, $x_1 \dots x_l$, that will cause p to have color c'. Our current implementation accomplishes the change by simply adding $c' - c$ to each of the selected color points. In the future, we would like to explore the relative merits of making the change on each color point proportional to its weight in the selected pixel.

All attributes in our system can be edited using “click and drag” operations, including color, edge blur, noise color, noise frequency, direction, and the variances of noise frequency and direction. This interface proved to be a very efficient and intuitive tool, as one just has to click at the place where changes should be applied rather than selecting and changing color points individually.

4. Adding Texture with Gabor Noise

Adding texture to vector graphics could be done with a recently proposed texture synthesis algorithm in the sense of “Texture by Numbers” presented by Hertzman et al. [HJO*01]. Problems with such an approach are the required definition of representative texture sample patches for different image regions, raster sampling issues, and the potential need to spatially change texture appearance in a gradual manner. Instead, we take a different route and define texture in a procedural way with Gabor noise [LLDD09]. Gabor noise parameters can be locally estimated in the input image, fitted to the curves just like colors and then used to steer the noise synthesis. The resulting DCIs allow for a number of rendering effects not possible with standard diffusion curves.

Here we briefly review the foundations of Gabor noise. The basic idea is to splat and accumulate Gabor kernels. A Gabor kernel $g(x,y)$ for position (x,y) is defined by a Gaussian envelope multiplied with a harmonic sinusoidal carrier. Both are defined with only 4 parameters, K, a, ω_0 and F_0:

$$g(x,y) = Ke^{-\pi a^2(x^2+y^2)} cos[2\pi F_0(x\cos\omega_0 + y\sin\omega_0)], \quad (2)$$

where ω_0 is the noise direction and F_0 is the scale of the sinusoidal carrier. K is the magnitude (fixed to 1 in this work) and a determines the Gaussian envelope diameter. We fixed this diameter to 40 pixels for rendering. In practice, we found that this looked better than using variable-sized splats. Note that a only determines the lowest possible noise frequency (which depends on the splat size), whereas F_0 defines the actual noise frequency. One desirable property is that all above parameters can be varied over an image plane, as was demonstrated in [BLV*10, LLD10].

To create a textured diffusion curve, we diffuse the curve colors, a *noise color* and four *noise parameters* that define its directional and frequency variation. Then the curve blur is applied to the resulting images containing color, noise color and noise parameters, respectively. Afterwards, a per pixel noise generation process is implemented that takes the noise parameters as input and generates a *Gabor noise image* in the range $[-1..1]$. The noise generation is implemented in a pixel shader based on the code provided by Lagae et al. [LLDD09]. Finally, for each pixel the noise image is multiplied with the noise color image and added to the diffused color image, resulting in the final image.

4.1. Noise Parameter Estimation

Given an input image and corresponding diffusion curves with color points, the task is to automatically derive noise parameters for the color points. After diffusing these parameters, we can then apply a noise generation process to approximate the original image. The fact that it is not possible to analyze the whole image at once excludes techniques like Fourier analysis, color channel decorrelation, histogram matching etc. that rely on coherent texture patches, e.g. [LVLD10, GDS10].

We start by locally estimating the Gabor parameters per pixel. These values are then fitted to the curves just like the colors (Sec. 3.2). For each pixel we apply Gabor filter analysis similar to [PZ89, FS89, BCG90]. More precisely, for the 40 pixel diameter circle around each pixel a set of Gabor wavelets is drawn from a filter bank and each is convolved with the input image. This is done with phase shifts of 0 and $\frac{\pi}{2}$ radians for the sinusoidal carrier and the L2 norm of the results is computed, making the filter response invariant to phase shift [FS89]. The responses of the three color channels are simply added. This works reasonably well in practice, but more elaborate methods could be employed [BCG90]. The filter bank consists of sinusoidal carriers at 15 degree angular rotations ω from 0 to 165 degrees. The Gabor kernel size (which determines the noise frequency) varies from 3.4 to 40 pixels with a factor of 1.4 between consecutive sizes, a total of 8 different sizes. After convolution the parameters with the maximum response are kept for each pixel. (Note that the analysis uses different kernel sizes, but for texture synthesis we use a fixed kernel width of 40 pixels. Also, because we only record a single dominant frequency, we only capture a rough impression of many textures.) These are then fitted to the curves exactly like the color fitting described in Sec. 3.2. An important technical detail is that rotation angles cannot directly be fitted because they cannot be interpolated. Instead, we use the tensor notation described by Zhang et al. [ZHT07]. The same tensor notation will is used in the diffusion and blur process during rendering.

(An)isotropy and scale variation: Most textures do not consist of a single frequency, but rather have a distribution of frequencies. Further, the direction of anisotropic features in a texture is likely to form a distribution rather than just being smooth flow lines. Thus, to capture a texture, we cannot simply fit smoothly varying Gabor kernel parameters to a region. We have to capture the distribution of those parameters.

In our system, we treat the noise parameters as Gaussian distributions, employing a simple maximum likelihood estimator for 4 noise parameters, the mean noise frequency μ_s, its variance σ_s^2, and the mean noise direction μ_d and its variance σ_d^2. The maximum likelihood for each of these parameters is computed on a per-pixel basis as follows: around each pixel a local neighborhood 40 pixels in diameter is considered. First mean values for μ_d and μ_s within the kernel are computed. Then the according variance σ_d^2 and σ_s^2 within the footprint f are computed as

$$\sigma_d^2 = \frac{1}{N}\sum_{i\in f}(d_i-\mu_d)^2, \qquad \sigma_s^2 = \frac{1}{N}\sum_{i\in f}(s_i-\mu_s)^2 \quad (3)$$

with N being the number of pixels in the footprint and d_i and s_i the estimated direction and size for the respective pixel i. During DCI rendering, the Gabor kernel splatting process simply chooses kernels with parameters drawn from the Gaussian distributions just described.

Noise color: We estimate the noise color (amplitude in RGB) similar to its frequency and direction: for each pixel an RGB mean color μ_c is computed using the same 40 pixel diameter footprint f. Then the variance

$$\sigma_c^2 = \frac{2}{N}\sum_{i\in f}(c_i-\mu_c)^2 \quad (4)$$

is computed with c_i being the noise color of a particular pixel in the footprint. Note that the factor 2 in 4 is needed because σ_c^2 is the *variance* of the color function. However, when multiplied with the Gabor noise, the *maximum variation* is required to retain the amount of color variation as in the original image. Because Gabor noise models variations with trigonometric functions, dividing the noise color by the variance of a trigonometric function (which is known to be $\frac{1}{2}$) provides the original contrast.

Curve-constraint sampling: In the parameter estimations described above, a circular region around each pixel is sampled. These regions might cross curves, which is not desirable as textures typically change on curves. While in general it is complicated to determine if the sampling process crossed a curve, a simple yet effective method was found that uses the combined weight image W again (see Sec. 3.2). Before sampling a region, the l most influencial color point IDs are read from W at the pixel position under consideration. Then, during the sampling process these color point IDs are also read for each sample in the circular region. If at least one of the l IDs matches an ID of the center position, the sample is considered to be *valid*. Otherwise, it is considered *invalid* and must not influence the sampling process, as it is presumably on the other side of a curve. When convolving with the Gabor filters, invalid pixels are assigned the average value of all valid pixels in the kernel, weighted by the Gaussion envelope. During the estimation of noise variation in direction, size and color, invalid pixels are just skipped and N is reduced accordingly in Equ. 3 and Equ. 4.

5. Application to Stippling and Hatching

The proposed fitting of parameters can also be used to drive other processes besides Gabor noise, such as stippling and hatching. After diffusing color, a DCI can directly be used to define the point density for a stippling process. We used the recursive Wang tile code provided by Kopf et al. [KCODL06] to generate a stippled image that allows for infinite zoom-ins. The points are generated on the CPU and rendered by the GPU. Hatches can be placed by the same process, as can be seen in Fig. 5.

6. Results

The DCI coloring and texturing algorithms decribed in sections 3 to 5 have been implemented on an Intel I7 920 CPU running at 2.7 GHz with 6 GB of RAM, an NVIDIA 8800 GTX graphics board and Windows 7 operating system. The diffusion and rendering were performed in DirectX10.

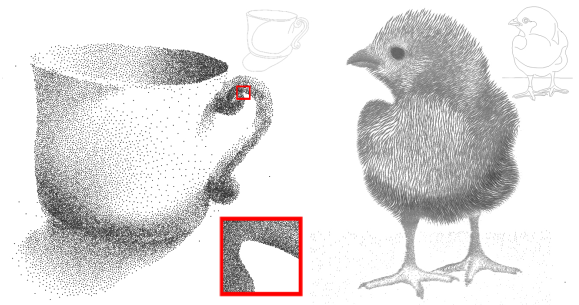

Figure 5: *Left: a cup rendering using diffusion curves and stippling. The insert shows the handle when zoomed in, still providing crisp boundaries. The right image shows the chick from Fig. 1 rendered with hatches, where the hatch directions are determined in the same way as noise directions.*

Figure #	1	2	6	7a	7b
Name	chick	boat	lena	beach	fish
Input image size	512	512	512	800	800
Number of curves	42	25	136	101	44
Color pnt. generation					
Sparse points (s)	2.4	1.9	5.5	17.4	7.5
Simplification (s)	1.6	1.4	5.5	5.3	2.8
Overall time (s)	4.0	3.3	11.0	22.7	10.3
Final number	283	176	682	586	361
[Orzan et al.08]	665	549	1178	850	754
Parameter fitting					
Gabor analysis (s)	17.7	17.8	23.2	46.3	44.3
Overall time (s)	19.7	19.6	26.6	49.9	47.5
Overall time (s)	23.7	22.9	37.7	72.6	57.8
Manual work					
Curve creation (min)	10	5	35	20	20
Parameter edit (min)	4	2	4	9	12

Table 1: *Statistics for the examples shown in the paper.*

Visual results. The chick graphic in Fig. 1 is a good example of the kind of result that can be achieved by adding noise texture to a DCI. Notice how our least squares coloring solution differs from the sampling approach in [OBW*08]. Because the edge of the chick is fuzzy, sampling the edge colors leads to numerous color outliers. In contrast, our approach has no problem with fuzzy and highly textured regions. The top right image shows how the automatic algorithm succesfully determined the directions of the feathers. Even the inflection point on the stomach is well reproduced. Some manual touchup was needed on this (and most images we tried), in part because the Gabor filters tend to latch onto large color gradients on fuzzy edges.

Most of the Lena image in Fig. 6 is well-reproduced by common diffusion curves. However, the shawl, hat and the hair look flat and blurred out without textures. By contrast, adding the Gabor noise textures provides a stylized visual impression that is more like the original photograph. Again, the automatic algorithm properly identified the directions and scale of the textures, which greatly reduced the time for manual editing afterwards.

The Gabor noise was also able to give a rough impression of the textures of the bottle in the beach example and the scales of the fish in Fig. 7. These subtle details could not be identified by the automatic algorithm, but were manually modeled using our interactive tool. Nevertheless, they demonstrate the usefulness of adding Gabor noise textures to DCIs. The directions and scales of the fin textures on the fish were automatically found, and make a great deal of difference in the visual impression of the rendering.

Timings. Table 1 displays statistics and timings for all of the examples shown in this paper. The number of curves for these graphics (25-136) is fairly typical of hand-generated DCIs. The overall color point generation took between 3 and 23 seconds, with the sparsification (see Sec.3.1) taking more than half of it. While this is not real time, it is certainly an acceptable wait in the context of an interactive editing session. All of the examples in the table took between 4 and 6 iterations for the point simplification, with most of the points being culled in the first pass. In general, the coloring solution of Orzan et al. [OBW*08] outputs 1.5 to 3 times as many color points as ours, and is prone to color overfitting. As an aside, we were quite surprised at how bad the overfitting can be in some instances, and by the fact that it can happen even when using the least squares solver to obtain the colors.

The time needed to fit attributes to the color points depends on two main steps. The first step is the generation of $\mathbf{A}^\mathbf{T}\mathbf{A}$ and $\mathbf{A}^\mathbf{T}\mathbf{b}$ from W, which includes a diffusion process and the matrix construction. Both of these steps depend primarily on the image resolution. We recorded times of 234 ms for 400×400 images and 468 ms for 800×800 images using our test setup. These numbers are fairly independent of curve complexity and the number of color points. The second step, the matrix solver, depends heavily on the number of color points and the matrix sparseness. Here, our test setup obtained speeds between 11 ms for 283 color points and 717 ms for 2700 color points.

Turning to the Gabor texture parameter fitting, overall the automatic fitting process is reasonbly fast (between 20 and 75 seconds). Most of the time is taken by the Gabor filter analysis (Sec.4.1). We note that our implementation is not optimized, and more elaborate methods could make it more practical. We did not observe a need to change the color point positions, nor their number after running the automatic generation process in any of the examples. While texturing parameter estimates were not always ideal, they did make a good starting point for manual editing. It seems unlikely that the color point placement and value assignments can be manually done at this quality in reasonable time without the automated assistance. The relatively few automatically generated color points, automated color and noise parameter

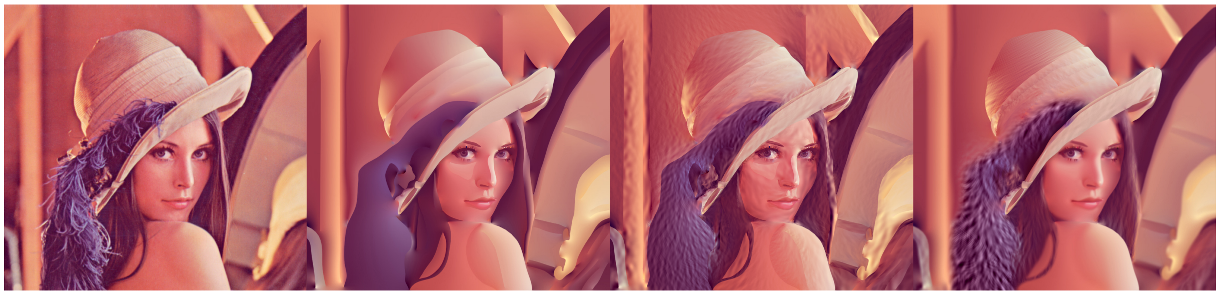

Figure 6: *The "Lena" image. (from left to right): original image, 682 automatically applied curve colors, result after the automatic texture parameter fitting, final image after manual editing.*

estimation, and the pixel-based editing interface all combine to make textured DCI authoring easier and more intuitive.

Rendering a textured DCI takes place in two stages, first the color and noise parameters are diffused over the image, then noise is rendered based on these parameters. At resolution 800 $\times$ 800, the fast diffusion method of Jeschke et al. [JCW09a] achieves 64 frames per second (FPS) when diffusing a single RGBA floating point image. Diffusing two additional images that contain noise color and parameters reduces the frame rate to 25 FPS. However, the most time consuming part of the rendering process is the Gabor noise application. As described in [LLDD09] the noise is rendered by dividing the domain into a spatial grid of cells into which Gabor kernels are splatted. The cells have the same width as a kernel, so rendering requires looking up the splats in a 3×3 grid neighborhood. Using 46 Gabor splats per cell provides interactive feedback with 12 FPS, but only at preview quality. With 128 splats very good image quality is achieved at 6.5 FPS. 256 splats provide excellent quality but only at 4 FPS. Example images are provided in a separate document due to space constraints. For high image quality, significantly more splats are required compared to [LLDD09], particularly for regions with high variation in noise direction and frequency. However, we note that our rendering is not yet optimized and cite [BLV*10] as a faster implementation.

Limitations Gabor noise as presented obviously has its limitations and can only model certain types of textures satisfactorily. Better noise models [PS99] that support multiple texture scales and directions, histogram matching, etc. would be desirable. In addition, the Gabor noise changes when the curves are animated as the underlying Gabor grid remains rigid. This becomes especially apparent for singular points in the tensor diffusion, which sometimes erratically change position during curve parameter editing.

7. Conclusions and Future Work

We have developed a method that automatically colors diffusion curves by choosing color points and calculating optimal colors for them. We showed that the algorithm works robustly, even when curves are placed inaccurately, and when small features and textures are present in the source image. Further, we derived parameters that drive a Gabor noise generation process in order to simulate textural details in vector graphics. Noise color, direction and scale variations were automatically estimated and an efficient tool was presented to edit curve parameters in an intuitive manner.

We believe that the topic of textured vector graphics will continue to be important in the future. In terms of extension to the work presented here, it would be interesting to be able to add more textural detail as one zooms in [HRRG08], for example. Automatically finding curves to separate different textured regions is another possibility for future work. In addition, storing separate control points for each parameter to be diffused might further improve the compactness and editability of our representation. We were also quite gratified by the success of the pixel-based parameter editor. We are planning to improve the interface and to apply similar ideas in different contexts. For example, our parameter fitting might be useful to efficiently compute harmonic coordinates [JMD*07].

Acknowledgements
This work was supported by the NSF and the Austrian Science Fund, Project FWF P20768-N13. We also thank Paul Guerrero for helping with the video.

References

[BCG90] BOVIK A. C., CLARK M., GEISLER W. S.: Multichannel texture analysis using localized spatial filters. *IEEE Trans. Pattern Anal. Mach. Intell. 12*, 1 (1990), 55–73. 3, 7

[BEDT10] BEZERRA H., EISEMANN E., DECARLO D., THOLLOT J.: Diffusion constraints for vector graphics. In *Proc. of NPAR '10* (2010), pp. 35–42. 3

[BLV*10] BÉNARD P., LAGAE A., VANGORP P., LEFEBVRE S., DRETTAKIS G., THOLLOT J.: A dynamic noise primitive for coherent stylization. *Comput. Graph. Forum 29*, 4 (june 2010), 1497–1506. 3, 6, 9

[DP73] DOUGLASS D., PEUCKER T.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer 10*, 2 (1973), 112–122. 3

[EG01] ELDER J. H., GOLDBERG R. M.: Image editing in the contour domain. *IEEE Trans. Pattern Anal. Mach. Intell. 23* (2001), 291–296. 2

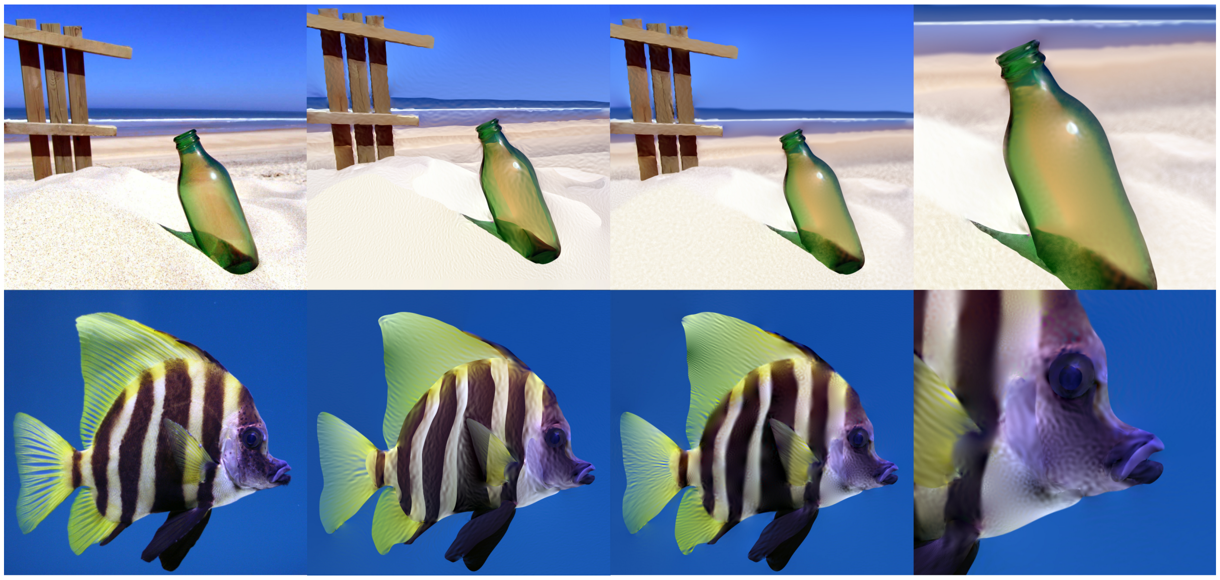

Figure 7: *Beach scene (top row) and fish scene (bottom row). From left to right: original images, images with 586 (361) automatically applied curve colors and noise parameters, final images after manual editing, closeups showing Gabor textures.*

[FS89] Fogel I., Sagi D.: Gabor filters as texture discriminator. *Biological Cybernetics 1*, 61 (1989), 103–113. 3, 7

[GDS10] Gilet G., Dischler J.-M., Soler L.: Procedural descriptions of anisotropic noisy textures by example. In *EG 2010 - short papers* (May 2010). 3, 6

[HGA*10] Hnaidi H., Guérin E., Akkouche S., Peytavie A., Galin E.: Feature based terrain generation using diffusion equation. *Computer Graphics Forum (Proceedings of Pacific Graphics) 29*, 7 (2010), 2179–2186. 3

[HJO*01] Hertzmann A., Jacobs C. E., Oliver N., Curless B., Salesin D. H.: Image analogies. In *Proc. of ACM SIGGRAPH '01* (2001), pp. 327–340. 6

[HRRG08] Han C., Risser E., Ramamoorthi R., Grinspun E.: Multiscale texture synthesis. *ACM Trans. Graph. 27*, 3 (2008), 1–8. 9

[JCW09a] Jeschke S., Cline D., Wonka P.: A GPU Laplacian solver for diffusion curves and Poisson image editing. *ACM Trans. Graph. 28*, 5 (2009), 1–8. 3, 4, 5, 9

[JCW09b] Jeschke S., Cline D., Wonka P.: Rendering surface details with diffusion curves. *ACM Trans. Graph. 28*, 5 (2009), 1–8. 3

[JMD*07] Joshi P., Meyer M., DeRose T., Green B., Sanocki T.: Harmonic coordinates for character articulation. *ACM Trans. Graph. 26* (2007). 9

[KCODL06] Kopf J., Cohen-Or D., Deussen O., Lischinski D.: Recursive Wang tiles for real-time blue noise. *ACM Trans. Graph. 25*, 3 (2006), 509–518. 2, 7

[LL06] Lecot G., Levy B.: Ardeco: Automatic region detection and conversion. In *Rendering Techniques 2006* (2006), pp. 349–360. 2

[LLC*10] Lagae A., Lefebvre S., Cook R., DeRose T., Drettakis G., Ebert D., Lewis J., Perlin K., Zwicker M.: State of the art in procedural noise functions. In *EG 2010 - State of the Art Reports* (May 2010). 3

[LLD10] Lagae A., Lefebvre S., Dutré P.: Improving Gabor noise. *IEEE Transactions on Visualization and Computer Graphics* (2010). to appear. 3, 6

[LLDD09] Lagae A., Lefebvre S., Drettakis G., Dutré P.: Procedural noise using sparse Gabor convolution. *ACM Trans. Graph. 28*, 3 (2009), 54:1–54:10. 2, 3, 6, 9

[LVLD10] Lagae A., Vangorp P., Lenaerts T., Dutré P.: Technical section: Procedural isotropic stochastic textures by example. *Comput. Graph. 34*, 4 (2010), 312–321. 3, 6

[MM96] Manjunath B. S., Ma W. Y.: Texture features for browsing and retrieval of image data. *IEEE Trans. Pattern Anal. Mach. Intell. 18*, 8 (1996), 837–842. 3

[OBW*08] Orzan A., Bousseau A., Winnemöller H., Barla P., Thollot J., Salesin D.: Diffusion curves: a vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (2008), 1–8. 1, 2, 3, 4, 8

[PB06] Price B., Barrett W.: Object-based vectorization for interactive image editing. *Vis. Comput. 22*, 9 (2006), 661–670. 2

[PS99] Portilla J., Simoncelli E. P.: Texture modeling and synthesis using joint statistics of complex wavelet coefficients. In *IEEE Workshop on SCTV* (June 1999). 9

[PZ89] Porat M., Zeevi Y.: Localized texture processing in vision: Analysis and synthesis in the Gaborian space. *IEEE Trans. Biomed. Eng. 36*, 1 (January 1989), 115–129. 3, 7

[PZ08] Parilov E., Zorin D.: Real-time rendering of textures with feature curves. *ACM Trans. Graph. 27*, 1 (2008), 1–15. 2

[SLWS07] Sun J., Liang L., Wen F., Shum H.-Y.: Image vectorization using optimized gradient meshes. *ACM Trans. Graph. 26*, 3 (2007), 11. 2

[WOBT09] Winnemöller H., Orzan A., Boissieux L., Thollot J.: Texture design and draping in 2d images. *Comput. Graph. Forum 28*, 4 (2009), 1091–1099. 3

[ZHT07] Zhang E., Hays J., Turk G.: Interactive tensor field design and visualization on surfaces. *IEEE Trans. Vis. Comput. Graph. 13*, 1 (2007), 94–107. 7