Graphical Models 73 (2011) 74–87

Contents lists available at ScienceDirect

Graphical Models

journal homepage: www.elsevier.com/locate/gmod

Structure grid for directional stippling ☆

Minjung Son [a], Yunjin Lee [b,*], Henry Kang [c], Seungyong Lee [a]

[a] Dept. of Computer Science and Engineering, POSTECH, Pohang 790-784, Republic of Korea
[b] Division of Digital Media, Ajou University, Suwon 443-749, Republic of Korea
[c] Dept. of Mathematics and Computer Science, University of Missouri, St. Louis, MO 63121, United States

ARTICLE INFO

Article history:
Received 24 August 2010
Received in revised form 2 December 2010
Accepted 8 December 2010
Available online 15 December 2010

Keywords:
Stippling
Hatching
Hedcut illustration
Texture synthesis

ABSTRACT

This paper presents a novel method to convert a photograph into a stipple illustration. Our method addresses *directional* stippling, where the collective flows of dots are directed parallel and/or orthogonal to the local feature orientations. To facilitate regular and directional spacing of dots, we introduce the notion of a *structure grid*, which is extracted from the smoothed feature orientation field. We represent a structure grid as a 2D texture and develop an efficient construction algorithm that outperforms conventional Lloyd's method in terms of the rigor of dot alignment. Moreover, the criss-crossing nature of a structure grid allows for the inclusion of line primitives, providing effective description of dark tone. Given a structure grid, we determine the appropriate positions and attributes of primitives in the final illustration via rapid pixel-based primitive rendering. Experimental results show that our directional stippling method nicely reproduces features and tones of various input images.

1. Introduction

Resynthesizing an image using points is a recurring problem in computer graphics. It has been addressed in different but related contexts such as halftoning, sampling, and stipple drawing. The main goals in this line of work are often summarized as how well the point distribution approximates the original tone without creating disturbing visual artifacts, e.g., point clustering or aliasing. One of the recent trends in halftoning and stippling is to incorporate structure or orientation information in generating dot distributions. For instance, the structure-aware halftoning techniques [1,2] are particularly useful in preserving visually identifiable structures and textures in the original image.

More closely related to our work is the *directional stippling* technique [3,4] which puts more emphasis on aesthetic distribution of points that collectively follows the smooth feature orientations in the original image. Such a stippling style is known as *hedcut*, in which a set of regularly spaced points tends to form a 'flow' that lines up strictly along and/or perpendicular to the dominant feature orientations nearby (Fig. 1a and b). The hedcut illustration style is originally "designed to emulate the look of woodcuts from old-style newspapers, and engravings on certificates and currency" [5], and has often been used by artists for illustrating portraits in newspapers and magazines. In addition to the directional spacing of dots, hedcut stippling has another notable difference. Conventional stippling usually describes tone by controlling the distribution density while keeping the dot size fixed. In contrast, hedcut stippling describes tone by controlling the dot size while keeping the distribution density fixed (Fig. 1a and b).

Many sampling/stippling methods employ Lloyd's algorithm (or its variants) [6–10] to provide regular dot spacing but they do not support such flow-like arrangement of dots. On the other hand, the method of [3] does generate hedcut-style stipple illustrations, based on an orientation-constrained Lloyd's algorithm. This method, however, is not without limitations. As the authors pointed out, it

☆ This paper has been recommended for acceptance by Jarek Rossignac and Douglas DeCarlo.

* Corresponding author. Fax: +82 31 219 1797.
E-mail addresses: sionson@postech.ac.kr (M. Son), yunjin@ajou.ac.kr (Y. Lee), kang@cs.umsl.edu (H. Kang), leesy@postech.ac.kr (S. Lee).

1524-0703/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.gmod.2010.12.001

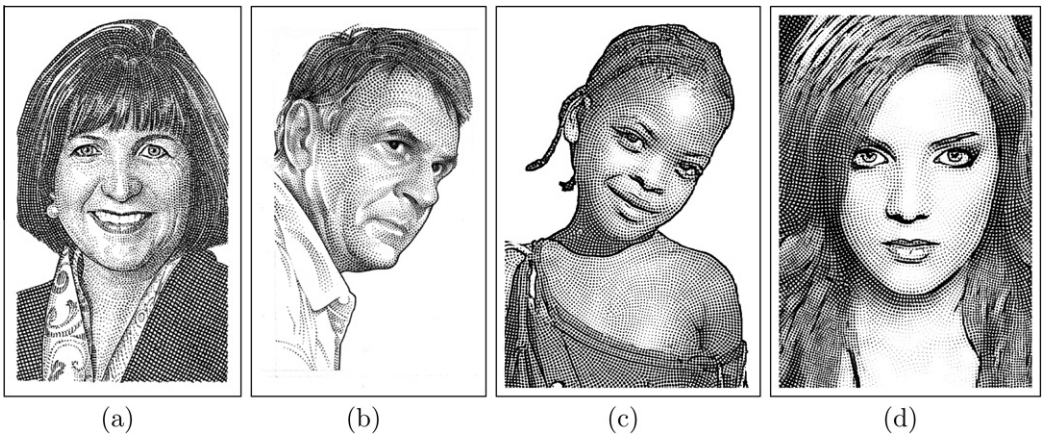


Fig. 1. Hedcut illustrations created by artists: (a) Kevin Sprouls (www.sprouls.com), (b) Randy Glass (www.randyglassstudio.com), and our method (c) and (d).

aligns points well along the feature lines, but not necessarily along the perpendicular directions and thus may lack the structural rigor of true hedcut style. In this paper, we present an entirely different approach, which is not based on Lloyd's method. The core idea is to construct a feature-adaptive grid, which we call a *structure grid*, using a smoothed feature orientation field. Unlike Lloyd's method, the use of a structure grid ensures regular spacing of dots along the feature directions as well as their perpendicular directions.

Another limitation of [3] (or stippling in general) is that despite the efforts to maintain the uniformity of point distribution, it is hard to avoid point clustering or unwanted patterns in dark regions. This problem may occur regardless of the dot management policy: varying dot size (dots too big) or varying dot density (dots too many). While reducing the sizes (or density) of dots could avoid point clustering, it would result in ineffective representation of dark tone. As shown in Fig. 1a, hedcut artists often use hatching lines to represent dark tone, as a way to prevent point clustering and its resulting distracting patterns. Our structure grid allows for a similar solution. Its strict alignment of points enables us to add line primitives connecting the dots along the tangential and normal directions of features. Consequently, we can easily handle the mixing of points and lines, as well as seamless transition from bright tone to dark tone. In this sense, a structure grid provides a unified platform for handling the mixture of feature-guided stippling and feature-guided hatching.

A structure grid, as described above, plays an essential role in our scheme. We define it as a smooth grid that curves along a vector field reflecting feature directions. We use a 'texture' instead of a set of explicit lines to represent a structure grid. Our texture representation contains at each pixel the tangential and normal distances to the nearest grid intersection. Such a representation allows us to construct a structure grid by synthesizing a grid texture from a small grid image with smoothed feature directions as constraints. Instead of relying on an existing 2D texture synthesis method, we develop an efficient technique for structure grid construction which synthesizes and merges two stripe patterns along tangential and normal directions of features. We also design a pixel-based primitive rendering algorithm that takes advantage of the structure grid to determine the positions and sizes of stipple dots and hatching lines in an illustration.

In summary, the main contributions of our paper are as follows: We introduce structure grid as an effective and unified tool for directional stippling. We present the definition, construction method, and rendering algorithm for a structure grid. As shown in Fig. 1c and d, the use of a structure grid provides the following benefits:

- A structure grid facilitates uniform spacing of dots along the tangential and normal directions of features, that is, the two major directions that constitute point distribution in a typical hedcut illustration. This produces dot distributions with stricter alignment than the existing methods based on Lloyd's algorithm [11,3].
- A structure grid gives an ability to describe continuous tone with a mixture of stippling and hatching, which leads to a more effective description of dark tone than stipple-only rendering.

2. Related work

2.1. Halftoning

Halftoning is a reproduction technique to approximate the original image with a reduced number of colors or grayscales. The output of halftoning often consists of a set of scattered black dots that are strategically placed to best describe the original tone. Recently, structure-aware halftoning techniques [1,2] have been developed based on iterative optimization and error diffusion, respectively. Their main goal is to improve the structure and texture similarities between the original and the halftone images, rather than generating stylistic and directional dot distributions as in hedcut illustrations.

2.2. Stippling/sampling

Deussen et al. [6] described stippling as a special kind of non-photorealistic rendering, having more freedom than

halftoning in terms of dot size, shape, and style of distribution. They showed that Lloyd's method produces aesthetic stipple drawing with rigorous dot spacing. Secord [7] presented a modified Lloyd's method that incorporates data-driven weights, for better protection of image features such as edges.

Stippling, as with halftoning, could similarly benefit from reducing aliasing artifacts by seeking blue noise spectral characteristics. Cohen et al. [12] used Lloyd's method to produce a tileable set of blue noise dot distributions. Kopf et al. [8] developed a recursive Wang-tiling method to enable real-time, dynamic control of blue noise dot distributions. Ostromoukhov et al. [13,9] presented fast hierarchical blue noise sampling algorithms by preprocessing Lloyd's relaxation step with Penrose tiling and rectifiable polyominoes, respectively. Balzer et al. [10] developed a capacity-constrained Lloyd's method that produces point distributions with blue noise characteristics but fewer regularity artifacts.

Mould [14] presented a stippling algorithm based on graph search (instead of Lloyd relaxation) for improved protection of image features, such as edges. There are a few methods based on analysis of hand-drawn stippling images [15–17]. Maciejewski et al. [15] quantified the difference between a computed-generated stippling and hand-drawn stippling of the same object. Kim et al. [16] used the same evaluation method as Maciejewski et al. [15] to reproduce the statistical characteristics represented in hand-drawn stippling. Martin et al. [17] developed an example-based method to produce scale-independent stipple illustrations for specific output size and resolution.

2.3. Directional stippling/hatching

None of the halftoning/sampling/stippling techniques mentioned above attempts to place the entire set of points in the image along curves with some smoothly varying orientations. Artistic rendering often involves such full-scale directional distribution of primitives, as seen in many of the existing painting and pen-and-ink algorithms [18,19]. Hedcut illustration, and thus also our work, has this property.

Hertzmann and Zorin [20] suggested an automatic method to generate directional and equally spaced hatching curves on 3D smooth surfaces, which however is not directly applicable to 2D images due to lack of information on the surface geometry, such as principal curvature directions.

Ostromoukhov's digital facial engraving system [21] takes a facial image as input and places a set of equally spaced engraving lines along the iso-parametric curves on the face, producing images reminiscent of hedcut illustrations, albeit without stippling. This system requires the user to manually segment the facial surface and carefully determine the parametric grid in each of the segmented regions.

Hausner's tile mosaics technique [11] employs Lloyd's method with Manhattan distance metric to arrange rectangular tiles along the feature flow while tightly packing the space. Kim et al. [3] developed a constrained Lloyd algorithm to align dots strictly along the smooth image feature flow, resulting in a hedcut-style illustration with more rigorous dot spacing than Hausner's approach. Recently, Kim et al. [4] incorporated shading information in the direction field for better description of the local shape and depth of the surface.

3. Overall process

Fig. 2 shows the overview of our directional stippling method, which contains the following three components.

- *Feature vector field generation*: From the input photograph, we first extract feature lines such as edges, then construct the feature vector field by interpolating the feature directions on the lines.
- *Structure grid construction*: Given the feature vector field, we generate a structure grid, with which to guide the placement of primitives. The structure grid is essentially a 2D rectangular grid that is smoothly deformed along the local feature directions. The criss-crossing grid lines represent two local axes, one along the feature (tangent) direction and the other along its perpendicular (normal) direction.
- *Primitive rendering*: Guided by the structure grid, we draw primitives, i.e., stipple dots and hatching lines. In principle, dots are placed at the grid intersections and lines are placed on the grid lines. The attributes

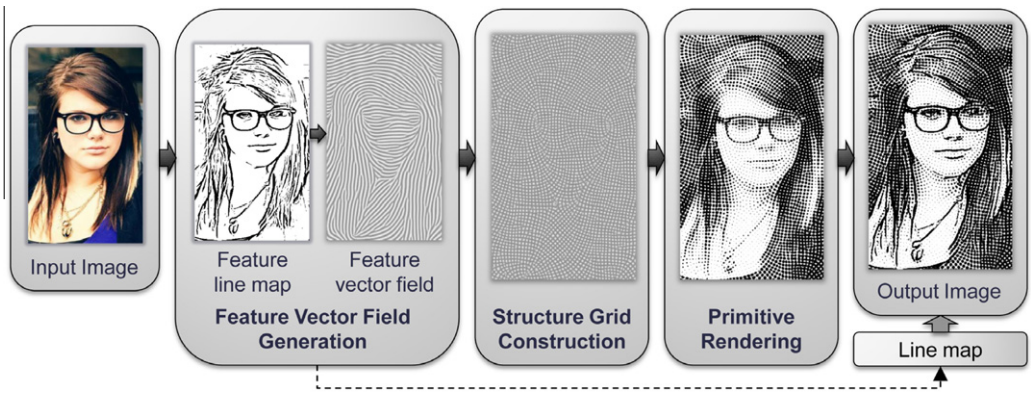


Fig. 2. Overall process of our directional stippling method.

(position, size, orientation) of each primitive are determined by examining the structure grid as well as the tone map. Finally, we add the extracted feature lines to the output in order to improve the clarity of illustration, as often done by professional artists.

4. Feature vector field generation

We first extract a black and white feature line map from the input image, where black pixels denote feature points. The tangential feature direction is also obtained at each feature point. The direction vectors at white (non-feature) pixels are then interpolated from the feature points. The result of this scattered orientation interpolation is our *feature vector field*, a smoothly varying vector field that records at each pixel the dominant feature directions nearby.

4.1. Feature line map extraction

For constructing the line map, we employ the flow-based line drawing method by Kang et al. [22], which constructs coherent and stylized feature lines. The tangential direction at each feature pixel is obtained from the edge tangent flow (ETF), a by-product of this line drawing algorithm (see [22] for more detail).

In professional illustrations, artists often place primitives along the principal curvature directions to depict the shape of a surface [23,20]. Since it is impossible to estimate such principal directions directly from a 2D image, we use *isophote curves* [24,4] to further assist in reflecting the approximate shape of the surface, under the assumption of Lambertian shading with a constant surface material property.

We first quantize the image intensities to a user-specified level and detect isophote curves along the quantization boundaries. The isophote pixels are then added to the feature line map together with the directions obtained from ETF. In our experiments, we use zero (no isophote curve) to four quantization levels.

Fig. 3b shows the feature line map for the input image in Fig. 3a. Black lines show the result of the line drawing algorithm [22]. Blue lines are the added pixels from isophote curves, which reflect the shape of the nose and the shades on the cheek.

4.2. Feature vector interpolation

To determine direction vectors for white (non-feature) pixels, we apply scattered data interpolation to the vectors at feature pixels. We find that scattered data interpolation outperforms ETF smoothing in creating consistent flow and also in reducing singularities in the vector field. We adopt multi-level B-spline interpolation [25], which runs fast and is easy to implement.

Before interpolation, we convert direction vectors to 2×2 structure tensors to avoid cancel-out of two opposite vectors (e.g., $(1,0)$ and $(-1,0)$) [26]. We apply scattered data interpolation to each component of the structure tensor. At each non-feature pixel, an eigenvector of the interpolated structure tensor serves as the new direction vector. Fig. 3c and d show the feature vector fields obtained from the feature line maps in Fig. 3a and b, respectively.

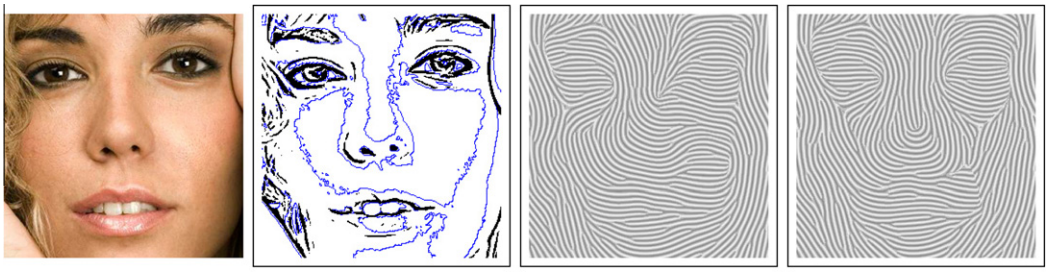
5. Structure grid construction

A *structure grid* consists of deformed 2D grid patches that conform to the given feature vector field (Fig. 5e). It facilitates aligned placement of primitives (dots and lines) with regular spacing along both tangent and normal directions of the features (Fig. 4).

In order to construct a structure grid, one could attempt to use texture synthesis (e.g., [27]) with a small 2D grid image as texture sample and with the feature vector field as constraints for local texture orientations. However, we found that existing texture synthesis techniques do not always perform well for such grid-shaped texture. In particular, it may obscure some grid intersections and grid lines (Fig. 5h). This is harmful because grid intersections and grid lines are where the stipple dots and hatching lines will be positioned, and thus directly affect the quality of stipple illustration. Instead, we synthesize a set of stripes in two orthogonal directions, such that they follow the tangential and normal directions of the feature vector field.

5.1. Structure grid and stripe pattern definition

A *structure grid* G is a vector-valued image (G: $p \rightarrow (t_0, t_1)$), where t_0 and t_1 denote the distances from pixel p



(a) Input image (b) Feature linemap with isophote curves (blue curves) (c) Feature vector field without using isophote curves (d) Feature vector field using isophote curves

Fig. 3. Feature vector field generation.

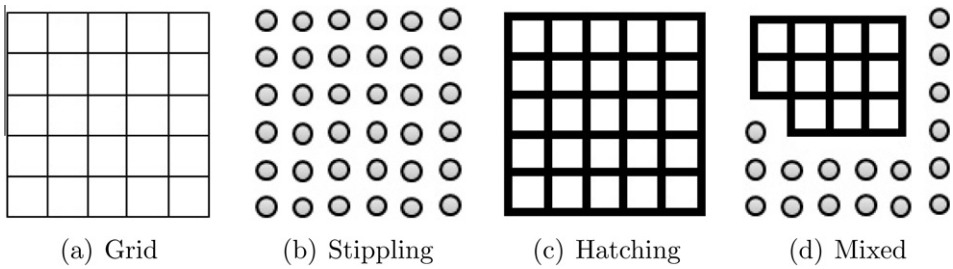


Fig. 4. Arranging stipple dots and hatching lines with a grid.

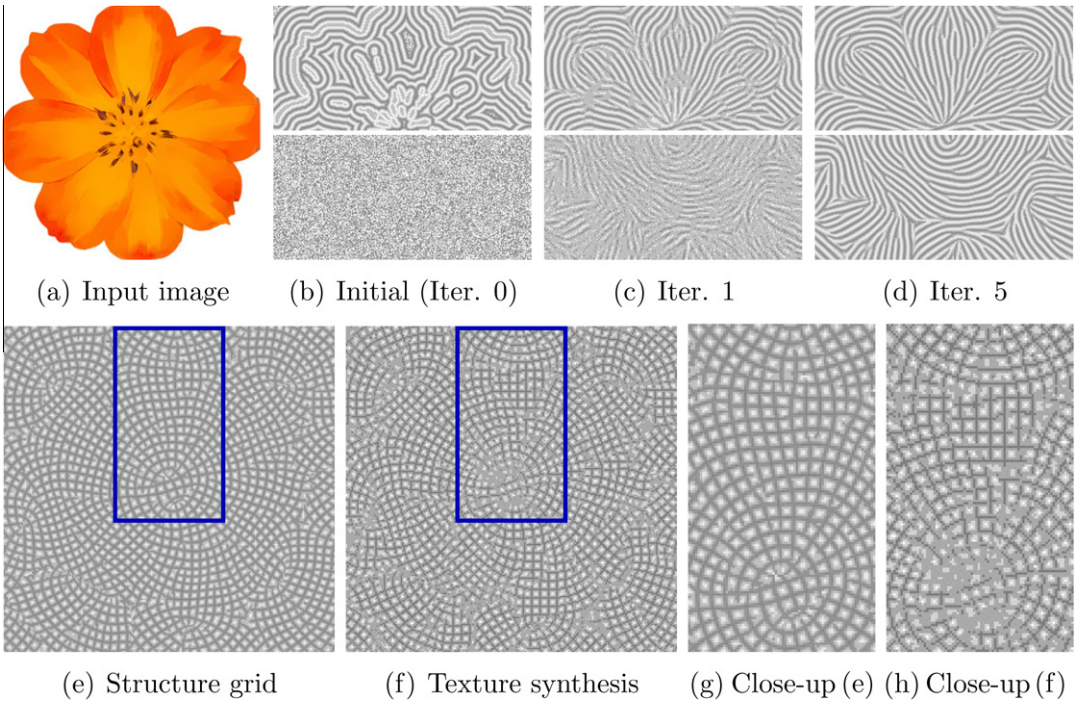


Fig. 5. Stripe pattern synthesis and the resulting structure grid: (a) input image, (b)–(d) Upper: stripe pattern following the feature directions, lower: stripe pattern perpendicular to the feature directions, where the pixel values are mapped to grayscale values, (e) structure grid, where the minimum among the two values at each pixel is mapped to a grayscale value, (f) grid pattern constructed by a 2D texture synthesis technique [27], (g) and (h) enlargements of (e) and (f).

to the nearest grid lines measured along the two perpendicular local axes, respectively. Let d denote the desired interval between neighboring grid lines. We then let t_i range in $[0,d/2]$, reflecting the periodic nature of the grid. We use the same interval d in both tangential and normal feature directions to ensure regular primitive spacing. The default value for d is six pixels. With this setting, stipple dots should be located at pixels with value pair $(0,0)$, while hatching lines should be placed along $(t_0,0)$ or $(0,t_1)$.

We construct a structure grid by running two passes of stripe pattern synthesis along tangential and normal feature directions, respectively. Here, a *stripe pattern P* refers to a scalar-valued image (P: $p \to t$) that accounts for one of the two distance values (t_0,t_1) at each pixel p (Fig. 5d and e).

5.2. Stripe pattern synthesis

Given a feature vector field F and a desired spacing d, we synthesize a stripe pattern P via iterative refinement of the distance values in P through local optimization. For the stripes along the tangential directions of F, we initialize each pixel with the distance from the nearest feature line. We compute these initial distances via jump-flooding algorithm [28]. The distance values are converted into real numbers in $[0,d/2]$ using a periodic reflection function,

$$S(x) = S(x+d) = \min(x, d-x), \tag{1}$$

where $0 \leqslant x < d$. Function S represents a 1D version of a regular grid, whose value corresponds to the distance from the nearest grid point. For the stripes along the normal

directions, there is no intuitive way to initialize the values in P, and thus we use random initial values in $[0, d/2]$.

For each pixel p in P, the goal is to obtain its optimal value t. We iteratively update t using a neighborhood window W of size $(2r+1) \times (2r+1)$ centered at p. In our implementation, we set $r = d$; the window size is twice the grid line spacing. Window W should be oriented to follow either the tangential or normal direction of F at p, depending on the target stripe orientation (Fig. 6, left, when $r = 3$). We obtain a 1D array of values by computing the average pixel value of each row in W along the target orientation (Fig. 6, the vertical array in the middle). The resulting 1D array of size $(2r+1)$, which is denoted by $w[i]$, $i = -r, \ldots, r$, summarizes the current t value distribution around the center pixel p.

Now to estimate the optimal t value for p, we find the best match of the array w against the periodic reflection function S. For element-by-element comparison of $w[i]$ and S, we define a template array $s_x[i]$ for S, which has the same size as $w[i]$, where $s_x[i] = S(i+x)$, $0 \leqslant x < d$ and $i = -r, \ldots, r$. This template array s_x contains shifted samples of the function S, where x is the shift amount (see Fig. 6, right, array s_x on top of function S). We then analytically find x that minimizes the difference between $w[i]$ and $s_x[i]$,

$$E(w, s_x) = \sum_{i=-r}^{r} (s_x[i] - w[i])^2. \tag{2}$$

We first divide the range of x into subranges $[m/2, m/2 + 0.5]$, $m = 0, \ldots, 2d-1$. Note that in each subrange, all elements in $s_x[i]$ can be determined without using min function in Eq. (1). That is, in a subrange, $s_x[i]$ is either $i + x$ or $d - (i + x)$ regardless of the value of x. Consequently, $E(w, s_x)$ is reduced to a quadratic function of x in each subrange, and we can easily compute x_m that minimizes $E(w, s_x)$ in the mth subrange. We select x that has the smallest cost $E(w, s_x)$ among the x_m values from all subranges. The selected x determines the sequence s_x that best matches the current neighborhood around p. The distance t at p is updated as $s_x[0]$, the central element of the array s_x.

This update operation is performed for every pixel in the stripe pattern P. The number of iterations for pixel updates in P is user-specified (typical number is between 3 and 8). The upper parts of Fig. 5b–d show the iterative process to generate a stripe pattern along the tangential directions of the feature flow F. Fig. 5b shows the initial tangential stripe pattern obtained using the distances from the feature lines, while Fig. 5c and d show iterative refinement results. A higher number of iterations usually means longer processing time but better synthesis result. After the tangential stripe pattern has been generated, the normal stripe pattern synthesis follows, and is conducted similarly. The lower parts of Fig. 5b–d show the process, where random values have been used for the initial normal stripe pattern. This second synthesis pass completes a structure grid G, shown in Fig. 5e. The whole process for structure grid construction consists of pixel-wise operations, and we implemented it on GPU.

By separating a structure grid into two orthogonal stripe patterns, we can outperform conventional 2D texture synthesis in preserving the integrity of the grid intersections and the grid lines (Fig. 5g and h). Moreover, this approach leads to significant acceleration. In 2D texture synthesis, to update the value pair (t_0, t_1), we need to consider all pixels in the $(2r+1) \times (2r+1)$ window W and the new value pair is chosen from the 2D range $\{(x,y) | x, y \in [0, d/2]\}$. The resulting operation count is $O(d^2(2r+1)^2)$. In our case, however, with the use of a 1D array w, a new value t can be chosen from the 1D range $[0, d/2]$. We synthesize two stripe patterns and the final operation count is $2 \cdot O(d(2r+1))$. See Appendix A for more detail of the operation counts.

When we place the window W to define the neighborhood of a pixel p, we can control the scale of W as well as the orientation. The scale factor is not related to the size of W, which is always $(2r+1) \times (2r+1)$. Instead, it determines the coverage of W for the neighbor pixels of p. When the scale factor is one, which is the default, W contains $(2r+1) \times (2r+1)$ pixels centered at p. If the scale factor is 1/2, the coverage of W reduces to $(2r+1)/2 \times (2r+1)/2$ pixels. We can use this scale factor to adaptively control

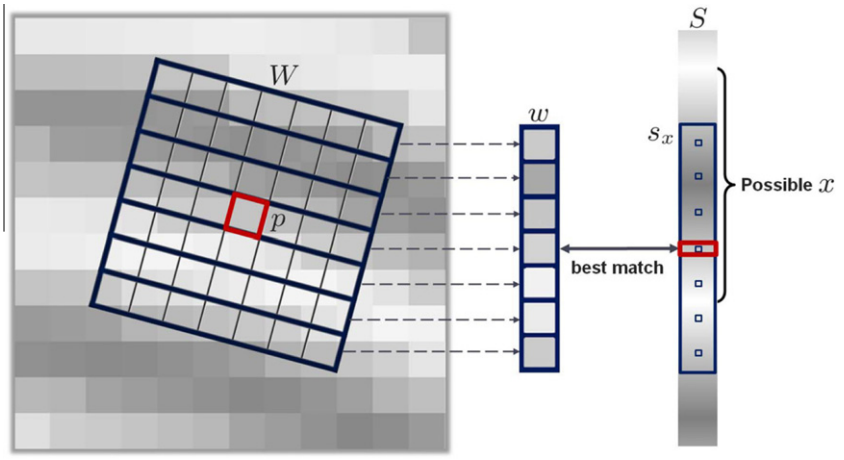


Fig. 6. Update of a pixel value in the stripe pattern synthesis.

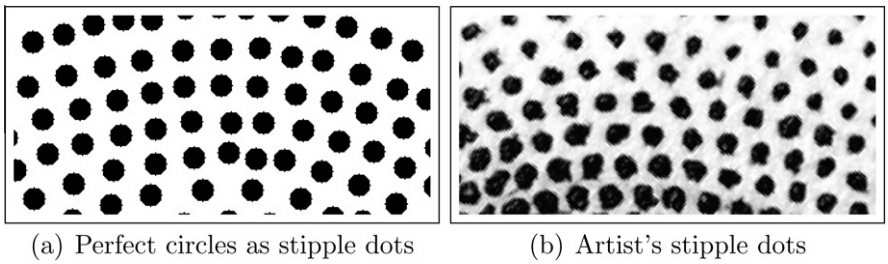


Fig. 7. Stipple dot shape comparison.

the spacing of stripes (and so grid lines), e.g., with image tone values.

6. Primitive rendering

Once the structure grid is constructed, we draw primitives (stipple dots and hatching lines). Attributes such as position, dot size, and line width are determined by combining the values in the structure grid with local tone of the input image. We obtain the tone map T from the grayscale version of the input image, which is then Gaussian smoothed to reduce noise and avoid abrupt tone changes.

6.1. Stipple dots

In directional stippling, the tone is described by varying the dot size. A straightforward approach would be to draw explicit circles at the pixels with value pair (0,0) in the

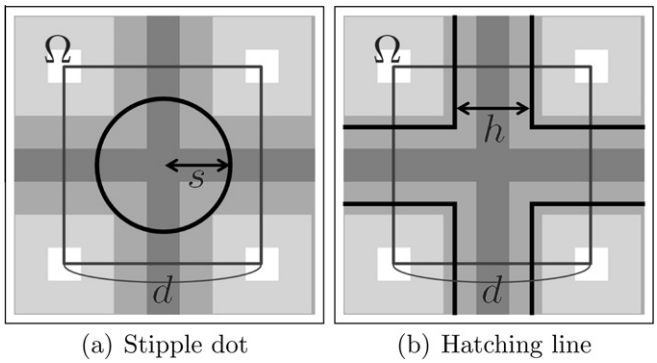


Fig. 8. Primitive size determination.

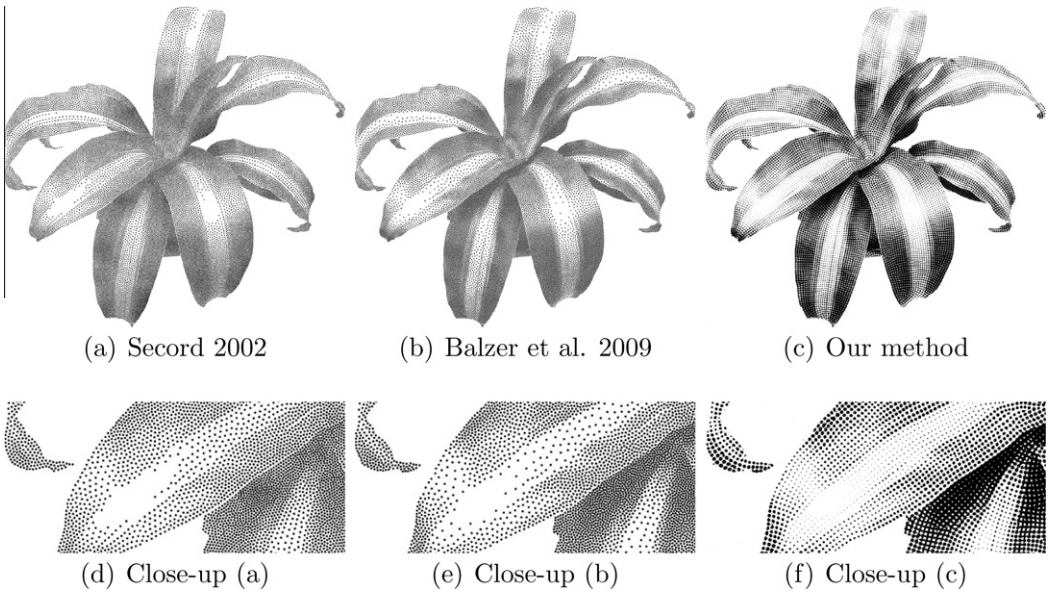


Fig. 9. Comparison with conventional stippling.

structure grid. However, it is known that using perfect circles as dots could easily produce overly mechanical look (Fig. 7a). Besides, in this case a slight imperfection in the dot distribution can be easily magnified. Professional illustrators thus often use somewhat irregular-shaped dots to create a more natural and comfortable look (Fig. 7b).

To this end, we take an implicit approach, and develop a pixel-based dot rendering algorithm. For each pixel, we determine its intensity by computing its probability to be included in the nearest dot. To compute this probability, we need to know two things: the size of the nearest dot and the distance to the dot center.

The distance d_s from pixel p to the nearest dot center can be easily obtained as $d_s = \sqrt{t_0^2 + t_1^2}$. The dot size is inversely proportional to the local tone. Let b denote the average tone in the $d \times d$ region Ω around the nearest dot center. For ease of computation, we approximate b by $T(p)$, which is justified somewhat by the Gaussian blurring of the tone map. Now we estimate the size of a black dot which is needed to reproduce the given tone b in the region Ω (Fig. 8a). The dot radius, denoted by s, should satisfy

$$1 - b = \frac{\pi s^2}{d^2}, \quad \text{that is,} \quad s = d\sqrt{\frac{1-b}{\pi}},$$

assuming the intensity ranges in $[0,1]$. If the radius is larger than $d/2$, neighboring dots overlap. In this case, we should adjust the radius computation to compensate the overlapping of stipples. See Appendix B for more detail.

Now that we have estimated both the dot size s and the distance d_s, the pixel intensity at p is determined as follows. We assign zero (black) if d_s is less than $s - \delta_s$, where δ_s is a user-specified parameter to control the width of the grayscale band around the dot boundary. This band is for antialiasing. If d_s is greater than s, the intensity becomes

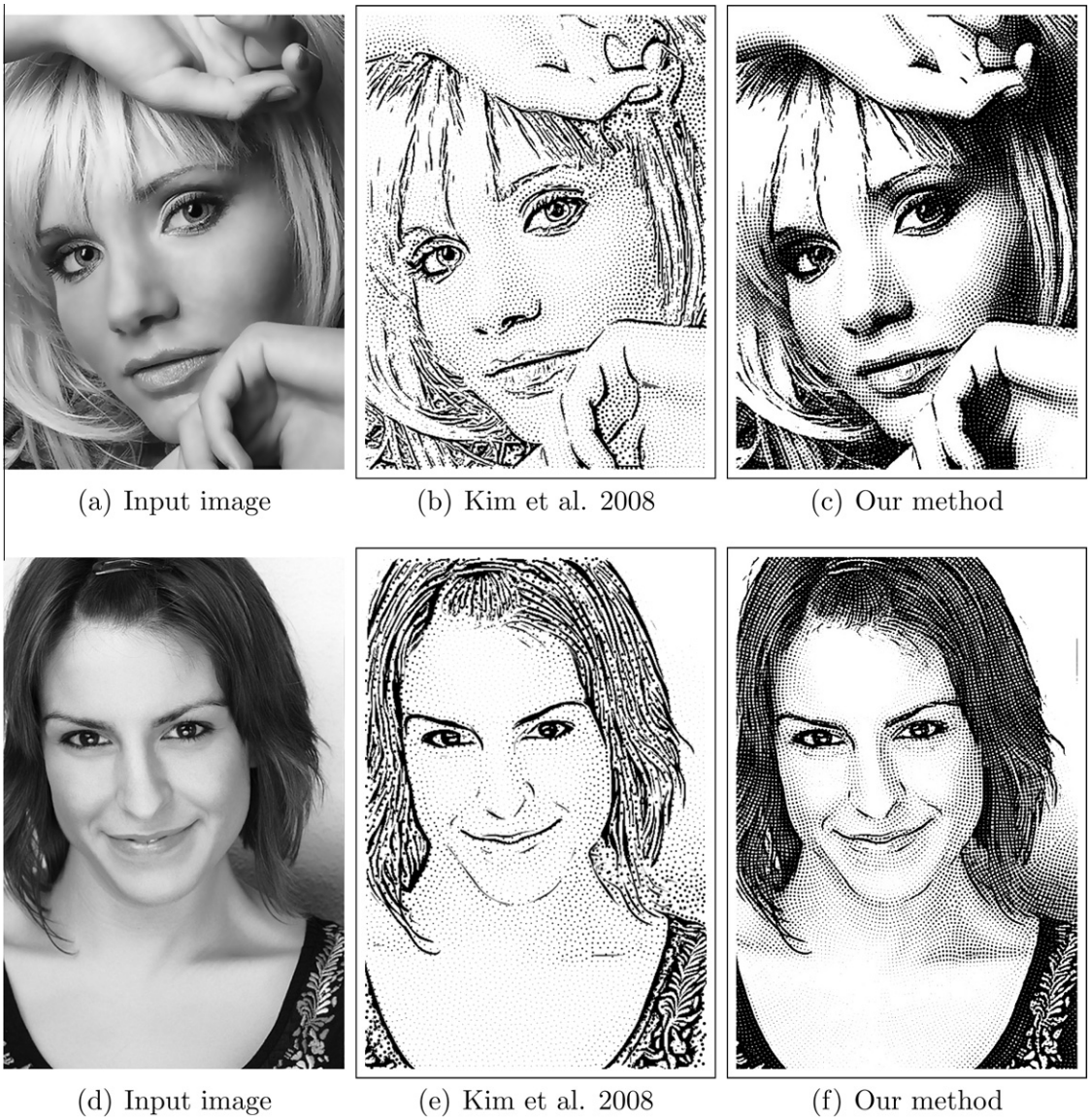


Fig. 10. Comparison with directional stippling.

one (white). Otherwise, the pixel p gets an interpolated grayscale value, $1 - (s - d_s)/\delta_s$. We used $\delta_s = 1$ (one pixel) for all examples.

Our approach allows for pixel-based dot rendering where the calculation of intensity is performed independently of the neighboring pixels. This leads to a simple and efficient GPU implementation. Our pixel-based dot rendering generates less-than-perfect circular dots, which is intended as described above. Also, the use of average tone value in the local window in computing the dot size leads to faithful reproduction of the overall tone.

6.2. Hatching lines

Hatching is conducted by placing lines through the pixels with value $t_0 = 0$ or $t_1 = 0$ in the structure grid. Again we control the line width to reflect the local tone. Like stippling, we process each pixel independently.

We determine the intensity of a pixel p by estimating the probability of p to be included in the nearest hatching line. Let Ω be the local region of size $d \times d$ centered at the grid intersection point (with value pair $(0,0)$) nearest to p. When a line of width h is drawn in the region Ω, the white pixels (no line) occupy area of $(d-h)^2$ (Fig. 8b). To preserve the average tone b in Ω, the width h should satisfy:

$$b = \frac{(d-h)^2}{d^2}, \quad \text{that is,} \quad h = d(1 - \sqrt{b}).$$

The distance d_h from p to the nearest hatching line in Ω is obtained by $d_h = \min\{t_0, t_1\}$. If d_h is less than $h - \delta_h$, the intensity of p is zero. Similarly to stippling, δ_h is a user-specified parameter to control the width of the grayscale band around the hatching line boundary, and $\delta_h = 1$ in our experiments. If d_h is greater than h, the intensity is one. Otherwise, the intensity is interpolated.

As with stippling, our pixel-based hatching line rendering method has similar benefits. The implementation is GPU-friendly, the shapes of hatching lines have natural variations with antialiasing, and the overall tone is well preserved.

6.3. Style mixing

As pointed out in Section 1, the stippling-only policy may lead to trouble in dark area due to the possibility of dot clustering. Reducing dot size or dot density is not a good solution either as we would then not be able to describe the dark tone effectively. To avoid this trouble, hedcut artists often use hatching to describe dark tone.

In general, such style mixing is a challenging task as it could easily generate visual artifacts around the border of the opposing styles. The introduction of our structure grid, however, naturally resolves this problem as it allows to handle both stippling and hatching in a unified manner, based on the same underlying grid. This ensures seamless blending of the two styles, as in professional hedcut illustrations.

In practice, we perform hatching if the computed dot size s is bigger than the user-specified maximum s_{max}. We can easily control the mixing style by changing s_{max}. With $s_{max} = 0$ we obtain pure hatching, while a large s_{max} would produce pure stippling.

6.4. Dot shape enhancement

The deformation of the regular grid inevitably disrupts the uniform spacing of grid intersections in a structure grid G. This could also lead to excessive deformation of dot shape, especially near singularities where the flow degenerates. To alleviate this problem, we may optionally revise the values t_0 and t_1 around the grid intersections.

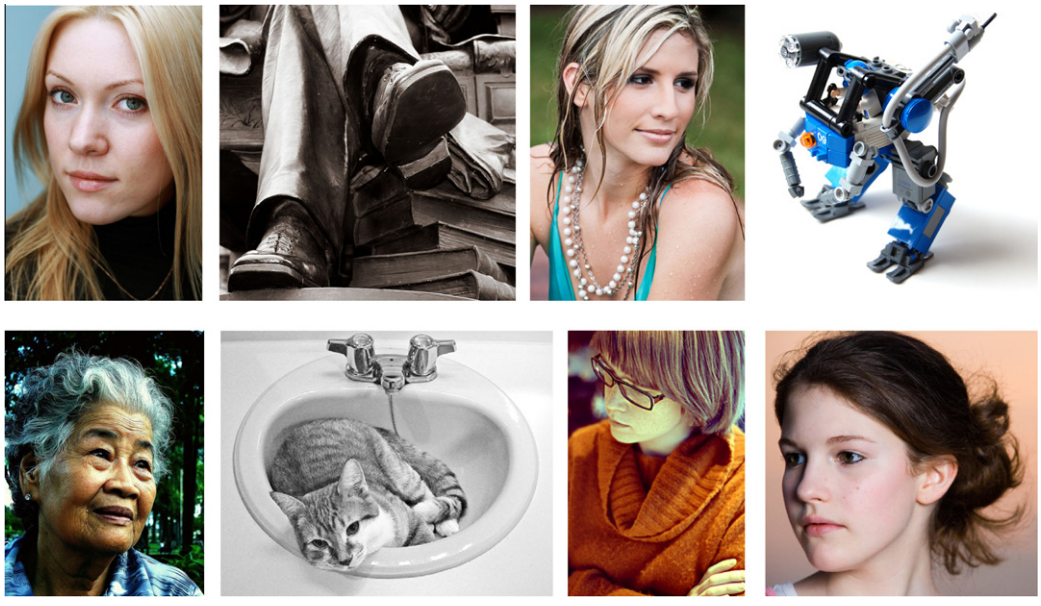


Fig. 11. Input images.

From the set of grid intersections (the local minima of $\sqrt{t_0^2 + t_1^2}$, practically), we compute the Voronoi diagram. Each pixel is then assigned new t_0 and t_1 by measuring its distances from the center of its Voronoi cell along the tangential and normal direction, respectively. Let G' denote this modified structure grid. To resolve the discontinuity of values t_0 and t_1 in G' near the Voronoi cell boundaries, we generate G'' by applying our stripe pattern synthesis separately to the t_0 and t_1 values in G'. We finalize the structure grid using the weighted average, $wG'(p) + (1 - w)G''(p)$, where the weight w at pixel p is inversely proportional to the distance from the Voronoi cell center of p. We use jump-flooding algorithm [28] to compute the Voronoi cells and the distances from the Voronoi centers.

7. Experimental results

7.1. Comparisons

Fig. 9 shows comparisons with existing stipple drawing methods. Note that in Fig. 9a and b, the dot density changes according to the tone, while locally keeping the uniformity of distribution. In contrast, our approach performs directional stippling that keeps the dot density fixed while shaping the distribution along the feature

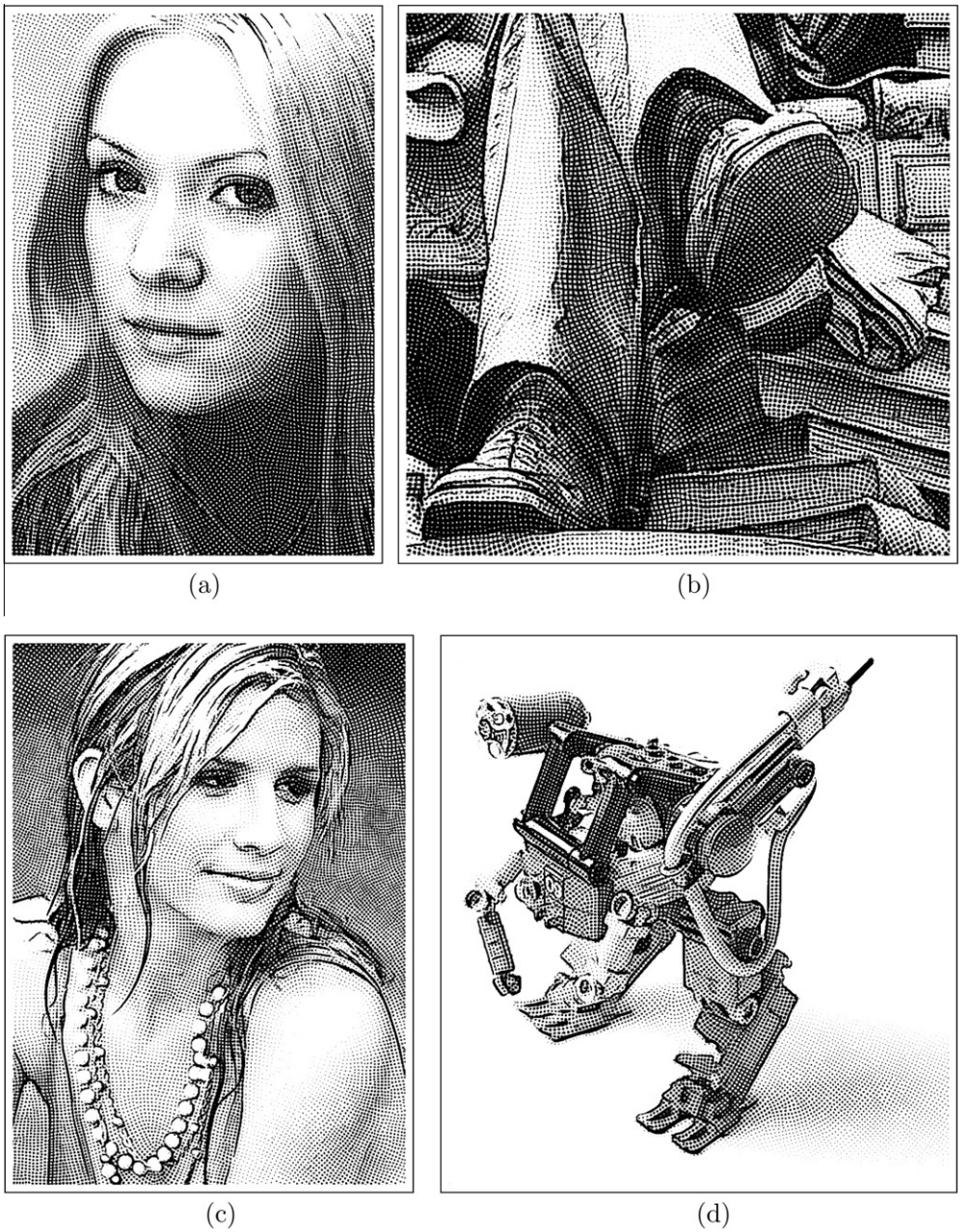


Fig. 12. Hedcut illustration results.

directions. Also, notice the effective description of dark tone with the substitution of hatching lines. These two traits of our approach, that is, the directional shaping of dots and the effective handling of dark tone, helps clarify the individual leaves of the plant. While Fig. 9a and b mainly focus on faithful tone reproduction, our method performs well in conveying features and avoiding dot clumping.

Fig. 10 shows a comparison with the state-of-the-art directional stippling method [3]. Our method outperforms in terms of aligning dots in both major directions in hedcut style. Also, our method effectively covers a wider range of tone without creating distribution artifacts.

7.2. Illustration results

In our method, all processing steps (with the exception of the multi-level B-spline interpolation) are pixel-based operations, and therefore GPU-friendly. We implemented them on a GPU using pixel shaders. The processing time depends mainly on the image size, primitive spacing (d), and window radius (r) and iteration number (n_{iter}) for stripe pattern synthesis. We tested our system on an Intel(R) Core(TM) i7 CPU with an nVIDIA GeForce GTX 285 Graphics card. For a 450×600 image with parameters $d = r = 6$ and $n_{iter} = 5$, the entire process takes about 1.0 s. In the case using $d = r = 8$, it takes about 1.5 s.

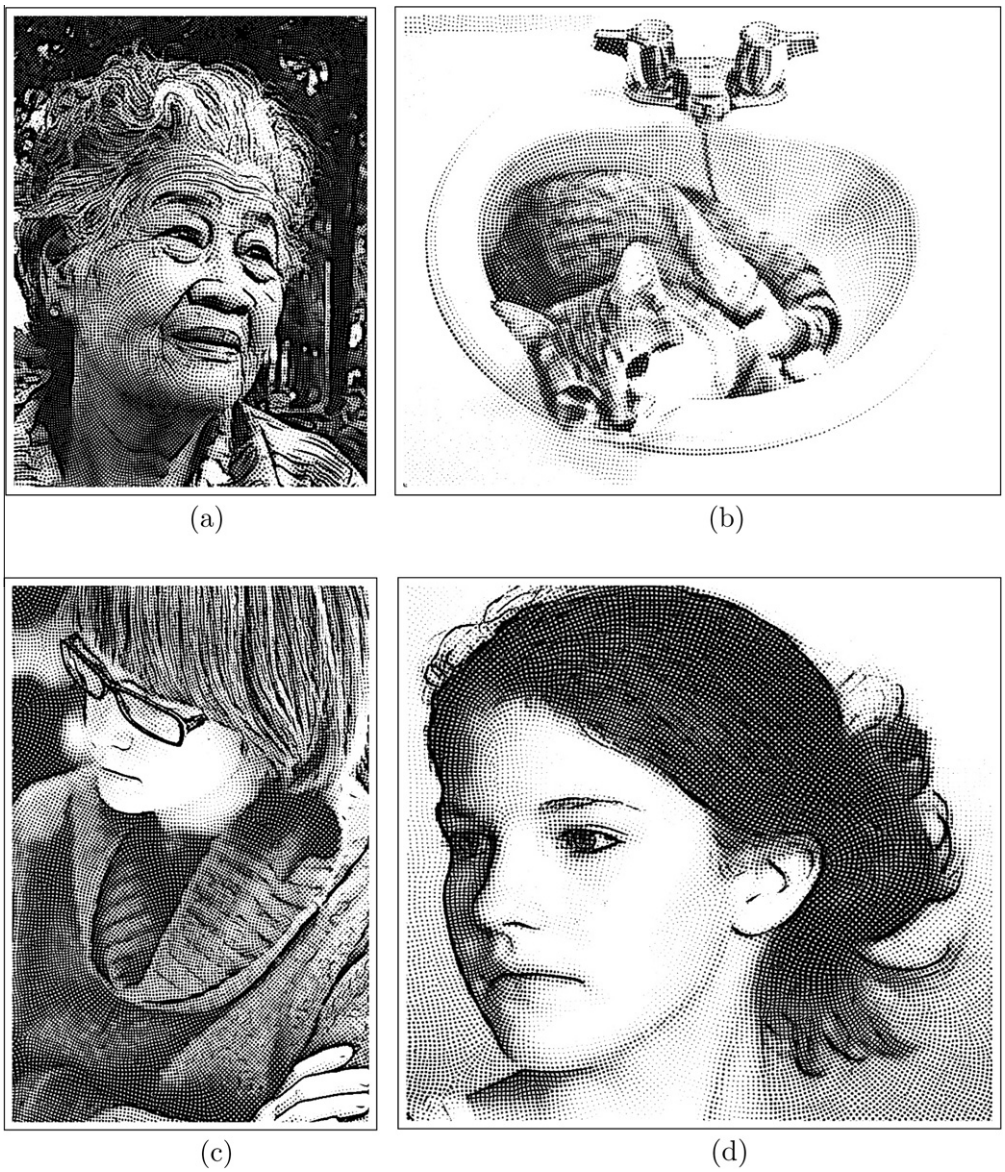


Fig. 13. Hedcut illustration results.

(a) Input image (b) Stippling by the proposed method (c) Mosaics by the proposed method

Fig. 14. Stained glass to stippling and tile mosaics.

Figs. 12 and 13 show hedcut illustrations generated by our system using the input images in Fig. 11. We fix both the spacing d and the window radius r to six, the number of iteration n_{iter} five, and the maximum stipple size s_{max} (for style mixing) 3.0. The number of quantization level to add isophote curves for feature vector field is set to zero for Fig. 12b and c, and three for others. No sophisticated parameter tweaking has been needed to produce consistent results from various input images.

We can additionally control the scale factor of the window W in stripe pattern synthesis. Fig. 12a, b, d and Fig. 13a show the illustration results when the scale factors are changed according to the tone values of pixels. Generally, we use a smaller scale factor for a darker tone. We can then reproduce dark tones effectively without using too large primitives due to the increased density of primitives.

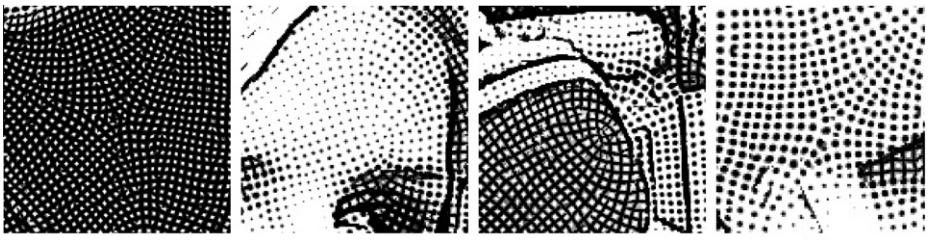
7.3. Tile mosaics

Structure grid can serve as a general framework for image stylization, where the primitives are arranged to follow the feature flow with some spacing constraint. For instance, it can be used to generate tile mosaics.

Given the Voronoi diagram used in the dot shape enhancement process (Section 6.4), we draw a square for each center of the Voronoi cell, with the orientation from the feature flow and the color from the input image. Let N denote the number of pixels in a Voronoi cell. Then the side length of the square is determined as $\sqrt{N} - s$, where s is the interval between neighboring squares. In Fig. 14, the same structure grid obtained from an input stained glass image was used to generate the hedcut illustration and tile mosaics.

8. Discussion and future work

We have addressed the problem of directional stippling, which appears to have been somewhat overlooked in the area of stippling. In particular, we have presented a novel approach to computerized hedcut illustration, based on the notion of a structure grid as a guiding tool for directional stippling. Compared to the state-of-the-art hedcut illustration method [3], our approach ensures more rigorous alignment of primitives, and supports effective description of dark tone based on seamless mixing of stippling and hatching, which are the key elements of professional hedcut style. We also showed that the usefulness of structure grid goes beyond directional stippling. It could be used to assist other NPR styles, such as mosaics, engraving, woodcuts, painting, and pen illustration, where the quality of output is dictated by placement, arrangement, and spacing of primitives.



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

Fig. 15. Unwanted patterns or singularities in our results. (a)–(d) are captured from Fig. 12a–c, and Fig. 13d, respectively.

Some limitations remain. Hedcut artists typically shape the flow of dots in such a way that it conforms to the true structure of the original 3D surface, while avoiding any unnatural patterns or singularities in the flow. Our automatic flow construction algorithm, however, may leave some unwanted patterns or singularities in the field. Although the use of isophotes and scattered orientation interpolation helps reduce these artifacts, there could still be some persistent ones (see Fig. 15). In a way this is inevitable unless we have the full knowledge on the original 3D shape of the subject, which however is unavailable in ordinary 2D images. Employing a sophisticated interactive technique such as [29] to restructure the flow may help in this regard. Nevertheless, let us point out that our system generally produces quite convincing stipple illustrations fully automatically without any user intervention.

Possible future research directions include extension to the hedcut-style video illustration, which would require a temporally coherent sequence of structure grids. Such image-based feature-following grid, we believe, may be useful in other applications, such as stroke-based rendering, flow visualization, image vectorization, and other flow-based stylization methods. In particular, the look of the structure grid is reminiscent of streamline-based flow visualization, which calls for its further investigation in the context of visualization.

Acknowledgments

We thank Thomas Hawk (thomashawk.com) for permission to use the picture in Fig. 14 and Flickr users for putting their images under a creative commons license. This work was supported by the new faculty research fund of Ajou University, the Brain Korea 21 Project in 2010, the Industrial Strategic Technology Development Program of MKE/MCST/KEIT (KI001820, Development of Computational Photography Technologies for Image and Video Contents), the Basic Science Research Program of MEST/NRF (2010-0019523), and the Engineering Research Center of Excellence Program of MEST/NRF (2010-0001721).

Appendix A. Complexity of optimal texture value computation

If we take the 2D texture synthesis approach to update the value pair (t_0, t_1), we should directly match the $(2r+1)\times(2r+1)$ window W with the 2D version of the periodic reflection function S, which is denoted by

$$S^2(x,y) = S^2(x,y+d) = S^2(x+d,y) = S^2(x+d,y+d) = (\min(x,d-x), \min(y,d-y)).$$

The template array $s_x[i]$ should be expanded to a vector-valued 2D array $s_{xy}[i][j]$,

$$s_{xy}[i][j] = S^2(i+x, j+y),$$

where $0 \leqslant x, y < d$ and $i,j = -r, \ldots, r$. We then find (x,y) that minimizes

$$\sum_{i=-r}^{r}\sum_{j=-r}^{r} \|s_{xy}[i][j] - W[i][j]\|^2.$$

Consequently, the total number of operations is $O(d^2(2r+1)^2)$. In our case, however, we use a 1D array w of size $(2r+1)$ for matching window W with function S, and optimize a 1D cost $\sum_{i=-r}^{r}(s_x[i]-w[i])^2$. Even though we synthesize two stripe patterns, the total number of operations is $2\cdot O(d\,(2r+1))$.

Appendix B. Overlapping dots

Given a desired spacing d, if the radius of a dot is larger than $d/2$, the dot does not entirely fit in the $d\times d$ window (Fig. B.16) and therefore overlapping of dots occurs. In this case, we consider only the area inside the window when computing the dot radius. For ease of computation, we approximate the area with the cross-shaped region plus the four quadrants, as shown in Fig. B.16. With this, the equation to compute the dot radius s becomes:

$$1-b = \frac{4k(d-k)+\pi\left(\frac{d}{2}-k\right)^2}{d^2}$$

$$s = \sqrt{k^2+\frac{d^2}{2}}, \quad k = \frac{d}{2} - d\sqrt{\frac{b}{4-\pi}}$$

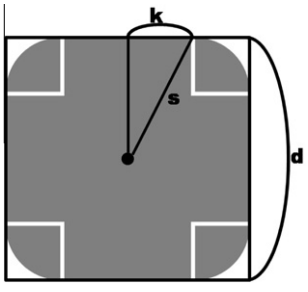


Fig. B.16. Approximating area of an overlapping dot.

References

[1] W.-M. Pang, Y. Qu, T.-T. Wong, D. Cohen-Or, P.-A. Heng, Structure-aware halftoning, ACM Transactions on Graphics (Proc. SIGGRAPH 2008) 27 (2008) 89:1–89:8.
[2] J. Chang, B. Alain, V. Ostromoukhov, Structure-aware error diffusion, ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2009) 28 (2009) 162:1–162:8.
[3] D. Kim, M. Son, Y. Lee, H. Kang, S. Lee, Feature-guided image stippling, Computer Graphics Forum 27 (2008) 1209–1216.
[4] S. Kim, I. Woo, R. Maciejewski, D.S. Ebert, Automated hedcut illustration using isophotes, in: Proceedings of Smart Graphics 2010, LNCS, vol. 6133, 2010, pp. 172–183.
[5] Wiki, 2010, <http://en.wikipedia.org/wiki/Hedcut>.
[6] O. Deussen, S. Hiller, K.V. Overveld, T. Strothotte, Floating points: a method for computing stipple drawings, Computer Graphics Forum 19 (2000) 40–51.
[7] A. Secord, Weighted voronoi stippling, in: Proceedings of the 2nd Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2002), ACM, 2002, pp. 37–43.
[8] J. Kopf, D. Cohen-Or, O. Deussen, D. Lischinski, Recursive wang tiles for real-time blue noise, ACM Transactions on Graphics (Proc. SIGGRAPH 2006) 25 (2006) 509–518.
[9] V. Ostromoukhov, Sampling with polyominoes, ACM Transactions on Graphics (Proc. SIGGRAPH 2007) 26 (2007) 78.
[10] M. Balzer, T. Schlömer, O. Deussen, Capacity-constrained point distributions: a variant of lloyd's method, ACM Transactions on Graphics (Proc. SIGGRAPH 2009) (2009) 86.
[11] A. Hausner, Simulating decorative mosaic, in: Proceedings of SIGGRAPH 2001, ACM, 2001, pp. 573–578.

[12] M.F. Cohen, J. Shade, S. Hiller, O. Deussen, Wang tiles for image and texture generation, ACM Transactions on Graphics (Proc. SIGGRAPH 2003) 22 (2003) 287–294.
[13] V. Ostromoukhov, C. Donohue, P.-M. Jodoin, Fast hierarchical importance sampling with blue noise properties, ACM Transactions on Graphics (Proc. SIGGRAPH 2004) 23 (2004) 488–495.
[14] D. Mould, Stipple placement using distance in a weighted graph, in: Proceedings of Computational Aesthetics, A.K. Peters, 2007, pp. 45–52.
[15] R. Maciejewski, T. Isenberg, W.M. Andrews, D.S. Ebert, M.C. Sousa, W. Chen, Measuring stipple aesthetics in hand-drawn and computer-generated images, IEEE Computer Graphics and Applications (2008) 62–74.
[16] S. Kim, R. Maciejewski, T. Isenberg, W. Andrews, W. Chen, M.C. Sousa, D. Ebert, Stippling by example, in: Proceedings of the 7th Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2009), ACM, 2009, pp. 41–50.
[17] D. Martin, G. Arroyo, M.V. Luzon, T. Isenberg, Example-based stippling using a scale-dependent grayscale process, in: Proceedings of the 8th Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2010), ACM, 2010, pp. 51–61.
[18] J. Hays, I. Essa, Image and video based painterly animation, in: Proceedings of the 3rd Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2004), ACM, 2004, pp. 113–120.
[19] M. Salisbury, M. Wong, J. Hughes, D. Salesin, Orientable textures for image-based pen-and-ink illustration, in: Proceedings of SIGGRAPH '97, ACM, 1997, pp. 401–406.
[20] A. Hertzmann, D. Zorin, Illustrating smooth surfaces, in: Proceedings of SIGGRAPH 2000, 2000, pp. 517–526.
[21] V. Ostromoukhov, Digital facial engraving, in: Proceedings of SIGGRAPH '99, ACM, 1999, pp. 417–424.
[22] H. Kang, S. Lee, C.K. Chui, Coherent line drawing, in: Proceedings of the 5th Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2007), ACM, 2007, pp. 43–50.
[23] A. Girshick, V. Interrante, S. Haker, T. Lemoine, Line direction matters: an argument for the use of principal directions in 3D line drawings, in: Proceedings of the 1st Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2000), ACM, 2000, pp. 43–52.
[24] T. Goodwin, I. Vollick, A. Hertzmann, Isophote distance: a shading approach to artistic stroke thickness, in: Proceedings of the 5th Symposium on Non-Photorealistic Animation and Rendering (Proc. NPAR 2007), ACM, 2007, pp. 53–62.
[25] S. Lee, G. Wolberg, S.Y. Shin, Scattered data interpolation with multilevel b-splines, IEEE Transactions on Visualization and Computer Graphics 3 (1997) 228–244.
[26] J.E. Kyprianidis, J. Döllner, Image abstraction by structure adaptive filtering, in: Proceedings of the EG UK Theory and Practice of Computer Graphics, Eurographics, 2008, pp. 51–58.
[27] J. Zhang, K. Zhou, L. Velho, B. Guo, H.-Y. Shum, Synthesis of progressively variant textures on arbitrary surfaces, ACM Transactions on Graphics (Proc. SIGGRAPH 2003) 22 (2003) 295–302.
[28] G. Rong, T. Tan, Jump flooding in GPU with applications to Voronoi diagram and distance transform, in: Proceedings of Symposium on Interactive 3D Graphics and Games, ACM, 2006, pp. 109–116.
[29] E. Zhang, J. Hays, G. Turk, Interactive tensor field design and visualization on surfaces, IEEE Transactions on Visualization and Computer Graphics 13 (2007) 94–107.

Minjung Son received the BS and MS degrees in computer science and engineering from Pohang University of Science and Technology (POSTECH). She is now a PhD student at POSTECH. Her research interests include computer graphics, non-photorealistic rendering, and image and video stylization.

Yunjin Lee received her BS degree in 1999 and her PhD degree in 2005, all in Computer Science and Engineering from POSTECH in Korea. She is currently an assistant professor in the Division of Digital Media at Ajou University. Her research interests include non-photorealistic rendering, 3D mesh processing, and data compression.

Henry Kang is an associate professor of computer science at the University of Missouri, St. Louis. He received his BS degree in computer science from Yonsei University, Korea (1994), and the MS (1996) and PhD (2002) degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST). His research interests include non-photorealistic rendering and animation, illustrative visualization, and image/video manipulation.

Seungyong Lee received the BS degree in computer science and statistics from Seoul National University in 1988 and the MS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1990 and 1995, respectively. He is now a professor of computer science and engineering at Pohang University of Science and Technology (POSTECH), Korea. From 1995 to 1996, he worked at the City College of New York as a postdoctoral research associate. Since 1996, he has been a faculty member and leading the Computer Graphics Group at POSTECH. From 2003 to 2004, he spent a sabbatical year at MPI Informatik, Germany, as a visiting senior researcher. His current research interests include image and video processing, non-photorealistic rendering, 3D mesh processing, 3D surface reconstruction, and graphics applications.