Learning Hatching for Pen-and-Ink Illustration of Surfaces

EVANGELOS KALOGERAKIS
University of Toronto and Stanford University
and
DEREK NOWROUZEZAHRAI
University of Toronto, Disney Research Zurich, and University of Montreal
and
SIMON BRESLAV
University of Toronto and Autodesk Research
and
AARON HERTZMANN
University of Toronto

This paper presents an algorithm for learning hatching styles from line drawings. An artist draws a single hatching illustration of a 3D object. Their strokes are analyzed to extract the following per-pixel properties: hatching level (hatching, cross-hatching, or no strokes), stroke orientation, spacing, intensity, length, and thickness. A mapping is learned from input geometric, contextual and shading features of the 3D object to these hatching properties, using classification, regression, and clustering techniques. Then, a new illustration can be generated in the artist's style, as follows. First, given a new view of a 3D object, the learned mapping is applied to synthesize target stroke properties for each pixel. A new illustration is then generated by synthesizing hatching strokes according to the target properties.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Line and curve generation*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Geometric algorithms, languages, and systems*; I.2.6 [**Artificial Intelligence**]: Learning—*Parameter learning*

General Terms: Algorithms

Additional Key Words and Phrases: learning surface hatching, data-driven hatching, hatching by example, illustrations by example, learning orientation fields

ACM Reference Format:

Kalogerakis E., Nowrouzezahrai D., Breslav S., Hertzmann A. 2010. Learning Hatching for Pen-and-Ink Illustration of Surfaces ACM Trans. Graph. VV, N, Article XXX (Month YYYY), 18 pages.
DOI = 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

We thank Seok-Hyung Bae, Patrick Coleman, Vikramaditya Dasgupta, Mark Hazen, Thomas Hendry, and Olga Vesselova for creating the hatched drawings. We thank Olga Veksler for the graph cut code and Robert Kalnins, Philip Davidson, and David Bourguignon for the jot code. We thank Aim@Shape, VAKHUN, and Cyberware repositories as well as Xiaobai Chen, Aleksey Golovinskiy, Thomas Funkhouser, Andrea Tagliasacchi and Richard Zhang for the 3D models used in this paper. This project was funded by NSERC, CIFAR, CFI, the Ontario MRI, and KAUST Global Collaborative Research.

{kalo, derek, breslav, hertzman}@dgp.toronto.edu

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

1. INTRODUCTION

Non-photorealistic rendering algorithms can create effective illustrations and appealing artistic imagery. To date, these algorithms are designed using insight and intuition. Designing new styles remains extremely challenging: there are many types of imagery that we do not know how to describe algorithmically. Algorithm design is not a suitable interface for an artist or designer. In contrast, an example-based approach can decrease the artist's workload, when it captures his style from his provided examples.

This paper presents a method for learning hatching for pen-and-ink illustration of surfaces. Given a single illustration of a 3D object, drawn by an artist, the algorithm learns a model of the artist's hatching style, and can apply this style to rendering new views or new objects. Hatching and cross-hatching illustrations use many finely-placed strokes to convey tone, shading, texture, and other qualities. Rather than trying to model individual strokes, we focus on *hatching properties* across an illustration: hatching level (hatching, cross-hatching, or no hatching), stroke orientation, spacing, intensity, length, and thickness. Whereas the strokes themselves may be loosely and randomly placed, hatching properties are more stable and predictable. Learning is based on piecewise-smooth mappings from geometric, contextual, and shading features to these hatching properties.

To generate a drawing for a novel view and/or object, a Lambertian-shaded rendering of the view is first generated, along with the se-

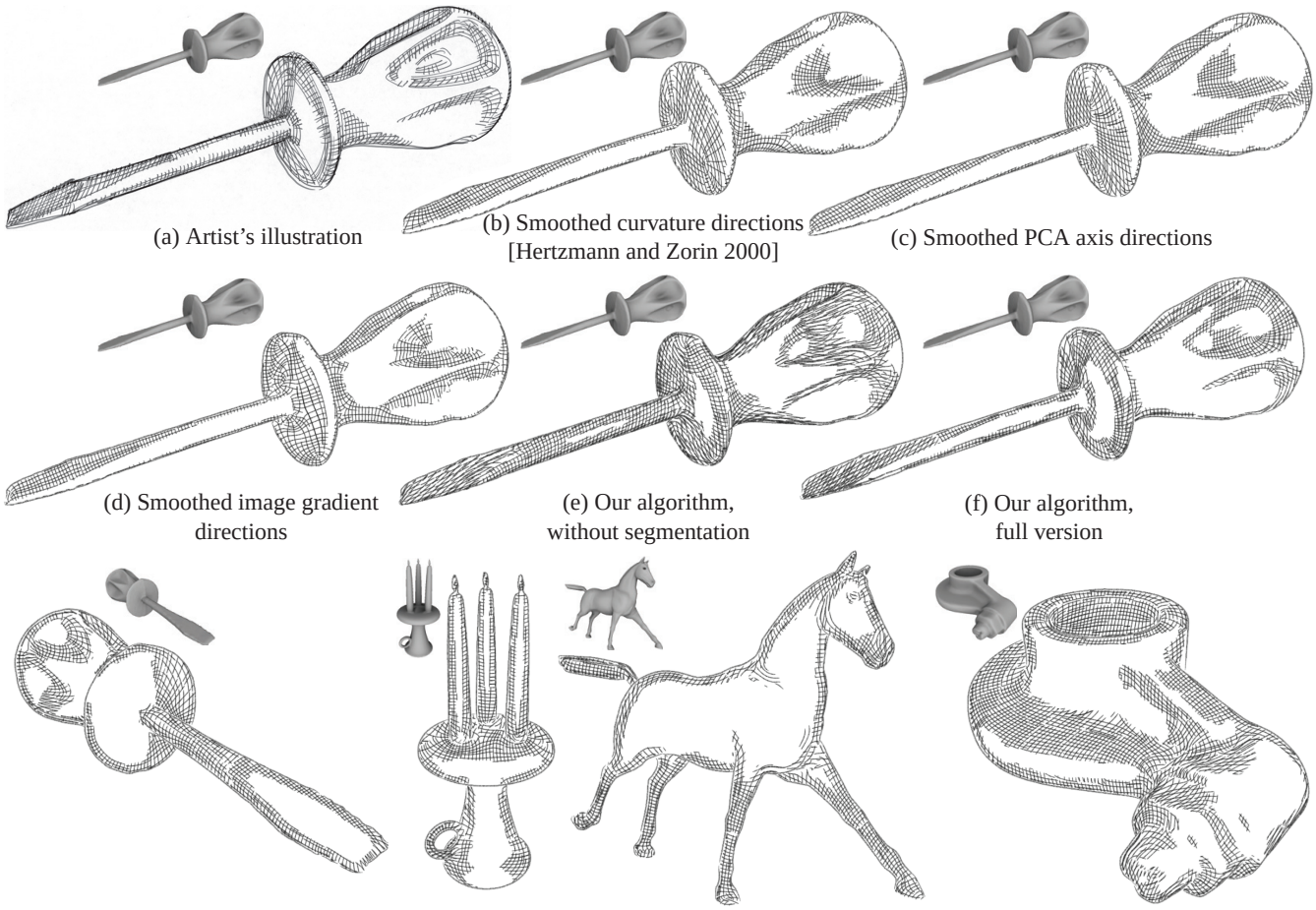

Fig. 1: ***Data-driven line art illustrations generated with our algorithm, and comparisons with alternative approaches. (a)*** *Artist's illustration of a screwdriver.* ***(b)*** *Illustration produced by the algorithm of Hertzmann and Zorin [2000]. Manual thresholding of $\vec{N} \cdot \vec{V}$ is used to match the tone of the hand-drawn illustration and globally-smoothed principal curvature directions are used for the stroke orientations.* ***(c)*** *Illustration produced with the same algorithm, but using local PCA axes for stroke orientations before smoothing.* ***(d)*** *Illustration produced with the same algorithm, but using the gradient of image intensity for stroke orientations.* ***(e)*** *Illustration whose properties are learned by our algorithm for the screwdriver, but without using segmentation (i.e., orientations are learned by fitting a single model to the whole drawing and no contextual features are used for learning the stroke properties).* ***(f)*** *Illustration learned by applying all steps of our algorithm. This result more faithfully matches the style of the input than the other approaches.* ***(g)*** *Results on new views and new objects.*

lected per-pixel features. The learned mappings are applied, in order to compute the desired per-pixel hatching properties. A stroke placement algorithm then places hatching strokes to match these target properties. We demonstrate results where the algorithm generalizes to different views of the training shape and/or different shapes.

Our work focuses on learning hatching properties; we use existing techniques to render feature curves, such as contours, and an existing stroke synthesis procedure. We do not learn properties like randomness, waviness, pentimenti, or stroke texture. Each style is learned from a single example, without performing analysis across a broader corpus of examples. Nonetheless, our method is still able to successfully reproduce many aspects of a specific hatching style even with a single training drawing.

2. RELATED WORK

Previous work has explored various formulas for hatching properties. Saito and Takahashi [1990] introduced hatching based on isoparametric and planar curves. Winkenbach and Salesin [1994; 1996] identify many principles of hand-drawn illustration, and describe methods for rendering polyhedral and smooth objects. Many other analytic formulas for hatching directions have been proposed, including principal curvature directions [Elber 1998; Hertzmann and Zorin 2000; Praun et al. 2001; Kim et al. 2008], isophotes [Kim et al. 2010], shading gradients [Singh and Schaefer 2010], parametric curves [Elber 1998] and user-defined direction fields (e.g., [Palacios and Zhang 2007]). Stroke tone and density are normally proportional to depth, shading, or texture, or else based on user-defined prioritized stroke textures [Praun et al. 2001; Winkenbach

and Salesin 1994; 1996]. In these methods, each hatching property is computed by a hand-picked function of a single feature of shape, shading, or texture (e.g., proportional to depth or curvature). As a result, it is very hard for such approaches to capture the variations evident in artistic hatching styles (Figure 1). We propose the first method to learn hatching of 3D objects from examples.

There have been a few previous methods for transferring properties of artistic rendering by example. Hamel and Strothotte [1999] transfer user-tuned rendering parameters from one 3D object to another. Hertzmann et al. [2001] transfer drawing and painting styles by example using non-parametric synthesis, given image data as input. This method maps directly from the input to stroke pixels. In general, the precise locations of strokes may be highly random—and thus hard to learn—and non-parametric pixel synthesis can make strokes become broken or blurred. Mertens et al. [2006] transfer spatially-varying textures from source to target geometry using non-parametric synthesis. Jodoin et al. [2002] model relative locations of strokes, but not conditioned on a target image or object. Kim et al. [2009] employ texture similarity metrics to transfer stipple features between images. In contrast to the above techniques, our method maps to hatching properties, such as desired tone. Hence, although our method models a narrower range of artistic styles, it can model these styles much more accurately.

A few 2D methods have also been proposed for transferring styles of individual curves [Freeman et al. 2003; Hertzmann et al. 2002; Kalnins et al. 2002] or stroke patterns [Barla et al. 2006], problems which are complementary to ours; such methods could be useful for the rendering step of our method.

A few previous methods use maching learning techniques to extract feature curves, such as contours and silhouettes. Lum and Ma [2005] use neural networks and Support Vector Machines to identify which subset of feature curves match a user sketch on a given drawing. Cole et al. [2008] fit regression models of feature curve locations to a large training set of hand-drawn images. These methods focus on learning locations of feature curves, whereas we focus on hatching. Hatching exhibits substantially greater complexity and randomness than feature curves, since hatches form a network of overlapping curves of varying orientation, thickness, density, and cross-hatching level. Hatching also exhibits significant variation in artistic style.

3. OVERVIEW

Our approach has two main phases. First, we analyze a hand-drawn pen-and-ink illustration of a 3D object, and learn a model of the artist's style that maps from input features of the 3D object to target hatching properties. This model can then be applied to synthesize renderings of new views and new 3D objects. Below we present an overview of the output hatching properties and input features. Then we summarize the steps of our method.

Hatching properties. Our goal is to model the way artists draw hatching strokes in line drawings of 3D objects. The actual placements of individual strokes exhibit much variation and apparent randomness, and so attempting to accurately predict individual strokes would be very difficult. However, we observe that the individual strokes themselves are less important than the overall appearance that they create together. Indeed, art instruction texts often focus on achieving particular qualities such as tone or shading (e.g., [Guptill 1997]). Hence, similar to previous work [Winkenbach and Salesin 1994; Hertzmann and Zorin 2000], we model the rendering process in terms of a set of intermediate *hatching properties* related to tone and orientation. Each pixel containing a stroke in a given illustration is labeled with the following properties:

- **Hatching level ($h \in \{0, 1, 2\}$)** indicates whether a region contains no hatching, single hatching, or cross-hatching.
- **Orientation ($\phi_1 \in [0...\pi]$)** is the stroke direction in image space, with 180-degree symmetry.
- **Cross-hatching orientation ($\phi_2 \in [0..\pi]$)** is the cross-hatch direction, when present. Hatches and cross-hatches are not constrained to be perpendicular.
- **Thickness ($t \in \Re^+$)** is the stroke width.
- **Intensity ($I \in [0..1]$)** is how light or dark the stroke is.
- **Spacing ($d \in \Re^+$)** is the distance between parallel strokes.
- **Length ($l \in \Re^+$)** is the length of the stroke.

The decomposition of an illustration into hatching properties is illustrated in Figure 2 (top). In the analysis process, these properties are estimated from hand-drawn images, and models are learned. During synthesis, the learned model generates these properties as targets for stroke synthesis.

Modeling artists' orientation fields presents special challenges. Previous work has used local geometric rules for determining stroke orientations, such as curvature [Hertzmann and Zorin 2000] or gradient of shading intensity [Singh and Schaefer 2010]. We find that, in many hand-drawn illustrations, no local geometric rule can explain all stroke orientations. For example, in Figure 3, the strokes on the cylindrical part of the screwdriver's shaft can be explained as following the gradient of the shaded rendering, whereas the strokes on the flat end of the handle can be explained by the gradient of ambient occlusion ∇a. Hence, we segment the drawing into regions with distinct rules for stroke orientation. We represent this segmentation by an additional per-pixel variable:

- **Segment label ($c \in \mathcal{C}$)** is a discrete assignment of the pixel to one of a fixed set of possible segment labels $\mathcal{C}$.

Each set of pixels with a given label will use a single rule to compute stroke orientations. For example, pixels with label c_1 might use principal curvature orientations, and those with c_2 might use a linear combination of isophote directions and local PCA axes. Our algorithm also uses the labels to create contextual features (Section 5.2), which are also taken into account for computing the rest of the hatching properties. For example, pixels with label c_1 may have thicker strokes.

Features. For a given 3D object and view, we define a set of features containing geometric, shading, and contextual information for each pixel, as described in Appendices B and C. There are two types of features: "scalar" features $\mathbf{x}$ (Appendix B) and "orientation" features $\boldsymbol{\theta}$ (Appendix C). The features include many object-space and image-space properties which may be relevant for hatching, including features that have been used by previous authors for feature curve extraction, shading, and surface part labeling. The features are also computed at multiple scales, in order to capture varying surface and image detail. These features are inputs to the learning algorithm, which map from features to hatching properties.

Data acquisition and preprocessing. The first step of our process is to gather training data and to preprocess it into features and hatching properties. The training data is based on a single drawing

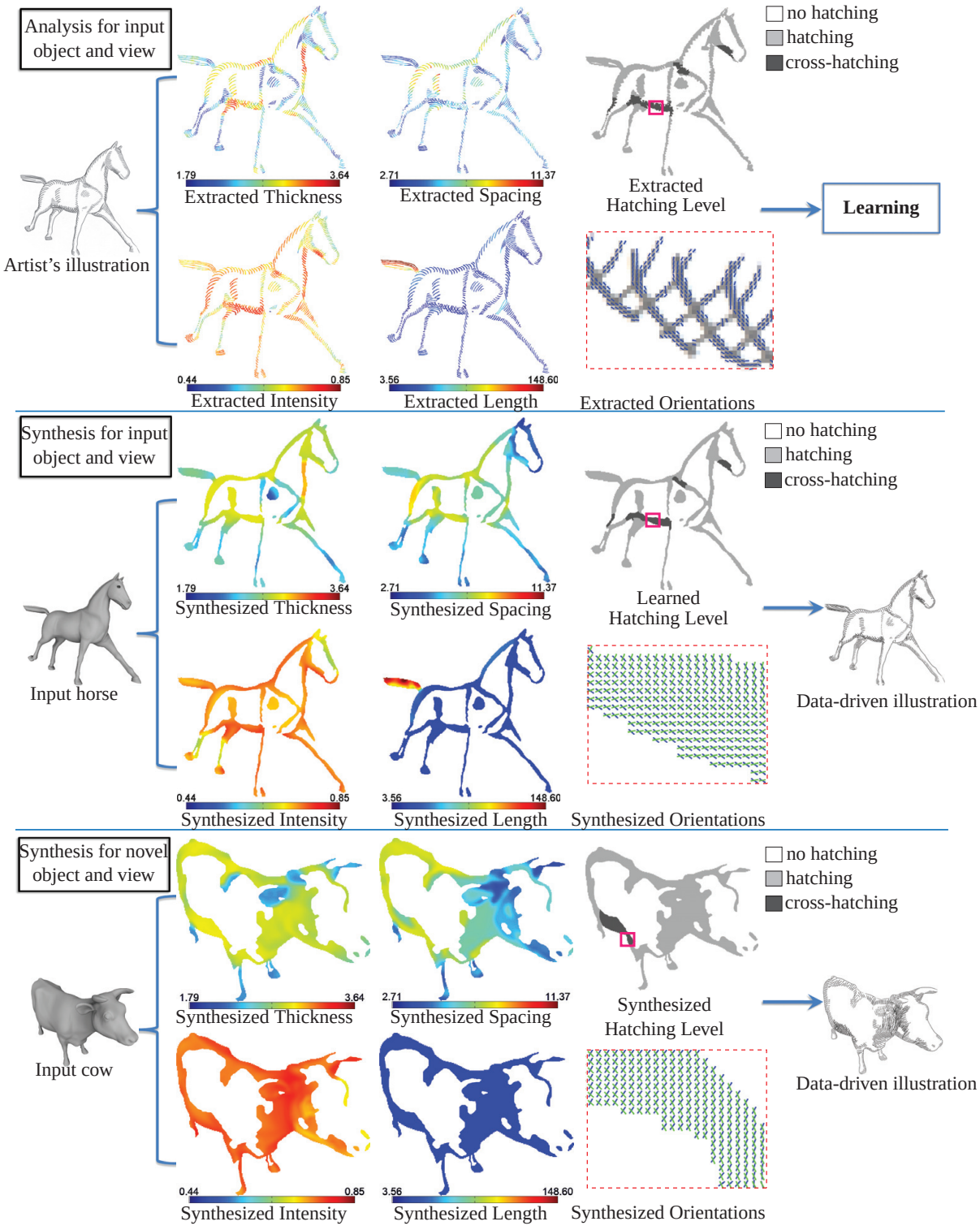

Fig. 2: ***Extraction of hatching properties from a drawing, and synthesis for new drawings. Top:*** *The algorithm decomposes a given artist's illustration into a set of hatching properties: stroke thickness, spacing, hatching level, intensity, length, orientations. A mapping from input geometry is learned for each of these properties.* ***Middle:*** *Synthesis of the hatching properties for the input object and view. Our algorithm automatically separates and learns the hatching (blue-colored field) and cross-hatching fields (green-colored fields).* ***Bottom:*** *Synthesis of the hatching properties for a novel object and view.*

of a 3D model. An artist first chooses an image from our collection of rendered images of 3D objects. The images are rendered with Lambertian reflectance, distant point lighting, and spherical harmonic self-occlusion [Sloan et al. 2002]. Then, the artist creates a line illustration, either by tracing over the illustration on paper with a light table, or in a software drawing package with a tablet. If the illustration is drawn on paper, we scan the illustration and align it to the rendering automatically by matching borders with brute force search. The artist is asked not to draw silhouette and feature curves, or to draw them only in pencil, so that they can be erased. The hatching properties (h, ϕ, t, I, d, l) for each pixel are estimated by the preprocessing procedure described in Appendix A.

Learning. The training data is comprised of a single illustration with features $\mathbf{x}, \boldsymbol{\theta}$ and hatching properties given for each pixel. The algorithm learns mappings from features to hatching properties (Section 5). The segmentation c and orientation properties ϕ are the most challenging to learn, because neither the segmentation c nor the orientation rules are immediately evident in the data; this represents a form of "chicken-and-egg" problem. We address this using a learning and clustering algorithm based on Mixtures-of-Experts (Section 5.1).

Once the input pixels are classified, a pixel classifier is learned using Conditional Random Fields with unary terms based on JointBoost (Section 5.2). Finally, each real-valued property is learned using boosting for regression (Section 5.3). We use boosting techniques for classification and regression since we do not know in advance which input features are the most important for different styles. Boosting can handle a large number of features, can select the most relevant features, and has a fast sequential learning algorithm.

Synthesis. A hatching style is transferred to a target novel view and/or object by first computing the features for each pixel, and then applying the learned mappings to compute the above hatching properties. A streamline synthesis algorithm [Hertzmann and Zorin 2000] then places hatching strokes to match the synthesized properties. Examples of this process are shown in Figure 2.

4. SYNTHESIS ALGORITHM

The algorithm for computing a pen-and-ink illustration of a view of a 3D object is as follows. For each pixel of the target image, the features $\mathbf{x}$ and $\boldsymbol{\theta}$ are first computed (Appendices B and C). The segment label and hatching level are each computed as a function of the scalar features $\mathbf{x}$, using image segmentation and recognition techniques. Given these segments, orientation fields for the target image are computed by interpolation of the orientation features $\boldsymbol{\theta}$. Then, the remaining hatching properties are computed by learning functions of the scalar features. Finally, a streamline synthesis algorithm [Hertzmann and Zorin 2000] renders strokes to match these synthesized properties. A streamline is terminated when it crosses an occlusion boundary, or the length grows past the value of the per-pixel target stroke length l, or violates the target stroke spacing d.

We now describe these steps in more detail. In Section 5, we will describe how the algorithm's parameters are learned.

4.1 Segmentation and labeling

For a given view of a 3D model, the algorithm first segments the image into regions with different orientation rules and levels of hatching. More precisely, given the feature set $\mathbf{x}$ for each pixel, the algorithm computes the per-pixel segment labels $c \in \mathcal{C}$ and hatching level $h \in \{0, 1, 2\}$. There are a few important considerations when choosing an appropriate segmentation and labeling algorithm. First, we do not know in advance which features in $\mathbf{x}$ are important, and so we must use a method that can perform feature selection. Second, neighboring labels are highly correlated, and performing classification on each pixel independently yields noisy results (Figure 3). Hence, we use a Conditional Random Field (CRF) recognition algorithm, with JointBoost unary terms [Kalogerakis et al. 2010; Shotton et al. 2009; Torralba et al. 2007]. One such model is learned for segment labels c, and a second for hatching level h. Learning these models is described in Section 5.2.

The CRF objective function includes unary terms that assess the consistency of pixels with labels, and pairwise terms that assess the consistency between labels of neighboring pixels. Inferring segment labels based on the CRF model corresponds to minimizing the following objective function:

$$E(\mathbf{c}) = \sum_i E_1(c_i; \mathbf{x}_i) + \sum_{i,j} E_2(c_i, c_j; \mathbf{x}_i, \mathbf{x}_j) \qquad (1)$$

where E_1 is the unary term defined for each pixel i, E_2 is the pairwise term defined for each pair of neighboring pixels $\{i, j\}$, where $j \in N(i)$ and $N(i)$ is defined using the 8-neighborhood of pixel i.

The unary term evaluates a JointBoost classifier that, given the feature set $\mathbf{x}_i$ for pixel i, determines the probability $P(c_i|\mathbf{x}_i)$ for each possible label c_i. The unary term is then:

$$E_1(c_i; \mathbf{x}) = -\log P(c_i|\mathbf{x}_i). \qquad (2)$$

The mapping from features to probabilities $P(c_i|\mathbf{x}_i)$ is learned from the training data using the JointBoost algorithm [Torralba et al. 2007].

The pairwise energy term scores the compatibility of adjacent pixel labels c_i and c_j, given their features $\mathbf{x}_i$ and $\mathbf{x}_j$. Let e_i be a binary random variable representing if the pixel i belongs to a boundary of hatching region or not. We define a binary JointBoost classifier that outputs the probability of boundaries of hatching regions $P(e|\mathbf{x})$ and compute the pairwise term as:

$$E_2(c_i, c_j; \mathbf{x}_i, \mathbf{x}_j) = -\ell \cdot I(c_i, c_j) \cdot (\log((P(e_i|\mathbf{x}_i) + P(e_j|\mathbf{x}_j))) + \mu) \qquad (3)$$

where ℓ, μ are the model parameters and $I(c_i, c_j)$ is an indicator function that is 1 when $c_i \neq c_j$ and 0 when $c_i = c_j$. The parameter ℓ controls the importance of the pairwise term while μ contributes to eliminating tiny segments and smoothing boundaries.

Similarly, inferring hatching levels based on the CRF model corresponds to minimizing the following objective function:

$$E(\mathbf{h}) = \sum_i E_1(h_i; \mathbf{x}_i) + \sum_{i,j} E_2(h_i, h_j; \mathbf{x}_i, \mathbf{x}_j) \qquad (4)$$

As above, the unary term evaluates another JointBoost classifier that, given the feature set $\mathbf{x}_i$ for pixel i, determines the probability $P(h_i|\mathbf{x}_i)$ for each hatching level $h \in \{0, 1, 2\}$. The pairwise term is also defined as:

$$E_2(h_i, h_j; \mathbf{x}_i, \mathbf{x}_j) = -\ell \cdot I(h_i, h_j) \cdot (\log((P(e_i|\mathbf{x}_i) + P(e_j|\mathbf{x}_j))) + \mu) \qquad (5)$$

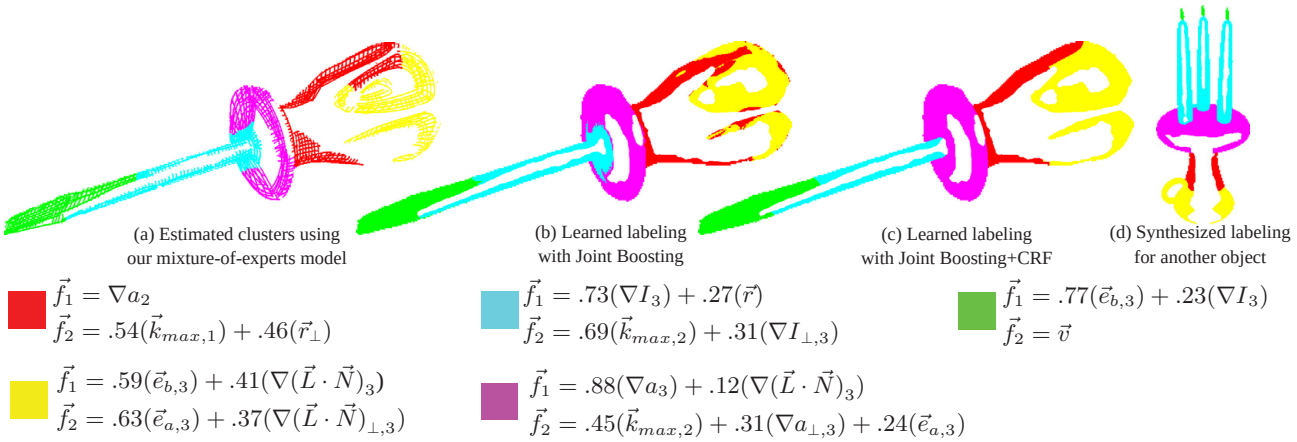

Fig. 3: ***Clustering orientations.*** *The algorithm clusters stroke orientations according to different orientation rules. Each cluster specifies rules for hatching ($\vec{f}_1$) and cross-hatching ($\vec{f}_2$) directions. Cluster labels are color-coded in the figure, with rules shown below. The cluster labels and the orientation rules are estimated simultaneously during learning.* ***(a)*** *Inferred cluster labels for an artist's illustration of a screwdriver.* ***(b)*** *Output of the labeling step using the most likely labels returned by the Joint Boosting classifier alone.* ***(c)*** *Output of the labeling step using our full CRF model.* ***(d)*** *Synthesis of part labels for a novel object.* ***Rules:*** *In the legend, we show the corresponding orientation functions for each region. In all cases, the learned models use one to three features. Subscripts $\{1,2,3\}$ indicates the scale used to compute the field. The $\perp$ operator rotates the field by 90 degrees in image-space. The orientation features used here are: maximum and minimum principal curvature directions ($\vec{k}_{max}$, $\vec{k}_{min}$), PCA directions corresponding to first and second largest eigenvalue ($\vec{e}_a$, $\vec{e}_b$), fields aligned with ridges and valleys respectively ($\vec{r}$, $\vec{v}$), Lambertian image gradient (∇I), gradient of ambient occlusion (∇a), and gradient of $\vec{L}\cdot\vec{N}$ ($\nabla(\vec{L}\cdot\vec{N})$). Features that arise as 3D vectors are projected to the image plane. See Appendix C for details.*

with the same values for the parameters of ℓ, μ as above.

The most probable labeling is the one that minimizes the CRF objective function $E(\mathbf{c})$ and $E(\mathbf{h})$, given their learned parameters. The CRFs are optimized using alpha-expansion graph-cuts [Boykov et al. 2001]. Details of learning the JointBoost classifiers and ℓ, μ are given in Section 5.2.

4.2 Computing orientations

Once the per-pixel segment labels c and hatching levels h are computed, the per-pixel orientations ϕ_1 and ϕ_2 are computed. The number of orientations to be synthesized is determined by h. When $h = 0$ (no hatching), no orientations are produced. When $h = 1$ (single hatching), only ϕ_1 is computed and, when $h = 2$ (cross-hatching), ϕ_2 is also computed.

Orientations are computed by regression on a subset of the orientation features $\boldsymbol{\theta}$ for each pixel. Each cluster c may use a different subset of features. The features used by a segment are indexed by a vector σ, i.e., the features' indices are $\sigma(1)$, $\sigma(2)$, ..., $\sigma(k)$. Each orientation feature represents an orientation field in image space, such as the image projection of principal curvature directions. In order to respect 2-symmetries in orientation, a single orientation θ is transformed to a vector as

$$\mathbf{v} = [\cos(2\theta), \sin(2\theta)]^T \tag{6}$$

The output orientation function is expressed as a weighted sum of selected orientation features.

$$f(\boldsymbol{\theta}; \mathbf{w}) = \sum_k w_{\sigma(k)} \mathbf{v}_{\sigma(k)} \tag{7}$$

where $\sigma(k)$ represents the index to the k-th orientation feature in the subset of selected orientation features, $v_{\sigma(k)}$ is its vector representation, and $\mathbf{w}$ is a vector of weight parameters. There is an orientation function $f(\boldsymbol{\theta}; \mathbf{w}_{c,1})$ for each label $c \in \mathcal{C}$ and, if the class contains cross-hatching regions, it has an additional orientation function $f(\boldsymbol{\theta}; \mathbf{w}_{c,2})$ for determining the cross-hatching directions. The resulting vector is computed to an image-space angle as $\phi = \text{atan2}(y, x)/2$.

The weights $\mathbf{w}$ and feature selection σ are learned by the gradient-based boosting for regression algorithm of Zemel and Pitassi [2001]. The learning of the parameters and the feature selection is described in Section 5.1.

4.3 Computing real-valued properties

The remaining hatching properties are real-valued quantities. Let y be a feature to be synthesized on a pixel with feature set $\mathbf{x}$. We use multiplicative models of the form:

$$y = \prod_k (a_k x_{\sigma(k)} + b_k)^{\alpha_k} \tag{8}$$

where $x_{\sigma(k)}$ is the index to the k-th scalar feature from $\mathbf{x}$. The use of a multiplicative model is inspired by Goodwin et al. [2007], who propose a model for stroke thickness that can be approximated by a product of radial curvature and inverse depth. The model is learned in the logarithmic domain, which reduces the problem to learning the weighted sum:

$$\log(y) = \sum_k \alpha_k \log(a_k x_{\sigma(k)} + b_k) \tag{9}$$

Learning the parameters $\alpha_k, a_k, b_k, \sigma(k)$ is again performed using gradient-based boosting [Zemel and Pitassi 2001], as described in Section 5.3.

5. LEARNING

We now describe how to learn the parameters of the functions used in the synthesis algorithm described in the previous section.

5.1 Learning segmentation and orientation functions

In our model, the hatching orientation for a single-hatching pixel is computed by first assigning the pixel to a cluster c, and then applying the orientation function $f(\boldsymbol{\theta}; \mathbf{w}_c)$ for that cluster. If we knew the clustering in advance, then it would be straightforward to learn the parameters $\mathbf{w}_c$ for each pixel. However, neither the cluster labels nor the parameters $\mathbf{w}_c$ are present in the training data. In order to solve this problem, we develop a technique inspired by Expectation-Maximization for Mixtures-of-Experts [Jordan and Jacobs 1994], but specialized to handle the particular issues of hatching.

The input to this step is a set of pixels from the source illustration with their corresponding orientation feature set $\boldsymbol{\theta}_i$, training orientations ϕ_i, and training hatching levels h_i. Pixels containing intersections of strokes or no strokes are not used. Each cluster c may contain either single-hatching or cross-hatching. Single-hatch clusters have a single orientation function (Equation 7), with unknown parameters $\mathbf{w}_{c1}$. Clusters with cross-hatches have two subclusters, each with an orientation function with unknown parameters $\mathbf{w}_{c1}$ and $\mathbf{w}_{c2}$. The two orientation functions are not constrained to produce directions orthogonal to each other. Every source pixel must belong to one of the top-level clusters, and every pixel belonging to a cross-hatching cluster must belong to one of its subclusters.

For each training pixel i, we define a labeling probability γ_{ic} indicating the probability that pixel i lies in top-level cluster c, such that $\sum_c \gamma_{ic} = 1$. Also, for each top-level cluster, we define a subcluster probability β_{icj}, where $j \in \{1, 2\}$, such that $\beta_{ic1} + \beta_{ic2} = 1$. The probability β_{icj} measures how likely the stroke orientation at pixel i corresponds to a hatching or cross-hatching direction. Single-hatching clusters have $\beta_{ic2} = 0$. The probability that pixel i belongs to the subcluster indexed by $\{c, j\}$ is $\gamma_{ic}\beta_{icj}$.

The labeling probabilities are modeled based on a mixture-of-Gaussians distribution [Bishop 2006]:

$$\gamma_{ic} = \frac{\pi_c \exp(-r_{ic}/2s)}{\sum_c \pi_c \exp(-r_{ic}/2s)} \tag{10}$$

$$\beta_{icj} = \frac{\pi_{cj} \exp(-r_{icj}/2s_c)}{\pi_{c1} \exp(-r_{ic1}/2s_c) + \pi_{c2} \exp(-r_{ic2}/2s_c)} \tag{11}$$

where π_c, π_{cj} are the mixture coefficients, s, s_c are the variances of the corresponding Gaussians, r_{icj} is the residual for pixel i with respect to the orientation function j in cluster c, and r_{ic} is defined as follows:

$$r_{ic} = \min_{j \in \{1,2\}} ||\mathbf{u}_i - f(\boldsymbol{\theta}_i; \mathbf{w}_{cj})||^2 \tag{12}$$

where $\mathbf{u}_i = [\cos(2\phi_i), \sin(2\phi_i)]^T$.

The process begins with an initial set of labels γ, β, and $\mathbf{w}$, and then alternates between updating two steps: the *model update step* where the orientation functions, the mixture coefficients and variances are updated, and the *label update step* where the labeling probabilities are updated.

Model update. Given the labeling, orientation functions for each cluster are updated by minimizing the boosting error function, described in Appendix D, using the initial per-pixel weights $\alpha_i = \gamma_{ic}\beta_{icj}$.

In order to avoid overfitting, a set of holdout-validation pixels are kept for each cluster. This set is found by selecting rectangles of random size and marking their containing pixels as holdout-validation pixels. Our algorithm stops when 25% of the cluster pixels are marked as holdout-validation pixels. The holdout-validation pixels are not considered for fitting the weight vector $\mathbf{w}_{cj}$. At each boosting iteration, our algorithm measures the holdout-validation error measured on these pixels. It terminates the boosting iterations when the holdout-validation error reaches a minimum. This helps avoid overfitting the training orientation data.

During this step, we also update the mixture coefficients and variances of the Gaussians in the mixture model, so that the data likelihood is maximized in this step [Bishop 2006]:

$$\pi_c = \sum_i \gamma_{ic}/N, \quad s = \sum_{ic} \gamma_{ic} r_{ic}/N \tag{13}$$

$$\pi_{cj} = \sum_i \beta_{icj}/N, \quad s_c = \sum_{ij} \beta_{icj} r_{icj}/N \tag{14}$$

where N is the total number of pixels with training orientations.

Label update. Given the estimated orientation functions from the above step, the algorithm computes the residual for each model and each orientation function. Median filtering is applied to the residuals, in order to enforce spatial smoothness: r_{ic} is replaced with the value of the median of r_{*c} in the local image neighborhood of pixel i (with radius equal to the local spacing S_i). Then the pixel labeling probabilities are updated according to Equations 10 and 11.

Initialization. The clustering is initialized using a constrained mean-shift clustering process with a flat kernel, similar to constrained K-means [Wagstaff et al. 2001]. The constraints arise from a region-growing strategy to enforce spatial continuity of the initial clusters. Each cluster grows by considering randomly-selected seed pixels in their neighborhood and adding them only if the difference between their orientation angle and the cluster's current mean orientation is below a threshold. In the case of cross-hatching clusters, the minimum difference between the two mean orientations is used. The threshold is automatically selected once during pre-processing by taking the median of each pixel's local neighborhood orientation angle differences. The process is repeated for new pixels and the cluster's mean orientation(s) are updated at each iteration. Clusters composed of more than 10% cross-hatch pixels are marked as cross-hatching clusters; the rest are marked as single-hatching clusters. The initial assignment of pixels to clusters gives a binary-valued initialization for γ. For cross-hatch pixels, if more than half the pixels in the cluster are assigned to orientation function $\mathbf{w}_{k2}$, our algorithm swaps $\mathbf{w}_{k1}$ and $\mathbf{w}_{k2}$. This ensures that the first hatching direction will correspond to the dominant orientation. This aids in maintaining orientation field consistency between neighboring regions.

An example of the resulting clustering for an artist's illustration of screwdriver is shown in Figure 3 (a). We also include the functions

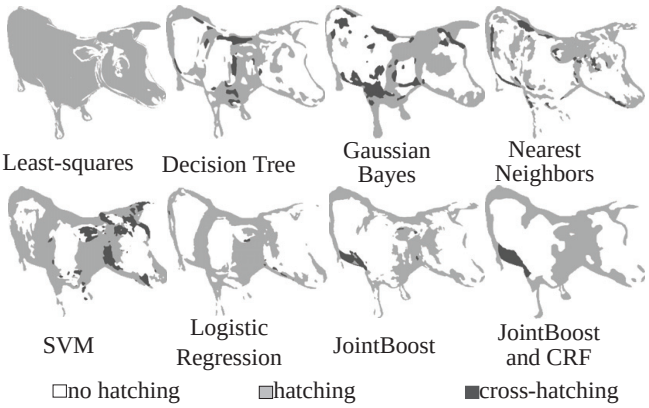

Fig. 4: ***Comparisons of various classifiers for learning the hatching level.*** *The training data is the extracted hatching level on the horse of Figure 2 and feature set* $\mathbf{x}$*.* ***Left to right:*** *least-squares for classification, decision tree (Matlab's implementation based on Gini's diversity index splitting criterion), Gaussian Naive Bayes, Nearest Neighbors, Support Vector Machine, Logistic Regression, Joint Boosting, Joint Boosting and Conditional Random Field (full version of our algorithm). The regularization parameters of SVMs, Gaussian Bayes, Logistic Regression are estimated by hold-out validation with the same procedure as in our algorithm.*

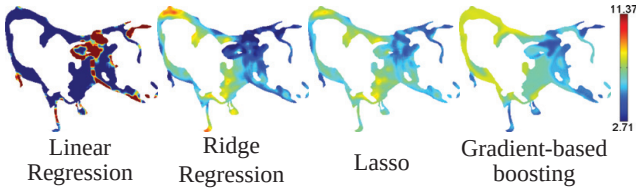

Fig. 5: *Comparisons of the generalization performance of various techniques for regression for the stroke spacing. The same training data are provided to the techniques based on the extracted spacing on the horse of Figure 2 and feature set* $\mathbf{x}$*.* ***Left to right:*** *Linear regression (least-squares without regularization), ridge regression, Lasso, gradient-based boosting. Fitting a model on such very high-dimensional space without any sparsity prior yields very poor generalization performance. Gradient-based boosting gives more reasonable results than ridge regression or Lasso, especially on the legs of the cow, where the predicted spacing values seem to be more consistent with the training values on the legs of the horse (see Figure 2). The regularization parameters of Ridge Regression and Lasso are estimated by hold-out validation with the same procedure as in our algorithm.*

learned for the hatching and cross-hatching orientation fields used in each resulting cluster.

5.2 Learning labeling with CRFs

Once the training labels are estimated, we learn a procedure to transfer them to new views and objects. Here we describe the procedure to learn the Conditional Random Field model of Equation 1 for assigning segment labels to pixels as well as the Conditional Random Field of Equation 4 for assigning hatching levels to pixels.

Learning to segment and label. Our goal here is to learn the parameters of the CRF energy terms (Equation 1). The input is the scalar feature set $\tilde{\mathbf{x}}_i$ for each stroke pixel i (described in Appendix B) and their associated labels c_i, as extracted in the previous step. Following [Tu 2008; Shotton et al. 2008; Kalogerakis et al. 2010], the parameters of the unary term are learned by running a cascade of JointBoost classifiers. The cascade is used to obtain contextual features which capture information about the relative distribution of cluster labels around each pixel. The cascade of classifiers is trained as follows.

The method begins with an initial JointBoost classifier using an initial feature set $\tilde{\mathbf{x}}$, containing the geometric and shading features, described in Appendix B. The classifier is applied to produce the probability $P(c_i|\tilde{\mathbf{x}}_i)$ for each possible label c_i given the feature set $\tilde{\mathbf{x}}_i$ of each pixel i. These probabilities are then binned in order to produce contextual features. In particular, for each pixel, the algorithm computes a histogram of these probabilities as a function of geodesic distances from it:

$$p_i^c = \sum_{j:\, d_b \leq \mathrm{dist}(i,j) < d_{b+1}} P(c_j)/N_b \qquad (15)$$

where the histogram bin b contains all pixels j with geodesic distance range $[d_b, d_{b+1}]$ from pixel i, and N_b is the total number of pixels in the histogram bin b. The geodesic distances are computed on the mesh and projected to image space. 4 bins are used, chosen in logarithmic space. The bin values p_i^c are normalized to sum to 1 per pixel. The total number of bins are $4|C|$. The values of these bins are used as contextual features, which are concatenated into $\tilde{\mathbf{x}}_i$ to form a new scalar feature set $\mathbf{x}_i$. Then, a second JointBoost classifier is learned, using the new feature set $\mathbf{x}$ as input and outputting updated probabilities $P(c_i|\mathbf{x}_i)$. These are used in turn to update the contextual features. The next classifier uses the contextual features generated by the previous one, and so on. Each JointBoost classifier is initialized with uniform weights and terminates when the holdout-validation error reaches to a minimum. The holdout-validation error is measured on pixels that are contained in random rectangles on the drawing, selected as above. The cascade terminates when the holdout-validation error of a JointBoost classifier is increased with respect to the holdout-validation error of the previous one. The unary term is defined based on the probabilities returned by the latter classifier.

To learn the pairwise term of Equation 3, the algorithm needs to estimate the probability of boundaries of hatching regions $P(e|\mathbf{x})$, which also serve as evidence for label boundaries. First, we observe that segment boundaries are likely to occur at particular parts of an image; for example, pixels separate by an occluding and suggestive contour are much less likely to be in the same segment as two pixels that are adjacent on the surface. For this reason, we define a binary JointBoost classifier, which maps to probabilities of boundaries of hatching regions for each pixel, given the subset of its features $\mathbf{x}$ computed from the feature curves of the mesh (see Appendix B). In this binary case, JointBoost reduces to an earlier algorithm called GentleBoost [Friedman et al. 2000]. The training data for this pairwise classifier are supplied by the marked boundaries of hatching regions of the source illustration (see Appendix A); pixels that are marked as boundaries have $e = 1$, otherwise $e = 0$. The classifier is initialized with more weight give to the pixels that contain boundaries of hatching level regions, since the training data contains many more non-boundary pixels. More specifically, if N_B are the total number of boundary pixels, and N_{NB} is the number of non-boundary pixels, then the weight is N_{NB}/N_B for boundary pixels and 1 for the rest. The boosting iterations terminate when the hold-out validation error measured on validation pixels (selected as above) is minimum.

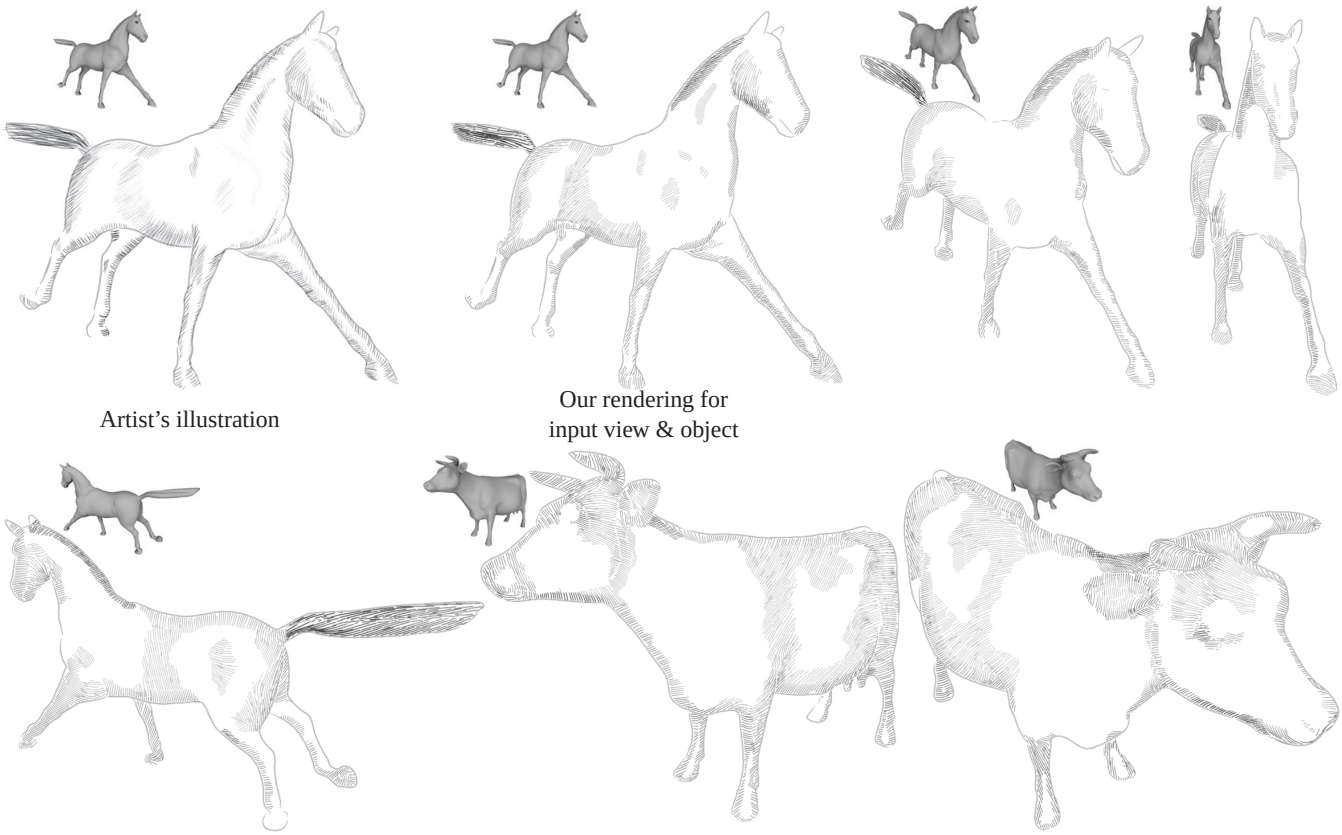

Fig. 6: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a horse. Rendering of the model with our learned style. Renderings of new views and new objects.*

Finally, the parameters ℓ and μ are optimized by maximizing the following energy term:

$$E_S = \sum_{i:c_i \neq c_j, j \in N(i)} P(e_i|\mathbf{x}) \tag{16}$$

where $N(i)$ is the 8-neighborhood of pixel i, and c_i, c_j are the labels for each pair of neighboring pixels i, j inferred using the CRF model of Equation 1 based on the learned parameters of its unary and pairwise classifier and using different values for ℓ, μ. This optimization attempts to "push" the segment label boundaries to be aligned with pixels that have higher probability to be boundaries. The energy is maximized using Matlab's implementation of Preconditioned Conjugate Gradient with numerically-estimated gradients.

Learning to generate hatching levels. The next step is to learn the hatching levels $h \in \{0, 1, 2\}$. The input here is the hatching level h_i per pixel contained inside the rendered area (as extracted during the pre-processing step (Appendix A) together with their full feature set $\mathbf{x}_i$ (including the contextual features as extracted above).

Our goal is to compute the parameters of the second CRF model used for inferring the hatching levels (Equation 4). Our algorithm first uses a JointBoost classifier that maps from the feature set $\mathbf{x}$ to the training hatching levels h. The classifier is initialized with uniform weights and terminates the boosting rounds when the hold-out validation error is minimized (the hold-out validation pixels are selected as above). The classifier outputs the probability $P(h_i|\mathbf{x}_i)$, which is used in the unary term of the CRF model. Finally, our algorithm uses the same pairwise term parameters trained with the CRF model of the segment labels to rectify the boundaries of the hatching levels.

Examples comparing our learned hatching algorithm to several alternatives are shown in Figure 4.

5.3 Learning real-valued stroke properties

Thickness, intensity, length, and spacing are all positive, real-valued quantities, and so the same learning procedure is used for each one in turn. The input to the algorithm are the values of the corresponding stroke properties, as extracted in the preprocessing step (Section A) and the full feature set $\mathbf{x}_i$ per pixel.

The multiplicative model of Equation 8 is used to map the features to the stroke properties. The model is learned in the log-domain, so that it can be learned as a linear sum of log functions. The model is learned with gradient-based boosting for regression (Appendix D). The weights for the training pixels are initialized as uniform. As above, the boosting iterations stop when the holdout-validation measured on randomly selected validation pixels is minimum.

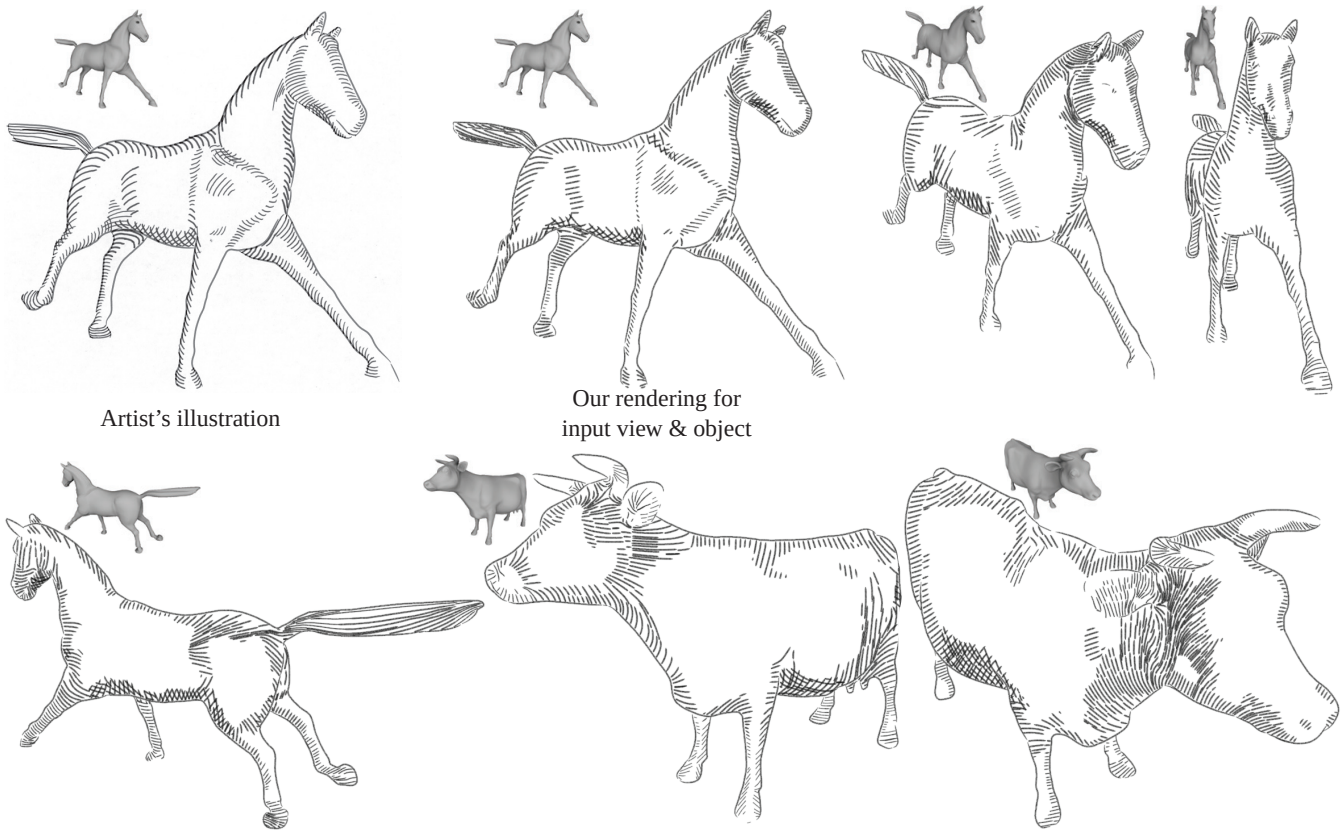

Fig. 7: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a horse with a different style than 6. Rendering of the model with our learned style. Renderings of new views and new objects.*

Examples comparing our method to several alternatives are shown in Figure 5.

6. RESULTS

The figures throughout our paper show synthesized line drawings of novel objects and views with our learning technique (Figures 1, 6, 7, 8, 9, 10, 11, 12, 13, 14). As can be seen in the examples, our method captures several aspects of the artist's drawing style, better than alternative previous approaches (Figure 1). Our algorithm adapts to different styles of drawing and successfully synthesizes them for different objects and views. For example, Figures 6 and 7 show different styles of illustrations for the same horse, applied to new views and objects. Figure 14 shows more examples of synthesis with various styles and objects.

However, subtleties are sometimes lost. For example, in Figure 12, the face is depicted with finer-scale detail than the clothing, which cannot be captured in our model. In Figure 13, our method loses variation in the character of the lines, and depiction of important details such as the eye. One reason for this is that the stroke placement algorithm attempts to match the target hatching properties, but does not optimize to match a target tone. These variations may also depend on types of parts (e.g., eyes versus torsos), and could be addressed given part labels [Kalogerakis et al. 2010]. Figure 11 exhibits randomness in stroke spacing and width that is not modeled by our technique.

Selected features. We show the frequency of orientation features selected by gradient-based boosting and averaged over all our nine drawings in Figure 15. Fields aligned with principal curvature directions as well as local principal axes (corresponding to candidate local planar symmetry axes) play very important roles for synthesizing the hatching orientations. Fields aligned with suggestive contours, ridges and valleys are also significant for determining orientations. Fields based on shading attributes have moderate influence.

We show the frequency of scalar features averaged selected by boosting and averaged over all our nine drawings in Figure 16 for learning the hatching level, thickness, spacing, intensity, length, and segment label. Shape descriptor features (based on PCA, shape contexts, shape diameter, average geodesic distance, distance from medial surface, contextual features) seem to have large influence on all the hatching properties. This means that the choice of tone is probably influenced by the type of shape part the artist draws. The segment label is mostly determined by the shape descriptor features, which is consistent with the previous work on shape segmentation and labeling [Kalogerakis et al. 2010]. The hatching level is mostly influenced by image intensity, $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$. The stroke thickness is mostly affected by shape descriptor features, curvature,

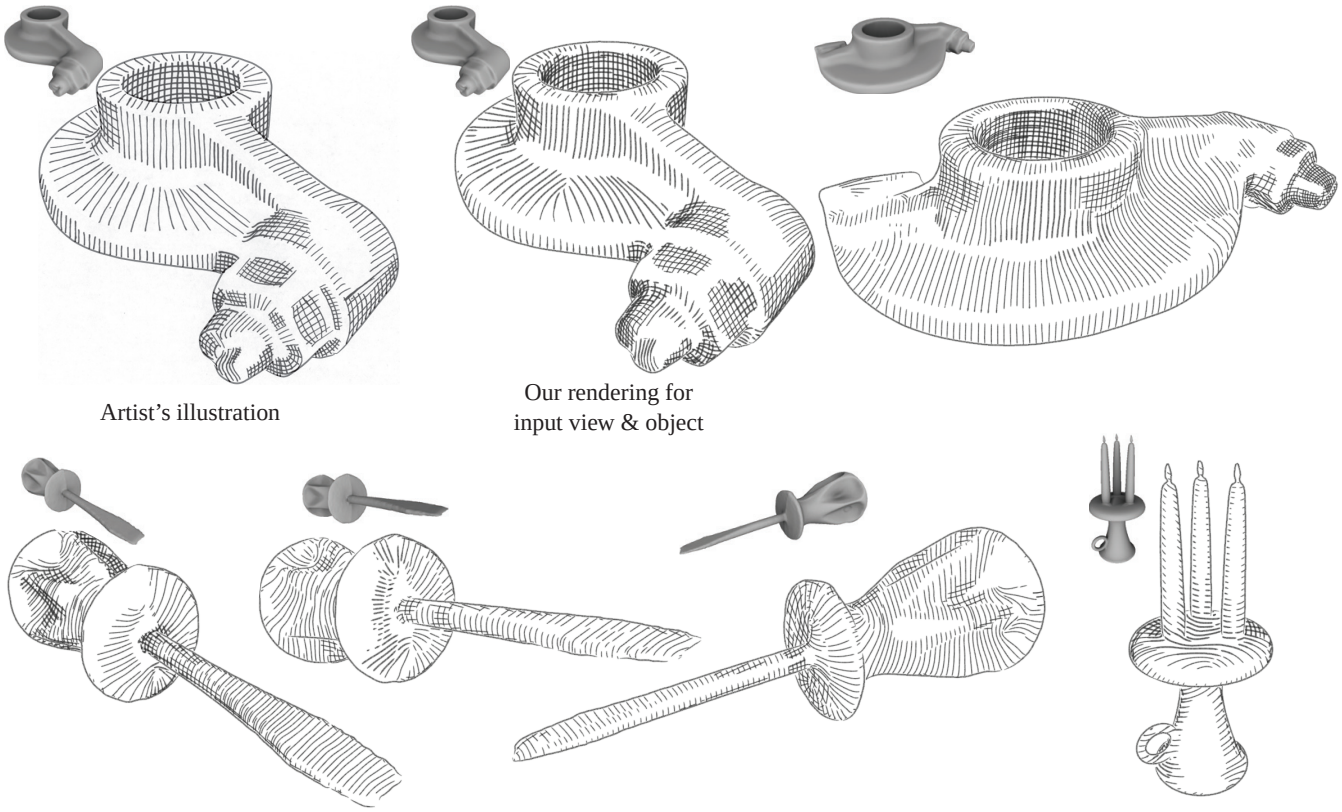

Fig. 8: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a rocker arm. Rendering of the model with our learned style. Renderings of new views and new objects.*

Fig. 9: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a pitcher. Rendering of the model with our learned style. Renderings of new views and new objects.*

$\vec{L} \cdot \vec{N}$, gradient of image intensity, the location of feature lines, and, finally, depth. Spacing is mostly influenced by shape descriptor features, curvature, derivatives of curvature, $\vec{L} \cdot \vec{N}$, and $\vec{V} \cdot \vec{N}$. The intensity is influenced by shape descriptor features, image intensity, $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$, depth, and the location of feature lines. The length is mostly determined by shape descriptor features, curvature, radial curvature, $\vec{L} \cdot \vec{N}$, image intensity and its gradient, and location of feature lines (mostly suggestive contours).

However, it is important to note that different features are learned for different input illustrations. For example, in Figure 11, the light directions mostly determine the orientations, which is not the case for the rest of the drawings. We include histograms of the frequency of orientation and scalar features used for each of the drawing in the supplementary material.

Computation time. In each case, learning a style from a source illustration takes 5 to 10 hours on a laptop with Intel i7 proces-

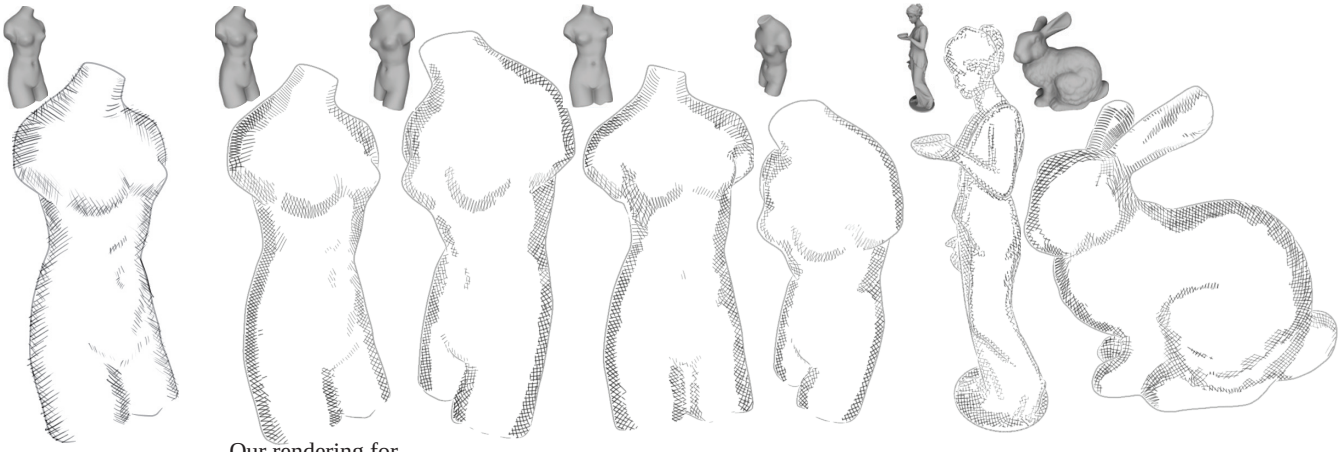

Fig. 10: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a Venus statue. Rendering of the model with our learned style. Renderings of new views and new objects.*

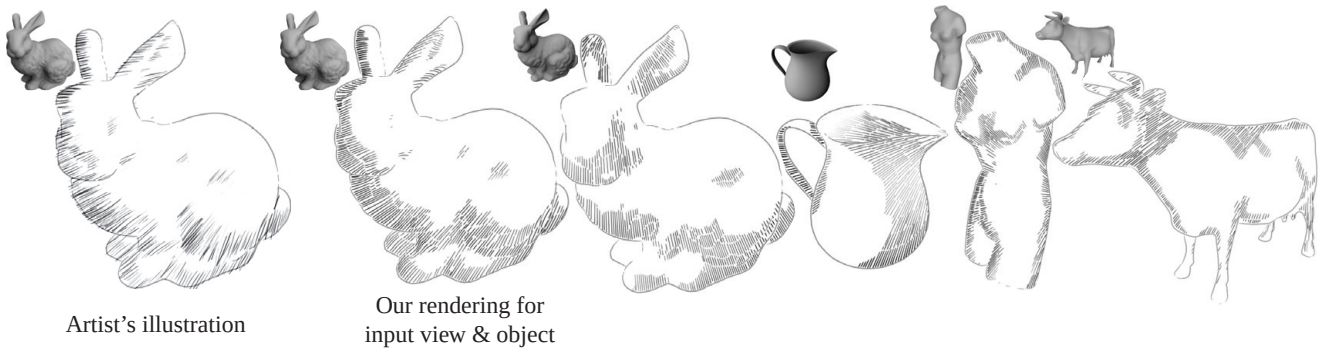

Fig. 11: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a bunny using a particular style; hatching orientations are mostly aligned with point light directions. Rendering of the model with our learned style. Renderings of new views and new objects.*

sor. Most of the time is consumed by the orientation and clustering step (Section 5.1) (about 50% of the time for the horse), which is implemented in Matlab. Learning segment labels and hatching levels (Section 5.2) represents about 25% of the training time (implemented in C++) and learning stroke properties (Section 5.3) takes about 10% of the training time (implemented in Matlab). The rest of the time is consumed for extracting the features (implemented in C++) and training hatching properties (implemented in Matlab). We note that our implementation is currently far from optimal, hence, running times could be improved. Once the model of the style is learned, it can be applied to different novel data. Given the predicted hatching and cross-hatching orientations, hatching level, thickness, intensity, spacing and stroke length at each pixel, our algorithm traces streamlines over the image to generate the final pen-and-ink illustration. Synthesis takes 30 to 60 minutes. Most of the time (about 60%) is consumed here for extracting the features. The implementation for feature extraction and tracing streamlines are also far from optimal.

7. SUMMARY AND FUTURE WORK

Ours is the first method to generate predictive models for synthesizing detailed line illustrations from examples. We model line illustrations with a machine learning approach using a set of features suspected to play a role in the human artistic process. The complexity of man-made illustrations is very difficult to reproduce; however, we believe our work takes a step towards replicating certain key aspects of the human artistic process. Our algorithm generalizes to novel views as well as objects of similar morphological class.

There are many aspects of hatching styles that we do not capture, including: stroke textures, stroke tapering, randomness in strokes (such as wavy or jittered lines), cross-hatching with more than two hatching directions, style of individual strokes, and continuous transitions in hatching level. Interactive edits to the hatching properties could be used to improve our results [Salisbury et al. 1994].

Since we learn from a single training drawing, the generalization capabilities of our method to novel views and objects are limited.

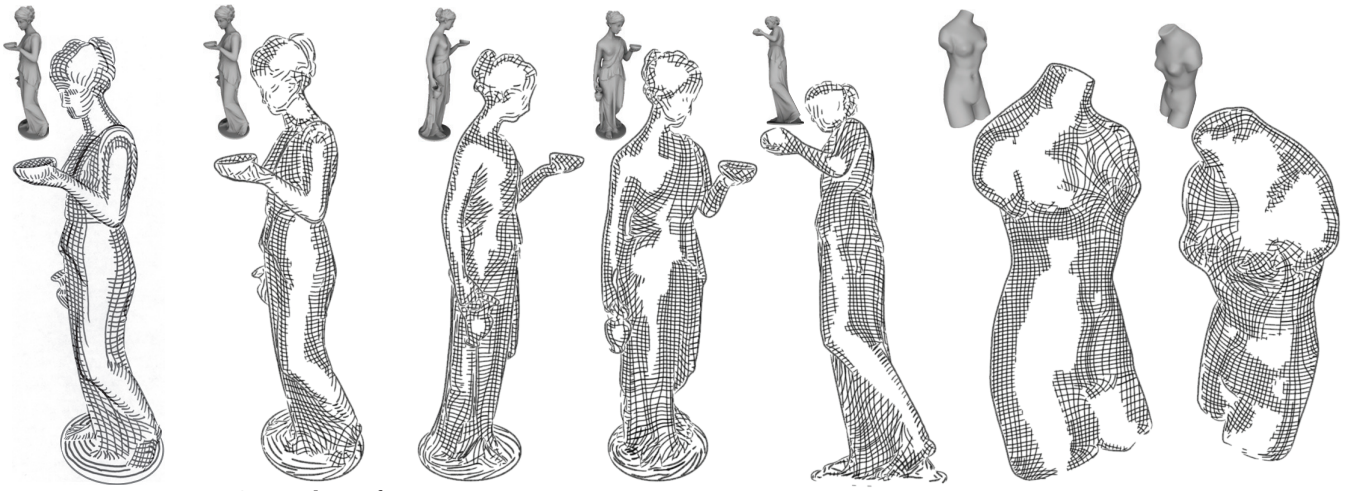

Fig. 12: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a statue. Rendering of the model with our learned style. Renderings of new views and new objects.*

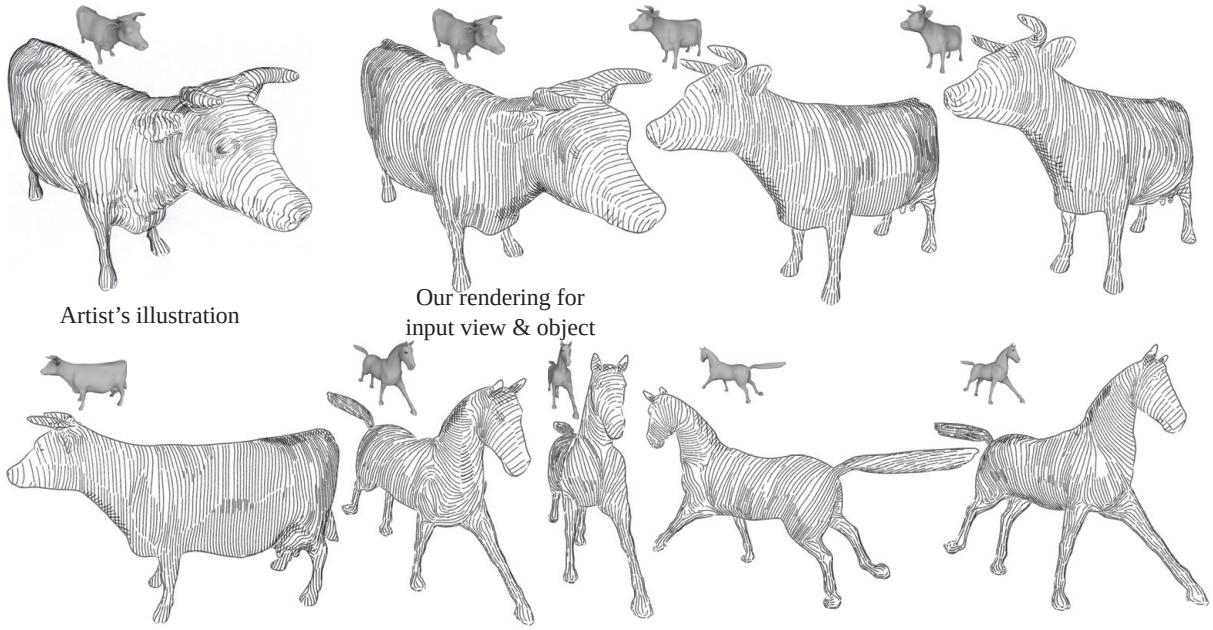

Fig. 13: *Data-driven line art illustrations generated with our algorithm.* ***From left to right:*** *Artist's illustration of a cow. Rendering of the model with our learned style. Renderings of new views and new objects.*

For example, if the relevant features differ significantly between the test views and objects, then our method will not generalize to them. Our method relies on holdout validation using randomly selected regions to avoid overfitting; this ignores the hatching information existing in these regions that might be valuable. Re-training the model is sometimes useful to improve results, since these regions are selected randomly. Learning from a broader corpus of examples could help with these issues, although this would require drawings where the hatching properties change consistently across different object and views. In addition, if none of the features or a combination of them can be mapped to a hatching property, then our method will also fail.

Finding what and how other features are relevant to artists' pen-and-ink illustrations is an open problem. Our method does not represent the dependence of style on part labels (e.g., eyes versus torsos), as previously done for painterly rendering of images [Zeng et al. 2009]. Given such labels, it could be possible to generalize the algorithm to take this information into account.

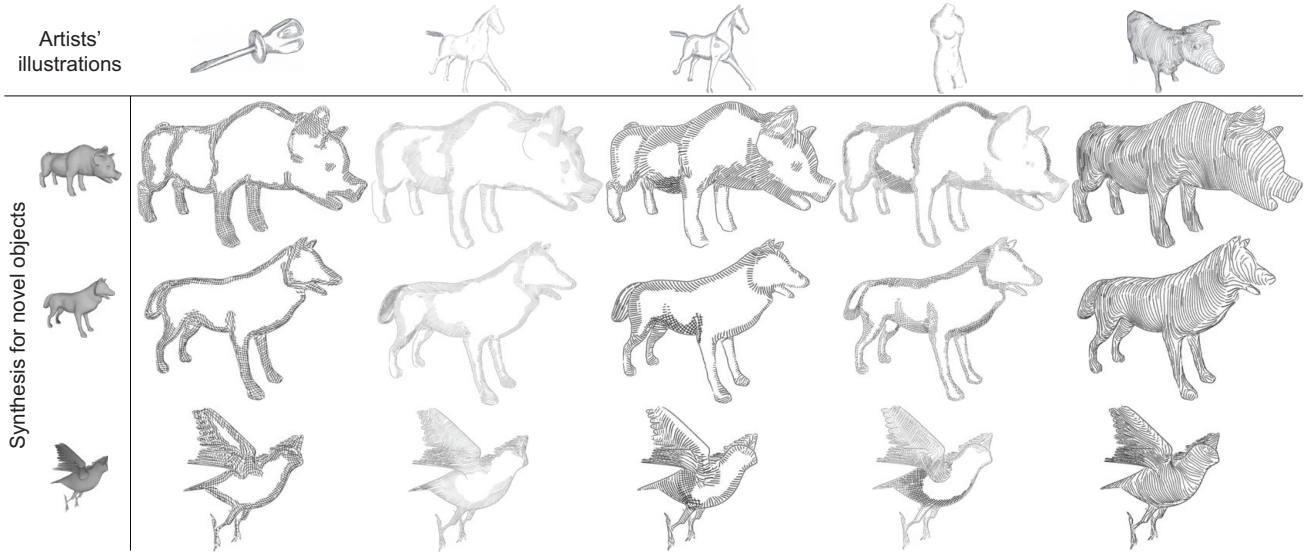

Fig. 14: *Data-driven line art illustrations generated with our algorithm based on the learned styles from the artists' drawings in Figures 1, 6, 7, 10, 13.*

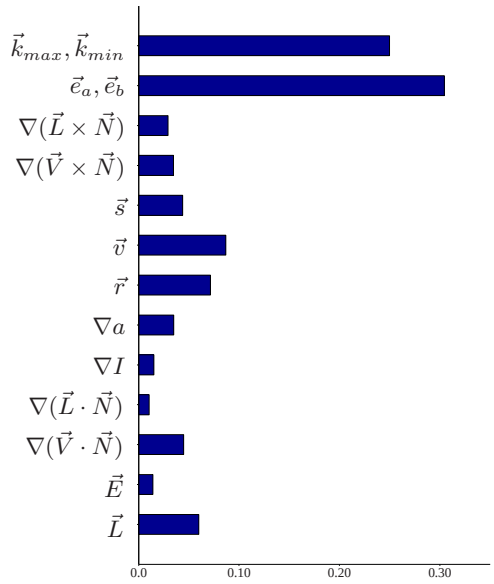

Fig. 15: ***Frequency of the first three orientation features selected by gradient-based boosting for learning the hatching orientation fields.*** *The frequency is averaged over all our nine training drawings (Figures 1, 6, 7, 8, 9, 10, 11, 12, 13). The contribution of each feature is also weighted by the total segment area where it is used. The orientation features are grouped based on their type: principal curvature directions ($\vec{k}_{max}, \vec{k}_{min}$), local principal axes directions ($\vec{e}_a, \vec{e}_b$), $\nabla(\vec{L} \times \vec{N})$, $\nabla(\vec{V} \times \vec{N})$, directions aligned with suggestive contours ($\vec{s}$), valleys ($\vec{v}$), ridges ($\vec{r}$), gradient of ambient occlusion (∇a), gradient of image intensity (∇I), gradient of $(\vec{L} \cdot \vec{N})$, gradient of $(\vec{V} \cdot \vec{N})$, vector irradiance ($\vec{E}$), projected light direction ($\vec{L}$).*

The quality of our results depend on how well the hatching properties were extracted from the training drawing during the preprocessing step. This step gives only coarse estimates, and depend on various thresholds. This preprocessing cannot handle highly-stylized strokes such as wavy lines or highly-textured strokes.

Example-based stroke synthesis [Freeman et al. 2003; Hertzmann et al. 2002; Kalnins et al. 2002] may be combined with our approach to generate styles with similar stroke texture. An optimization technique [Turk and Banks 1996] might be used to place streamlines appropriately in order to match a target tone. Our method focuses only on hatching, and renders feature curves separately. Learning the feature curves is an interesting future direction. Another direction for future work is hatching for animated scenes, possibly based on a data-driven model similar to [Kalogerakis et al. 2009]. Finally, we believe that aspects of our approach may be applicable to other applications in geometry processing and artistic rendering, especially for vector field design.

REFERENCES

ARVO, J. 1995. Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *Proc. SIGGRAPH*. 335–342.

BARLA, P., BRESLAV, S., THOLLOT, J., SILLION, F., AND MARKOSIAN, L. 2006. Stroke pattern analysis and synthesis. *Comput. Graph. Forum 25*, 3.

BELONGIE, S., MALIK, J., AND PUZICHA, J. 2002. Shape Matching and Object Recognition Using Shape Contexts. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 4.

BISHOP, C. M. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag.

BOYKOV, Y., VEKSLER, O., AND ZABIH, R. 2001. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Trans. Pattern Anal. Mach. Intell. 23*, 11.

COLE, F., GOLOVINSKIY, A., LIMPAECHER, A., BARROS, H. S., FINKELSTEIN, A., FUNKHOUSER, T., AND RUSINKIEWICZ, S. 2008. Where Do People Draw Lines? *ACM Trans. Graph. 27*, 3.

DECARLO, D. AND RUSINKIEWICZ, S. 2007. Highlight lines for conveying shape. In *NPAR*.

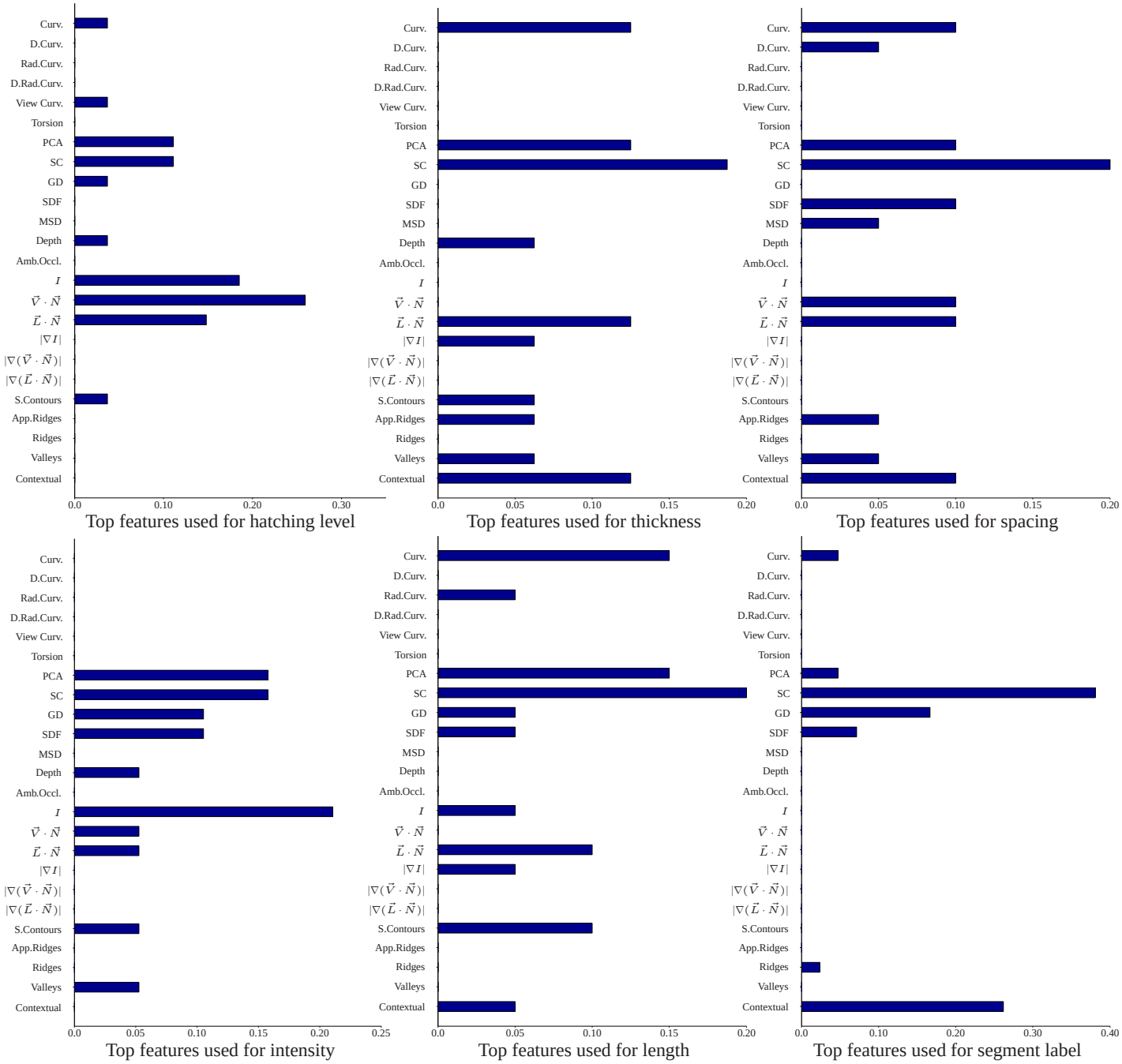

Fig. 16: *Frequency of the first three scalar features selected by the boosting techniques used in our algorithm for learning the scalar hatching properties. The frequency is averaged over all nine training drawings. The scalar features are grouped based on their type: Curvature (Curv.), Derivatives of Curvature (D. Curv.), Radial Curvature (Rad. Curv.),Derivative of Radial Curvature (D. Rad. Curv.), Torsion, features based on PCA analysis on local shape neighborhoods, features based Shape Context histograms [Belongie et al. 2002], features based on geodesic distance descriptor [Hilaga et al. 2001], shape diameter function features [Shapira et al. 2010], distance from medial surface features [Liu et al. 2009], depth, ambient occlusion, image intensity (I), $\vec{V}\cdot\vec{N}$, $\vec{L}\cdot\vec{N}$, gradient magnitudes of the last three, strength of suggestive contours, strength of apparent ridges, strength of ridges and values, contextual label features.*

ELBER, G. 1998. Line Art Illustrations of Parametric and Implicit Forms. *IEEE TVCG 4*, 1, 71–81.

FREEMAN, W. T., TENENBAUM, J., AND PASZTOR, E. 2003. Learning style translation for the lines of a drawing. *ACM Trans. Graph. 22*, 1,

33–46.

FRIEDMAN, J., HASTIE, T., AND TIBSHIRANI, R. 2000. Additive Logistic Regression: a Statistical View of Boosting. *The Annals of Statistics 38*, 2.

GOODWIN, T., VOLLICK, I., AND HERTZMANN, A. 2007. Isophote Distance: A Shading Approach to Artistic Stroke Thickness. In *Proc. NPAR*. 53–62.

GUPTILL, A. L. 1997. *Rendering in Pen and Ink*. Watson-Guptill, edited by Susan E. Meyer.

HAMEL, J. AND STROTHOTTE, T. 1999. Capturing and Re-Using Rendition Styles for Non-Photorealistic Rendering. *Computer Graphics Forum 18*, 3, 173–182.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image Analogies. *Proc. SIGGRAPH*.

HERTZMANN, A., OLIVER, N., CURLESS, B., AND SEITZ, S. M. 2002. Curve Analogies. In *Proc. EGWR*.

HERTZMANN, A. AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proc. SIGGRAPH*. 517–526.

HILAGA, M., SHINAGAWA, Y., KOHMURA, T., AND KUNII, T. L. 2001. Topology Matching for Fully Automatic Similarity Estimation of 3d Shapes. In *SIGGRAPH*.

JODOIN, P.-M., EPSTEIN, E., GRANGER-PICHÉ, M., AND OSTROMOUKHOV, V. 2002. Hatching by Example: a Statistical Approach. In *Proc. NPAR*. 29–36.

JORDAN, M. I. AND JACOBS, R. A. 1994. Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation 6*, 181–214.

JUDD, T., DURAND, F., AND ADELSON, E. 2007. Apparent ridges for line drawing. *ACM Trans. Graph. 26*, 3.

KALNINS, R., MARKOSIAN, L., MEIER, B., KOWALSKI, M., LEE, J., DAVIDSON, P., WEBB, M., HUGHES, J., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: drawing strokes directly on 3D models. In *Proc. SIGGRAPH*. 755–762.

KALOGERAKIS, E., HERTZMANN, A., AND SINGH, K. 2010. Learning 3d mesh segmentation and labeling. *ACM Trans. Graph. 29*, 3.

KALOGERAKIS, E., NOWROUZEZAHRAI, D., SIMARI, P., MCCRAE, J., HERTZMANN, A., AND SINGH, K. 2009. Data-driven curvature for real-time line drawing of dynamic scenes. *ACM Trans. Graphics 28*, 1.

KIM, S., MACIEJEWSKI, R., ISENBERG, T., ANDREWS, W. M., CHEN, W., SOUSA, M. C., AND EBERT, D. S. 2009. Stippling by Example. In *NPAR*.

KIM, S., WOO, I., MACIEJEWSKI, R., , AND EBERT., D. S. 2010. Automated Hedcut Illustration using Isophotes. In *Proc. Smart Graphics*.

KIM, Y., YU, J., YU, X., AND LEE, S. 2008. Line-art Illustration of Dynamic and Specular Surfaces. *ACM Trans. Graphics*.

LIU, R. F., ZHANG, H., SHAMIR, A., AND COHEN-OR, D. 2009. A Part-Aware Surface Metric for Shape Analysis. *Computer Graphics Forum, (Eurographics 2009) 28*, 2.

LUM, E. B. AND MA, K.-L. 2005. Expressive line selection by example. *The Visual Computer 21*, 8, 811–820.

MERTENS, T., KAUTZ, J., CHEN, J., BEKAERT, P., AND DURAND., F. 2006. Texture Transfer Using Geometry Correlation. In *EGSR*.

PALACIOS, J. AND ZHANG, E. 2007. Rotational Symmetry Field Design on Surfaces. *ACM Trans. Graph.*.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-Time Hatching. In *Proc. SIGGRAPH*.

RUSINKIEWICZ, S. AND DECARLO, D. 2007. rtsc library. http://www.cs.princeton.edu/gfx/proj/sugcon/.

SAITO, T. AND TAKAHASHI, T. 1990. Comprehensible Rendering of 3-D Shapes. In *Proc. SIGGRAPH*. 197–206.

SALISBURY, M. P., ANDERSON, S. E., BARZEL, R., AND SALESIN, D. H. 1994. Interactive pen-and-ink illustration. In *SIGGRAPH*. 101–108.

SHAPIRA, L., SHALOM, S., SHAMIR, A., ZHANG, R. H., AND COHEN-OR, D. 2010. Contextual Part Analogies in 3D Objects. *International Journal of Computer Vision*.

SHOTTON, J., JOHNSON, M., AND CIPOLLA, R. 2008. Semantic Texton Forests for Image Categorization and Segmentation. In *Proc. CVPR*.

SHOTTON, J., WINN, J., ROTHER, C., AND CRIMINISI, A. 2009. TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context. *Int. J. Comput. Vision 81*, 1.

SIMARI, P., KALOGERAKIS, E., AND SINGH, K. 2006. Folding Meshes: Hierarchical Mesh Segmentation Based on Planar Symmetry. In *SGP*.

SINGH, M. AND SCHAEFER, S. 2010. Suggestive Hatching. In *Proc. Computational Aesthetics*.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *Proc. SIGGRAPH*. 527–536.

TORRALBA, A., MURPHY, K. P., AND FREEMAN, W. T. 2007. Sharing Visual Features for Multiclass and Multiview Object Detection. *IEEE Trans. Pattern Anal. Mach. Intell. 29*, 5.

TU, Z. 2008. Auto-context and its Application to High-level Vision Tasks. In *Proc. CVPR*.

TURK, G. AND BANKS, D. 1996. Image-guided streamline placement. In *SIGGRAPH*.

WAGSTAFF, K., CARDIE, C., ROGERS, S., AND SCHRÖDL, S. 2001. Constrained k-means clustering with background knowledge. In *ICML*.

WINKENBACH, G. AND SALESIN, D. 1994. Computer-generated pen-and-ink illustration. In *Proc. SIGGRAPH*. 91–100.

WINKENBACH, G. AND SALESIN, D. 1996. Rendering parametric surfaces in pen and ink. In *Proc. SIGGRAPH*. 469–476.

ZEMEL, R. AND PITASSI, T. 2001. A gradient-based boosting algorithm for regression problems. In *Neural Information Processing Systems*.

ZENG, K., ZHAO, M., XIONG, C., AND ZHU, S.-C. 2009. From image parsing to painterly rendering. *ACM Trans. Graph. 29*.

APPENDIX

A. IMAGE PREPROCESSING

Given an input illustration drawn by an artist, we apply the following steps to determine the hatching properties for each stroke pixel. First, we scan the illustration and align it to the rendering automatically by matching borders with brute force search. The following steps are sufficiently accurate to provide training data for our algorithms.

Intensity: The intensity I_i is set to the grayscale intensity of the pixel i of the drawing. It is normalized within the range $[0, 1]$.
Thickness: Thinning is first applied to identify a single-pixel-wide skeleton for the drawing. Then, from each skeletal pixel, a Breadth-First Search (BFS) is performed to find the nearest pixel in the source image with intensity less than half of the start pixel. The distance to this pixel is the stroke thickness.
Orientation: The structure tensor of the local image neighborhood is computed at the scale of the previously-computed thickness of the stroke. The dominant orientation in this neighborhood is given by the eigenvector corresponding to the smallest eigenvalue of the structure tensor. Intersection points are also detected, so that they can be omitted from orientation learning. Our algorithm marks as intersection points those points detected by a Harris corner detector in both the original drawing and the skeleton image. Finally, in

order to remove spurious intersection points, pairs of intersection points are found with distance less than the local stroke thickness, and their centroid is marked as an intersection instead.

Spacing: For each skeletal pixel, a circular region is grown around the pixel. At each radius, the connected components of the region are computed. If at least 3 pixels in the region are not connected to the center pixel, with orientation within $\pi/6$ of the center pixel's orientation, then the process halts. The spacing at the center pixel is set to the final radius.

Length: A BFS is executed on the skeletal pixels to count the number of pixels per stroke. In order to follow a single stroke (excluding pixels from overlapping cross-hatching strokes), at each BFS expansion, pixels are considered inside the current neighborhood with similar orientation (at most $\pi/12$ angular difference from the current pixel's orientation).

Hatching Level: For each stroke pixel, an ellipsoidal mask is created with its semi-minor axis aligned to the extracted orientation, and major radius equal to its spacing. All pixels belonging to any of these masks are given label $H_i = 1$. For each intersection pixel, a circular mask is also created around it with radius equal to its spacing. All connected components are computed from the union of these masks. If any connected component contains more than 4 intersection pixels, the pixels of the component are assigned with label $H_i = 2$. Two horizontal and vertical strokes give rise to a minimum cross-hatching region (with 4 intersections).

Hatching region boundaries: Pixels are marked as boundaries if they belong to boundaries of the hatching regions or if they are endpoints of the skeleton of the drawing.

We perform a final smoothing step (with a Gaussian kernel of width equal to the median of the spacing values) to denoise the properties.

B. SCALAR FEATURES

There are 1204 scalar features ($\tilde{\mathbf{x}} \in \Re^{760}$) for learning the scalar properties of the drawing. The first 90 are mean curvature, Gaussian curvature, maximum and minimum principal curvatures by sign and absolute value, derivatives of curvature, radial curvature and its derivative, view-dependent minimum and maximum curvatures [Judd et al. 2007], geodesic torsion in the projected viewing direction [DeCarlo and Rusinkiewicz 2007]. These are measured in three scales (1%, 2%, 5% relative to the median of all-pairs geodesic distances in the mesh) for each vertex. We also include their absolute values, since some hatching properties may be insensitive to sign. The above features are first computed in object-space and then, projected to image-space.

The next 110 features are based on local shape descriptors, also used in [Kalogerakis et al. 2010] for labeling parts. We compute the singular values s_1, s_2, s_3 of the covariance of vertices inside patches of various geodesic radii (5%, 10%, 20%) around each vertex, and also add the following features for each patch: $s_1/(s_1+s_2+s_3)$, $s_2/(s_1+s_2+s_3)$, $s_3/(s_1+s_2+s_3)$, $(s_1+s_2)/(s_1+s_2+s_3)$, $(s_1+s_3)/(s_1+s_2+s_3)$, $(s_2+s_3)/(s_1+s_2+s_3)$, s_1/s_2, s_1/s_3, s_2/s_3, $s_1/s_2+s_1/s_3$, $s_1/s_2+s_2/s_3$, $s_1/s_3+s_2/s_3$, yielding 45 features total. We also include 24 features based on the Shape Diameter Function (SDF) [Shapira et al. 2010] and distance from medial surface [Liu et al. 2009]. The SDF features are computed using cones of angles 60, 90, and 120 per vertex. For each cone, we get the weighted average of the samples and their logarithmized versions with different normalizing parameters $\alpha = 1$, $\alpha = 2$, $\alpha = 4$. For each of the cones above, we also compute the distance of medial surface from each vertex. We measure the diameter of the maximal inscribed sphere touching each vertex. The corresponding medial surface point will be roughly its center. Then we send rays from this point uniformly sampled on a Gaussian sphere, gather the intersection points and measure the ray lengths. As with the shape diameter features, we use the weighted average of the samples, we normalize and logarithmize them with the same above normalizing parameters. In addition, we use the average, squared mean, $10th, 20th, ..., 90th$ percentile of the geodesic distances of each vertex to all the other mesh vertices, yielding 11 features. Finally, we use 30 shape context features [Belongie et al. 2002], based on the implementation of [Kalogerakis et al. 2010]. All the above features are first computed in object-space per vertex and then, projected to image-space.

The next 53 features are based on functions of the rendered 3D object in image space. We use maximum and minimum image curvature, image intensity, and image gradient magnitude features, computed with derivative-of-Gaussian kernels with $\sigma = 1, 2, 3, 5$, yielding 16 features. The next 12 features are based on shading under different models: $\vec{V} \cdot \vec{N}$, $\vec{L} \cdot \vec{N}$ (both clamped at zero), ambient occlusion, where $\vec{V}$, $\vec{L}$, and $\vec{N}$ are the view, light, and normal vectors at a point. These are also smoothed with Gaussian kernels of $\sigma = 1, 2, 3, 5$. We also include the corresponding gradient magnitude, the maximum and minimum curvature of $\vec{V} \cdot \vec{N}$ and $\vec{L} \cdot \vec{N}$ features, yielding 24 more features. We finally include the depth value for each pixel.

We finally include the per pixel intensity of occluding and suggestive contours, ridges, valleys and apparent ridges extracted by the rtsc software package [Rusinkiewicz and DeCarlo 2007]. We use 4 different thresholds for extracting each feature line (the rtsc thresholds are chosen from the logarithmic space $[0.001, 0.1]$ for suggestive contours and valleys and $[0.01, 0.1]$ for ridges and apparent ridges). We also produce dilated versions of these features lines by convolving their image with Gaussian kernels with $\sigma = 5, 10, 20$, yielding in total 48 features.

Finally, we also include all the above 301 features with their powers of 2 (quadratic features), -1 (inverse features), -2 (inverse quadratic features), yielding 1204 features in total. For the inverse features, we prevent divisions by zero, by truncating near-zero values to $1e-6$ (or $-1e-6$ if they are negative). Using these transformations on the features yielded slightly better results for our predictions.

C. ORIENTATION FEATURES

There are 70 orientation features ($\boldsymbol{\theta}$) for learning the hatching and cross-hatching orientations. Each orientation feature is a direction in image-space; orientation features that begin as 3D vectors are projected to 2D. The first six features are based on surface principal curvature directions computed at 3 scales as above. Then, the next six features are based on surface local PCA axes projected on the tangent plane of each vertex corresponding to the two larger singular values of the covariance of multi-scale surfaces patches computed as above. Note that the local PCA axes correspond to candidate local planar symmetry axes [Simari et al. 2006]. The next features are: $\vec{L} \times \vec{N}$ and $\vec{V} \times \vec{N}$. The above orientation fields are undefined at some points (near umbilic points for curvature directions, near planar and spherical patches for the PCA axes, and near $\vec{L} \cdot \vec{N} = 0$ and $\vec{V} \cdot \vec{N} = 0$ for the rest). Hence, we use globally-smoothed direction based on the technique of [Hertzmann and

Zorin 2000]. Next, we include $\vec{L}$, and vector irradiance $\vec{E}$ [Arvo 1995]. The next 3 features are vector fields aligned with the occluding and suggestive contours (given the view direction), ridges and valleys of the mesh. The next 16 features are image-space gradients of the following scalar features: $\nabla(\vec{V}\cdot\vec{N})$, $\nabla(\vec{L}\cdot\vec{N})$, ambient occlusion and image intensity ∇I computed at 4 scales as above. The remaining orientation features are the directions of the first 35 features rotated by 90 degrees in the image-space.

D. BOOSTING FOR REGRESSION

The stroke orientations as well as the thickness, intensity, length and spacing are learned with the gradient-based boosting technique of Zemel and Pitassi [2001]. Given input features $\mathbf{x}$, the gradient-based boosting technique aims at learning an additive model of the following form to approximate a target property:

$$\tau(\mathbf{x}) = \sum_k r_k \psi_{\sigma(k)}(\mathbf{x}) \tag{17}$$

where $\psi_{\sigma(k)}$ is a function on the k-th selected feature with index $\sigma(k)$ and r_k is its corresponding weight. For stroke orientations, the functions are simply single orientation features: $\psi_{\sigma(k)}(\mathbf{v}) = \mathbf{v}_{\sigma(k)}$. Hence, in this case, the above equation represents a weighted combination (i.e., interpolation) of the orientation features, as expressed in Equation 7 with $r_k = w_{\sigma(k)}$. For the thickness, spacing, intensity and length, we use functions of the form: $\psi_{\sigma(k)}(\mathbf{x}) = \log(a_k x_{\sigma(k)} + b_k)$, so that the selected feature is scaled and translated properly to match the target stroke property, as expressed in Equation 9 with $r_k = \alpha_{\sigma(k)}$.

Given N training pairs $\{\mathbf{x}_i, t_i\}, i = \{1, 2, ..., N\}$, where t_i are exemplar values of the target property, the gradient-based boosting algorithm attempts to minimize the average error of the models of the single features with respect to the weight vector $\mathbf{r}$:

$$L(\mathbf{r}) = \frac{1}{N}\sum_{i=1}^{N}\left(\prod_{k=1}^{K} {r_k}^{-0.5}\right)\exp\left(\sum_{k=1}^{K} r_k \cdot (t_i - \psi_k(\mathbf{x}_i))^2\right) \tag{18}$$

This objective function is minimized iteratively by updating a set of weights $\{\omega_i\}$ on the training samples $\{\mathbf{x}_i, t_i\}$. The weights are initialized to be uniform i.e. $\omega_i = 1/N$, unless there is a prior confidence on each sample. In this case, the weights can be initialized accordingly as in Section 5.1. Then, our algorithm initiates the boosting iterations that have the following steps:

- for each feature f in $\mathbf{x}$, the following function is minimized:

$$L_f = \sum_{i=1}^{N} \omega_i (r_k^{-0.5} \exp\left(r_k (t_i - \psi_f(\mathbf{x}_i))\right)^2) \tag{19}$$

  with respect to r_k as well as the parameters of a_k, b_k in the case of learning stroke properties. The parameter r_k is optimized using Matlab's active-set algorithm including the constraint that $r_k \in (0, 1]$ (with initial estimate set to 0.5). For the first boosting iteration $k = 1$, $r_k = 1$ is used always. For stroke properties, our algorithm alternates between optimizing for the parameters a_k, b_k with Matlab's BFGS implemenation, keeping r_k constant and optimizing for the parameter r_k, keeping the rest constant, until convergence or until 10 iterations are completed.
- the feature f is selected that yields the lowest value for L_f, hence $\sigma(k) = \arg\min_f L_f$.
- the weights on the training pairs are updated as follows:

$$\omega_i = \omega_i \cdot r_k^{-0.5} \exp\left(r_k \cdot (t_i - \psi_{\sigma(k)}(\mathbf{x}_i)\right)^2 \tag{20}$$

- The $\omega_i = \omega_i / \sum_i \omega_i$ are normalized so that they sum to 1.
- the hold-out validation error is measured: if it is increased, the loop is terminated and the selected feature of the current iteration are disregarded.

Finally, the weights $r_k = r_k / \sum_k r_k$ are normalized so that they sum to 1.

Received October 2010; accepted Month XXXX