Programmable Rendering of Line Drawing from 3D Scenes

STÉPHANE GRABLI and EMMANUEL TURQUIN
Université de Grenoble, INRIA
FRÉDO DURAND
Computer Science and Artificial Intelligence Lab, MIT
and
FRANÇOIS X. SILLION
INRIA, Université de Grenoble

This article introduces a programmable approach to nonphotorealistic line drawings from 3D models, inspired by programmable shaders in traditional rendering. This approach relies on the assumption generally made in NPR that style attributes (color, thickness, etc.) are chosen depending on generic properties of the scene such as line characteristics or depth discontinuities, etc. We propose a new image creation model where all operations are controlled through user-defined procedures in which the relations between style attributes and scene properties are specified. A *view map* describing all relevant support lines in the drawing and their topological arrangement is first created from the 3D model so as to ensure the continuity of all scene properties along its edges; a number of style modules operate on this map, by procedurally selecting, chaining, or splitting lines, before creating strokes and assigning drawing attributes. Consistent access to properties of the scene is provided from the different elements of the map that are manipulated throughout the whole process. The resulting drawing system permits flexible control of all elements of drawing style: First, different style modules can be applied to different types of lines in a view; second, the topology and geometry of strokes are entirely controlled from the programmable modules; and third, stroke attributes are assigned procedurally and can be correlated at will with various scene or view properties. We illustrate the components of our system and show how style modules successfully encode stylized visual characteristics that can be applied across a wide range of models.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Viewing algorithms*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Line drawing, nonphotorealistic rendering (NPR), style

ACM Reference Format:

Grabli, S., Turquin, E., Durand, F., and Sillion, F. X. 2010. Programmable rendering of line drawing from 3D scenes. ACM Trans. Graph. 29, 2, Article 18 (March 2010), 20 pages. DOI = 10.1145/1731047.1731056 http://doi.acm.org/10.1145/1731047.1731056

1. INTRODUCTION

1.1 Motivation and Approach

Art offers great flexibility in representation. Working with elements such as color and geometry, artists have long used this flexibility to fulfill goals of abstraction, simplification, or emphasis that make illustration popular and useful. The way each artist uses this freedom defines his visual identity, his *style*. In NonPhotorealistic Rendering (NPR), while many techniques enable the creation of compelling images [Gooch and Gooch 2001; Strothotte and Schlechtweg 2002], the control given to the user over the rendering style is insufficient and is generally limited to a fixed set of attributes. The approach presented in this article serves two objectives: first, it provides the user with as much control as possible over the rendering style and, second, it ensures separation between style and content, making style descriptions reusable with any 3D scene. We conduct our analysis of style using techniques inspired by modern linguistics, similar to Willats' [1997] and Durand's [2002; Willats and Durand 2005].

We choose to focus on line drawings for still pictures and more precisely on *contour drawings*. We leave aside all aspects of shading, including hatching lines or shadow lines. Contour drawing offers a great variety of styles and is often the preferred technique

This work was supported in part by "Région Rhône-Alpes" (DEREVE project and EURODOC grant).
Authors' addresses: S. Grabli, ARTIS project-team, Université de Grenoble, INRIA, Laboratoire Jean Kuntzmann UMR 5524 CNRS-UJF-Grenoble INP, 665 av. de l'Europe, 38334 Saint Ismier Cedex, France; email: stephane.grabli@gmail.com; E. Turquin, ARTIS project-team, Université de Grenoble, INRIA, Laboratoire Jean Kuntzmann UMR 5524 CNRS-UJF-Grenoble INP, 665 av. de l'Europe, 38334 Saint Ismier Cedex, France; email: Emmanuel.Turquin@imag.fr; F. Durand, MIT CSAIL, The Stata Center, 32 Vassar Street, Cambridge, MA 02139; email: fredo@mit.edu; F. X. Sillion, ARTIS project-team, Université de Grenoble, INRIA, Laboratoire Jean Kuntzmann UMR 5524 CNRS-UJF-Grenoble INP, 665 av. de l'Europe, 38334 Saint Ismier Cedex, France; email: Francois.Sillion@inrialpes.fr.

for abstraction. In addition, the line as a 1-dimensional primitive presents substantial differences with classic 0-dimensional pixels of computer graphics and therefore represents an intriguing challenge.

Our approach contributes to improving style control within the standard object-space NPR pipeline for line drawings from 3D scenes. This pipeline is made of the following stages: first, feature lines are extracted from the 3D model and properties of the scene such as visibility are computed. Then, stylized strokes are built from these feature lines. The final image is obtained by rendering these strokes. Line drawing simplification can optionally be done at this step by omitting superfluous strokes.

Our model for image creation relies on the postulate that style attributes (e.g., stroke thickness or line omission) are not chosen randomly by the artist but partly depend on properties of both the scene (e.g., nature of lines, distance to the viewer) and the drawing (e.g., local density of strokes). This postulate is implicitly assumed by many previous NPR algorithms for style depiction, such as Sousa and Prusinkiewicz [2003] and Gooch et al. [1998], however, it was never explicitly exploited nor articulated. In contrast, we make this assumption the cornerstone of our approach and explore the flexibility it offers. To develop an intuition for this postulate, we consider a few real illustrations and try to articulate their style.

The style of Figure 1(a) is characterized by the use of different line weights in the drawing. The weight is stronger when the line marks a larger depth discontinuity, that is, as the two surface patches adjacent to this line in the image plane are further from one another in the view direction.

In Figure 1(b), the artist brings forward the character by shortening the background lines, creating a "halo" effect. This time, it is the "line adjacency" property that is used to decide if a line must be shortened.

In Figure 1(c), despite the perspective effect that tends to cause line clutter, the overall illustration remains clear. The artist omits lines in regions where visual complexity is already high, thus maintaining a constant low density. This can best be seen on the paved ground. Here the line omission style attribute is correlated to the density property.

In this article we show that the programmable approach follows naturally from these considerations and represents an efficient way to get both flexible control over style and separation between style and content.

We chose to dedicate our approach to static image depiction, which means that it does notinclude any special treatment for animation such as the maintenance of temporal coherence from frame to frame [Kalnins et al. 2003; Bourdev 1998]. This choice is discussed further in Section 7.

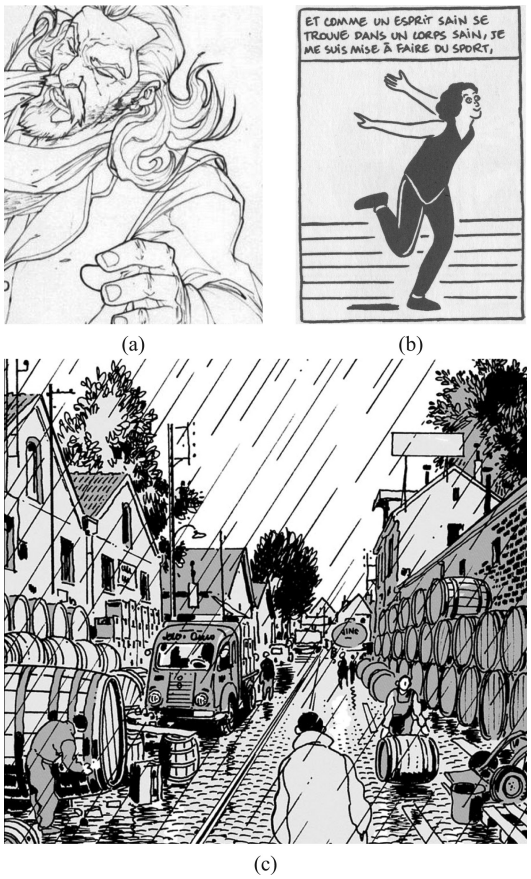


Fig. 1. These three drawings illustrate the postulate that style attributes are related to properties of the scene and the drawing. (a) Here, the main characteristic of the style is the subtle use of different line weights based on depth discontinuity. Image from "13 Chambers," courtesy of Denis Medri, © Denis Medri. (b) The artist creates a "haloing" effect by deliberately shortening the strokes around the main character: style relies partly on the property of adjacency between the different lines of the drawing. Image from "Persepolis," courtesy of Marjane Satrapi, © L'Association. (c) The artist avoids clutter around the vanishing point by omitting lines as the drawing density becomes too high: The line omission style attribute is driven by the density property. Image from "Nestor Burma," courtesy of Jacques Tardi, © Casterman.

1.2 Overview

Our main contribution is the introduction of a programmable approach to describe style in non photorealistic rendering of line drawings. More specifically, our contributions include:

—a control over style attributes of strokes, including high-level attributes like stroke topology;

—the identification of properties of the scene useful to style description;

—a structure for feature lines, the view map, that is flexible, compact, and provides the shader writer with a relevant access to scene property values;

—a new formalization of the drawing process as a sequence of programmable operations;

—the specification of line density measures for automatic line drawing simplification.

The general scheme of our approach is shown Figure 2. The input to the pipeline is a 3D mesh and the output is a set of stylized strokes that can be rendered as an image.

User interaction only takes place at the style sheet creation stage (orange arrow in Figure 2).

In the first phase we extract feature lines from the 3D scene. In our implementation, we include silhouettes, creases, borders, and suggestive contours [DeCarlo et al. 2003]. In practice, these feature lines are the real input to our method and nothing is assumed about their computation. Any set of 3D lines lying on the surfaces of the scene is a valid input.

At the same time, relevant *properties* are computed on the scene and cached for further access in the style description. Section 2.2

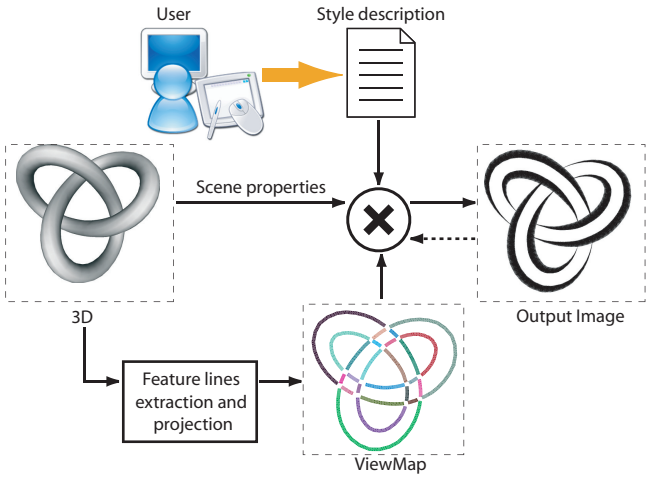


Fig. 2. General pipeline.

presents the set of properties we consider to be useful for describing style.

The feature lines are broken/merged to ensure continuity of the scene properties, and are then structured as a planar graph to reflect their 2D spatial organization in the view. The resulting graph, called the *view map*, is defined in Section 3. We also show how scene property values can be queried appropriately from view map elements. The view map is a versatile input structure to 3D-based line drawing systems and is not conditioned by the programmable aspect of our approach. Section 3 is therefore relevant to any reader concerned with the development of 3D line drawing software.

Next, using a set of *operators*, the user programs the *style sheet* which describes how edges of the view map should be turned into strokes based on the scene properties provided by the system. In particular, all style attributes such as color or thickness (see Section 2.1) are set through this process. Each operator is programmable in the sense that it is, either completely or partially, defined by the user. Section 4 presents the set of operators proposed to describe style, and their organization as a pipeline. This section is relevant to readers interested in the "shading language" itself.

Section 6 provides concrete examples and demonstrates the variety and quality of the styles afforded by our approach. Finally, in Section 7 we discuss the limitations and possible extensions of our approach.

The software (Freestyle) that was developed to support this research was released under the GPL license. All the source for both the system itself and the shaders can be found on the project Web site: http://freestyle.sourceforge.net. The Web site also contains prebuilt binaries for certain platforms and additional results. There is an ongoing effort to integrate Freestyle to the Blender package and many impressive results generated by artists of the Blender community can be found on Blender forums (e.g., http://blenderartists.org).

1.3 Link to the Authors' Previous Work

This article is an extension of a previous publication at the Eurographics symposium on rendering [Grabli et al. 2004a]. In Section 2.3 we also include a brief summary of the main ideas presented in our Pacific Graphics paper [Grabli et al. 2004b] which defines density measures that are essential to line clutter control, a significant aspect of line drawing that we believe is best expressed though our programmable system. The rest of the article extends our EGSR paper.

Section 3 is mostly new and gives a comprehensive description of the structure central to our approach, the view map. It also more thoroughly discusses the critical problem of access to scene property values from points and lines. Other key differences include Section 4.2, which motivates the choice of our basis of operators and Section 4.3 which addresses the possible organization of these operators as a pipeline. Section 6 presents several new illustrations, including code, pseudocode, and figures. Finally, as a result of the review process, the vocabulary used in this article is slightly different from that of our previous articles. In particular, we use the term "properties of the scene" in place of "information from the scene" to describe the various quantities computed on the scene (e.g., depth, position).

1.4 Related Work

Style has received much attention in NPR and is addressed in a great variety of papers. The approaches that relate specifically to our work can be divided into three categories: automatic, interactive, and mixed approaches.

Automatic approaches are the typical "black box" approaches: The style recipe is embedded in the rendering engine which can then render any 3D scene in the target style. The main advantage of these approaches is that style is not bound to a particular scene and can be reused automatically with any 3D model. On the other hand, each system targets a particular style over which the user has a fairly poor control, generally limited to a few parameter knobs. In particular, a significant change in the rendering style can only be obtained by developing another black box encoding a different style. Our approach builds upon the wealth of techniques falling into that category, such as Goodwin et al. [2007], Sousa and Prusinkiewicz [2003], Way et al. [2002], Kowalski et al. [1999], Gooch et al. [1998], and Winkenbach and Salesin [1994] but aims at offering the user more control over the rendering style.

Interactive approaches consist in modeling a style by interacting directly on the scene, in the spirit of drawing software such as Illustrator™ or Photoshop™. The visual attributes of the feature lines can usually be set through a graphical user interface which gives the user flexible and intuitive control over the style of the drawing. However, the resulting style is tied to the scene on which it was designed and can at best be used for an animated sequence, but not to render different scenes. The main research system implementing this approach was presented by Kalnins et al. [2002]. In contrast, our approach requires a programming task from the user: Style is explicitly articulated and, as a result, can be applied to any 3D scene. Kalnins et al.'s system additionally permits one to specify stroke style in an example-based manner: The user sketches a stroke whose style attributes are then learnt by the system and automatically replicated to other feature lines. We refer to such semi-interactive techniques as *mixed approaches* and discuss them in the next section.

Mixed approaches combine an interactive aspect through an editing stage and an automatic aspect through the rendering engine. During the editing stage the user interactively specifies a template of style, which is then used to automatically provide stylistic directions during rendering on any 3D scene.

So far, the techniques most commonly chosen for the editing stage are based on statistics and consist in learning the style, or part of the style, from examples. For instance, Hamel and Strothotte [1999] build histograms during an interactive session to correlate simple style attributes of hatching lines (e.g., thickness) with some properties of the scene (e.g., 3D curvature). These histograms serve as a

style template used to infer the style attributes when rendering other 3D scenes. In the same spirit, Hertzman [2002] focuses on learning statistical models of 2D curves to reproduce subtle geometry variations from an example curve.

Example-based approaches are intuitive and require little work from the user. However, we believe that only low-level aspects of style can be well captured this way, and many works have shown that subtle variations at the global level can result in significant changes in style [Winkenbach and Salesin 1994; Durand et al. 2001; Hertzmann 2001; DeCarlo and Santella 2002]. Our approach belongs to the mixed-approaches category, but in contrast to the example-based strategy, we obtain more flexibility by requiring the user to make all aspects of style explicit.

Dooley and Cohen [1990] describe a mixed approach in which the style rules are explicitely specified by the user. Their system focuses on technical illustrations and is limited to a smaller number of scene properties and style attributes than ours, making it possible for the user to specify style rules interactively through a graphical user interface. By leveraging a programmable approach, it is possible to provide a more fundamental basis of operators and to handle a larger number of scene properties as well as more complex style attributes. As a result, users of our system can describe more elaborate style rules and produce a greater variety of illustrations.

The work that most inspires our approach is the Renderman programming interface [Upstill 1989; Apodaca and Gritz 1999; Hanrahan and Lawson 1990; Cook 1984] and more precisely its shading language which permits the design of an infinite variety of rich and complex appearances. Quoting Upstill: "The key idea in the Renderman Shading Language is to control shading, not only by adjusting parameters, but by *telling the shader what you want it to do directly* in the form of a procedure" [Upstill 1989, p. 275]. We believe that such an approach could benefit NPR even more, for it requires an even greater flexibility than photorealistic rendering. While we draw inspiration from such existing programmable rendering systems, we observe that the application to line drawings entails major differences. Most importantly, the use of lines as atomic drawing elements, to which a number of procedures are applied, means that we operate on objects that have significant extent in the image, as opposed to, for example, points for Renderman. Two additional properties of line drawing also contribute to this nonlocality of rendering. First, properties of the drawing at a certain scale, such as its overall density, may affect individual lines and strokes. Second, stroke primitives carry a visual meaning that extends well beyond their actual shape, as they typically depict some region in 2D or 3D. Another difference with existing procedural shaders is that, due in part to the nonlocality just mentioned, the drawing is created by the accumulation of marks in the image and therefore is produced in a sequential manner: the order of operations, and the resulting sequence of strokes drawn, matter in the final result.

Halper et al. [2002, 2003] have introduced OpenNPAR [2002] a C++ library built on top of OpenInventor™ and designed to facilitate the development of NPAR real-time software by providing a suite of built-in components commonly used in NPR systems (e.g., standard silhouette extraction algorithms, geometric structures). The typical user of OpenNPAR would be a software engineer interested in designing and developing a complete NPR system. Although both our approaches are programmable, they differ in many respects: while their goal is to facilitate the development of applications, ours is to facilitate the direct creation of images; while their target audience is developers, ours is technical directors; an analogy with well-known systems applies accurately to our respective approaches and is helpful in understanding better their singularities: their system can be compared to OpenInventor™, ours more resembles Renderman™; the differences in intended goal, users, and philosophy between these two systems are analog to those between OpenNPAR and our approach.

Fig. 3. Stroke topology. The 3D model used here can be seen in Figure 2. (a): The planar graph for the considered viewpoint. (c) and (d): The same path in the graph drawn with the same low-level attributes but with three strokes and one stroke respectively.

Recently, Eisemann et al. [2008] presented an extension to our programmable line drawing approach [Grabli et al. 2004a] focusing on clip-art styles. They build on the operators proposed in our work and the user can program new styles for both contours and regions.

2. STYLE ATTRIBUTES AND SCENE PROPERTIES

2.1 Style Attributes

We distinguish two levels of style attributes: low-level and high-level. Low-level attributes determine the appearance of a single stroke. They are the most easily identifiable and are the most widely used in other NPR systems. We consider four low-level attributes:

—*Geometry*. The stroke backbone geometry.

—*Thickness*. The stroke thickness at each of its vertices on both sides of the backbone.

—*Color*. The stroke color at each vertex

—*Texture*. The stroke texture simulating the interaction of the tool and the medium on the support.

High-level attributes carry more global properties such as the spatial distribution of strokes or the path each stroke follows in the drawing. Although they contribute significantly to the style of a drawing, these attributes are less obvious than the low-level ones and were given less attention in previous works. We identify three high-level attributes.

Stroke topology. It is the path followed by the artist's tool while drawing a single stroke without lifting the "pen", as illustrated in Figure 3. The choice made for this path will strongly influence the drawing appearance [Willats 1997; Durand 2002]. Indeed, as shown in Figure 3(c) and 3(d), although the same set of contours was depicted with the same low-level attributes, the two drawings are visually different just by virtue of choosing different stroke topologies. It is important to observe that stroke topology is not tied to the object's 3D topology but rather relies on the 2D arrangement of the feature lines in the image, and that resulting continuous strokes may be composed of the projections of several disjoint 3D curves, as illustrated in Figure 4.

Stroke topology was studied and used in other work, for example, Isenberg and Brennecke [2006]. However, to our knowledge, it was never a style attribute over which the user has explicit control.

Line omission. As mentioned in the Introduction, in contrast to photography, drawings afford abstraction or omission of details, and artists have developed a number of pictorial techniques to prevent clutter while preserving shape and information. For example, they omit structures that are too small, exploit repetition in the scene,

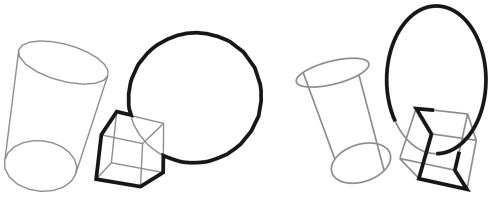
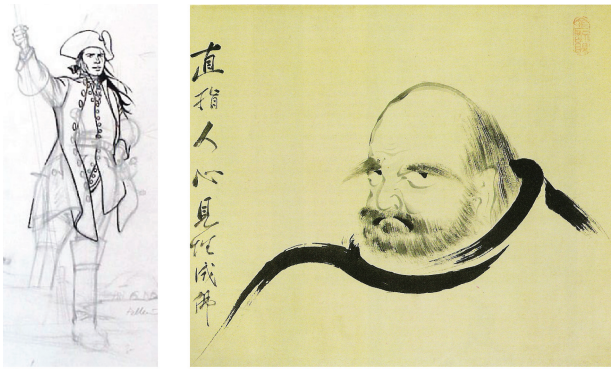


Fig. 4. Stroke paths rely on the 2D topology of the scene rather than on its 3D topology. Strokes are connected paths in 2D but not necessarily in 3D. The 3D scene on which these strokes were computed can be seen in Figure 8(a).



(a) (b)

Fig. 5. Style layering consists in combining different visual appearances within the same drawing. (a) This illustration in progress combines two visual appearances for the strokes: first, some sketchy strokes to draw the scene blueprint and second, more precise pen-and-ink strokes for the first permanent lines. For these last lines, both styles were applied leading to two sets of overlapping strokes. Image courtesy of Pellerin © Dupuis. (b) This Japanese illustration also combines two visual appearances: the lines of the face were drawn with fine faint strokes whereas the cloth is represented using very thick and strong strokes. In contrast to the previous example, the two styles never overlap. "Daruma" by Daishin Gito.

and omit texture detail. They carefully control the local amount of strokes, or *density*, in order to avoid clutter, focus attention, and create dynamism.

We identify two pictorial strategies predominantly used by artists to address clutter in line drawing of repetitive or near-regular structures: first, *uniform pruning* which ensures low complexity by omitting lines homogeneously, as can be seen in Figure 1(c) on the rooves and the pavement, and second, *indication* which suggests the overall complexity of repetitive structures by drawing in full details only a few small regions [Winkenbach and Salesin 1994], as can be seen in Figure 7(d). They differ in their focus (emphasize versus exploit repetition) and visual style (uniform versus spatially-varying drawing complexity).

Style layering. Style layering refers to the use of different visual aspects for different categories of strokes in the drawing, as illustrated in Figure 5. These categories can overlap, when some lines are represented by several strokes of different styles (Figure 5(a)), or they can can be distinct, in which case each line is represented in a single style (Figure 5(b)).

Line nature	List of line families (silhouette, suggestive contour, crease, border) to which a given line belongs.
Quantitative invisibility	Number of surfaces occluding a given point (defined by Appel [1967]). A point whose quantitative invisibility is 0 is visible.
Occluding surfaces	List of surfaces (each surface being marked by an identifier) occluding a given point.
Occluded surfaces	List of surfaces, at a given 3D point, occluded by the surface this point belongs to.
Depth	Distance from the viewpoint to a given 3D point.
Depth discontinuity	Distance between a given 3D point and the next object behind it in the view direction.
3D Normal	The 3D normals at a given surface point. There will be a single normal if the surface is smooth at the given point, two otherwise.
3D Coordinates	The X,Y,Z 3D coordinates of a given point.
Material	Material color (*e.g.* diffuse, specular) of a surface at a given point.
3D line length	The length of a feature line in world space.
Objects ID	Unique identifier associated to each surface.
Line adjacency	Connectivity information encoded in the planar graph built from the feature line projections. It is defined at each 2D vertex of this graph as the list of graph edges connected to it.
2D Curvature	The 2D curvature computed at a given 2D line point.
2D Normal	The 2D normal computed at a given 2D line point.
2D Coordinates	The x,y 2D coordinates of a given point in the image plane.
2D arc-length	The arc-length at a given 2D line point.
2D line length	The length of a projected feature line in image space.
A priori density	The *a priori* density is a measure of the visual complexity of the potential drawing.
Image aspect ratio	The aspect ratio of the final image.
Camera	Properties related to the camera (world space position, focal length, etc...).
Bounding Box	The bounding box of the scene in world space.

(a)

Causal density	It is a measure of the visual complexity of the drawing. It is updated as strokes are added to the drawing.

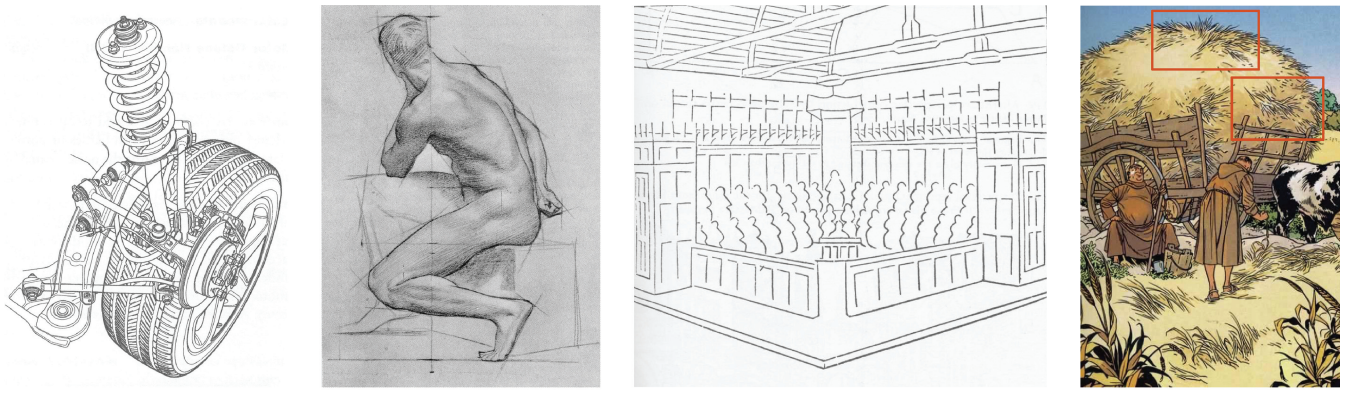
(b)

Fig. 6. (a) Properties of the scene. (b) Properties of the drawing.

2.2 Scene and Drawing Properties

We believe that style is driven by properties of the 3D scene and the drawing. Visibility or line nature are examples of properties belonging to the 3D scene, whereas drawing density is a property obtained from the drawing itself. Figure 6(a) sums up the properties that can be extracted from the 3D scene and that we believe useful in the context of style description. Some of these properties were already introduced and used in previous papers: depth, for instance, is commonly used to control line thickness, where a line closer to the eye is drawn thicker. While some properties are useful in almost all styles, like the line adjacency property that generally helps define stroke topology, some other properties only apply to very specific styles. For example, quantitative invisibility or the list of occluding surfaces is useful for technical illustrations but rarely used for artistic ones. The a priori density is a property that describes scene complexity: it is computed on the complete arrangement of lines from the view, that is, on all lines that might be turned into strokes. It corresponds to the a priori knowledge an artist has of the scene he is drawing (refer to Section 2.3)

Similarly, Figure 6(b) shows useful properties found in the drawing. It only contains the causal density, a measure of the density of strokes in the drawing. In contrast to the a priori density, the causal density is computed on the image where the strokes are rendered and measures the spatial complexity of the current state of the drawing as strokes are added, allowing for clutter control through line omission or stylization (refer to Section 2.3).



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

Fig. 7. (a) The style of this technical illustration consists in omitting lines to emphasize a subset of the assembly elements. Two properties are particularly important in this context: First, the set of occluding surfaces, which is used to discard lines whose at least one occluder belongs to the subset of interest, second, the set of occluded surfaces, which serves to omit lines that occlude the emphasized subset. Image from "Road & Track Illustrated Automative Dictionary," page 152, by John Dinkel, © Bentley Publishers. (b) The guiding lines visible in this illustration often match quasi-rectilinear feature lines, thus 2D curvature is implicitly used to decide where these guiding lines should start and stop. Image from "The Practice and Science of Drawing," plate XVIII, by Harold Spencer. (c) The lines present in this third illustration are those "turned" toward a point central to the scene: It is the 3D normal property that permits the artist to determine which lines to keep. "Interior of the Palais de Justice" by Jean Pelerin. (d) In this illustration, the artist uses *indication* to simplify the drawing: complexity is suggested through a few regions drawn in more details. This simplification strategy involves both the a priori density and the causal density. Image from "Les maîtres de l'Orge," courtesy of Van Hamme and Vallès, © Editions Glénat.

Figure 7 shows a few illustrations where the influence of scene and drawing properties on the drawing style is relatively obvious.

Most of these properties are either simple and straightforward to obtain, or can be computed using standard techniques, such as ray casting for visibility. However, we could not find satisfactory definitions for line density measures in the literature and introduced our own, the a priori density and the causal density, which, combined, allow advanced automatic simplification strategies for line drawings, as presented in Section 2.3.

Note that the addition of lighting quantity to this list does not represent any theoretical complication. In fact, the user can already implement shaders that capture lighting using the 3D normal value.

2.3 Density Measure

In this section we summarize two density measures, a priori and causal, which we introduced in an earlier article. They estimate the complexity of the view and the drawing [Grabli et al. 2004b].

2.3.1 *A Priori Density.* A priori density is measured on the *view* made of all the visible feature lines. This view represents the potential drawing as it would be if all visible lines were drawn without any style. The role of the a priori density is to give information about the complexity of the view at any location, both in a quantitative and a qualitative way. Indeed, due to the 1-dimensional nature of lines, directions play a significant role in the resulting clutter and have to be taken into account. Similarly, visual complexity is inherently linked to scale; lines appear cluttered or not depending on the scale at which they are considered.

Intuitively, we define density at a given point and at scale s as the sum of the length of the lines included in a circle of radius s normalized by the area of the circle. This normalization is important to ensure scale independence. In practice, we use a spatially-weighted average with a circular Gaussian kernel of variance σ. We also decouple information about different orientations and define density for a given direction $\vec{u}$ using a falloff w_o on the direction of the line.

The density of a set of lines $\mathcal{L}$ at a point Q, for scale σ and orientation $\vec{u}$ is then

$$d(Q,\sigma,\vec{u}) = \int_{P\in\mathcal{L}} w_d(P,Q,\sigma) w_o(P,\vec{u})\, dl, \tag{1}$$

where w_d is the normalized circular Gaussian function of standard deviation σ

$$w_d(P,Q,\sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{\|\overline{PQ}\|^2}{2\sigma^2}} \tag{2}$$

and w_o the orientation weighting function that depends on $\theta(P)$, the angle between the line tangent at point P and $\vec{u}$

$$w_o(P,\vec{u}) = \begin{cases} |cos(\frac{n\theta(P)}{2})|, & \text{if } \theta(P) \in [-\frac{\pi}{n}, \frac{\pi}{n}], \\ 0 \text{ otherwise.} \end{cases} \tag{3}$$

The notation $P \in \mathcal{L}$ in Eq. (1) is a shortcut to designate the integration space made of the set of points lying on the lines of $\mathcal{L}$. n controls the range of angles a given point of $\mathcal{L}$ contributes to.

A thorough analysis of this measure as well as implementation details can be found in Grabli et al. [2004b].

The a priori density affords a systematic approach for characterizing the structure of cluttered regions in terms of geometry, scale, and directionality. However, this measure alone does not provide fine control over stroke placement and we propose another measure of density to do so.

2.3.2 *Causal Density.* This measure complements the a priori density which cannot guarantee that the complexity of the actual drawing does not exceed a given threshold, nor that any pair of strokes is too close in the image. The causal density is updated after each stroke is drawn and can be used to stylize or decide to omit subsequent strokes. The causal density estimator works on the arrangement of strokes and, at a point Q in image I[1], it can be

[1]In the case where the strokes are drawn over a white background, $I(P) = 1 - Intensity(P)$.

written as

$$d(Q,\sigma) = \int_{P\in I} w_d(P,Q,\sigma)I(P)\,dP, \tag{4}$$

where w_d is the Gaussian function defined by formula (2). It indicates how "dark" the drawing is locally with respect to the given scale defined by σ. Each added stroke "darkens" the image an amount that depends on its color, size, thickness, and on the scale σ.

In our approach, the causal density can be queried at multiple scales but for performance reasons we have chosen not to encode it at multiple orientations. In contrast to the line density estimator it does not take directionality into account. Nonetheless, a directionality dependency might be taken into account using the directional information afforded by the a priori density measure.

3. DATA STRUCTURE AND ACCESS

To both facilitate access to properties of the scene and to make stroke creation versatile, we define a data structure, the *view map*, which combines geometry, topology, and the rest of the properties found on feature lines. This data structure makes it easy to query scene property values and manipulate feature lines in the style module.

3.1 Feature Line Generators and Apparent Feature Lines

Feature lines form the true input to our system. They are both the foundation of the drawing, since strokes are built from these lines, and the link between the drawing and the scene, as scene properties are accessed through them. In our implementation we work with some of the usual families of feature lines, namely silhouettes, creases, and suggestive contours [DeCarlo et al. 2003]. However, our approach is not restricted to a given set of lines and can easily be extended to include new types of lines. Recent studies [Cole et al. 2009; 2008] have shown that different families of lines should be combined to effectively depict a large variety of shapes and, as future work, we would like to add apparent ridges [Judd et al. 2007], suggestive highlights [DeCarlo and Rusinkiewicz 2007], and demarcating curves [Kolomenkin et al. 2008] to our set of feature lines.

We define feature lines in a topological way that is both unambiguous and can encompass any curve lying on the surface of a 3D object. To make the description easier we adopt some conventions used in Cipolla and Giblin's work [2000], and distinguish *feature line generators*, noted Γ_i, which live on the 3-dimensional surfaces of the scene, from *apparent feature lines*, γ_i, which are their projections in the image plane. We define the set Γ of *feature line generators* as the 3D curves formed by the feature points lying on the surfaces such that each curve is, first, either closed or has extremities corresponding to an ending or a branching on the surface, and, second, is regular.[2] This can be more formally written as

$$\Gamma = \left\{ \Gamma_i \;\middle|\; \begin{array}{l} \Gamma_i \subseteq \mathcal{P} \\ \Gamma_i \textit{ open and connected} \\ \forall P \in \Gamma_i, \exists \epsilon / \mathcal{B}_\epsilon(P) \cap \mathcal{P} \textit{ is a 1-manifold} \\ \Gamma_i \textit{ maximal wrt inclusion} \\ \Gamma_i \textit{ regular} \end{array} \right\},$$

where $\mathcal{P}$ is the set of feature points and $\mathcal{B}_\epsilon(P)$ an ϵ neighborhood around P. An example set Γ is illustrated in Figure 8.

[2]A differentiable curve is said to be regular if its derivative never vanishes.

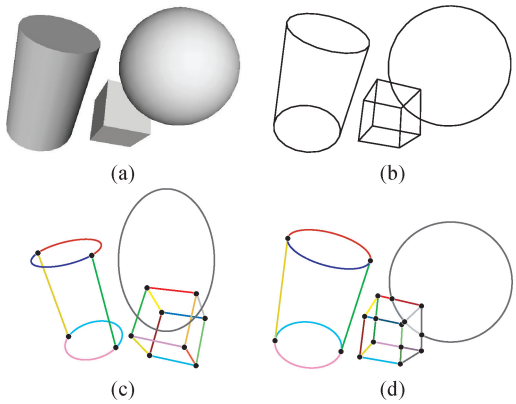


Fig. 8. (a) A 3D scene, (b) the set $\mathcal{P}$ of feature points for the considered viewpoint, (c) the corresponding initial feature lines Γ_i seen from another viewpoint. (d) The view map for this scene.

This definition excludes the possibility of multiple line generators at a given point on a surface. As an example, a point being both silhouette and crease would belong to a single generator that is both a silhouette and a crease at the same time.

The *apparent feature* lines γ_i are the projections of these curves on the image plane. Because they live in the same space as the future strokes, it is the apparent feature lines rather than the feature line generators that we work with in a style module. For example, the vertices of a stroke result from a regular sampling of an apparent line, that is, a regular sampling in image space.

In the stylization process, we want our shaders to fully utilize the properties described in Section 2.2; first, as a way to manipulate lines, for instance selecting a set of lines based on their quantitative invisibility; and second, to drive stylization attributes, for example, varying the stroke thickness proportionally to the distance to the viewpoint. So far, our definition of apparent lines does not consider scene properties whose value can vary arbitrarily along a given γ_i or even be locally undefined in case of discontinuity. This means that querying scene property values from apparent lines in this form is at best unreliable.

First, we need to ensure that there are no conflicting properties, such as different values for quantitative invisibility, on the same feature line. Second, all scene properties must be well defined at any point along each line. For these two reasons, it is required to further split feature lines to produce shorter lines along which scene properties are continuous. This will guarantee consistency in subsequent access to property values. We first consider where scene properties are naturally located to understand if and when apparent lines constitute a valid domain for their query.

3.2 Location of Scene Properties

Properties of the scene can be partitioned according to the five following geometric contexts.

Point on an apparent line. It is the first geometric context naturally available to us when working with lines in image space. Some of the properties located in that context include 2D image space coordinates, 2D normal, 2D curvature, and image space arc length.

Apparent line. It is the image-space curve itself. Properties from that context include the 2D length of the line in image space.

Point on a line generator. A lot of the properties we wish to consider when working with apparent lines are defined in 3D, on line

generators. Thus, another possible context is a point on a line generator. An important portion of the scene properties is located in that context, for instance, world coordinates, depth, depth discontinuity, quantitative invisibility, feature line family (silhouette, crease, etc.), normal to the surface, and arc length in world space.

Line generator. As with apparent lines, the line generator itself constitutes a possible location for properties such as line length in world space.

Scene. Finally, some properties are not bound to the lines themselves and are located at a more global level in the scene. They can then be queried in an a-contextual way. For example, information of camera, bounding box of the scene, image resolution, and image aspect ratio are located at the scene level. Note that these quantities can always be unambiguously queried and do not participate in refining the lines. For this reason we do not discuss it in the remainder of this section.

Because we are working with curves, we can always assume the context of a curve around a point.

3.3 Structure of Apparent Lines Based on Scene Properties

We now focus on accessing value of scene properties belonging to any geometric context from apparent feature lines.

For properties defined on apparent lines themselves or at their points, continuity can be directly studied without ambiguity. For properties at the line generator level, it can be shown that the continuity of their value along an apparent line is the same as along its line generator except at a few identified points.

PROPERTY 1. *Let $\mathcal{I}$ be a function giving the value of a property defined on a line generator Γ and let γ be the apparent line corresponding to Γ. If $\mathcal{I}$ is C^1 at a given point r of Γ and if both Γ and γ are 1-manifolds respectively around r and around the projection p of r, $\mathcal{I}$ is also C^1 at p except if p is a cusp, that is, if the tangent to Γ at r is collinear to the view direction.*[3]

The proof can be found in Appendix A. This property gives us a recipe to build a set of apparent lines with continuous scene properties. First, split the feature line generators where world-space properties are discontinuous, at cusps and at points corresponding to *multiple points* (self-intersections of the apparent line). And second, split the resulting apparent lines where image-space properties are discontinuous. We call λ the set of scene-property-based apparent feature lines λ_i built this way.

λ is a structure of the set of feature points that is as compact as possible with respect to scene properties. We show in Section 3.5 that it is also relevant to the general problem of accessing properties of the scene in the context of line drawing.

Section 3.4 characterizes discontinuity points for all the properties presented in Section 2.2 and shows a practical construction method for λ.

3.3.1 *Graph Structure.* Artists often take advantage of the 2D arrangement of lines in the image, in particular to draw strokes independently from underlying surface topology. We build a planar graph (the view map) from the set λ of apparent feature lines defined earlier to reflect this 2D arrangement. In practice, intersections of these feature lines in the image lead to graph vertices. This graph, illustrated in Figure 8(d), makes definition of stroke topology very flexible as it allows any possible path. As all properties are continuous along the lines of λ, they are also defined and continuous along each view map edge. On the other hand, they are generally discontinuous at view map vertices (which include initial feature line extremities). Vertices and edges of the view map will be referred to as *view vertices* and *view edges*, respectively.

3.4 Practical Construction of the View Map

Using the tools defined Section 3.3, we propose an analysis of our set of scene properties, and a practical construction for the view map. It consists of the two following steps: First, build the set λ of scene-property-based apparent feature lines and second, compute the corresponding planar graph in the view. We show that the view map construction is fairly straightforward since, with the exception of cusps, discontinuity points for our scene properties are a subset of the intersection vertices introduced when creating the planar graph. Figure 9 lists all our scene properties, gives the context in which they are naturally located, and characterizes their discontinuity points (we omit the study of properties that are constant at the scene level). These discontinuities may happen at Y-junctions (for line nature, quantitative invisibility and 3D normal), T-vertices (for quantitative invisibility, occluding surfaces, occluded surfaces, depth discontinuity, and any property located on line generators when the T-vertex corresponds to a self-intersection of the projected curve), and cusps (quantitative invisibility, 2D curvature, 2D normal, and any property located on line generators). These points are illustrated in Figure 10, for the specific example of quantitative invisibility. For quantities located on line generators, we systematically rely on Property 3.3 to translate continuity results from line generators to apparent lines.

We assume that the triangle meshes that compose the scene are good C^0 approximations of piecewise smooth surfaces on which we can safely use discrete average-based estimators to compute differential geometric quantities such as curvature. Because of the difficulty of computing higher-order geometric discontinuities on a triangle mesh, we choose to ignore discontinuities of order $k > 1$. For instance, a discontinuity in curvature that does not correspond to a discontinuity in normals is neglected. We also assume that the scene is viewed under a general viewpoint, which means that a small perturbation of that viewpoint does not affect the configuration of the line drawing. With piecewise smooth surfaces, silhouette generators and suggestive contour generator are smooth space curves.[4] Finally we assume that crease line generators are piecewise smooth as well. Under these assumptions, apparent feature lines are also smooth except at cusps [Cipolla and Giblin 2000].

As mentioned earlier, except for cusps, all discontinuity vertices are either vertices of the initial set Γ of line generators or vertices corresponding to intersections of apparent lines in the image plane. Therefore, once the Γ_i are available, computing the view map simply involves adding cusp vertices to these generators and building the planar graph of their apparent counterparts.

There is no fundamental difficulty in adding more families of feature lines to the view map. The main task consists in identifying how the scene properties behave at intersections with these new lines. The fact that a family is view-dependent or view-independent doest not require any additional consideration since our shaders do not hadle temporal events.

[3]Note that even if Γ is a 1-manifold, it is not necessarily the case of its projection γ which can, for instance, self-intersect. In this case, Γ has to be split to ensure that γ is a 1-manifold.

[4]To be accurate we should mention that silhouette generators actually have two types of singularities: beaks and lips [Cipolla and Giblin 2000]. However both these singularities disappear under the general viewpoint assumption and can therefore be ignored.

Line nature (generator)	Continuous along the apparent feature lines λ_i. In our case, each line generator Γ_i has a single nature, except for silhouette and crease that can coexist at a given point. A transition to/from this dual nature only happens at Y-junctions involving a crease and a silhouette [Nalwa 1988] (already a subset of the vertices of Γ).
Quantitative invisibility (generator)	Three types of discontinuities along a line: when it passes behind a silhouette (T-junctions), when it intersects a silhouette in 3D (Y-junctions), and when it passes through a cusp [Markosian et al. 1997]. These events are illustrated in Figure 10.
Occluding surfaces (generator)	Discontinuity along a line when it passes behind a contour line, *i.e.* a silhouette corresponding to an object's external contour in the image plane.
Occluded surfaces (generator)	Discontinuity along a line when it passes in front of a contour line.
3D coordinates, Depth (generator)	Continuous on line generators (continuous curves in world space), continuous along apparent lines (from property 3.3).
Depth discontinuity (generator)	Discontinuity along a line when it passes in front of a silhouette line, with no surface lying between these two lines.
3D Normal (generator)	Continuous. Discontinuities along a line generator iff it intersects a crease line on the surface. These intersections belong to the initial set of vertices of Γ so after property 3.3 this quantity is continuous along the λ_i.
Objects identity (generator)	By definition, constant on feature line generators. Indeed, since the topology of a line of Γ follows that of its surface, only one surface can be spanned by a single line generator.
Material (generator)	Continuous, for the same reason as above, assuming each object is assigned a single material.
3D line length (generator)	By definition, constant on a line generator.
2D Curvature, 2D Normal (apparent)	For smooth surfaces, discontinuities along the γ_i happen only at cusps [Cipolla and Giblin 2000]. For surfaces with sharp edges, discontinuities can also happen at points corresponding to Y-junctions involving a crease line generator. Thus both these quantities are continuous along the λ_i.
2D coordinates (apparent)	Continuous (λ_i are curves in image space).
2D arc-length (apparent)	By definition, continuous along apparent lines.
2D line length (apparent)	By definition, constant on apparent lines.
A priori density (apparent)	Considering that the *a priori* density is computed in image plane as a convolution of a Gaussian function with a bound function (the image intensity), it is continuous all along the lines of λ.

Fig. 9. Discontinuities of scene properties. For each property, the natural context is indicated in parentheses, "generator" or "apparent" for properties located or varying along, respectively, feature line generators or apparent feature lines.

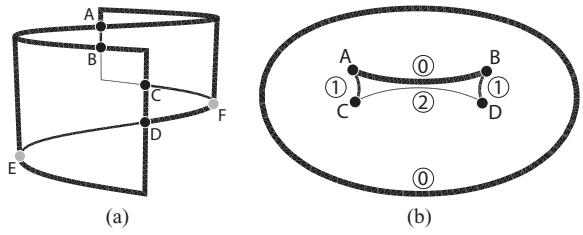


Fig. 10. The quantitative invisibility can change at T-junctions (points A,B,C, and D in (a)), at Y-junctions, (points E,F in (a)) and at cusps (points A,B,C, and D in (b)).

The view map proved useful in other illustrative applications. In particular, Eisemann et al. [2009, 2008] leverage the view map to generate 2D editable vector illustration from 3D scenes.

3.5 Access to Property Values

Since style attributes rely strongly on properties of the scene, it is critical to be able to query them from different primitives (lines, points) manipulated during the drawing process. In the general context of NPR line drawing, the difficulties related to such a mechanism result mainly from the three following facts.

—NPR eventually leads to purely 2D elements (e.g., the apparent feature lines) from which some 3D quantities must be queried.

—Scene properties are a priori discontinuous along initial apparent feature lines (the γ_i in our notation).

—The primitives that are rendered are traditionally 1-dimensional and direct access to scene properties from these primitives is often desirable (although most properties are defined at points).

In our case, the primitives that are handled by the user within the drawing process can be 1-dimensional or 0-dimensional. The 1-dimensional primitives are view edges or strokes and the 0-dimensional ones are view edge points or stroke points. We must provide functions to access scene properties from any of these primitives.

3.5.1 *0-Dimensional Queries.* As seen in the previous section, the view map structure successfully isolates points where scene properties are discontinuous (view vertices), and ensures their continuity over view edges. 0-dimensional queries can therefore be made safely at any view edge point. However, as we will see in Section 4, strokes are built as a path in the view map and can potentially span several connected view edges. As a result, strokes may also include points that correspond to view vertices and we need to consider that queries can be made at these points.

In the general case, a view vertex connected to *n* view edges can be associated to *n* different values of the vector of scene properties. However, since in our approach points always belong to lines, we know that every query at a point is performed in the context of a 1D primitive. In practice, the points handled in the drawing process always come from an iteration loop over the vertices of a stroke or view edge. This 1D context reduces the number of possible values from *n* to 2, making it easier for the user to make a proper decision, either choosing one of the two values or combining both.

In our implementation, such contextual queries to properties are made through iterators on points. These iterators embed the 1D context as they allow the traversal of the view edge or stroke the point belongs to. As a result, at any view vertex, the two possible values for a given property are available to the user. Iterators also make it possible to compute on-the-fly differential-type values.

3.5.2 *1-Dimensional Queries.* As explained earlier, 1-dimensional primitives are central to the approach and need to afford a direct access to scene properties. For properties natively located at the line level (e.g., line length) or properties that take discrete values (e.g., quantitative invisibility), 1-dimensional queries are natural. However, for properties located at points, we want to compute a single value for the whole 1D primitive. This can be done by *aggregating* several values obtained at points along the line. We provide basic aggregation operators, such as mean, min/max, variance, union, etc., as built-in operations and the user can also easily define his own aggregation operators.

As long as the manipulated 1D primitives are view edges, this process generally leads to consistent results, since properties are continuous along each of them. By contrast, in the case of strokes, the query of a single property value for the whole primitive might make no sense. For instance, when dealing with discrete properties

such as quantitative invisibility, what result should we expect for a stroke sharing different values? And which aggregation operator can be used? There is no general answer to such questions as they depend both on the involved property and on the target application. Indeed, due to the extreme variety in scene properties, although several standard aggregation mechanisms can be provided, they must be used with care. For instance, choosing the max operator to compute quantitative invisibility on either a stroke or a view edge makes sense (it returns the highest number of surfaces occluding a part of the 1D line), but using it for the nature property leads to a meaningless answer. In addition, using the mean operator to compute quantitative invisibility on a line can be correct if it is continuous all along it (case of a view edge, for instance), or incorrect, as explained before, in the opposite case.

Note that even in the case of view edges, the use of statistics, such as the average, for 1D queries can lead to unexpected results. Indeed, the return value is computed from the set of values obtained through 0D queries at points along the apparent feature line. The location for these points is defined by the regular image-space parameterization of this apparent line. But for world-space properties, we would generally rather choose query points according to a regular world-space parameterization of the feature line generator. As future work, we would like to allow for multiple-space parameterizations per line, including image-space and world-space regular parameterizations.

In essence, it is necessary to provide aggregation operators for access to properties from 1-dimensional primitives since they are often needed and are well defined most of the time; as a result they can greatly help reducing the user's coding task. However, it is the user's responsibility to properly recognize the relevance of such a query.

4. PROGRAMMABLE STYLE

In this section, we present the heart of our approach: the "shading language" provided to the user for style description. In practice, we build on top of the Python scripting language and add a set of specialized operators (select, chain, split, shade, sort, draw) that work on the view map. Together these operators allow the user to model the whole drawing process.

We define a style sheet as a set of layers, the *style modules*, which each describes part of the style. This is a natural way to break up complex styles made of several different visual appearances into simpler styles, each focusing on a single visual appearance, as illustrated in Figure 11.

We first show a simple example made of three style modules to illustrate the operators idea. Then, each operator is thoroughly described and motivated. Finally, we discuss issues related to the style module pipeline organization and synchronization.

4.1 Simple Style Module Example

Consider as an example the style sheet illustrated in Figure 12. It uses three style modules.

(1) The first module *selects* edges on the external contour of the drawing, *chains* all edges on the external contour, and *assigns* a calligraphic (direction-dependent) thickness.

(2) The second module *selects* all visible edges, *chains* edges along silhouettes, *splits* chains of edges at points of high curvature, *alters* the stroke geometry making them tangent to their center, and *assigns* a sketchy texture and standard thickness.

(3) The third module *selects* all visible edges that are not external contours, *chains* along silhouettes and crease lines, and *assigns* plain attributes to the strokes.

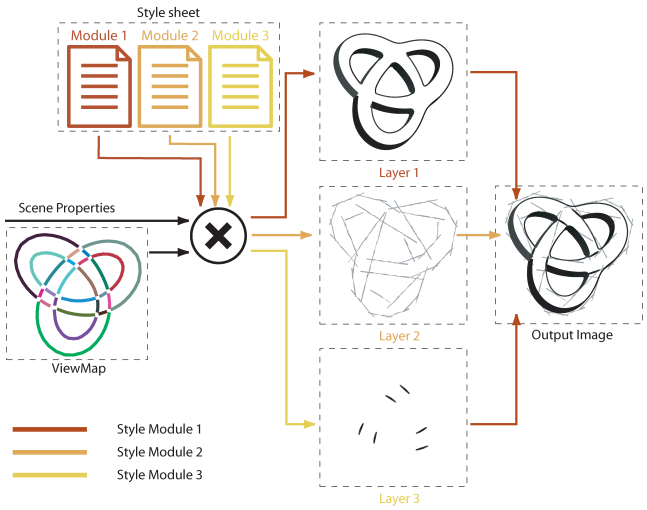


Fig. 11. Style module organization: The complex style shown on the right is the result of superimposing three simpler style modules.

Figure 13 shows the code for one of the style modules. In particular, it shows that a style module is made of a main body and of a set of style rules. As we see in the next section, *style rules* encapsulate the behavioral part of each operator and are a way to both minimize the user's coding task and to reuse existing elements. Each style module creates a layer of the final image as seen on the right of Figure 12. The bottom of Figure 13 shows the drawings obtained with this style sheet on different models. Notice the style consistency accross these illustrations.

4.2 Operators

We define a set of operators that follows from the list of style attributes established earlier. It is important that these operators strike the right balance in granularity: they must be high-level enough to easily describe a style and low-level enough not to compromise flexibility. They should also be modular to reduce the user's coding task as much as possible though reusability of existing components. We take inspiration from STL algorithms which successfully implement similar design goals. Each of our operators is a fixed algorithm/procedure parameterized by a programmable component, which we call *a style rule*, that encodes the actual behavioral part of the operator, in the same way that STL algorithms are fixed procedure parameterized by functors.[5] In our implementation, style rules are usually C++ functors derived from built-in base classes. One key difference with STL algorithms is that the set of primitives is never explicitly handled (see Section 4.3) in the style module and is therefore not part of the operators' arguments. Depending on the operator, there might be one or several style rules of different natures to embody its behavior. There are three types of style rules: predicates[6] (unary or binary, the latter for comparisons, working on either 0D or 1D primitives), functions (in particular to query scene properties, at 0D or 1D primitives), and iterators over view edges to traverse the view map.

Figure 14 shows the formal declarations of all operators (their C++ signature) and describe their input, output, and arguments.

[5]A functor is a function object.

[6]A predicate is a function that evaluates a condition and returns a boolean value.

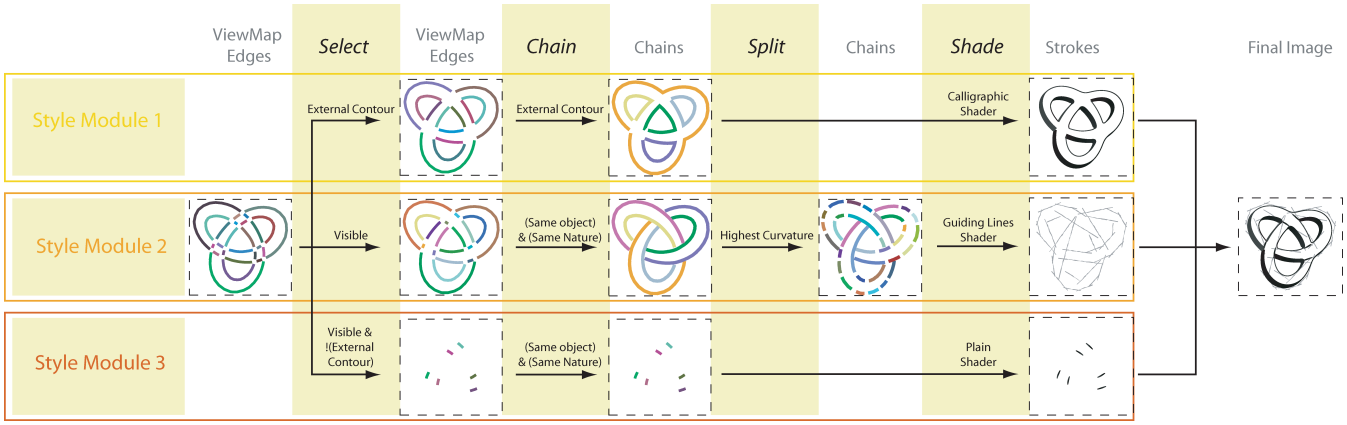


Fig. 12. Three simple style module pipelines.

```
from Freestyle import *
from logical_operators import *
from vector import *
from calligraphic_shader import *

upred = ExternalContourUP1D()

# select external contour view edges
Operators.select( upred )

# Chain using a chaining iterator
# parameterized by our predicate
Operators.bidirectionalChain(
    ChainPredicateIterator( upred, TrueBP1D() ),
    NotUP1D( upred ) )

# create = select + shade + draw
shaders_list = [
    ConstantColorShader(0,0,0,1),
    pyCalligraphicShader(1, 10, Vec2(1,1))
        ]
Operators.create(TrueUP1D(), shaders_list)
```

```
from Freestyle import *
from vector import *

class pyCalligraphicShader(StrokeShader):
    def __init__ (self,
        iMinThickness,
        iMaxThickness,
        iOri):
        StrokeShader.__init__(self)
        self._tMin = iMinThickness
        self._tMax = iMaxThickness
        self._ori  = iOri/iOri.length()

    def shade(self, stroke):
        func = VertexOrientation2DF0D()
        it = stroke.strokeVerticesBegin()
        while it.isEnd() == 0:
            v = it.getObject()
            tan = func(v)
            ori = Vec2(-tan.y(), tan.x())
            scal = min(1, max(0, ori * self._ori))
            t = max(0,self._tMin +
                scal * (self._tMax-self._tMin))
            v.attribute().setThickness(t/2.0, t/2.0)
            it.increment()
```

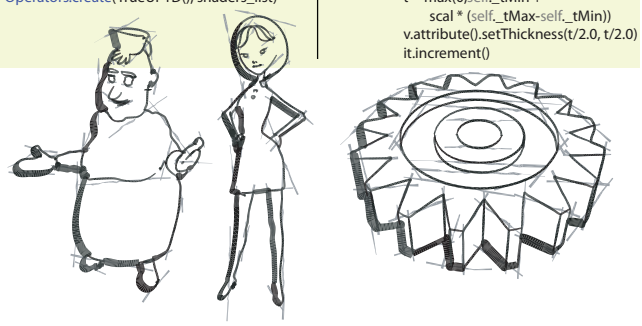


Fig. 13. Code for the first style module from Figure 12. On the left is the main body and on the right is the code of the calligraphic shader. The other style rules used in this style module (e.g., ExternalContourUP1D, ChainPredicateIterator) are built-in. The create operator on the last line is specific to our implementation and is explained in Section 4.3. The bottom drawings illustrate how the same style sheet can be applied to any 3D model.

We now describe each operator and how it relates to style attributes.

Selection operator. The *selection* operator selects a subset of an input set of 1D primitives (see Figure 14(a)). With the selection operator the user can implement the high-level style attributes of line omission and style layering. Indeed, it can be used to, for instance, select the subset of view edges that correspond to silhouettes or external contours. It can also be used on strokes to discard the ones that lie in regions that are already too dense. In practice, a selection operator extracts a subset of the active set of 1D primitives to define the new active set. Built-in predicates are provided that permit testing the properties of the scene described in Section 2.2. For example, selection can be based on quantitative invisibility, on the object ID, or on edge nature (crease, silhouette, or border).

Stroke topology operators. The view of a scene as encoded in the view map provides graph information. However, line drawings consist of 1D paths. Stroke topology operators allow the user to define such 1D paths from the view map graph. We note that there can be multiple strokes to represent the same feature line, and that strokes can span multiple feature lines. We have identified two kinds of decisions to control stroke topology. First, we must decide for each vertex of the graph which path to follow. Second, we must decide where to start and where to stop strokes. In our approach, the former is handled by *chaining* operators, and the latter by *splitting* operators. Figure 15 illustrates this process.

The formal declaration of the chaining operator as well as a description of its input, output, and arguments can be found in Figure 14(b). A chaining operator is invoked successively on all view edges in the selection, and builds a stroke (i.e., a connected list of view edges at this stage) originating from each, by traversing the view map, optionally tagging each view edge as it is processed. It is controlled through style rules specifying where to turn at a view vertex, and when to stop.

Multiple chaining of the same view edge can be desired to produce overlapping strokes that convincingly simulate sketchy looks as illustrated in Figure 16(c). The tagging mechanism mentioned previously is provided for that purpose, and controls or prevents multiple chaining. In addition, chaining can be either bidirectional or unidirectional, the former meaning that a chain extends in both directions from the first view edge. In addition, chaining can either be constrained to remain inside the selection, or unconstrained. In the latter case, each chain starts on a view edge from the selection but can contain arbitrary view edges. Figure 16 illustrates various chaining strategies for an initial selection of external silhouette edges only.

Our system provides several built-in chaining strategies, such as chaining view edges of the same nature while respecting surface topology, following contours or external contours, or chaining the same view edges multiple times (sketchy look).

`void` **select**`(UnaryPredicate1D& pred);`	
Input	A set of 1D primitives (view edges or strokes).
Output	A subset of the input.
Style rules	**pred:** A unary predicate that evaluates a condition on a 1D primitive. If that condition is true, the primitive is kept otherwise it is discarded.

(a)

`void` **chain**`(ChainingIterator& it, UnaryPredicate1D& pred);`	
Input	A set of view edges.
Output	A set of strokes. After the chaining operator, strokes are chains of view edges. Each input view edge initiates the creation of a stroke.
Style rules	**it:** An iterator over view edges whose increment method encodes the algorithm to choose the next view edge among those adjacent to the current one. **pred:** A unary predicate working on 1D primitives that decides when to stop the chain. Examples of stopping conditions include reaching a certain length, running into an occlusion or turning with too high curvature.

(b)

`void` **sequentialSplit** `(UnaryPredicate0D& startingPred, UnaryPredicate0D& stoppingPred, float sampling);`	
Input	A set of strokes.
Output	A new set of strokes.
Style rules	**startingPred:** A unary predicate working on points evaluating the starting condition. If true, a new stroke is started at the point at which the predicate is evaluated. **stoppingPred:** A unary predicate working on points evaluating the stopping condition. If true, the last new stroke that was started finishes at the point on which the predicate is evaluated. **sampling:** Since we may want to split the stroke in places other than vertices from the input model, our system operates on a sampled version of the curve with a user-controlled sampling rate. Temporary vertices at this sampling rate are iteratively created as the chain is traversed, but they are not stored permanently.

(c)

`void` **recursiveSplit** `(UnaryFunction0D<double>& func, UnaryPredicate0D& pred0d, UnaryPredicate1D& pred, float sampling);`	
Input	A set of strokes.
Output	A new set of strokes.
Style rules	**func:** A real function evaluated on points. The point realizing the function's minimum identifies the splitting location. **pred0d:** A unary predicate working on points that filters splitting candidates. This rule can be used to prevent splitting at certain points, for instance those too close to the stroke extremities. **pred:** The recursion unary predicate working on strokes that decides whether to continue the recursion or not. **sampling:** Same as the sequential splitting operator.

(d)

`void` **shade**`(Stroke& stroke);`

(e)

`void` **sort**`(BinaryPredicate1D& pred);`	
Input	A set of 1D primitives (view edges or strokes).
Output	The ordered set of 1D primitives.
Style rules	**pred:** A comparison binary predicate working on any pair of 1D primitives and based on properties of the scene such as length, depth, etc.

(f)

Fig. 14. Operators.

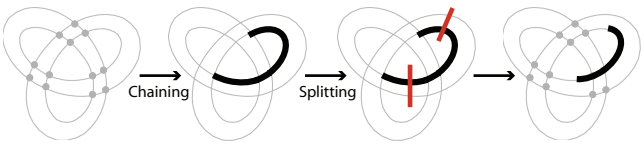


Fig. 15. The topology of the strokes is controlled through a chaining operator and a splitting operator. The stroke shown on the extreme right is built by first chaining three full view edges and then splitting the chain to obtain the desired extremities.

Once a first topology is defined for the strokes, refinement is afforded through the splitting operator. We identify two different ways of splitting a stroke: a *sequential* one and a *recursive* one.

(a) (b) (c)

Fig. 16. Examples of simple chaining predicates, applied to the set of ViewEdges on the external contour of the drawing: (a) follows external contour (b) follows silhouettes on same object (c) follows silhouettes on same object and allows multiple chaining of the same ViewEdges. Note in (b) and (c) how the chaining operation includes edges that did not belong to the original selection.

The declaration for the *sequential splitting* operator can be seen in Figure 14(c). In its basic version, sequential splitting iterates over the stroke vertices and evaluates a condition to decide where to split. A new stroke is created at each splitting point. For more flexibility the strategy is refined by decoupling the splitting conditions for the beginning and end of the stroke, and by evaluating each in a separate pass. This process can lead to strokes constituting a partition of the initial stroke (when the beginning and ending conditions are the same), a set of overlapping strokes, or a set of isolated strokes. This operator produces sketchy looks if a configuration leading to overlapping strokes is chosen.

The *recursive split* (see Figure 14(d)) has a more global behavior. It evaluates a function for each point of a stroke and splits at its minimum point. The operator is then applied recursively to the two resulting substrokes until a recursion condition is no longer true. This is the appropriate approach to split a stroke at the points of highest curvature, as seen in Figure 12.

Shading operator. Once the stroke topology is specified, the last task consists in assigning low-level visual attributes such as color or thickness. Because this step is the most similar to traditional shading systems we name the operator responsible for setting these attributes the *shading* operator. However, in contrast to classic shaders, we operate on 1-dimensional strokes rather than on 0-dimensional fragments.

The shading operator traverses a stroke and assigns or modifies its attributes. As opposed to previous operators which were implemented as fixed procedures parameterized by user-defined style rules, the shading operator is completely implemented by the user. Indeed, given the algorithmic variety of the shading operation, it seems difficult to provide it as a parameterized fixed procedure without compromising its functionality. For this reason, Figure 14(e) only contains a formal declaration for the shading operator, but no input, output, or argument description.

We remark that strokes rarely exactly follow the underlying geometry (except for technical illustration). Therefore, we also consider the spatial position of the points along the stroke as a style attribute and the shading operator can modify the geometry of the stroke backbone, in the spirit of displacement mapping techniques [Ebert et al. 1994].

Strokes can be resampled in image space to account for the various sampling rate requirements of specific styles. The attributes that were previously assigned are interpolated at the new locations. A number of atomic operations on strokes (such as removing a vertex, resampling using a given number of desired points) are available and can be used in the context of geometry modification.

Fig. 17. It is essential to order the lines prior to using causal density. (a) is a rendering with all visible lines. (b) uses causal density to omit lines that were sorted using depth discontinuity, and (c) omits lines that were not ordered.

Several shading operators can be applied sequentially to a stroke to facilitate the control of different attributes within the same style module. In addition, attributes can be assigned to a stroke in an absolute manner (where previous values are replaced) or in a relative manner (where previous values are modified).

Simple shading operators are provided, such as the assignment of constant attributes or the mapping of a given texture. Furthermore, several useful standard techniques that have long been used in NPR for sketchy effects, such as noise, stroke displacement, or smoothing [Markosian et al. 1997; Kalnins et al. 2002; Sousa and Prusinkiewicz 2003] are built-in components of our system.

Drawing operator. Finally, as we discuss further in the next section, it is important that the user can directly trigger the actual drawing of a stroke. A *drawing* operator is provided for that purpose. It is directly connected to the rendering engine and does not include any programmable part. Its input is the final set of strokes and its output can be a bitmap or vector image, a text file, etc., depending on the renderer.

Ordering operator. In our approach, some operators work in a greedy way and the sequence in which view edges or strokes are treated can influence the drawing. In the chaining operator for instance, a time-stamp mechanism prevents the reuse of a view edge. More importantly, when using the selection operator combined with the causal density for line omission, density evolves as strokes are added and it is therefore essential to treat strokes that are most important first, so that they are less likely to be removed. The *sorting* operator orders a set of 1D primitives according to a user-specified binary predicate (see Figure 14(f)) . The definition of a relevant ordering of view edges or strokes can be very tedious and requires the evaluation and integration of many kinds of properties that are best specified using a programmable approach. Figure 17 shows two simplified versions of a sunflower obtained using causal density. In the first (b), the ordering gives the highest priority to strokes with high depth discontinuities. In the second (c), strokes are drawn in an arbitrary order. Notice how in the first version the limit of the flower's center appears more pronounced and the shape of the seeds is better suggested.

4.3 Style Module Pipeline

Each style module encodes the drawing process as a sequence of calls to the operators. At any given stage of the pipeline, only one set of lines is available: Initially, it is all the view edges. After a selection operator, it is the set of selected view edges, etc. This set, referred to as "active", is never explicitly manipulated by the user since it naturally follows from the sequence of calls to the operators. Note that this active set is either made of view edges or strokes depending on whether it is considered before or after a call to the chaining operator.

The sequence of calls to the operators is partially fixed by the methodology following from the properties of the operators, which are as follows.

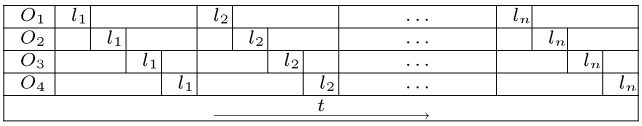
Input and output. Each operator's input and output determine the set of operators it can follow or precede. For instance, the splitting operator necessarily comes after a chaining operator since it works only on strokes.

Uniqueness. This property indicates if an operator can appear more than once or not in the pipeline. Chaining and drawing are the only operators that cannot appear more than once.

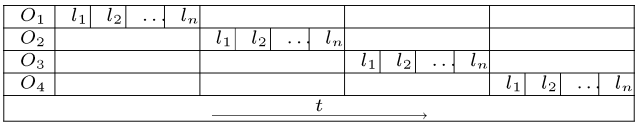
Optionality. Whereas some operators *must* appear in the style module for the drawing process to be valid, others are optional and can be omitted. Only chaining and drawing operators are not optional.

In addition, the sequence is also determined by the choice of the synchronization mechanism made for each operator, that is, the regulation of stream of data through an operator. We identify two synchronization strategies to regulate the sequential flow of a set of lines (view edges then strokes) through the pipeline of operators.

Data-based synchronization. In this scheme, each input view edge triggers a run of the full pipeline before the next one is processed, as illustrated with four operators and n lines in the following table[7]:



Operator-based synchronization. In this scheme, all primitives are processed by an operator before being passed to the next operator.



Choosing one or the other synchronization mode has repercussions on the information available during the process. Operator-based synchronization provides all the information of processing the complete sequence of lines through the previous operators to the current one. For example, it is useful to know within a chaining operator whether a given view edge was previously selected or not. Operators like the ordering operators have to consider the full set of lines at once and must be synchronized this way. However, with the operator-based synchronization, information resulting from a complete traversal of the pipeline by previous lines is obviously not available. In particular, since the last operation always consists in rendering the strokes, no information related to the current drawing can be used. The data-based synchronization is useful to address this precise need.

[7]Note that the illustration is slightly abusive since it is rarely the same element that goes from one end of the pipeline to the other; generally, we start with a view edge and end up with a stroke that potentially spans other view edges.

Consider, for instance, a pipeline for line omission using causal density made of a selection operator followed by the drawing operator. The operators must be synchronized according to the data-based strategy. Indeed, since the selection operator needs the information of stroke density, which is updated each time a stroke is added to the drawing, it is essential that each stroke is drawn (i.e., traverses the pipeline) before the next one is processed by the selection operator. More generally, as soon as information from the drawing is needed, the synchronization scheme must be chosen.

In practice, we want some operators to be synchronized one way and some others to be synchronized the other way. The two modes therefore coexist in the style module pipeline that consists of several subpipelines, each working with its own synchronization mode. It is mainly the need for information from the drawing that determines how an operator is synchronized. Note that an operator that is called several times in a style module can have different synchronization strategies in the different subpipelines. The style module pipeline Φ we chose is made of two subpipelines Φ_{op} and Φ_{dat}, which respectively use the operator-based synchronization and the data-based synchronization.

$$\begin{aligned}\Phi_{op} &= \Big([sort][select]^*\Big)^* \; chain\; ([sort][select]^*\Big)^* \setminus \\ &\quad [split]^*\Big([sort][select]^*\Big)^* \\ \Phi_{dat} &= [select]^*[shade]^*draw\end{aligned}$$

We implemented the entire Φ_{dat} subpipeline as a single extra operator *create*. This hides the details of synchronization from the user. The set of style rules that drives the create operator is the union of all the style rules appearing in the Φ_{dat} subpipeline. The create operator is controlled via the combination of the style rules that control the embedded operators.

5. MARK BACK END

The mark system is orthogonal to our programmable line drawing approach. Our mark rendering engine uses OpenGL for its interactive view and Postscript for high-quality resolution-independent output.[8] Such an output is especially desirable for technical illustrations and also allows further editing in vectorial packages such as Illustrator™. Strokes are rendered as triangle strips, defined by the geometry of the backbone and the thickness values at its vertices, as shown in Figure 18(a). Standard techniques are used to prevent singularities of the offset curve at high curvature [Strothotte and Schlechtweg 2002, Chapter 3].

We use actual stroke textures as alpha maps to increase visual quality. The use of transparency alone allows us to control the color of each stroke, as specified by its attributes. We use OpenGL blending modes to emulate various physical medium types. In practice, we render the inverse of the image, so that a blank canvas corresponds to $(0, 0, 0)$. This facilitates the use of blending and the simulation of the subtractive nature of most media. We use a simple replace mode for thick media such as oil paint. Additive blending (which becomes subtractive in our inverse context) is well-suited for wet materials such as ink. Finally, the *minimum* blending mode provided by OpenGL 1.2 [Woo et al. 1999] can imitate graphite and other dry media. Figures 18(b) to 18(d) illustrate these various blending modes.

[8]Note that the Postscript specification doesn't support texture mapping, which prevents realistic media simulation as afforded by OpenGL.

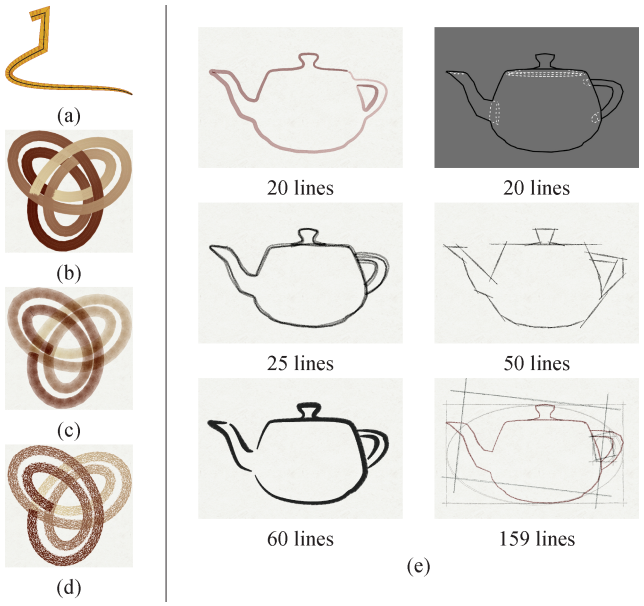


Fig. 18. Strokes are rendered as triangle strips using OpenGL, as shown in (a) where the stroke's backbone is drawn in black. Using different OpenGL blending modes, we simulate thick media (b), wet media (c), and dry media (d). (e) Lines of code required to model various styles.

A background canvas texture can also be applied. However, it is rendered only for the final drawing and does not affect the density computation.

6. IMPLEMENTATION AND RESULTS

We implemented our core system in C++ and chose the Python interpreted language as our style description language. A Python library including operators, built-in style rules, and functions to access properties of the scene is provided to the user for style description. The implementation of this library as well as the application itself are written in C++. We chose two languages instead of one so as to differentiate more the style modules from the system. In addition, these languages are both object-oriented and show nice compatibility features that offer essential interaction capabilities. In particular, objects and functions defined in C++ can be used in Python, and C++ classes can be specialized in Python and run back in C++. Thanks to this last feature, style rules can be user-defined in Python and passed as arguments to our C++ operators. We used Swig [Beazley 1996] to generate the binding between the C++ core system and Python style descriptions.

The silhouette lines are computed according to Hertzmann and Zorin's technique [2000] which leads to smooth 3D lines that better fit stylization parameterization requirements than polygonal approximations. Although the evaluation of the visibility for these lines is problematic and we could not find in the literature a perfectly robust approach for their computation, we obtain satisfactory quality by leveraging information given by the view map. First, we compute the visibility at a set of sample points along a given view edge using ray-casting combined with a subdivision approach inspired by Hertzmann and Zorin [2000]. Then, since we know that each view edge has constant visibility, we proceed to a vote among these samples to determine the view edge visibility.

It takes between a few seconds and a few minutes to compute the view map for a model of approximately 50K polygons using ray

```
def edgeStop (x, sigma):                          #for anisotropy
    return exp(-x*x/(2*sigma*sigma))              #Gaussian

class pyDiffusion2Shader(StrokeShader):
    def __init__(self, lambda1, sigma, nbIter):   #constructor
        StrokeShader.__init__(self)               #from parent
        self.lambda = lambda1                     #diffusion rate
        self.nbIter = nbIter                      #nb of iteration
        self.sigma = sigma                        #anisotropy scale

    def shade(self, stroke):                      #shader itself
        for k in range (1, self.nbIter):          #diffusion loop
            it = stroke.strokeVerticesBegin()     #create iterator
            while it.isEnd() == 0:                #loop on vertices
                v=it.getObject()                  #get vertex
                c=curvatureInfo(it)               #local curvature
                n=normalInfo(it)
                dV=self.lambda * c *
                    edgeSTop(c, self.sigma)       #[Desbrun, Perona]
                v.SetPoint(v.getPoint()+n*dV)     #update coordinates
                it.increment()                    #increment vertex
```

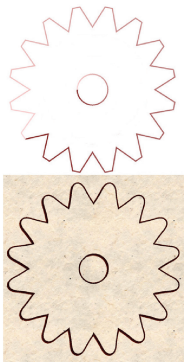


Fig. 19. Left: Python code for a user-defined anisotropic smoothing operator. Right: The shader is applied on the external contour of a gear. Top: without smoothing. Bottom: with smoothing.

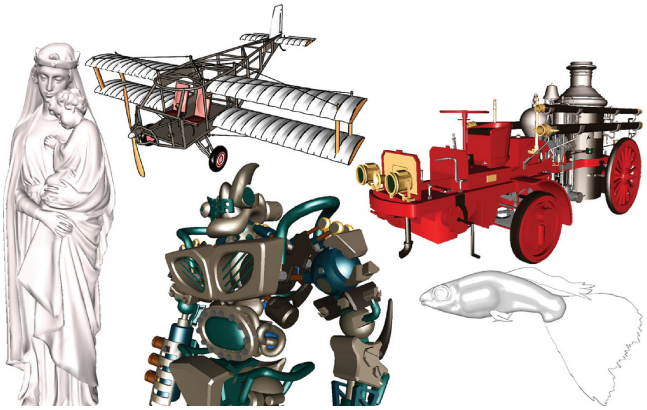


Fig. 20. Most of the 3D models used for examples in this section.

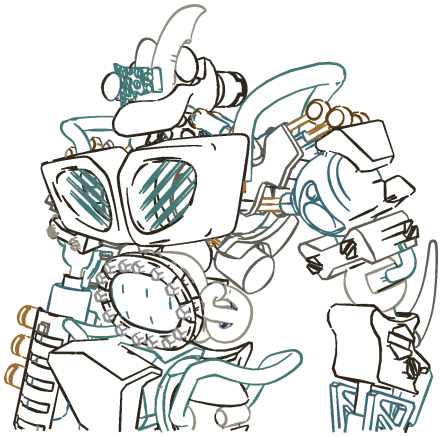


Fig. 21. Stroke color is automatically computed from material color.

casting for visibility computations. Stroke creation takes a similar amount of time, depending on the number of strokes and on the style module complexity. The use of density induces a significant drop in performance because of frame buffer read-back cost.

A more interesting measure of our system's performance is the time needed to develop a style module. As an example, we spent a total of three hours producing the images in Figure 23 (this includes style module coding, experimentation, and aesthetic evaluation). The style modules contain about 500 lines of code, half of which are straightforward use of built-in mechanisms. The system includes many standard functions, predicates, shaders, and chaining iterators that facilitate the elaboration of new styles. The development of any new base object can benefit from standard sampling, noising, smoothing, and 1D integration components. Similarly, all scene properties are accessed through standard contextual query mechanisms. In Figure 18(e) we show a range of styles and indicate for each the approximate number of lines of code that were written.

6.1 Code Example and Results

Figure 19 shows the actual code of a shading operator that performs feature-preserving smoothing using anisotropic diffusion inspired by mesh smoothing techniques [Desbrun et al. 1999, 2000]. It uses curvature flow and moves vertices in the direction normal to the stroke at a rate proportional to local curvature. An edge-stopping function prevents smoothing at sharp curvature points. The parameters of the shader are declared in the constructor `init`. Normals and curvature are part of the available scene properties, as described in Section 2.2. We use an iterator to traverse the vertices of the strokes. Stroke geometry is modified using the method `setPoint()`. More code examples, including the code used to generate all the illustrations in this article, can be found on the freestyle Web site.

In the remainder of this section, we show images that were produced with our system to illustrate its flexibility as well as the great variety of styles that can be described. Although the effects shown in these images are often new and can only be produced through the precise control offered by our programmable system, our contribution is not in these effects but in the unified and flexible approach to stylized line drawing rendering. The 3D models used to generate the illustrations are shown in Figure 20.

Figure 21 illustrates a very common style in cartoon animation where the color of the strokes is a variation of the surface material color. We can easily implement this style as a shader. For example, the color of the material is translated in the LUV space [Wyszecki and Stiles 1982] in the negative L direction if its actual L value is higher than a user-defined threshold (to get a darker color), and in the opposite direction otherwise (to get a brighter color).

Technical illustration is one of the main applications of our approach. Figure 22 demonstrates how a subpart of an assembly can be emphasized. A first style module draws all the lines that do not belong to the subset of interest in an imprecise way using light-tone strokes. Four other style modules draw the lines of that subset with different attributes depending on their visibility and which surfaces are occuding them (surfaces from the subset itself or not). In addition, standard technical illustration conventions are evoked through the use of different colors for creases and silhouettes. The resulting style clearly emphasizes the subset of interest while keeping its surrounding environment.

Figure 23 shows a complex style made of eight styles modules applied to a Virgin statue model. Three of these modules alter the geometry of the strokes to generate a blueprint and sketchy lines, two modules select and draw small strokes in high density areas and the remaining modules display longer strokes in a lighter tone, also using density value and a fade along the Y axis.

Figure 24 uses similar attributes as the previous example.

Figure 25 imitates the Japanese line drawing style using two style modules simulating different brushes. Both use line shortening and

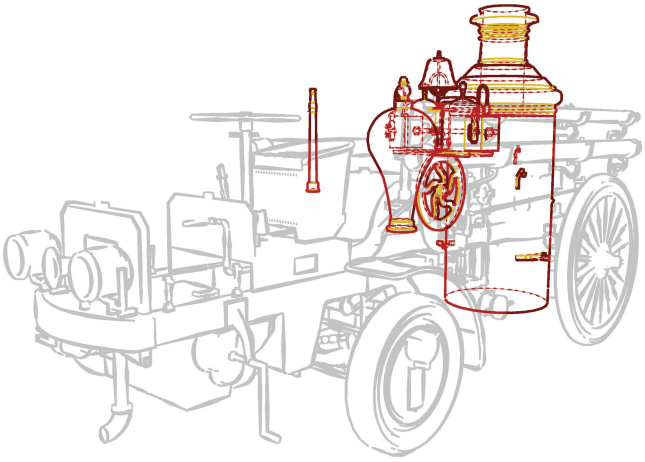


Fig. 22. In this technical illustration, the occlusion property is used to emphasize a subset of elements.

(a) (b) (c)

Fig. 23. This Virgin model was rendered with a "Renaissance-like" style using a combination of several style modules. The illustrations show the progressive addition of the three groups of style modules.

string tapering. The large brush layer also uses density evaluation to avoid clutter.

Figure 26 illustrates how 3D properties can be used to drive advanced chaining. A chaining iterator that depends on depth value forms strokes that delimitate clusters of objects lying at different depth.

Figure 27 illustrates the use of a complex chaining operation as well as causal density to build a simplified representation of a dense structure with occlusion. For the grid, a chain is created for each bar by connecting all view edges including short occluded ones. These chains are sorted by length and subjected to the causal density operator with a variable Gaussian kernel size depending on depth. This allows us to keep only a single stroke for each bar and to remove exactly half of the bars. The compressor behind the grid also uses an advanced chaining iterator to avoid the dashed line effect shown in Figure 28(b).

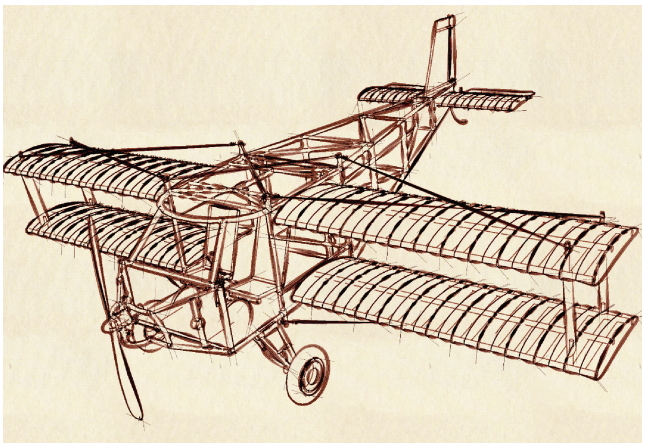


Fig. 24. Renaissance-like technical illustration.

Fig. 25. Japanese line drawing style.

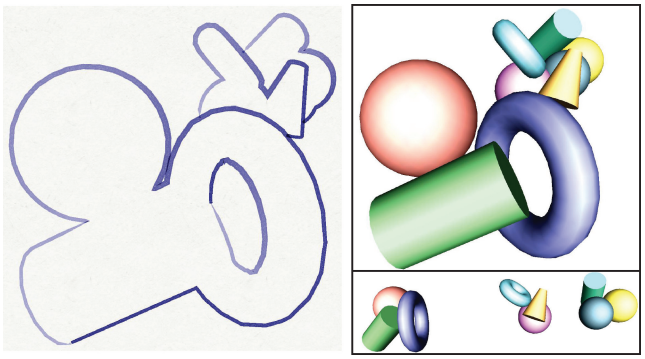


Fig. 26. Left: Chaining based on depth value that draws strokes around object groups in the foreground, middle, and background. Right (top): The 3D scene from the same viewpoint. Right (bottom): The scene from a top viewpoint emphasizing the distances between objects.

Figure 29 illustrates three different style modules for line drawing simplification using density. For all style modules, visible view edges are first selected and chained together to form an initial set of visible strokes. For the pipeline on the left, these strokes are directly processed through a selection operator that discards strokes

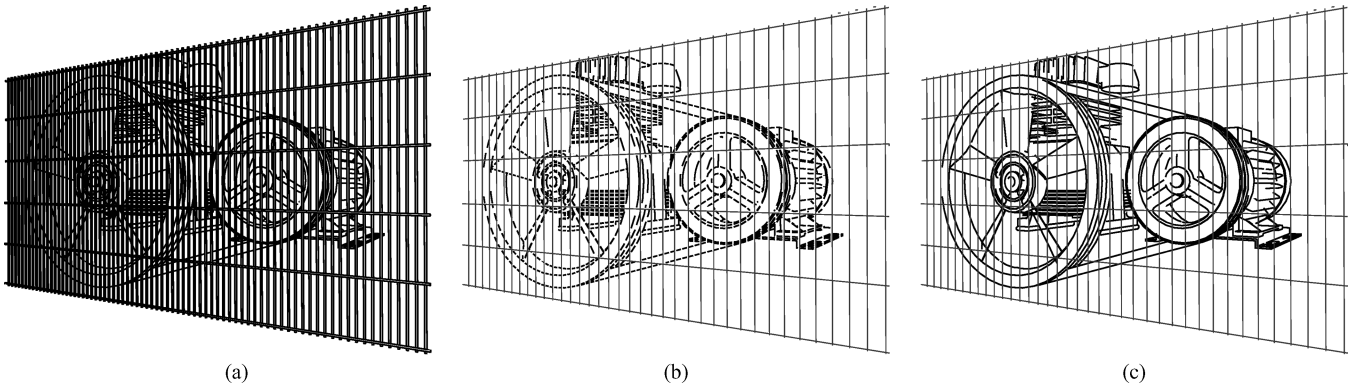


Fig. 27. Density-based simplification combined with advanced chaining. (a) The initial set of lines. (b) The grid was uniformly simplified to enhance the drawing clarity. Without any treatment, this results in a dashed-line effect for the compressor lines whose visibility is not up-to-date anymore (see close-up in Figure 28). (c) An advanced chaining operator chains through small occlusions to fill holes due to grid line omission.

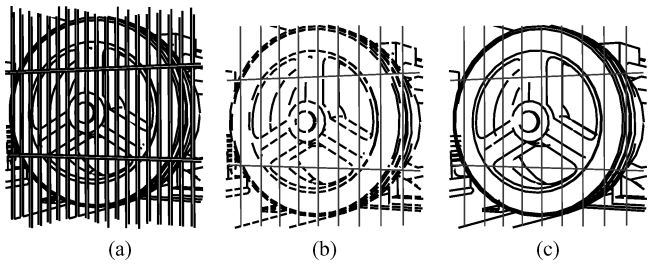


Fig. 28. Close-up on the small wheel of the compressor from Figure 27, drawn with all lines (a), after simplification of the grid (notice the dashed line effect) (b) and after chaining (c).

for which the causal density is too high. Note that without proper sorting, the result looks rather unstructured. By sorting the strokes according to depth, the result is visibly improved, as shown in the middle pipeline of Figure 29. This style module performs uniform pruning simplification and its pseudocode is shown in Figures 30(a) and 30(b).

A slight modification of this pipeline allows one to transform the simplification strategy from uniform pruning to indication. This is done by changing the threshold used in the final selection to vary proportionally to the gradient computed on the a priori density. This results in lines lying near the boundaries of dense regions to be drawn first. This style module is shown on the right of Figure 29 and its pseudocode is shown in Figures 30(a) and 30(c).

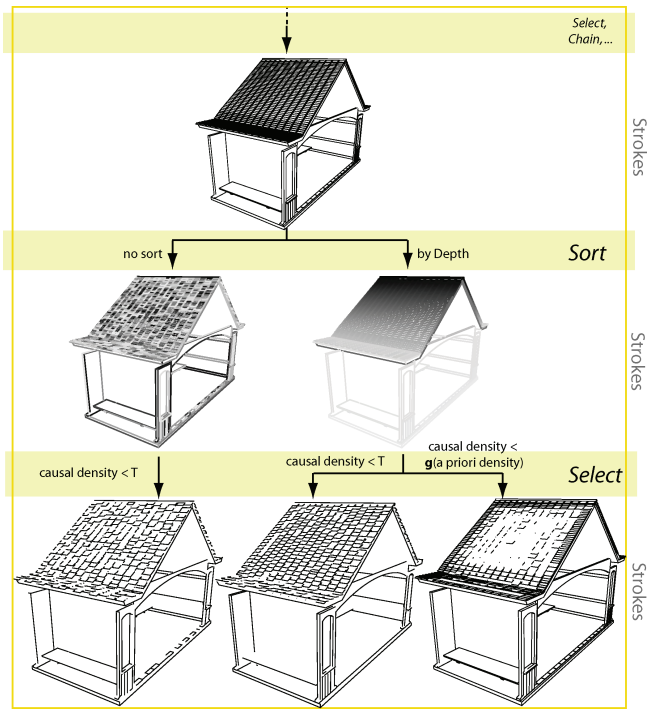


Fig. 29. Three pipeline examples performing line drawing simplification using density. The pipeline on the left implements uniform pruning without any sorting. The middle pipeline is similar but additionally sorts the strokes based on their depth, which clearly improves the result. The pipeline on the right corresponds to the indication strategy and is a slight variation of the previous one: the threshold T for omitting strokes is not constant anymore but varies proportionally to the gradient g computed on the a priori density. As a result, strokes are kept near the boundaries of dense regions.

7. CONCLUSIONS

7.1 Summary of Contributions

We described a new formulation of the image creation process for generating line drawings from 3D models. Our approach is based on programmable operators that can be arranged to create style modules. Through the development of this approach we made the following contributions:

—a better control over essential style attributes, especially high-level attributes such as line omission or stroke topology;

—the identification of scene properties useful for style description;

—the specification of a structure (the view map) of the feature lines appropriate for describing style based on scene properties, both in terms of line manipulation and access to scene property values;

—a decomposition of the drawing process as a pipeline of programmable operators;

```
select(Visible)
chain(Visible)
sort(Z)
shade(plain)
select(LowCausalDensity)
draw()
```
(a)

```
def LowCausalDensity(1DElement e)
  if CausalDensity(e) < threshold
    return true
  return false
```
(b)

```
def LowCausalDensity(1DElement e)
  t = APrioriDensityGradient(e)
  if CausalDensity(e) < a*t+b
    return true
  return false
```
(c)

Fig. 30. Pseudocode for the line omission style module implementing either the uniform pruning strategy or the indication strategy. (a) The style module body common to both strategies. (b) The style rule for uniform pruning, and (c) for indication. In these examples, built-in components and user-defined ones are written in blue and red respectively.

—the use of density measures to control automatic line drawing simplification strategies.

7.2 Limitations

View map topology constraint. Although the strokes do not necessarily follow the topology of the 3D scene, the 3D feature lines do. In that respect, the view map defines the drawing process as a 3D to 2D projection rather than as a 2D depiction of the 3D world. In particular this means that the topology of the view cannot be changed and that no extra line can appear in the drawing, whereas both features could be useful, especially for abstraction [Barla et al. 2005].

Still images. Our approach is currently dedicated to still images and cannot handle animation properly. In particular, temporal coherence issues arise with view-dependent lines such as silhouettes, whose position on the surface, geometry, and even topology change from frame to frame resulting in visual artifacts. Techniques exist to maintain temporal coherence by propagating parameterization from frame to frame [Kalnins et al. 2003]. However, most of the time this would force the system to disregard the style specification. This then requires a compromise between style specifications and temporal coherence. For this reason, we found it preferable to focus on static images as we think that current techniques designed to maintain temporal coherence do not suit our application. The next section includes some thoughts about an alternative approach to temporal coherence.

7.3 Future Work

Here are several directions for future work we would like to explore.

Generalize the programmable approach. Our system is currently limited to contour drawings but many other aspects of NPR could benefit from a programmable approach. We find especially interesting the research of programmable operators for stylized depiction of regions, as studied by Eisemann et al. [2008]. A programmable mark back end could also be of great interest. Finally, feature line extraction itself could be made programmable so as to facilitate the exploration of new feature lines.

Temporal coherence. As explained earlier, we believe that existing techniques [Kalnins et al. 2003; Bourdev 1998] are too intrusive for a system like ours. We would like to leave the decisions relative to temporal coherence to the user: in the context of an animated sequence (rather than an interactive session), visibility events, which are the source of temporal discontinuities, could be added as parameters to the operators. The user could then account for temporal events in the style description. The time management would therefore be a full part of the pictorial style.

APPENDIX

A. Proof for Property 3.3

PROOF. We prove this property by using the inverse function theorem to show that the perspective projection is a local C^1 diffeomorphism from Γ to γ except at cusp points.

For two 1-dimensional smooth manifolds M and N, the local inverse function theorem states that for a C^1 map h from M to N and for a point $r \in M$ such that the derivative Dh of h is nonzero at r, there exists an open neighborhood U of r such that $h_{|U} : U \to h(U)$ is a C^1 diffeomorphism. By writing Γ in its parametric form

$$\begin{array}{rccc} r: & I \subset \mathbb{R} & \longrightarrow & \mathbb{R}^3 \\ & t & \longmapsto & (x(t), y(t), z(t)) \end{array}$$

the perspective projection from Γ to γ is a C^∞ map that can be defined as

$$\begin{array}{rccc} h: & \Gamma & \longrightarrow & \gamma \\ & (x(t), y(t), z(t)) & \longmapsto & \left(\frac{x(t)}{z(t)}, \frac{y(t)}{z(t)}\right). \end{array}$$

Let us now study the derivative Dh of h along Γ. Note that at a given point r of Γ, Dh is a linear map from the tangent space $T\Gamma_r$ of Γ around r to the tangent space $T\gamma_{h(r)}$ to γ around the projection of r and that both of these spaces are of dimension 1.

The derivative of h at a point r of Γ is $D(h \circ r)(t)$, which can also be written as $Dh(r) \cdot Dr(t)$ where $Dr(t) = (x', y', z')$ is the tangent of Γ at r and $Dh(r)$ is the Jacobian matrix of the perspective projection at r

$$Dh(x, y, z) = \begin{pmatrix} \frac{1}{z} & 0 & -\frac{x}{z^2} \\ 0 & \frac{1}{z} & -\frac{y}{z^2} \end{pmatrix}.$$

Since Γ is a regular curve, $Dr(t)$ is nonnull. Therefore

$$\begin{aligned} Dh(r) \cdot Dr(t) = 0 &\Leftrightarrow \begin{pmatrix} \frac{1}{z} & 0 & -\frac{x}{z^2} \\ 0 & \frac{1}{z} & -\frac{y}{z^2} \end{pmatrix} \cdot (x', y', z')^T = 0 \\ &\Leftrightarrow \begin{cases} \frac{x'}{z'} = \frac{x}{z} \\ \frac{y'}{z'} = \frac{y}{z} \end{cases}. \end{aligned}$$

In other words, the derivative of the perspective projection along Γ vanishes at a point r only when the tangent to Γ at r is collinear to the view direction, that is, when r is a cusp. As a result, if both Γ and γ are 1-manifolds, after the inversion theorem, the perspective projection is a C^1 diffeomorphism between a line generator Γ and its projection γ except at cusp points.

This means that, outside cusps, a parameterization of γ is a *reparameterization* of Γ. Consequently, a quantity that is C^1 on Γ is also C^1 on γ. □

ACKNOWLEDGMENT

Thanks to E. Eisemann for helping us formulate our ideas on access to scene property values and to T. Judd for proof-reading the final version of this document. Thanks also to the anonymous reviewers whose comments helped improve this document considerably. M. Curioni, J.-L. Peurière and T. Kajiyama are to be praised for undertaking the heroic task of integrating Freestyle to the open source 3D package Blender.

REFERENCES

APODACA, A. AND GRITZ, L., Eds. 1999. *Advanced Renderman : Creating CGI for Motion Pictures*. Morgan Kaufmann.

APPEL, A. 1967. The notion of quantitative invisibility and the machine rending of solids. In *Proceedings of the 22nd National Conference*. ACM Press, 387–393.

BARLA, P., THOLLOT, J., AND SILLION, F. 2005. Geometric clustering for line drawing simplification. In *Proceedings of the Eurographics Symposium on Rendering*.

BEAZLEY, D. M. 1996. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop*. USENIX, 129–139.

BOURDEV, L. 1998. Rendering nonphotorealistic strokes with temporal and arc-length coherence. M.S. thesis, Brown University.

CIPOLLA, R. AND GIBLIN, P. 2000. *Visual Motion of Curves and Surfaces*. Cambridge University Press. Cambridge, UK.

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.

COLE, F., SANIK, K., DECARLO, D., FINKELSTEIN, A., FUNKHOUSER, T., RUSINKIEWICZ, S., AND SINGH, M. 2009. How well do line drawings depict shape? *ACM Trans. Graph. 28*.

COOK, R. L. 1984. Shade trees. In *Proceedings of the SIGGRAPH Conference*.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Trans. Graph. 22,* 3.

DECARLO, D. AND RUSINKIEWICZ, S. 2007. Highlight lines for conveying shape. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*.

DECARLO, D. AND SANTELLA, A. 2002. Stylization and abstraction of photographs. *ACM Trans. Graph. 21,* 3.

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the SIGGRAPH Conference*.

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. 2000. Anisotropic feature-preserving denoising of height fields and bivariate data. In *Proceedings of the Graphics Interface Conference*. 145–152.

DOOLEY, D. AND COHEN, M. F. 1990. Automatic illustration of 3d geometric models: Surfaces. *IEEE Comput. Graph. Appl. 13,* 2, 307–314.

DURAND. 2002. An invitation to discuss computer depiction. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*.

DURAND, F., OSTROMOUKHOV, V., MILLER, M., DURANLEAU, F., AND DORSEY, J. 2001. Decoupling strokes and high-level attributes for interactive traditional drawing. In *Proceedings of the Eurographics Workshop on Rendering*.

EBERT, D., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY. 1994. *Texturing and Modeling: A Procedural Approach*. Academic Press.

EISEMANN, E., PARIS, S., AND DURAND, F. 2009. A visibility algorithm for converting 3d meshes into editable 2d vector graphics. *ACM Trans. Graph. 28,* 3, 1–8.

EISEMANN, E., WINNEMÖLLER, H., HART, J. C., AND SALESIN, D. 2008. Stylized vector art from 3d models with region support. *Comput. Graph. Forum 27,* 4.

GOOCH AND GOOCH. 2001. *Non-Photorealistic Rendering*. AK-Peters.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the SIGGRAPH Conference*.

GOODWIN, T., VOLLICK, I., AND HERTZMANN, A. 2007. Isophote distance: A shading approach to artistic stroke thickness. In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering (NPAR'07)*. ACM Press, New York, 53–62.

GRABLI, S., DURAND, F., AND SILLION, F. 2004b. Density measure for line-drawing simplification. In *Proceedings of Pacific Graphics*.

GRABLI, S., TURQUIN, E., DURAND, F., AND SILLION, F. 2004a. Programmable style for npr line drawing. In *Proceedings of the Eurographics Symposium on Rendering*.

HALPER, SCHLECHTWEG, AND STROTHOTTE. 2002. Creating non-photorealistic images the designer's way. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*.

HALPER, N., ISENBERG, T., RITTER, F., FREUDENBERG, B., MERUVIA, O., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. Opennpar: A system for developing, programming, and designing non-photorealistic animation and rendering. In *Proceedings of the Pacific Conference on Computer Graphics and Applications*. 424–428.

HAMEL, J. AND STROTHOTTE, T. 1999. Capturing and re-using rendition styles for non-photorealistic rendering. *Comput. Graph. Forum 18,* 3, 173–182.

HANRAHAN, P. AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proceedings of the SIGGRAPH Conference*.

HERTZMANN. 2001. Paint by relaxation. In *Proceedings of the Computer Graphics International Conference (CGI)*. 47–54.

HERTZMANN, A., OLIVER, N., CURLESS, B., AND SEITZ, S. M. 2002. Curve analogies. In *Proceedings of the 13th Eurographics Workshop on Rendering (EGRW'02)*. Eurographics Association, 233–246.

HERTZMANN, A. AND ZORIN, D. 2000. Illustrating smooth surfaces. In *Proceedings of the SIGGRAPH Conference*.

ISENBERG, T. AND BRENNECKE, A. 2006. G-strokes: A concept for simplifying line stylization. *Comput. Graph. 30,* 5, 754–766.

JUDD, T., DURAND, F., AND ADELSON, E. 2007. Apparent ridges for line drawing. *ACM Trans. Graph. 26,* 3, 19.

KALNINS, MARKOSIAN, MEIER, KOWALSKI, LEE, DAVIDSON, WEBB, HUGHES, AND FINKELSTEIN. 2002. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Trans. Graph. 21,* 3.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent stylized silhouettes. *ACM Trans. Graph. 22,* 3.

KOLOMENKIN, M., SHIMSHONI, I., AND TAL, A. 2008. Demarcating curves for shape illustration. *ACM Trans. Graph. 27,* 5, 1–9.

KOWALSKI, M., MARKOSIAN, L., NORTHRUP, J. D., BOURDEV, L., BARZEL, R., HOLDEN, L., AND HUGHES, J. 1999. Art-Based rendering of fur, grass, and trees. In *Proceedings of the SIGGRAPH Conference*.

MARKOSIAN, L., KOWALSKI, M., TRYCHIN, S., BOURDEV, L., GOLDSTEIN, D., AND HUGHES, J. 1997. Real-Time nonphotorealistic rendering. In *Proceedings of the SIGGRAPH Conference*.

NALWA, V. 1988. Line-Drawing interpretation: A Mathematical framework. *Int. J. Comput. Vision, 2, 2*, 103–124.

OPEN NPAR. 2002. http://www.opennpar.org/.

SOUSA, M. AND PRUSINKIEWICZ, P. 2003. A few good lines: Suggestive drawing of 3d models. *Comput. Graph. Forum 22,* 3.

STROTHOTTE, T. AND SCHLECHTWEG, S. 2002. *Non-Photorealistic Computer Graphics. Modeling, Rendering, and Animation*. Morgan Kaufmann.

UPSTILL, S. 1989. *The Renderman Companion*. Addison-Wesley, Reading, MA.

WAY, D.-L., LIN, Y.-R., AND SHIH, Z.-C. 2002. The synthesis of trees in chinese landscape painting using silhouette and texture strokes. *J. WSCG 10*.

WILLATS, J. 1997. *Art and Representation*. Princeton University Press.

WILLATS, J. AND DURAND, F. 2005. Defining pictorial style: Lessons from linguistics and computer graphics. *Axiomathes 15,* 2.

WINKENBACH, G. AND SALESIN, D. 1994. Computer-Generated pen-and-ink illustration. In *Proceedings of the SIGGRAPH Conference*.

WOO, M. ET AL. 1999. *OpenGL Programming Guide: The Official Guide to Learning OpenGL Ver. 1.2*, 3rd Ed. Addison-Wesley, Reading, MA.

WYSZECKI, G. AND STILES, W. S. 1982. *Color Science: Concepts and Methods, Quantitative Data and Formulae*, 2nd Ed. John Wiley and Sons.

Received April 2008; revised September 2009; accepted November 2009