Eurographics Symposium on Rendering (2004)
H. W. Jensen, A. Keller (Editors)

Programmable Style for NPR Line Drawing

Stéphane Grabli[1], Emmanuel Turquin[1], Frédo Durand[2], and François X. Sillion[1]
[1] ARTIS†/GRAVIR-IMAG - INRIA [2] MIT

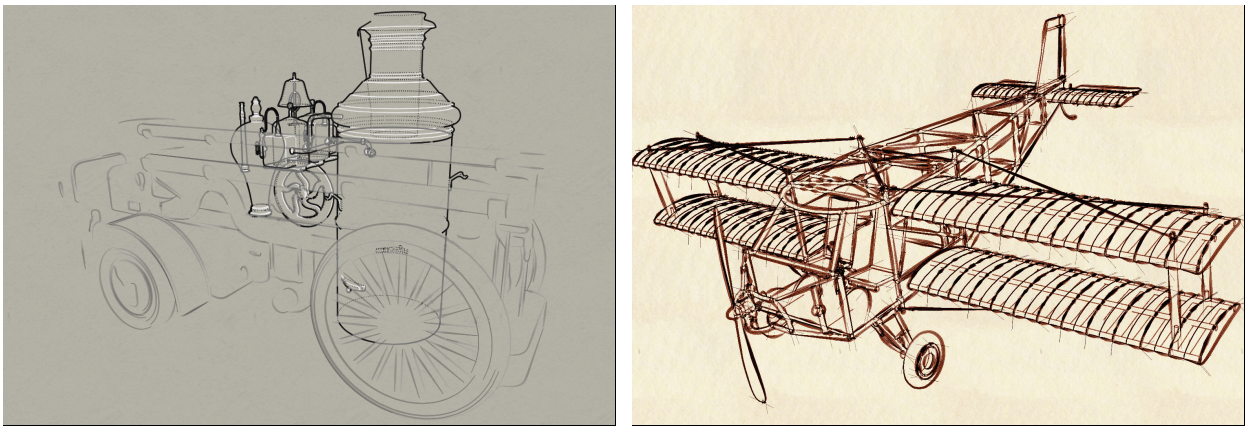


Figure 1: *A programmable approach unifies the specification of line drawing styles, allowing for various complex styles descriptions.* ***Left:*** *Use of occlusion information to emphasize some parts in the scene.* ***Right:*** *A sketchy style based on the use of "construction lines".*

Abstract

This paper introduces a programmable approach to non-photorealistic line drawing from 3D models, inspired by programmable shaders in traditional rendering. We propose a new image creation model where all operations are controlled through user-defined procedures. A view map *describing all relevant support lines in the drawing and their topological arrangement is first created from the 3D model; a number of style modules operate on this map, by procedurally selecting, chaining or splitting lines, before creating strokes and assigning drawing attributes. 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. Finally, we propose new density control strategies where strokes can be adapted or omitted to avoid visual clutter. We illustrate the components of our system and show how style modules successfully capture stylized visual characteristics that can be applied across a wide range of models.*

1. Introduction

The appeal of line drawing lies in its expressiveness and abstraction. It is widely used in contexts as different as technical and scientific illustration, appliance manuals, maps, signs and art. In addition to its pleasing aesthetic qualities, line drawing can prevent clutter, focus attention on relevant parts and omit superfluous details. The field of Non-Photorealistic Rendering [GG01, SS02, Dur02] has proposed a variety of techniques to create compelling line drawings from 3D models. Unfortunately, these methods are generally hard-coded in monolithic software, and while a variety of parameters usually allows the user to vary the style of the drawing, there is a need for more flexibility and power in the specification of pictorial style.

In contrast, the shading languages available in photorealistic renderers such as Pixar Renderman [Coo84, HL90, Ups89, AG99] permit the design of an

† ARTIS is a research project in the GRAVIR/IMAG laboratory, a joint unit of CNRS, INPG, INRIA and UJF

© The Eurographics Association 2004.

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" [Ups89] p. 275.

In this paper, we introduce a flexible programmable approach to NPR line drawing. Similar to traditional shaders, the style of a line drawing can be specified by implementing procedures that describe how the silhouettes and other feature lines from the 3D model should be turned into strokes. We focus on pure line drawing of static scenes, leaving hatching, tonal modeling and temporal coherence as future work. Historically, the development of shading languages has dramatically facilitated the exploration and development of realistic shading. We hope that NPR can similarly benefit from a flexible programmable approach. Our system has been distributed to a few research institutions and will be made broadly available on the Internet during fall 2004.

1.1. Related work

Style has received much attention in Non-Photorealistic Rendering and computer vision. Most NPR techniques offer a control of style through a set of *parameters*, e.g. [Her98], or through direct user interaction, e.g. [KaM*02]. In particular, the WYSIWYG NPR approach [KaM*02] presents an approach for interactively "painting" strokes directly on the 3D model. In contrast, our approach is programmable and the style of the drawing is controlled via procedures. This requires programming skills similar to that of a Technical Director in production rendering, but it provides more control on style. This includes the use of scene information, full control over stroke topology and placement and it allows style to be independent from the 3D model. In terms of traditional appearance modeling, this is the difference between 3D painting of texture maps and procedural shaders. Both approaches are now used in conjunction, and we believe that our work is a similar complement to interactive data-driven techniques such as WYSIWYG NPR.

Machine learning is a popular approach to capture style from examples, e.g. [TF97, FTP99, HJO*01, KaM*02]. Machine-learning approaches usually focus on the low-level and statistical aspects of styles. Hamel and Strothotte [HS99] capture and re-use style using templates that control the parameters of a line renderer. In addition, the subtle variation of style within an image has been shown to be crucial to make the image more lively and focus attention, e.g. [WS94, DOM*01, Her01, DS02]. In contrast to these approaches, the user of our system specifies style with a set of rules that govern the drawing process.

Our line-drawing approach builds upon the wealth of techniques developed in NPR, e.g. [MKT*97, HZ00, DFRS03].

Little attention has been given to the potential of a programmable approach to NPR. In [KMN*99], Kowalski et al. use procedural stroke-based textures to render 3D complexity using indications [WS94]. Although their system is dedicated to several specific types of objects, it introduces many of the concepts our more general approach builds upon. The research work closest to ours is the OpenNPAR system [HSS02, Ope02], an API for the development of real-time NPR software. The authors also used their system to develop an impressive graphical user interface for the exploration of simple styles [HSS02]. Our approach focuses on line drawing and on more complex style development. In summary, their system is a programming toolbox in the spirit of Open-Inventor [SC92] while our approach is inspired by Renderman shaders.

Recently, a major company has released information about their in-house NPR system [Tee03]. It allows for very flexible ink rendering of 3D model and uses a philosophy of "shaders" similar to our system. Their technique is not as flexible as our approach and focuses mostly on ink lines, but this restriction allows them to better handle temporal coherence. Production software has recently been augmented with "toon" shaders, and non-photorealistic styles can be obtained with Pixar RenderMan, e.g. [AG99], p330 and p477.

While we draw inspiration from existing programmable rendering systems such as Renderman, we observe that the application to line drawing 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 e.g. pixels for Renderman. Two additional properties of line drawing also contribute to this non-locality 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 non-locality 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.

1.2. Overview of contributions

We introduce a programmable approach to line drawing from 3D models, and propose a consistent architecture for a drawing system. The novelty of the proposed model lies in its representation of drawing as a *process*, its explicit realization of the sequential nature of drawing, and its exploitation and support of the non-locality of line drawing primitives. We present a consistent decomposition of the drawing process into individual operators for the selection, chaining, splitting and ordering of lines and strokes. We show how the resulting strokes can be further processed and modified to

© The Eurographics Association 2004.

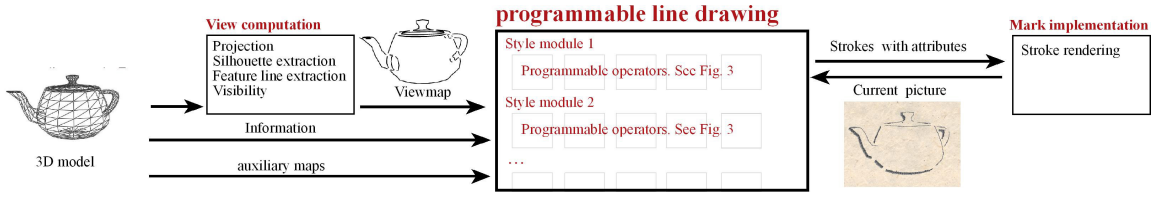


Figure 2: *System architecture.*

obtain interesting graphical styles. Finally we manage the information necessary at all stages of the drawing process.

2. Design decisions and architecture

Our work is based on the assumption that line drawing is a sequential process where decisions are based on information afforded by the scene and the current drawing. These decisions are often explicit, such as in technical illustrations, where precise thickness or color attributes are applied to the different natures of feature lines [GSG*99]. They can also be implicit, and a goal of our research is to make them explicit and embed them into software. We address two key elements in line drawing: how strokes are drawn, and which ones are drawn. The appearance of strokes is determined by their attributes such as thickness, color, or wiggliness. Equally important is the choice of what strokes to draw, and what path they should follow. In addition, style variations in different parts of the drawing have a large impact on the resulting visual aspect. This motivates the main features of our system architecture:

- Layering and selection mechanisms permit the application of the appropriate style to sub-parts of the picture
- Information from the 3D input scene and the current drawing is available at all stages
- The topology and path of the strokes can be finely controlled
- Programmable operators provide control of the stroke geometry and attributes.

2.1. Architecture

The overall architecture is summarized in Fig. 2. The input is a polygonal 3D model, possibly augmented with information such as color or the subjective importance of each object.

The first stage is the computation of a *view*: an arrangement of curves in the image plane, which will support all the elements of the drawing (Fig. 3 a). Following Willats [Wil97], we assume that the line drawing is based on a number of feature lines of the model, which we first identify using established techniques. Our current implementation uses silhouettes and feature lines such as creases or suggestive contours [HZ00, IFH*03, DFRS03]. These lines are projected in the image plane, using a standard visibility computation algorithm, and they are arranged in a planar graph. An important objective is to retain maximum flexibility in the definition of drawing marks, thus the graph must

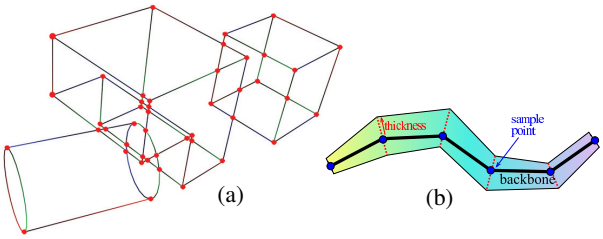


Figure 3: ***(a)*** *View map data structure. The ViewEdges are represented with a color gradient and ViewVertices are the red dots.* ***(b)*** *Stroke representation.*

encode complete topological data such as line intersections and neighboring information. This leads to a slightly simplified, thus more compact, graph which we call a *view map*: our atomic unit is a set of connected line segments in the projected view that share the same nature (i.e. silhouette, crease, or contour) and the same visibility status (occluders and occludees). Such sets are called *ViewEdges*, and connect *ViewVertices* representing points of interest. A ViewVertex can either correspond to an actual vertex of the scene, to the visual intersection of two edges (T-vertex), or to an end junction (cusp). With the view map we also compute a number of auxiliary images such as a depth buffer or line density maps as explained later.

The second stage is the heart of our approach, the programmable line drawing process. It takes as input the view map and the auxiliary images, and creates the *strokes* that compose the drawing. Strokes are described by a one-dimensional backbone and a set of attributes that vary along its length (thickness, color, transparency, texture, etc.), e.g. [HL94, SS02]. This is illustrated in Fig. 3(b). Our choice of ViewEdges as the atomic element in the viewmap realizes an excellent compromise between generality (although a ViewEdge has consistent visibility status and nature along its entire length we shall see that is is still possible to create multiple strokes to depict a single ViewEdge) and compactness (when possible or necessary a stroke can span multiple ViewEdges, for example a single stroke can depict the external outline of an object).

Finally, the mark system is responsible for the actual rendering of the stroke primitives with their attributes. For example, the same stroke with given thickness and color can be rendered with different mark styles such as crayon or oil painting.

We call the set of procedures that implement a given pictorial style a *style sheet*. It is usually decomposed into a se-

© The Eurographics Association 2004.

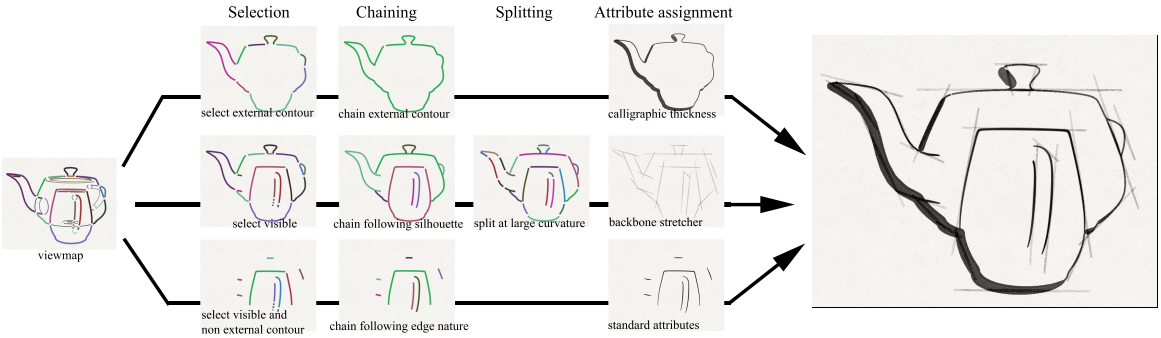


Figure 4: *Operators used within three style modules constituing a simple style sheet. See also Figure 5.*

ries of *style modules* that are responsible for sub-parts (or "layers") of the drawing. Style modules are a natural way to vary the style within the drawing. For example, the main object can be drawn using a different style module from the rest of the scene, or hidden lines can be drawn with a different style module from the visible ones.

The operation of a style module consists mainly in controlling the topological characteristics of strokes as well as their visual attributes. Extensive experimentation with stylized drawings led us to define the following set of topological operators, which we have found to be sufficient to create all the stylistic effects we have attempted to realize. First, the *selection* of a set of ViewEdges, which is the mechanism used to restrict actions to a subset of a drawing. Then *chaining* operators let us build a one-dimensional sequence of ViewEdges, starting from a given edge in the selection. Next, *splitting* operators let us refine the drawing elements by breaking chains at appropriate locations (e.g. points of high curvature); Strokes directly correspond to the chains resulting from this topology-definition process. Finally, a last class of operators *assigns attributes* to the strokes (e.g. color, width, texture, transparency. . .). These operators are entirely programmable, and are applied iteratively in a pipelined manner.

2.2. A simple example

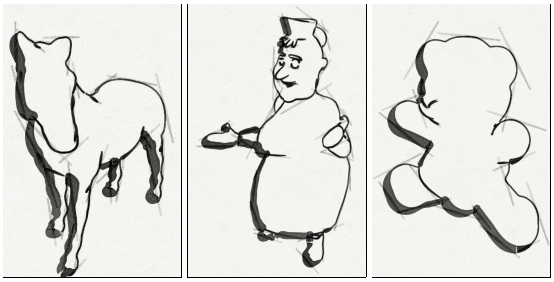


Figure 5: *Results obtained with the style sheet of Figure 4.*

Consider as an example the style sheet illustrated in Figure 4. 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, **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, **assigns** a sketchy texture and standard thickness.
3. The third module **selects** all visible edges that are not external contour, **chains** along silhouettes and crease lines, **assigns** smoothly varying width to the strokes.

Each style module creates a layer of the final image as seen on the right of Figure 4. Figure 5 shows the drawings obtained with this style sheet on different models. Notice how a consistent visual aspect is obtained.

2.3. Language

We chose to use an interpreted language, Python, rather than a C++ API for our style description language. This allows for interactive development and exploration of style modules without recompilation. More importantly, it clearly draws the line between the programmable style interface and the low-level technical aspects of the system. We extend Python with our subset of style definition instructions and classes, e.g. operators (section 3) and data structures (section 2.4), augmented with all information access mechanisms (section 2.6). The user's coding task mainly consists in overloading functions, predicates or shaders and in using operators and built-in helpers.

2.4. Data Structure

As discussed above, the uniqueness of line drawing is that it requires handling one-dimensional objects such as `ViewEdges` and `Strokes` in addition to simpler zero-dimensional points.

The ViewMap is encoded as a graph data structure, composed of `ViewVertices` and `ViewEdges`. The traditional adjacencies between them are stored.

The central objects manipulated by our operators are one-dimensional chains that will eventually lead to the creation

© The Eurographics Association 2004.

of strokes. A `Chain` describes a path in the ViewMap graph (Table 1). It is a connected set of `ViewEdges`. It does not necessarily start at a `ViewVertex`, which is why it also stores an initial and final point. A `Chain` can implicitly access points by generating samples along its `ViewEdges`. For memory efficiency and for sampling flexibility, these sample points are not stored. Sample points are parameterized by their arclength.

The `Chains` are finally derived into `Strokes` that contain additional information for the drawing creation. In contrast to `Chains`, `Strokes` store an explicit set of sample points for their backbone (see Fig. 3 b). A set of *attributes* is stored for each of these sample points (thickness, color, and transparency.)

```
Chain
  List of ViewEdges
  Sample point
    initial and final
  Double length
```

```
Stroke : derives from Chain
  Array of sample
      2D vertices [nb_ samples]
  Array of attributes [nb_samples]
    Thickness [left, right]
    Color
    Transparency
  Mark Rendering style
```

Table 1: *Chain and stroke data structures.*

The appropriate sampling of strokes is necessary to capture attribute variation. We provide a stroke resampling mechanism that takes as input a maximum length between samples and generates new points when needed. Attributes that were previously assigned are interpolated at the newly-created samples. We currently use linear interpolation, but smoother interpolation could also be used. The notion of attribute sampling rate is closely related to Renderman's shading rate [Ups89, AG99].

2.5. View computation

Similar to the philosophy of shading languages, our approach assumes nothing about how the view map is computed, as long as it provides the adjacency data structure described in Section 2.4. In our implementation, the view map is computed in three steps.

We first extract all relevant lines from the model, with respect to the view. We compute silhouettes using the technique by Hertzman and Zorin [HZ00]. Suggestive contours follow the method proposed by DeCarlo et al. [DFRS03]. We also include creases (defined by edges whose vertices share a location but not a normal) and boundary edges (edges with only one adjacent polygon).

The 2D intersections of these feature lines are then computed and define `TVertices`. The `ViewVertices` include these `TVertices`, the vertices of the mesh that form corners and the cusps. A `ViewEdge` is an arc of the view map linking two `ViewVertices` (see Fig. 3 a), and is represented as a connected set of line segments as described in the previous section.

Location	Data
3D Scene	3D coordinates normal, 3D curvature color, material ID object ID, object importance
Auxiliary maps	local average depth item buffer local depth variance local view density
ViewMap	2D coordinates,ViewEdge length ViewEdge type, Adjacency information Quantitative invisibility Occluded object (for silhouettes) Occluding objects (for hidden edges) depth discontinuity (for silhouettes) 2D orientation, 2D curvature
Current drawing	Local stroke density

Table 2: *Information provided by our system.*

Finally, we compute the quantitative invisibility of each ViewEdge [App67, MKT*97]. Hidden-lines are not eliminated at this point because some styles can elect to display them.

2.6. Information access

As pointed out by Hanrahan and Lawson [HL90], defining the possible exchange of information at the interface between the rendering program and the shading modules is a crucial decision. As discussed above, non-photorealistic rendering requires even more information, partially because the process tends to be less local. We decided to make rendering information available to all operators, so that it can influence all of their decisions.

Information is always queried in the context of a one-dimensional support element (which can be a ViewEdge, a Chain, or a Stroke depending on the operator we are in). It can be queried at a given point, or globally for the support element, in which case simple statistics about the queried quantity are made available (i.e. average, extremal values, and variance). The one-dimensional context is often needed when evaluating some information at a point. Indeed, evaluating the 3D normal, for example, at a T-vertex is undefined unless the 1D context allows us to remove the ambiguity. In our technique, 0D information is always queried in the context of a 1D chain or stroke.

Information can come from four different sources: the 3D scene, auxiliary maps, the viewmap, and the current drawing, as summarized in Table 2. Types of available information include scalars, vectors (normal direction), colors, and image maps.

© The Eurographics Association 2004.

The information afforded by the 3D scene is similar to that provided by traditional shading languages, and also includes object identifiers (to treat different objects differently), and an optional subjective object importance that is important to separate style from content. For example, a repair-manual drawing server can use an importance tag to draw the failing part with more emphasis.

Differential information such as 2D normal and 2D curvature are computed from the viewmap. Note that these differential properties are provided in the context of a 1D element, and that they might not be well-defined outside of this context. In particular, the curvature at a ViewVertex depends on the context chain.

We also use a set of auxiliary maps: an item buffer provides information such as average local depth or object identification at any point. Multiresolution line density maps are created by rendering the viewmap, counting the number of edges drawn at each pixel, and building image pyramids. In addition, the current drawing is refreshed as the drawing creation proceeds, and the local stroke density can be queried. This current density information is computed upon request, using a parameterized Gaussian smoothing operator to allow queries at different scales.

3. Programmable line drawing

The heart of our approach is the programmable style modules. They permit the explicit specification of the rules that govern the drawing process for both stroke topology and stroke attributes. Recall that such modules are applied sequentially to obtain successive layers in the drawing. A style module works on the view map and produces a set of strokes through a pipeline organization as illustrated in Fig. 4. Some of the operators (e.g. selection) apply to any 1D element and can be used at any stage of the pipeline, whereas others (e.g. chaining) only apply to a specific type and must therefore be called at specific locations in the pipeline. In this section we first describe the operators that deal with the topological creation of strokes. Then, we detail the last class of operators, attribute assignment. A style module is built as a sequence of calls to these various operators.

The system must provide the user a convenient way to specify the rules that apply to each operator. In our implementation we heavily use functors [Ale01], working either on 0D or 1D elements, as rules specifiers. Although we provide numerous basic rules for the different operators, all of them can be user-defined in Python. The different types of rules applying to the different operators are summarized in Fig. 6.

3.1. Stroke creation operators

Selection Selection is fundamental to layer the drawing, apply a style to a sub-part of the features lines or to omit

```
Selection
  Selection predicate (1D element)
Ordering
  Comparison predicate (1D element, 1D element)
Chaining
  Function nextEdge (currentVertex)
Splitting
  Sequential
     End predicate, Start predicate
  Recursive
    Function(sampleVertex) to maximize,
    splitting interdiction predicate(sampleVertex)
    recursion predicate(1D chain),
Attribute assignment (a.k.a. shader)
  shade(stroke), method that modifies the fields of stroke
```

Figure 6: *Specification of our operators.*

lines that are unnecessary. We provide selection mechanisms through a *selection* operator. This operator works on any 1D element (ViewEdge, Chain, or Stroke). In practice, a selection operator extracts a subset of the active set of 1D elements to define the new active set. The selection rule is specified as a unary predicate on a 1D element. Built-in predicates are provided that permit the test of information described in Section 2.6; For example, selection can be based on quantitative invisibility, on the object ID, on the nature (crease, silhouette, or border). These predicates can be combined with classical logic operators. Developers can also implement new predicates based on more complex functions of the scene. The selection operator is most of the time used at the beginning of our programmable pipeline, directly after the viewmap computation. However, as mentioned earlier, it is available at any stage of the pipeline, and can be useful to refine selection after chaining, when information about the the potential topology of a stroke is accessible.

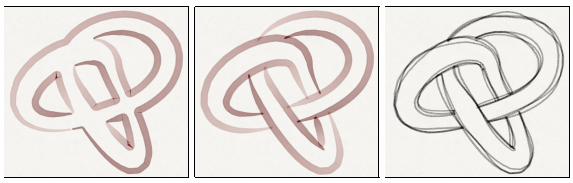
Chaining The view of an object, as encoded in the viewmap, provides graph information, while line drawings consist of 1D paths. The choice of the path and length of strokes has important repercussions on the appearance of the drawing [Wil97, Dur02]. We note that there can be multiple strokes to represent the same feature line, and that strokes can span multiple feature lines (see the external contour in Fig. 4). When creating a stroke, we have identified two kinds of decisions: First, we must decide for each vertex of the graph which path to follow. Second, we must decide where to start or stop strokes, or where to split them in order to generate shorter strokes. In our approach, the former is handled by chaining operators, and the latter by splitting operators.

Chaining operators create connected lists of ViewEdges, which we call *chains*. A chaining operator is invoked successively on all ViewEdges in the selection, and builds a chain originating from each, optionally tagging each ViewEdge as it is processed. A chaining rule must answer two questions: when to stop, and where to turn at a ViewVertex. In our implementation, we chose to embed this rule as an iterator [GHJV95]. The incrementation method of this iterator decides which is the next ViewEdge among those adjacent to the ViewVertex. The iterator stopping criterion decides whether the chain should be stopped. For example, it can

© The Eurographics Association 2004.

stop when a certain length is reached, if an occlusion is encountered, or when the curvature is too high.

Drawing style can allow multiple strokes to overlap or not. Strokes overlapping can be useful to produce sketchy looks as illustrated in Figure 7 (c). We provide a tagging mechanism to control or prevent multiple chaining of the same ViewEdge. In addition, chaining can be either bi-directional or unidirectional, the former meaning that a chain extends in both directions from the first ViewEdge. Second, chaining can either be constrained to remain inside the selection, or can be unconstrained. In the latter case, each chain starts on a ViewEdge from the selection but they can contain arbitrary ViewEdges. This can be useful for example to select only edges on the external silhouette of an object, but to allow chaining to extend to (unselected) internal silhouette as shown in Figure 7. Our system provides several standard chaining strategies, such as chaining ViewEdges of same nature, following contours or external contours, chaining several times the same ViewEdges (sketchy look), as built-in.



(a) (b) (c)

Figure 7: *Examples of simple chaining predicates, applied to the set of ViewEdges on the external contour of the drawing:* ***(a)*** *follow external contour* ***(b)*** *follow silhouettes on same object* ***(c)*** *follow silhouettes on same object and allows multiple chaining of same ViewEdges. Note how in cases* ***(b)*** *and* ***(c)*** *the chaining operation includes edges that did not belong to the original selection.*

Chain splitting As discussed above, a given chain might be depicted with multiple strokes of smaller size. This is the role of the *chain splitting* operator. It takes a chain as input, and creates a number of strokes that depict it. The rule given to this operator mainly decides where the chain should be split.

We developed two different strategies to choose the length of strokes and split points. In the sequential split, we traverse the chain sequentially and decide to split based on a predicate such as maximum length, nature of vertex, or local density. This mechanism is easy to specify but only takes its decision in a greedy way and based on local information. In contrast, the recursive split takes a global decision on the whole chain and recursively splits along the minimum of a user-specified function.

Before discussing them, it is important to note that we may want to split the chain in places other than ViewVertices or vertices from the input model. This is why 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.

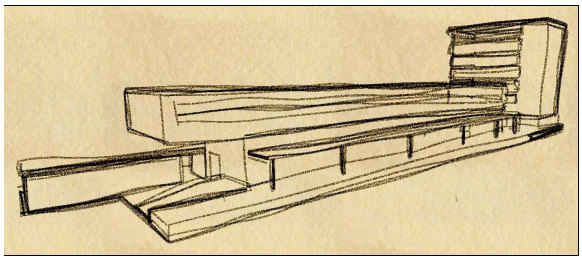


Figure 8: *Use of multiple strokes per chain for rendering a building in a sketchy style.*

The basic sequential split traverses the chain and evaluates a splitting predicate to decide where to split. A new chain is started at each split. In order to handle sketchy styles with multiple strokes per chain (Fig 8), we refine this strategy by decoupling the starting and stopping criteria and perform two traversals in parallel. This process can lead to a partition of the original chain, when the starting and stopping predicates are the same, to a set of overlapping chains or to a set of isolated chains. Note that by using the curvilinear distance on the chain in the splitting predicate, it is very easy to enforce a minimum or maximum stroke length.

The recursive split acts in a global way. It evaluates a function for each sample point, and it splits along the minimum of this function. In addition, the user can prevent splitting at some points specified by a predicate. The split is applied recursively to the two sub-chains, until a recursion predicate is no longer true. As an example, the recursive split is ideal to split at the points of highest curvature.

In our experience, often the same effect can be obtained by first creating long strokes and later splitting them, or by specifying a more aggressive stopping criterion during chaining. Different users envision the drawing process in different ways and may favor one or the other option.

Ordering In our approach, the sequence in which ViewEdges, Chains or Strokes are treated can influence the drawing: In the chaining operator for instance, a timestamp mechanism prevents the reuse of a ViewEdge. In addition, the stroke density information evolves with the current drawing. For instance, when the density information is used to avoid clutter, it is important to treat ViewEdges that are visually more important first, so that they are less likely to be omitted, as described in the next Section. Thus we provide an *ordering* operator, which permits the sorting of any 1D element. The ordering rule is expressed as a comparison predicate based on length, importance, depth, local depth

© The Eurographics Association 2004.

variations, 2D spatial locations, etc. The definition of a relevant ordering of ViewEdges or Strokes can be very tedious and require the evaluation and integration of many different kinds of information that can only be specified using a programmable approach.

3.2. Attribute assignment

Now that we have created strokes, we need to *assign their attributes* such as color, varying thickness, transparency, and spatial location. This step is the most similar to traditional shading systems, except that we operate on 1-dimensional strokes rather than on 0-dimensional fragments. We note that the strokes of a drawing often do not exactly follow the underlying geometry. This is why our shaders can modify the spatial location of the backbone points. This is in a sense similar to the technique of displacement mapping.

In our system, attribute assigners are implemented as procedures working on strokes. As discussed earlier, any information can be queried on the stroke for proper stylistic decisions within the procedure. As a special case, we allow attribute operators to delete strokes in order to avoid clutter.

Multiple attribute assignment operators can be applied to a stroke sequentially. This is useful to control different attributes. One operator assigns color while a second one assigns thickness. In addition, attributes may be assigned in an absolute manner (the previous value is replaced), or in a relative manner (the previous value is modulated). This can be useful to apply a small amount of relative noise after other shaders have set a mean value for an attribute.

As described in Section 2.4, strokes can be resampled 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 StrokeVertex, resampling using a given number of desired points) are available and can be used in the context of strokes geometry modification.

Simple operators such as the assignment of constant attributes are provided. A special attribute operator assigns the mark style used for the rendering of the stroke, as described in Section 5. Other simple shaders include a "Tip Remover" that trims the final and initial portion of the stroke. This permits for example the classical "line haloing" for better depth perception. Furthermore, several useful standard techniques that have long been used in NPR for sketchy effects, such as Noise, stroke displacement, or smoothing, e.g. [MKT*97, KaM*02, SP03] are provided as base components in our system.

4. Density and omission

Stroke omission is a crucial pictorial tool to prevent visual clutter or derive minimalist styles. One way to achieve this is to conditionally create strokes only when the local density in the affected region of the drawing is sufficiently low. Thus strokes are less likely to be created in crowded parts of the image.

The proper omission of strokes relies on two major components: One must estimate the visual density in regions of the drawing, and one must properly prioritize strokes to make sure that "important" strokes are drawn first to avoid their omission due to excessive density. Winkenbach, Salisbury et al. solved the latter problem in the special case of textured areas [WS94], in our system we can insert any ordering operator to control the process.

As described in Section 2.6 the system allows queries for the density of the current drawing density at a given point and scale. This function can be used, for instance within a simple threshold predicate to discard candidate strokes when the drawing is already cluttered in their neighborhood.

The mechanism just described is *causal*, in the sense that the density is evaluated based on what has been drawn so far. In many cases it is necessary to have an idea of the potential density of the drawing, if all lines were drawn. This is obtained from the precomputed view density maps. Note that the combination of density estimates at different scales provides very rich information about the local complexity. Density information can also be used to modulate stroke attributes such as thickness and color.

Fig. 9 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 viewedges 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 the bottom-right image.

Figure 14 shows how to use non-causal density information to put the focus on certain regions; in this case, they correspond to the faces of the characters and the hands of Maria, which are the areas of highest density (see section 6 for more details).

5. Mark back end

The mark system is orthogonal to our programmable line drawing approach. The development of a programmable mark back end is an exciting avenue of future work.

Our mark rendering system uses the standard OpenGL API. Strokes are rendered as triangle strips, determined by the backbone and thickness samples. Standard techniques are used to prevent singularities of the offset curve at high curvature, e.g. [SS02] chapter 3.

© The Eurographics Association 2004.

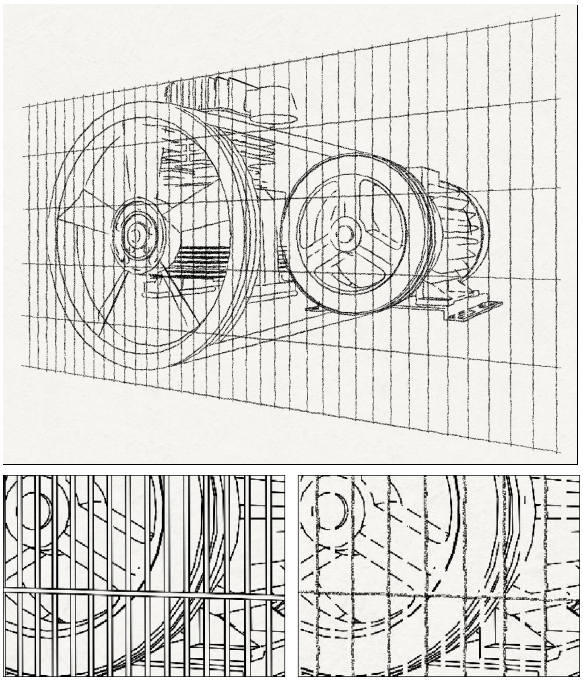


Figure 9: ***Top:*** *Simplified drawing of a complex view.* ***Bottom-left:*** *close-up view with all visible lines showing the unwanted visual complexity. The grid is simplified using causal density in such a way that half of the bars are omitted and each bar is drawn with a single stroke. Note how the grid simplification allows a much clearer view of the engine behind it. However this simplification alone would create problems as shown at the* ***bottom-right****. We use an advanced chaining operator that follows object silhouette or crease lines including short occluded sections, to create continuous strokes.*

We use real 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 [W*99] can imitate graphite and other dry media.

A background texture can be applied. It is however rendered only for the final drawing and does not affect the density computation.

6. Implementation and results

We have implemented our system in C++, using SWIG [Bea96] to generate the Python binding needed for communication between the core system and the style description. The system is distributed under the GPL license (see `http://artis.imag.fr/Projects/Style`).

It takes between a few seconds and a few minutes to compute the ViewMap for a model of approximately 50K polygons, using ray 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 performance hit because of the readback cost. Thus the system is not interactive, mainly due to the poor performance of the interpreted Python language.

A more relevant measure of our system's performance is the time needed to develop a style module. As an example, we spent a total of 3 hours producing the images in Figure 14 (including style modules coding, experimentation and aesthetic evaluation). The style modules comprise about 500 lines of code, half of which are straightforward use of built-in mechanisms. Indeed 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, 1D integration components. Similarly, all information is afforded through standard contextual query mechanisms.

```
def edgeStop (x, sigma):                                  #for anisottropy
  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
```

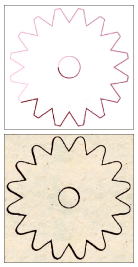


Figure 10: ***Left:*** *Python code for a user-defined anisotropic smoothing operator.* ***Right:*** *The shader is applied on the external contour of a gear. Top: without smoothing. Bottom: with smoothing.*

Figure 10 shows the code of a shader that performs feature-preserving smoothing using anisotropic diffusion inspired by mesh smoothing techniques [DMSB99, DMSB00]. 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`. Normal and curvature are part of the available information, as described in Section 2.6. We use an iterator to traverse the vertices of the strokes. Stroke geometry is modified

© The Eurographics Association 2004.

using the method `setPoint()`. More code examples can be found in the supplemental material .

Figure 1 at the start of the paper shows the application of procedural drawing styles to 3D models. The vintage engine (left) is drawn using five different style modules that emphasize a subset of objects. Standard technical illustration conventions are used to draw creases in white and silhouettes in black. In addition to the visible lines, we also draw lines that are hidden by other objects through a selection using the visibility information about occluders. Lines delineating self-occluded parts of these objects are drawn with dashed lines. Finally, the remainder of the scene is simplified, drawing only the longest strokes in a lighter tone. The sketchy style used for the plane (right) involves "construction lines" based on the backbone-stretch operation. We use a painted stroke for visible lines, and finally add small set of hidden lines with faint strokes.

Figure 11: *Japanese drawing using shortened strokes and density evaluation. The 3D model is shown in the lower-left.*

Figure 11 shows an attempt to imitate the Japanese line-drawing style using two style modules simulating different brushes. Both use line shortening and string tapering. The large brush layer also uses density evaluation to avoid clutter.

Fig. 12 illustrates how programmable chaining operators can generate multiple parameterizations of a given element. This control over stroke topology, demonstrated here with simple attribute-changing shaders, is a key contribution of our system.

Fig. 13 illustrates how 3D Information can be used to drive advanced chaining, with a chaining rule using depth information.

Figure 14 shows a complex style made of eight styles modules applied to a virgin statue model. Three of these modules are responsible for drawing a blueprint: the first two generate bounding boxes out of strokes, giving a coarse approximation of the shape of the model, and the other one draws temporary pencil strokes, trying to mimic hand-drawn style by making use of smoothing and noising shaders. The next two modules mark the beginning of what is supposed to

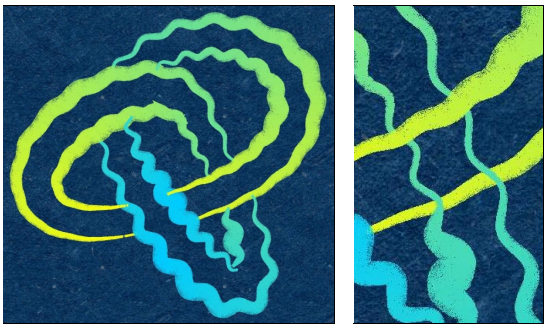


Figure 12: *In this image different shaders use two distinct parameterizations to control attributes: one that covers the entire length of the silhouette line, including occluded portions; and another one that separates and covers the visible segments of the silhouette.* ***Left:*** *color and detail geometry are varied according to the full-length parameterization, whereas stroke thickness varies along the visible-segment-based parameterization.* ***Right:*** *Close-up on the bottom-right part of the scene, in which we added the occluded parts of the silhouette. Comparing this image with the left image, notice that the wavy detail pattern continuously traverses invisible parts.*

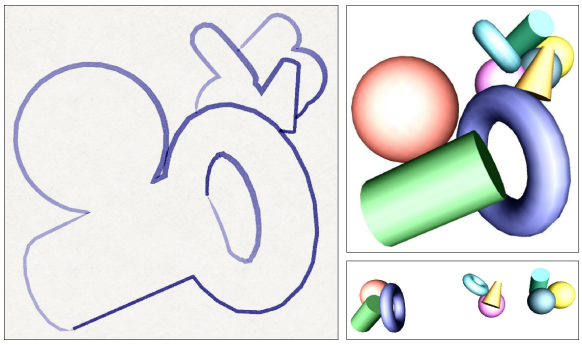


Figure 13: ***Left:*** *Chaining based on depth information to draw strokes around foreground,middle and far distance groups.* ***Right (top):*** *The 3D scene from the same viewpoint.* ***Right (bottom):*** *The scene from a top viewpoint emphasizing the distance between objects.*

be the definitive drawing, by selecting small strokes (among silhouettes and suggestive contours) in high density areas and displaying them in a dark tone. The remaining modules display longer strokes in a lighter tone, also using density information and a fade along the Y axis. The combination of all these modules results in an "unachieved Renaissance-like" style: if we compare it to a straightforward drawing of all the visible lines, apart from purely aesthetic criteria, it is obvious that controlled line omission really plays its role and draws the viewer's attention to the areas of the picture we have decided to emphasize.

© The Eurographics Association 2004.

Figure 14: *A complex stylesheet composed by several style modules.* ***Left:*** *Final rendering.* ***Right:*** *The style modules separated in three groups: one that generates a blueprint, another one that draws short strokes, and a last one taking care of longer strokes. As a comparison, a simple rendering of all the visible lines is shown at the bottom-right corner.*

7. Discussion

We have described a new formulation of the image creation process for the generation of line drawings from 3D models. Our approach is based on programmable operators that can be arranged to create style modules.

We would like to point out that our main contribution is not in new visual effects but rather on a unified and flexible approach for stylized line drawing. However, the control and flexibility afforded by the approach make many new effects possible through:

- Advanced Layering (Fig. 1 right, Fig. 14 and Fig. 1 left).
- Control over stroke topology (Fig. 7 a, Fig. 13, Fig. 7 c, Fig. 9, Fig. 12, Fig. 4).
- Stroke geometry displacement (Blueprint of Fig. 14, guiding lines of Fig. 4).
- Density Control (Fig. 9, Fig. 14).

One of the major benefits of our approach is its natural redundancy, implying great flexibility: most effects can be obtained (a) by modifying individual operators (using a programming interface), (b) by changing the set of operators in a style module or controlling their behavior, or by adding and scheduling specialized style modules (using a graphical user interface). Furthermore, the style module descriptions can be modified online before being interpreted by the system, making the stylized rendering session a truly interactive experience.

The introduction of programmable "shaders" for non-photorealistic rendering opens many interesting avenues for graphical design and styles, all the more so since by definition non-photorealistic styles allow the greatest freedom for geometric and visual modifications of the underlying model. The line smoothing operator described in the paper is a good example, as well as the generation of "construction lines" from the rendering strokes.

In our experience, the development of a style sheet is comparable to that of procedural shaders in traditional rendering. The most challenging step is often to formulate the goal in terms that can be translated into programs. After this step, experimentation and refinement proceed smoothly.

Our system is currently limited to line drawings composed of the set of edges in our view map. Natural extensions would include a consistent treatment of tonal and hatching lines. Future work also includes similar procedural treatments for shading and all other NPR components.

Temporal coherence Our system does not yet address the issue of temporal coherence. Its very flexibility could in some cases make temporal coherence issues more pronounced. We believe that this can be addressed using specific style modules and some extensions to the system, in particular consistent stroke parameterization [KDMF03]. Nevertheless, topological operations such as chain splitting can exhibit temporal discontinuities, thus requiring explicit consideration of time. Recent work in NPR for animation [KDMF03, KSC*01, CTP*03], however, reinforces our belief that a programmable approach is important to control the variety of tradeoffs and stylistic choices related to temporal depiction.

Acknowledgments

This work was supported in part by "Région Rhône-Alpes" (DEREVE project and EURODOC grant).

References

[AG99] APODACA A., GRITZ L. (Eds.): *Advanced Renderman : Creating CGI for Motion Pictures*. Morgan Kaufmann, 1999. 1, 2, 5

[Ale01] ALEXANDRESCU A.: *Modern C++ Design*. Addison-Wesley C++ In-Depth. Addison-Wesley Publishing Company, New York, NY, 2001. 6

[App67] APPEL A.: The notion of quantitative invisibility and the machine rendering of solids. *Proc. ACM Natl. Mtg.* (1967), 387. 5

[Bea96] BEAZLEY D. M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In *4th Annual Tcl/Tk Workshop* (July 1996), USENIX, (Ed.), pp. 129–139. 9

[Coo84] COOK R. L.: Shade trees. In *Proc. SIGGRAPH* (1984). 1

[CTP*03] CUNZI, THOLLOT, PARIS, DEBUNNE, GASCUEL,

© The Eurographics Association 2004.

DURAND: Dynamic canvas for non-photorealistic walkthroughs. In *Proc. Graphics Interface* (2003). 11

[DFRS03] DECARLO D., FINKELSTEIN A., RUSINKIEWICZ S., SANTELLA A.: Suggestive contours for conveying shape. *ACM Trans. on Graphics 22*, 3 (2003). 2, 3, 5

[DMSB99] DESBRUN M., MEYER M., SCHRÖDER P., BARR A.: Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proc. SIGGRAPH* (1999). 9

[DMSB00] DESBRUN M., MEYER M., SCHRÖDER P., BARR A.: Anisotropic feature-preserving denoising of height fields and bivariate data. In *Graphics Interface* (2000), pp. 145–152. ISBN 1-55860-632-7. 9

[DOM*01] DURAND F., OSTROMOUKHOV V., MILLER M., DURANLEAU F., DORSEY J.: Decoupling strokes and high-level attributes for interactive traditional drawing. In *Eurographics Workshop on Rendering* (2001). 2

[DS02] DECARLO D., SANTELLA A.: Stylization and abstraction of photographs. *ACM Trans. on Graphics 21*, 3 (2002). (Proc. SIGGRAPH). 2

[Dur02] DURAND: An invitation to discuss computer depiction. In *Proc. NPAR* (2002). 1, 6

[FTP99] FREEMAN W., TENENBAUM J., PASZTOR E.: *An example-based approach to style translation for line drawings*. Tech. Rep. 99-11, MERL, 1999. 2

[GG01] GOOCH, GOOCH: *Non-Photorealistic Rendering*. AK-Peters, 2001. 1

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. New York, NY, 1995. 6

[GSG*99] GOOCH B., SLOAN P., GOOCH A., SHIRLEY P., RIESENFELD R.: Interactive Technical Illustration. *ACM Symp. on Interactive 3D Graphics* (1999). 3

[Her98] HERTZMANN A.: Painterly rendering with curved brush strokes of multiple sizes. *Proc. SIGGRAPH* (1998). 2

[Her01] HERTZMANN A.: Paint By Relaxation. In *CGI* (2001), pp. 47–54. 2

[HJO*01] HERTZMANN A., JACOBS C., OLIVER N., CURLESS B., SALESIN D.: Image analogies. *Proc. SIGGRAPH* (2001). 2

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *Proc. SIGGRAPH)* (1990). 1, 5

[HL94] HSU S. C., LEE I. H. H.: Drawing and animation using skeletal strokes. *Proc. SIGGRAPH 94* (1994). 3

[HS99] HAMEL J., STROTHOTTE T.: Capturing and re-using rendition styles for non-photorealistic rendering. In *Proc. Eurographics* (1999). 2

[HSS02] HALPER, SCHLECHTWEG, STROTHOTTE: Creating non-photorealistic images the designer's way. In *Proc. NPAR* (2002). 2

[HZ00] HERTZMANN A., ZORIN D.: Illustrating smooth surfaces. *Proc. SIGGRAPH* (2000). 2, 3, 5

[IFH*03] ISENBERG T., FREUDENBERG B., HALPER N., SCHLECHTWEG S., STROTHOTTE T.: A developerŠs guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications special issue on NPR* (2003). 3

[KaM*02] KALNINS, ARKOSIAN, MEIER, KOWALSKI, LEE, DAVIDSON, WEBB, HUGHES, FINKELSTEIN: Wysiwyg npr: Drawing strokes directly on 3d models. *ACM ToG 21*, 3 (2002). (Proc. SIGGRAPH). 2, 8

[KDMF03] KALNINS R. D., DAVIDSON P. L., MARKOSIAN L., FINKELSTEIN A.: Coherent stylized silhouettes. *ACM Trans. on Graphics 22*, 3 (2003). 11

[KMN*99] KOWALSKI M., MARKOSIAN L., NORTHRUP J. D., BOURDEV L., BARZEL R., HOLDEN L., HUGHES J.: Art-based rendering of fur, grass, and trees. *Proc. SIGGRAPH* (1999). 2

[KSC*01] KLEIN A. W., SLOAN P.-P. J., COLBURN A., FINKELSTEIN A., COHEN M. F.: *Video Cubism*. Tech. Rep. MSR-TR-2001-45, Microsoft Research, 2001. 11

[MKT*97] MARKOSIAN L., KOWALSKI M., TRYCHIN S., BOURDEV L., GOLDSTEIN D., HUGHES J.: Real-time nonphotorealistic rendering. *Proc. SIGGRAPH* (1997). 2, 5, 8

[Ope02] OPEN NPAR:. http://www.opennpar.org/, 2002. 2

[SC92] STRAUSS P. S., CAREY R.: An object-oriented 3d graphics toolkit. In *Computer Graphics (Proc. of SIGGRAPH 92)* (July 1992), vol. 26, pp. 341–349. 2

[SP03] SOUSA M., PRUSINKIEWICZ P.: A few good lines: Suggestive drawing of 3d models. *Computer Graphics Forum (Proc. of EuroGraphics '03) 22*, 3 (2003). 8

[SS02] STROTHOTTE T., SCHLECHTWEG S.: *Non-Photorealistic Computer Graphics. Modeling, Rendering, and Animation*. Morgan Kaufmann, 2002. 1, 3, 8

[Tee03] TEECE D.: Ink line rendering for film production. In *Theory and practice of Non-Photorealistic Graphics: Algorithms, Methods and Production Systems, SIGGRAPH Course Notes*, Sousa M. C., (Ed.). SIGGRAPH, 2003. 2

[TF97] TENENBAUM J., FREEMAN W.: Separating style and content. In *Advances in Neural Information Processing Systems* (1997), vol. 9, p. 662. 2

[Ups89] UPSTILL S.: *The Renderman Companion*. Addison-Wesley, Reading, MA, 1989. 1, 2, 5

[W*99] WOO M., ET AL.: *OpenGL programming guide: the official guide to learning OpenGL, ver. 1.2*, third ed. Addison-Wesley, Reading, MA, USA, 1999. 9

[Wil97] WILLATS J.: *Art and Representation*. Princeton U. Pr., 1997. 3, 6

[WS94] WINKENBACH G., SALESIN D.: Computer-generated pen-and-ink illustration. *Proc. SIGGRAPH* (1994). 2, 8

© The Eurographics Association 2004.