A Procedural Approach to Style for NPR Line Drawing from 3D models

Stéphane Grabli, Frédo Durand, Emmanuel Turquin, François X. Sillion

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Procedural Approach to Style for NPR Line Drawing from 3D models

Stéphane Grabli — Frédo Durand — Emmanuel Turquin — François Sillion

N° 4724

Février 2003

THÈME 3

ISSN 0249-6399

A Procedural Approach to Style for NPR Line Drawing from 3D models

Stéphane Grabli* , Frédo Durand†, Emmanuel Turquin‡ , François Sillion§

Thème 3 —Interaction homme-machine,
images, données, connaissances
Projets ARTIS et MIT Graphics Group

Rapport de recherche n° 4724 —Février 2003 —25 pages

Abstract: This paper introduces a procedural approach to non-photorealistic line drawing from 3D models. The approach is inspired both by procedural shaders in classical rendering and by the power of procedural modeling. We propose a new image creation model where all operations are controlled procedurally. A *view map* describing all relevant support lines in the drawing is first created from the 3d model; a number of style modules operate on this map, by procedurally selecting and chaining lines before creating strokes and assigning drawing attributes. Two different levels of user control are provided, ranging from a low-level programming API to a parameterized building-block assembly mechanism. The resulting drawing system allows very flexible control of all elements of drawing style: first, different style modules can be applied to different types of lines in a view; second, 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 results of its application.

Key-words: Non-Photorealistic Rendering, Procedural Rendering, style, line drawing

* ARTIS
† MIT Graphics Group
‡ ARTIS
§ ARTIS

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Modélisation et application procédurale de styles pour le rendu expressif

Résumé : Ce document introduit une approche procédurale pour le rendu de dessins au trait à partir de modèles 3D. Cette approche s'inspire à la fois des *shaders* procéduraux utilisés en rendu traditionnel et de la modélisation procédurale. Nous proposons un nouveau modèle de création d'images dans lequel toutes les opérations sont contrôlées de manière procédurale. Dans un premier temps, un *graphe de vue* décrivant tous les supports des lignes pertinentes du dessin est créé à partir du modèle 3D. Des *modules de style* s'appuient alors sur ce graphe pour sélectionner et chaîner les lignes de manière procédurale, avant de créer les *traits* et de leur affecter des attributs de dessin. Deux niveaux de contrôle différents sont fournis à l'utilisateur, allant de l'interface de programmation bas-niveau à un mécanisme d'assemblage de composants paramétrés. Le système résultant permet un contrôle très souple de tous les éléments constitutifs du style de dessin: premièrement, différents modules de style peuvent s'appliquer à différents types de lignes; deuxièmement, les attributs des traits sont affectés de manière procédurale et peuvent dépendre à volonté des différentes propriétés de la scène ou de la vue. Nous illustrons les composants de ce système et montrons les résultats de son application.

Mots-clés : Rendu non photoréaliste, rendu procédural, style, dessin

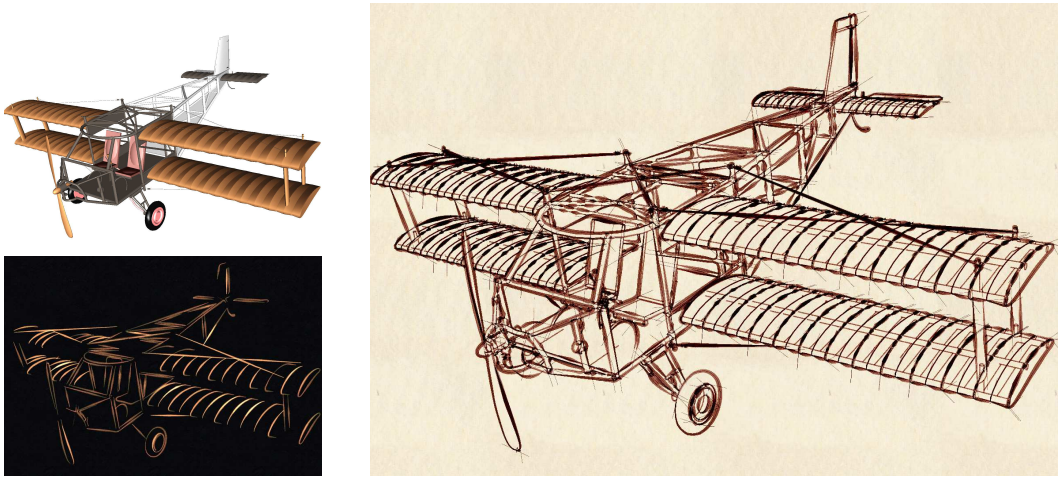


Figure 1: Two line drawings generated from a 3D model using our system.

1 Introduction

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 lack a flexible and formal specification framework. In contrast, the shading languages available in photorealistic renderers such as Pixar Renderman [Coo84, HL90, Ups89, AG99] permit the design of an infinite variety of rich and complex appearances. In this paper, we introduce a flexible procedural approach to NPR line drawing. We focus on pure line drawing and leave hatching and tonal modeling as future work. Style can be specified by implementing procedures that describe how the silhouettes and other feature lines from the 3D model should be turned into strokes. This allows for a great variety of styles, including local stylistic variations within the drawing.

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 the attention on relevant part and omit superfluous details. Despite its simplicity, line drawing permits a broad variety of styles. Style can be varied by changing the medium (e.g. chalk, pencil, pen and ink), but also by varying the attributes of the strokes. This includes thickness, tone, transparency, or texture. The art and craft of drawing carefully chooses these attributes to reveal shape, texture, ambiance, or to place emphasis. Equally important, the omission of strokes such as parts of the silhouettes allows the artist to prevent clutter, to choose focal point, and to call to the imagination of the beholder in order to produce a more lively drawing. It is therefore crucial to provide a fine control and the possibility to vary style across the picture.

In this paper, we present an approach to style in line drawing inspired by photorealistic production rendering software such as Renderman [Coo84, HL90, Ups89, AG99]. 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. Similarly, the style of a line drawing can be specified in our approach by implementing procedures that describe how the silhouettes and other feature lines from the 3D model should be turned into strokes. Historically, the development of shading languages has dramatically facilitated the exploration and development of realistic shading, and it has contributed to a dramatic improvement in the quality of production rendering. We hope that NPR can similarly benefit from a flexible procedural approach.

We see two major application domains for our approach. First, the approach is ideally suited for situations where drawings have to be produced automatically in a given style, such as repair or assembly manuals, CAD-CAM, etc., where different features need to be rendered in different styles, or where some parts need to be emphasized, e.g. [SF91]. For example, the new field of feature modeling encodes semantic and high-level geometric information about parts of objects. At rendering time, these features needs to be visualized in a different style [BBN02]. Our approach is ideally suited for this, since it can use this high-level information and render objects in a flexible variety of styles. Which leads to our second application domain. The flexibility of our approach greatly facilitates the exploration of style and depiction issues for line drawing. A new effect can be implemented very easily and interactively explored, encouraging creativity.

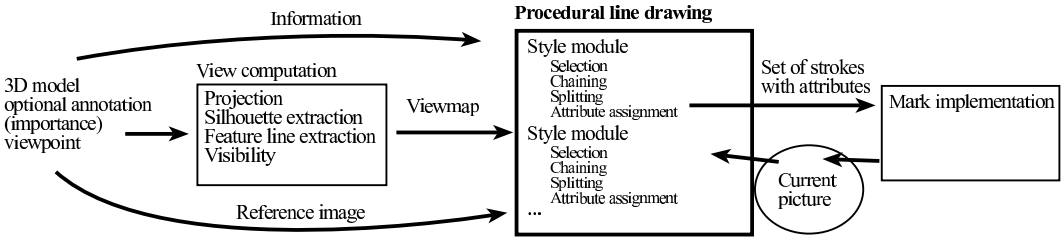


Figure 2: System architecture.

1.1 Related work

Non-photorealistic rendering has proposed a wealth of techniques to generate line drawings from 3D models, e.g. [MKT+97, HZ00, RC99, BS00, Elb98, SB99, GSG+99, KMM+02]. We build upon these techniques and extend them to propose a flexible procedural control of the drawing's appearance.

Style has received much attention in Non-Photorealistic Rendering and computer vision. *Machine learning* is the most popular approach, e.g. [TF97, FTP99, BH00, HJO+01, JEGPO02, KMM+02]. However, these work focus on the low-level and statistical aspects of styles. Most Non-photorealistic techniques offer a control of style through a set of *parameters*, e.g. [Her98], or

through direct user interaction, e.g. [Hae90, Sch96, KMM+02]. 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, SD02, DS02].

The work closest to ours is the OpenNPAR system [HSS02, Ope02]. They developed an API for the development of real-time NPR software. They provide an impressive graphical user interface for the exploration of simple styles. Our approach focuses on line drawing and on more complex style development. We believe that ideally, a line drawing style could be explored first in our system before porting a more optimized real-time implementation in OpenNPAR.

We also borrow from procedural modeling, which is a great technique to augment the complexity of graphics models [EMP+94, LDG01, CDM+02]. Similarly, in our system, the richness of a style can be generated by procedures specified by the developer.

Our approach is also inspired by the style sheets present in word processing systems such as LaTeX. They separate style from content, and the writer needs only provide some high-level semantic information such as what is a section header. The developer of a style writes a set of procedures that describe the appearance of the document, based on the structure information provided by the writer. The structure of 3D scenes and pictures is not as simple to specify, and we will rely more on geometry and semantic information provided by the user such as the importance of objects.

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. Impressive results can be obtained, but the styles are usually limited to plain or wiggly lines, because no real one-dimensional stroke data structure is available.

While we draw some of our inspiration from the existing procedural 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, on 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 actual sequence of strokes drawn, matters in the final result.

1.2 Overview of contributions

We introduce a procedural 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 and ordering of lines and strokes. We show how the resulting strokes can be further processed and modified to obtain

interesting graphical styles. Finally we manage the information necessary at all stages of the drawing process. The resulting system allows both high-level descriptions and control and low-level control through a programming interface, and provides a highly customizable and versatile non-photorealistic line drawing environment.

2 Architecture

The architecture of our Drawing creation system is shown in Fig. 2. It takes as input a 3D model, possibly augmented with information such as color or the subjective importance of each object, for instance, according to the application at hand.

The first step is the computation of the view: well-known algorithms are applied to perform perspective projection, silhouette extraction, and visibility computations,e.g. [HZ00, IFH+03]. The output is a *view map*, which is the planar graph describing the view. It is composed of ViewEdges and ViewVertices (Fig. 3). A ViewEdge can be a silhouette, a crease, or a border edge. A ViewVertex can correspond to an actual vertex of the scene, to the visual intersection of two edges (T-vertex), or to an end junction (cusp). A *reference image* is also computed at this stage, to provide an image space map of object ID and depth using image buffers.

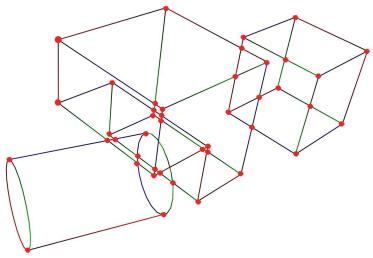


Figure 3: View map data structure. The ViewEdges are represented with a color gradient and ViewVertices are the red dots.

The second step is the procedural line drawing itself, which constitutes the major contribution of this article. It takes as input the view map and the reference image, and creates the *stroke* primitives that constitute 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. 4. Multiple strokes can be created to depict a single ViewEdge (for example in a sketchy style), and a stroke can span multiple ViewEdges (for example, a single stroke can be used to depict the external outline of a given object.)

Finally, the mark system is responsible for the actual rendering of the stroke primitives with their attributes. The same stroke with given thickness and color can for example be rendered with

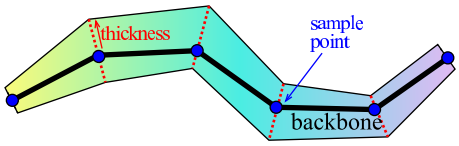


Figure 4: Stroke representation.

different mark styles such as crayon or oil painting. However, the precise mark style is often specified in the procedural drawing. Although we describe them as separate processes, stroke rendering is interleaved with stroke creation, that is, the drawing is refreshed each time a new stroke is created. This is crucial to allow the current state of the drawing to influence subsequent decisions. For example, if the local density of strokes is too high, a potential new stroke can be omitted to prevent clutter.

We call the set of procedures that implement a given pictorial style a *procedural line style*. It is usually decomposed into a series 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 (but it is not the only one). 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.

A style module is based on a pipeline of four classes of *operators* that are applied iteratively at the programmer's will. These operate on ViewEdges to build and alter strokes. These operators permit the *selection* of a set of ViewEdges, which is the way they restrict their action to a subset of the drawing. Then, starting from a given edge in the selection, a one-dimensional *chain* of ViewEdges is built. These chains are *split* at appropriate locations (e.g. points of high curvature) and strokes are created to depict the chain, which includes the decision of how many separate strokes are used to depict a given ViewEdge, and where strokes start and end. Finally, an *Attribute assignment* operator specifies, e.g., the color, width, texture, transparency for each point along the stroke.

2.1 A simple example

To make these operators more concrete, let us consider as an example a simple procedural line style, illustrated in Figure 5. In this example we use three style modules:

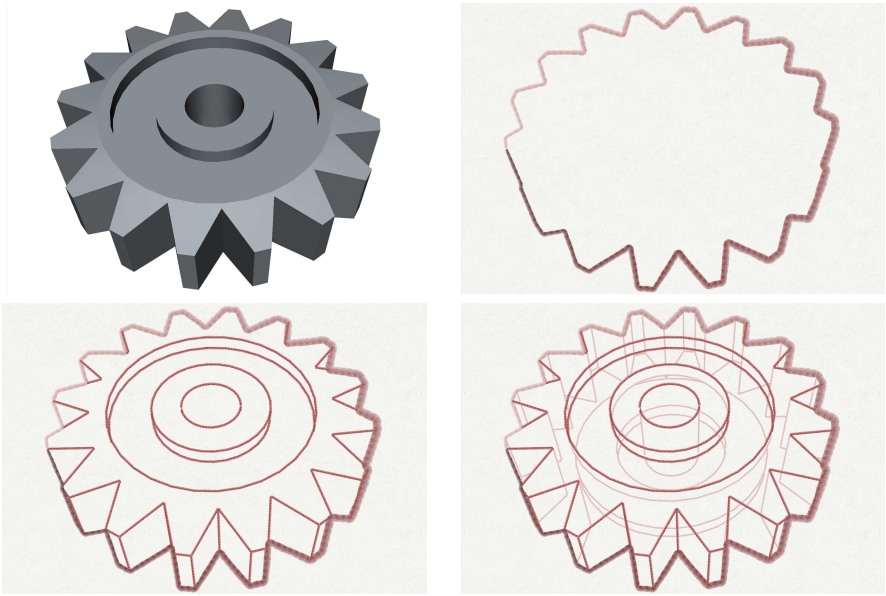


Figure 5: Simple procedural line style using three style modules.

The first module
- **selects** edges on the external contour of the drawing
- **chains** all edges on the external contour
- **smoothes** the stroke
- **assigns** varying color/width to the stroke

The second module
- **selects** all visible edges (excluding the external contour)
- **chains** edges along silhouettes and crease lines
- **assigns** constant width to the strokes

The third module
- **selects** all edges that are hidden
- **chains** along silhouettes and crease lines
- **assigns** fine width to the strokes

2.2 Interface

The specification of a procedural line style is performed at two levels. The lower level is a C++ API where operators are implemented. We provide a number of standard operators, but the user can always implement new operators and benefit from the infrastructure we provide. This task is similar to that of technical directors who implement realistic shaders in production companies.

The higher-level interface handles the specification of style modules where parameters are controlled, and some expressions such as selection predicates or ordering criteria can be set. This level requires less familiarity with the underlying technical specifics and allows for a rapid exploration of style, since it can be performed at runtime. In our implementation, we use either an XML representation or a specially designed description language (which is easier to read for humans!). The description of the style module for external contours from Section 2.1 is listed below:

```
module ExternalContour {

operators:

  Select<ViewEdge>(QuantitativeInvisibility<ViewEdge>(0) &&
                   ExternalContour<ViewEdge>())
  BidirectionalChain<ViewEdge>(ChainPredicateIterator(all),
                                  QuantitativeInvisibility<ViewEdge>(0) &&
                                  ExternalContour<ViewEdge>()),
                               Null<ViewEdge>() ||
                                  EqualToTimeStamp<ViewEdge>() ||
                                  QuantitativeInvisibility<ViewEdge>(0),
                               Increment(TimeStamp))
  Create<Chain>()

shaders:

  Thickness("thickness01.bmp")
  IncreasingColor(0.5, 0.5, 0.8, 1)
}
```

Table 1: Description of a style module.

Finally, a graphical user interface allows the user to load style modules and organize the various layers composing the drawing.

2.3 Data Structure

As discussed above, the specificity of line drawing is that it requires the handling of one-dimensional objects such as view edges and strokes in additional to simpler zero-dimensional points.

The ViewMap is encoded as a graph data structure, composed of `ViewVertices` and `ViewEdges` (Table 2). The traditional adjacencies between them are stored. In addition, ViewVertices and ViewEdges contain information about the 3D scene (Table 2). In particular, ViewEdges contain the description of their 3D embedding as a list of 3D points, as well as their type (silhouette, crease, or border). Similarly ViewVertices store their image-space coordinates (as well as the distance to the camera), and their type (cusp, T-vertex, scene vertex.)

```
ViewVertex:
  Point3D WorldCoordinates
  Point3D CameraCoordinates
  List of adjacent ViewEdges
  Type <enum: T-vertex, Scene Vertex, Cusp>

ViewEdge
  ViewVertex *V1, *V2
  Type <enum: Silhouette, Crease, Border>
  Object ID
  List of sample points from embedding
  Int quantitative invisibility
  List of Occluders
  Occluded object // for silhouettes
```

Table 2: Viewmap data structures.

The central objects manipulated by our operators are one-dimensional chains that will eventually lead to the creation of strokes. `Chains` describe a path in the ViewMap graph (Table 3). They are connected sets of `ViewEdges`. Note however that they do not necessarily start at a `ViewVertex`, which is why they also store an initial and final point. Chains can implicitly access points by generating samples along their `ViewEdges`. Note however that for memory efficiency and for sampling flexibility, these sample points are not stored. Sample points are characterized 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. 4). A set of *attributes* is stored for each of these sample points (thickness, color, and transparency.)

The appropriate sampling of strokes can be important to capture attribute variation. We provide a stroke resampling mechanism that takes a maximum length between samples and generates new points when needed. Attributes are interpolated, currently using 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].

2.4 Information access

As pointed out by Hanrahan and Lawson Hanrahan:1990:ALF, defining the possible exchange of information at the interface between the rendering program and the shading modules is a crucial

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

Stroke : derives from Chain
  List of ViewEdges
  Sample point initial and final
  Double length
  Array of sample 2D vertices [nb_samples]
  Array of attributes [nb_samples]
     Thickness [left, right]
     Color
     Transparency
  Mark Rendering style
```

Table 3: Chain and stroke data structures.

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, minimum and maximum value, or variance).

Information can come from four different sources: the 3D scene, a reference view, the view map, and the current drawing, as summarized in Table 4. Types of available information include scalars, vectors (normal direction), colors, and image maps.

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 more emphasized.

The information encoded in the view map data structures described above are made available. In addition, differential information such as 2D normal and 2D curvatures can be computed. The formulae for the discrete evaluation of these properties is given in the appendix. 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 reference view computed using standard z-buffering. It provides information such as the average local depth or the color obtained using classical shading. In addition, the current drawing is refreshed as the drawing creation proceeds, and the local stroke density can be queried. Density information is computed upon request, using a parameterized Gaussian smoothing operator to allow queries at different scales. This use of density is related to the work of Winkenbach et al.

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

Table 4: Information provided by our system.

[WS94] and Salisbury et al. [SABS94, SWHS97], However in our system, density is used to control the creation of feature strokes rather than for the tonal modeling in textured areas.

3 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 2.3. In our implementation, the view map is computed in three steps.

We first extract all relevant lines from the model, with respect to the chosen view parameters. In practice, we have chosen to use the technique by Hertzman and Zorin [HZ00] for its robustness. The view map contains silhouettes, 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 silhouettes, creases and boundary edges are then computed and define `ViewVertices`. A `ViewEdge` is an arc of the view map linking two `ViewVertices` (see Fig. 3), and is represented as a connected set of line segments as described in the previous section.

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. We also store the

list of occluders for each `ViewEdge`. This information can later be used, e.g., to treat differently edges hidden by some important object.

The addition of other features in the view map is a promising avenue for future extensions of our system, as it would allow different types of lines to be used as a basis for the drawing. Possible such features include parabolic curves or skeletal curves [HBK02, Kø84].

4 Procedural line drawing

We now discuss the different operators involved in the application of a style module. Recall that such modules are applied sequentially to obtain successive layers in the procedural drawing.

We emphasize again that there are two levels of programmability in our system: each operator is a procedure, that can either be chosen in a library of predefined operators using our XML or text input at runtime, or overloaded by a programmer using the C++ API. In addition, the particular choice and ordering of operators in a style module provides another level of control.

A style module works on the view map and produces a set of strokes through a pipeline organization illustrated in Fig. 6. In this Section we describe the first steps of the pipeline that deal with the topological creation of strokes from the view map. The description of the last class of operator, attribute assignment, is deferred to the next Section.

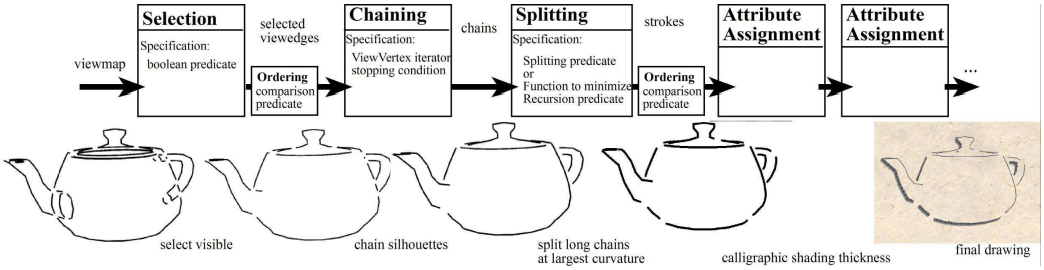


Figure 6: Organization of a style module as a pipeline of four types of operators, plus ordering operations.

4.1 Selection

The first class of operators we employ is a *selection* operator. Its role is to extract a subset of the ViewEdges from the view map and place them in the list of active ViewEdges for further processing. Selection is the main mechanism to layer the drawing and restrict the effect of a style module to particular ViewEdges. Selection is specified with unary predicates. Basic predicates are provided that permit the test of the value of any type of information described in Section 2.4; For example, selection can be based on the quantitative invisibility value, on the object ID, on the ViewEdge nature (crease, silhouette, or border), etc. In addition, these predicates can be combined with classical logic

operators using the XML/language interface. Developers can also implement new predicates based on more complex functions of the scene.

4.2 Chaining

Chaining operators are responsible for creating connected lists of ViewEdges, which we call *chains*. A chain therefore represents a one-dimensional graphical element in image space [Wil97, Dur02], which will be used to control stroke creation. Chaining is responsible for converting from the graph nature of the ViewMap to the one-dimensional nature of strokes. However we note that there is not necessarily a simple correspondence between chains and strokes: in particular while chaining is very useful to build consistent graphical elements, it is often necessary to split the obtain chains in various ways to better control the creation of strokes, as will be discussed shortly.

A chaining operator is invoked successively on all ViewEdges in the selection, and builds a chain originating from each. A chaining operator involves two kind of decisions: when to stop, and where to turn at a ViewVertex. It is implemented as a C++ iterator in the spirit of the STL [SLM00]. The iterator incrementation 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 stop when a certain length is reached, if an occlusion is encountered, or when the curvature is too high.

The standard chaining operators offered by our system can follow a silhouette, follow the external contour, or attempt to always follow the ViewEdge that yields the straightest chain. Their stopping criterion is specified by a predicate similar to those discussed above.

We provide an optional tagging mechanism to prevent multiple chaining of the same ViewEdge. We also offer the following chaining options, independently from the actual chaining algorithm. Chaining can be either bidirectional 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.

4.3 Chain splitting and stroke creation

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 main role of the operator is to decide where the chain should be split.

We propose two standard mechanisms to do this: a sequential split that traverses the chain and decides to split based on a predicate, and a recursive split that 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. Our system permits the traversal of a sampled version of the curve by specifying a sampling rate. Temporary vertices at this sampling rate are iteratively created as the chain is traversed.

The sequential split then iterates on these sample points and evaluates a predicate. When it is true, the chain is split at this point, and the process starts again from there. Note that by using the

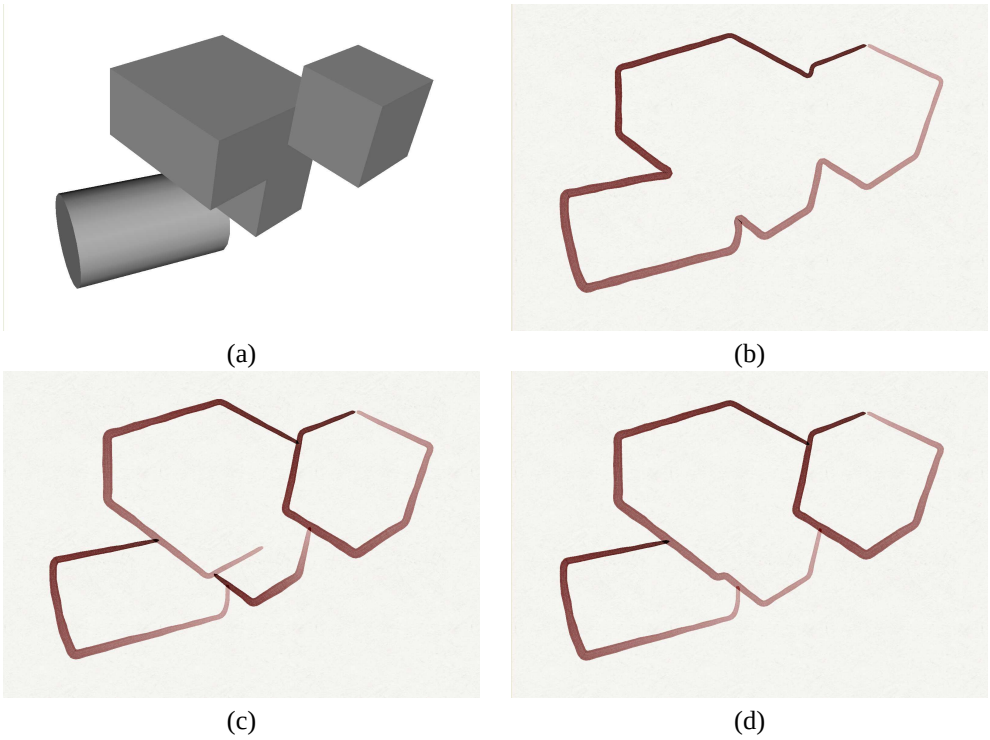


Figure 7: Examples of simple chaining predicates, applied to the set of ViewEdges on the external contour of the drawing: (a) shaded view (b) follow external contour (c) follow silhouettes on same object (d) follow contour of same object. Note how in cases (c) and (d) the chaining operation includes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

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 more global way. It evaluates a function for each sample point, and it splits along the minimum of this function. It is then applied recursively to the two sub-chains, until a recursion predicate is no longer true. The recursive split is for example ideal to split along the points of highest curvature.

4.4 Ordering

In our approach, the sequence in which ViewEdges, Chains or Strokes are treated can influence the drawing: in the chaining operator for instance, the timestamp mechanism prevents from re-using a ViewEdge. In addition, the stroke density information evolves with the current drawing. For in-

stance, 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. Thus we provide an *ordering* operator, which permits the sorting of ViewEdges or Strokes. It is based on a comparison predicate or any user-defined procedure, for instance using comparisons of length, importance, depth, local depth variations, 2D spatial locations etc.

5 Attribute assignment

Now that we have created strokes, we need to *assign their attributes* such as color, varying thickness, transparency, but also 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.

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

As a special case, we allow attribute operators to delete a stroke. This is useful to discard strokes in order to avoid clutter.

As described in Section 2.3, strokes can be resampled to account for the various sampling rate requirements of specific styles. This can be done at the API level by the programmer of an attribute operator. For example, our noise operator adapts the sampling of the stroke to the frequency of the noise specified by the user. Resampling can also be called as a stand-alone operator in the XML/language interface.

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 6. 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.

In the rest of this section, we describe some of the standard attribute operators offered by our system. More complex operators can be implemented by developers at the API level.

5.1 Functors

One of the simplest yet powerful way to specify attribute operators is through the use of functors, similar to the STL concept [SLM00]. A functor is a function that is applied to each sample vertex of the stroke. It can be any function that takes the information defined in Section 2.4 as input and that outputs a thickness, color, or transparency. Functors can be specified at the XML/language level and provide a quick way to vary attributes. For example, we used a functor on z to vary thickness and color in Fig. 12(b) and 13(a).

5.2 Spatial modification of strokes

Attribute operators can modify the spatial location of the backbone points. This is in a sense similar to the technique of displacement mapping. Spatial displacement is ideal to generate sketchy styles.

We provide a spatial noise operator that moves points along the normal to the polyline. A 1D Perlin turbulence is used, with either a simple linear interpolation for sharp corners, or a smoother Lanczos-Windowed sinc interpolation for smoother variation [Tur90]. The user controls the amount of noise, its scale, the number of octaves in the turbulence, and the interpolation used.

We also provide a "backbone stretcher" that extends strokes beyond their initial and final points. We use it for sketchy styles and to add construction lines, as in Fig. 1.

We developed an operator to smooth the stroke (Fig. 8). It uses normal or curvature flow and converges towards either a straight line or a curve of constant curvature (circle). In order to preserve sharp corners, we introduce an anisotropic edge-stopping function that prevents the diffusion across corners (Fig. 8, right). The details are described in the appendix. We also provide the option to keep feature vertices such as ViewVertices fixed, in order to preserve the topology of the drawing. This can be easily enforced in the diffusion framework. Other approaches such as spline approximation could also be studied.

Smoothing is ideal to make drawings more abstract and minimalist. In addition, it can be used to remove the artifacts of the tessellation of smooth objects. Smoothing is made much more effective by the use of appropriate chaining. The topological information generated during chaining makes it possible to perform diffusion between adjacent samples.

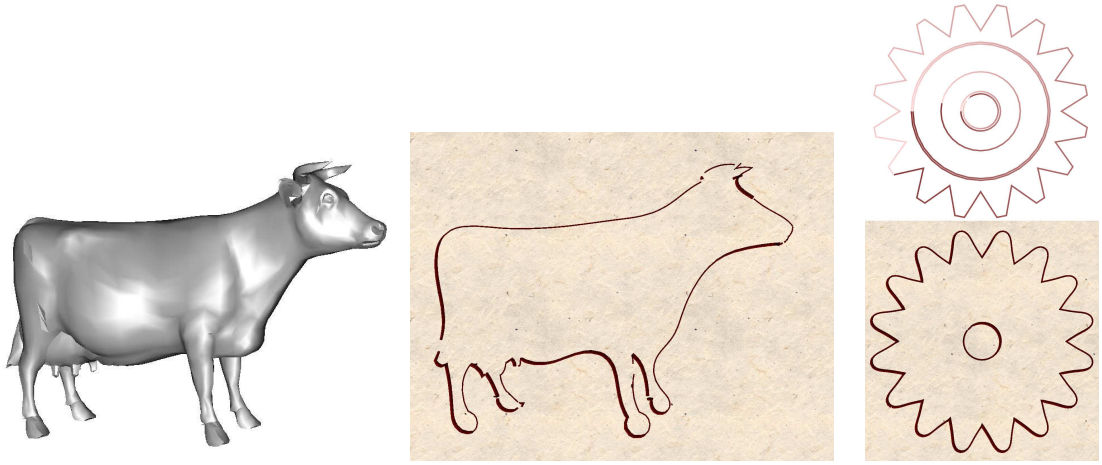
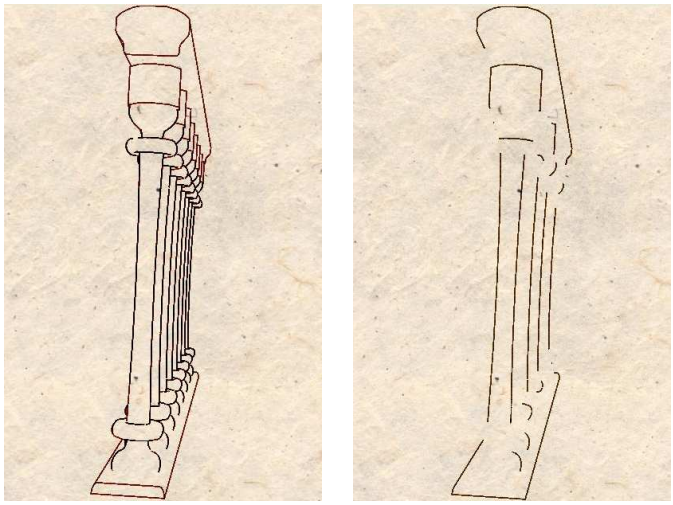


Figure 8: Use of the smoothing operator to produce a minimalist style. In the rightmost image, sharp corners are preserved using an edge-stopping function.

5.3 Use of density

As described in Section 2.4, our system provides access to the current density of strokes in a given part of the drawing. This can be used by an attribute operator to omit strokes where density is too high. The system includes a standard operator that uses a threshold to discard strokes when the drawing becomes too cluttered. The user controls the threshold, but also the radius of the area around the stroke where density is queried.

Fig. 9 illustrates the use of the density operator. Combined with depth or stroke-length ordering, it is ideal to simplify distant objects.



(a) (b)

Figure 9: Use of the density map to limit clutter.(a) drawing with all visible edges (b) Stroke creation prevented in dense areas.

6 Mark back end

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.

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 inversed 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.

7 Implementation and results

Figure 1 at the start of the paper shows the application of different procedural drawing styles to a 3d model of a plane. The sketchy style is obtained by indicating "construction lines" using the backbone stretch operation, using a painted stroke for visible lines and indicated some hidden lines using faint strokes.

Figure 11-(a) shows how quantitative (in)visibility information can be used for a simple control of style attributes. Figure 11-(b) shows another sketchy style, using the chain split operator to break chains into multiple strokes. In this case it is apparent that the size of the strokes is decoupled both from the edge length in the 3d model and from the chain length in the drawing. Figure 11-(c) employs a varying stroke width as a function of depth to emphasize the 3d perception, and differentiates shading attributes based on the nature of edges (silhouettes or crease lines).

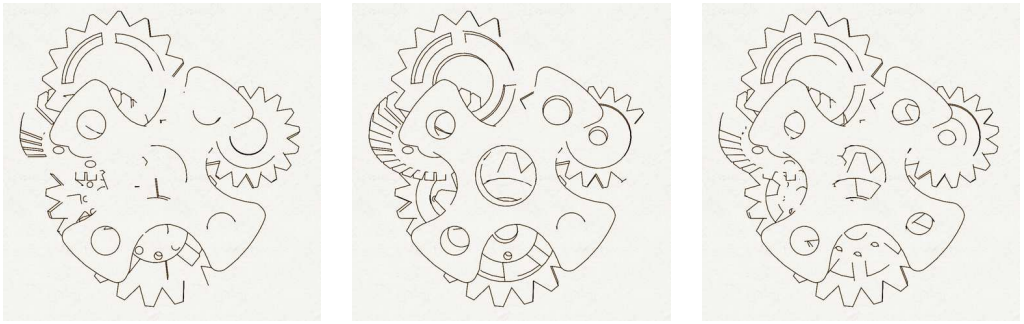
8 Discussion

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 procedural operators that can be arranged to create style modules. We presented a complete system for line drawing generation, which orchestrates the action of all style modules and operators, provides the relevant information for all stages of the calculation, and creates images.

While we have presented renderings in a variety of drawing styles, our contribution is not in the recipe for the visual effects achieved, but rather in the definition and organization of the drawing system. In particular, the flexible and modular nature of our style modules makes experimentation with styles especially easy.

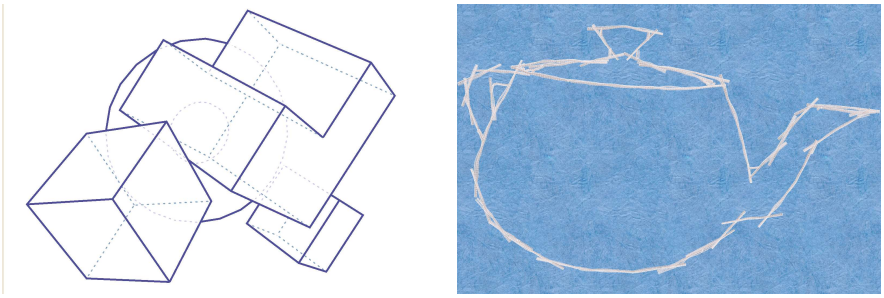
One of the major benefits of our approach is its natural redundancy, implying great flexibility: most effects can be obtained either (a) by modifying individual operators (using a programming interface), (b) by changing the set of operators in a style module or controlling their behavior (using a specific module description language), 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 procedural "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.



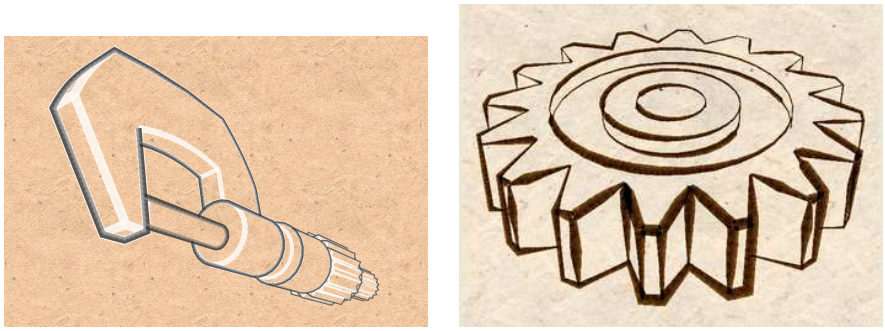
(a) (b) (c)

Figure 10: Effect of different chain ordering predicates on density control. The same density limiting parameters were used on all three images to limit the number of strokes, therefore the order in which chains are drawn matters greatly. (a) no ordering (b) chain length sort (b) depth discontinuity sort.



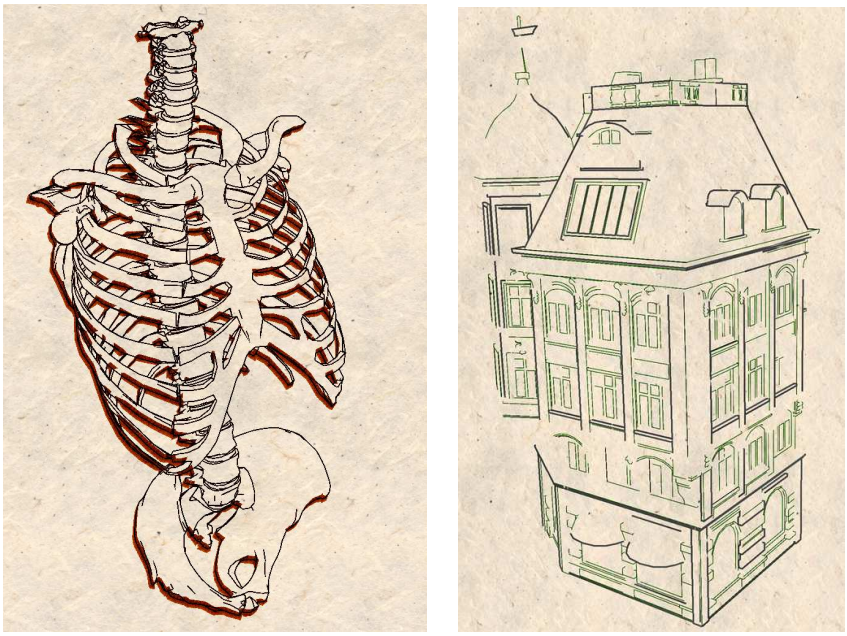
(a) (b)

Figure 11: Gallery of styles obtained using our approach.



(a) (b)

Figure 12: Gallery of styles obtained using our approach.



(a) (b)

Figure 13: a) An ink-style drawing using two style modules. The external silhouette is rendered with a calligraphic thickness that provides a shading effect, while other silhouettes and creases are rendered with a thickness and color that vary subtly with depth. b) House rendered using density limitation.

The line smoothing operator described in the paper is a good example, as well as the generation of "construction lines" from the rendering strokes.

Our system is currently limited to line drawings composed from the set of edges in our view map. Natural extensions would include (a) an extended set of line primitives in the view map, by including more general feature lines, and (b) a consistent treatment of tonal and hatching lines. Future work also includes similar procedural treatments for shading and all other NPR components.

Acknowledgements

Many thanks to Mathieu Desbrun for his help on curvature computation and to Bryan Fink for his work on mark rendering.

References

[AG99] A. Apodaca and L. Gritz, editors. *Advanced Renderman : Creating CGI for Motion Pictures*. Morgan Kaufmann, 1999.

[App67] Arthur Appel. The notion of quantitative invisibility and the machine rendering of solids. *Proc. ACM Natl. Mtg.*, page 387, 1967.

[BBN02] W.F. Bronsvoort, R. Bidarra, and A Noort. Feature model visualization. *Computer Graphics Forum*, 21(4):661–673, December 2002.

[BH00] Matthew Brand and Aaron Hertzmann. Style machines. In *Computer Graphics (Proc. SIGGRAPH)*, 2000.

[BS00] J. Buchanan and M. Sousa. The edge buffer: A data structure for easy silhouette rendering. In *Non-Photorealistic Animation and Rendering*, 2000.

[CDM+02] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Transactions on Graphics*, 21(3):302–311, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[Coo84] Robert L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 223–231, Minneapolis, Minnesota, July 1984.

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature fbw. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 317–324, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

[DMSB00] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Anisotropic feature-preserving denoising of height fi elds and bivariate data. In *Graphics Interface*, pages 145–152, 2000. ISBN 1-55860-632-7.

[DOM+01] F. Durand, V. Ostromoukhov, M. Miller, F. Duranleau, and J. Dorsey. Decoupling strokes and high-level attributes for interactive traditional drawing. In *Eurographics Workshop on Rendering*, 2001.

[DS02] Doug DeCarlo and Anthony Santella. Stylization and abstraction of photographs. *ACM Transactions on Graphics*, 21(3):769–776, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[Dur02] F. Durand. An invitation to discuss computer depiction. In *Proc. of the ACM/Eurographics Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, 2002.

[Elb98] G. Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE TVCG*, 4(1), 1998.

[EMP+94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.

[FTP99] W. Freeman, J. Tenenbaum, and E. Pasztor. An example-based approach to style translation for line drawings. Technical Report 99-11, MERL, 1999.

[GG01] Gooch and Gooch. *Non-Photorealistic Rendering*. AK-Peters, 2001.

[GSG+99] B. Gooch, P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive Technical Illustration. *ACM Symposium on Interactive 3D Graphics*, 1999.

[Hae90] P. Haeberli. Paint by numbers: Abstract image representations. *Proce. SIGGRAPH*, 1990.

[HBK02] M. Hisada, A. Belyaev, and T. Kunii. A skeleton-based approach for detection of perceptually salient features on polygonal surfaces. *Computer Graphics Forum*, 21(4):689–700, December 2002.

[Her98] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. *Proc. SIGGRAPH*, 1998.

[Her01] Aaron Hertzmann. Paint By Relaxation. In *Proceedings Computer Graphics International 2001 (Hong Kong, July 2001)*, pages 47–54, Los Alamitos, 2001. IEEE Computer Society Press.

[HJO+01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 327–340. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 289–298, Dallas, Texas, August 1990. ISBN 0-201-50933-4.

[HL94] Siu Chi Hsu and Irene H. H. Lee. Drawing and animation using skeletal strokes. *Proceedings of SIGGRAPH 94*, pages 109–118, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[HS99] J. Hamel and T. Strothotte. Capturing and re-using rendition styles for non-photorealistic rendering. In *Proc. Eurographics*, 1999.

[HSS02] Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. Creating non-photorealistic images the designer's way. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June2002)*, 2002.

[HZ00] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. *Proc. SIGGRAPH*, 2000.

[IFH+03] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications*, special issue on NPR, 2003.

[JEGPO02] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by Example: a Statistical Approach. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June2002)*, 2002.

[KMM+02] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[Kø84] J. J. Kœnderink. What does the occluding contour tell us about solid shape? *Perception*, 13:321–330, 1984.

[LDG01] Justin Legakis, Julie Dorsey, and Steven J. Gortler. Feature-based cellular texturing for architectural models. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 309–316. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[MKT+97] L. Markosian, M. Kowalski, S. Trychin, L. Bourdev, D. Goldstein, and J. Hughes. Real-time nonphotorealistic rendering. *Proc. SIGGRAPH*, 1997.

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

[PM90] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE PAMI*, 12(7):629–639, 1990.

[RC99] R. Raskar and M. Cohen. Image Precision Silhouette Edges. In *Proc. 1999 ACM Symp. on Interactive 3D Graphics*, 1999.

[SABS94] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, pages 101–108, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[SB99] Mario Costa Sousa and John W. Buchanan. Computer-generated graphite pencil rendering of 3d polygonal models. *Computer Graphics Forum*, 18(3):195–208, September 1999.

[Sch96] Simon Schofi eld. Piranesi: A 3-D Paint System. *Eurographics UK 96 Conference Proceedings*, 1996.

[SD02] Anthony Santella and Doug DeCarlo. Abstracted Painterly Renderings Using Eye-Tracking Data. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June2002)*, 2002.

[SF91] Dorée Duncan Seligmann and Steven Feiner. Automated generation of intent-based 3d illustrations. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 123–132, Las Vegas, Nevada, July 1991. ISBN 0-201-56291-X.

[SLM00] Alexander Stepanov, Meng Lee, and David Musser. *The C++ Standard Template Library*. Prentice Hall, 2000.

[SS02] Thomas Strothotte and Stefan Schlechtweg. *Non-Photorealistic Computer Graphics. Modeling, Rendering, and Animation*. Morgan Kaufmann, San Francisco, 2002.

[SWHS97] M. Salisbury, M. Wong, J. Hughes, and D. Salesin. Orientable textures for image-based pen-and-ink illustration. *Proc. SIGGRAPH*, 1997.

[TF97] J. Tenenbaum and W. Freeman. Separating style and content. In M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 662. The MIT Pr., 1997.

[Tur90] Ken Turkowski. Filters for common resampling tasks. In *Graphics Gems*, pages 147–165. Academic Press, Boston, 1990. ISBN 0-12-286166-3.

[Ups89] S. Upstill. *The Renderman Companion*. Addison-Wesley, Reading, MA, 1989.

[W+99] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley, Reading, MA, USA, third edition, 1999.

[Wil97] J. Willats. *Art and Representation*. Princeton U. Pr., 1997.

[WS94] G. Winkenbach and D. Salesin. Computer-generated pen-and-ink illustration. *Proc. SIGGRAPH*, 1994.

A Anisotropic line curvature flow

We present a 2D equivalent to curvature flow [DMSB99], where we diffuse the signed curvature on a polyline. Consider a sequence of points A, B, C on a polyline. We compute the oriented direction using $\vec{AB} + \vec{BC}$. We deduce an oriented unit normal $\vec{n}$. We then compute the curvature normal $\vec{n_C}$

using $\frac{1}{||BA||+||BC||}\left(\frac{\vec{BA}}{||BA||}+\frac{\vec{BC}}{||BC||}\right)$. We finally obtain the signed curvature $\gamma_B = \vec{n_C}\cdot\vec{n}$. If the signed curvature is positive, the curve is "turning left".

In order to smooth the polyline, we iteratively diffuse the curvature. A point P_i tries to obtain the same discrete curvatures γ as its neighbor P_{i-1} and $P_{i+1}$by moving along its normal.
$P_i^{t+1} = P^t + \lambda\,\vec{n}\sum_{k=i-1,k=i+1}\gamma_k - \gamma_i,$
where λ is the diffusion rate. The amount of smoothing depends on both λ and the number of iterations. The limit curve is a circle (constant curvature.)

We can preserve sharp corners using anisotropic curvature flow, similar to Desbrun et al. on height field [DMSB00] We add an edge-stopping function g, e.g. [PM90], that prevents curvature flow when the neighboring curvature is too different: $P_i^{t+1} = P^t + \lambda\,\vec{n}\sum_{k=i-1,k=i+1} g(\gamma_k-\gamma_i)(\gamma_k-\gamma_i),$

We can similarly diffuse the normals, in which case the limit curve is a straight line.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399