G-Strokes: A Concept for Simplifying Line Stylization

Tobias Isenberg
Department of Computer Science
University of Calgary, Canada
isenberg@cpsc.ucalgary.ca

Angela Brennecke
Department of Simulation and Graphics
Otto-von-Guericke University of Magdeburg, Germany
abrennec@isg.cs.uni-magdeburg.de

Abstract

In previous systems, only very few line properties have been used for stylization in line rendering. This is due to a complicated handling and two-way dependency of these properties and the stages of the stylization pipeline. In this paper we propose the concept of G-strokes to efficiently deal with these and many more stroke properties. This new concept allows a much simpler way of representing and handling various line properties and stylization stages. By this we make it easy to enrich the set of line stylization means by adding more properties and stylization stages and, thus, allowing more freedom and creativity for generating expressive line drawings. We show a number of possible G-strokes using both simple and complex examples to demonstrate the power of our approach.

Keywords: Non-photorealistic rendering, line rendering, line stylization, stroke pipeline, G-strokes, G-buffers.

1 Introduction

In the past decade or two, line rendering has been established as one of the major areas of research within the ever-growing field of non-photorealistic rendering (NPR) [Gooch and Gooch 2001; Strothotte and Schlechtweg 2002]. Fueled by the development of a variety of silhouette extraction algorithms [Isenberg et al. 2003] as well as feature detection techniques (e. g., [DeCarlo et al. 2003]), numerous methods for line and stroke-based rendering using a wide range of styles have been and are being conceived. In particular, the use of object-space edge extraction facilitates the further stylization and processing of these edges as so-called *strokes* since they are available in analytic form.

Typically, the stylization process is implemented using a so-called *stylization pipeline* within which strokes are processed. In general, a stylization pipeline comprises a sequence of pipeline elements or stages, each of them modifying existing, adding new, or simply preparing all data for the next stage in line. The sequential approach of this procedure, therefore, is appropriate for the analytic stroke stylization process where strokes are to be stylized in a number of steps [Northrup and Markosian 2000; Halper et al. 2003; Grabli et al. 2004].

However, the more stylization stages are being created and added to the stylization pipeline, the more difficult the stylization process itself becomes. This is because a new stage may introduce a new line property which not only has to be captured but also has to be processed (e. g., visibility or line thickness). Consequently, all other pipeline elements that already have been implemented need to be adapted as well in order to handle the new property. Only then the old stages can be used together with the new one. Likewise, every new stage also has to ensure that it can handle the already existing line properties. Therefore, a two-way dependency between the pipeline's stages and the properties of the processed stroke exists. This makes the development of a comprehensive line stylization and rendering toolkit increasingly complex and difficult.

Inspired by the groundbreaking work of Saito and Takahashi on *G-buffers* [Saito and Takahashi 1990], we propose the concept of *G-strokes* as a solution to this problem. We regard all properties added to the stroke geometry by a pipeline element as so-called G-strokes that are maintained parallel to the underlying geometry (see Figure 3). However, in contrast to Saito and Takahashi's G-buffers, G-strokes might need to be adapted during the stylization process as the underlying geometry or *topology* of the stroke may change. We demonstrate how this can be achieved and how the necessary programming work can be minimized.

The remainder of this paper is structured as follows. In Section 2 we discuss related work with respect to the concept presented in this paper. The following Section 3 discusses the problems arising from the previous handling of stylization pipelines and introduces our G-strokes concept to overcome these. In Section 4 we address a number of implementation issues and design decisions we made to realize the concept. Afterwards, Section 5 presents a number of case studies in order to illustrate the flexibility of a G-strokes based stylization. Finally, in Section 6 we summarize our contribution and discuss a few directions for future work.

2 Related Work

The field of non-photorealistic rendering has diversified and grown considerably in recent years [Gooch and Gooch 2001; Strothotte and Schlechtweg 2002]. However, line rendering has been one of the first issues that were discussed [Saito and Takahashi 1990; Dooley and Cohen 1990; Elber 1995] and this topic continues to be one of the major areas of NPR (e. g., [Kalnins et al. 2003; DeCarlo et al. 2003; Isenberg et al. 2003; Sousa and Prusinkiewicz 2003; **?**]). As one of the earliest and most important contributions for the area, Saito and Takahashi presented the G-buffer concept for enhancing the expressiveness of renditions [Saito and Takahashi 1990]. In their paper, the authors describe how to extract additional data during the rendering process, store it in the so-called G-buffers, and use them for computing NPR primitives. These primitives (silhouettes and feature lines) are then composited into the image to extend the comprehensibility of the shown objects. It is important to note that G-buffers use the same underlying topology as the rendition they were generated for, i. e., the $x \times y$ pixel matrix of the image. Thus, G-buffers form a stack of images, each recording a different property.

Although Saito and Takahashi used their G-buffers to store extracted linear features from 3D data, this happened entirely in image-space. Besides this pixel-based approach there are also two different approaches to extract edges and render strokes—hybrid methods and techniques in object-space [Isenberg et al. 2003]. In particular the latter group is of interest for this paper as it simply offers a greater freedom in terms of line parametrization and further processing than the other two. In the area of object-space stroke generation the concept of using line stylization pipelines has emerged. The pipeline's elements are used to extract significant edges from a model, concatenate them to strokes, stylize these strokes according to certain properties and parameters, and finally render them

[Halper et al. 2003; Sousa and Prusinkiewicz 2003; Grabli et al. 2004]. In particular, GRABLI et al. discuss the process of line stylization in a pipeline in greater detail. This is achieved by a modular system that first extracts and stores feature edges into a so-called viewmap. Depending on the user's needs certain edges are selected and chained to create strokes. Due to the data dimensions the system stores certain properties, e. g. an ID per edge (3D) or a line thickness per stroke (2D). This way different line styles can be combined in one image. In summary, their system follows a step-by-step approach per stroke style which is based on a local edge-to-stroke concept.

3 G-Strokes

Based on previous line stylization pipelines we first analyze the problems that occur and explain in detail the above mentioned two-way dependency between pipeline elements and processed stroke properties. We will then introduce and discuss the new G-strokes concept that eliminates this dependency making it much easier to implement and add new elements as well as properties to the stroke pipeline.

3.1 Stroke Pipelines

In previous stroke-based rendering systems (e. g., [Halper et al. 2003; Sousa and Prusinkiewicz 2003; Grabli et al. 2004]), the procedure of rendering the images followed the previously mentioned stylization pipeline approach. However, since typically not only the stylization itself but also the assembling of different strokes is performed in the pipeline we will refer to the term *stroke pipeline* instead.

A common way to represent a stroke is to store the stroke's segments as an indexed list with pointers to the actual coordinates of the strokes' vertices, each two segments being separated by a -1 (see Figure 1). This allows to store strokes efficiently since vertices typically occur at least twice (with the exception of vertices where strokes end). Stroke properties are also kept as data tracks parallel to the stroke's indices. This way they can either encode the property for the specific vertex or for the following stroke segment.

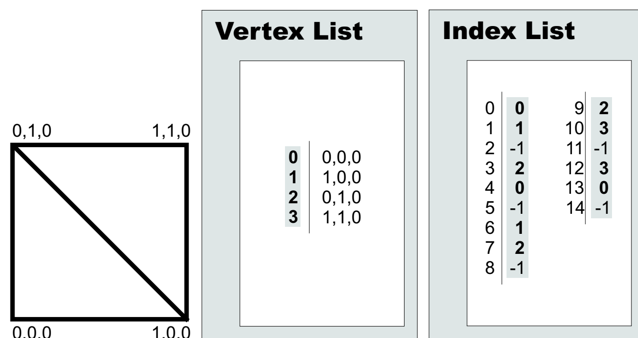


Figure 1: Vertex and index list for a simple set of strokes.

At the first stage of a stroke pipeline, certain significant edges are extracted from the model and added to the pipeline as stroke data—typically the silhouette and specific feature edges (refer to Figure 2). At the next stage, these are concatenated using adjacency information to form the strokes that will later be stylized. Each stroke then consists of a chain of stroke segments and is terminated by a -1 itself. The concatenation of single segments to segment chains simulates the human approach to line drawing where several long strokes are used to depict the objects. To simplify matters, we will refer to the term *stroke* as the set of all strokes derived from the model, therefore describing the strokes' topology.

After the strokes have been formed, typically the visible subset of them is determined (e. g., [Northrup and Markosian 2000; Isenberg et al. 2002]). Also, when using polygonal meshes as the underlying 3D models, certain artifacts such as zig-zags are removed (see, e. g., [Northrup and Markosian 2000; Isenberg et al. 2002]). Subsequently, a set of base strokes has been identified that can now be stylized in order to simulate, for example, certain traditional drawing utensils.

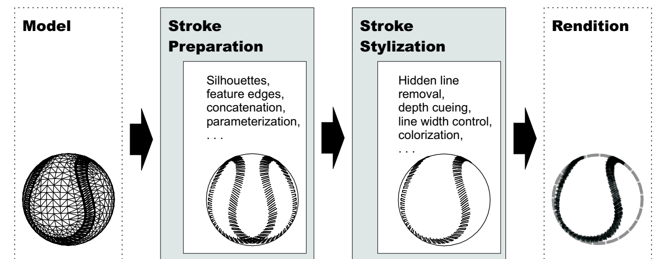


Figure 2: Typical process of stroke stylization in NPR.

The stylization process itself can only be performed with the proper stroke data and, therefore, has to be conducted at a later stage. For example, a parametrization may be assigned to the strokes to ensure a balanced scaling of textures and frame-coherent animation [Kalnins et al. 2003]. Then, the line width may be modified, e. g., according to the distance from the viewer to add depth-cueing. Also, the geometry of the strokes themselves may be modified, for example, by introducing overshooting that is meant to simulate a very sketchy look (e. g., [Grabli et al. 2004]). In order to improve the overall appearance, a spline curve may be fitted to the stroke. Finally, a texture could be assigned that simulates specific characteristics of the simulated traditional drawing utensil.

In summary, during each pipeline stage, tasks of three types are being performed: the adding of new edges/strokes to the pipeline (e. g., the silhouette and feature line extraction), the adding of properties to the strokes in the pipeline (e. g., the parametrization or the line width manipulation), and the modification of the underlying geometry of the strokes (e. g., concatenating edges, artifact removal, overshooting, and spline fitting). This includes also the removal of certain vertices or segments. However, there is no inherent sequence in which these actions have to be performed (other than that the first step has to add an initial set of edges to the pipeline). Thus, even after certain properties have been derived, the stroke geometry may change. As a consequence, each pipeline element that does change the stroke's geometry has to ensure that all existing properties are updated accordingly in order to maintain a consistent stroke representation.

Consequently, the implementation of each new geometry-modifying stylization element has to ensure that all previously implemented stroke properties are adapted as well. Furthermore, when implementing a new stylization element that introduces a new property to the pipeline, the programmer has to ensure that all previously implemented pipeline stages handle this new property accordingly. Therefore, a two-way dependency between pipeline stages and stroke properties exists that hinders the extension of line rendering systems. The more stages and properties have previously been implemented the more difficult it becomes to further extend the system.

3.2 A New Stroke Concept

In order to support the creative process of generating expressive line renditions, a wide variety of stylization elements and stroke properties need to be available to the artist. Therefore, the problem discussed above hinders the creation of truly powerful line rendering systems. In the following, we present the concept of G-strokes that not only can overcome the mentioned limitations but also reduce both the amount of necessary coding for each new stylization element and the complexity of the resulting stroke pipeline.

3.2.1 Refined Stroke Definition

Previously, a *stroke* has been defined as a path (usually a set of concatenated edges extracted from a 3D model) that is modified by a *line style* [Schlechtweg et al. 1998]. The line style itself consists of a *style curve* and, in particular, it's parameters including the deviation from a straight *style line*. In our approach we re-define the term *stroke* to be a unique sequence of indices each representing a pointer into a list of 3D coordinates. This captures the geometric aspect of the stroke and is similar to the previous path. As suggested by GRABLI et al. [Grabli et al. 2004], the style of the lines has to be captured by tracking a number of attributes. Inspired by SAITO and TAKAHASHI's *G-buffers* [Saito and Takahashi 1990], our new notion of a stroke, therefore, includes a set of usually geometric properties being maintained parallel to the index sequence that we call *G-strokes* (see Figure 3).

Similar to the G-buffers, each of the G-strokes is unique and only represents exactly one property. In contrast to the fairly static G-buffers, however, the G-strokes have to be adapted to a potentially changing stroke geometry and are, therefore, a dynamic data structure. This also results in a two-way dependency between stroke and G-strokes: the G-strokes have to be adapted according to changes of the stroke's geometry while they themselves can initiate such a change in geometry during stroke stylization.

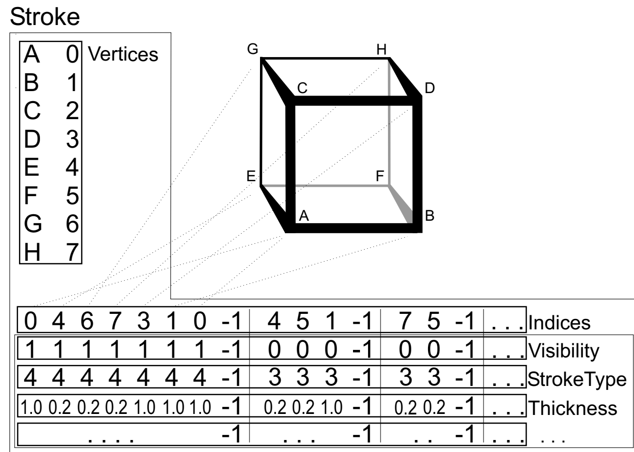


Figure 3: Parallel handling of stroke and G-strokes. Each stroke is terminated by a -1.

Hence, in contrast to, e. g., GRABLI et al.'s local stroke concept [Grabli et al. 2004], our concept pursues a global approach. All edges are extracted and chained to strokes only once. The stylization itself comprises both the process of deriving new and the modification of existing G-stroke data as well as the change of the underlying geometry of the stroke and the necessary adaption of affected G-strokes. Finally, the line style is controlled by the G-strokes which will be explained in detail in Section 3.2.4.

3.2.2 Hierarchy of G-Strokes

Depending on the type of data that is stored, G-strokes can have different data types (see Figure 4). For example, the visibility stroke that captures whether a stroke segment is visible or not is a binary property and is, therefore, stored as `BOOLEAN`. However, since each G-stroke also has to include the termination marker -1 that separates consecutive strokes (see Figure 1) we use a pseudo-`BOOLEAN` data type. A different G-stroke is the width stroke that records the line width using the `FLOAT` data type. In general, there is a limited number of basis data types that may be used for G-strokes (see Figure 4). So far we are using `BOOLEAN` (e. g., visibility), `INTEGER` (e. g., edge type and object ID), `FLOAT` (e. g., thickness and texture parameter), `Vector2D` (e. g., 2D orientation), and `Vector3D` (e. g., normal vectors and colors). Each G-stroke either denotes a property of its associated stroke vertex or of the following stroke segment.

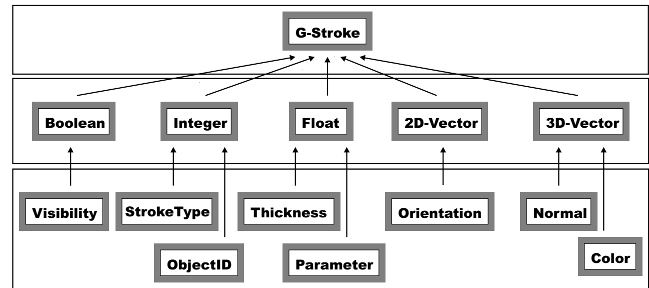


Figure 4: Hierarchy of G-strokes.

3.2.3 Separate Data Management

The logical separation of stroke geometry (coordinates and indices) and stroke properties (G-strokes) facilitates a new handling of stroke data. In this context it is important to note that the stroke properties depend on the stroke geometry and, therefore, have to adapt to potential changes of the geometry. Also, since the stroke is the underlying basis of the G-strokes they also have to adapt to changes in stroke topology.

The stroke maintains a list of its G-strokes. Hence, whenever a pipeline element changes the stroke's geometry, the stroke can call an update function in each of its G-strokes. The G-strokes, on the other hand, all implement these update functions and modify themselves accordingly. Naturally, this modification is specific to the data type and the data of each G-stroke. However, the G-Strokes' self-administration in terms of geometric modifications is crucial in this regard.

We identified the following five types of modifications that each G-stroke has to implement:

1. vertex insertion,
2. vertex removal,
3. vertex coordinate modification,
4. vertex splitting, and
5. vertex joining.

Vertex insertion is needed whenever a new vertex has to be added somewhere in an already existing stroke segment. This type of modification occurs, e. g., when deriving the visibility G-stroke (see Figure 5). In this case, some G-strokes have to interpolate their data according to the new position while others just have to replicate their values. The second modification is the deletion of a vertex from the stroke. This is necessary, for example, for artifact removal modules. It might require just the deletion of the respective G-stroke data although there may be cases where more complex adaptations may be necessary. The vertex coordinate modification is similar to inserting a new vertex. It also may require new interpolations of G-stroke values. Vertex splitting is necessary whenever a stroke has to be separated into two at a certain vertex. On the G-stroke side, this usually only requires replicating the respective data. Finally, it has to be possible to join two G-strokes at a vertex that both share (geometrically or on the 3D mesh). The handling of G-strokes in this case may be tricky since the two strokes may store different values at the vertex. Some G-strokes, in particular those that store their data with respect to segments, may just keep the according data. However, in other cases special interpolation or more complex computations may be necessary. In general, the specific G-strokes have to implement all these modification operations according to what their data represents. This is not specific to the data types but will typically even vary among G-strokes using the same base data type.

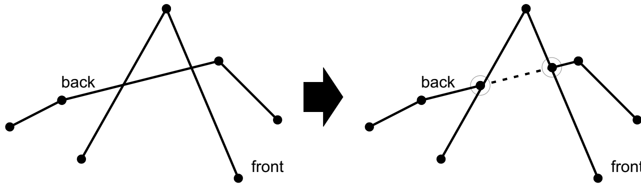


Figure 5: Inserting new vertices for visibility G-stroke.

Since the G-Strokes use the above explained self-administration, the only data that has to be exchanged is the stroke itself. With this new scheme we developed a clearly defined interface between pipeline elements, strokes, and their properties that does not change when new elements are implemented. Hence, old elements do not have to be adapted anymore when a new one implements a new property.

3.2.4 Stylization Using G-Strokes

In order to use the information stored in the G-strokes for line stylization three methods can be used, each requiring the implementation of a novel pipeline element. As our implementation of the G-strokes concept is based on the Open Inventor scene graph API the following figures refer to scene graphs and nodes rather than to pipelines and pipeline elements or stages. Nevertheless, the concept suits any pipeline or scene graph approach other than Open Inventor.

The first and most flexible way is to use a *filter element*. A filter element monitors one specified G-stroke and filters only those segments of a stroke where the G-stroke fulfills a certain condition. At the same time, it can also serve as a root element for a pipeline subtree. Hence, the pipeline subtree can now be used to stylize the selected stroke subset in a specific way. This can be realized with a scene graph approach excellently. By using several filter elements in one stroke pipeline different properties can be filtered and stylized differently. Figure 6 shows an example that demonstrates the use of an actual filter node to stylize the visible part of the strokes one way and the invisible part in a second.

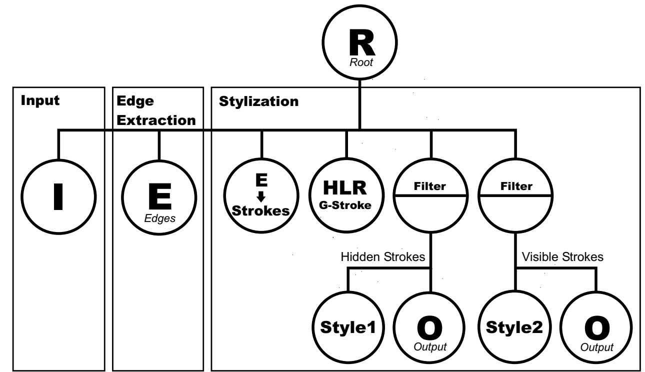


Figure 6: From the input 3D model edges are extracted, concatenated, and their visibility determined. Now, filter nodes are employed to depict visibility using separate subgraphs.

A second way to use the G-stroke data is to implement a *style element*. Such a style element uses the data in one or more G-strokes and generates output according to it—either in the form of other G-strokes or by rendering directly. Hence, a style element is very specific in the results it produces, i. e., to produce a different result a new style element has to be implemented. As the name suggests, a style element represents a complete style that is applied to the entire set of strokes at the same time (see scene graph in Figure 7). It is useful especially when a style is to be used several times.

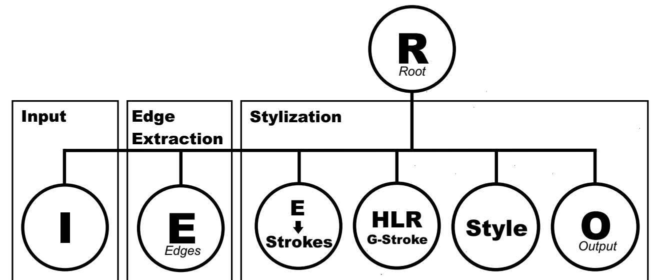


Figure 7: Instead of using filter nodes as in Figure 6, stylization can also be done with a single style node.

Finally, a hybrid form between the two methods discussed before is to use a *filter-style element*. This element applies a certain predefined style (similar to the style element) to a subset of the strokes that is being filtered similar to the filter element. That means to fully stylize a set of strokes typically a number of filter-style element have to be used (see Figure 8). Therefore, it is more flexible than the style element but not as flexible as the filter element. In addition, the scene graphs created using the filter-style element are not as big as the ones created using the filter element but bigger than the ones using the style element. The rendering time is fastest whenever the style element is used because no copying operation has to be applied to the stroke data. However, this element is most restricted in its flexibility.

4 Implementation Issues

The consequences of implementing the G-strokes concept in a NPR rendering system based on the Open Inventor scene graph API are briefly discussed in the following. In addition, in order to realize the separate data management as laid out above we used the observer

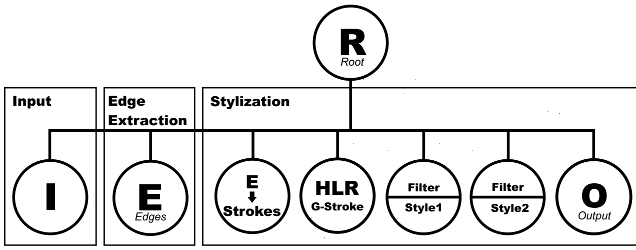


Figure 8: Stylization using a filter-style node is a hybrid between the the methods shown in Figures 6 and 7.

and the singleton pattern, two object-oriented design patterns. Finally, we explain the created G-strokes hierarchy and what has to be done to implement additional G-strokes.

4.1 Scene Graph API

Using a scene graph API has a number of important advantages. It not only allows us to use, e. g., the available rendering and 3D model handling capabilities. It also provides means to implement the stroke pipeline as a subtree of the scene graph. This is important, in particular, for implementing the filter element that uses subtrees for stylizing selected parts of the stroke set (see Figure 6). Among other things, the use of caching is required to re-establish the previous state of the pipeline after the subgraph has been traversed. We used the Open Inventor API which meets these demands. In addition, although the stroke pipeline is now a hierarchical entity its linear character is still preserved since the traversal of the scene graph imposes a linear sequence upon it.

The Open Inventor scene graph API also allows us to prepare specific scene graphs ahead of time that implement specific stylization functionality. These pre-defined subgraphs can be integrated into the stroke pipeline easily and interactively whenever this specific style is requested by the user. In addition, the scene graphs can be stored as a file and can be reloaded into the program at any time.

4.2 Implementing New G-Strokes and Pipeline Nodes

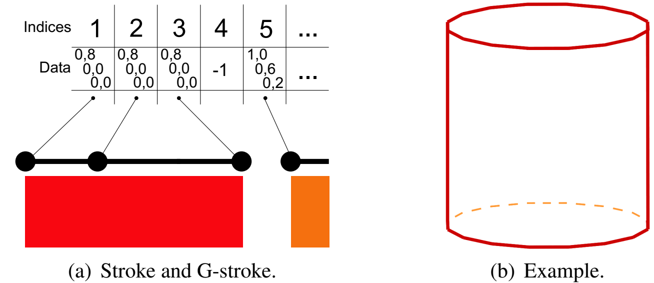
Based on the G-stroke hierarchy described in Section 3.2.2, new G-stroke classes can easily be implemented by sub-classing one of the abstract base data type G-stroke classes. Much of the behavior of a G-stroke is already implemented in these base classes or even in the main G-stroke class, such as the observer behavior and the basic data handling. A new G-stroke class only has to implement its update function.

Each new pipeline node typically just has to work with the stroke geometry and/or stroke topology. The G-strokes adapt according to their implemented update behavior. Moreover, if the new node does not alter the stroke itself, it can add data to a new or modify data of an existing G-stroke. Reading out data from a specific G-stroke can also be easily achieved by accessing individual G-strokes using their unique IDs. In any case, it is not necessary anymore to update the line properties data in the pipeline nodes.

5 Case Study and Examples

In this section we will first discuss a number of the implemented G-strokes separately and show some related simple examples to demonstrate their effect. In this overview we omit some of the obvious properties of strokes such as line width, line saturation, surface normal, and stroke orientation that are also kept as G-strokes and can be both influenced by the values in other G-strokes and used in stylization. Afterwards, we will talk about and show a few more elaborate examples to demonstrate the power of our approach.

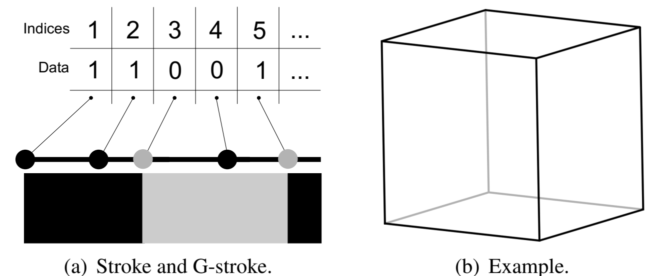
The *color G-stroke* is used to encode the color property of the strokes as 3D vectors representing red, green, and blue (see Figure 9). It can either be interpreted as the color of the starting stroke segment or as the color of the vertex. In the latter case the color would need to be interpolated along the segment between two consecutive vertices. For now, however, we use the former method and encode the color of segments directly. The color G-stroke is particularly useful in illustrations to emphasize certain objects. Although coloring of strokes has previously been possible by assigning a color directly before rendering the stroke, now this color can be varied even within one stroke pipeline.



(a) Stroke and G-stroke. (b) Example.

Figure 9: Color G-stroke.

The *visibility G-stroke* captures the visibility of the segment starting in a particular vertex using a simple `BOOLEAN` value (see Figure 10). Previously, the invisible part of a set of strokes used to be removed (hidden line removal) so that these strokes could not be used anymore in an illustration. Now, the former hidden line removal node just determines the visibility of a segment into the visibility G-stroke and later on the information can be used for stylization as shown in the example in Figure 10(b). Potentially, this requires adding new vertices to a segment when the visibility changes as was illustrated in Figure 5.

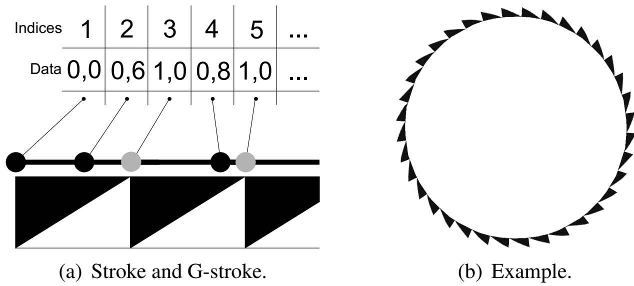


(a) Stroke and G-stroke. (b) Example.

Figure 10: Visibility G-stroke. Gray dots in (a) denote newly inserted vertices while black dots denote the original ones.

As noted before, it is necessary to track a parameter property of the strokes to ensure that textures are scaled evenly across the whole

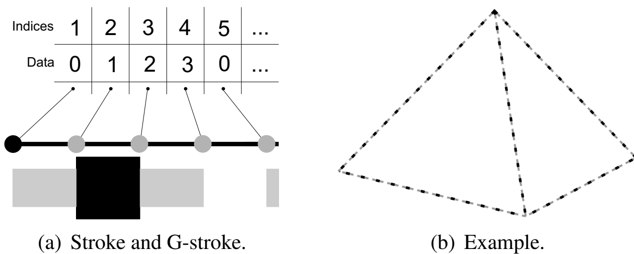
rendition. This is where the *parameter G-stroke* is employed (see Figure 11). It is determined using the projected coordinates of the strokes and stores `FLOAT` values between 0.0 and 1.0, 0.0 denoting the start of a texture and 1.0 denoting its end. In the actual system 0.0 denotes both the values of 0.0 and 1.0 at the same time since at these points both a parameter segment ends and a new one starts. Similar to the visibility G-stroke, deriving the parameter G-stroke might also require adding one or more new vertices within a segment where the parameter value reaches 1.0.



(a) Stroke and G-stroke. (b) Example.

Figure 11: Parameter G-stroke.

The *dashing G-stroke* can be used to generate a wide variety of line patterns. It subdivides each parameter segment (ranging from 0.0 to 1.0) into n evenly sized subsegments and assigns an `INTEGER` dashing ID between 0 and $n-1$ to them (see Figure 12). This can now be used, for example, to assign different textures to each of the dashing IDs and assemble a unique dasing pattern as shown in Figure 12(b).



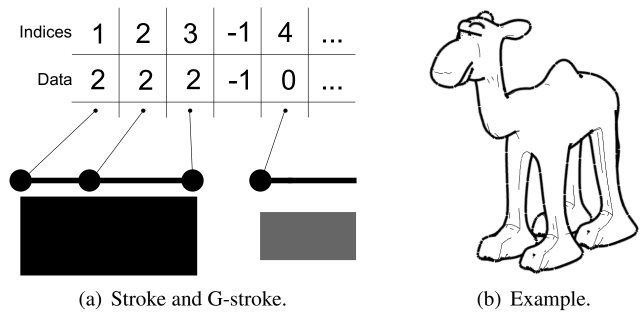
(a) Stroke and G-stroke. (b) Example.

Figure 12: Dashing G-stroke.

A very important property of the strokes in a line drawing is the type of algorithm used to extract its edges. This property is captured by the *edge type G-stroke* (see Figure 13). It can distinguish, e. g., between silhouettes and the various types of feature edges—each of them denoted by a unique ID. Even the different methods to extract silhouettes from a 3D model—edge-based or sub-polygon-based—can be assigned different edge type IDs. When used in stroke rendering, this can lead to very nice effects (see, e. g., Figure 13(b)) since in traditional renditions the different edge types are also depicted using different styles.

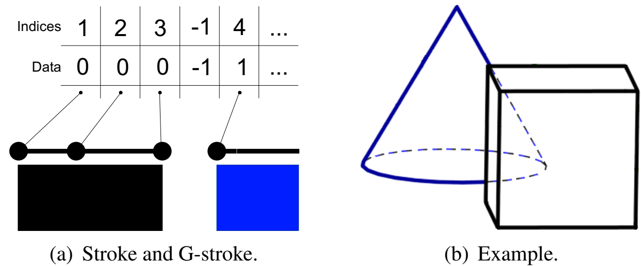
Very important for line renditions used as illustrations is to render different objects using different styles. This can easily be achieved using the *object ID G-stroke* (see Figure 14). The object ID property is extracted very early when the strokes are initially extracted from the 3D model. The values are typically later used to influence the values of other G-strokes (see Figure 14(b)) such as the color, line width, and line saturation G-strokes.

Based on the simple examples presented above, we will now show more complex examples that partially make use of more than one



(a) Stroke and G-stroke. (b) Example.

Figure 13: Edge type G-stroke.



(a) Stroke and G-stroke. (b) Example.

Figure 14: Object ID G-stroke.

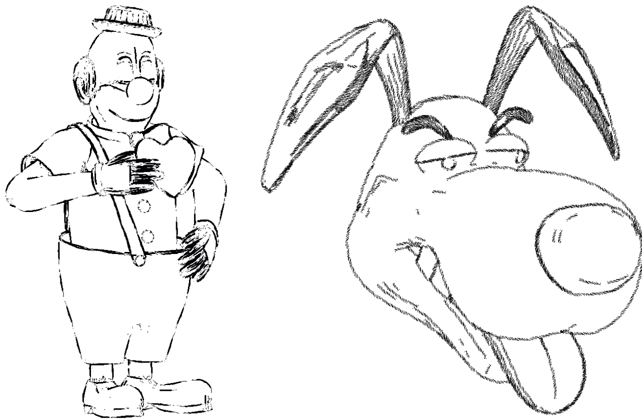
G-stroke at the same time. The first two examples in Figures 15(a) and 15(b) demonstrate the use of the parameter and the edge type G-strokes, respectively. The first example shows that, using this information, it is possible to improve the quality of textured line drawings by applying evenly scaled textures throughout the image. The second example demonstrates that the use of different stroke stylizations for different edge types (silhouettes and angle-thresholded feature lines) may be subtle yet very powerful.

The next example in Figure 15(c) shows the use of both edge type and visibility G-strokes at the same time. This type of drawing employs the visibility G-stroke to render hidden lines using a dashed texture and could easily be used for architectural or archaeological illustrations where it is important to also reveal the internal structure of buildings. Again, the edge type G-stroke is used to create the subtle but powerful effect to hint more structure than just silhouettes but not to disturb from the main shape.

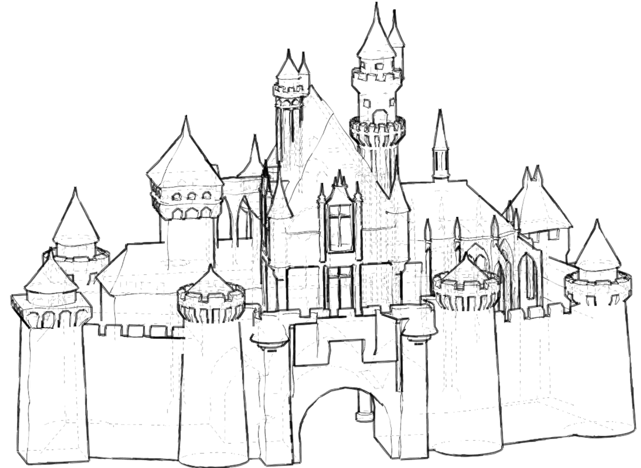
Combining the dashing G-stroke with the object ID and the color G-strokes produces the result shown in Figure 15(d). Using the object ID G-stroke the individual objects are identified. Consecutively, this data is used to assign the color G-stroke with a different color to each object ID. The dashed G-stroke is used to create a dashing pattern that is partially in the object's color and partially in black.

Finally, the last two examples show that coloring certain objects may easily be used in medical illustration. In particular in this domain it is common to color certain organs in very specific colors, such as the arteries in red as done in the first image in Figure 15(e). Of course, it is also possible to emphasize different objects using other colors as in the second image in Figure 15(e).

Using this example we show a comparison of the previous way of stylizing with the new G-strokes method with respect to the necessary scene graphs in Figure 16. Previously, each group of objects that needed to be stylized differently was extracted from the 3D model and a separate stylization pipeline was applied to it. This leads to a very complex and computationally expensive scene graph

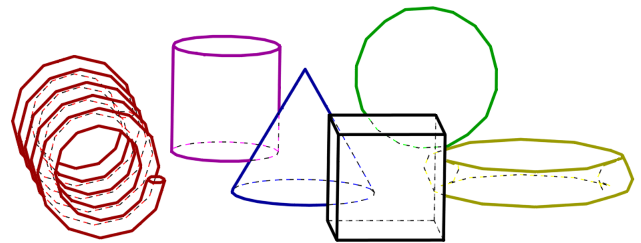


(a) Use of parameter G-stroke.

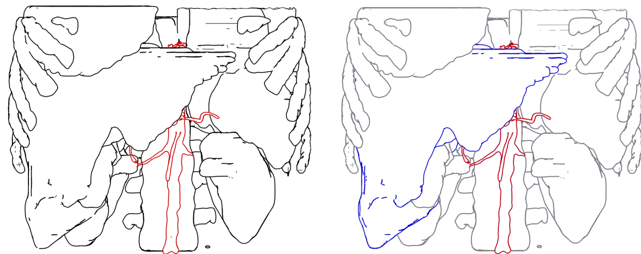


(c) Use of both edge type and visibility G-strokes.

(b) Use of edge type G-stroke.



(d) Use of dashing, object ID, and color G-strokes.



(e) Object ID G-stroke used to emphasize objects in illustrations.

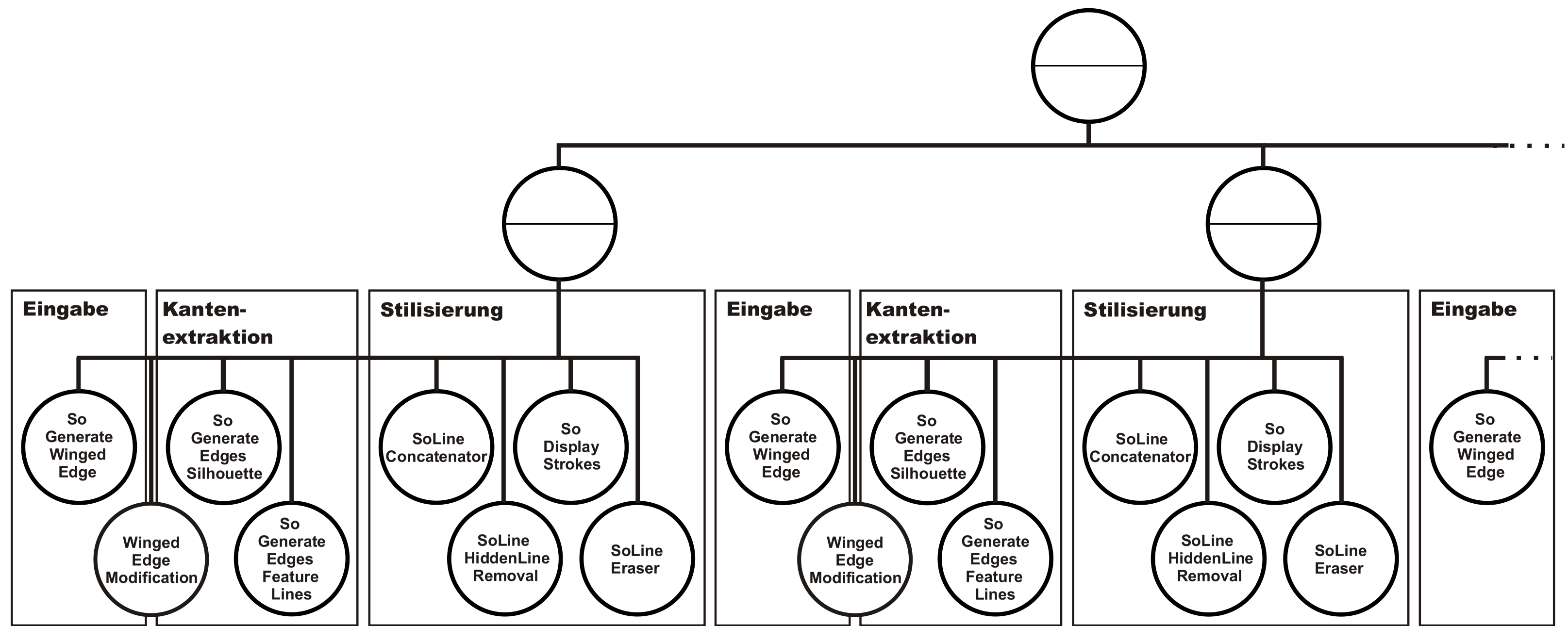
Figure 15: More complex examples for using G-stroke to stylize line renderings.

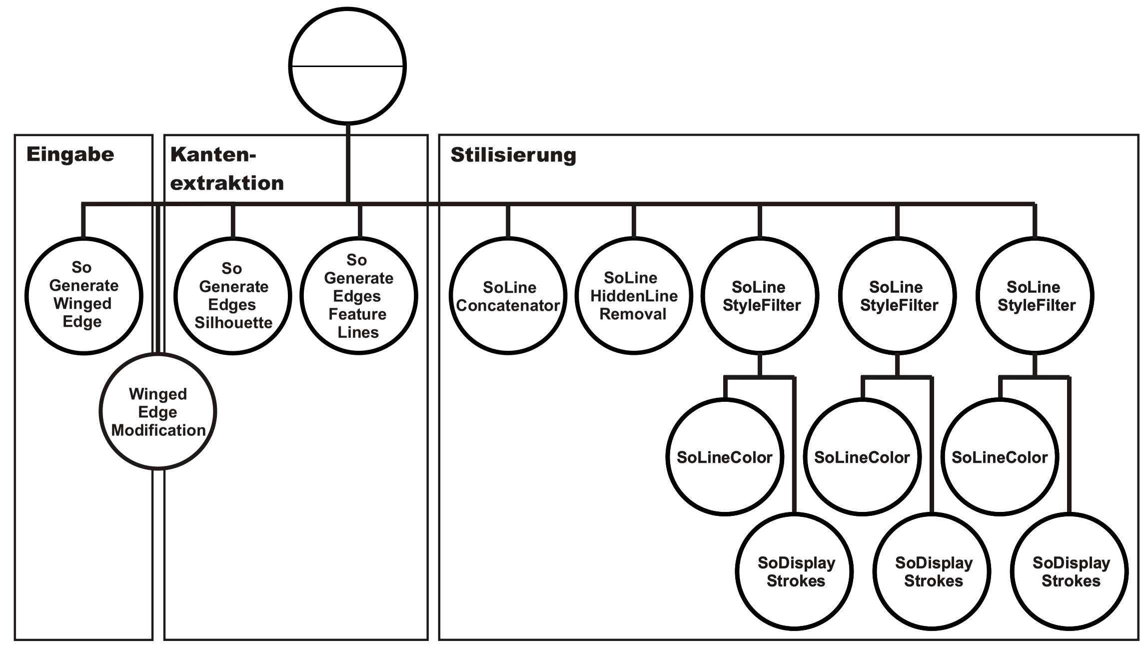
(upper part of Figure 16). With the G-stroke approach this is not necessary anymore. Now, we can simply use one single stylization pipeline and filter the generated strokes according to the automatically extracted object ID G-stroke. Afterwards, we just have to use a few sub-pipelines to stylize and render each group of objects accordingly (lower part of Figure 16).

6 Summary and Future Work

In this paper we presented an approach that allows to handle all stroke properties that may occur in a line rendering system. In contrast to previous local methods we propose a global technique that allows to store and manage the attributes in a coherent and consistent way. We demonstrated that there is a dependency between strokes and their properties—called G-strokes—and how this can be solved by separating the data management between strokes and G-strokes. We discussed what types of G-strokes may occur in a typical system and how these can be arranged in a G-strokes hierarchy. This hierarchy makes the development of new strokes very easy because most of the functionality is implemented in the upper levels so it simply needs to be re-defined. Moreover, we demonstrated the use of many G-strokes both with simple and more advanced examples. Finally, we could show that our approach simplifies the creation of elaborate stylization techniques by significantly reducing the size of the used scene graphs.

Future work includes the extension of the G-strokes concept in terms of creating new specific G-strokes. For example, the several notions of curvature as, e. g., in [Sousa et al. 2003] could be tracked each as an individual G-stroke, the degree of an extracted feature edge (e. g., in terms of angle) could be stored as a G-stroke and used to influence the line width, and many more. Also, new nodes have to be implemented that make use of the G-stroke data for stroke modification and stroke stylization. For example, over-





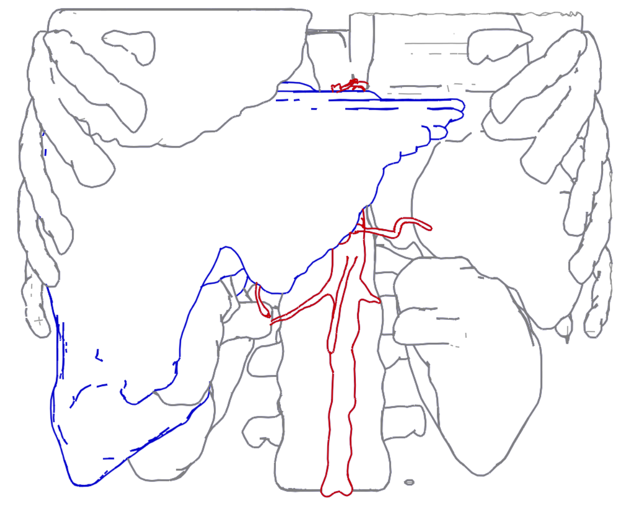


Figure 16: Comparison of a scene graph that uses the previous way for stylization (above) with one that uses G-strokes (below) for rendering objects in different colors.

shooting could be implemented as in [Grabli et al. 2004], the curvature G-strokes could be used to influence the line thickness as in [Sousa et al. 2003], etc.

Very important for future line rendering systems is the development of good and efficient interaction metaphors and/or interfaces that allow users to intuitively specify rendering styles. For example, the system presented by HALPER et al. [Halper et al. 2002] could serve as a starting point for this task.

In addition, the G-buffer concept can also be applied to domains other than pixel images or strokes. In fact, the variety of texture types that are used in regular rendering could considered to be G-properties of the 3D surface model.

References

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive Contours for Conveying Shape. *ACM Transactions on Graphics 22*, 3 (July), 848–855.

DOOLEY, D. L., AND COHEN, M. F. 1990. Automatic Illustration of 3D Geometric Models: Lines. In *Proc. 1990 Symp. on Interactive 3D Graphics*, ACM Press, New York, 77–82.

ELBER, G. 1995. Line Illustrations $\in$ Computer Graphics. *The Visual Computer 11*, 6 (June), 290–296.

GOOCH, B., AND GOOCH, A. 2001. *Non-Photorealistic Rendering*. A K Peters, Ltd., Natick.

GRABLI, S., TURQUIN, E., DURAND, F., AND SILLION, F. X. 2004. Programmable Style for NPR Line Drawing. In *Proc. Rendering Techniques 2004*, EUROGRAPHICS Association, 33–44, 407.

HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2002. Creating Non-Photorealistic Images the Designer's Way. In *Proc. NPAR 2002*, ACM Press, New York, 97–104.

HALPER, N., ISENBERG, T., RITTER, F., FREUDENBERG, B., MERUVIA, O., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. OpenNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering. In *Proc. Pacific Graphics 2003*, IEEE, Los Alamitos, CA, 424–428.

ISENBERG, T., HALPER, N., AND STROTHOTTE, T. 2002. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. *Computer Graphics Forum 21*, 3 (Sept.), 249–258.

ISENBERG, T., FREUDENBERG, B., HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications 23*, 4 (July/Aug.), 28–37.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent Stylized Silhouettes. *ACM Transactions on Graphics 22*, 3 (July), 856–861.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic Silhouettes: A Hybrid Approach. In *Proc. NPAR 2000*, ACM Press, New York, 31–37.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible Rendering of 3-D Shapes. In *Proc. SIGGRAPH 90*, ACM Press, New York, 197–206.

SCHLECHTWEG, S., SCHÖNWÄLDER, B., SCHUMANN, L., AND STROTHOTTE, T. 1998. Surfaces to Lines: Rendering Rich Line Drawings. In *Proc. WSCG'98*, vol. 2, 354–361.

SOUSA, M. C., AND PRUSINKIEWICZ, P. 2003. A Few Good Lines: Suggestive Drawing of 3D Models. *Computer Graphics Forum 22*, 3 (Sept.), 381–390.

SOUSA, M. C., FOSTER, K., WYVILL, B., AND SAMAVATI, F. 2003. Precise Ink Drawing of 3D Models. *Computer Graphics Forum 22*, 3 (Sept.), 369–379.

STROTHOTTE, T., AND SCHLECHTWEG, S. 2002. *Non-Photorealistic Computer Graphics. Modelling, Animation, and Rendering*. Morgan Kaufmann Publishers, San Francisco.