G-Strokes: A Concept for Simplifying Line Stylization

Tobias Isenberg [a] and Angela Brennecke [b]

[a] *Department of Computer Science, University of Calgary, Canada*

[b] *Department of Simulation and Graphics, Otto-von-Guericke University of Magdeburg, Germany*

Abstract

In most previous NPR line rendering systems, geometric properties have been directly used to extract and stylize certain edges. However, this approach is bound to a tight stylization of strokes as the focus lies on the edge extraction. Styles are applied to the currently extracted edges, making it necessary to re-do certain computations whenever several different styles are to appear concurrently in the same rendition. Consequently, the generation of renditions is often constrained to one or two styles to keep computational cost low. To broaden the possibilities of generating highly expressive line drawings we introduce the concept of G-strokes. In contrast to the above-mentioned approach, we propose to keep all edges and to extract the geometric properties instead. According to these properties, one style could be applied to a particular set of edges and another style could be applied to another set of edges without having to extract the designated edges anew. This makes it easy to enrich the set of line stylization means, allowing more freedom and creativity for generating varied line drawings. We show a number of possible G-strokes using both simple and complex examples to demonstrate the power of our approach.

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

1 Introduction

In the past two decades, line rendering has been established as one of the major areas of research within the growing field of non-photorealistic rendering (NPR) [1,2]. Fueled by the development of a variety of silhouette extraction algorithms [3] as well as feature detection techniques (e. g., [4]), numerous methods for line and stroke-based rendering using a wide range of styles have

Article published in Computers & Graphics 30(5) (2006) 754–766

been and are being conceived. In particular, the use of object-space edge extraction facilitates the further stylization and processing of these edges as *strokes* since they are available in analytic form.

Typically, the stylization process is implemented using a *stylization pipeline* within which strokes are processed. In general, a stylization pipeline comprises a sequence of pipeline elements. At each stage of the pipeline, data is either modified, added, or simply prepared for the next element 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 [5,6,7]. The interconnection between the line drawing and its generation technique is crucial to the below-stated problem and the necessity of the G-Strokes concept: The rendition's explanatory power depends on the stroke's topology and style which are in turn established and altered by the pipeline elements. Therefore, the pipeline elements as well as their combination have to be as flexible as possible to achieve the favored line drawings.

However, the more stylization elements are being created and added to the stylization pipeline, the more difficult the stylization process itself becomes. This is because a new element may introduce new data that not only has to be captured but also has to be processed. For instance, texture or line thickness parameters may be added to the coordinates and their indices of the current stroke set. Whenever the indexing of the strokes changes, the parameters also have to change. Consequently, all pipeline elements that already have been implemented need to be adapted as well in order to handle the new data. Only then can the old elements be used together with the new one. Likewise, every new element also has to ensure that it can handle the increasing number of already existing data sets. Therefore, a two-way dependency between the pipeline's elements and the processed stroke data exists. This makes the development of a comprehensive line stylization and rendering toolkit increasingly complex and difficult to manage.

Inspired by the groundbreaking work of Saito and Takahashi on *G-buffers* [8], we propose the concept of *G-strokes* as a solution to this problem. We regard all data added to the stroke (coordinates and indices) by a pipeline element as geometric stroke properties and call them *G-strokes*. These are maintained parallel to the underlying geometry. In this context, for example, the stroke's parameterization (e. g., for texturization) and visibility are geometric properties. The latter could be captured in a G-stroke telling the current stroke set which strokes are visible and which are not and could then be used to apply a certain style to the extracted edges (see Figure 3).

In contrast to Saito and Takahashi's G-buffers, G-strokes might need to be adapted during the stylization process since 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, making it easy to add new pipeline elements without having to care for the existing data sets.

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

2 Related Work

The field of non-photorealistic rendering has diversified and grown considerably in recent years [1,2]. However, line rendering was one of the first issues to be discussed [8,9,10] and this topic continues to be one of the major areas of NPR (e. g., [3,4,7,11,12]). 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 [8]. In their paper, the authors describe how to extract additional data during the rendering process, store it in what they call G-buffers, and use it 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 [3]. In particular, the latter group is of interest for this paper as it offers a greater freedom in terms of line parametrization and further processing than hybrid and image-based techniques. 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 [6,7,12]. In particular, Grabli et al. discuss the process of line stylization in a pipeline in greater detail. Their main contribution to NPR line drawing is the separation of lines or edges from the attributes which guide the line stylization. This separation is achieved by collecting as much information on the scene as possible. The gathered data is categorized into 3D

scene information (3D coordinates, normals, object IDs, ect.), auxiliary maps (local average depth, item buffer, etc.), the view map (a planar graph which is received by projecting the extracted feature edges into the view plane), and the current drawing (local stroke density). This data is then used to create style sheets for stylizing the lines extracted from a 3D model. These can now be arranged to model different NPR-pipelines. The individual style sheet modules operate on the 2D view map's edges and consist of selecting, chaining, splitting, and assigning attribute operation. Furthermore, several of these modules can be used simultaneously. In a final step, each resulting image layer is combined into a single image leading to a huge amount of different styles which can be used in one image. However, in contrast to our technique, they are bound to a sequential pipeline in a greater extent. Moreover, the information acquiring step is fairly complex and could be handled in an easier way.

3 G-Strokes

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. The realization of previous stylization pipelines hinders the creation of truly powerful line rendering systems. In the following, we present the concept of G-strokes that not only can overcome these limitations but also reduce both the amount of necessary coding for each new stylization element and the complexity of the resulting stroke pipeline.

3.1 A New Stroke Concept

A common way to represent an edge is to store the edge's segments as an indexed list with pointers to the actual coordinates of the edge's vertices, each segment being separated by a -1 (see Figure 1). This method is more efficient since vertices typically occur at least twice (with the exception of vertices where strokes end).

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* [13]. 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.

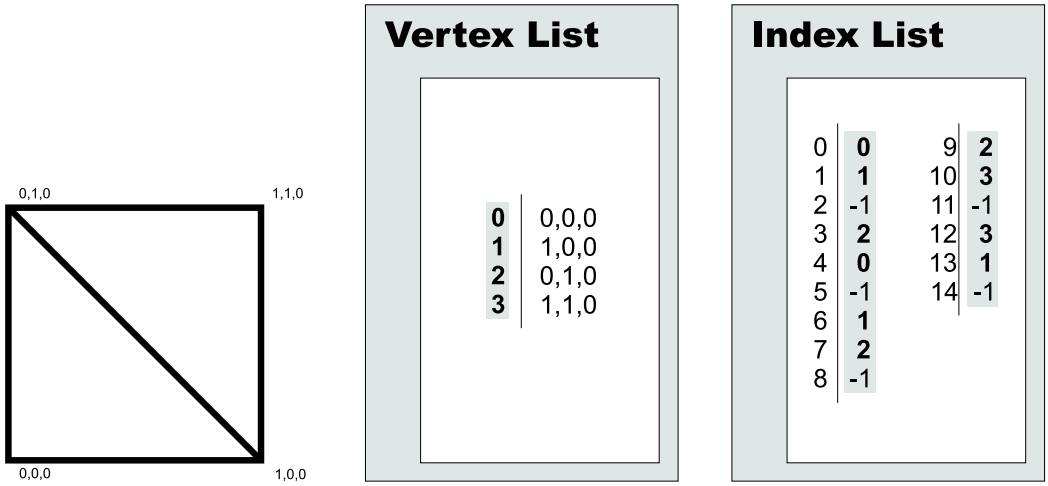


Fig. 1. Vertex and index list for a simple set of strokes.

3.1.1 G-Stroke Definition

As suggested by Grabli et al. [7], the style of the lines has to be captured by tracking a number of attributes. Inspired by Saito and Takahashi's *G-buffers* [8], our new notion of a stroke includes a set of usually geometric properties being maintained parallel to the index sequence that we call *G-strokes* (see Figure 2).

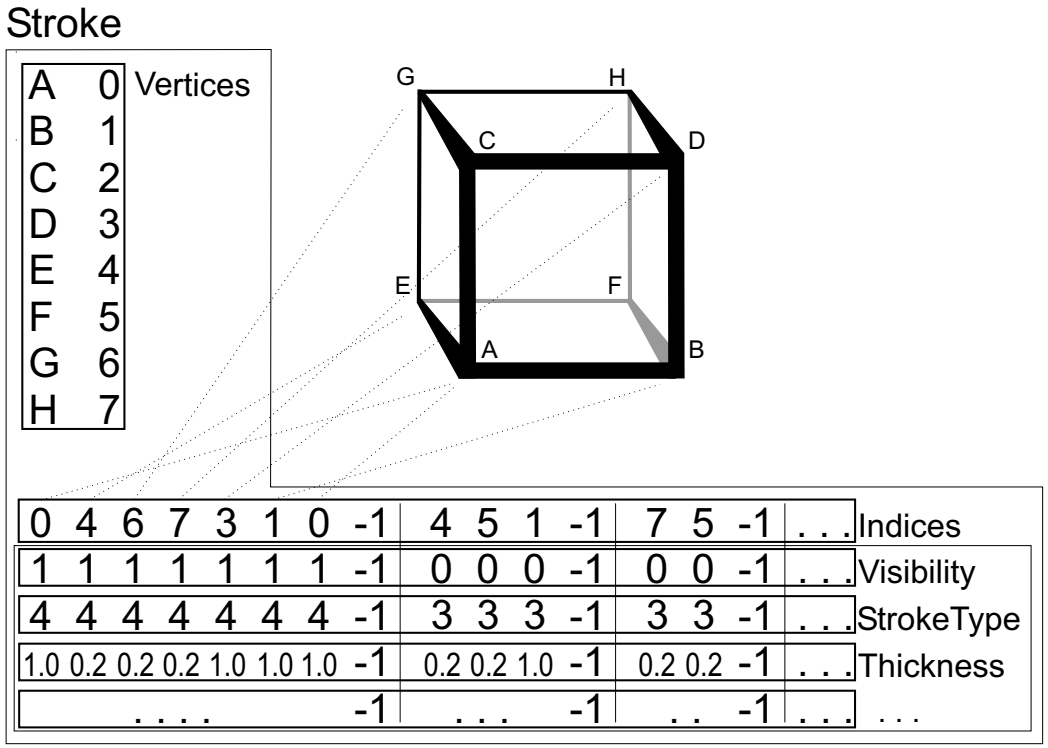


Fig. 2. Parallel handling of stroke and G-strokes. Each stroke is terminated by a -1.

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 or topology and are, therefore, a dynamic data structure. This results in a two-way dependency between stroke and G-strokes and is crucial for the G-stroke's definition: the G-strokes have to be adapted according to changes in the stroke's geometry while G-strokes themselves can initiate such a change in geometry during stroke stylization. Phrased differently, a G-stroke represents a certain property of the stroke and can be used whenever a style is to be applied to a stroke or its segments. Whenever the stroke changes (e. g., due to artefact removal), the G-stroke has to adjust to this modification. In turn, when the stroke is being stylized, the G-stroke can be used to alter the stroke's topology (e. g., different drawing of visible and invisible edges, see Figure 3). G-strokes may capture geometric properties (such as orientations, parameterizations etc.) as well as non-geometric properties (e. g., edge type, color etc.).

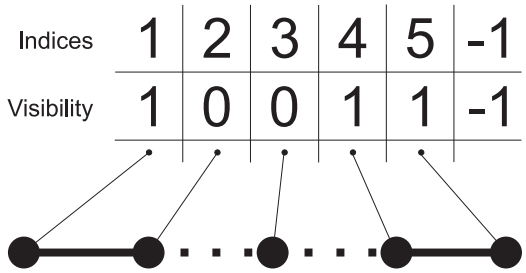


Fig. 3. A stroke and its visibility property captured by a G-stroke.

3.1.2 Hierarchy of G-Strokes

In order to encapsulate central G-stroke features that are to be inherent to any G-stroke we designed an object-oriented G-stroke hierarchy (see Figure 4). The root of this hierarchy is a general G-stroke, primarily maintaining the functionality of G-stroke adaptation to stroke modification. Many specific G-strokes are based on the same abstract data type (e. g., an edge type ID G-stroke and an object ID G-stroke that both are based on `INTEGER` values). These G-strokes exhibit similar behavior which can also be encapsulated in the G-stroke hierarchy for frequently used abstract data types. This ensures that the common functionality of G-strokes only has to be maintained once and can be inherited by concrete stroke properties.

One important difference between individual G-strokes is whether the values of the G-stroke denote a property of the associated stroke vertices or of the following stroke segments. This fact has to be observed in all specific G-strokes of the hierarchy.

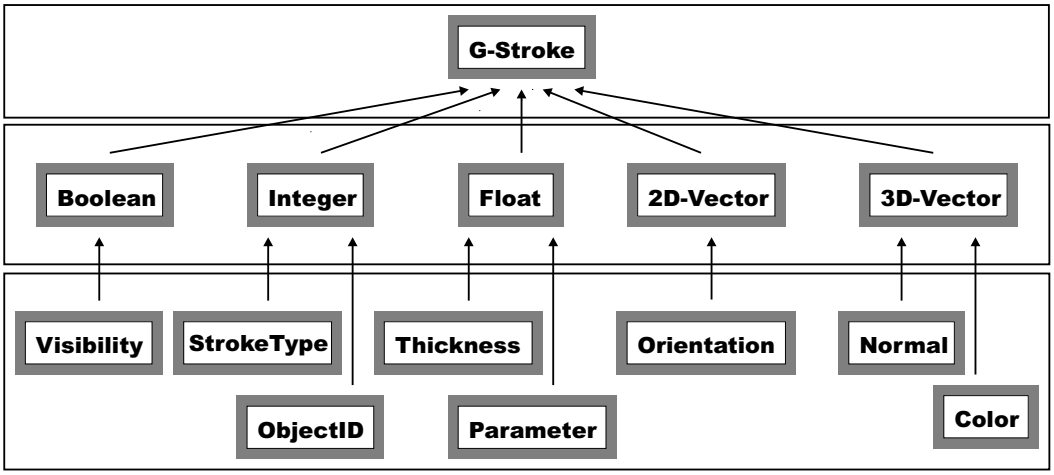


Fig. 4. Hierarchy of G-strokes.

3.2 Previous Stroke Pipelines

In former stroke-based rendering systems (e. g., [6,7,12]), the procedure of rendering the images followed the previously mentioned stylization pipeline approach. However, since typically not only the stylization itself but also the extraction and assembling of different strokes is performed in the pipeline we will instead refer to the combination of both these stages as the *stroke pipeline*.

At the first stage of a stroke pipeline, certain significant edges are extracted from the model and added to the pipeline as geometry data—typically the silhouette and specific feature edges (refer to Figure 5). There are a number of methods for silhouette edge extraction ranging from the trivial method (selecting edges that share a front- and a back-facing polygon) to pre-processing techniques on to hardware acceleration via the GPU [14,15,16,17,18,19]. In contrast to silhouette edges, feature edges rather support the object's inner shape. There are several types of feature edges extracted by different systems including creases, border edges, self-intersections, and suggestive contours [3,4,9,18,20].

At the next stage, the extracted edges are concatenated using adjacency information to form the strokes that will later be stylized. Each stroke then consists of a chain of edges or stroke segments and is terminated by a -1. 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 generic term for 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., [5,21]). Also, when using polygonal meshes as the underlying 3D models, certain artifacts such as zig-zags are removed (see, e. g., [5,21]).

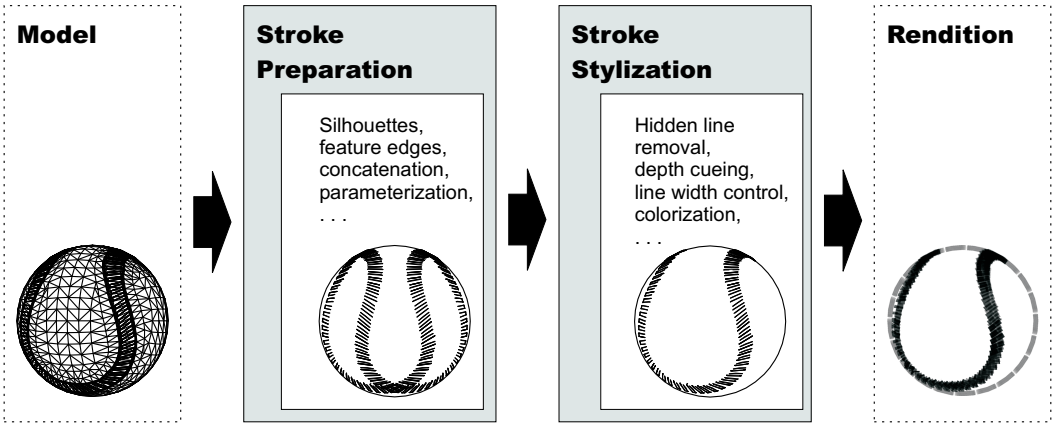


Fig. 5. Typical stroke pipeline in NPR.

Subsequently, a set of base strokes has been identified that can then be stylized.

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, one or more parameterizations may be assigned to the strokes to ensure a balanced scaling of textures and frame-coherent animation [11]. 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., [7]). 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 further data sets (*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 must be to add an initial set of edges to the pipeline). Thus, even after certain properties have been derived, the stroke geometry or topology may change. As a consequence, each pipeline element that does change the stroke's geometry or topology has to ensure that all existing properties are being updated accordingly in order to maintain a consistent stroke representation.

Consequently, the implementation of each new geometry- or topology-modifying stylization element has to ensure that all previously added 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 elements and stroke properties exists that hinders the extension of line rendering systems. The more elements and properties have previously been implemented, the more difficult it becomes to further extend the system.

3.3 New Stroke Pipelines

The logical separation of stroke geometry (coordinates and indices) and stroke properties (G-strokes) facilitates a new handling of stroke data. G-strokes as the stroke's properties now depend on the stroke's geometry and topology and, therefore, can automatically adapt to potential changes of the stroke. Hence, for being able to apply manipulations to the stroke that potentially lead to changes of its geometry and topology, only the coordinates and indices themselves have to be adapted—all other necessary changes are performed automatically.

In order to realize the above dependency, we provide the stroke with a list of its G-strokes. Consequently, whenever a pipeline element changes the stroke's geometry, the stroke calls 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. This is necessary because of the G-stroke hierarchy's encapsulation of behavior that allows for the consistent handling of G-strokes. Naturally, this modification is specific to the data type and the actual data of each G-stroke.

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 6). 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 value, though there may be cases where more complex adaptations may be necessary. The vertex coordinate modification is similar to inserting a new

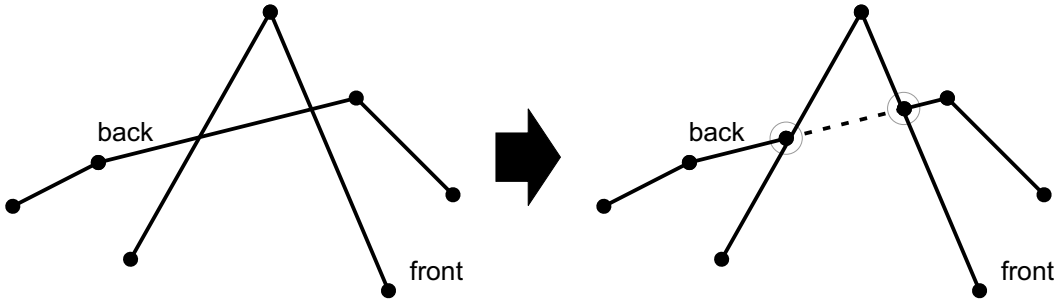


Fig. 6. Inserting new vertices for visibility G-stroke.

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. To solve issues that may arise in this context, we developed the priority G-stroke, which prevents the deletion of important vertices. This boolean G-stroke stores a 1 for vertices or segments with a high priority and a 0 for low priority values. A high priority prohibits deletion whereas a low priority permits the deletion. Whenever a G-stroke is composed, the priority of the current value has to be confirmed.

In general, the specific G-strokes have to implement all these modification operations according to what their data represents. The necessary modification operation is not specific to the data types but will typically even vary among G-strokes using the same base data type.

Since the G-Strokes use the above explained self-administration, the only data that has to be exchanged is the stroke's coordinates and indices. 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.

3.4 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 traditional 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. For example, a filter element could be used to filter out the invisible segments of a stroke by observing the visibility stroke. The element can then serve as a root element for a pipeline subtree that stylizes the filtered stroke segments in a specified way. By using several filter elements in one stroke pipeline, different properties can be filtered and stylized differently. Figure 7 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. This is similar to the concurrent handling of the style modules by Grabli et al. [7].

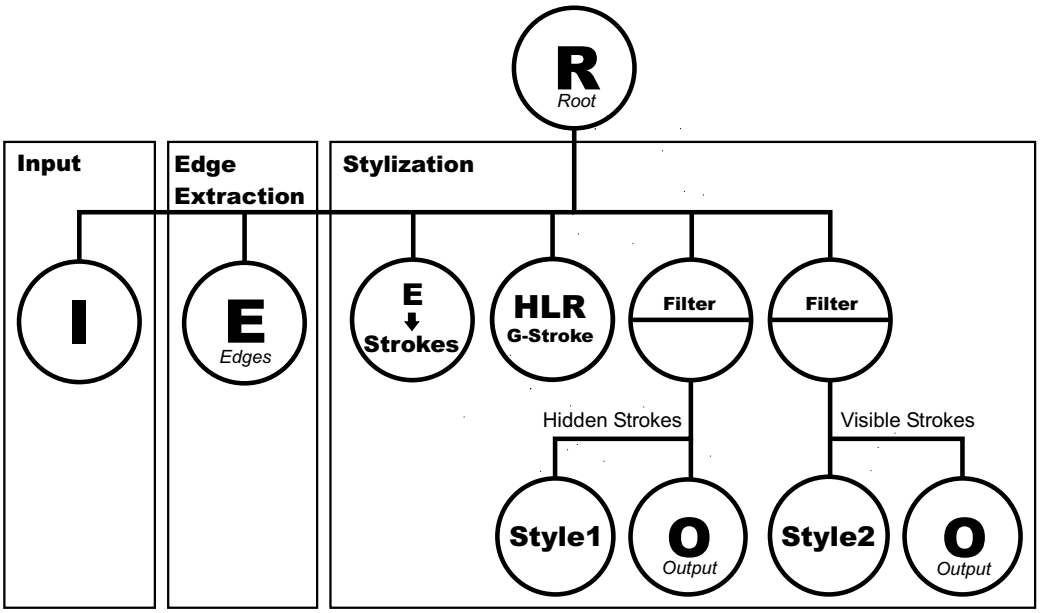


Fig. 7. From the input 3D model edges are extracted, concatenated, and their visibility determined. Now, filter nodes are employed to depict visibility using separate subgraphs.

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 8). It is useful, in particular, when a style is to be used several times.

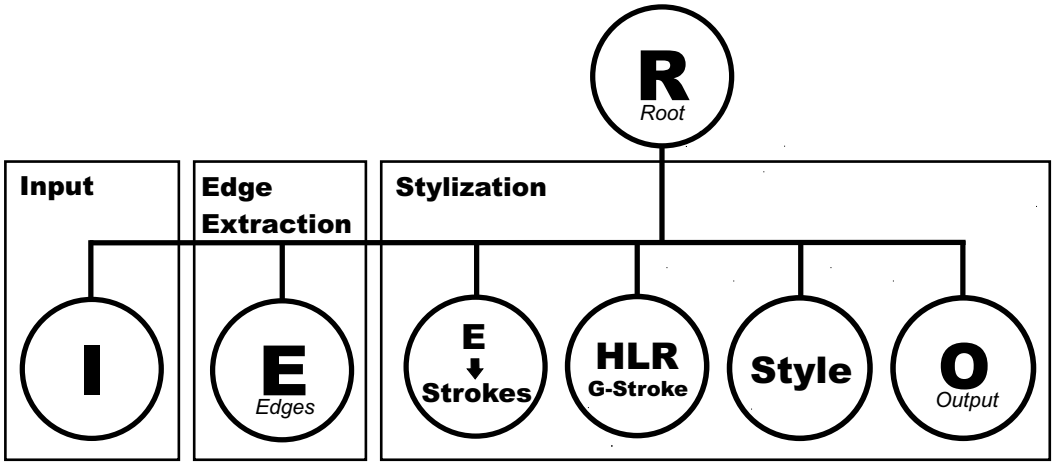


Fig. 8. Instead of using filter nodes as in Figure 7, stylization can also be done with a single style node.

Finally, a hybrid form between the two methods discussed before is to use a *filter-style element*. This element applies a certain pre-defined style (similar to the style element) to a subset of the strokes that is being filtered similar to the filter element. This means that, to fully stylize a set of strokes, typically a number of filter-style elements have to be used (see Figure 9). 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.

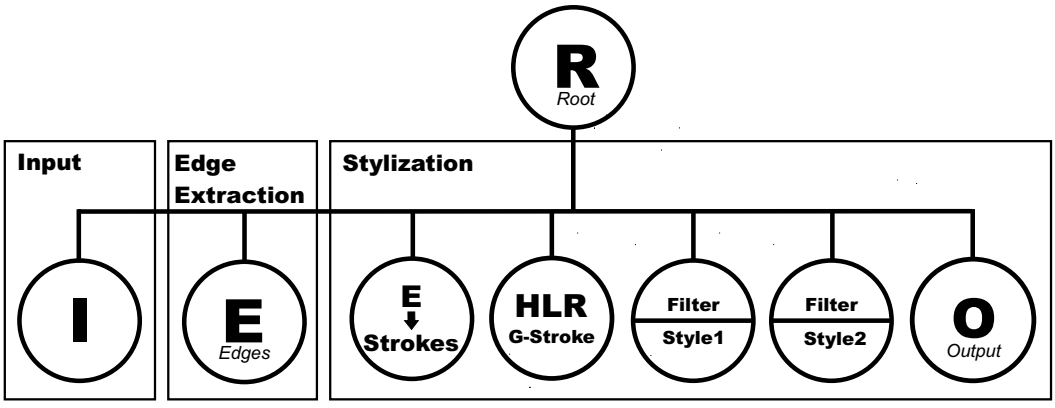


Fig. 9. Stylization using a filter-style node is a hybrid between the the methods shown in Figures 7 and 8.

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. Finally, we explain the created G-strokes hierarchy and what has to be done in order to implement additional G-strokes.

In addition, in order to achieve an implementation close to the concept laid out above, we used two object-oriented design patterns. The first one, the *singleton pattern*, describes how to limit the number of an object's instances to a single one and, thus, suited the realization of the G-Stroke's uniqueness perfectly. The second design pattern we deployed was the *observer pattern*. It is based on the relationship between one subject and (many) observers. Whenever the subject changes, all observers adapt to this change and it, therefore, was suitable for realizing the relationship between Stroke and G-stroke.

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, but also provides a 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 7). Among other things, the use of caching is required to reestablish 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 3.1.2, 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 im-

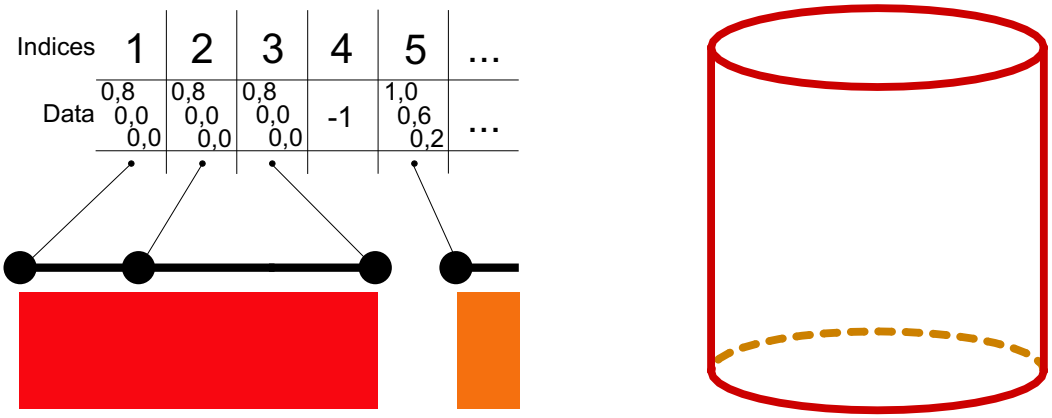
plemented in these base classes or even in the main G-stroke class, such as the observer behavior and the basic data handling. A new G-stroke class only has to implement its own update function so that the G-Stroke adapts to topology changes correctly.

Each new pipeline node typically just has to work with the stroke geometry and/or stroke topology data. 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 G-stroke 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. In any case, it is not necessary to update specific 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 10). It can either be



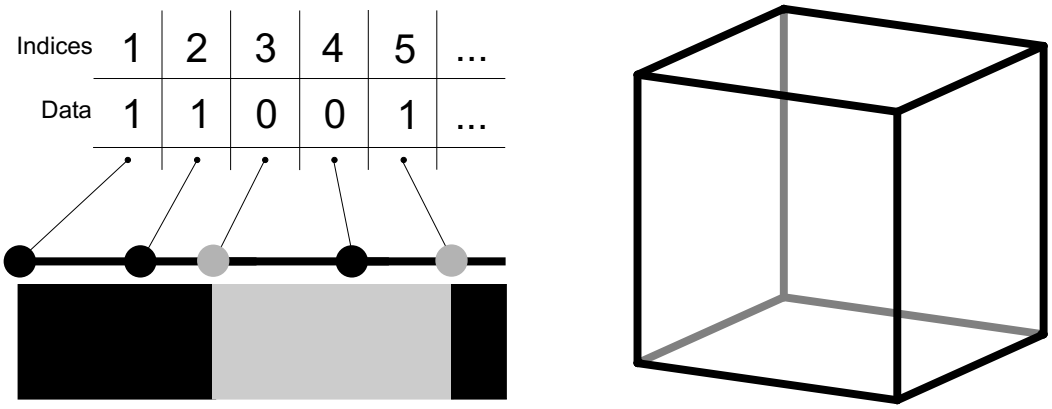
(a) Stroke and G-stroke. (b) Example.

Fig. 10. Color G-stroke. It stores the color to be used by a specific stroke segment.

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 entire stroke segments directly. The color G-stroke is particularly useful in illustrations to emphasize certain objects. Although coloring of strokes has previously been accomplished by assigning a color directly before rendering the stroke, we allow the color to be varied even within one stroke (see example in Figure 16(e)).

The *visibility G-stroke* captures the visibility of the segment starting at a particular vertex using a simple `BOOLEAN` value (see Figure 11). In many



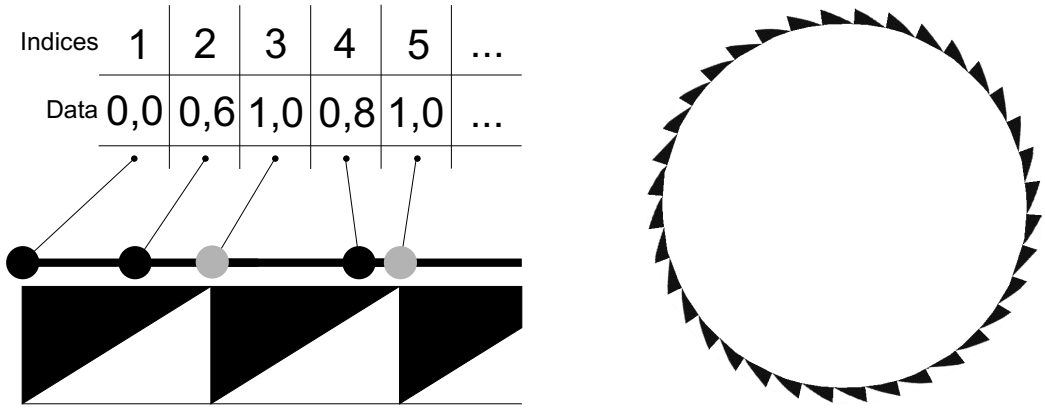
(a) Stroke and G-stroke. (b) Example.

Fig. 11. Visibility G-stroke. Gray dots in (a) denote newly inserted vertices while black dots denote the original ones. These can now be used to stylize hidden lines different from visible ones.

previous approaches, 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 11(b). Potentially, this requires adding new vertices to a segment when the visibility changes, as was illustrated in Figure 6.

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 12). 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.

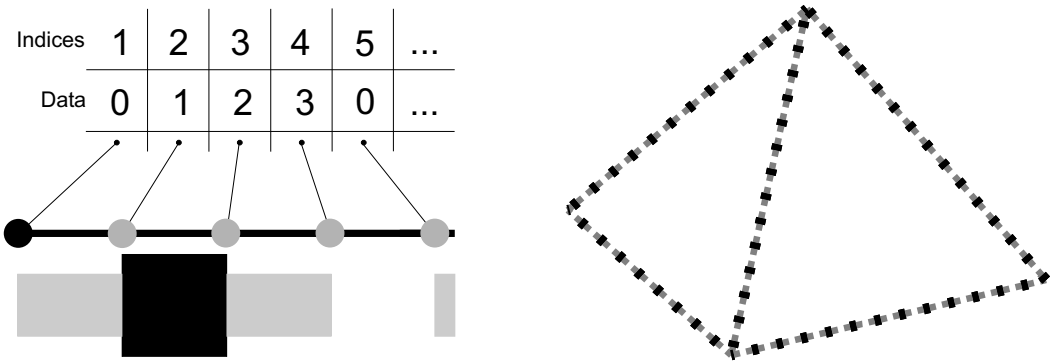
The *dashing G-stroke* can be used to generate a wide variety of line patterns. It subdivides each parameter segment (ranging from 0.0 to 1.0) into n evenly



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

Fig. 12. Parameter G-stroke. It allows to assign parametrized stroke textures.

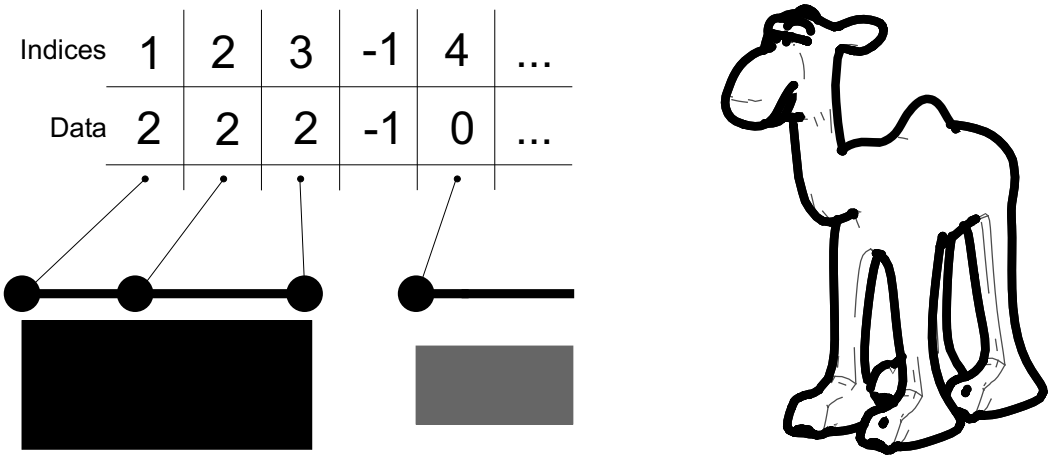
sized subsegments and assigns an `INTEGER` dashing ID between 0 and $n-1$ to them (see Figure 13). This can now be used, for example, to assign different textures to each of the dashing IDs and assemble a unique dasing pattern as shown in Figure 13(b).



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

Fig. 13. Dashing G-stroke. It can be used to create many different repeating patterns.

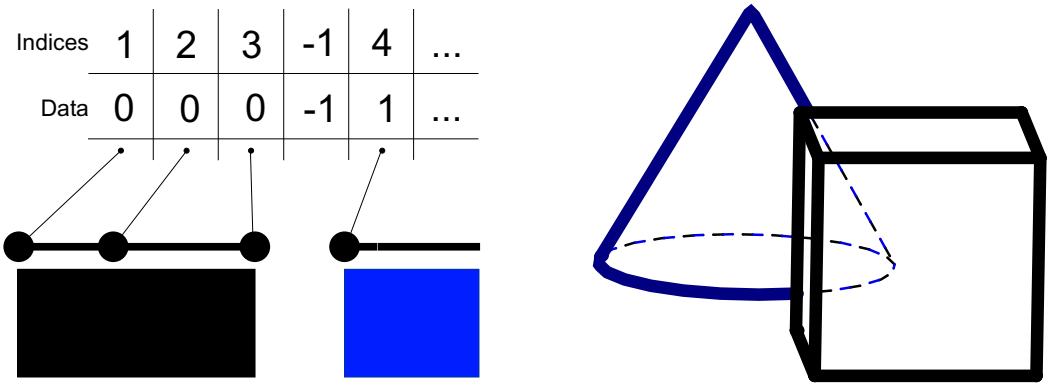
A very important property of the strokes in a line drawing is the type of algorithm used to extract its edges. This property is captured by the *edge type G-stroke* (see Figure 14). It can distinguish, e. g., between silhouettes and the various types of feature edges—each of them denoted by a unique 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 14(b)) since, in traditional renditions, the different edge types are also depicted using different styles. In perspective, it is certainly possible to add more line types such as hatching lines and wireframe lines etc. or types of strokes that were not derived from the 3D model such as a grid for the background.



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

Fig. 14. Edge type G-stroke. It allows to stylize different edge types differently.

It is very important for line renditions used as illustrations to render different objects using different styles. This can easily be achieved using the *object ID G-stroke* (see Figure 15). The object ID property is extracted early when the



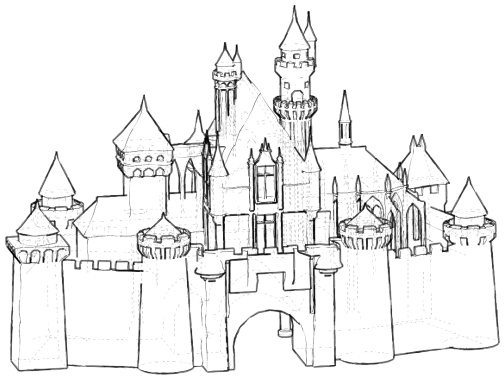
(a) Stroke and G-stroke. (b) Example.

Fig. 15. Object ID G-stroke. Notice that it is used to treat both objects differently: the cone's hidden lines are shown while they are not rendered for the box.

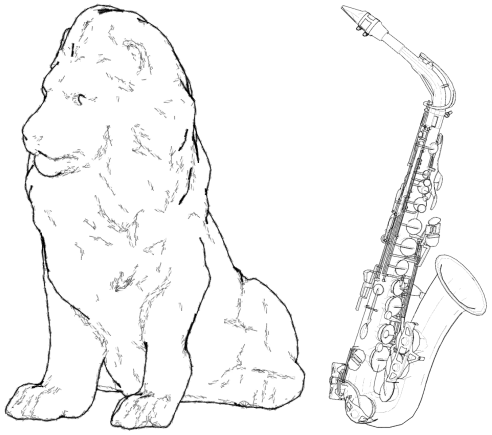
strokes are initially extracted from the 3D model. The values are typically later used to influence the values of other G-strokes (see Figure 15(b)), such as the color, line width, and line saturation G-strokes.

Based on the simple examples presented above, we will now show more complex examples that partially make use of more than one G-stroke at the same time. The first two examples in Figures 16(a) and 16(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

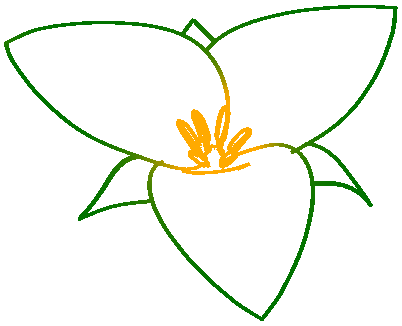
(a) Use of parameter G-stroke.



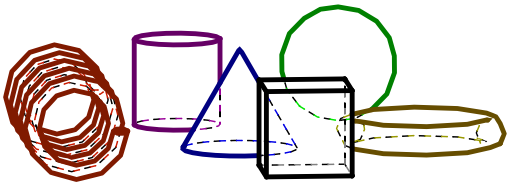
(d) Use of both edge type and visibility G-strokes.



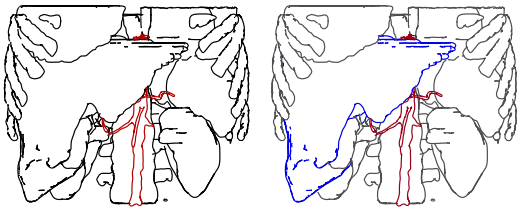
(b) Use of edge type G-stroke.



(e) Modifying the color G-stroke with NPR Lenses.



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



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

Fig. 16. More complex examples for using G-stroke to stylize line renderings.

for different edge types (silhouettes and angle-thresholded feature lines) may be subtle, yet very powerful.

Combining the dashing G-stroke with the object ID and the color G-strokes produces the result shown in Figure 16(c). 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.

The next example in Figure 16(d) 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.

Figure 16(e) demonstrates how G-strokes can be used to stylize line drawings independent of the scene's object structure and how the values of the color G-stroke can be adjusted even within a stroke. Using NPR Lenses [22], a recently developed interaction technique for NPR line stylization based on G-strokes, the flower was colorized in the center with the focus color smoothly blended to the background.

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 render certain organs in very specific colors, such as the arteries in red, as seen in the first image in Figure 16(f). Of course, it is also possible to emphasize different objects using other colors, as in the second image in Figure 16(f).

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 17. 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 (upper part of Figure 17). With the G-stroke approach, this complexity is no longer necessary. Now, we can simply use one 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 17).

6 Summary and Future Work

In this paper, we presented an approach that enables the consistent handling of all stroke properties that may occur in a line rendering system. In contrast to previous direct methods, we propose a global technique that allows coherent and consistent storage and management of the attributes. 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

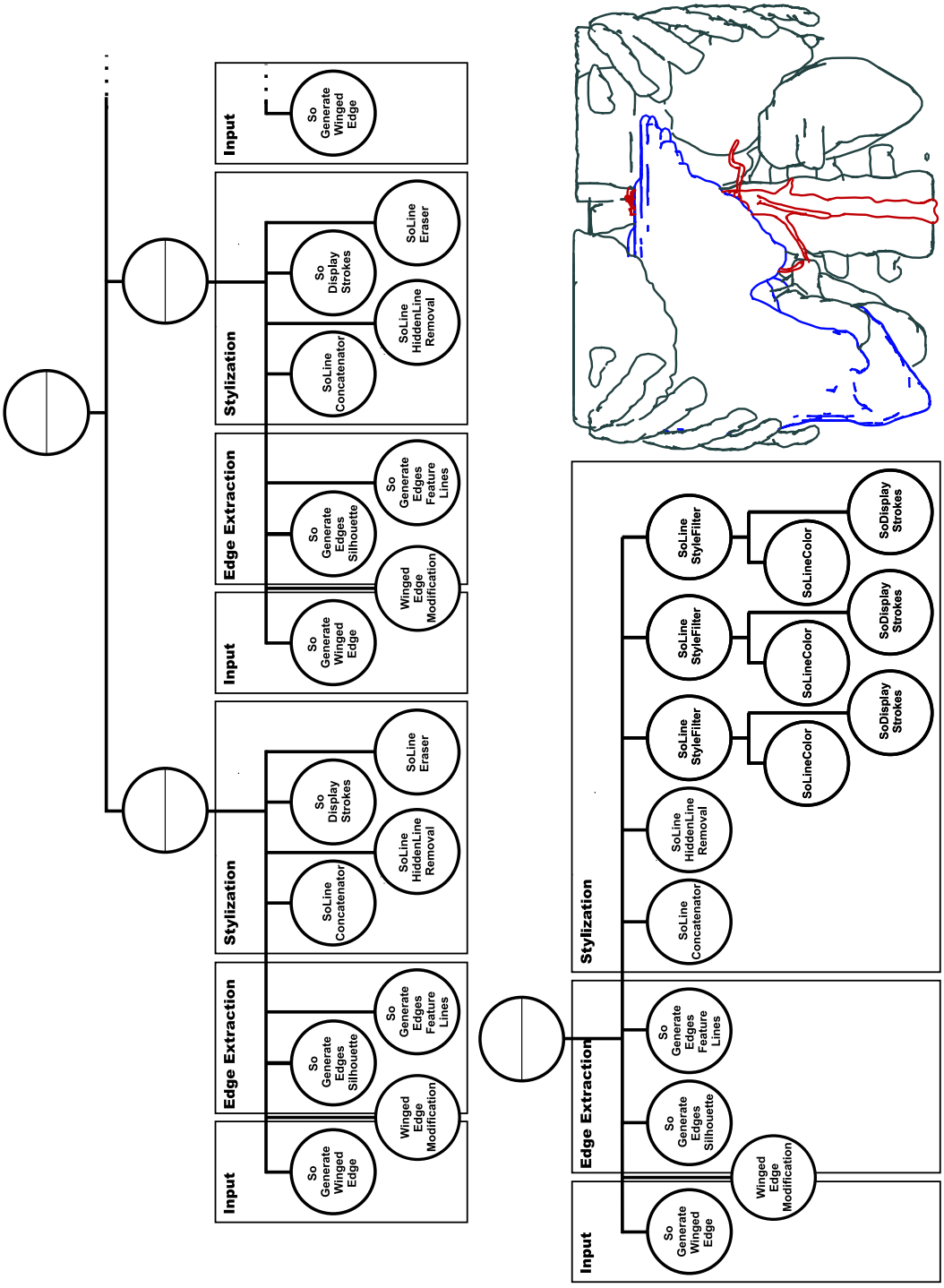


Fig. 17. Comparison of a scene graph that uses the previous way for stylization (above) with one that uses G-strokes (below) for rendering objects in different colors.

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 leading to simpler and faster

computation.

The presented approach of computing the necessary changes to all G-strokes and the stroke itself "on the fly" naturally raises two questions: (1) is this technique necessary to achieve the desired effects and (2) how big is the performance impact. The answer to the first question—that it is indeed an effective approach—is given by the previously mentioned two-way dependency between properties and stroke geometry and topology. Changes to the stroke lead to changes in the properties as well as vice versa. Thus, the computation "on the fly" is essential in order to keep the specification of line stylizations as simple as possible. With respect to the performance impact, we have not noticed any measurable slowdown of the hierarchical stylization "pipeline" when compared to the previously used linear one. The system allows interactive to real-time line stylization depending on the size of the used model with the line extraction being the bottleneck.

Future work includes the extension of the G-strokes concept in terms of creating new specific G-strokes. For example, several notions of curvature such as in [23] could each be tracked 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 can be implemented that make use of the G-stroke data for stroke modification and stroke stylization. For example, overshooting could be implemented as in [7], the curvature G-strokes could be used to influence the line thickness as in [23], 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. [24] could serve as a starting point for this task.

Although the G-strokes concept lends itself very easily to local modifications of the strokes, it can also be used to do more global modifications. For example, it could be used to modify stroke density in the resulting drawings depending on the region of the model from which the strokes originated. For that purpose, a pipeline node would use the 3D coordinates of the stroke data that it maintains in the pipeline and derive the density based on a second data source to decide whether a specific stroke or portion of it would be drawn.

Another direction for future work needs to address the issue of temporal correlations such as, e. g., in [11]. This would require tracking the parametrization of strokes as in their system which could be handled by an additional node that maintains a separate data structure for this purpose. 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 be

considered to be G-properties of the 3D surface model. The development of a G-strokes system for surface textures of 3D models will be challenging future work.

Acknowledgments

We would like to gratefully acknowledge the support of our funding providers, Alberta Ingenuity (AI) and the State of Saxony-Anhalt. In addition, we would like to thank Christian Tietjen for his insight in Open Inventor peculiarities, Mario Costa Sousa and Stefan Schlechtweg for their valuable comments on the paper, and Mark Hancock for proofreading.

References

[1] B. Gooch, A. Gooch, Non-Photorealistic Rendering, A K Peters, Ltd., Natick, 2001.

[2] T. Strothotte, S. Schlechtweg, Non-Photorealistic Computer Graphics. Modelling, Animation, and Rendering, Morgan Kaufmann Publishers, San Francisco, 2002.

[3] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, T. Strothotte, A Developer's Guide to Silhouette Algorithms for Polygonal Models, IEEE Computer Graphics and Applications 23 (4) (2003) 28–37.

[4] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, A. Santella, Suggestive Contours for Conveying Shape, ACM Transactions on Graphics 22 (3) (2003) 848–855.

[5] J. D. Northrup, L. Markosian, Artistic Silhouettes: A Hybrid Approach, in: J.-D. Fekete, D. Salesin (Eds.), Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000, Annecy, France, June 5–7 2000), ACM Press, New York, 2000, pp. 31–37.

[6] N. Halper, T. Isenberg, F. Ritter, B. Freudenberg, O. Meruvia, S. Schlechtweg, T. Strothotte, OpenNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering, in: J. Rokne, R. Klein, W. Wang (Eds.), Proceedings of the 11$^{\text{th}}$ Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2003, Canmore, Alberta, Canada, October 8–10, 2003), IEEE Computer Society, Los Alamitos, CA, USA, 2003, pp. 424–428.

[7] S. Grabli, E. Turquin, F. Durand, F. X. Sillion, Programmable Style for NPR Line Drawing, in: A. Keller, H. W. Jensen (Eds.), Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering 2004 (Norrköping,

Sweden, June 21–23, 2004), EUROGRAPHICS Association, 2004, pp. 33–44, 407.

[8] T. Saito, T. Takahashi, Comprehensible Rendering of 3-D Shapes, ACM SIGGRAPH Computer Graphics 24 (3) (1990) 197–206.

[9] D. L. Dooley, M. F. Cohen, Automatic Illustration of 3D Geometric Models: Lines, in: R. Riesenfeld, C. Séquin, M. Zyda (Eds.), Proceedings of 1990 Symposium on Interactive 3D Graphics (Snowbird, UT, USA, March 25–28, 1990), ACM Press, New York, 1990, pp. 77–82.

[10] G. Elber, Line Illustrations ∈ Computer Graphics, The Visual Computer 11 (6) (1995) 290–296.

[11] R. D. Kalnins, P. L. Davidson, L. Markosian, A. Finkelstein, Coherent Stylized Silhouettes, ACM Transactions on Graphics 22 (3) (2003) 856–861.

[12] M. C. Sousa, P. Prusinkiewicz, A Few Good Lines: Suggestive Drawing of 3D Models, Computer Graphics Forum 22 (3) (2003) 381–390.

[13] S. Schlechtweg, B. Schönwälder, L. Schumann, T. Strothotte, Surfaces to Lines: Rendering Rich Line Drawings, in: V. Skala (Ed.), Proceedings of the 6^{th} International Conference in Central Europe on Computer Graphics and Visualization (WSCG 1998, Plzen, Czech Republic, February 9–13, 1998), Vol. 2, 1998, pp. 354–361.

[14] A. Appel, The Notion of Quantitative Invisibility and the Machine Rendering of Solids, in: S. Rosenthal (Ed.), Proceedings of the 22^{nd} ACM National Conference 1967 (Washington, D.C., USA), ACM Press, New York, 1967, pp. 387–393.

[15] A. Hertzmann, Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines, in: S. Green (Ed.), ACM SIGGRAPH 1999 Course Notes, ACM SIGGRAPH, ACM Press, New York, 1999, course 17: Non-Photorealistic Rendering.

[16] F. Benichou, G. Elber, Output Sensitive Extraction of Silhouettes from Polygonal Geometry, in: B. Werner (Ed.), Proceedings of the 7^{th} Pacific Conference on Computer Graphics and Applications (Pacific Graphics 1999, Seoul, Korea, October 5–7, 1999), IEEE Computer Society Press, Los Alamitos, CA, USA, 1999, pp. 60–69.

[17] J. W. Buchanan, M. C. Sousa, The Edge Buffer: A Data Structure for Easy Silhouette Rendering, in: J.-D. Fekete, D. Salesin (Eds.), Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000, Annecy, France, June 5–7 2000), ACM Press, New York, 2000, pp. 39–42.

[18] A. Hertzmann, D. Zorin, Illustrating Smooth Surfaces, in: J. R. Brown, K. Akeley (Eds.), Proceedings of ACM SIGGRAPH 2000 (New Orleans, LA, July 23–28, 2000), ACM Press, New York, 2000, pp. 517–526.

[19] M. McGuire, J. F. Hughes, Hardware-Determined Feature Edges, in: A. Hertzmann, C. Kaplan (Eds.), Proceedings of the Third International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2004, Annecy, France, June 7–9, 2004), ACM Press, New York, 2004, pp. 35–44.

[20] B. Gooch, P.-P. J. Sloan, A. A. Gooch, P. Shirley, R. Riesenfeld, Interactive Technical Illustration, in: J. Rossignac, J. Hodgins, J. D. Foley (Eds.), Proceedings of the 1999 Symposium on Interactive 3D Graphics (Atlanta, GA, USA, April 26–29, 1999), ACM Press, New York, 1999, pp. 31–38.

[21] T. Isenberg, N. Halper, T. Strothotte, Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes, Computer Graphics Forum 21 (3) (2002) 249–258.

[22] P. Neumann, T. Isenberg, S. Carpendale, NPR Lenses: Local Effect Control for Non-Photorealistic Line Drawings, in: D. DeCarlo, L. Markosian (Eds.), Poster Presentations of the Fourth International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2006, Annecy, France, June 5–7, 2006), 2006.

[23] M. C. Sousa, K. Foster, B. Wyvill, F. Samavati, Precise Ink Drawing of 3D Models, Computer Graphics Forum 22 (3) (2003) 369–379.

[24] N. Halper, S. Schlechtweg, T. Strothotte, Creating Non-Photorealistic Images the Designer's Way, in: A. Finkelstein (Ed.), Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2002, Annecy, France, June 3–5, 2002), ACM Press, New York, 2002, pp. 97–104.