WYSIWYG NPR: Interactive Stylization for Stroke-Based Rendering of 3D Animation

Robert D. Kalnins

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

June 2004

Abstract

Artists and illustrators have developed a large repertoire of techniques to communicate information effectively using traditional media. Recent work in computer graphics has begun to leverage these techniques in the form of non-photorealistic rendering (NPR) algorithms for 3D, but, to date, little research has addressed flexible, interactive tools to make such algorithms accessible to content creators. In this dissertation we demonstrate the importance of developing these tools for NPR. In particular, we show that "hands-on" NPR systems can provide the designer with new levels of æsthetic flexibility, and the means to achieve effects tedious or even impossible to attain by traditional methods.

We approach this open problem for stroke-based NPR of animated 3D geometry. Our system employs a tablet interface to provide the designer with an interactive paradigm in which stroke effects are sketched directly into the scene. The artist imparts his unique æsthetic by sketching strokes over the outlines of objects, and drawing details and hatching effects onto their surfaces.

These tools raise new questions regarding how annotations drawn by the designer in one view should appear when the scene is rendered from novel viewpoints. It is unreasonable to expect the user to annotate geometry from all conceivable poses. Rather, we develop rendering algorithms that can exhibit a range of behaviors which the designer can tune to specific goals, either by adjusting parameters or directly sketching the desired effects. But it is not sufficient for the stylization rendered in each frame to maintain consistency only with the designer's annotations. To be suitable for animation, they must also exhibit coherence between consecutive frames. We consider this for all effects in our system, but this is particularly challenging for those stylizing view-dependent silhouettes. For these, we develop a flexible and robust new framework that can provide coherence tailored to a particular æsthetic. These tools make possible a new class of effects that would be daunting to animate by hand.

This research shares applications with those of traditional hand-generated techniques, including architectural and product design, technical and medical illustration, storytelling (e.g. children's books), games, fine arts, and animation.

Acknowledgments

This research would not have been possible without the generous support and guidance of my advisor, Adam Finkelstein. I would also like to extend special thanks to Lee Markosian for the multitude of insightful discussions that assisted, in no small part, to conceive and shape this project. Many individuals deserve recognition for their contributions to this work, including (but not limited to) Philip L. Davidson, Matthew Webb, Michael A. Kowalski, Barbara J. Meier, Joe Lee, John F. Hughes, Doug DeCarlo, Szymon Rusinkiewicz, Grady Klein, Matthew Hibbs, Salman Butt, Dan Crosta, and David Bourginon. And kudos to Melissa Lawson for always going the extra mile to ensure that her students' affairs are in order. As any international student will affirm, one is truly lost without such an excellent graduate coordinator.

I would like to extend deepest appreciation to my family for the limitless support they have provided me over the course of my endeavors. Many thanks also to my friends (both at home and in Princeton), without whom my sanity would most certainly have been unseated somewhere along the way. And finally, to Melanie, I cannot fully express all my gratitude for your support, encouragement, patience, and inspiration that made this journey possible.

Funding for this work was provided in part by Intel Labs, the Alfred P. Sloan Foundation, and the National Science Foundation (CCR-9875562). Portions of this dissertation have also been published as [46] and [47].

For Melanie...

Contents

List of Figures

Chapter 1

Introduction

1.1 Why NPR tools?

Research in 3D computer graphics has traditionally focused on systems that produce photorealistic imagery. As a result, the adoption of these technologies by new users has been limited to a small number of specific applications. One key reason is that photorealistic content is extremely difficult and expensive to create. Such rendering carries the burden of exhaustive simulation of the physical processes by which light propagates through the scene, interacts with surfaces, and ultimately arrives at the camera [32]. Even if some "black box" were available to easily perform this task, Markosian [55] argues that the cost of modeling the underlying geometry with sufficient detail is still prohibitive. While methods have been developed to scan existing 3D objects [15], and capture their motion [62, 70], the results ultimately offer minimal creative control, and are certainly of little use if the target result exists only in the designer's imagination.

On the other hand, artists and illustrators working with traditional media have developed a large array of techniques to convey visual information both efficiently and effectively via non-photorealistic means. In fact, such approaches are often preferable to photorealism when the primary purpose is to communicate, explain, inform or entertain [55]. With the right tools, systems that produce stylized or abstract rendering should be significantly easier for the content creator to employ – requiring less underlying geometric detail, and relaxing the rendering demands of physical realism. Accordingly, a growing branch of computer graphics research has focused on producing non-photorealistic rendering (NPR) of 3D geometry by adapting these traditional methods. Strong arguments for the usefulness of this line of inquiry are made by many researchers (e.g., [23, 52, 61, 84, 92]).

To date, much of the research in NPR has targeted a particular style of imagery and developed algorithms to produce it automatically when rendering 3D scenes. Relatively little emphasis has been placed on the related problem of how to provide direct, flexible interfaces that enable a designer to describe the stylistic appearance of such renderings in the first place, much less specify how they behave dynamically. Instead, the usual approach is to rely on pre-scripted behavior, or cumbersome programmatic descriptions. Meier [60] and Seims [78] argue that effective interfaces are essential for these technologies to be accepted by content creators. Indeed, a key reason is that NPR imagery often reflects a designer's personal judgment regarding what details to emphasize or omit. Thus, the challenge facing NPR researchers is to develop systems that present a designer with interfaces offering such direct control over rendering æsthetics.

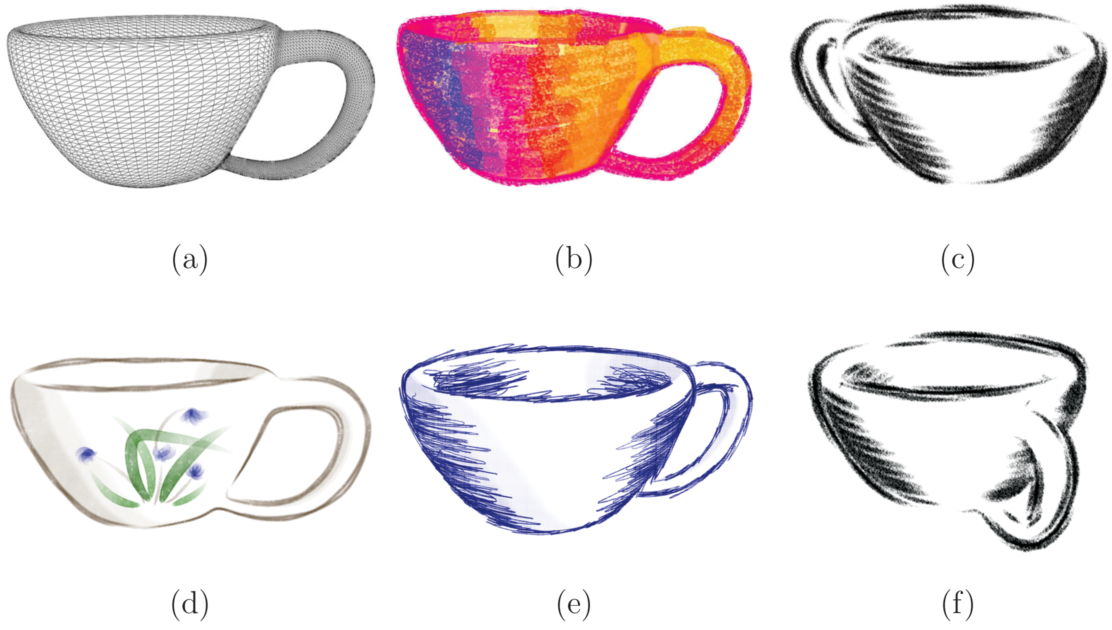
Furthermore, while the last decade has seen a blossoming of work on NPR algorithms in a variety of styles (see [30] for a survey), much of this work addresses the production of still images. However, a particular strength of NPR is the possibility to relieve the burden of painstakingly creating many frames by hand when constructing animations. Some systems for rendering 3D scenes have explicitly addressed the challenge of NPR rendering for animation (e.g. [21, 61]). But to date, the results lack significant æsthetic flexibility or practical accessibility to the artist.

Additional challenges arise when rendering NPR the real-time interactive domain. Maintaining frame-to-frame coherence of sequential renderings while under real-time computational budgets, especially when the camera paths are unknown *a priori*, are all issues that must be addressed in interactive NPR systems. These challenges are common to such applications as visualization and design tools, video games, interactive story books, etc. Some work in the literature has begun to address aspects of rendering in a real-time settings (e.g. [31, 50, 51, 57, 68]), but the development of interactive NPR design tools remains an open problem.

1.2 Thesis

In this work we seek to demonstrate the importance of developing flexible, interactive design tools for animated NPR. Our goal – or thesis – is to show that these "hands-on" systems can provide the designer with new levels of æsthetic flexibility, and the means to achieve effects challenging or impossible to attain by traditional methods.

The development of these systems will entail a range of challenges spanning from the front-end interfaces upon which the design tools are built, to the back-end algorithms that produce the final imagery. We are specifically interested in "hands-on" tools – those providing the artist with particularly direct, interactive control over æsthetic qualities. Often, such tools employ interfaces that function in a manner consistent with the type of stylization implemented by the system. For example, painting with a haptic brush device is well-suited for defining the stylizations in a paint-based NPR system [2]. With these interfaces and appropriately designed tools, a user should be able to apply any existing skills with traditional media (e.g., indication [92]) and personal judgment to impart his unique æsthetic to the rendering. Such processes are typically most challenging to emulate in the black-box engines common in the literature (e.g. [16]).


(a) (b) (c)

(d) (e) (f)

Figure 1.1: By drawing stroke-based NPR effects directly into the 3D scene, the artist annotates the teacup in (a) to produce various distinct styles. The stylization in (c) is preserved coherently by the system as the camera changes to (f).

However, these systems would be of little interest if they sought only to emulate traditional media. Their appeal derives from when the machine is exploited to accomplish the tedious aspects of content creation. For instance, if a designer draws stylization into a 3D scene, a useful system might render the stylized geometry from new camera poses automatically. This raises the question of how some stylization created in one view should adapt when seen from another. Solutions will need to investigate both the designer's controls to describe these adaptations, and the rendering algorithms that implement them. Furthermore, if a system can be made to maintain frame-to-frame coherence between novel views, it becomes possible to produce animations automatically. Effective solutions can open the door to effects so rich they would be virtually impossible to produce coherently by hand.

In short, effective NPR systems are those that provide hands-on tools that enable artists to do what they are good at, and handy tools that offload tedious tasks the CPU is good at. We explain in Chapter 2 why we use the term *WYSIWYG NPR* for systems composed of such tools. However, as we outline in the previous section, little work has yet been done to produce them. We address this open problem with the development of *Jot* – a proof-of-concept WYSIWYG NPR system – with which we advance our thesis. Using Jot, an artist can annotate 3D geometry like the mesh in Figure 1.1a by drawing stroke-based effects directly into the scene. Hands-on tools enable the designer to achieve a wide spectrum of æsthetics, a few examples of which are given in Figures 1.1b-e. Additional controls provide means to describe how stylization behaves under the dynamic conditions that arise rendering novel views, as in Figure 1.1f. The system maintains the frame-to-frame coherence of these stylizations during animation or real-time interaction, regardless of their visual complexity. It is our aim with Jot to demonstrate that, even with a small set of stroke-based effects, WYSIWYG NPR tools can offer designers a new degree of æsthetic flexibility, and the means to produce dynamic imagery unattainable via traditional media.

1.3 Organization

We begin in Chapter 2 by fleshing out the design challenges presented by WYSIWYG NPR tools, and then introduce the Jot test system used to investigate them in this work. The chapter concludes by mapping out the design challenges posed by our thesis in the context of the Jot platform.

In our test system, we chose to investigate the range of traditional media styles available to stroke-based NPR. Jot implements several of these effects using a stroke primitive, described in Chapter 3, that is designed to provide æsthetic flexibility, while maintaining interactive performance rates.

The stroke-based effects we have chosen to address in this system fall into two fundamental categories. The first of these are the line-based effects described in Chapter 4. These involve the stylization of curves arising from the 3D geometry, such as the static feature lines described in Section 4.2, or the view-dependent contours explored in Section 4.3. Figure 1.1 provides several examples of silhouettes stylized with these tools. The particularly challenging problem of maintaining the coherence of these stylizations is reserved for Section 4.4.

The other class of annotations are the surface-based effects explored in Chapter 5. We consider different types of surface markings delineated by the semantic information they convey. First among them are the detail marks like the flower in Figure 1.1d, which we discuss in Section 5.2. Strokes that impart surface tonal effects, such as hatching on the cup in Figure 1.1c, comprise the final variety in Section 5.3.

We conclude discussions in Chapter 6 with a summary of our findings and recommendations for avenues of future research.

Chapter 2

WYSIWYG NPR

2.1 Overview

The term "What You See Is What You Get," or WYSIWYG, originated at Xerox PARC in the late 1970's to describe their text editor Bravo [91]. The first of its kind, Bravo was designed to present the document being edited in a format identical to the final printed page. In more contemporary times, WYSIWYG has come to describe any interface that allows manipulation of output in its final form, without requiring any knowledge of the underlying machinery. WYSIWYG webpage editors, for instance, enable authors to directly layout their pages while the system automatically generates the actual HTML code. Stated most broadly, WYSIWYG describes content creation systems that enable the user to arrange the final design elements in a hands-on environment while providing immediate visual feedback. This paradigm describes the type of NPR design tools we propose in Section 1.2, thus we use the term *WYSIWYG NPR* to describe systems that adhere to these design philosophies.

In this work, we develop a WYSIWYG NPR test-bed system called *Jot*. But this is not the first example of WYSIWYG in the context of computer graphics design tools. The WYSIWYG technique for creating texture maps was proposed by

Hanrahan and Haeberli [34] and is now available in various commercial modeling systems. These tools let the designer paint texture maps directly onto 3D models in real-time by projecting screen-space paint strokes onto the 3D surface and into texture space where they are composited on the final map. Of course, the strokes in these conventional texture maps simply remain fixed on the surface. They do not adapt to dynamic conditions, such as changes in lighting model or viewpoint. In contrast, the strokes in our NPR system can be tailored by the designer to respond stylistically in dynamic 3D environments.

Our system also draws inspiration from others that provide direct drawing interfaces for creating stylized scenes. Arguably, the "Holy Grail" of NPR research is to enable the artist to simply draw in the image plane and thereby express a complete, stylized 3D world. The systems of Tolba et al. [86], Cohen et al. [12], and Bourguignon et al. [5], each pursue this goal by making simplifying assumptions about the structure of the underlying geometry. To avoid making any such constraining assumptions in our work, we assume that the underlying 3D geometry has already been supplied in some way. NPR stylization effects will then be created by drawing directly into the scene, and onto this geometry. Our goal is to eventually integrate WYSIWYG NPR tools directly into a modeling system that also uses a drawing interface for constructing the geometry, like SKETCH [95] or Teddy [42].

In pursuit of the goal of bringing WYSIWYG tools to bear on NPR, we cannot hope to address all possible styles and effects. Instead, we focus upon NPR of stroke-based styles, as they encompass a wide range of traditional media. Furthermore, we concentrate our efforts on two fundamental effects: (1) stylization of individual lines and contours that form the basis of simple line drawings, and (2) stylized patterns and hatching that depict surface detail or tonal properties. While by no means complete, we will show that this set of effects offer significant flexibility when implemented with WYSIWYG NPR tools.

In the next section, we describe how the Jot system is organized with an example from the user's perspective. This is followed by a summary of the two primary modes of the system – Section 2.3 briefly describes Jot's *parametric controls* (e.g., sliders and buttons), while Section 2.4 introduces the more hands-on *direct controls* that form the focus of this WYSIWYG NPR system. Finally, we map out the challenges that we must address to develop them in Section 2.5.

2.2 Example Session

The Jot system has three primary editing modes. The first mode employs conventional interface elements (e.g. buttons, sliders) to offer various *parametric controls* as described in Section 2.3. With these tools the user adjusts the more mundane system settings like paper textures, background images, brush styles, lighting models, etc.

In contrast, the two remaining modes of the system leverage a tablet interface to provide *direct controls* for stroke-based effects. When the artist sketches with the pen and tablet interface, the gestural input defines stroke paths, while the pressure information modulates stroke attributes like width or opacity. In the first of these modes, *line stylization* (Chapter 4), the artist can stylize features like the crease and silhouette contours typical to line drawings, while in *surface stylization* mode (Chapter 5) strokes are sketched to create surface detail and shading effects.

As an illustration of how these tools work in concert, we summarize the steps followed by a designer to create a stylized depiction of an apple sauce tin from a cylindrical mesh (Figure 2.1):

a) A connected, polygonal mesh of a can was modeled using 3rd party software.

b) The stylization process begins when the artist imports this mesh into Jot. It is initially displayed over top of the default background, in a generic outlining style, and without any surface shaders.

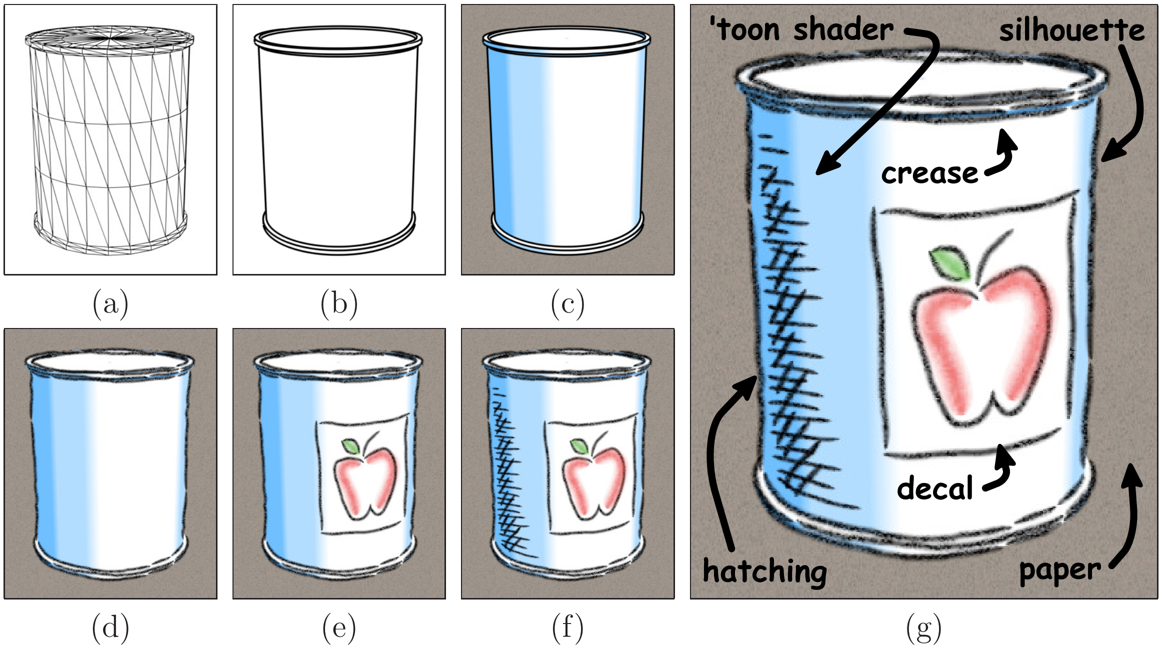


Figure 2.1: Jot: (a) A mesh is, (b) loaded into Jot, (c) paper and shaders are chosen, (d) strokes are drawn over outlines, (e) a label is sketched with decal strokes, and (f) shading depicted with hatching strokes. These elements are labeled in (g).

c) First the artist engages *parametric control* mode, causing various menus, buttons, etc. to be presented. He selects a pre-generated paper texture and applies it to a gray background color. Next, he picks a ‘toon shader for the model [51], chooses its 1D texture from a set of profiles, and adjusts it to a blue color. And finally, the lighting controls are manipulated so that the shader darkens the left of the can.

d) Now the designer engages *line stylization* mode and picks a black pencil style from amongst the presets. Using the tablet, he selects the silhouette contour with a tap gesture, and draws a sketchy line prototype to stylize it. A second tap gesture propagates the stylization to the remaining silhouette on the other side of the can. To stylize the remaining feature lines, he selects the creases along top and bottom lips and over-sketches them with a stylization similar to the silhouettes.

e) Next *surface stylization* mode is engaged by the artist, and he indicates that his coming gestures will depict detail. The apple label is then created by strokes drawn directly onto the surface. During this process, he moves the viewpoint as necessary, and changes the color, width and other brush parameters as shown.

f) Finally, the user indicates that the remaining gestures imply tone. Drawing roughly parallel lines he produces hatching on the side of the can. A tap gesture completes the stroke group, and is followed by another at angle to the first. A final tap gesture completes the cross-hatching. The system infers a camera-space lighting model that keeps the hatching on the left of the tin in new views.

2.3 Parametric Controls

The goal of this project is to bring the WYSIWYG paradigm to NPR design tools. The hallmark of these systems are the hands-on *direct control* modes we outline in the next section. These interfaces and supporting systems will compose the bulk of our discussions (Chapters 4 and 5). Nevertheless, the more traditional *parametric controls* common in existing off-line or black-box style NPR systems were found necessary to bring the Jot tool set up to minimally functional levels. For the sake of completeness, we take a moment to sketch out options available to the designer in this mode.

Environment

Various controls govern an array of environmental settings. These include such static elements such as the background colors and images which the scene objects will be composited over. Also, as part of a media simulation effect described in Section 3.3, a paper texture map and associated tunable parameters can be assigned to the window plane. Other controls are associated with the lighting environment, which can influence the rendering of some surface shaders described in the next section.

Basecoats

The designer can assign to each patch of surface a *basecoat* – a composite of shaders that draw the patch triangles. Shaders include flat coloring, conventional Gouraud shading and texture mapping, and 'toon shading (Figure 2.1).

Appropriate interfaces exist to configure each of these shaders, such as color sliders, texture menus, and lighting controls. For 'toon shaders, as described by Lake et al. [51], the designer can also create custom 1D texture maps describing the color spectrum. Our 1D textures may also include an alpha component that is input for our media simulation (Figure 3.4).

Media Simulation

To increase the range of NPR styles, a media simulation effect (Section 3.3) emulates the deposition of pigment by shaders and strokes onto an underlying paper medium. This is governed by a height-field texture that is assigned to the window as part of the environmental settings described above. Additional controls influence how paper textures are assigned on a per basecoat, and per stroke effect basis, providing more stylistic flexibility. Furthermore, these additional height fields need not remain fixed in the view plane. To combat shower-door effects [61], these textures can be "pinned" to geometry so that they move in screen-space (Section 3.3).

Brush Styles

When sketching stroke effects in the direct stylization modes, the designer can adjust the current "brush style" that affects stroke properties. In practice, the artist will carefully design combinations of parameters that together produce a desired look, then save that brush into a collection of presets available in future sessions. Section 3.1 briefly describes the parameters that govern the rendering of the stroke primitives.

2.4 Direct Controls

At this point, we turn our attention to Jot's *direct controls*. These comprise the tablet-driven WYSIWYG NPR tools and supporting systems that are the primary focus of this investigation. As revealed in the example session (Section 2.2), these controls are separated into a mode for manipulating *line stylization effects* (Chapter 4), and another for *surface stylization effects* (Chapter 5). Before we delve into the their implementation details, we first summarize these effects to introduce concepts and terminology useful when mapping out our approach to their development in the next section.

2.4.1 Line Stylization Effects

Certain types of lines, such as silhouette contours, play a critical role in our visual interpretation of 3D shape. Artists and designers therefore often emphasize them, either by drawing them outright or by contrast enhancement across line boundaries. In fact, the most minimal hand-drawn image of a given 3D scene will often consist of little more than markings that depict these lines.

In this mode of the system, the goal is to provide tools enabling the artist to directly describe the stylized appearance of these lines. The user will select one of these curves in the 3D scene, and then specify stylization by directly sketching strokes over it with a chosen brush style. The system's task is then to maintain the character imparted by the artist automatically under changing views, animation, or other dynamic influences.

We sort our line stylization effects into two categories based upon whether they remain static on the geometry, or arise dynamically as a function of the view. The latter will present additional challenges because of their dynamic nature.

Static Feature Lines remain fixed on the 3D surface. These arise on the surface to represent sharp creases, borders between different regions, and other salient features. Some examples of these are rim on the sauce tin in Figure 2.1 and all the lines depicted in Figure 4.1 (page 32).

View Dependent Contours arise dynamically as a function of surface orientation in the camera frame. The most familiar variety of these are *silhouette contours* that separate front and back facing regions of surface, but our system also supports the *suggestive contours* of Decarlo et al. [19]. All the outlines composing the images in Figure 1.1 are examples of stylized silhouette contours.

2.4.2 Surface Stylization Effects

Another critical element of traditional illustration are the groups of strokes designers use to elicit surface properties. With the appropriate marks, they can convey surface shape through shading, reveal material properties, and impart other forms of detail.

We support some of these effects in this mode of the system. Here, surface stylization is generated by sketching groups of new strokes directly onto 3D surfaces. This differs markedly from line stylization effects, which involve over-sketching to stylize existing lines. Once defined, however, the system again manages these stylizations automatically under changing views, animation, or other dynamic influences.

We examine two types of surface effects, categorized by the type information they are intended to convey. This distinction will govern their dynamic behavior.

Detail Marks are groups of strokes used to depict surface details. Examples include decals, such as the label on the tin in Figure 2.1, and suggestions of material properties, like the hairs on the bust in Figure 5.1c (page 74).

Tonal Marks are strokes used to depict tone or lighting effects. The conventional cross-hatching on the side of the tin in Figure 2.1 is an example this effect. A more free-form variety of tonal marks indicate shading on objects in Figure 5.7 (page 87).

			Stroke-Based NPR Effects			
			Line Stylization Effects (i.e. sketching stylizes existing stroke paths)		Surface Stylization Effects (i.e. sketching creates new stroke paths)	
			Static Feature Lines (e.g. sharp creases)	View-Dependent Contours (e.g. silhouettes)	Detail Marks (e.g. decals)	Tonal Marks (e.g. hatching)
WYSIWYG Design Elements	Annotation	Stylization Interfaces	· over-sketch, §4.2.1	· prototype sketch, §4.3.2	· direct sketching, §5.2.1	· structured sketch, §5.3.1 · free-form sketch, §5.3.2
		Stylization Synthesis	· rubber-stamp, §4.2.2 · markov synth, §4.2.2	· rubber-stamp, §4.3.2 · proto. propagation, §4.4.6	· *pattern synthesis*, §5.4	· *hatching synthesis*, §5.4
	Dynamics	Scripted Behavior	· level of detail, §4.2.3 · parameterization, §4.4	· level of detail, §4.3.3 · parameterization, §4.4	· dynamic strokes, §5.2.2	· structured hatching, §5.3.1 · lighting, §5.3.3
		Customizable Behavior	· *level of detail*, §4.5	· anim. stylization, §4.3.5 · *localized stylization*, §4.5	· level of detail, §5.2.3	· free-form hatching, §5.3.2
	Interactivity	Temporal Coherence	· intrinsic, §4.2.5 · coherence pipeline, §4.4	· simple approach, §4.3.4 · coherence pipeline, §4.4	· mostly intrinsic, §5.2.4	· mostly intrinsic, §5.3.4
		Real-Time Performance	· visibility, §4.2.5	· fast extraction, §4.3.1	· visibility, §5.2.4	· uv issues, §5.2.4

Table 2.1: Mapping out the problem space: WYSIWYG Design Elements vs. Stroke-Based NPR Effects. Section references point to discussions of each WYSIWYG design consideration in the context of tool development for a given NPR effect. Items in *italics* typeface have not been investigated, but discussed as possible future avenues of research.

2.5 Design Considerations

The development of Jot's direct controls will entail a range of challenges spanning from their front-end interfaces, to their back-end rendering algorithms. We identify these challenges in this section by examining key design elements fundamental to the WYSIWYG NPR tools proposed in Chapter 1. These elements will organize our approach to developing such tools for the stroke-based effects in our system.

We can map out the resulting problem space as shown in Table 2.4.2. Each cell represents one of the key design issues described in this section, taken in the context of tool development for one of our stroke-based effects. If that concern has been addressed in the implementation of Jot, pertinent section references are given. Issues that are only discussed in terms of their merit for future research are referenced in *italics* typeface. This table forms the road map for the remaining chapters.

To identify these design issues, recall the summary from Section 1.2 – i.e., effective WYSIWYG NPR systems are those that provide hands-on tools enabling artists to do what they are good at, and handy tools for offloading tedious tasks the CPU is good at. We apply these principles to three fundamental aspects of the system: (1) *annotation* interfaces and algorithms for describing the stylization of a scene, (2) interfaces and algorithms that describe how animation or camera *dynamics* influence the stylization, and (3) constraints imposed by the *interactivity* of the system. We examine each of these aspects of the system in turn, identifying the important design considerations to address.

2.5.1 Annotation

Foremost, we seek hands-on tools enabling the user to annotate 3D scenes with their desired æsthetic. This entails not only flexible and direct *stylization interfaces*, but also algorithms that can automatically *synthesize stylization* from the original input.

With these tools the artist can impart the desired appearance to complex scenes, without performing an exhaustive annotation of all constituent elements.

Stylization Interfaces

Appropriately designed tools should offer the artist a high degree of direct control, even providing opportunity to apply existing skills with traditional media. One aspect of this is choosing input devices that are matched to the task at hand. For instance, a pen and tablet interface is well-suited to perform stylization tasks in a stroke-based NPR system. Unlike a conventional mouse, the tablet provides high precision for small strokes, additional pressure and tilt information, and a degree of similarity to traditional tools.

Of course, the input device is just one aspect of interface design. Systems must be built that employ them in an effective fashion. Hatching stylization, for example, could be controlled by literally sketching it onto 3D surfaces. WYSIWYG NPR systems should aspire to these direct style controls where possible, and avoid encumbering the designer unnecessarily with parametric controls such as sliders, buttons, and the like.

Stylization Synthesis

The stylization interfaces described above imply a localized form of annotation, such as sketching a label onto a surface (Figure 2.1d). However, it can be tedious to manually stylize all components of complex scenes in this way. A useful system should also provide annotation tools that do not unreasonably burden the user, especially when repetition can be exploited by the CPU. Stylization synthesis tools could replicate the designer's annotations, or even generate unique stylization similar in some way to the original input. By sketching only a few bricks, for instance, the designer might stylize an entire brick wall.

2.5.2 Dynamics

Arguably, the most attractive aspect of 3D NPR is the ability to automatically render annotated geometry from novel viewpoints. However, this raises new questions concerning how annotations made in one view should appear when rendered in another. To facilitate this, stroke rendering algorithms should produce effects that respond to such influences as changing level-of-detail under camera motion. This is something that can occur automatically for common or predictable effects via *scripted behavior*. But in WYSIWYG NPR systems, interfaces should also exist to enable a designer to ascribe a desired *customizable behavior* for these dynamic responses.

Scripted Behavior

Certain dynamic effects lend themselves to familiar “pre-cooked” or scripted behavior. As a camera zooms out from a scene, for instance, it is natural to shrink the widths of the strokes that depict it. For some stroke-based effects, a given dynamic influence may lead to a number of common behaviors. A flexible system could provide a spectrum of these, along with any pertinent parametric controls to tweak them.

Customizable Behavior

While it is useful to provide default scripts, the dynamic responses of some stylizations cannot be easily modeled by simple processes. For these cases, tools are required that can employ the designer’s skills and judgment more directly to describe customizable effects. The same tools are, of course, also useful whenever the designer simply wishes to exercise additional æsthetic control over the results. Figure 5.7 (page 87) shows an example of customizable level of detail (LOD) behavior as the camera zooms in. The artist directly sketched the tonal marking from multiple camera distances, thereby expressing the additional LODs. Simple parametric controls and scripted behavior cannot offer this type of flexibility.

2.5.3 Interactivity

To provide the designer with a flexible, hands-on environment, we seek systems which are fully interactive. This requirement will place constraints upon design choices throughout the system, including the interfaces and algorithms developed to address the two previously outlined design issues. Obviously, this implies *real-time performance*, placing limited computational budgets upon rendering and other supporting algorithms. But interactivity also means that each frame rendered by the system cannot be considered independently. Because they are viewed in real-time sequential animation, rendered imagery should also exhibit frame-to-frame *coherence*.

Real-time Performance

Achieving interactive frame rates will impact many elements of the system. Certain challenges will require novel solutions, while others can employ more established acceleration methods. For instance, frame-buffer tricks are a common way to trade off speed vs. accuracy by exploiting the efficiency graphics hardware at the cost image precision. Indeed, such tricks will be described in the coming chapters, but care must be taken not to fundamentally compromise the fidelity of results should the designer need to "crank up the quality dial" for final production output.

Coherence

The ability to render 3D scenes under dynamic conditions does not automatically imply that the results are coherent under animation. There are many examples in the literature that render stylized 3D geometry from novel view points (e.g. [40, 92]), but do so by considering each frame in isolation. Such imagery is nearly always unsuitable for animation because the rendering primitives depicting them (e.g. strokes) have no correspondence between frames. The animations they produce exhibit noisy, distracting visual artifacts.

To avoid these effects, interactive WYSIWYG NPR systems must employ coherent rendering algorithms. Furthermore, as we will show in Section 4.4, the coherence mechanism itself is subject to various æsthetic considerations. So, it is also necessary to consider design controls that govern this aspect of stylization.

Of course, frame-to-frame coherence is not unique to real-time rendering. It is desirable even if the annotated scenes are rendered to depict scripted animated sequences that are viewed off-line. However, in interactive settings, achieving coherence poses an additional complication. When enforcing frame-to-frame correspondence of rendering elements in any frame, the system can only draw upon information in past frames – the future frames are as yet unknown.

2.5.4 Mapping the Problem Space

As stated at the beginning of this section, these design issues can be mapped out in a table versus each of the stroke-based effects we support in Jot. The resulting problem space (Table 2.4.2) organizes our approach to the development of these tools in the remainder of this dissertation. By addressing the majority of these challenges, we will demonstrate that WYSIWYG systems lead to a new level of æsthetic flexibility for NPR, and rich dynamic effects unavailable to traditional methods.

Chapter 3

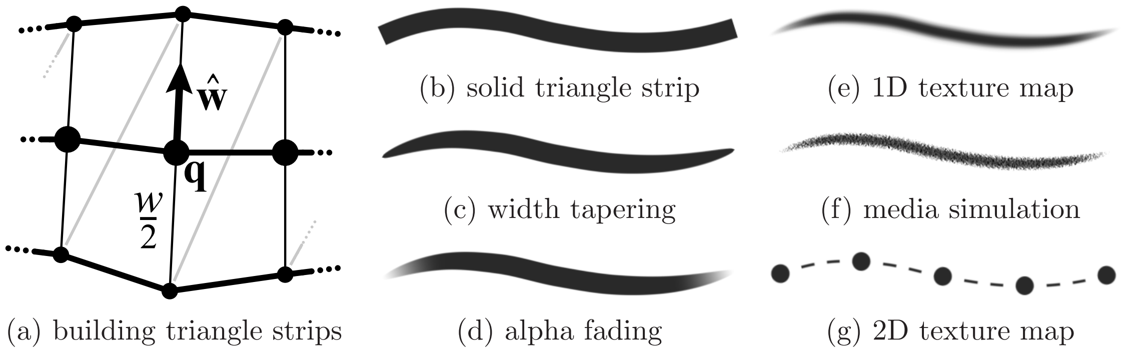
Stroke Primitive

This chapter touches upon several aspects of the rendering primitive that forms the foundation of Jot's stroke-based effects. In Sections 3.1 and 3.2, we describe various design choices and implementation details geared toward providing stylistic flexibility, while achieving real-time performance rates. The æsthetic range of the system is further extended by an efficient media simulation effect presented in Section 3.3.

3.1 Representation

Our stroke primitive is based upon that of Northrup and Markosian [64]. Their approach is motivated by a similar goal of rendering silhouettes in real-time using strokes that resemble natural media. The main difference in our implementation is that vertices describing a stroke path are control points to a Catmull-Rom spline [27] rather than polyline vertices. This vastly improves the robustness and quality of rendering since we can always tessellate the spline smoothly in screen space.

Each stroke is represented by a list of control points $\{\mathbf{q}_i\}$ called the *base path*, and a set of style parameters describing the appearance of the stroke rendered along it. These parameters include a screen-space width, color, and alpha value. Two additional parameters describe how sharply the width and alpha value taper off at the ends of the stroke. Optionally, pressure information from the tablet interface can modulate the width and/or alpha profile along the stroke.



(a) building triangle strips

(b) solid triangle strip

(c) width tapering

(d) alpha fading

(e) 1D texture map

(f) media simulation

(g) 2D texture map

Figure 3.1: Various parameters control the triangle-strip-based stroke primtitive.

Because they convey stylistic features that are inherently 2D in nature, each stroke is ultimately rendered as a triangle strip that lies in the image plane. These primitives can also be rendered efficiently on modern graphics processing units (GPUs). The triangle strip is constructed by stepping along the spline at a desired screen-space resolution. At each point **q**, we generate a pair a vertices separated in the transverse directions ±$\hat{\mathbf{w}}$ by the appropriate screen-space width w, as dictated by the style parameters. The color and alpha values of each vertex pair are also set accordingly.

Additional flexibility is available through the use of alpha texture maps. An optional 1D texture map can modulate alpha along the “cross-section” of the stroke, or a 2D texture map can be used to modulate alpha in both the transverse and longitudinal directions. For instance, a 1D texture could be used to diminish alpha toward the perimeter of a stroke as in Figure 3.1e, or cause it to saturate abruptly as in Figure 3.3a. On the other hand, 2D texture maps might be used to produce effects like arrows heads, or broken lines as in Figure 3.1g.

Another dimension of expressiveness is achieved through the use of *offset lists* that introduce screen-space displacements to the base path [56]. Each offset records a parametric position along the base path and a screen-space displacement perpendicular from that point. To render strokes with offset lists, we first map the list out along the base path spline, perturbing the path appropriately in the normal direction. This will define new stroke splines that can then be rendered as described above.

Offset lists can be used to achieve a myriad of effects, and are leveraged extensively in Chapter 4. For instance, they can impart wiggles about the base path, such as the stylized creases on the gear in Figure 4.1b (page 32). They can also introduce breaks, as demonstrated by the thorny stylization of the cactus silhouettes in Figure 4.4a (page 45). Offset lists can even progress forward and backward along the base path to produce effects like the scribbled silhouettes in Figure 1.1e (page 4).

3.2 Visibility

The triangles composing both our strokes and 3D meshes (Section 2.3) can be rendered efficiently by the modern GPU. Unfortunately, the visibility of these elements cannot be resolved by the standard z-buffer. Even though the base path of a stroke may fall upon a mesh surface, there is no guarantee that the triangle strips constructed to depict it will not interpenetrate the surface. This is so because the triangle strips possess a non-zero width and lie parallel to the view plane. The problem cannot even be resolved by carefully constructing the triangle strips in the surface's tangent plane at all points – it would still interpenetrate in non-convex regions.

Instead, we adapt the method of Northrup and Markosian [64], that resolves visibility using an item-buffer called the *ID reference image*. Into this off-screen region, we render all mesh surfaces as triangles and any feature lines as polylines, taking care to encode each element in a unique ID color. The conventional z-buffer algorithm will ensure the frame-buffer receives the ID color belonging to the visible element at each pixel (Figure 4.6, page 51). We can read back this buffer at 60Hz on modern graphics hardware. Now, stroke visibility is resolved at interactive rates by querying the ID image appropriately for the expected ID values in O(1) time. With this mechanism, the stages to render a single frame in Jot appear as in Table 3.1.

Pass	Action	Comments	Ref	z-Rd	z-Wrt
1.	draw mesh tris	in ID colors	§3.2, 4.4.6	√	√
2.	draw feature lines	in ID colors	§3.2, 4.4.6	√	X
	–read back ID image–				
3.	draw bkgd. image	via media shader	§2.3, 3.3	√	√
4.	composite mesh tris	via surf. shaders	§2.3, 3.3	√	√
5.	composite stroke effects	vis. w/ ID image	§3, 4, 5	X	X

Table 3.1: The ID image is rendered prior to the stylized scene in each frame.

In Sections 4.3.1 and 4.4 we present additional techniques for using the ID reference image to compute visibility for silhouettes and hidden lines effects, as well as achieve coherence for stylized features and contours.

3.3 Media Simulation

We would like to expand the æsthetic range of the system to encompass strokes styles that resemble traditional media. This is accomplished by subjecting the strokes described in Section 3.1 to a media simulation effect. When a stroke primitive is composited into the frame buffer, a shader models the effect of a brush stroke depositing pigment onto an uneven paper surface. By choosing the appropriate stroke properties and paper texture, a diverse variety of effects can be achieved (Figure 3.3).

Like Curtis et al. [16] and Durand et al. [24], we begin by representing the paper surface with a height field texture. To model deposition of pigment onto it by a brush stroke, we interpret the incoming primitive's color as the brush pigment, and the alpha component as the brush pressure. As usual, the amount of pigment deposited at a given point will be proportional to the pressure. But to account for the uneven surface, this pressure will first be modified by the local height of the paper texture.

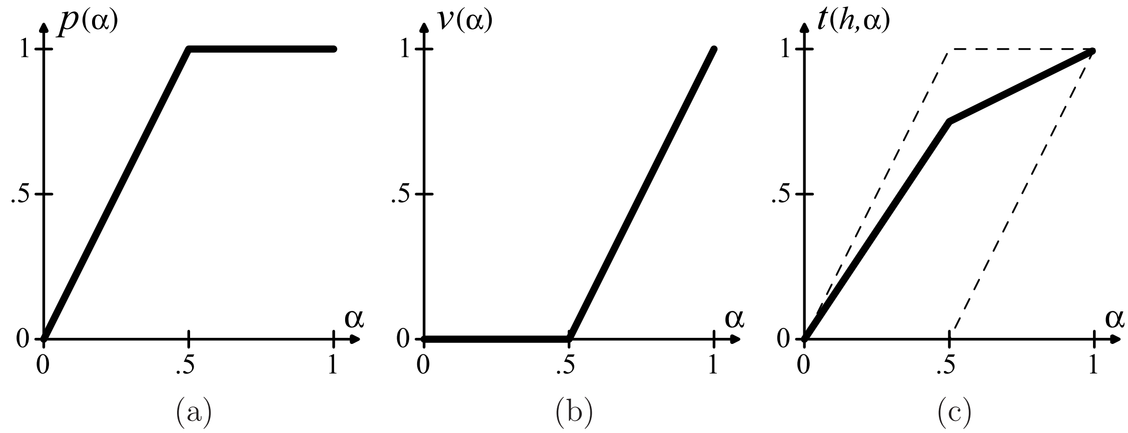


Figure 3.2: Plots of media simulation transfer functions: (a) the canonical peak, (b) the canonical valley, and (c) an intermediate height function, shown for $h = 0.7$.

In formal terms, the paper texture encodes a height field $h \in [0, 1]$. At peaks ($h = 1$) the paper easily catches pigment, whereas in valleys ($h = 0$) it resists pigment. We model this process by applying a transfer function to the α component of the incoming primitive. For peaks, we use the transfer function $p(\alpha)$, and, for valleys, we use $v(\alpha)$, each shown below. At intermediate height h, we employ transfer function $t(h, \alpha)$ that reflects the relative peak-like or valley-like character of the point:

$$p(\alpha) = \texttt{clamp}(2\alpha) \tag{3.1}$$

$$v(\alpha) = \texttt{clamp}(2\alpha - 1) \tag{3.2}$$

$$t(h, \alpha) = h \cdot p(\alpha) + (1 - h) \cdot v(\alpha) \tag{3.3}$$

Figures 3.2a and b contain plots of the canonical peak and valley transfer functions, $p(\alpha)$, and $v(\alpha)$. Clearly, peaks will accrue additional pigment, while valleys will resist it. The general transfer function, $t(h, \alpha)$, is plotted in Figure 3.2c for the case of $h = 0.7$, but any arbitrary h will yield similar functions within the dashed envelope of $p(\alpha)$ and $v(\alpha)$. One can show that, as long as h is evenly distributed about $h = 0.5$, the average tone over a region will not be changed by the transfer function.

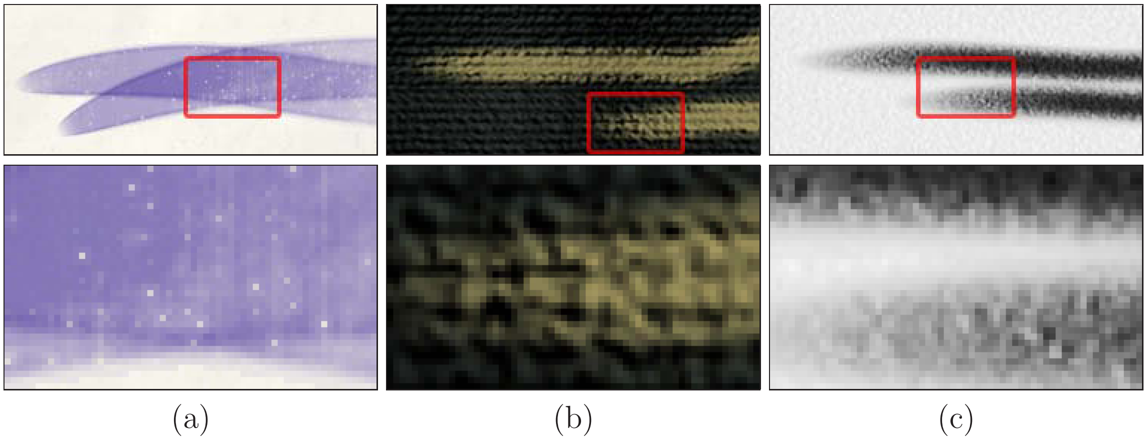


Figure 3.3: Adjusting stroke and media parameters can yield a wide variety of effects.

The final effect is achieved by implementing $t(h, \alpha)$ as a pixel shader. For each incoming stroke pixel (r, g, b, α) the shader determines h of the underlying paper, and remaps α through $t(h, \alpha)$. The final pigment (r, g, b) is composited with additive α-blending. Additional implementation details are found in Appendix A.

This transfer function was chosen because it is natively available in 1st generation programmable graphics hardware. As such, the effect can be implemented on any modern GPU without incurring additional performance cost. Despite the apparent simplicity of the transfer functions, we find that with appropriate parameters and textures, a wide range of effects can be achieved. In fact, we create a library of brush presets for the designer that resemble such traditional effects as ball-point pen (Figure 1.1e, page 4), oil paint on rough canvas (Figure 3.3b), lead pencil on paper (Figure 3.3c), watercolor (Figure 4.4c, page 45), crayon (Figure 5.9, page 90), etc.

The media effect is not limited to strokes. Any primitive that possesses an α-component can be composited with the $t(h, \alpha)$ shader. In Figure 3.4, the silhouettes of a statue are rendered using stylization resembling a chalk on rough paper effect, but the same media shader is also applied to the triangles of the mesh. The α-component of the surface comes from a 'toon shader with a narrow α-band at grazing angles, leading to a subtle effect near silhouette boundaries.

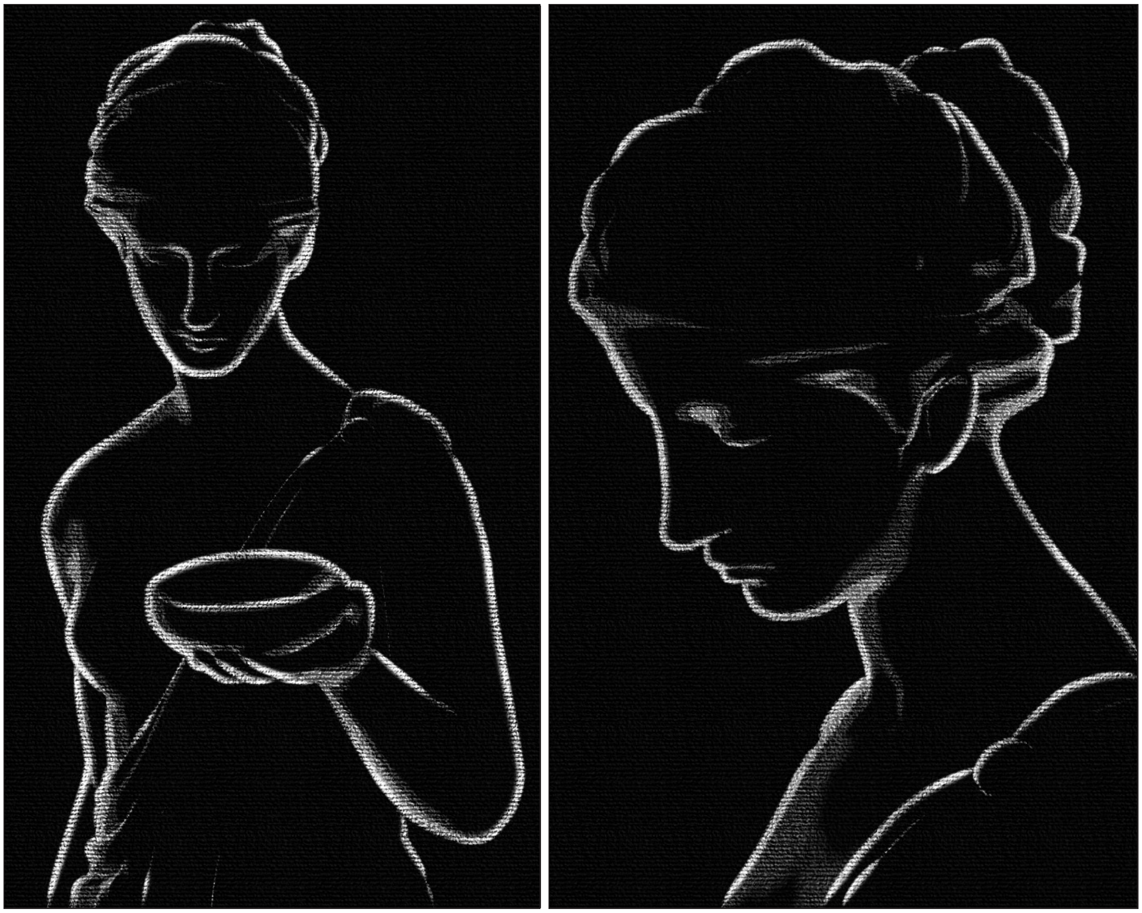


Figure 3.4: Wide silhouette strokes are rendered on a coarse paper texture. A narrow 'toon shader in the alpha channel achieves a subtle effect via the media simulation.

Pursuing the analogy that the viewport represents a physical surface, it is natural to fix the paper texture to the view plane. However, as described by Meier [61], rendering dynamic 3D scenes with static screen-space effects can produce an undesirable "shower door effect". We combat this effect by giving the user an option to "pin" the coordinate frame of a paper texture to the bounding box of an object. If each primitive depicting an object uses a paper texture so defined, we find that the shower door effect can be mitigated. In fact, the user is free to assign an entirely different paper texture to each stroke and surface effect in the scene, even multiple paper textures for a single object.

When textures are pinned to moving objects, the shower door effect is not generally apparent under object translation and some rotations. However, when an object is magnified by relative camera motion or zoom, the shower door effect returns because the paper texture is not simultaneously magnified in screen-space. A possible avenue of future work could investigate multi-resolution paper textures. Cunzi et al. [14] have investigated this using turbulence functions [65] in a framework that can both track object motion and support magnification.

Our general approach is similar to a strategy described by Durand et al. [24]. They use a transfer function to re-map tones to produce bi-level (halftone) output, and a modification to blur this bi-level output for better results on color displays. In contrast, our transfer functions were designed to reproduce patterns we observed in scanned images of real crayon and pencil on paper. They are simpler and better suited to implementation on current pixel-shading hardware. And, because we remap alpha instead of tone, our method handles arbitrary colors more flexibly in both source and destination pixels.

Chapter 4

Line Effects

4.1 Introduction

Certain lines, such as silhouettes and sharp creases, factor critically into the perception of 3D shape from drawings. Accordingly, graphics researchers have developed a number of algorithms to find and render these lines from 3D models. In fact, a major branch of NPR work addresses the depiction of these lines with strokes that wiggle, vary in width and texture, or resemble in other ways strokes drawn by hand with natural media. General arguments for the value of these stylizations have already been made [23, 30, 85]. They can, for example, suggest surface texture (Figure 4.4a, page 45), depict rich hand-drawn stylization (Figure 4.4c, page 45), or simply call out special features such as hidden lines (Figure 4.4b, page 45).

Many researchers have approached the goal of rendering silhouettes at interactive rates. Several techniques have been developed to accomplish this entirely on the GPU by carefully tuning the conventional rendering pipeline [31, 71, 72]. However, the results do not offer any significant æsthetic flexibility. Other techniques achieve slightly more stylistic range by first extracting the feature lines from the geometry, and then rendering stylized elements that resolve visibility in the z-buffer [51, 73, 95].

More expressive stylization is possible with techniques that, prior to rendering, first extract feature lines from the geometry, determine visibility, and produce smooth stroke paths [40, 58, 92]. More recent work has explored how to achieve this in real-time by exploiting the z-buffer and rendering hardware [7, 44, 56, 64]. But, to date, virtually no work has addressed the issue of frame-to-frame coherence for these stylizations, even in off-line systems. Furthermore, these systems offer only limited parametric control over the look and behavior of æsthetics.

In this chapter, we describe the development of a set of WYSIWYG tools that provide the designer with very direct control over these stylizations. Using a drawing interface, the artist can literally sketch into the 3D scene to describe line stylization with a flexible stroke-based primitive. The system will ensure that the artist's stylizations retain frame-to-frame coherence, even during real-time interactive rendering.

We consider first in Section 4.2 the stylization of *static feature lines* that remain stationary on the 3D surface, while we reserve the more challenging problem of *dynamic contours* for Section 4.3. Finally, we develop a general framework for ensuring the frame-to-frame *coherence* of these stylizations in Section 4.4.

4.2 Static Feature Lines

In Jot, static feature lines are represented as chains of edges or polylines that lay on 3D meshes. They can arise on surfaces at sharp creases (Figure 4.1a), boundaries between distinct patches of surface, borders of non-manifold surfaces, and other salient features at the designer's discretion. These lines are typically loaded as part of a 3D model's description, but we also provide controls for automatic crease generation via dihedral angle testing. A bending threshold controls heuristics that chain the sharp edge networks into static feature lines.

4.2.1 Over-Sketching Stylization

To develop WYSIWYG NPR tools for static features lines, we first consider the stylization interfaces with which the designer annotates them. As described in Sections 2.3 and 3.1, we provide various parametric controls to customize the brush styles available with our stroke primitive. Although means exist to adjust many different stroke qualities, they do not result in a particularly hands-on degree of control.

However, an additional dimension of stylization lends itself to more a WYSIWYG style interface. The offset lists described at the end of Section 3.1 enable our stroke to exhibit a wide spectrum of effects by deviating about the base path. By exposing this flexibility through an over-sketching interface (Section 2.2), we allow the artist to more directly impart a personalized æsthetic.

When a user over-sketches a selected feature line with a brush stroke, the result is recorded into an offset list. We accomplish this by treating the selected feature line as the base path associated with the sketched out stroke. For each point along the stroke, a perpendicular screen-space offset is recorded, as measured from the nearest parameterized point along the base path. If necessary, the end-tangents are extended to accommodate offsets that overshoot the path.

The artist is free to sketch over a given feature line any number of times, even using a unique brush style if desired. Each stroke will result in an additional set of style parameters (i.e. brush style and offset list) that are associated with the selected feature line. It is then straightforward to maintain these stylizations in novel views. Because static feature lines remain fixed on the mesh, they are easily matched with their respective style parameters. In a given frame, the system simply projects each feature line into screen-space, and renders the visible portion using a primitive with the corresponding parameters.

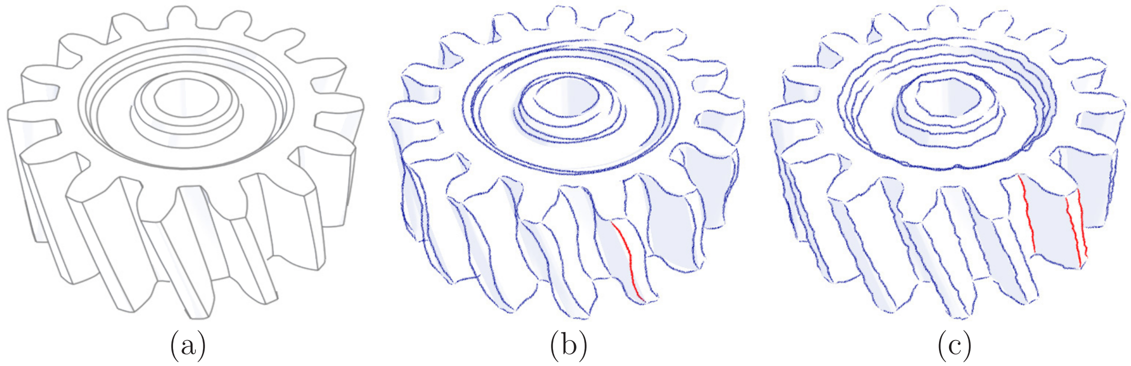


Figure 4.1: Stylization synthesis: The gear in (a) is stylized via (b) rubber stamping a single example stroke, and (c) markov synthesis using three example strokes.

4.2.2 Stylization Synthesis

Via over-sketching, the designer is free to stylize each static feature line individually. While offering a high degree of direct control, this approach can be tedious when annotating arbitrarily complex scenes. Ideally, we would enable the designer to automatically stylize many lines at once, while still retaining hands-on æsthetic control. This is accomplished by providing tools to synthesize stylization from the designer's own annotations. With these tools, the designer selects multiple feature lines and stylizes them all at once by sketching only a few examples.

Jot provides two such schemes. The first of these, *rubber-stamping*, copies the style parameters from a chosen feature line to all others targeted for synthesis. The crease lines of the gear in Figure 4.1b were stylized with this method. A single crease was annotated with a wavy line shown highlighted in red, the remaining lines are synthesized by rubber-stamping it. Though limited, this scheme is useful for intrinsically repetitive stylizations like dashed lines.

Another useful synthesis method might provide unique results that exhibit similarity to the example annotations. Freeman et al. [28] describe a method of transferring stylization from a set of example strokes to a new line drawing. Unfortunately, their methods require a large set of examples (e.g. > 100) which does not meet our criteria for reducing the annotation burden. We describe a method, *Markov synthesis*, that can work well with just a few example strokes. To synthesize new annotations, we perform one-dimensional texture synthesis on the offset list using Markov random fields. We can then create variety by synthesizing a new offset list for each feature line requiring stylization. Figure 4.1c shows the gear model with three creases annotated by the designer and highlighted in red, while the remaining creases were synthesized from the example set. Markov random fields have been recently used in graphics in other data-synthesis applications [6, 26, 89]. Our algorithm closely follows the "video textures" algorithm presented in Schödl et al. [76]; rather than generating new video sequences from a set of example clips, we synthesize new sequences of offsets from a set of example strokes. The synthesis algorithm constructs a Markov random field where each state corresponds to an offset in an example list; the transition probability between two states is a function of the local similarity between the two points. With this method, we can generate many offset lists containing features from a small example set. Additional details are given in Kalnins et al. [47]. The scheme works best with stochastic stylizations (i.e., non-periodic), but more recent work handles structured stylizations more effectively [38]. A more complete WYSIWYG NPR system could offer stylization synthesis via a range of such algorithms.

4.2.3 Level of Detail

As mentioned in Section 4.2.1, it is straightforward to maintain annotations on the static feature lines in novel views. In a given frame, the system simply projects each feature line into screen-space, and renders the visible portion as a stroke base path using the corresponding style parameters. Even if the surface of the mesh deforms via some animation, the locations of the static feature lines still remain well-defined. We have, however, glossed over precisely how offset lists are mapped out along these new base paths. It is these details that largely define how the stylizations behave in response to dynamic influences.

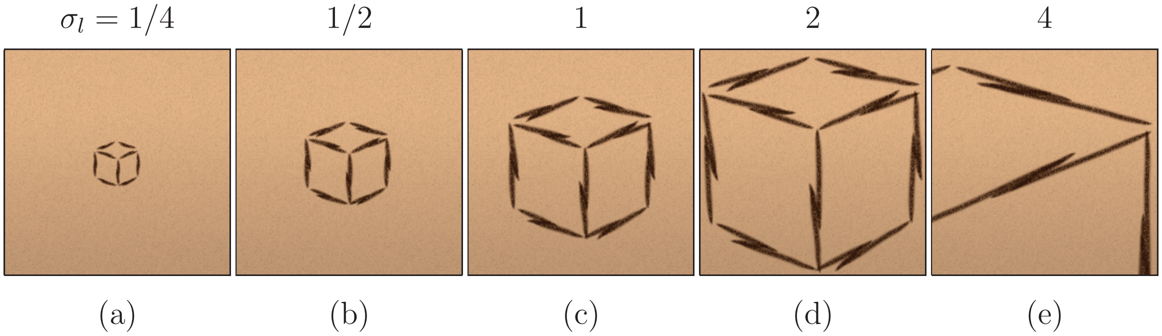


Figure 4.2: A cube with annotated creases. The magnitude of the stylization offsets are modulated by $\texttt{min}(\sigma_l, 1)$.

One such detail concerns how offset magnitudes respond to changes in level of detail. This effect arises when the screen-space projection of a feature line varies in length due to foreshortening or changing magnification. It can result from camera motion, animation in the scene, are some combination of both.

When base paths shorten in new views, offsets maintaining their original magnitude will cause an ever increasing amount of apparent detail to "pile up" along a vanishing length. We find this policy to be unnatural and distracting for most conceivable annotations. In these situations, it is necessary to reduce the level of detail imparted by the offset list by shrinking the magnitude of the displacements.

Conversely, when changing views cause base paths to lengthen, we find multiple policies that are reasonable. If the displacements grow in proportion with their base paths, the stylization can appear to describe geometric perturbations to the underlying 3D feature lines. However, if the offsets maintain their original magnitude, this invites a 2D screen-space interpretation of the stylization. In some sense, the designer's annotations continue to live in screen-space, even though they depict an underlying dynamic 3D scene.

In Jot, we choose to implement a default scripted behavior based upon the later policy. For each offset list, we define σ_l as the ratio of the current 2D length of the feature line, to its original 2D length when annotated by the designer. In novel views, the offset magnitudes are modulated by $\texttt{min}(\sigma_l, 1)$, causing displacements to scale down when the base path minifies, but remain unchanged during magnification. Stated another way, the relative level of detail of an annotation diminishes as it deviates from the context in which it was made. Figure 4.2 provides an example of this behavior. The creases of the cube were originally annotated in Figure 4.2c ($\sigma_l = 1$). As the cube recedes in the image ($\sigma_l < 1$), the offsets are diminished; while as the cube grows ($\sigma_l > 1$), the displacements remain constant.

While this scripted behavior is suitable for a wide class of stylizations, a more complete system might enable the designer to choose from amongst a collection of such scripts – presumably assigning one deemed suitable for each given stylization.

4.2.4 Parameterization

When offset lists are generated by annotations, each displacement is associated with a parameterized position along its underlying feature line. Before the offsets can be remapped in a new view, the feature lines must again be parameterized. Under dynamic viewing conditions, the chosen parameterization scheme will have a significant impact upon semantic aspects of the final æsthetic.

Arc-length parameterizations are the most natural to consider. For instance, if feature lines are parameterized using 3D arc-length, their annotations can invite a geometric interpretation under dynamic conditions. This is because elements of the stylization “stick” to the 3D line, often causing the displacements to appear like perturbation in the actual geometry. On the other hand, 2D arc-length parameterization does not exhibit 3D cues, such as perspective foreshortening, lending itself to more of a screen-space interpretation.

We choose to implement 2D arc-length parameterization as the default scripted behavior in Jot. This is consistent with our default policies governing offset magnitudes that also imply 2D character by their dynamic behavior. The system could offer the designer a variety of scripted behaviors to chose from, but we will revisit this policy again after considering the coherence of view-dependent contours. With the framework developed in Section 4.4, it will become possible for the designer to tailor the parameterization to strike a particular balance between 2D and 3D behavior. However, for applications in which this framework is too laborious to implement, the simple schemes above may be useful.

4.2.5 Interactivity

The systems described for stylizing static feature lines do not present any additional hurdles to interactive performance rates. Visibility of static feature lines is readily determined in real-time using the ID image described in Section 3.2. Rendering then proceeds via a stroke primitive that is designed for efficient implementation on the GPU. Furthermore, frame-to-frame coherence is inherent for the stylizations we have described. Because they are static on the geometry, the feature lines are trivially associated with their respective stylizations in each frame and their arc-length parameterization is intrinsically coherent.

4.3 View-Dependent Contours

The view-dependent contours supported in our system take the form of polylines on the 3D mesh. In general, they arise dynamically from a surface as some function of its orientation in the camera frame. The most familiar variety are *silhouette contours* that form the boundary between front-facing and back-facing regions of surface; however, Jot also enables the annotation of *suggestive contours* [19]. From a stylization standpoint, though, these contours do not differ appreciably from silhouettes. In this section, we confine our discussions to silhouette contours with the understanding that, unless otherwise noted, the results apply equally to suggestive contours.

4.3.1 Silhouette Contours

Before silhouettes can be annotated or rendered with strokes, they must be extracted from the underlying geometry for the given view. In the case of polygonal meshes, the most common definition of the "silhouette" is the set of silhouette edges – i.e., edges of the mesh that share a front- and back-facing polygon. It is straightforward to check each edge of a mesh in brute force fashion to extract the silhouettes for a given frame. As scene complexity increases, however, this approach can become too expensive for real-time applications.

Recently, several techniques have been proposed for detecting silhouettes without traversing the entire mesh [1, 11, 40, 43, 66, 75]. These methods employ hierarchical data structures to accelerate the search for silhouettes, yielding significant performance improvements for finely tessellated models. Unfortunately, these methods are not well-suited to real-time animation because the data structures must be rebuilt whenever the geometry deforms. Instead, we employ the stochastic methods of Markosian et al. [56] that query the geometry directly without need for an intermediate data structure. This approach can locate all of the silhouette edges to within a chosen level of confidence by testing an appropriate random subset of the mesh.

Once located, the silhouette edges cannot be rendered directly using stylized strokes. It is first necessary to process them into smooth screen-space paths. This problem may not be visually evident if the edges are rendered as a collection of simple line segments as this will yield an apparently smooth result in screen-space. However, the chains of edges actually zig-zag in 3D and bifurcate into arbitrarily complex networks that can backtrack in the image plane. Northrup and Markosian [64] propose a set of heuristics that attempt to clean up the silhouette edges and produce paths suitable for strokes. In practice, however, we find it more effective to pursue an alternate definition of the silhouette that avoids these difficulties altogether.

We will use the stochastic approach of Markosian, but instead of finding silhouette edges, we adapt it to find the silhouette contours defined by Hertzmann and Zorin [40]. For a given vertex $\mathbf{p}$ with normal $\hat{\mathbf{n}}$, they define camera vector $\mathbf{v}$ and scalar field g:

$$\mathbf{v} = \mathbf{c} - \mathbf{p} \tag{4.1}$$

$$g = \hat{\mathbf{n}} \cdot \mathbf{v} \tag{4.2}$$

and extend g to triangle interiors by linear interpolation. The silhouette contour is then given by the zero-set of g. This takes the form of clean polylines whose segments traverse the faces of the mesh, forming loops or terminating on feature lines. We call these the *silhouette loops*. To find them at run-time, we sample a small pool of randomly selected faces in the mesh. For each face, we determine g at the vertices and test its sign. If the signs differ within a given triangle, then the silhouette contour must cross it, exiting from two of three sides into neighboring triangles. When a loop is found, we trace it out until returning to the starting triangle or intersecting a feature line. As described by Markosian [55], we can locate the entire family of silhouette loops up to a desired level of confidence by testing a pool of $O(\sqrt{n})$ random triangles. This performance improves even further by populating the pool with triangles containing the contour from the previous frame.

The final step is to separate the silhouette loops into contiguous visible *silhouette paths* suitable for stroke rendering. As described by Hertzmann and Zorin, the silhouette contour will be occluded by the surface it borders in regions where $\nabla g \cdot \mathbf{v} < 0$. Therefore, we break silhouette loops at the cusps ($\nabla g \cdot \mathbf{v} = 0$) and discard the "back-facing" regions. However, the remaining silhouette paths may still suffer occlusion, either by distant regions of the same surface, or by another object in the scene. We compute the remaining visibility efficiently using the ID reference image described in Section 3.2. As shown in Figure 4.6 (page 51), silhouette polylines and mesh triangles are rendered with unique ID colors into an off-screen buffer, and the z-buffer resolves visibility. Each loop is assigned a unique 24-bit ID using three color components. The remaining 8-bit component is varied along the length of each loop. We test this image along the remaining polylines, culling away portions that yield incorrect colors.

By using the zero contour of g instead of the set of silhouette edges, we easily produce smooth paths amenable to rendering with our stroke primitive. However these contours lie on the interior of triangles, and, therefore, do not describe the true silhouette of the polygonal model. In fact, they correspond to the silhouette contours of a smooth surface to which the polygonal mesh is only an approximation. This becomes apparent if mesh is viewed under magnification so that the triangles become large in screen-space. Under these conditions, the screen-space disparity between the edges of the polygonal mesh and the contours can become visually significant. This will also interfere with visibility of the contours where they lie on back-facing triangles. To combat this, we render the ID image using `GL_POLYGON_OFFSET` [81] to push the mesh triangles back relative to the contours.

Even though this solution still breaks down under high magnification, given the quality of the stroke paths we can achieve in real-time versus edge methods, we find this limitation to be an acceptable trade-off. Furthermore, we find that via subdivision meshes [41, 54], we can tessellate most models sufficiently to avoid the edge-contour disparity under typical interactive viewing conditions. And, of course, this is easily guaranteed for scripted animations. Nevertheless, a more complete solution could implement an adaptive subdivision scheme. By locally tessellating the surface near the contour, the disparity might be maintained at sub-pixel levels.

4.3.2 Sketching Prototypes

Once dynamic contours have been extracted from the geometry, we can present the designer with WYSIWYG tools to annotate them. As described for static feature lines, simple brush style adjustments do not offer a sufficient degree of hands-on control. We augment this with an interface for directly sketching over the contours using a tablet, enabling the artist to more directly impart a particular æsthetic.

However, we cannot implement the same solution presented in Section 4.2.1. There, each annotation produces an offset list associated with the chosen static feature line. This type of one-to-one correspondence cannot be defined for contours which are dynamic on the geometry. To resolve this ambiguity, we *rubber-stamp* all silhouette contours of the selected surface using the *prototype* defined by the artist's annotation. In the event that some contour exceeds the length of the given prototype, it can be replicated periodically.

While the designer may impart this prototype by sketching over a particular contour, in practice we find it more effective to sketch against a horizontal baseline. Any number of constituent strokes can be sketched along such a line, and the final result is encoded into an offset list. In novel views, the system simply renders all the silhouette contours of the given surface with the style parameters of the prototype.

Each type of dynamic contour need not share the same stylization. The silhouettes and suggestive contours of a given mesh, for instance, can each be annotated with different prototypes. To extend the flexibility further, the surface of a mesh can be divided into multiple patches. The contours that arise from each individual patch may be assigned their own individual prototypes. The system easily associates the appropriate prototype with the contours extracted in each frame.

4.3.3 Level of Detail

Once again, we have glossed over the precise details of how offset lists are mapped out along their respective base paths. These particular details will govern how the stylizations behave in response to dynamic influences. While these issues are explored in Sections 4.2.3 and 4.2.4 for static feature lines, they must be re-examined in the context of dynamic contours.

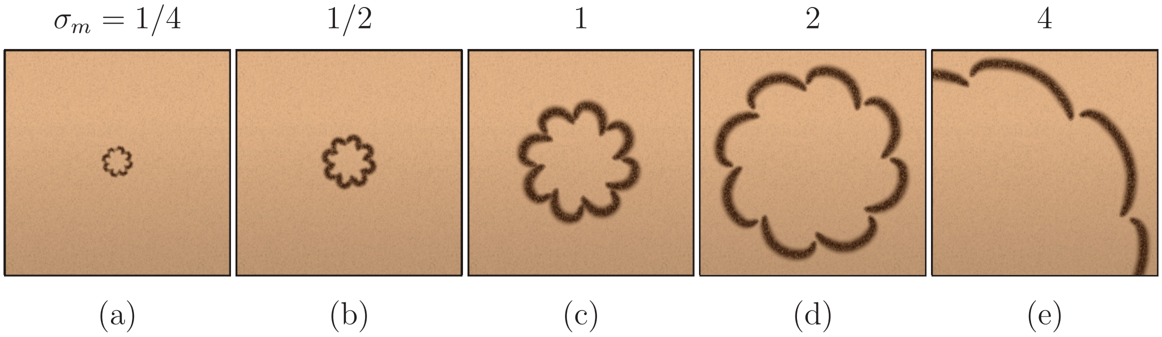


Figure 4.3: A sphere with annotated silhouettes. The period and magnitude of the stylization offsets are modulated by σ_m and $\texttt{min}(\sigma_m, 1)$, respecively.

The first of these details concerns how offset magnitudes respond to changes in level of detail. The importance of this becomes clear with an example. Consider a mesh shrinking in the image plane as it recedes from the camera. If the annotations maintain their original magnitude, an increasing amount of detail will "pile up" along silhouettes, ultimately overwhelming the mesh. We find this policy generally distracting for most conceivable annotations. In these situations, it is preferable to reduce the level of detail imparted by the stylization by shrinking the offset displacements.

In views that cause the model to grow beyond its original annotation size, there are multiple policies that may be viable. For instance, if the displacements grow in proportion with the mesh, silhouette deviations may suggest geometric perturbations to the underlying surface. If the offsets maintain their original magnitude, on the other hand, the stylizations instead appear to inhabit the image plane.

We choose to implement a default scripted behavior that exhibits the image-space character of the later policy. This is consistent with the choice made for stylizations on static features lines. However, we cannot implement this with the same ratio, σ_l, that measures the changing screen-size of the 3D feature line. Because the silhouette contours can arise from any arbitrary region of the mesh, they cannot be associated with any one single 3D path. Instead, we must find a metric that reflects the changing size of the mesh itself.

Accordingly, we define σ_m as the ratio of the current 2D radius of the bounding sphere of the mesh to its original 2D radius when the annotation was made by the designer (Figure 4.3c). In novel views, the offset magnitudes are modulated by $\mathtt{min}(\sigma_m, 1)$. With this scheme, displacements scale down when the mesh shrinks on screen (Figures 4.3a,b), but remain unchanged when it is magnified (Figures 4.3d,e). In some sense, the annotations diminish in detail as their mesh deviates from the context in which they were made.

We find that this scripted behavior is suitable for a wide class of stylizations. Nevertheless, a more complete system should enable the designer to choose from amongst a collection of such scripts, so as to assign one deemed suitable to a given stylization. Furthermore, σ_m is not well-suited to all meshes. For those with large depth extent (e.g., landscapes) it will be necessary to find a more local metric of scale.

4.3.4 Parameterization

Before the prototype offset list can be mapped out along the silhouettes, the visible silhouette paths must be parameterized. Under dynamic viewing conditions, the chosen parameterization scheme will have a significant impact upon the frame-to-frame coherence of the stylization, as well as semantic aspects of the final æsthetic.

The problem of coherence is particularly challenging for silhouette contours. Although the silhouettes exhibit a natural coherence in the image plane, their evolution on the underlying geometry can be arbitrarily complex between successive frames. The frame-to-frame correspondence of screen-space silhouettes cannot be directly established from their 3D representation. For this reason, solutions based on 3D arc-length described for static feature lines (Section 4.2.4) are unsuitable. Nevertheless, we have already reduced the burden of achieving these correspondences. By rubber-stamping a single prototype stylization along all silhouette paths, the parameterization at any point remains unique only up to the period of the offset list.

We now examine the remaining natural parameterization; namely, 2D arc-length along the visible silhouettes. To assign such a parameterization to a given silhouette path, we simply choose a starting parameter and the remaining parameterization is defined by screen-space arc-length. Because the starting parameter is only unique up to the period of the prototype, we call it a phase. This is the only degree of freedom for parameterizations assigned with this scheme.

However, as described, this parameterization is of limited value. Consider the sphere in Figure 4.3c annotated with a prototype that repeats eight times around the silhouette. If the sphere doubles in image size, our parameterization scheme would produce sixteen periods of the stylization. Between these two states, the additional periods would “swim” into position along the silhouette. We find this dynamic behavior unnatural and distracting for the vast majority of conceivable annotations. To combat this, we observe that, as the size of the mesh increases in the image plane, the lengths of its silhouettes grow in approximate proportion with its perimeter. If we scale the parameterization with σ_m, which measures the changing circumference of the 2D bounding sphere (Section 4.3.3), we can diminish the swimming considerably. Figure 4.3 demonstrates this for a sphere as it changes over four size scales while maintaining eight periods of the prototype.

It still remains to determine the phase of the parameterization assigned to each silhouette path. If we arbitrarily set this to zero in all cases, a limited degree of coherence can be achieved over some view changes. But, in general, as silhouette paths break or join, phase mis-matches will lead to distracting “pops”. We have investigated means to set phases more intelligently using information from the previous frame, but the results have limited value. As we explain in Section 4.4, parameterizations using 2D arc-length alone are too rigid to describe the dynamic behavior of many stylizations. Instead, we develop a framework that enables the designer to tailor a coherent parameterization with a particular dynamic to each annotation. Nevertheless, for applications in which the machinery of Section 4.4 is too elaborate, the simple parameterization scheme described above may be an effective solution.

4.3.5 Animated Stylization

We provide the designer with tools to describe a customizable dynamic inspired by Tom Snyder's Squigglevision™ technique. This effect was popularized in the cartoon television programme 'Dr. Katz'. Snyder developed a technique to provide a low cost method for turning still illustrations into apparent animations. By tracing over the outlines of the characters several times with wobbly lines and looping the result in random order, the scenes appear dynamic, even though the characters never actually move. To implement this in Jot, we enable the designer to annotate the silhouettes multiple times. The result is a set of prototypes that will be randomly looped as with the original effect. Additional parameters control the speed of the loop and the degree of randomness in the order and timing of the cycle.

Unlike the hand generated case, we can easily extend the effect to moving objects. This is possible because we have an underlying coherent parameterization. Consecutive frames of the stylization may appear to be changing randomly due to randomness in the cycle's order and timing, but the loop comes back into correspondence every couple of frames. We find that if the underlying parameterization changes randomly, the resulting æsthetic is wholly different.

An unexpected side benefit of this effect is the opportunity to employ simpler parameterization schemes. While we do find that a lack of coherence can destroy Squigglevision™, the effect can function satisfactorily with parameterizations that exhibit only marginal coherence. It can, for instance, mask the shortcomings described for the 2D arc-length scheme in Section 4.3.4. If a particular application renders contours with animated stylization only, it may not be necessary to implement the full coherence apparatus described in the next section.

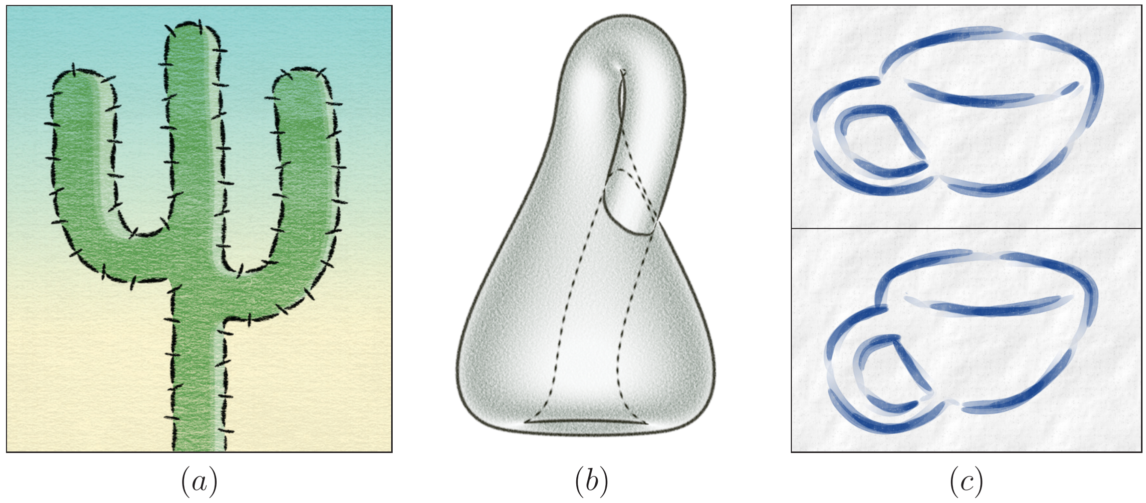


Figure 4.4: Different stylizations require different forms of coherence to (a) maintain correspondences between features on the 3D shape, (b) maintain the uniform 2D arc-length parameterization, or (c) strike some balance between (a) and (b).

4.4 Generalized Coherence Framework

As discussed in Sections 4.2 and 4.3, the frame-to-frame coherence of our stylizations is fundamentally tied to that of the underlying parameterization. But, thus far, we have limited our investigation to parameterizations that arise naturally from the elements being stylized – i.e., 2D and 3D arc-length parameterizations. These schemes are straightforward to compute, but yield significant coherence only on static feature lines. Furthermore, they exhibit either a rigid 2D or 3D dynamic that is suitable only for certain types of annotations.

In this section, we develop a more general framework to coherently parameterize static or view-dependent lines by leveraging information from the previous frame. This approach will enable the designer to tailor the dynamic behavior of the coherence appropriately for a wide range of different stylizations (Figure 4.4).

4.4.1 Overview

In general, stylized lines possess two properties that influence their coherence: the path in the image plane over which the stroke primitive is applied, and the parameterization that defines how stylization is mapped along that path. The first issue poses no particular challenge for static features lines. They remain fixed on the geometry, so their image-space paths enjoy intrinsic coherence. On the other hand, dynamic contours can evolve in an arbitrarily complex fashion in the 3D domain. Silhouette paths, in particular, tend to break and re-join in screen-space in response to these changes. Since their 2D paths pose the greater challenge to coherence, we will focus on silhouettes in this section – the results will extend naturally to creases, etc. To address the second issue, parameterization, we must answer two separate questions: first, what is the unit of stylization to be parameterized? and second, how do we assign coherent parameterization to those units?

There is a seemingly-obvious answer to the first question: one can parameterize individual silhouette paths. Both Bourdev [4] and Masuch et al. [59] investigated this approach independently. This strategy can work well for some sequences of frames; however, coherence will be broken whenever two silhouette paths in one frame merge into a single path in the next frame. There will be a visual "pop" if single parameterization is assigned, unless by chance the two paths have matching parameterizations. To avoid this, Masuch et al. caused the two abutting paths to match in limited cases by enforcing coherence along the underlying silhouette loop. While this strategy prevents some popping, in general, it exacerbates a problem arising whenever a single parameterization is maintained for long silhouette paths. Changes in foreshortening on one area of the model can influence other more distant areas, leading to "swimming" artifacts wherein the stylization appears to slide along the visible silhouette path.

To address these popping artifacts, we do *not* require a single parameterization to be assigned per silhouette path. Instead, we enforce coherence for separate portions of the silhouette path that we call *brush paths*. These paths will also help to ameliorate the swimming problem because stylization becomes more localized. Intuitively, a brush path may be thought of as the path of a single stroke, though, in practice, many strokes may be applied in concert to a single brush path (e.g. for dashed lines). Section 4.4.4 provides a detailed description of how silhouette paths are divided into brush paths using parameterization information propagated from the previous frame (Section 4.4.3).

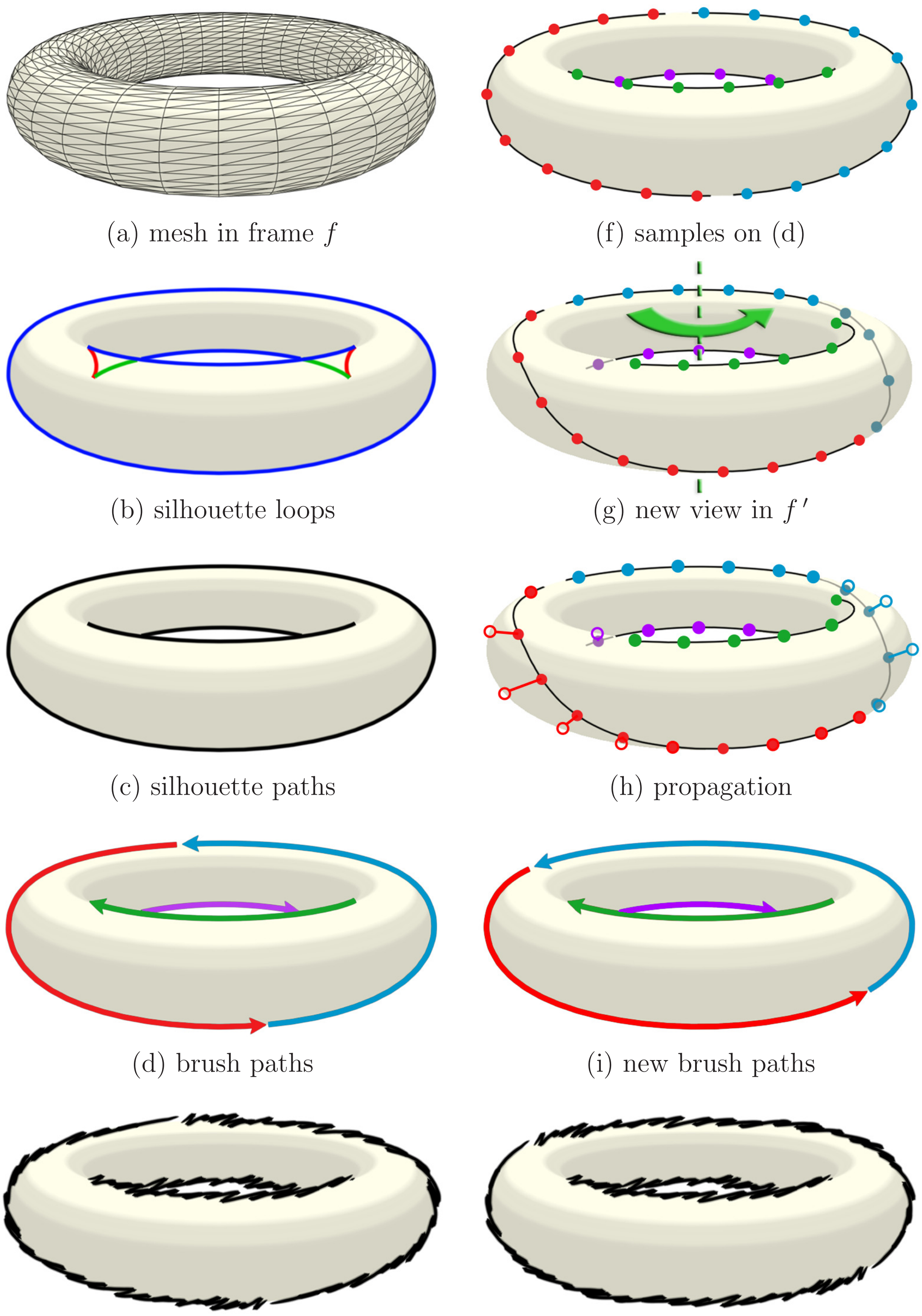
Having identified brush paths as the unit of coherence, we now address the second question above: how do we parameterize them? We demonstrate shortly that there are competing aims: correspondence on the 3D shape (Figure 4.4a) versus uniform 2D arc-length parameterization (Figure 4.4b). The semantics of a particular stylization govern the relative priority of these two goals, generally falling somewhere between these extremes (Figure 4.4c). In Section 4.4.5 we describe a parameterization scheme that can balance these behaviors at the discretion of the designer.

4.4.2 Coherent Rendering Pipeline

We now recap the stages of Jot's rendering pipeline to include those implementing the coherence framework. The pipeline is demonstrated in Figure 4.5, where silhouettes in frame f' are stylized coherently using information propagated from previous frame f.

Silhouette Loop Extraction

The first stage extracts the *silhouette loops* from the underlying mesh (Figures 4.5a and b). We describe a stochastic approach to efficiently locate the zero set of $g = \hat{\mathbf{n}} \cdot \mathbf{v}$ in Section 4.3.1.



(a) mesh in frame f

(f) samples on (d)

(b) silhouette loops

(g) new view in f'

(c) silhouette paths

(h) propagation

(d) brush paths

(i) new brush paths

(e) strokes in f

(j) new strokes in f'

Figure 4.5: Overview of the rendering pipeline for coherent stylized silhouettes. *Left column:* stylizing silhouettes in frame f. *Right column:* propagation to frame f'.

Visible Path Generation

The silhouette loops are then tested for visibility and converted into smooth *silhouette paths* suitable for stroke rendering (Figure 4.5c). As explained in Sections 3.2 and 4.3.1, we achieve this in real-time via an ID reference image.

Sample Propagation

To assign stylization coherently in frame f', we leverage parameterization samples propagated from the previous frame f (Figure 4.5f) via a two-step process described in Section 4.4.3. First, the samples are transformed to their new 3D locations in f' (Figure 4.5g), then a search through the image plane registers their votes along the new silhouette paths (Figure 4.5h).

Brush Path Generation

Brush paths are then generated along the new silhouette paths in correspondence with the votes from the previous frame. The color coding in Figure 4.5i illustrates how the old brush paths in f give rise to the new brush paths in f', as explained in Section 4.4.4

Brush Path Parameterization

The new brush paths are then assigned parameterization derived from their respective votes. In Section 4.4.5 we describe a process to achieve correspondence with the votes, while balancing different coherence objectives appropriately for a given stylization.

Stroke Rendering

Finally, each brush path is rendered using the stroke primitive described in Chapter 3 (Figure 4.4j). The parameterization can be used to map out an *offset list* to describe one or more wobbly strokes (Figures 4.4a and c), or define the longitudinal texture coordinate for a single triangle strip (Figure 4.4b).

4.4.3 Sample Propagation

During rendering, stylization is mapped onto brush paths via the parameter t. To achieve temporal coherence, we propagate this parameterization between frames in the form of discrete samples. In a given frame f we create a set of samples evenly spaced along the brush paths. Each sample $(\mathbf{p}, b, t)$ contains its location on the 3D surface $\mathbf{p}$, the ID of its brush path b (as color-coded in Figure 4.5f), and the stylization parameter t. We store $\mathbf{p}$ as a barycentric location on a mesh triangle so that samples can track the changing surface during animation. The brush path ID will help to coherently define the new brush paths in f' (Section 4.4.4), while the associated t samples will be leveraged to parameterize them (Section 4.4.5).

In subsequent frame f', we seek to propagate parameterization by registering the samples from f against the new silhouette paths in f'. For each sample, this entails determining the 3D correspondence $\mathbf{p} \rightarrow \mathbf{p}'$, where $\mathbf{p}'$ is the location on a silhouette loop in f' receiving the sample. For static feature lines the obvious choice is $\mathbf{p}' = \mathbf{p}$, since the paths remain fixed on the 3D surface. Unfortunately, it is not clear how to establish meaningful 3D correspondences for silhouettes (and other view-dependent contours) because their evolution on the 3D surface is arbitrary between f and f'.

However, the successive frames of a coherent animation will typically exhibit only moderate camera motion and mesh deformation. Under these conditions, silhouettes (and suggestive contours [18]) enjoy a coherent evolution in screen-space. If we define $proj_{f'}(\mathbf{p})$ as the projection of $\mathbf{p}$ into the f' image plane, we can exploit this screen-space coherence to find reasonable 2D correspondences $proj_{f'}(\mathbf{p}) \rightarrow proj_{f'}(\mathbf{p}')$. This is accomplished efficiently using the ID reference image described in Sections 3.2 and 4.3.1. Each sample projects to a location in the ID image of f' as shown in Figure 4.6. If frame-to-frame animation is coherent, the samples will generally appear near the new silhouette paths. The correspondence for a given sample can then be found by searching nearby $proj_{f'}(\mathbf{p})$ for silhouette pixels, and determining a suitable $\mathbf{p}'$.

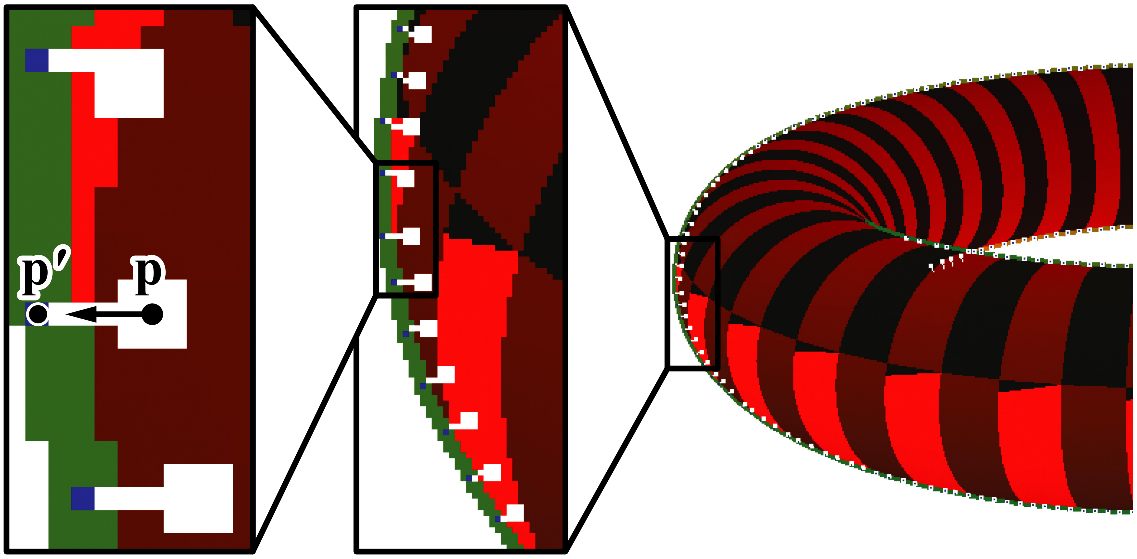


Figure 4.6: An example of sample propagation in the ID reference image of a torus, shown in successive enlargements. Silhouette path IDs appear in green shades, while mesh faces are shown in shades of red. Search locations are represented in white, with successful termination indicated by blue pixels. (Note: The striping is an artifact of the order in which ID colors are assigned to triangle strips on subdivision meshes.)

It still remains to define precisely how these 2D correspondences are determined. A natural choice is *closest-point correspondence* – for each $\mathbf{p}$ the corresponding $\mathbf{p}'$ is taken as the location on a visible silhouette path in f' that minimizes distance in the image plane:

$$\| \, proj_{f'}(\mathbf{p}') - proj_{f'}(\mathbf{p}) \, \| \tag{4.3}$$

By using the ID image, we avoid the cost of an exhaustive search for $\mathbf{p}'$. For each sample, the ID image is queried in a one pixel radius about $proj_{f'}(\mathbf{p})$ and any silhouette IDs are recorded. The radius is incremented and the search is repeated until at least one silhouette ID is found, or the radius exceeds some threshold. Once the search terminates, $\mathbf{p}'$ is computed by minimizing the closest-point criteria (4.3) on the subset of silhouette loops located within the search radius.

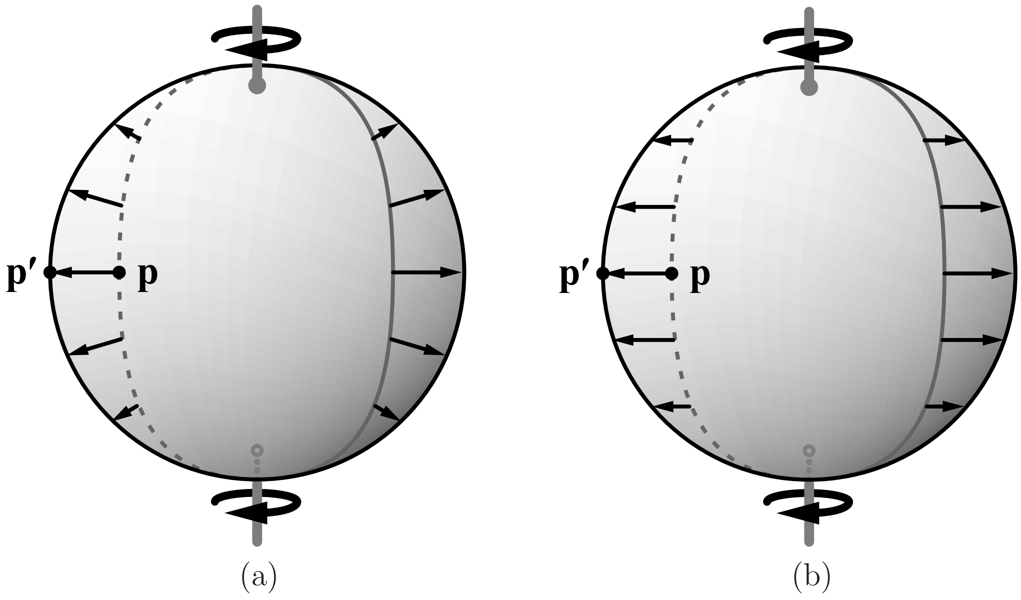


Figure 4.7: Propagating the samples from frame f to f' for a spinning sphere. In (a) *closest-point correspondence* propagates samples toward the nearest silhouette, and in (b) *epipolar correspondence* propagates samples according to camera motion.

We have also investigated a variant of this algorithm, *accelerated closest-point correspondence*. As before, we search a one pixel radius about $proj_{f'}(\mathbf{p})$ and record any silhouette IDs. If none are found, we then continue the search along a single direction only, terminating when a silhouette pixel is found or a search distance threshold is exceeded. This is the scheme depicted in Figure 4.6. The idea is to choose a direction that will lead quickly to silhouette pixels. If $proj_{f'}(\mathbf{p})$ is near the silhouette then searching along the screen-space projection of ∇g at $\mathbf{p}$ will generally lead to a nearby silhouette where $g = 0$. In fact, if the surface is smooth near $\mathbf{p}$, we can even search along the screen-space projection of the normal, as we presented in [46]. In any case, $\mathbf{p}'$ is again computed by minimizing (4.3) on the silhouette loops found prior to search termination. The 8-bit length encoding component of the 32-bit ID color (Section 4.3.1) is used to accelerate the closest-point test on the loop.

While the closest-point correspondence algorithm will produce seemingly reasonable results, it suffers from a critical flaw. This is revealed by considering the sphere spinning about its vertical axis in Figure 4.7a, or an equivalent stationary sphere with camera orbiting in the equatorial plane. Samples along the silhouette in f will project to points along the elliptical paths shown in $f\,'$. The closest-point criteria will then propagate each sample to the nearest silhouette location $\mathbf{p}'$ along the radial direction. Even though the silhouette appears stationary in the image plane, over time this propagation scheme will cause parameterization to "pile up" at the poles of the sphere. This result runs counter to the expectation that parameterization will remain fixed on the apparently stationary silhouettes – a consequence of ignoring the effects of camera motion while determining 2D correspondences. Ideally, the features of a coherent stylization will track the visual flow of the annotated geometry from f to $f\,'$. Accordingly, the underlying parameterization should propagate in a fashion sympathetic with the associated camera motion from $\mathbf{c}$ to $\mathbf{c}\,'$:

$$\Delta\mathbf{c} = \mathbf{c}' - \mathbf{c} \tag{4.4}$$

The desired result of such a scheme is depicted in Figure 4.7b. Here, correspondences produce the anticipated stationary result, rather than the artifacts introduced by the arbitrary closest-point strategy.

We can implement this by adapting the solution to a related problem in computer vision – the reconstruction of unknown 3D surfaces from the motion of silhouette contours in the image plane [10]. In this context, the 3D position $\mathbf{p}$ of a point on the silhouette in f is known only in terms of its direction $\hat{\mathbf{v}}$ relative to the camera. From (4.1), we express its unknown depth as the magnitude of the camera vector:

$$\mathbf{p} \;=\; \mathbf{c} \;-\; \|\mathbf{v}\|\,\hat{\mathbf{v}} \tag{4.5}$$

As shown in Figure 4.8, this point on the silhouette is associated with a corresponding silhouette location $\hat{\mathbf{v}}\,'$ in the next frame by searching the image plane along its intersection with the *epipolar plane* defined by $\hat{\mathbf{v}}$ and the moving camera $\mathbf{c}$, $\Delta\mathbf{c}$. The correspondence $\hat{\mathbf{v}} \rightarrow \hat{\mathbf{v}}\,'$ can be used to determine the depth $\|\mathbf{v}\|$ of point $\mathbf{p}$.

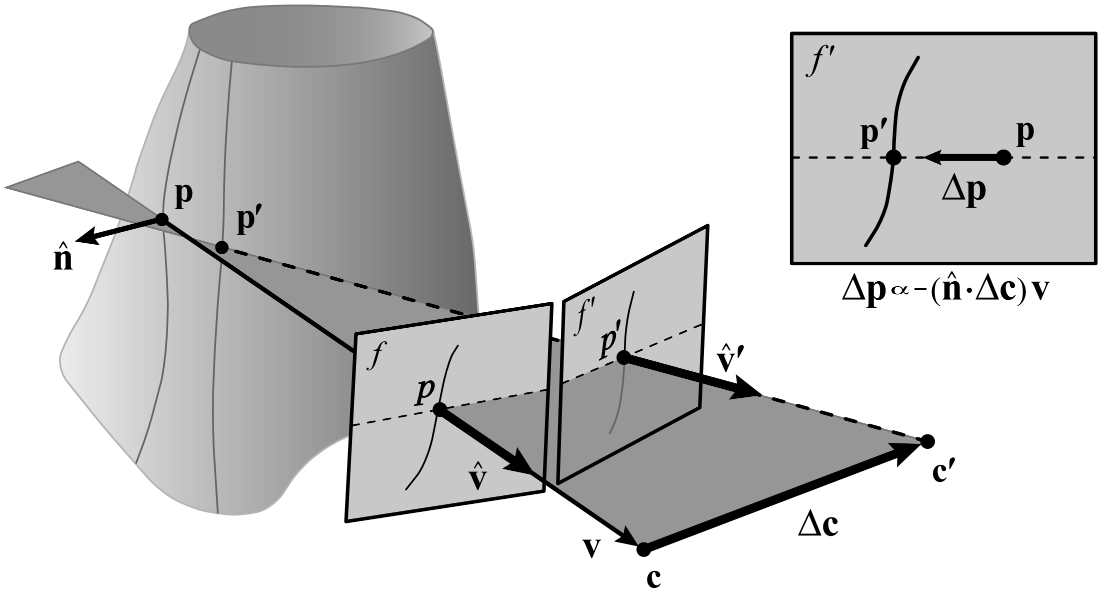


Figure 4.8: The epipolar plane containing $\mathbf{p}$, $\mathbf{c}$, and $\Delta\mathbf{c}$ defines corresponding points on the silhouette in nearby frames f and f' (after [8]). *Inset:* The direction $\Delta\mathbf{p}$ describes the appropriate search direction in the f' image plane.

The epipolar plane is a natural choice for establishing the correspondences because their velocity in the image will describe the position and curvature of the underlying geometry via simple expressions [9]. This is a direct consequence of defining the correspondences to lie in the same plane as the moving camera. Because of this consistency with $\Delta\mathbf{c}$, the same epipolar correspondences will produce the propagation scheme we seek in Figure 4.7b.

To adapt this approach to our correspondence problem $proj_{f'}(\mathbf{p}) \rightarrow proj_{f'}(\mathbf{p}')$, we simply modify the closest-point correspondence scheme by constraining the result to lie on the epipolar plane. This defines *epipolar correspondence* – for each $\mathbf{p}$ the corresponding $\mathbf{p}'$ is taken as the location on a visible silhouette path in f' that lies on the epipolar plane $(\,\mathbf{p}\,,\mathbf{c}\,,\Delta\mathbf{c}\,)$ and minimizes the closest-point criterion (4.3).

Instead of brute force, we again exploit the ID image to determine this efficiently. We call this procedure *accelerated epipolar correspondence*, as it bears similarity to the accelerated closest-point correspondence scheme described previously. Each sample $\mathbf{p}$ is first projected to its location in the ID image of f' as shown in the inset of

Figure 4.8. The intersection of the epipolar plane with the image plane will yield an *epipolar line* through $proj_{f'}(\mathbf{p})$. As before, we search within a one pixel radius about this projection and record any silhouette IDs. If none are found, this time we continue the search along the epipolar line, terminating when a silhouette pixel is found, or a search distance threshold is exceeded. Finally, $\mathbf{p}'$ is chosen as the particular intersection of the epipolar plane amongst the encountered silhouette loops that minimizes (4.3). It remains only to determine which direction to search along the epipolar line. For an infinitesimal camera displacement $\delta\mathbf{c}$, one can show (Appendix B) that a point $\mathbf{p}$ on the silhouette in frame f will "slip" along the surface in the epipolar plane by:

$$\delta\mathbf{p} \;=\; -\frac{\hat{\mathbf{n}} \cdot \delta\mathbf{c}}{\kappa_r \, \|\mathbf{v}\|^2}\,\mathbf{v} \tag{4.6}$$

where κ_r is the *radial curvature* [19, 48] of the surface at $\mathbf{p}$. This result is consistent with our expectations for propagation on the spinning sphere in Figure 4.7b – the velocity of samples is maximum at the equator and diminishes to zero at the poles (or *frontier points*) where $\delta\mathbf{c}$ lies in the tangent plane. Because the velocity is inversely proportional to curvature, it will also tend to zero for the limiting case of sharp creases – consistent with our assertion that $\mathbf{p}' = \mathbf{p}$ for static feature lines. Finally, we observe that $\kappa_r > 0$ for all points on visible silhouettes (equivalent to the $\nabla g \cdot \mathbf{v} > 0$ requirement described in Section 4.3.1). Therefore, for some camera motion $\Delta\mathbf{c}$ equation (4.6) can be reduced to a simpler form that preserves its direction:

$$\Delta\mathbf{p} \;\propto\; -\left(\hat{\mathbf{n}} \cdot \Delta\mathbf{c}\right)\,\mathbf{v} \tag{4.7}$$

This disambiguates the direction from $proj_{f'}(\mathbf{p})$ along which to search the epipolar line, as shown in the inset of Figure 4.8. In practice, we evaluate this in object space to account for rigid body animation. See equation (B.12) in Appendix B for a version of (4.6) which also accounts for mesh deformation under animation.

While there is no “correct” way to propagate parameterization, we prefer the more principled epipolar approach – it yields the expected result for test cases, and reflects visual flow – over the more ad-hoc closest-point method. We find that a search threshold of 1% of image size (e.g. $\sim$6 pixels at 640×480) is generous enough for most samples to successfully propagate in typical animations. Once mesh deformation or camera motion become too exteme to propagate samples with this threshold, we have observed that any coherence, or lack thereof, is imperceptible anyway.

Finally, for each sample that successfully finds a correspondence $\mathbf{p}'$ along some silhouette path P, we determine the arc-length parameter s of this location along the path. The sample is now recorded as a parameterization *vote* $(\, s\, ,\, b\, ,\, t\,)$ on P. The next two sections describe how these votes are used to generate and then parameterize brush paths along the new silhouette paths of $f\,'$.

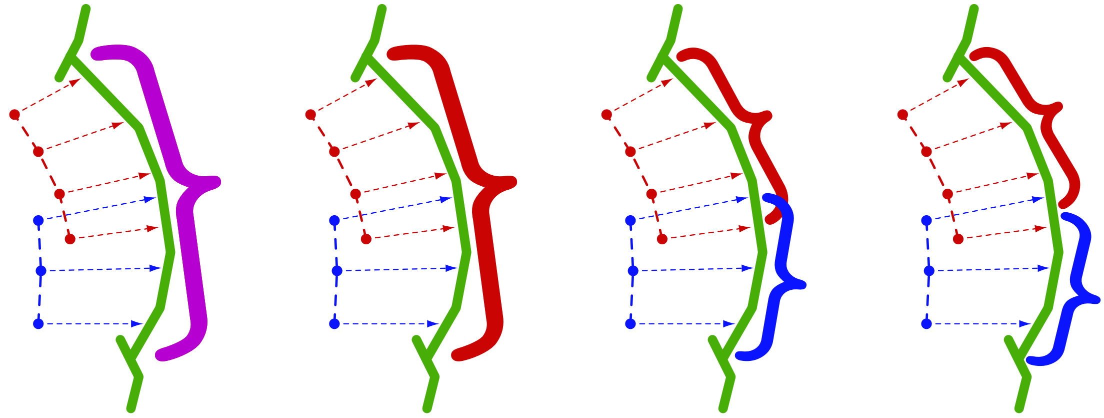
4.4.4 Brush Path Generation

We generate one or more brush paths along each silhouette path P by taking into account the votes it receives. This is accomplished by sorting the votes registered with P into *vote groups* $\{(s_i, t_i)\}_b$ based on their brush path ID. Each vote group will potentially lead us to create a corresponding brush path along its silhouette path.

Under reasonable animation, most of the votes from a given brush path will reach a single silhouette path P en masse – evenly spaced, and in the original order they were generated. We call these *coherent vote groups.* Whenever such a group is found, we may confidently generate a new brush path in its corresponding region on P. But, in general, the votes from a given brush path may propagate to multiple silhouette paths, land in regions already populated by votes from other brush paths, and even arrive with anomalous ordering or spacing. First, we will describe a procedure that can “clean up” these anomalies to yield a collection of coherent vote groups. We then present several strategies for generating brush paths when multiple coherent vote groups arrive on P.

To test the coherence of a vote group, we sort the votes by increasing arc-length position s along P. A coherent vote group $\{(s_i, t_i)\}_b$ can now be defined as a sequence of votes along P that (1) are evenly spaced similar to the sampling distance, and (2) correspond 1-to-1 with a sub-sequence of samples from b. We generally find that few vote groups actually meet this strict criteria, even under coherent animation. For instance, the sudden appearance or disappearance of a silhouette path, or some other occlusion can cause gaps to appear in vote groups. Another common artifact involves locations where the end-point of one silhouette path lies along the length of another. Often, small fragmentary vote groups and corresponding gaps will be produced at these points when samples are traded between them. And, if the same effect occurs where a silhouette loops back upon itself, the same interchange of samples can cause votes within a group to arrive out of order. Nevertheless, we find that these artifacts generally comprise only a few votes amongst otherwise coherent vote groups. Therefore, we present a strategy for extracting the coherent sequences from these "mostly" coherent vote groups.

Rather than seek to eliminate outliers (which we have found to be difficult to achieve robustly), we employ an aggressive procedure that simply splits each vote group between pairs of votes that violate (1) or (2) above, and then chooses the coherent vote groups from the resulting candidates. To meet property (1), each vote group is split wherever vote spacing $s_{i+1} - s_i$ deviates significantly from its mean, and resulting groups are retained if their mean spacing is similar to the sampling distance. To meet (2), we split each remaining vote group wherever samples appear out of order, i.e. $t_{i+1} < t_i$ (since parameterization increases monotonically, as described in the next section). While the remaining vote groups all conform to our strict coherence criteria, many of them will consist of only one or two spurious votes excised by the splitting procedures. To retain only the meaningful groups, we discard those that contain fewer than 3 votes.



(a) mixed (b) majority (c) 1-*to*-1 (d) trimmed

Figure 4.9: Four different policies for generating brush paths on a silhouette path given the same two vote groups.

This procedure will err on the side of discarding useful votes that might otherwise be recovered by more rigorous data mining efforts. But, in practice, we find that the sheer number of samples (see Figure 4.6) makes this unnecessary. In regions where propagation artifacts lead us to discard votes, the parameterization schemes in the next section will extrapolate the missing information from the ample remainder. Furthermore, the aggressive splitting of vote groups will be repaired by a "healing" process also described in Section 4.4.5.

With only coherent vote groups remaining, we now assign their corresponding brush paths. If fewer than two vote groups survive on P, we simply assign a single brush path. However, when there are two or more surviving vote groups, we consider several policies for covering P with brush paths (Figure 4.9):

Mixed. We generate a single brush path along P and parameterize it by mixing information from *all* the votes. This is similar to the the policy described by Bourdev [4] that parameterizes the silhouette paths directly. One problem with this policy is that it averages together data from different sources in a way that is not meaningful, in effect throwing away coherence information. It also exhibits "popping" artifacts whenever it assigns a single parameterization to disagreeing vote groups as we describe for the next policy.

Majority. This method still assigns a single parameterized brush path to P, but avoids mixing together votes from disparate sources. Each vote group is assigned a confidence based on its size. The vote group with the highest confidence is then used to parameterize the entire brush path, while the remaining votes are discarded. This will exhibit coherent behavior across a majority of P, but still yields poor temporal coherence. For example, in Figure 4.9b, all of P is parameterized consistently with the "red" vote group. The parameterization in the "blue" region is extrapolated from the red votes, and, unless both vote groups chance to describe a similar parameterization, a visual "pop" will occur wherever the red parameterization replaces the blue.

1-*to*-1. We can overcome this drawback by generating one brush path for each vote group. If any part of P remains uncovered, the nearest brush paths are extended to cover the gap. This approach retains all of the coherence information, and also avoids averaging together any unrelated data. While coherent, this approach can lead to cumulative fragmentation over time. A profusion of short brush paths covering P can overlap arbitrarily, leading to an accumulation of visual "clutter." This will impact upon the final æsthetic of the stylization in an uncontrollable fashion. In practice, we find it more effective to prohibit brush paths from overlapping, as described in the next policy. This way, the artist can control any overlapping effects by describing them intentionally in the stylization prototype (Section 4.3.2).

Trimmed 1-*to*-1. We favor this policy which eliminates the overlaps of 1-*to*-1. Once again, each vote group is assigned a confidence based on its size. We then generate brush paths as in 1-*to*-1 coverage, but trim away overlapping portions, leaving only those of highest confidence. While this does avoid visual clutter, the approach still suffers from cumulative fragmentation over time. However, we can counteract this effect by merging abutting brush paths with similar parameterizations. We promote this via the "healing" process described at the end of the next section.

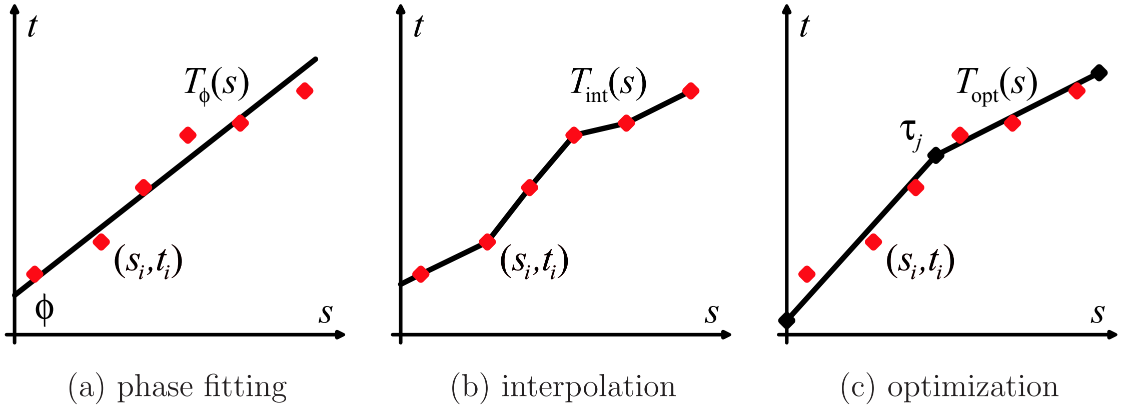
4.4.5 Brush Path Parameterization

Once the brush paths are generated, it remains only to parameterize them. Each new brush path b' will have an associated vote group $\{(s_i, t_i)\}_b$ that describes a series of parameterization samples t_i at their respective arc-length positions s_i along P. If these are plotted as in Figure 4.10, the task of assigning coherent parameterization in f' clearly boils down to fitting some function $T(s)$ to the samples from f. It now remains only to determine an appropriate objective function to define the fit.

As first noted by Bourdev [4], there are competing objectives when assigning a coherent parameterization: 2D arc-length consistency vs. tracking of the 3D shape. The relative priority of these objectives depends upon æsthetic factors. Screen-space effects (like dotted lines, Figure 4.11d) generally benefit from a parameterization consistent with 2D arc-length. On the other hand, stylizations depicting apparent 3D details (like cactus thorns, Figure 4.11a) behave more reasonably when parameterization "sticks" to the 3D shape. Shortly, we examine both these objectives in detail and develop specific fitting schemes to achieve them. Finally, we conclude with a more flexible method that can optimize for these (and other) objectives at the discretion of the designer.

But, first we must consider how to parameterize the brush paths that receive no votes. This, of course, will be the case for all brush paths in the first frame of an animation. But it also occurs sporadically when vote groups fail to meet our coherence criteria (Section 4.4.4), or fail to propagate entirely due to extreme view changes. In this case, we are again faced with the 2D arc-length vs. 3D shape tracking trade-off. Accordingly, we provide the designer with two choices: arc-length parameterization $T(s) = s$, or *scaled* arc-length parameterization $T(s) = \rho s$.

While the former policy is obvious, the latter deserves further explanation. Scaled arc-length has already been described in Section 4.3.4 as a reasonable policy when no coherent propagation framework is available. In brief, the scale factor $\sigma_m = 1/\rho$



(a) phase fitting (b) interpolation (c) optimization

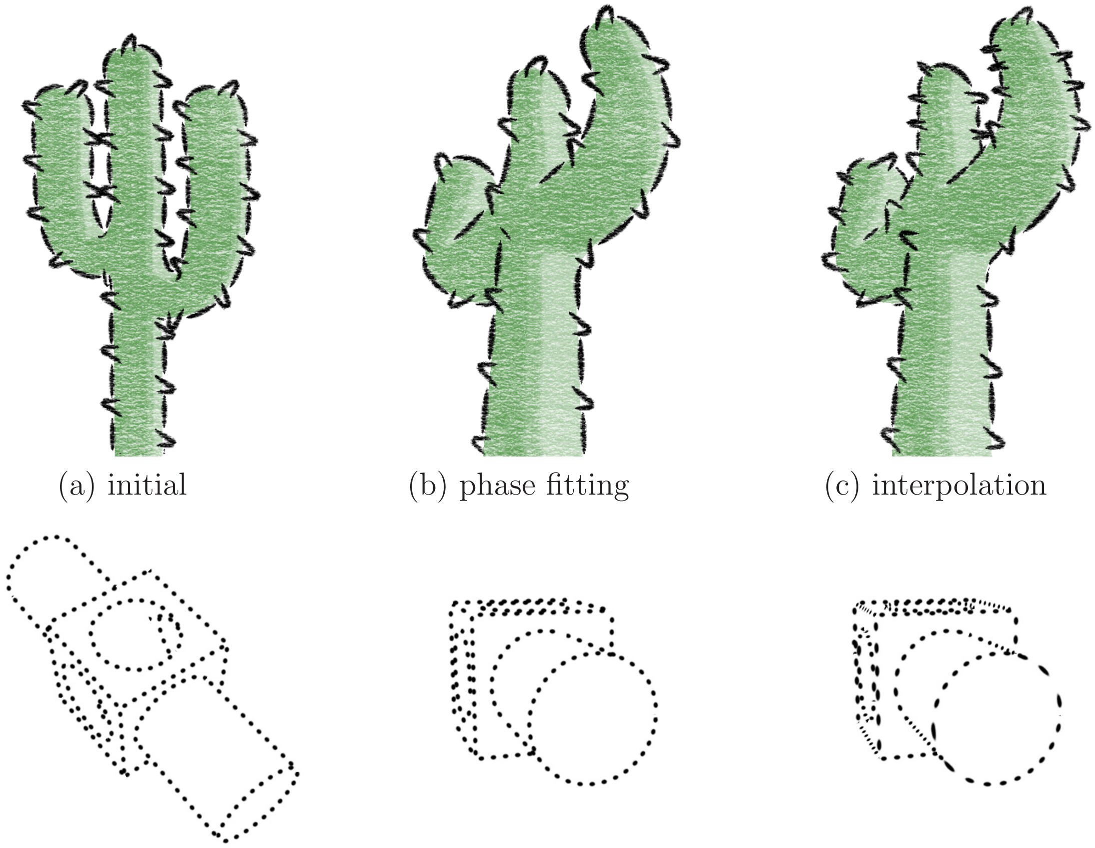
Figure 4.10: Three schemes for assigning a parameterization $T(s)$ to a brush path given the vote group $\{(s_i, t_i)\}_b$.

is the ratio of a mesh's current size to its initial size when stylization is first applied. We measure this size as the screen-space circumference of the bounding sphere. So, for example, if the cactus in Figure 4.11a doubles in size on the screen, we expect its outline to double in perimeter. Accordingly, $\rho = 0.5$ will stretch the parameterization to double the thorn spacing, consistent with their interpretation as 3D detail.

We now move on to describe means for parameterizing a brush path when votes are present. The first two methods strive to meet one of the competing goals described above, while the third provides a means to balance between multiple objectives.

Phase fitting (Figure 4.10a): This method computes a uniform 2D arc-length parameterization $T_\phi(s) = \rho s + \phi$. Least-squares minimization will yield the phase ϕ that best fits the votes. The scheme is well-suited to a 2D style like dotted-lines if we take $\rho = 1$. At all times, it will produce a parameterization that maintains constant dot spacing while maximizing coherence with the previous frame (Figure 4.11e).

We can attempt to adapt this policy to stylizations that convey 3D detail by taking $\rho = 1/\sigma_m$. Now, a stylization like that in Figure 4.11a will behave reasonably under panning and zooming. Changes in foreshortening, however, lead to an inconsistent "swimming" effect. As shown in Figure 4.11b, instead of remaining relatively fixed on the outline, the spines flow around the arms of the cactus.


(a) initial (b) phase fitting (c) interpolation

(d) initial (e) phase fitting (f) interpolation

Figure 4.11: The appropriate parameterization fitting scheme depends upon semantics of the stylization. *Top row:* 3D styles require the tracking behavior shown in (c). *Bottom row:* 2D stylizations require the arc-length uniformity shown in (e).

Interpolation (Figure 4.10b): In Section 4.2.4 we found that the 3D arc-length parameterization of static feature lines will cause stylization to track the 3D shape. However, we cannot appeal to 3D arc-length for view-dependent contours. Instead, this method promotes coherence on the 3D shape by interpolating the votes – i.e., $T_{int}(s)$ is a polyline connecting the samples $\{(s_i, t_i)\}_b$. Under this policy, the cactus spines in Figure 4.11c appear fixed on the model as expected. This is a direct consequence of our propagation scheme (Section 4.4.3) that causes samples to track the evolving silhouettes from frame-to-frame. Obviously, the effect is not applicable for styles that are mostly 2D in nature. This is demonstrated by dotted-lines in Figure 4.11f that undergo undesirable distortion in dot spacing under foreshortening.

Optimization (Figure 4.10c): We could simply offer the user a choice between phase fitting or interpolation methods, but, in practice, we find this inadequate. Phase fitting is only useful for stylizations that exhibit a purely 2D character, such as the canonical dotted line. For these, the swimming effect is accepted because the dots must maintain equal spacing to yield the expected æsthetic. Many other apparent 2D effects, like the paint strokes in Figure 4.4c, do not require such a rigid screen-space uniformity. They benefit, instead, from some measure of 3D tracking to more closely match the incoming votes, and reduce swimming artifacts. On the other hand, strict interpolation is undesirable even for stylizations that exhibit strong 3D characteristics. The cactus thorns, for instance, will suffer cumulative distortion after many successive frames of animation. This will eventually lead to an arbitrarily dense "piling up" or "thinning out" of the stylization. An interpolation scheme that also promotes some degree of 2D arc-length consistency could relax these distortions.

In general, we find that all but the most extreme stylizations benefit from some particular balance between 2D arc-length uniformity and 3D shape tracking objectives. To this end, we propose a method that can balance these goals in a fashion controllable by the designer. We implement this optimal parameterization $T_{opt}(s)$ by minimizing a total energy defined as a weighted sum of energies corresponding to the different goals:

$$E = E_{votes} + \omega_\rho E_\rho + \omega_b E_{bend} + \omega_h E_{heal} \tag{4.8}$$

The first two terms of this energy correspond, respectively, to how well the parameterization (1) tracks the 3D shape by interpolating the votes, and (2) exhibits a uniform scaled 2D arc-length. We also include two additional terms that (3) measure the local distortion, and (4) promote a "healing" process that we describe shortly. The non-negative weights ω_ρ, ω_b, and ω_h permit the designer to balance between all these objectives with arbitrary relative priority.

We find that expressing the parameterization t as a piecewise-linear function of s is sufficient to fit the votes well without producing visual artifacts at tangent discontinuities. Thus, we take $T_{opt}(s)$ to be a polyline interpolating a vertex sequence $\{\sigma_j, \tau_j\}$, where the m knots σ_j are evenly-spaced arc-length parameters over the brush path.[1] Given the set of incoming votes and chosen weights, the optimization solves a system of equations to find τ_j's minimizing the total energy E. Because its terms are all linear by design, this can be solved efficiently at run-time – an important requirement given that arbitrary brush paths may populate any frame.

The first term of the energy equation (4.8) measures how well parameterization tracks the 3D shape. Analogous to interpolation fitting, this energy gauges how well the $T_{opt}(s)$ polyline interpolates the incoming votes. This is computed as a sum of square differences:

$$E_{votes} = \frac{1}{n} \sum_{i=1}^{n} [\, T_{opt}(s_i) - t_i \,]^2 \tag{4.9}$$

Because each term $T_{opt}(s_i)$ depends only upon the segment j of the polyline in to which s_i falls, it is linear in the two free variables τ_j and τ_{j+1}.

The second term measures how well the $T_{opt}(s)$ corresponds to a scaled 2D arc-length parameterization, identical to that described for phase fitting. This energy term is most easily expressed if we "normalize" the τ_j's by taking $\tilde{\tau}_j \equiv \tau_j - \rho\sigma_j$. Now, measuring how the τ_j's deviate from ρ-scaled arc-length is equivalent to measuring how the *normalized* $\tilde{\tau}_j$'s deviate from their average $\tilde{\tau}_{ave}$:

$$E_{\rho} = \frac{1}{m} \sum_{j=1}^{m} [\, \tilde{\tau}_{ave} - \tilde{\tau}_j \,]^2 \tag{4.10}$$

Depending upon the relative 2D or 3D character of the stylization, the designer may choose $\rho = 1$ or $1/\sigma_m$, respectively, as deemed appropriate.

[1] In practice, we find it effective to choose the knot spacing as a small multiple of the sampling distance – e.g., we sample brush paths at 6 pixel intervals and choose an 18 pixel knot spacing.

While the first two terms already encompass the competing objectives we have identified, two additional energies were also found to be particularly effective. First, it is desirable to provide some control over local distortion in the parameterization (i.e., bending of the $T_{opt}(s)$ polyline). While (4.10) already discourages bending, it does so by promoting a globally uniform 2D arc-length parameterization. For cases where a global arc-length consistency is not imperative, we can define a more local gauge of distortion. The third term implements this via a "thin-plate" energy that measures bending between adjacent polyline segments:

$$E_{bend} = \frac{1}{m} \sum_{j=1}^{m-2} \left[\tau_j - 2\tau_{j+1} + \tau_{j+2} \right]^2 \tag{4.11}$$

Lastly, as mentioned at the end of Section 4.4.4, two abutting brush paths sharing a silhouette path may be merged when their parameter discontinuity is small. We would like this *healing* to happen often to combat cumulative fragmentation due to aggressive vote group culling, and the natural evolution of the silhouette paths. Thus, whenever the discontinuity is small between brush paths, but not yet small enough to merge them, we coerce the parameterizations on either side of the break to approach their average t_{ave}. This is accomplished by creating an additional *healing vote* (s_k, t_{ave}) at each abutting endpoint s_k of the brush paths. Like the vote-fitting equation (4.9), these healing votes contribute via least-squares differences (with 0, 1 or 2 terms):

$$E_{heal} = \sum_{k} \left[T_{opt}(s_k) - t_{ave} \right]^2 \tag{4.12}$$

To finally compute the optimal fit $T_{opt}(s)$, we minimize the total energy E by solving the system of equations given by $\partial E / \partial \tau_j = 0$. That can be accomplished efficiently using standard LU decomposition [69]. The system will be non-singular as long as there is at least one vote and ω_ρ is non-zero.

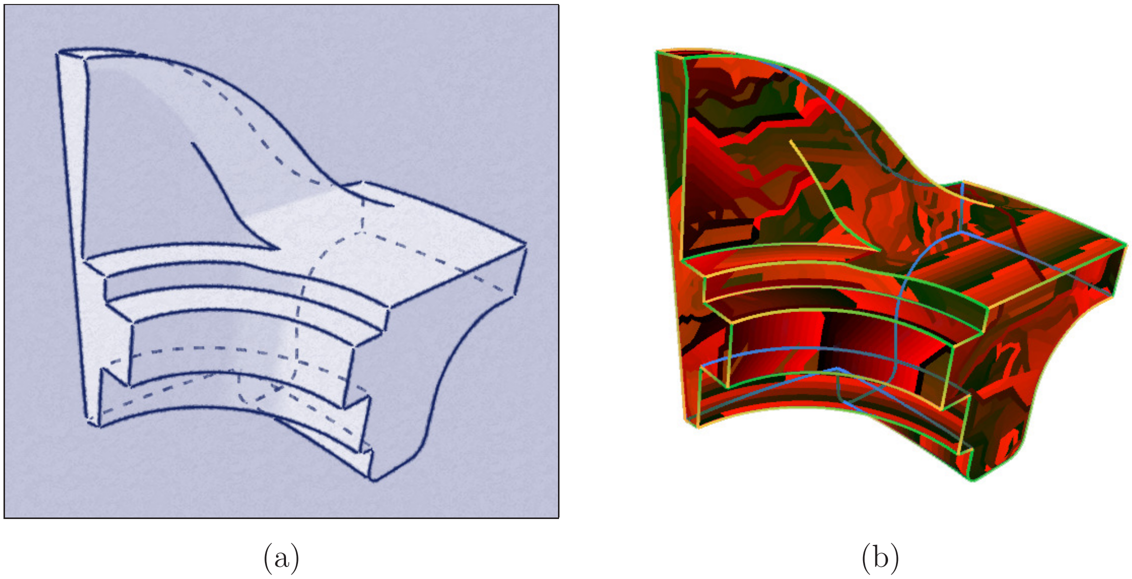


Figure 4.12: Extensions of the coherence framework: In (a) creases are easily handled, and in (b) the ID image reveals how the hidden lines are encoded.

4.4.6 Extensions

The coherence framework we have presented can be easily extended in several ways.

Beyond Silhouettes

While we have focused on silhouette contours in this section, the results extend to other types of lines. Static feature lines are trivially implemented, as their frame-to-frame correspondences are known *a priori*. In lieu of the rigid 2D or 3D arc-length parameterizations discussed in Section 4.2.4, the coherence framework can assign a parameterization that balances between these extremes with any arbitrary priority.

This work can also extend to other view-dependent contours. DeCarlo et al. [18] recently analyzed the dynamics of suggestive contours under camera motion and offer culling methods to reject those exhibiting poor temporal coherence. Therefore, these contours are also amenable to correspondence tracking in the image-plane. It remains only to derive expressions for their velocity in the epipolar plane, analogous to equations (4.6) and (4.7) for silhouette contours (see Appendix B).

Pass	Objects	Components	ID Color	z-Rd	z-Wrt
1.	see-thru	mesh triangles	normal	✓	✓
2.	see-thru	hidden lines	hidden	X	X
3.	see-thru	all lines	normal	✓	X
4.	opaque	mesh triangles	normal	✓	✓
5.	opaque	all lines	normal	✓	X

Table 4.1: Rendering the ID image in two stages to encode hidden lines.

Beyond Visible Lines

It is possible to extend this framework to hidden lines by adapting a frame-buffer trick described by Rossignac et al. [73]. They render silhouettes on the GPU using `GL_LINES` in two passes. The first pass disables z-buffer testing so that hidden lines will be rendered, then a second pass enables the test to render visible silhouettes in a different color. The second pass will overwrite the visible portions only, leaving behind the hidden lines in the first color.

We use the same technique in the ID image to render the hidden components of the lines with a different set of ID colors. Figure 4.12b shows the ID image corresponding to the stylized rendering on its left. As in Figure 4.6, mesh triangles appear in reds and visible lines appear in greens, but now their hidden portions are also visible in shades of blue. When the designer chooses to reveal and stylize hidden lines, the pipeline simply changes over to testing for the blue IDs during their visibility testing, and sample propagation phases.

We implement this in two stages (Table 4.1), first rendering the *see-thru* objects (those that reveal hidden lines), and then rendering the remaining *opaque* objects and their lines. By drawing the opaque objects in the final pass, they have the opportunity to occlude the hidden lines of see-thru objects. The second stage (passes 4 and 5) are identical to the original ID image rendering passes (1 and 2, Table 3.1, Page 24).

Beyond Rubber-Stamping

The choice to rubber-stamp a single prototype along all silhouettes in Sections 4.3.2 and 4.3.4, was made to reduce the burden of determining correspondences for these view-dependent contours. However, the sample propagation stage of the coherence pipeline reestablishes these correspondences. With this framework, it is possible to assign a unique stylization prototype to each brush path.

When brush paths are created in f', any silhouette path that receives no votes from f is assigned a single brush path with default parameterization. A prototype synthesis scheme, like Markov synthesis (Section 4.2.2), could also assign a unique prototype π to each new brush path b. Instead of sketching a single prototype to rubber-stamp, the designer can now sketch a collection of prototypes to seed the generator. We then modify the coherence pipeline to propagate the unique prototype π along with the parameterization samples as follows:

1. Each *new* brush path b is assigned unique prototype $\rightarrow \pi$
2. Samples along b now include its prototype $\rightarrow (\mathbf{p}, b, \pi, t)$
3. Corresponding votes also carry the prototype $\rightarrow (s, b, \pi, t)$
4. Votes groups from b each share a unique $\pi \rightarrow \{(s_i, t_i)\}_{b,\pi}$
5. Each *propagated* brush path b' receives unique prototype $\rightarrow \pi$

Now, brush paths at time-zero, and those that receive no votes, are assigned unique stylizations retained by corresponding brush paths in future frames. A given stylization prototype will persist in the image until all brush paths that carry it fail to propagate samples to any visible silhouette paths.

Beyond $\mathbf{E}_{\mathbf{votes}}$, $\mathbf{E}_{\rho}$, $\mathbf{E}_{\mathbf{bend}}$, $\mathbf{E}_{\mathbf{heal}}$

One strength of this coherence framework is the ease with which additional coherence goals may be integrated into the total energy expression. The four terms we have presented in Jot cover the primary coherence objectives, but more sophisticated terms might be conceived. For instance, one might formulate terms that account for how extreme the view change is between f and f'. During small changes coherence could enforce 3D tracking, while larger changes could favor arc-length to relieve distortion. Such an approach provides a higher degree of correspondence when it is most perceptible, and takes advantage of extreme view changes to hide the swimming that accompanies relaxing distortion. A more complete WYSIWYG NPR system might offer the designer control over these and other potentially useful energy terms.

4.5 Discussion

Summary of Line Effects

In this chapter, various aspects of WYSIWYG NPR tool development for line stylization effects were explored (Table 2.4.2, page 15). We demonstrated the means for designers to annotate silhouettes and feature lines by directly sketching over them with stylized strokes. Additional tools synthesize annotations automatically from a small set of examples, simplifying the task of stylizing complex scenes. Coupled with a flexible stroke primitive and media simulation, these tools provide the artist with a wide range of styles, and the hands-on means to impart one's own personal æsthetic. Some diverse examples can be found in Figures 4.4 (page 45) and 6.1 (page 95).

Once annotated, the 3D scene can be rendered automatically from novel views. The system assumes the task of maintaining the artist's original annotations, while dynamically adapting properties such as stroke widths and offsets to changing level of detail. The user can tune the underlying parameterization of feature lines and silhouettes to exhibit a spectrum of dynamics, ranging from a purely 2D æsthetic, to one that tracks the 3D shape. And, for lines that are annotated in multiple passes, the stylization can animate cyclically, producing a pseudo-random Squigglevision™ effect (Figure 6.4, page 99).

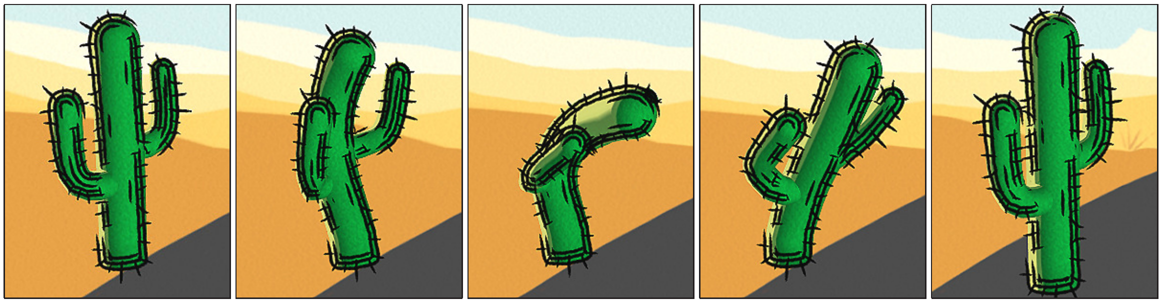


Figure 4.13: The designer annotate only the first frame of this animation. The system handles the tedious task of maintaining the thorny stylization in the remainder.

We ensure that all of these tools function effectively in an interactive setting, providing the designer with a responsive, hands-on environment. By employing a stochastic silhouette extraction scheme, an ID image for visibility and sample propagation, and exploiting the GPU to render our media simulation, we achieve the necessary real-time performance rates. Furthermore, the stylizations applied to silhouette contours and feature lines maintain their frame-to-frame coherence under real-time scene navigation, as well as during the playback of scripted animations.

Because of their view-dependence, achieving this coherence for stylized silhouettes was particularly challenging. To this end, we present a new, robust framework which can accomplish this for both view-dependent contours and static feature lines. Our approach can optimize for multiple coherence objectives, striking a particular balance appropriate to a given stylization as chosen by the designer. To our knowledge, this framework is the first system to provide effective, controllable coherence for stylized silhouettes, either in real-time or for off-line animation.

Without coherent parameterization, the designer cannot use strokes that wiggle, taper, or vary in color along their length – these risk introducing coherence artifacts during animation. This is true even of hand generated cell-animation, which nearly always exhibits such uniform outlines. With these WYSIWYG tools, the artist can now explore these previously inaccessible styles, while the system assumes the tedious or intractable task of ensuring coherence (Figures 4.13 and 6.7, page 99).

Future Work

Localized Stylization

In Section 4.2.1 we describe how to stylize static feature lines locally – each time a static feature is annotated by the designer, a unique prototype is unambiguously associated with it. However, this type of localization is not possible with view-dependent contours. Thus far, our approach can only localize the stylization of dynamic contours up to a given patch of surface. On any patch each different line type (e.g., hidden silhouettes, visible suggestive contours, etc.) may be assigned a unique stylization. But as the view changes and contours cross patch boundaries, we currently just break them at the patch interface. An important open question concerns how moving contours might transition smoothly between unique stylizations at these boundaries. Another open problem involves how these patches are defined in the first place. Presently, we generate them using traditional modeling software, but ultimately we imagine a WYSIWYG approach whereby the designer's own localized annotations give rise to corresponding patches of stylization.

Level of Detail

At present, our system can diminish stylization detail by shrinking offsets and stroke widths as an object recedes from the camera. But under these conditions, human-crafted illustrations typically also omit lines to reduce the image complexity. An interesting question is how to omit or even merge strokes in such scenes automatically, and to do so with temporal coherence as the camera zooms in or out. The same mechanism could be useful for meshes that contain many small surface details, leading to silhouettes that are fragmented into many tiny loops. The resulting paths could benefit from simplification into longer, smoother connected paths in screen-space.

While automatic level of detail systems are useful, determining appropriate details to omit is often a judgment best left with the designer. Therefore, it is also worth pursuing systems that enable the artist to customize this behavior, perhaps by directly annotating the silhouettes at various levels of detail. Once again, this raises the open question of how a contour can smoothly transition from one stylization to another.

Offline-Coherence

The coherence framework presented in this chapter propagates information between pairs of consecutive frames f and f' in a time-forward fashion. This design choice was made to fit the real-time constraints imposed on rendering in interactive systems. However, the off-line rendering of scripted animations can also benefit significantly from this work. In this case, we are at liberty to propagate coherence information both forward and backward in time. Future work might investigate optimizations over entire sequences of frames, rather than just consecutive isolated pairs.

These global optimizations can arrange for stylization to evolve in a fashion that improves coherence by “anticipating” potential coherence breaking events in future frames. Furthermore, global optimization could also permit the designer to “key-frame” the stylization – an essential feature for production-level NPR animation tools.

Chapter 5

Surface Effects

5.1 Introduction

To complement our methods for stylizing the existing outlines of objects, this chapter presents a set of tools for generating stroke-based effects on their surfaces. With these tools, the artist can create groups of strokes to elicit various surface properties ranging from shading effects to more detail oriented markings. In the example session from Section 2.2 (seen in Figure 5.1b) an artist demonstrates shading effects using cross-hatching, and employs detail markings to depict a label. These tools can be used to achieve a wide range of styles – narrow brushes can generate detailed line drawings (Figure 5.1d), while many wide strokes produce richly painted surfaces (Figure 5.1a), where just a few lines craft more minimal æsthetics (Figure 5.1c).

Various researchers have explored algorithms to produce these types of line-art and painterly styles automatically from 2D image input [16, 33, 36, 74, 79, 80]. Other work has explored how to extend these techniques to video image sequences by tracking visual flow to achieve coherence between frames [39, 53]. However, these approaches assume no *a priori* knowledge about the underlying 3D geometry. With this additional 3D information, Meier demonstrates that coherence is easily enforced by attaching stroke rendering primitives to the underlying surfaces during animation [61]. Recent research has even explored how to implement this in real-time settings [13, 83].

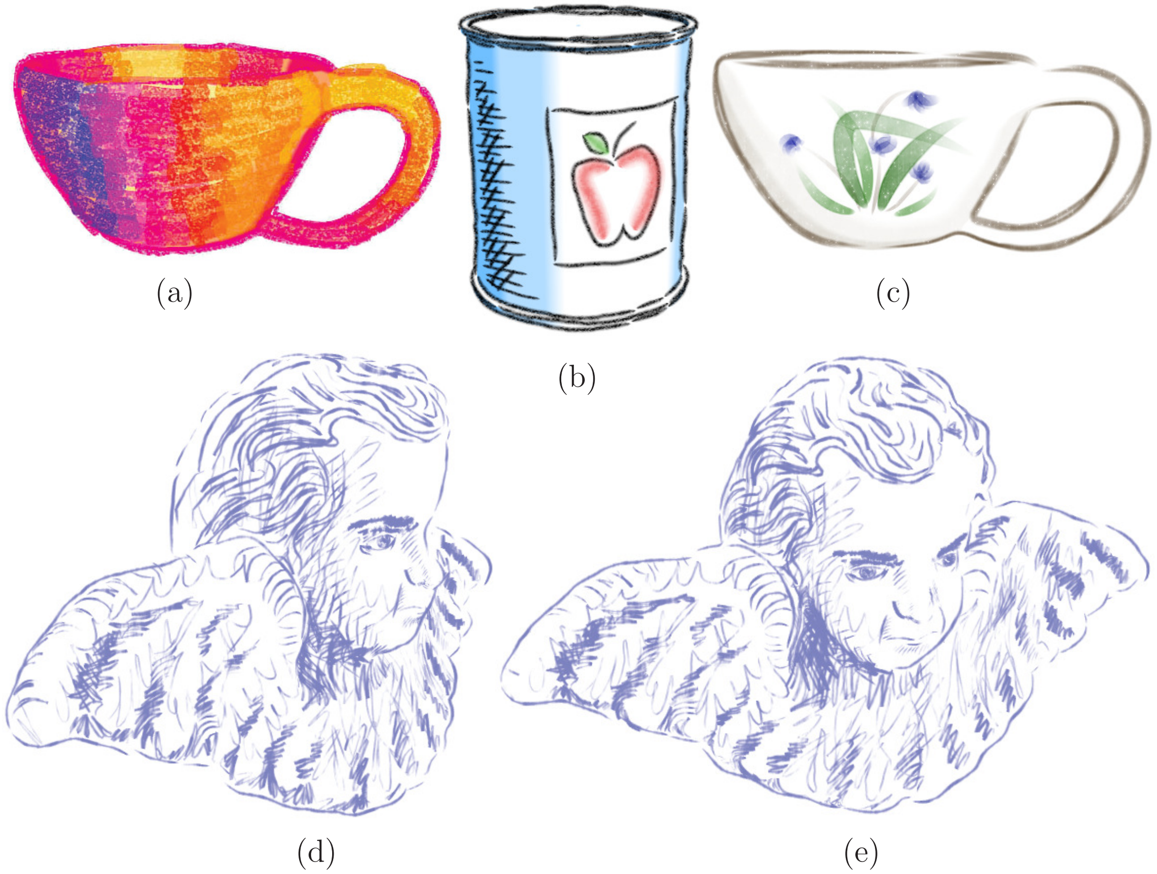


Figure 5.1: Strokes on surfaces achieve various effects: (a) wide brushes stroke create a richly painted surface, (b) hatching conveys shading and details provide a label, (c) a few brush strokes depict a flower, (d) thin strokes create a line drawing revealing surface shape and hair details, and (e) the system can automatically render new views.

All of these systems derive the actual placement, appearance and behavior of strokes from hands-off image processing methods. Meier, for instance, randomly populates surfaces with strokes and employs scripting to describe how they are influenced by the depth buffer, normal maps, and color reference images generated from the underlying 3D geometry.

To bring these tools into the WYSIWYG NPR domain we follow Meier's approach, but implement hands-on tools for the designer to describe the stroke-based effects. Instead of using scripts and algorithms, the user describes the placement of strokes by drawing them interactively onto the 3D surface. We also provide associated controls to describe their dynamic behavior during real-time navigation and animation. This approach follows a similar one to Disney's Deep Canvas system[1] which was used to create painterly effects on dynamic 3D backgrounds in the movie Tarzan, as briefly described by Daniels [17].

In our work, we classify these strokes based upon the type of information they convey about the underlying surface. Groups of strokes which achieve a broad range of different surface detail effects are described first in Section 5.2. We consider the more specific case of tonal stroke groups, like hatching, in Section 5.3. This distinction will impact the dynamic behavior exhibited by the strokes under changing viewing and lighting conditions.

5.2 Detail Marks

The most straightforward surface annotations in Jot are details marks. An artist can use these to depict a myriad of effects as shown in Figure 5.1 – except for the cross-hatching on the apple sauce tin in (b), all of these effects are achieved using groups of detail strokes.

5.2.1 Annotating Surfaces

Our hands-on approach to annotating surfaces with strokes was originally inspired by the WYSIWYG texture painting system of Hanrahan and Haeberli [34]. In their work, the artist's strokes are projected into texture maps on parameterized 3D surfaces. In contrast, we project strokes onto 3D objects to create spline curves that are ultimately rendered using the stroke primitive in Chapter 3. Furthermore, we do not require a parameterization of the surface. Each control point is simply represented as a barycentric location on a particular mesh triangle. When the mesh deforms in animation, the control points track the surface in the same fashion as a texture map.

[1] Some details can be found in the SIGGRAPH sketch [17], and filed with U.S. pat. no. 6,268,865.

As with line stylization effects of Chapter 4, the designer first chooses a particular brush style, and then draws strokes into the 3D scene using the tablet interface. Strokes are associated with the surface under the tablet pen at the beginning of the stroke, and are clipped to that surface for the remainder of the gesture. This permits the designer to sketch near the edge of an object without concern for strokes landing on background geometry. While annotating surfaces, the artist can freely navigate the scene, zooming in and rotating about to ensure that the scene is “interesting” from various distances and viewpoints, including the “backside.”

Collections of strokes that share brush styles and work in concert to achieve some æsthetic effect can be grouped together. During sketching, a *double-tap* gesture with the tablet pen is used to indicate that a group of strokes has been completed. In the next two sections, we demonstrate how dynamic properties of stroke are controlled by the designer on a per-group basis.

5.2.2 Dynamic Stroke Behavior

Obviously, our individually rendered strokes do not suffer the blurring or aliasing artifacts of those in conventional texture maps. But more significant differences become apparent when these strokes are examined during dynamic viewing conditions. As shown in Figure 5.2, the strokes can maintain their image-space width when rendered from oblique perspectives. Furthermore, the width can be modulated independently from the underlying surface. In the example, the strokes narrow in the distance, but grow only minimally under magnification. In this section we describe the designer’s controls for specifying the dynamic behavior of stroke widths under both changing magnification and foreshortening.

Recall from Figure 3.1a (page 22) in Chapter 3 that a stroke is represented by a list of spline control points $\{\mathbf{q}_i\}$ called the base path. In each frame, the 3D stroke spline is rendered by tessellating it at a chosen screen-space resolution and generating a

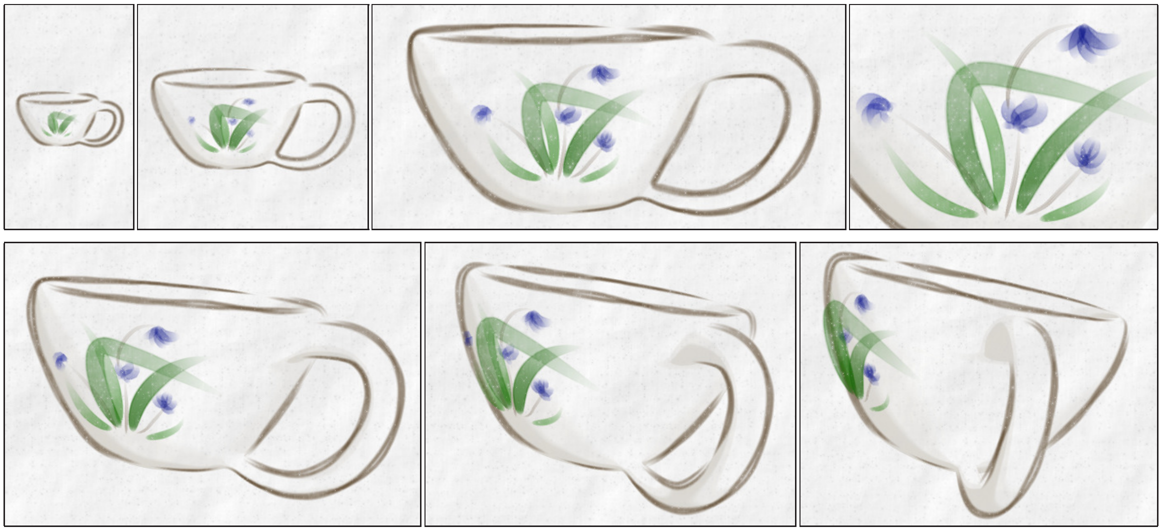


Figure 5.2: Detail strokes change in size independent of their underlying object. *Upper:* Stroke widths shrink under minification, but grow slowly in magnification. *Lower:* Stroke widths remain constant despite the foreshortening in oblique views.

triangle strip along it. Each tessellated point $\mathbf{q}$ is projected into the image plane, and a pair of vertices are generated, separated by w along the transverse directions $\pm\hat{\mathbf{w}}$. To control the dynamic behavior of strokes under magnification and foreshortening, we provide the designer with parameters that influence w and $\hat{\mathbf{w}}$, respectively.

We first consider the effects of changing magnification. To measure the degree to which the object has changed in size, we proceed as in Section 4.3.3 using the ratio σ_m of the object's current image size to its size when the stroke group was completed. If strokes maintain their original brush width defined at $\sigma_m = 1$ in all other views $\sigma_m \neq 1$, then they will exhibit an æsthetic consistent with "real" brushes of fixed size in 2D image plane. However, this will cause details to "pile up" eventually as the object recedes from the camera. This can be avoided by scaling widths in proportion to σ_m, achieving an æsthetic more consistent with 3D details. Unfortunately, now the strokes can grow unbounded under magnification.

We might consider a hybrid factor like $\texttt{max}(\sigma_m, 1)$, but, in practice, we find that scaling strictly by 1 or σ_m is simply too rigid – strokes appear to behave most naturally when they strike some balance between these extremes. Since this trade-off generally depends upon semantic aspects of the stylization, and even changes between minification and magnification, we leave this policy up to the designer to adjust appropriately. For each stroke group, we provide the parameters β^-, $\beta^+ \in [0,1]$ which balance between 1 and σ_m. These describe the scale factor ρ_w that modulates stroke width as magnification changes:

$$\rho_w = \begin{cases} \beta^- + \ (1-\beta^-)\,\sigma_m & \text{if } \ \sigma_m < 1, \\ \beta^+ + \ (1-\beta^+)\,\sigma_m & \text{if } \ \sigma_m \geq 1. \end{cases} \tag{5.1}$$

In Figure 5.2, the artist chooses a $\beta^- \sim 0.9$ causing strokes to shrink slightly more slowly than real 3D features – an effect that we find avoids "pile ups" while suggesting a degree of "2D-ness." Similarly, a choice $\beta^+ \sim 0.1$ allows strokes to grow slowly without becoming absurdly large under magnification. A subtle effect is also achieved by setting β^- slightly larger for the central green stroke, causing this feature to dominate when the object is viewed distantly.

The other parameter under artist control influences how strokes behave when their underlying surface is viewed obliquely. Though we always render the 3D stroke path $\{\mathbf{q}_i\}$ by constructing our triangle strips in the 2D image plane, the particular choice of the 2D transverse direction $\hat{\mathbf{w}}$ at each point will lead to different dynamics. We define $\dot{\mathbf{q}}$ as the 3D tangent vector to the stroke spline at $\mathbf{q}$. A purely 2D æsthetic can be achieved with $\hat{\mathbf{w}}_{\text{2d}}$ (5.2), the perpendicular to the image projection of the stroke tangent. On the other hand, we can produce a 3D dynamic with $\hat{\mathbf{w}}_{\text{3d}}$ (5.3), the projection of the vector both perpendicular to the stroke and contained in the tangent plane. The approximate equalities indicate that these must be normalized:

$$\hat{\mathbf{w}}_{\text{2d}} \ \simeq \ \mathit{proj}_{\perp}(\dot{\mathbf{q}}) \tag{5.2}$$

$$\hat{\mathbf{w}}_{\text{3d}} \ \simeq \ \mathit{proj}\,(\dot{\mathbf{q}} \times \hat{\mathbf{n}}) \tag{5.3}$$

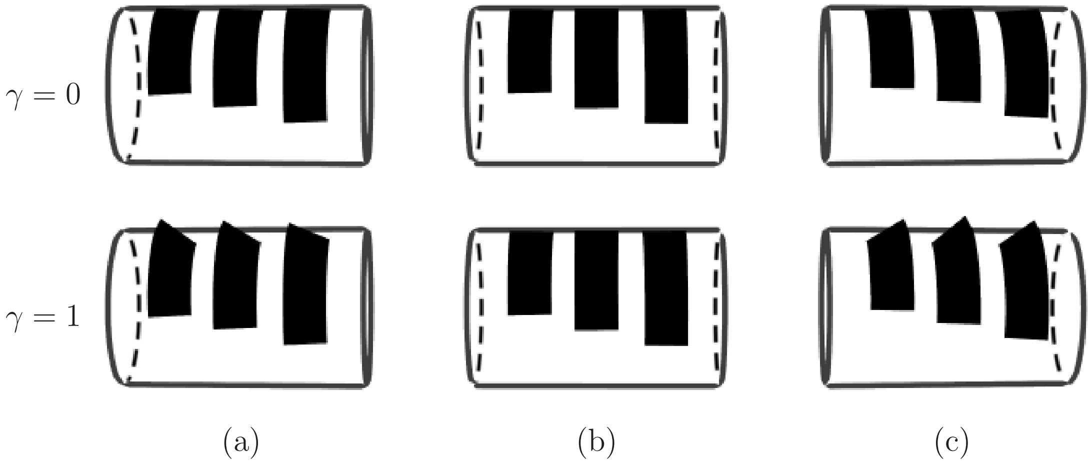


Figure 5.3: Three views of a cylinder annotated with wide strokes – exaggerated with un-stylized triangle strips as in Figure 3.1b. *Upper:* Strokes that are constructed in the surface's tangent plane are well-behaved in all regimes. *Lower:* Strokes that are constructed *purely* in screen-space are unstable when parallel to the silhouette.

For each stroke group, we again provide the designer with a parameter $\gamma \in [0, 1]$ which balances between these two extreme dynamics:

$$\hat{\mathbf{w}} \;\simeq\; \gamma\,\hat{\mathbf{w}}_{\text{2d}} \;+\; (1-\gamma)\,\hat{\mathbf{w}}_{\text{3d}} \tag{5.4}$$

While the designer is free to employ arbitrary γ, any choice $\gamma > 0$ can lead to artifacts near the silhouette. As shown in Figure 5.3, if the stroke tangent is nearly parallel to the camera direction $\dot{\mathbf{q}} \cdot \hat{\mathbf{v}} \sim 1$, then its projection in the image plane becomes highly sensitive to small camera changes. This instability in $\hat{\mathbf{w}}_{\text{2d}}$ is revealed by wide strokes as the transverse direction bends abruptly at the silhouette. Rather than force the designer to employ $\gamma = 0$ universally, we reduce this artifact by enforcing it locally as $|\,\dot{\mathbf{q}} \cdot \hat{\mathbf{v}}\,| \to 1$. The effective γ is now given by:

$$\gamma_{\text{eff}} \;=\; \gamma\,[\,1 - (\dot{\mathbf{q}} \cdot \hat{\mathbf{v}})^2\,] \tag{5.5}$$

In the lower row of Figure 5.2 the artist's choice of $\gamma \sim 1$ produces a mainly 2D effect by preserving stroke widths under foreshortening. In the rightmost image, strokes parallel to the silhouette protrude off the surface as desired, unaffected by (5.5).

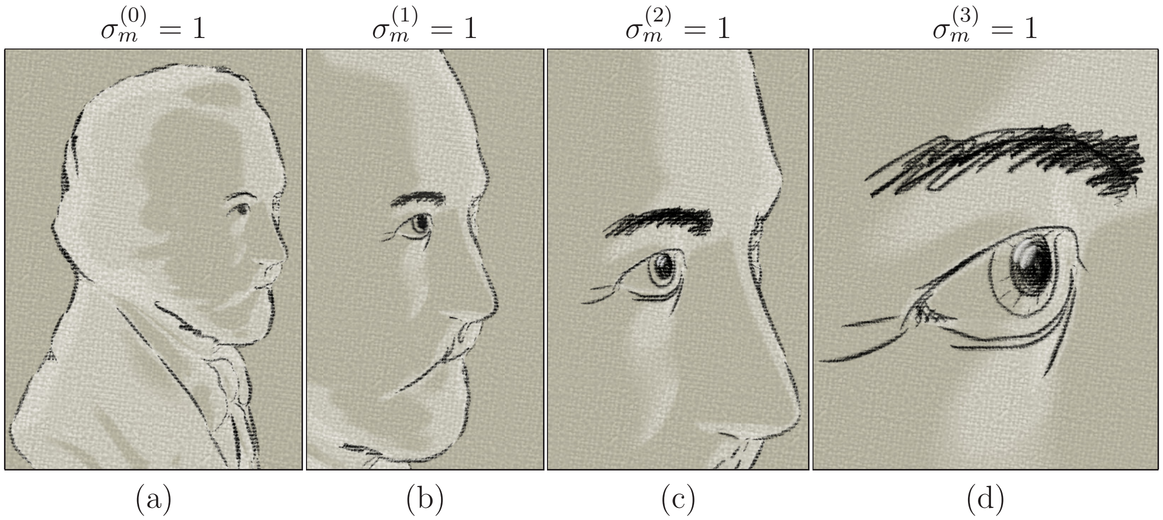


Figure 5.4: Four levels of detail for an eye effect: (a) initially there is little detail, (b) additional strokes soon reveal an iris and eyebrow details, (c) a highlight appears, and (d) fine details annotate the iris and the corner of the eye.

5.2.3 Level of Detail

Using only the tools described thus far, it is difficult to annotate an object with detail marks over a wide range of size scales. In any intermediate view, coarse details will become highly magnified $\sigma_m \gg 1$, while fine details will be squashed together $\sigma_m \ll 1$. This constrains the designer to using β^-, $\beta^+ \sim 1$. However, we find that when stroke widths scale closely with the size of the mesh over too wide a range, the result disturbs the painterly æsthetic – approaching an effect more akin to texture mapping. In this section, we describe a means for the artist to divide stroke groups into discrete levels of detail that are rendered only at appropriate scales, alleviating these constraints on more richly detailed scenes.

While composing a group of detail strokes, the user signifies the completion of each separate level of detail (LOD) with a *single-tap* gesture of the tablet pen. Once all of the desired LODs have been created, the group as a whole is again completed with a *double-tap* gesture as in Section 5.2.1. Instead of tracking object scale using a single σ_m for the whole group, now each separate LOD is associated with its own ratio $\sigma_m^{(\mathrm{i})}$ of the object's current size to that when the LOD was completed.

Once complete, the LODs are sorted in order of ascending scale so that $\sigma_m^{(0)}$ corresponds to the details visible when the object is most distant. The decision whether or not to render a given LOD is based upon its $\sigma_m^{(\mathrm{i})}$. If the object is so distant that $\sigma_m^{(0)} \ll 1$, then no LODs are rendered at all. The first LOD only becomes visible as $\sigma_m^{(0)} \to 1$. Then, as scale changes, the strokes in this LOD vary in width according to (5.1) using their respective $\sigma_m^{(0)}$. Once magnification increases such that $\sigma_m^{(1)} \to 1$, the system presents the designer with two possibilities: the higher LOD can either *augment* the strokes in the previous one, or simply *replace* them altogether to avoid excessive magnification. The process continues analogously for the remaining LODs, reversing when magnification decreases.

In the example of Figure 5.4, the artist created four LODs to depict an eye using the augmentation policy. It is worth mentioning that each LOD was not drawn at its respective scale. Instead, the designer composed all four LODs from camera pose (d) to gain increased precision. After each LOD was drawn, the camera was adjusted to its respective pose before "locking-in" the scale with a single-tap gesture. As we progressively zoom in upon the stylized bust, stroke widths grow only gradually, while additional details transition in to augment the sparse details from coarser scales.

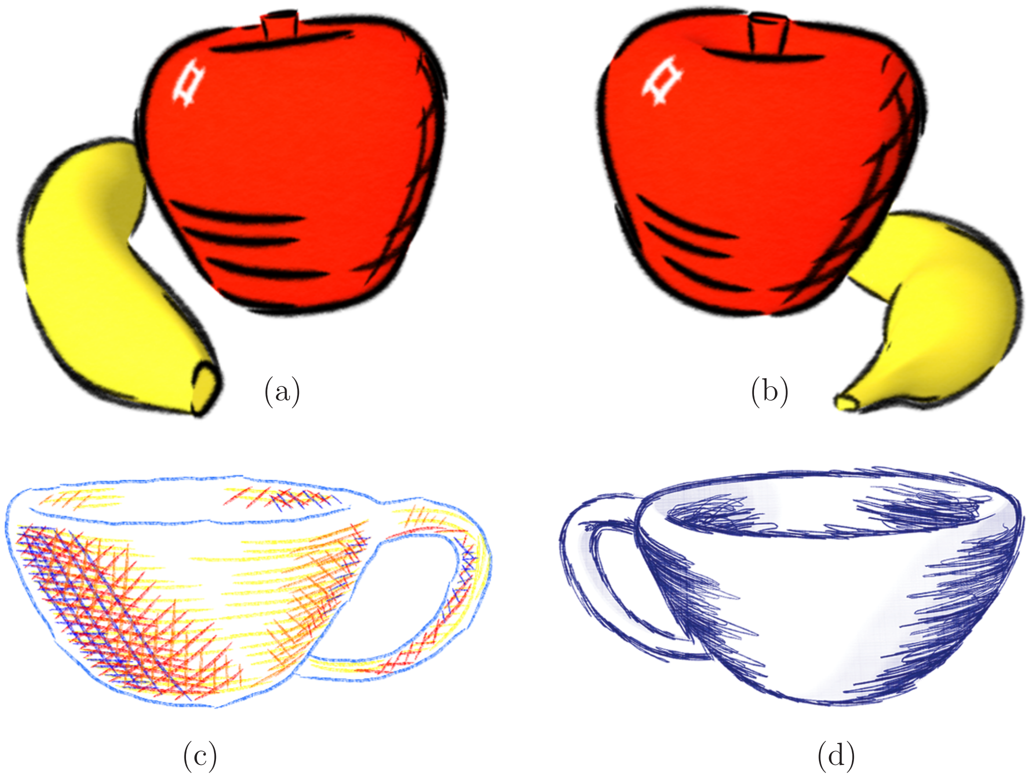
Care must be taken not to introduce coherence artifacts during discrete changes in LOD. Interactive scene navigation makes this especially challenging because the camera paths are unknown *a priori*. We approach this by *scheduling* an animated transition whenever the $\sigma_m^{(\mathrm{i})}$ test determines that some LOD should appear. The animation modulates one or both of width and α to cause strokes to grow smoothly into existence, and performs the reverse as they vanish. Once begun, transitions always run to completion so that LODs never persist in a state inconsistent with the designer's chosen brush parameters. Furthermore, we use hysteresis in the scale test to reduce spurious transitions. A given LOD will not be scheduled to appear until $\sigma_m^{(\mathrm{i})} > 1+\delta^+$. And, once visible, it will not be scheduled to vanish until $\sigma_m^{(\mathrm{i})} < 1-\delta^-$.

In total, this section presents a particularly large collection of parameters to control the dynamics exhibited by each group of detail marks: β^-, β^+, γ, δ^-, δ^+, and a few other LOD policy choices including replacement vs. augmentation, and animated transition settings. However, in practice, we have found that we can set many of these parameters to defaults that are reasonable for the vast majority of stylizations. In Jot's GUI, the more esoteric settings are concealed in roll-out panels that are only opened when the designer chooses to tweak some default parameter to achieve a more specific effect.

5.2.4 Interactivity

Like static feature lines (Section 4.2), detail marks present few challenges to achieving interactive performance rates. Their visibility can be resolved in real-time by again using the ID reference image (Section 3.2), though this time we test for triangle IDs instead of feature lines. Rendering then proceeds via the efficient stroke primitive described in Chapter 3.

Furthermore, the frame-to-frame coherence of these strokes is essentially intrinsic. Because they remain fixed on the underlying surface, their coherence will simply reflect that of any dynamic effects. The magnification and foreshortening behaviors described in Section 5.2.2 are both smooth functions of the view, so these do not introduce new artifacts. However, explicit effort was necessary to reduce the impact of the discrete LOD transitions in Section 5.2.3. We mitigate these with a hysteresis test that reduces spurious transition events. When transitions do finally occur, strokes are introduced and removed from the image gradually using smooth animations.


(a)

(b)

(c)

(d)

Figure 5.5: Tonal marks convey lighting-induced shading effects. Jot implements (a) roughly parallel *structured hatching* and (c) cross-hatching stroke groups, as well as (d) a flexible *free-form* hatching. In (b), the annotations move under implied lighting.

5.3 Tonal Marks

The other class of surface annotations we address in Jot are tonal marks. More rigidly defined than detail marks, these are groups of strokes that act collectively to convey tone, or shading effects that arise from lighting considerations. We focus primarily upon a common form of tonal markings called *hatching* – one or more overlapping sets of roughly parallel strokes, as in Figure 5.5a and c. Several NPR researchers have investigated methods to use procedural textures or principle curvature directions to define hatching strokes on the 3D surface consistent with a lighting environment [40, 82, 92, 93]. Recently, others have extended this work into real-time via novel application of GPU texturing hardware, and programmable shaders [29, 68, 88]. But all of these approaches are hands-off, taking as input only a description of the lighting, and the stroke textures or parameters governing how strokes populate the surface.

The images rendered by these parametric systems generally consist of large numbers of small strokes that collectively approximate the Gouraud shading model. They cannot, for instance, produce hatching effects which are more localized, conform to non-physical lighting models, or consist of only a few carefully chosen strokes. For these difficult to simulate effects – like the examples in Figure 5.5 – it is necessary to provide the designer with hands-on controls to describe the æsthetic. In this section, we describe a preliminary approach to implementing such tools. The process begins with the artist sketching the desired hatching effects directly into the scene. A lighting model is directly inferred from the annotations, such as those in Figure 5.5a, and the system adjusts them accordingly in novel views, as in Figure 5.5b.

We explore two types of tonal stroke groups – *structured hatching* (Section 5.3.1) which is composed of roughly parallel marks, and a *free-form hatching* (Section 5.3.2), which imposes no restriction upon stroke arrangement. These two categories are distinguished primarily by how their LOD behavior preserves tone over changing size scales. We reserve for Section 5.3.3 the discussion of how these hatching annotations define a lighting environment to govern their behavior in novel views.

5.3.1 Structured Hatching

In illustrations, one of the most common forms of tonal marks are the roughly parallel lines we call structured hatching. These groups of annotations collectively convey shading on the surface in proportion to their stroke density in the image. In Jot, the designer creates these effects by sketching them into the scene. After each group is drawn, the camera pose is adjusted until the hatching exhibits the desired spacing – a final tap gesture “locks-in” this target tone for the group. The result is a set of spline curves on the object surface that are rendered in a similar fashion to the detail marks in Section 5.2.

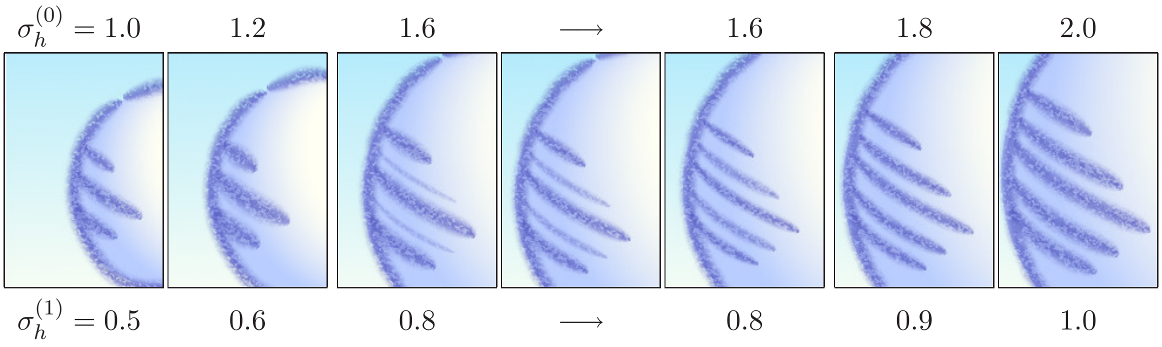


Figure 5.6: As the camera approaches structured hatching, level of detail effects preserve stroke density. Stroke widths change to preserve tone between transitions.

However, unlike detail strokes, structured hatching groups exhibit only a particular form of dynamic behavior. When magnification or foreshortening lead to changes in stroke density, the system compensates with effects geared toward maintaining the target tone. One way to affect this is by modulating stroke widths, but, as we found for detail marks, this is only æsthetically viable over small scales. Therefore, we again employ a discrete level of detail (LOD) framework (Section 5.2.3). This time, however, the designer need not sketch each LOD. Instead, we exploit the regularity of structured hatching to do this automatically.

Each time hatching density falls too low, we introduce a new LOD that *augments* the existing hatching strokes to restore their spacing. This is governed by the ratio $\sigma_h^{(0)}$ of the current mean spacing between hatching strokes, to that when the designer locked the group. By definition, hatching exhibits the desired tone if $\sigma_h^{(0)} = 1$. As shown in Figure 5.6, when magnification leads this spacing to double $\sigma_h^{(0)} = 2$, a new LOD restores the desired tone by doubling the density. The new strokes are created automatically in the interstices of the existing ones via interpolation. As before, each LOD is associated with its own ratio $\sigma_h^{(\mathrm{i})}$ which controls its behavior. The first synthesized LOD restores the target tone $\sigma_h^{(1)} = 1$ when $\sigma_h^{(0)} = 2$. This proceeds analogously for each successive doubling of the original spacing $\sigma_h^{(0)} = 2^i$, where an associated LOD re-establishes target tone $\sigma_h^{(\mathrm{i})} = 1$ by again doubling density.

Between discrete LOD transitions, we compensate for disparity in target density by modulating brush widths, as described for detail marks in equation (5.1). When stroke spacing exceeds or falls short of the target, the original brush widths expand or narrow, respectively. However, we move the LOD transition threshold from $\sigma_m^{(\mathrm{i})} = 1 \pm \delta^{\pm}$ for detail groups, to $\sigma_h^{(\mathrm{i})} = {}^3\!/_4 \pm \delta^{\pm}$ for hatching groups. This ensures that the active LOD is always the one which most closely approximates the original density. As the camera approaches a hatching group in Figure 5.6, spacing begins to exceed the target ($\sigma_h^{(0)} > 1$) and widths increase in response to preserve collective tone. Eventually the threshold for the next LOD is crossed (here $\delta^{+} = 0.05$), and new strokes grow in during the scheduled transition animation. Spacing now falls short of the target ($\sigma_h^{(1)} < 1$), so widths scale back from their original brush size until the desired density is reached again at $\sigma_h^{(1)} = 1$.

Because of the coherence cost incurred during each stroke-doubling event, we have investigated LOD schemes that amortize this impact by introducing fewer strokes more often. However, we find that this actually accentuates the problem. Though individually such transitions are less disruptive, in practice they often draw more attention by virtue of their frequency. Nevertheless, more complete systems might offer these or other novel tone-preserving dynamic effects.

5.3.2 Free-Form Hatching

For some tonal effects, the artist may find structured hatching too constraining. Therefore, we also provide a *free-form hatching* that enables the designer to explicitly build the tone-preserving LODs. These are composed by directly drawing arbitrary arrangements of strokes, as in Figure 5.7. Indeed, this is precisely the same LOD framework described for detail marks in Section 5.2.3. But there is an important distinction – as described in the next section, both structured and free-form hatching groups can imply a lighting model to dynamically control them in novel views.

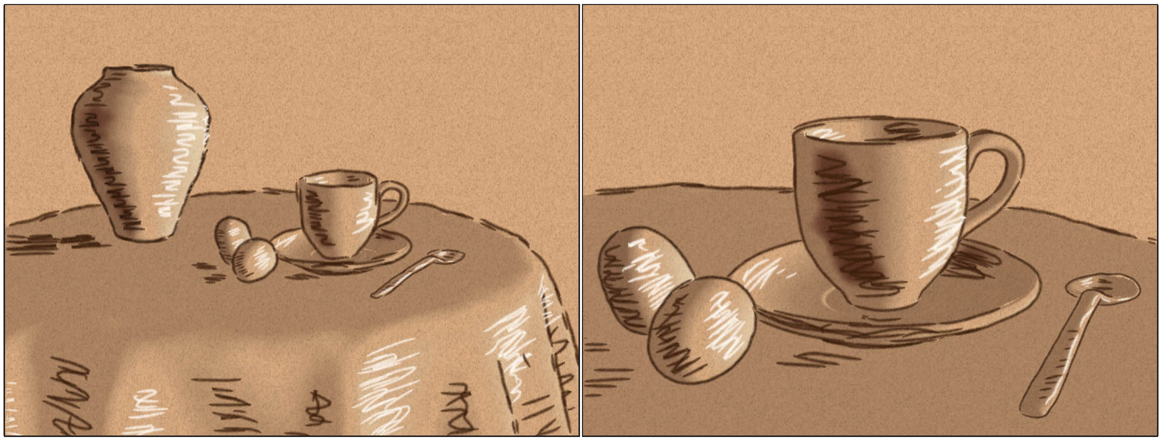


Figure 5.7: Using free-form hatching, the designer can achieve arbitrary tonal effects. Custom generated level of detail strokes (that preserve tone) are revealed in a close-up.

5.3.3 Dynamic Lighting

Thus far, we have presented dynamic effects that strive to preserve the tone of hatching groups under changing views. As their density changes, hatching strokes appear, disappear, and change in width, but otherwise are stationary on the underlying surface. As a result, the shading effects these markings convey will imply consistency with lighting models fixed in the world frame. However, it is natural to investigate the impact of dynamic lighting models. This is easily accomplished in hands-off systems, as they render imagery by starting from a description of the lighting environment itself. Typically, many small strokes on the surface are used to approximate the standard Gouraud shading model, in what is effectively a half-toning problem. In contrast, we have chosen an approach which provides the designer means to directly sketch the final hatching effects. It is not obvious how to derive an explicit description of the lighting model from the designer's annotations. In fact, the artist is entirely free to produce imagery that does not exhibit consistency with any global lighting environment. And, even if such a lighting description could be extracted from the scene, it is still unclear how the designer's annotation should respond if it were to change dynamically.

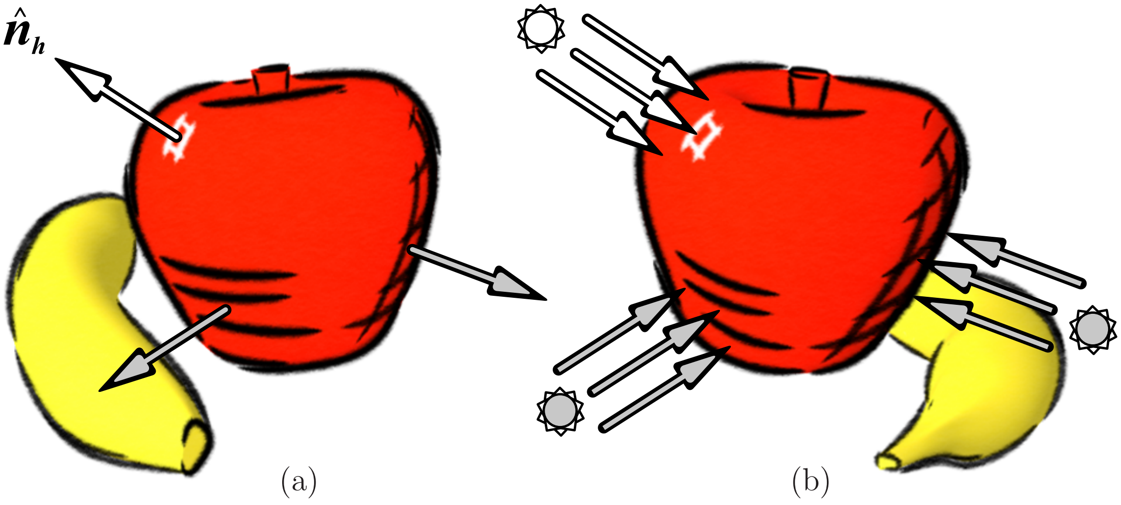


Figure 5.8: Dynamic lighting effects apply identically to highlights and shadows: (a) each hatching group defines a directional light along the local surface normal, and (b) in novel views hatching groups move because lighting is fixed in the camera frame.

To approach this challenge, we proceed from observations of these effects in traditional illustrations. Often, artists will employ shading near the silhouettes of objects, suggestive of lighting cast from somewhere behind the camera. In novel views of the scene, these shading marks do not remain fixed on the surface, but tend to appear again near the silhouettes. This behavior is consistent with lighting which is fixed in the camera frame. Because we cannot hope to determine a globally consistent lighting model, we will pursue this observed dynamic on a per-hatching group basis. That is, if the designer annotates the "left side" of an object with a tonal marks (e.g., on the snowman in Figure 5.9), the system will maintain these annotations on the "left side" of the object in new views. We call this *mobile hatching.* The artist can employ this dynamic with both structured and free-form hatching, as shown on the snowman and trees, respectively, in Figure 5.9.

We accomplish this with a straightforward model that applies equally to tonal marks suggestive either of highlights or shadows. Each hatching group created by the artist is associated with a pseudo directional light source opposite to the local surface normal $\hat{\mathbf{n}}_h$. By definition, this source will illuminate the surface most strongly at the location of the annotation. For highlights, we imagine a light source in the usual sense producing this tone. However, for shading strokes, we think of a “dark light” shining darkness onto the local surface. As shown in Figure 5.8, these pseudo sources are held fixed in the camera frame in novel views. The dynamic effect is achieved as each mobile hatching group moves about the surface to maintain a location which is strongly illuminated by its respective source.

Motivated by the goal of real-time rendering, we currently implement this general technique only for surfaces that satisfy a narrow set of constraints. We consider smooth surface regions with roughly uniform *uv*-parameterization, and further restrict the motion of mobile hatching to particular curves in parameter space. Specifically, we choose the direction along which surfaces exhibit a “wrapping” in the parameterization (e.g., along the longitudinal lines of a sphere or cylinder). We will call this the u direction. When the artist completes a mobile hatching group, the system projects the strokes into *uv*-space, computes their convex hull, locates its centroid (u_c, v_c), and records $\hat{\mathbf{n}}_h$ as the surface normal at (u_c, v_c). The parametric path over which hatching travels is taken to be the line $v = v_c$. For new views, we sample the surface normal along the line $\hat{\mathbf{n}}(u, v_c)$, and evaluate a Lambertian lighting function $\ell(u) = \hat{\mathbf{n}}_h \cdot \hat{\mathbf{n}}(u, v_c)$. Note that, when obtaining either $\hat{\mathbf{n}}_h$ or $\ell(u)$, we use a copy of $\hat{\mathbf{n}}(u, v_c)$ cached with the hatching group. Each copy is smoothed via a filter kernel with the extent of its respective stroke group, thereby accounting for the normal across the entire band of surface inhabited by the hatching, and reducing sensitivity to small-scale surface variations. Next, at each local maximum in $\ell(u)$ we create a mobile hatching group whose extent is governed by the width of the peak. We omit maxima where $\ell(u)$ is less than a confidence threshold T (we use $^1/_2$) that prevents hatching from being placed at insignificant local maxima arising from the constrained motion of the group. Finally, as $\ell(u)$ approaches T we fade out the group to achieve a coherent transition.

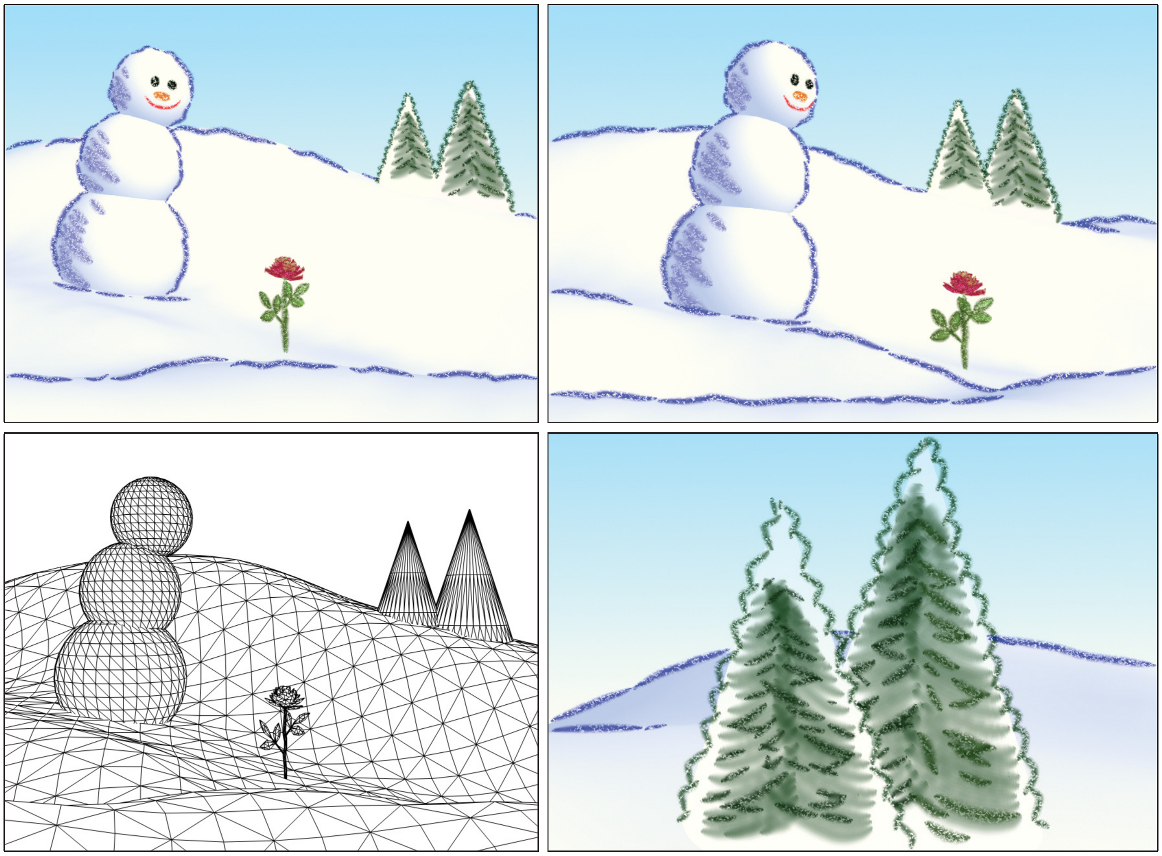


Figure 5.9: Simple geometry provides a "blank canvas" for stroke-based details. A 'toon shader and structured hatching suggest lighting on the snowman, detail strokes depict a face, and uneven blue crayon silhouettes imply bumpy snow. Conical trees are annotated with sawtooth crayon silhouettes plus two layers of hatching effects.

While this particular implementation is limited, it could be easily extended to handle arbitrary unparameterized surfaces. This is possible with the use of "proxy surfaces." As demonstrated by Markosian [55], compelling hatching effects can be generated by composing roughly parallel strokes in the image plane, and then simply clipping them to an underlying 3D surface – optionally modulating their widths to account for curvature, etc. Instead of the image plane, we could enable the designer to sketch mobile hatching effects onto a proxy surface, such as a bounding cylinder with the necessary *uv*-parameterization. By clipping the final hatching to the underlying general surface, we can achieve a similar effect to Markosian, but with additional dynamic behavior arising from our lighting model.

While the “dark light” model may not be physically accurate, it does yield plausible cartoonish lighting effects. It is also extremely easy for the artist to specify, and does not constrain him to depict lighting that is globally consistent. Still, the problem of inferring lighting in response to hand-drawn shading merits further investigation. Other researchers, for example, Poulin et al. [67] and Schoeneman et al. [77], have addressed this problem with greater rigor in the context of photorealism.

5.3.4 Interactivity

The key challenges addressed in achieving interactivity for tonal marks are essentially identical to those for detail marks (Section 5.2.4). A notable exception is our approach to achieving real-time rendering for mobile hatching. For this effect, it was necessary to constrain the solution to parameterized surfaces to simplify the task of evaluating the pseudo lighting criterion. Additionally, each parameterized region which receives mobile hatching annotations is stored into spatial data structure – one that permits O(1) look-up of the triangle containing a given *uv*-coordinate. This is necessary to facilitate efficient rendering as mobile hatching traverses parameter-space.

5.4 Discussion

Summary of Surface Effects

In this chapter, we addressed various aspects of WYSIWYG NPR tool development for surface stylization effects (Table 2.4.2, page 15). We began by describing our approach to annotating surfaces via the sketching of strokes directly into the scene. Such tools for creating detail marks immediately lead to a wide array of effects, ranging from rich painterly styles (Figure 5.1a), to detailed line drawings (Figure 5.1d). Similar tools for hatching enable the artist to create tonal effects directly and locally, unconstrained by global lighting models. In concert with a flexible stroke primitive and media simulation, these systems provide æsthetic flexibility, and the hands-on means for designers to impart their personal æsthetic (Figure 6.1).

The annotated surfaces can be freely rendered under animation and novel viewing conditions. The system preserves the artist's annotations while dynamically adapting stroke properties to changing scales and perspective. The designer can tune this scripted behavior to elicit effects that balance between 2D stroke-like behavior and 3D texture mapping æsthetics. Additional controls provide LOD effects customized through direct sketching at the desired scales, while the regularity of hatching is exploited to accomplish this automatically. We also present a new cartoonish lighting model that is inferred directly from hatching annotations, which imposes no physical consistency constraints upon the implied illumination.

And, once again, we ensure that these tools function effectively in an interactive domain – providing the designer with a responsive, hands-on environment. Various design choices coupled with rendering algorithms that exploit the GPU for visibility and stylized rendering, achieve real-time performance rates for scenes of reasonable complexity [46, 47]. Furthermore, groups of strokes on the surface maintain their frame-to-frame coherence during LOD changes. This is accomplished through scripted transition animations that mitigate the impact of discrete LOD events, both under real-time navigation and during the playback of animated sequences (Figure 6.6).

With WYSIWYG NPR tools, the artist may explore stylizations previously inaccessible in animation. When the system assumes the coherence of the annotations, the artist is free to exploit detailed stroke-based effects which would be daunting to produce by hand (e.g., Figures 5.1c and d). In fact, the vast majority of hand-generated cell-animation employs absolutely no stroke-based effects at all. Surfaces are generally portrayed in only solid colors, or at most, a few discrete bands. Clearly, these tools can open the way to new styles for the animator.

Future Work

Pattern/Hatching Synthesis

We have yet to explore tools that can synthesize stylization from the annotations made on surfaces, as we have for stylized lines in Chapter 4. This could significantly reduce the burden on the designer when stylizing many elements in complex scenes. For example, the artist might sketch only a few bricks, and then direct the system to generate similar effects over an entire wall. This challenge has been pursued in the context of repetitive image synthesis from example input [25, 26, 37, 89], and extended to the explicit case of texture maps on surfaces [87, 90, 94]. But with the exception of some work on a particular form of hatching effects by Jodoin et al. [45], the synthesis of stroke-based patterns remains an entirely open problem.

Automatic Abstraction

Hand crafted illustrations typically omit details to reduce image complexity as scenes are viewed more distantly. At present, our system supports this sort of abstraction in the form of the explicitly constructed LODs for detail and hatching stroke groups. An interesting question is how to omit or even merge strokes in scenes automatically, and to do so with temporal coherence as the camera zooms in or out. For instance, the designer might specify surface annotations at their highest LOD, while the system assumes the task of omitting and merging strokes to automatically generate the more abstract representations. This type of effect has been investigated in 2D painterly rendering systems, and other applications in NPR (see DeCarlo et al. [20] for a brief survey), but the problem of achieving this with coherence remains unaddressed.

Chapter 6

Conclusions

6.1 Summary

In this dissertation, we have begun to target the key challenges that have so far limited the use of stylization and abstraction in computer graphics. The inability to apply these techniques has greatly restricted the adoption of 3D rendering systems by content creators beyond a small number of particular applications. The reason is that photorealistic content is extremely difficult and expensive to create, and, in many cases, is not the most effective medium of expression. Over the centuries, artists have developed a broad range of techniques which convey visual information more effectively and efficiently for many applications. A growing branch of graphics research now explores NPR technologies that bring these methods into the 3D graphics domain, but relatively little research has addressed the problem of creating flexible tools at the disposal of the designer. Thus far, the focus has been upon "black boxes" which can replicate certain traditional styles. Though such systems are certainly useful, our goal in this work was to demonstrate the fundamental importance of developing a more hands-on, flexible class of tools.

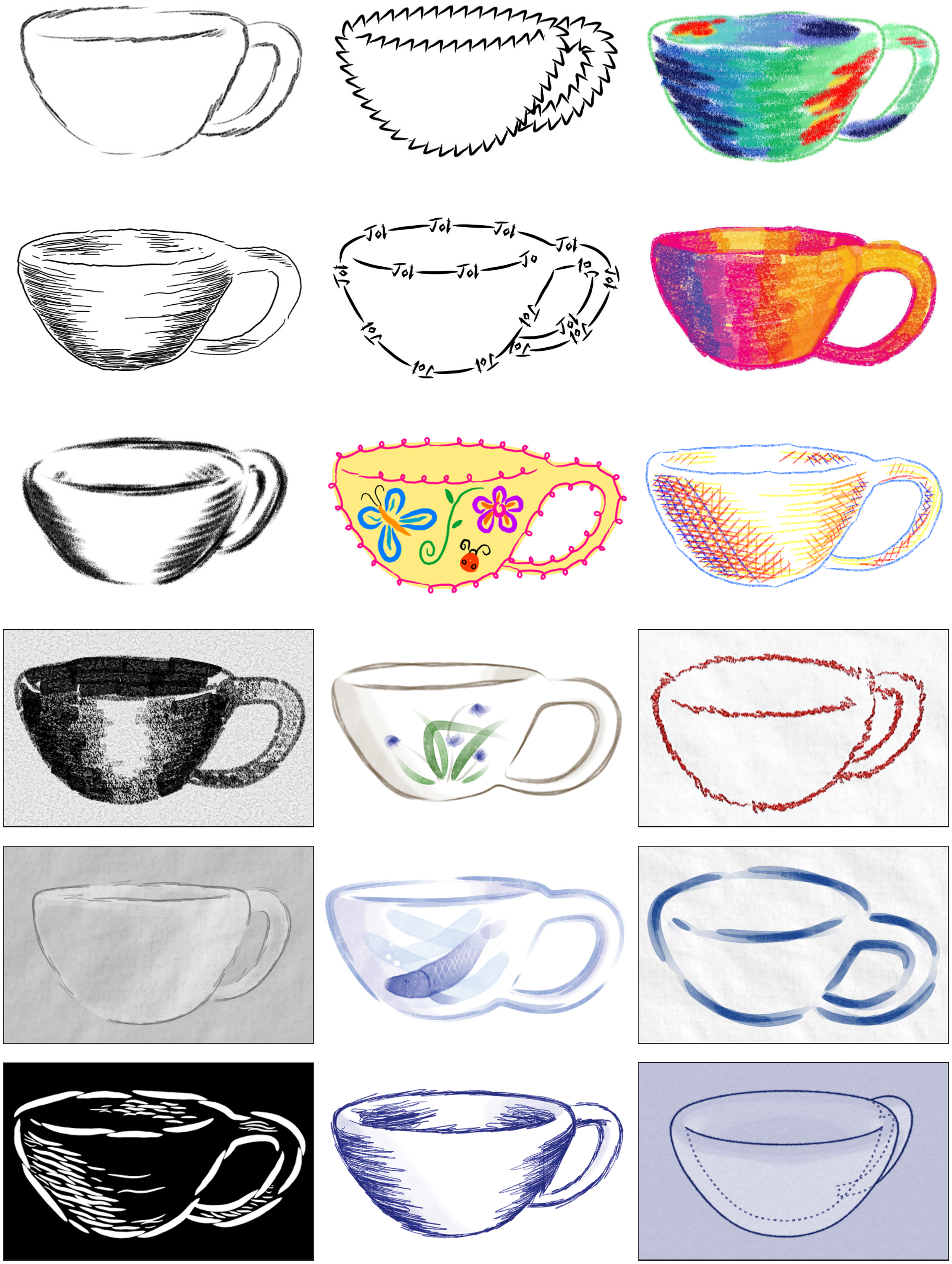


Figure 6.1: WYSIWYG annotation tools provide wide degree of æsthetic flexibility. The same cup mesh has been stylized in 18 different ways.

To this end, we proposed WYSIWYG NPR – design tools that offer flexible interfaces to exploit the designer's particular skills and judgment, and useful algorithms he can exploit to offload the more tedious aspect of content creation. We approached this challenge by focusing on three key aspects: annotation tools for describing the stylization of 3D scenes, interfaces and algorithms that govern the dynamic behavior of stylization in novel views, and the challenges that arise when implementing fully interactive NPR systems.

Though we explored these tools for a small set of stroke-based effects, we find the pay-off in æsthetic range to be large. Direct, hands-on annotation interfaces immediately lead to a wide degree of flexibility in stylization (Figure 6.1). Complex geometry tends to offer the designer details that can be "revealed" through media shaders and contour stylization (Figures 6.2 and 3.4, page 27), whereas simpler objects offer a "blank slate" on which to create new details where none exist (Figure 5.9, page 90). By bringing the artist into the loop, we exploit his unique discretion to produce subtle or complex effects that would be extremely difficult to simulate in automatic rendering systems. Nevertheless, to expedite stylization of dense scenes, the user can still call upon the system to automatically generate annotations. Because these are synthesized from the artist's own examples, he still remains in control of the final æsthetic.

The rewards of WYSIWYG NPR are further realized in dynamic settings. Once geometry is annotated by the designer, he can freely render the scene from novel views while the system maintains the annotations automatically. To accomplish this, it was necessary to consider how annotations from one view evolve under dynamic viewing conditions – a problem unique to hands-on tools. We met this with various rendering algorithms that can exhibit a range of behavior which the designer can adjust to specific goals, either by tuning parameters or directly sketching the desired effects.

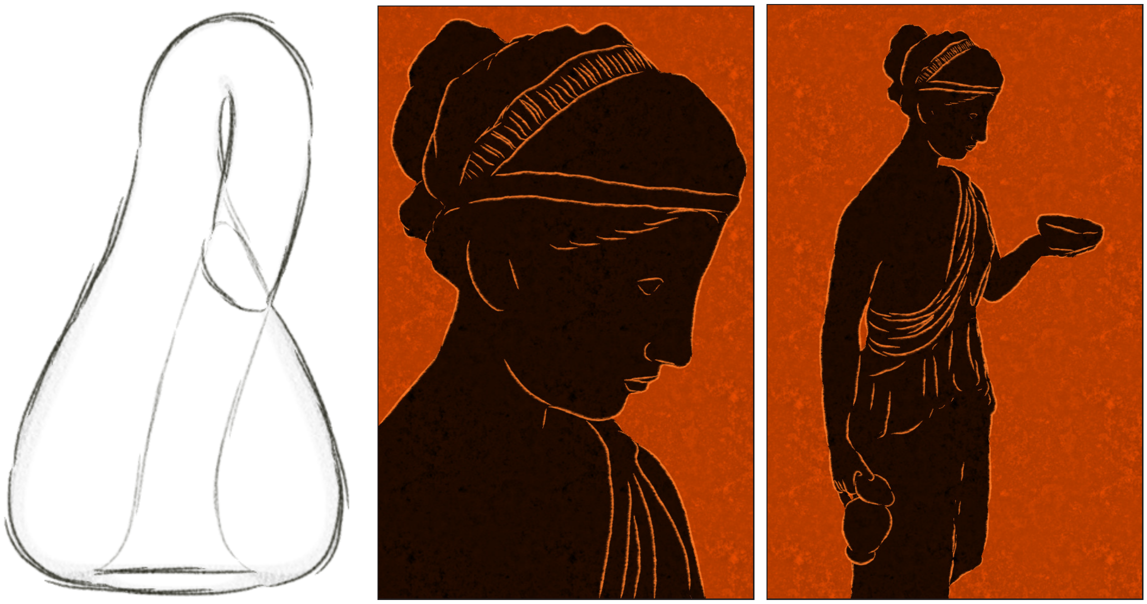


Figure 6.2: The objects from Figures 4.4b and Figure 3.4 annotated in new styles.

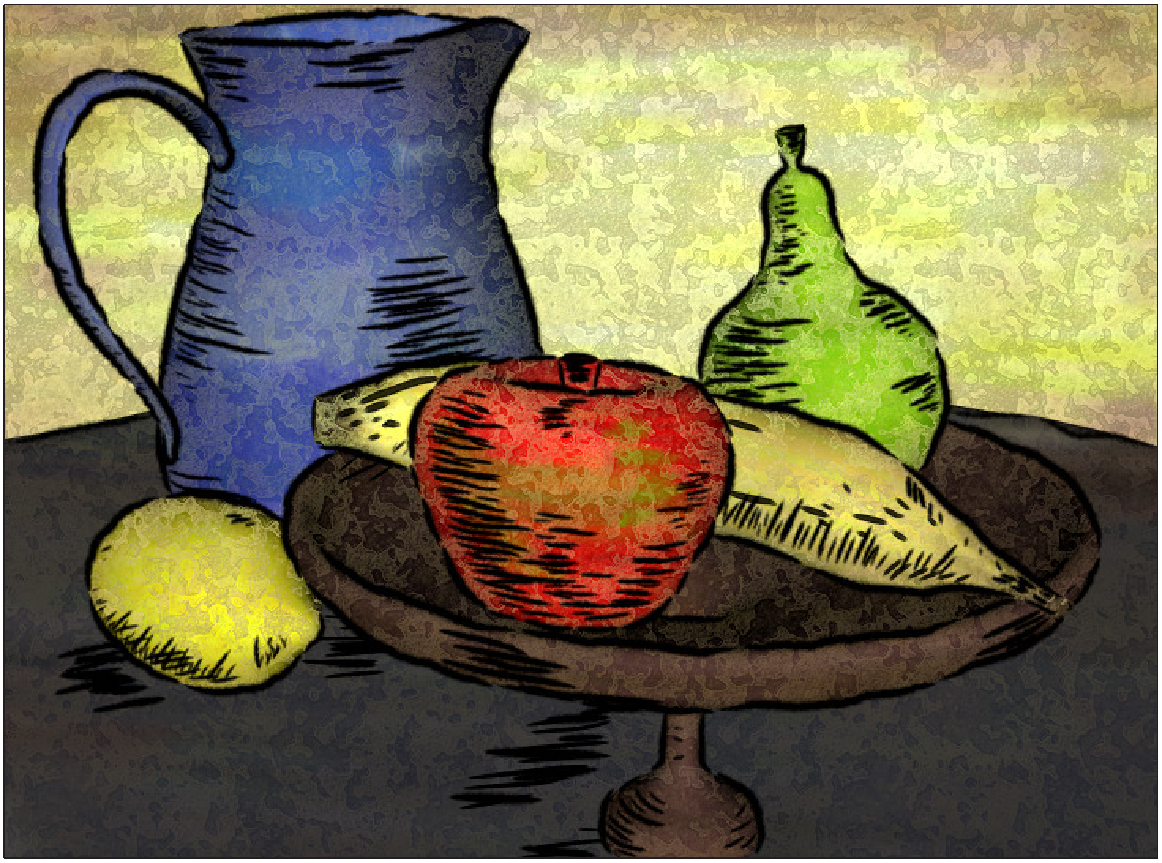


Figure 6.3: To create this still life, the artist annotated silhouettes, and sketched hatching effects. Surfaces are lit, textured, and composited on splotchy paper.

And, finally, all of these systems were developed to meet the demands of interactive environments. We presented solutions which ensure the necessary performance rates, including several that exploit the GPU to achieve real-time visibility, sample propagation, media simulation, etc. Furthermore, we enforce the coherence of stylizations during real-time interactive navigation (and off-line rendering). While this was relatively straightforward for most of the effects in Jot, it presented significant challenges for view-dependent contours. We addressed these with a new coherence framework that significantly surpasses previous work, opening the door to much richer stylizations for animation. The designer can stylize just one frame of an animated sequence, as in Figure 6.6, and the system assumes the task of enforcing a coherence in future frames. The behavior of the coherence can be tailored from a purely 2D effect (as on the animated gears in Figure 6.5), through to a 3D shape tracking dynamic (like the dancing cactus in Figure 6.7), and even animated stylizations (Figures 6.4).

The value of WYSIWYG NPR tools is clear. These hands-on systems provide the artist with a new range of æsthetic flexibility and convenient tools to off-load the tedious aspects of content creation, even opening the way to effects previously challenging or impossible to attain by traditional means.

6.2 Future Work

To increase the expressive power of 3D graphics for the content creator, in Chapters 1 and 2 we propose WYSIWYG NPR tools that bring the techniques of stylization and abstraction to bear in a hands-on domain. We organized our approach identifying key design considerations, and mapped the problem space for our test system in Table 2.4.2 (page 15). The majority of these issues were considered in this project, while a few remain for future work. We have made substantial progress toward this new class of tools, but even if we had addressed all aspects in Table 2.4.2, this work would still be far from complete. In fact, we have taken only a few first steps toward systems that can vastly enhance the expressive power of computer graphics.

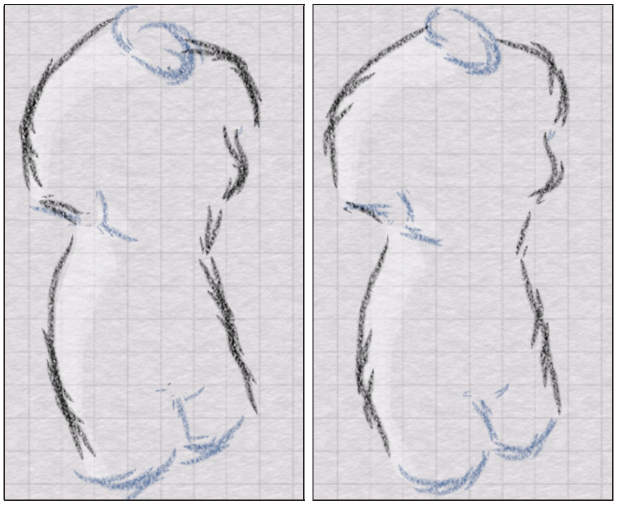


Figure 6.4: A Squigglevision™ effect

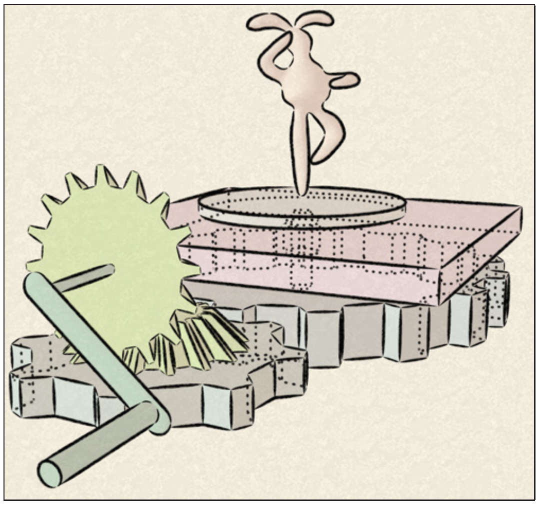


Figure 6.5: Animation of rigid bodies

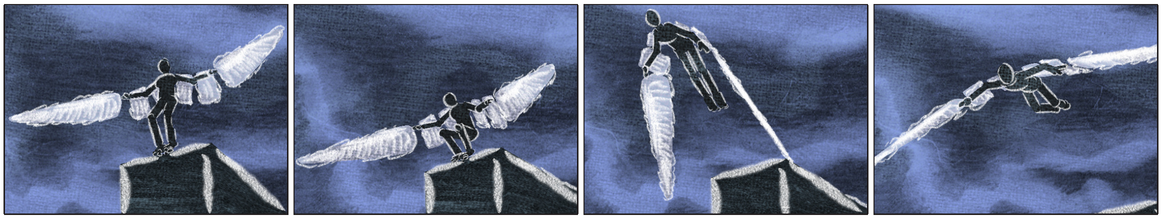


Figure 6.6: Four frames from an animation with objects undergoing full deformation. Silhouette stylization and free-form hatching produce the shown æsthetic.

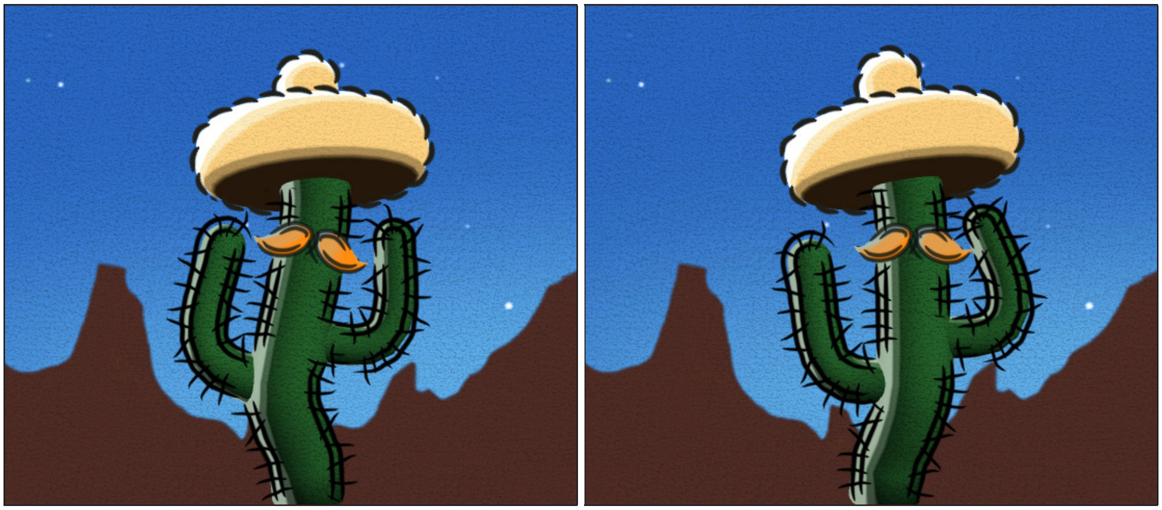


Figure 6.7: The "Boogie Cactus." A stylization depicting thorns sticks to the silhouette of the smooth underlying mesh as the figure deforms under animation.

Markosian [55] observes that there is a fundamental difference in how one gauges the success of photorealistic imagery versus that of NPR. We recognize that photorealism has achieved its goal when the result *looks photorealistic*. But, while research in non-photorealism commonly describes success when the result *looks good*, this metric should instead reflect how close the result *looks as the designer intended*. Indeed, the clearest path to this goal is the continued development of tools that provide the designer with the highest degree of hands-on control and æsthetic flexibility.

In addition to our recommendations in Sections 4.5 and 5.4, we can imagine such future work as additions to Table 2.4.2 that extend the flexibility of WYSIWYG NPR. New rows, for instance, represent additional design considerations. Our current focus has been upon annotation of 3D scenes for real-time interaction. However, if the target is the production of scripted animation, then tools are required for specifying stylization effects in the time domain – perhaps by sketching stylizations at key-frames. On the other hand, additional columns for the table represent new forms of stylization. One might investigate more complex stroke-based æsthetics, like blended painterly styles where underlying strokes begin to interact or bleed together, or perhaps dense markings like stippling. Another interesting problem involves the effects that are a hybrid of geometry and stylization, such as graftals that depict details like leaves, and fur [50, 57]. Tools for these effects could be useful for adding apparent geometric details to much simpler underlying 3D objects.

Tools for the annotation of existing 3D geometry are, however, only half of the content creation equation. Indeed, in Section 2.1 we posit that the “Holy Grail” of NPR research in to enable the artist to simply draw in the image plane and thereby express a complete, stylized 3D scene. Full WYSIWYG systems should also enable the designer to create the underlying geometry through similar hands-on methods. One possible approach would be to integrate WYSIWYG NPR tools like Jot, with sketch-based modeling systems such as Teddy [42]. In fact, we have already begun preliminary work on such tools. The teacup seen throughout this thesis, and the "still life" in Figure 6.3, were all created entirely through gestural input from the tablet in Jot. An additional mode of the system (under development) offers a gestural interface for creating objects and composing them in the scene, the results of which can then be annotated with the WYSIWYG NPR tools we have described.

But, perhaps not all NPR effects are best expressed as annotations atop a conventional 3D mesh. Bourguignon et al. [5], for instance, describe a system that generates just enough geometry to present a stylized, view-dependent result consistent with each input gesture from the designer. Yet the system never explicitly creates the actual 3D object that these gestures serve to depict. This can be particularly useful when the artist needs to quickly convey a "back of the envelope" sketch, but does not have the time or resources to find or create the requisite 3D geometry. However, these approaches [12, 86] presently place significant restrictions on the structure of this geometry as well as final rendering æsthetics.

Nevertheless, with the ultimate goal of presenting a "blank canvas" with tools capable of expressing stylized virtual scenes that can meet the designer's intentions, it is necessary to pursue the ongoing development and integration of WYSIWYG modeling and annotation tools, as well as these hybrid systems which blur such distinctions.

6.3 Jot

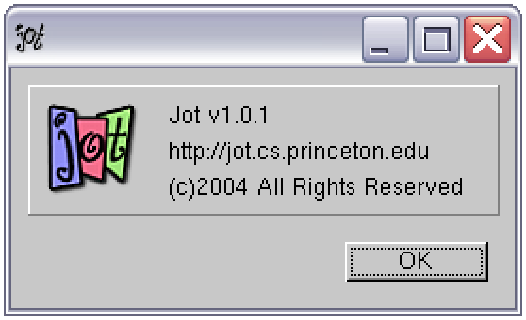
What is the status of Jot?

- The WYSIWYG NPR system described in this dissertation can be downloaded for personal, educational, and research purposes.

- A binary is available for Windows platforms, but Linux and OS X are planned.

- Source code will ultimately be released under some license.

Where can you get it?



Jot v1.0.1
http://jot.cs.princeton.edu
(c)2004 All Rights Reserved

Appendix A

Media Simulation Details

A.1 Overview

In this section, we describe an efficient GPU-based implementation for the media effect from Section 3.3. The goal is to simulate the process of depositing pigment onto an uneven paper surface as fragments are composited into the frame-buffer. In practice, we imagine that the α-component of each incoming fragment $(\, r \;\; g \;\; b \;\; \alpha \,)$ represents the "pressure" with which pigment $(\, r \;\; g \;\; b \,)$ is deposited. A paper height-field texture is used to control a remapping $\alpha \rightarrow t(\alpha, h)$, where the transfer function at each pixel accounts for the underlying paper height and incoming pressure value. The final pigment $(\, r \;\; g \;\; b \,)$ is then composited through alpha blending into the frame-buffer according to the remapped component $t(\alpha, h)$.

This effect can be applied to any rendering primitive with an α-component. For strokes, α can be viewed as the pressure profile across a brush. With effectively chosen stroke parameters (Section 3.1) and paper textures, various forms of traditional media can be emulated, as in Figure 3.3 (page 26). But, this effect applies equally to mesh triangles. In Section 2.3, we describe how objects can be assigned a basecoat consisting of shaders that render the object's triangles. These include solid shading, conventional Gouraud lighting and texture mapping, and ‘toon shaders – all of which can be configured to assign an α-component in some way, as demonstrated with the ‘toon shader in Figure 3.4 (page 27).

In this appendix, we demonstrate the GPU code that implements such a ‘toon shader coupled with the media simulation effect. For clarity we simplify the implementation by removing certain functionality. For instance, our actual shader supports both directional and point source ‘toon lighting, as well as providing brightness and contrast parameters to adjust the paper height field texture. The latter was found extremely useful for tuning the media simulation æsthetic.

A.2 Design Considerations

We implement the media simulation efficiently in hardware using both vertex and fragment programs. The vertex program is necessary to set the texture coordinate of each vertex to its screen-space location, causing the paper height field texture to appear fixed in the image plane. The associated fragment program can then look up the paper height at each pixel to compute the alpha transfer function given by equation (3.3).

The vertex and fragment programs that implement a ‘toon shader are given in Sections A.3 and A.4, respectively. The ‘toon profile texture is bound to `Tex0`, while the paper height field texture is bound to `Tex1`. The vertex program sets the `Tex0` coordinates to $u_0 = (\, s \;\; t \;\; r \;\; q \,) = (\, \hat{\mathbf{n}} \cdot \hat{\ell} \;\; 0 \;\; 0 \;\; 1 \,)$ appropriate for a directional light $\hat{\ell}$, while the `Tex1` coordinates u_1 are set according to the screen-space position of the vertex. At points interior to a triangle, the fragment program modulates each pixel by the local value of the ‘toon texture `Tex0`($\tilde{u}_0$), and re-maps $\alpha \to t(\alpha, h)$ using the paper height at the location of the pixel `Tex1`($\tilde{u}_1$). The texture coordinates $\tilde{u}_0$, $\tilde{u}_1$ of each fragment are interpolated across triangle faces from the values at the vertices.

And this is the catch: we must be careful when setting u_0 to the screen-space location of the vertex. The GPU will determine texture coordinates $\tilde{u}_1$ for each pixel inside a triangle by *perspective-correct* interpolation of the coordinates u_1 at its vertices [3, 35]. Because the paper texture lies in the image plane (unlike the 'toon texture), we must ensure that these texture coordinates are only *linearly interpolated* in the image plane. Failure to do so will cause the media effect to warp as the camera approaches 3D objects, violating the expected behavior of an image-space height field. Given object vertex $\mathbf{p} = (\, x \;\; y \;\; z \;\; 1\,)$ and concatenated 4×4 modelview-projection matrix $\mathbf{MP}$, the homogeneous clipping coordinates of the vertex are:

$$\mathbf{p}' \;=\; (\, x' \;\; y' \;\; z' \;\; w'\,) \;=\; \mathbf{p}\,\mathbf{MP}. \tag{A.1}$$

Once clipped to the view frustum, the final screen-space coordinates are given by:

$$\mathbf{p}'' \;=\; (\, x'' \;\; y'' \;\; z'' \;\; 1\,) \;=\; (\, x'/w' \;\; y'/w' \;\; z'/w' \;\; w'/w'\,) \tag{A.2}$$

We require for each vertex that the `Tex1` coordinate u_1 reflect its screen-space location:

$$u_1 \;=\; (\, s \;\; t \;\; r \;\; q\,) \;=\; (\, x'' \;\; y'' \;\; 0 \;\; 1\,) \tag{A.3}$$

However, this choice causes distortion at triangle interiors where perspective correction determines $\tilde{u}_1$ via pre-multiplying by $1/w'$ to linearly interpolate the perspective corrected quantity below, before post-multiplying by w'/q:

$$u_1 \;\xrightarrow{\times 1/w'}\; (\, s/w' \;\; t/w' \;\; r/w' \;\; q/w'\,) \;\xrightarrow{\times w'/q}\; \tilde{u}_1 \tag{A.4}$$

Therefore, we choose $u_1 = (\, x' \;\; y' \;\; 0 \;\; w'\,)$. By equations (A.4) and (A.2), this leads to linear interpolation of screen-space location $(\, x'/w' \;\; y'/w' \;\; 0 \;\; w'/w'\,) = (\, x'' \;\; y'' \;\; 0 \;\; 1\,)$, as we desire. Because this is post-multiplied by w'/w'= 1, the final texture coordinate $\tilde{u}_1$ gives back the screen-space location we seek in equation (A.3).

A.3 Vertex Program

This GL_VERTEX_PROGRAM_ARB sets the texture coordinates appropriately for a ‘toon shader with light $\hat{\ell}$, and the media simulation of Section 3.3.

```
!!ARBvp1.0

ATTRIB iCol = vertex.color;
ATTRIB iPos = vertex.position;
ATTRIB iNorm = vertex.normal;

OUTPUT oCol = result.color;
OUTPUT oPos = result.position;
OUTPUT oTex0 = result.texcoord[0];
OUTPUT oTex1 = result.texcoord[1];

PARAM cOne = 1.0,1.0,1.0,1.0;
PARAM cZero = 0.0,0.0,0.0,0.0;

PARAM mMP[4] =  state.matrix.mvp ;
PARAM mTex0[4] =  state.matrix.texture[0] ;
PARAM mTex1[4] =  state.matrix.texture[1] ;

PARAM cL = program.env[0];

TEMP tPprime;
TEMP tNDotL;

### Compute p’ = p MP for media effect
DP4 tPprime.x, mMP[0], iPos;
DP4 tPprime.y, mMP[1], iPos;
DP4 tPprime.z, mMP[2], iPos;
DP4 tPprime.w, mMP[3], iPos;

### Compute N dot L for toon shader
DP3 tNDotL.x,iNorm,cL;
MOV tNDotL.yz,cZero;
MOV tNDotL.w, cOne;

DP4 oTex0.x, mTex0[0], tNDotL;
DP4 oTex0.y, mTex0[1], tNDotL;
DP4 oTex0.z, mTex0[2], tNDotL;
DP4 oTex0.w, mTex0[3], tNDotL;

DP4 oTex1.x, mTex1[0], tPprime;
DP4 oTex1.y, mTex1[1], tPprime;
DP4 oTex1.z, mTex1[2], tPprime;
DP4 oTex1.w, mTex1[3], tPprime;

MOV oPos, tPprime;
MOV oCol, iCol;

END
```

A.4 Fragment Program

This GL_FRAGMENT_PROGRAM_ARB applies the ‘toon texture in Tex0, and then remaps α according to equation (3.3) using the paper height in Tex1. The media simulation of Section 3.3 is completed by compositing the result with alpha blending.

```
!!ARBfp1.0

ATTRIB iCol = fragment.color;
OUTPUT oCol = result.color;

ATTRIB iTex0 = fragment.texcoord[0];
ATTRIB iTex1 = fragment.texcoord[1];

PARAM cOne = { 1.0, 1.0, 1.0, 1.0 };
PARAM cTwo = { 2.0, 2.0, 2.0, 2.0 };

TEMP tPeak;
TEMP tValley;
TEMP tResult;

### Conventional 2D texture modulation

TXP tResult, iTex0, texture[0], 2D;
MUL tResult, iCol, tResult;

### Peak and valley transfer functions

MUL_SAT tPeak.a, cTwo, tResult;
MAD_SAT tValley.a, cTwo, tResult, -cOne;

### Weighted sum of peak and valleys functions

TXP tResult.a, iTex1, texture[1], 2D;
LRP tResult.a, tResult, tPeak, tValley;

### Pre-multiply by alpha
MUL tResult.rgb, tResult, tResult.a;

### Final result should be composited using
###  glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)

MOV oCol, tResult;

END
```

Appendix B

Epipolar Silhouette Motion

To ensure the frame-to-frame coherence of stylized silhouettes, we assign their parameterization based on samples propagated from the previous frame. Corresponding points on silhouette are established between frames through a search of the epipolar plane, as described in Section 4.4.3. The appropriate search direction is given by equation (4.6), page 55, the displacement $\delta\mathbf{p}$ in the epipolar plane of point $\mathbf{p}$ on the silhouette contour due to camera motion $\delta\mathbf{c}$. We derive this following a similar analysis for suggestive contours presented by DeCarlo et al. [18]. They also provide a brief background in differential geometry that encompasses the notation and concepts applied here, but additional information can be found in [10, 22, 49].

Section 4.3.1 defines the silhouette contour as the zero-set of a scalar function g, evaluated at triangle vertices, and extended to interiors through linear interpolation. Given the camera location $\mathbf{c}$, the silhouette contour consists of points $\mathbf{p}$ that satisfy:

$$\mathbf{v} \;=\; \mathbf{c} - \mathbf{p} \tag{B.1}$$

$$g \;=\; \hat{\mathbf{n}} \cdot \mathbf{v} \;=\; 0 \tag{B.2}$$

We seek to determine how these silhouette contours evolve under camera motion. This can be derived from the implicit function theorem [63], which relates the partial derivatives of implicit functions at level sets. We are interested in the change in position $\mathbf{p}$ with respect to camera location $\mathbf{c}$ at the zero set of g on the surface. One can show that, for a small camera displacement $\delta\mathbf{c}$, such a point will be displaced through the tangent plane in direction $\mathbf{x}$ by:

$$\delta\mathbf{p} \;=\; -\left(\frac{\partial g}{\partial \mathbf{c}} \cdot \delta\mathbf{c}\right)\left(\frac{D_{\mathbf{x}} g}{\|\mathbf{x}\|}\right)^{-1} \frac{\mathbf{x}}{\|\mathbf{x}\|} \tag{B.3}$$

Where $D_{\mathbf{x}}g$ denotes the *directional derivative* of function g on the surface in tangent direction $\mathbf{x}$, while $\partial g/\partial\mathbf{c}$ is a regular derivative with respect to vector $\mathbf{c}$. We are concerned with motion through the epipolar plane, which intersects the tangent plane along the $\mathbf{v}$ direction. Substituting and applying the product rule gives:

$$\delta\mathbf{p} \;=\; -\frac{\left(\dfrac{\partial \hat{\mathbf{n}}}{\partial \mathbf{c}}\,\delta\mathbf{c}\right)\cdot\mathbf{v} \;+\; \hat{\mathbf{n}}\cdot\left(\dfrac{\partial \mathbf{v}}{\partial \mathbf{c}}\,\delta\mathbf{c}\right)}{\left(D_{\mathbf{v}}\hat{\mathbf{n}}\right)\cdot\mathbf{v} \;+\; \hat{\mathbf{n}}\cdot\left(D_{\mathbf{v}}\mathbf{v}\right)}\,\mathbf{v} \tag{B.4}$$

If the camera displacement $\delta\mathbf{c}$ is accompanied by animation that deforms the mesh at point $\mathbf{p}$, we denote this displacement and change in surface normal as $\delta\mathbf{p}_a$ and $\delta\hat{\mathbf{n}}_a$, respectively. The differential quantities can now be expressed as:

$$\begin{aligned}
\frac{\partial \hat{\mathbf{n}}}{\partial \mathbf{c}}\,\delta\mathbf{c} &\;=\; \delta\hat{\mathbf{n}}_{\mathbf{a}} && \text{(B.5)}\\
\frac{\partial \mathbf{v}}{\partial \mathbf{c}}\,\delta\mathbf{c} &\;=\; \delta\mathbf{c} - \delta\mathbf{p}_a && \text{(B.6)}\\
\hat{\mathbf{n}}\cdot\left(D_{\mathbf{v}}\mathbf{v}\right) &\;=\; -\hat{\mathbf{n}}\cdot\mathbf{v} \;=\; 0 && \text{(B.7)}\\
\left(D_{\mathbf{v}}\hat{\mathbf{n}}\right)\cdot\mathbf{v} &\;=\; \mathbf{II}(\mathbf{v},\mathbf{v}) && \text{(B.8)}
\end{aligned}$$

Here, $\mathbf{II}$ is the *second fundamental form* which describes how the local surface changes at $\mathbf{p}$ as a function of direction [22]. We substitute these quantities:

$$\delta\mathbf{p} \;=\; -\frac{\mathbf{v}\cdot\delta\hat{\mathbf{n}}_a \;+\; \hat{\mathbf{n}}\cdot(\delta\mathbf{c} - \delta\mathbf{p}_a)}{\mathbf{II}(\mathbf{v},\mathbf{v})}\,\mathbf{v} \tag{B.9}$$

The second fundamental form yields the *normal curvature* $\kappa_n(\mathbf{x})$ of the surface at $\mathbf{p}$ in tangent direction $\mathbf{x}$ as:

$$\kappa_n(\mathbf{x}) \;=\; \frac{\mathbf{II}(\mathbf{x},\mathbf{x})}{\mathbf{x}\cdot\mathbf{x}} \tag{B.10}$$

But the camera vector $\mathbf{v}$ lies in the tangent plane at $\mathbf{p}$, so this normal curvature is equivalent to the *radial curvature*:

$$\kappa_r \;=\; \kappa_n(\mathbf{v}) \tag{B.11}$$

Collecting terms arising from animated mesh deformation provides the final result:

$$\delta\mathbf{p} \;=\; -\frac{\hat{\mathbf{n}}\cdot\delta\mathbf{c} \;+\; \big(\mathbf{v}\cdot\delta\hat{\mathbf{n}}_a - \hat{\mathbf{n}}\cdot\delta\mathbf{p}_a\big)}{\kappa_r\,\|\mathbf{v}\|^2}\,\mathbf{v} \tag{B.12}$$

For animations that involve only camera displacement this reduces to equation (4.6):

$$\delta\mathbf{p} \;=\; -\frac{\hat{\mathbf{n}}\cdot\delta\mathbf{c}}{\kappa_r\,\|\mathbf{v}\|^2}\,\mathbf{v} \tag{B.13}$$

In practice, we evaluate this using a camera displacement derived from $\mathbf{c}$ and $\mathbf{c}'$ after each are transformed into object space in frames f and f', respectively. This accounts for rigid body animation achieved through variation in an object's world transform matrix. Thus, the example case of the spinning sphere in Figure 4.7 is equivalent to a stationary sphere with camera orbiting in the equatorial plane. However, to account for general animation that deforms the surface, we return to equation (B.12).

Finally, it is interesting to rearrange equation (B.13) into the following format:

$$\delta\mathbf{p} \;=\; -\frac{\hat{\mathbf{n}}\cdot(\delta\mathbf{c}/\|\mathbf{v}\|)}{\kappa_r}\,\hat{\mathbf{v}} \tag{B.14}$$

Now, it becomes evident that the angular velocity $\delta\mathbf{c}/\|\mathbf{v}\|$ of the camera about a point $\mathbf{p}$ on the silhouette governs its motion $\delta\mathbf{p}$ under epipolar correspondences.

Bibliography

[1] Gill Barequet, Christian A. Duncan, Michael T. Goodrich, Wenjing Huang, Subodh Kumar, and Mihai Pop. Efficient Perspective-Accurate Silhouette Computation. *Fourth CGC workshop on Computational Geometry*, October 1999.

[2] William Baxter, Vincent Scheib, Ming Lin, and Dinesh Manocha. dAb: Interactive Haptic Painting with 3D Virtual Brushes. *Proceedings of ACM SIGGRAPH 2001*, pages 461–468, 2001.

[3] James F. Blinn. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications*, 12(4):89–94, 1992.

[4] Lubomir Bourdev. Rendering Non-Photorealistic Strokes with Temporal and Arc-Length Coherence. Master's thesis, Brown University, May 1998. URL: http://www.cs.brown.edu/research/graphics/art/bourdev-thesis.pdf.

[5] David Bourguignon, Marie Paule Cani, and George Drettakis. Drawing for Illustration and Annotation in 3D. In *Computer Graphics Forum*, volume 20:3, pages 114–122. Blackwell Publishers, 2001.

[6] Matthew Brand and Aaron Hertzmann. Style Machines. *Proceedings of SIGGRAPH 2000*, pages 183–192, 2000.

[7] David J. Bremer and John F. Hughes. Rapid Approximate Silhouette Rendering of Implicit Surfaces. In *Proceedings of Implicit Surfaces*, pages 155–164, 1998.

[8] Roberto Cipolla. The Visual Motion of Curves and Surfaces. *Philosophical Transactions of the Royal Society of London*, 356(A):1103–1121, 1998.

[9] Roberto Cipolla and Andrew Blake. Surface Shape from the Deformation of Apparent Contours. *Int. J. Computer Vision*, 9(2):83–112, 1992.

[10] Roberto Cipolla and Peter Giblin. *Visual Motion of Curves and Surfaces*. Cambridge University Press, 2000.

[11] Johan Claes, Fabian Di Fiore, Gert Vansichem, and Frank Van Reeth. Fast 3D Cartoon Rendering with Improved Quality by Exploiting Graphics Hardware. *Image and Vision Computing*, pages 13–18, 2001.

[12] Jonathan M. Cohen, John F. Hughes, and Robert C. Zeleznik. Harold: A World Made of Drawings. *Proceedings of NPAR 2000*, pages 83–90, 2000.

[13] Derek Cornish, Andrea Rowan, and David Luebke. View-Dependent Particles for Interactive Non-Photorealistic Rendering. *Graphics Interface 2001*, June 2001.

[14] Matthieu Cunzi, Joëlle Thollot, Sylvain Paris, Gilles Debunne, Jean-Dominique Gascuel, and Frédo Durand. Dynamic Canvas for Immersive Non-Photorealistic Walkthroughs. In *Proc. Graphics Interface*. A. K. Peters, June 2003.

[15] Brian Curless and Marc Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 303–312. ACM Press, 1996.

[16] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. *Proceedings of SIGGRAPH 97*, pages 421–430, 1997.

[17] Eric Daniels. Deep Canvas in Disney's Tarzan. In *ACM SIGGRAPH 99 Conference Abstracts and Applications*, page 200. ACM Press, 1999.

[18] Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive Rendering of Suggestive Contours with Temporal Coherence. *Proceedings of NPAR 2004 (to appear)*, 2004.

[19] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive Contours for Conveying Shape. *ACM Transactions on Graphics*, 22(3):848–855, July 2003.

[20] Doug DeCarlo and Anthony Santella. Stylization and Abstraction of Photographs. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 769–776. ACM Press, 2002.

[21] Oliver Deussen and Thomas Strothotte. Computer-Generated Pen-and-Ink Illustration of Trees. *Proceedings of SIGGRAPH 2000*, pages 13–18, 2000.

[22] Manfredo do Carmo. *Differential Geometry of Curves and Surfaces.* Prentice Hall, 1976.

[23] Frédo Durand. An Invitation to Discuss Computer Depiction. *Proceedings of NPAR 2002*, pages 111–124, 2002.

[24] Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling Strokes and High-Level Attributes for Interactive Traditional Drawing. In *12th Eurographics Workshop on Rendering*, pages 71–82, London, June 2001.

[25] Alexei A. Efros and William T. Freeman. Image Quilting for Texture Synthesis and Transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, August 2001.

[26] Alexei A. Efros and Thomas K. Leung. Texture Synthesis by Non-Parametric Sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, Corfu, Greece, September 1999.

[27] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, 2nd Edition.* Addison-Wesley, 1990.

[28] William T. Freeman, Josh B. Tenenbaum, and Egon C. Pasztor. An Example-Based Approach to Style Translation for Line Drawings. Technical Report TR99-11, MERL, Cambridge, MA, 1999. http://www.merl.com/papers/TR99-11.

[29] Bert Freudenberg. Real-Time Stroke Textures. In *ACM SIGGRAPH 2001 Conference Abstracts and Applications*, page 252. ACM Press, 2001.

[30] Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering.* A.K. Peters, 2001.

[31] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter S. Shirley, and Rich Riesenfeld. Interactive Technical Illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[32] Donald P. Greenberg, Kenneth E. Torrance, Peter Shirley, James Arvo, Eric Lafortune, James A. Ferwerda, Bruce Walter, Ben Trumbore, Sumanta Pattanaik, and Sing-Choong Foo. A Framework for Realistic Image Synthesis. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 477–494. ACM Press, 1997.

[33] Paul Haeberli. Paint by Numbers: Abstract Image Representations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 207–214. ACM Press, 1990.

[34] Pat Hanrahan and Paul Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes. *Proceedings of SIGGRAPH 90*, pages 215–223, 1990.

[35] Paul Heckbert and Henry Moreton. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.

[36] Aaron Hertzmann. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. *Proceedings of SIGGRAPH 98*, pages 453–460, July 1998.

[37] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image Analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM Press, 2001.

[38] Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. Curve Analogies. In *Proc. of 13th Eurographics Workshop*, pages 233–246, 2002.

[39] Aaron Hertzmann and Ken Perlin. Painterly Rendering for Video and Interaction. In *Proceedings of NPAR 2000*, Annecy, France, June 2000.

[40] Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. In *Proceedings of ACM SIGGRAPH 2000*, pages 517–526, July 2000.

[41] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise Smooth Surface Reconstruction. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pages 295–302. ACM Press, 1994.

[42] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *Proc. of SIGGRAPH 99*, pages 409–416, 1999.

[43] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications*, 23(4):28–37, July 2003.

[44] Tobias Isenberg, Nick Halper, and Thomas Strothotte. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. *Computer Graphics Forum (Proceedings of Eurographics 2002)*, 21(3), 2002.

[45] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by Example: A Statistical Approach. In *Proceedings of NPAR 2002*, pages 29–36. ACM Press, 2002.

[46] Robert D. Kalnins, Philip L. Davidson, Lee Markosian, and Adam Finkelstein. Coherent Stylized Silhouettes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '03)*, 22(3):856–861, 2003.

[47] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '02)*, 21(3):755–762, 2002.

[48] Jan J. Koenderink. What Does the Occluding Contour Tell Us About Solid Shape? *Perception*, 13:321–330, 1984.

[49] Jan J. Koenderink. *Solid Shape.* MIT Press, 1990.

[50] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-Based Rendering of Fur, Grass, and Trees. In *Proceedings of SIGGRAPH 99*, pages 433–438, 1999.

[51] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized Rendering Techniques for Scalable Real-Time 3D Animation. In *Proceedings of NPAR 2000*, pages 13–20, 2000.

[52] John Lansdown and Simon Schofield. Expressive Rendering: A Review of Non-Photorealistic Techniques. *IEEE Comp. Graph. and Apps.*, 15(3):29–37, 1995.

[53] Peter Litwinowicz. Processing images and video for an impressionist effect. *Proceedings of SIGGRAPH 97*, pages 407–414, August 1997.

[54] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Dept. of Mathetmatics, 1987.

[55] Lee Markosian. *Art-Based Modeling and Rendering for Computer Graphics*. PhD thesis, Brown University, May 2000.

[56] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Non-Photorealistic Rendering. *Proceedings of SIGGRAPH 97*, pages 415–420, 1997.

[57] Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-Based Rendering with Continuous Levels of Detail. In *Proceedings of NPAR 2000*, pages 59–66. ACM SIGGRAPH / Eurographics, 2000.

[58] Maic Masuch, Stefan Schlechtweg, and Bert Schönwälder. daLi! – Drawing Animated Lines! In *Proceedings of Simulation und Animation '97*, pages 87–96. SCS Europe, 1997.

[59] Maic Masuch, Lars Schumann, and Stefan Schlechtweg. Animating Frame-to-Frame Coherent Line Drawings for Illustrative Purposes. In *Proceedings of Simulation und Visualisierung*, pages 101–112. SCS Europe, 1998.

[60] Barbara Meier. Computers for Artists Who Work Alone. *Computer Graphics*, 33(1):50–51, February 1999.

[61] Barbara J. Meier. Painterly Rendering for Animation. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 477–484. ACM SIGGRAPH / Addison Wesley, August 1996.

[62] Thomas B. Moeslund and Erik Granum. A Survey of Computer Vision-Based Human Motion Capture. *Computer Vision and Image Understanding: CVIU*, 81(3):231–268, 2001.

[63] James R. Munkres. *Analysis on Manifolds*. Addison Wesley Publish. Co., 1991.

[64] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. *Proceedings of NPAR 2000*, pages 31–38, 2000.

[65] Ken Perlin. An Image Synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*, pages 287–296. ACM Press, 1985.

[66] Mihai Pop, Gil Barequet, Christian A. Duncan, Michael T. Goodrich, Wenjing Huang, and Subodh Kumar. Efficient Perspective-accurate Silhouette Computation. *Proceedings of the 17th Annual ACM Symposium on Computational Geometry (SoCG)*, June 2001.

[67] Pierre Poulin, Karim Ratib, and Marco Jacques. Sketching Shadows and Highlights to Position Lights. In *Proceedings of Computer Graphics International 97*, pages 56–63. IEEE Computer Society, June 1997.

[68] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-Time Hatching. *Proceedings of SIGGRAPH 2001*, pages 579–584, 2001.

[69] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[70] Katherine Pullen and Christoph Bregler. Motion Capture Assisted Animation: Texturing and Synthesis. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 501–508. ACM Press, 2002.

[71] Ramesh Raskar. Hardware Support for Non-photorealistic Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001.

[72] Ramesh Raskar and Michael F. Cohen. Image Precision Silhouette Edges. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 135–140, 1999.

[73] Jarek Rossignac and Maarten van Emmerik. Hidden Contours on a Framebuffer. *Proceedings of 7th Workshop on Computer Graphics Hardware*, 1992.

[74] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-Dependent Reproduction of Pen-and-Ink Illustrations. In *SIGGRAPH '96 Conference Proceedings*, Annual Conference Series, pages 461–468. ACM SIGGRAPH, Addison Wesley, August 1996.

[75] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette Clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 327–334. ACM Press/Addison-Wesley Publishing Co., 2000.

[76] Arno Schödl, Richard Szeliski, David Salisin, and Irfan Essa. Video textures. *Proceedings of SIGGRAPH 2000*, pages 489–498, 2000.

[77] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. Painting with Light. In *Proc. of SIGGRAPH 93*, pages 143–146, 1993.

[78] Joshua Seims. Putting the Artist in the Loop. *Computer Graphics*, 33(1):52–53, February 1999.

[79] Michio Shiraishi and Yasushi Yamaguchi. Image Moment-Based Stroke Placement. Technical Report ID no. skapps3794, University of Tokyo, Tokyo Japan, May 1999.

[80] Michio Shiraishi and Yasushi Yamaguchi. An Algorithm for Automatic Painterly Rendering based on Local Source Image Approximation. In *Non-Photorealistic Animation and Rendering 2000 (Proceedings of NPAR 2000)*, Annecy, France, June, 2000.

[81] Dave Shreiner, Mason Woo, Jackie Neckler, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[82] Mario Costa Sousa and John W. Buchanan. Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. *Computer Graphics Forum (Proceedings of EuroGraphics '99)*, 18(3):195–207, 1999.

[83] Daniel Sperl. Künstlerisches Rendering für Echtzeit-Applikationen (Artistic Rendering for Real-time Applications). Master's thesis, Fachhochschule Hagenberg, Medientechnik, Austria, 2003. http://www.incognitek.com/painterly/.

[84] Thomas Strothotte, Bernard Preim, Andreas Raab, Jutta Schumann, and David R. Forsey. How to render frames and influence people. *Computer Graphics Forum*, 13(3):455–466, 1994.

[85] Thomas Strothotte and Stefan Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*. Morgan Kaufman, 2002.

[86] Osama Tolba, Julie Dorsey, and Leonard McMillan. A Projective Drawing System. In *ACM Symposium on Interactive 3D Graphics*, pages 25–34, March 2001.

[87] Greg Turk. Texture Synthesis on Surfaces. In *Proceedings of SIGGRAPH 2001*, pages 347–354. ACM Press, 2001.

[88] Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Fine Tone Control in Hardware Hatching. In *NPAR 2002: Second International Symposium on Non-Photorealistic Rendering*, pages 53–58, June 2002.

[89] Li-Yi Wei and Marc Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. *Proc. SIGGRAPH 2000*, pages 479–488, 2000.

[90] Li-Yi Wei and Marc Levoy. Texture Synthesis Over Arbitrary Manifold Surfaces. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 355–360. ACM Press, 2001.

[91] Wikipedia, The Free Enclyclopedia. Search topic: WYSIWYG, March 2004. http://en.wikipedia.org/wiki/WYSIWYG.

[92] Georges Winkenbach and David H. Salesin. Computer-Generated Pen-and-Ink Illustration. *Proceedings of SIGGRAPH '94*, pages 91–100, 1994.

[93] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. In *SIGGRAPH '96 Conference Proceedings*, Annual Conference Series, pages 469–476. ACM SIGGRAPH, Addison Wesley, August 1996.

[94] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and Shape Synthesis on Surfaces. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 301–312. Springer-Verlag, 2001.

[95] Robert Zeleznik, Kenneth Herndon, and John F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In *Proceedings of SIGGRAPH '96*, Annual Conference Series, pages 469–476. ACM SIGGRAPH, Addison Wesley, August 1996.