WYSIWYG NPR: Drawing Strokes Directly on 3D Models

Robert D. Kalnins[1] Lee Markosian[1] Barbara J. Meier[2] Michael A. Kowalski[2] Joseph C. Lee[2]
Philip L. Davidson[1] Matthew Webb[1] John F. Hughes[2] Adam Finkelstein[1]

[1]Princeton University [2]Brown University

Abstract

We present a system that lets a designer directly annotate a 3D model with strokes, imparting a personal aesthetic to the non-photorealistic rendering of the object. The artist chooses a "brush" style, then draws strokes over the model from one or more viewpoints. When the system renders the scene from any new viewpoint, it adapts the number and placement of the strokes appropriately to maintain the original look.

Keywords: Interactive techniques, non-photorealism.

1 Introduction

Artists and designers apply techniques of 3D computer graphics to create images that communicate information for some purpose. Depending on that purpose, photorealistic imagery may or may not be preferred. Thus, a growing branch of computer graphics research focuses on techniques for producing non-photorealistic renderings (NPR) from 3D models. Strong arguments for the usefulness of this line of inquiry are made by many researchers (e.g., [Durand 2002; Lansdown and Schofield 1995; Meier 1996; Strothotte et al. 1994; Winkenbach and Salesin 1994]).

Much of the research in NPR has targeted a particular style of imagery and developed *algorithms* to reproduce that style when rendering appropriately-annotated 3D scenes. Relatively little emphasis has been placed on the separate problem of how to provide direct, flexible *interfaces* that a designer can use to make those annotations in the first place. Instead, the usual approach is to rely on complex scripting or programming. Meier [1999] and Seims [1999] argue that effective interfaces are essential for these algorithms to be accepted by content creators. One reason is that NPR imagery must often reflect a designer's judgments regarding what details to emphasize or omit. Thus, a key challenge facing NPR researchers is to provide algorithms coupled with direct user interfaces that together give designers flexible control over the look of a scene. In this paper we begin to address this challenge in the context of interactive NPR for 3D models.

Copyright © 2002 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212-869-0481 or e-mail permissions@acm.org.
© 2002 ACM 1-58113-521-1/02/0007 $5.00

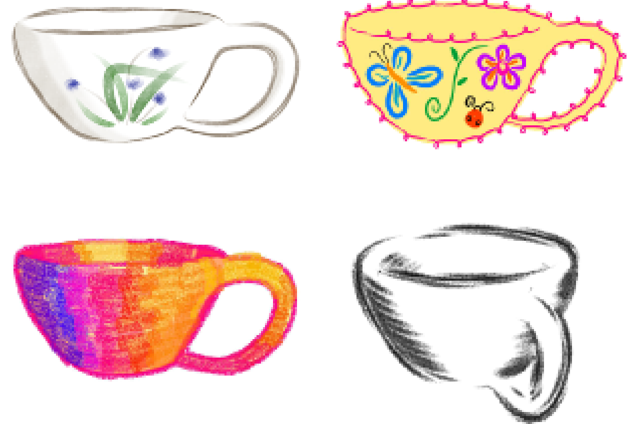


Figure 1: Artists directly annotated the same 3D teacup model to produce four distinct rendering styles.

We present a system called WYSIWYG NPR, for "what you see is what you get non-photorealistic rendering." We focus on stroke-based rendering algorithms, with three main categories of strokes: (1) silhouette and crease lines that form the basis of simple line drawings; (2) decal strokes that suggest surface features, and (3) hatching strokes to convey lighting and tone. In each case, we provide an interface for direct user control, and real-time rendering algorithms to support the required interactivity. The designer can apply strokes in each category with significant stylistic variation, and thus in combination achieve a broad range of effects, as we demonstrate in both figures and video.

In this paper we focus on tools that give the artist control over the *look* of a scene. We also provide limited control over how strokes *animate* during level of detail transitions, recognizing that the ideal system would provide more complete control over the animation of strokes as a design element in its own right.

The applications for this work are those of NPR, including architectural and product design, technical and medical illustration, storytelling (e.g., children's books), games, fine arts, and animation.

The main contributions of this paper are to identify the goal of providing direct control to the user as being key to NPR, and to demonstrate with a working system the payoff that can result from targeting this problem. In support of this goal, we offer several new NPR algorithms, including improved schemes for detecting and rendering silhouettes, an algorithm for synthesizing stroke styles by example, methods for view-dependent hatching under artistic control, and an efficient technique for simulating various types of natural media such as pencil or crayon on rough paper.

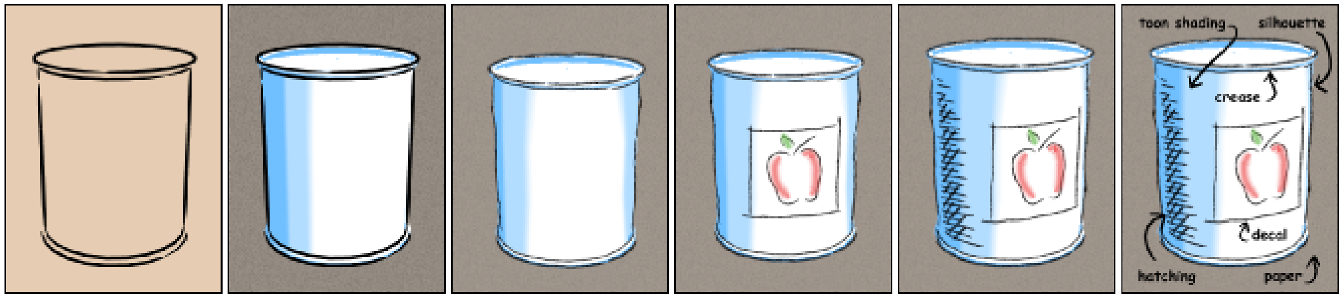


Figure 2: Example session (L to R): load model; shading and paper; stylize outlines; add decals; hatching; labeled features.

2 Related Work

The last decade has seen a blossoming of work on NPR algorithms in a variety of styles. For a survey, see Gooch and Gooch [2001]. Much of this work addresses the production of still images, while some systems for rendering 3D scenes have addressed the challenge of providing temporal coherence for animations [Deussen and Strothotte 2000; Meier 1996]. Our work falls into the latter category, within the subset of systems that address *interactive* rendering (e.g., [Gooch et al. 1999; Kowalski et al. 1999; Lake et al. 2000; Markosian et al. 2000; Praun et al. 2001]), wherein the challenge is to maintain temporal coherence, using limited run-time computation, when camera paths are not known in advance.

Most work in NPR has focused on algorithms that are controlled by parameter setting or scripting—the designer has no direct control over where marks are made. An inspiration for our work is the technique for direct WYSIWYG painting on 3D surfaces proposed by Hanrahan and Haeberli [1990], which is now available in various commercial modeling systems. These tools let the designer paint texture maps *directly* onto 3D models by projecting screen-space paint strokes onto the 3D surface and then into texture space, where they are composited with other strokes. Strokes then remain fixed on the surface and do not adapt to changes in lighting or viewpoint. In contrast, strokes in our system are automatically added, removed, or modified in response to changes in lighting or viewing conditions.

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 allow an artist to simply draw in the image plane and thereby express a complete, stylized 3D world. The systems of Tolba et al. [2001], Cohen et al. [2000], and Bourguignon et al. [2001], each pursue this goal by making simplifying assumptions about the underlying geometry. In contrast, we start with a 3D model, and draw directly on it. Our goal is eventually to integrate the WYSIWYG NPR interface directly into a modeling system, ideally one that uses a drawing interface for constructing geometry, like SKETCH [Zeleznik et al. 1996] or Teddy [Igarashi et al. 1999].

Previous efforts have addressed finding silhouettes on 3D models and rendering them with stylization [Markosian et al. 1997; Masuch et al. 1997; Northrup and Markosian 2000]. Other systems have addressed generating hatching or structured patterns of strokes on surfaces to convey tone or texture [Hertzmann and Zorin 2000; Mitani et al. 2000; Praun et al. 2001; Winkenbach and Salesin 1994]. In our system the challenge is to synthesize strokes with temporal coherence, in real time, based on a user-given style, where the user specifies where particular strokes should be placed.

3 An Interaction Example

To give a sense of the system from the artist's viewpoint, we now briefly describe the organization of the user interface, and then narrate a step-by-step interaction session.

To support a *direct* interface for creating stroke-based NPR styles, we provide a pen and tablet for drawing input. Pressure data from the tablet pen can be used to vary stroke attributes such as width and darkness. The UI makes use of conventional menus and dialog boxes as needed.

The system has three editing modes. In the first, the artist can position each object and set its "base coat" (described in Section 4.1). In outline mode the artist can draw decals and stylize silhouettes and creases. In hatching mode, he can draw hatching strokes. In any mode, the artist can modify the current "brush style" that affects stroke properties (see Section 4.2). In practice, the artist will carefully design combinations of parameters that together produce a desired look, then save that brush style to use again in future sessions. The artist can also select a background color or image for the scene, and control the directional lighting that affects the "toon" shaders used for object base coats. All color selections are made via HSV sliders.

We now describe how an artist annotates a simple 3D model to depict a stylized fruit can. Stages of the process are shown in Figure 2; the names of various design elements appear in the rightmost image.

1. We begin by loading a model of a can, which is initially displayed in a generic silhouette style with a tan base coat and background color.

2. We choose a toon base coat for the model from a previously-created list. Next we set the background color to gray, and apply a previously created paper texture to it. We adjust the lighting so the right side of the can is lit.

3. From a list of previously saved brushes, we choose a black pencil style. We select a silhouette, which highlights in yellow. We draw over the highlighted portion with sketchy strokes and then click an "apply" button to propagate the sketchy style to all silhouettes. Next we select a crease where the lip of the can meets the cylindrical part, oversketching this and other creases in a sketchy style similar to that of the silhouettes.

4. To add a label to the can, we draw decal strokes directly on the surface, moving the viewpoint and changing the color and other attributes of the brush as needed.

5. We now switch to hatching mode and draw roughly parallel lines where we want shading to appear. To finish the hatch group, we tap the background. We create a second set at an angle to the first and tap again. The drawing is done!

4 Rendering Basics

The models accepted by our system are triangle meshes (possibly animated). A mesh is divided into one or more *patches*, each rendered with several procedural textures, or *shaders*. One shader draws a "base coat," another handles creases, a third draws silhouettes, and a fourth applies hatching strokes.

4.1 Background and Base Coat

The focus of our work is to let the designer annotate a model with strokes, but there are two other elements under user control that provide significant flexibility. First, the designer can choose a background color or image that fills the rendering window (e.g. the "sky" in Figure 8.) Second, the designer can select for each patch a *base coat* – a shader that draws the triangles of the patch in some style. Examples in this paper include drawing the patch using cartoon (or "toon") shading as in Figure 1, upper left, or in a solid color as in Figure 1, upper and lower right. As described by Lake et al. [2000], the designer can create custom 1D texture maps containing the color spectrum for each toon shader. Our 1D toon textures may include an alpha component to be used by the media simulation algorithm of Section 4.4, as in Figure 3. In such cases the base coat is typically rendered in two passes: an opaque layer followed by the toon shader.

4.2 Strokes

Our stroke primitive is based on that of Northrup and Markosian [2000]. The main difference is that the path of the stroke, called the *base path*, is represented as a Catmull-Rom spline. Elements under user control are: color, alpha, width, the degree to which strokes taper at their endpoints, and "halo" (the degree to which a stroke is trimmed back at an endpoint due to occlusion). These parameters are controlled by sliders, though width and alpha can be made to vary along the stroke due to pressure data from the tablet. The designer may choose an optional 1D texture specifying the image (including alpha) of a "cross-section" of the stroke. (We found that using a 2D texture parameterized by stroke length and width led to temporal artifacts arising from abrupt changes in stroke lengths.) The stroke is rendered as a triangle strip that follows the base path and matches the varying width, alpha, and color or texture of the stroke.

As will be described in Section 5, we may further stylize a silhouette or crease stroke using an *offset list* that represents small-scale wiggles relative to the base path [Markosian et al. 1997]. Each offset records a parametric position along the base path and a perpendicular screen-space displacement from that point (in pixels). The offset list may progress forward and backward along the stroke, and may contain breaks. To render the offset list, we map it along the base path to define one or more triangle strips in image space.

4.3 Stroke Visibility

The z-buffer cannot directly compute visibility for strokes because the triangle strip for a *stylized* base path would generally interpenetrate the surface. Instead, we adopt the method of Northrup and Markosian [2000], which resolves visibility using an *ID reference image*. Into this off-screen buffer, we render the mesh faces, crease edges, silhouette polylines – each in a unique color. The visible portions of a particular stroke are found by sampling the ID reference image along the base path, checking for the correct color.

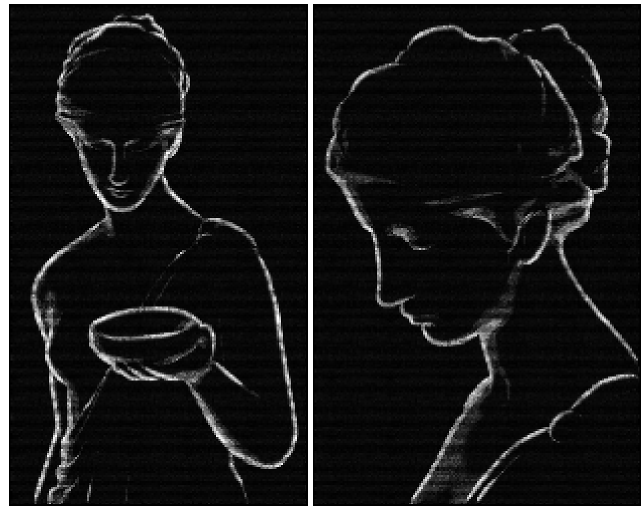


Figure 3: Wide silhouette strokes and a subtle toon shader are rendered over a coarse paper texture.

4.4 Media Simulation

To simulate natural media, the artist can apply a *paper effect* to any semi-transparent scene primitive: background image, toon shader, or stroke. We modulate the per-pixel color of the primitive during scan conversion by the texel of a chosen *paper texture*. Conceptually following Curtis et al. [1997] and Durand et al. [2001], 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 color. For peaks we use the transfer function $p(\alpha) = \texttt{clamp}(2\alpha)$, and for valleys we use $v(\alpha) = \texttt{clamp}(2\alpha - 1)$. For intermediate height h we use the interpolated function $t(\alpha) = p(\alpha)h + v(\alpha)(1 - h)$. We composite the incoming color into the framebuffer using the standard "over" operation with transparency $t(\alpha)$. We have implemented this simple algorithm as a pixel program on our graphics card. The paper effect is clear in Figures 3 and 4.

Durand et al. [2001] describe a similar strategy, using 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. Because we re-map alpha instead of tone, our method more flexibly handles arbitrary colors in both source and destination pixels.

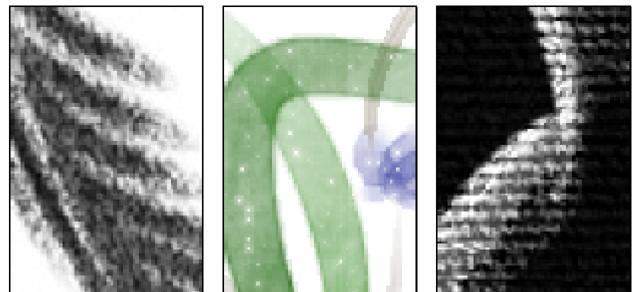


Figure 4: Details of paper effect from Figures 1 and 3.

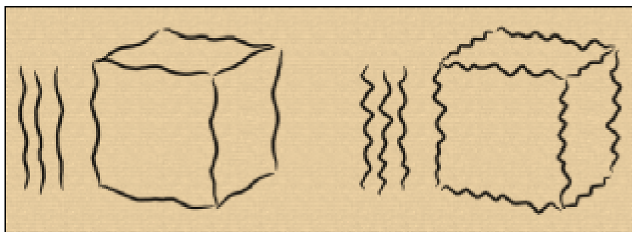


Figure 5: The same cube rendered in two different styles. Crease strokes were synthesized from the examples shown.

5 Decals, Creases, and Silhouettes

In this section we treat the placement, modification and rendering of individual strokes. Section 6 will address level of detail issues relating to management of groups of strokes.

5.1 Decal Strokes

The simplest annotation in our system is the "decal" stroke. The artist draws a stroke directly onto a surface, and it sticks like a decal. These strokes were used to draw the flowers and the butterfly on the upper cups in Figure 1.

The interface for decal strokes was inspired by the system of Hanrahan and Haeberli [1990], which projects strokes into a texture map, and renders them using conventional texture mapping. In contrast, our system represents decal strokes as spline curves whose control points are projected onto the surface and rendered as described in Section 4.2. Our representation offers three benefits over texture mapping. First, decal strokes do not require a parameterization of the surface. Second, we avoid sampling artifacts at magnified and oblique views. Third, with texture mapping the apparent brush size depends on the obliquity of the surface, whereas we can maintain the screen-space width of strokes to be consistent with the artist's brush choice under all views.

5.2 Crease Strokes

The artist may also annotate *creases* – chains of mesh edges that follow sharp features on the model. Such features are manually tagged by the modeler or automatically found by thresholding dihedral angles in the mesh. The strokes in Figures 5 and 6 all arise from annotated creases.

When the artist oversketches a chosen crease, perhaps more than once, the system records these "detail" functions in offset lists as described in Section 4.2. For every vertex of the input stroke, an offset is recorded as a perpendicular pixel displacement from the nearest point along the arc-length parameterized crease path. The crease endpoints are extended along the respective tangents to accommodate sketching beyond the crease.

In subsequent views, the parametric positions of the offsets remain fixed. However, we shrink the offsets when the model shrinks in screen space, because we found this to appear more natural than using fixed-magnitude offsets. We take the ratio σ_c of the current screen-space length of the crease to that when it was oversketched, and scale the magnitudes of the offsets by $\mathtt{min}(\sigma_c, 1)$. I.e., we scale them down but never up.

Figure 6: Victorian Storefront. Four example strokes were used to synthesize strokes along all creases, combining the best of both worlds: direct drawing to create the look, and automation to avoid the tedium of sketching every stroke.

5.3 Synthesizing Strokes by Example

Since some models contain many creases, we provide two techniques for automatically assigning them offsets based on examples provided by the artist. The first technique, *rubber-stamping,* repeats a sequence of example offsets along each crease, using arc-length parameterization.

To produce less obviously repetitive strokes that still resemble the example strokes, we provide a second technique that synthesizes new strokes from a given set of example strokes. Freeman et al. [1999] describe a method of transferring stylization from a set of example strokes to a new line drawing. Their method requires a large set of examples (typically over 100), with each example stroke given in each of the various "styles" to be supported. In contrast, the method we describe below works with just a few example strokes (we typically use three or four). This is possible because we separate each stroke into an unstylized "base path" plus detail "offsets" as described in Section 4.2. We perform one-dimensional texture synthesis on the stroke offsets using Markov random fields, and can apply the result to any new stroke base path. To maintain variety, we synthesize a new offset list for each stroke requiring stylization.

Markov random fields have been recently used in graphics in other data-synthesis applications [Brand and Hertzmann 2000; Efros and Leung 1999; Wei and Levoy 2000]. Our algorithm closely follows the "video textures" algorithm presented in Schödl et al. [2000]; rather than synthesize new sequences of frames from an example video, we synthesize new sequences of stroke 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 stroke; the transition probability between two states is a function of the local stroke similarity between the two points. With this method, we generate offset sequences containing features of the example strokes, as shown in Figures 5 and 6. Implementation details are given in Appendix A.

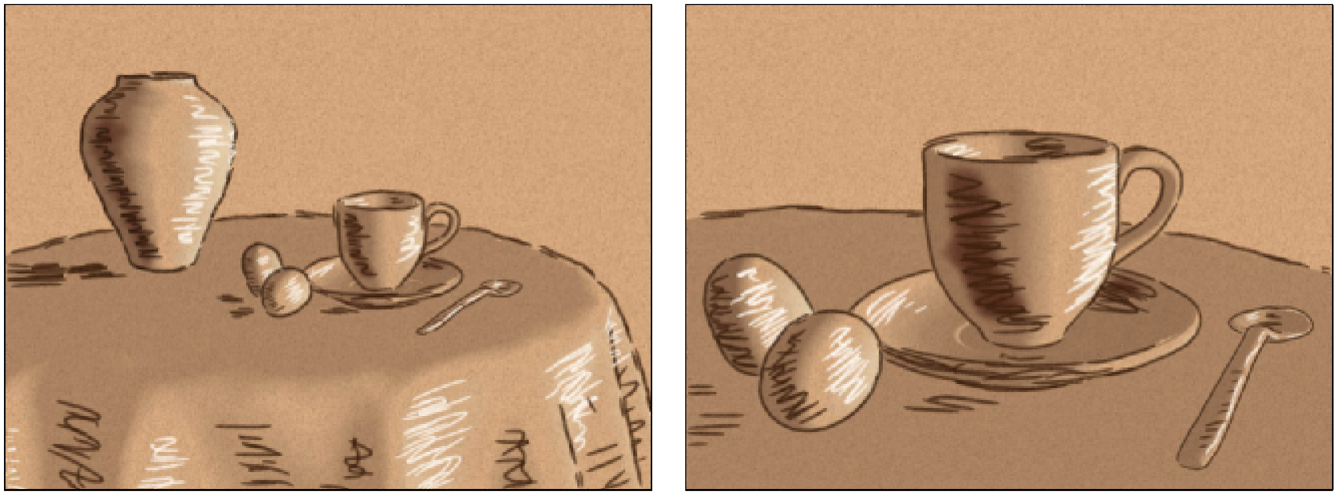


Figure 7: The artist chose sepia-toned paper and brown and white conte crayon brushes to create this breakfast scene. The toon shading, which is translucent brown in the dark areas and completely transparent in the light areas, implies an ink wash. Multiple levels of detail are revealed in the free hatching when the camera zooms in for a closer view (right).

5.4 Silhouettes

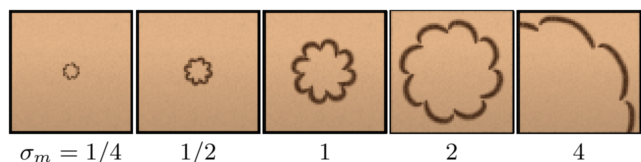
Unlike creases, silhouettes are view dependent: their number, locations, and lengths vary from one frame to the next. It is not obvious how the artist could annotate individual silhouettes with unique stylizations that persist over time. Therefore, our tools let the artist sketch a *single* prototype stroke that is rubber-stamped wherever silhouettes are found. In most other respects, silhouettes behave like creases from the point of view of the artist.

We have adapted the silhouette detection algorithm of Markosian et al. [1997], which finds networks of silhouette edges on the mesh. Our goal is to turn visible portions of these networks into strokes. While the silhouette generally looks smooth, it often bifurcates and zig-zags in 3D and therefore backtracks in the image plane – an obstacle for the construction of smooth strokes along the path. Northrup and Markosian [2000] propose a set of heuristics for cleaning up the resulting 2D paths before forming strokes.

Hertzmann and Zorin [2000] use an alternate definition of silhouettes that avoids such problems. For a given vertex v with normal $\mathbf{n}_v$ and vector $\mathbf{c}_v$ to the camera, we define the scalar field $f(v) = \mathbf{n}_v \cdot \mathbf{c}_v$, extending f to triangle interiors by linear interpolation. Silhouettes are taken to be the zero-set of f, yielding clean, closed polylines whose segments traverse faces in the mesh (rather than following edges, as in the Markosian and Northrup methods.) Hertzmann and Zorin also describe a fast, deterministic method for finding these silhouettes at runtime, based on a pre-computed data structure. A drawback is that their method is not suitable for animated meshes (e.g. Figure 10). Therefore, we use their silhouette definition, but adapt the stochastic method for finding the polylines, as follows. Sample a small number of faces in the mesh. At each, test the sign of the function f at its three vertices; if they are not all the same, then the silhouette must cross this triangle and exit from two of the three sides into neighboring triangles; locate those triangles and continue tracing until returning to the starting triangle. (If f is *exactly* zero at a vertex, slightly perturbing the normal removes the degeneracy.) We cull "back-facing" segments by checking if ∇f points away from the camera, and test visibility of the remaining portions as in Section 4.3.

Since we render silhouettes with stylization (Section 4.2), assigning them consistent parameterization is critical for temporal coherence. This is easy for creases because they are fixed on the model. But silhouettes are view-dependent and do not necessarily have correspondence from frame to frame. This paper does not fully address the challenge of assigning consistent parameterization from one frame to the next. Nevertheless, we adopt a simple heuristic described by Bourdev [1998]. We begin by assigning all strokes arc-length parameterization. In each frame, we sample visible silhouettes, saving for each sample its surface location and parameter value (with respect to the stylization). In the next frame, we project the sample from 3D into the image plane. Then, to find nearby silhouette paths, we search the ID reference image (Section 4.3) by stepping a few pixels along the surface normal projected to image space. If a new path is found, we register the sample's parameter value with it. Since each new path generally receives many samples, we use a simple voting and averaging scheme to parameterize it; more rigorous analysis of clustering or voting schemes merits future work.

Once we have parameterized the silhouette paths, we can apply stylization as we do with creases (Section 5.2), with one notable difference. Because the silhouette paths have varying lengths, we truncate or repeat the stylization to fully cover each path. When the artist oversketches a silhouette to provide stylization, the length of the oversketched base path is taken as the arc-length period for repeating the offset pattern. We scale the period and magnitude of the silhouette offsets just as for creases (Section 5.2), except for the definition of the scaling factor (which for a crease depends on a fixed set of edges). Instead we use the ratio σ_m of the screen-space model diameter to that when the oversketch was performed. We scale the offset magnitudes only when zoomed out, using $\mathtt{min}(\sigma_m, 1)$, as on this sphere:



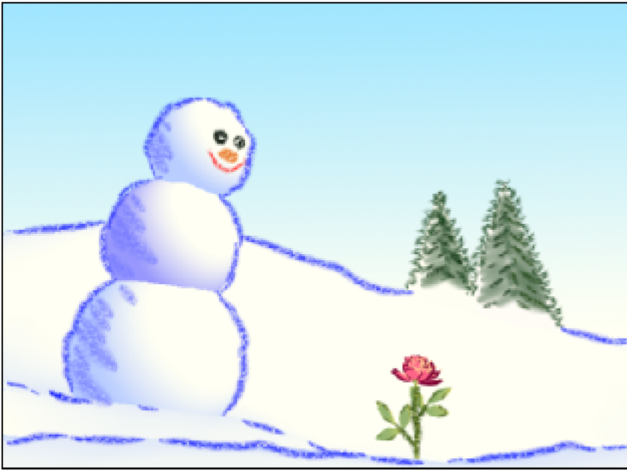


Figure 8: A toon shader and mobile hatching suggest lighting on the snowman, decal strokes depict a face, and uneven blue crayon silhouettes imply bumpy snow. Conical trees are annotated with sawtooth crayon silhouettes plus two layers of mobile hatching.

6 Hatching

Our system provides tools for drawing several forms of *hatching* – groups of strokes drawn on the surface that collectively convey tone, material or form.

6.1 Structured Hatching

In illustrations, perhaps the most common form of hatching could be characterized as a group of roughly parallel strokes. We call such annotations *structured hatching* and leverage their regular arrangement to provide automatic level-of-detail (LOD) control. First, the artist draws a group of roughly parallel and regularly-spaced strokes from a particular vantage point. Then, he may move the camera closer or further to adjust the screen-space stroke density before "locking" it with a button. In subsequent views, the system attempts to maintain the chosen density by successively doubling (or halving) the stroke count when the group size doubles (or halves). To compensate for the discrete nature of these LODs, we modulate stroke widths between levels.

In each frame we calculate a factor σ_h that approximates the multiplier on the original stroke count necessary to maintain density. We take σ_h to be the ratio of the current size of the stroke group (measured transverse to the strokes) to that when it was locked. Figure 9 shows the first LOD transition as the camera approaches the locked stroke group. The strokes do not double until σ_h reaches the threshold t_+ (1.6 here). For $\sigma_h \in [1.0, t_+)$ the strokes thicken as their density falls. When σ_h reaches t_+ a brief animation (0.5s by default) begins, during which existing strokes narrow while new strokes grow in the interstices, as shown in the middle three images. New stroke paths linearly interpolate those of their neighbors. Finally, for $\sigma_h \in (t_+, 2.0]$ strokes thicken again to eventually reach the original width. The doubling process repeats analogously for larger $\sigma_h \in [2^n, 2^{n+1}]$. If the camera path is reversed (zooming out) the process is inverted using a threshold $t_- < t_+$, providing hysteresis for stability. The designer may set a maximum LOD after which strokes simply fatten rather than divide.

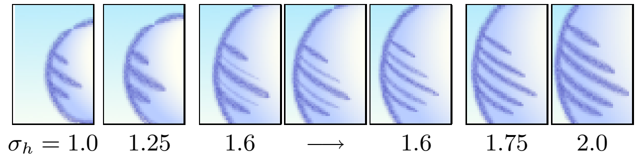


Figure 9: As the camera approaches a hatch group, level of detail effects preserve stroke density. See Section 6.1.

6.2 Free Hatching

For some effects, an artist may find structured hatching too constraining. Therefore, we also provide *free hatching* with which the artist draws arbitrary arrangements of strokes to explicitly build LODs. See, for example, Figure 7.

The scheme is simple. As before, the artist draws a set of strokes from a particular view, and "locks" them at the desired density. Then he moves to a closer view where higher density is desired and adds more strokes, choosing whether the new stroke group *replaces* or *augments* the previous. This is repeated until the artist builds sufficient LODs. For novel views, LOD transitions are handled using the same policy as for structured hatching. (The user-adjustable constants t_+ and t_- are particularly useful for tuning free-hatching LOD transitions.) However, there is no obvious way to measure σ_h for free hatching, so instead we use the ratio σ_m based on mesh sizes (Section 5.4). When the first LOD is drawn, $\sigma_m = 1$. Each additional LOD is associated with the value of σ_m at the time it is drawn.

6.3 Mobile Hatching

Artists often use shading near outlines, suggestive of light cast from somewhere behind the camera. To achieve this effect when an object or camera animates, the hatching must move on the surface. Our system provides a tool for annotating the model with such dynamic strokes, which we call *mobile hatching*. This effect is used, for example, on the snowman Figure 8, and readers with access to the accompanying video can see it in action.

Like *stationary hatching*, described in Sections 6.1 and 6.2, mobile hatching may also be either structured or free. The artist enters "mobile mode" and then sketches hatch groups as before, but now each group implies a "directional light." The model we use is simple, and applies equally to hatching suggestive of either highlights or shadows. From the drawn group, we infer a light direction opposite to the local surface normal. 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 on the local surface. As the view changes, mobile hatch groups move over the surface consistent with this pseudo lighting rule.

Our implementation is presently limited to smooth surface regions with roughly uniform uv-parameterization, and we further constrain the motion of mobile hatch groups to straight lines in either u or v – let's say u. When the artist completes a mobile hatching group, the system (1) projects the strokes into uv-space, (2) computes their convex hull, (3) finds its centroid c, and (4) records the normal $\hat{n}_c$ of the surface at c. The parametric path over which hatch groups travel is taken to be the line in u passing through c. For new views, we sample the surface normal $\hat{n}(u)$ along the line, and evaluate a Lambertian lighting function $\ell(u) = \hat{n}_c \cdot \hat{n}(u)$. Prior to obtaining either $\hat{n}_c$ or $\ell(u)$, we smooth $\hat{n}(u)$ with a filter whose kernel has the extent of the hatch group, to

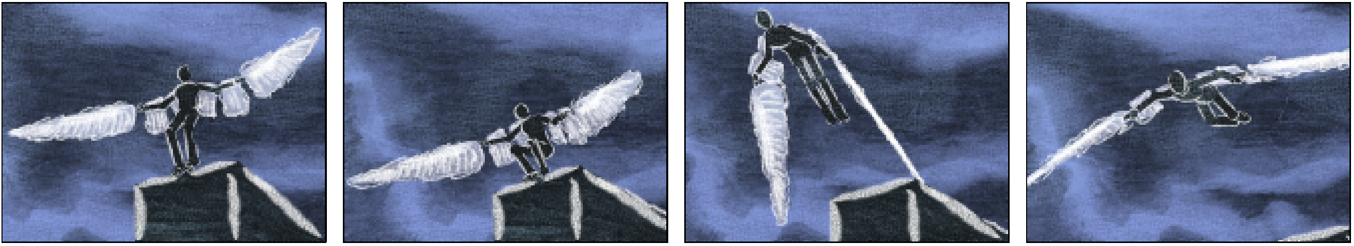


Figure 10: Four frames from an animation rendered by our system. Wings are drawn with semi-transparent free hatching.

reduce sensitivity to minor 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 $\frac{1}{2}$) that prevents hatch groups 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 avoid "popping."

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 it 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 research. Other researchers, for example, Schoeneman et al. [1993] and Poulin et al. [1997], have addressed this problem with greater rigor in the context of photorealism.

7 Results and Discussion[1]

The greatest strength of this system is the degree of control given to the artist: the choice of brush styles and paper textures, background and basecoats, and the look and location of each mark. In our experience, working with each new style requires a learning period but, as with a traditional medium, it becomes natural once mastered.

We find that complex geometry offers detail that can be "revealed" through the more automatic features of our system (e.g., toon shaders or silhouettes), whereas simple geometry offers a blank slate on which to create new details where none exist. For example, in Figure 6, we simplified the appearance of the intricate building with a sparse line drawing representation. Four example strokes were used to automatically synthesize detail over the 8,000 crease edges in the model. This scene was completed in under fifteen minutes, including time to experiment with media, textures, and lighting. In contrast, the snowman scene in Figure 8 called for stroke-based detail to augment its simple geometry (spheres and cones). This took about an hour to complete.

For interactive exploration, the artist may need to design appropriate detail to ensure that a scene is "interesting" from disparate views. In the scene shown in Figure 7, we created three LODs for each object and also annotated the "backside," so of course the drawing took longer to create than a still image. However, the scene in Figure 8 did not have such additional overhead because mobile, structured hatching automatically adapts to a moving camera.

While most of the images in this paper were captured from interactive exploration of static scenes, our system also supports offline rendering of scenes with animated geometry, for example the winged character in Figure 10.

[1]The electronic version of this paper contains an Appendix B with additional results.

While the system supports a diversity of styles, it does not work well for those based on short strokes, such as stippling or pointillism. Furthermore, silhouette stylization presently cannot depict features local to a particular surface region, since the style applies globally. Also, the system does not yet support object interplay such as shadows or reflections.

Table 1 reports model complexity and typical frame rates for various models rendered with a 1.5 GHz Pentium IV CPU and Geforce3 GPU. All of the models render at interactive frame rates, except for the building which uses a large number of strokes. Stroke count has the greatest influence on performance, while polygonal complexity has limited impact. Reading back the ID reference image (Section 4.3) imposes a significant but fixed overhead, so that trivial models render at only roughly 25 fps.

Figure	Faces (K)	Strokes	Frames/sec
1: cup	5	25	20
3: statue	120	100	10
6: building	16	7,000	3
7: breakfast	25	400	9
8: snowman	10	250	11

Table 1: Frame rates and sizes of various models.

8 Conclusion and Future Work

We have demonstrated a system for drawing stroke-based NPR styles directly on 3D models. The system offers control over brush and paper styles, as well as the placement of individual marks and their view-dependent behavior. Combined, these afford a degree of aesthetic flexibility not found in previous systems for creating stylized 3D scenes. We presented new algorithms for finding and rendering silhouettes, synthesizing stroke detail by example, simulating natural media, and hatching with dynamic behavior.

As future work, we hope to address some of the limitations of our system and extend it, for example, to encompass such styles as stippling and pointillism. We believe that the stroke synthesis currently available for annotating creases could be adapted to create regions of hatching strokes or other structured patterns based on artist example, thus reducing the tedium of creating every stroke manually. We would also like to consider how we can help designers show object-to-object interactions such as shadows, and create artistic effects like smudging one object into another. Most important is to put these tools in the hands of artists.

Acknowledgments

We thank Carl Marshall and Adam Lake for encouragement and advice, and Trina Avery for her super-human edit. This research was supported by Intel Labs, the Alfred P. Sloan Foundation, and the NSF (CISE 9875562, EIA 8920219).

References

BOURDEV, L. 1998. *Rendering Nonphotorealistic Strokes with Temporal and Arc-Length Coherence*. Master's thesis, Brown University. www.cs.brown.edu/research/graphics/art/bourdev-thesis.pdf.

BOURGUIGNON, D., CANI, M. P., AND DRETTAKIS, G. 2001. Drawing for illustration and annotation in 3D. In *Computer Graphics Forum*, Blackwell Publishers, vol. 20:3, 114–122.

BRAND, M., AND HERTZMANN, A. 2000. Style machines. *Proceedings of SIGGRAPH 2000*, 183–192.

COHEN, J. M., HUGHES, J. F., AND ZELEZNIK, R. C. 2000. Harold: A world made of drawings. *Proceedings of NPAR 2000*, 83–90.

CURTIS, C. J., ANDERSON, S. E., SEIMS, J. E., FLEISCHER, K. W., AND SALESIN, D. H. 1997. Computer-generated watercolor. *Proceedings of SIGGRAPH 97*, 421–430.

DEUSSEN, O., AND STROTHOTTE, T. 2000. Computer-generated pen-and-ink illustration of trees. *Proc. of SIGGRAPH 2000*, 13–18.

DURAND, F., OSTROMOUKHOV, V., MILLER, M., DURANLEAU, F., AND DORSEY, J. 2001. Decoupling strokes and high-level attributes for interactive traditional drawing. In *12th Eurographics Workshop on Rendering*, 71–82.

DURAND, F. 2002. An invitation to discuss computer depiction. *Proceedings of NPAR 2002*.

EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

FREEMAN, W. T., TENENBAUM, J. B., AND PASZTOR, E. 1999. An example-based approach to style translation for line drawings. Tech. Rep. TR99-11, MERL, Cambridge, MA. http://www.merl.com/papers/TR99-11.

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

GOOCH, B., SLOAN, P.-P. J., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics*, 31–38.

HANRAHAN, P., AND HAEBERLI, P. 1990. Direct WYSIWYG painting and texturing on 3D shapes. *Proc. of SIGGRAPH 90*, 215–223.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, 517–526.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3d freeform design. *Proc. of SIGGRAPH 99*, 409–416.

KOWALSKI, M. A., MARKOSIAN, L., NORTHRUP, J. D., BOURDEV, L., BARZEL, R., HOLDEN, L. S., AND HUGHES, J. F. 1999. Art-based rendering of fur, grass, and trees. *Proc. SIGGRAPH 99*, 433–438.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3D animation. *Proceedings of NPAR 2000*, 13–20.

LANSDOWN, J., AND SCHOFIELD, S. 1995. Expressive rendering: A review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications 15*, 3, 29–37.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97*, 415–420.

MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., HOLDEN, L. S., NORTHRUP, J. D., AND HUGHES, J. F. 2000. Art-based rendering with continuous levels of detail. *Proc. of NPAR 2000*, 59–66.

MASUCH, M., SCHLECHTWEG, S., AND SCHÖNWÄLDER, B. 1997. daLi! – Drawing Animated Lines! In *Proceedings of Simulation und Animation '97*, SCS Europe, 87–96.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of SIGGRAPH 96*, ACM SIGGRAPH / Addison Wesley, Computer Graphics Proceedings, Annual Conference Series, 477–484.

MEIER, B. 1999. Computers for artists who work alone. *Computer Graphics 33*, 1 (February), 50–51.

MITANI, J., SUZUKI, H., AND KIMURA, F. 2000. 3D Sketch: Sketch-based model reconstruction and rendering. *Seventh IFIP WG 5.2 Workshop on Geometric Modeling*. http://kaemart.unipr.it/geo7/.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic silhouettes: A hybrid approach. *Proceedings of NPAR 2000*, 31–38.

POULIN, P., RATIB, K., AND JACQUES, M. 1997. Sketching shadows and highlights to position lights. In *Proceedings of Computer Graphics International 97*, IEEE Computer Society, 56–63.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-time hatching. *Proceedings of SIGGRAPH 2001*, 579–584.

SCHÖDL, A., SZELISKI, R., SALISIN, D., AND ESSA, I. 2000. Video textures. *Proceedings of SIGGRAPH 2000*, 489–498.

SCHOENEMAN, C., DORSEY, J., SMITS, B., ARVO, J., AND GREENBERG, D. 1993. Painting with light. In *Proc. of SIGGRAPH 93*, 143–146.

SEIMS, J. 1999. Putting the artist in the loop. *Computer Graphics 33*, 1 (February), 52–53.

STROTHOTTE, T., PREIM, B., RAAB, A., SCHUMANN, J., AND FORSEY, D. R. 1994. How to render frames and influence people. *Computer Graphics Forum 13*, 3, 455–466.

TOLBA, O., DORSEY, J., AND MCMILLAN, L. 2001. A projective drawing system. In *ACM Symposium on Interactive 3D Graphics*, 25–34.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. *Proc. SIGGRAPH 2000*, 479–488.

WINKENBACH, G., AND SALESIN, D. H. 1994. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, 91–100.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. SKETCH: An interface for sketching 3D scenes. *Proceedings of SIGGRAPH 96*, 163–170.

A Stroke Synthesis

We follow the notation of Schödl et al. [2000]. The first step of the stroke synthesis algorithm is to resample each example stroke into offsets in the normal direction, regularly spaced in four-pixel increments along the length of the stroke (the synthesis algorithm does not yet handle breaks or backtracking along the stroke). We then add a "separator" offset to the end of each example stroke and concatenate the example offsets together into a single vector y, keeping track of the locations of the separator offsets. We calculate the matrix D, where D_{ij} is the "distance" from y_i to y_j:

$$D_{ij} = \begin{cases} K, \text{ when } y_j \text{ is a separator} \\ 0, \text{ when } y_i \text{ is a separator and } y_j \text{ is not} \\ |y_i - y_j| \text{ otherwise} \end{cases}$$

where K is a large constant (we use 10^4). To take the surrounding offsets into account, we filter the distance matrix with a diagonal kernel $[w_0, \ldots, w_m]$: $D'_{ij} = \sum_{k=0}^{m} w_k \cdot D_{i-k,j-k}$, where out-of-range entries in D are assumed to be zero. In our implementation, $m = 4$ and $w_i = 1/m$. D' represents the difference between two windows of offsets, but it does not accurately represent the future cost of transitions. For instance, a window may have a low difference to windows near the end of an example stroke; if the algorithm chose to transition to this window, it would have a high probability of prematurely ending the stroke. We want to assign high costs to such "dead-end" windows. To calculate the future costs, we use Q-learning. We initialize D'' to the values in D'. We then iteratively compute $D''_{ij} \leftarrow (D'_{ij})^p + \alpha \min_k D''_{jk}$ until convergence; we use $p = 2.0$ and $\alpha = 0.999$ in our implementation; Schödl et al. [2000] discusses optimizations. Finally, we calculate the matrix whose entries P_{ij} are the probability of transitioning from offset y_i to offset y_j: $P_{ij} \propto \exp(-D''_{i+1,j}/\rho)$, where ρ is a small multiple (we use 0.01) of the average magnitude of the entries of D'' that are greater than zero and less than the magnitude of the largest example offset.

To synthesize a new list of offsets using y and P, we first choose at random an offset y_i that is the start of an example stroke. We then transition to some other offset j with probability proportional to P_{ij} and iterate until the desired number of offsets are generated.