Art-based Modeling and Rendering for Computer Graphics

by
Lee Markosian

B.A., English, University of Rochester, 1981
Single Subject Teaching Credential, Mathematics, San Francisco State University, 1990
M.Sc., Computer Science, Brown University, 1995

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2000

This dissertation by Lee Markosian is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date 11/12/99 [illegible]
John F. Hughes, Director

Recommended to the Graduate Council

Date 11/12/99 [illegible]
David Laidlaw, Reader

Date 11/12/99 [illegible]
Adam Finkelstein, Reader
Princeton University

Approved by the Graduate Council

Date Jan 26, 2000 [illegible]
Peder J. Estrup
Dean of the Graduate School and Research

Vita

Vitals

Lee Markosian was born on July 23, 1959 in New Brunswick, NJ. In the 80's he worked as a short order cook and counselor to the mentally retarded while writing and performing for no pay with a group of nouveau dadaists in Northampton, MA. In the 90's he got certified and briefly taught as a high school math teacher, before embarking on a new career as a grad student in computer science. He now lives in Providence, RI with his wife Gillis Kallem and their darling daughter Till.

Education

Ph.D. in Computer Science, May 2000.
Brown University, Providence, RI.

M.Sc. in Computer Science, May 1995.
Brown University, Providence, RI.

Single Subject Teaching Credential, Mathematics, 1990.
San Francisco State University, San Francisco, CA.

B.A. in English, May 1981.
University of Rochester, Rochester, NY.

Honors

Ph.D. Fellowship, Intel Foundation, 1998-99.

Graduate Research Fellowship, Brown University, 1993-94.

Stoddard Prize, Mathematics, University of Rochester, 1979.

Refereed Conference Articles

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.* Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment. To be held in June, 2000.

J. D. Northrup and Lee Markosian. *Artistic Silhouettes: A Hybrid Approach.* Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment. To be held in June, 2000.

Lee Markosian, Jonathan M. Cohen, Thomas Crulli and John F. Hughes. *Skin: A Constructive Approach to Modeling Free-form Shapes.* Proceedings of SIGGRAPH 99.

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.* Proceedings of SIGGRAPH 99.

Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes and Ronen Barzel. *An Interface for Sketching 3D Curves.* 1999 ACM Symposium on Interactive 3D Graphics.

Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein and John F. Hughes. *Real-Time Nonphotorealistic Rendering.* Proceedings of SIGGRAPH 97.

D. B. Conner, M. Cutts, R. Fish, H. Fuchs, L. Holden, M. Jacobs, B. Loss, L. Markosian, R. Riesenfeld, and G. Turk. *An Immersive Tool for Wide-Area Collaborative Design.* TeamCAD, the First Graphics, Visualization, and Usability (GVU) Workshop on Collaborative Design. 1997.

Acknowledgements

The ideas and parts of the text presented in this thesis were developed in collaboration with the following people: Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, Jonathan M. Cohen, J. D. Northrup, Thomas Crulli, Robert C. Zeleznik, Loring S. Holden, Ronen Barzel and John F. Hughes. Parts of this thesis have previously appeared as [15, 44, 49, 50].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why art-based graphics?

There's the joke about the computer graphics researcher who goes hiking with his wife in the white mountains of New Hampshire. Standing on a ridge at daybreak they survey the scene spread out before them: Dawn's light playing over towering cloud formations, distant undulating hills thick with trees like some exotic moss, tiny dew drops glistening on overgrown brambles and the tall tangled grass at their feet, hoary lichen clinging to rocky outcrops, stunted pine trees hunched against the wind. His wife takes his arm and murmurs, "Isn't it beautiful?" Depressed by it all, he answers, "But how will we ever be able to render it?"

The standard computer graphics approach to rendering 3D scenes is to represent objects of the scene as a collection of surfaces, each of which reflects light according to a bidirectional reflectance distribution function, or BRDF [30]. These BRDFs may vary across each surface. Given a collection of surfaces and their associated BRDFs, the rendering algorithm must simulate the transport of light through the scene, as it ricochets between each bit of surface according to its angle of incidence and the BRDF there, to arrive finally at the "camera" where it forms an image.

This approach is straightforward in that it simply models the physical process by which light moves through a scene to arrive at a (real) camera's lens, to be captured by the photographic film within. The problem with this approach when applied to a scene as complex as the one described above – and the reason for the graphics researcher's pessimism – is that the amount of data required to faithfully describe the "surfaces" of the scene is vast. And the number of computational steps required to simulate the interaction of light between each of the bits of surface is even more vast.

But suppose the graphics researcher acquired a "black box" that could magically carry out the simulation of light through scenes as complex as the one described, to produce photorealistic results in just seconds. Would that solve the problem? Could he finally enjoy the scene? No, for then he might rightfully ask, "But how will we ever *model* it all?"

It's not clear that any "black box" will ever be able to fully remove this difficulty. The problem is that the scene inherently consists of a vast amount of particular information. Having complete control over the content of that information requires that it be specified. Delegating the job to a black box means giving up control over how that's done. If the goal is to produce a detailed, believable scene from some world that has only ever existed in the imagination of an artist, an automated black box won't be much help.

The point is that realistic depiction of complex objects - clouds, brambles, dewdrops, ants, polar fleece, people - will always involve a trade-off: The more it's automated, the less control the designer has. The more control he has, the more time, attention and care must be taken to model the scene. The source of the difficulty is that realism is inherently costly in terms of the sheer quantity of data that must be specified, bit by bit, then stored, transmitted, and processed to produce an image.

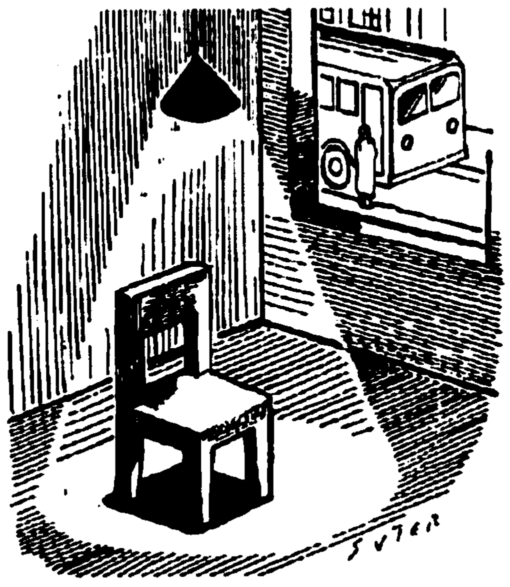


Figure 1.1: Illustration for the New York Times by David Suter.

But all of this assumes that realism is necessarily what's wanted in computer-generated virtual scenes. Yet it's obvious that realism is not always the appropriate choice, depending on how the image is to be used. Consider the illustration by David Suter shown in Figure 1.1 which accompanied a New York Times op-ed piece [75]. Suter's drawing is meant to communicate an abstract idea, and the chosen style serves that purpose. A photograph of a chair lit by an overhead light would be all wrong - too specific, too particular.

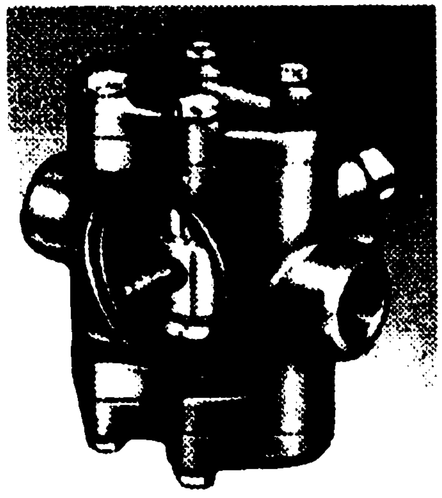


Figure 1.2: Illustration of a piston pump from a supply catalog.

The image in Figure 1.2 is taken from a catalog of mechanical parts [74]. This catalog is full of such images - not photographs, but illustrations that better convey information about shapes and their features. For similar reasons hand-drawn illustrations are widely used in owner's manuals, repair manuals, and medical texts. Lansdown and Schofield [45] make the same point, asking: "How much use is a photograph to mechanics when they already have the real thing in front of them?"

The omission of extraneous details and emphasis of important features - that is, the control of detail in an image for purposes of communication - is a hallmark of handcrafted images. This control of image detail is often combined with stylization to evoke complexity without representing it explicitly. The cover of the Black Hole comic book [14] by Charles Burns is one example (see Figure 1.3). Burns evokes an impression of glossy hair with stylized brush strokes. The skin and clothes have almost no shading information, yet this does not detract from our ability to perceive "skin" and "cloth." In reality, both skin and

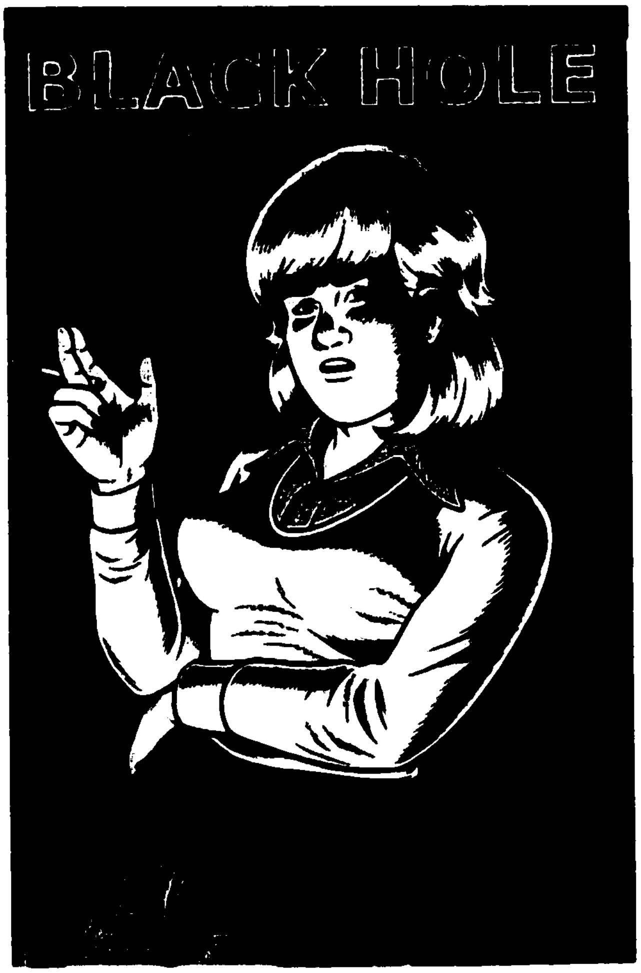


Figure 1.3: Cover of Black Hole #6 comic book by Charles Burns.

cloth have extremely complex surface structure that is partially apparent to the eye. A "realistic" computer rendering that attempts to depict such surfaces with subtle shading runs the risk of getting it subtly wrong, with a result that looks like something very particular - plastic or rubber, perhaps - but not what was intended. Burns' drawing leaves out so much information that this pitfall is avoided. Also note that the background foliage is conveyed clearly with few strokes, and the whole image is clearly delineated though it represents a nighttime scene! Of course, stylization serves other purposes, such as aesthetic ones. In this case Burns' style suits the ominous mood of his story.

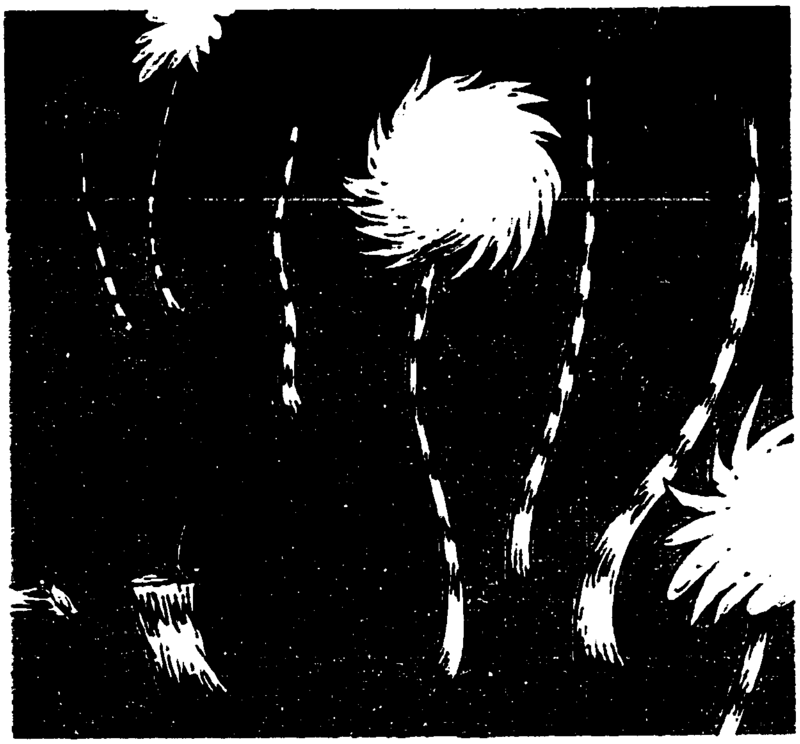


Figure 1.4: A detail of the truffula trees from *The Lorax* by Dr. Seuss [24].

The drawing by Dr. Seuss [24] in Figure 1.4, though very different in style, is another example of the use of simple, stylized strokes to evoke complex geometry indirectly - in this case a landscape with vegetation. Would photographs have served Dr. Seuss' purpose as well? Probably not, for two reasons: Children - the intended audience - apparently respond better to simple, brightly colored images, and less well to photographs, which can be harder to interpret. The other problem is that the subjects of these drawings don't exist except in Dr. Seuss' imagination. To photograph them, it would first be necessary to construct them. Simple drawings like this have the dual advantage of being easy to produce and better serving the required purpose of communication.

1.2 Our thesis

This brings us to the main point - the thesis - of the present work: *We can apply techniques from art and illustration to increase the expressive power of 3D computer graphics.* The task of modeling the landscape around a ridge in the white mountains, for example, need not necessarily be ruinously expensive – a stylized approach that evokes the complexity of the scene indirectly might communicate to the intended audience better than a photorealistic depiction, and be orders of magnitude simpler to produce, store, transmit and render. The same argument applies to depictions of humans or other characters, which have always been favorite subjects of images.

But there's a catch. To truly take advantage of art-based techniques for controlling image detail and evoking complexity, powerful new modeling tools are needed. The designer of a stylized virtual scene will want to specify the geometry of the scene with some appropriate representation and at some appropriate level of detail. For example, to create a 3D version of the figure in Burns' drawing, the hair might be represented as curved sheets whose shape is view-dependent, on top of which stylized strokes are drawn (also view-dependently). The face might be a 3D model, but certainly not one with the detail of a high-resolution model scanned from life. In addition to geometry, the designer will want to create or customize the stylization – including the behavior of the strokes used to achieve it.

Behavior of strokes is an issue because they will typically be view-dependent. Silhouettes are view-dependent, as are shading strokes (more are needed when the camera zooms in) and highlight strokes. In Burns' drawing there is a distinct "halo" around the figure and the tree directly behind her, in which background objects are not drawn. This common technique that provides additional depth cues requires that background objects be view-dependent.

In addition, the modeling tools should let the designer exploit his existing drawings skills to define both geometry and the stylized rendering applied to it. This seems especially desirable considering that so many "art-based" techniques for depicting 3D scenes with controlled detail and stylization involve mark-making, at which artists typically develop considerable skill from working with traditional media.

What's needed, then, is a modeling system that lets a designer exploit his or her skills to create stylized 3D scenes that are rendered according to view-dependent algorithms that he specifies. It turns out this objective entails a significant amount of new research. The various chapters of this dissertation address aspects of the problem of providing such a modeling system. The work is not done, and certainly the ideal system, with the characteristics described above, does not yet exist.

1.3 Description of ensuing chapters

The next chapter describes an early attempt to provide algorithms for rendering finely tessellated smooth surfaces in the style of simple line drawings and at interactive rates. In Chapter 3 we develop a framework for dividing up a model into distinct surface patches, each of which can be rendered differently (with different algorithms) according to what each portion of surface represents – for example, bricks on the walls of a castle, wood planks on the drawbridge. This framework allows rendering styles to be composed, so that more complex effects can be built up from combinations of simple ones. Chapters 4 and 5 build on this framework to develop classes of more sophisticated rendering algorithms (or textures) – one class performs simple hatched shading; the other adopts techniques of Dr. Seuss and others to render fur, grass and trees in a stylized manner.

All of that work is concerned with new rendering algorithms for producing art-based effects – it does not address the separate requirement of the ideal system that it allow a designer to create 3D geometry, preferably by drawing it (with perhaps some additional input). Chapters 6 and 7 in part II of this thesis begin to address this. In Chapter 6, we develop techniques for constructing 3D curves by drawing them – and their shadows – as they would appear from a given viewpoint, using 2D input strokes.

The importance of 3D curves (for modeling) is that they can be used to build up 3D shapes such as generalized cylinders and other "swept" objects. The plan is to develop an interface for rapidly sketching such objects – collections of generalized cylinders and others smooth primitives – in order to define more complex and general free-form surfaces that wrap over the primitives, conforming to their collective shape – like a skin. A new algorithm for computing such a skin – with multi-resolution properties that allow creases and bumps to be added as details to a roughly defined smooth shape – is described in Chapter 7. The approach to modeling we have chosen to pursue – building up a complex shape from collections of smooth primitives or masses, then oversketching a smooth shape and refining it with the addition of surface detail – is directly analogous to methods typically taught by art instructors and used by artists. The approach is "natural" in that artists with acquired skill can exploit that skill to sketch compelling free-form shapes like human figures.

The natural next step is to integrate the various parts into a single system for sketching both 3D shapes and the stylized rendering algorithms used to depict them.

Part I

Rendering

Chapter 2

Silhouettes

2.1 Introduction

A growing body of research in computer graphics has recognized the power and usefulness of nonphotorealistic imagery [17, 32, 45, 46, 52, 65, 73, 83, 84, 87]. Until now[1], though, nonphotorealistic rendering (NPR) methods have primarily been batch-oriented rather than interactive. (An exception is Zeleznik's SKETCH system [87], which makes crude nonphotorealistic renderings using tricks in the standard polygon-rendering pipeline). One obstacle to achieving real-time nonphotorealistic rendering is the problem of determining visibility, since a straightforward use of z-buffering may give incorrect results. This can occur when what is drawn on the screen does not correspond literally to the geometry of the scene. For example, a line segment between two vertices of a triangle mesh may be rendered in a wobbly, hand-drawn style. Any part of the wobbly line which does not directly correspond to the original line segment may be clipped during z-buffering.

This chapter presents a new real-time NPR technique based on an *economy of line* – the idea that a great deal of information can be effectively conveyed by very few strokes. Certain key features of images can convey a great deal of information; our algorithm preferentially renders silhouettes, certain user-chosen key features (e.g., creases), and some minimal shading of surface regions. To accomplish this at interactive rates, we rely on approximate data: not every silhouette is rendered in every frame, although large silhouettes are rendered with high probability. The key steps that support this scheme are:

[1]The material in this chapter was originally published in 1997 [50].

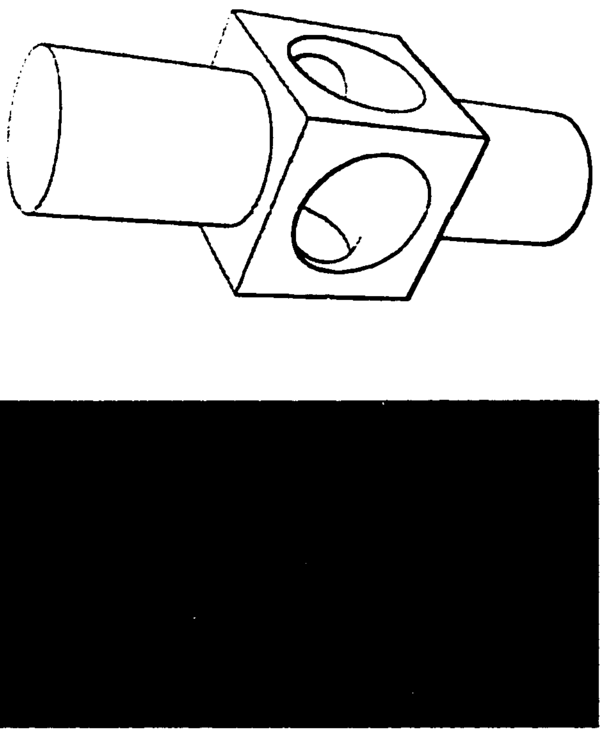


Figure 2.1: A mechanical part rendered with plain strokes (top) and wobbly strokes (bottom). Model courtesy of the University of Washington.

- rapid (probabilistic) detection of silhouette edges,
- fast visibility determination using a modified Appel's algorithm,
- drawing the resulting visible lines with various stylizations, and
- optionally drawing additional shading strokes over interior surfaces.

We demonstrate the use of these techniques to support a variety of rendering styles, all of which are produced at interactive rates. These include a spare line-rendering style suitable for illustrations and a variety of sketchy hand-drawn styles suitable for approximate models (see Figure 2.1), optional rendering of hidden lines (Figure 2.2), and a style that adds shading strokes to basic visible-line renderings in order to better convey 3D information while preserving a hand-drawn effect (see Figure 2.6). Using the methods we describe, our software renderer is able to produce basic visible line drawings of free-form (tessellated) surfaces at an effective rate of over 1 million model polygons per second on a modern workstation. (In 1997 when we ran our tests this was competetive with current graphics hardware.)

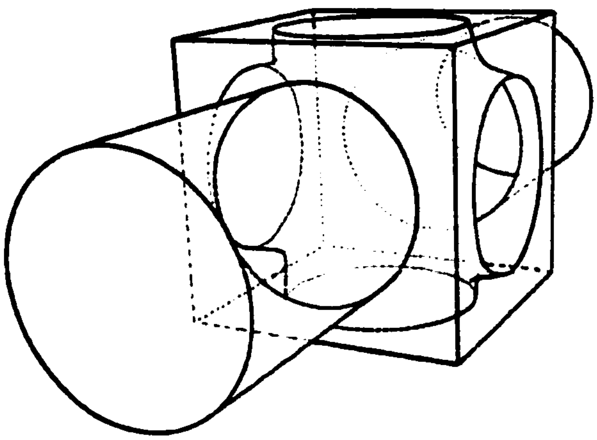


Figure 2.2: A mechanical part showing occluded lines in varied styles.

2.2 Assumptions and Definitions

First, we assume the model to be rendered is represented by a non-self-intersecting polygon mesh, no edge of which has more than two adjacent faces - i.e., the mesh is a topological manifold. To make our second assumption precise, we need some definitions:

Definition 1 *A polygon is* **front-facing** *if the dot product of its outward normal and a vector from a point on the polygon to the camera position is positive. Otherwise the polygon is* **back-facing**. *A* **silhouette edge** *is an edge adjacent to one front-facing and one back-facing polygon. A* **border edge** *is an edge adjacent to just one face. A silhouette edge is* **front-facing** *if its adjacent face nearest the camera is front-facing. Other silhouette edges are* **back-facing**.

Our second assumption is that in every image that we render, the view is *generic* in the following sense:

Definition 2 *A view is* **generic** *if (i) the multiplicity of the image of the silhouette curves is everywhere one, except at a finite number of points where it is two; (ii) these multiplicity-two points do not coincide with the projection of any vertices of the mesh; and (iii) their number is invariant under small changes in viewing direction.*[2]

Our method for determining visibility (described in section 2.4.1) may fail for non-generic views, but we have not observed this in practice when computations are performed with double-precision (64 bit) floating point numbers.

[2]This definition is adopted from [82].

2.3 A fast randomized algorithm for finding silhouettes

In this section we address the problem of finding the silhouette edges of a polyhedral model and given camera parameters. A simple algorithm that does this is as follows:

Algorithm 1 (Exhaustive silhouette detection)
Check each edge of the model and record those that are silhouette edges.

This simple, effective algorithm has just one drawback: it requires that we check every edge of the model every frame. If the model is finely tessellated to approximate a smooth surface, then typically just a small fraction of all edges will be silhouettes. It's interesting to note that the silhouette edges will tend to be organized into long, connected groups, or *chains*. Indeed, it's easy to prove that a silhouette edge can never exist in isolation – one will always find neighboring silhouette edges adjacent to each endpoint. One way to state this is to say that silhouette edges exhibit spatial coherence. A second type of coherence we can expect to observe is temporal coherence. That is, under a succession of small changes in viewing parameters, as we would have in an animation or during interactive viewing of the model, it will nearly always be the case that a given chain of silhouette edges in one frame will contain edges that were also silhouette edges in the previous frame. These two observations led us to develop the following randomized algorithm for detecting silhouettes that avoids checking all but a small fraction of the model's edges:

Algorithm 2 (Randomized silhouette detection)

(1) Check just a small fraction of the model's edges to look for silhouette edges. When a silhouette edge is found, check each edge adjacent to its endpoints to look for additional silhouette edges, repeating the process until the entire chain of silhouette edges is found.

(2) To increase the chances of detecting each silhouette chain, we also check each edge in the current frame that was found to be a silhouette edge in the previous frame.

We'd like to analyze the running time of the randomized algorithm and determine to what extent it improves on that of the exhaustive silhouette detection algorithm, which has running time of $O(n)$, where n is the number of edges in the model. At the same time we should be able to say something about how effective the randomized algorithm is at finding silhouettes. To make this meaningful we introduce some assumptions. First, we assume the model is a tessellated approximation of some "ideal" surface, and that the tessellation is carried out according to a refinement scheme that lets us better approximate the ideal surface by using more triangles. Further, we'll assume that a given silhouette chain can be identified and tracked over each refinement, and that the number of edges in a chain doubles with each refinement, whereas the total number of edges of the model quadruples.[3]

Proposition 1 *Under these assumptions, we can apply step (1) of the randomized algorithm to check just $O(\sqrt{n})$ edges of the model while maintaining a chosen probability of detecting a given silhouette chain.*

Proof Let n be the number of edges in a given refinement of the model, c be the number of edges in the silhouette chain at that level of refinement, and k be the number of edges of the model that we will check to look for silhouettes. Each of these variables increases with each refinement. Because c doubles while n quadruples, it's easy to show that there is a constant β for which $c = \beta\sqrt{n}$. (Specifically, take $\beta = c_0/\sqrt{n_0}$, where c_0 and n_0 are the number of edges in the chain and the model, respectively, at the coarsest level of refinement.)

Now let's take $k = \alpha\sqrt{n}$, for some choice of α. We adopt the policy that each time we check an edge, we leave it in the list of edges, so that we run a slight chance of checking a given edge more than once. We will detect all the edges of the chain if we detect just one of them, so to miss the chain we would have to miss finding an edge of the chain each time we check an edge. Thus, the probability that we miss the entire silhouette chain is:

$$\begin{aligned} P(\text{miss}) &= \left(\frac{n-c}{n}\right)^k \\ &= \left(1 - \frac{\beta\sqrt{n}}{n}\right)^{\alpha\sqrt{n}} \\ &= \left(1 - \frac{\beta}{\sqrt{n}}\right)^{\alpha\sqrt{n}} \\ &< e^{-\alpha\beta}. \end{aligned}$$

So the probability of hitting the chain is:

$$P(\text{hit}) > 1 - e^{-\alpha\beta}.$$

In other words, as n increases with each refinement, we can check $\alpha\sqrt{n}$ edges and our probability of detecting the given chain will always remain above the fixed value $1 - e^{-\alpha\beta}$. This completes the proof. □

[3]The refinement scheme that splits each edge at its midpoint, dividing each triangle into four congruent sub-triangles, has these properties. It seems reasonable to expect refinement schemes that incorporate smoothing to have roughly these properties, especially as the tessellation approaches the ideal surface.

The argument just given actually tells us how to adjust our choice of α to attain a desired probability p of detecting a particular chain: calculating β as above, we just take $\alpha = -\ln(1-p)/\beta$. For example, if at the coarsest level of refinement our mesh has 128 edges and we would like to detect a chain of 8 edges with probability $p = 0.95$, then $\beta = 1/\sqrt{2}$ and so we must take $\alpha \approx 4.24$.

Of course, detecting a given silhouette chain 95% of the time means we miss it roughly once in every 20 frames – which would be intolerably distracting. We observe in practice that step (2) of the randomized algorithm is very effective at ensuring that each silhouette chain continues to be detected in every frame after it is first detected (as long as it persists and under incremental changes to the camera parameters). We occasionally notice that a silhouette is detected a few frames later than it could have been, but once detected the algorithm continues to find the chain reliably.

As we report in section 2.6, the randomized silhouette detection algorithm improves the *total* running time of our rendering algorithm by a factor of up to five for our test models.

2.4 Determining visibility

2.4.1 Appel's algorithm

Appel's hidden-line algorithm [3], as well as those of Galimberti [23] and Loutrel [48], is based on a notion of *quantitative invisibility* (QI), which counts the number of front-facing polygons between a point of an object and the camera. The algorithm is applied to the entire mesh of edges in a polyhedral model to determine QI at all points; those with QI = 0 are visible and are drawn. Good descriptions of the basic algorithm can be found in [8, 21, 76]. We summarize a few key ideas here.

The algorithm first identifies all silhouette edges, because as we traverse the interior of an edge, QI changes only when the edge crosses behind a silhouette. In a generic view, QI can also change at a vertex, but only when the vertex lies on a silhouette edge. This fact is characterized by several authors [21, 76] as a "complication" of the algorithm; we'll discuss this further below.

The algorithm proceeds by determining (via raytracing, for example), the QI of some point in each connected set of edges, and then propagating QI out from this point, taking care to note changes as the edge along which it is propagated passes behind silhouettes, or when a vertex through which it is propagated lies on a silhouette. In this way the number of ray tests is minimized by exploiting "edge coherence."

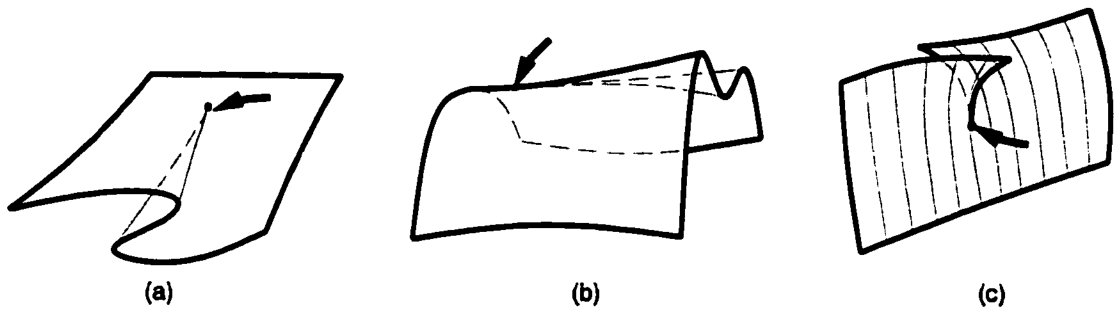


Figure 2.3: Arrows indicate cusps. (a) A typical cusp. (b) A more exotic cusp. (c) A border cusp (the two edges meeting at the center of the sheet are border edges).

2.4.2 Silhouettes and cusps

The "complication" in Appel's algorithm arises because the mapping from the surface to the plane is singular along silhouette edges. Understanding this complicated case better allows us to avoid some unneeded computation. To this end, we first (following [18]) redefine QI to be the number of layers of surface (front- and back-facing) obscuring a point. We then observe that, for generic views, QI along a silhouette curve can change at a vertex only if that vertex is of a special type, which we call a *cusp*:

Definition 3 *A vertex is called a* **cusp vertex** *(or* **cusp***) if one of the following holds (see Figure 2.3):*

1. *it is adjacent to exactly 2 silhouette edges, one front-facing and the other back-facing,*
2. *it is adjacent to more than 2 silhouette edges, or*
3. *it is adjacent to a border edge*[4]*.*

The QI along a non-silhouette curve which intersects a silhouette curve at a vertex can change as it passes through the vertex. Appel's algorithm thus requires a local test at every vertex belonging to a silhouette. But we are interested primarily in propagating QI *along* silhouette curves, so testing for changes in QI just at cusp vertices provides a significant savings in computation time.[5]

[4]This case is necessary, as shown by Figure 2.3(c), which contradicts corollary 5.1.5 of [18].

[5]Note that front-facing and back-facing silhouette edges (used in identifying cusps) can be detected according to whether the surface along the edge is convex or concave; the convexity of each edge can be determined in a pre-process step once-and-for-all.

2.4.3 Avoiding ray tests

Next, we show how to avoid some of the ray tests required by Appel's algorithm. First, if we assume that all objects in the scene are completely in view of the camera, then any edge which touches the 2D bounding box (in image space) of all silhouettes does not require a ray test – it is automatically visible. Hence, no ray test is required for any connected set (or **chain**) of silhouette edges containing such an edge.

Appel's algorithm would now proceed with (1) a ray test to establish QI at some distinguished point on each chain, followed by (2) the propagation step in which QI is assigned to the remaining points of each chain. By reversing this order, we can sometimes eliminate the need for a ray test altogether, since the second step is often sufficient to determine that an entire silhouette curve is occluded. (See Figure 2.4).

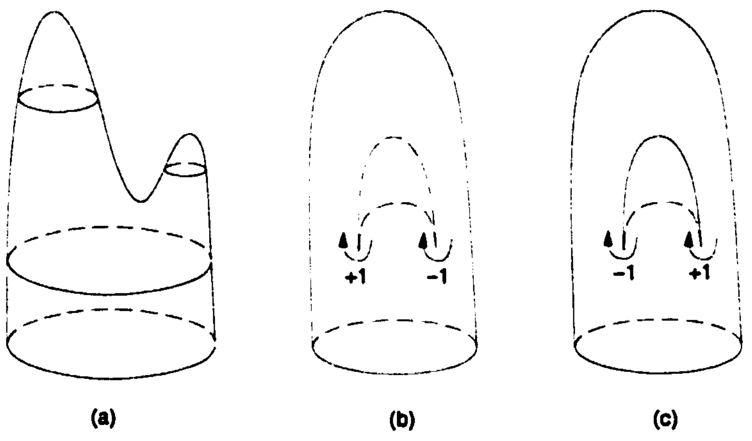


Figure 2.4: (a) A surface: side view. (b) Smaller branch is in rear. (c) Smaller branch is in front. The change in QI at cusps is indicated. Traversing the inner silhouette in (b) is sufficient to determine that the silhouette is totally occluded.

For each connected silhouette chain, we first choose an edge and a point on it infinitesimally close to one of its vertices. We call this point the **base point** of the chain. Let b denote QI at the base point. QI at all other points of the chain will be defined via offsets from b. We assign a preliminary lower-bound value of 0 to b. We then calculate the offsets with a graph search, taking into account image space intersections of edges of the current chain with any silhouette edges, as well as cusp vertices encountered in the traversal. (A curve's QI increases by two when it passes behind a silhouette, and may change by an arbitrary, locally measurable amount at a cusp vertex). We record the minimum QI, m, encountered during the search. If $m < 0$, we may safely increment b by $-m$. It's easy to show that on a closed surface, front-facing silhouette edges must have even QI and back-facing silhouette edges must have odd QI. For such surfaces, we add 1 to b if needed to correct its parity. (In that case the chain is totally occluded - Figure 2.4 shows an example of this).

Finally, we examine each intersection involving edges from different chains. In this situation, if n is the QI of the occluding edge, and m is the QI of the occluded edge along its unobscured portion, then we must have $m \geq n$. If we find that for our estimated QI values $m < n$, we can increment the base QI of the chain containing the occluded edge by $n - m$, and propagate this information to other chains as well.

In practice, these observations often account for all chains, and consequently no ray tests are required in the current frame. In the remaining cases we perform the needed ray tests efficiently through a technique we call **walking**.

2.4.4 Walking

Once relative QI values at all points of a silhouette chain have been determined with respect to the QI b at the base point, we must determine the correct value of b. The following technique does this, assuming all objects in the scene are in front of the camera. (We briefly discuss how to render immersive scenes below).

When one silhouette chain is totally enclosed by another (in image space), the enclosing silhouette may be the boundary of a region which may totally obscure the enclosed silhouette. (See Figure 2.4 (b) and (c). In (b), the enclosed silhouette is totally occluded, in (c) it is not). We detect such enclosures and their consequent occlusions as follows. First, we disregard silhouette curves which are already known to be totally occluded. We also disregard any silhouette curve which touches the image space box, B, that bounds all silhouette edges (as it can't be totally enclosed). On each remaining silhouette curve, we choose a point U with currently assigned QI of 0. Let U_p denote the projection of U. We identify enclosing silhouettes by tracing a path in image space from U_p toward the boundary of B. From each enclosing silhouette curve S encountered at an image space point V_p, we find the corresponding point V on S. We choose a branch of surface adjacent to S at V along which we can begin tracing a path whose projection heads back toward U_p, if such a branch of surface exists. (Either both branches of surface satisfy this condition or both do not - in which case we proceed to the next enclosing silhouette). We then traverse the surface from V along the path whose projection retraces (in reverse direction) the original path from

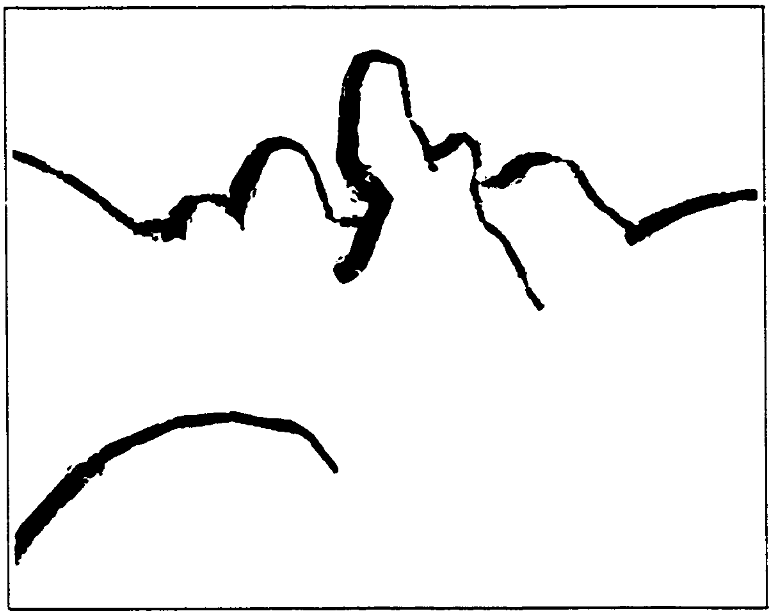


Figure 2.5: A charcoal-like rendering of terrain with texture-mapped strokes.

U_p. If this surface walk succeeds in arriving at a point which projects to U_p, a depth test determines whether U is occluded by that portion of surface.

Our walking method does not work in general for immersive scenes in which geometry may surround the camera. An alternative approach is to perform ray tests efficiently with the use of an octree data structure which can be used to find intersections of a line segment with any triangles in the scene. One problem with this approach is that if there are any silhouette curves in the scene which have gone undetected by the randomized algorithm for finding silhouettes, it's possible for a small region of occlusion in a detected silhouette to be propagated (incorrectly) throughout the entire silhouette. This can occur since intersections with the undetected silhouettes are not taken into account, but the ray test may count occlusions due to surfaces bounded by the undetected silhouettes. (The walking method does not count such surfaces). Taking steps to decrease the probability of missing silhouette curves that lie within the viewing frustum is one approach for minimizing this problem.

The discussion to this point has tacitly assumed that edges of interest are all silhouette edges. These methods easily accommodate border edges and other non-silhouette edges (such as creases or decorative edges) as well. Border edges cause a change of ± 1 in QI of edges passing behind them. Other edges cause no change in QI of edges passing behind them.

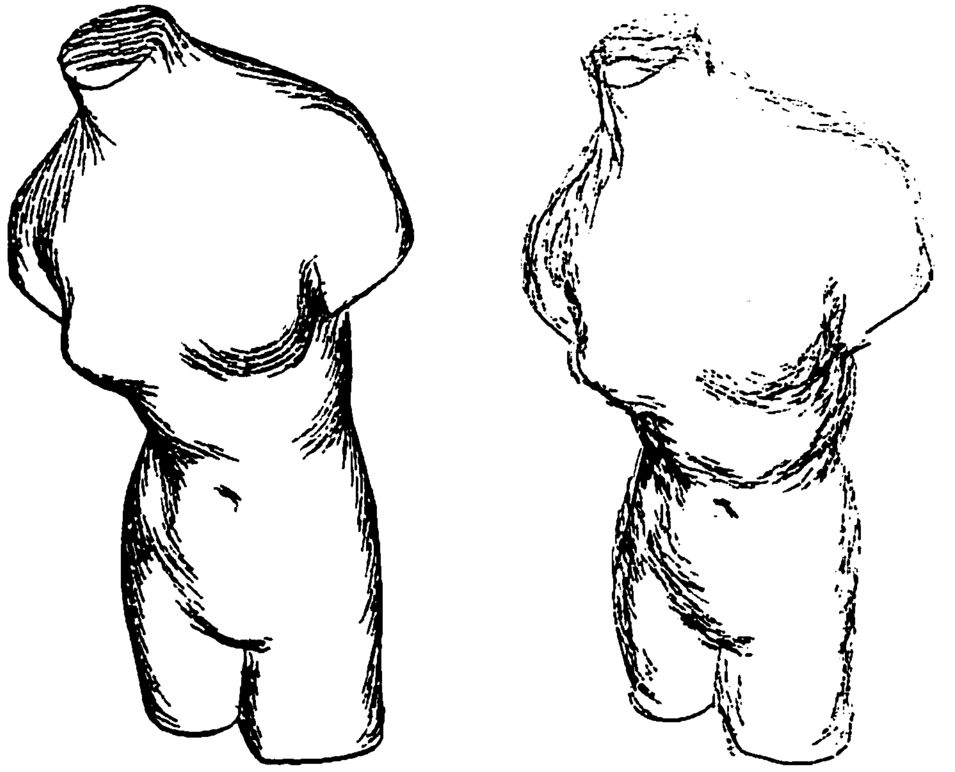


Figure 2.6: The venus model with simple shading strokes (left), and wobbly ones (right).

2.4.5 Implementation details

We follow Loutrel's [48] approach of projecting the silhouette edges into image space and finding their intersections there. This can be done with a sweep-line algorithm in $O(k \log k)$ time, where k is the number of silhouette edges (see e.g. [57]). We found it more convenient to use a spatial subdivision data structure which divides the image space bounding box of the silhouette edges into a grid of cells. Each silhouette edge is "scan converted" into the grid; only edges which share a cell need be tested for intersection. This method has worst-case complexity of $O(k^2)$ but performs well on average. We re-use the spatial subdivision grid in the walking step, in order to find enclosing silhouettes whose projection intersects the image-space path from U_p.

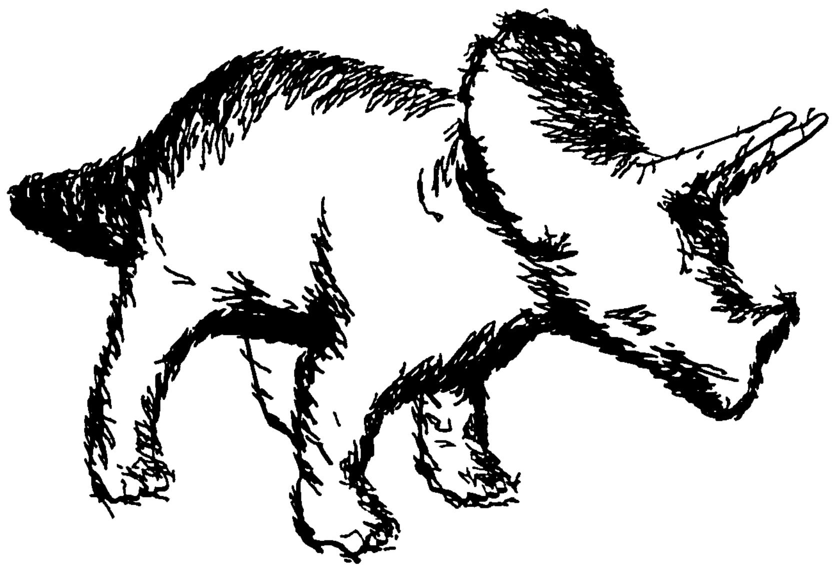


Figure 2.7: Triceratops with shading strokes.

2.5 Rendering visible lines and surfaces

We demonstrate the use of our visibility algorithm to produce several styles of nonphotorealistic renderings at interactive rates.

World-space polylines to be rendered are first projected into the film plane. Artistic or expressive strokes are then generated by modifying the resulting 2D polylines. We use three techniques for generating expressive strokes: drawing the polylines directly, with slight enhancements such as variations in line width or color (see Figure 2.1 top); high-resolution "artistically" perturbed strokes defined by adding offsets to the polyline (Figure 2.1 bottom); and texture-mapped strokes which follow the shape of the polyline (Figure 2.5). A variation on the first method is to render occluded lines in a style which depends on the number of layers of surface occluding them (Figure 2.2).

In the second method we first parameterize the polyline by arc length. We then define a new parametric curve $\mathbf{q}(t)$ based on the original parametric curve $\mathbf{p}(t)$ by adding a vector offset $\mathbf{v}(t)$ defined in the tangent-normal basis, i.e.:

$$\mathbf{q}(t) = \mathbf{p}(t) + v_x(t)\mathbf{p}'(t) + v_y(t)\mathbf{n}(t).$$

The use of vector offsets allows $\mathbf{q}(t)$ to double back on itself or form loops. Using the tangent-normal basis allows perturbation patterns to follow silhouette curvature. These offset vectors can either be precomputed and stored in lookup tables or computed on the fly. (We have implemented both techniques).

For precomputed offsets, we use a file format which specifies vector offsets. This format also incorporates "break" tags which signal the renderer to leave selected adjacent vertices unconnected in $\mathbf{q}(t)$, allowing strokes to incorporate disconnected shapes, such as circles, dashes, or letters. Variations on a small number of fundamental stroke classes (sawtooth, parabolic undulations, noise) produce a wide variety of stroke styles: high frequency sawtooth curves produce a charcoal style; low-frequency parabolic curves produce a wandering, lazy style; high-frequency, low-magnitude noise applied along the stroke normal and tangent directions produces a jittery hand-drawn style; low-frequency, high-magnitude offsets along the stroke tangent produces a jerky, rough-sketched look.

An alternative method for computing offsets is to use a spatially-coherent noise function indexed by screen-space location. We use a Perlin noise function [56] to define displacements along visible lines.[6]

The third method builds a texture-mapped mesh using the polyline as a reference spine. Each texture map represents a single brush stroke. We repeat the texture along the reference spine, approximately preserving its original aspect ratio. In order to generate the mesh, we walk along the spine adding a perpendicular crossbar at each vertex in the polyline and at each seam between repeating brush strokes. Additionally, the width of the stroke can be made to vary with lighting computed at the polyline vertices, becoming thicker in darker areas and thinner in lighter areas. Our simple implementation does not handle self-intersections of the texture map mesh due to areas of high curvature.

Lastly we demonstrate a technique for generating curved shading strokes in order to produce a richer artistic effect and to better convey 3D information. (See Figures 2.6 - 2.7). Here, the principle of "economy of line" supports both the esthetic goals and that of maintaining interactive frame rates. We use an extension of the hidden line algorithm which allows us to derive visibility information across surface regions. This method was described by Hornung [38].

We place shading strokes (or *particles*) in world space (on the surface) rather than define them in screen space. This is the approach used by Meier [52] in her "painterly rendering" system. One advantage of this approach is that it maintains frame-to-frame coherence. We make the simplifying assumption that lighting comes from a point source located at

[6]We thank Paul Haeberli for this rendering method and the source code for implementing it.

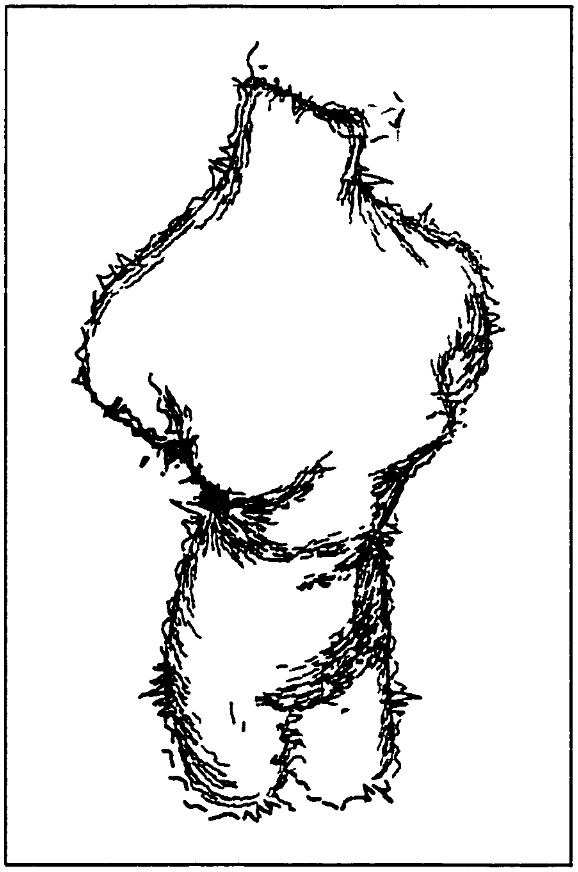


Figure 2.8: Similar to Figure 2.6, but the silhouettes are more loose and sketchy.

the camera position. This greatly simplifies the task of computing stroke placement and density to achieve a target tone. An even distribution of strokes on the surface produces higher apparent densities in regions slanting away from the light – which is exactly where we want a darker tone. Our initial implementation assigns one stroke particle to the center of each triangle, which assumes a sufficiently even triangulation. Strokes are not drawn when occluded or when the computed gray value (using a lambertian shading model) falls below a threshold.

Stroke directions are defined by the cross product of local surface normal and the ray from the camera to the stroke location, so that strokes line up with silhouette lines. Strokes have a preset world-space length; those with sufficiently large screen-space length are drawn as polylines. The direction of each segment of the polyline is computed as above, with local surface normal taken as a blend of normals at the vertices of the triangle at which the stroke is centered. Finally, we render the strokes using any of the artistic rendering methods described above.

2.6 Performance

We treat our models as subdivision surfaces, which allows us to refine a given mesh so that it approximates a smooth surface with an arbitrary degree of accuracy. (See [36] for a description of the type of subdivision surfaces we use). The following tables list performance statistics for our renderer operating on models which have been subdivided to the indicated number of polygons. Our test machine is a 200 MHz Sun Ultra ^{TM} 2 Model 2200 with Creator 3D graphics.[7] Our method performs particularly well on smooth meshes, since these have fewer cusps and intersecting silhouette edges than irregular or bumpy surfaces.

In contrast, Winkenbach's [83] pen-and-ink rendering system produces decidedly finer images, but takes several minutes per frame to do so. (Over half that time is spent on visibility determination).

Model	Triangles	Frames/sec	Triangles/sec
Two tori	65,536	30.58	2,004,091
Mechanical part	64,512	14.69	947,681
Venus	90,752	17.83	1,618,108

Table 2.1: Performance of the basic visible-line renderer. Times were measured on a 200 MHz Ultrasparc.

Model	Triangles	Frames/sec	Slowdown
Two tori	65,536	5.27	5.8
Mechanical part	64,512	4.30	3.4
Venus	90,752	3.47	5.1

Table 2.2: Performance of basic line renderer when checking all edges each frame – the slowdown is in comparison with the same models listed in table 2.1.

Model	Triangles	Frames/sec	Triangles/sec
Blobby teddy bear	7,776	6.37	49,588
Venus	5,672	9.2	52,195

Table 2.3: Performance of the shaded line renderer.

[7]We use the graphics capabilities for rendering lines only.

Chapter 3

Procedural textures

3.1 The need for view-dependent "textures"

In the last chapter, we discussed fast algorithms for rapidly detecting silhouette curves and determining their visibility. We applied these algorithms to build a simple silhouette-based nonphotorealistic renderer. This renderer suffers from several limitations – particularly its lack of flexibility. The renderer applies a single "style" to the whole scene, where the style just consists of parameters affecting the appearance of the strokes used to render the scene. The strokes themselves are placed according to a fixed procedure that does not take into account what the various parts of the model are intended to represent.

Say we want to create a stylized 3D scene that looks like the print by M. C. Escher shown in Figure 3.1. Specific, distinct patterns of strokes must be assigned to the various surface regions corresponding to stone columns, insect wings, cloth and so on. In a sense we want each pattern of strokes to be like a texture map, except that the stroke patterns must be view-dependent. For example, fewer shading strokes are required to draw a surface when it is far away. Also, the conspicuous haloing around the mantis is a view-dependent phenomenon. We explained more in Chapter 1 about why an art-based renderer must adaptively control the amount of visual detail used to depict each object in a scene. Recognizing the need to associate view-dependent stroke patterns with surfaces in a given scene led us to develop a new framework for representing procedural textures.

Another drawback of the rendering system described in the last chapter is that our modified Appel's algorithm for determining visibility is complicated to implement and does not work when objects in the scene surround the camera. The procedural texture framework

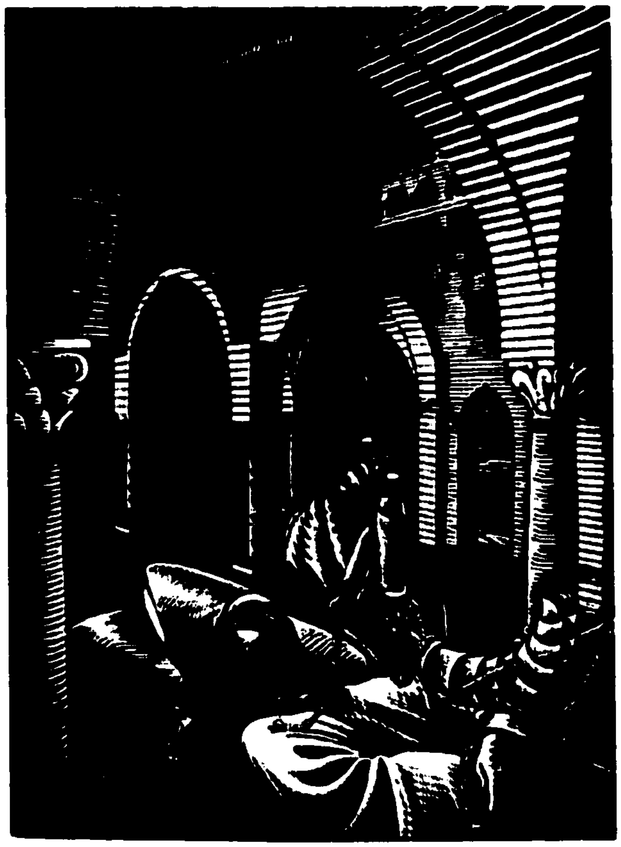


Figure 3.1: Wood engraving by M. C. Escher.

described in this chapter includes an alternative method for determining visibility that avoids these problems. The trade-off is that it makes greater demands on the graphics hardware. The randomized algorithm for detecting silhouettes described in the last chapter continues to be useful within our new system.

3.2 A new framework

Our procedural texture framework is implemented within *jot*, a collection of C++ libraries developed at Brown to support interactive 3D applications, both immersive and desktop, on multiple platforms.

The primary representation for 3D objects is a *mesh*. This is a triangulated polyhedron that stores lists of vertices, edges and faces (triangles), with the usual connectivity information. Each mesh is divided into one or more *patches*, which are collections of faces. Each patch is assigned a procedural *texture* that is responsible for drawing the part of the model corresponding to the patch. The notion of dividing a model into distinct patches and rendering each with its own texture is standard. An important distinction of our system is that the textures are procedural; the stroke-based textures described in Chapters 4 and 5 were implemented within this framework.

The "procedure" that defines a texture needn't be complicated – many simply draw their patch in some conventional style (e.g., smooth-shaded or wireframe). One of our textures performs Floyd-Steinberg dithering [21]. Others perform a variety of hatching effects as described in Chapter 4.

An important component of the system is the provision of *reference images*. These are off-screen renderings of the scene, subsequently read from frame-buffer memory to main memory and made available to the procedural textures. (Meier's "painterly rendering" system [52] and Saito and Takahashi's "comprehensible rendering" techniques [65] used reference images as well.) We currently use two kinds of reference images: a *color reference image* and an *ID reference image*. The color reference image can be used in various ways – the rendering algorithms described in the following two chapters use the color reference image as a tone map, generating more strokes (or fatter strokes) in dark regions and fewer (or thinner) strokes in regions of light tones. The ID reference image supports $O(1)$ picking, as we explain below. The idea is that each face is given a unique ID that lets us look up the face in a table.[1] To prepare the ID reference image, we draw the scene, coloring each face with a color that corresponds to its ID. Lighting and blending are disabled so that the colors are preserved exactly. Then, given a pixel in the ID reference image, we can read the color at that pixel and look up the designated face.

After the ID reference image is prepared, all of its pixels are checked in one pass: when a pixel contains the valid ID of a face, that pixel location is stored in a list on the patch that contains the face. Later, the texture of the patch can access the list of pixel locations in its rendering procedure. For example, the dithering texture simply runs the Floyd-Steinberg algorithm on the pixels of its patch.

[1]Edges and vertices can be assigned IDs in the same way, and looked up using the ID reference image.

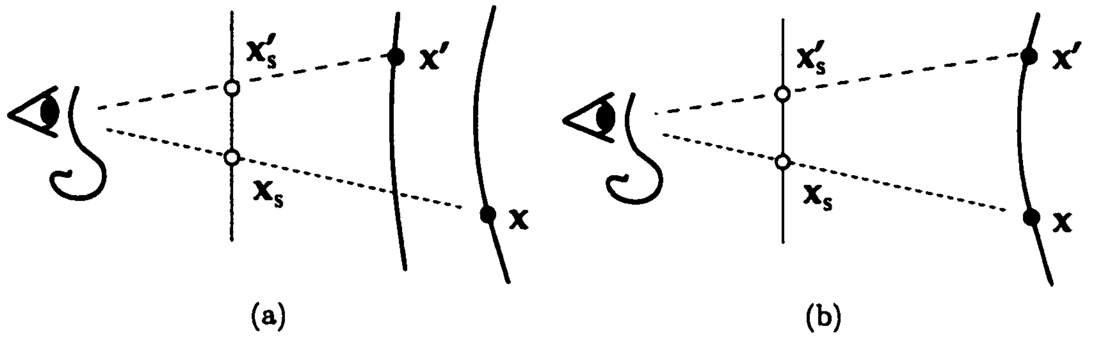


Figure 3.2: A point $\mathbf{x}$ on a surface (cutaway view) has screen-space location $\mathbf{x}_s$. At that location in the ID reference image we find the ID not of triangle T containing $\mathbf{x}$, but of another triangle T'. (The triangles are not shown.) We denote by $\mathbf{x}'$ that point on T' whose screen-space location $\mathbf{x}'_s$ is closest to $\mathbf{x}_s$. In this situation, $\mathbf{x}$ may or may not be occluded. In (a) it is occluded; in (b) it is not.

3.3 Determining visibility using the ID reference image

The ID reference image can be used to answer the following question: Given the current camera parameters and any point $\mathbf{x}$ on a triangle T of a model, is $\mathbf{x}$ visible? This is the question addressed by our modified Appel's algorithm described in the last chapter. The method we now describe for determining visibility is a simpler and more robust alternative to that algorithm. (E.g., the modified Appel's algorithm does not work if any part of the scene is behind the camera.) Knowledge of what is visible in a given scene is crucial for many art-based effects. As explained in Chapter 2, the usual technique for handling occlusion in computer graphics – z-buffering – cannot be used when strokes are drawn with screen-space stylizations, since then the shape of a stroke may deviate from the shape of the underlying geometry, causing parts of the stroke to be clipped incorrectly. If we know what portions of the model are visible, we can disable z-buffering and render just the visible strokes with whatever stylizations we like. Knowledge of visibility can also be used to implement effects like haloing, as was done by Dooley and Cohen [17].

The first step in answering the question stated above is to determine whether $\mathbf{x}$ lies in the camera's viewing frustum.[2] If it lies outside the frustum, $\mathbf{x}$ is not visible. If it lies inside the frustum, one might then proceed by projecting $\mathbf{x}$ to its screen space position $\mathbf{x}_s$, then looking up the triangle found there in the ID reference image. We conclude $\mathbf{x}$ is visible if and only if the triangle we find is T.

This method has the advantage of being simple and intuitive. Its main disadvantage

[2]An explanation of cameras and frusta is given in [21].

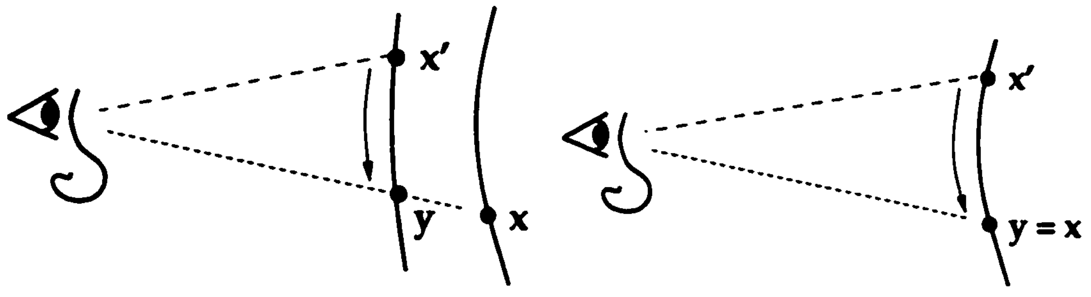


Figure 3.3: The local search procedure yields a point **y** whose screen-space location is the same as that of **x**. In (a) we find that $\mathbf{y} \neq \mathbf{x}$, and conclude that **x** is occluded (by **y**). In (b) $\mathbf{y} = \mathbf{x}$, hence **x** is visible.

is that it doesn't work. For example, the ID of T may be found at screen location $\mathbf{x}_s$ even though **x** is not visible. This could happen if **x** is just behind the silhouette of some occluding object, because the pattern of pixel rasterization cannot in general match the exact shape of the triangles being rasterized. But if **x** is well away from any silhouette (by more than a pixel), then it *is* true that finding the ID of triangle T at screen location $\mathbf{x}_s$ implies **x** is visible. The converse is not true: Finding the ID of some other triangle at $\mathbf{x}_s$ does *not* imply that **x** is occluded. For example, the triangle T may be very small in screen space and may simply not have contributed to the rasterization of the ID reference image. Or **x** may be visible but lie near the edge of triangle T, and the ID of a neighboring triangle may have been written to location $\mathbf{x}_s$. In such cases, we would still like to be able to answer the question: Is **x** visible?

To proceed, we first assume that there are no z-buffer artifacts, such as part of one surface showing through another surface that actually covers the first. We also assume surfaces do not intersect themselves or each other, and that $\mathbf{x}_s$ is more than a pixel away from any (visible) silhouette. With these assumptions, we can answer the question. The idea is this: since **x** is not near a silhouette, there are two possibilities: either there is some front-facing portion of surface totally covering it, or there is none. (See Figure 3.2.)

In the latter case, the triangle T' whose ID is found in the ID reference image at screen location $\mathbf{x}_s$ must lie on the same portion of surface as **x**. But if **x** is occluded, then T' must be part of the occluding surface. To determine which case holds, we first find the point $\mathbf{x}'$ on T' whose screen location $\mathbf{x}'_s$ is closest to $\mathbf{x}_s$. If $\mathbf{x}'_s = \mathbf{x}_s$ but $T' \neq T$, then $\mathbf{x}'$ itself occludes **x**, and the question is answered. Otherwise, we update $\mathbf{x}'$ using a local search procedure that is essentially the same as the one described in Chapter 7 (see Figure 7.3),

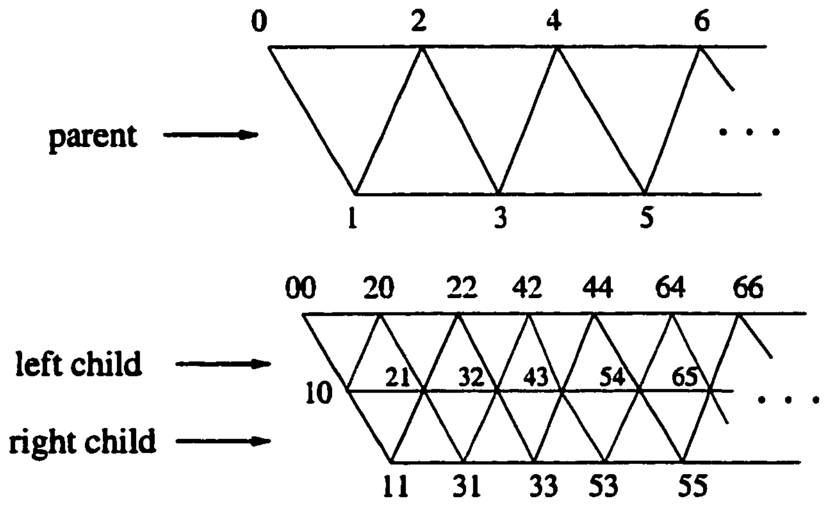


Figure 3.4: A triangle strip in the control mesh (top) defines two triangle strips in the next-level subdivision mesh (bottom). Vertices in the next-level subdivision mesh are denoted by jj if generated by vertex j in the control mesh, and by ij if generated by the edge from vertex i to vertex j in the control mesh.

except that here we operate in screen space. (See Figure 3.3.) Specifically, we traverse the mesh from T', moving from triangle to triangle along a path whose screen-space projection leads to $\mathbf{x}_s$.[3] This procedure returns a point $\mathbf{y}$ whose screen-space location is $\mathbf{x}_s$. Finally the question is answered: $\mathbf{x}$ is visible if and only if $\mathbf{y} = \mathbf{x}$.

3.4 Triangle strips and subdivision meshes

One service provided by the patch is to find and draw triangle strips. The use of triangle strips can significantly increase rendering speed on modern graphics hardware. Evans *et al.* [19] survey methods for processing a mesh to extract good-quality triangle strips.

Meshes in our system may be subdivided; we use loop subdivision [36, 47, 69]. In this section we briefly describe a simple, effective and fast algorithm for computing triangle strips of a subdivision mesh, given a set of triangle strips for the control mesh.

Our approach is similar in spirit to that of Velho *et al.* [80]. Their method efficiently constructs generalized triangle strips in the context of a hierarchical mesh. That is, a base-level (coarse) mesh is given, together with a sequence of refinement operations. These operations are applied simultaneously to the triangles of the mesh and to its triangle strips.

[3]We will be able to reach $\mathbf{x}_s$ unless a silhouette is encountered, in which case we fail to determine the visibility of $\mathbf{x}$.

The result is that good-quality triangle strips are generated for each refinement of the mesh with no additional work beyond bookkeeping.

The difference between our method and that of Velho *et al.* is that our method is valid for the particular type of refinement used in Loop subdivision. Also our method computes regular (not generalized) triangle strips; OpenGL [12] is optimized for regular triangle strips. In both methods, the key idea is to compute triangle strips on the base-level mesh – where there are few triangles – then exploit the special structure of the refinement method to simply "read off" good-quality strips in the refined meshes.

Our method is quite simple. Each strip in a given mesh generates two strips in the next-level subdivision mesh. We call the original strip the "parent," and the two subdivision strips the "left" and "right" children. We illustrate this in Figure 3.4.

Given that the representation for the parent strip is just the list of its vertices:

{0, 1, 2, 3, 4, ... },

we can read off the list of vertices for the two child strips according to the following pattern:

parent	0	1	2	3	4	5	6	...
left child	00	10	20, 21, 22	32	42, 43, 44	54	64, 65, 66	...
right child	10	11	21	31, 32, 33	43	53, 54, 55	65	...

Table 3.1: The vertices of the two child strips are related to those of the parent strip according to the pattern shown. The notation is the same as described in Figure 3.4.

With this scheme, the average length of strips doubles with each level of subdivision (and there are twice as many of them). Thus, with a few levels of subdivision the strips are essentially optimal, as far as graphics performance is concerned. (We don't know if an optimal stripping of the control mesh implies an optimal stripping of the subdivision mesh using this scheme).

This method for finding triangle strips in a subdivision mesh can be useful particularly when the control mesh is retriangulated interactively while a subdivision mesh several levels down is being displayed. This could be the case if a subdivision mesh is displayed to the user, who performs edits that indirectly alter the triangulation and vertex positions of the control mesh, then sees the resulting changes directly in the subdivision mesh. The usefulness is that the stripping algorithm is fast and effective, and so increases the interactivity of the system.

Chapter 4

Shading with hatched strokes

4.1 Introduction

In this chapter we develop several techniques to convey grey tones using hatched shading strokes with variable widths and spacing. We begin with a simple approach based on a screen-space arrangement of strokes, and show how the reference image framework described in the previous chapter can be used to determine stroke visibility as well as thickness and spacing to convey grey tone. We then extend this basic approach to include a method for bending the strokes to suggest the curvature of the shaded model. This extension relies on some annotated 3D shape information, but is primarily carried out in screen-space.

4.2 A simple screen-space approach

In this section we describe a simple technique for shading a patch with parallel strokes aligned in some fixed screen-space direction. The strokes have fixed spacing but vary their width to convey tone (and are clipped to the visible portions of the patch).

4.2.1 Stroke placement and visibility

At each frame, we first calculate a suitable arrangement of strokes that covers the entire patch. We begin by computing the pixel bounding box of the patch.[1] This gives us a rectangular frame within which we can arrange the strokes.

[1]Recall from Chapter 3 that the patch stores a list of its pixels for the current frame.

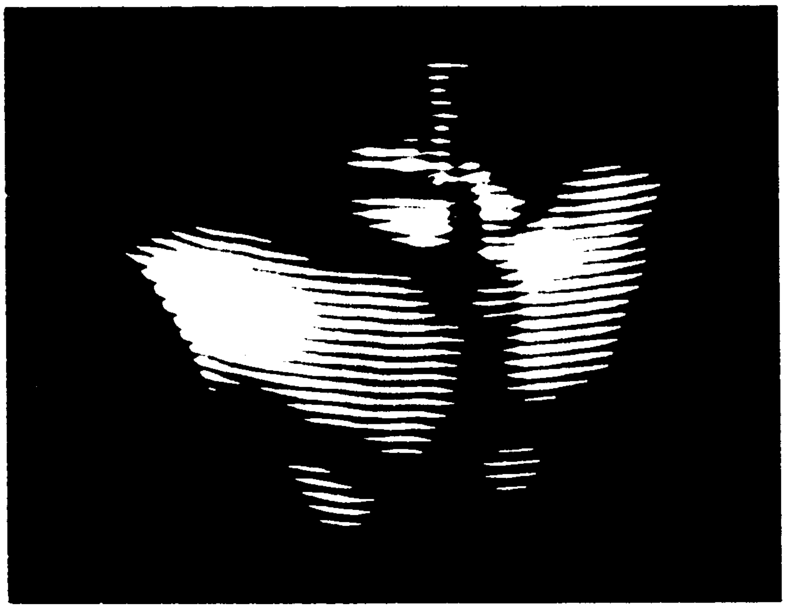


Figure 4.1: This image shows off cross-hatching in the dark areas and an additional layer of highlight strokes. "Bent strokes," described in section 4.3, are used for the highlight layer and for one of the cross-hatch layers. All three layers use tapered wiggly strokes.

Strokes are generated by creating line segments in the given direction with the given spacing and clipping them to the bounding box. Each stroke is then sampled at some fixed spacing (about ten pixels in all of our examples). At each sample point we can use the ID reference image to tell in $O(1)$ time whether that point lies in the given patch. Strokes are clipped to consist of segments that lie entirely in the patch. Once visibility of strokes is determined, screen-space stylizations (e.g. wiggles, as in Chapter 2) can be applied to the strokes to reduce the mechanical appearance of the straight, parallel lines. More information on how we implement a variety of stylized strokes in OpenGL is provided in an up-coming paper [54].

4.2.2 Shading

We "shade" strokes by adjusting their widths to convey grey tones. Since the spacing of the strokes is known, it's straightforward to compute a width at a point on the stroke given the target grey tone there. If the tone is given on a scale of 0 (white) to 1 (black), then the stroke width at a point is simply the stroke spacing multiplied by the target tone at that point. Once the stroke path and its varying widths have been decided, it can be rendered as a quad strip with a pair of vertices at each sample point along the path, offset toward opposite sides of the path by half the width. That is, we use a simple implementation of "skeletal strokes," described in [40, 39]. The endpoints of strokes can be tapered – see for example the "highlight" strokes in Figure 4.1.

This scheme supports multiple layers of strokes, as occurs with cross-hatching or the use of both darkening and lightening strokes on a mid-tone background. To achieve cross-hatching, we apply strokes in two passes. In the first pass, strokes only perform shading up to some maximum target darkness, T. In a second pass, we apply strokes in a distinct direction. To compute shading, strokes first subtract T from the tone values found in the color reference image. This is done to account for the amount of tone already conveyed by strokes in the first pass.

Salisbury *et al.* [66] take great care to account precisely for the "lost" darkness due to overlapping strokes. This could be added to our algorithm, but seems unnecessary in practice. Indeed, we sometimes, for aesthetic effect, alter the mapping that goes from the darkness image to the stroke size, so the precise mapping isn't critical. Examples of effects achieved this way are shading only in very dark regions, or shading lightly everywhere. We currently alter the tone mapping by letting the user explicitly assign a tone value below which the stroke will have zero width and a tone above which it will have full width. The default is to have zero width at zero tone and full width at full tone. To shade only in dark regions, for example, the minimum width is set high, and the maximum width is set to full tone. Tones found below this unusually high minimum width will be rendered with zero width – i.e., be invisible. This simple mechanism gives the illustrator extra flexibility.

4.3 Extensions to the basic rendering technique

Extension 1: Layering. We can render with layers of styles. The user can choose from variety of base coats ranging from a solid color wash to various qualities of two-tone texture. The user can also layer stroke types to create effects like cross-hatching and highlighting strokes, as shown in Figure 4.1.

Two-tone tricks: At the most basic level, we generate two-tone shading by thresholding, on a per-vertex value, the dot-product of the vertex normal and the camera-to-vertex direction vector. The mesh is divided into triangles below and above the threshold; those below are rendered in one color; those above in another. Triangles straddling the threshold

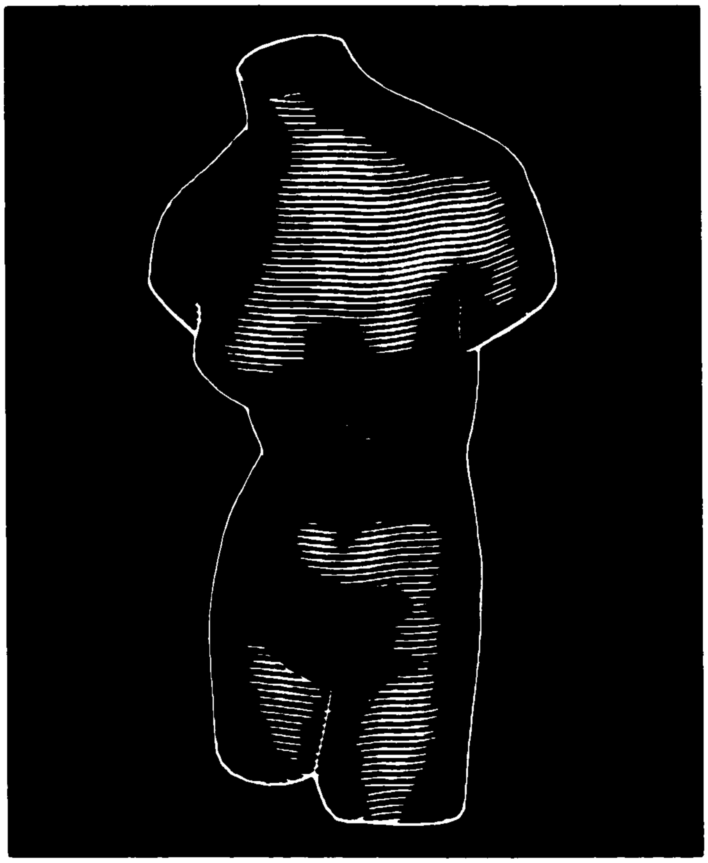


Figure 4.2: This model shows how strokes can be made to bend to indicate curvature. Our bends are not completely faithful to the underlying geometry, but nor are those used by many artists - surprisingly often they are mere indications of shape rather than precise representations.

are divided by linearly interpolating the value over them, and the two pieces are each rendered in the appropriate color. Further, when drawing strokes, we can detect the threshold value and taper the stroke when we pass the threshold, tapering to zero over a few pixels, creating a sawtooth effect often used by Charles Burns [14] (an example of his work is shown in Figure 1.3). We have even arranged for strokes to turn at the threshold boundary and continue in a new direction, normal to the threshold boundary, and taper, to more closely mimic Burns' style. This sawtooth effect can be seen on the mattress in Figure 4.3.

Extension 2: Bent strokes. There is a common illustration shorthand where the artist suggests the curvature of a surface by slightly bending strokes as they approach edges. This works well for roughly tubular shapes such as torsos, arms, legs, horns, etc.

Figure 4.2 shows an example of this type of effect generated by the following technique.

To use bent strokes, the user creates a 3D path through the object, called the *spine*. The user then sketches widths at selected points along the spine to indicate the radius of a generalized cylinder at those points – a cylinder that's supposed to approximate the underlying object. (The radius at other points of the spine is interpolated from the sketched examples). At each frame, the spine is projected into screen space and its projection is evenly sampled at a fixed pixel spacing (about 10 pixels in our examples). At each sample point, we find the corresponding point on the spine, and consider the circular cross-section of the generalized cylinder there. We compute the projection, B, of a point some fraction of the way[2] along the radius from the spine point to the nearest (to the camera) point of the cross-section, and the projections, A and C of the two endpoints of the film-plane-parallel diameter of that cross-section. We then create a curved stroke from A through B to C, clipping to the patch as usual. The shape of the curve is not particularly important – we've used a half-period of a scaled sine function in our examples, but other shapes we've tried have also looked good. At points where the spine's tangent is parallel to the film plane, A, B, and C will be collinear and there will be no curvature; where the spine is tilted out of the film plane, the curvature will be larger.

Drawbacks: Where the spine is tilted far out of plane, the "front half circle" that we're drawing doesn't always convey the whole shape – the "back" half-circle may be visible too. And where the curvature of the screen-projection of the spine is high, adjacent strokes may be compressed on the convex side of the spine, even to the point of crossing one another. Because of this, we actually stop drawing curved strokes at places where the spine-tangent and view are too close to parallel.

4.4 Discussion

We can create real-time renderings of moderately complex polygonal models in a stroke-based style that allows user annotation of stroke style, spacing, and placement (in addition to base coats). There are still limitations – for example, curved strokes self-intersecting. And if the object being rendered is not well-approximated by a generalized cylinder, then the curved-stroke method fails.

As with most stroke-based rendering schemes, inter-frame coherence is a problem – when a new stroke is added, it "pops" into existence, and this can be distracting. And

[2]About 0.6 in our examples.

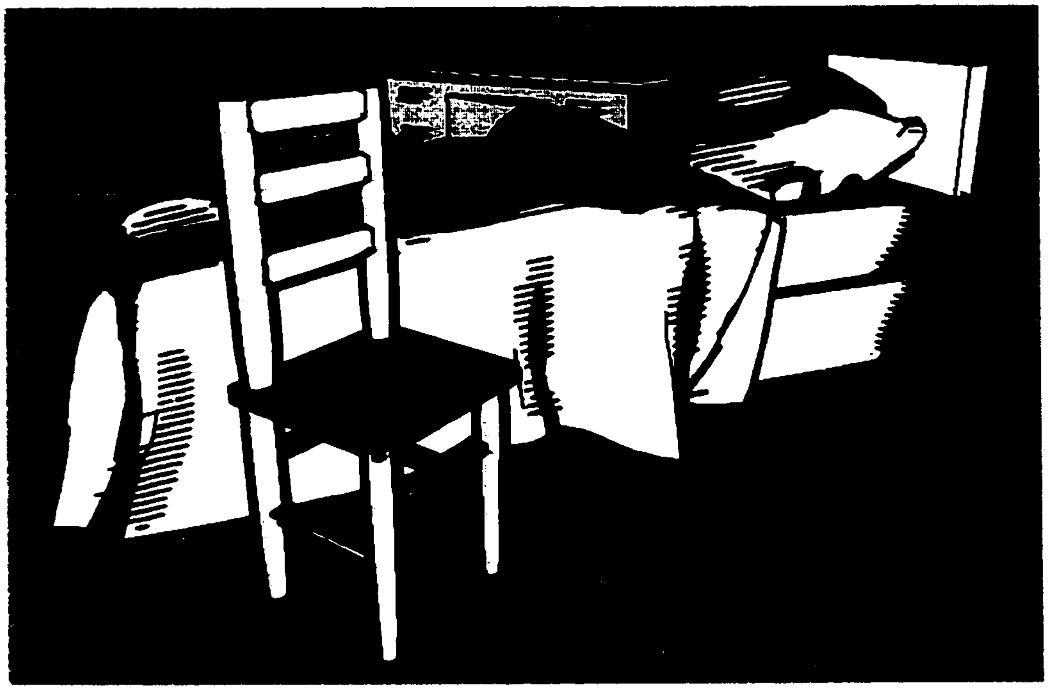


Figure 4.3: Note the haloed shading strokes on the blanket, and the two-toned shading with sawtooth effect on the mattress.

when objects are scaled, the strokes, which remain in constant position on the screen, seem to "swim" inwards through a peculiar optical illusion. We're eager to address these limitations in future work, and to extend our stroke-based rendering scheme to new styles.

Chapter 5

Fur, grass and trees

5.1 Introduction

Any art student can rapidly draw a teddy bear or a grassy field. But for computer graphics, fur and grass are complex and time-consuming. Even so, the artist's few-stroke rendering may have greater persuasive or evocative power than the usual computer-graphics rendering.

How does the artist effectively communicate the teddy bear or grass? By rapidly creating an impression of free-form shape – difficult to do with conventional 3D modeling systems – and then drawing a few well-chosen strokes. This chapter describes some of our efforts to expand the expressive power of 3D graphics by adopting techniques for depicting complexity from the centuries-old disciplines of art and illustration.

Three goals in our work on art-based graphics are to give the designer of a scene control over the style of rendering; to ease the burden of modeling complex scenes by treating the rendering strategy as an aspect of modeling; and to provide interframe coherence for the kinds of stylized renderings we've developed. Other goals, less directly related to the work in this chapter, include the development of systems for rapidly creating free-form shapes [49], and the control of scene composition.

Our approach to creating complex expressive renderings is to target the kinds of images made by artists and illustrators and reproduce the effects and techniques we observe in those images. Much 2D art and illustration is created by making strokes on a flat surface (paper, canvas), so we have based our work on what we call "stroke-based textures." We started from the drawings of Theodore Geisel ("Dr. Seuss") [24, 25], in part because they are such an extreme departure from the domain of conventional computer graphics.

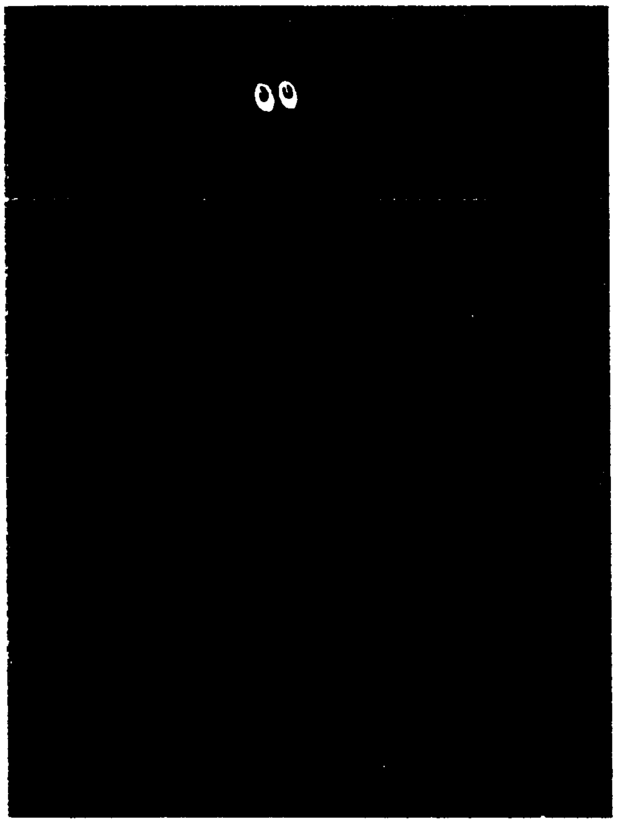


Figure 5.1: A furry creature, after Dr. Seuss. The fur is generated in a view-dependent way by a procedural stroke-based texture that places it near silhouettes, varying the style according to surface orientation.

There are two main research challenges: (1) the development of a software framework and algorithms for rendering procedural stroke-based textures, and (2) the development of a user interface for a free-form modeling system that lets a designer create and customize such procedural textures. In this chapter, we build on the software framework described in Chapter 3 to develop a new algorithm that addresses the first of these challenges. The second challenge remains future work, though in an up-coming paper [51] we describe new work in this direction.

We have developed a system to generate and render stroke-based textures that mimic the styles of two artists, and a framework for generalizing this to other techniques. The system renders the images shown in this chapter at several frames per second on a Sun Sparc Ultra 2 model 2/300 with Creator 3D graphics, and even faster on a high-end PC.

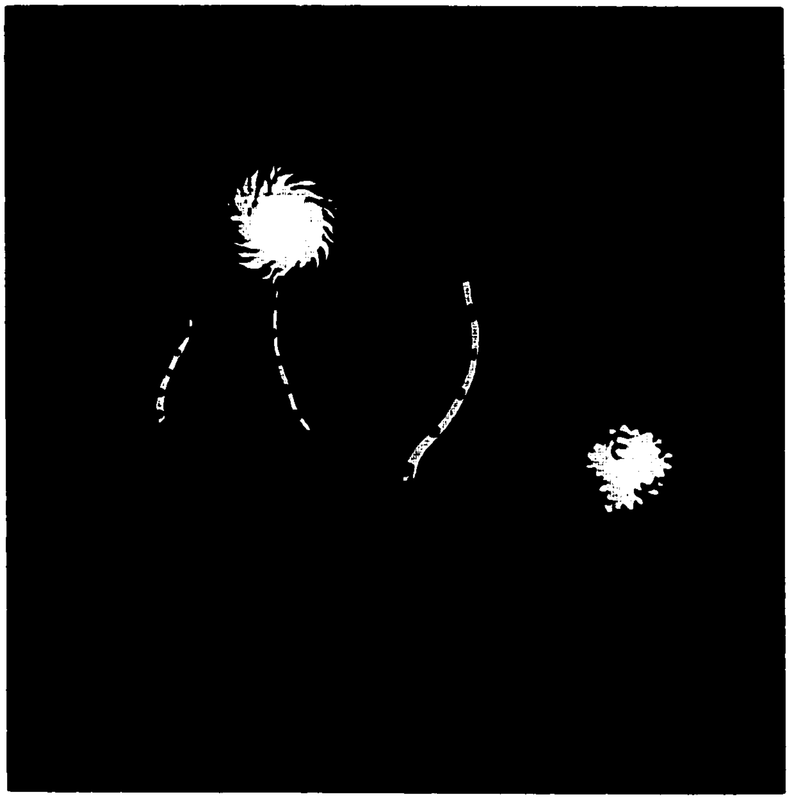


Figure 5.2: A more complex scene, again based on the style of Dr. Seuss. The grass, bushes, and truffula treetops are implemented with graftal textures that use the same basic algorithm to place graftals with a variety of shapes and drawing rules. The truffula tree trunks are drawn by stroke textures (not graftal textures) assigned to ribbon-like surfaces that always face the viewer.

Our main contributions are the system architecture, the (partial) temporal coherence of the texture elements, and the particular methods used to mimic Dr. Seuss's and Geoffrey Hayes's [33] styles.

5.2 Prior Work

Using art as a motivation for computer graphics techniques is not new, and our work builds on the efforts of many others. Fundamental to our ideas are the particle systems of Reeves [60, 61], which he used to create trees, fireworks, and other complex imagery from relatively simple geometry. Alvy Ray Smith's later use of particles, together with recursively

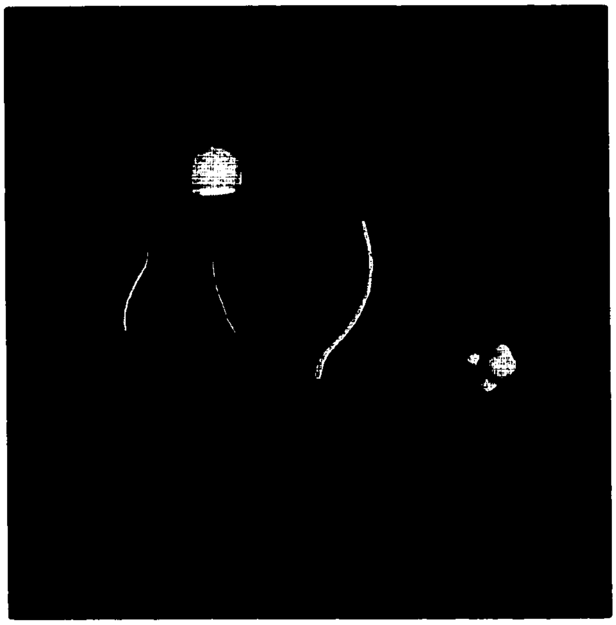


Figure 5.3: The same scene as in Figure 5.2 rendered without graftal textures or the stroke-based textures on the truffula trunks.

defined L-systems that he called "graftals," extended this to more biologically accurate tree and plant models [71]. His "Cartoon Tree" is a direct precursor to the work described here. Graftals have since come to be described more generally: according to Badler and Glassner [5], "Fractals and graftals create surfaces via an implicit model that produces data when requested." We use the word "graftal" in this much more general sense.

We use a modified version of the "difference image" stroke-placing algorithm of Salisbury *et al.* [66] to place procedural texture elements at specific areas of the surface. Winkenbach and Salesin [84] described the use of "indication" (showing a texture on part of an object) in pen-and-ink rendering. And Strothotte *et al.* [73] extensively discuss the use of artistic styles to evoke particular effects or perceptions.

At a more mechanical level, Meier's work on particle-based brush strokes [52] was a major inspiration in two ways: first, her use of particles to govern strokes that suggest complexity in her Monet-like renderings showed that not all complexity need be geometric; second, the fixed spacing of the particles on the objects, which limited how closely one could zoom into the scene, inspired us to seek a similar but hybrid screen/object space technique.

The work in this chapter builds on the procedural texture framework we developed in

Chapter 3. For instance, the fur texture on the creature in Figure 5.1 is applied over most of the body but not the face, even in profile. Other images in this chapter show more examples of the selective use of distinct nonphotorealistic textures applied to objects in a scene according to what each represents.

5.3 Graftal Textures

The textures described in this section place fur, leaves, grass or other geometric elements into the scene procedurally, usually to achieve a particular aesthetic effect (e.g., indicating fur at silhouettes but tending to omit it in interior surface regions). We'll call this class of textures *graftal textures*. They all share the same basic procedure for placing tufts, leaves, grass, etc., all of which we call *graftals*. The key requirements are that graftals be placed with controlled screen-space density in a manner matching the aesthetic requirements of the particular textures, but at the same time seem to "stick" to surfaces in the scene, providing interframe coherence and a sense of depth through parallax.

5.3.1 Placing graftals with the difference image algorithm

To meet these requirements, we have adapted the "difference image" algorithm (*DIA*) used by Salisbury *et al.* [66] to produce pen-and-ink-style drawing from grayscale images. Their algorithm controls the density of hatching strokes in order to match the gray tones of the target image. For each output stroke drawn, a blurred image of the stroke is subtracted from a "difference" image (initially the input image). The next output stroke is placed by searching in the difference image for that pixel most (proportionally) in need of darkening, and initiating a stroke there. The resulting image consists of marks whose density conveys the gray tones of the original.

The DIA meets our first requirement of placing marks (or in our case, graftals) with a controlled screen-space density. To control graftal placement according to a particular aesthetic requirement, each graftal texture simply draws its patch into the color reference image so that darker tones correspond to regions requiring a denser distribution of graftals. We call the result the *desire image*, and the value at a pixel in that image measures the *desire* that graftals be placed there. For example, to render the furry creature in Figure 5.1, the reference image is drawn darker near silhouettes – easily done by placing a point light near the camera position. Also, some regions (e.g., the feet) can be explicitly darkened by the designer to promote a greater density of graftals there.[1]

To meet the requirement that graftals appear to stick to surfaces in the scene, we must convert the 2D screen position of a graftal (assigned to it by the DIA) to a 3D position on some surface. This is achieved in $O(1)$ time (per graftal) by using the ID reference image (described in Chapter 3) to find the triangle (and the exact point on the triangle with a ray-test) corresponding to a given screen position.

This now allows graftals to be distributed over surfaces in the scene to achieve a desired screen-space density – for a single frame. To create some interframe coherence, we modify the algorithm:

- In the first frame, graftals are placed according to the DIA.
- In each successive frame, the graftal texture first attempts to place the graftals from the preceding frame.
- Then, when all the "old" graftals have been considered for placement and accepted or rejected, the graftal texture executes the DIA to place new graftals into the scene as needed.

An existing graftal may fail to be placed in a frame for two reasons: (i) the graftal is not visible (it is occluded or off-screen); (ii) there is insufficient desire in the desire image at the graftal's screen position. This can happen if the original desire value at the graftal's screen position was small (e.g. the graftal is far from a silhouette). It can also happen if the camera has zoomed out and the graftal's neighbors, now closer to it in screen space, have already subtracted the available "desire" in the vicinity. When a graftal fails to be placed in a frame, it is discarded. Otherwise it updates its attributes (as described below) and is drawn.

We make a final modification to the DIA: we use a bucket-sort data structure to find the pixel with the greatest desire. This key step can be completed in $O(1)$ time rather than the $O(\log(n))$ quad-tree method of Salisbury *et al.*, where n is the number of pixels.[2] This lets the algorithm run at interactive speeds on simple scenes.

[1]To further encourage drawing near silhouettes, we filter the desire image, replacing each desire value d with $2d - d^2$, where d ranges from 0 (no desire) to 1 (maximum desire).

[2]We thank Ken Lao for suggesting this idea.

Figure 5.4: A tree rendered in the style of Geoffrey Hayes [33]. The leaves are drawn with graftals based on OpenGL triangle fans, rather than the triangle strip-based type of graftal shown in Figure 5.5. The interior is shaded with the technical-illustrator shader of Gooch *et al.*[27].

5.3.2 Subtracting the blurred image

When a graftal is placed in the scene (either initially or in subsequent frames) it subtracts a blurred "image" of itself from the difference image. For this, graftals are treated as points with a given (variable) screen size, so the blurred image is just a Gaussian dot.

Pixels in the desire image are encoded with values ranging from zero (no desire) to one (maximum desire). Each graftal has an associated "volume" that determines how much total "desire" it subtracts from the desire image. This volume is proportional to the graftal's approximate screen space area. Intuitively, a visually large graftal subtracts a large volume, corresponding to a wide blurred dot: this eliminates desire in a wide region near the graftal, preventing others from being placed there.

Graftals can scale their geometry and volume so that they tend to maintain a desired screen-space size and relative density. For example, strictly adhering to the laws of perspective when zooming away from the model could result in graftals being drawn too small to be individually discernible. An artist might choose to draw them larger than they would realistically appear in this case. In any case, graftals that appear smaller in screen space should scale their volume accordingly in the DIA , or they will be placed too sparsely.

To perform such compensatory scaling, each graftal must keep track of its approximate screen space size. It does so by first converting its object-space length L to a screen-space measurement s in every frame (ignoring foreshortening). Then it chooses a scale factor r by which to multiply L as follows. As part of its definition, the graftal is given a desired screen space length d and corresponding volume v_0 (chosen by the user). At one extreme the graftal could take $r = d/s$, so that it always appears the same size on the screen regardless of distance. At the other extreme it could take $r = 1$, which would be strictly realistic. In our examples, we have taken a weighted average between the two extremes:

$$r = w(d/s) + (1 - w),$$

with weight $w = 0.25$. This approach moderates the degree to which the graftal scales with distance, providing a measure of resistance to change from its ideal size. Finally, the volume in each frame is calculated as $v = v_0(rs/d)^2$ to keep it proportional to the graftal's current screen size.

Now let d_0 be the value in the desire image at the graftal's screen position $\mathbf{x}_0$ (which has been verified as visible). Let $v > 0$ be the volume of the graftal. We seek a 2D Gaussian function g such that

$$g(\mathbf{0}) = d_0 \qquad \text{and} \qquad \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(\mathbf{x})\, dx dy = v.$$

This is given by $g(\mathbf{x}) = d_0 e^{-\pi d_0 |\mathbf{x}|^2 / v}$. Thus, to subtract the "blurred image" at $\mathbf{x}_0$, we subtract $g(\mathbf{x} - \mathbf{x}_0)$ from each pixel location $\mathbf{x}$ in the desire image. Of course, outside of some radius values of g are negligible. If we take m to be the minimum value we can represent in the 8 bits we use to store desire, then this radius is $(v \log(2d_0/m)/(\pi d_0))^{1/2}$. We only subtract g from pixel locations within this radius of the graftal's screen position.

As the graftal subtracts this Gaussian from the desire image, it records the total desire subtracted. (It can't subtract more from a pixel than is stored there.) When all goes well, this quantity should equal the volume v (ignoring discretization errors and the small portion of the Gaussian outside the maximum radius above). If the total is less than v, the graftal may draw itself with a reduced level of detail. If the total is too low (below 0.5 in all our examples), the graftal reports failure to its texture and is removed from the scene. To avoid "popping" when graftals appear and disappear, they may initially be drawn with reduced detail, quickly increasing to full detail over a short time, and reversing the process when they are removed. This has its limitations, though, as we discuss below.

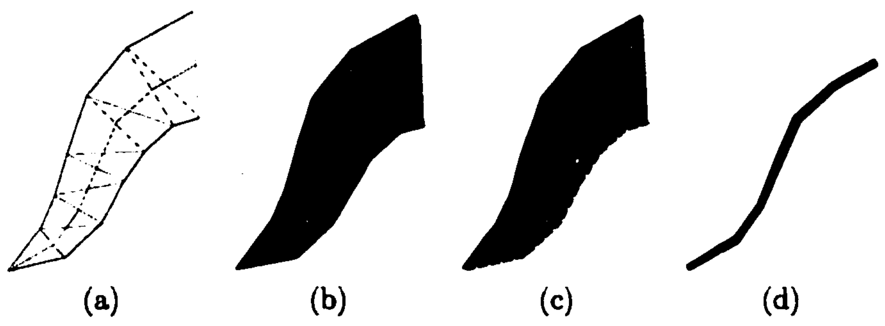


Figure 5.5: A fur graftal is based on a planar polyline and table of widths, used to construct a GL triangle strip (a). The graftal can render itself in three ways: It can draw a set of filled polygons with strokes along both borders (b) or just one (c); or it can draw just the spine (d).

5.3.3 Details of fur graftals

A fur graftal – the kind used for the furry creature in Figure 5.1, and for the truffula tufts and grassy mounds at the base of the trees in Figure 5.2 – is not particularly complex. It is based on a flat tapering shape by a gradually reducing width about a central spine (see Figure 5.5). The central spine is a planar polyline, and the taper widths are recorded in an array. For the model in Figure 5.1, just a few taper widths were assigned; for the truffula tufts there were about seven. The shape of the central spine was drawn on graph paper and entered by hand.

After being placed with the DIA, each fur graftal determines how to orient and draw itself by computing the dot product d of the unit surface normal $\vec{n}$ at the graftal's position with the unit view vector $\vec{v}$ from the camera to its position (see Figure 5.6). For varying values of d, the tuft may be drawn filled, filled with one edge, filled with both edges, as a spine only, or not at all. For the fur in Figure 5.1, we draw just the spine for $-0.75 < d < -0.6$. We draw the filled tuft with both edges for $-0.55 < d < 0$. Other schemes are possible, and we believe that adjusting the thresholds and drawing styles during "fade-in" and "fade-out" might help smooth these transitions.

Finally, the fur graftals are oriented to face the camera – that is, to lie in the plane containing the underlying surface normal and most nearly orthogonal to the view vector. They're placed so that in general they bend down. This behavior can be modified (as in the truffula tufts) so that they point clockwise, or so that they follow directions that have been "painted" onto the graftal texture's patch, as in the feet in Figure 5.1.

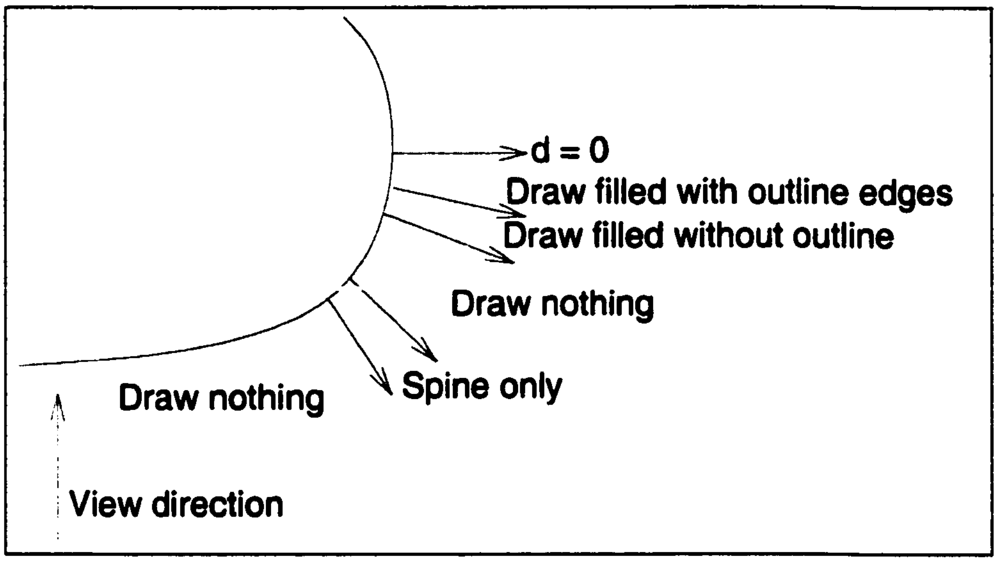


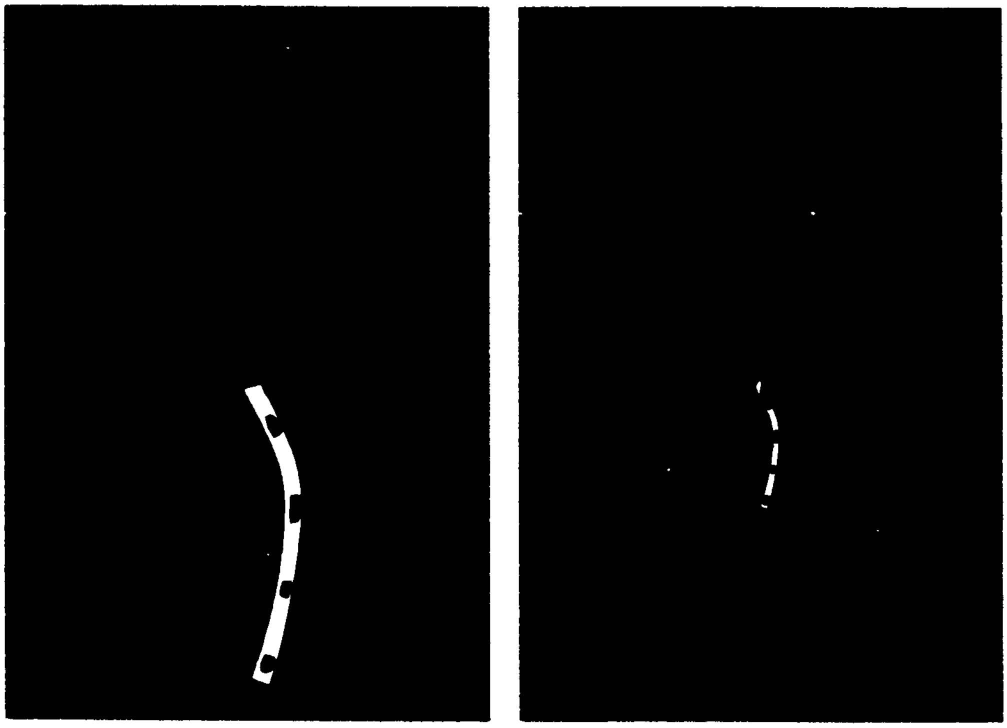
Figure 5.6: The dot product *d* of the view vector and surface normal determines a graftal's drawing style. In Figure 5.1 the "Draw filled without outline" region is empty: we transition directly from "filled with outline" to "draw nothing."

5.4 Results and Future Work

Our system can produce scenes that evoke a remarkable sense of complexity, in a style that's new to 3D graphics, and at interactive rates. Figures 5.2, 5.4 and 5.9 show the kinds of results that can be achieved with graftal textures. In each case the underlying geometry was simple to produce, yet the renderings have an expressiveness often lacking in computer graphics imagery. With our system, even the truffula scene can be rendered at several frames per second on a high-end PC.

The Siggraph 99 Conference Proceedings Video Tape shows our system in action. It includes sequences captured in real-time and animation sequences rendered off-line and played back at significantly higher frame rates. The problem of poor frame-to-frame coherence stands out most noticeably in the latter case. Graftals that persist from frame to frame maintain geometric coherence (if we simply redistributed graftals at every frame, the flicker would be overwhelming); unfortunately, the DIA has no inherent interframe consistency, so it's easy for a graftal to be "crowded out" in one frame, replaced in the next, and so on, causing the flickering artifacts that are so noticeable in the video.

We have recently begun experimenting with some strategies to address this problem. One possibility is to make much greater use of fading and alpha blending when introducing graftals into the scene and taking them out. The degree to which this approach is usable



(a) (b)

Figure 5.7: A truffula treetop with static graftals organized in a three-level hierarchy. (a) When the camera is close, all three levels are drawn. (b) As the camera zooms out, only two levels are drawn. (See Figure 5.8.)

depends quite a lot on the specific style being targeted. Fading in a large yellow truffula tuft outlined in black against a blue sky may be just as jarring as introducing it suddenly; but fading in semi-transparent blades of grass rendered with a watercolor-like effect over green terrain might seem perfectly acceptable.

One problem we have encountered in our early experiments with fading in tufts of fur like those in Figure 5.1 occurs when all the tufts along a silhouette are newly introduced, and thus nearly transparent. The model then appears (briefly) to be missing its fur along that silhouette. A possible solution that we have not yet implemented is to maintain a separate population of tufts drawn on back-facing surfaces. This requires an auxiliary ID reference image prepared with front-facing triangles culled. It also requires two separate calls to our modified DIA each frame - one to place front-facing tufts, another to place back-facing ones. The point is that tufts emerging into view from behind a silhouette (as the object turns) would already be drawn and thus would not "pop in." Each frame might

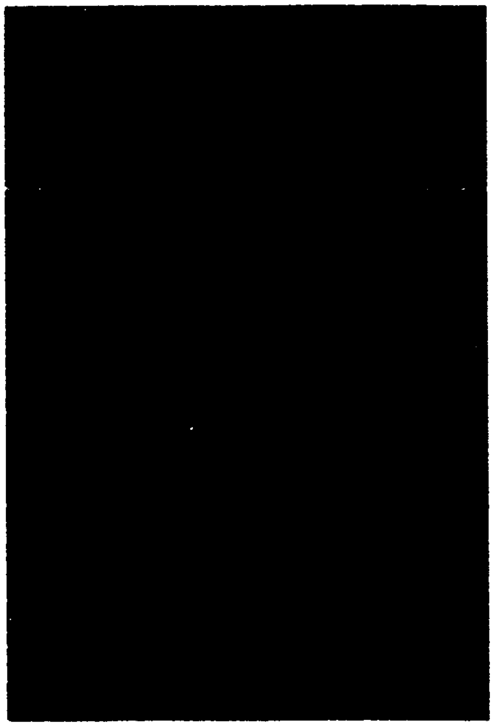


Figure 5.8: The final, coarsest level of the three-level hierarchy of Figure 5.7.

take twice as long to render - possibly a worthwhile trade-off if the resulting animations are significantly more watchable.

Another strategy we have experimented with is to use *static* graftals (see Figures 5.7 and 5.8). With this approach, graftals are assigned fixed positions on the surface, rather than being generated each frame as needed. They still draw in a view-dependent way - those far from a silhouette, say, may not draw at all. This works quite well as long as the camera does not zoom out too far: in that case the graftals are drawn too densely in screen space. We can overcome this by assigning graftals several levels of priority - say numbered 0 through 2. Each level is distributed evenly over the surface, with those in a given level outnumbering those in the next lower level by a factor of about four. In a given frame, every graftal at level 0 "draws" itself view-dependently (possibly not at all if far from a silhouette). Each also subtracts its blurred image from the desire image, as in the DIA , and measures its success rate. If, collectively, this rate is high enough, the next level is given the chance to draw, and goes through the same procedure to decide whether the last level should also have the chance to draw. This strategy is suitable for localized objects - we have tested it on versions of the truffula treetops – but not for landscapes where the choice of what level graftal to draw must vary over the surface according to distance from the camera. Our preliminary results indicate the effectiveness of this strategy. We demonstrate this in the Siggraph 99 Conference Proceedings Video Tape. Figures 5.7 and 5.8 shows a truffula tree top with static graftals drawn at three levels of detail.

An up-coming paper [51] describes our recent work that extends the idea of static graftals, and in which we develop a more flexible framework for creating a wide range of graftal shapes and specifying their behaviors under different viewing conditions.

A separate area for future work is to explore a range of new styles. For example, we have begun to explore rendering fur using two layers, one light and one dark, on a neutral background (see Figure 5.9). This is a common technique in traditional drawing that can suggest more complex lighting effects (such as glossiness) while omitting unnecessary detail: nothing is drawn in mid-tone regions, which are represented by the unaltered background color.

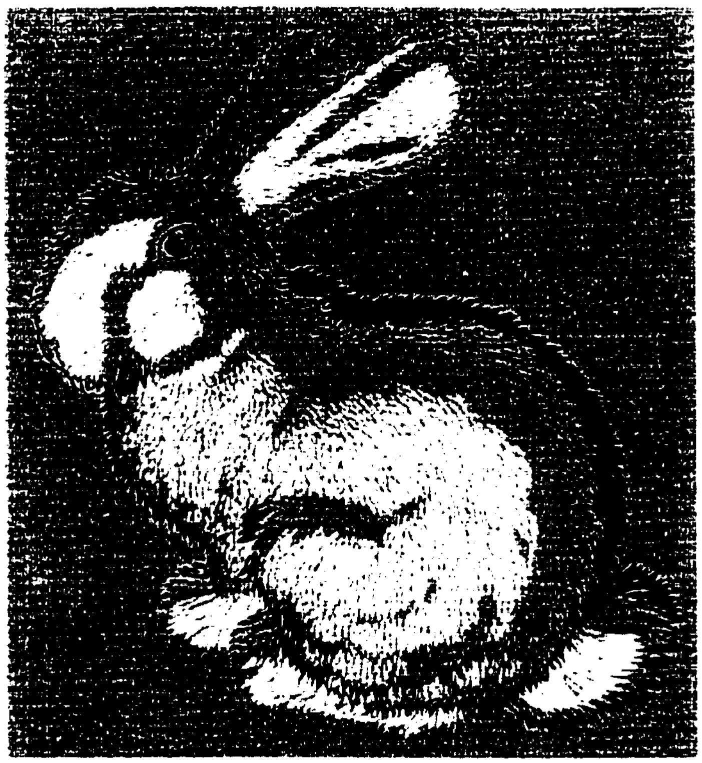


Figure 5.9: Fur rendered in light and dark layers on a neutral background. Model courtesy of Stanford University.

Part II

Modeling

Chapter 6

Sketching 3D curves

6.1 Introduction

Specifying 3D curves is one of the most important tasks that a 3D user interface must support. Curves are used in modeling and CAD systems to specify surface patches [2, 4], as skeletal shapes for implicit surfaces [11], and to define controls for object deformations [70]. Animation systems and VR applications use curves to specify motion and camera paths [16, 41, 55].

Many authors have recognized the importance of specifying curves directly [7, 22, 31, 42, 87, 88]. Although sketched curves are imprecise by nature, sketching allows a user to quickly create a curve that is close to the desired result, even if she has little experience with the underlying curve representation. A novice user can quickly create approximate curves because little overhead is required to learn the interface. A trained artist can apply her existing drawing skills to produce accurate curves because the interface more closely matches the one she is used to – namely pencil and paper.

The technique we present is an extension of the idea used in [34, 87] that a point in 3D can be determined from its image-space projection together with that of its "shadow." (The "shadow" is just the vertical projection of the point onto some horizontal surface.) We apply this idea to a connected set of 3D points to define a curve. With this approach, the user sketches a curve directly into a scene in two strokes: first drawing the curve as it appears from the current viewpoint, and then sketching its approximate "shadow." The effect is to redefine the 3D shape of the curve while leaving its appearance unchanged. The user can then refine portions of the curve by over-sketching either its projected image or that of its shadow. Although this technique is less precise than previous methods, it lets the user quickly sketch a reasonably correct shape that may be further refined with more conventional methods.

6.2 Previous work

Some techniques for editing curves are indirect in that they require the user to modify parameters, e.g. spline control points or knot values, that in turn affect the curve's shape. Other techniques allow direct manipulation of the curve itself, such as the overdrawing paradigm described by Baudel [7] and direct manipulation of spline curves [22, 31, 88]. The technique we present falls into this latter category.

Although much work has been done in sketching 2D curves [6, 7, 42, 68], few systems have addressed the issue of sketching curves in 3D. One notable exception is the 3-Draw system [64], which uses Polhemus trackers attached to a stylus to allow a designer to sketch in 3D with arm motions.

Commercial 3D modeling systems such as Maya and 3DStudio [2, 4] give the user a variety of techniques for creating and editing curves. Most of these, however, are indirect (e.g. the user edits spline control points or intersects two surfaces). In Maya a user can sketch a curve directly onto a user-defined plane, or more generally onto a surface. Of these techniques, only the latter constitutes a direct method for specifying nonplanar curves. But the user can draw on a surface only where nothing occludes it. To draw all the way around a sphere, for example, the user must draw from multiple camera positions. Thus, there are restrictions on the types of curves that can be sketched from a single view.

The interface we present complements existing 3D modeling systems by providing additional flexibility for directly sketching nonplanar curves.

6.3 Overview of the system

We support four basic operations for sketching curves: drawing a new curve in some plane, "overdrawing" a section of an existing curve, redefining a curve's entire shadow, and overdrawing a section of a curve's shadow. Figure 6.1 illustrates the steps involved in creating and editing a curve. To distinguish between operations that edit the shadow and operations that edit the curve, the user selects either shadow mode or curve mode via a menu or keyboard shortcut.

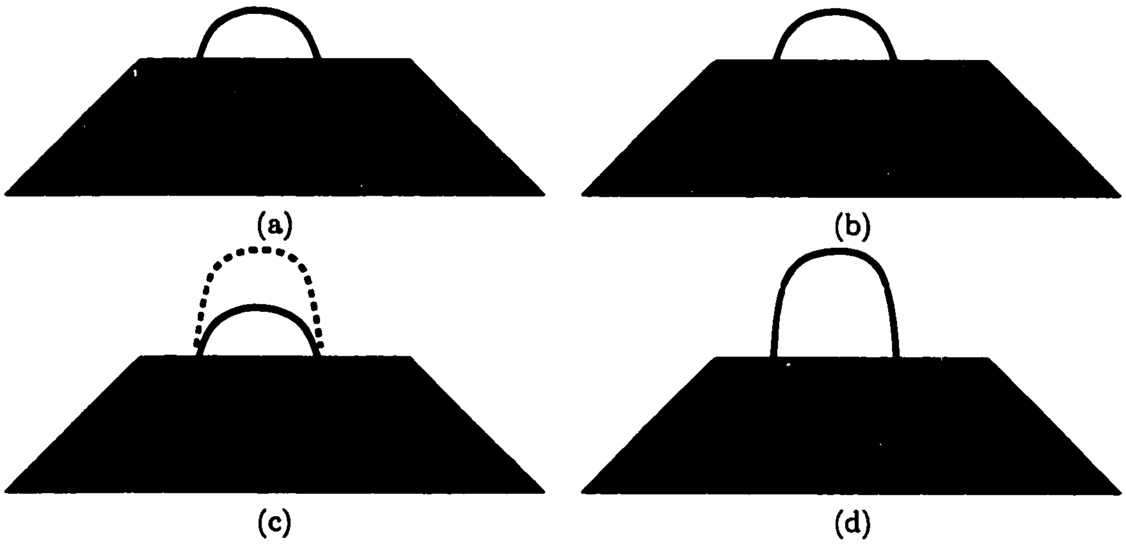


Figure 6.1: (a) A single stroke creates the initial curve. (b) A second stroke defines the curve's shadow and hence its 3D shape. (c) The dashed line indicates an overdraw stroke. (d) The system blends the overdraw with the original curve to get the final result.

When the user draws a stroke in curve mode or shadow mode, the system determines whether the stroke is an "overdraw" by checking whether it starts and ends near and nearly parallel to an existing curve or shadow. If so, we merge it into the existing curve using a method similar to that described in [7]. In curve mode, if the stroke is not an overdraw, the system interprets it as a new curve that is projected onto a plane; the plane is determined by a set of heuristics described below.

To define a shadow, the user (in shadow mode) draws a stroke beneath the curve to be modified. If the stroke appears to be an overdraw, the system blends it into an existing shadow. If there is no existing shadow with which the new shadow can be merged, we test whether its endpoints lie approximately below some curve's endpoints. If so, we take this to mean than the the curve's shadow was entirely redrawn. (If not, the stroke is rejected.) Finally, we reproject the curve back into the scene to match its new shadow.

6.3.1 Drawing Curves and Shadows

We represent 3D curves as parameterized polylines, i.e., as piecewise linear curves defined by a mapping from $[0, 1] \rightarrow \mathbf{R}^3$. Before they are used to define curves or shadows, input strokes are smoothed in the following way. First we filter the stroke to remove all points whose screen-space distance is less than some threshold (e.g. 25 pixels) from the previous point. We fit a Catmull-Rom spline [20] to the remaining points and sample the spline every few pixels to generate a smooth-looking polyline.[1]

When a curve is first drawn, we project the 2D stroke onto a plane in world space to create a 3D planar curve. We choose the plane as follows. If either endpoint appears to lie on an existing object (or curve), we take this as intentional and place the endpoint in 3D so that it lies on the existing object. We then choose a plane that contains the endpoint (or points). Since one or two points do not uniquely determine a plane, we choose, among all planes containing them, the one that is most nearly screen-parallel.

If neither endpoint appears to lie on an existing object in the scene, we determine which plane to use from the angle of the camera. If the camera is looking down, we use the floor plane, and if the camera is at an oblique angle, we use the plane perpendicular to the floor plane that is most nearly screen-parallel.

A *shadow* is a 3D curve obtained by projecting another 3D curve along a fixed vector, which we call the *projection vector*, onto some surface. In this discussion, we always use the world Y axis as the projection vector, and we always project onto the floor plane. These choices are arbitrary - we could just as easily use the world X vector and let the user draw shadows on a wall. Also, note that in all of our examples, the shadow is a planar curve. This assumption is not necessary for any of the algorithms described below. Thus, we could project shadows onto rolling terrain, for instance.

The key feature of this system is the ability to edit a curve via its shadow. As noted in [34, 86, 87], a point's location is determined uniquely by its appearance from an oblique camera position and by its shadow. We extend this idea to curves: the shape of a 3D curve is determined by its image-space projection and its shadow.[2] Thus, to modify a curve's shape in our system, the user redraws its shadow. This redefines the curve's shape while leaving its appearance from the current camera position unchanged.

It can be difficult to draw a valid shadow for a given curve. To facilitate this, we draw vertical guidelines at both ends of the curve. These lines provide feedback that helps the user align the shadow with the curve. Also, the matching algorithm does not require that the curve and shadow be exactly aligned, only that they be "close," as explained in the next section.

[1]This smoothing step, while independent of the overall technique, is important since noise in the input device propagates to the final 3D curves.

[2]In certain cases described below, the curve's 3D shape is not determined uniquely.

6.3.2 Correlating Curves with Shadows

Once a curve's shadow or image-space projection has been redefined, we project the curve back into the scene using the following method.

We assume either a perspective or orthogonal projection, with the restriction that the camera's "look vector" is not close to parallel with the projection vector. In a perspective projection, the vanishing point for vertical lines must be off the screen. This allows us to define a left-to-right ordering of 3D points (see Figure 6.2).

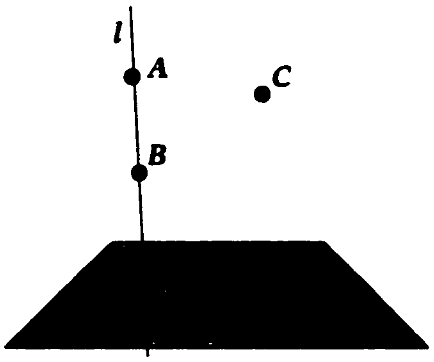


Figure 6.2: A and B are image-space aligned through the line l, and C is image-space right of both of them.

To test if a point A is left or right of another point B, we project A into the image. Then we take the line parallel to the projection vector running through A's world location and project this line into the image. This line (call it l) partitions the image into two sections, one to the left and one to the right. If the image-space projection of B is to the left of l, we say B is *image-space left* of A and similarly for *image-space right.* If B lies on l, we say A and B are *image-space aligned.* If B is within a distance d of l, we say B is *aligned to within d of A.*[3]

We define a point on a curve to be a *critical point* if some neighborhood of the point lies entirely to the right or left of the point in image space. Note that by this definition, the first and last points of a curve are critical points (see Figure 6.3 (a)). A *span* is the section of a curve between two critical points.

The key observation we use to match a curve with its shadow is that the shadow defines a ruled surface formed by extruding the shadow along the projection vector. An interior critical point in the shadow corresponds to a silhouette of the surface, as shown in Figure 6.3 (b).

[3]Note that in a perspective projection, *aligned to within d* is not a symmetric relation.

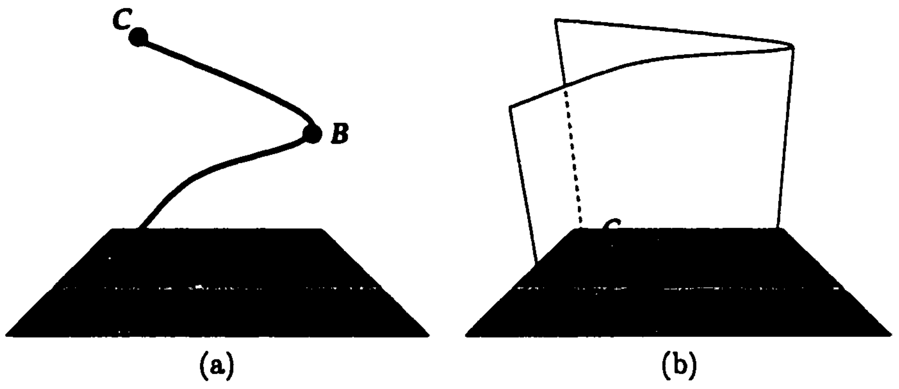


Figure 6.3: (a) A, B, and C are critical points of this curve. (b) The shadow defines a ruled surface with a silhouette above the interior critical point of the shadow, B.

To redefine a curve's 3D shape from its shadow, we must project the curve onto this possibly many-layered surface. To do this, we must determine onto which layer of the shadow surface we should project each point of the curve. In general, a critical point on the shadow must correspond with a critical point on the curve. This is because the curve must turn around where the shadow surface turns around in order to stay on the shadow surface, as shown in Figure 6.4 (a). Note that there may be critical points in the curve that do not correspond to critical points of the shadow, as in Figure 6.4 (b).

If there is no way to project the curve onto the surface so that the resulting 3D curve is continuous, the shadow is invalid and we reject it. Figure 6.5 shows an invalid shadow.

To create the correspondence between layers of the shadow surface and spans of the curve, we need to match critical points of the shadow with critical points of the curve. First, we find the critical points of the shadow and curve. Let $c_0, c_1, \ldots, c_n$ be the list of

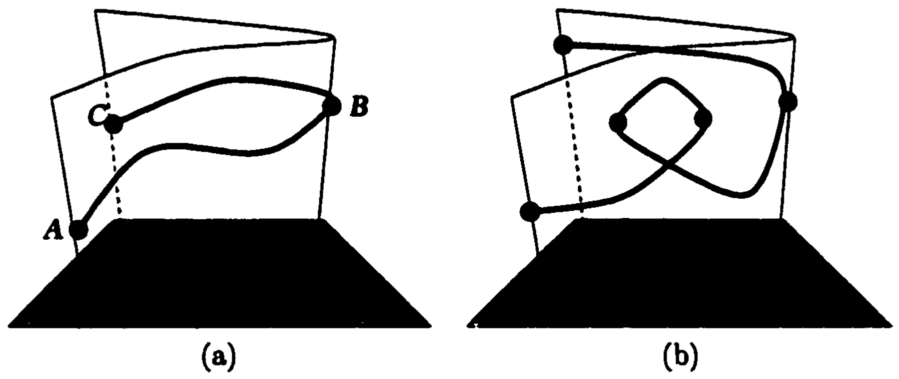


Figure 6.4: (a) The curve must turn around at B to stay on the surface. (b) The curve may have more critical points than the shadow and still be valid.

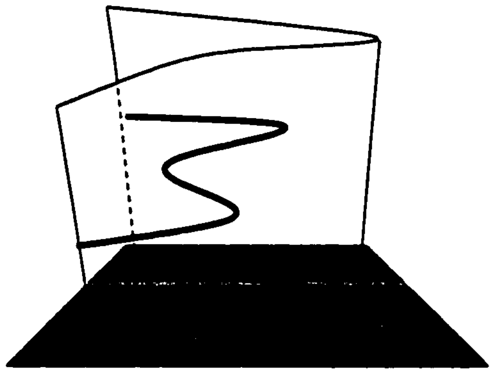


Figure 6.5: There is no way to project this curve continuously onto the shadow surface.

critical points of the curve, sorted by parameter value, and $s_0, s_1, \ldots, s_m$ be the sorted list of the shadow's critical points.

To begin, we verify that c_0 corresponds with s_0. (At this point, we may have to reverse the parameterization of the curve to make these points match.) We iterate through the critical points of the shadow in order and attempt to match each point s_i with some critical point on the curve. We do this as follows.

First, let c_j be the next unmatched critical point on the curve. If c_j is aligned to within a small tolerance of s_i (we use 25 pixels), then c_j is matched with s_i, and we go on to s_{i+1}. Otherwise, we test if c_j is between s_{i-1} and s_i (in terms of the image-space left-to-right ordering). If not, then there is no valid match because there is no shadow underneath some span of the curve adjacent to c_j. Also, if c_j is the final endpoint of the curve and is unmatched, then there is no valid match. Finally, we increment j and repeat for the next critical point on the curve.

The system can match a shadow with a curve even when the shadow does not align exactly with the curve, as follows. After creating a correspondence between critical points, we deform the shadow so that all matching critical points are precisely image-space aligned. This is done by rotating and scaling each span of the shadow to align it with the corresponding span of the curve, as shown in Figure 6.6. This step allows the user to sketch an approximate shadow, leaving it to the system to ensure that curve and shadow are precisely aligned.

We now have a valid aligned shadow and a correspondence between each span of the shadow and some span of the curve. Just as the shadow defines a particular surface, we can think of the curve as defining a unique surface containing all rays extending from the camera through the image-space projection of the curve. We intersect each portion of the

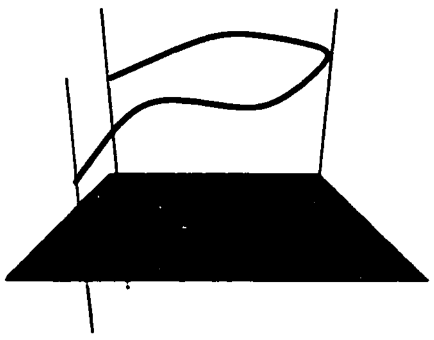


Figure 6.6: The dashed line indicates the original shadow. The solid line indicates the shadow after it has been adjusted to match the curve.

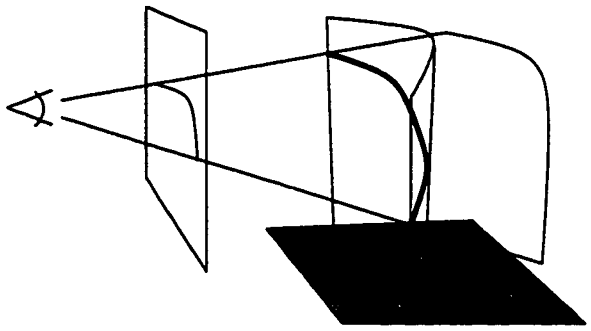


Figure 6.7: The curve defines a surface containing all rays from the viewpoint through each point on the curve. We intersect this with the shadow surface to get the final 3D curve.

curve surface with the corresponding layer of the shadow surface to produce a section of the 3D curve, as shown in Figure 6.7. Because we use a piecewise linear representation for our curves, we intersect these two surfaces by breaking them up into planar segments and intersecting the corresponding segments. We splice all such sections together to get the final 3D curve.

Near a critical point on the shadow, the tangent plane to the shadow surface is oriented nearly edge-on to the camera. This has the effect of magnifying noise in the 2D input: that is, small variations in the input stroke result in large variations in depth for the 3D curve. To alleviate this problem, we remove points that are nearly aligned to critical points of the shadow, replacing them with a smooth spline that joins neighboring sections of the 3D curve. Finally, we perform the same filtering and smoothing operations described above to improve the smoothness of the final curve.[4]

[4]Because we perform these filtering and smoothing steps, the appearance of the curve is not constant – it often changes by a few pixels after each edit.

In certain cases this algorithm may produce an unintended result. This can happen when the curve and shadow have multiple critical points that are image-space aligned. One example is shown in Figure 6.8. In a case such as this, the image-space curve and shadow do not define a unique 3D curve. Our algorithm will find one possible 3D curve, but it might not be the intended one.

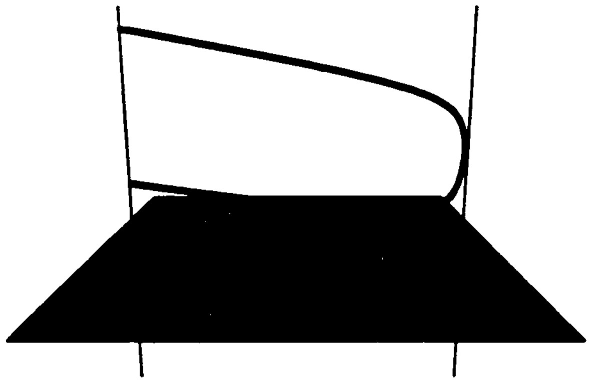


Figure 6.8: The shadow and curve do not determine a unique 3D curve. This is because all four endpoints are image-space aligned.

6.4 Discussion

This system is well suited for applications that require fast specification of approximate 3D curves. Applications that require more precise curves might still benefit from this technique, because of the lack of overhead required and the simplicity of the interface. In one scenario, the user would quickly sketch an approximate curve, then refine its shape with more conventional techniques.

We mentioned previously that our technique can be extended to allow shadows on walls or nonplanar surfaces. There is also no reason to restrict the user to drawing the curve and shadow from the same point of view. The user might draw the shadow from an overhead camera position (thus specifying the shadow more accurately), then sketch the curve from an oblique viewpoint.

A limitation of this method is that it can be quite hard to judge what the shadow should look like for complex 3D curves, especially from an oblique viewpoint. We have observed that users in our lab, even those with artistic training, have considerable difficulty drawing corkscrews and other spiraling shapes. In such cases, a better solution might be to use a 3D input device such as a Phantom or a 3D tracker.

6.5 Future work

We use context-sensitive commands to indicate over-sketching operations and keyboard modifiers to indicate modes and to differentiate between different editing operations. Although this works, it is neither consistent nor supported by user studies. We would like to find a more streamlined user interface, perhaps using marking menus or gestural commands [63]. We would also like to have more users try this system, especially users with artistic training but little experience with computer graphics tools.

Finally, we have started to use this curve-sketching technique within a sketch-based free-form modeling system. We believe this interface is a good starting point from which to build a modeling system that leverages a user's talent with pencil and paper to create more complicated shapes than was possible with the original SKETCH system [87].

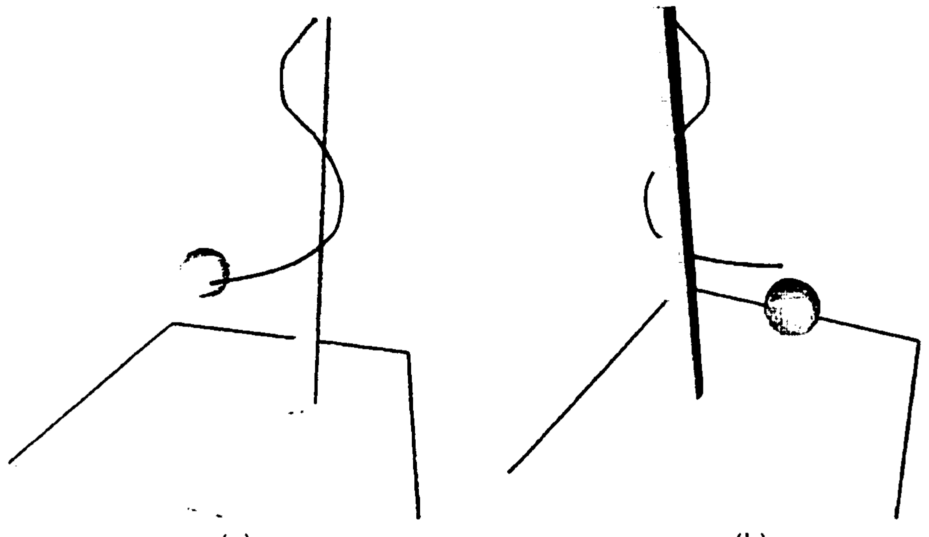

(a) (b)

Figure 6.9: A tetherball rope before its shadow has been oversketched. The default placement is not what's desired. (a) Shows the original view; (b) shows another view.

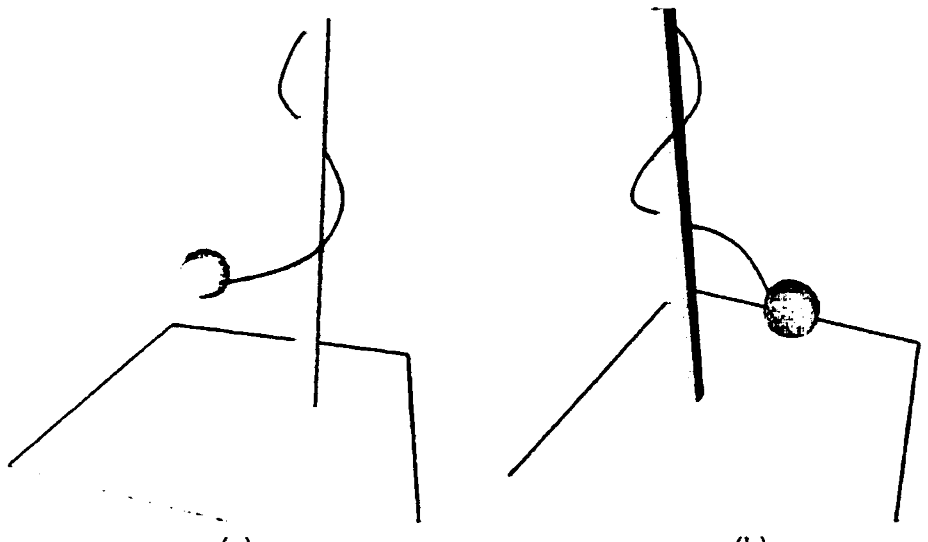

(a) (b)

Figure 6.10: Two views of the tetherball rope from Figure 6.9, now with its shadow oversketched.

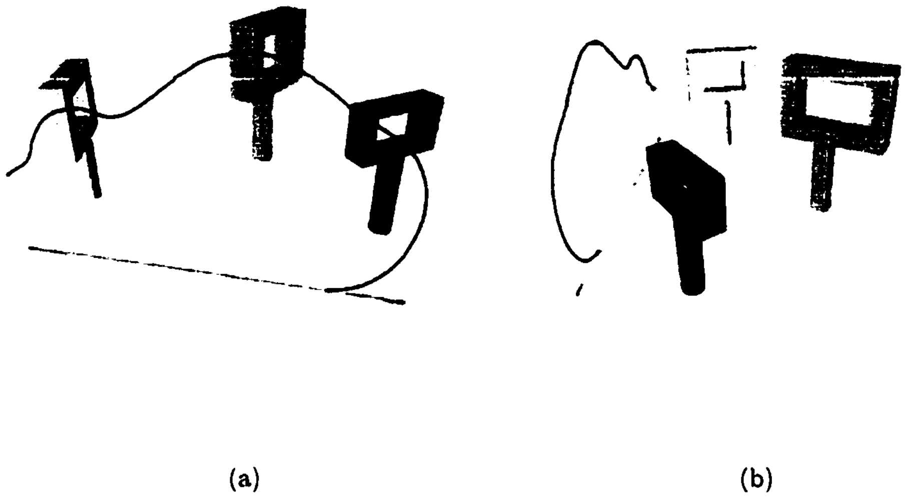


Figure 6.11: The user seeks to thread a curve through some obstacles. Without oversketching the shadow, the default placement of the curve is wrong. (a) Original view; (b) alternate view.

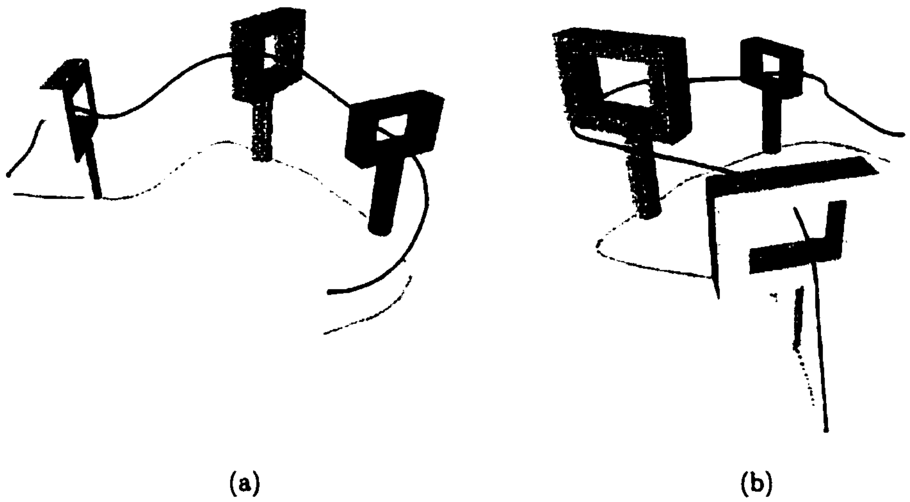


Figure 6.12: After oversketching the shadow, the curve passes through the obstacles. Same views as in Figure 6.11.

Chapter 7

"Skin" – a new surface representation

7.1 Introduction

The algorithm described in this chapter derives from our ongoing effort to build a free-form modeling system with a direct interface – that is, one in which the user defines shapes by sketching them directly as they would appear from a given viewpoint. In particular, we seek to extend the principles developed in the SKETCH system [87] – rapid construction of approximate shapes via direct interaction – to the problem of free-form modeling. The Teddy system [43] targets similar goals, though we seek to provide more fine-grained control over the resulting shape, including the ability to add surface detail at multiple scales. An important goal is that the interface should allow a user with skill at drawing on paper to apply this skill to model compelling 3D forms.

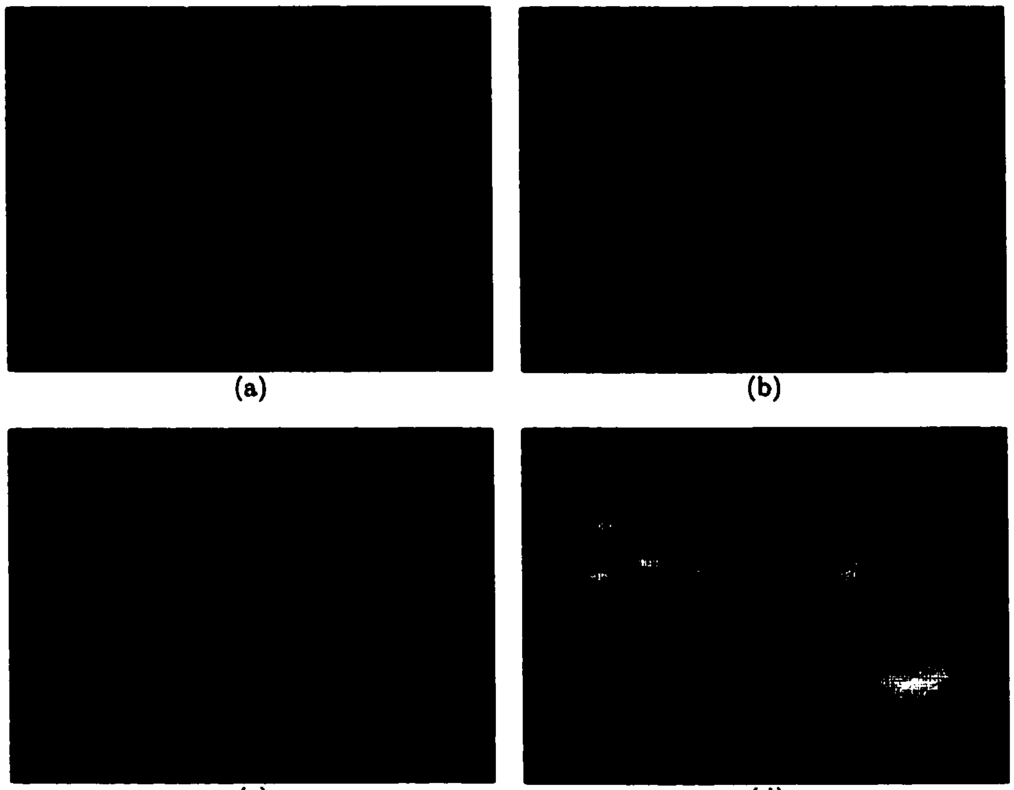
With that in mind, we seek to build a system in which the user models complex surfaces by first constructing simplified representations of the underlying masses that give them their form, then "oversketches" these masses with a smooth surface that approximates their collective shape. We chose this approach because of its tractability (SKETCH and Teddy, as well as our system for sketching 3D curves [15], describe techniques that could be applied to define the "underlying masses"), and because it is natural – that is, it resembles techniques commonly used by artists and recommended in books on art instruction [1, 13, 28, 35, 59, 67, 78].

This chapter describes our solution to a sub-problem that arises in implementing our approach: given a collection of "masses" (or *skeletons*) represented as polygon meshes, each with an associated offset (typically small), construct a smooth surface that approximates the distance surface defined by the skeletons and their offsets. Distance surfaces, discussed in [10] and defined in section 7.3 below, typically have sharp creases that are not desired. Thus the final surface should smoothly approximate the distance surface, not fit it exactly. The solution should work in an interactive setting, continually updating the surface as skeletons are added, manipulated, or removed.

Our solution, the *skin* algorithm, works by constructing a subdivision surface control mesh that roughly fits the shape of the distance surface. The control mesh in turn defines a smooth surface with a similar shape. A key contribution of the present work is to provide an algorithm for generating a *suitable* control mesh for use in subdivision. Simply matching a given shape is not sufficient – the triangulation of the control mesh has a strong influence on the qualities of the resulting surface. Smaller triangles provide less "smoothing" effect, yielding a surface that more closely fits the control mesh, and – more importantly – irregular triangulations can greatly reduce the fairness of the limit surface. This can be seen in Figure 7.1.

The skin algorithm is good for *rapid* construction of *approximate* free-form shapes. It is not suitable for applications that require exact control of the final shape. Skin supports fast, intuitive editing of features at multiple resolutions and produces piecewise smooth surfaces with creases or sharp corners where desired. Since the algorithm manipulates a polygon mesh, rendering can be done efficiently on common graphics hardware.

Together with these advantages, the skin algorithm has some drawbacks: it is based on an iterative algorithm that is not guaranteed to converge to a fixed triangulation – and sometimes doesn't. The exact results depend not just on the primitives used to define the surface, but also on the order of the user's operations. Finally, the behavior of the skin depends on the size of its triangles (which the user can control). It would be preferable not to expose the user to the issue of triangle size as a means of controlling surface shape. The other difficulties can be alleviated by providing tools to let the user control the skin's behavior, as we will illustrate. Because we designed skin for use in an interactive setting, we do not consider any of these problems to be severe.



(a) (b) (c) (d)

Figure 7.1: Triangulation matters. Two triangulations of the same shape ((a) and (c)) produce different results when subdivided ((b) and (d)). The skin algorithm produces even triangulations, avoiding skinny triangles like those in (a) that lead to unexpected wiggles and bumps when subdivided.

7.2 Related work

The skin algorithm resembles the methods for iteratively evolving a triangulation described in [36, 37, 72, 79, 81]. It has much in common with Miller *et al.*'s "volumetrically deformed models" [53] in which a balloon-like surface is inflated under constraints to match a volumetrically defined implicit function. Also, because the skin algorithm treats vertices as particles that interact with their neighbors while being guided by an implicit function, it resembles the particle system methods described in [77, 85]. (An important difference is that skin particles have explicit connectivity information.) Skin is related to the large body of work on implicit surfaces, particularly offset and convolution surfaces [10, 62] and blobby modeling [9, 11], in the constructive approach to modeling it provides.

For multiresolution editing, we use Loop subdivision meshes [47], including the rules for defining creases and corners described in [36, 69]. Previous work on subdivision surfaces has primarily focused on specifying subdivision rules and analyzing properties of the limit surface, *given a control mesh.* Hoppe *et al.* [36] present a means to construct a good-quality control mesh for a subdivision surface that approximates a given polygon mesh (typically scanned from a real model). We are not aware of any work that discusses ways to construct suitable control meshes for subdivision surfaces *from scratch.* Recently Zorin *et al.* [89] and Pulli [58] presented systems that let the user add detail to a subdivision surface (given the control mesh). With these systems the user edits details of the subdivision surface (at a given resolution) by manipulating individual vertices directly. The skin algorithm, though originally designed to construct a good-quality control mesh in the first place, is readily extended to provide a mechanism for adding detail at finer scales more conveniently than by individual control point manipulations.

7.3 Terms and definitions

Our meshes are triangle meshes, represented in the usual way: each consists of a collection of vertices, edges and faces with local connectivity information. That is, each vertex stores pointers to its adjacent edges; each edge stores pointers to the two vertices that define it and to at most two adjacent faces; each face stores pointers to the three edges and vertices that define it. *Skeletons*, represented with this mesh data structure, may be non-self-intersecting closed surfaces, surfaces with boundary, polylines or even isolated points. With each skeleton is associated a positive real-valued *offset.* The skin algorithm iteratively modifies the *skin* mesh (a closed surface) to roughly fit the distance surface defined by the skeletons and their offsets. Each skeleton also has an associated *target length* that specifies the desired size of triangles the skin should generate when growing over that skeleton. We refer to vertices of the skin mesh as *particles*. It is convenient to associate with each particle a *reference length* that measures the scale of the skin mesh near that particle. We take the reference length to be the average length of the particle's adjacent edges.

Let S denote the set of skeletons. For a given skeleton $s \in S$ with associated offset r_s, and a point $\mathbf{x}$ in space, let $d_s(\mathbf{x})$ denote the signed distance from $\mathbf{x}$ to s. That is, $d_s(\mathbf{x})$ is the distance from $\mathbf{x}$ to the nearest point on s, but if s is a closed surface and $\mathbf{x}$ is inside it, the "distance" is taken to be negative. Now let $f_s(\mathbf{x}) = d_s(\mathbf{x}) - r_s$. The *distance surface* defined by a single skeleton and its offset is just the implicit surface defined by $f_s(\mathbf{x}) = 0$. The distance surface defined by the collection of skeletons and their offsets is the implicit surface defined by $F(\mathbf{x}) = 0$, where F is the continuous function given by:

$$F(\mathbf{x}) = \min\{f_s(\mathbf{x}) : s \in S\}.$$

The rules for positioning particles described in section 7.4.1 are designed so that the skin approximates this surface. They also allow us to evaluate the implicit function efficiently.

7.4 The skin algorithm

At a high level, the skin algorithm closely resembles algorithms described in [37] and [81] for iteratively evolving a triangle mesh. The primary difference is in the implementation of step (1) of the main loop:

```
repeat
    (1) reposition particles
    (2) modify the skin's connectivity
until no changes occur
```

Intuitively, in step (1) particles move toward the implicit surface while tending to distribute themselves more evenly and smoothing out wrinkles and bumps in the skin. In step (2), the connectivity of the skin is modified to produce triangles that are more nearly equilateral and whose size is roughly the target length specified by nearby skeletons. We now describe steps (1) and (2) in detail.

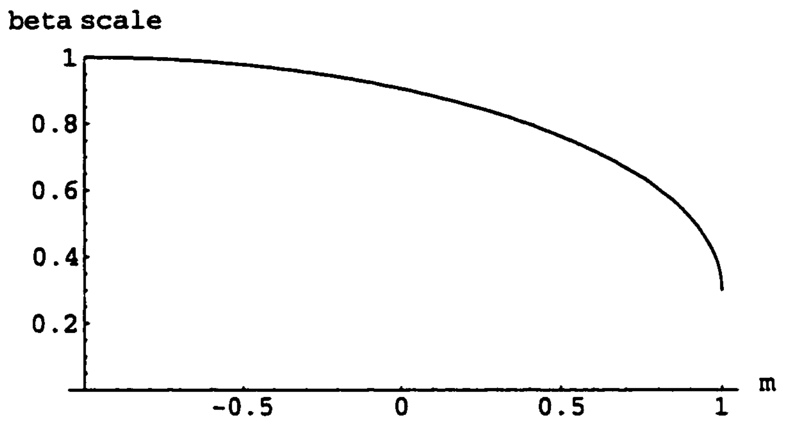


Figure 7.2: The coefficient β_{scale} chosen as a function of the minimum dot product m. This function is a quarter ellipse.

7.4.1 Repositioning particles

The rules for repositioning particles are designed to allow the skin to grow adaptively to conform to the changing implicit surface as skeletons are added, repositioned, removed, or their offsets are edited. In step (1) of the skin algorithm, each particle is visited in turn. The new position $\mathbf{x}'_p$ of particle p is computed as a weighted sum of its current position $\mathbf{x}_p$, the centroid $\mathbf{c}_p$ of its neighbors' positions, and a target position $\mathbf{t}_p$ chosen to bring the particle closer to the implicit surface:

$$\mathbf{x}'_p = \alpha \mathbf{x}_p + \beta \mathbf{c}_p + \gamma \mathbf{t}_p.$$

The weights α, β, and γ are non-negative and sum to one. We always take $\alpha = 0.3$, and choose β according to how smooth the skin is near p. (Then γ is taken to be $1 - \alpha - \beta$.) We estimate the smoothness of the skin near p by calculating the minimum, over each edge adjacent to p, of the dot product of the normals of the two faces adjacent to that edge. This value (call it m) lies between 1 and -1. When m is close to 1, the skin is smooth near p; when m is close to -1, the skin contains one or more sharp edges near p. A large value for β results in a greater smoothing effect, so we choose $\beta = (1 - \alpha)\beta_{scale}$, where β_{scale} is the function of m shown in Figure 7.2. This function, chosen heuristically, seems to produce good results. It has the effect of letting the skin move briskly toward the implicit surface where the skin is reasonably smooth. Where the skin is bumpy or uneven, it slows or stops moving until it smooths out again, then resumes its motion toward the implicit surface.

The target position $\mathbf{t}_p$ is chosen so that the skin tends to expand when inside the implicit surface, stay fixed when on it, and contract when outside it. We can determine whether the particle is inside, on, or outside the implicit surface according to whether $F(\mathbf{x}_p)$ is negative, zero, or positive, respectively. Also, by definition, the magnitude of this value specifies a lower bound on the distance from the particle to the nearest point on the implicit surface. We thus set the target position to be a displacement along the particle's surface normal $\vec{\mathbf{n}}_p$ by an amount d determined by $F(\mathbf{x}_p)$:

$$\mathbf{t}_p = \mathbf{x}_p + d\vec{\mathbf{n}}_p.$$

We can take $d = -F(\mathbf{x}_p)$ unless this value is large (that is, the particle is far from the implicit surface). To prevent the particle from taking excessively large steps, we restrict the magnitude of d to be no more than the reference length of the particle.

Because β is always nonzero, skin particles tend not to arrive exactly at the implicit surface. (Where it is concave they lie outside it, and where it is convex they lie inside it.) As mentioned earlier, we prefer this behavior, since the distance surface defined by the skeletons typically has unwanted sharp creases. For this reason, we can think of the implicit function as guiding rather than defining the skin's final shape.

7.4.2 Evaluating the implicit function

The skin algorithm is intended to work with skeletons that may be finely tessellated (as when representing muscle masses, such as those in Figure 7.16). For the algorithm to be usable in an interactive setting, it is important to evaluate the implicit function efficiently. The particles do this by exploiting locality in two ways. To explain this, let's first assume there is only a single skeleton, s. Each particle p tracks a point $\mathbf{y}_p$ on s that is locally closest to p.[1] We refer to $\mathbf{y}_p$ as p's *track point*.[2] After updating its position $\mathbf{x}_p$, p updates its track point using the LOCAL-SEARCH procedure (Figure 7.3), passing in a face f containing the track point:

$$\mathbf{y}_p \leftarrow \text{LOCAL-SEARCH}(f, \mathbf{x}_p).$$

Evaluating $f_s(\mathbf{x}_p)$ is now simple: For a non-closed skeleton surface, we evaluate the distance d from $\mathbf{x}_p$ to $\mathbf{y}_p$, and subtract r_s. For a closed surface, we negate d if the particle is inside it, and then subtract r_s. (Each skeleton surface is checked in a pre-process step to determine whether it is closed.) For closed surfaces, we can determine when a particle is inside provided that the particle has actually tracked the globally closest point $\mathbf{y}_p$ on

[1]By locally closest, we mean that all points on s in a neighborhood of $\mathbf{y}_p$ are farther away.

[2]In fact, p stores a pointer to a face of s containing $\mathbf{y}_p$.

```
LOCAL-SEARCH($f$, $\mathbf{x}$)
    $\mathbf{y} \leftarrow$ closest point to $\mathbf{x}$ on $f$
    if $\mathbf{y}$ is on boundary of $f$
        foreach adjacent face $f_i$ containing $\mathbf{y}$
            if closest point to $\mathbf{x}$ on $f_i \neq \mathbf{y}$
                return LOCAL-SEARCH($f_i$, $\mathbf{x}$)
    return $\mathbf{y}$
```

Figure 7.3: The LOCAL-SEARCH algorithm, starting from face f on a mesh M, traverses faces of M, always moving closer to point $\mathbf{x}$, to reach a point $\mathbf{y}$ on M that is locally closest to $\mathbf{x}$.

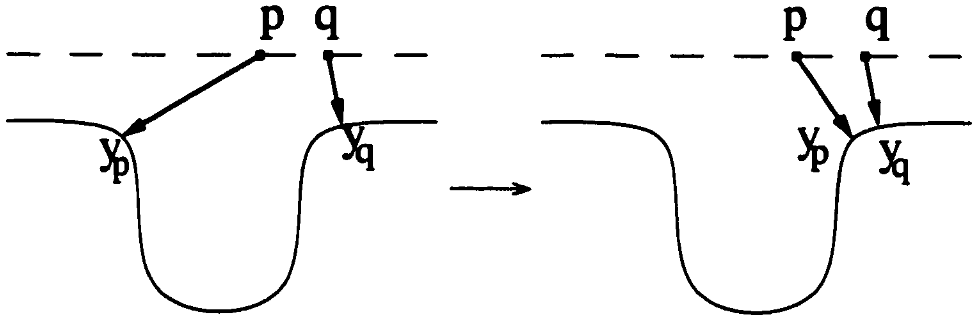


Figure 7.4: A particle p can use its neighbor's track point $\mathbf{y}_q$ to get out of a local minimum.

the skeleton. For then the plane containing $\mathbf{y}_p$ that is perpendicular to the vector from $\mathbf{y}_p$ to $\mathbf{x}_p$ divides space into two parts. The skeleton locally near $\mathbf{y}_p$ lies all in one part and the particle lies in the other. An examination of the face normals around $\mathbf{y}_p$ determines whether p is inside or outside the surface.

Of course, the LOCAL-SEARCH procedure may return a point on the skeleton that is only *locally* closest to p (see Figure 7.4). To overcome this, particles exploit a second kind of locality: they share information with their immediate neighbors. Thus, to evaluate $f_s(\mathbf{x}_p)$, p runs the LOCAL-SEARCH procedure starting from its own track point, and then again starting from the track points of each of its neighbors in turn. The particle selects the result yielding the smallest distance, and stores this as its new track point.

When there are multiple skeletons, p can switch to a track point on a different skeleton if doing so yields a smaller *value* for the implicit function. It can switch to a different track point on the same skeleton, though, only if doing so yields a smaller *distance*. The distinction between these cases occurs only when the skeleton is a closed surface. In this case, the particle should first find the closest point on the skeleton, then evaluate the implicit

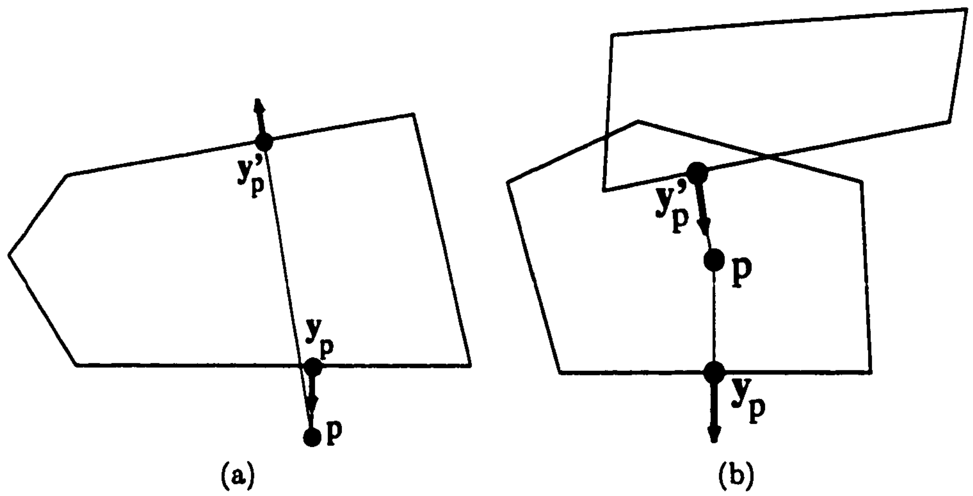


Figure 7.5: In (a), p is deciding between tracking $\mathbf{y}_p$ or $\mathbf{y}'_p$. Since both points lie on the same skeleton, it will choose $\mathbf{y}_p$. Note that the surface normal at $\mathbf{y}'_p$ is facing away from p, so tracking $\mathbf{y}'_p$ would yield a negative value for the implicit function, while $\mathbf{y}_p$ would yield a positive value. In (b), $\mathbf{y}'_p$ is closer. However, $\mathbf{y}_p$ is on a different skeleton, and will yield a lower implicit function value because its surface normal is facing away from p. p should therefore choose to track $\mathbf{y}_p$.

function from the signed distance to this point. Figure 7.5 shows examples of these two cases.

When a new skeleton is added, it is sufficient for a single particle (lying inside the skeleton's distance surface) to begin tracking its closest point on the skeleton. That particle will then recruit its neighbors to track points on the new skeleton, and in the next iteration they will recruit their neighbors, and so on.

It sometimes happens that a particle has no track point: for example, when a skeleton is removed, particles tracking points on it forget their track points. In this case particles simply take $\beta = 1 - \alpha$, which causes them to move towards the centroids of their neighbors. The region of skin that covered that skeleton will wither away, until all of its particles are either destroyed in edge collapses (described in the next section) or acquire new track points on other skeletons from their neighbors.

There are two other cases in which particles forget their track points. If a particle is outside of the implicit surface but its surface normal points toward its track point, the particle would, according to the rules described above, tend to back away from the track point, moving farther and farther from the implicit surface (see Figure 7.6). To prevent this, particles in this case simply forget their track points and take $\beta = 1 - \alpha$ as above. The

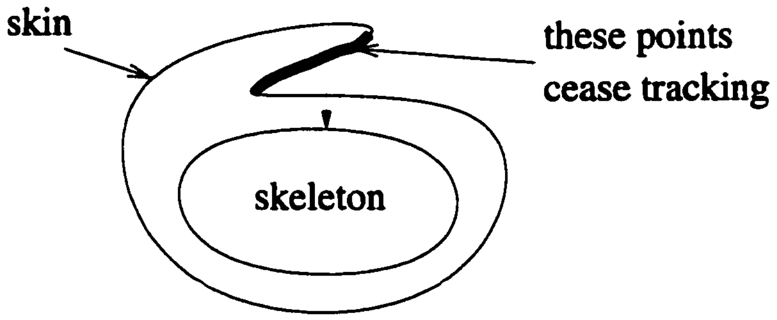


Figure 7.6: When the skin folds over so that an exterior particle's normal points towards the skeleton, the particle ignores the track point to avoid "backing away."

second case occurs when a particle is adjacent to a *stressed* edge – that is, an edge whose two adjacent faces form an angle of less than 60 degrees. We discuss this case further in section 7.5.

7.4.3 Modifying the mesh connectivity

In step (2) of the skin algorithm we modify the skin's connectivity by performing the edge operations shown in Figure 7.7. These operations were described in [37, 81]; conditions under which the operations are valid are discussed in [37].

We swap an edge when doing so increases the minimum angle within its adjacent faces. As noted by Welch and Witkin [81], repeated application of this swap operation (always increasing the minimum angle) computes a constrained Delaunay triangulation. That is, it maximizes the minimum angle over all the triangles of the mesh.

Each edge has an associated target length. We split an edge if it is longer than 1.5 times its target length, and collapse the edge if it is less than half its target length. These numbers are chosen in part so that a collapse operation will not immediately be invoked on an edge that has just been split. We compute the target length for each edge as follows.

As mentioned in section 7.3, each skeleton has an associated target length that the user can set. This specifies the desired size of triangles for portions of skin growing over that skeleton. We also store a target length value in each particle. In the repositioning step, each particle sets its target length to a weighted sum of its current value, the average target length of the particle's neighbors, and the target length of the skeleton currently tracked by the particle. (We use weights of 0.4, 0.4 and 0.2, respectively.) The target length of an edge is taken to be the average of the target lengths of its two adjacent particles.

Splitting and collapsing edges as described tends to produce edges of approximately

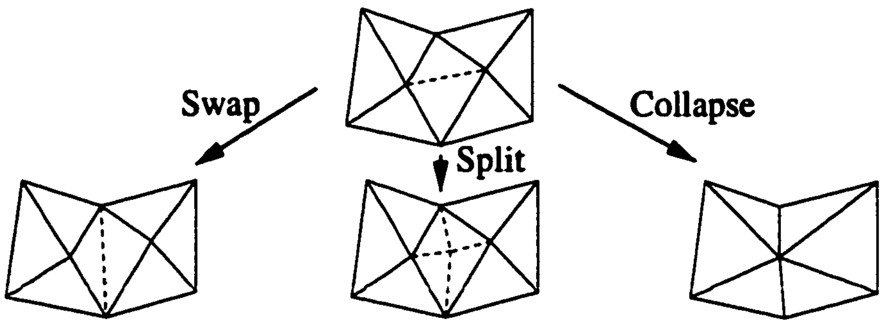


Figure 7.7: We *swap* to improve the minimum angle of a triangle. A *split* introduces a new vertex by splitting a long edge. A *collapse* removes a vertex by destroying a short edge.

the desired length. Interleaving steps (1) and (2) of the skin algorithm further tends to equalize edge lengths, allowing particles to redistribute themselves more evenly after edge operations are performed. In fact, we control the number of split and collapse operations that can occur between repositioning steps by sorting edges by the ratio of their actual length to their target length, then considering split operations just on the last 5% of edges and collapse operations just on the first 5% of edges. Any edge that is too long (or too short) is still guaranteed to be split (or collapsed) eventually.

Each edge operation affects the valence of nearby vertices. Since Loop subdivision performs best on meshes with a majority of valence-six vertices, we modify the rules slightly to favor the formation of such vertices. For example, a proposed edge swap is evaluated according to: (1) how much it will increase or decrease the minimum angle interior to its adjacent faces, and (2) the extent to which it will increase (or decrease) the number of valence-six vertices around it. The other operations are "handicapped" in a similar way. Such handicapping yields a small but noticeable improvement in the quality of the resulting subdivision surfaces.

7.4.4 Termination

In the repositioning step, we actually assign a particle its new position only if it is appreciably different from the current position. Specifically, the new position must differ from the current one by more than .01 times the particle's reference length. This prevents the skin from continuing to make visually unnoticeable adjustments to the particle's positions after the surface is essentially stable.

Still, we have observed two cases in which the algorithm does not terminate. The first can occur when the topology of the skeletons' distance surface differs from that of the skin.

The rules for modifying the skin's connectivity do not change its surface topology. If the user grows a skin over two skeletons that are initially close together, and then pulls them far apart, the skin does not spontaneously separate into two pieces. (In section 7.7 we describe how the user can explicitly invoke this type of topological change.) Instead, it remains joined by a thin strand, along which particles continue to move as edges are repeatedly split and collapsed. If the strand is cut, the skin heals itself and stabilizes.

The other case can occur when adjacent regions of skin have very different target lengths. In this case particles may continuously "swim" out from the region with small triangles toward the one with larger ones. This happens when a particle between the regions drifts toward the centroid of its neighbors in the region with larger triangles. As this happens, the particle's shorter edges are stretched, leading them to split and create new particles that repeat the process. We do not currently provide a solution to this, other than letting the user "freeze" the skin in that region.

Another problem involves premature termination. This can occur when the target length associated with a skeleton is large relative to one of its dimensions. In that case, the region of skin attempting to grow over the skeleton may necessarily have sharp edges. The tendency for particles to move to their neighbors' centroids may then outweigh their tendency to move toward the implicit surface, and the skin stops. An effective remedy is for the user to reduce the target length associated with the skeleton.

7.4.5 Analysis of convergence

The skin algorithm was designed to produce a surface that approximates, but smooths out, the collective shape of a set of polyhedral "skeletons." We observe that in practice it successfully performs this task. But one might reasonably ask, "What does the skin algorithm actually compute? Does it converge to some well-defined surface that we can characterize?" The answer is that, under certain assumptions, it does.

Recall that at each step of the iteration we compute the new location $\mathbf{x}_{k+1}$ of a vertex as a weighted sum of its current location $\mathbf{x}_k$, the centroid $\mathbf{c}_k$ of its neighbors, and a computed "target position" $\mathbf{t}_k$:

$$\mathbf{x}_{k+1} = \alpha\mathbf{x}_k + \beta\mathbf{c}_k + \gamma\mathbf{t}_k,$$

where $\alpha+\beta+\gamma = 1$. In what follows we will assume that α, β and γ are positive constants.

Suppose that at iteration k, the mesh has n vertices, and let X_k, C_k and T_k denote the $(n \times 3)$ column matrices of vertex positions, centroids and target positions, respectively.

By definition, we must have $C_k = CX_k$, where C is a stochastic matrix. That is, C is nonnegative and its rows sum to 1. For information on stochastic matrices, see for example the book by Graham [29].

The skin algorithm can now be expressed as the iteration:

$$X_{k+1} = \alpha X_k + \beta C X_k + \gamma T_k.$$

We can rewrite the first two terms:

$$\begin{aligned} \alpha X_k + \beta C X_k &= (\alpha I + \beta C) X_k \\ &= (\alpha + \beta) \left(\frac{\alpha}{\alpha+\beta} I + \frac{\beta}{\alpha+\beta} C \right) X_k \\ &= (\alpha+\beta) S X_k, \end{aligned}$$

where S is a convex combination of the stochastic matrices I (the identity) and C, and so is itself stochastic. Using $\alpha + \beta = 1 - \gamma$, the skin algorithm can now be expressed as the iteration:

$$X_{k+1} = (1-\gamma) S X_k + \gamma T_k, \tag{7.1}$$

Note that the dimensions of all matrices and the coefficients of S depend on the mesh connectivity. T_k depends on the vertex positions and the skeletons, and on whether the skin particles correctly maintain track points on the skeletons. Because both S and T_k vary, this iteration is complicated to analyze. To make the analysis more tractable, we introduce two assumptions: that the mesh connectivity is constant, and that the sequence T_k is constant.

These assumptions are certainly convenient, but they are also reasonable: the retriangulation operations stop (usually) once the skin approaches its final shape; and in any case the user can always suspend retriangulation explicitly. Also, we have observed that we can modify the repositioning rules to cause skin particles to interpolate the iso-surface associated with the skeletons.[3] Let's say we are given a fixed set of skeletons and offsets, and that we run the modified skin algorithm to yield an evenly triangulated mesh with vertex positions X_0 that interpolate the iso-surface. If we now suspend the retriangulation operations and run the regular skin algorithm (Equation 7.1), we will initially have $T_0 = X_0$, because X_0 interpolates the iso-surface.

[3] To do this, we ensure that the displacement toward the target position along the particle's normal is at least a bit larger than the normal component of the displacement to the centroid – if the displacements point in opposite directions. In that case the particle will always make some minimal progress in moving toward the iso-surface.

If we now assume that the displacement due to smoothing will lie largely along the direction of the surface normal at each vertex (reasonable since the initial mesh is evenly triangulated), and that the deformation due to smoothing does not perturb the surface normals (it will, but not much), then the computed target positions at each iteration will remain equal to X_0. The iteration thus becomes:

$$X_{k+1} = (1-\gamma)SX_k + \gamma X_0.$$

Because S is stochastic, its spectral radius is 1. Thus the matrix

$$R = (1-\gamma)S$$

has spectral radius $\rho(R) = 1-\gamma < 1$. Let $T = \gamma X_0$. (The following conclusions still hold if T is chosen to be any constant $(n \times 3)$ matrix.) The iteration is finally expressed as:

$$X_{k+1} = RX_k + T, \tag{7.2}$$

We know that a limit of this sequence must satisfy the equation:

$$X = RX + T,$$

or $(I-R)X = T$. Because $\rho(R) < 1$, $(I-R)$ must be nonsingular. (Otherwise $\lambda = 1$ would be an eigenvalue of R, contradicting $\rho(R) < 1$.) The equation thus has a unique solution: $X = (I-R)^{-1}T$.

Proposition 2 *The iteration defined in Equation 7.2 converges to $X = (I-R)^{-1}T$, regardless of the choice of X_0.*

Proof (Adopted from Golub and Van Loan [26].)
Let $E_k = X_k - X$ denote the error at step k of the iteration. Then:

$$\begin{aligned} E_{k+1} &= X_{k+1} - X \\ &= RX_k + T - (RX+T) \\ &= R(X_k - X) \\ &= RE_k. \end{aligned}$$

By induction, $E_k = R^k E_0$.
It follows that $E_k \rightarrow \mathbf{0}$, since $\|E_k\| = \|R^k E_0\| \leq (1-\gamma)^k \|E_0\|$. □

Figure 7.8: Indentations were created in the foot model using crease curves.

7.4.6 Geometric constraints – creases

The user might want to constrain the skin's triangulation to include a sequence of edges aligned along some curve. Such an ability is required to model piecewise-smooth surfaces that include sharp "creases." Many organic forms, including human figures, can be described as having such features. To support this, we let the user first create a curve interactively by drawing onto the skin surface. We then split the triangles of the skin through which the curve passes to produce a sequence of edges aligned along this curve. Finally, the skin retriangulates in the neighborhood of the curve to maintain a good triangulation subject to the constraint that edges remain aligned along the curve. The user can then interact with the curve to change the shape of the surface. The foot model in Figure 7.8 was created by drawing curves to model the creases between the toes, and then interactively pushing each curve into the foot to create the indentations. In this way, the user can create features that are not present in the implicit surface. Because the rules for repositioning vertices do not force them to interpolate the implicit surface, the region of skin around the crease blends smoothly to meet it.

For particles and edges lying on a crease, we modify the skin algorithm as follows. First, edges lying along a crease are not allowed to swap. Second, a particle p lying on a crease curve C is constrained to stay on C. If p lies on an endpoint of C, its position is constrained to remain at that endpoint. Otherwise, it computes its target position $\mathbf{t}_p$ by taking a convex combination of its current position $\mathbf{x}_p$ and the midpoint $\mathbf{c}_p$ between its two neighboring particles on the crease curve. Then, instead of moving under the influence of the implicit function, p sets its new position $\mathbf{x}'_p$ to be the point on the curve that is closest to $\mathbf{t}_p$:

$$\mathbf{x}'_p = \underset{\mathbf{x} \in C}{\operatorname{argmin}} |\mathbf{t}_p - \mathbf{x}|.$$

We use a global search to compute this point, but a number of other techniques, such as gradient descent, could be applied depending on the underlying curve representation.

7.5 Preventing self-intersection

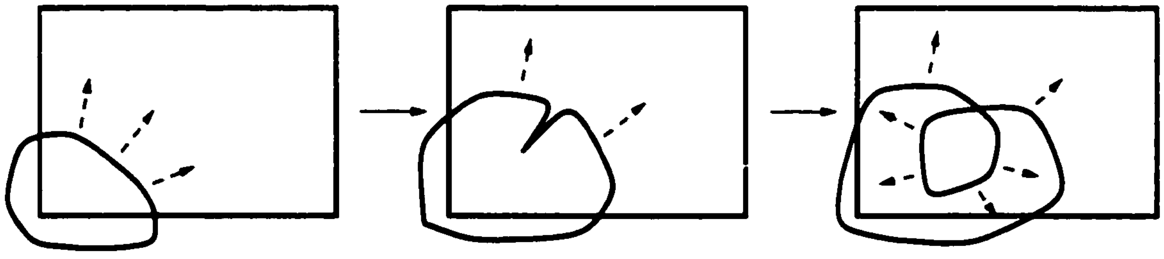


Figure 7.9: Uneven expansion of a skin can lead to self-intersection.

One problem with the skin algorithm as stated is shown in Figure 7.9. During rapid expansion, one region of the skin may grow at a different rate from a neighboring region. The skin begins to buckle, and soon two adjacent regions of skin interpenetrate, forming a "bubble" that expands to produce a second, unwanted layer of skin.

In this section we discuss how we can prevent such self-intersection efficiently by reusing the particles' abilities to track a locally closest point on some mesh, and by sharing information as in section 7.4.2. A key observation is that local interpenetration typically occurs between two portions of surface separated by a chain of stressed edges. Our strategy is to initiate a non-penetration behavior only as needed, starting at stressed edges and propagating along opposing surface regions. We illustrate the idea in Figure 7.10.

In Figure 7.10(a), we see the skin surface, in cross section, starting to buckle along the stressed edge s. Particle a, which shares a face with s, receives notification from s to begin

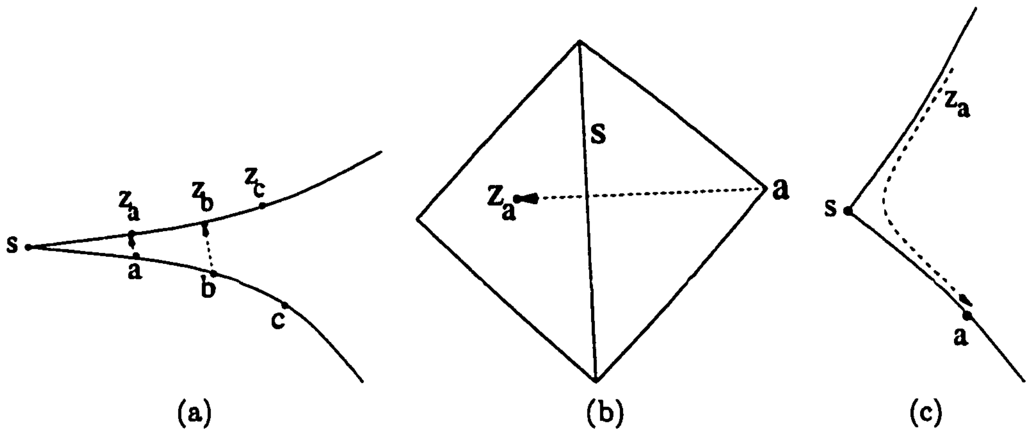


Figure 7.10: In (a), a and b track their penetration points $\mathbf{z}_a$ and $\mathbf{z}_b$. c is too far away from $\mathbf{z}_c$, so it has no penetration point. In (b), the stressed edge s tells adjacent particle a to track $\mathbf{z}_a$. In (c), the surface has smoothed out, so a's LOCAL-SEARCH returns to a.

tracking its closest point $\mathbf{z}_a$ on the opposite face, shown in Figure 7.10(b). We call $\mathbf{z}_a$ the *penetration point* of a.

As particles are repositioned, each updates its penetration point using the LOCAL-SEARCH procedure starting from its previous penetration point, as well as each of its neighbors' penetration points, if they have them. If a local search starting from any of these points returns a point that is not the particle's location, but is at a distance of less than 1.5 times the particle's reference length, the particle stores this point as its new penetration point. Particles are in this way always attempting to borrow penetration points from their neighbors.

Thus, b acquires a penetration point $\mathbf{z}_b$ by using the LOCAL-SEARCH starting from $\mathbf{z}_a$. When c does the same – borrowing from b – the result $\mathbf{z}_c$ is too distant (more than 1.5 times c's reference length) so c "forgets" the penetration point.

Whenever a particle with a valid penetration point passes through the surface containing its penetration point, it suspends the normal repositioning rules and moves back to the correct side of the surface. The result is that a crease temporarily forms in the skin but then resolves itself (Figure 7.11). As the surface flattens out, particles track their penetration points back to themselves (Figure 7.10(c)), upon which they forget their penetration point and revert to the normal repositioning rules.

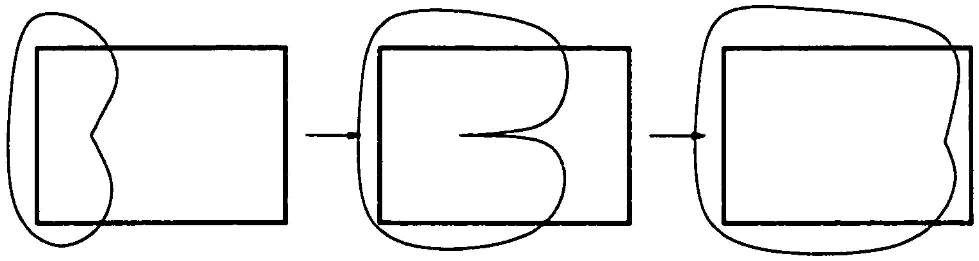


Figure 7.11: With the non-penetration behavior, a crease forms along the stressed edge, and then resolves itself as the skin grows.

7.6 Multiresolution editing

The approach to modeling we have targeted – sculpting complex surfaces by constructing the underlying masses that define their shape – naturally allows the user to work in a coarse-to-fine manner. That is, the user can begin with a coarse cylindrical shape to represent a torso, and then refine the surface, adding masses to define the shapes of individual muscles, bones, and tendons. The subdivision surface framework is naturally compatible with such an approach. With each level of refinement, the surface can effectively "resolve" detail at finer scales. The systems described in [58, 89] allow the user to add detail to a subdivision surface at multiple scales by manipulating vertices individually. We now explain how to extend the skin algorithm with little modification to support *subdivision skeletons*. A subdivision skeleton is like a regular skeleton except that it is active only at specific levels of subdivision. When the skin is subdivided, its vertices act as *subdivision particles* that track points on the subdivision skeletons active at that level.

Intuitively, subdivision particles that lie inside the implicit surface of a skeleton should move in the normal way (toward the implicit surface). A particle far from any subdivision skeleton should remain at the position it was initially assigned through subdivision, which we call its *base point*. In some range near the skeleton, a particle should smoothly blend between its base point and the displacement induced by the skeleton in the region of skin near the particle.

We achieve this as follows. Each particle evaluates the implicit function F as before, sharing information with its neighbors and tracking a point on some skeleton active at the current level of subdivision. Each particle has an associated "cutoff distance" M, equal to 3 times its reference length, beyond which it is unaffected by any skeleton. As before (section 7.4.1), its new position is computed from a weighted average of its current position, the centroid of its neighbors, and its target position:

$$\mathbf{x}'_p = \alpha \mathbf{x}_p + \beta \mathbf{c}_p + \gamma \mathbf{t}_p.$$

We now choose the target position and weights α, β, and γ according to the value of the implicit function: when $F(\mathbf{x}_p) < 0$, the particle sets its target as before and uses weights $\alpha = 0.3$, $\beta = 0.3$, and $\gamma = 0.4$. When the implicit function is 0 or M we choose values for α and γ according to the table in Figure 7.12, setting $\beta = 1 - \alpha - \gamma$. For values of $F(\mathbf{x}_p)$ between 0 and M we choose the target position and weights by interpolating the corresponding values listed in the table.

$F(\mathbf{x}_p)$	α	γ	$\mathbf{t}_p$
0	.3	.4	$\mathbf{x}_p$
$\geq M$	0	1	$\mathbf{b}_p$

Figure 7.12: Computing the target position for subdivision particles.

7.7 Topology changes

At each iteration, the skin evolves through local operations, using local information. Global properties of the skin surface, such as its genus, are not taken into account (or modified). Consequently, we do not automatically detect topological changes to the implicit surface. Figure 7.13 shows an example of a skin growing over a toroidal skeleton. When the new skeleton is first added, it changes the genus of the implicit surface. The skin grows around the torus and through itself in an attempt to fit the implicit surface. We could perhaps detect this interpenetration and prevent it. However, since it is global and not local, the method given in section 7.5 will not work.

Instead, we allow the user to explicitly change the genus of the skin by joining two sections of surface or by cutting a region of skin into two parts. Because the skin will evolve a poor triangulation into a good one, we can perform these operations without worrying about leaving a good triangulation.

To cut the skin, we split the edges and faces that cross a user-specified cutting plane, and stitch up the resulting holes with a simple triangulation. (If desired, we can place crease curves along the cut to preserve its shape.) To join two portions of skin surface, we remove a face on each portion to create two holes, then connect the holes with a bridge of triangles. In either case, when editing the topology of the skin it's important to edit the skeletons accordingly so that the implicit surface matches the skin's new topology.

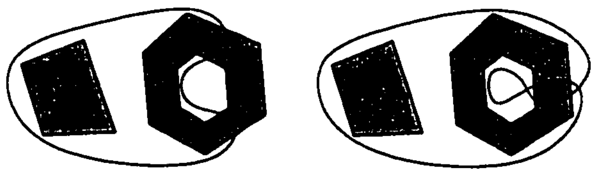


Figure 7.13: The skin does not detect the topology of the isosurface, which is toroidal, and hence will interpenetrate unless the user intervenes.

7.8 Software framework

We developed the skin algorithm within a larger system consisting of a collection of C++ libraries that provide extensive support for 3D modeling and interaction. The mesh class consists of a collection of vertices, edges, and faces (generically called simplices) that store connectivity information as described in section 7.3. To implement skin, we first introduced a separate class, Tessellator, that operates on a mesh's simplices according to some procedure, typically by editing connectivity and vertex positions. We made a small modification to the existing simplex classes to allow a tessellator to store arbitrary data directly on them.

The skin algorithm is implemented by a skin tessellator, which is a type of 2D tessellator - that is, it operates on a region of surface. Other types of tessellators include point tessellators and curve tessellators, which are 0D and 1D tessellators, respectively. Each Tessellator receives a callback every frame in which it iterates over its simplices and edits them according to its procedure. We enforce the rule that if multiple tessellators attempt to edit the same simplex, the tessellator with the lowest dimension wins. For example, we represent a curve constraint with a 1D tessellator that acts on its vertices and edges to approximate the curve. A skin tessellator operating on the surrounding surface is not allowed to edit the vertices and edges of the curve.

A vertex is considered active only if it has moved within the last 32 frames. Edges and faces are considered active if some contained vertex is active. We also activate simplices when the user performs an operation that could cause particles to move, such as changing a skeleton's offset distance or target length, or interactively moving a constraint curve. Tessellators operate only on simplices that are active. Since users typically edit just a local region of a surface at a time, this can significantly increase the interactivity of the system - particularly when the user edits a subdivision skin, which may have a large number of inactive vertices.

7.9 Discussion

As stated in the introduction, skin was designed to be used in an interactive free-form modeling setting. Since our ultimate aim is to create high-quality models, we believe our results should be judged by what types of models we can create within this framework. Although we have not designed the final interface for our system, we do have a preliminary implementation that lets the user create primitives either using SKETCH [87] or by importing them from other modeling packages. The user can interactively create a ball of skin and grow it over a primitive. Editing operations include instructing the skin to grow over a new skeleton, adjusting a skeleton's offset distance or target length, drawing and editing crease curves, and changing the genus of the skin surface.

With this system we have created the models shown in Figures 7.8, 7.14, 7.15, and 7.16. The torso model was created by growing skin over the skeleton shown in the top image of Figure 7.16. The skeletons were created by a user with no artistic training using SKETCH. Some skeletons were created by growing skin over a sketched primitive. The skin surface was edited at both one and two levels of subdivision. For example, the muscles of the neck were added as subdivision skeletons. The foot and hand models demonstrate how skin can be used to create cartoonish effects. Both models are expressive, yet were built by growing skin over simple skeletons. The face model is more complex. Note the use of crease curves to delineate the mouth, nose, and eye sockets. This model would be difficult to create using a surface patch representation. The skeletons, created by a skilled computer artist using a conventional modeling system, are shown in the top image of Figure 7.14.

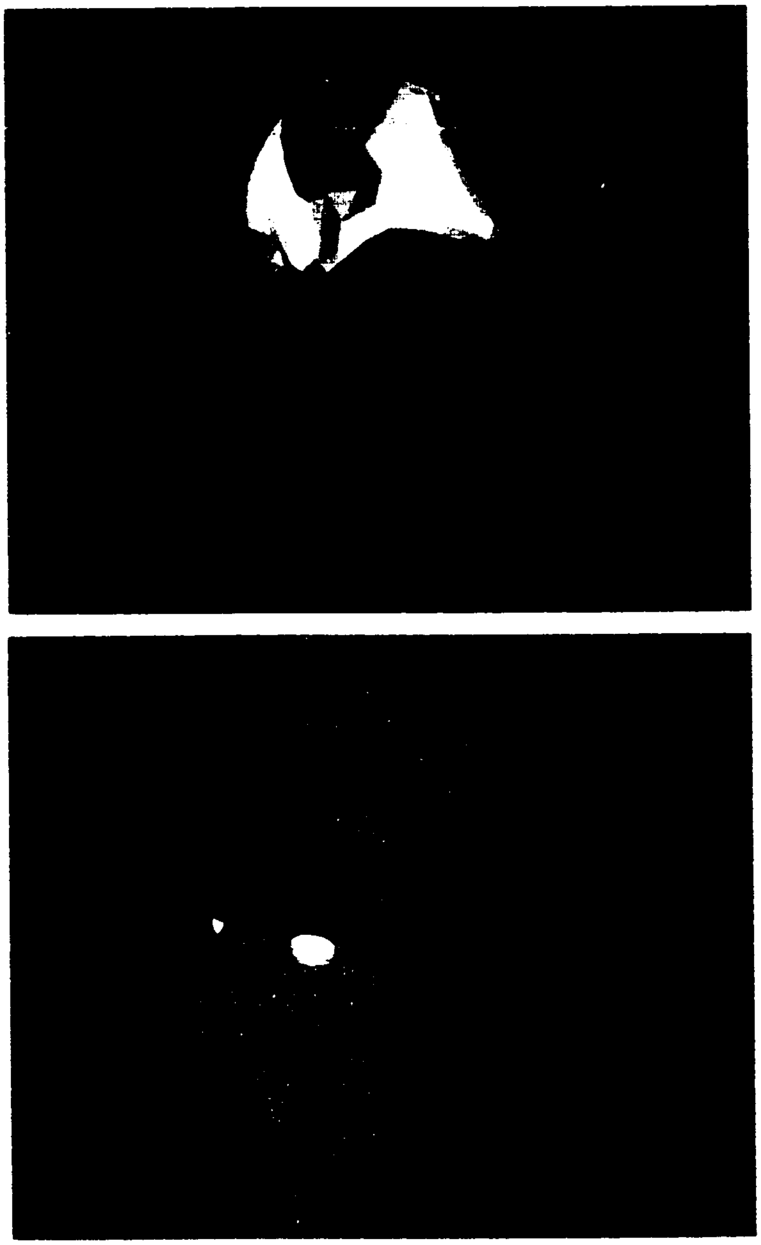


Figure 7.14: This face model consists of a skin surface with about 3500 particles. The skeleton, shown in the upper image, was created using a commercial modeling package. Notice the fine detail around the nose and eyes, which was created using creases in the base mesh, and subdivision skeletons.

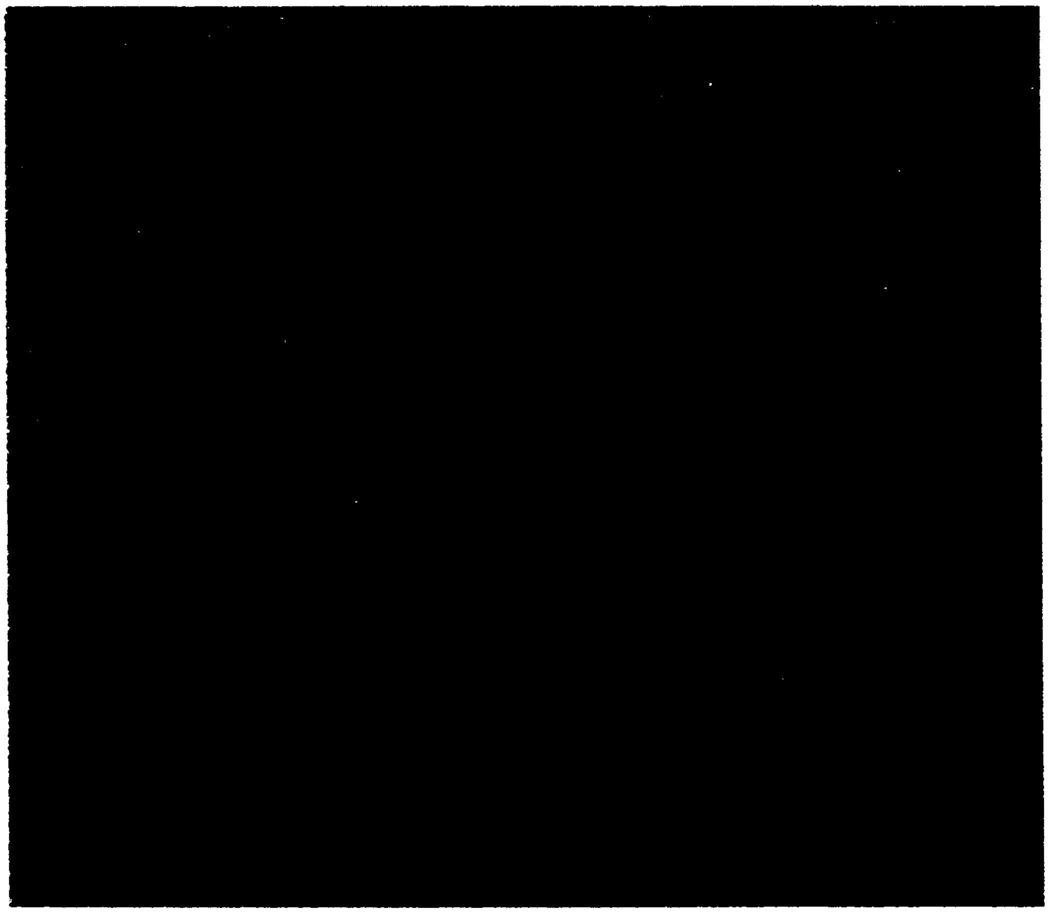


Figure 7.15: This frog hand is based on an image from a children's book. It was made from 10 skeleton meshes. Notice the smooth webbing between the fingers.

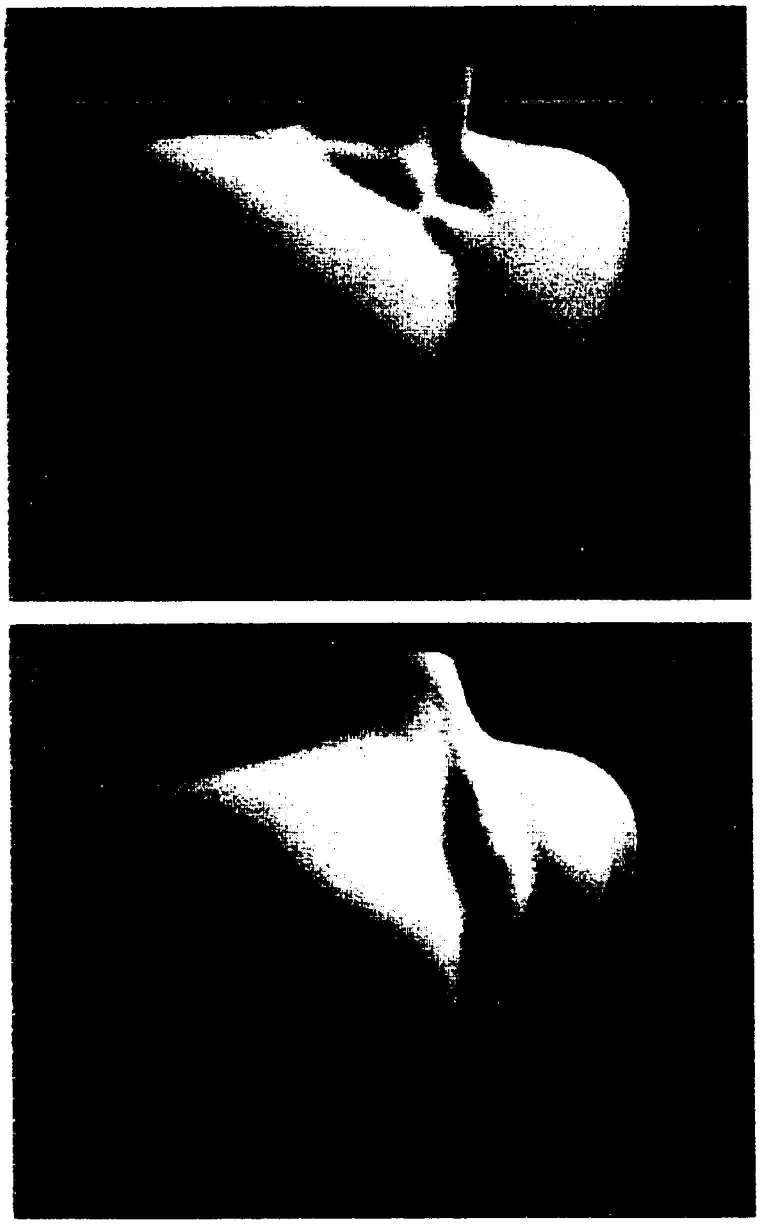


Figure 7.16: The torso model was edited at 2 levels of subdivision. For example, the vertical neck muscles and the spine were added as subdivision skeletons. Notice the indications of underlying muscles, especially around the neck.

Figure 7.17: The skeletons used to model the torso in Figure 7.16.

Chapter 8

Conclusions and Future Work

8.1 Summary

This work began with a basic observation: photorealistic depiction of 3D virtual scenes comes at a high cost, both to the creator of the scene, who must supply vast amounts of specific and accurate data, and in rendering time spent processing all that data each frame to produce the final image. Yet photorealistic imagery is certainly the wrong choice in some cases – depending on the target audience or intended meaning of the image. In some cases, the costs of modeling, storing, transmitting and rendering a particular scene photorealistically may simply be too great, so that some other scene, subject matter or story must be substituted for the preferred one.

When we started this work, the costs associated with nonphotorealism were even greater than for photorealism (including fake "photorealistic" techniques like Gouraud shading and texture maps). In Chapter 2 we showed that it's possible to adopt a stripped-down rendering style resembling hand-made line drawings in which the rendering costs are appropriately stripped-down as well. The key idea – our randomized silhouette detection algorithm – is simple to implement, effective, and fast, as we both proved analytically and confirmed experimentally.

In Chapter 3 we introduced a procedural texture framework and a new visibility determination algorithm that served as the basis for the two new classes of rendering algorithms described in Chapters 4 and 5. In Chapter 4 we developed a variety of hatching techniques that work on arbitrary polygonal models. Using the framework of Chapter 3 these techniques work at interactive rates.

In Chapter 5 we developed a new class of rendering algorithms that borrow strategies from human artists and illustrators to depict complex objects and scenery with relatively few strokes. The payoffs are both in terms of speed and expressive power. A single basic technique is used to create a variety of distinct visual styles for depicting fur, leaves, grass, and other "textures" made up of collections of small detail elements. An interesting feature of this method is that only detail elements that contribute to the final image are drawn. Detail elements associated with off-screen regions simply cease to exist. This means that the running time of the algorithm is proportional to the area of the image, if we ignore the cost of rendering the needed reference images.

In part II we described two powerful new modeling techniques. The first lets a designer apply her existing drawing skills to define 3D curves from 2D drawings of them. The second technique, skin, lets us build up complex, free-form surfaces from collections of simpler masses. This approach holds particular promise for modeling organic shapes such as human figures, as we demonstrated. The two modeling techniques specifically lend themselves to use within an integrated system that leverages the artist's drawing skills.

Taken together, this work has broken significant new ground in several areas relating to modeling and rendering of stylized virtual scenes. The potential benefits of following through on our original observation – that art-based techniques can be mined to increase the expressive power of 3D graphics – are certainly not exhausted.

8.2 Future work

The "ideal" modeling system we described in Chapter 1 still does not exist. That ideal system would allow the artist to transfer her skills to the problem of describing 3D shapes and imparting the stylized "look" used to depict them. The artist should be able to quickly sketch a shape – say a human figure – and get a compelling result, quickly, that bears the artist's distinctive style.

The drawing in Figure 8.1 is an example of the type of thing we have in mind. It took just a few minutes to draw. It's really not very complicated. It may not be drawn well – nonetheless, our ideal system would allow the artist to generate a 3D model resembling the figure in the drawing, including the suggestion of hair. Strokes would be assigned to depict significant features of the model – dimples, creases, muscles – as in the drawing. Letting the artist specify which features are "significant" gives her more control over the final rendering, while potentially easing the modeling burden, since the underlying geometry need

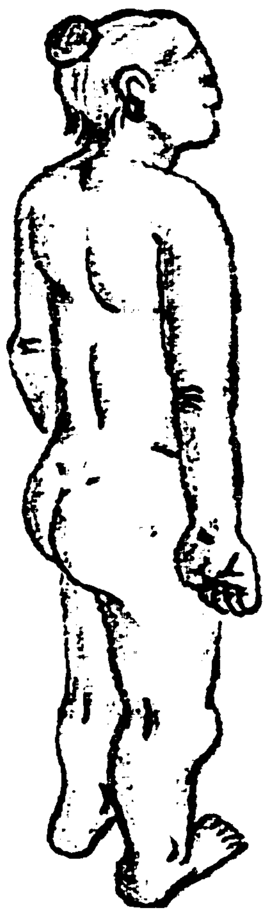


Figure 8.1: A simple drawing.

not be specified as carefully. The rendering should lightly apply shading that resembles the pencil strokes in the drawing, freeing the artist from having to specify detailed reflectance properties that vary over the surface geometry. This simple "less is more" approach to shading also avoids the pitfall of having the figure appear to be made of plastic or rubber due to incorrect data or assumptions used in a "photorealistic" method.

An important question we have yet to address is this: How does the artist create the textures and give them their distinctive looks and view-dependent behavior? This is a crucial issue, but also a hard one. Certainly rendering algorithms must exist before there can be a user interface to assign and parameterize them. But the reason it's a crucial issue has to do with the nature of nonphotorealism.

How do we recognize that a photorealistic rendering is right?

Answer: when it looks photorealistic.

How do we know when a nonphotorealistic rendering is right?

The correct answer should be: when it looks *as its designer intended.* Art-based rendering, by its nature, really only makes sense as a means of communication. Stylization and control of detail must be in service of some *intention.* Up to now the bulk of research into nonphotorealistic rendering for computer graphics has ignored this issue. The commonly used yardstick is that it's right when it looks *good.* This is okay for the early stages of research into this new area, but this simplistic approach must evolve toward providing tools for artists to actively model the style of the scene according to each one's intention and distinctive vision. We intend to address this in the future.

A new paper, to appear in the spring of 2000, describes our recent work on "static graftal" textures [51]. This work makes two contributions - we develop a new graftal texture framework that provides much more flexibility to the designer, to create the shapes, looks and behaviors of a broad range of graftal types. The second contribution is that our new framework is significantly more temporally coherent than the method described in Chapter 5. The next step is already underway: developing a powerful UI to let artists directly sketch graftal textures – or shading stroke textures – by example into a 3D scene.

We have begun work to revamp the UI for sketching 3D curves, both to generalize the methods (defining a curve from two views, say) and to incorporate other curve-sketching techniques. We intend to integrate this UI within a free-form sketching system based on smooth primitives and new types of skin. A high priority will be to integrate the texturing and modeling aspects of the interface into a single system – which may finally begin to approach the "ideal system" we originally envisioned.

As we begin to attain these abilities it will become increasingly important to work with artists, to let them use the system, give feedback and recommend changes. This will mark the beginning of a new phase for the work, as we embark on an exploration of the huge range of possibilities for stylized depiction of 3D shape.

Bibliography

[1] Greg Albert, editor. *Basic Figure Drawing Techniques.* North Light Books, Cincinnati, Ohio, 1994.

[2] Alias / Wavefront. *Maya*, 1.0 edition, 1998.

[3] A. Appel. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. In *Proceedings of ACM National Conference*, pages 387–393, 1967.

[4] Autodesk. *3D Studio MAX*, 1996.

[5] Normal I. Badler and Andrew S. Glassner. 3D Object Modeling. In *SIGGRAPH '97 Introduction to Computer Graphics Course Notes.* ACM SIGGRAPH, August 1997.

[6] Michael J. Banks and Elaine Cohen. Realtime Spline Curves from Interactively Sketched Data. *1990 Symposium on Interactive 3D Graphics*, 24(2):99–107, March 1990. ISBN 0-89791-351-5.

[7] Thomas Baudel. A Mark-Based Interaction Paradigm for Free-Hand Drawing. In *Proceedings of UIST 94*, pages 185–192. ACM SIGGRAPH, 1994.

[8] J. Blinn. *Jim Blinn's Corner*, chapter 10, pages 91–102. Morgan Kaufmann, 1996.

[9] James F. Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[10] Jules Bloomenthal and Ken Shoemake. Convolution Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):251–257, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[11] Jules Bloomenthal and Brian Wyvill. Interactive Techniques for Implicit Modeling. *1990 Symposium on Interactive 3D Graphics*, 24(2):109–116, March 1990. ISBN 0-89791-351-5.

[12] OpenGL Architecture Review Board. *OpenGL Reference Manual, 2nd Edition.* Addison-Wesley Developers Press, 1996.

[13] George B. Bridgman. *Constructive Anatomy.* Dover Publications, Inc., New York, 1973.

[14] Charles Burns. Black Hole # 6. Comic book series, 1998. Fantagraphic Books.

[15] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. An Interface for Sketching 3D Curves. *1999 ACM Symposium on Interactive 3D Graphics*, pages 17–22, April 1999. ISBN 1-58113-082-1.

[16] Michael F. Cohen. Interactive spacetime control for animation. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):293–302, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[17] Debra Dooley and Michael Cohen. Automatic Illustration of 3D Geometric Models: Lines. *1990 Symposium on Interactive 3D Graphics*, 24(2):77–82, March 1990. ISBN 0-89791-351-5.

[18] Gershon Elber and Elaine Cohen. Hidden Curve Removal for Free Form Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):95–104, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[19] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing Triangle Strips for Fast Rendering. *IEEE Visualization '96*, pages 319–326, October 1996. ISBN 0-89791-864-9.

[20] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, 3rd edition, 1993.

[21] J. Foley, A. van Dam, S. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*, chapter 15, pages 666–667. Addison-Wesley, 1992.

[22] Barry Fowler and Richard Bartels. Constraint-based curve manipulation. *IEEE Computer Graphics & Applications*, 13(5):43–49, September 1993.

[23] R. Galimberti and U. Montanari. An Algorithm for Hidden Line Elimination. *Communications of the ACM*, 12(4):206–211, April 1969.

[24] Dr. Seuss (Theodor Geisel). *The Lorax.* Random House, New York, 1971.

[25] Dr. Seuss (Theodor Geisel). *The Foot Book.* Random House, New York, 1988.

[26] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press., Baltimore, 3rd edition, 1996. ISBN 0-8018-5414-8.

[27] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. *Proceedings of SIGGRAPH 98*, pages 447–452, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[28] Louise Gordon. *How to Draw the Human Figure.* Penguin Books, New York, 1979.

[29] Alexander Graham. *Nonnegative Matrices and Applicable Topics in Linear Algebra.* Ellis Horwood Ltd., Chichester, England, 1987. ISBN 0-470-20855-4.

[30] Donald P. Greenberg, Kenneth E. Torrance, Peter Shirley, James Arvo, James A. Ferwerda, Sumanta Pattanaik, Eric P. F. Lafortune, Bruce Walter, Sing-Choong Foo, and Ben Trumbore. A Framework for Realistic Image Synthesis. *Proceedings of SIGGRAPH 97*, pages 477–494, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[31] Cindy Grimm and Matthew Ayers. A Framework for Synchronized Editing of Multiple Curve Representations. *Computer Graphics Forum*, 17(3):31–40, 1998. ISSN 1067-7055.

[32] Paul E. Haeberli. Paint By Numbers: Abstract Image Representations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):207–214, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[33] Geoffrey Hayes. *Patrick and Ted.* Scholastic, Inc., New York, 1984.

[34] Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, S. Scott Snibbe, and Andries van Dam. Interactive Shadows. In *Proceedings of UIST 92*, pages 1–6. ACM SIGGRAPH, November 1992.

[35] Burne Hogarth. *Dynamic Anatomy.* Watson-Guptill Publications, New York, paperback edition, 1990.

[36] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise Smooth Surface Reconstruction. *Proceedings of SIGGRAPH 94*, pages 295–302, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[37] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH 93*, pages 19–26, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[38] C. Hornung. A Method for Solving the Visibility Problem. *IEEE Computer Graphics & Applications*, 4:26–33, July 1984.

[39] S. C. Hsu, I. H. H. Lee, and H. E. Wiseman. Skeletal Strokes. In *Proceedings of UIST '93*, pages 197–206, November 1993.

[40] Siu Chi Hsu and Irene H. H. Lee. Drawing and Animation Using Skeletal Strokes. *Proceedings of SIGGRAPH 94*, pages 109–118, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[41] T. Igarashi, R. Kadobayashi, K. Mase, and H. Tanaka. Path Drawing for 3D Walk-through. In *Proceedings of UIST 98*, pages 173–174. ACM SIGGRAPH, 1998.

[42] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Pegasus: A Drawing System for Rapid Geometric Design. In *CHI'98 Summary (ACM Conference on Human Factors in Computing Systems)*, pages 24–25, 1998.

[43] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *Proceedings of SIGGRAPH 99*, pages 409–416, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[44] 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. *Proceedings of SIGGRAPH 99*, pages 433–438, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[45] John Lansdown and Simon Schofield. Expressive Rendering: A Review of Nonphotorealistic Techniques. *IEEE Computer Graphics & Applications*, 15(3):29–37, May 1995.

[46] W. Leister. Computer Generated Copper Plates. *Computer Graphics Forum*, 13(1):69–77, January 1994.

[47] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[48] P. Loutrel. A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra. *IEEE Transactions on Computers*, C-19(3):205–213, March 1970.

[49] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John F. Hughes. Skin: A Constructive Approach to Modeling Free-form Shapes. *Proceedings of SIGGRAPH 99*, pages 393–400, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[50] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. *Proceedings of SIGGRAPH 97*, pages 415–420, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[51] 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 the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. To be held in Annecy, France.

[52] Barbara J. Meier. Painterly Rendering for Animation. *Proceedings of SIGGRAPH 96*, pages 477–484, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[53] James V. Miller, David E. Breen, William E. Lorensen, Robert M. O'Bara, and Michael J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):217–226, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[54] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. To be held in Annecy, France.

[55] Randy Pausch, Tommy Burnette, Dan Brockway, and Michael E. Weiblen. Navigation and Locomotion in Virtual Worlds via Flight Into Hand-Held Miniatures. *Proceedings of SIGGRAPH 95*, pages 399–400, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[56] Ken Perlin. An Image Synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985. Held in San Francisco, California.

[57] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*, chapter 7. Springer-Verlag, 1985.

[58] Kari Pulli and Michael Lounsbery. Hierarchical Editing and Rendering of Subdivision Surfaces. Technical report, University of Washington, 1997.

[59] Walt Reed, editor. *The Figure: An Approach to Drawing and Construction*. North Light Books, Cincinnati, Ohio, 2nd edition, 1984.

[60] W. T. Reeves. Particle Systems - a Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics*, 2(2):91–108, April 1983. Held in USA.

[61] William T. Reeves and Ricki Blau. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):313–322, July 1985. Held in San Francisco, California.

[62] A.A.G. Requicha. Toward a theory of geometric tolerancing. *International journal of robotics research*, 2(4):45–49, 1983.

[63] Dean Rubine. Specifying gestures by example. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):329–337, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[64] Emanuel Sachs, Andrew Roberts, and David Stoops. 3-Draw: A tool for designing 3D shapes. *IEEE Computer Graphics & Applications*, 11(6):18–26, November 1991.

[65] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):197–206, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[66] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image-Based Pen-and-Ink Illustration. *Proceedings of SIGGRAPH 97*, pages 401–406, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[67] Fritz Schider. *An Atlas of Anatomy for Artists*. Dover Publications, Inc., New York, 3rd American edition, 1957.

[68] P.H. Schneider. An Algorithm for Automatically Fitting Digitized Curves. In A. Glassner, editor, *Graphics Gems*. Academic Press, 1990.

[69] Jean Schweitzer. *Analysis and Application of Subdivision Surfaces.* PhD thesis, University of Washington, 1996.

[70] Karan Singh and Eugene Fiume. Wires: A Geometric Deformation Technique. *Proceedings of SIGGRAPH 98*, pages 405–414, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[71] Alvy Ray Smith. Plants, Fractals and Formal Languages. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):1–10, July 1984. Held in Minneapolis, Minnesota.

[72] Barton T. Stander and John C. Hart. Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling. *Proceedings of SIGGRAPH 97*, pages 279–286, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[73] T. Strothotte, B. Preim, A. Raab, J. Schumann, and D. R. Forsey. How to Render Frames and Influence People. *Computer Graphics Forum*, 13(3):455–466, 1994.

[74] McMASTER-CARR supply company. Catalog 98. book, 1998. P.O. Box 440, New Brunswick, NJ 08903-0440.

[75] David Suter. Illustration for the New York Times, May 19, 1997.

[76] I. Sutherland, R. Sproull, and R. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys*, 6(1):1–55, March 1974.

[77] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):185–194, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[78] Hal Tollison. *Cartooning.* Walter Foster Publishing, Inc., Laguna Hills, CA, 1989.

[79] C.W.A.M van Overveld and B. Wyvill. Shrinkwrap: an adaptive algorithm for polygonizing and implicit surface. Technical report, University of Calgary, 1993.

[80] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999. ISSN 0178-2789.

[81] William Welch and Andrew Witkin. Free-Form Shape Design Using Triangulated Surfaces. *Proceedings of SIGGRAPH 94*, pages 247–256, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[82] L. R. Williams. Topological Reconstruction of a Smooth Manifold-Solid from its Occluding Contour. Technical Report 94-04, University of Massachusetts, Amherst, MA, 1994.

[83] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[84] Georges Winkenbach and David H. Salesin. Computer-Generated Pen-And-Ink Illustration. *Proceedings of SIGGRAPH 94*, pages 91–100, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[85] Andrew P. Witkin and Paul S. Heckbert. Using Particles to Sample and Control Implicit Surfaces. *Proceedings of SIGGRAPH 94*, pages 269–278, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[86] Robert C. Zeleznik, Andrew S. Forsberg, and Paul S. Strauss. Two Pointer Input For 3D Interaction. *1997 Symposium on Interactive 3D Graphics*, pages 115–120, April 1997. ISBN 0-89791-884-3.

[87] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. *Proceedings of SIGGRAPH 96*, pages 163–170, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[88] J. M. Zheng, K. W. Chan, and I. Gibson. A New Approach for Direct Manipulation of Free-Form Curve. *Computer Graphics Forum*, 17(3):327–334, 1998. ISSN 1067-7055.

[89] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive Multiresolution Mesh Editing. *Proceedings of SIGGRAPH 97*, pages 259–268, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.