Computing Smooth Surface Contours with Accurate Topology

Pierre Bénard, Aaron Hertzmann, Michael Kass

▶ **To cite this version:**

Pierre Bénard, Aaron Hertzmann, Michael Kass. Computing Smooth Surface Contours with Accurate Topology. ACM Transactions on Graphics, 2014, 33 (2), 10.1145/2558307 . hal-00924273v2

HAL Id: hal-00924273

https://inria.hal.science/hal-00924273v2

Submitted on 19 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computing Smooth Surface Contours with Accurate Topology

PIERRE BÉNARD
University of Toronto, Université de Bordeaux, LaBRI, CNRS, and Inria
AARON HERTZMANN
Adobe Research, University of Toronto, and Pixar Animation Studios
and
MICHAEL KASS
Pixar Animation Studios

This paper introduces a method for accurately computing the visible contours of a smooth 3D surface for stylization. This is a surprisingly difficult problem, and previous methods are prone to topological errors, such as gaps in the outline. Our approach is to generate, for each viewpoint, a new triangle mesh with contours that are topologically-equivalent and geometrically close to those of the original smooth surface. The contours of the mesh can then be rendered with exact visibility. The core of the approach is *Contour-Consistency,* a way to prove topological equivalence between the contours of two surfaces. Producing a surface tessellation that satisfies this property is itself challenging; to this end, we introduce a type of triangle that ensures consistency at the contour. We then introduce an iterative mesh generation procedure, based on these ideas. This procedure does not fully guarantee consistency, but errors are not noticeable in our experiments. Our algorithm can operate on any smooth input surface representation; we use Catmull-Clark subdivision surfaces in our implementation. We demonstrate results computing contours of complex 3D objects, on which our method eliminates the contour artifacts of other methods.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Line and curve generation*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*

General Terms: Algorithms

Additional Key Words and Phrases: Non-photorealistic rendering, line drawing, silhouettes, visibility, geometry processing

ACM Reference Format:

Pierre Bénard, Aaron Hertzmann, Michael Kass. 2014. Computing Smooth Surface Contours with Accurate Topology. ACM Trans. Graph. 33, 2, Article 19 (March 2014), 21 pages.
DOI: http://dx.doi.org/10.1145/2558307

This work was supported in part by NSERC and CIFAR.

Authors' addresses: P. Bénard, University of Toronto, Canada, Universite de Bordeaux, LaBRI, CNRS, and Inria, France pierre.benard@laposte.net; A. Hertzmann, Adobe Research, CA, University of Toronto, Canada, and Pixar Animation Studios, CA; M. Kass, Pixar Animation Studios, CA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2014 ACM 0730-0301/2014/14-ART19 $15.00
DOI: http://dx.doi.org/10.1145/2558307

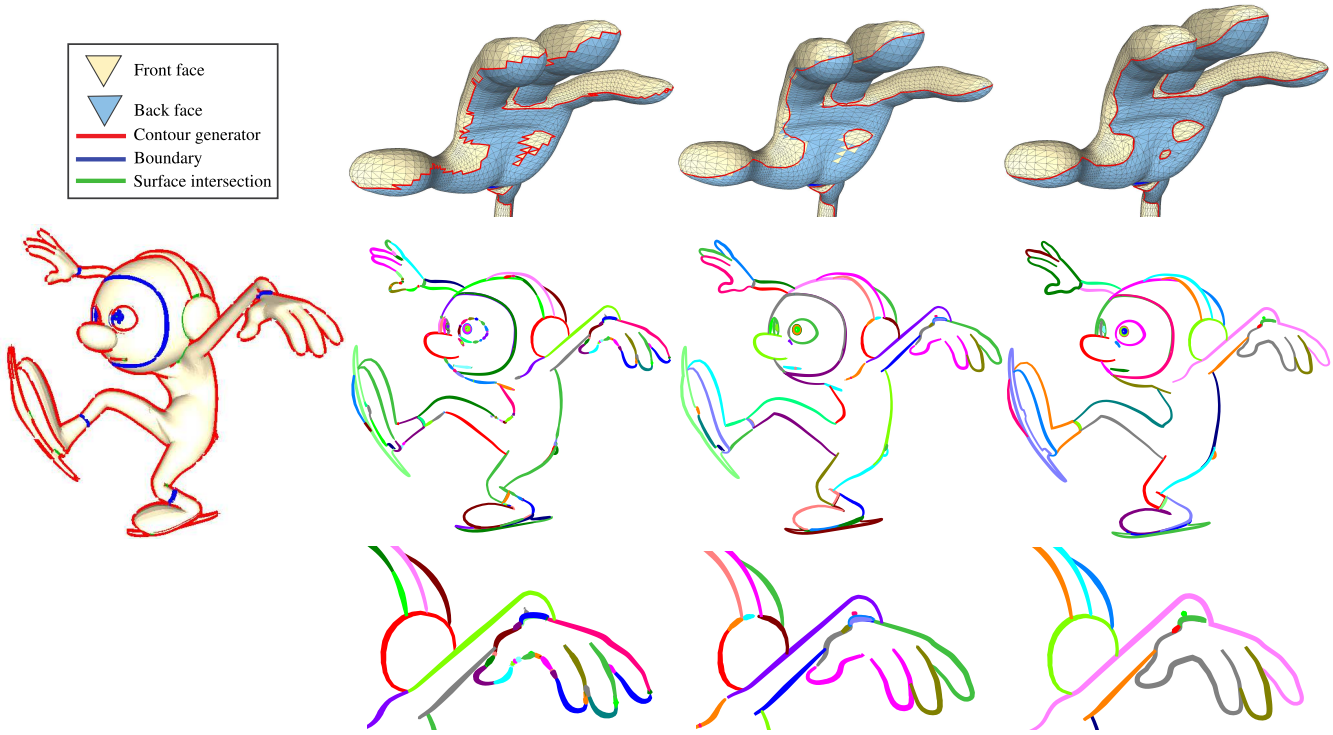
1. INTRODUCTION

Computing the occluding contours[1] of 3D surfaces is one of the oldest problems in computer graphics. Contours were first used for hidden-line rendering, and then for non-photorealistic rendering. Stylizing contours is fundamental to 3D non-photorealistic rendering, because contours mimic many hand-drawn curves. Despite this long history, computing and rendering the contour for smooth surfaces with correct visibility remains an unsolved problem. The problem is surprisingly difficult to solve perfectly, and existing methods are prone to pathological visibility artifacts. For example, during an animation, outline curves may incorrectly appear and disappear, seemingly at random, and strokes may be arbitrarily broken into multiple segments. For high-quality animation, the artifacts are extremely objectionable; the animation of a cartoon character would be ruined by a stylized outline curve that randomly breaks into pieces. Likewise, in an industrial design rendering, small gaps in curves would distort the perceived object shape. Previous research has focused on applications that can tolerate imperfection, such as rendering very simple surfaces, curves without stylization, sketchy real-time rendering, and static imagery. Even for sketchy rendering, we argue that sketchiness should be controllable, instead of arising from unpredictable errors.

This paper introduces a new method for computing topologically-accurate contours of smooth surfaces. Given a smooth surface and a camera viewpoint, the algorithm produces a triangle mesh, such that the contour generator of the mesh is topologically-equivalent and geometrically close to the smooth surface's contours. The visible contours of the mesh are mesh edges, and can be accurately computed by standard methods. The core of the approach is the notion of *Contour Consistency,* a way to prove topological equivalence between the contours of a smooth surface and a triangle mesh. In order to produce a Contour Consistent mesh, we define Radial Triangles, which ensure consistency near the contour. We introduce a tessellation algorithm that produces these triangles through a sequence of local mesh transformations. Our algorithm is not strictly guaranteed to produce a topologically-consistent mesh, as it can yield isolated inconsistencies in the mesh. However, these inconsistencies are identified, and our method can bypass them to produce correct curves. Numerical sampling is used to detect contours, so contours could hypothetically be missed, though this does not appear to be an issue in practice.

The method works for orientable surfaces in general position, with the assumption that only front-facing surface points are visi-

[1] In this paper, we use the term *contour* to refer to the occluding contours of a surface. They are also sometimes referred to as silhouettes.



(a) 3D model (b) Conventional mesh contours (c) Interpolated contours (d) Our method

Fig. 1. **Contour extraction from a smooth 3D surface. (a)** The object is modeled as a collection of Catmull-Clark surfaces. We extract contours (red), boundaries (blue), and surface-surface intersection curves (green). **(b)** Contours of a conventional surface tessellation exhibit overly-complex topology. **(c)** Smooth interpolated contours [Hertzmann and Zorin 2000] have simpler topology. However, because they are not true contours of the tessellation, there are visibility errors, such as the gap in the left hand, and broken outline curves. **(d)** Our method generates a mesh tessellation such that the mesh contours have accurate topology for the underlying surface. **Colors:** Front-faces are drawn in yellow, and back-faces in blue. All strokes are rendered with tapering and random colors, in order to visualize the extracted topology. Red © Disney/Pixar

ble. We apply the algorithm to Catmull-Clark subdivision surfaces. In principle, the approach can be applied to any piecewise-smooth piecewise-parametric surface representation.

An unexpected observation from our work is that the true contours of smooth surfaces can exhibit tiny kinks, loops, and other undesirable topological complexity. These details create undesirable breaks and overlaps in the stylized curves. These effects are surprising, because they can appear well below the apparent scale of the surface. We show examples of this undesirable complexity, and introduce a procedure for topological simplification of contours.

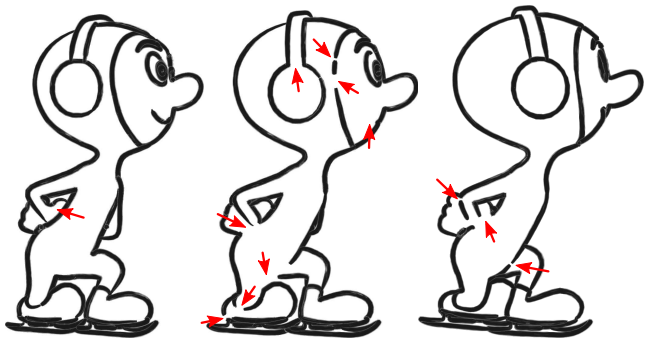
For animation production, surface intersection curves play an important role. Our paper describes why these curves are important and discusses methods for handling them, for the first time.

The renderings we produce are correct almost everywhere, according to a surface that is slightly perturbed from the input surface. The precise statement of our method's guarantees is as follows. First, the 3D contour generator is guaranteed to be topologically-equivalent (ambient isotopic) to those of the input surface, excluding small, sub-triangle loops. If desired, one may control the maximum size of these loops, e.g., by splitting all triangles larger than a given image-space area. Second, the geometric positions of the contour generator are accurate up to tiny perturbations. Third, visibility is computed correctly with respect to some surface that is slightly perturbed from the input surface, except in a few places where the algorithm is unable to obtain a Consistent surface. However, these locations are small and isolated, and local propagation usually produces correct visibility. The resulting renderings will have plausible topology, e.g., impossible gaps in the object silhouette are eliminated.

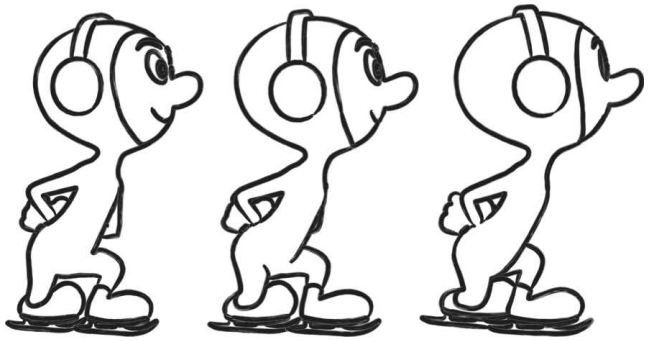
Contour extraction is just the first step in the hypothetical pipeline for exact computation and stylization of animated strokes. Several other substantial exact components will also be necessary, including complete topological simplification, spacetime correspondence between the extracted curves [Buchholz et al. 2011; Kalnins et al. 2003], and effective spacetime stylization. Each of these steps is an open problem in its own right, and we rely on previous approximate methods to generate stylized results. As illustrated in Figure 2, contour extraction is a critical part of this process, since minor errors at this step (e.g., gaps or breaks) get amplified into much bigger artifacts after stylization (temporal ones in particular).

2. PREVIOUS APPROACHES

This section surveys previous approaches to computing the visible contours for smooth surfaces. At first, the problem might appear to be quite straightforward, with many previously-published approaches to choose from. Previous methods work well for very simple, smooth surfaces, and for real-time rendering styles in which some errors are tolerable. However, each of the existing methods exhibit unexpected errors that are subtle, but pernicious for high-quality rendering. Here we analyze these problems and their un-



(a) Interpolated contours



(b) Our method

Fig. 2. **Contours stylized with tapered strokes. (a)** Interpolated contours exhibit many breaks and gaps (red arrows) which lead to objectionable temporal artifacts after stylization. **(b)** Our method avoids those issues, producing more coherent animated strokes. Red © Disney/Pixar

derlying causes. Our approach builds on many ideas from previous work, but is designed to avoid these problems.

The contour generator for any piecewise-smooth surface (e.g., subdivision surface or triangle mesh) is defined as the boundary between the front- and back-facing regions of the surface. The contour is the visible subset of the contour generator (Figure 3). Formally, the contour is the 2D projection, but, for brevity, we use the term to refer to both the 3D and 2D portions of the curve.

The fundamental difficulty in computing smooth contours is in computing visibility: for contours, visibility queries involve testing intersection between implicitly-defined quantities. For example, to determine if a point on the contour generator is visible, one may test if a camera ray through the point intersects the surface. Unfortunately, neither the contour generator nor the visible subset of a surface have closed-form expressions. Moreover, by definition, local surface visibility is unstable near the contour generator, in the sense that tiny perturbations in a ray test can change the result. These factors make direct visibility computation unreliable. An alternative approach is to modify the surface before determining the contours and visibility. Most commonly, this is done by tesselating the surface into a triangle mesh, but this produces messy, inaccurate contours. Many methods instead tessellate the contour and the surface separately, causing errors because the tessellated contour is no longer the contour of the tessellated surface. Trying to clean up these mismatches between representations leads to seemingly-endless headaches.

A popular approach to stroke rendering is to use image buffers and graphics hardware (e.g., [Cole and Finkelstein 2009; Hertzmann 1999; Saito and Takahashi 1990]) to directly identify contours. These methods can be very efficient, but cannot guarantee topological accuracy.

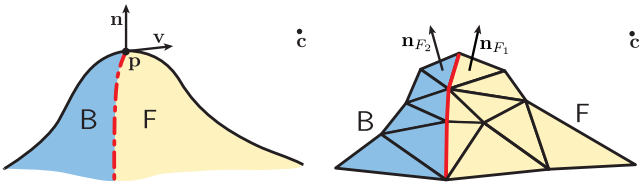


Fig. 3. **Contours.** The contour generator is the boundary between the front-facing and back-facing parts of a surface, as seen from a camera center $\mathbf{c}$. **Left:** For a smooth surface, these points have a surface normal $\mathbf{n}$ perpendicular to the view vector $\mathbf{v} = \mathbf{c} - \mathbf{p}$. **Right:** For a triangle mesh, the contour generator comprises the mesh edges between front-faces and back-faces.

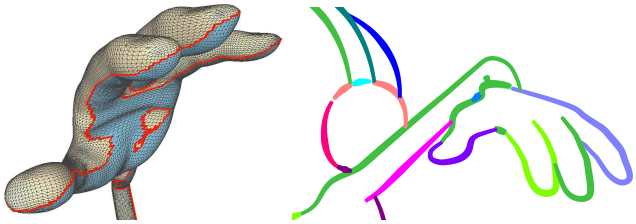


Fig. 4. **Subdividing a surface does not fix all topological problems. Left:** The mesh contour of a subdivided surface exhibits jaggy edges. **Right:** The interpolated contour computed from this mesh improves over Figure 1(b), but still exhibits many errors. Red © Disney/Pixar

2.1 Mesh Contours

The simplest way to compute contours of a smooth surface is to generate an approximate triangle mesh, and then render the contours of the triangle mesh. (In some situations, one begins with a triangle mesh without ever computing an explicit smooth representation). As noted by many previous authors, the contours of the approximating mesh exhibit much more complex topology than those of the smooth surface (Figure 1(b)). Subdividing the surface to a finer tessellation does not solve the problem (Figure 4).

Several methods topologically simplify the mesh contour using heuristics [Eisemann et al. 2008; Isenberg et al. 2002; Kirsanov et al. 2003; Northrup and Markosian 2000]. These heuristics often obtain appealing results, but cannot guarantee correct topology with respect to the original smooth surface. Depending on user-defined thresholds, they may merge unrelated curves, or leave the topology overly-complex. We apply similar topological simplification operators, but only once the correct topology is known.

2.2 Ray-Tracing Smooth Contours

A second approach is to perform computations directly on a smooth surface representation. Elber and Cohen [1990] detect contours on NURBS patches, using a root-finding method, and adaptively subdivide surface patches to refine the contour to a desired accuracy. Curves are split into curve segments whenever they are overlapped by a contour or boundary curve. Ray-tracing is used to determine which isoparametric curve segment is visible, and visibility is propagated to contours along these visible curves. More recent methods

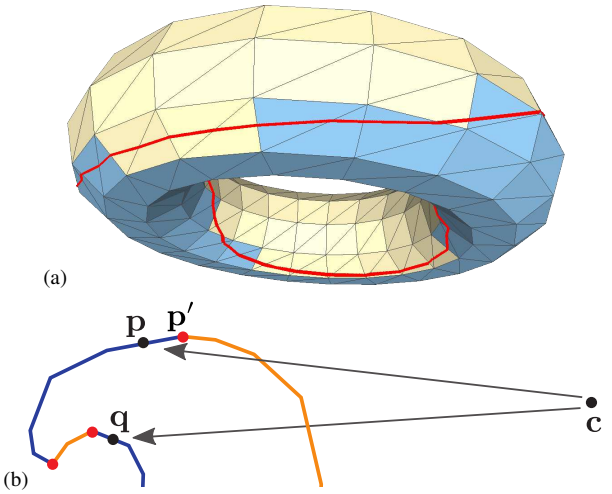


Fig. 5. **Difficulty of computing visibility with interpolated contours. (a)** The interpolated contour (red) does not correspond to the mesh contour, but instead crosses between front-faces and back-faces. Naive ray-tracing would mark the back-facing segments as invisible. **(b)** 2D cross-section of a surface, illustrating some of the difficulties in visibility for interpolated contours. Here, both $\mathbf{p}$ and $\mathbf{q}$ are points on a back-facing interpolated contour, each near a mesh contour (red). One approach is to ignore the occlusion of $\mathbf{p}$ by $\mathbf{p}'$, because $\mathbf{p}'$ is "near" $\mathbf{p}$ on the surface. However, there is no generic way to define "nearby" that will always work.

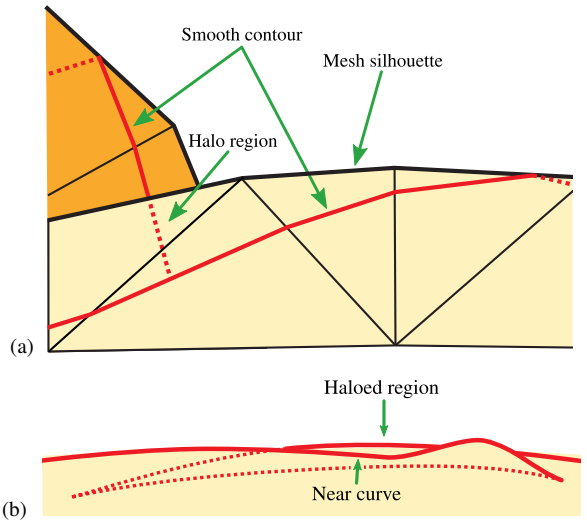


Fig. 6. **Haloing problem of interpolated contours. (a)** As pointed out by Eisemann et al. [2008], in image space, the interpolated contours lie inside the mesh contour, which affects planar map stylization. However, it also affects visibility, as there is a "halo region" between the two contours in which the mesh occludes any curves behind it, but the interpolated contour does not connect to the visible part. **(b)** An image-space illustration of highly-foreshortened curves in which the nearer curves will cause visibility errors for more distant curves. Here, the near curve's halo hides the far curve. Situations like this often arise for surfaces with noisy, nearly-flat regions, such as the Stanford bunny.

skip adaptive subdivision and iterative root-finding. Gooch [1998] uses linear interpolation to find contour generator points on B-spline patches, but does not compute visibility. Hertzmann and Zorin [2000] apply the same approach to arbitrary triangle meshes without any explicit underlying smooth surface, and compute visibility by image-space intersections and ray tests.

The fundamental difficulty in these methods is in robustly determining the visible portions of the smooth contour generator.

Contour ray test problems. An obvious approach is to perform contour ray tests directly against the smooth surface. These tests must be performed numerically, because both the smooth contour and the ray-surface intersection are defined implicitly. As noted by Elber and Cohen [1990], such tests are numerically unstable, since the true contour lies exactly on the boundary between visible and invisible. To our knowledge, this approach has not been attempted in the literature.

One may instead use a piecewise-linear approximation to the contour generator, and perform ray tests with respect to an approximate triangle mesh [Hertzmann and Zorin 2000]. Unfortunately, the approximate contour generator is not the contour generator of the mesh (Figure 1(c)). Roughly half of the approximate contour generator will lie on back-faces, and thus be occluded by some nearby triangles of the mesh. There are several heuristics one may use to combat this effect, such as voting with multiple ray-tests [Hertzmann and Zorin 2000], and ignoring occlusions from triangles adjacent to the contour's face [Grabli et al. 2010], but these heuristics are not robust (Figure 5(b)). Subdividing the surface to a higher resolution does not solve these problems, since, under fairly general assumptions, each face containing contour generator has 50% probability of being backfacing, and increasing the surface resolution does not change this.

The mismatch between the contour and the surface representations causes other problems. Surface curves that lie near the contour will have unreliable visibility (Figure 5). Curves may also be occluded by the surface but lie outside the contour, creating a tiny halo of invisibility around the surface (Figure 6). As pointed out by Eisemann et al. [2008], the mismatch also interferes with stylization of the interiors of objects (Section 2.3). As illustrated in Figure 1(c), these issues cause errors in the results.

Visibility propagation problems. Instead of performing ray tests at the contour, one may perform them elsewhere on the surface, and then propagate visibility based on image-space relationships between curves. Elber and Cohen [1990] perform ray tests on isoparametric curves, and assume that every visible contour is connected to some visible isoparametric curve, which is not true for complex models. More generally, one might propagate Quantitative Invisibility (QI) [Appel 1967; Markosian et al. 1997] along arbitrary curves to determine visibility. The QI of a point is defined as the number of surfaces that occlude the point; a point is visible if and only if it has a QI of zero. However, QI propagation depends on computing image-space intersections between curves. Robustly computing image-space intersections between smooth contours would be very difficult, and we are not aware of published methods for doing so. A single missed or spurious intersection can ruin many visibility computations.

2.3 Planar Map Methods

Another approach is to compute the contours of a Planar Map for the scene, thereby ensuring closed contours. The Planar Map is a data structure that represents the mapping from every point in the image to points in the scene. Since it often also represents the topology of the projections of scene objects, it generalizes the graph of visible contours. Winkenbach and Salesin [1996] compute the Pla-

nar Map of a polyhedral surface approximation. Because they numerically refine the contours, their contours might not match the edges of the Planar Map. They discretize all points to an image-space lattice, which may mitigate some of the mismatch. Karsch and Hart [2011] perform mesh segmentation using a snake evolution strategy, though using a number of heuristics for snake initialization, evolution, and handling of smooth contours. Both methods demonstrate good results for static renderings of simple surfaces. However, the Planar Map computed using any piecewise-constant contour approximation will result in many small tiny regions in the vicinity of the contour. These regions can be pruned by heuristics, but these heuristics will be problematic on more complex surfaces that exhibit many small, nearby contours.

3. OVERVIEW

Given a smooth surface $\mathcal{S}$ viewed from a camera position $\mathbf{c}$, our goal is to render the contours of the surface in a manner that is both topologically and geometrically accurate. Our approach is to generate a new mesh $\mathcal{M}$ by tessellating $\mathcal{S}$, *such that the contour generators of* $\mathcal{M}$ *are topologically equivalent to those of* $\mathcal{S}$. The mesh contour can then be rendered precisely. For example, given the smooth surface in Figure 7(a), we generate the mesh in Figure 7(e), with the contours in Figure 7(f). A new mesh is computed for each camera view.

Topological equivalence is a global property, which makes it difficult to enforce directly. The core of our approach is **Contour-Consistency,** a way to assess topological correctness based on local correspondences (Section 4). Contour Consistency requires that every triangle orientation (front- or back-facing) match the orientation of the corresponding region of the smooth surface. This correspondence can only be exactly defined up to the triangle sampling resolution, which, we show, causes the omission only of tiny contour loops.

Obtaining a consistent mesh $\mathcal{M}$ is not trivial. For example, we show that one can sample a surface to arbitrarily-fine resolution and still have inconsistency (Section 5). Based on this analysis, we define a **Radial Triangle**, a type of triangle at the contour that is guaranteed to be consistent. We find that consistency is most difficult to attain near the contour, and thus we focus most of our effort in this region.

Given a smooth surface $\mathcal{S}$, we describe an algorithm that attempts to compute a Contour-Consistent mesh $\mathcal{M}$ (Section 6). The method begins with a triangle mesh that approximates the smooth surface. Mesh edges are introduced that approximate the contour of the smooth surface. Radial triangles are then introduced along the contour, and heuristics are used to remove any remaining inconsistency. Each of these steps is implemented by incremental, local operators that are much simpler to implement and understand than global remeshing procedures. The resulting mesh does not always achieve perfect consistency, but the inconsistent regions are very small, and tagged as such.

Given the mesh $\mathcal{M}$, the visible contours are then computed (Section 8), and we describe modifications to existing methods in order to handle intersections between surfaces, and numerical errors that can arise in contour visibility. We find that undesirable tiny loops and junctions often occur in the true contours, and describe topological simplification operators for the contour (Section 9). Results of our method are described in Section 10. The main steps of the process are illustrated in Figure 7.

Input surface representations. In this paper, we apply our approach to smooth surfaces. We use Catmull-Clark subdivision surfaces [1978] as our smooth surface representation, but the approach is applicable to any piecewise-smooth surface.

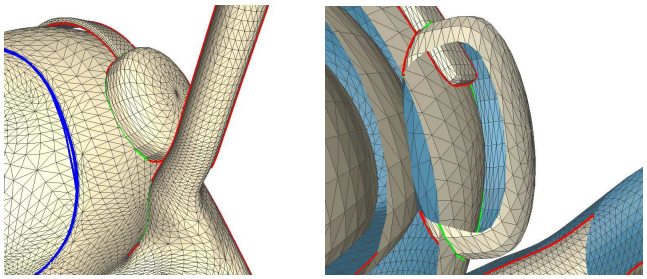


Fig. 8. **Surface intersection curves (green) are common in 3D animation.** This character's earmuff passes through his head; the earmuff intersects the strap; the arms intersect the hoodie; the skates intersect the feet; the lips intersect each other (Figure 1). Correctly handling surface intersections is crucial for accurate rendering of production geometry. Red © Disney/Pixar

We assume that the input smooth surfaces are oriented so that back-faces are never visible from the given viewpoint, a property that is typically true of 3D models used in animated film production. For example, closed surfaces satisfy this condition. Most surfaces we use are not closed, and the camera is never positioned to see the gaps. We also assume that all inputs are in general position. (If necessary, tiny random perturbations can be used to put any surface into general position, though this was not done for any of the models in this paper.)

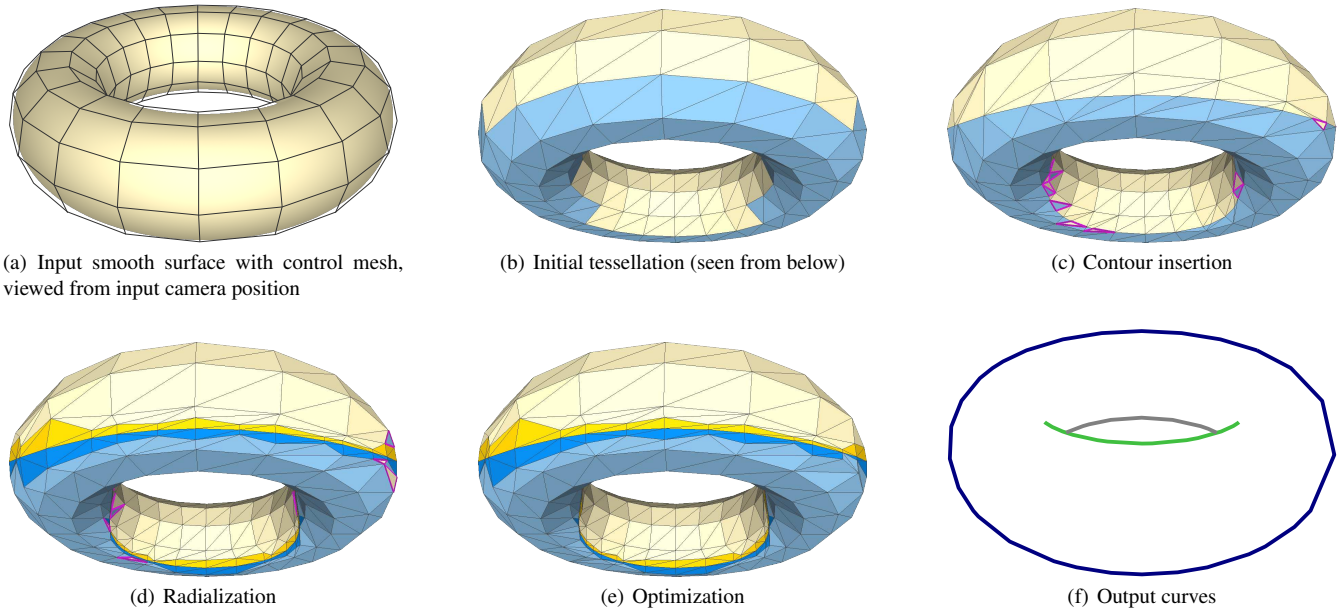
The smooth surfaces we handle are typical of high-quality smooth geometry used in animated film production. These surfaces often exhibit significant 3D intersections between different parts of the surface. We find that surface intersections are common in production models. Unless special care is taken, they occur just about any time that one object is attached to another. For example, Figure 8 shows some of the intersections that occur in a professionally-modeled character: the earmuff intersects the hood and the strap. In many cases, these curves must be connected to visible contours, e.g., for this model, the visible outline of the earmuff consists of both contour and surface intersection curves. Additionally, objects often intersect each other simply as a by-product of animation, e.g., the arm intersects the hood in this example. Consequently, handling surface intersections is crucial to the goal of handling production animation.

4. A THEORY OF CONTOUR CONSISTENCY

This section describes the theoretical core of our approach: a way to assess topological correctness of the orientation of a mesh $\mathcal{M}$ that is in correspondence with an input surface $\mathcal{S}$. This section does not consider geometric accuracy (i.e., that the position of the computed contour should be close to the true contour), which is evaluated separately from topology, as described in Section 6.

Let $\mathcal{S}$ be a surface viewed with camera center $\mathbf{c}$. Let $\mathbf{p}$ be a surface point with normal $\mathbf{n}$, and view vector $\mathbf{v} = \mathbf{c} - \mathbf{p}$. A point is front-facing if $\mathbf{n} \cdot \mathbf{v} > 0$, and back-facing if $\mathbf{n} \cdot \mathbf{v} < 0$. We assume that the surface $\mathcal{S}$ has outward-facing normals, so that back-facing points are never visible (Figure 3).

Our goal is to accurately compute the *contour* of the surface: the set of curves that separate visible regions on the surface from invisible regions. More precisely, the *contour generator* is the set of 3D points that separates front-facing regions from back-facing regions,



(a) Input smooth surface with control mesh, viewed from input camera position

(b) Initial tessellation (seen from below)

(c) Contour insertion

(d) Radialization

(e) Optimization

(f) Output curves

Fig. 7. **Steps of our algorithm, for a subdivision surface. Top:** An initial tessellation is created. All faces that cross the contour are inconsistent. Edges are inserted along the contour. Some of the resulting faces are inconsistent (outlined in magenta). **Bottom:** Strips of radial triangles (yellow/dark blue) are created on each side of the contour. Finally, optimization is applied to eliminate inconsistency, and the contour can then be directly rendered. We also apply topological simplification to the contour, but it is not needed in this case.

and the contour is the visible subset of the contour generator. For a smooth surface, the contour generator comprises the points with $\mathbf{n}\cdot\mathbf{v} = 0$, and, for a triangle mesh, the contour generator comprises the edges separating back-faces from front-faces.

We use the labels F, B, and C to denote points that are front-facing, back-facing, or contour, respectively. A triangle may be F or B. (Because we have assumed general position, there are no faces parallel to their view direction.)

Let $\mathcal{P}$ be the base mesh of the surface. The base mesh provides a piecewise-parameterization of the smooth surface: for any given base mesh point $\mathbf{u} \in \mathcal{P}$, there is a corresponding point $\mathbf{f}(\mathbf{u})$ on the smooth (limit) surface, and a corresponding limit normal $\mathbf{n}(\mathbf{u})$ (Figure 9). The point $\mathbf{u}$ is called the preimage of $\mathbf{f}(\mathbf{u})$. We define the *orientation function* as

$$g(\mathbf{u}) = (\mathbf{c} - \mathbf{f}(\mathbf{u})) \cdot \mathbf{n}(\mathbf{u}) \tag{1}$$

The zero set of $g(\mathbf{u})$ is the preimage of the contour generator of $\mathcal{S}$. (Due to limitations of our Catmull-Clark library, computation of the orientation function in the neighbourhood of extraordinary vertices required special treatment, as detailed in Appendix A.)

We define the *orientation* $R_{\mathcal{S}}(\mathbf{u})$ of a point $\mathbf{u}$ as F, B, or C depending on whether $g(\mathbf{u}) > 0, g(\mathbf{u}) < 0$, or $g(\mathbf{u}) = 0$, respectively. Hence, the contour generator may also be written as the set $\{\mathbf{f}(\mathbf{u}) : R_{\mathcal{S}}(\mathbf{u}) = \mathsf{C}\}$.

When generating a new mesh $\mathcal{M}$, each new vertex i will be assigned to some point $\mathbf{u}_i$ on the base mesh. Linear interpolation of these assignments defines a mapping from $\mathcal{M}$ to $\mathcal{P}$, and we require that the mapping be continuous and bijective.

Hence, every point on the output triangle mesh has a preimage $\mathbf{u}$, and thus corresponds to a point $\mathbf{f}(\mathbf{u})$ of the smooth surface, with orientation $R_{\mathcal{S}}(\mathbf{u}) \in \{\mathsf{F}, \mathsf{B}, \mathsf{C}\}$. Likewise, we can compute the orientation $R_{\mathcal{M}}(\mathbf{u})$ of the point on the triangle mesh: $\{\mathsf{F}, \mathsf{B}\}$ for points inside or adjacent to front- or back-facing triangles, respectively, and C for points on contour edges.

The idea of our approach is to ensure that these two orientations "agree" as much as possible: that is, we would like to choose the mesh $\mathcal{M}$ so that the orientations match ($R_{\mathcal{M}}(\mathbf{u}) = R_{\mathcal{S}}(\mathbf{u})$) everywhere. For general smooth surfaces, the contour generator is not piecewise-linear, and so we measure this equivalence only to triangle sampling resolution. We decompose the analysis to consider individual triangles. Each triangle on $\mathcal{M}$ corresponds to a patch on $\mathcal{S}$. For example, if all three vertices correspond to front-facing points on the smooth surface, then we would want the triangle to be front-facing as well. As a short-hand, the triangle orientations are written as FFF in this case. Conversely, a triangle edge where both vertices correspond to contour points (CC) should be adjacent to one front-face and one back-face.

These ideas are formalized as follows. We first define the Vertex-Based Orientation of a triangle in $\mathcal{M}$, which indicates the orientation of the corresponding patch of the surface $\mathcal{S}$, up to sampling resolution.

DEFINITION 1. *The **Vertex-Based Orientation (VBO)** of a triangle is:*

— F *if the face's vertices correspond to points on the smooth surface oriented as* FFF, CFF, CCF, *or a permutation of one of these (e.g.,* CFC*);*

— B *if the face's vertices are* BBB, CBB, CCB, *or a permutation of one of these, and*

— *undefined if the face's vertices are* CCC, CFB, FBB, FFB, *or a permutation of one of these.*

Then, we can evaluate whether the orientation of the face matches the corresponding patch on the smooth surface.

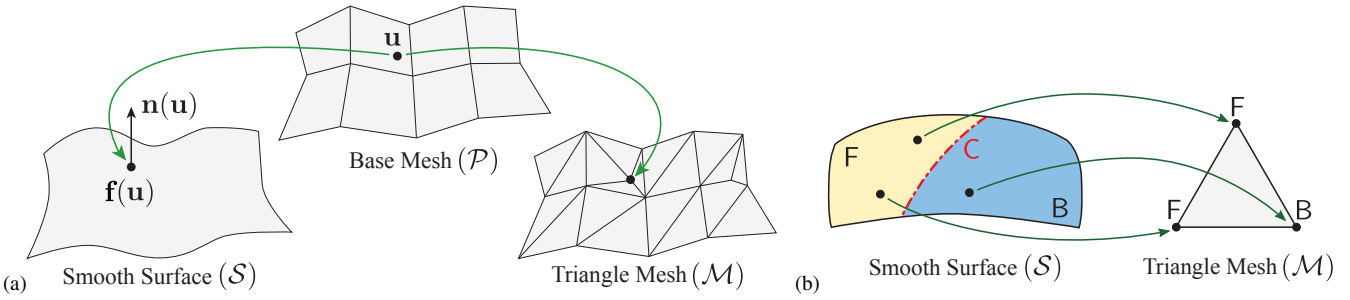


Fig. 9. **Surface parameterization**, for a subdivision surface. **(a)** Each preimage point $\mathbf{u}$ on the base mesh $\mathcal{P}$ maps to a point $\mathbf{f}(\mathbf{u})$ on the smooth surface $\mathcal{S}$, with normal $\mathbf{n}(\mathbf{u})$. Each vertex on our output triangle mesh $\mathcal{M}$ also corresponds to a point on the base mesh. **(b)** Each point on the smooth surface is labeled according to facing direction: front-facing (F), back-facing (B), or contour (C). Each vertex on the triangle mesh inherits its label from the corresponding smooth surface point. The triangle shown here cannot be consistent, because its edges cross the smooth contours. Hence, its VBO is undefined.

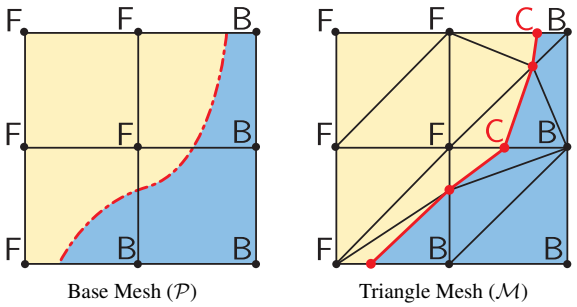


Fig. 10. **Consistent Tessellation. Left:** Base Mesh, with colors indicating the orientation function $R_{\mathcal{S}}(\mathbf{u})$ of the smooth surface. **Right:** A Triangle Mesh tessellation of this surface, in which all triangles are consistent. Colors indicate per-face orientations ($R_{\mathcal{M}}(\mathbf{u})$).

DEFINITION 2. *A face j in $\mathcal{M}$ is **Contour-Consistent** if and only if its VBO is defined, and equal to the face's orientation.*

That is, a face is consistent if the face is front-facing and the VBO is F, or the face is back-facing and the VBO is B.

The idea is illustrated in Figure 10: front-facing regions in the input surface correspond to front-facing triangles, and the C points lie on the contours of both the input surface and the triangle mesh. Moreover, no spurious contours can occur on the mesh. We call a triangle inconsistent if is not contour-consistent.

Contour-Consistency does not ensure topological correctness, because there may be zero-crossings on the surface that are missing from the mesh. We address this by adding stronger constraints. We first introduce the notion of degeneracy for mesh edges:

DEFINITION 3. *An edge in $\mathcal{M}$ is **degenerate** if and only if it has two contour vertices (CC), and the edge is shared by two triangles with the same VBO.*

In other words, a CC edge is degenerate if it is shared by two CCF triangles or two CCB triangles (Figure 11). It is not possible for a mesh to be both consistent and to exhibit a mesh contour at a degenerate edge. (In particular, for a degenerate edge, the two triangles adjacent to the edge have the same VBO, and, thus, by consistency, they must have the same orientation, either both front-facing or both back-facing. Hence, the edge cannot be a mesh contour.)

DEFINITION 4. *A face in $\mathcal{M}$ is **Strongly Contour-Consistent** if (a) it is contour-consistent, and (b) within every edge that is incident on an F or B vertex, there are no zero-crossings of g.*

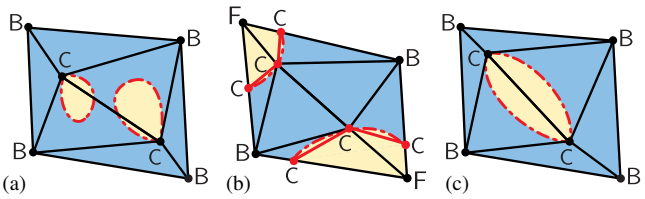


Fig. 11. **Degenerate edges.** Three examples of degenerate edges: each edge has two contour vertices CC, but the opposite vertices of the two adjacent triangles have the same orientation (in this case, B).

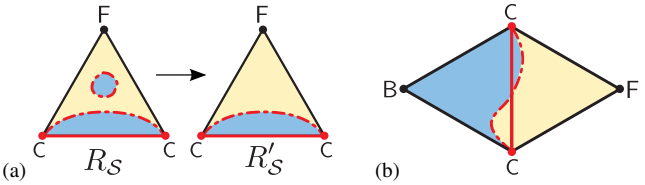


Fig. 12. **Strong Consistency. (a)** Simplification removes regions that do not intersect mesh edges. **(b)** For Strong Consistency, the true contour does not necessarily coincide with the CC edge, but cannot intersect the FC or BC edges. Figure 19 shows a case where it does.

In order to establish the significance of Strong Contour-Consistency, we first define the Simplified Orientation $R'_{\mathcal{S}}(\mathbf{u})$. The Simplified Orientation removes all contour regions that do not intersect some mesh edge, as illustrated in Figure 12(a). The contour set $R'_{\mathcal{S}}(\mathbf{u}) = \mathsf{C}$ (Simplified Contour) is a subset of the original contour set $R(\mathbf{u}) = \mathsf{C}$, with tiny loops eliminated.

Then, we can show that Strong Contour Consistency ensures topological equivalence (ambient isotopy) between the contours of $\mathcal{M}$ and those of the Simplified Orientations of the original surface:

THEOREM 1. *Let $\mathcal{S}$ be a smooth surface, viewed from viewpoint $\mathbf{c}$, and let $\mathcal{M}$ be a mesh in which every triangle is Strongly Contour-Consistent with respect to $\mathcal{S}$, and no edges are degenerate. The orientations of the mesh faces form a partition of the surface into front-facing and back-facing regions that is ambient isotopic to the Simplified Orientation $R'_{\mathcal{S}}(\mathbf{u})$ of $\mathcal{S}$.*

PROOF. To prove topological equivalence, it is sufficient to show that we can smoothly deform the orientations of $\mathcal{M}$ to those of $\mathcal{S}$, specifically, using a smooth bijection $\mathbf{u}' = \mu(\mathbf{u})$, so that

$R'_{\mathcal{S}}(\mu(\mathbf{u})) = R_{\mathcal{M}}(\mathbf{u})$. First, we fix the map to be the identity ($\mathbf{u} = \mu(\mathbf{u})$) at all mesh vertices and over all triangles that are not adjacent to mesh contours (FFF, FFC, BBB, BBC). Strong Consistency implies that $R'_{\mathcal{S}}(\mu(\mathbf{u})) = R_{\mathcal{M}}(\mathbf{u})$, because the orientations must match at the vertices, and zero-crossings cannot occur within the edges; zero-crossings cannot occur within these triangles in the Simplified Orientations.

Finally, consider a pair of triangles sharing a CC edge, illustrated in Figure 12(b). The smooth contour $R'_{\mathcal{S}}(\mathbf{u}) = \mathsf{C}$ must pass through the two C vertices, and cannot exit the triangle through any other edges. Non-degeneracy implies that one of the faces is front-facing, and the other is back-facing, and consistency implies that these orientations match those of the smooth surface. Under the generic position assumption, the smooth contour must be a simple curve. Hence, the topology of the front/back regions of the two surfaces are equivalent, and one can define μ in these faces as an arbitrary smooth mapping that maps from contour to contour, and reduces to the identity at the non-contour edges. Hence, we have defined μ over the entire surface to satisfy $R'_{\mathcal{S}}(\mu(\mathbf{u})) = R_{\mathcal{M}}(\mathbf{u})$. □

The power of contour-consistency is that it takes a global, topological goal and converts it to a local, per-triangle property. Furthermore, while the algorithms we describe later in the paper do not ensure consistency everywhere, consistency tells us which triangles are reliable, and so we can normally achieve topologically-correct results even when the mesh approximation is not perfectly consistent.

5. RADIAL TRIANGLES

So far, we have described conditions for consistency, but not how to obtain a consistent surface. It might seem that sampling a smooth surface densely and then inserting contour edges should ensure consistent triangles. Unfortunately, this is not the case.

To understand how triangles can become inconsistent for smooth surfaces, consider the contour of an elliptical surface region. We place two mesh vertices $\mathbf{p}$ and $\mathbf{q}$ on the contour, and a third, $\mathbf{r}$, on the front-facing part of the surface. Figure 13 shows two possible placements for $\mathbf{r}$, one of which is consistent, and one is not. In this example, if $\mathbf{r}$ is placed anywhere on the left-hand side of the line $\mathbf{pq}$ in image-space, the face is inconsistent. Naively refining the surface to a finer scale does not fix this problem: for a triangle where $\mathbf{p}$ and $\mathbf{q}$ are closer together (while still on the contour), $\mathbf{r}$ may still be to the left of these points. Subdivision could refine for arbitrarily-many steps without resolving the inconsistency. One can also attempt a range of heuristics for improving consistency at the contour, but we were unable to find a robust set of heuristics, particularly near cusps and other regions of low radial curvature.

On the other hand, triangles far from the contour are unlikely to be inconsistent, as the surface becomes more fronto-parallel. This intuition is borne out by our experiments. Hence, we rely more on simple search procedures for these cases.

Based on this observation, we propose a strategy for ensuring consistent triangles along the contour. Intuitively, if we can place $\mathbf{r}$ along an image-space ray that is perpendicular to the image-space tangent to the contour at $\mathbf{p}$, then it is very unlikely that $\mathbf{r}$ will project between the smooth contour and the line $\mathbf{pq}$. We can identify this point in 3D as follows (Figure 14). Let the view vector at $\mathbf{p}$ be $\mathbf{v} = (\mathbf{c} - \mathbf{p})$, and let $\mathbf{n_p}$ be the surface normal at $\mathbf{p}$. Let the radial plane [DeCarlo et al. 2003] at $\mathbf{p}$ be the plane through $\mathbf{p}$ with normal $\mathbf{n_p} \times \mathbf{v}$. Then, we want $\mathbf{r}$ to lie on the radial plane. Equivalently, we want to place $\mathbf{r}$ on the radial curve, which is the intersection of the radial plane with the surface. The third vertex $\mathbf{q}$ can be anywhere nearby, except on the radial plane of $\mathbf{p}$.

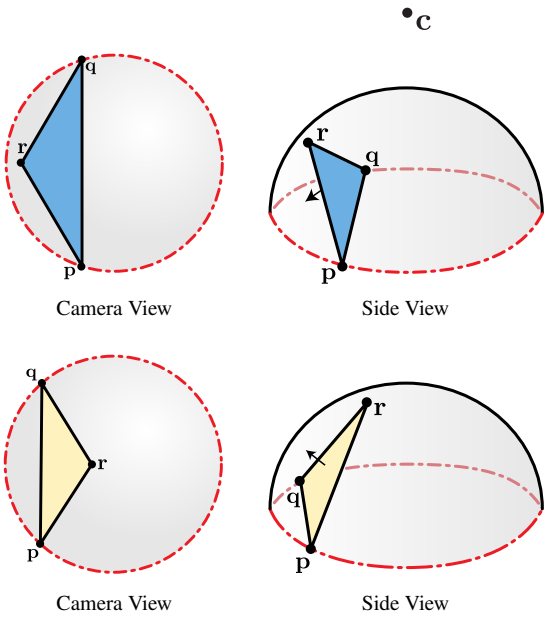


Fig. 13. **Understanding how faces become inconsistent.** Consider a small portion of the contour (red) at an elliptical region, viewed from a camera position $\mathbf{c}$. Here, only a visible portion of the surface is shown. A triangle $\mathbf{pqr}$ is placed so that $\mathbf{p}$ and $\mathbf{q}$ lie on the contour. The triangle must be front-facing in order to be consistent. For a given $\mathbf{p}$ and $\mathbf{q}$, the placement of $\mathbf{r}$ determines whether the face is front-facing or back-facing: specifically, which side of the line $\mathbf{pq}$ the point $\mathbf{r}$ projects to in image space.

DEFINITION 5. *A **Radial Triangle** is a triangle with one vertex $\mathbf{p}$ on the contour, and another vertex $\mathbf{r}$ on the radial plane of $\mathbf{p}$. The edge $\overline{\mathbf{pr}}$ is called a **Radial Edge**.*

The key observation is that, as long as a radial triangle does not cross contours or folds in the surface, it is guaranteed to be consistent.

THEOREM 2. *Let $\mathbf{p}, \mathbf{q}, \mathbf{r}$ be distinct points on an orientable smooth surface, viewed from a camera position $\mathbf{c}$, such that $\mathbf{p}$ is on the contour, and $\mathbf{r}$ is on the radial plane of $\mathbf{p}$ (Figure 14). Assume the following.*

—The triangle has a well-defined Vertex-Based Orientation (VBO).

—The triangle vertices are enumerated clockwise in the base domain (e.g., base mesh).

—There is no contour on the radial curve between $\mathbf{p}$ and $\mathbf{r}$.

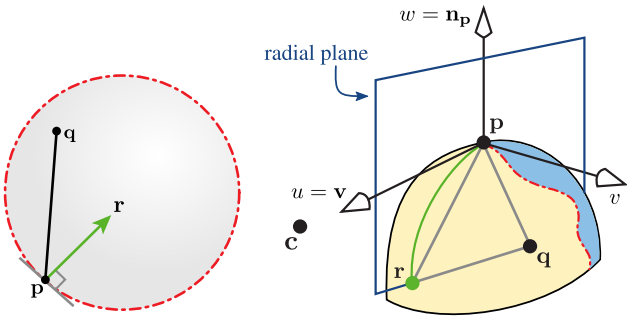
—The surface can be represented as a graph (Monge patch) around the radial curve, and on some curve from $\mathbf{p}$ to $\mathbf{q}$.

—The points $\mathbf{r}$ and $\mathbf{p}$ are on the same side of the camera center, when projected onto the tangent plane at $\mathbf{p}$.

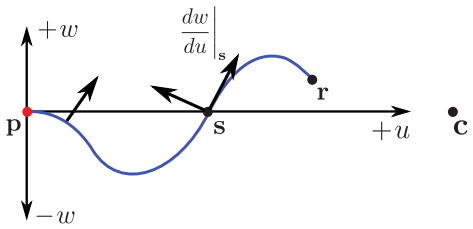
—The three points are all on the same side of the image plane.

Then, the triangle $\mathbf{pqr}$ is consistent with respect to the smooth surface.

PROOF. There are multiple cases to consider. For clarity, we will prove the theorem for the following case first: $\mathbf{r}$ is front-facing, $\mathbf{r}$ is nearer to the camera than $\mathbf{p}$, and the triangle $\mathbf{pqr}$ is clockwise in the domain. The other cases will be handled afterward.



(a) Camera View (b) Monge patch in side view



(c) Radial curve

Fig. 14. **Radial Triangles. (a)** In image space, the radial direction is perpendicular to the contour. Placing the point $\mathbf{r}$ in the radial direction from $\mathbf{p}$ makes consistency very likely. **(b)** Visualization of a radial triangle on the Monge patch (see text for details). **(c)** Radial curve for proof by contradiction that $r_w < 0$ in Theorem 2.

Given the clockwise orientation, the face normal is $\mathbf{n}_F = (\mathbf{r} - \mathbf{p}) \times (\mathbf{q} - \mathbf{p})$. Since we assume the orientation of $\mathbf{r}$ is front-facing (F), the triangle must be front-facing to be consistent.

We describe the surface $\mathcal{S}$ locally in a (u, v, w) coordinate system. The origin of the coordinate system is at $\mathbf{p}$, the u-axis is aligned to the view direction, and the w-axis is aligned to the surface normal. In these coordinates, we have $\mathbf{p} = (0, 0, 0), \mathbf{c} = (c_u, 0, 0)$, with $c_u > 0$. The radial plane is given by $v = 0$, and so $\mathbf{r} = (r_u, 0, r_w)$, with $r_u > 0$. The triangle orientation and the Monge patch assumptions imply that $\mathbf{q} = (q_u, q_v, q_w)$ with $q_v > 0$.

We can then write the orientation of $\mathbf{pqr}$ as:

$$\begin{aligned} \mathbf{n}_F \cdot (\mathbf{c} - \mathbf{p}) &= \det(\mathbf{r} - \mathbf{p},\ \mathbf{q} - \mathbf{p},\ \mathbf{c} - \mathbf{p}) \\ &= -r_w q_v c_u \end{aligned} \tag{2}$$

Since $q_v, c_u > 0$, it remains to determine the sign of r_w, which corresponds to whether $\mathbf{r}$ is above or below the tangent plane at $\mathbf{p}$. We will use proof by contradiction to show that $r_w < 0$: if $\mathbf{r}$ is above the tangent plane, then there must be a contour point on the radial curve segment between $\mathbf{p}$ and $\mathbf{r}$, which violates the assumptions.

Consider the radial curve, as illustrated in Figure 14(c). We will first relate the tangent of the radial curve to surface orientation. Since the surface is locally a graph about the radial plane ($v = 0$), a surface point can be written $\mathbf{x}(u, v) = (u, v, w(u, v))$. We first show that, for any point $\mathbf{s}$ on the ray $w = 0, v = 0$, the orientation of the point is given by $\frac{dw}{du}\Big|_{\mathbf{s}}$, independent of the other tangent direction. The surface normal is given by $\frac{d\mathbf{x}}{du} \times \frac{d\mathbf{x}}{dv}\Big|_{\mathbf{s}} = (1, 0, \frac{dw}{du}\Big|_{\mathbf{s}}) \times (0, 1, \frac{dw}{dv}\Big|_{\mathbf{s}}) = (-\frac{dw}{du}\Big|_{\mathbf{s}}, -\frac{dw}{dv}\Big|_{\mathbf{s}}, 1)$, and the view vector is $\mathbf{v} = (c_u - s_u, 0, 0)$. Hence, the surface orientation at $\mathbf{s}$ is $\mathbf{n} \cdot \mathbf{v} = -(c_u - s_u)\frac{dw}{du}$. Since the points are on the same side of the camera, $c_u > s_u$. Hence, $\mathbf{s}$ is back-facing iff $\frac{dw}{du}\Big|_{\mathbf{s}} > 0$, front-facing iff $\frac{dw}{du}\Big|_{\mathbf{s}} < 0$, and contour otherwise. (Effectively, the graph assumption makes $\frac{dw}{dv}\Big|_{\mathbf{s}}$ redundant for determining the orientation, given the derivative along the ray direction.)

Since we have assumed that $\mathbf{r}$ is front-facing, and there is no contour between $\mathbf{p}$ and $\mathbf{r}$, the entire curve segment is front-facing. Since $\mathbf{r}$ is nearer to the camera than $\mathbf{p}$, the surface must bend to face the camera at $\mathbf{p}$; at a non-cusp, this corresponds to the fact that radial curvature is negative [Koenderink 1984]. Hence, $w(u, 0) < 0$ infinitesimally close to $\mathbf{p}$ in the view direction.

Finally, we show that $r_w < 0$, using proof by contradiction. Suppose it were the case that $r_w \geq 0$. Continuity of the surface implies that there must be a point $\mathbf{s}$ on the radial curve segment such that $s_w = 0$ and $\frac{dw}{du}\Big|_{\mathbf{s}} > 0$, and, thus this point is back-facing. This contradicts the assumption that there is no contour on the segment. Hence, $r_w < 0$.

Hence, by Equation 2, the triangle is front-facing, and thus consistent.

We can now consider the other cases: $\mathbf{r}$ might be a front- or back-facing point ($g(\mathbf{r}) > 0$ or $g(\mathbf{r}) < 0$), and it might be in front of or behind $\mathbf{p}$ ($r_u > 0$ or $r_u < 0$). The triangle might be clockwise as $\mathbf{pqr}$ or as $\mathbf{prq}$, and we define $a = 1$ for the former case and $a = -1$ for the latter. The normal of the triangle is then $\mathbf{n}_F = a(\mathbf{r} - \mathbf{p}) \times (\mathbf{q} - \mathbf{p})$. If we have $r_u < 0$ and the triangle is $\mathbf{pqr}$, then $q_v < 0$. More generally, we can write: $\operatorname{sgn}(q_v) = \operatorname{sgn}(a r_u)$. The value of r_w is also affected by these cases. By applying the same radial curve reasoning for r_w as above, we find that $\mathbf{r}$ is above the tangent plane when $r_u > 0$ and $g(\mathbf{r}) < 0$ or $r_u < 0$ and $g(\mathbf{r}) > 0$. This can be written more concisely as: $\operatorname{sgn}(r_w) = -\operatorname{sgn}(r_u g(\mathbf{r}))$. Then, using Equation 2:

$$\operatorname{sgn}(\mathbf{n}_F \cdot \mathbf{v}) = -\operatorname{sgn}(a r_w q_v) = \operatorname{sgn}(a^2 r_u^2 g(\mathbf{r})) = \operatorname{sgn}(g(\mathbf{r})) \tag{3}$$

Thus, the triangle is front-facing iff the point $\mathbf{r}$ is front-facing, and thus the triangle is consistent. □

The assumption that the faces are clockwise in the base domain arises from the assumption that the surface is orientable, and all triangles should have an orientation consistent with the surface. The same proof steps could also be applied for a surface oriented with all triangles counter-clockwise.

It should be possible to generalize the proof to the case where $\mathbf{p}$ is not a contour point, by aligning the u-axis to the view direction, and keeping the w-axis within the radial plane but normal to the view vector. However, the more fronto-parallel the surface is, the less likely the graph assumption is to hold in this new coordinate system, for any real sampling of points.

There are other assumptions one can make to achieve consistency. The most basic requirements are that $\mathbf{r}$ lies on the correct side of $\mathbf{p}$'s tangent plane, and $\mathbf{q}$ lies on the correct side of the radial plane. These two conditions could also be checked in implementation to determine which edge to subdivide in an inconsistent radial triangle. (We do not use this in our algorithm, since we have never observed it to produce inconsistent radial triangles.)

It is straightforward to see that some consistent radial triangle may always be constructed around any contour point $\mathbf{p}$, provided that one chooses $\mathbf{r}$ and $\mathbf{q}$ close enough to $\mathbf{p}$. For example, given a candidate triangle at $\mathbf{p}$ that violates the assumptions, one could move $\mathbf{r}$ toward $\mathbf{p}$ on the surface until it lies on the correct side of the tangent plane, and then move $\mathbf{q}$ toward $\mathbf{p}$ until it lies on the correct side of the radial plane while also staying on the correctly-oriented region of the surface. More formally, one can show that, for a given point $\mathbf{p}$, there exists a finite distance d such that moving $\mathbf{q}$ and $\mathbf{r}$ to within this distance guarantees that the triangle is radial.

6. MESH GENERATION ALGORITHM

We now describe an algorithm that takes a smooth surface $\mathcal{S}$ and a viewpoint $\mathbf{c}$, and generates a triangle mesh $\mathcal{M}$ tessellation of the surface. This tessellation includes edges corresponding to the smooth contour, and radial triangles adjacent to these contour edges. The algorithm is designed to produce Strongly-Contour Consistent meshes almost everywhere. We found it not necessary to strictly enforce it, and we discuss these decisions further in Section 7.

The algorithm operates in sequence of processing phases, illustrated in Figures 7 and 15:

(1) **Initialization:** An initial triangle mesh is created, in correspondence with the smooth surface.
(2) **Contour Insertion:** Contour edges and cusp vertices are inserted into the mesh. After this phase, all triangles have well-defined VBOs, and each cusp lies on a vertex with a particular topology (described below).
(3) **Radialization:** The triangles along the contour are converted into radial triangles.
(4) **Optimization:** Additional perturbations are applied to the mesh to improve consistency.

Each of these steps is implemented as sequences of subphases, each of which iterates over the mesh, applying local transformations to a few triangles at a time, such as Edge Splits (inserting a vertex into a mesh edge), and Vertex Shifts (adjusting the preimage coordinates $\mathbf{u}$ of a vertex, along with its 3D position).

We initially devised this algorithm as global remeshing algorithm that traced a new grid with two edges aligned to the radial direction at each vertex. However, this strategy led to an explosion of devilish special cases that were very difficult to reason about or to implement. In contrast, the strategy we present here decomposes the problem into much more manageable subproblems.

6.1 Initialization

The mesh $\mathcal{M}$ is initialized by once subdividing the base mesh of the Catmull-Clark surface, in order to eliminate non-quad faces, splitting all quads into triangles (Figure 15(b)), and moving all vertices to their limit positions. Each vertex i of the mesh $\mathcal{M}$ has a corresponding preimage $\mathbf{u}_i$. (Initially, all preimages are vertices of the base mesh $\mathcal{P}$.) Consequently, each vertex may be labeled as either F or B, depending on whether the corresponding smooth surface point is front- or back-facing. The 3D location of each mesh vertex i is the limit position $\mathbf{f}(\mathbf{u}_i)$, this is true as well for new vertices created by subsequent steps. Catmull-Clark limit positions and normals are computed by Halstead et al.'s method [1993].

Next, the algorithm searches the mesh for FF and BB edges with sign-crossings of $\mathbf{n} \cdot \mathbf{v}$. In particular, $R_S(\mathbf{u})$ is sampled densely along each edge; we use 10 sample points per edge. (Recall that the correspondence between surfaces is defined by linear interpolation: the preimages along a mesh edge with vertex preimages $\mathbf{u}_0$ and $\mathbf{u}_1$ are given by $\mathbf{u}_{01} = (1-t)\mathbf{u}_0 + t\mathbf{u}_1$, according to the interpolation procedure in Appendix B.) If $R_S(\mathbf{u}_{01})$ ever differs from the values at the two endpoints, the edge is split into two FB edges by inserting a new vertex at its limit position $\mathbf{f}(\mathbf{u}_{01})$ (Figure 15(c) and 16(a)).

6.2 Contour Insertion

The Contour Insertion phase inserts contour (C) vertices in all FB edges. Following this step, all triangles have defined VBOs, and the CC edges form a piecewise-linear approximation to the smooth surface contour.

This phase also ensures that each contour cusp has a corresponding C vertex with valence 4, in the configuration shown in Figure 18(d). This post-condition is a necessary precondition for the following radialization stages, so that radialization produces cusps in the configuration of Figure 18(e). Directly inserting cusp vertices before other contour insertion steps would lead ultimately to the configuration in Figure 18(f), which would be very difficult to fix in a general way.

Cusp detection (Figure 15(d)). The algorithm first detects cusps on the smooth surface's contour. When a cusp is detected, a mesh edge will be split or shifted, so that the cusp's preimage $\mathbf{u}_{cusp}$ lies exactly on a mesh edge. However, the cusp vertex is not inserted yet (Figure 18(b)).

The algorithm checks each mesh triangle for cusps by a bisection search, as follows. A cusp occurs at any point $\mathbf{u}_{cusp}$ for which $g(\mathbf{u}) = 0$ and $\kappa(\mathbf{u}) = 0$, where $\kappa(\mathbf{u})$ is the radial curvature at a point (Figure 18(a)). The values of g and κ are evaluated at each mesh vertex. If both functions exhibit sign changes, then the triangle is bisected along the edge that is longest in parameter space. The procedure then recurses into the two new triangles. When the recursion detects a very small triangle (area below a threshold, 10^{-20} in our implementation in quad-precision) that exhibits sign crossings in both functions, the centroid $\mathbf{u}_{cusp}$ of that triangle is returned as a cusp preimage location. (Note that the above bisections do not modify the actual mesh; they are only for searching).

Next, the mesh is modified so that each cusp lies on an FB edge. The face containing the cusp must have exactly one FF or BB edge. This edge is normally split by a vertex insertion, such that the new edge contains the cusp (Figure 16(b)). It is straightforward to determine this split by a line intersection in parameter space. However, if the cusp lies very close to a triangle edge, the insertion would create a triangle with bad aspect ratio. In this case, we can shift one of the triangle vertices so that the mesh lies on the edge. There are a number of additional conditions on when vertices can be shifted, discussed in Appendix C. If the vertex is not shiftable, the original split is used.

Contour Insertion (Figures 15(e-f)). This step inserts contour vertices into the mesh, so that each face has a well-defined VBO. For each zero-crossing edge (FB) in the mesh, the contour point C is identified by finding the point location $\mathbf{u}_{root}$ that solves $g(\mathbf{u}) = 0$ along the edge. The root is found by bisection search between the endpoints and inserted as a vertex with position $\mathbf{f}(\mathbf{u}_{root})$ into the mesh, splitting the two adjacent faces (Figure 16(c)). The inserted vertex is tagged as C. This process repeats until there are no zero-crossing edges remaining. Edges containing cusps are saved for last, in order to ensure that they will have valence four (Figures 18(c-d)).

In some cases, the root may be very close to one of the vertices of the edge. In these cases, instead of inserting a new vertex, the existing vertex position is shifted to the contour, i.e., its source $\mathbf{u}$, position, and normal are set to those of the contour point, and the orientation tagged as C, provided it is shiftable (Appendix C).

6.3 Radialization

Some inconsistent triangles usually remain after contour edge insertion (Figure 7(c)). The next step is to convert the triangles along the contour into radial triangles (Section 5), in order to ensure that the region near the contour is consistent.

Radial Edge Insertion (Figure 15(g)). The algorithm first inserts radial edges into the mesh. The algorithm iterates over each face,

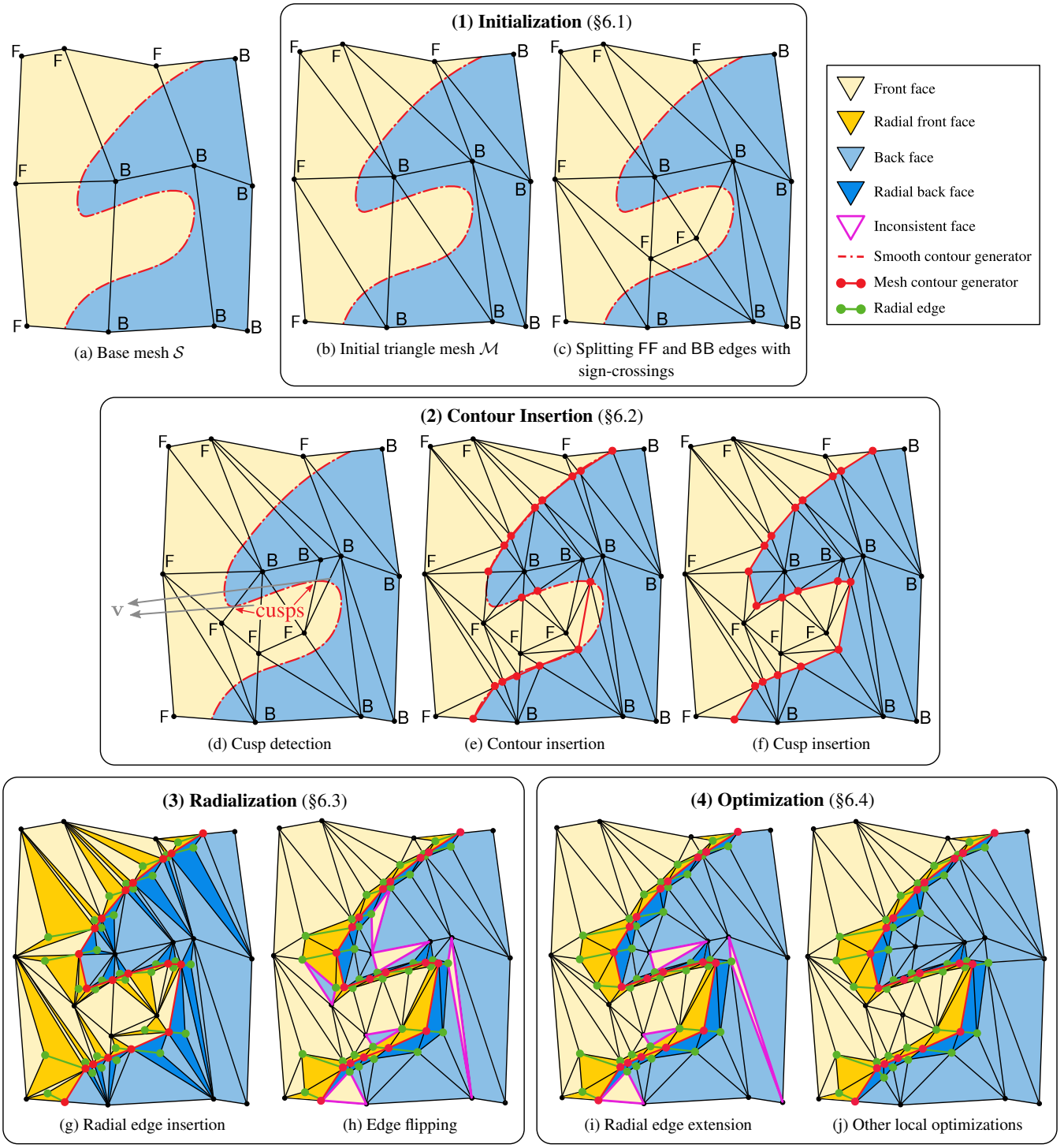
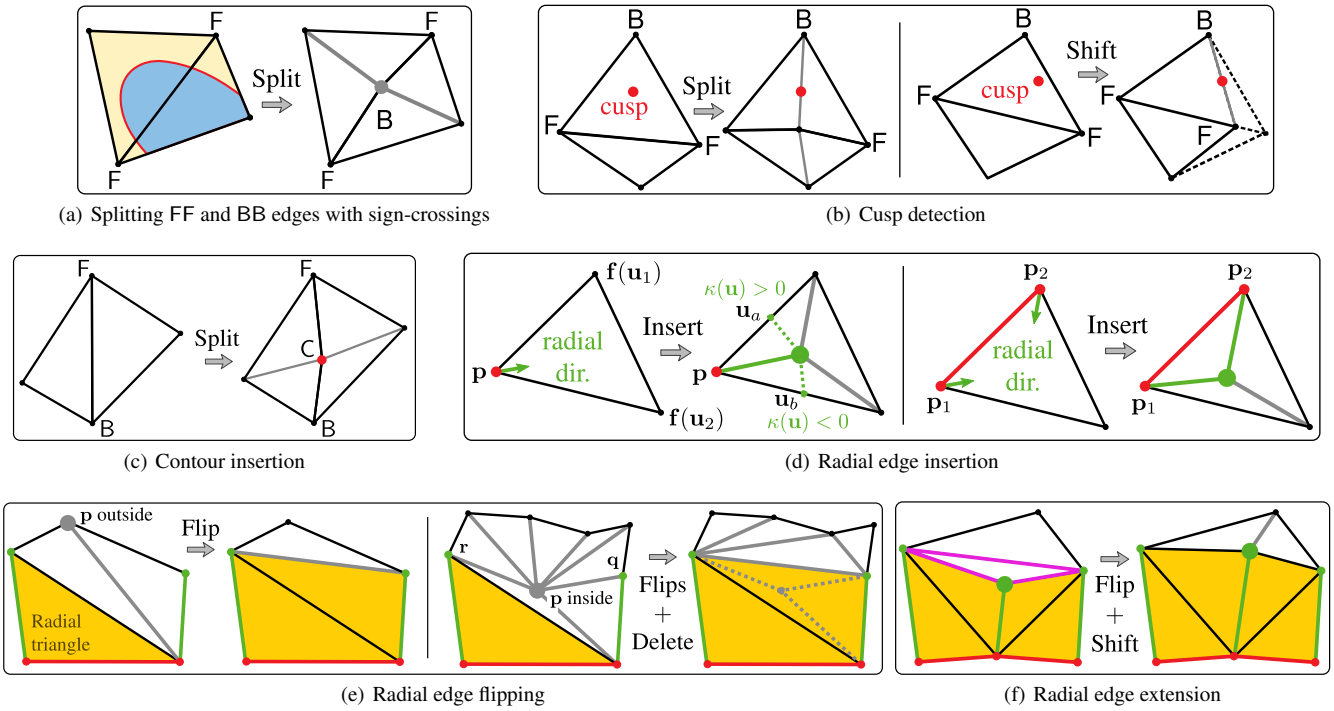


Fig. 15. **Overview of the mesh generation algorithm. (a)** Starting from the base mesh of $\mathcal{S}$, **(b)** the mesh $\mathcal{M}$ is initialized by splitting all quads into triangles and **(c)** every FF and BB edges with sign-crossings. Contours are then inserted in the mesh by 3 subphases: **(d)** cusps detection, **(e)** contour edges insertion and **(f)** cusp insertion. The radialization step first **(g)** inserts radial edges in the mesh, before **(h)** flipping edges to convert every face touching the contour into a radial triangle. Consistency of the mesh is finally improved by **(i)** extending radial edges and **(j)** local perturbations.



(a) Splitting FF and BB edges with sign-crossings

(b) Cusp detection

(c) Contour insertion

(d) Radial edge insertion

(e) Radial edge flipping

(f) Radial edge extension

Fig. 16. **Mesh generation algorithm.** The algorithm is decomposed into a set of local transformations (Edge Split, Vertex Shift, Insert and Delete) applied on a few triangles at a time. They are combined into six operations designed to produce Strongly-Contour Consistent meshes. (See text for details.)

and checks if the face has exactly one contour (C) vertex $\mathbf{p}_0$ such that the radial plane from that vertex intersects the surface through the opposite edge of the face. If so, a new vertex is inserted at a point on the radial curve near the centroid of the face; the new vertex splits the face into three (Figure 16(d), left).

In more detail: the radial plane at $\mathbf{p}$ is defined as $h_{\mathbf{p}}(\mathbf{q}) = (\mathbf{n}_{\mathbf{p}} \times (\mathbf{c}-\mathbf{p})) \cdot (\mathbf{q}-\mathbf{p}) = 0$, where $\mathbf{n}_{\mathbf{p}}$ is the limit normal at $\mathbf{p}$. The opposite edge is parameterized as $\mathbf{u}_{12}(t) = (1-t)\mathbf{u}_1 + t\mathbf{u}_2$, where $\mathbf{u}_1$ and $\mathbf{u}_2$ are the preimages of the two opposite vertices. The radial plane intersects the opposite edge surface if $\mathrm{sgn}(h_{\mathbf{p}}(\mathbf{f}(\mathbf{u}_1))) \neq \mathrm{sgn}(h_{\mathbf{p}}(\mathbf{f}(\mathbf{u}_2)))$. To determine an insertion point, we select the midpoints $\mathbf{u}_a$ and $\mathbf{u}_b$ along the triangle edges incident on $\mathbf{p}$, and perform root-finding of $h_{\mathbf{p}}(\mathbf{f}(\mathbf{u}_{ab}(t)))$ on the curve between them.

In the case where there are two contour vertices $\mathbf{p}_1$ and $\mathbf{p}_2$, such that both have radial planes in the face, and there exists a surface point in the face that intersects both radial planes, then a new vertex is inserted at this point (Figure 16(d), right). As with cusp detection, this point is found by bisection search to find a point $\mathbf{u}$ in the triangle that is simultaneously a zero of $h_{\mathbf{p}_2}(\mathbf{f}(\mathbf{u}))$ and $h_{\mathbf{p}_1}(\mathbf{f}(\mathbf{u}))$. In all other cases (e.g., the intersection point does not exist), the radial edges of the two C vertices are inserted in turn using the one-vertex technique described above.

In either case, splitting the face can create a fold in the mesh, producing an inconsistent triangle which is difficult to fix later. A potential fold is detected by projecting the new insertion point to the plane of the original face (Figure 17); a fold occurs when the projection lies outside the face. When possible, the fold is eliminated by flipping the adjacent edge of the folding triangle. If the flip is not possible (e.g., the face is opposite a surface boundary), then the radial edge insertion is skipped.

Figure 18 shows how cusps are treated by this process, and how radial triangles are ensured on the "outside" of a cusp, i.e., the portion of the surface where two radial edges emerge from the cusp (in the Figure, the front-facing region). We do not attempt to create radial triangles in the "inside" region, as it is more difficult to create radial triangles in a way that always works for these regions. In general, because these inside regions are much closer to being fronto-parallel, triangles in these inside regions are usually consistent, or easy to make consistent with heuristics, and so radialization is not necessary.

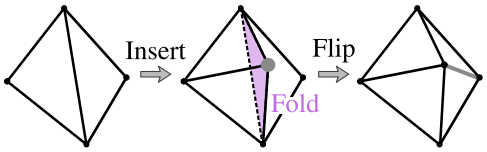


Fig. 17. **Prevention of folds.** Because vertices are inserted at their limit positions, inserting a vertex (gray) can create a fold in the surface. A fold occurs when the projection of the new vertex to the plane of the original triangle lies outside the face. Folds are fixed by flipping an edge of the adjacent face (in pink).

Edge Flipping (Figure 15(h)). Following the previous phase, every contour vertex has two radial edges. Every triangle with a contour vertex (C) and a radial edge is already a radial triangle. However, there are still non-radial triangles that touch the contour, and the goal of the Edge Flipping step is to convert these to radial triangles. For this step, we define a Standard Radial Triangle as one that either includes a contour (CC) edge, or else includes a second vertex on another radial edge. All Radial Triangles touching non-cusp contours will be Standard Radial Triangles. The algorithm iterates over all non-radial edges incident on contour vertices, i.e., all non-radial FC and BC edges. For each edge, the algorithm tests the ef-

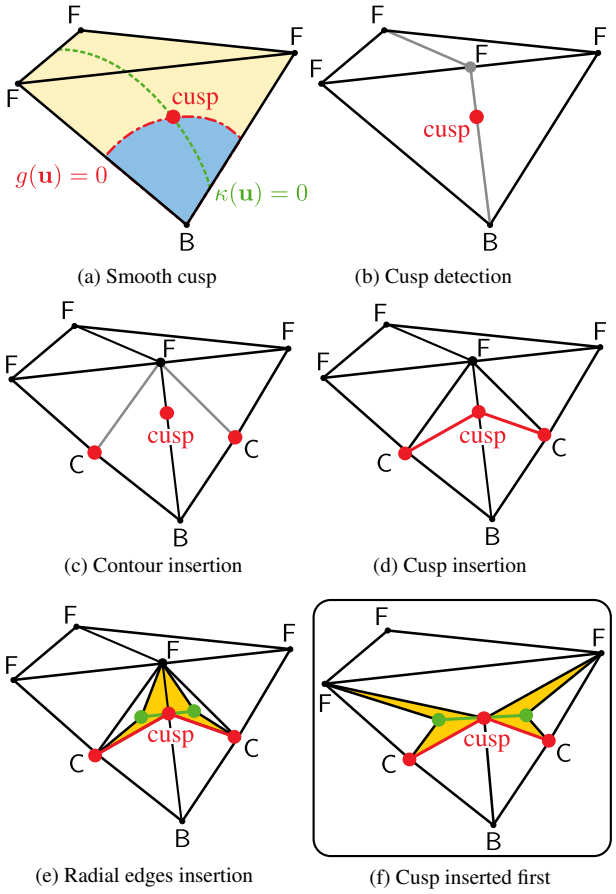


Fig. 18. **Cusp lifetime. (a)** Cusps occurs at the intersection of the two functions $g(\mathbf{u}) = 0$ and $\kappa(\mathbf{u}) = 0$. **(b)** The detected cusp is not inserted immediately because it can lead to undesirable radial triangles configurations such as **(f)**. Instead, a mesh edge is split (or shift) so that the pre-image of the cusp lies on this edge. **(c)** Contours are then inserted by splitting FB edges, but the previously created cusp edge is ignored. **(d)** The cusp is inserted last, ensuring valence 4 at this vertex. **(e)** The radial triangles produced at the radial edges insertion step are surrounding the cusp. **(f)** If we were to insert the cusp vertices before contour insertion, then this difficult configuration would arise. Here, there are non-radial triangles touching the cusp vertex, and no simple way to fix the neighborhood that will work in general.

fect of flipping the edge (Figure 16(e), left). If the flip increases the number of Standard Radial Triangles, without introducing folds, the flip is applied to the mesh. The iteration repeats until no more flips can be performed.

Flipping an edge can sometimes produce a fold. In such a case, the algorithm tries to flip all the edges incident on $\mathbf{p}$ until the vertex can be deleted from the mesh and the two adjacent faces merged (Figure 16(e), right). More specifically, the algorithm considers in order every face in the triangle fan delimited by the edges $\overline{\mathbf{pq}}$ and $\overline{\mathbf{pr}}$, with $\mathbf{q}$ and $\mathbf{r}$ the radial edge extremities adjacent to $\mathbf{p}$. For each face, the edge incident on $\mathbf{p}$ is flipped as long as it does not produce folds. If it would, the algorithm starts iterating for the other side of the fan. Once $\mathbf{p}$ is only connected to $\mathbf{q}$, $\mathbf{r}$ and the contour, this vertex is deleted creating the edge $\overline{\mathbf{qr}}$ and a Standard Radial Triangle.

6.4 Optimization

Following the above steps, the triangles along the contour will generally be consistent. Triangles away from the contour will occasionally be inconsistent, but they are usually isolated, and amenable to improvement through local perturbations. Each of the following phases iterates over the entire mesh in turn, making local improvements to the mesh.

Radial Edge Extension (Figure 15(i)). Most triangles touching the contour (i.e., with one or two C vertices) are now radial and, consequently, Contour-Consistent. Yet, some faces directly beyond the radial triangles are inconsistent, in the configuration illustrated in Figure 16(f). During this step, these radial edges are extended by shifting their non-contour endpoint and flipping their opposite edge.

The algorithm iterates over every radial edge $\overline{\mathbf{pq}}$ whose non-contour endpoint $\mathbf{q}$ touches an inconsistent face. It tests whether the radial plane $h_{\mathbf{p}}$ of the endpoint $\mathbf{p}$ intersects the surface through the opposite edge of the inconsistent face. If so, it samples the inconsistent triangle and its opposite adjacent face to find the shifted location that maximizes the number of consistent faces while avoiding folds. The process is repeated as long as the number of inconsistent triangles decreases.

Edge Flipping. The algorithm tests whether flipping any of the three edges of inconsistent faces would improve consistency. It selects (if any) the flip that both produces the best improvement and creates triangles with the best aspect ratio (using the shape quality metric of Bank and Smith [1997]). Flips that would generate tiny triangles or folds are discarded.

Vertex Wiggling in Parameter Space. The algorithm iterates over the vertices of inconsistant triangles. Each face in the vertex one-ring is regularly sampled at 100 locations in parameter space. The vertex is shifted to the new position that minimizes the number of inconsistencies and maximizes the minimum aspect ratio of the triangles in the one-ring. Folds are avoided following the previously-described approach. The process repeats iterating over all vertices until no further improvements can be made.

Edge Splitting. Wiggling large triangles cannot always resolve inconsistencies. Splitting them can give the wiggling step more opportunities to find consistent positions. Edges of inconsistent triangles are sampled at 10 locations and the best one (using previous criteria) is selected.

Vertex Wiggling in the Normal Direction. During this final step, mesh vertices are allowed to deviate from the limit surface to improve consistency. For each vertex, the algorithm searches over its normal line ($\ell(\alpha) = \mathbf{p} + \alpha \mathbf{n}_{\mathbf{p}}$) to find the new position $\mathbf{p}_i$ that minimizes the number of inconsistent faces adjacent to the triangle. In particular, the algorithm samples 100 offsets α around the point, with maximum displacement equal to the average distance to the vertices in the one-ring. If multiple offsets give the same number of inconsistent triangles, the result is chosen that minimizes $|\alpha|$.

7. MESH GENERATION ALGORITHM PROPERTIES

In this section, we discuss the properties that are partially or fully guaranteed by our method, and what additional steps would be necessary to complete the guarantees. In trying different approaches to this problem, we found that many heuristics that "ought to work" failed in difficult special cases. Consequently, our strategy was to

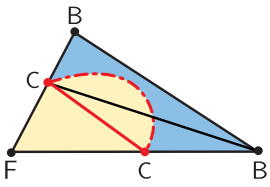


Fig. 19. **Missing zero-crossing on a CB edge.** A zero-crossing is missed by our algorithm; the CBB and CCB triangles are not Strongly Contour-Consistent. Yet, the mesh coutour (CC edge) is still topologically equivalent to the smooth contour segment (dashed curve).

develop a theory and approach that guarantee correct results, but to only implement enough to achieve good results.

The main goal of the method is topological equivalence to the true contours (Theorem 1), which requires consistency on all faces, zero-crossings on non-contour edges, and no degenerate edges.

Contour-Consistency. Because contour vertices are always inserted in FB edges and all FB edges are split (Section 6.2), the contour insertion step always creates triangles with valid VBOs (i.e., not "undefined"). It is straightforward to see that none of the subsequent steps introduce triangles with undefined VBOs (note that the shiftability rules in Appendix C explicitly prevent creation of CCC triangles).

Theorem 2 requires that the surface be C^2 along the radial curve. Catmull-Clark surfaces are guaranteed to be C^2 except at extraordinary points. Fortunately, extraordinary points do not pose a problem. Since the points $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$ are all generated by the algorithm, under the assumption of generic position neither these points nor any point on the radial curve will coincide with an extraordinary point. Thus Catmull-Clark surfaces meet the necessary continuity conditions of the theorem.

Theorem 2 assumes that the surface is locally a graph. In principle, our method could produce inconsistent radial triangles by violating these assumptions, though we do not observe this in practice. The algorithm could be modified to make this guarantee by repeatedly subdividing whichever edge exhibits a fold in the tangent plane.

Triangles away from the contour have no consistency guarantees, but one would expect that they are unlikely to be inconsistent. We find this empirically to be the case. Since only a small fraction of triangles effectively affect the final rendering, and most of those are near the contours, errors due to these inconsistencies are marginal. It might be possible to generalize our theory to non-contour triangles, which we leave for future work.

Zero-crossings. Strong Contour-Consistency (Definition 4) requires there to be no zero-crossings on non-contour-generator edges. It is possible for the initial sampling step to miss zero-crossings. Empirically, we do seem to be finding all zero crossings on the initial FF and BB edges. One could implement interval analysis [Elber and Cohen 1990; Plantinga and Vegter 2006] to find all zero crossings, since, with non-normalized $\mathbf{n}$ and $\mathbf{v}$, the function $\mathbf{n} \cdot \mathbf{v}$ is a piecewise polynomial for subdivision surfaces. This would give an algorithm theoretically guaranteed to find all contours, although the convergence rate of the interval analysis may or may not be fast enough to be practical. Additionally, one could add zero-crossing checks after every modification to an edge.

The method never checks for zero-crossings on CF and CB edges, though these zero-crossing checks could easily be added. However, these checks are generally not necessary. A typical case is illustrated in Figure 19: although a zero-crossing is missed, the contours are still topologically correct. It is possible to devise configurations where these missed zero-crossings lead to topologically-incorrect contours, but these configurations are extremely intricate and unlikely. (For example, one can devise a configuration in which

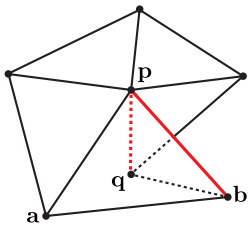


Fig. 20. **A curtain-fold cusp in the contour**. Edge $\overline{\mathbf{pb}}$ is an **exterior contour**. Edge $\overline{\mathbf{pq}}$ is an **interior contour**, indicating that it is occluded by a nearby face. Vertex $\mathbf{p}$ is thus a curtain-Fold cusp [Markosian et al. 1997]. (A cusp of the mesh boundary curve occurs in the same configuration, but with triangle $\mathbf{pqb}$ removed.)

every edge is incident on some C vertex, but the contours are not topologically correct.) Hypothetically, we might also miss tiny contour loops that run adjacent to the detected contours; such loops are unlikely to be desired, since they would hew very closely to known contours.

Non-degeneracy. The algorithm cannot introduce degenerate edges (Definition 3), since each C vertex is introduced between an F and a B vertex.

Summary. Our implementation does not guarantee Contour-Consistency, but achieves it almost everywhere. We do not guarantee finding all zero crossings, but empirically we appear to be finding all zero crossings that are necessary for topological correctness. We guarantee non-degeneracy. Consequently, we normally have high confidence in the correctness of the extracted contour generators.

8. VISIBLE CURVE COMPUTATION

Given the computed triangle mesh $\mathcal{M}$, we can then compute its contour generator and contour using standard methods (e.g., [Grabli et al. 2010]). We modify these algorithms in order to be robust to numerical error and to inconsistent faces, though inconsistent faces are rare. From this point on, the original smooth surface $\mathcal{S}$ is no longer used.

The algorithm first detects all contour, boundary, and surface intersection line segments; other types of curves could easily be handled as well. Each of these curves is represented as a 3D polyline on the surface. Contour and boundary segments lie on mesh edges, and Surface Intersections lie in the interiors of faces. Visibility is computed for individual line segments through a combination of ray tests and local occlusion tests. A **view graph** data structure is constructed that represents the network of curves, and their connections at cusps and intersections. Since visibility can be unreliable in local tests, visibility is propagated through this data structure.

The contour generator comprises the edges that connect a triangle with front-facing VBO to a triangle with back-facing VBO. For consistent triangles, this is just the contour generator of the mesh $\mathcal{M}$. Surface Intersection edges are detected by computing intersections between all pairs of triangles, using the method of Guigue and Devillers [2003], accelerated with a 3D grid data structure.

8.1 Cusps

Cusps are detected in boundary and contour curves. There are two kinds of cusps: bifurcations and curtain folds[2]. Bifurcations occur where three or more contour generator edges meet at a vertex. Note

[2]The terminology around cusps is inconsistent in the previous literature. Here we haven chosen the terms we believe are clearest. The term "curtain fold" is taken from Blinn [1978], and "bifurcation" is our name for case 2 of Definition 3 of Markosian et al. [1997].

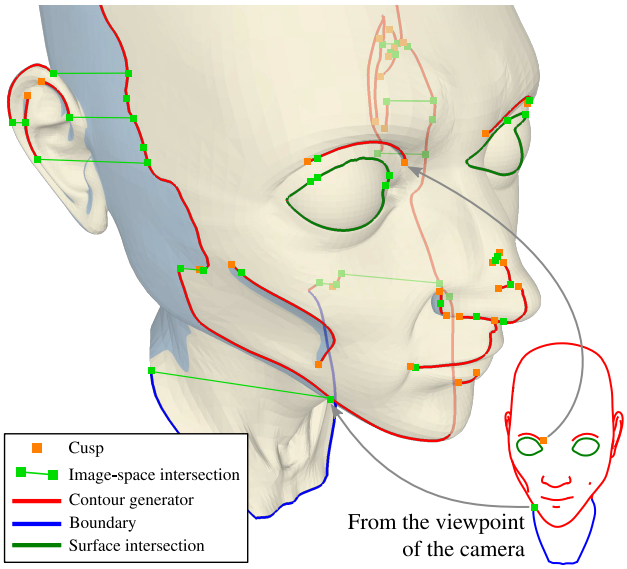


Fig. 21. **View graph** (visible chains only). Line segments (contour generators, boundaries and surface intersections) are combined into chains that terminate at cusp and image-space intersection. This network of chains is called the view graph. Angela © Chris Landreth

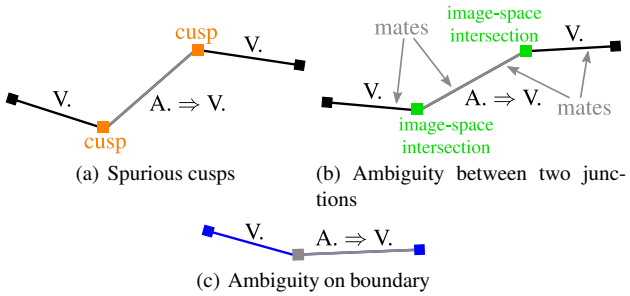
that, unlike generic mesh contours, our mesh contours cannot exhibit bifurcations, because they are topologically-equivalent to the smooth contours, which cannot have bifurcations in generic position. Thus bifurcations need not be handled.

A curtain fold is a cusp in which a single curve becomes locally self-occluded. In particular, a curtain fold occurs at a mesh vertex with two outgoing curve edges on the mesh, such that exactly one of the edges is obscured by a face in the vertex's one-ring (Figure 20). For mesh contour vertices, we use a modified version of Markosian's [1997] test. Specifically, an **exterior contour edge** is defined as one in which the near face has front-facing VBO, and an **interior contour edge** has a near face with back-facing VBO. A cusp then occurs at any vertex that connects an interior edge to an exterior edge. We detect boundary curtain-fold cusps by a procedure that tests the image-space overlap of the boundary curve by any non-adjacent triangle.[3]

One can interpret this formulation as computing a discrete analogue to the sign of radial curvature [Koenderink 1984]. On the smooth surface, the sign of κ indicates whether a contour is interior or exterior, and curtain fold cusps occur at the zero-crossings ($\kappa = 0$).

This procedure can sometimes give spurious cusps or miss cusps in inconsistent regions. However, it is guaranteed to produce at least one cusp nearby along the curve, because this procedure detects the transition from interior to exterior (analogous to a zero-crossing of κ). Hence, numerical error can shift a cusp slightly along a curve, but not eliminate it.

[3]We determine the near face as follows. Let $\mathbf{a}$ and $\mathbf{b}$ be the vertices of the edge, let $\mathbf{p}_1$ be the third vertex of adjacent face 1, and $\mathbf{p}_2$ be the third vertex of adjacent face 2. Let the unit view vector $\mathbf{v}$ be the vector to the camera center from the edge's midpoint, normalized. The view vector is projected to each face and normalized, yielding two in-face vectors $\mathbf{w}_1$ and $\mathbf{w}_2$. Each in-face vector is flipped if necessary to point from the midpoint to the face's opposite vertex. Face 1 is the near face if $\mathbf{w}_1 \cdot \mathbf{v} > \mathbf{w}_2 \cdot \mathbf{v}$. However, if $|\mathbf{v} \cdot (\mathbf{w}_1 - \mathbf{w}_2)| < 1$, we consider the test to be unreliable.



(a) Spurious cusps (b) Ambiguity between two junctions

(c) Ambiguity on boundary

Fig. 22. **Resolution of visibility ambiguities.** Three heursitics are used to resolved visibility of ambiguous chains. **(a)** An ambiguous chain connecting two visible cusps is considered visible. **(b)** An ambiguous chain connecting two image-space intersections whose mates are both visible is also visible. **(c)** An ambigous boundary chain connected to a visible chain not at an image-space intersection is marked visible.

8.2 Image-space Intersections

All image-space intersections between pairs of curve segments are detected using the sweep-line algorithm. However, detecting intersections between curves that intersect on mesh edges is not robust if done naively, and we separately handle edges that intersect on the surface. In particular, Surface Intersection curves intersect Contour edges at their endpoints, thus image-space detection is unreliable for these cases.

8.3 Chaining and View Graph

Next, visibility is computed on all the line segments. These line segments can be combined into **chains** that terminate at cusps, and image-space intersections. The chains form a view graph [Grabli et al. 2010], with graph vertices at cusps and image-space intersections between curves (Figure 21).

8.4 Visibility

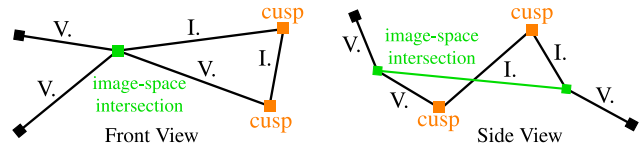
After image-space splitting, all line segments and all chains must be entirely visible or invisible. The algorithm then performs visibility tests for each line segment in each curve.

Per-Segment Visibility. All Surface Intersection curves that lie within back-faces are marked invisible. All interior contour edges (Section 8.1) are also marked invisible (except when adjacent to cusps, because the near-face test can be unreliable on rare occasions in these cases).

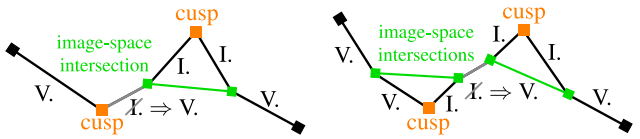
Then, for each segment on which visibility remains unknown, a ray test is performed to determine if the segment is occluded by some other object in the scene. There are several situations in which ray tests are unreliable: testing segments that lie on or adjacent to an inconsistent face; testing segments adjacent to a cusp; ray tests that intersect an inconsistent face. For these cases, the segment visibility is initially marked as ambiguous.

Chain Visibility. A chain is marked as visible if the majority plus one of its segments is visible. If the visibility of all its segments is ambiguous (e.g., because all ray tests were unreliable), the visibility of the chain is considered ambiguous.

Ambiguity Resolution. The visibility of any remaining ambiguous chains is resolved by propagating visibility between chains. In particular, any ambiguous chain that directly connects cusps whose



(a) Expected configuration near two isolated cusps



(b) Inconsistent configuration: one incorrect image-space intersection

(c) Inconsistent configuration: two incorrect image-space intersections

Fig. 23. **Cusps visibility consistency. (a)** For an isolated pair of cusps, a specific configuration is expected. Yet this configuration is highly sensitive to the precision of the detection of image-space intersections. Two simple cases **(b)** and **(c)** are detected, and chain visibility is modified to match the expected result.

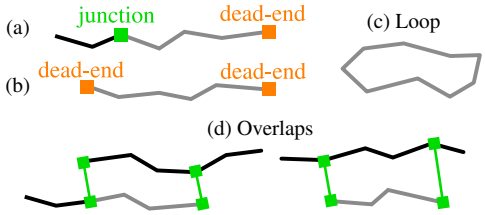


Fig. 24. **Topological simplification.** Four cases are considered for simplification (candidate chain depicted in grey): **(a)** junction to dead-end connection; **(b)** dead-end to dead-end connection; **(c)** small closed loop; **(d)** small overlaping pieces of curve between two junctions.

outgoing chains are both visible is marked visible (Figure 22(a)), since at least one cusp is spurious (i.e., produced by numerical error) in this configuration. Any ambiguous chain joining two image-space intersections whose arc length is below a small threshold and whose mates (where two chains that connect at an image-space intersection are "mates" if they are both the near chains or are both the far chains of the intersection) are both visible is also considered visible (Figure 22(b)). Finally, any ambiguous chain on the boundary connected to a visible chain not at an image-space intersection (e.g. bifurcation) is marked visible (Figure 22(c)). The process repeats until no more propagation is possible. Any remaining ambiguous chains are marked as invisible.

Cusp Visibility Consistency. Correct 2D contour connectivity near cusps is especially difficult to ensure, since a single inaccurate image-space intersection can break the curve. Yet, the most simple and common configuration near an isolated pair of cusps has known visibility (Figure 23(a)). The algorithm tests whether the detected image-space intersections and previously computed visibility are consistent with the expected model, and try to fix inconsistent configurations. In particular, if the image-space intersection is detected between the two cusps (Figure 23(b)), the edge connecting the image-space intersection to the visible cusp is marked visible. More complex configurations such as Figure 23(c) are fixed similarly.

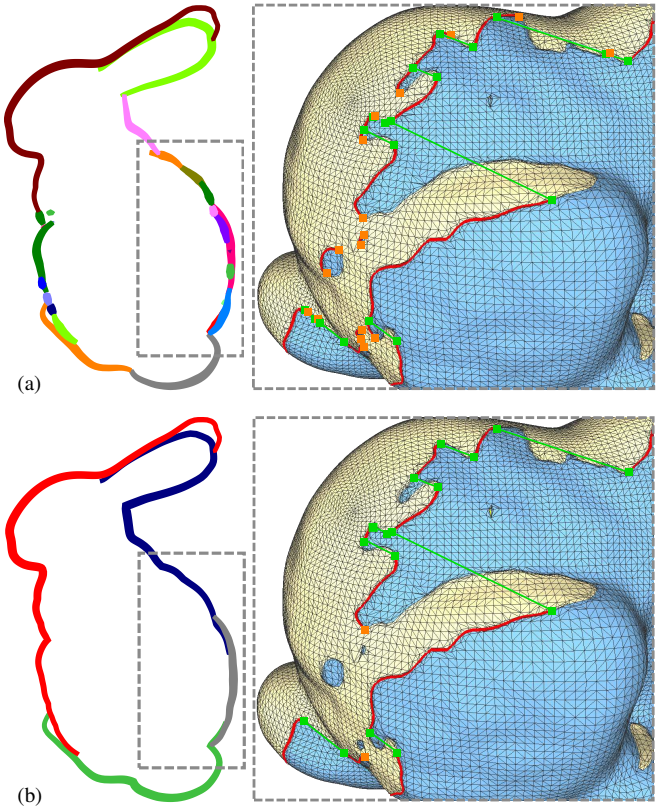


Fig. 25. **Result of topological simplification on the Stanford Bunny. (a)** The original contour is highly complex due to the bumpiness of the Bunny surface. **(b)** Topological simplification turn this small pieces of contours into two long curves. In both cases the contours are extracted from the optimized mesh generated by our algorithm (Section 6). For clarity, radial triangles are not color-coded separately. Bunny © Stanford Computer Graphics Laboratory

9. TOPOLOGICAL SIMPLIFICATION

The computed set of curves and their visibility can be rendered and stylized with standard techniques. However, we find that the true contours often exhibit small, unappealing topological details. For example, a cusp can introduce an unexpected break in a contour curve, and tiny loops on the surface can introduce tiny curves, as illustrated in Figure 26. We emphasize that these are undesirable properties of the *true* contour of the smooth surface, and not artifacts of our tessellation. (We have verified that the cusps occur on the true contour by plotting the radial curvature of the smooth surface $\mathcal{S}$). Furthermore, inconsistency can sometimes introduce spurious extra curves in the graph.

We perform topological simplification to improve the computed curves. Topological simplification is a stylistic control, one that depends on the scale of the objects as well as the rendering style; for example, one would generally want to simplify a distant object more than a near one. Our simplification is similar to previous approaches [Isenberg et al. 2002; Northrup and Markosian 2000], but these methods were used to estimate topology from noisy curves, and thus mix topological estimation and stylization. In contrast, we begin with the true curve topology and use simplification only for stylization.

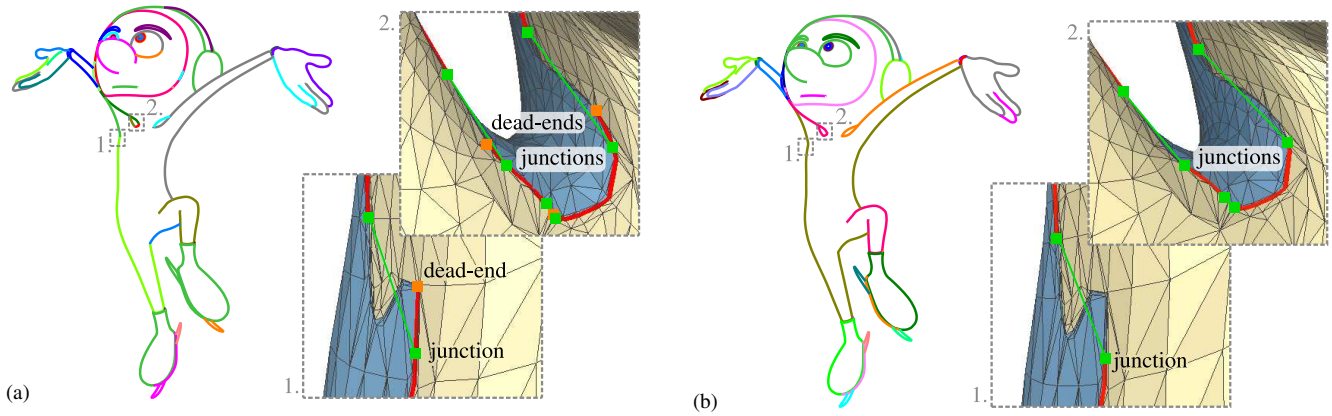


Fig. 26. **Result of topological simplification on Red.** Closeup on Red's shoulder and armpit, with genuine cusps, **(a)** before and **(b)** after topological simplification. In both cases the contours are extracted from the optimized mesh generated by our algorithm (Section 6), even though radial triangles are not color-coded in dark blue/yellow for the sake of readability. Red © Disney/Pixar

Table I. **Statistics of the mesh generation algorithm**

Sequence	Input faces	Output faces	Inconsistent faces	Time
Stanford Bunny*	42,928	51,314	11	7 min
Angela's face	38,568	50,150	10	7 min
Walking man	123,264	141,627	4	9 min
Red (Figure 1)	85,336	115,997	53	26 min†

Each result listed is for a single, typical frame of each input sequence. Timing is only for mesh generation; times for chaining, visibility, and rendering are not included. The number of input faces is the number of triangles in the initial tessellation (i.e., after one round of subdivision). *Base quad mesh obtained from the the original triangle mesh using Ray et al.'s [2006] method. †The first three timings were computed on a 2.7GHz i7 GHz MacBook Pro, whereas the fourth timing is from an older 2GHz i7 MacBook Pro. However, the slower time reflects the much more high-curvature geometry.

We categorize view graph vertices by the number of visible curves they connect: a **dead-end** vertex is adjacent to a single visible curve (e.g., a visible curtain fold); a **connector** vertex is adjacent to two visible curves, and a **junction** vertex is adjacent to more than two vertices, i.e., bifurcations and image-space intersection vertices.

We define a candidate chain as any connected sequence of visible curves that do not contain any junctions, but with image-space arc length less than a user-specified threshold (between 10 and 20 pixels in our experiments). The algorithm marks as invisible any candidate chain that (a) connects a junction to a dead-end, (b) connects a dead-end to a dead-end, (c) connects a vertex to itself, or (d) is overlapped in 2D by another chain. These cases are illustrated in Figure 24. This process is iterated until there are no more changes to be made.

This procedure is sufficient for Catmull-Clark surfaces at the scales shown here. Thanks to the accurate visibility provided by our approach, even contours with many cusps and junctions, such as the silhouettes of the Stanford Bunny, are effectively simplified (Figure 25).

10. RESULTS

We have applied our method to a number of challenging cases, including a detailed 3D character modeled with Catmull-Clark surfaces (Figure 1). Timing information for the mesh generation process is given in Table I. Mesh generation typically takes a few minutes for large meshes, and yields just a few inconsistent triangles.

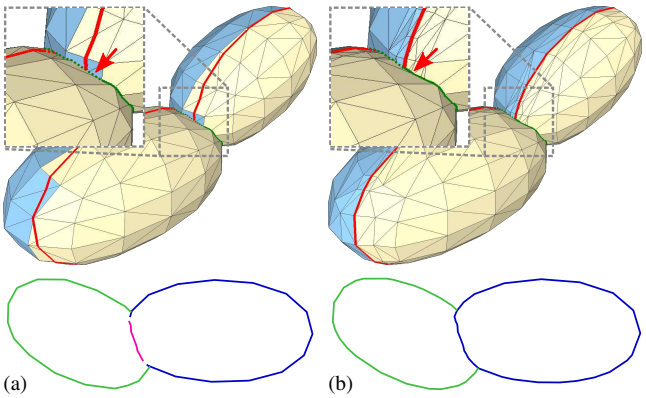
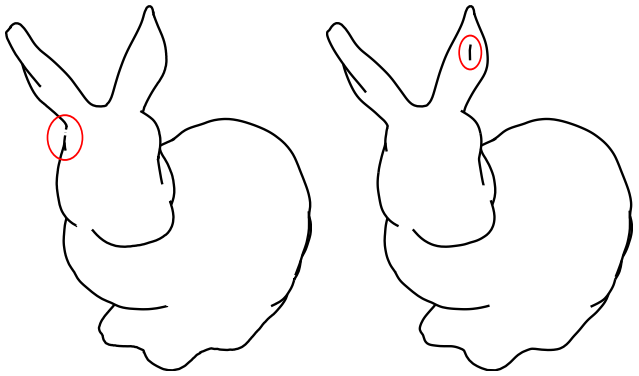


Fig. 27. **Visibility of Surface Intersections.** Defining visibility for curves near interpolated contours is difficult, such as the Surface Intersection curves (green). **(a)** Heuristics can be used to get reasonable interpolated contour visibility in simple cases, but the Surface Intersection curves still give errors (red arrow). **(b)** Our method allows to compute accurate visibility for all curves, preventing gaps in the chains.

Given this mesh, chaining, visibility, and rendering are performed by a heavily-modified version of the Freestyle software [Grabli et al. 2010]; this process takes typically takes a few more minutes. The implementations are not at all fine-tuned, and there is considerable room for speed-up by more careful implementation.

One shortcoming of using interpolated contours [Hertzmann and Zorin 2000] is that gaps sometimes appear in the object silhouettes. While these errors only occur in a few frames, even a few such errors would cause major problems for stylization and temporal coherence. In the accompanying video, we compare our algorithm to interpolated contours, using a plain rendering style that highlights these gaps. Our method does not exhibit gaps in the silhouette, as illustrated by Figures 27, 28 and 30. Also note that applying topological simplification heuristic to mesh or interpolated contours does not solve visibility errors (Figure 29).

We use the parameterization scheme of Kalnins et al. [2003] to generate stylized results such as the temporally coherent tapered curves of Figures 30 and 32 and the textured strokes of Figure 31. (Also see the supplementary video.) Our results still exhibit flickering due to the stylization procedures. Most of the flickering re-



(a) Interpolated contours (b) Our method

Fig. 28. **Bunny contours,** rendered in black-and-white without topological simplification, to better-visualize gaps in the silhouette. The interpolated contour **(a)** introduces a gap in the silhouette and does not detect one piece of contour (red circles), whereas our method **(b)** produces a solid silhouette. Bunny © Stanford Computer Graphics Laboratory

(a) Mesh contours (b) Interpolated contours

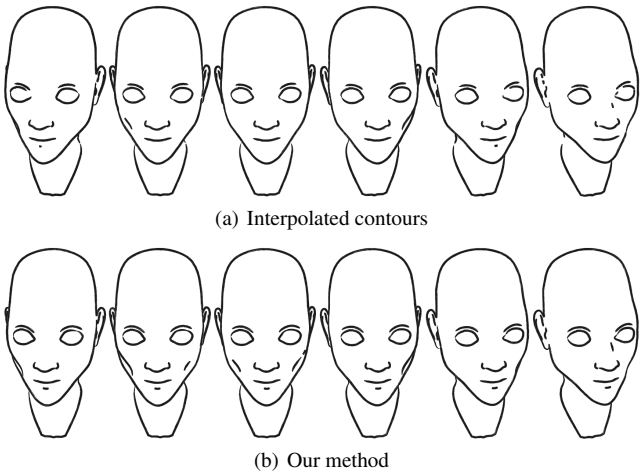
Fig. 29. Applying topological simplification to existing methods does not eliminate problems. (Compare to Figure 1). Red © Disney/Pixar

sults from topological simplification, since the algorithm operates on each frame independently, and thus does not know which curves will grow or shrink in adjacent frames. A few artifacts also appear due to suboptimal decisions in coherent stroke stylization. These issues indicate a need for more research on these subproblems.

11. DISCUSSION AND FUTURE WORK

This paper presents the first theory of consistent mesh tessellation for contours, and algorithm for obtaining consistent meshes. The algorithm produces results that are consistent almost everywhere, sufficient to produce stylization without the artifacts of previous methods. Our framework should guide the way to developing algorithms that fully guarantee consistency. Conversely, an approximate GPU-based tessellation based on our theory could produce accurate contours in real-time.

Our work provides several different types of guarantees. A mesh that is Strongly Contour-Consistent is guaranteed to have a 3D contour generator that is topologically equivalent to those of the input smooth surface, with small, subtriangle loops omitted. Radial triangles are guaranteed to be consistent, provided they are small enough that the surface is locally approximately second-order. Our algorithm produces surfaces that are Contour-Consistent almost everywhere. Regions of inconsistency are identified as such, so that



(a) Interpolated contours

(b) Our method

Fig. 30. **Stylization results on Angela's face.** Temporally coherent tapering based on Kalnins et al. [2003]. **(a)** Interpolated contours are incomplete and exhibit many breaks. **(b)** Our method produces long temporally smooth silhouettes. (Also see the supplementary video.) Angela © Chris Landreth

visibility computations can be modified accordingly. We do not strictly ensure Strong Consistency, because we use dense sampling, though interval analysis could be used instead.

The problem of computing 3D visible contours is surprisingly difficult, and has plagued non-photorealistic rendering for many years. We began this research with an approach based on using interpolated contours to redefine visibility, but this method failed near complex, subtriangle curves. We then tried a method that modified the surface to have exact silhouettes at the interpolated contours, using CSG-like operations, but this method failed when the implied topology became too complex. Based on observations from this experience, we developed the notion of Contour Consistency, which we attempted to optimize by resampling the mesh, by gradient-based numerical optimization, and by combinations of these approaches. These methods never achieved satisfactory results at the contour, particularly near cusps and other regions of low radial curvature. We developed Radial Triangles, but attempted to creat them with a front-propagation approach that became too difficult to implement. Finally, we developed the iterative algorithms introduced here, which achieve consistency nearly everywhere.

Based on this experience, we can say with some confidence that the problem is fundamentally difficult. The main reason is that visibility entails making long-range topological decisions (e.g., the visibility or connectivity of a large stroke) based on unstable numerical predicates. We often developed heuristic methods that seemed like they should be "good enough," but Murphy's Law always applied, and obscure-seeming failure cases inevitably arose. The method described in this paper is far more robust than any of our previous attempts, particularly in the contour regions where the Radial Triangle guarantees apply.

Building a complete pipeline for high-quality animation of contours will require addressing several open problems, most of which are only touched upon by previous research. First, robust visibility for meshes with very small triangles remains a problem. As discussed in Section 8, our implementation of curve visibility suffered from some numerical issues. We devised effective heuristics that resolved most ambiguities for our test sequences. Second, our experience shows that topological simplification will be a necessary step for any non-trivial geometry. Topological simplification is a

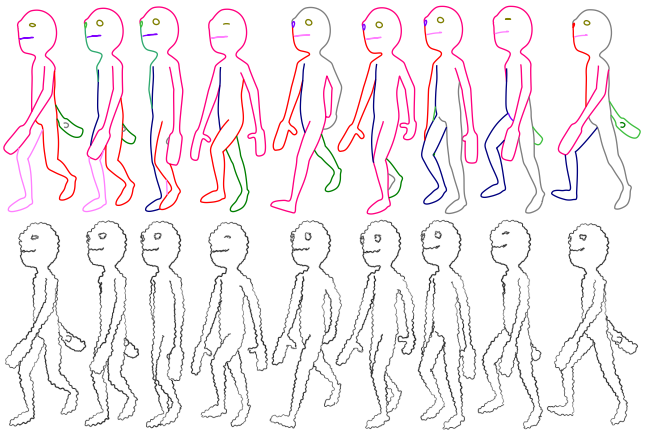


Fig. 31. **Stylization results on the walking man.** Colors propagation and temporally coherent parameterization based on Kalnins et al. [2003]. (Also see the supplementary video.) Hank © Ryan Dale

stylization step, one that must take into account topology, image-space density [Grabli et al. 2010], temporal coherence, and other stylistic factors. Third, space-time curve correspondence and parameterization [Bénard et al. 2012; Buchholz et al. 2011; Kalnins et al. 2003] is necessary for temporal stylization. Finally, it is also necessary to combine contours with other types of curves, such as suggestive contours [DeCarlo et al. 2003], which must meet the contours at cusps in each frame. Contour Consistency provides the necessary foundation upon which principled solutions to each of the above problems can be built.

Though we focus on contour computation, our approach can also be used to compute consistent Planar Maps, which are also important for stylization [Eisemann et al. 2008; Karsch and Hart 2011; Winkenbach and Salesin 1996]. Generalizing the above pipeline for computing and stylizing space-time Planar Maps is a more general open research problem, that could provide a very powerful tool for high-quality and general stylization.

Even when our contours have correct visibility, we do not guarantee topological correctness of the 2D line drawing, because infinitesimal perturbations in contour positions can, in principle, change the projected topology and visibility. It may be possible to develop an approach that does provide this guarantee, by creating an image-space analogue to consistency. However, this would likely require a procedure for exact ray-tests on smooth surfaces, which might not be robust. For non-photorealistic animation, we do not believe that perfect topological accuracy in image-space is necessary, but it maybe useful in other domains.

ACKNOWLEDGMENTS

The animated character “Red” was created by Andrew Schmidt, Brian Tindall, Bernhard Haux and Paul Aichele, based on the original design of Teddy Newton. Thanks to Chris Landreth for permission to use his “Angela” character. Thanks to Kurt Fleischer and John Hancock for technical assistance, to Bardia Sadri and Vladlen Koltun for discussions. Thanks to the reviewers for their extremely diligent comments that significantly improved this paper.

REFERENCES

APPEL, A. 1967. The notion of quantitative invisibility and the machine rendering of solids. In *Proc. ACM*. 387–393.

BANK, R. E. AND SMITH, R. K. 1997. Mesh smoothing using a posteriori error estimates. *SIAM J. Numer. Anal. 34,* 3, 979–997.

BÉNARD, P., LU, J., COLE, F., FINKELSTEIN, A., AND THOLLOT, J. 2012. Active Strokes: Coherent Line Stylization for Animated 3D Models. In *Proc. NPAR*. 37–46.

BLINN, J. F. 1978. A scan line algorithm for displaying parametrically defined surfaces. *Proc. SIGGRAPH 12,* SI, 1–7.

BUCHHOLZ, B., FARAJ, N., PARIS, S., EISEMANN, E., AND BOUBEKEUR, T. 2011. Spatio-Temporal Analysis for Parameterizing Animated Lines. In *Proc. NPAR*. 85–92.

CATMULL, E. AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10,* 6.

COLE, F. AND FINKELSTEIN, A. 2009. Two Fast Methods for High-Quality Line Visibility. *IEEE TVCG 8,* 1.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive Contours for Conveying Shape. *ACM Trans. Graph. 22,* 3, 848–855.

EISEMANN, E., WINNEMÖLLER, H., HART, J. C., AND SALESIN, D. 2008. Stylized Vector Art from 3D Models with Region Support. In *Proc. EGSR*.

ELBER, G. AND COHEN, E. 1990. Hidden Curve Removal for Free Form Surfaces. In *Proc. SIGGRAPH*. Vol. 24. 95–104.

GOOCH, A. 1998. Interactive non-photorealistic technical illustration. M.S. thesis, University of Utah.

GRABLI, S., TURQUIN, E., DURAND, F., AND SILLION, F. X. 2010. Programmable Rendering of Line Drawing from 3D Scenes. *ACM Trans. Graphics 29,* 2. Article 18.

GUIGUE, P. AND DEVILLERS, O. 2003. Fast and Robust Triangle-Triangle Overlap Test using Orientation Predicates. *J. Graphics Tools 8,* 1, 25–42.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, fair interpolation using Catmull-Clark surfaces. In *Proc. SIGGRAPH*. 35–44.

HERTZMANN, A. 1999. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In *SIGGRAPH Course on NPR*, S. Green, Ed.

HERTZMANN, A. AND ZORIN, D. 2000. Illustrating Smooth Surfaces. In *Proc. SIGGRAPH*. 517–526.

ISENBERG, T., HALPER, N., AND STROTHOTTE, T. 2002. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. In *Proc. EG*.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent Stylized Silhouettes. *ACM Trans. Graphics*.

KARSCH, K. AND HART, J. C. 2011. Snaxels on a Plane. In *Proc. NPAR*.

KIRSANOV, D., SANDER, P., AND GORTLER, S. 2003. Simple Silhouettes for Complex Surfaces. In *Proc. SGP*.

KOENDERINK, J. J. 1984. What does the occluding contour tell us about solid shape? *Perception 13,* 3, 321–330.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-Time Nonphotorealistic Rendering. In *Proc. SIGGRAPH*. 415–420.

NORTHRUP, J. D. AND MARKOSIAN, L. 2000. Artistic Silhouettes: A Hybrid Approach. In *Proc. NPAR*. 31–38.

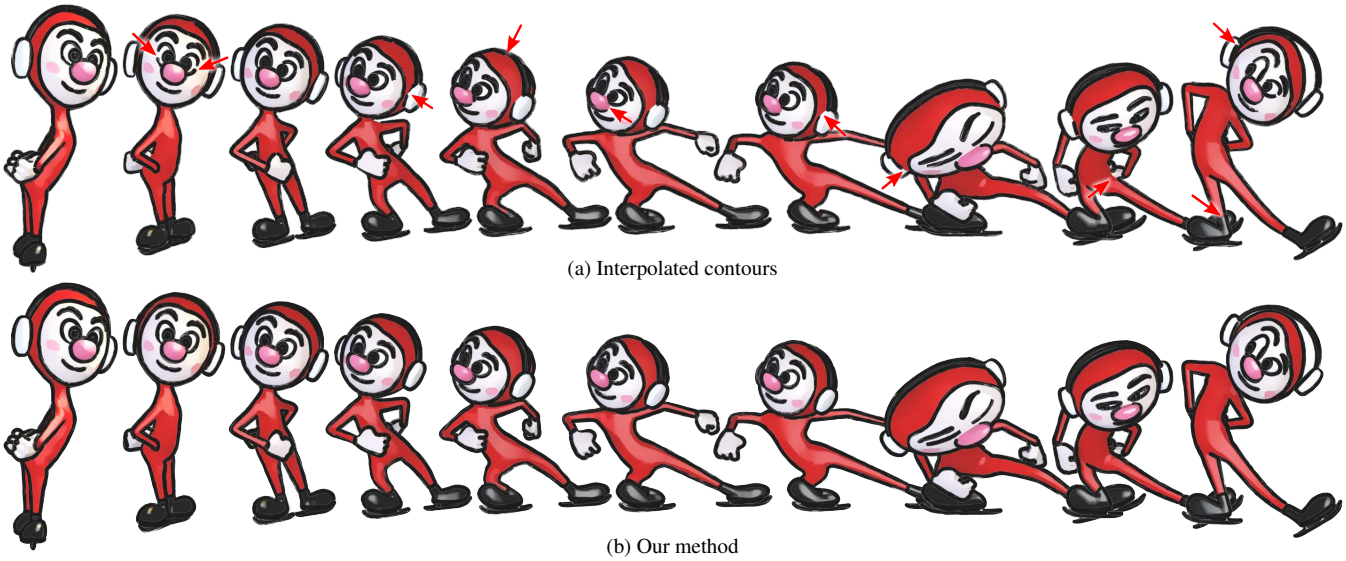
PLANTINGA, S. AND VEGTER, G. 2006. Computing Contour Generators of Evolving Implicit Surfaces. *ACM Trans. Graph. 25,* 4, 1243–1280.

RAY, N., LI, W. C., LÉVY, B., SHEFFER, A., AND ALLIEZ, P. 2006. Periodic global parameterization. *ACM Trans. Graph. 25,* 4, 1460–1485.

SAITO, T. AND TAKAHASHI, T. 1990. Comprehensible Rendering of 3-D Shapes. In *Proc. SIGGRAPH*. Vol. 24. 197–206.

STAM, J. 1998. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proc. SIGGRAPH*. 395–404.

WINKENBACH, G. AND SALESIN, D. H. 1996. Rendering Parametric Surfaces in Pen and Ink. In *Proc. SIGGRAPH*. 469–476.



(a) Interpolated contours

(b) Our method

Fig. 32. **Tapered strokes composited with toon shading.** Red © Disney/Pixar

APPENDIX

A. EXTRAORDINARY VERTICES

Our Catmull-Clark library does not support exact evaluation in a small, fixed-size neighborhood of any extraordinary point (vertex of the base mesh having a valence not equal to four). Instead, we use the following approximation at these points. To evaluate, a tessellation of the neighborhood around the extraordinary vertex is computed, bounded by locations that can be exactly evaluated. The tessellation is star-shaped, with the extraordinary point at the center. The level function $g(\mathbf{u})$ are computed exactly at the vertices, and then linearly interpolated within the neighborhood to produce a new level function $g_I(\mathbf{u})$ for the neighborhood. Similarly, radial curvature $\kappa(\mathbf{u})$ is linearly interpolated within this neighborhood. The evaluation procedure of Stam [1998] would avoid this special treatment of extraordinary points.

B. INTERPOLATION DETAILS

Various procedures in mesh generation (Section 6) require the ability to interpolate in parameter space, that is, to compute $\mathbf{u}(t) = (1-t)\mathbf{u}_0 + t\mathbf{u}_1$, for two preimage points $\mathbf{u}_0$ and $\mathbf{u}_1$. Because our initial mesh is obtained from the base mesh, $\mathbf{u}_0$ and $\mathbf{u}_1$ usually share a face on the base mesh, e.g., they are vertices on a mesh edge or lie inside the face. Hence, interpolation usually reduces to simple interpolation of the original points.

However, the vertex shifting operation (Section 6.2) can create situations in which we later wish to interpolate $\mathbf{u}$ coordinates on nearby base mesh faces. There are many ways this could be implemented, and any reasonable interpolation is sufficient for our purposes: the principal constraint is that it not lead to folds in the surface, e.g., a series of convex interpolations of the vertices of a triangle should not yield a point outside that triangle. A generic approach would be to interpolate along mesh geodesics, but we take a simpler approach here.

Our approach is based on constructing local *charts*. A vertex of the base mesh is defined as regular if it has four adjacent faces and no adjacent boundary. Because our input surfaces are user-defined and then subdivided once, almost all base mesh vertices are regular. A chart is a bijective mapping between the one-ring of a regular vertex to the 2D Euclidean plane, and is easy to construct by mapping the four base mesh quadrilaterals to the four quadrants around the origin of the 2D plane. We interpolate two points $\mathbf{u}_0$ and $\mathbf{u}_1$ by finding a vertex for which both points are within the one-ring, and then by interpolating the chart coordinates. If no such vertex can be found (e.g., if both vertices have been shifted to adjacent triangles), then the interpolation fails and the attempted insertion is cancelled.

C. CONDITIONS OF SHIFTABILITY

Several steps in mesh generation (Section 6) shift vertices to new positions. Shifting a vertex entails changing its preimage to new coordinates $\mathbf{u}$, and updating the vertex's 3D position $\mathbf{f}(\mathbf{u})$ and normal $\mathbf{n}(\mathbf{u})$ for the new location.

There are a number of conditions that must be satisfied for a point to be shiftable. First, the shift must not create a fold in the mesh. Second, the shift must not create any CCC triangles. Third, the shift must not create an "X" shape in the contour, which would happen if the vertex's one-ring has more than two zero-crossings (C vertices and FB edges) in its outer boundary. Finally, the shift must not prevent interpolation along the edges, as described in Appendix B. Shifting is performed whenever the distance from the root to the nearest vertex is less than 20% of the length of the edge, and the above conditions are met. Shifting vertices also affects how base mesh vertices are interpolated (Appendix B).