



HAL
open science

ConTesse: Accurate Occluding Contours for Subdivision Surfaces

Chenxi Liu, Pierre B enard, Aaron Hertzmann, Shayan Hoshyari

► **To cite this version:**

Chenxi Liu, Pierre B enard, Aaron Hertzmann, Shayan Hoshyari. ConTesse: Accurate Occluding Contours for Subdivision Surfaces. ACM Transactions on Graphics, 2023, 42 (1), pp.16. 10.1145/3544778 . hal-03699140

HAL Id: hal-03699140

<https://inria.hal.science/hal-03699140>

Submitted on 20 Jun 2022

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 destin ee au d ep ot et  a la diffusion de documents scientifiques de niveau recherche, publi es ou non,  emanant des  tablissements d'enseignement et de recherche fran ais ou  trangers, des laboratoires publics ou priv es.

ConTesse: Accurate Occluding Contours for Subdivision Surfaces

CHENXI LIU, University of British Columbia, Canada

PIERRE BÉNARD, Univ. Bordeaux, CNRS, Bordeaux INP, INRIA, LaBRI, UMR 5800, France

AARON HERTZMANN, Adobe Research, USA

SHAYAN HOSHYARI, Adobe, USA

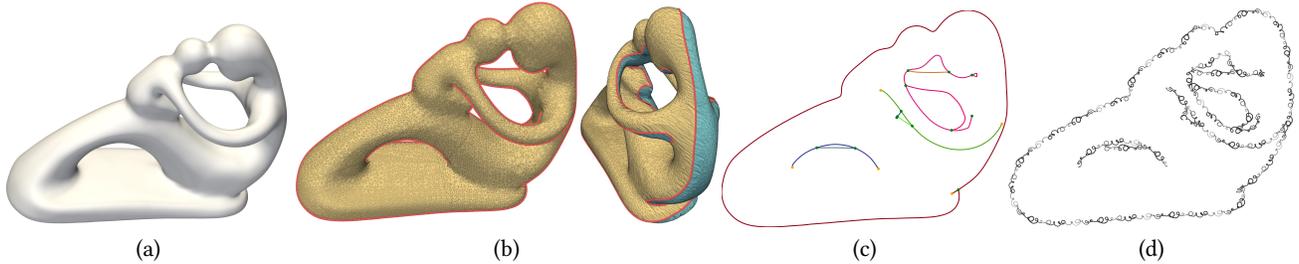


Fig. 1. Given (a) a smooth 3D surface S and a camera viewpoint, our method produces (b) a triangle mesh M where the occluding contour of the mesh accurately approximates the occluding contour of the smooth surface. Standard algorithms may then be used to extract (c) the view map of occluding contours, and to (d) stylize them. (Fertility courtesy UU from AIM@SHAPE-VISIONAIR Shape Repository).

This paper proposes a method for computing the visible occluding contours of subdivision surfaces. The paper first introduces new theory for contour visibility of smooth surfaces. Necessary and sufficient conditions are introduced for when a sampled occluding contour is valid, that is, when it may be assigned consistent visibility. Previous methods do not guarantee these conditions, which helps explain why smooth contour visibility has been such a challenging problem in the past. The paper then proposes an algorithm that, given a subdivision surface, finds sampled contours satisfying these conditions, and then generates a new triangle mesh matching the given occluding contours. The contours of the output triangle mesh may then be rendered with standard non-photorealistic rendering algorithms, using the mesh for visibility computation. The method can be applied to any triangle mesh, by treating it as the base mesh of a subdivision surface.

CCS Concepts: • **Computing methodologies** → **Computer Graphics-Non-photorealistic rendering**; *Visibility*; Shape analysis.

ACM Reference Format:

Chenxi Liu, Pierre Bénard, Aaron Hertzmann, and Shayan Hoshary. 2022. ConTesse: Accurate Occluding Contours for Subdivision Surfaces. *ACM Trans. Graph.* 1, 1, Article 1 (January 2022), 16 pages. <https://doi.org/10.1145/3544778>

1 INTRODUCTION

Computing occluding contours is one of the fundamental building blocks of 3D Non-Photorealistic Rendering, because these curves allow us to produce beautiful and stylized artistic images and animations in many styles [Grabli et al. 2010]. The occluding contours follow occlusion boundaries in an image, i.e., where one part of a

surface occludes another. These contours accurately capture many of the lines that artists draw [Cole et al. 2008].

The problem of computing the occluding contours for smooth surfaces dates back to the earliest days of computer graphics [Appel 1967; Weiss 1966]. Yet, despite more than a half-century of effort, this problem remains unsolved for smooth surfaces [Bénard and Hertzmann 2019]. No existing method guarantees results that both accurately capture the occluding contours (e.g., no curves missing or merged), while also producing topologically-consistent results (e.g., no gaps in the silhouette).

The most accurate current method is that of Bénard et al. [2014]. This method generates a new mesh, so that the contour of the new mesh approximates the contours of the input surface. Then, existing methods may be used to compute the occluding contour of the mesh, in a consistent fashion. However, this method is extremely slow and often produces meshes with a few spurious contours, which could, in theory, lead to some incorrect results. Moreover, some styles require an accurate planar map [Eisemann et al. 2008; Grabli et al. 2010; Winkenbach and Salesin 1994], which would be impossible to guarantee with such errors. Importantly, there remain a key theoretical gap: when can we expect that a valid mesh M exists?

This paper introduces new theoretical and practical advances for computing the visible occluding contour for smooth surfaces. We first derive a new theory that describes when there exists a 3D mesh M consistent with a set of sampled occluding contours C . Previous methods produce curves that do not guarantee these conditions, and thus often produce curves without any possible valid visibility. These observations provide new insight into why contour visibility has been such a challenging problem, and how to address it.

We then propose ConTesse (contour tessellation), a new method for computing the visible occluding contours of a subdivision surface. The method can be applied to any triangle mesh, by treating it as the base mesh of a subdivision surface. Like previous methods, we first

Authors' addresses: Chenxi Liu, University of British Columbia, Canada; Pierre Bénard, Univ. Bordeaux, CNRS, Bordeaux INP, INRIA, LaBRI, UMR 5800, France; Aaron Hertzmann, Adobe Research, USA; Shayan Hoshary, Adobe, USA.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics*, <https://doi.org/10.1145/3544778>.

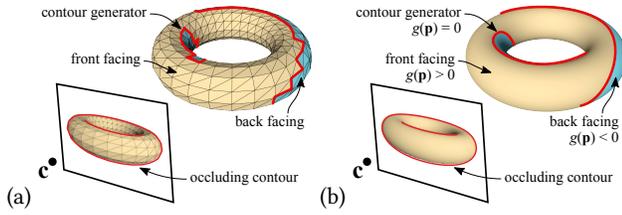


Fig. 2. Definition of occluding contour (or apparent contour) and occluding contour generator, for meshes (a) and smooth surfaces (b), from [BH§3.4.7.2]. The occluding contour describes occlusion boundaries in the image. Throughout the paper, we use the following color scheme: contour points are red, front-facing points are yellow, and back-facing points are blue.

sample the occluding contour into piecewise-linear curves (poly-lines). Our method then refines these poly-lines until they satisfy our new validity conditions. The method then produces a new triangle mesh that fits these sampled contours, based on a new image-space approach for ensuring consistent triangle orientation. Finally, standard algorithms compute and stylize the occluding contours, using the triangle mesh to determine visibility. We show that the resulting method is substantially faster and less memory-intensive than that of Bénard et al. [2014], and, arguably, conceptually simpler, while producing higher-quality results. Finally, we discuss possible future improvements to our approach.

1.1 Problem Definitions and Background

We now briefly review the problem of determining the visible occluding contour, and why it is difficult. For an in-depth tutorial on occluding contour algorithms, see [Bénard and Hertzmann 2019]. For brevity, in the rest of this paper, we refer to sections of [Bénard and Hertzmann 2019] with concise notation, e.g., [BH§3] for Section 3.

Suppose we view a 3D surface from a camera position c , in general position. For a point p on the surface with associated surface normal n , the orientation of the point is

$$g(p) = (c - p) \cdot n. \quad (1)$$

Points are front-facing when $g(p) > 0$, and back-facing when $g(p) < 0$. Assuming that back-faces are never visible, the boundaries between the front-facing and back-facing regions correspond to occlusion boundaries in the image. Specifically, for a smooth surface, the *contour generator* is the set of points where $g(p) = 0$, and, for a triangle mesh, the contour generator is the set of mesh edges that connect front faces to back faces (Figure 2).

The *occluding contour* (or *apparent contour*) is defined as the image-space projection of the visible portion of the contour generator (Figure 2). Note that visibility is part of this definition; hidden points are not part of the occluding contour.

This paper focuses on the problem of computing the occluding contour for a smooth surface. At first, this task seems like it ought to be quite simple. Yet, an extensive body of literature has identified this problem and attempted to solve it by a variety of clever approaches, none of which guarantee correct results.

The simplest approach is to take a triangle mesh as input, and output the mesh’s occluding contours. However, when the mesh

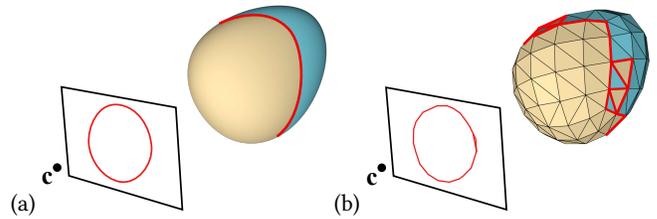


Fig. 3. Convex contour tessellation. (a) The input is a convex smooth surface viewed from a camera position c . The contour projects to a simple convex curve. (b) Naively converting the smooth surface to a triangle mesh produces a mesh for which the occluding contour projects to a curve with very different topology. The new topology can be badly behaved, leading to many visibility errors [Bénard et al. 2014].

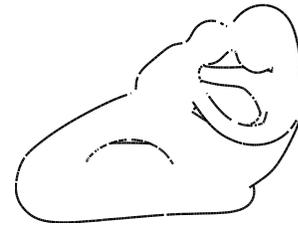


Fig. 4. Naive visibility. In this example, the occluding contour has been sampled into line segments, and then visibility for each segment was computed by a separate ray test against a dense triangulation of the surface. This naive strategy produces unreliable visibility, such as incorrect gaps in the contours. Most algorithms use a combination of heuristics that can greatly improve over a naive result, but they cannot remove all visibility errors. (Fertility by UU from AIM@SHAPE-VISIONAIR Shape Repository).

represents a smooth surfaces, the mesh contours produce knotty, incorrect topologies (Figure 3(b)). Using heuristics to untangle them [Eisemann et al. 2008; Northrup and Markosian 2000] produces unreliable results.

Many approaches instead sample polyline approximations to the smooth occluding contour generator, which is guaranteed to have simpler topology. These methods then perform visibility tests against a triangle mesh [Appel 1967; Grabli et al. 2010; Hertzmann and Zorin 2000; Karsch and Hart 2011; Markosian et al. 1997; Winkenbach and Salesin 1996]. However, these ray tests are not reliable because a triangle mesh has different visibility from the underlying smooth surface [Bénard et al. 2014; Eisemann et al. 2008; Grabli et al. 2010] (Figure 4). Such small errors in visibility tests can propagate to produce topologically-invalid drawings, such as objects with large gaps in their silhouettes. Previous methods use voting schemes and other heuristics to fix visibility, but none are exact. Computing curve visibility with image buffers [Cole and Finkelstein 2010; Eisemann et al. 2008; Saito and Takahashi 1990] or ray-tests with smooth geometry [Elber and Cohen 1990] have similar numerical problems. Our method builds most directly on the method of Bénard et al. [2014], which frequently guarantees correct results, but is very expensive to compute. Moreover, as we show in this paper, some poly-lines simply cannot be assigned valid visibility, which affects all the methods described above.

2 CONVEX SURFACE ALGORITHM

In order to provide basic intuitions for our theory and algorithms, we first describe a highly simplified version of the algorithm for convex, closed surfaces. We generalize these ideas to non-convex surfaces in subsequent sections.

The input to the algorithm is a strictly convex, oriented smooth surface \mathcal{S} , viewed from camera position c . The occluding contour of this surface must be a convex curve in the image (Figure 3(a)).

Our goal is to produce a triangle mesh \mathcal{M} whose contour generator is a good representation for the true contour generator of \mathcal{S} . Specifically, we want the contour generator of \mathcal{M} to partition the mesh into two regions, one containing only front-facing triangles, and one with only back-facing triangles. We also want \mathcal{S} and \mathcal{M} to be geometrically similar. Once the mesh is computed, the contour visibility can be computed from the mesh using standard exact techniques, such as ray tests. No existing algorithm provably achieves this goal [BH§7.6]. For example, directly using the contour generators of a triangle mesh will produce spurious 2D self-intersections [BH§6.1] (Figure 3(b)).

The first step is to sample a polyline C on the smooth surface, where each vertex on the polygon lies on the contour generator, and the polyline projects to a simple, closed polygon in 2D (Figure 5(a)). For example, this can be achieved by root-finding Eq. 1 on a mesh representation of the surface, see [Bénard et al. 2014]§6.2.

Our goal now is to generate a mesh for which C is the mesh's contour generator: C will partition the mesh into one region containing only front-facing triangles, and another region containing only back-facing triangles, and the front-facing region will be nearer to the camera. To generate the front-facing region, we proceed as follows (Figure 5):

- (1) Project the polyline C to a 2D image plane.
- (2) Tessellate the 2D polygon, using, for example, Constrained Delaunay Triangulation (CDT) [Chew 1989].
- (3) Project each new interior vertex back to 3D by casting a ray from the camera through the 2D position, and intersecting it with the front-facing region in \mathcal{S} .

The back-facing region is meshed by the same procedure. Finally, the output mesh \mathcal{M} is produced by stitching these two regions at C .

The key insight of this algorithm is that we can guarantee front-facing orientations by triangulating in image space and then projecting to 3D. This is because any valid 2D triangulation will produce only clockwise (or only counter-clockwise) triangles, and, furthermore, projecting a triangle from 2D to 3D preserves orientation (Appendix A). Hence, this procedure correctly produces a mesh with a front-facing region and a back-facing region, separated by C . Moreover, the output mesh \mathcal{M} is topologically equivalent to \mathcal{S} , and geometrically similar due to the use of C and the ray-casting step. The geometric accuracy can be improved arbitrarily by refining the 3D contour sampling and the 2D triangulation before ray-casting. Hence, this algorithm solves the contour meshing problem for the convex case.

Once the mesh \mathcal{M} is computed, it can be rendered and stylized with standard non-photorealistic rendering methods. The main benefit of having the mesh is that it can be used to determine curve

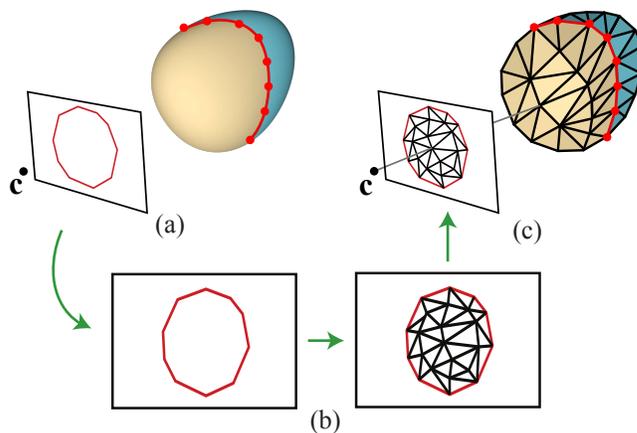


Fig. 5. Steps of the convex surface tessellation algorithm. (a) Points are sampled along the smooth surface contour. These points are projected to a 2D polygon in image space. (b) The polygon is triangulated in image space. (c) 2D triangles are lifted to 3D, by projecting their vertices back onto the original surface, in both the front-facing and back-facing regions.

visibility with ray tests, for example, in a scene comprising multiple convex objects. Note that computing the back-facing region is unnecessary for computing contour visibility, provided the scene is set up appropriately. We include back-facing regions in this paper solely for completeness and for visualization.

3 GENERAL CONTOUR REGIONS AND POLYGONS

Suppose we sample the contour generators of a closed smooth surface into polylines C , where the surface is not necessarily convex. When can those contours be triangulated into a new mesh? That is, does there exist a new mesh where C are its contour generators, and the mesh has the same topology as the smooth surface? This section describes the types of contours that may occur on smooth surfaces, which allows us to identify which kinds of polygons can and cannot be triangulated.

These questions are important because the existence of a triangulation implies that there exists a valid visibility labeling for the curves. Conversely, if no triangulation exists, then a plausible visibility labeling may likewise be impossible.

We consider an oriented smooth surface \mathcal{S} , viewed from camera position c , for which all back-facing points are invisible due to occlusion. We assume general position (a.k.a. generic position) [BH§3.5]. For the discussion in this section, we assume that surfaces are closed and do not contain self-intersections. As a consequence, the occluding contour generator of \mathcal{S} is a set of closed loops that partition the surface into front-facing and back-facing *regions*.

Hence, for a given polyline sampling of the contour generator of \mathcal{S} , the question of whether or not a valid triangulation exists is equivalent to the question of whether or not all of the regions enclosed by those contours can be meshed with the appropriate orientations, i.e., all front-facing or all back-facing. As discussed in the previous section, this reduces to the question of whether each region can be triangulated in image-space.

3.1 Types of Regions

This section categorizes the different types of 3D surface regions, in terms of the types of curves their boundaries project to. Each region must be entirely front-facing or entirely back-facing. The categorization applies equally to smooth surface regions and polygonal regions. This categorization is nested: each category is more general than the previous ones, and a triangulation algorithm that works for the final category applies to all of these cases. This categorization applies regardless of whether or not the regions are bounded by contours.

Simple curves (Figure 6(a)). The easiest case is where a region is bounded by a simple 2D curve, i.e., a curve that does not intersect itself in image space. A valid sampling of a simple curve produces a simple polygon in 2D, which can be triangulated in 2D with methods such as CDT (Figure 5).

Self-overlapping polygons (Figure 6(b)). When one part of a region overlaps a separate part in image space, the boundary curve is called *self-overlapping*. Self-overlapping polygons can be triangulated using the algorithm of Shor and Van Wyk [1992]§4.

While self-overlapping polygons can have multiple incompatible triangulations [Shor and Van Wyk 1992]§3, we have not observed incompatible triangulations in practice. If needed, these incompatibilities could be resolved by using “crossing” constraints [Eppstein and Mumford 2009], i.e., the depth ordering at 2D intersections implied by the 3D locations of the curves.

Weakly self-overlapping (WSO) (Figure 6(c)). A curtain fold cusp ([BH§4.3]) in the contour generator creates a singularity in the occluding contour where the surface self-overlaps. The corresponding polygon also has a singularity. Singularities are marked with red dots in Figure 6. Weber and Zorin [2014]§3, call a singular polygon that overlaps “weakly self-overlapping” (**WSO**), and provide an algorithm for triangulating WSO polygons with singular vertices tagged in the input. Note that this algorithm can also triangulate simple curves and self-overlapping curves, which are all considered to be WSO. We give formal definitions of WSO in Section 3.3.

Weakly-Self-Overlapping with Holes (WSOH) (Figure 6(d)). In the most general case, which we call **WSOH**, a region may have holes, and possibly handles. Polygonal regions with holes may be triangulated by first introducing a cut to remove holes and handles, and then applying the WSO algorithm. We formally define WSOH in Section 3.3.

Because this is the most general case, we develop an algorithm that works for WSOH regions, and it automatically handles the simpler cases described above. Simple curves, self-overlapping curves, and weakly-self-overlapping curves are all considered WSOH.

3.2 Invalid Polygons

Sometimes, sampling a contour curve produces a polygon that cannot be triangulated in 2D. For such a polygon, the 3D polyline cannot be triangulated with solely front-facing (or back-facing) triangles. We call such polygons *invalid*. The example in Figure 7(a) shows a case where undersampling introduces a self-intersection into the

curve; we call this structure a *twist*. See Shor and Van Wyk [1992]§1 for more discussion and examples of invalid curves.

3.3 Theorems

We now prove theorems that establish the significance of the WSO and WSOH properties for occluding contours. The first two theorems apply to triangle meshes, and the latter two to smooth surfaces.

3.3.1 Meshes. We first review the formal definition of WSO for 2D meshes.

DEFINITION 1 ([WEBER AND ZORIN 2014]). *A polygon P is weakly self-overlapping (WSO) if there is a map f from some planar mesh M , homeomorphic to a disk, to the plane such that $f(\partial M) = P$, all triangles are mapped with positive orientation, and $\Theta = 2\pi$ for each internal vertex in $f(M)$, where Θ is the sum of triangle angles around a vertex. $f(M)$ is called a triangulation of P .*

The mapping is illustrated in Figure 8, and Θ in Figure 9(a).

Overlaps ($\Theta > 2\pi$) may occur at boundary vertices; these are singularities. If $\Theta \in [0, 2\pi]$ for all boundary vertices as well, then the polygon is self-overlapping.

The condition $\Theta = 2\pi$ rules out a peculiar violation of local injectivity. In normal situations, the fact that all triangles have positive orientation implies that $\Theta = 2\pi$ for each internal vertex (Figure 9(a)). However, Weber and Zorin point out that a spiral structure (Figure 9(b)) produces a non-injective local structure with $\Theta = 4\pi$, or, more generally, a positive integer multiple of 2π . On a 3D mesh, we call a vertex *spiral* if, for some viewpoint, either the sums of the positive angles or the sums of the negative angles lie outside $[-2\pi, 2\pi]$. This generalized definition will be useful later for ruling out another hypothetical structure called fusilli cusps [BH§4.7]. We have never observed spiral vertices on any real meshes.

We now generalize the definition of WSO to regions with holes.

DEFINITION 2. *Let \mathcal{P} be a set of K polygons. This set is called weakly self-overlapping with holes (WSOH) if there exists a genus- $(K - 1)$ 2D mesh M such that $f(\partial M) = \mathcal{P}$, all triangles are mapped with positive orientation, and $\Theta = 2\pi$ for each internal vertex in $f(M)$. $f(M)$ is called a triangulation of \mathcal{P} .*

These quantities are illustrated in Figure 8. Note that a single WSO polygon P is a special case of a WSOH set, with $K = 1$. The associated mesh has disk topology (genus 0).

THEOREM 1. *Let T_{3D} be a connected triangle mesh in 3D. Let c be a camera position with an associated image plane. Assume all vertices in T_{3D} have positive depth from the camera and no vertices are spiral. Let T_{2D} be the projection of the mesh on the image plane. T_{2D} is the triangulation of a set of WSOH polygons \mathcal{P} if and only if all triangles in T_{3D} are front-facing or all back-facing. Moreover, T_{2D} is WSO if and only if the above conditions hold, and T_{3D} is genus 0.*

PROOF. (if) Suppose all triangles in T_{3D} are front-facing. Then, all triangles in T_{2D} have positive orientation, because projection preserves orientation (Appendix A). All interior vertices have $\Theta_i = 2\pi$ because of the positive orientation of the adjacent triangles and the non-spiral condition. T_{2D} is bounded by K polygons, where K is one more than the genus of T_{3D} .

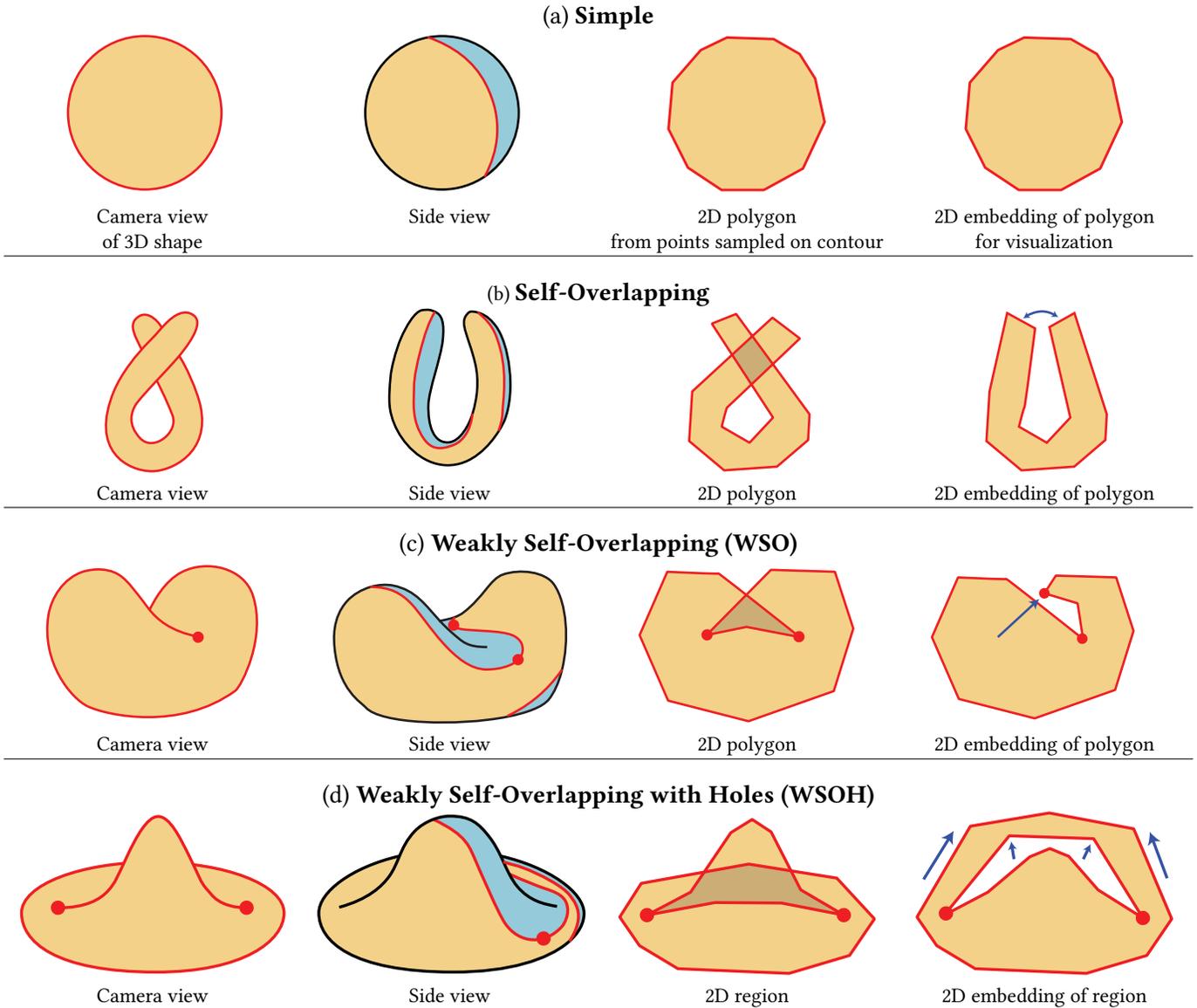


Fig. 6. The four categories of contour regions in 3D, and how they project to 2D. Each row shows a smooth surface from the camera view and from a side view (first and second columns) with the contour in red, the front-facing region in yellow, back-facing regions in blue, and cusps/singularities as red dots. A 2D projection of the front-facing region is shown (third column), with vertices sampled from the contour. The final column is meant to aid in understanding the 2D region; vertices are translated to unfold the region.

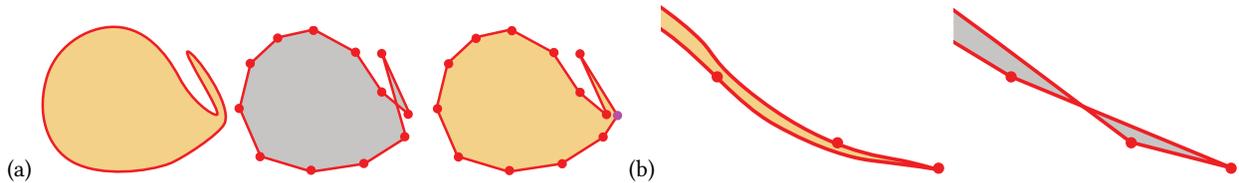


Fig. 7. Invalid curves. (a) An example in which sparsely-sampled points around a simple smooth curve produce an invalid 2D polygon (in grey). The polygon self-intersects, and cannot be triangulated without introducing a twist in the resulting 3D surface. In this case, adding a sample point (purple) makes the polygon valid. (b) A common case where the polygon is very skinny near a cusp, and undersampling introduces a loop near the cusp.

a surface point is

$$\text{sign}((\mathbf{p}_u \times \mathbf{p}_v) \cdot (\mathbf{c} - \mathbf{p})) = \text{sign} \det(\mathbf{p}_u, \mathbf{p}_v, -\mathbf{p}) \quad (3)$$

$$= \text{sign} \det(\mathbf{f}_u z + \mathbf{f}_z u, \mathbf{f}_v z + \mathbf{f}_z v, -\mathbf{f}) \quad (4)$$

$$= \text{sign } z^3 \det(\mathbf{f}_u, \mathbf{f}_v, -\mathbf{f}) \quad (5)$$

$$= \text{sign} \det(\mathbf{f}_u, \mathbf{f}_v, -\mathbf{f}) \quad (6)$$

Equation 5 follows from properties of the determinant.

Since Equations 2 and 6 are the same, the surface is front-facing everywhere if and only if the parameterization f has positive orientation everywhere. Hence, a curve being WSO implies that a front-facing surface exists that projects to this curve, and vice versa. \square

Hence, all valid regions on smooth surfaces will be WSOH, and we must produce WSOH sets of polygons from these curves in order to be able to triangulate them.

3.4 Consequences for Contour Visibility Algorithms

These observations give new insight into why the contour visibility problem has proven so troublesome.

Existing algorithms that compute smooth occluding contours can be grouped into two categories. Planar map algorithms [Eisemann et al. 2008; Karsch and Hart 2011; Winkenbach and Salesin 1996] modify curves until they create a valid planar map; they guarantee consistent visibility but do not make any guarantees about accuracy of the contours. All other existing methods computed sampled representations of the occluding contour, and then compute visibility for these polylines [Bénard et al. 2014; Hertzmann and Zorin 2000; Weiss 1966], including methods that use numerical ray tests and curve sampling [Elber and Cohen 1990]. For this latter category, we find that naively sampling contours often produces invalid polygons, e.g., Figure 11. Further experiments are shown in Section 5.

And no possible algorithm can produce a consistent visibility assignment for invalid contour polygons, because these curves cannot be triangulated. Thus, all existing methods will fail in some cases.

These algorithms often do find correct results, for example, sampled polygons often do happen to be WSOH, and problematic areas are often completely occluded, so that the resulting drawing is valid. But errors inevitably occur as well.

4 CONTESSE ALGORITHM

We now describe the ConTesse meshing algorithm in full. This procedure follows the same high-level sequence of steps as in Section 2, but these steps are made more involved by non-convexity. Moreover, we must take steps to ensure that sampled contours are WSOH. The steps of our algorithm are illustrated in Figure 12.

Related algorithms for reconstructing shape from a network of contours have been developed in computer vision, e.g., [Roberts 1963], and sketch-based modeling, e.g., [Karpenko and Hughes 2006]. Our case is distinct in that we begin with a 3D input surface rather than 2D measurements. Related problems occur in untangling self-intersecting volumes as well, e.g., [Li and Barbič 2018; Sacht et al. 2013].

4.1 Input and Problem Statement

The algorithm takes a mesh as input and camera position \mathbf{c} . We treat the mesh as the base mesh of a subdivision surface, and we require that the surface be orientable, in general position with the camera, with back-faces never visible [BH§3.3,3.5]. The surface may have boundaries. We assume the entire scene has positive depth from the camera, i.e., no part of the scene is behind the camera. We assume no spiral vertices—including no fusilli cusps [BH§4.7]—hypothetical structures that we have never observed with real meshes. We discuss self-intersections in Section 6, which could be handled as a post-process.

The subdivision surface is parameterized by a base mesh \mathcal{P} , so that for any given base mesh point $\mathbf{u} \in \mathcal{P}$ there is a corresponding point $\mathbf{p}(\mathbf{u}) \in \mathcal{S}$ on the surface. Each point is either front-facing $g(\mathbf{u}) > 0$, back-facing $g(\mathbf{u}) < 0$, or contour generator $g(\mathbf{u}) = 0$, denoted respectively F, B, or C for short.

We aim to produce a new mesh \mathcal{M} with the following properties:

- (1) The mesh has the same topology as the smooth surface. There exists a smooth bijection that defines the correspondence between points on the surfaces. Mesh vertices have the same 3D locations as their corresponding smooth surface points.
- (2) Let C be the mesh’s contour generator, which partitions the mesh into regions; each region comprises entirely front-facing triangles or entirely back-facing triangles.
- (3) Mesh vertices must correspond to the following types of smooth surface points: mesh vertices in C correspond to C points on the smooth surface; vertices inside front-facing regions correspond to F points on the smooth surface; back-facing vertices correspond to B points. Every triangle must have at least one non-C vertex.

These conditions are equivalent to “Contour-Consistency” in [Bénard et al. 2014]§4.

4.2 Contour insertion

The first stage of our algorithm is to create an initial surface triangulation that includes a sampling of the contour generator, that is, new contour vertices C , at locations corresponding to contour points of \mathcal{S} . In the output, there are no edges containing sign-crossings of $g(\mathbf{u})$, and no CCC triangles. We use a version of the method in [Bénard et al. 2014]§6.1–6.2, with simplified handling of cusps, as follows.

Vertex insertion. The first step produces an initial mesh \mathcal{M} by uniformly subdividing the base mesh \mathcal{P} a predetermined number of levels (§6.1). Root-finding on $g(\mathbf{u})$ is applied to every FF and BB edge, and edges are split whenever roots are found. Specifically, if \mathbf{u}_0 and \mathbf{u}_1 are the parameter locations (preimages) of two adjacent vertices, root-finding densely samples $g(\mathbf{u}(t)) = g((1-t)\mathbf{u}_0 + t\mathbf{u}_1)$ along each edge. Root-finding and splitting is repeated on any new FF and BB edges (up to a maximum of five recursions). Next, contour insertion is performed on all FB edges, as in §6.2, but with no special handling for cusps. Finally, we perform root-finding to find cusps, by repeatedly bisecting any triangles with sign-crossings in both g and radial curvature (§6.2). If a cusp is detected in the interior of a triangle, a new vertex is inserted at the cusp, and the triangle is

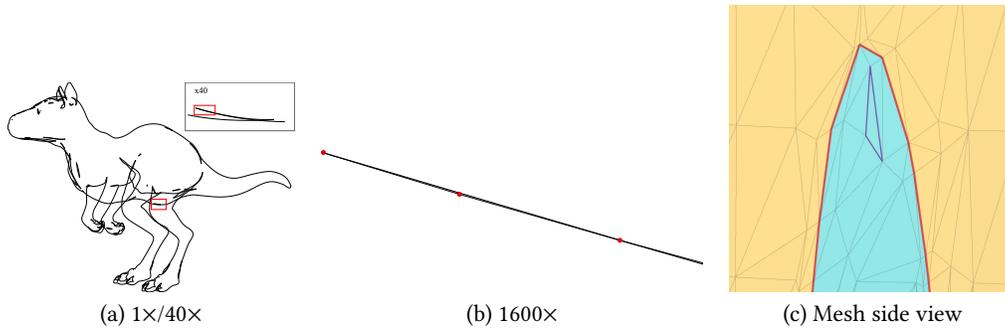


Fig. 11. Invalid contours with [Bénard et al. 2014]. (a) Bénard et al. produce highly refined contours using root-finding. Nonetheless, the contours may fail to be WSO. In the example here, a twist occurs in the sampling near a cusp, similar to the example in Figure 7(b). (b) In image-space, this twist is an *extremely* thin structure, nearly invisible even at the 1600 \times zoom shown here. (c) As a result, Bénard et al.'s triangulation produces an inconsistent triangle in this region, shown in a side view highlighted in purple. This instance is occluded and thus does not affect visibility, but there is no guarantee that this will always be the case. For this example, the output includes 11 inconsistent triangles out of 546,624 that were generated. Our method produces 100% consistent triangles, generating only 55,476 triangles for this view (Figure 14), and performing substantially faster (2 minutes for our method, and 10 minutes for Bénard et al.) (Killeroo © headus.com.au)

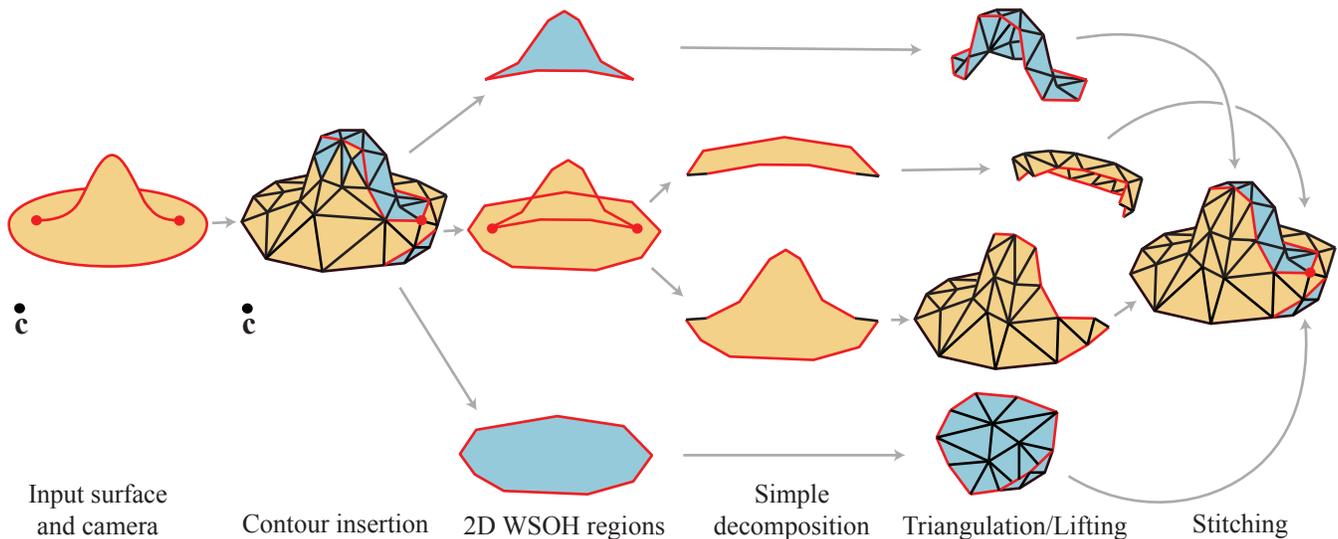


Fig. 12. Overview of the ConTesse algorithm. The input is a smooth surface viewed from a camera center c . An initial triangle mesh is computed, with contour edges inserted by root-finding. The contours partition the object into WSOH regions. Each region is decomposed into simple polygons, using cuts to remove holes and WSO triangulation to decompose self-overlapping regions. Each simple polygon is then triangulated and lifted to 3D, and these triangulations are stitched to produce the output mesh. This mesh can then be used as input to conventional non-photorealistic rendering algorithms.

split into three triangles. However, if a cusp is detected close to an existing vertex (either in image-space, world-space, or uv -space), then the existing vertex is shifted to the cusp location. This shifted vertex produces a CCC triangle, which is resolved by an edge flip.

Singularity labeling. Next, the algorithm tags singularities in the image-space contours, where the polygon must locally overlap in image space. Singularities should correspond to cusps in the contour generator, and so all cusps found by root-finding in the previous step are tagged as singularities.

A vertex can be only singular for one of the two regions it is adjacent to. For the singular region, the triangles in the one-ring self-overlap in image space (Figure 13(a)). In the tangent plane of a smooth cusp, the self-overlapping side is the convex side of the contour (Figure 13(b)). For the discrete curves we have, we compute the discrete Laplacian ($v + x - 2w$) of the contour loop in 3D, and then determine which of the two regions the Laplacian vector points to by projecting it onto the one-ring (Figure 13(c)). The other region – that it does not point to – gets a singularity label at this vertex.

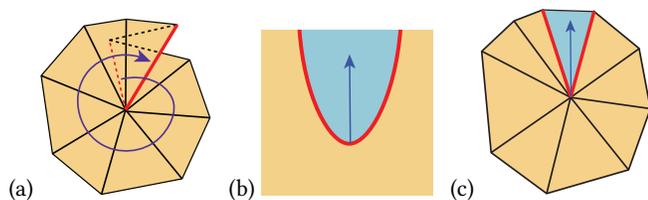


Fig. 13. Singularity side determination. (a) At a cusp, one of the two adjacent regions self-overlaps in image-space, i.e., the total rotation angle of the polygons $\Theta > 2\pi$. (b) In the tangent plane of a cusp on a smooth surface, the curve Laplacian points toward the smaller, non-overlapping region, which is back-facing (blue) in this example. (c) Since we are operating with a discrete mesh output, we compute the discrete curve Laplacian, and tag a singularity in the region (yellow) that it does not point to.

In some cases, the contour polygons have additional singularities missed during the previous step, e.g., see Figure 19 of [Bénard et al. 2014]. We detect these as follows. To test a contour vertex w , the algorithm bidirectionally traces along the contour generator to find two nearby contour vertices v and x , such that each of them is at least 10^{-8} from w in 3D. If the image space angle $\angle vwx$ is less than $\pi/3$, then w is marked as a singular vertex.

WSO checking and twist removal. While the smooth surface’s contour generator must be WSOH in 2D, sometimes a polygon sampled from the contour is not. This means that the polygon is invalid in 2D and cannot be triangulated consistently (Section 3.2). We address these cases with simple heuristics, and, when necessary, additional subdivision levels to increase the contour sampling. Whether a curve is WSO is checked by the algorithm of Weber and Zorin [2014]. All contours should converge to WSO with sufficient sampling density, but we use heuristics to avoid this extra computation when possible. We leave the problem of efficient WSO sampling as future work. If any contours fail to be WSO, then we first employ a series of heuristics to attempt to correct any sampling errors. Because contour insertion produces a reasonably dense sampling of contour generator, invalid portions are typically localized to a few structures that we call *twists*. The heuristics we use to detect and resolve twists are given in Appendix B.

If any curves are still not WSO after applying the heuristics, then our algorithm subdivides the original mesh to a finer level than before, and repeats all of the steps in this section. This process repeats until all curves are WSO. This new sampling is then passed to the next step, below.

4.3 Region Decomposition

At the end of the previous stage, we have partitioned the shape into a set of regions that project to WSOH regions in 2D. In this stage we decompose these regions into simple polygons in 2D, by removing holes and then applying an existing triangulation algorithm to find self-overlaps.

Removing holes. After insertion, some regions may have holes. If the region has holes, we introduce a cut, which is a set of mesh edges added to the region boundary. Adding cuts to the region boundary

produces a new region without holes, where the new boundary of the region traverses each cut twice.

To find a cut, we run the cut-to-disk algorithm of Gu et al. [2002]§3.2 on every surface region. The method of Gu et al. assumes a valid input triangulation, but our input triangulation may pass outside the polygon in image space. As a result, some possible cuts may pass outside the polygon in image space. In order to avoid bad cuts, we modify the algorithm to avoid cusps, image-space intersections, and triangles facing the wrong direction for their region, where possible. Details are provided in Appendix C.

Initial triangulation. After removing holes, the mesh can be partitioned into front-facing regions and back-facing regions. Each region is bounded by a polygon in 3D, comprising the contours and/or boundaries surrounding the region. For each region, the boundary polygon is projected to 2D, and triangulated in 2D using the WSO triangulation algorithm of Weber and Zorin [2014]§3.1–3.2, which takes polygons with labeled singularities as input. Mapping this triangulation to the 3D contour polygon gives an initial valid triangulation in 3D. However, this triangulation only uses contour vertices and so cannot accurately represent surface shape in the interior of the polygon.

Simple decomposition. We then decompose the triangulation into simple polygons, with the method of Weber and Zorin [2014]§4; see the *Simple Decomposition* step of Figure 12. Once this step is completed for the whole surface, we have decomposed it into simple 2D regions, each of which is entirely front-facing or entirely back-facing.

4.4 Triangulation and Lifting

In this stage, we generate a 3D triangulation for each simple polygon from the previous step.

Our approach is to first identify a set of 3D surface points that lie within the simple polygon, and then triangulate these points. While it may be possible instead to use the ray-casting procedure in Section 2, extra steps would be required to disambiguate rays that intersect the WSO region multiple times.

The procedure for finding these points is as follows. Initially, each non-contour edge in a 2D polygon connects two contour vertices from the 3D mesh. We first search for a path on the 3D surface connecting these vertices that projects to the line containing the edge in 2D. If we find such a path, then we march along it, and periodically produce sample points on the smooth surface.

This process skips samples that do not move in the direction of the endpoint in image space, to avoid folds due to inconsistencies. Specifically, let the endpoints of an edge be \mathbf{a} and \mathbf{b} ; after a sample \mathbf{v}_i is inserted, the next sample is inserted as \mathbf{v}_{i+1} only if $(\mathbf{v}_{i+1} - \mathbf{v}_i) \cdot (\mathbf{b} - \mathbf{a}) > 0$. Samples are skipped also if they would be within an image-space distance threshold to an existing vertex. This process is repeated for each edge of the initial triangulation.

This produces a new set of 2D/3D sample points within the polygon. The triangulation is then computed by CDT [Chew 1989] on these sample points and the bounding polygon in 2D.

In some cases our method fails to find a path between vertices in the original polygon, which produces very long edges in the

triangulation; this happens most often when one of the vertices is a cusp. For these edges, we identify a large (five-ring) neighborhood around one of the endpoints, and then apply Delaunay edge-flipping for all edges in the neighborhood, which effectively removes long edges.

This process may produce CCC triangles, i.e., triangles where each vertex lies on a contour edge, which in turn can lead to a degenerate mesh when two adjacent regions have triangles formed of the same three contour vertices. For each CCC triangle, we randomly pick an edge between two vertices that is not a contour edge, and split this edge, and then perturb the new vertex toward the camera for front-facing patches, or away for back-facing patches.

4.5 Final output

The final output mesh is produced by stitching the triangulated regions from the previous step. The occluding contours of this surface correspond to the occluding contours of the input smooth surface. This mesh can then be supplied to standard mesh contour detection and stylization algorithms [BH§9].

Assuming that all regions are WSOH, the output mesh satisfies the goals set out in Section 4.1 by design, and the contour generator of the output mesh is the contour generator sampled from the input surface. The contour generator’s visibility can be computed by applying standard visibility algorithms for mesh contours.

5 EXPERIMENTS

We implemented our method using Catmull-Clark subdivision surfaces, with exact limit position and normal evaluation using the algorithm and code from Lacewell and Burley [2007]. To compute radial curvatures for cusp detection, we use finite differences to estimate surface derivatives, and then compute radial curvature analytically from these estimates.

Our system takes a 3D model and a viewpoint c as input. The system outputs a remeshed version of the input surface. The system then computes the visible contours of this mesh using standard methods [BH§4], which are output as a tagged SVG file. These may be further stylized by standard methods; we use topological simplification [BH§9.3] and stroke texturing in our examples [BH§9.2].

Figure 14 shows typical inputs and outputs of the method, illustrating the different complexities of meshes that can be handled correctly. In each example, the output mesh is completely consistent, and simple, clean view maps are produced, which can be simplified further for stylization (Figure 15).

Finer-scale meshes may be obtained by increasing the initial subdivision level (Figure 16) or by increasing the sampling density in the triangulation and lifting step.

Dataset tests. In order to test the robustness of our method, we gathered 35 meshes from various sources. Most of the meshes are quad meshes, some including isolated triangles, and a few are purely triangle meshes. For each model, we set up 26 camera views, equally spaced around the model in a turntable configuration. Additionally, we obtained the three non-proprietary animation sequences used by Bénard et al. [2014] (Angela, bunny, walking man). Together, the turntable sequences and animations comprise 1580 distinct model/view combinations. Our implementation obtains correct

WSOH results for each one, with at most four levels of subdivision. Computation times and robustness are reported in Table 17. In nearly all cases, our algorithm requires less than 2 minutes to complete, often much less for smaller meshes. Computation times and output density depend significantly on the number of subdivision levels selected by the algorithm (Section 4.2). In some cases, the number of output triangles is substantially lower than on the input mesh; additional vertices could easily be inserted if desired. Our heuristics were developed on this test set, and so more subdivision levels and time may be required for other models.

In order to test with a very challenging model, we separately tested with the genus-131 model “Yeah, Right.” Results for two viewpoints are shown in Figure 18. Due to the complexity of the model, we ran a maximum of 3 subdivision levels, and the method succeeded in 21 of 26 viewpoints; the average run-time for successful views was 62 minutes. In contrast, the method of Bénard et al. [2014] failed to produce a fully-consistent mesh on any viewpoint, averaging 86 inconsistent triangles per frame. As illustrated in the figure, our method produces valid visibility despite the exceedingly complex topology in both 2D and 3D.

Disabling twist heuristics. We also experimented with running our method without the twist-removal heuristics, with a limit of 5 subdivision levels. The method successfully obtained WSOH results in 96% of the cases, but with greater computation times, sometimes taking many hours. It is possible that the remaining cases would have succeeded at higher subdivision levels.

Comparison to state-of-the-art. We compare to Bénard et al. [2014]’s statistics in Table 1, using the three available animation sequences for which they reported numerical results; the fourth, “Red” was proprietary. Whereas that method produced a handful of inconsistent faces for each mesh, our method produces perfectly consistent meshes. Moreover, our method operates an order-of-magnitude faster: 10× on “Stanford Bunny”, 6× on “Angela’s face”, and 13× on “Walking Man”. It also produces roughly half as many output triangles, making our output more compact (more triangles may easily be added, if desired). We also believe our method is simpler to understand and simpler to implement, and the WSOH insights here will lead to more elegant algorithms in the future.

How common are invalid polygons? As explained in Section 3, prior methods can fail if they sample invalid polygons. How common are invalid polygons? Our method uses extensive root-finding and careful sampling to compute polygons, and yet our method still frequently requires multiple subdivision levels and twist-removal heuristics in order to find valid WSOH regions. Bénard et al. [2014] use similarly-careful root-finding procedures, but do no WSOH checks; their method always has at least a handful of invalid triangles. Other methods sample the contour far less carefully, e.g., [Hertzmann and Zorin 2000]. Based on these experimental observations, **we believe that all previous methods frequently produce invalid polygons.**

6 DISCUSSION

The problem of computing visible occluding contours for smooth surfaces dates back to Weiss [1966]; we have shown, for the first

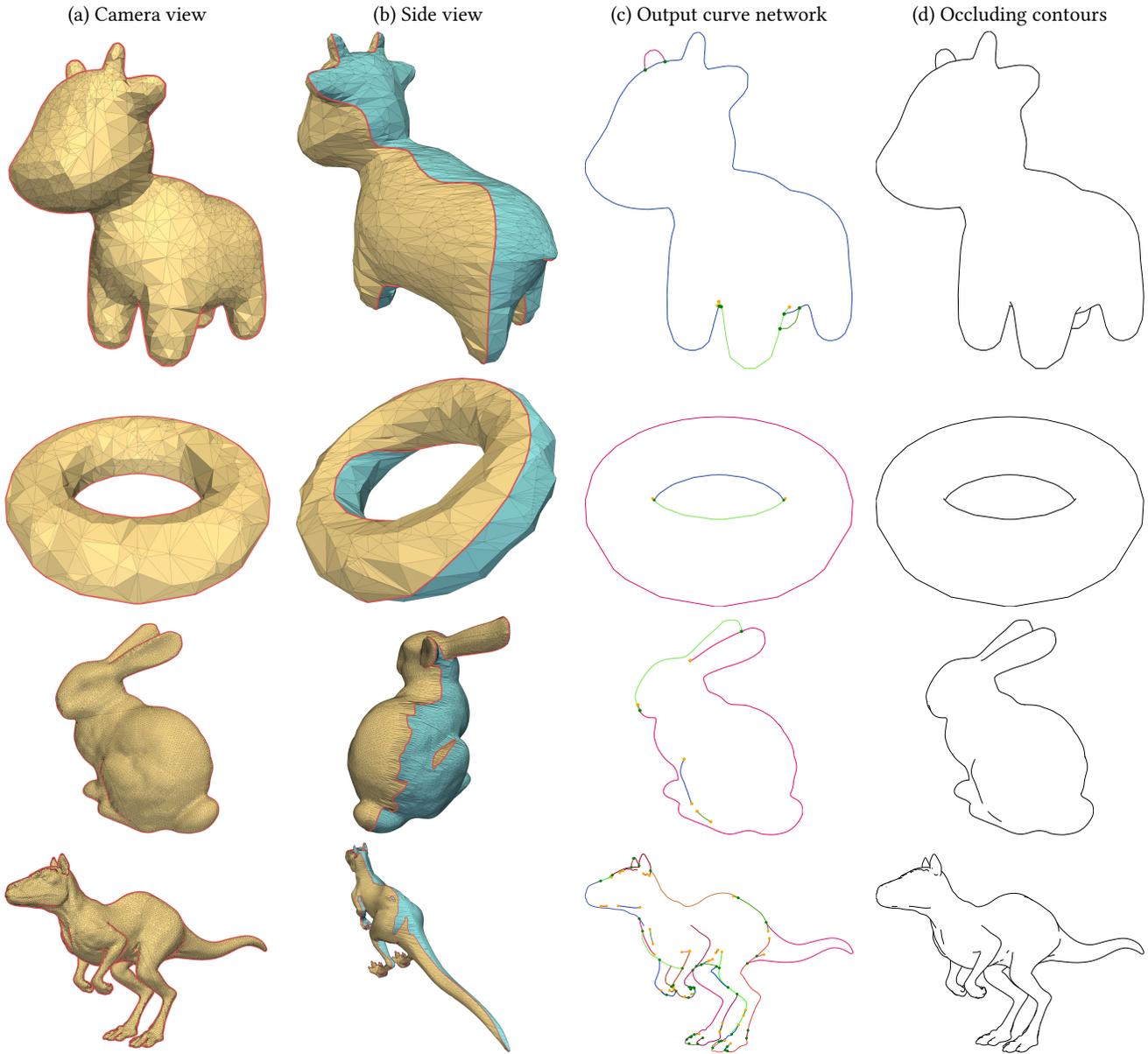


Fig. 14. Examples of the ConTesse algorithm applied to various surfaces and camera views. Each example is a commonly-used mesh in geometry processing, treated as a Catmull-Clark base mesh. (a) Camera view of the output mesh, (b) side view of the output mesh, (c) view graph (curve network) of the visible occluding contours of the output mesh, with cusps marked in orange and 2D intersections in green, and (d) occluding contours, after computing visibility. (Public domain Spot model by Keenan Crane, Killeroo © headus.com.au.)

time, how to characterize valid contours. Based on these insights, we have presented an algorithm that achieves state-of-the-art results on the problem.

Having a mathematical characterization of the space of valid solutions means that this problem is now in the domain of robust geometric computation. One important question is: how can we sample a contour curve in a way that guarantees a WSOH polygon? Simple strategies for refining the sampling seem like they ought

to produce a WSOH curve eventually, but we do not have a proof. Eliminating the need for twist and cut heuristics would also make the algorithm simpler. Some problems arise due to limitations of vertex insertion scheme and data structures, e.g., Fig 19 of [Bénard et al. 2014].

Our definition of valid triangulations does not, in itself, preserve depth ordering; instead, this is ensured by sampling all vertices from the smooth surface. We considered requiring preservation of

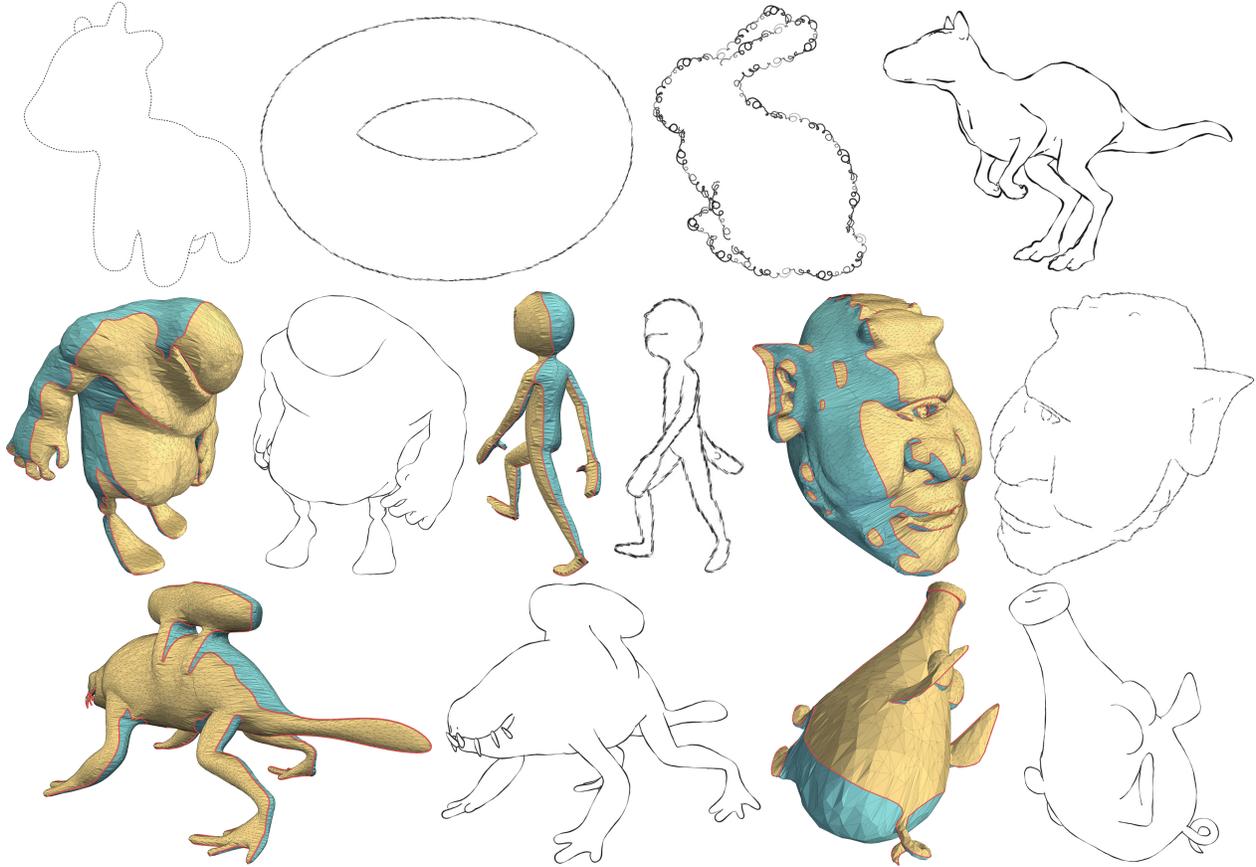


Fig. 15. Stylized versions of the contours from Figure 14, and results from three more models in our dataset. Note that we do not render mesh self-intersections. (Public domain Pig, ogre, and Spot models by Keenan Crane, Killeroo © headus.com.au, Bigguy and Monster Frog © Bay Raitt, Walking Man © Ryan Dale.)

Sequence	Input faces	Frames	Bénard et al. [2014]			Our method		
			Output faces	Inconsistent faces	Time (sec.)	Output faces	Inconsistent faces	Time (sec.)
Stanford Bunny	42,928	400	$51,336 \pm 1,383$	7.6 ± 4.4	120 ± 35	$23,424 \pm 518.1$	0	12 ± 2.2
Angela's face	39,576	150	$50,907 \pm 693$	14 ± 4.4	170 ± 42	$26,642 \pm 6,286$	0	28 ± 22
Walking Man	30,912	120	$158,184 \pm 663$	2.3 ± 2.1	160 ± 15	$10,825 \pm 2,620$	0	12 ± 10

Table 1. Statistics of our mesh generation algorithm on three of the animation sequences tested by [Bénard et al. 2014]. The “Red” animation is omitted because it is proprietary. The numbers are averaged over animation frames, which we list together with the standard deviation in each case. Our method produces no inconsistencies, far fewer output faces (roughly half), and runs much faster ($6\times$ to $13\times$ speedup). These values are for generating front and back faces, though back-faces would almost never be generated in practice. Details: Both methods were run with a single thread on the same MacBook Pro (3.1GHz Intel Core i5 CPU, 8GB of memory). The number of input faces is after one round of subdivision, and computation times are for mesh generation only, not stylization. We report fewer input faces for “Walking Man” because our method using 1 subdivision level as the minimum subdivision level, rather than 2 subdivisions used by Bénard et al.

contour convexity/concavity [BH§4.2,7.4]; while theoretically more elegant, it seemed unnecessarily complex in practice and potentially numerically sensitive. Likewise, we considered using Quantitative Invisibility (QI) [BH§4.7] to check WSO, following Eppstein and Mumford [2009]. QI uses depth ordering constraints, making it potentially much more efficient than checking with triangulation. However, QI would need to be generalized to handle holes and

convexity/concavity/cusps, which is potentially quite challenging in our case, but worth future study.

Our method can be applied to self-intersecting surfaces, although we have not tested this. That is, self-intersections do not need to be treated specially during triangulation; they can be detected on the output mesh during mesh contour extraction. However, the intersections produced by this method may be jagged; a smoothing

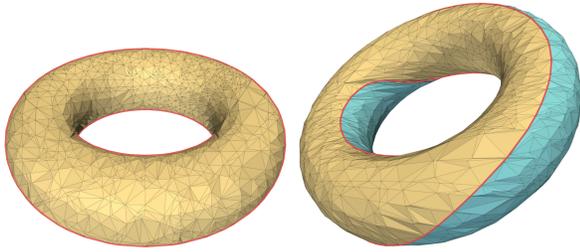


Fig. 16. Finer-resolution meshes may be obtained by increasing the number of initial subdivision levels. Here the torus was subdivided twice initially, as compared to once in Figure 14.

step could be added, or the triangulations modified to accurately track the self-intersections of the subdivision surface. Our method also does not prevent spurious self-intersections, would could also be handled by an extra detection and mesh refinement step.

In our results, we do not make any effort to control mesh quality, as mesh quality is generally not important for line rendering, though it may be useful for other applications. Improving mesh quality would be straightforward, by adjusting the 2D sample points input to the CDT.

At present, our algorithm computes many quantities that may not be used in a final rendering, e.g., many of the triangles from the output mesh may not be needed. Lazy computations could improve efficiency.

A more intriguing possibility is to compute an output planar map directly, rather than computing an intermediate mesh. We chose to focus instead on mesh generation in the belief that it would give the most insight; our results show what the mesh \mathcal{M} looks like, and now future work can explore computing visibility without explicitly computing \mathcal{M} . This will present several new challenges, such as accurately determining occlusion order without a mesh. A mesh may still be needed in some regions, such as when there are self-intersections, and for some kinds of planar map rendering.

Finally, we wonder if it is possible to apply these ideas directly to a polygonal mesh, without any explicit smooth surface representation. For example, could we adjust interpolated contours [Hertzmann and Zorin 2000] to make them WSOH, thereby avoiding the complexity of smooth surface representations?

7 ACKNOWLEDGEMENTS

We are grateful to Denis Zorin and Qingnan Zhou for very helpful discussions, to Alec Jacobson and Danny Kaufman for comments on a draft, to Hanxiao Shen for providing a WSO implementation online, and to Alla Sheffer for support. P. B enard is supported in part by the ANR MoStyle project (ANR-20-CE33-0002). Thanks to Keenan Crane for sharing the pig, ogre, and Spot models, to Ryan Dale for the Walking Man, to Bay Raitt for Big Guy and Monster Frog, to AIM@Shape for Fertility, and headus.com.au for Killeroo.

REFERENCES

Arthur Appel. 1967. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. In *Proceedings of the 1967 22nd National Conference (ACM '67)*. ACM, 387–393. <https://doi.org/10.1145/800196.806007>

- Pierre B enard and Aaron Hertzmann. 2019. Line Drawings from 3D Models. *Foundations and Trends in Computer Graphics and Vision* 11, 1-2 (2019), 1–159. <https://doi.org/10.1561/06000000075>
- Pierre B enard, Aaron Hertzmann, and Michael Kass. 2014. Computing Smooth Surface Contours with Accurate Topology. *ACM Trans. Graph.* 33, 2, Article 19 (2014), 21 pages. <https://doi.org/10.1145/2558307>
- L. Paul Chew. 1989. Constrained delaunay triangulations. *Algorithmica* 4, 1 (01 Jun 1989), 97–108. <https://doi.org/10.1007/BF01553881>
- Forrester Cole and Adam Finkelstein. 2010. Two Fast Methods for High-Quality Line Visibility. *IEEE Transactions on Visualization and Computer Graphics* 16, 5 (2010), 707–717. <https://doi.org/10.1109/TVCG.2009.102>
- Forrester Cole, Aleksey Golovinskiy, Alex Limpaecher, Heather Stoddart Barros, Adam Finkelstein, Thomas Funkhouser, and Szymon Rusinkiewicz. 2008. Where Do People Draw Lines? *ACM Trans. Graph.* 27, 3, Article 88 (2008), 11 pages. <https://doi.org/10.1145/1360612.1360687>
- Elmar Eisemann, Holger Winnem oller, John C. Hart, and David Salesin. 2008. Stylized Vector Art from 3D Models with Region Support. In *Proceedings of the Nineteenth Eurographics Conference on Rendering (EGSR '08)*. Eurographics Association, 1199–1207. <https://doi.org/10.1111/j.1467-8659.2008.01258.x>
- Gershon Elber and Elaine Cohen. 1990. Hidden Curve Removal for Free Form Surfaces. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '90)*. ACM, 95–104. <https://doi.org/10.1145/97879.97890>
- David Eppstein and Elena Mumford. 2009. Self-Overlapping Curves Revisited. In *Proceedings 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*. SIAM, 160–169.
- St ephane Grabli, Emmanuel Turquin, Fr edo Durand, and Fran ois X. Sillion. 2010. Programmable Rendering of Line Drawing from 3D Scenes. *ACM Trans. Graph.* 29, 2, Article 18 (2010), 20 pages. <https://doi.org/10.1145/1731047.1731056>
- Xianfeng Gu, Steven Gortler, and Hugues Hoppe. 2002. Geometry Images. *ACM Trans. Graphics* 21, 3 (2002). <https://doi.org/10.1145/566654.566589>
- Aaron Hertzmann and Denis Zorin. 2000. Illustrating Smooth Surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., 517–526. <https://doi.org/10.1145/344779.345074>
- Olga A. Karpenko and John F. Hughes. 2006. SmoothSketch: 3D Free-Form Shapes from Complex Sketches. In *ACM SIGGRAPH 2006 Papers (Boston, Massachusetts) (SIGGRAPH '06)*. Association for Computing Machinery, New York, NY, USA, 589–598. <https://doi.org/10.1145/1179352.1141928>
- Kevin Karsch and John C. Hart. 2011. Snaxels on a Plane. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering (NPAR '11)*. ACM, 35–42. <https://doi.org/10.1145/2024676.2024683>
- Dylan Laceywell and Brent Burley. 2007. Exact Evaluation of Catmull-Clark Subdivision Surfaces Near B-Spline Boundaries. *Journal of Graphics Tools* 12, 3 (2007), 7–15.
- Yijing Li and Jernej Barbi c. 2018. Immersion of Self-Intersecting Solids and Surfaces. *ACM Trans. Graph.* 37, 4, Article 45 (2018), 14 pages. <https://doi.org/10.1145/3197517.3201327>
- Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. 1997. Real-time Nonphotorealistic Rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., 415–420. <https://doi.org/10.1145/258734.258894>
- J. D. Northrup and Lee Markosian. 2000. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering (NPAR '00)*. ACM, 31–37. <https://doi.org/10.1145/340916.340920>
- Lawrence Roberts. 1963. *Machine Perception of Three-Dimensional Solids*. Ph. D. Dissertation. Massachusetts Institute of Technology. Dept. of Electrical Engineering.
- Leonardo Sacht, Alec Jacobson, Daniele Panozzo, Christian Sch uller, and Olga Sorkine-Hornung. 2013. Consistent Volumetric Discretizations Inside Self-Intersecting Surfaces. *Computer Graphics Forum (Proc. SGP)* 32, 5 (2013), 147–156.
- Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '90)*. ACM, 197–206. <https://doi.org/10.1145/97879.97901>
- Peter W. Shor and Christopher J. Van Wyk. 1992. Detecting and decomposing self-overlapping curves. *Computational Geometry* 2, 1 (Aug. 1992), 31–50. [https://doi.org/10.1016/0925-7721\(92\)90019-O](https://doi.org/10.1016/0925-7721(92)90019-O)
- Ofir Weber and Denis Zorin. 2014. Locally injective parametrization with arbitrary fixed boundaries. *ACM Trans. Graph.* 33, 4 (2014). <https://doi.org/10.1145/2601097.2601227>
- Ruth A. Weiss. 1966. BE VISION, A Package of IBM 7090 FORTRAN Programs to Draw Orthographic Views of Combinations of Plane and Quadric Surfaces. *J. ACM* 13, 2 (1966), 194–204. <https://doi.org/10.1145/321328.321330>
- Georges Winkenbach and David H. Salesin. 1994. Computer-generated Pen-and-ink Illustration. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. ACM, 91–100. <https://doi.org/10.1145/192161.192184>

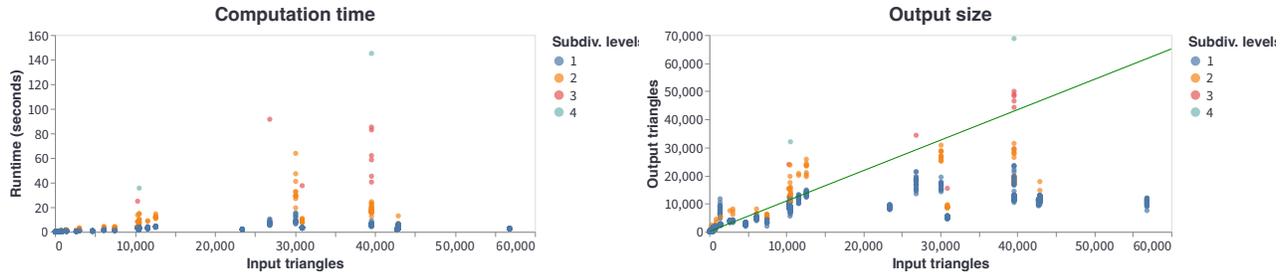


Fig. 17. Computation time and output size for each of the inputs. These results are for generating front-faces only; output triangles for back-facing regions are not generated in these plots, since they would almost never be used in practice. Each dot represents one or more of the 1580 test cases (mesh, camera), with the number of subdivision rounds required for that case color-coded. The vertical strip of dots at 39,576 input triangles are different tests on the Angela mesh. The green diagonal line shows where unity values would occur on the plot (i.e., one input for one output). These computations were performed on a MacBook Pro M1, 3.2Ghz, 16Gb memory.

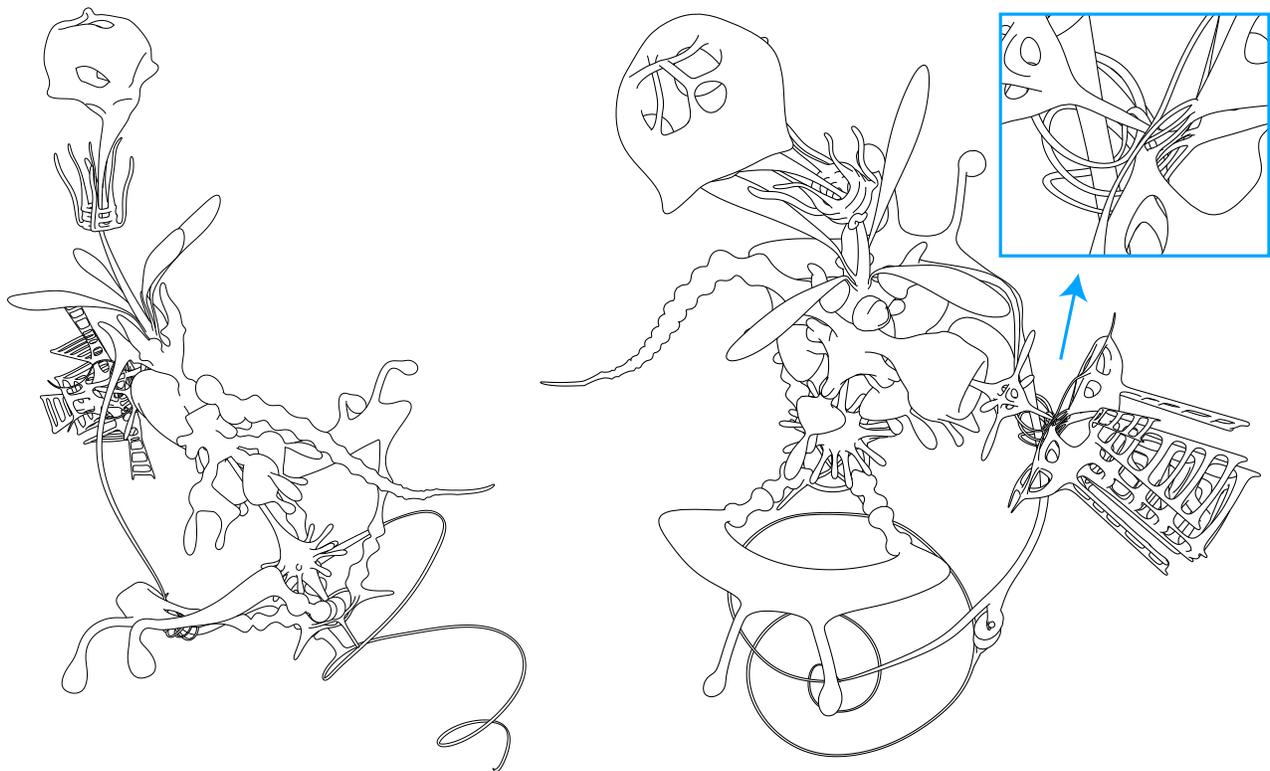


Fig. 18. Two views of the complex “Yeah, Right” model, which has genus 131. (Public domain model by Keenan Crane.)

Georges Winkenbach and David H. Salesin. 1996. Rendering Parametric Surfaces in Pen and Ink. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, 469–476. <https://doi.org/10.1145/237170.237287>

A PROJECTION PRESERVES TRIANGLE ORIENTATION

We now show that the orientation of a triangle is preserved when projecting it from the image plane to 3D. Suppose we define coordinates so that the camera is at the origin $c = (0, 0, 0)$, facing in the $+z$ direction. The vertices of a triangle Δpqr have image coordinates

$p = (p_x, p_y, 1)$, and so on. The orientation of the triangle is given by $(c - p) \cdot ((r - p) \times (q - p)) = \det(-p, r - p, q - p) = \det(p, q, r)$. For clockwise 2D triangles, the orientation is positive: $\det(p, q, r) > 0$

We construct the 3D points by selecting depths $p_z, q_z, r_z > 0$, and then projecting into 3D: $p' = p_z p, q' = q_z q, r' = r_z r$. The orientation of this new triangle is $(c - p') \cdot ((r' - p') \times (q' - p')) = \det(p', q', r') = \det(p_z p, q_z q, r_z r) = p_z q_z r_z \det(p, q, r) > 0$. Thus, the orientation of the 3D triangle is the same as the orientation of

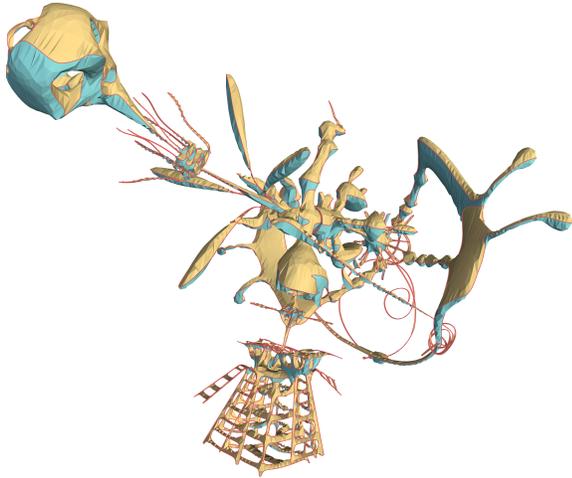


Fig. 19. Side view of the output triangulation of “Yeah, Right,” for the left-most rendering in Figure 18

the 2d triangle: clockwise triangles become front-facing and counter-clockwise triangles become back-facing.

B REMOVING TWISTS

We observe the following types of twists (Figure 20), and use the following heuristics to correct them:

- (1) A very small polygon comprises a simple loop with the wrong orientation (e.g., CW when it should be CCW). In this case, we simply remove the loop from the surface. These polygons are typically smaller than a pixel in image-space, and would be removed during stylization regardless.
- (2) A pair of nearby cusps on a contour is on the wrong side of the contour forming an inverted fishtail. This configuration is detected twice; we check for these before and after testing the other four cases.
- (3) The curve is twisted near a cusp/singularity. We correct this by inserting vertices on each curve at the intersection point, and moving the new vertices apart in image space.
- (4) A skinny portion of the polygon crosses over itself. We insert vertices at the intersection points and shift them apart in image space.
- (5) A cusp on a hole occurs outside the hole. We shift the cusp in image space.

Each of these cases applies only to the contours within a region, e.g., intersections are not detected between regions.

Case 1. If a patch is simple but fails the WSO triangulation, our algorithm removes it if the 2D contour length is below 100px. Otherwise, the patch will be left unchanged, and the algorithm will need to be rerun with a higher subdivision level. To remove a loop, the corresponding faces are assigned to the adjacent patch and the corresponding contour edges are unlabelled.

Case 2. Case 2 is handled by collapsing any loop formed by a 2D intersection (valid or not) that attaches a 2D contour loop whose

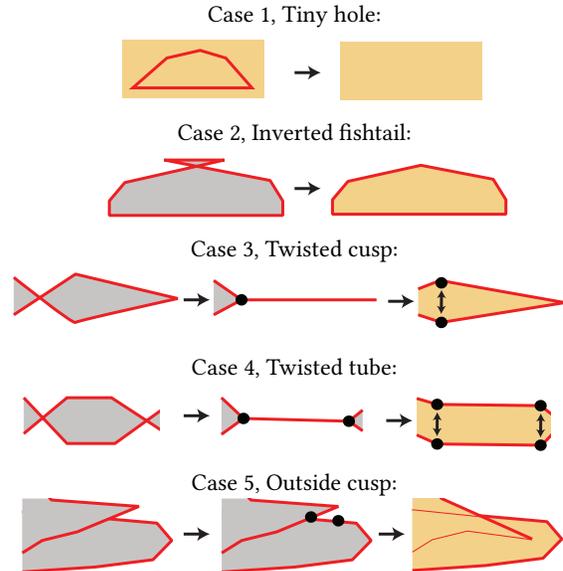


Fig. 20. The five cases of twists that we detect and resolve; see text for details.

2D arc-length is less than 10^{-3} px. Note that this removal step may delete valid fishtails [BH§4.6], but these are subpixel fishtails that would normally be removed during stylization [BH§9.3].

Cases 3, 4, 5. These three cases are detected as follows. First, the algorithm finds all intersections where two contour edges intersect in image space. The following steps are then run on each intersection separately.

An image-space intersection comprises two distinct 3D points on the surface. We first wish to determine if the intersection points are directly connected by geometry, which indicates a twist we may wish to remove. Specifically, we compute a 3D plane that contains the two intersection points and the bisector of the larger angle between the two intersecting edges in 2D. Then we intersect the plane with the region and check if this intersection creates a path between the two intersection points that stays within the region (i.e., does not cross any contours). If the path is valid, then we conclude that there is a twist at this intersection.

The next step is to determine which type of twist occurs. From each intersection point, we can trace around the contour loop until returning to the intersection point. The tracing direction is the one that makes the adjacent front facing patch lie on the left side of the contour. If the sub-loop traced in this process has the wrong orientation (CW or CCW), then that sub-loop has an error. The case depends on how this tracing returns to the intersection: (a) Case 3: return back to the intersection point via a crossing edge. (b) Case 4: meet another invalid intersection point.

While case 3 and 4 are distinguished by whether the tracing returns or reaches another invalid intersection point, case 5 is separated from case 3 and 4 by heuristics. We have the observation that the twisting is caused by under-sampling and thus is supposed to have a small scale.

Let the tracing distance in 2D from an invalid intersection point to either another invalid intersection or itself be D_1 and D_2 where $D_1 = D_2$ in case 5. Let the total lengths of the 2D contour chains containing D_1 and D_2 be c_1 and c_2 respectively. If both tracing paths belong to the same chain, we have $c_1 = c_2$. Since the tracing directions determined by the orientation rule could be incorrect if the under-sampling causes a fishtail to untwist into a Z-shape structure, we also consider the flipped tracing directions. Let the tracing distance of the flipped tracing directions be D'_1 and D'_2 respectively. Let the tracing distance in 2D from the invalid intersection point to the corresponding cusp and its projection be d_1 and d_2 . We categorize the twisting as case 3 if all the following conditions are satisfied:

- Scale violation in case 3 and 4: $\frac{D_1}{c_1} > 0.9$ OR $D_1 > 100\text{px}$ OR $\frac{D_2}{c_2} > 0.9$ OR $D_2 > 100\text{px}$.
- Contrast between the flipped tracing directions and case 5: $\frac{\max(D'_1, D'_2)}{\min(d_1, d_2)} > 4$ OR $\min(d_1, d_2) < 2\text{px}$.
- Safety check in case 5: $\frac{d_1}{c_1} < 0.5$ AND $d_1 < 40\text{px}$ AND $\frac{d_2}{c_2} < 0.5$ AND $d_2 < 40\text{px}$.

If the first condition is satisfied yet either of the latter two is not, we choose to trace in the flipped directions. If the first condition is not satisfied, we trace in the directions initially determined by the orientation rule.

C CUT-TO-DISK ALGORITHM

We now describe our modification to Gu et al. [2002]’s cut-to-disk algorithm to avoid cuts that pass outside the region boundary in image space.

We first mark the following sets of edges that the cut should not pass through: (a) any edge adjacent to any inconsistent face, (b) any edge adjacent to a cusp, and (c) any edge that intersects a nearby contour in image-space. For this last case, we test for intersections between each edge, and an contour edge within its 5-ring, as well as all other contour edges with 5 edges of the contour edge.

For each contour loop in the region, we check if the contour loop has no valid edges coming out of it. In this case, we find a consistent triangle that touches the contour at a single vertex, and split this triangle to produce a usable edge.

We then modify the cut-to-disk algorithm to keep the above edges out of the cut. Specifically, the initial seeds are all faces adjacent to any of the above edges. We keep track of the connected components of faces during region growing. If two regions become adjacent, then they are merged and the edge between them is removed rather than becoming part of the cut. Finally, we apply the same shrinking step from Gu et al.

The initial seeds are all faces adjacent to any invalid edge. Each seed is assigned a unique label and non-seed faces is viewed as unassigned. To indicate the cut result, each edge has a flag showing whether it is in the cut.

In the first phase, the method grows and merges labeled face regions by processing edges adjacent to any assigned face. These edges are stored in a priority queue with three priority levels: 0 to 2, where 2 is the highest one. The priority of an edges is determined based on its two neighboring faces and its validity: (a) priority 2: if the two faces are assigned and the edge is invalid; (b) priority 1: if

one of the two faces is unassigned; (c) priority 0: if the two faces are assigned and the edge is valid. The method iteratively draws from the priority queue until the queue is empty. An edge is processed as follows,

- If the edge has priority 0 or 2, and the two faces have the same label, the edge is in the cut; if the two faces have difference labels, the two labels are merged and the edge is not in the cut.
- If the edge has priority 1, the unassigned face is set to the label of the other face and the edge is set to be not in the cut. The other two edges of the newly assigned face, if have not been processed, are added to the queue with priorities determined as above.

The special case is the boundary edge. They are always set to be in the cut and the region growing stops once reaches these edges.

In the second phase, the method repeatedly removes cut edges adjacent to valence-1 vertex in the same way as the second phase of Gu et al. 2002. Intuitively, this phase removes tree structures from the cut.

This algorithm finds a cut fully consisting of valid edges if it exists, or finds a cut containing invalid edges otherwise. This is ensured by the initialization and the priority order. Note that the initialization enqueues all invalid edges with top priorities and thus they are processed before any other types of edges. An invalid edge would be in the cut if and only if its two neighboring faces have the same label. Since we assign each such face a unique label, this situation would only happen when the corresponding labeled region has a non-disk topology and only has invalid edges in its interior. This means there exists no cut that only consists of valid edges. Once the top priority edges are fully processed, the method continues in a fashion similar to the original Gu et al. 2002. Finally, we apply the same shrinking step from Gu et al.

It is possible that there is a contour loop with no adjacent consistent triangles, and so there is no feasible cut. In this case, the above algorithm will find a cut that touches the loop via an invalid edge. We use three priority levels in the cut-to-disk priority queue, in order to ensure that some cut is found.