Putting Holes in Holey Geometry: Topology Change for Arbitrary Surfaces

Gilbert Louis Bernstein*
University of Washington
Stanford University

Chris Wojtan†
IST Austria

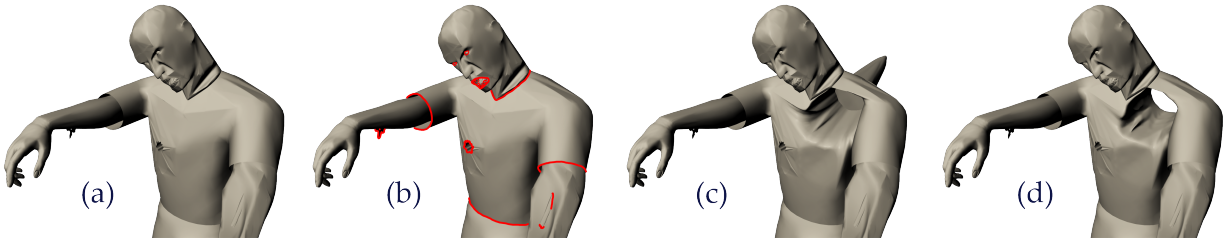


Figure 1: *This zombie model (a) has numerous open surfaces, non-manifold edges, and self-intersections, displayed in red here (b). Nonetheless, using the technique described in this paper, we are able to (c) poke through the zombie's chest and (d) create the desired tunnel/hole.*

Abstract

This paper presents a method for computing topology changes for triangle meshes in an interactive geometric modeling environment. Most triangle meshes in practice do not exhibit desirable geometric properties, so we develop a solution that is independent of standard assumptions and robust to geometric errors. Specifically, we provide the first method for topology change applicable to arbitrary non-solid, non-manifold, non-closed, self-intersecting surfaces. We prove that this new method for topology change produces the expected conventional results when applied to solid (closed, manifold, non-self-intersecting) surfaces—that is, we prove a backwards-compatibility property relative to prior work. Beyond solid surfaces, we present empirical evidence that our method remains tolerant to a variety of surface aberrations through the incorporation of a novel error correction scheme. Finally, we demonstrate how topology change applied to non-solid objects enables wholly new and useful behaviors.

CR Categories: I.3.7 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems;

Keywords: topology, intersections, deformations, sculpting, 3d modeling, non-manifold geometry

Links: DL PDF WEB VIDEO CODE

1 Introduction

Programs for the 3d modeling of surfaces must support ways to change the topology of a surface or else be severely limited in their capabilities. For instance, without some way to edit or change the topology of a surface, it is impossible to model a donut starting from a sphere. The ability to model changes in topology is also critical for the assembly of surfaces from parts, as well as permitting surfaces to merge or split as they are manipulated, to name just a few more consequences.

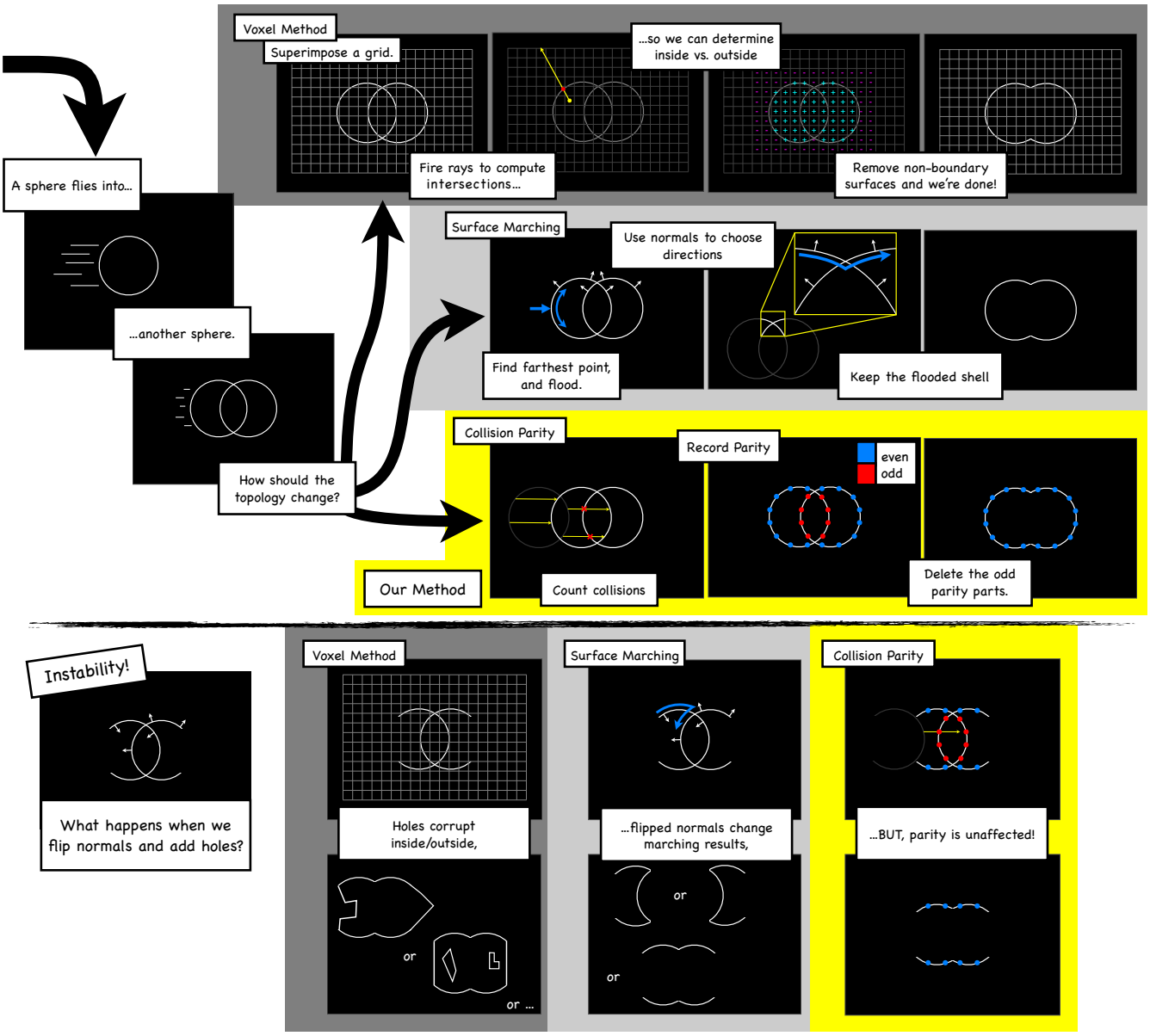
While the ability to change topology is critical for all 3d modeling software, strategies vary widely depending on the representation of the surface and the modeling paradigm in use. For instance, in traditional CAD-derived modeling software like Maya [2013b], or 3DS Max [2013a], special tools allow the user to directly edit the connectivity of the polygons comprising the mesh. Sketch-based modelers in the vein of Teddy [Igarashi et al. 1999] incorporate special stroke gestures which allow users to add tunnels or handles to a surface. Meanwhile voxel-based modeling, exemplified by 3D Coat [2013] or the game Minecraft [2013], naturally incorporates changing topology as a by-product of the representation.

In this paper we propose a novel method for supporting **topology change** in surface-deformation modeling software (e.g. Zbrush [2013b], Sculptris [2013a], Mudbox [2013c]). Like voxel modeling, we would like our topology change to be *incidental*, occurring as a natural side effect of using existing tools/brushes. In contrast, note that CAD-like and sketch-based modelers require specialized tools for topology change. By choosing incidental topology change over specialized tools, we can achieve greater *parsimony* (§8) in our modeling system.

Having made the choice to incorporate topology change incidentally, a number of methods for topology change primarily used in the simulation literature are available to us [Wojtan et al. 2009; Brochu and Bridson 2009]. Unfortunately, these methods all require that the surface represents a solid object—one that can be faithfully represented by a voxel grid. Many surface models available in the wild (over 90% in our measurements §2.2) fail to meet this criterion. One simple example is a height-mapped or planar grid of quadrilaterals. In general, character models and other objects are built to function in 3d applications where skinning, animation, and visual appearance trump other concerns like solidity or physical manufacturability. Like a facade on the set of a spaghetti western, these models have been tailored to tell stories. Compounding this problem, most existing programs do not guarantee that exported models are "solid". So, in order to design a modeling system which fully interoperates with the existing ecosystem, we have to handle *all* surfaces, not just the conveniently solid ones.

To achieve the goal of topology change for arbitrary surfaces, we rely on one key observation: the motion of a surface during editing is sufficient to determine how the topology of that surface should

*e-mail: gilbert@gilbertbernstein.com
†e-mail: wojtan@ist.ac.at



change, even in the absence of reliable surface normals or the enclosure of space—both surrogates for solidity. By simply tracking points on the surface as they move over time and counting the number of times they experience collisions, we can determine whether that part of the surface should be kept around or deleted: a point which collides with the surface an odd number of times is deleted, while one which collides an even number of times is retained.

This simple idea—using collision parity to drive topology change—works well for solid surfaces and many non-solid surfaces as well. However, it can be sensitive to collision detection errors, surface holes, boundaries, and other aberrations. To increase the reliability of our framework in the presence of such imperfections, we introduce an error correction scheme based on graph partitioning.

Contributions We provide the first method for topology change applicable to arbitrary non-solid, non-manifold, non-closed, self-intersecting surfaces. We prove that this new method produces expected, conventional results when applied to solid (closed, manifold, non-self-intersecting) surfaces—that is, we prove a backwards-compatibility property relative to prior work. Beyond solid surfaces, we present empirical evidence that our method remains tolerant to a variety of surface aberrations through the incorporation of a novel error correction scheme. Finally, we demonstrate how topology change applied to non-solid objects enables wholly new and useful behaviors.

2 Related Work & Background

2.1 What Makes a Mesh Solid?

Throughout this paper, we will work with triangle meshes that can be specified as a list of vertex positions and a list of triangles (triples of indices into the vertex list). This rules out the possibility of any isolated vertices or dangling edges, but does allow for a variety of interesting mesh types.

Many surface classifications can be determined locally from looking at the degree of mesh edges (i.e. the number of triangles which ring around an edge). An edge of the mesh is **manifold** if it has

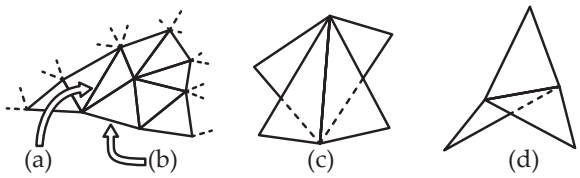


Figure 2: *Mesh type locally depends on the number of triangles incident to an edge: (a) manifold edge, (b) boundary edge, (c) closed edge, (d) non-manifold, non-closed, non-boundary edge*

degree 2, a **boundary** edge if it has degree 1, and **closed** if it has even degree. If all of the edges in a mesh are closed, we say the mesh itself is closed. Not all closed meshes need be manifold.

A second critical property of surfaces is whether or not they are self-intersecting. If any two triangles of the mesh have a non-trivial intersection (that is an intersection along something other than a shared vertex or edge), then we say the mesh is **self-intersecting**.

We say that a mesh which is both closed and non-self-intersecting is **solid**. Solid meshes can also be characterized as those meshes which partition the ambient space $\mathbb{R}^3$ into two parts: those points inside the surface, and those outside of it. Based on this classification, every solid mesh has a canonical surface normal field with normals pointing outward. Sometimes the term "watertight" is found instead of solid. We prefer to use solid, since the definition of watertight is inconsistent and imprecise across the literature.

Outside the class of solid meshes, we say that a mesh is **unoriented** if any pair of triangles sharing a manifold edge are oriented facing opposite directions. There are some meshes which are inherently **non-orientable**, the most famous example being a Möbius band. Non-orientable surfaces cannot be assigned a continuous surface normal field, preventing us from relying on normals as a source of information for topology change of arbitrary meshes.

2.2 Why Aren't All Meshes Solid?

Not only do bad meshes exist in the wild, they are quite common. To demonstrate this fact, we ran statistics on the Brown mesh set [McGuire 2004]: less than 25% $(263/1136)$ of the meshes were closed manifolds, and over 40% $(472/1136)$ were neither manifold, nor closed; over 90% $(1046/1136)$ were self-intersecting. To illustrate various conditions which might arise, we developed the coffee mug example(figure 3).

Besides accidental causes, there are reasons people choose to model these "undesirable" features. In character models for animation,

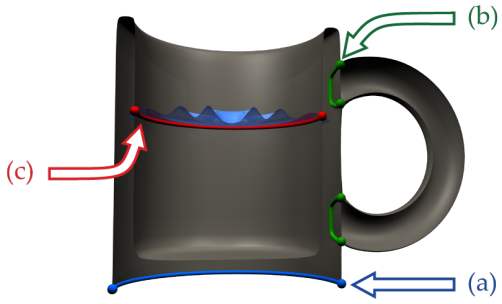


Figure 3: *An example mesh exhibiting "undesirable" mesh properties. Since the artist expects the coffee mug to always sit on a table, they chose (a) to leave the bottom open. Furthermore the handle has been (b) attached to the mug body in a non-closed manner. Lastly, the water surface has been left (c) intersecting the walls of the mug in order to permit flexible animation later.*

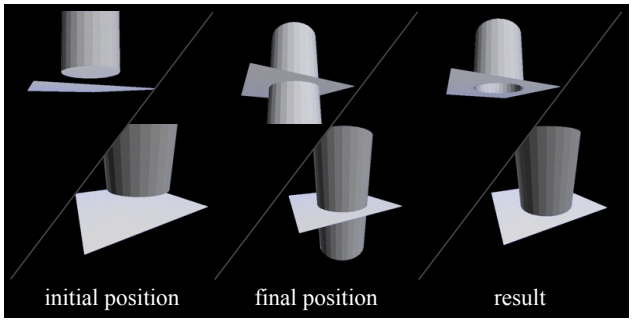


Figure 4: *When a cylinder is plunged downwards into a ground plane, our method reacts as depicted. However, methods which only inspect the final position of the mesh are faced with an ambiguously symmetric problem.*

skinning models regularly requires armpits and other creased areas of geometry to self-intersect in order to achieve reasonable appearances. Thin objects like cloth and flags are regularly modeled as thin sheets, without any intention of representing closed objects. Scanned and reconstructed geometry, especially of large objects like buildings is often incomplete, producing meshes with holes in order to preserve fidelity relative to the raw scan data.

2.3 Prior Work

Most methods available today for computing topology changes frame the problem in the context of mesh-repair [Attene et al. 2013; Ju 2009]. After a surface is deformed, a mesh-repair-like algorithm is run on the surface at the final position. This can be accomplished through converting the surface into an implicit function representation [Wojtan et al. 2009; Wojtan et al. 2010], into a BSP-based volume representation [Campen and Kobbelt 2010] or by marching around the outside of the mesh, using normals [Zaharescu et al. 2011]. All of these techniques are used to infer a volumetric interpretation (inside vs. outside) of the mesh and discard those parts of the surface which are not necessary to enclose the inferred volume. We believe that the dynamesh feature added to ZBrush [2013b] in late 2012 uses a similar kind of volumetric method.

Because these methods (a) force all surfaces to represent solids, and (b) ignore motion data allowing for the inference of what has *changed* between two frames, they are unable to handle arbitrary meshes (see comic figure). All self-intersecting surfaces must be "corrected" and open surfaces must have their holes filled, even when doing so would lead to nonsensical results (e.g. hole-filling a ground plane). These methods cannot disambiguate symmetric geometric arrangements (e.g. a cylinder penetrating a ground plane, figure 4) because they only inspect the final position of the mesh, not the full motion.

More similar to our approach, Brochu & Bridson [2009], and Stãnculescu et al. [2011] make use of motion data. Rather than attempting to repair the final position of the mesh, they attempt to stitch proximate or colliding pieces of the mesh together as the deformation progresses. Brochu & Bridson use a combination of collision detection, local remeshing, and time-step control to get the surfaces of the mesh close but not touching, allowing for a tunnel to be stitched. Similarly, Stãnculescu et al. tightly control the size of both mesh elements and time-steps in order to create safe conditions for tunnel stitching.

Both Brochu & Bridson, as well as Stãnculescu et al. assume that displacements are small and can be rewound as necessary to prevent collision, coercing the rest of their simulation and modeling systems (respectively) to satisfy these constraints. However, nei-

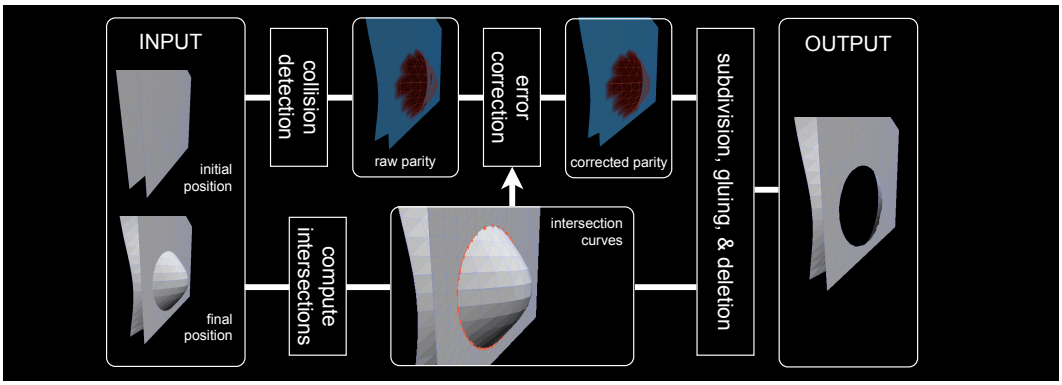


Figure 5: *Overview of our method and its components*

ther of these constraints are necessarily true in a modeling system, as evidenced by our 'grab' brush(§4). Furthermore, these methods closely interweave local surface remeshing with topology change. Such a conflation poses problems when one attempts to integrate topology change with an existing commercial modeler that uses different, and more sophisticated surface re-meshing techniques.

In contrast to prior work, we make no special assumptions about what kinds of meshes our algorithm is presented with.

Currently, artists tend to directly edit the connectivity of their meshes, or make use of Boolean operations (aka. CSG). However, Boolean operations are only defined on closed, orientable surfaces [Requicha 1977]. When working with open surfaces, artists are forced to directly edit the connectivity — no other options are available to them.

3 Method

3.1 Overview

Our approach to topology change relies on existing, well understood computations as components: collision detection, static intersection identification, and triangulation. Perhaps the only exception is a formulation of graph partitioning specialized to serve as a form of error correction. We string these components together in order to compute a **parity field** over the mesh, recording the parity of the number of times that point collided with the surface during the given motion. This field determines whether the surface should be deleted (odd parity) or preserved (even parity). Throughout this process we store an approximation to this (conceptually) continuous field by sampling its value at mesh vertices.

As input for our algorithm, we require a triangle mesh at the initial time/frame and a linear displacement of vertices transporting the mesh to its position at the final time/frame. We begin (figure 5) by running collision detection to compute a **raw parity field**. If we only worked with solid surfaces, this raw field would suffice, but because we expect our mesh to exhibit aberrations or other shortcomings we treat the raw parity field as if it has some (relatively) small number of corrupted entries. To correct these errors and produce a more desirable parity field, we perform an error correction step informed by the mesh's self-intersections at the final frame's position. With the **corrected parity field** in hand, we can then subdivide the mesh, glue the mesh, and delete the appropriate triangles. This results in our final output mesh with suitably altered topology.

3.2 Collision Detection

Our method for topology change relies on knowing the parity of the number of collisions each vertex makes with the rest of the surface. In order to tabulate the number of collisions, we must first run continuous collision detection for every vertex of the mesh.

For every vertex of the mesh and every triangle not containing that vertex, we compute the roots of the usual cubic equation (Appendix A) and the associated barycentric coordinates, and we use them to determine whether any collisions occurred. We arrive at the desired collision parity value by accumulating the results of this computation for each vertex across all potentially intersecting triangles.

In order to accelerate this computation so that less than quadratically many collisions must be tested, we use an acceleration structure, namely an axis-aligned bounding volume hierarchy. We build this AABVH over the line segments traced out by the moving vertices. We then stream the triangles over this structure to identify potential collisions.

Rather than attempt to maintain this acceleration structure between frames, we use a fast, median-split, top-down, divide-and-conquer build inspired by work on real-time ray tracing [Wald 2007]. At each node, a dimension (x, y, or z) is selected and the geometry is rearranged via a quick select search for the median. The build then recurses on the two halves. The resulting build algorithm takes time comparable to a quick sort of the geometry and consumes a small fraction (10%) of the total time spent performing collision detection. Since no heuristics are used to ensure a quality acceleration structure, and since we don't make use of narrow-phase collision culling techniques, we expect that the overall cost of collision detection could be significantly improved by just applying existing methods [Teschner et al. 2005]. Nonetheless, this simple strategy suffices to demonstrate our prototype.

Besides speed, correctness is frequently a problem with collision detection algorithms. When used in applications like cloth simulation, accurate collision detection becomes critical to keep surfaces from snagging on themselves. However, implementing truly robust collisions is a difficult problem. Standard robustness methods (§3.3) are restricted to handling rational arithmetic, but collision detection requires cubic root finding. Brochu et al. [2012] recently gave a very clever solution to this problem, formulating collision detection as a series of predicates relying solely on rational arithmetic.

Here we only need an approximation to the parity field that we compute from collision detection. Since we already intend to run an "error correction" step to tolerate mesh aberrations, we can likewise tolerate the small number of errors in the parity field which result from imprecise collision detection. Unless we are presented

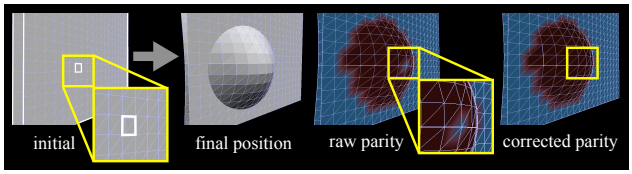


Figure 6: *When surfaces have holes (inset, initial) the raw parity field can be corrupted (raw parity, inset). Running error correction removes these isolated errors.*

with a highly degenerate motion (e.g. the collision of two perfectly aligned grids) these numeric errors are rare. To help ensure that they remain rare, we apply a slight perturbation (similar to perturbations for the intersection computation in the following section). We observed that in the case of two aligned grids, adding a perturbation eliminated 100% of the numeric errors.

3.3 Static Intersections

Once the mesh has reached its final position at the end of a frame of movement, we determine the region where the final mesh statically intersects itself. We find these static intersection curves for two reasons: First, this information is used to guide the error correction procedure. Secondly, the intersection curves are used to cleanly segment the mesh into preserved and deleted portions.

We find the intersection curves in three steps. First, we identify all edge-triangle intersection points present. Second, we infer the set of all intersection edges from these points. Finally, we identify any points formed by the intersection of three triangles.

To efficiently find edge-triangle intersections, we use a second AABVH, like the one computed for collision detection, this time built over the edges of the mesh. Triangles are streamed over this structure and tested for intersection with the edges they encounter.

For each intersecting edge-triangle pair (e, t), we collect the set of triangles t_i with e as an edge. Then, emanating from the point where e and t intersect, there must be exactly one intersection edge on t for each triangle t_i, which is uniquely identified by the pair $\{t, t_i\}$. We accumulate all such pairs identified via this combinatorial generation procedure and eliminate duplicates to form a set of triangle-triangle intersection edges.

To complete the intersection computation, we identify triples of triangles $\{t_1, t_2, t_3\}$ such that $\{t_1, t_2\}$, $\{t_1, t_3\}$, and $\{t_2, t_3\}$ are all intersection edges. Each triple is tested to see whether the three triangles intersect in a point. Triangle-triangle-triangle intersection candidates are rare, so this step takes a negligible amount of time.

Geometric robustness Unfortunately, unlike collisions, we rely on robustly computed intersections. This is because our intersection curves are used as input to a triangle subdivision algorithm. If these curves reveal any inconsistencies, then the subdivision algorithm could segmentation fault or produce non-sensical output.

To ensure consistent results from our intersection computation at a reasonable overhead, we use a variety of techniques from robust geometric computation. First, we use floating point filters [Shewchuk 1997] to efficiently decide the results of most intersection predicates. When these filters fail, we fallback to exact big number arithmetic, ensuring that all predicates are correctly computed. (The actual coordinates of intersection are always computed to machine precision in big number arithmetic.) However, degenerate cases may still occur (e.g. when testing a potential edge-triangle intersection, the edge being tested might pass *exactly* through an edge of the triangle being tested). Because degenerate cases are measure-zero

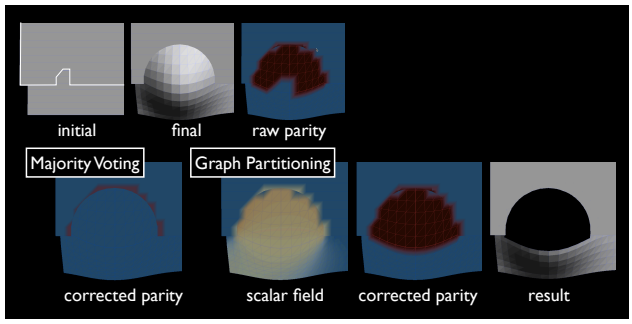


Figure 7: *Majority voting works poorly when intersection curves don't separate the surface. Graph partitioning allows us to introduce new cuts into the mesh.*

events, by definition they can be eliminated through perturbation of the geometry [Seidel 1994]. While symbolic perturbation [Edelsbrunner and Mücke 1990] is one popular way to achieve this end, we instead rely on explicit numeric perturbation of coordinates on a scale well beneath the smallest resolution of triangle edges in use. (We perturb on the order of 10^{-5} units in our prototype code.)

If any degeneracies are detected during the computation of intersections, we abort the computation, perturb all geometry and try again. On our test cases with highly degenerate geometry, this strategy sufficed to eliminate degeneracies after a single perturbation. As with acceleration structures, we chose this strategy for expedient implementation, not because we advocate the choice as the best option for production code.

3.4 Error Correction

"Errors" of various sorts and sources may result in collision detection producing a corrupted parity field. Some of these errors may be due to numeric inaccuracies (§3.2) while others may be due to aberrations (e.g. holes) in the mesh itself. We view the process of computing a more appropriate parity field as a form of error correction. In this section we will propose two such error correction schemes, ultimately discarding the first in favor of the second.

One strategy is suppressing outliers in the parity field through **majority voting**. We split them surface into connected components using the static intersection curves (computed at the mesh's final position) and vote within each component independently. Concretely, we take the vertex-edge graph given by the triangle mesh and remove all edges which are cut by some intersection curve. The connected components of the resulting graph form the connected components of the surface. Within each component, a vertex gets one vote with weight proportional to the area of the surface that it represents. Whichever parity (even or odd) gets more votes is assigned to all vertices in the component.

Majority voting works well when the intersection curves neatly partition the surface into primarily odd or even components. However, when the user manipulates an open surface near its boundary (Figure 7), the resulting intersections rarely partition the surface nicely.

To address this shortcoming, we propose a second error correction scheme based on **graph partioning**. Graph partitioning algorithms label the vertices of a graph with one of two possible labels, splitting the graph. In image segmentation, labels such as foreground/background are used; here we compute even/odd parity labels. The desired labeling is found by solving an optimization problem with both a binary "smoothness" term (minimizing the size of the cut between the two regions) and a unary "accuracy" term. (deviation from the raw parity field) Vision researchers have ex-

plored a range of different formulations and algorithms for graph partitioning [Boykov et al. 2001; Grady and Schwartz 2006].

Following previous examples, we can formulate our graph partitioning problem as follows:

$$\min_x \sum_{(i,j)\in E_o} w_{ij}(x_i - x_j)^2 + \gamma \sum_{i\in V} w_i(x_i - r_i)^2 \qquad (3.1)$$

(Here and for the remainder of this section, we let r_i denote the value of the raw parity field at vertex i, and x_i denote the continuous error corrected value which we solve for. $r_i = 1$ if the vertex is marked even and $r_i = -1$ if it is marked odd. E_o denotes only those edges of the triangle mesh which are not already cut by an intersection edge. Edge and vertex weights (w_{ij}, w_i) are explained at the end of this section.) The energy in equation 3.1 can be optimized by solving a linear system and rounding all $x_i > 0$ to 1; $x_i < 0$ to -1. Alternatively, the energy can be optimized using a min-cut algorithm [Boykov et al. 2001]. We tried both approaches.

In the above formulation, the value of γ is critical. On the one hand, we want to ensure that the solution boundary between the even and odd portions of the surface is smooth (low γ). However, when γ is set too low, accuracy will be ignored. When we played with trying to tweak this balance between the binary and unary terms, we were unable to find a happy medium. Every setting would break some test case.

Further complications arose when we tried to apply the min-cut algorithm. Although we were able to get reasonable results on most test cases, ignoring edges cut by intersections led to failed cases. In the following, we will see how both prying apart values near the intersections (eq. 3.2) and penalizing unused intersections (eq. 3.3) helps to coerce more desirable behavior.

To get around these problems we take a cue from isoperimetric graph partitioning [Grady and Schwartz 2006] and use a two-stage partitioning algorithm. In the first stage, we compute a continuous relaxation of the graph partitioning problem using an energy which prioritizes smoothness. The resulting field encodes a constrained set of possible cuts in the form of its isocontours. By focusing on the smoothness term at this stage, we can limit our search for a cut to only smooth candidates. In the second stage, we select one of these isocontours by choosing how to round the continuous solution into a discrete one. By re-introducing a stronger accuracy term only after we reach this second stage, we ensure a reasonable degree of accuracy while simultaneously guaranteeing smoothness.

In stage 1, we minimize the following quadratic energy within each connected-component:

$$\begin{aligned} & \textstyle\sum_{(i,j)\in E_o} w_{ij}(x_i - x_j)^2 \\ + \quad & \textstyle\sum_{(i,j)\in E_c} w_{ij}(x_i - x_j - (r_i - r_j))^2 \\ + \quad & \gamma \textstyle\sum_{i\in V}(x_i - r_i)^2 \end{aligned} \qquad (3.2)$$

(Appendix B explains how this problem is solved in more detail.) Unfortunately, since we make γ small ($= 10^{-3}$) the solved values x will be unreliably small. To avoid this problem, we introduce a few extra binary terms. Each new binary term is associated with an edge $(i,j) \in E_c$ which is cut by some intersection curve. The extra terms pry apart the values of x on opposite sides of the intersection curve, using the observed difference in the raw parity field. Aside from our dependence on Unfortunately, this heuristic term is necessarily asymmetric, allowing it to be compromised when there are a sizable number of raw parity errors near the intersection curves. This did not turn out to be a significant problem in our test cases.

In stage 2, we sort the vertices by their x values and perform a sweep cut to determine the rounding threshold x^0. Each vertex with

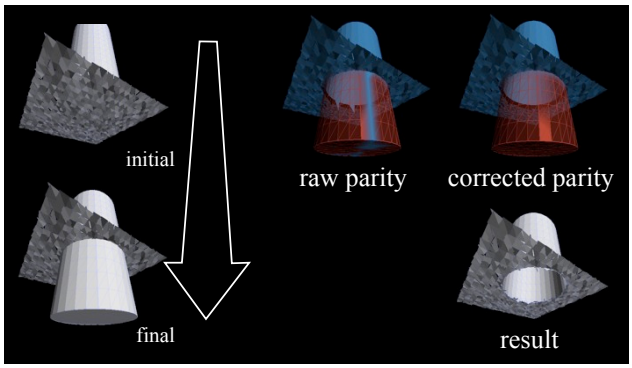


Figure 8: *Our method even works on some triangle soups. In this variation on an earlier case, we disconnect all of the triangles in the ground plane and jitter their vertices; same result.*

$x_i < x^0$ will be rounded down to -1 (odd) and otherwise will be rounded up to 1 (even). We begin with x^0 below all x_i and "sweep" upwards until x^0 is greater than all x_i. We select whichever x^0 minimizes the cut energy:

$$\begin{aligned} & \delta \textstyle\sum_{i\in V} w_i \mathbb{1}[r_i \neq y_i] \\ + \quad & \left(\textstyle\sum_{(i,j)\in E} w_{ij} \mathbb{1} \left[\begin{array}{l} (i,j) \in E_c \text{ and } y_i = y_j \text{ or} \\ (i,j) \in E_o \text{ and } y_i \neq y_j \end{array} \right] \right)^2 \end{aligned} \qquad (3.3)$$

(where y_i is the rounded value of x_i and $\mathbb{1}[\cdot]$ is an indicator function.) Note that we penalize both newly introduced boundary $(i,j) \in E_o$, as well as unused intersections $(i,j) \in E_c$. In order to balance these two terms, we use $\delta = 2\pi$ (half the isoperimetric constant) and square the boundary length term. This results in a geometrically nice, scale-free energy.

Edge and vertex weights Edge weights w_{ij} and vertex weights w_i should be set to suitably encode the geometry of the mesh. Summing all of the edge weights between the even and odd parity vertices should approximate the length of that perimeter curve, while summing the vertex weights should approximate surface area. To accomplish this, we use the following scheme: Conceptually, we can think of each triangle as being divided into 3 quadrilaterals by edges which connect the midpoint of each edge to the barycenter of the triangle. Then, the area of this each quadrilateral can be added to each vertex weight, while the length of the midpoint-barycenter line segment can be added to the appropriate edge weight.

3.5 Subdivision, Gluing, and Deletion

Once we have successfully computed intersection curves and a corrected parity field, we have all the data necessary to actually change the topology of the surface. We complete this change in four steps:

1. *Decide which curves to resolve.* When beginning this step, we have all of the intersection curves already detected. We will discard some of these curves, and add in new curves with the goal of producing a set of curves to separate the even parity portion of the surface from the odd parity portion of the surface. First, we group the intersection curves into connected components. Any connected component of intersections which is adjacent to only even or only odd parity regions is removed from consideration (i.e. it will be left un-subdivided).

Next, we identify all edges of the triangle mesh (not intersection edges) which both separate even and odd parity vertices, but are not themselves crossed by an intersection curve. "Intersection" vertices are inserted midway along each of these edges. Then, we examine all triangles untouched by an intersection curve, but with both even

and odd parity vertices. Each such triangle will have false "intersection" vertices inserted along exactly two edges. We then introduce a false "intersection" edge connecting these vertices.

(Note that the resulting false intersection curves will terminate just before entering a triangle containing an actual intersection curve. This produces slight gaps in the curve separating even and odd parity. Closing these gaps explicitly involves more complex geometric reasoning. Instead, we will fix up this minor shortcoming with a post-process after subdividing.)

2. *Subdivide the mesh.* We solve a constrained triangulation problem on a triangle-by-triangle basis. We accumulate all of the points and edges of intersection lying on each triangle and run Shewchuk's Triangle[1996] to produce a constrained triangulation.

3. *Glue vertices and edges.* Except for the false intersection curves inserted during step 1, intersections necessarily lie on two different parts of the surface. Therefore we must glue together the duplicate curves by replacing duplicated vertices and edges with unique vertices and edges. In effect, this step connects the surfaces along those intersections we did not discard in step 1.

4. *Delete triangles with odd parity.* Finally, we are back to a normal mesh without intersection curves. We must now propagate the parity field onto the triangles and delete the odd portions. This occurs in two sub-steps. First, observe that no triangle in the subdivision can have both odd and even parity vertices after subdivision. This is because we inserted some kind of intersection vertex into every edge of the original mesh with opposite parity endpoints. Taking advantage of this observation, we decide the parity of the majority of triangles by simply copying values from the vertices.

However, because of both the gaps introduced by step 1 and the potential for complicated intersections happening entirely within a single triangle, there may be some triangles which lack parity. To cover these triangles, we perform a 50 round iterated diffusion. The parity at already decided triangles is held constant, while values are propagated between any two adjacent triangles not separated by an intersection curve. Triangles still undecided at the end of this process are marked with even parity. Finally, the odd parity triangles are deleted.

4 Integration with Mesh Modeler

To demonstrate our method of topology change, we constructed a rudimentary surface deformation modeler (See supplementary material). This modeler is equipped with four brushes:

Inflate/Deflate paint to cause instantaneous displacement in the direction of the surface normal

Smooth paint to cause instantaneous explicit Laplacian smoothing

Grab hold onto part of the surface (with a distance-based falloff in influence) and drag it around

Move hold onto an entire connected component of the surface rigidly and drag it around

Between every pair of frames we run an edge-based (split/collapse) remeshing algorithm to dynamically adapt mesh resolution.

For the inflate and smooth brushes, we incorporate topology change in the obvious manner. Between every pair of frames (before remeshing) we run our algorithm. Doing so produces instantaneous topology change.

For the grab and move brushes, user interaction often involves the exploration of different options by dragging through them. If the user decides against taking the action they were contemplating, returning the mouse to the position at which the drag began effectively cancels the action. To incorporate topology change in this scenario, we take a different approach. While the mouse is depressed, we run collision detection each frame, accumulating new collisions into the running tally kept by the raw parity field. Only when the mouse is released do we execute the full topology change algorithm (using the accumulated raw parity) and commit the result. In order to aid the user, we also compute the corrected parity field every frame and display the result as immediate feedback.

5 Analysis

Collision parity tends to produce similar results compared to methods which require solid meshes. In this sense, we can think of parity as being backwards-compatible—semantically—with prior work. In order to allow for a more formal exploration of this claim, we restrict our attention to the case of solid surfaces, where all methods' behaviors are well defined. Given this restricted setting, we can make the following claims.

Theorem 5.1. *Given a solid mesh and a deformation thereof, parity-based topology change will produce a solid mesh.*
Theorem 5.2. *Given two solid meshes undergoing independent rigid motion, parity-based topology change will yield their Boolean union as if computed via CSG.*
Proposition 5.3 (informal)**.** *For most common cases, given a deforming solid mesh, parity-based topology change will compute the expected result, given a solid interpretation.*

In Appendix C, we formalize the final proposition in to a more precise claim, from which the first two claims follow as corollaries. In total, this argument serves to establish a notion of soundness for our method: solid surfaces are handled as expected.

In addition to formal arguments, we also measured the time and space usage of our prototype system. The code for this system is being made available under the LGPL. We are able to operate on meshes of up to $\sim 10,000$ triangles at interactive rates using less than 120MB RAM. The majority of time is spent computing intersections and collision detection, so we expect that the method can be scaled to larger meshes through the use of more aggressive acceleration structures. As a point of comparison, the grid-based method used by Wojtan et al. [2009] can be used for interactive sculpting of solid surfaces with up to $\sim 40,000$ triangles.

6 Applications

Mr. potato head modeling — part assembly (Figure 9) Very easy to use modelers can be built around the idea of assembling a model from existing parts [Chaudhuri et al. 2011; Schmidt and Singh 2010; Hecker et al. 2008]. Such modelers depend on some algorithm to allow users to join various parts together. Methods like the one used in Mesh Mixer [Schmidt and Singh 2010] require that the target surface patch be manifold and disk-like (i.e. simply connected), and that the part/surface being attached has a single loop boundary which can be stitched into this disk.

By contrast the method presented here enables the attachment of parts as a side effect when used with our *move* tool. Unlike the more specialized algorithms, ours is universally applicable, allowing any two surfaces to become attached wherever the user desires.

Pinch closing (Figure 10) Many brushes in mesh sculpting programs may cause the surface to become pinched. For instance, painting a crease in the surface with a displacement along normal

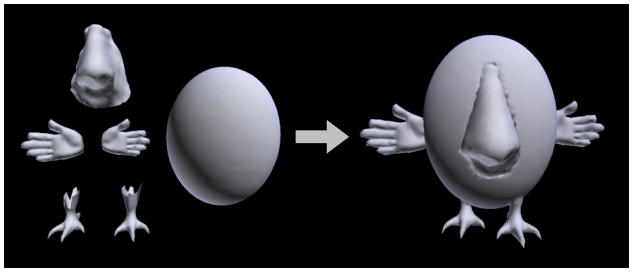


Figure 9: *Using topology change to model-by-parts. The hands are closed, while the feet and nose are open surfaces.*

("inflate") brush often causes the surface to pinch. When topology change is added to the system, these pinches are correctly deleted as they close up. This ensures that the surface remains well behaved as successive operations are applied near the crease.

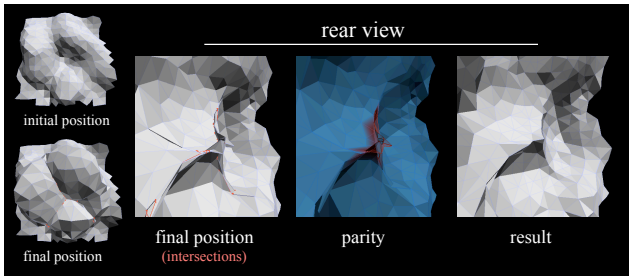


Figure 10: *Topology change helps clean up undesirable features that arise while sculpting, like pinched surfaces.*

Paper mache (Figure 11) While not designed for this purpose, topology change can be used to join two surfaces by slotting one into the other. In conjunction with a Laplacian smoothing brush, surfaces can be joined and smoothed together as if one were paper mache-ing an object.

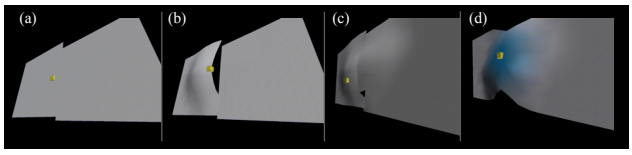


Figure 11: *An improvised use of topology change. By pulling a surface edge around (b) and repeatedly into another surface, (c) the two surfaces become connected. Once connected, the 'smooth' brush with dynamic remeshing can be used (d) to clean up the join, achieving a paper-mache-like process.*

More complex example (Figure 12) This example exhibits models of a cow and two wings consisting of about 25,000 triangles in total. This cow model was purchased by a colleague as part of a mesh set. As provided, the cow model is composed of hundreds of disconnected mesh components, many of which intersect each other. Needless to say, this presents a challenging case for topology change, representative of the issues which arise in practice. Nevertheless, our method is able to successfully join the wing to the cow body.

7 Limitations

Formal properties of our method are proven for the continuous setting. However, the discretization of the parity field which we use (sampling only at mesh vertices) can miss important collisions. For instance, when a cylinder is plunged into a single large triangle (Figure 13), no parity samples lie on the triangle, within the cylinder. The result is incorrect topology change. This shortcoming can be eliminated by adaptively refining the mesh in response to collisions. In the particular scenario we give, adding a single vertex to the large triangle (placed inside the cylinder) would result in correct topology change. In order to keep our prototype implementation simpler, we chose not to address this issue.

We have shown some examples of our method dealing with multiple disconnected mesh components (Figures 8 and 12). Our method is able to successfully handle some of these cases, but arbitrary triangle soups may still pose a problem because the error correction algorithm in Section 3.4 assumes a connected graph.

Our method is designed for general purpose use, not tailored to specific applications. Unfortunately, this choice also leads to unexpected behavior in some more specific contexts. For instance, when assembling models from existing parts, some of those parts will have open boundaries on the side where they are attached. This can lead to the preservation of occluded surface components whose presence the user is unaware of. To take another example, when mimicking a paper mache process, bringing two sheets together will often create tunnels/holes in the process of connecting the two surfaces. A different form of topology change more focused on merging colliding surfaces would probably be more appropriate.

Another issue arrises when self-intersecting geometry begins to move. Currently, we do not resolve self-intersections so long as they remain static. However, once the intersecting portion of the surface begins moving, the resulting raw parity field may overwhelm the suppressing effect of error correction and cause holes to open in the mesh as a result.

When discussing this system with others, we repeatedly received opinions that our system (or variants of it) produce "intuitive" or "unintuitive" behavior. Frequently, we would receive both opinions about the same example. When we restrict ourselves to the case of solid objects, these disagreements disappear, but in non-solid cases disagreement is common. Consequently, we observe that the goal of producing "intuitive" behavior for a method applicable to arbitrary surfaces is likely to be a fool's errand. None-the-less we can propose methods for topology change which may be useful to artists and whose idiosyncrasies can be learnt over time.

Some of these shortcomings could probably be addressed (at least in particular cases) with specialized hacks. However, from our experience experimenting with error correction schemes, we observed that fixing one problem generates a host of others. In the interest of keeping our approach simple and focused, we chose not to explore complicating fixes.

Some of these issues, like those with paper mache-ing, go beyond hacks. In that case, a different kind of topology change with a different behavior might be more appropriate. Consider the behavior of a soap film when two soap bubbles collide at low speed. Rather than annihilate each other, the two colliding surfaces merge into a single interface. Examples like this one suggest that multiple different kinds of topology change behavior are necessary to satisfy users' expectations. One interesting direction for future research is to explore how to organize and categorize different kinds of topology change, working under the premise that there cannot be one universally appropriate behavior.

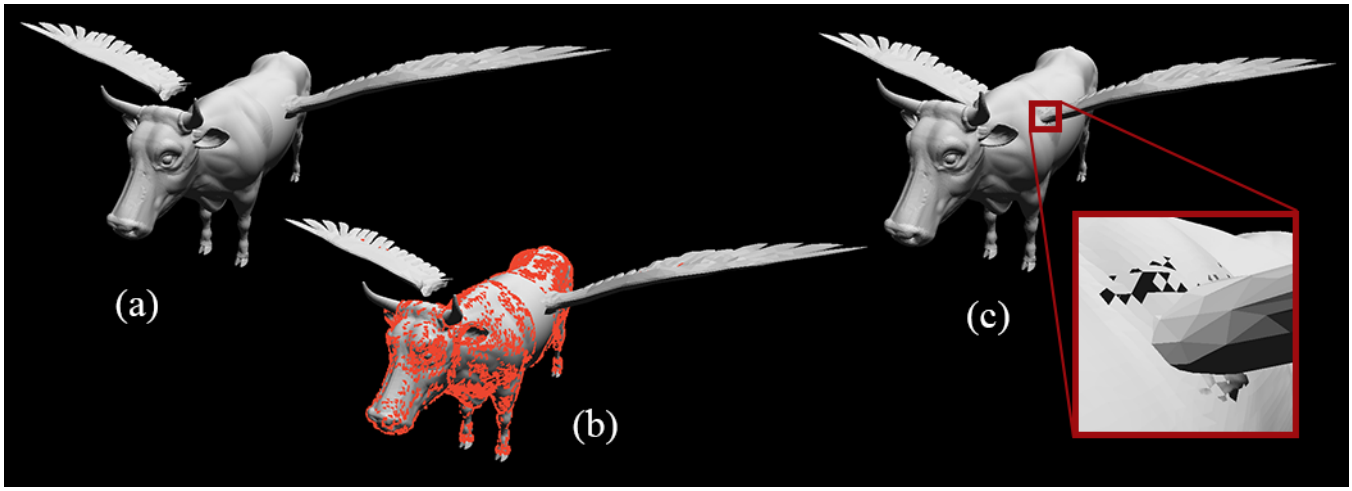


Figure 12: *We attach a pair of wings to a cow model (a) composed of many disconnected components, intersecting as shown (b) in red. Nonetheless, we are able to (c) compute topology change. (Inset) Error correction cannot fix completely disconnected components.*

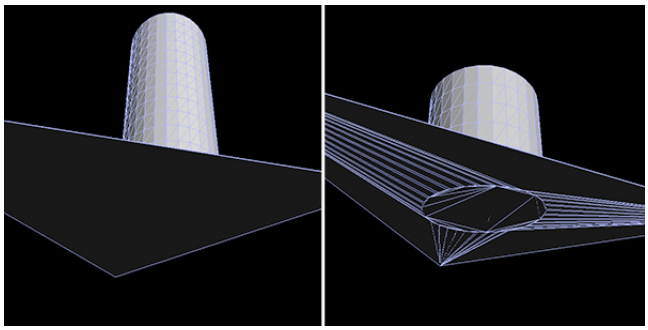


Figure 13: *A failure case of our algorithm. A closed cylinder plunges downward through a single giant triangle (left). Because there are no parity samples on the triangle itself, our algorithm does not properly delete the hole caused by the cylinder. Instead, it causes a non-manifold junction between both models (right).*

8 Conclusions and Outlook

In closing, we have provided a method for computing topological changes on triangle meshes that are not subject to the typical restrictions common among most geometric algorithms. Our method does not require the input meshes to represent solid objects, so it is more widely applicable than most existing algorithms. To further improve the method's reliability, we provide a novel error correction mechanism that both handles poor-quality meshes and tolerates inaccuracies in collision detection computations.

Because our method is error-tolerant and applicable to such a wide range of potential inputs, we believe that it can integrate nicely into geometric modeling applications of the future. The algorithm's efficiency and simplicity allow topology change to happen naturally with any deformation tool, instead of needing to be explicitly effected through a special tool.

As a consequence of this ease of integration, we believe our method has the potential to significantly improve the standard work flow of a geometric modeler. Once a user understands the general idea of a behavior like topology change and how their different tools can be used to deform a surface, they can discover new strategies through exploration and continued use. By way of comparison, sketch-based modeling programs must incorporate special new gesture types, which the user must then learn how to execute in order to introduce tunnels or handles in an existing surface. These tools are less likely to lead the user to discover serendipitous uses or combinations. We see our method as a way to easily incorporate topology change into a 3d modeler, similar to the way Harmon et al. [2011] incorporate collision detection and response.

We suspect that this line of thinking may lead to further innovations in 3d modeling tools. How can we incorporate more behaviors, (e.g. topology change, collision response) natural or otherwise, with compounding effects into our tools? How few tools and behaviors can we use to build a parsimonious, compelling and flexible modeling system?

Acknowledgements

Most of all we would like to thank Zoran Popovic for his help and guidance with earlier iterations of this research. Thank you to the researchers involved in compiling and maintaining both the Brown Mesh Set and Princeton Shape Benchmark. Thank you also to Siddhartha Chaudhuri for providing the cow model we used. This research was supported via the NSF GRFP DGE-0718124.

References

3D-Coat, 2013. 3D-Coat.

Attene, M., Campen, M., and Kobbelt, L. 2013. Polygon mesh repairing: An application perspective. *ACM Computing Surveys*. To appear.

Autodesk, 2013. 3ds Max.

Autodesk, 2013. Maya.

Autodesk, 2013. Mudbox.

Boykov, Y., Veksler, O., and Zabih, R. 2001. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell. 23*, 11 (Nov.), 1222–1239.

Brochu, T., and Bridson, R. 2009. Robust topological operations for dynamic explicit surfaces. *SIAM J. Sci. Comput. 31*, 4, 2472–2493.

Brochu, T., Edwards, E., and Bridson, R. 2012. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph. 31*, 4 (July), 96:1–96:7.

CAMPEN, M., AND KOBBELT, L. 2010. Exact and robust (self-)intersections for polygonal meshes. *Computer Graphics Forum 29*, 2, 397–406.

CHAUDHURI, S., KALOGERAKIS, E., GUIBAS, L., AND KOLTUN, V. 2011. Probabilistic reasoning for assembly-based 3D modeling. *ACM Transactions on Graphics (Proc. SIGGRAPH) 30*, 4.

EDELSBRUNNER, H., AND MÜCKE, E. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (TOG 9*, 1 (Jan).

GRADY, L., AND SCHWARTZ, E. L. 2006. Isoperimetric graph partitioning for image segmentation. *IEEE Trans. on Pat. Anal. and Mach. Int 28*, 469–475.

HARMON, D., PANOZZO, D., SORKINE, O., AND ZORIN, D. 2011. Interference-aware geometric modeling. *ACM Trans. Graph. 30* (Dec.), 137:1–137:10.

HECKER, C., RAABE, B., ENSLOW, R. W., DEWEESE, J., MAYNARD, J., AND VAN PROOIJEN, K. 2008. Real-time motion retargeting to highly varied user-created morphologies. In *ACM SIGGRAPH 2008 papers*, ACM, New York, NY, USA, SIGGRAPH '08, 27:1–27:11.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3d freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '99, 409–416.

JU, T. 2009. Fixing geometric errors on polygonal models: a survey. *J. Comput. Sci. Technol. 24*, 1 (Jan.), 19–29.

MCGUIRE, M. 2004. Observations on silhouette sizes. *journal of graphics, gpu, and game tools 9*, 1, 1–12.

MOJANG, 2013. Minecraft.

PIXOLOGIC, 2013. Sculptris.

PIXOLOGIC, 2013. ZBrush.

REQUICHA, A. A. G. 1977. Mathematical models of rigid solid objects. Tech. Rep. TM-28, Production Automation Project, University of Rochester, Rochester, New York 14627, November.

SCHMIDT, R., AND SINGH, K. 2010. meshmixer: an interface for rapid mesh composition. In *ACM SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, SIGGRAPH '10, 6:1–6:1.

SEIDEL, R. 1994. The nature and meaning of perturbations in geometric computing. In *STACS '94: Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag, London, UK, 3–17.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Selected papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, Springer-Verlag, London, UK, UK, FCRC '96/WACG '96, 203–222.

SHEWCHUK, J. R. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry 18*, 3 (Oct.), 305–363.

STÃNCULESCU, L., CHAINE, R., AND CANI, M.-P. 2011. Smi 2011: Full paper: Freestyle: Sculpting meshes with self-adaptive topology. *Comput. Graph. 35*, 3 (June), 614–622.

TESCHNER, M., HEIDELBERGER, B., MANOCHA, D., GOVINDARAJU, N., ZACHMANN, G., KIMMERLE, S., MEZGER, J., AND FUHRMANN, A. 2005. Collision handling in dynamic simulation environments. In *Eurographics 2005: Tutorial Notes*.

WALD, I. 2007. On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, USA, RT '07, 33–40.

WOJTAN, C., THÜREY, N., GROSS, M., AND TURK, G. 2009. Deforming meshes that split and merge. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, ACM, New York, NY, USA, 1–10.

WOJTAN, C., THÜREY, N., GROSS, M., AND TURK, G. 2010. Physics-inspired topology changes for thin fluid features. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, 1–8.

ZAHARESCU, A., BOYER, E., AND HORAUD, R. 2011. Topology-adaptive mesh deformation for surface evolution, morphing, and multiview reconstruction. *IEEE Transactions on Pattern Analysis and Machine Intelligence 33*, 823–837.

Appendix A

Given a point p and triangle (a, b, c) in motion from p_0 to p_1 and from (a_0, b_0, c_0) to (a_1, b_1, c_1), the point collides with the triangle's plane whenever the triple product

$$\begin{bmatrix} (1-t)(a_0-p_0)+t(a_1-p_1) & ; \\ (1-t)(b_0-p_0)+t(b_1-p_1) & ; \\ (1-t)(c_0-p_0)+t(c_1-p_1) & \end{bmatrix}$$

vanishes. This produces a cubic equation in t whose roots are found via interval search. Candidate intersection points are then tested for containment in the triangle and in the time interval $(0, 1)$. Standard floating-point arithmetic is used. Exact methods [Brochu et al. 2012] could be used instead to avoid inaccuracies due to floating-point, but are unnecessary due to error correction.

Appendix B

The quadratic energy used during error correction is minimized by solving a linear equation $Ax = b$ using a sparse Cholesky solver. This system can be set up using the following formulas.

$$\begin{aligned} A_{ij} &= \begin{cases} 0, & (i,j) \text{ is cut with } r_i = r_j \\ -w_{ij}, & \text{otherwise} \end{cases} \\ A_{ii} &= -\sum_{j \neq i} A_{ij} + \gamma \\ b_i &= \gamma r_i + \sum_j w_{ij}(r_i - r_j) \text{ where } (i,j) is \text{ cut and } r_i \neq r_j \end{aligned}$$

Appendix C

We begin by defining and/or recalling a few concepts related to point in polygon tests. After that, we present a lemma and corollaries.

Recall that every solid surface (closed and non-self-intersecting) has a canonical normal field with normals pointing outward(§2.1). In the following, we assume that all solid surfaces are equipped with this canonical normal field. Furthermore, we assume that every surface begins and ends its motion in general position. We make this

assumption here to simplify the analysis. In our code, we use perturbation and exact geometric predicates to treat general position-related issues—we do not ignore them.

Given a closed surface equipped with a normal field, we define the **containment number** $\omega(p)$ of a point p not on the surface as follows: Select any continuous path starting at p and diverging towards infinity. If this path never crosses the surface, then $\omega(p) = 0$. Otherwise, we may compute $\omega(p)$ by the following procedure. Start with $\omega(p) = 0$. Then, trace along the path beginning at p. Every time the path crosses the surface, increment the containment number if we cross the surface in the same direction as the surface normal, and decrement the containment number if we cross in the opposite direction. For closed surfaces the quantity computed by this procedure is invariant to the choice of path. As the surface moves, the containment number at p is preserved so long as the surface does not pass over p. So, as the surface moves, it will adjust the containment number at p only as it passes over p. The rules for this update derive from the above definition. If the surface passes over p traveling in the direction of its surface normal, then the containment number at p is incremented; otherwise it is decremented. In 2d, this containment number is equivalent to the winding number of a curve.

Containment numbers can be used to classify the space around a surface into space contained *inside* the surface ($\omega(p) = 0$) and space *outside* the surface ($\omega(p) = 1$). This **binary interpretation** of containment numbers suffices for solid surfaces; we consider two possible extensions to self-intersecting but closed surfaces:

- The **sign interpretation** says that a point p is inside if $\omega(p) > 0$ and outside if $\omega(p) \leq 0$.
- The **modulo interpretation** says that a point p is inside if $\omega(p) \equiv 1$ or $2 (\text{mod } 4)$ and inside if $\omega(p) \equiv 0$ or $-1 (\text{mod } 4)$

Of these two interpretations the sign interpretation is probably more natural. However, without arguing their relative merits, we can observe that so long as $\omega(p) \in \{-1, 0, 1, 2\}$, both interpretations agree. (i.e. so long as the containment number deviates by at most one from the expected values 0 and 1)

Lemma 8.1. *Given an initially solid surface undergoing deformation, tracking collision parity results in topology change equivalent to the modulo interpretation for containment numbers.*

Proof sketch. Consider a point p of the surface undergoing deformation, a point p^+ displaced infinitesimally away from p in the normal direction and a point p^- displaced infinitesimally far away from p in the opposite direction. Before the surface begins to move $\omega(p^+) = 0$ and $\omega(p^-) = 1$. During movement p collides with other pieces of the surface an even or odd number of times.

If p collides an even number of times, then the containment numbers of p^+ and p^- must both change by the same amount: some multiple of 2, say $2k$. So either $\omega(p^+) \equiv 0 (\text{mod } 4)$ while $\omega(p^-) \equiv 1 (\text{mod } 4)$, or $\omega(p^+) \equiv 2 (\text{mod } 4)$ while $\omega(p^-) \equiv -1 (\text{mod } 4)$. Under the modulo interpretation of containment number, this means that either p^+ is outside while p^- is inside, or vice-versa. So, under the modulo interpretation we would choose to not delete points p on the surface which collide an even number of times.

If p collides an odd number of times, then the containment numbers of p^+ and p^- must both change by the same amount: $2k + 1$. So either $\omega(p^+) \equiv 1 (\text{mod } 4)$ while $\omega(p^-) \equiv 2 (\text{mod } 4)$, or $\omega(p^+) \equiv -1 (\text{mod } 4)$ while $\omega(p^-) \equiv 0 (\text{mod } 4)$. Under the modulo interpretation of containment number, this means that either p^+ and p^- are both inside, or they are both outside. In either case, we would choose to delete p since it does not serve to separate inside from outside. □

The first theorem(§5.1) follows directly from this lemma, since we just demonstrated that parity based topology change can be interpreted as providing an inside/outside classification for closed surfaces.

Corollary 8.2 (Theorem 5.2)**.** *Given two solid meshes undergoing independent rigid motion, parity-based topology change will yield their Boolean union as if computed via CSG.*

Proof. Consider two solid meshes undergoing independent rigid motion. When they are done moving every point p of space is either outside of both meshes, inside one of the meshes or inside both. Thus $\omega(p) \in \{0, 1, 2\}$. Under the modulo interpretation of containment number, we interpret the value 0 as outside and both 1 and 2 as inside. Leveraging lemma 8.1, we know that computing parity-based topology change will yield a solid mesh under the modulo interpretation. Therefore this resulting mesh will represent the Boolean union of the two solid meshes in question. □

Finally, to the extent that we are willing to accept the informal proposition (i.e. observation) that $\omega(p)$ is often confined to the set of values $\{-1, 0, 1, 2\}$, then we observe that computing parity-based topology change for initially solid surfaces often yields the same result as under a sign interpretation of containment numbers.