Mandoline: Robust Cut-Cell Generation for Arbitrary Triangle Meshes

MICHAEL TAO, University of Toronto, Canada
CHRISTOPHER BATTY, University of Waterloo, Canada
EUGENE FIUME, Simon Fraser University, Canada and University of Toronto, Canada
DAVID I.W. LEVIN, University of Toronto, Canada

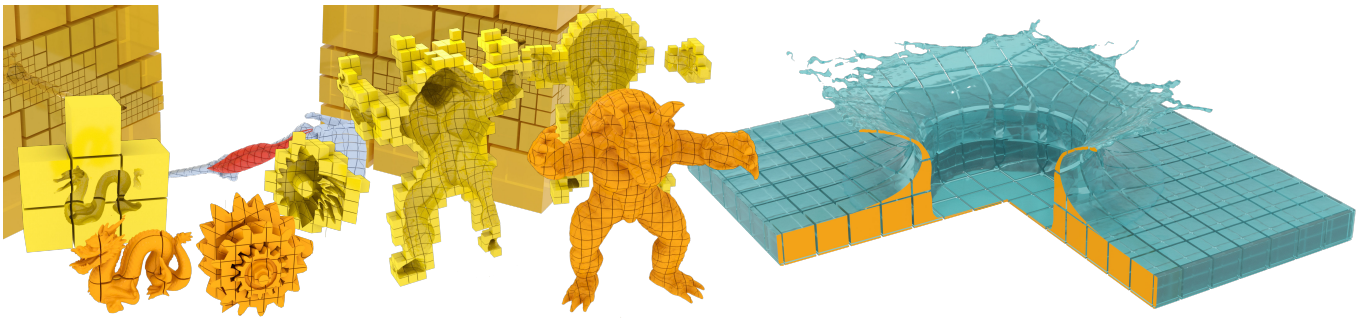


Fig. 1. Various cut-cell meshes generated by our method based on surface geometries drawn from both graphics and engineering. To visualize the cut-cells, we slice the meshes along user-defined cut planes, omitting empty background grid cells except in the adaptive case.

Although geometry arising "in the wild" most often comes in the form of a surface representation, a plethora of geometrical and physical applications require the construction of volumetric embeddings either of the geometry itself or the domain surrounding it. Cartesian cut-cell-based mesh generation provides an attractive solution in which volumetric elements are constructed from the intersection of the input surface geometry with a uniform or adaptive hexahedral grid. This choice, especially common in computational fluid dynamics, has the potential to efficiently generate accurate, surface-conforming cells; unfortunately, current solutions are often slow, fragile, or cannot handle many common topological situations. We therefore propose a novel, robust cut-cell construction technique for triangle surface meshes that explicitly computes the precise geometry of the intersection cells, even on meshes that are open or non-manifold. Its fundamental geometric primitive is the intersection of an arbitrary segment with an axis-aligned plane. Beginning from the set of intersection points between triangle mesh edges and grid planes, our bottom-up approach robustly determines cut-edges, cut-faces, and finally cut-cells, in a manner designed to guarantee topological correctness. We demonstrate its effectiveness and speed on a wide range of input meshes and grid resolutions, and make the code available as open source.

CCS Concepts: • **Mathematics of computing → Mesh generation**; • **Computing methodologies → Volumetric models**.

Authors' addresses: Michael Tao, mtao@dgp.toronto.edu, University of Toronto, Toronto, Canada; Christopher Batty, christopher.batty@uwaterloo.ca, University of Waterloo, Waterloo, Canada; Eugene Fiume, efiume@sfu.edu, Simon Fraser University, Burnaby, Canada, University of Toronto, Toronto, Canada; David I.W. Levin, diwlevin@cs.toronto.edu, University of Toronto, Toronto, Canada.

0730-0301/2019/11-ART179 $15.00
https://doi.org/10.1145/3355089.3356543

Additional Key Words and Phrases: cut-cells, volumetric meshing

ACM Reference Format:
Michael Tao, Christopher Batty, Eugene Fiume, and David I.W. Levin. 2019. Mandoline: Robust Cut-Cell Generation for Arbitrary Triangle Meshes. *ACM Trans. Graph.* 38, 6, Article 179 (November 2019), 17 pages. https://doi.org/10.1145/3355089.3356543

1 INTRODUCTION

Mesh generation is a fundamental geometric problem with broad applications. Its task is to subdivide a given spatial domain, described by its boundary, into a set of non-overlapping elements or cells. The resulting *mesh* is typically employed as the basis for discretizations of partial differential equations drawn from diverse application domains such as solid mechanics, thermodynamics, electromagnetism, or fluid dynamics. A plethora of geometric tasks also exploit volumetric discretizations, such as skinning weights [Dionne and de Lasa 2013], functional maps [Ovsjanikov et al. 2012], topology optimization [Lian et al. 2017], and surface tracking [Wojtan et al. 2010].

One popular mesh generation strategy, particularly common in fluid dynamics, is the use of Cartesian *cut-cell* meshes. These are constructed by clipping the cells of a uniform Cartesian grid against the given input boundary geometry, which in many cases can be described by a triangle mesh. This ensures that every strictly interior cell is a perfect cube, which is convenient for memory efficiency and discretization accuracy; at the same time, the surface geometry of the input domain is preserved by the boundary-conforming polyhedral cut-cells created by the clipping process. There exists today a wide variety of discretization strategies appropriate for cut-cell or polyhedral meshes, including traditional embedded/immersed

boundary/interface methods [Lee and LeVeque 2003; Mittal and Iaccarino 2005], polyhedral (dis)continuous Galerkin methods [Antonietti and Mazzieri 2018; Martin et al. 2008], and structure-preserving mimetic methods [Lipnikov et al. 2014], among many others.

While the basic task of cut-cell mesh generation is simple to describe, designing an ideal algorithm is highly non-trivial due to several conflicting criteria. It should be *robust* to degenerate and near-degenerate situations that arise frequently in real-world geometry, such as input vertices, edges, or faces coinciding with components of the uniform grid. It should be *fast*, since many practical use cases involve time-dependent domains and therefore require that mesh generation be performed at every time step. It should be *general*, in the sense that relatively few requirements should be placed on the input triangle mesh and its relationship to the regular grid. Our definition of generality includes support for: (1) open meshes whose boundaries yield partial cuts, (2) non-manifold triangle meshes, (3) meshes with small or infinitesimal thickness leading to fully split cells, and (4) meshes with relatively fine topological features, leading to sub-cell tunneling, nesting, and branching.

The tension among these requirements has led to a variety of algorithms and codes that make compromises on one criterion or another. For example, converting the input geometry to a signed distance field reduces the problem to marching cubes cases [Mittal and Iaccarino 2005] but destroys geometric detail and precludes open/non-manifold meshes. Adopting expensive but robust Boolean routines designed for intersecting arbitrary triangle meshes [Zhou et al. 2016] fails to exploit the inherent axis-aligned planar structure of uniform grids, while placing conditions on the winding number of the input mesh. Circumventing the challenges of geometric degeneracies by random numerical perturbations of the grid has been suggested in the past [Edwards and Bridson 2014; Kim and Tautges 2010], but this provides no guarantees and cannot avoid many topologically challenging cases we consider.

1.1 Contribution

Our contribution is *Mandoline*, a cut-cell generation algorithm that is empirically robust, fast, and general. It partitions input triangle mesh geometry agnostic of input topology and uses only local planar structures to produce valid cut-cell meshes with correct connectivity. We demonstrate its efficacy by generating cut-cells from thousands of input meshes including geometrically complex open and closed surfaces at various grid resolutions. To further both the use of, and research into, cut-cell methods, we will release the source code for Mandoline under a permissive software license.

Our bottom-up approach is demonstrated to be robust despite being completely implemented with floating point numbers and arithmetic. This is achieved through a design that minimizes dependency on numerical vertex positions and the geometric structure of the mesh, and instead relies on the combinatorial mesh structure wherever possible. Core to this approach is a local vertex representation that includes bitmasks identifying whether vertices lie precisely on grid vertices, edges, or faces. Early in our algorithm this vertex representation allows the identification of redundant vertices when evaluating mesh intersections along different axis-aligned planes, and later on helps to safely accelerate additional collinearity, coplanarity, and redundancy checks. In our broad testing, Mandoline always generated consistent and correct geometry.

2 RELATED WORK

While unstructured and semi-structured meshes of tetrahedra [Hu et al. 2018; Si 2015] or hexahedra [Ray et al. 2018] are popular for a wide range of graphical and scientific computing needs, cut-cell meshes are also widely used for applications such as fluid mechanics [Azevedo et al. 2016; Edwards and Bridson 2014], isogeometric analysis [Safdari et al. 2016], and elasticity simulation [Patterson et al. 2012; Theillard et al. 2013]. Cut-cells are particularly attractive for application domains (like fluids) in which meshes must be quickly regenerated in order to track moving features of a simulated material. The cut-cell approach means the mesh generation effort is concentrated near the moving boundary while interior cells maintain a convenient structured arrangement. Berger [2017] provides a useful recent review of this methodology. Cut-cell approaches can be characterized based on their basic cut-cell representation along with the types of input they can reliably digest (Table 1).

Cell Geometry: Modern cut-cell approaches use a variety of boundary representations. Some use only *hexahedral* cells [Kim and Tautges 2010] at the boundary and represent the cut itself implicitly via either *cut edge lengths* or *area/volume* fractions. Others reduce the boundary to a single planar segment [Colella et al. 2014], or allow only marching cubes-type cell structures [Ferstl et al. 2014; Meinke et al. 2013]. Still other methods, including Mandoline, explicitly split or clip the underlying hexahedral grid cells against the boundary, storing detailed, boundary-conforming cells [Azevedo et al. 2016; Edwards and Bridson 2014]. In these cases, ensuring robustness is a perennial challenge; for example, Edwards and Bridson resort to perturbing the underlying grid at each simulation step to circumvent degeneracies.

In the context of surface triangulations, intrinsic Delaunay triangulations [Fisher et al. 2007] require data structures that augment a coarse discretization of a domain with a second, finer one. Mandoline's intrinsic cut-vertex representation shares similar qualities with the *signpost* data structures used in recent work on these triangulations [Sharp et al. 2019].

Several discretization schemes, such as virtual node algorithms and extended finite element methods, use quadrature based on duplicated/virtual tetrahedra for each cut-cell (e.g., [Koschier et al. 2017]) by multiplying each tetrahedral basis function by a characteristic function for the cut-cell. Like the regular grid case, various restrictions on valid cuts have been considered, from requiring each cut-cell to include an input vertex [Molino et al. 2004] to fully general cutting of tetrahedra with triangle soups [Sifakis et al. 2007]. In particular, the method of Sifakis et al. shares some broad similarities with Mandoline. However, it assumes manifold cut geometry, does not benefit from our axis-aligned plane structure, and omits details of the complex topological stitching process (*"boundary tracking"*) needed to produce a valid final cut-mesh.

Non-Manifold Meshes: General non-manifold configurations, such as an edge shared by several surrounding triangles, can readily arise

Table 1. Algorithms for cut-cell mesh generation and their support for important features. "Not Done" denotes features that are not demonstrated in publication but are theoretically achievable. "?" denotes that the feature was not demonstrated or discussed in literature. "Open" denotes support for open input meshes. "Non-manifold" indicates the code directly accepts non-manifold inputs. "Intersect" indicates that the code handles intersecting (component-wise) inputs.

Algorithm	Cell Data	Split Cells	Tunnels	Adaptive	Open	Non-Manifold	Intersect	Code
Cart3D [Aftosmis et al. 1998]	Polygons	Yes	No	Yes	No	No	Yes	No
EB-Chombo [Colella et al. 2014]	Hex	Yes	No	Yes	No	No	No	Yes
EB-Mesh [Kim and Tautges 2010]	Hex	No?	No?	Not Done	No	Yes	Yes	Yes
Edwards/Bridson [2014]	Triangles	No	No?	No	No	No	No	No
Arrangements [Zhou et al. 2016]	Triangles	Yes	Yes	Yes	No	Yes	Yes	Yes
Azevedo et al. [2016]	Triangles	Yes	No?	No	Yes	No	No	No
Mandoline	Polygons	Yes	Yes	Yes	Yes	Yes	Yes	Yes

in practical settings. One source of such problems are component-wise CAD models; before the advent of robust mesh Booleans [Zhou et al. 2016] some cut-cell packages implemented their own mesh-mesh intersection routines to merge component-based triangulated CAD models [Aftosmis et al. 1998] into a single input mesh. Mandoline relies on existing robust mesh-intersection tools [Jacobson et al. 2016] to merge component-based inputs in a preprocess.

Open Meshes: Many simulation tasks require the ability to simulate both volumetric objects and thin shells, such as fluid flow around a deforming cloth [Guendelman et al. 2005]. Such thin objects are most compactly represented by their medial surface, often parameterized as an open (or closed) surface mesh (e.g., [Grinspun et al. 2003]), and are a common source of partial cuts and split cells.

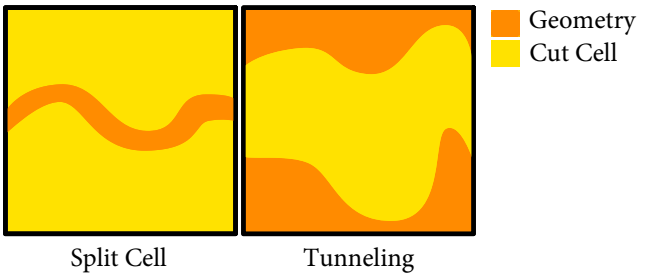


Fig. 2. Split cells and tunneling are common problem cases encountered during cut-cell generation. Split cells occur when surface geometry divides a cell into two or more disconnected parts, while tunneling occurs when the geometry encloses a region that crosses only through the faces of a cell.

Split Cells and Tunnels: A split cell is created when input geometry splits a grid cell into multiple distinct components. Tunneling occurs when a topologically distinct tube of geometry passes entirely through a cell without intersecting the cell's edges or vertices. These cases, illustrated in 2D in Figure 2, occur routinely for shells and open triangle meshes, near thin or sharp geometric features, and when the size of the grid cells is large compared to geometric feature size. A related challenge in deformable simulation considered by Li and Barbič [2018] is constructing topologically-correct immersions of detailed triangle meshes into coarse tetrahedra; however, their approach is restricted to closed, orientable triangle meshes and focuses on mesh embedding rather than generating clipped cells.

Adaptivity: Some cut-cell approaches allow the use of spatially adaptive grids to improve accuracy at the boundary or other regions of interest. Adaptive approaches are most commonly used to reduce the local complexity of input geometry relative to a single grid cell, in hopes of circumventing the need to store detailed sub-grid geometry (e.g., sharp features) and/or topology (e.g., split cells and tunnels) [Aftosmis et al. 1998; Colella et al. 2014]. Mandoline maintains a uniform resolution for cells that intersect the input, but allows octree-style coarsening away from it.

Code: The implementation of cut-cell algorithms is challenging, and as such there are few examples of readily accessible software. Unfortunately, some of the most widely used [Aftosmis et al. 1998] and state-of-the-art [Azevedo et al. 2016; Edwards and Bridson 2014] are not readily accessible for general use.

Context: Mandoline is the first algorithm for cut-cell mesh generation that meets all the requirements for both high resolution engineering simulations and state-of-the-art coarse grid methods (Table 1). Aspects of Mandoline are related to work on robust Mesh Arrangements [Zhou et al. 2016], but Mandoline offers significant improvements in generality and speed in the cut-cell context, for example, by allowing open surface geometry and exploiting the underlying hexahedral lattice. We will show in Section 5.7 that, for the closed, manifold meshes that Mesh Arrangements can handle, Mandoline is substantially faster. When compared to previous research and commercial algorithms, Mandoline offers a higher fidelity representation of cut-cell boundaries and correctly handles split cells and tunneling, which permits the generation of extremely coarse cut-cell meshes. Figure 1 shows some examples of detailed cut-cell geometries generated using Mandoline.

The computational fluid dynamics (CFD) community has traditionally preferred to "fully resolve" boundary geometry with fine grids, while graphics researchers have instead pushed the limits of detail offered by topologically correct coarse cells [Azevedo et al. 2016; Edwards and Bridson 2014]. However, even in CFD settings, refinement alone will not eliminate all cases of split cells: they can easily be created by sharp input features [Meinke et al. 2013] and on coarser levels of geometric multigrid schemes [Crockett et al. 2011; Dick et al. 2016].

Lastly, advanced finite element schemes are beginning to support general non-convex polytopal elements (including holes!) [Jaśkowiec et al. 2016], which can allow, for example, topology optimization

tasks to be computed with intricate Escher-tile elements [Paulino and Gain 2015]. Such modern discretization methods suggest a pathway towards fast, accurate, and flexible physical simulation, provided they are paired with the robust arbitrary cut-cell meshes generated by Mandoline. Indeed, NASA's recent *CFD Vision 2030* study [Slotnick et al. 2014] highlighted the demand for robust and scalable meshing capabilities for complex geometry, noting that *"paradigm shifts in meshing technology (i.e., **cut-cell methods**, strand grids, meshless methods) may lead to revolutionary advances in simulation capabilities."*

3 OVERVIEW

Given an input triangle mesh, our algorithm should output a cut-cell mesh comprised of simply-connected cells, each having a piecewise-constant winding number. Our algorithm strives to minimize dependence on vertex positions and geometric structure, focusing instead on the combinatorial mesh structure. In particular, we seek to avoid floating point operations as much as possible, ensuring that the overall method can robustly generate appropriate cut-elements despite near-degenerate polyhedra.

From a high level, our pipeline involves generating cut-elements (i.e., vertices, edges, faces, and cut-cells induced by intersections) from the bottom up, using different sub-complexes of the input triangulation. The new vertices can be computed from the triangles' edges and vertices, the planarity of the input triangles enables us to robustly produce the intersection-induced edges and faces, and the normals of the input triangles aid us in discerning the final cut-cells. Pseudocode for our algorithm is given in Algorithm 1 and a visual summary is provided in Figure 3.

Algorithm 1: Overview of our algorithm

```
igl::remesh_self_intersections();
removeDegenerateTriangles();
for e ∈ E do
    V_e, E_e ← findEdgeIntersections(e); # Sections 3.3 and 4.1.1
end
F ← {};
for each face f do
    E^H ← computeHybridEdges(f, {V_e}_{e∈∂f}); # Section 4.1.2
    for each edge e in E^H do
        V_e^h, E_e^h ← findEdgeIntersections(e);
    end
    V_f ← collectAndUnifyVertices({V_e^H}); # Section 4.2
    E_f ← collectEdges(V_f, {E_e^H});
    F_f ← computeFaces(∂F, {E_f}); # Section 3.4
end
E ← collectEdges({E_e}_e, {E_f}_f);
for each plane on axis d and coordinate c ∈ ℤ do
    Ê ← getEdgesInPlane(E, d, c); # Section 4.2.1
    F_{d,c} ←computeFaces(Ê); # Section 3.4
end
F ← collectFaces({F_f}_f, {F_{d,c}}_{d,c});
C ← computeCells(F); # Section 3.5
return C
```

3.1 Notation

We first outline some nomenclature. For clarity and brevity, we will refer to the input triangle mesh as the *tri-mesh*, the regular cubic lattice grid as the *grid*, and the output polyhedral cut-cell structure as the *cut-mesh*. Wherever possible, we will follow a simple naming structure of prefixing quantities of interest (vertex, edge, face) with the type of mesh that they belong to or within (tri-, grid-, cut-), except where it is irrelevant or unambiguous. The squares that define the faces of our regular grid each sit on an axis-aligned plane defined by an implicit equation of the form $\{x^d = c \in \mathbb{Z}\}$, where $x \in \mathbb{R}^3$, c is a constant integer, and the superscript d indicates the component of x; we call these *axial planes*.

In the overview below we describe our algorithm's key stages while considering only cut-cells which we refer to as *simple*; that is, cells whose set of bounding cut-edges consists of a single connected component. Then, in Section 4, we will investigate the details for generalizing the algorithm to arbitrary inputs. Some additional edge cases are enumerated in Appendix A.

3.2 Cut-Vertices

Cut-Vertex Types. Our input provides two types of vertices: tri-vertices, which come from our input tri-mesh, and grid-vertices, which sit exactly where grid-edges meet each other (Figure 4, left pair). Both types are also cut-vertices, since they will necessarily be present in the final cut-mesh. The clipping process, discussed momentarily, yields two new types of cut-vertices: (A) points where grid-edges pierce tri-faces and (B) points where tri-edges pierce grid-faces (Figure 4, right pair). Every involved edge and face will store a reference to the corresponding cut-vertices resulting from the intersections. Both of these cut-vertex types involve one grid element and one tri-mesh element. To avoid ambiguity, we identify them by the element of the tri-mesh involved, as in tri-face-intersections (A) and tri-edge-intersections (B).

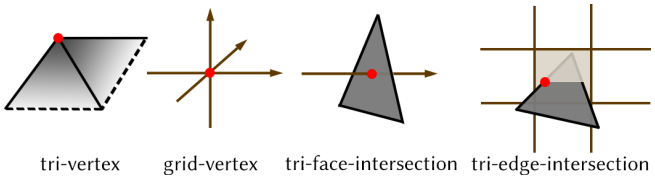


Fig. 4. Types of Vertices

Cut-Vertex Representation. Key to our dicing strategy is the ability to represent any position $p \in \mathbb{R}^3$ in terms of integer grid-cell indices c, cell-local trilinear coordinates q, and a bitmask b:

$$(c, q, b) \in \mathbb{Z}^3 \times [0, 1)^3 \times \mathbb{Z}_2^3. \tag{1}$$

We can reconstruct the corresponding world position as

$$p = dx\,(c + q), \tag{2}$$

where dx is the grid resolution. The additional bitmask b is a length-3 vector of binary flags used to track when p lies *exactly* on grid-faces, grid-edges, or grid-vertices. Specifically, an $x/y/z$ bit is 1 if the point in question lies exactly on the corresponding axial plane. For example, $b = 011$ indicates that a point sits exactly on both a y and

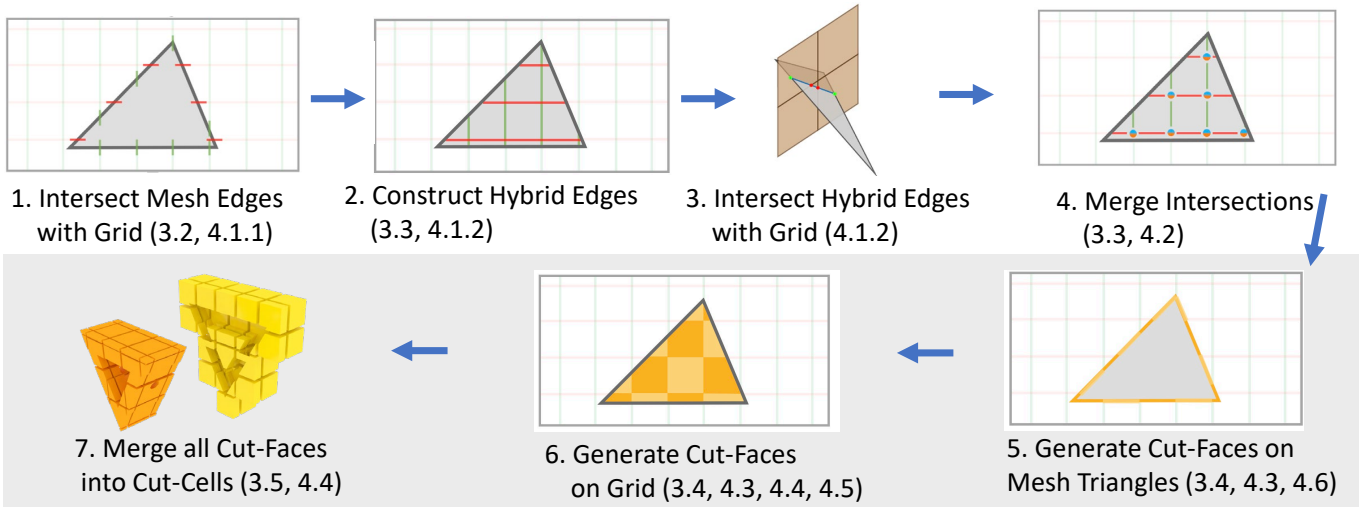


Fig. 3. A Visual Overview of Mandoline (2D Projection): Starting from input triangles, we find intersection points of triangle edges with the grid's axial planes. These define the endpoints of new *hybrid-edges* lying within each triangle (i.e., triangle-plane intersection lines). Hybrid-edges subdivide the triangles into a collection of cut-edges. Cut-edges per triangle are traversed and assembled into cut-faces (orange). The initial triangle intersection stages produce additional edges lying strictly within the axial planes; these can be processed similarly and in parallel to produce in-plane cut-edges and cut-faces. Finally, the full collection of cut-faces from both triangles and grid planes is traversed to assemble the cut-cells. (Parenthesized numbers refer to relevant sections of the paper.)

a z axial plane, i.e., on an x grid-edge. Our input tri-vertices are easily converted into this representation in a preprocess, where for a given axis d, the b^d value is set active if $q^d = 0$ (superscript d indicates the component of the vector). When we later compute intersections that give rise to cut-vertices, the grid structure ensures that their associated bitmasks can be determined unambiguously. As we will discuss later, this data is important for robustly and exactly identifying and handling redundant elements, and will also allow us to quickly determine the type of grid-intersection that a given vertex corresponds to. Although the bitmasks are in theory a redundant structure defined by $b_i = 1 \leftrightarrow q_i = 0$ and therefore could potentially be elided in implementation, maintaining an explicit representation simplifies implementation by both acting as a cache when handling redundancies and allowing a consistent, unified treatment with respect to bitmasks on edges and faces.

3.3 Cut-Edges

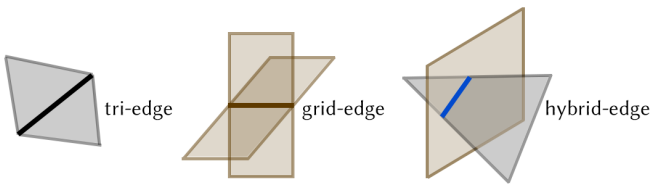


Fig. 5. Types of Parent Edges

Our cut-edge representation is simply an ordered pair of cut-vertex indices. Cut-edges are created by subdividing existing "parent" edges. Figure 5 shows the three kinds of parent edges that will give rise to cut-edges in the output cut-mesh: input tri-edges, input grid-edges, and new edges generated at lines of intersection between tri-faces and grid-faces, which we denote as *hybrid-edges*. Parent edges are subsequently subdivided into one or more final cut-edges based on their endpoints and the intersection points (cut-vertices) that lie along them.

Dividing Parent Edges into Cut-Edges. Observe that tri-edge-intersections (recall these are points where a tri-edge pierces a grid-face) necessarily interpolate their parent tri-edge's endpoint positions (Figure 6); by sorting the tri-edge-intersections by their interpolation value along the parent, we can derive the adjacency information that determines the cut-edges. Exactly the same procedure can be applied in an analogous way to subdivide both grid-edges and hybrid-edges into cut-edges based on the tri-*face*-intersection points lying along them.

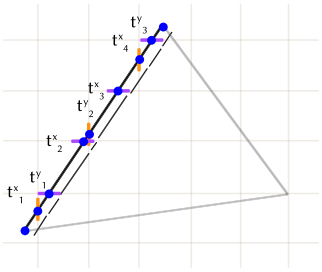


Fig. 6. Tri-edge-intersections are sorted along their parent tri-edge to determine the resulting cut-edge connectivity, illustrated in 2D. (In 3D, the tri-edge will also have intersections with z-planes.)

Bitmasks on Cut-Edges. Each of the tri-edge-intersections has at least one active bitmask entry b^d, as each corresponds to the intersection of an edge and an axial plane. When we compute these intersections, if an intersection is determined by a d-axial plane then we can safely set $b^d = 1$. Next consider tri-face-intersections. Since a hybrid-edge, defined by a vertex-pair (i, j), is already the result of intersecting a triangle with an axial plane (Figure 5, right), it lies entirely in that plane and therefore every point on it initially has at least one bitmask entry active (i.e., for some $\hat{d}$, $b_i^{\hat{d}} = b_j^{\hat{d}} = 1$). This means that every tri-face-intersection, where a hybrid-edge crosses a *second* plane, has at least *two* b^d active: one for the $\hat{d}$ that created the hybrid-edge and another for the second plane that the hybrid-edge intersects (see bottom of Figure 7).

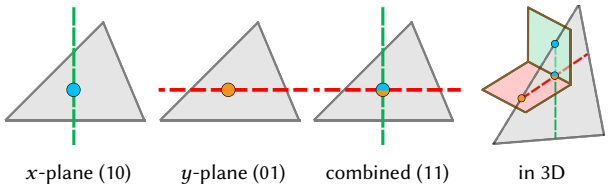


Fig. 7. The same intersection is found when tracing along both an x-plane (green) and a y-plane (red). Our system of bitmasks lets us identify and unify these redundancies.

Resolving Redundant Cut-Vertices. An important challenge for robustness is that processing the intersections of hybrid-edges arising from a particular tri-face can result in redundant vertices. For instance, consider a pair of hybrid-edges arising from the same tri-face. Projected to a plane, as seen in Figure 7, each hybrid-edge finds its own edge-intersections but floating point equality cannot always determine whether vertices on two different hybrid-edges represent the same point. In this example, when we process the intersections (i.e., cut-vertices) along the green x-hybrid-edge we will find the intersection of the tri-face with the grid-edge going through the blue dot, which is located at the intersection of the green and red hybrid-edges. When processing the intersections along the red y-hybrid-edge, we will find the orange intersection, once again located at the intersection of the two hybrid-edges. Although these two intersections are evoked by different parent hybrid-edges and their numerical realizations are constructed independently, they are really the same vertex.

We use our bitmask data to robustly merge them as follows. As discussed above, any tri-face-intersection along a hybrid-edge must have at least two nonzero b^d entries, i.e., the point has two coordinates specified exactly. Naturally it also lies within the plane of the input tri-face. Now, if we find two intersections created by the same tri-face with at least two matching (c^d, b^d) pairs then they must represent the same vertex. This is a straightforward consequence of the implicit function theorem: if two coordinates of a point on a given plane in $\mathbb{R}^3$ are known, the third coordinate is uniquely determined.

3.4 Assembling Cut-Faces

Cut-Faces as Planar Subdivisions. The preceding cut-vertices and cut-edges, together with the existing planar surfaces of the grid and tri-mesh, are sufficient for reconstructing the entire cut-mesh decomposition. Our next task is to gather this data into topologically consistent cut-faces (and later cut-cells) through appropriate traversals. Cut-faces are polygons that each lie entirely within a single planar surface present in our input data. These planar surfaces are of two types: the tri-faces from the input mesh, and the axial planes of our regular grid. Therefore, we will conceptually project our cut-vertices and cut-edges into two-dimensional parameterizations of the planes (axial planes or tri-faces) that they reside on, so we can depend on planar embeddings of our edge-graphs to discern cut-face topology. The collection of cut-edges for each input plane partitions that plane into polygonal regions; these are our cut-faces.

Reconstructing Faces by Polar Ordering Edges. The connectivity of each planar partition can be determined by looking at the edges incident to each vertex: two edges that share a vertex lie on the boundary of the same face only if there are no other edges lying between them in rotational order around the shared vertex. More specifically, consider a vertex p_i and its incident edges within the same plane, and choose an arbitrary basis on the plane. We can represent the edges (i, j) as outward-facing vectors $p_j - p_i$ in polar coordinates (r_j^i, θ_j^i) around the chosen vertex. Any two edges with adjacent angles θ_j^i are part of the same face.

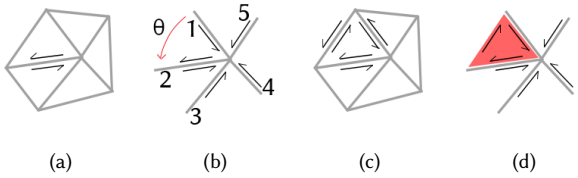
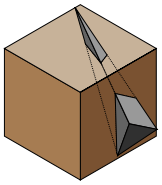


Fig. 8. We begin with half-edges that are only aware of their dual half-edge (a). Half-edges' *next* pointers are determined by rotational ordering around vertices, (b), which completes the half-edge mesh structure (c). Faces are constructed by ordered boundary loops of half-edges (d).

This fact can be used to recover any planar face encapsulated by a single planar closed curve, by walking from edge to edge (Figure 8); this applies to all our cut-faces, since each lies either in a planar tri-face or an axial plane. For cut-faces lying in the axial planes of the grid, we can use the two coordinates of that plane as the local basis for ordering of edges. For cut-faces on the tri-mesh, we collect the cut-edges on each tri-face and use that tri-face's barycentric coordinates as a local basis to again determine the polar coordinates needed for ordering. Note that this "projection" to the plane of each tri-face simply requires reading off the natural barycentric coordinates and requires nominal floating point computation: tri-face-intersections are stored with barycentric coordinates, tri-edge-intersections with line coordinate t have barycentric coordinates of the form $(t, 1 - t, 0)$, and tri-vertices have barycentric coordinates of the form $(1, 0, 0)$.



It is also possible for an input tri-mesh to produce cut-faces whose boundaries are *not* a single closed boundary curve, as we have assumed thus far. For example, in the inset figure, a tetrahedron protrudes through two grid-faces without crossing any grid-edges. On each intersected grid-face, the boundary of the resulting (outer) cut-faces consist of *two* disjoint boundary curves. This sort of situation is what is commonly called tunneling. We defer a thorough discussion of these challenges to Section 4.5.

3.5 Assembling Cut-Cells

Cut-cells are constructed in a manner much like cut-faces, but instead of ordering edges radially around vertices, we order faces around edges. For now, we limit ourselves to the case of cut-faces that consist of planar simply-connected curves with no self-intersections. For every directed cut-edge in the cut-mesh, we can collect the oriented cut-faces sharing the cut-edge, identify the cut-faces' associated normal vectors, and project those vectors to a plane orthogonal to the edge using an arbitrary basis for the plane, visualized as light blue in Figure 9. We then order the incident faces based on polar coordinate representations of their normals, and let adjacent faces belong to the same cell.

In practice, polar ordering can be performed on any plane that projects vectors bijectively with respect to the desired orthogonal plane. Therefore, in order to minimize floating point operations, for an edge with tangent vector T, we instead use the *axial* plane that is most-orthogonal to T: $\arg\min_i |T_i|$. Coordinates of the normal vectors in that plane can then be used rather than generating new floating point coordinates in an orthogonal plane.

A normal vector for each planar cut-face is induced by the orientation of its boundary curve. The cut-face normals are all oriented in the same direction around the edge because cut-faces derived from an input triangle (tri-cut-faces) are constructed in the same orientation as the triangle, and cut-faces derived from grid-faces (grid-cut-faces) are clockwise curves on $\mathbb{R}^2$ through the projection $\pi^d(p) = \left(p^{(d+1) \bmod 3}, p^{(d+2) \bmod 3}\right)$ where d is the axis of the grid-face that the cut-face lies on. Clockwise-normals for tri-cut-faces are therefore given by the clockwise-normals of the tri-faces that they lie within. For grid-cut-faces we use the normal $e^{(d+1) \bmod 3} \times e^{(d+2) \bmod 3}$. The above definitions are defined locally to each individual triangle, so we can easily support non-manifold input triangle meshes with arbitrary edge-face valences.

Perhaps a more intuitive ordering approach, similar to our edge-ordering step, would be to construct for each face a vector perpendicular to the shared edge and lying in the plane of that face, and use these vectors for ordering. With convex polygonal faces, as is the case in the work of Zhou et al. [2016], we could construct the vector by simply picking any vertex in the face that does not lie in the line of the chosen edge, such as another vertex. However, our faces are not necessarily convex. Our alternative solution given above uses the normal of each incident face for ordering because these normals are all an identical in-plane $\frac{\pi}{2}$ rotation from the corresponding outward-pointing vectors (see Figure 9).

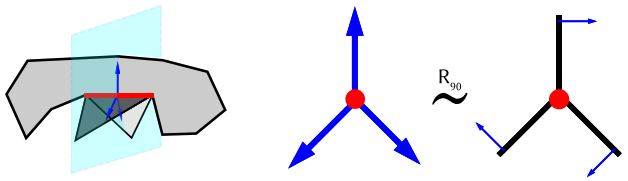


Fig. 9. To compute cells we need to order faces around a plane orthogonal to an edge. Rather than doing cell-plane intersections to find outward-facing tangent vectors we use oriented normals, which are a 90-degree rotation from the these tangent vectors.

Given the orderings of cut-faces around cut-edges, we unify oriented cut-faces into cut-cells by discerning which half-faces are the boundaries of the same cut-cell and using a union-find data structure to identify unique cut-cells. A variety of degeneracies can occur, which we discuss later (Section 4.6).

4 TECHNICAL DETAILS

Having outlined the algorithm's general flow, we can now describe the finer details as well as degeneracies that can occur and our corresponding solutions. To simplify intersection computation, our methodology takes advantage of two facts. First, triangles and edges are strictly convex. Second, viewing a triangle mesh as a collection of vertices, *open* edges, and *open* triangles produces a unique partition of points on the triangle mesh. This approach ensures that intersections are uniquely defined on a single object (such as an edge) rather than on multiple elements (such as redundantly on two triangles that share the edge).

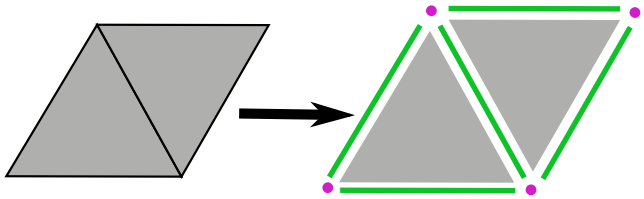


Fig. 10. We decompose the input geometry into the open elements to compute intersections on open sets.

4.1 Grid-Mesh Intersections

Given the input collection of non-intersecting triangles with non-zero volume, our first task is to determine the set of cut-vertices and cut-edges on the interior of the tri-faces and grid-faces. The primitive operation we adopt for finding cut-vertices and cut-edges is the intersection of an arbitrary open segment and an axial plane, rather than axis-aligned raycasts against arbitrary triangles (as adopted by some previous meshing tools [Kim and Tautges 2010; Müller 2009]). The former operation is sufficient for finding the cut-vertices that arise when parent tri-edges and hybrid-edges cross axial planes, but what about cut-vertices that arise at grid-line intersections with triangles? These do not need to be (re-)computed separately because they are the exact same cut-vertices already found when a hybrid-edge crosses an axial plane; segment vs. axial plane tests therefore

suffice. Given all the cut-vertices, cut-edges are simply the spaces between cut-vertices on the parent edges.

An important issue is that a cut-vertex on a hybrid-edge can be redundant with respect to a cut-vertex generated by a hybrid-edge lying within a different axial plane (see Figure 7). In especially rare situations, numerical inaccuracies can even result in vertices that are found to belong to the interior of a simplex through one computation path, and to its boundary through another. Rather than remove redundant vertices, we keep all redundant vertices from every line and annotate equivalence classes among intersection points (Section 4.2). Thus we can leave the various data structures alone and preserve all metadata that any given vertex may have, including the type and index of the simplex that it was spawned by, as well as its local coordinates with respect to its parent simplex.

4.1.1 Computing Edge-Intersections. We first compute tri-edge-intersections, which are generated by the intersection of tri-edges with axial planes (grid-faces). Later on, for tri-face-intersections we will use this same procedure on hybrid-edges.

Input: The two endpoint positions of a parent edge, given in the (c, q, b) representation from (1).
Output: The set of intersection points of the parent edge with all axial planes and the associated set of cut-edges.

Algorithm: For parent edge (i, j), for each axis, we compute the integers in the interval $(c_i^d + q_i^d, c_j^d + q_j^d)$, where we assume $i < j$. For every such integer value z, we get an intersection with line coordinate

$$t = \frac{z - (c_i^d + q_i^d)}{(c_j^d + q_j^d) - (c_j^d + q_i^d)} \tag{3}$$

and activate the bitmask entry for that intersection's axis: $b^d = 1$. We use the t values to sort these edge-intersections and thereby construct the child cut-edges generated by this parent edge.

4.1.2 Computing Hybrid Edges. We will next want to compute tri-face-intersections, induced by the intersection of tri-faces with axial lines (grid-edges). However, as noted earlier, we prefer to compute them using our edge-vs.-axial-plane primitive operation, as intersections of hybrid-edges with the grid's axial planes. This requires a method to find hybrid-edges; we can then use the procedure from Section 4.1.1 to subdivide them.

Input: A tri-face, its tri-vertices, and the tri-edge-intersections of its edges.
Output: The hybrid-edges belonging to the tri-face.

Algorithm: Convexity tells us that a line that crosses through the *interior* of a convex domain will intersect its boundary twice. A hybrid-edge is a line (lying within an axial plane), and a triangle is a convex domain; as such, finding a hybrid-edge is equivalent to finding two points on the boundary of a tri-face that lie on a single axial plane. Recall that we have already found all the points on tri-face boundaries that lie within axial planes: these are either our tri-edge-intersections or input tri-vertices. Hybrid-edges, therefore, terminate at tri-vertices or tri-edge-intersections on the tri-face's boundary edges. We need only identify the appropriate point-pairs.

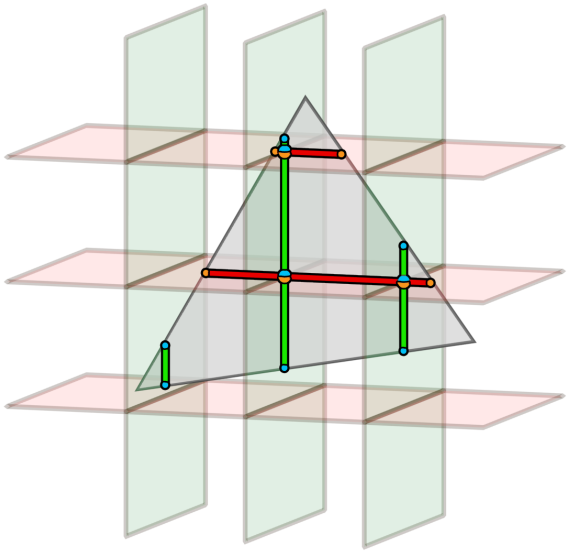


Fig. 11. Hybrid-edges (green and red lines) are the intersections of axial planes (light green, light red) and triangle faces (grey); their endpoints are either tri-edge-intersections (pure orange and pure blue points) or tri-vertices. Tri-face-intersections along hybrid-edge are shown in mixed orange/blue.

For each tri-face, we bin the tri-vertices and tri-edge-intersections that have mask entries b_i^d active according to a hash (d, c_i^d), where d is a coordinate with an active mask. For every bin containing two of these cut-vertices we create a hybrid-edge that connects them and subdivide it into cut-edges as we did for tri-edges. The endpoints of each hybrid-edge are known to have at least one integral coordinate based on the plane that contains it (the c_i^d from the bin); therefore, tri-face-intersections computed along each hybrid-edge must have at least two integral coordinates, since they lie on grid-lines.

Depending on how a tri-face penetrates through a given axial plane, there are a variety of potential hybrid-edge cases that may occur, as detailed in Figure 12. For any given bin (d, i) we only want to create hybrid-edges that lie on the interior of the given tri-face. While using bins with precisely two elements suffices for finding non-degenerate hybrid edges on *closed* triangles, we only want the hybrid edges that lie on *open* triangles. We therefore only use bins that contain two tri-edge-intersections or contain one tri-edge-intersection and one tri-vertex. This ensures that any hybrid-edges we create are strictly on the interior of tri-faces as desired, and as a result, we do not introduce new hybrid-edges that are redundant with respect to existing tri-edges that happen to lie in an axial plane.

Our bitmasks help us coalesce the cut-vertices that belong on grid-edges: any cut-vertex with exactly two active entries lies on a grid-edge. We collect such cut-vertices onto the grid-edges they lie on and generate cut-edges according to how they subdivide their grid-edges.

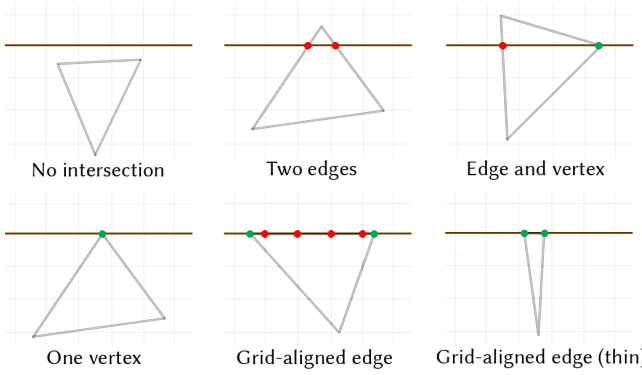


Fig. 12. The different types of intersections between an axial plane (black horizontal line) and a triangle, shown in a 2D projection.

4.2 Reducing Redundancies

Our collection of hybrid-edges intersects with axial planes to form our set of tri-face-intersections. However, this set may contain redundant vertices, as seen in Figure 7. With only floating point positions it can be difficult to identify these redundancies due to numerical artifacts. Likewise, we may have the same sorts of redundant intersections on tri-edges. For these reasons we developed a process to robustly identify these redundancies, using the inverse function theorem and our bitmasks b_i.

Input: Intersection points on the interior of a tri-element (i.e., tri-edge or tri-face).
Output: An exact identification of equivalent vertices in that element.

Algorithm: Consider the k-plane (i.e., infinite plane or line) that contains a given k-element (i.e., finite face or edge, respectively) and temporarily assume that the k-plane is not parallel to any axial lines. Then we know that if $k-1$ coordinates of two points on the k-plane are identical, it follows by the inverse function theorem that they are the same point. In our setting, when we have two points p, q with $k-1$ matching active bitmask entries and integer coordinates (i.e., $k-1$ values of d for which $b_p^d = 1$, $b_q^d = 1$, $c_p^d = c_q^d$), we know that these are indeed the same point. If we have a k-plane that is parallel to one or more axial edges, we simply increase the required number of matching bitmask coordinates by the number of axial planes intersected (i.e., we need $k + a - 1$ shared masks, where a is the number of parallel axes).

However, as a side effect of numerical imprecision, it is also possible for a redundant intersection pair to erroneously have different numbers of active bitmask entries. Consider Figure 13, in which a tri-edge cuts diagonally through a grid-vertex; the X-intersection is found to lie on both axes, while the Y-intersection is identified as lying only on the adjacent edge. Without proper handling, these masking differences can lead to the creation of unnecessary and nearly degenerate edges. Fortunately, we can resolve this ambiguity by selecting the vertex with the larger number of masked entries and uniting the other with it. This choice reduces the total number of vertices and edges that we process and removes some infinitesimal cut-cells. We will now mathematically formalize this procedure to deterministically remove all such redundancies.

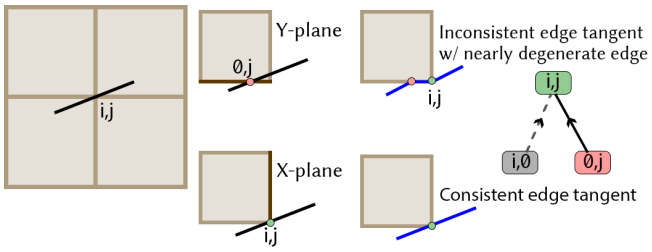


Fig. 13. We opportunistically upgrade intersections by their bitmasks to create better elements. Here we see that computing the intersection of a Y-plane finds a grid-edge, while intersecting with an X-plane finds a grid-vertex. The bitmask partial ordering identifies that these two points should be the same point and merges them.

4.2.1 Bitmask Partial Ordering. Each point's bitmask and integer coordinates specify the particular grid-cell, -face, -edge, or -vertex that it lies within. However, cut-vertices lying *close* to the boundaries of the grid-cells they reside within raise the concern that they may in fact lie on the boundary. Equivalently, we can say that such a cut-vertex may need at least one more active bitmask entry. When we have a collection of cut-vertices that we know must sit on a single plane, we will show that we can discern redundant vertices with a least upper bound operation on the directed acyclic graph (DAG) induced by adding active bitmask entries. This DAG is geometrically related to the boundary operator, and the induced partial ordering is determined by the number of bitmask entries, which is geometrically equivalent to the dimension of the grid elements being considered.

More formally, consider the map $\rho : \mathbb{R} \to \mathbb{Z}_\varnothing$ for $\mathbb{Z}_\varnothing = \{\varnothing\} \cup \mathbb{Z}$ given by

$$\rho(x) = \begin{cases} x, & x \in \mathbb{Z} \\ \varnothing, & \text{otherwise} \end{cases} \tag{4}$$

This map specifies that a real coordinate x is mapped to $\varnothing$ if it is not on a grid-plane (integer). With a healthy distrust of floating point computations we think of $\varnothing$ as values that may in fact be integral, but we do not yet have the confidence to claim so. We express this confidence with a partial ordering by saying for $u, v \in \mathbb{Z}_\varnothing$,

$$u \le v \Leftrightarrow [u = v \text{ or } (u = \varnothing \text{ and } v \neq \varnothing)]\,. \tag{5}$$

This partial ordering is naturally extended to vectors with the map $\rho : \mathbb{R}^n \to \mathbb{Z}_\varnothing^n$ where we apply the scalar map to each component, adopting the product (coordinatewise) order. The dimension of the geometry holding x can be incorporated by defining the semi-norm $|\rho(x)| : \mathbb{Z}_\varnothing^n \to \mathbb{Z}$ to be the number of non-$\varnothing$ components of $\rho(x)$. In order to extend this concept to tri-edges and tri-faces we define this map on sets $S \subset \mathbb{R}^3$ as the minimal value of ρ according to this semi-norm:

$$\rho(S) = \min_{z \in S} \rho(z). \tag{6}$$

We can efficiently evaluate ρ on polygons by taking the greatest lower bound of its vertices (i.e., $\rho(S)^d = \varnothing$ if $\rho(v)^d = \varnothing$ or $\rho(x)^d \neq \rho(y)^d$ for any $x, y \in S$).

Because we represent our cut-vertices as triplets (c, q, b) rather than simple vectors in $\mathbb{R}^3$, this is implemented using

$$\rho(x)^d = \rho(c, q, b)^d = \begin{cases} c^d, & b^d \\ \varnothing, & \neg b^d \end{cases}, \tag{7}$$

$$\rho(x) = \sum_d b^d. \tag{8}$$

4.2.2 Removing Redundancies. Now that we have formalized our vertices as a DAG we can discuss how this DAG structure removes redundancies. Note that when formalizing cut-vertices of a k-plane P^k, taking into account collinearity of P^k with axial planes requires a modification to ρ

$$\rho_{P^k}(x)^d = \begin{cases} e_d \cdot TP^k \neq 0 & \rho(x)^d \\ \text{o.w} & \varnothing. \end{cases} \tag{9}$$

where the dot product records the axes that P^k varies on, by checking that its tangent plane utilizes basis element e_d. A cut-vertex on P^k is therefore any vertex x such that $|\rho_{P^k}(x)| \geq k$.

Although ρ_{P^k} will be convenient for a quick derivation in a moment, it is worth noting that we do not need to explicitly compute it; in implementation we will only need inequality comparisons. This can be seen as follows: for any $\hat{d}$ such that $\rho_{P^k}(x)^{\hat{d}} \neq \rho(x)^{\hat{d}}$ we know that for every $y \in P^k$, $\rho(y)^{\hat{d}} = \rho(x)^{\hat{d}}$ because P^k, by definition, does not vary in that axis. That implies that $\rho(x) < \rho(y) \Leftrightarrow \rho_{P^k}(x) < \rho_{P^k}(y)$.

Our main claim is that every cut-vertex on a k-plane P^k is a local maximum according to this DAG.

$$\forall x, y \text{ s.t } |\rho_{P^k}(x)| \geq k \;\&\; |\rho_{P^k}(y)| \geq k \Rightarrow \rho(x) \not< \rho(y). \tag{10}$$

This claim allows us to say that for any potential cut-vertices x, y where $\rho(x) \leq \rho(y)$ we know that $x = y$. We implement this by filtering potential cut-vertices from each input triangle for redundancies by checking each pair of vertices for whether $\rho(x) < \rho(y)$ and removing vertices x whenever we find them.

The proof is fairly straightforward: the implicit function theorem implies that if x, y share k constraints when projected to the k-plane then they are indeed the same point. Say $\rho(x) \leq \rho(y)$; then for any d, $\rho_{P^k}(x)^d \neq \varnothing$ implies $\rho_{P^k}(y)^d = \rho_{P^k}(x)^d \neq \varnothing$ so x, y share at least k entries, and so $x = y$; therefore for any cut-vertices x, y, $\rho(x) \not< \rho(y)$.

4.3 Constructing Cut-Faces using Half-Edges

We use a collection of half-edge meshes to represent our cut-faces: one half-edge mesh for cut-edges on each axial plane in use and one half-edge mesh for cut-edges on each tri-face of our tri-mesh. Each half-edge mesh comprises a collection $\mathcal{H}$ of half-edges h_i, which are objects $h_i \in \mathcal{H}$ with connectivity defined by a set of functions

$$(n, d, v, f) \colon \mathcal{H} \to \mathcal{H}^2 \times \mathbb{Z}^2. \tag{11}$$

Applied to a particular half-edge h, n gives the half-edge that follows h; v gives the vertex that the half-edge h points towards; d gives the dual half-edge of h that stores the vertex at its other end; and f gives the face that h belongs to. A full directed edge (i.e., ordered vertex pair) can be constructed from a half-edge h using $(v(d(h)), v(h))$. The convenience of this representation is that, although it is typically used for triangle meshes, it can support simple polygons. As we discuss in Section 4.5, it can also be extended to support the more complex faces needed for tunneling.

Input: The set of edges in a closed simple polygon.
Output: A collection of open simple polygons that may contain one another.

Algorithm: Every tri-face holds its own half-edge mesh structure, as does each axial plane. Each cut-edge e is placed in the half-edge meshes of every face it is contained by and every axial plane it lies within. We generate cut-faces by first defining the results of the next operator $n(h)$. After creating a counterclockwise ordering of the N edges $\{h_i\}_{i=1}^N$ around the face, as described in Section 3.4, we unify adjacent half-edges by setting next pointers as:

$$n(h_i) := d\left(h_{(i+1) \bmod N}\right).$$

In order to create simple closed curves from open planar surfaces, we follow this procedure even for vertices with only one neighbor, which we call dangling edges. Handling dangling edges turns out to be required for computing cut-cell meshes, even for closed input tri-meshes. This is because dangling edges are created by tri-edges that lie on an axial plane, particularly when the two incident faces to that edge lie entirely on one side of the axial plane (Figure 14).

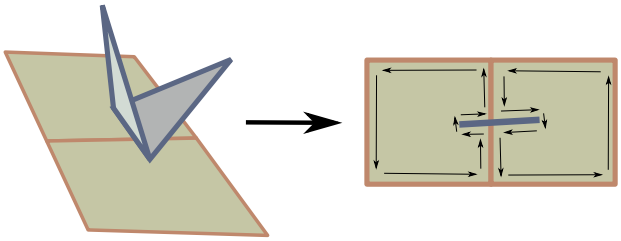


Fig. 14. A tri-edge that sits on an axial plane forms a non-manifold planar mesh.

After we have constructed the n operator, we can construct the faces, creating equivalence classes

$$f(h) \equiv f(n(h)).$$

We perform this same procedure for every axial plane and every tri-face.

4.3.1 Boundary Faces. To combine our cut-cell mesh with other, more regular, hexahedral mesh structures (e.g., uniform staggered grids or octrees), we only want to create a cut-mesh in a narrow *stencil* of grid-cells containing the tri-mesh surface, just large enough so that its boundary faces are all unaltered squares. While our intersection routines generate data only on or inside stencil cells, our mechanism for assembling the half-edge meshes has no notion of embedding; it will therefore generate spurious cut-faces (i.e., exterior boundary loops) that span interior gaps in the stencil or traverse

around the outer boundary of the stencil (see Figure 15). These must be explicitly identified and discarded.

Input: A collection of simple closed cut-faces.
Output: A subset of the input cut-faces such that every cell's interior is inside the stencil.

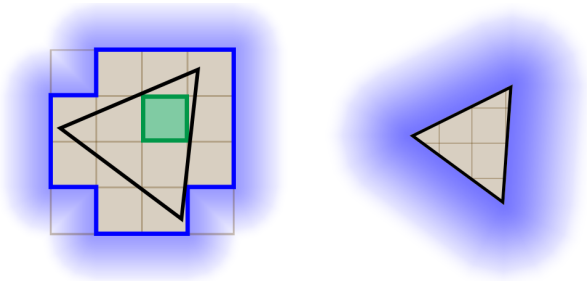


Fig. 15. Our face-creation algorithm identifies and discards infinite-volume exterior faces (blue) and spurious interior faces lying outside the cut-mesh stencil (green).

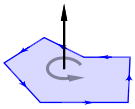
Algorithm: We discern and remove unnecessary faces by first annotating the grid-vertices that lie on stencil boundaries as boundary vertices on each axial plane. What we would ideally like to do is define boundary-edges as edges comprised of only boundary vertices, and remove cut-faces that are entirely comprised of boundary-edges. However, stencils that are a single grid-cell wide disallow this straightforward algorithm, since their faces would be be erroneously discarded. We correct for this by additionally retaining a face if its grid-edges form the boundary of a single grid-face *and* the face is in the stencil.

4.4 Cut-Cell Construction

We combine cut-faces into cut-cells by examining edges shared by multiple cut-faces. Similar to the technique we applied with half-edges, we split each cut-face into halves, each with opposing normals, and collect half-faces to form cut-cells.

Input: A collection of simple closed polygons with appropriate face-normals.
Output: A collection of cut-cells.

Algorithm: Our cut-faces are defined by oriented boundary curves and we let each half-face correspond to one of the two orderings of those boundary curves (i.e., one is the original cut-face orientation, the other is the same boundary curve reversed).



Each face is planar and we assign it a normal N orthogonal to it. In turn, each half-face is assigned either N or $-N$ as its normal according to the orientation of the outermost boundary curve (it must form a counter-clockwise loop on the plane orthogonal to the normal). We collect half-faces onto every oriented edge and use the half-faces' normals to determine their ordering about that edge.

4.4.1 Face Normals. It is crucial to accurately compute an appropriate normal for each half-face. Rather than construct new normals for polygonal cut-faces, we keep track of the underlying plane that a given cut-face comes from (i.e., an axial plane or a tri-face) and use its normal instead. This ensures that if the normals of the input geometry are well-defined, so too are those assigned to the cut-faces. This cell construction strategy is necessarily contingent on the input having reasonable normals, which could be achieved in practice by pre-processing the mesh to remove (near-)zero-area triangles. In our examples we only remove zero-area triangles.

4.5 Tunneling

Tunneling occurs when a cut-cell passes entirely through a grid-cell without intersecting any of its grid-edges. Support for this feature is equivalent to supporting cut-faces with holes, which can only occur on axial planes. Without support for tunneling the existing cut-faces result in overlapping faces, while Mandoline's cell construction procedure depends on having non-overlapping cut-faces.

Keeping with our boundary loop representation of cut-faces, handling tunneling requires detecting and representing faces whose set of boundary edges consists of multiple distinct loops. Each valid cut-face should have one outer loop enclosing all other inner loops, and the inner loops must not contain one another (i.e., the interior of the face is a single continuous component with holes). Assuming directed boundary loops, the outer loop has winding number $w = 1$ on its interior and every inner loop has winding number $w = -1$ on its interior.

Input: A collection of simple closed polygons with possibly overlapping interiors.
Output: A collection of polygons with holes and no overlapping interiors.

Algorithm: We compute the required non-simple cut-faces by, for each axial plane, finding an ordering of oriented closed boundary curves that contain one another and finding, for each $w = -1$ boundary loop γ_n, a "parent" $w = 1$ boundary loop γ_p.

This interior check can be performed by confirming that every point on a loop γ has constant winding number with respect to a potential parent, i.e., $\forall p \in \gamma, w_\gamma(p) = C, C \in \{-1, 1\}$. Because the boundary loops do not intersect one another, we need only evaluate the winding number on the vertices of γ_n and the midpoint of at most one edge, as explained in Figure 16. We therefore check whether one curve subsumes another's vertex set, find the novel edge, and check if its midpoint is on the interior or exterior of the other.

Having a collection of intersection-free curves for input here is advantageous for identifying tunneled faces within tunneled faces. This is because being intersection-free implies that the DAG of simple curves (loops) that contain one another is in fact a tree, i.e., boundary loops are uniquely contained by one another, so a bottom-up traversal of loops guarantees that merging loops into non-simple polygons yields the true cut-face topology. We therefore repeatedly find the smallest-area γ_p and join it with every unused γ_n contained within it, where area here denotes the hole-free simple polygon that the loop bounds, until we have gone through every γ_p.

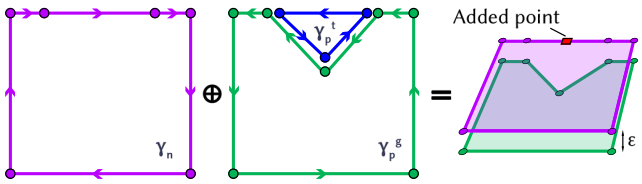
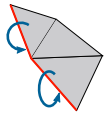


Fig. 16. Here we have three boundary loops: two $w = 1$ curves, the green loop γ_p^g and the triangle γ_p^t, and a $w = -1$ boundary loop in magenta, γ_n. Each vertex of γ_n lies on the loop γ_p^g, so in order to determine that γ_n is not nested inside γ_p^g we must check the winding number of the midpoint of an unshared edge (red square).

4.6 Degenerate Edges in Face Creation

Support for open tri-meshes implies that faces may have boundary edges with no opposing face; i.e., both "halves" of an edge refer to the same face. This is trivially supported by allowing our cell-construction algorithm to run on edges with only one adjacent face, as seen in the inset.



However, even without open meshes, degenerate cases such as Figure 14 are also an issue as our face-to-cell construction algorithm will generate multiple half-faces for each directed face around the problematic edge. Without careful treatment these redundant half-faces would disrupt our face-ordering process. Occasionally, directed edges $i \rightarrow j$ and $j \rightarrow i$ will both appear in a single face, which represents somewhere the face is "pinched" to have zero thickness; the offending edges need to be removed and the face subdivided. This process, although mathematically simple, can result in the creation of new boundary curves that must be stitched together from multiple components. Without this process, as seen in Figure 17, the claw in Figure 21 would not have individual fingers.

Input: The boundary of an open polygon.
Output: The boundary of degeneracy-free open polygons whose union is the input.

Algorithm: If we take our input boundary curve to be an open polygon these degenerate edges are equivalent to sets of points that would disappear under the closure operator.

Given that we represent boundary curves by an array of indices, this means we have to separate this array into multiple smaller arrays. Our method for removing such degeneracies is to identify the non-degenerate edges, create an adjacency list from those non-degenerate edges, and then traverse the adjacency list to fill new arrays of indices. We can easily avoid making redundant boundary loops by popping visited entries in the adjacency list.

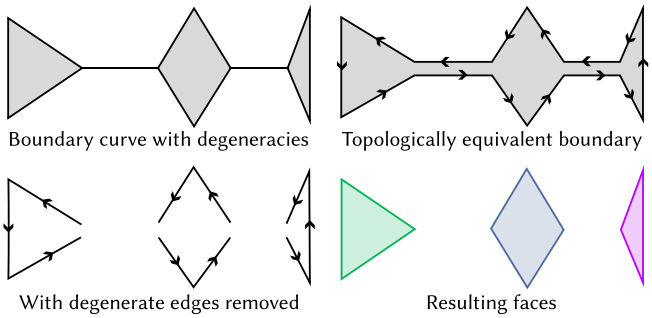


Fig. 17. A single boundary curve can have degeneracies that affect cell creation. We remove such degeneracies.

4.7 Triangulation

For the purposes of rendering we must triangulate the polygonal cut-faces that make up our cut-mesh. To triangulate faces with holes due to tunneling we call upon Triangle [Shewchuk 1996]. To improve performance we use traditional ear-clipping on faces that are simple polygons. Finally, cut-faces lying on input tri-faces are guaranteed to be convex (they are the result of triangles clipped against planes) so for these we simply use triangle fans.

5 RESULTS

We have used Mandoline to generate cut-cell meshes for a wide variety of geometries, as illustrated in Figures 1, 23, and in the supplement. Below we take a closer look at some of Mandoline's features and performance.

5.1 Split Cells and Tunneling

Figure 18 shows examples of split cells and tunnels created by Mandoline. Here we generate tunnels using a hollow, closed Hilbert curve and split cells using a wavy hexahedron. Mandoline resolves cases of multiple splits and multiple tunnels, and is robust to small geometric features that may result from "almost" split cells. We are unaware of any other cut-cell generation method that demonstrates this type of reliability.

5.2 Effect of Grid Resolution

Figure 19 shows two cut-cell meshes of the same Dragon surface generated at different resolution. Notice that both coarse and fine cut-cells generated by Mandoline conform perfectly to the input surface geometry and that small features, like the Dragon's toes, are handled correctly.

5.3 Adaptive Meshes

Nontrivial cut-cells are only generated in grid cells in a local neighborhood around the input triangle mesh. For simulation purposes, we typically fill the remainder of the desired domain with cubes. However, for a given cut-cell grid resolution we can also pair the generated cut-cell mesh with a standard octree to create an adaptive mesh; this handles the common case in which a uniform finest grid scale is desired around the input surface, with coarser cells further away from it. We show a few examples of this feature in Figure 23.

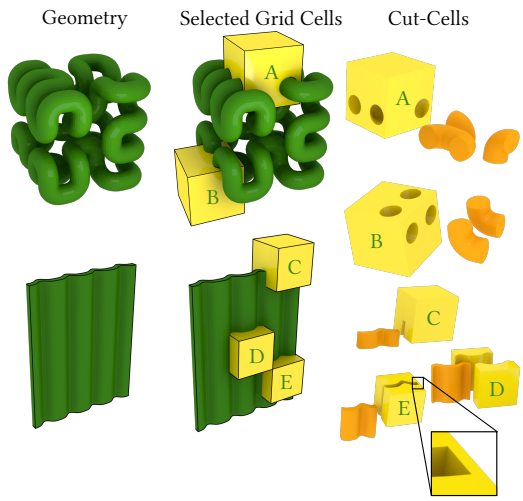


Fig. 18. Mandoline handles both tunneling (top) and split cells (bottom). Cutting a Hilbert curve (top) produces cut-cells with multiple tunnels (A,B). Using a wavy plane (bottom) to cut through the background grid produces cells that are partially (C,E) and fully (D) split. Note the narrow cell geometry in (E) caused by the small gap between grid and geometry.

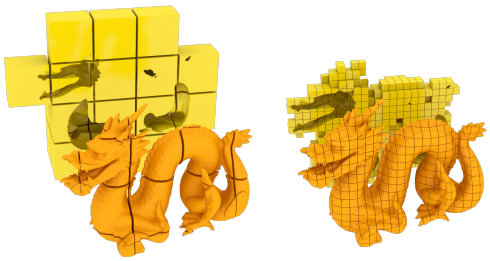


Fig. 19. Mandoline can produce cut-cells using a wide range of background grid resolutions. The cut-cells produced using a coarse background grid (left) are often more complex than their fine conterparts (right) because they contain more of the surface geometry. Of note is how Mandoline robustly handles small cells produced by the dragon's toes.

5.4 Open Meshes

Figure 20 shows an example of a cut-cell mesh generated from an open surface. Here special care must be taken at the open boundary where the surface may not fully split the cell. In these cases Mandoline generates topologically correct *partially* cut-cells, embedding a double-sided portion of the open mesh into such cells.

5.5 Difficult Cases

Mandoline handles several types of challenging degenerate situations. In this section we outline a few exemplars, as seen in Figure 21.

Wedge (a). This case has two grid-cells with a tetrahedron cut out of them through a boundary. This presents a case where, on the boundary face between the two cells, using pointwise winding number computations presents an ambiguity between whether a negative winding number boundary curve is included within a positive winding number boundary curve, as seen in Figure 16. This is resolved by finding a point on the negative winding number curve that isn't shared with the positive winding number curve, as discussed in Section 4.5.

Empty cone (b). The input mesh for the *Empty cone* test is the difference of two nested tetrahedra that share their top face. This shared face forms the opening of the cone and lies exactly on an axial plane. Despite the triangle on the top grid cell being tangent to the axial plane, Mandoline successfully constructs the original geometry, a single interior cell, and an exterior cell surrounding the cone.

Claw (c). The *Claw* test represents a case where a single boundary curve, which traces around each of the claws and over the webbing between the claws, must be separated into three distinct boundary curves forming the base of each claw. This is the case seen in Figure 17; in order to identify the base of each claw as a separate polygon we must remove degenerate edges and discern the three separate boundary loops.

Oscillator (d). The *Oscillator* presents a case of tunneling (Section 4.5) where the axial plane between two grid cells contains many embedded boundary curves. The input geometry is a cylinder whose top oscillates several times in the radial direction, with the final oscillation ending exactly on the middle axial plane. The cut-faces along this axial plane are a series of punctured disks, which create a collection of embedded cells for both the input geometry and the exterior space.

5.6 Validation

We depended on a few metrics to determine whether our cut-cell meshes incorporated the input triangle mesh and grid in a lossless fashion and created well-defined volumetric elements. For every mesh from the Thingi10K dataset, we checked that the following conditions were satisfied.

Grid cell isolation. Cut-cells should be fully contained within a single grid cell.

Piecewise-constant winding number (PWN). Cells should have piecewise-constant winding number using `libigl`'s `piecewise_constant_winding_number` function on triangulated faces. We removed cut-faces corresponding to partial cuts for this computation because they are only implicitly known to represent two faces glued together with opposite orientation, which would generate a PWN value of 0.

Surface area. The cut-faces generated should produce a partitioning subdivision of the input mesh. We computed the area fractions of tri-cut-faces relative to their parent triangles, and confirmed that the sum of the weights were uniformly 1 for each triangle.

Input Volume. For closed inputs without self-intersections, the volume of the cells on the interior of the input geometry should be identical to the volume of the input geometry.

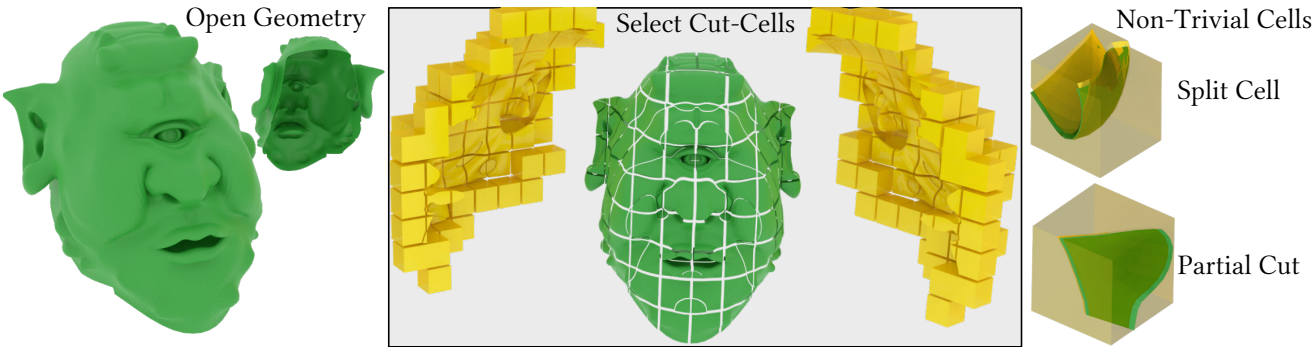


Fig. 20. Cut-cells generated for an open surface. At the boundary, some grid cells will not be split completely. In such cases Mandoline correctly identifies and produces proper partially cut-cells.

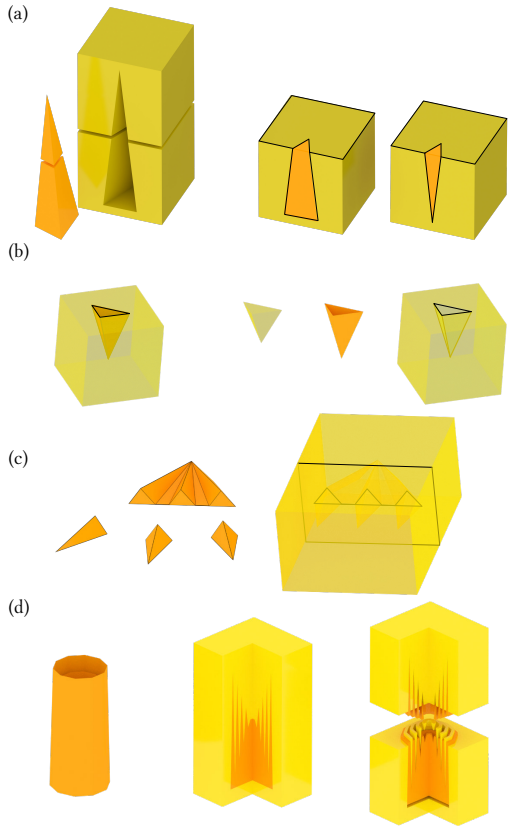


Fig. 21. Examples of a few challenging cases, described in Section 5.5.

5.7 Performance

Mandoline performs well on a wide range of examples and at a wide range of grid resolutions (Table 2). Even for complicated meshes such as the Ogre (Figure 20), it only takes ~9 seconds to generate a result. We also compared with the only comparable open source alternative to Mandoline: libigl's mesh arrangements [Zhou et al. 2016]. However, mesh arrangements is much slower than Mandoline even for small grid sizes. For instance, Mandoline generates a $(5 \times 5 \times 5)$ dragon cut-cell mesh in ~5 *seconds* while libigl takes over *40 seconds*. We coerced mesh arrangements to generate cut-cells for this experiment by triangulating a grid of cubes and intersecting them with the dragon mesh.

We ran performance tests to check the scalability of our method using the Thingi10K dataset [Zhou and Jacobson 2016], the results of which are available in our supplemental document. We found that Mandoline's performance was roughly linear with respect to the number of grid cells and number of input triangles.

6 CONCLUSIONS AND FUTURE WORK

We have presented Mandoline, a robust algorithm for rapidly generating hexahedral cut-cell meshes suitable for applications in engineering and computer graphics. We have demonstrated that Mandoline can generate well-formed cut-cells from both open and closed surfaces described by potentially non-manifold triangle meshes at arbitrary grid resolution. The code for Mandoline is readily available at `github.com/mtao/mandoline`. This includes the mesh generator, supporting all desired cut-cell features (Table 1), and the tools to reproduce our tests.

For isolated tri-mesh geometry, enclosed entirely within one grid cell (such as an air bubble falling inside a single liquid cell in a fluid simulation), Mandoline will generate two identical cut-cells, with winding numbers of opposing signs. The appropriate duplicate cell needs to be removed as a post-process. A potential avenue for resolving these floating cut-cells is to utilize the same winding number strategy we apply for tunneling in cut-faces in Section 4.5.

Mandoline also makes no attempt to eliminate relatively small cells with negligible volume that may be generated. For some downstream applications small cells may be problematic; e.g., they can induce stability issues for certain CFD schemes, although a range of possible treatments have been proposed [Berger 2017]. An interesting extension would be to develop a geometric post-process that can robustly provide a practical lower bound on cell sizes, perhaps via a cell-merging strategy.

However, the most exciting avenues of future work are those unlocked by the availability of a tool like Mandoline. In the context of physical simulation, for example, we look forward to investigating Mandoline's use in concert with modern high-order finite

Table 2. Run-times of Mandoline for various inputs as a function of the number of grid cells (indicated by column headings). All times are given in **milliseconds**. Benchmarks were run on an Intel Xeon E5-2630 (2.4 Ghz) with 64 GB of RAM.

Example	$2 \times 2 \times 2$	$4 \times 4 \times 4$	$5 \times 5 \times 5$	$10 \times 10 \times 10$	$25 \times 25 \times 25$	$50 \times 50 \times 50$
Bunny	3323	3126	3165	3225	3224	3909
Bunny Watertight	2694	2749	2883	3424	5527	14316
Cow	173	217	240	333	831	3216
Cube	14	16	29	142	1263	9193
Dragon	4740	4972	4965	5441	8035	16603
Fandisk	407	459	438	662	1783	5947
Joint	54	70	81	287	2028	11257
Ogre	1265	1301	1435	1658	3214	8970
Noisy Sphere	36	29	55	142	1190	6504

difference, finite volume, and finite element schemes for solid and fluid mechanics problems, including fluid-structure interaction and multiphase flows. We believe that Mandoline's greatest benefit will be allowing the community as a whole to more fully explore the use of cut-cell meshes for geometry processing, simulation, and animation tasks. For a sneak peek, look at Appendix B.

ACKNOWLEDGMENTS

This work is graciously supported by NSERC Discovery Grants (RGPIN-04360-2014 & RGPIN-2017-05524), NSERC Accelerator Grant (RGPAS-2017-507909), Connaught Fund (503114), and the Canada Research Chairs Program. We thank Tim Jeruzalski for his substantial efforts to render our results, as well as Rahul Arora, Nicole Sultanum, and Sarah Kushner for assistance with figure creation, and Ryan Goldade for the fluid crown splash model.

REFERENCES

Michael J Aftosmis, Marsha J Berger, and John E Melton. 1998. Robust and efficient Cartesian mesh generation for component-based geometry. *AIAA journal* 36, 6 (1998), 952–960.

PF Antonietti and I Mazzieri. 2018. High-order Discontinuous Galerkin methods for the elastodynamics equation on polygonal and polyhedral meshes. *Computer Methods in Applied Mechanics and Engineering* 342 (2018), 414–437.

Vinicius C. Azevedo, Christopher Batty, and Manuel M. Oliveira. 2016. Preserving Geometry and Topology for Fluid Flows with Thin Obstacles and Narrow Gaps. *ACM Trans. Graph.* 35, Article 97 (2016), 97:1–97:12 pages. Issue 4. Proceedings of SIGGRAPH 2016.

M. Berger. 2017. Chapter 1 - Cut Cells: Meshes and Solvers. In *Handbook of Numerical Methods for Hyperbolic Problems*, Rémi Abgrall and Chi-Wang Shu (Eds.). Handbook of Numerical Analysis, Vol. 18. Elsevier, 1 – 22. https://doi.org/10.1016/bs.hna.2016.10.008

P. Colella, D. T. Graves, T. J. Ligocki, G. Miller, D. Modiano, P. O. Schwartz, B. Van Straalen, J. Pilliod, D. Trebotich, M. Barad, B. Keen, A. Nonaka, and C. Shen. 2014. *EBChombo software package for Cartesian grid, embedded boundary applications.* Technical Report.

RK Crockett, Phillip Colella, and Daniel T Graves. 2011. A Cartesian grid embedded boundary method for solving the Poisson and heat equations with discontinuous coefficients in three dimensions. *J. Comput. Phys.* 230, 7 (2011), 2451–2469.

Christian Dick, Marcus Rogowsky, and Rüdiger Westermann. 2016. Solving the fluid pressure Poisson equation using multigrid—evaluation and improvements. *IEEE transactions on visualization and computer graphics* 22, 11 (2016), 2480–2492.

Olivier Dionne and Martin de Lasa. 2013. Geodesic voxel binding for production character meshes. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 173–180.

Essex Edwards and Robert Bridson. 2014. Detailed water with coarse grids: combining surface meshes and adaptive discontinuous Galerkin. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 136.

Florian Ferstl, Rüdiger Westermann, and Christian Dick. 2014. Large-scale liquid simulation on adaptive hexahedral grids. *IEEE transactions on visualization and computer graphics* 20, 10 (2014), 1405–1417.

Matthew Fisher, Boris Springborn, Peter Schröder, and Alexander I Bobenko. 2007. An algorithm for the construction of intrinsic delaunay triangulations with applications to digital geometry processing. *Computing* 81, 2-3 (2007), 199–213.

Eitan Grinspun, Anil N. Hirani, Mathieu Desbrun, and Peter Schröder. 2003. Discrete Shells. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '03)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 62–67. http://dl.acm.org/citation.cfm?id=846276.846284

Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. 2005. Coupling water and smoke to thin deformable and rigid shells. In *ACM Transactions on Graphics (TOG)*, Vol. 24. ACM, 973–981.

Anil Nirmal Hirani. 2003. *Discrete exterior calculus.* Ph.D. Dissertation. California Institute of Technology.

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201353

Alec Jacobson, Daniele Panozzo, C Schüller, Olga Diamanti, Qingnan Zhou, N Pietroni, et al. 2016. libigl: A simple C++ geometry processing library.

Jan Jaśkowiec, Piotr Pluciński, and Anna Stankiewicz. 2016. Discontinuous Galerkin method with arbitrary polygonal finite elements. *Finite Elements in Analysis and Design* 120 (2016), 1–17.

Hong-Jun Kim and Timothy J Tautges. 2010. EBMesh: An embedded boundary meshing tool. In *Proceedings of the 19th International Meshing Roundtable*. Springer, 227–242.

Dan Koschier, Jan Bender, and Nils Thuerey. 2017. Robust eXtended finite elements for complex cutting of deformables. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 55.

Long Lee and Randall J LeVeque. 2003. An immersed interface method for incompressible Navier–Stokes equations. *SIAM Journal on Scientific Computing* 25, 3 (2003), 832–856.

Yijing Li and Jernej Barbič. 2018. Immersion of self-intersecting solids and surfaces. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 45.

Haojie Lian, Asger N Christiansen, Daniel A Tortorelli, Ole Sigmund, and Niels Aage. 2017. Combined shape and topology optimization for minimization of maximal von Mises stress. *Structural and Multidisciplinary Optimization* 55, 5 (2017), 1541–1557.

Konstantin Lipnikov, Gianmarco Manzini, and Mikhail Shashkov. 2014. Mimetic finite difference method. *J. Comput. Phys.* 257 (2014), 1163–1227.

Sebastian Martin, Peter Kaufmann, Mario Botsch, Martin Wicke, and Markus Gross. 2008. Polyhedral finite elements using harmonic basis functions. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 1521–1529.

Matthias Meinke, Lennart Schneiders, Claudia Günther, and Wolfgang Schröder. 2013. A cut-cell method for sharp moving boundaries in Cartesian grids. *Computers & Fluids* 85 (2013), 135–142.

Rajat Mittal and Gianluca Iaccarino. 2005. Immersed boundary methods. *Annu. Rev. Fluid Mech.* 37 (2005), 239–261.

Neil Molino, Zhaosheng Bao, and Ron Fedkiw. 2004. A virtual node algorithm for changing mesh topology during simulation. In *ACM Transactions on Graphics (TOG)*, Vol. 23. ACM, 385–392.

Matthias Müller. 2009. Fast and robust tracking of fluid surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 237–245.

Maks Ovsjanikov, Mirela Ben-Chen, Justin Solomon, Adrian Butscher, and Leonidas Guibas. 2012. Functional maps: a flexible representation of maps between shapes. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 30.

Taylor Patterson, Nathan Mitchell, and Eftychios Sifakis. 2012. Simulation of complex nonlinear elastic bodies using lattice deformers. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 197.

Glaucio H Paulino and Arun L Gain. 2015. Bridging art and engineering using Escher-based virtual elements. *Structural and Multidisciplinary Optimization* 51, 4 (2015), 867–883.
Nicholas Ray, Dmitry Sokolov, Maxence Reberol, Franck Ledoux, and Bruno Levy. 2018. Hex-dominant meshing: Mind the gap! *Computer-Aided Design* 102 (2018), 94–103.
Masoud Safdari, Ahmad R Najafi, Nancy R Sottos, and Philippe H Geubelle. 2016. A NURBS-based generalized finite element scheme for 3D simulation of heterogeneous materials. *J. Comput. Phys.* 318 (2016), 373–390.
Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019. Navigating intrinsic triangulations. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 55.
Jonathan Richard Shewchuk. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, Ming C. Lin and Dinesh Manocha (Eds.). Lecture Notes in Computer Science, Vol. 1148. Springer-Verlag, 203–222. From the First ACM Workshop on Applied Computational Geometry.
Hang Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)* 41, 2 (2015), 11.
Eftychios Sifakis, Kevin G Der, and Ronald Fedkiw. 2007. Arbitrary cutting of deformable tetrahedralized objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 73–80.
Jeffrey Slotnick, Abdollah Khodadoust, Juan Alonso, David Darmofal, William Gropp, Elizabeth Lurie, and Dimitri Mavriplis. 2014. CFD vision 2030 study: a path to revolutionary computational aerosciences. (2014).
Maxime Theillard, Landry Fokoua Djodom, Jean-Léopold Vié, and Frédéric Gibou. 2013. A second-order sharp numerical method for solving the linear elasticity equations on irregular domains and adaptive grids–application to shape optimization. *J. Comput. Phys.* 233 (2013), 430–448.
Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. 2010. Physics-inspired topology changes for thin fluid features. *ACM Trans. Graph.* 29, 4 (2010), 1–8. https://doi.org/10.1145/1778765.1778787
Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016).
Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

A DEGENERACIES

Although Mandoline is formulated to simplify cut-cell generation as much as possible, various types of degeneracies nevertheless require explicit handling; we review some of these below. The general principle is that in many components of our pipeline we encounter redundant geometries, algebraically volume-free elements, or non-manifold elements that do not affect the final topology. All of these elements can simply be removed from the construction of the topology and, for completeness, added back to the geometry afterwards.

Bad Input Meshes. The first step of Mandoline is to preprocess its input for three issues: self-intersecting triangles, redundancies, and empty triangles. We treat these by first running `libigl`'s `remesh_self_intersections` on the input. We then collapse vertices having *identical* floating point coordinates, remove triangles sharing the same three indices, and remove empty (topologically zero-area) triangles. For topological consistency, we stitch the remaining triangles together after removing empty triangles.

Maintaining Open Simplex Assumption. Key to Mandoline is the ability to create cut-vertices and cut-edges that are robust to floating point errors. Cut-edges are computed by analyzing how cut-vertices partition closed tri- and hybrid-edges, including their endpoints. To guarantee that these endpoints remain endpoints of these edges, we mathematically require that edge-intersections lie on an open edge. Within the local parameterization of each edge this is the same as saying we only accept edge-intersections that lie in the open interval $(0, 1)$.

Planar Triangles. Recall that the intersection of an axial plane with an open triangle is either the entire triangle itself or an open segment, i.e., a new hybrid-edge. When we encounter a triangle contained by an axial plane we neglect searching for hybrid-edges along that particular axis.

Redundant Edges and Faces. When the input mesh has edges or faces that lie entirely in axial edges or planes, respectively, we let the input mesh take priority and remove the grid-based edges and faces. In order to discern redundant cut-elements we use the bitmasks to hash cut-elements to the grid-edges or grid-faces they lie within and search for cut-elements on grid-elements with the same indices. One optimization on this procedure takes advantage of the fact that tri-cut-faces only contain a single boundary curve. This implies that only grid-cut-faces that have more than one boundary curve can be ignored. We further accelerate the comparison of cut-elements by making use of a total ordering of their indices. For edges this means we sort the indices, but for cut-faces there is some subtlety required because the boundary loops representing the potentially redundant cut-face can be ordered in the opposing orientation. This check is done by finding the smallest index of both curves and circulating in two directions for one of the curves to see if it is identical to the other.

B POISSON PROBLEM

Fig. 22. We simulate the effect of putting a bust of Max Plank in a wind tunnel. In the background we show the resulting pressure field as per-cut-cell colors and in the foreground we show the resulting flow with arrows. Note how the arrows flow around the bust.

A crucial question is whether the cut-cells generated by Mandoline can be used for simulation. We were successfully able to solve a Poisson problem using existing techniques, without any preprocessing such as removing small cells or faces. In particular, we used a DEC-styled Laplacian [Hirani 2003] approach where the dual vertices are chosen to be the grid-cell centers, as proposed by Azevedo et al. [2016]. This sort of Laplacian depends on only three geometric quantities: the adjacency structure of the cut-cells,

the surface areas of the cut-faces, and the distance between dual vertices.

The adjacency structure is given by Mandoline, the surface areas of cut-faces can be computed from the boundary loop structures of the (untriangulated) cut-faces, and the distances between dual vertices is straightforward. This is where the advantage of Mandoline's explicit cut-cell computation lies: the cut-face surface areas are readily available from the geometry and do not need to be approximated. For example, although the background grid in Figure 22 is quite coarse, with the obstacle's neck only ~4 grid cells long, we are able to accurately resolve a visibly smooth flow around it.

Figure 22 is the result of a "wind tunnel" test using potential flow, which solves for static airflow with given boundary conditions. The flow is described by the gradient of a scalar potential ϕ, found by solving a Poisson problem with boundary conditions on its gradient. We solve for the L^2-minimal solution with two types of Neumann conditions. On the grid boundary we presume $\nabla\phi \cdot D = \nabla\phi \cdot N$, where ϕ is the solution, D is a given direction, and N is the normal of a boundary face. On the input mesh boundary we presume no penetration: $\nabla\phi \cdot N = 0$. MIC0-preconditioned conjugate gradient with a residual norm tolerance of 10^{-10} consistently converged for this linear system.

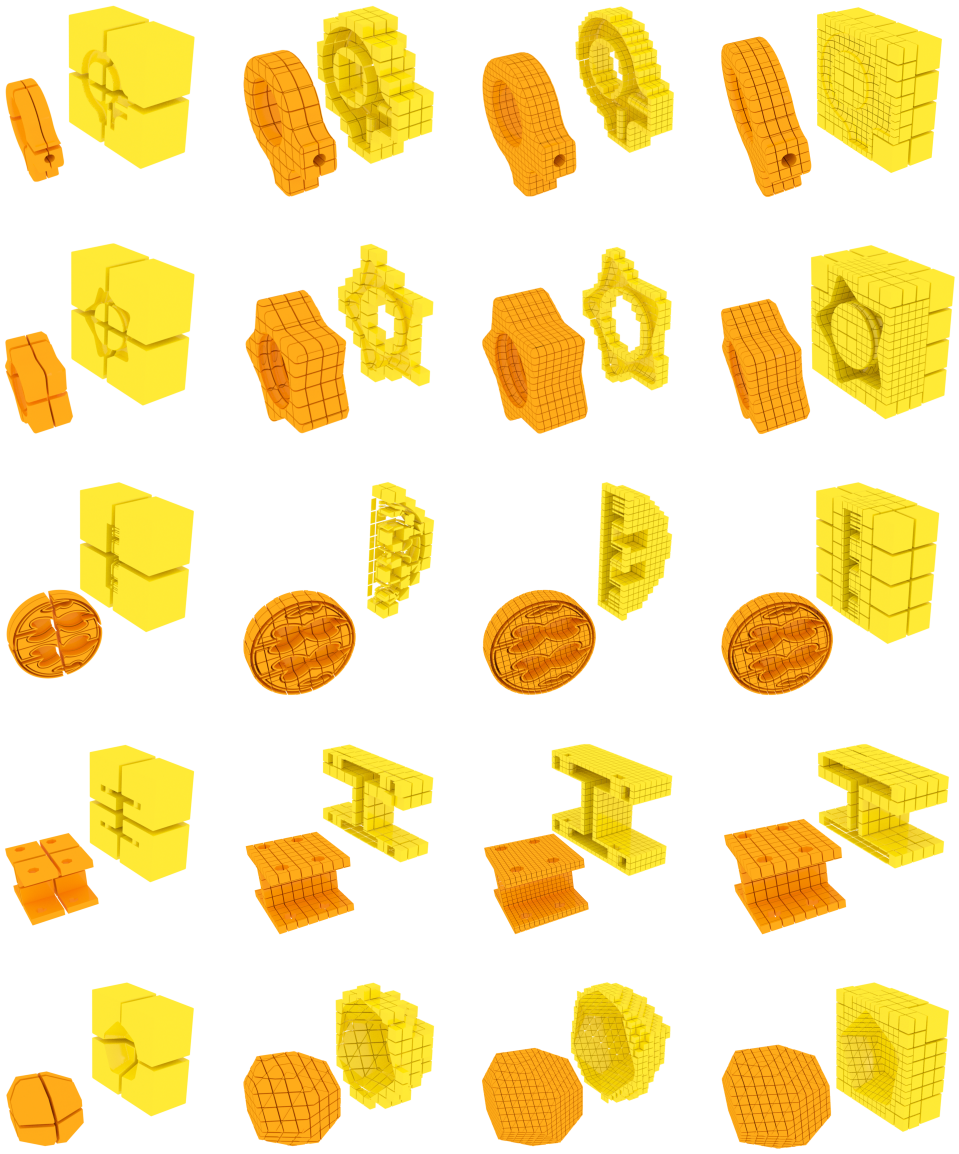


Fig. 23. Mandoline works at various grid resolutions and can generate adaptive grids. Different slices through the exterior region (yellow) were chosen for each image to expose interesting features.