SIAM J. SCI. COMPUT. © 2009 Society for Industrial and Applied Mathematics
Vol. 31, No. 4, pp. 2472–2493

ROBUST TOPOLOGICAL OPERATIONS FOR DYNAMIC EXPLICIT SURFACES[*]

TYSON BROCHU[†] AND ROBERT BRIDSON[†]

Abstract. We present a solution to the mesh tangling problem in surface tracking. Using an explicit triangle mesh to track the location of a surface as it moves in three dimensions has many potential advantages for accuracy and efficiency, compared to implicit capturing methods such as level sets. However, particularly when "mesh surgery" is required for topological changes, this approach is prone to tangling: The mesh may self-intersect or otherwise no longer represent a physical interface. Our new approach uses robust collision testing to determine when a mesh operation—such as motion, adaptive refinement, coarsening, or topological change—will lead to an invalid state; we then either roll back noncritical operations or apply robust collision response algorithms, minimally perturbing the mesh to guarantee validity. We present numerical examples demonstrating the robustness and accuracy of the method.

Key words. interface tracking, dynamic surfaces, triangle meshes, geometric flows

AMS subject classifications. 65D17, 65D18

DOI. 10.1137/080737617

1. Introduction.

Methods for discretizing and evolving a surface embedded in Euclidean space can be lumped into two broad categories: implicit and explicit. Both have advantages and disadvantages; our goal in this paper is to overcome one of the critical challenges of explicit methods, namely, robustness in the face of difficult operations such as topological changes.

Implicit surface methods do not track the location of points on the surface but rather utilize fixed data points located at regular intervals throughout the computational domain to reconstruct the surface when needed. They are sometimes called "front-capturing" since they do not explicitly track points on the surface but rather contain the information needed to reconstruct the surface. A popular example of an implicit method is the level set method [22], in which the zero-isocontour of a scalar function—usually a numerically approximated signed distance field—defines a surface.

Explicit surface methods, by contrast, discretize the surface using a set of connected points. This is sometimes called "front tracking," as points on the surface are followed as the surface evolves, and these points define the surface. Perhaps the best-known general purpose front tracking method is that developed by Glimm, Grove, and Li [10]. They discretize the surface as a simplex mesh (line segments in two dimensions and triangles in three dimensions).

By *dynamic surfaces*, we mean surfaces that move over time. Surface motion can be driven by geometric flows, such as motion in the normal direction or motion by mean curvature, or by physical simulation, such as two-phase flow simulation, where the surface defines the material interface.

[*]Received by the editors October 8, 2008; accepted for publication (in revised form) March 3, 2009; published electronically June 12, 2009. This work was partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada. The research of the first author was funded by a PGSD doctoral scholarship from the Natural Sciences and Engineering Research Council of Canada.
http://www.siam.org/journals/sisc/31-4/73761.html

[†]Computer Science Department, University of British Columbia, Vancouver, BC V6T 1Z4, Canada (tbrochu@cs.ubc.ca, rbridson@cs.ubc.ca).

Implicit surface functions are commonly used to represent dynamic surfaces because they easily handle topological changes, such as merging and pinching-off: No special effort is required. (Indeed, implicit methods become problematic if topological change is ruled out by the underlying dynamics.) These changes are notoriously hard to handle with explicit surfaces—meshes often become "tangled" if they are advected into a self-intersecting state or if "mesh surgery" introduces holes that must be remeshed, and it can then be difficult to reconstruct a consistent, intersection-free surface.

Implicit methods, however, do suffer from drawbacks such as numerical dissipation and the inability to reliably capture detail near or below the resolution of the underlying grid. For example, numerical dissipation in level set methods smoothes out high curvature regions and can cause parts of the surface to vanish, even under rigid motion. In fact, very thin yet smooth (low curvature) surfaces require excessively refined grids to avoid this problem, for example. In addition, the user has no control to *prevent* topology change if the surface physics of the problem dictate a more subtle behavior. A further drawback of implicit surface tracking is that the extraction of an explicit surface from an implicit representation is still a nontrivial operation—many of the challenges that face explicit surface tracking methods also apply here—and thus the complexity is shifted from surface tracking to surface extraction.

In this paper, we present a framework for robustly handling topological changes in explicit surfaces in a step towards tractable explicit surface tracking that does not suffer from the drawbacks of an implicit method. We consider two-dimensional (2D) surfaces embedded in $\mathbb{R}^3$, discretized as triangular meshes.

Our key idea is to require that every mesh operation should leave the mesh in a consistent, nonintersecting state—as opposed to attempting to recover such a state after the fact. We thus first use robust collision detection methods to ensure we detect every possible violation. Once a collision is detected, we either roll back the operation if it is deemed noncritical and may be delayed to a subsequent time step when it may succeed, or minimally perturb mesh positions to guaranteeably avoid the problem (patterned after frictionless inelastic collision response in a physical contact problem).

We present numerical experiments verifying convergence of our method for geometric flows with topology change, highlighting its ability to robustly and efficiently capture extremely thin and delicate details.

2. Previous work.

Since its introduction [23], the level set method has become very popular for implicit surface capturing. The particle level set method [8], which supplements the implicit surface function with a set of marker particles, has proven especially accurate in some applications. Even greater accuracy has been achieved through the use of an octree grid structure [18], which can effectively increase the resolution of the level set function discretization around the interface.

Volume of fluid methods were introduced by DeBar [5] and Noh and Woodward [21] for the simulation of fluids. These methods operate on a fixed volumetric voxel grid, maintaining the fraction of volume occupied by fluid at each voxel. At the beginning of the simulation, these volume fractions are initialized using the known geometric interface. At subsequent steps in the simulation, these volume fractions are evolved according to the advection equation. The actual interface must then be reconstructed from these volume fractions when required. See work by Rider and Kothe [25] for more details and an overview of some advancements in this technique over the past several decades.

The use of passive marker particles to track a surface is common in applications such as fluid simulation. For example, the marker and cell method [13] uses an Eulerian grid to drive the fluid simulation and passive marker particles to indicate the interior of the fluid. An explicit surface can then in principle be reconstructed as needed, though we note determining an accurate smooth surface from the marker particles remains an open research problem. We might classify this approach as both Lagrangian and implicit since it uses the Lagrangian frame of reference but still *captures* only information necessary to reconstruct a surface.

A hybrid implicit/explicit surface tracking method called "grid-based front tracking" was introduced in by Glimm et al. [11]. In this method, explicit surfaces that have been advected into an intersecting state are treated in one of two ways. First, a grid-free untangling is attempted, which cuts the triangle mesh along intersection contours and retriangulates. If this fails, a global, grid-based reconstruction is performed. (A similar effect is achieved by Bargteil et al. [1], where the explicit surface is regenerated from a level set discretization at each step.) Du et al. [6] refined this method, performing only local grid-based reconstruction around the intersecting mesh components. Fixed grid nodes are used to determine when the surface is in an intersecting state, and thus sub-grid-scale intersections may be missed; the grid reconstruction similarly eliminates any details at or below the grid resoution, including smooth but thin parts of the surface.

Pons and Boissonnat [24] handle topological changes in explicit surfaces by embedding the surface mesh in a tetrahedralization of the surface vertices. The tetrahedralization is updated at each step, rejecting those tetrahedrons whose circumcenters lie outside the surface mesh. The exterior triangles of the remaining tetrahedrons form a new triangulation of the surface. This tetrahedralization is shown to be a good approximation of the input surface mesh but is not guaranteed to conform to the surface triangles used in the previous step.

Perhaps the method most similar to the one presented in this paper was introduced by Lachaud and Taton [17], applied to the segmentation of 3D images. The authors use dynamic, purely explicit surfaces with interference detection to determine when topological changes should occur. Their method of interference detection relies on maintaining a regular triangulation of the surface mesh and detecting when any two vertices are close to each other. Our method differs in that we use robust geometric predicates for interference detection, allowing for nonregular, anisotropic triangulations while guaranteeing intersection-free meshes. Our method also permits surfaces to approach much closer than the length-scale of a triangle, critical for handling thin yet smooth geometry efficiently.

Robust intersection detection and handling have been important components of cloth simulation in computer graphics for several years. Similar to the explicit surfaces discussed in this paper, cloth is generally discretized as a triangle mesh. Bridson, Fedkiw, and Anderson introduced a method for guaranteeing intersection-free cloth, which uses continuous collision detection, penalty forces, impulse-based collision solving, and a fail-safe rigid motion projection to ensure that the cloth surface will never advect into an intersecting state, eliminating the need for mesh untangling altogether [2]. This approach has been simplified by Harmon et al. with a unified projection method [14] and concurrently by Sifakis, Marino, and Teran with a globally coupled impulse-based method [27]. In this paper, we adopt this approach of preventing intersections rather than allowing them to occur and then trying to untangle them.

Table 3.1
Symbols used in this paper.

$q(t)$:	mesh configuration at time t
$\mathbf{x}(t)$:	mesh vertex positions
$\mathbf{T}(t)$:	mesh triangles
$\mathbf{u}(\mathbf{x}, t)$:	velocity field
$\hat{f}(t_m)$:	*predicted* value of function f at future time t_m
α:	maximum edge length
β:	minimum edge length
γ:	maximum volume change
ξ:	initial average edge length
ϵ_p:	proximity threshold

3. Method.

After an overview of notation, the discussion of our method will begin with a description of the input and output data, providing a framework for our algorithm, followed by a short description of the main steps in our algorithm. We then describe in detail our methods for interference detection, mesh maintenance, topological changes, and collision resolution.

3.1. Symbols.

Table 3.1 lists the symbols used in what follows. They are defined as they are introduced in the text, but we include them here for easy reference.

3.2. Framework.

Our method operates on triangle meshes. Let $q(t)$ be a mesh configuration at time t, consisting of the pair $(\mathbf{x}(t), \mathbf{T}(t))$, a set of vertex positions and a set of triangles defining the mesh connectivity. Each triangle is a triple of vertex indices, oriented consistently over all triangles.

$\mathbf{T}$ is a function of time, indicating that connectivity can change over time, but note that connectivity changes occur at discrete events, for instance, when a vertex is added or removed. In contrast, $\mathbf{x}(t)$ is usually an approximation of a continuous trajectory of vertex positions.

Our method takes as input a mesh configuration, the "current mesh" $q(t_n)$, and a pointer to a velocity function $\mathbf{u}(\mathbf{x}, t)$ which will define the "predicted" mesh $\hat{q}(t_n+\Delta t)$. The choices of input mesh configuration, velocity function, and time step Δt are determined by the user, usually the results of a physical simulation or geometric flow. Our system produces as output a "final" mesh configuration $q(t_n + \Delta t)$, which is as close as possible to the "predicted" mesh $\hat{q}(t_n + \Delta t)$ but is guaranteed to be intersection-free. In general, the final mesh will not share the same connectivity as the current mesh and may have a different number of vertices. We ensure that, as long as the current mesh is intersection-free, the final mesh will be intersection-free as well, even if the intermediate, predicted mesh is not.

We note that the underlying simulation may use any time integration scheme to get the linear velocity per vertex. For example, the simulation might advect a surface vertex using a high-order, multistep time integration scheme (in a collision-naive way) from time t_n to $t_n + \Delta t$ and then return $\mathbf{u}(\mathbf{x}, t) = (\mathbf{x}(t_n + \Delta t) - \mathbf{x}(t))/\Delta t$. If this trajectory is indeed free of interference, the resulting integration will retain the accuracy of the multistep scheme.

There are a few inadmissible mesh triangulations which we will not accept as input and will not produce as output. For simplicity of presentation, we do not allow boundary edges (edges incident on fewer than two triangles) but return to this later in section 4.4 when extending the method to handle open surfaces. We do not allow two triangles to share the same three vertices (creating a zero-volume tetrahedron). We *do* allow some nonmanifold surfaces. In particular, we allow more than two triangles to be incident on an edge. However, each triangle incident on an edge must have another triangle with a consistent orientation incident on the same edge. We also allow only an even number of triangles to be incident on a single edge. Loosely speaking, we allow only meshes that partition the computational domain into an interior and an exterior; i.e., we ensure the mesh is the boundary of an open set.

This restriction allows us to perform mesh operations which are usually restricted to manifold surfaces. For example, an edge flipping operation (see section 3.5.2) usually assumes that the edge is incident on two triangles with consistent orientation. Our requirement above ensures that there are *at least* two triangles with consistent orientation incident on the edge, so we can find two such triangles and apply the edge flip operation.

Apart from these restrictions, the user has the freedom to supply any mesh configuration and velocity field. As a very simple example, consider a mesh under motion from an external velocity field $\mathbf{u}(\mathbf{x}, t)$ with forward Euler time integration. Then at the nth time step (when $t = t_n$) we have as inputs

$$\begin{aligned}&\mathbf{x}_{\texttt{current}} = \mathbf{x}(t_n),\\&\mathbf{T}_{\texttt{current}} = \mathbf{T}(t_n),\\&\mathbf{u}(\mathbf{x}, t).\end{aligned}$$

Our method will internally compute the intermediate mesh using forward Euler time integration:

$$\mathbf{x}_{\texttt{predicted}} = \hat{\mathbf{x}}(t_n + \Delta t) = \mathbf{x}(t_n) + \Delta t\, \mathbf{u}\left(\mathbf{x}(t_n), t_n\right).$$

It will output the intersection-free mesh positions and possibly modified triangulation:

$$\begin{aligned}&\mathbf{x}_{\texttt{final}} = \mathbf{x}(t_n + \Delta t),\\&\mathbf{T}_{\texttt{final}} = \mathbf{T}(t_n + \Delta t).\end{aligned}$$

We can then use this output as $q(t_{n+1})$, the input $q_{\texttt{current}}$ in the next time step, and we have a complete time integration scheme.

3.3. Algorithm overview. Here we provide a high-level outline of our algorithm. Each nontrivial step is explained in detail in what follows. The first and fourth steps, splitting long edges and null-space smoothing, are mesh maintenance steps and can be omitted at the user's discretion. The second and third steps, edge flipping and short edge collapsing, are mainly used for mesh maintenance but are also key to allowing surface separation (as described in section 3.6), and thus omitting these steps will prevent separation events. We also allow the option of using these steps but preventing topology changes by adding extra checks to prevent surface separation. The zippering step can also be omitted if the user wishes to avoid topological changes altogether, as might be appropriate in some applications.

Our software implementation allows the user to toggle three boolean settings: whether to perform mesh improvement operations, whether to allow topological changes, and whether to enforce intersection-free surfaces. Figure 3.1 depicts the flow of control in our algorithm.

Split long edges (*section* 3.5.1). Subdivide all edges with length greater than the user-defined maximum edge length.

Flip non-Delaunay edges (*section* 3.5.2). Repeatedly search the mesh for non-Delaunay edges and replace them.

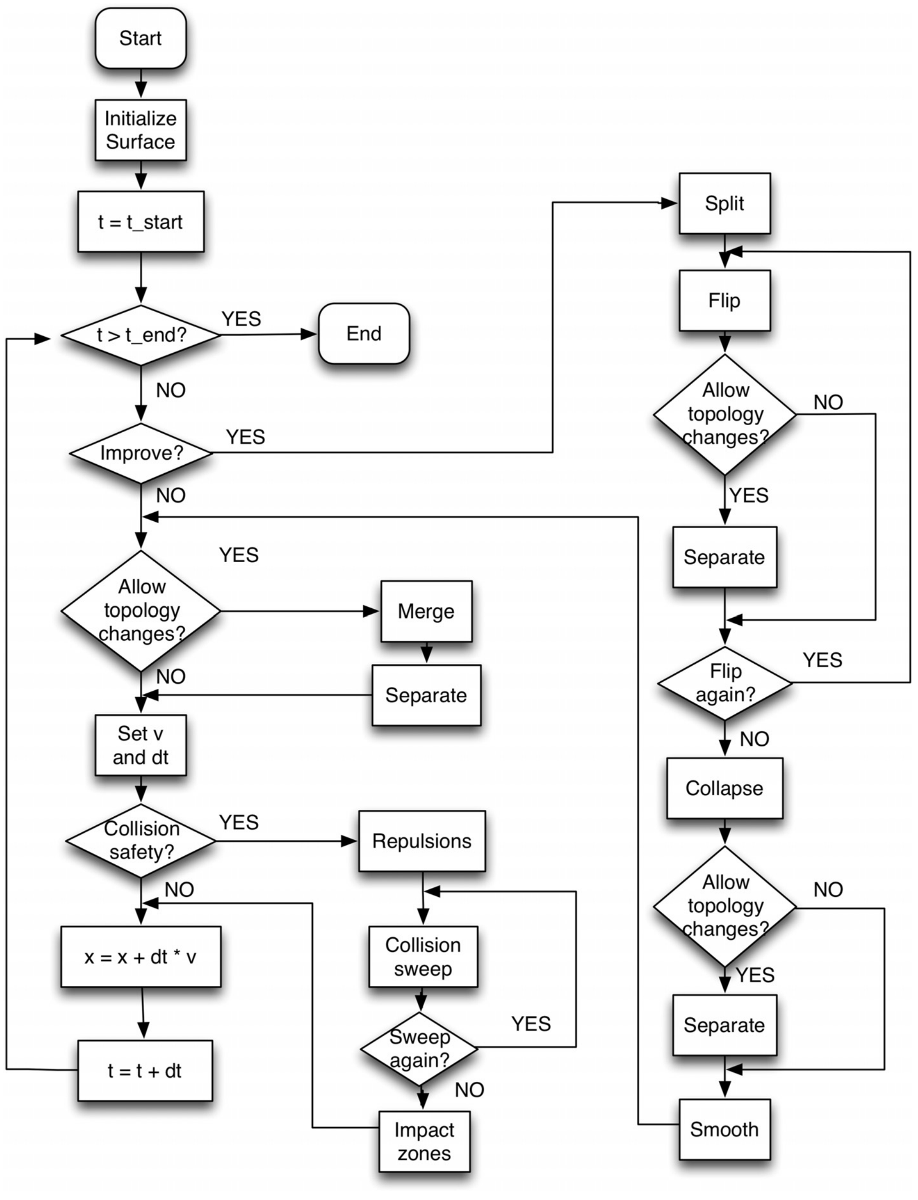


FIG. 3.1. *Flow chart outlining our algorithm.*

Collapse short edges (*section* 3.5.3). Delete all edges with length less than the user-defined minimum edge length, replacing the edge with a single vertex.

Null-space smoothing (*section* 3.6.1). Apply a Laplacian-type filter to the vertex positions, moving each vertex only in the null space of its local quadric metric tensor.

Merging/zippering (*section* 3.7). Detect edges that are near each other and attempt to merge surfaces by deleting their incident triangles and zippering the resulting holes.

Set vertex velocities. Use the given velocity function to assign a velocity vector to each vertex.

Proximity detection and repulsion forces (*section* 3.8). Detect elements that are near each other and apply a repulsion force between them by adjusting the vertex velocities.

Compute predicted vertex locations. Use the adjusted vertex velocities to compute the predicted vertex locations with forward Euler integration.

Impulse-based collision resolution (*section* 3.8). Detect individual collisions using continuous collision detection and apply impulses which will prevent an intersection.

Impact zones (*section* 3.8). Detect remaining collisions, group colliding elements into impact zones, and solve for a set of impulses which will prevent intersection.

Compute final vertex locations. Use the vertex velocities and collision impulses to get an intersection-free mesh configuration.

3.4. Interference detection. Before diving into the details of our approach, we briefly discuss techniques for interference detection. We differentiate between three types of geometric interference detection: *intersection* detection, *proximity* detection, and *collision* detection. We use all three of these types at different steps in the algorithm.

Static *intersection detection* detects if and where a mesh intersects itself for a given mesh configuration (i.e., at one instant in time). This can generally be decomposed into primitive tests discovering where an edge is penetrating a triangle, but we must take care to identify degenerate cases, such as an edge penetrating a surface only at an edge or at a vertex. We also use a static point-in-tetrahedron test during mesh maintenance (described below).

Static *proximity detection* detects when mesh elements are closer than a specified tolerance (in particular, when a vertex is close to a triangle or when two edges are close to each other). We denote this proximity tolerance by ϵ_p.

Proximity detection finds the two points on the mesh elements that are closest to each other. If we denote the set of four barycentric coordinates of these two points as $\mathbf{a}$ (setting $a_i = 1$ if i is the vertex in a vertex-triangle collision), then, to find the distance between the mesh elements, we multiply the barycentric coordinates by -1 if they refer to a point on the triangle or on the second edge in an edge-edge proximity, to get a new set of coordinates $\bar{\mathbf{a}}$. Then taking the sum of vertex locations weighted by these scaled barycentric coordinates yields a vector between these closest points. If $\mathbf{p}$ is the vector of indices of the vertices involved, then the shortest distance is given by the length of this vector:

$$d = \left\| \sum_{i=1}^{4} \bar{a}_i \mathbf{x}_{p_i} \right\|.$$

Our proximity detection function can also return a "collision normal" $\mathbf{n}$, which, when an impulse is applied along it, will increase the distance between elements.

Continuous collision detection (CCD) detects whether a collision between a moving vertex and a moving triangle or between two moving edges will occur during some specified time span.

In our framework, two mesh configurations are given: one at time t_n and one at time $t_n + \Delta t$. We assume vertices move in a linear trajectory from their positions at time t_n to their positions at time $t_n + \Delta t$. Given these two configurations, continuous collision detection will return any point-triangle and edge-edge collisions, as well as the time that the collision occurs (sometime between t_n and $t_n + \Delta t$), the collision normal, the set of barycentric coordinates describing the point of contact, and the computed relative displacement.

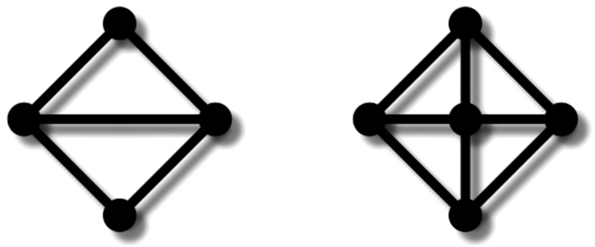


FIG. 3.2. *Edge split operation on the mesh graph.*

We use a robust CCD approach recently introduced by Brochu and Bridson [4]. This method augments the spatial coordinate system by one dimension, representing time. We can then apply numerically robust predicates from computational geometry to detect intersections in space-time. These predicates use only multiplication and addition, so we can easily bound the maximum round-off error accumulated in their computation. From this forward error analysis, we can identify and handle degenerate geometric configurations without resorting to user-tuned error tolerances. This approach detects all collisions (no false negatives) and produces potentially false positives only when the numerical error accumulated during the CCD computation exceeds the magnitude of the results of the computation.

3.5. Mesh quality improvement. Triangles with small areas or poor aspect ratios can adversely affect collision detection, topological operations, and any boundary-integral-based simulation. To improve the quality of our surface discretization, we use a few common operations. The need for these operations and their effectiveness have been argued by others [16] and are orthogonal to the main contribution of our paper—robust topological changes. However, we include this section for completeness.

3.5.1. Edge split. If an edge is longer than a user-defined maximum edge length (denoted α), we subdivide it by introducing a new vertex (see Figure 3.2). The new vertex can be placed at the edge midpoint, which will not introduce any new intersections. However, we may wish to offset the new vertex from the current surface using a subdivision scheme to maintain curvature. To ensure intersection safety in this case, we can make use of the continuous collision detection framework introduced earlier. We begin by introducing the new vertex at the edge midpoint. We then compute the "predicted" location of the new vertex via the subdivision scheme. These two points define a *pseudomotion*. We check the new vertex and its incident triangles and edges as it moves from the edge midpoint to its predicted point to ensure that it doesn't collide with any existing mesh elements (which do not move during this pseudomotion). If a collision does occur, we revert to using the edge midpoint, which is guaranteed to not introduce any new intersections.

We do not attempt to subdivide nonmanifold edges (edges incident on more than two triangles), although if handled carefully, these edges could probably be treated as well. In our experience, these edges are rare enough that failing to subdivide them does not introduce significant error.

3.5.2. Edge flip. We employ an edge flip operation as a way of maintaining good triangle aspect ratios. For each edge incident on two triangles, we check whether the distance between the two points *not* on the edge is less than the length of the edge. If so, we remove the edge and create a new edge between these two points (see Figure

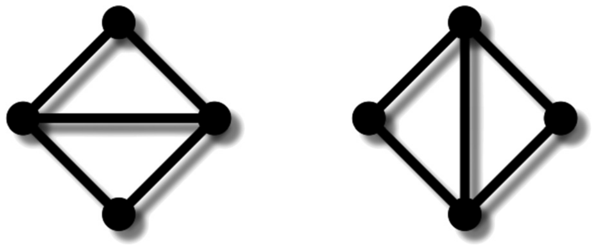


FIG. 3.3. *Edge flip operation on the mesh graph.*

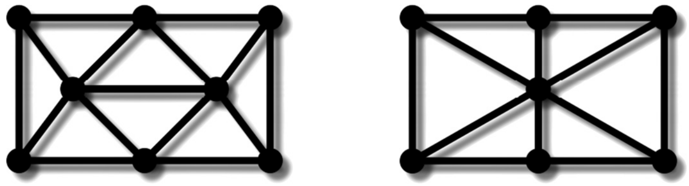


FIG. 3.4. *Edge collapse operation on the mesh graph.*

3.3). Again, we must ensure this operation does not introduce any intersections. A simple way of doing this is to check that no existing edge intersects the two new triangles and that no point lies inside the tetrahedron formed by the two new and two old triangles. We also reject the edge flip if it introduces a change in volume greater than a user-defined maximum volume change (we denote this maximum volume change by γ and usually set it to be $0.1\xi^3$, where ξ is the average edge length at the beginning of the simulation; for simulations involving extremely thin surfaces such as our curl noise example later, this may need to be further reduced). We extend this operation to handle nonmanifold edges by choosing *any pair* of incident triangles with consistent orientation and applying these steps to the edge and the chosen pair of triangles.

Flipping a single edge may introduce new triangles with poor aspect ratios, so we iteratively sweep over all edges in the mesh until no flip is performed or until we reach a maximum number of sweeps (in our implementation, we set this maximum to five). We also require that the new edge length decrease by a minimum amount to prevent the same edge from flipping back and forth on subsequent sweeps.

3.5.3. Edge collapse. If an edge is shorter than a user-defined minimum edge length (denoted β), we attempt to collapse it by replacing it with a single vertex as shown in Figure 3.4. As with edge splitting, we treat only manifold edges, skipping edges incident on more than two triangles. We again use a subdivision scheme to choose the location of the new single vertex in the general case. However, we also use an eigen-decomposition of the quadric metric tensor to detect vertices that lie on ridges or creases [15]. If one edge end point lies on a ridge and the other lies on a smooth patch of the surface, we set the new vertex position to be the position of the existing vertex on the ridge. In other words, we wish to prevent vertices moving off of the ridge, which tends to introduce bumps or jagged edges.

To ensure collision safety, we can use the same pseudomotion collision detection described in section 3.5.1, this time with two vertices in motion: the end points of the edge. These end vertices will have the same predicted location: the location chosen by the subdivision algorithm. If a collision is detected during this pseudomotion, we can try again, this time moving the vertices towards the edge midpoint. Unlike

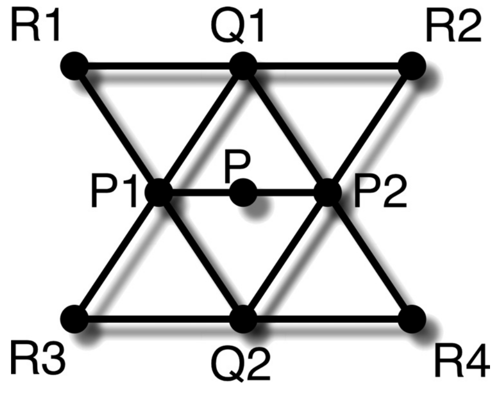


FIG. 3.5. *Butterfly subdivision.*

edge splitting, however, we have no safe fallback vertex location. If we cannot find a collision-free trajectory, the edge collapse must be abandoned.

We use simple minimum and maximum edge lengths for determining when to split and collapse edges. In practice, we compute the average edge length when the mesh is initialized and set the minimum and maximum edge length parameters to be some fractions of the initial average length ξ. This has the effect of keeping the edge lengths within some range of the initial average using split and collapse operations. In our examples, we allow edge lengths to vary between 0.5 and 1.5 of the initial average edge length; however, these parameters did not require tuning and our system remains stable for other values. More sophisticated criteria for triggering a split or collapse exist, such as detecting triangles whose areas are too small or too large, or aspect ratios that are too far from unity (c.f. Jiao [15]).

When choosing locations for new vertices during an edge collapse or split operation, there are a number of schemes that can be used. We use traditional *butterfly subdivision* [7] due to the simplicity of its implementation and because it is free of parameters. Quadric error minimization schemes [9] are promising, but in our experience the simplicity and quality of butterfly subdivision made it more attractive.

Butterfly subdivision determines the location of a new edge midpoint P_{new} as

$$P_{new} = \frac{1}{16}(8(P_1 + P_2) + 2(Q_1 + Q_2) - (R_1 + R_2 + R_3 + R_4)),$$

where P_1 and P_2 are end points of the edge, Q_1 and Q_2 are the vertices on the two triangles incident on the edge which are not the edge end points, and $R_1 \ldots R_4$ are the vertices on the four triangles adjacent to the triangles incident on the edge (see Figure 3.5).

3.6. Mesh separation. As mentioned in section 3.2, we do allow surfaces which are not strictly manifold. In particular, we allow more than two triangles to be incident on an edge. We do not, however, allow two triangles to share the same three vertices, thus creating a zero-volume tetrahedron. An edge collapse or flip operation may introduce such degenerate tetrahedra, so after performing either of these operations, we search the surface meshes for degenerate tetrahedra and delete the two offending triangles. We also delete triangles that may have repeated vertices ("collapsed" triangles).

After this sweep, we deal with surfaces which may be connected only at a single vertex. These so-called "singular" vertices can be detected if their incident triangles are not all connected. If this is the case, we partition the set of incident triangles into connected components. For each component, we create a duplicate vertex and map all triangles in the component to this new vertex. A similar procedure is described by Guéziec et al. [12]. We also move the duplicate vertices very slightly towards the centroid of their associated triangles to avoid problems with collision detection and resolution which may occur when two vertices occupy exactly the same point in space.

The mesh zippering operations described in section 3.7 cannot separate or "pinch" a mesh to create two disjoint volumes. However, the combination of removing degenerate tetrahedrons and the "duplicate-and-separate" operation on singular vertices does allow for mesh separation. If we wish to avoid topological changes altogether, as may be the case, we modify the edge collapse and edge flip operations to check if any degeneracies or singular vertices would result and, if so, abort the operation.

3.6.1. Null-space smoothing. A powerful mesh improvement technique was recently introduced by Jiao [15]. Applying a Laplacian filter to the vertex locations would move each vertex to the average of its neighbors' locations. This usually has the desirable effect of equalizing edge lengths. However, it will also shrink the volume enclosed by the surface and smooth away sharp features. We instead move the vertices only in the null-space of their associated quadric metric tensors. If the vertex is on a flat or smoothly curved patch of surface, the null space will correspond to the plane tangential to the surface at the vertex. If the vertex is on a ridge, the null space will be the infinite line defined by the ridge, and the smoothing operation preserves the ridge feature. If the vertex is at a corner, the null space will be empty and the vertex will not move, preserving the corner.

To ensure no mesh intersection, we treat the global smoothing operation as a pseudotrajectory on all vertices and apply collision resolution as if the surface was moving under the influence of an external velocity field (see section 3.8).

3.7. Topological changes. We have described several mesh quality maintenance operations that can be performed without introducing geometric intersections. To make our general surface tracking algorithm useful for a wider range of applications (e.g., fluid simulation), we must allow surfaces to change topology. We have seen in section 3.6 how our method allows meshes to separate when they become too thin. We now describe a method for allowing surface patches which are close to each other to merge without introducing any intersections.

To achieve this, we seek out edges that are close together and attempt to merge the surface. We use proximity detection to search for edges that are closer than a specified tolerance (considering only edges that are incident on two triangles). We sort the pairs of edges in order of increasing separation distance so that the nearest edges are merged first. For each pair in the sorted list, we first remove the triangles incident on each edge. This introduces two temporary "holes" in the mesh, each hole consisting of a loop of four boundary edges. We create eight new triangles between the two holes, using a closed-form triangulation. We then use intersection testing to determine if these new triangles intersect any existing mesh elements or each other (treating degenerate cases as intersections for safety). If an intersection is found, we discard the new triangles and replace the original triangles incident on the proximal edges, abandoning the topology change.

Similar to edge collapsing and flipping above, this merge operation may introduce degenerate tetrahedra and singular vertices which must be handled as described in section 3.6.

3.8. Collision resolution. After performing mesh improvement and any topological operations, all that remains is determining the surface velocity and integrating the vertex positions forward in a collision-free manner. We allow the user to specify per-vertex velocities, and the goal of our time integration scheme is to produce a final mesh configuration that is as close to the specified trajectory as possible while being free of intersections.

Our collision resolution procedure is based on the filtering approach for handling collisions [2] and operates in three phases. First, we run proximity detection as described in section 3.4 to obtain pairs of elements that are closer to each other than ϵ_p. For each pair of proximal elements, we compute the relative normal velocity of the elements. We then perturb the vertex velocities so that the new relative normal velocity is positive and large enough to carry the vertices at least ϵ_p away from each other if they were integrated forward for Δt without further interference. Attempting to maintain this small minimum separation significantly helps in avoiding degenerate geometric cases which would otherwise slow subsequent floating-point-based collision detection and resolution.

As described in section 3.4, for a pair of elements, proximity detection returns a distance d and a set of scaled barycentric coordinates $\bar{\mathbf{a}}$. If $\mathbf{p}$ is the vector of element vertice indices and $\mathbf{u}$ is the vector of vertex velocities, then the relative velocity is

$$\mathbf{u}_{\text{rel}} = \sum_{i=1}^{4} \bar{a}_i \mathbf{u}[p_i].$$

If $\mathbf{n}$ is the unit-length collision normal, the impulse J we apply is computed as

$$\delta = \frac{\epsilon_p - d}{\Delta t} - \mathbf{n} \cdot \mathbf{u}_{\text{rel}},$$
$$J = \frac{\delta}{\langle \bar{\mathbf{a}}, \bar{\mathbf{a}} \rangle_{M^{-1}}},$$

where M is the diagonal matrix of vertex masses. (In problems where there is no natural mass for a surface vertex, we simply use a unit weighting $M = I$.)

Then, for each vertex in proximity, we distribute the impulse J to perturb the predicted velocity field:

$$\mathbf{u}_{p_i} = \mathbf{u}_{p_i} + J \frac{\bar{a}}{M_{p_i}} \mathbf{n}.$$

Note that this will not immediately resolve any of the proximities detected, as the "current" vertex positions are left untouched; it aims to resolve the proximity at the next time step. More importantly, it tends to dramatically reduce the number of collisions that must be dealt with in the next phase.

In the second phase of collision resolution, we use continuous collision detection to determine pairs of colliding elements. Our CCD function returns the relative displacement of the elements in the direction of the collision normal (which we can scale by $1/\Delta t$ to compute the relative normal velocity), as well as the barycentric coordinates that should be used to distribute the corrective impulse. For each pair of colliding elements we encounter, we apply an impulse that sets the relative normal velocity between the elements to zero, thus preventing the collision from occurring. This is similar to the repulsion impulses applied in the previous phase except that the impulse magnitude is

$$\delta = -\frac{\mathbf{n} \cdot \mathbf{\Delta x}_{\mathrm{rel}}}{\Delta t}.$$

This is equivalent to introducing an impulse that instantaneously changes the velocity while minimizing the velocity change in the normal direction in a least-squares sense. (Minimizing the normal velocity change in this way ensures that momentum is conserved if the least-squares metric is kinetic energy.) One sweep through all mesh elements will not prevent all collisions, as resolving one collision may introduce a new collision between a pair of elements that was already checked. We have found that applying three sweeps of this individual collision resolution handles most collisions; however we must use a fail-safe to ensure that all collisions are handled.

For our fail-safe, we use the simultaneous treatment of collisions developed by Harmon et al. [14]. After three passes of individual collision resolution, we detect all pairs of elements that are still colliding. We lump colliding pairs of elements into "impact zones" based on adjacency and resolve all collisions in each zone simultaneously using one linear solve. We can think of our desired new velocities $\mathbf{u}'$ as being the solution to a constrained minimization problem:

$$\begin{aligned}&\min ||\mathbf{u}' - \mathbf{u}||_M^2\\ &\text{subject to } \mathbf{n} \cdot \mathbf{u}'_{\mathrm{rel}} = 0 \quad \text{for all collisions.}\end{aligned}$$

We can rewrite the constraint as a linear operator on the vertex velocities by building a matrix C, where each row C_i corresponds to one collision and has nonzero entries in block columns $\mathbf{j} = [3v, 3v+1, 3v+2]$, where v is one of the four vertices involved in collision i. Setting $C_{i,\mathbf{j}} = \bar{a}_v \mathbf{n}^T$, our constrained optimization problem becomes

$$\begin{aligned}&\min ||\mathbf{u}' - \mathbf{u}||_M^2\\ &\text{subject to } C\mathbf{u} = \mathbf{0}.\end{aligned}$$

Solving this using the method of Lagrange multipliers yields the system

$$CM^{-1}C^T\lambda = C\mathbf{u}.$$

We can think of λ as the set of impulses which, when applied, yields zero relative normal velocities for all collisions. We update the vertex velocities according to

$$\mathbf{u}' = \mathbf{u} + M^{-1}C^T\lambda.$$

The application of these impulses may result in new collisions, so we run collision detection again and add any additional collisions to the set of impact zones, repeating the process until no new collisions are detected. This is guaranteed to terminate, assuming adequately accurate linear solves, since each additional constraint reduces the finite dimension of the solution space; in practice, it proves to be very efficient.

At the end of each time step, we verify that no tangling has occurred by running intersection tests on all edge-triangle pairs; while not necessary for the method as presented, this is a useful practice for detecting programming errors during software development.

3.8.1. Error introduced by collision resolution. Each individual collision response and repulsion force perturbs the vertex location by $O(\Delta t)$. We posit that the number of collision events for a given vertex in any numerically-resolved simulation should be small and finite for a fixed end-time, with the collision events in the limit becoming a set of measure zero. Since each collision perturbation has magnitude at most $O(\Delta t)$, this implies the collision resolution should introduce at worst a global $O(\Delta t)$ error, so we should achieve at least first-order convergence. When surface elements collide or merge, we introduce an error similar to that introduced into the level set method by the kink in the signed distance field; at a fundamental level topological changes are nonsmooth and unlikely to permit greater than first-order accuracy in any numerical method.

As a caveat, we note that the $O(\Delta t)$ error produced by the impact zone solver could potentially involve a very large constant, since it will perturb the velocity field at many points. However, we have found that impact zones are seldom used in practice, after one pass of repulsion forces and three passes of impulse-based collision resolution. Adaptively cutting the time step size has been a successful strategy for reducing the number of collisions [2] and could also be used in the case where impact zones grew too large. However, we shall see in the next section that our method shows convergence under mesh refinement, even without cutting the time step size; this does not appear to be a concern in practice.

4. Numerical examples.

Armed with techniques for guaranteeing intersection-free meshes, changing mesh topology, and maintaining mesh quality, we attack some dynamic surface problems traditionally handled by implicit surfaces.

4.1. Motion in the normal direction.

We compare motion in the normal direction using an explicit method with a level set method. Our explicit method uses the entropy solution of the *face offsetting* algorithm [15] to achieve normal motion. For this advection scheme to function properly, we must guard against operations that will invert a patch on the surface. Thus we reject any edge collapse or edge flip operation that results in a triangle with a normal that is too different from the original triangle normals. We also adjust the time step to avoid inverting any triangles over a single offsetting step following the method described in the original face offsetting paper [15].

We ran motion in the normal direction on two disjoint spheres with speed of 0.2 for $t = [0, 1)$, and then with speed of -0.2 for $t = [1, 2]$. The spheres initially have centers at $(-0.25, 0, 0)$ and $(0.25, 0, 0)$, and radius of 0.2. We used marching tiles [28] to generate a mesh with an initial average edge length of approximately 0.02. Let α be the maximum edge length, β be the minimum edge length, γ be the maximum volume change, and ξ be the initial average edge length. We set $\alpha = 1.5\xi$, $\beta = 0.5\xi$, and $\gamma = 0.1\xi^3$. The time step Δt was generally set to 0.005 but was shortened for some time steps to avoid inverting triangles as mentioned above.

Figure 4.1 shows our method at $t = 0, 1,$ and 2. Note that our method cleanly handles merging of the two spheres. In this figure we compare our method against the level set method, using the toolbox of level set methods [20]. The grid resolution used was $100 \times 50 \times 50$, resulting in a grid spacing of $\Delta x = 0.02$, similar to the average initial edge length of the triangle mesh. We used a 5th-order weighted essentially non-oscillatory (WENO5) spatial derivative approximation and a third-order TVD Runge–Kutta (RK) time integrator with the same time step size of $\Delta t = 0.005$.

We also ran this example on lower- and higher-resolution initial meshes to determine convergence. Table 4.1 compares the initial average edge length to the L_∞ and

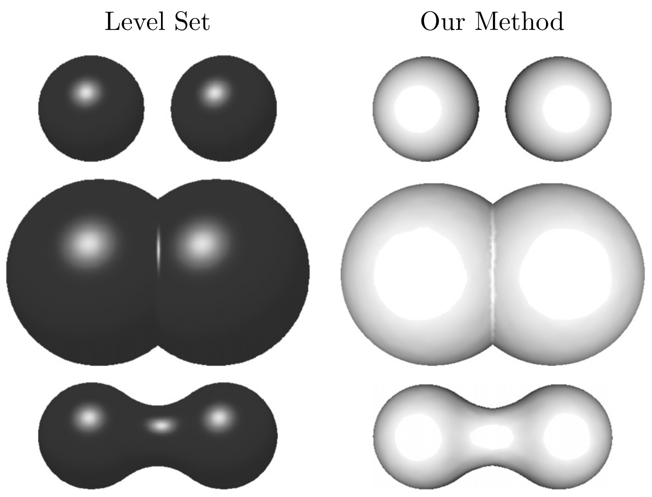


FIG. 4.1. *Motion in the normal direction.*

TABLE 4.1
Motion in the normal direction: error measures for the new method.

Initial average edge length	L_∞ error at $t = 2$	L_1 error at $t = 2$
0.04	0.00811073	0.00114493
0.02	0.00236906	0.00042705
0.01	0.00129011	0.000174606

TABLE 4.2
Motion in the normal direction: error measures for the level set method.

Grid spacing (Δx)	L_∞ error at $t = 2$
0.04	0.0118
0.02	0.0053
0.01	0.0031
0.005	0.0017

L_1 errors, obtained by comparing with the analytic exact solution for time $t = 2$. The results are consistent with first-order convergence. We used the same initialization values for Δt and set α, β, and γ according to the same fraction of ξ throughout. Table 4.2 shows a comparable rate of convergence when using the level set method with RK3 time integration and upwind WENO5 spatial derivatives. Note that, despite the use of high-resolution numerical methods, the topology change reduces the level set method to first-order accuracy, due to the kink in the signed distance field at the merging event.

A first instinct may be to use a simpler scheme for specifying normal motion, namely, using "vertex normals" to specify the direction of motion, and then simply advecting the vertices according to this direction. However, this scheme does not produce the entropy solution, as it does not correctly handle flow fields with merging characteristics (as argued by Enright et al. [8], for example). To confirm this, we computed vertex normals as the area-weighted average of normals of incident triangles. We then ran our test again, advecting the vertices according to these computed normals. Comparing with the analytic entropy solution revealed that this method

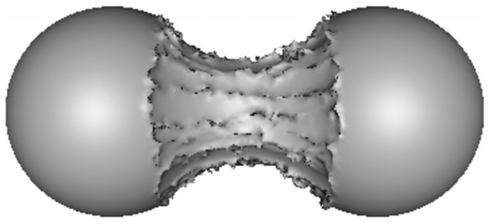


FIG. 4.2. *Motion in the normal direction using vertex normals.*

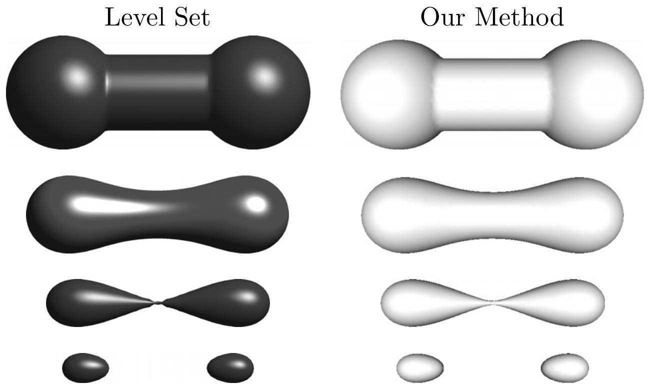


FIG. 4.3. *Motion by mean curvature at $t = 0$, $t = 0.0125$, $t = 0.0219$, and $t = 0.025$.*

does not converge as the mesh is refined. Figure 4.2 shows the results of this test on a high-resolution mesh.

4.2. Motion by mean curvature. Motion in the normal direction with speed proportional to mean curvature is a classic geometric flow used for testing surface tracking methods. We ran motion by mean curvature on a dumbbell-shaped surface as described by Sethian [26]. This causes surface separation, as the dumbbell "handle" shrinks faster due to its higher mean curvature. We estimated mean curvature times the surface normal at each vertex using the scheme introduced by Meyer et al. [19] and then moved vertices using simple forward Euler time integration, with time step size limited by the CFL condition. Initial conditions were generated by running marching tiles on a $28 \times 14 \times 14$ grid, generating an average initial edge length ξ of approximately 0.02. We again set $\alpha = 1.5\xi$, $\beta = 0.5\xi$, and $\gamma = 0.1\xi^3$.

Figure 4.3 shows a comparison with the level set method. For the level set example, we again use the toolbox of level set methods with a $100 \times 50 \times 50$ grid with $\Delta x = 0.02$, a second-order spatial derivative scheme for estimating curvature, and a third-order TVD RK time integration scheme with time step size dictated by the CFL condition.

To determine convergence in this case, we compare our solution against a high-resolution ($200 \times 100 \times 100$) solution produced by the toolbox of level set methods. Table 4.3 compares the initial average edge length to the error after integration. The results are consistent with first-order convergence, which again is probably very hard to improve upon in the presence of topological change.

4.3. Motion by external flows. We subjected our method to the *Enright test*, which was developed to test the accuracy of surface tracking methods [8]. The initial

TABLE 4.3
Motion by mean curvature: error measures.

Initial edge length	L_∞ error at $t = 0.025$	L_1 error at $t = 0.025$
0.04	6.18499×10^{-4}	2.62302×10^{-4}
0.02	1.44355×10^{-4}	7.02612×10^{-5}
0.01	1.47687×10^{-5}	1.54722×10^{-5}

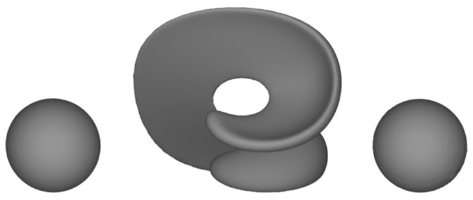


FIG. 4.4. *The "Enright test."*

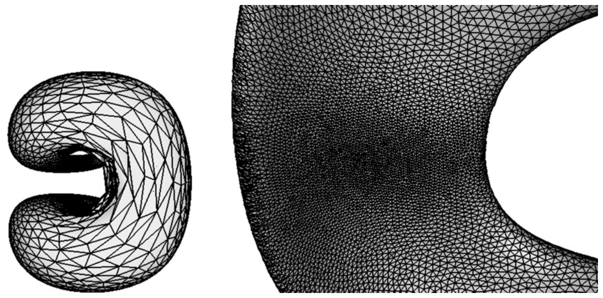
mesh is a sphere centred at $(0.35, 0.35, 0.35)$ with radius 0.15. The mesh is advected by the velocity field given by

$$u(x, y, z) = 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z),$$
$$v(x, y, z) = -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z),$$
$$w(x, y, z) = -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z).$$

This velocity field is modulated by the term $\sin(2/3\pi t)$ to achieve a smooth, periodic motion. The initial mesh is generated by marching tiles from a $14 \times 14 \times 14$ grid, generating an average initial edge length of $\xi = 0.01$. We initialize the mesh maintenance parameters as $\alpha = 1.5\xi$, $\beta = 0.5\xi$, and $\gamma = 0.1\xi^3$. We use a fourth order Runge–Kutta scheme to advect the mesh vertices, with a time step of 0.01. After one period of motion, the volume enclosed by the surface has changed by just 2.48899×10^{-5}, resulting in a relative error of 0.1764 percent. Figure 4.4 shows the initial, mid-period, and final surface.

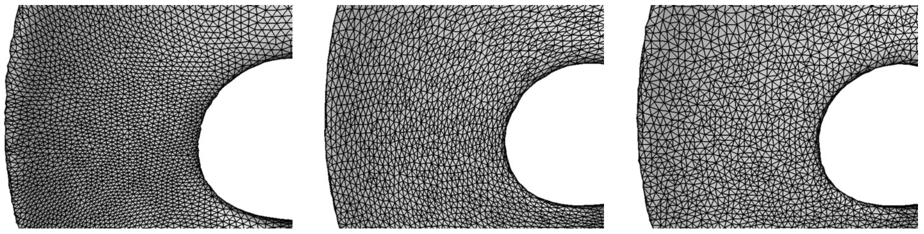
To illustrate the effects of our mesh adaptivity operations, we also ran the Enright test with various mesh maintenance operations turned off. Figure 4.5(a) shows a mesh where no edge splitting had been performed. Note that the resulting large triangles poorly capture the curvature of the surface. Turning off edge collapse generates high triangle density when the surface is contracting, as in Figure 4.5(b), resulting in wasted computational effort. Figure 4.6 shows the nonuniform triangulations resulting when edge flipping and null-space smoothing operations are turned off. Since the patch shown has relatively low curvature, a uniform triangulation is desirable but without flipping and null-space smoothing, the resulting triangulation is irregular.

We also advect an initially spherical surface with a smooth, pseudorandom, divergence-free velocity field using curl-noise [3] with RK4 time integration (see Figure 4.7). It would be very challenging for a grid-based, implicit method with similar resolution to resolve the extremely thin structures which are produced. We restrict the rotational motion to be planar by generating a random spline potential with zero x- and y-components and taking the curl of this potential. We do this only to aid visualization, eliminating occlusion that occurs when using a fully 3D rotation field. We disallowed topological changes and set the maximum allowed change in volume γ to be very small ($5 \times 10^{-4}\xi^3$) to faithfully capture the thin structures. At time $t = 30$, the total volume enclosed by the surface has changed by 0.9211 percent.



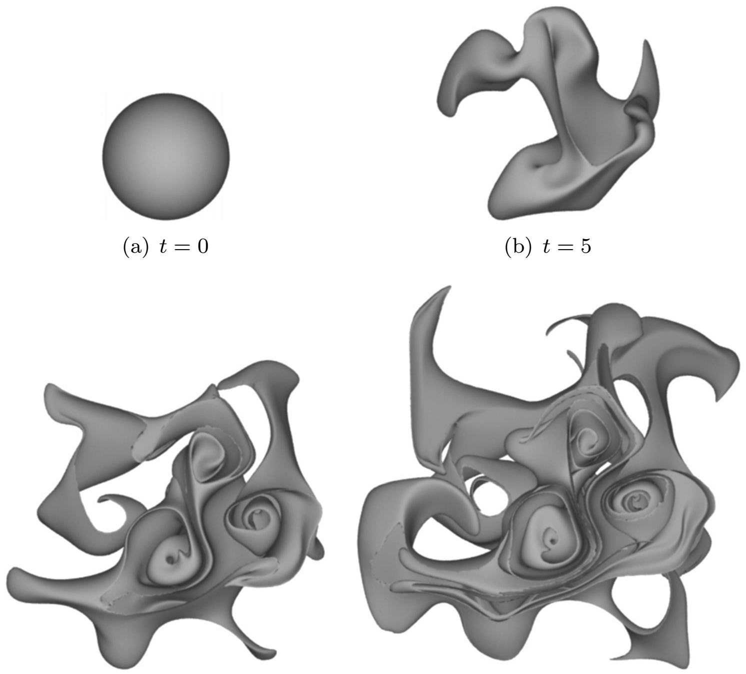
(a) No edge split (b) No edge collapse

FIG. 4.5. *Effect of edge splitting and collapsing.*



(a) Flipping and smoothing (b) No edge flip (c) No smoothing

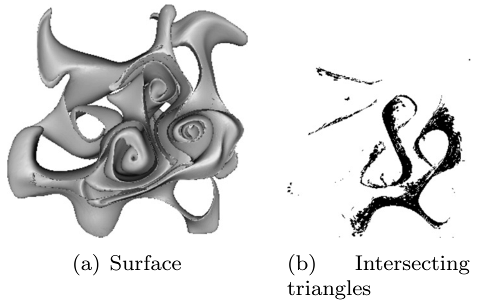
FIG. 4.6. *Effect of edge flipping and vertex smoothing.*



(a) $t = 0$ (b) $t = 5$

(c) $t = 15$ (d) $t = 30$

FIG. 4.7. *Motion by curl noise.*



(a) Surface (b) Intersecting triangles

FIG. 4.8. *Motion by curl noise with no collision resolution.*

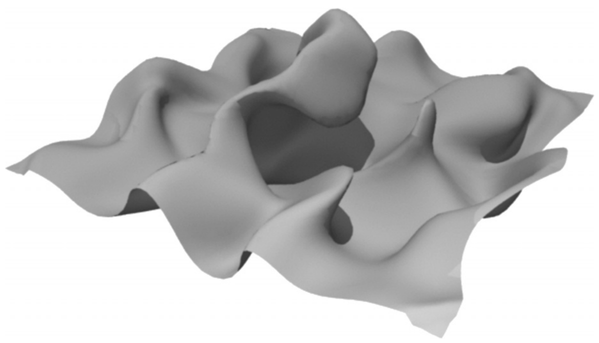


FIG. 4.9. *Motion by curl noise on an open surface.*

We ran this test a second time with collision detection turned off, allowing mesh elements to intersect. Notice that, with a smooth velocity field defined everywhere in space, the surface should never self-intersect, but—even with a high-order time integration scheme—self-intersections do occur due to discretization into triangles, numerical error, and collision-oblivious mesh improvement operations. Figure 4.8 shows one screen capture showing the entire mesh and one showing only the set of intersecting triangles.

4.4. Extension to open surfaces. With some modification, we can extend our algorithm to handle open surfaces. Figure 4.9 shows a curl noise velocity field advecting an open surface. In such scenarios, we must be careful when dealing with edges incident on only one triangle. In our example, we handle these edges in the simplest way possible: We disallow flipping, collapsing, and splitting of any such edges, as well as topology changes. The remaining triangles and edges on the surface are unchanged.

The ability to handle open surfaces suggests a possible further extension to periodic surfaces, although we have not yet attempted to implement this. Another further extension would be allowing an odd number of triangles incident on an edge in order to represent, for example, a solid-fluid-air triple point in a fluid simulation.

4.5. Performance. Table 4.4 shows timings for our method running the motion-in-the-normal-direction example. We list the time for setting velocity on the surface, detecting and handling collisions, performing topological operations, and improving mesh quality per time step. All computations were performed on a single core of a 2.4 GHz Intel Core2 Duo with 4 GB main memory. We have not attempted a parallel

Table 4.4
Execution time per step (in seconds).

Δx	Triangles	Improvement	Topology	Set velocity	Collisions
0.04	1476–3512	0.1104	0.04171	0.04031	0.06529
0.02	5896–14222	0.5203	0.2115	0.1602	0.3123
0.01	23372–56390	2.07845	0.845781	0.617641	1.41106

Table 4.5
Mesh resolution and run time.

Example	Triangles	Run time	Level set resolution	Level set run time
Normal direction	5896–14251	617s	$100 \times 50 \times 50$	1435s
Mean curvature	3354–15468	1537s	$100 \times 50 \times 50$	666s
Enright test	6614–25176	597s	$100 \times 100 \times 100$	N/A
Curl noise	8798–381092	608m	N/A	N/A

implementation, but as the majority of our operations are local in nature, it should be possible to spread the work over several processors.

The timings for topology change and mesh improvement operations include the time taken for intersection and collision queries to ensure collision safety. The speed of such interference detection depends greatly on the broad-phase collision culling strategy. We use a simple regular grid of bounding boxes for each mesh element type (vertex, edge, and triangle), which theoretically scales linearly with the number of elements. We do achieve linear scaling in the number of objects tested after broad-phase culling but only near-linear scaling in execution time. Our immediate future work will be to investigate and optimize our broad-phase algorithm to achieve linear scaling in actual execution time.

Table 4.5 lists the minimum and maximum numbers of triangles for each example, as well as the total run time. Where appropriate, we also list the grid resolution of the level set grid used for comparison. We did not run the Enright test using the level set method, but we include the resolution of the grid used in the original particle level set paper [8] for comparison. Note that, in the particle level set method, the grid is augmented with marker particles (64 particles in each grid cell located within 3 cells of the initial interface), effectively increasing the resolution of their method even further.

Since we are using a MATLAB implementation of the level set algorithms, direct comparison of timings against our unoptimized C++ implementation is difficult, but we include run times for a very general idea of how our method compares. We ran the level set examples for motion in the normal direction and motion by mean curvature on a Sun x4600 M2 with 4 dual core Intel x64 processors (at 2.8 GHz) and 128 GB shared main memory.

5. Conclusions and future work.

We presented a method for robustly handling topological changes in surfaces represented as triangle meshes, addressing one of the major obstacles in using such explicit surface tracking methods. The use of robust interference detection, topological operations, and fail-safe collision handling provides the framework for guaranteeing intersection-free surfaces while still allowing merging and separation. We presented a collection of mesh maintenance operations to improve the quality of the surface discretization. Finally, we presented results of numerical experiments, comparing our method to the level set method for geometric flows.

A public domain C++ implementation of our algorithm, including all the examples presented in this paper, is available on the web at http://www.cs.ubc.ca/labs/imager/tr/2009/eltopo/.

The next obvious step is the use of our method in more concrete applications. Immediate application domains include the physical simulation of multiphase fluid flows and deformable models for volume segmentation in medical imaging.

Anisotropic mesh adaptation as described by Jiao et al. [16] is a promising technique for accurately resolving highly curved and thin surfaces, and we would like to introduce it into our system. Their method involves variations to the same four basic mesh maintenance techniques we use in this paper and so should not be hard to integrate.

As collision detection is a major bottleneck of our system, we hope to investigate and implement a more efficient broad-phase culling technique to speed up our method.

The interplay between edge splitting, flipping, and collapsing is subtle, and the parameters that determine when these operations are performed are important to the resulting quality. A more thorough analysis of how to choose these parameters will likely prove very useful.

Acknowledgments. We would like to thank the anonymous reviewers for their constructive comments and suggestions.

REFERENCES

[1] A. W. Bargteil, T. G. Goktekin, J. F. O'Brien, and J. A. Strain, *A semi-Lagrangian contouring method for fluid simulation*, ACM Trans. Graph., 25 (2006), pp. 19–38.

[2] R. Bridson, R. Fedkiw, and J. Anderson, *Robust treatment of collisions, contact and friction for cloth animation*, ACM Trans. Graph., 21 (2002), pp. 594–603.

[3] R. Bridson, J. Houriham, and M. Nordenstam, *Curl-noise for procedural fluid flow*, ACM Trans. Graph., 26 (2007), p. 46.

[4] T. Brochu and R. Bridson, *Numerically Robust Continuous Collision Detection for Dynamic Explicit Surfaces*, Technical report TR-2009-03, University of British Columbia, Vancouver, 2009.

[5] R. DeBar, *Fundamentals of the KRAKEN code*, Technical report UCID-17366, California Univ., Livermore (USA), Lawrence Livermore Laboratories, 1974.

[6] J. Du, B. Fix, J. Glimm, X. Jia, X. Li, Y. Li, and L. Wu, *A simple package for front tracking*, J. Comput. Phys., 213 (2006), pp. 613–628.

[7] N. Dyn, D. Levine, and J. A. Gregory, *A butterfly subdivision scheme for surface interpolation with tension control*, ACM Trans. Graph., 9 (1990), pp. 160–169.

[8] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell, *A hybrid particle level set method for improved interface capturing*, J. Comput. Phys., 183 (2002), pp. 83–116.

[9] M. Garland and P. Heckbert, *Surface simplification using quadric error metrics*, in Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97), New York, ACM Press/Addison–Wesley Publishing, 1997, pp. 209–216.

[10] J. Glimm, J. W. Grove, X. L. Li, K. Shyue, Y. Zeng, and S. Zhang, *Three dimensional front tracking*, SIAM J. Sci. Comput., 19 (1998), pp. 703–727.

[11] J. Glimm, J. W. Grove, X. L. Li, and D. C. Tan, *Robust computational algorithms for dynamic interface tracking in three dimensions*, SIAM J. Sci. Comput., 21 (2000), pp. 2240–2256.

[12] A. Guéziec, G. Taubin, F. Lazarus, and B. Horn, *Cutting and stitching: Converting sets of polygons to manifold surfaces*, IEEE Trans. Vis. Comput. Graph., 7 (2001), pp. 136–151.

[13] F. H. Harlow and J. E. Welch, *Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface*, Phys. Fluids, 8 (1965), pp. 2182–2189.

[14] D. Harmon, E. Vouga, R. Tamstorf, and E. Grinspun, *Robust treatment of simultaneous collisions*, ACM Trans. Graph., 27 (2008), pp. 1–4.

[15] X. Jiao, *Face offsetting: A unified framework for explicit moving interfaces*, J. Comput. Phys., 220 (2007), pp. 612–625.

[16] X. Jiao, A. Colombi, X. Ni, and J. C. Hart, *Anisotropic mesh adaptation for evolving triangulated surfaces*, in Proceedings of the International Meshing Roundtable, Birmingham, AL, Sandia National Laboratory, 2006, pp. 173–190.

[17] J.-O. Lachaud and B. Taton, *Deformable model with adaptive mesh and automated topology changes*, in Proceedings of the 4th International Conference on 3-D Digital Imaging and Modeling (3DIM'2003), Banff, Alberta, Canada, M. Rioux, P. Boulanger, and G. Godin, eds., IEEE Computer Society Press, Piscataway, NJ, 2003.

[18] F. Losasso, F. Gibou, and R. Fedkiw, *Simulating water and smoke with an octree data structure*, ACM Trans. Graph., 23 (2004), pp. 457–462.

[19] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr, *Discrete differential-geometry operators for triangulated* 2*-manifolds*, in Proceedings of Visualization and Mathematics (VisMath) 2002 Berlin, Technical University of Berlin, 2002.

[20] I. M. Mitchell, *The flexible, extensible and efficient toolbox of level set methods*, J. Sci. Comput., 35 (2008), pp. 300–329.

[21] W. Noh and P. Woodward, *SLIC (simple line interface calculation)*, in Proceedings of the 5th International Conference on Numerical Methods in Fluid Dynamics Seoul, Lecture Notes in Physics 59, A. I. van de Vooren and P. J. Zandbergen, eds., Springer-Verlag, Berlin, 1976, pp. 330–340.

[22] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Appl. Math. Sci. 153, Springer-Verlag, New York, 2003.

[23] S. Osher and J. A. Sethian, *Fronts propagating with curvature dependent speed: Algorithms based on Hamilton-Jacobi formulations*, J. Comput. Phys., 79 (1988), pp. 12–49.

[24] J.-P. Pons and J.-D. Boissonnat, *Delaunay deformable models: Topology-adaptive meshes based on the restricted Delaunay triangulation*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2007, Minneapolis, MN, 2007.

[25] W. Rider and D. Kothe, *Reconstructing volume tracking*, J. Comput. Phys., 141 (1998), pp. 141–112.

[26] J. A. Sethian, *Level Set Methods and Fast Marching Methods*, Cambridge University Press, Cambridge, 1999.

[27] E. Sifakis, S. Marino, and J. Teran, *Globally coupled impulse-based collision handling for cloth simulation*, in Proceedings of the ACM Siggraph/Eurographics Symposium on Computer Animation (SCA), Dublin, Ireland, 2008, pp. 147–153.

[28] B. W. Williams, *Fluid Surface Reconstruction from Particles*, Master's thesis, University of British Columbia, Vancouver, 2008.