ABSTRACT

Electrophysiological modeling of dendrites and other neurological processes is generally done in a simplified manner, by treating these structures as a series of cylinders (aka cable models). 3D reconstruction from serial section transmission electron microscopy (ssTEM) reveals highly complex and densely packed neurons that, if accurately modeled, promises to improve our discovery of their electrical functions. We present a method for reconstruction of spatially realistic and topologically correct 3D models. Previous work in 3D reconstruction from serial contours has focused on reconstructing one object at a time, potentially producing inter-object intersections between slices. We have developed a robust algorithm that removes these intersections using a geometric approach. Our method not only removes intersections but can guarantee a given minimum separation of objects. This paper describes the algorithm for geometric adjustment and presents several results of our high-fidelity modeling.

Categories and Subject Descriptors
H.4 [Biomedical applications of shape and solid modeling]: Curve, surface, and manifold modeling
Mesh processing

1. INTRODUCTION

Understanding of the brain and other biological structures and systems relies heavily on accurate modeling and simulation in a variety of settings. Previously this modeling has been done using the simplified cases of treating the dendritic arbor as a series of cylinders. However, serial section transmission electron microscopy (ssTEM) reveals highly complex geometries among neuronal processes (e.g. axons, dendrites, glial cells), including high tortuosity, varying caliber, spiny protrusions and extremely tight packing. Our work of simulating neuronal electrophysiological function at high resolution requires very accurate and topologically correct 3D reconstructions of neuronal processes [????]. These reconstructions can be used in a variety of modeling experiments using high-fidelity versions of the traditional cable model approach [????] or, more recently, an approach based on meshless Weighted Extended B-spline-based finite element methods [????].

The foundation for generating these neuronal reconstructions is generally ssTEM imagery. Neuronal contours are generated from these images by neurobiology experts who trace out the contours using a variety of tools [19]. Once these contours are available, the problem can be cast into a classic 3D reconstruction problem using planar, serial contours as the input. A large amount of research has been done on methods solving this problem.

Unfortunately these methods generally reconstruct one object at a time, impervious to how other objects in the scene may be reconstructed. While not generally a problem in reconstruction of single objects (e.g. heart, skeleton), it can cause great difficulty when dealing with multiple objects, especially when these objects are of highly complex geometries and tightly packed. Such is the problem in neuronal reconstruction.

The problem lies in the anisotropy of the data – pixel spacing in the xy plane of neuronal ssTEM images is roughly 2-5 nm, while spacing between slices is closer to 45 nm. In addition, extracellular spacing (spacing between neuronal processes) is on the order tens of nanometers [20, 15]. This close spacing, combined with the comparatively large distance between slices can cause inter-object intersections between slices. This topological incorrectness prohibits any type of analysis based on finite element methods.

Very little work has been done to ensure inter-object topological correctness, and this work aims to fill the gap. We have developed an intersection removal method that acts in concert with any surface reconstruction method, provided it conforms to several criteria. Our approach is to adjust surface vertices and induced vertices along the z axis. This approach can remove intersections in polynomial time without causing additional intersections. Using an approach that moves vertices in x and or y in addition to z could generate more faithful surfaces but at considerably greater computational complexity, including potential optimization problems.

We first give a brief outline of work that has been done both in single component and multi-component surface reconstruction from cross-sectional contour data (Section 2). We then build up a set of rules and theorems to prove correctness and robustness of our algorithm (Section 3), fol-
followed by a discussion of the surface removal algorithm including complexity bounds (Section 4). We then present results of our implementation (Section 5).

2. RELATED WORK

A large amount of work has been done to solve the single-component reconstruction problem.

Methods of single-component reconstruction have been around for many years. Fuchs et al. [11] presented the problem and proposed a solution based on triangulations guided by a toroidal graph. Barequet and Sharir [5] introduced a method using linear interpolations between slices of medical images. Bajaj et al. [2] expanded on their work by using medial axes to tile regions with no legal slice-to-slice tiling. Oliva et al. [18] specifically targeted difficult objects (objects with multiple branches, holes, and other irregularities) and used Voronoi diagrams to construct topologically correct surfaces.

Somewhat more recent works of the same class interpolate contours using 2D skeletons in valid tile regions [4], Delaunay triangulation [21] and contour morphing [17]. Most works use some type of linear interpolation and thus require a smoothing post-processing step. One approach that performs non-linear smoothing during tiling is found in [6]. Other recent approaches reconstruct surfaces from non-parallel contours [16, 7].

Very different approaches that work directly from image data rather than contours exist, such as energy-minimizing 3D snakes [14] and a 3D extension to path cost-minimizing LiveWire [10]. These approaches require a level of user interactivity, however, and often don’t scale well to massive datasets with large numbers of components.

Surprisingly little work has dealt with the issue of intersections between multiple components. Bajaj and Gillette [1] produced a method for removing intersections by removing contour overlaps in intermediate planes. This algorithm uses an inflated medial axis surface to separate mid-slice contours, but there is no guaranteed bound on the number of mid-slice contours to be separated. In addition, branching of components is treated in a pre-processing step, where branch points are moved as needed to enable intersection removal. Our algorithm has provably finite computational bounds and it also handles branching of components without treating them as a special case.

Of course, there are many algorithms that perform general boolean operations on surfaces [13]. Our algorithm, however, uses the wealth of constraints available to us to perform a highly optimized and robust intersection removal. Guarantees of distance of object separation with very little additional computation also naturally follow as a result of our approach.

3. RULES

Our method is correct and robust and, in addition, it has bounded computational complexity and requires neither parameter optimization nor refinements. The only modifications made to the original surface are the addition of induced points and moving of points parallel to the z-axis. We prove that moving these points in z in a constrained manner can remove intersections without causing additional intersections among other components. In addition, no two points on different components will be moved to be more than $\epsilon$ (an application-specific parameter) away in euclidean distance during the intersection removal. This is an important guarantee for applications that are sensitive to inter-component spacings.

We state here criteria on which our method relies, as well as various theorems which prove correctness and completeness. We define the xy plane to be the plane of the original images (and slice contours), and the z axis to be perpendicular to the xy plane.

**Criterion 1.** The reconstructed surface is a piecewise closed surface of polyhedra.

**Criterion 2.** Any vertical (parallel to the z axis) line segment between two adjacent slices intersects a single component exactly 0 or 1 times, or along exactly one line segment.

**Criterion 3.** Slicing the reconstructed surface on any of the original slices produces exactly the input contours.

**Criterion 4.** No two contours on the same slice intersect.
Figure 2: $s_1$ and $s_2$ are two adjacent slices and the lower plane is an arbitrary $xy$ plane showing containing contours of various points. $p_2$ and $p_3$ are on the green component, while $p_1$ cannot be according to lemma 1. $\mathcal{E}(p_1) = \emptyset$, $\mathcal{E}(p_2) = \{c_1, c_2\}$, $\mathcal{E}(p_3) = \{c_2\}$.

**Criterion 5.** A contour cannot be nested inside a different colored contour.

Criteria 1-3 are borrowed from [2] and are required to ensure high quality and topologically correct surfaces. Criterion 1 ensures that each component surface given to our algorithm are topologically correct and water-tight. Note that this does not guarantee that intersections between components won’t exist. Criterion 2 also applies only to single components and helps to avoid topologies that are unlikely. Without it, a host of additional correspondences are possible, most of which are incorrect. In order to avoid additional complexity and a large number of false positive correspondences, this criterion is carried through into our work. It assumes, however, that the slice spacing is close enough to enable reasonable reconstructions even with the criterion in place. Criterion 3 ensures that interpolation is bounded by only that required to generate a likely topology and avoids adding information to the original contours.

Criterion 4 may seem redundant, as by definition contours on manifold surfaces are non-intersecting. But in practice we’ve found that contours are sometimes imperfect and can contain intersections. We therefore make this criterion explicit. If, in fact, the given contours contain intersections, these can easily be removed (given that each contour is simple) as a pre-processing step. We remove the intersections using a simple overlap removal algorithm.

Criterion 5 is currently required as nested contours are not currently supported. In other words, components are treated as solids without any nested components.

Our algorithm works on slice pairs. That is, all components on a pair of slices are tiled, and then intersections occurring between slices are removed. The algorithm then proceeds to the next pair of slices. In the following discussion, it is assumed that the data we are dealing with lies on and between a pair of adjacent slices. We also deal with only two components at a time, $C^g$ and $C^y$, the green and yellow components, respectively. We will see that working with only two components at a time is justified.

Proofs of the following lemmas and theorems can be found in the appendix.

**Definition 1.** Suppose $p'$ is the projection of point $p$ onto the $xy$ plane. We say that contour $c$ is a containing contour of $p$ if $p'$ is inside or on the boundary of $c'$ (see figure 2). We define $\mathcal{E}(p)$ to be the set of all containing contours of point $p$ and $\mathcal{E}^g(p)$ to be $\mathcal{E}(p)$ restricted to all green contours $\{c^g\}$, i.e., $\mathcal{E}^g(p) = \mathcal{E}(p) \cap \{c^g\}$.

**Lemma 1.** Suppose $p^g$ is a point on the surface $\partial C^g$ of component $C^g$. Then $1 \leq |\mathcal{E}(p^g)| \leq 2$.

This lemma states that any point on a surface must have at least one containing contour (of any color) but no more than two. In figure 2, $p_1$ has no containing contour and so cannot exist on the surface of the reconstructed component.

**Lemma 2.** If $|\mathcal{E}(p^g)| = 2$ then the two contours in $\mathcal{E}(p^g)$ lie on separate slices.

**Definition 2.** $S^g$ is the set of all points $p^g \in \partial C^g$ such that $|\mathcal{E}^g(p^g)| = 2$. These points lie “sandwiched” between two green contours. Similarly, $U^g$ is the set of all points $p^g \in \partial C^g$ such that $|\mathcal{E}^g(p^g)| = 1$. These points have no containing contour on one of the slices. For a point $p^g \in U^g$, we call its single containing green contour the penumbral contour $\mathcal{P}(p^g)$.

In figure 2, $\mathcal{P}(p_3) = c_2$ and the penumbral contour for the other two points is undefined.

As it happens, points in $S^g$ cannot lie on the inside of both contours, but must lie on the boundary of at least one, but this distinction is not necessary for our analysis here.

**Definition 3.** Suppose $p^g \in U^g$. Then $z^g(p^g)$ is the $z$ coordinate value for $\mathcal{P}(p^g)$. We call this the $z$-home value for $p^g$. $z^g(p^g)$ is undefined for all $q^g \notin U^g$.

**Definition 4.** Point $p^g \in U^g$ is called a conflict point if there is some point $p^g \in U^g$ such that $p^g = p^g$ and $|y^g - z^g(p^g)| \geq |y^g - z^g(p^g)|$.

where $p^g_z$ is the $z$ coordinate of $p^g$. In other words, if two points on separate contours both exist on a vertical line, and the yellow point is closer to green’s $z$-home value, then these points are in conflict. See figure 3.

**Lemma 3.** No point $p^g \in S^g$ can be a conflict point.

**Lemma 4.** Let $p^g \in \partial C^g$ be a conflict point. Then there is no other point $q^g \in \partial C^g$ such that $p^g = q^g$.

**Corollary 1.** Let $p^g \in \partial C^g$ be a conflict point. Then there is exactly one point $q^g \in \partial C^{g,y}$ such that $p^g = q^g$.

Note that this corollary shows that not only are $p^g$ and $p^g$ unique on their own components, but there also cannot be any third point from any other component in the vertical line passing through $p^g$ and $p^g$.

**Theorem 1.** Two components $C^g$ and $C^y$ intersect if and only if there is at least one conflict point on the surface of either component.

**Theorem 2.** Moving any point $p^g \in U^g$ in the direction of $z^g(p^g)$ will not generate any additional conflict points among any pair of components.

**Theorem 3.** Conflict points exist on triangulated surfaces if and only if at least one conflict point exists either at a triangle vertex or along a vertical line passing through an intersection point of the $xy$ projection of triangle edges.
4. IMPLEMENTATION

Our algorithm removes conflict points in a manner conforming to theorem 2, i.e., no additional conflict points are introduced at any step. And since removing all conflict points at locations defined in theorem 3 effectively removes all conflict points, our algorithm is bounded by the number of those locations.

The algorithm removes conflict points between pairs of components. While it is provably true that no three (or more) components can have conflict points along the same vertical line, this fact is not necessary to justify dealing with only two components at a time. Rather, we rely on theorem 2. We can safely remove intersections from two components at a time as that action will not increase the number of conflict points among any pair of components.

In addition to working with only two components at a time, we also deal only with reconstructions between two given tiles at a time. There are no component intersections on the slices themselves (by criteria 3 and 4), and the algorithm does not modify any points on the slices. So the contours on the slices act as natural boundary points between intra-slice reconstructions, even after intersection removal.

The algorithm is as follows:

1. Remove contour intersections in slices $z_i$ and $z_{i+1}$
2. Run single component tiling on each component in slices
3. Determine conflict points
4. Trace cut from conflict point to its exit in a tile
5. Triangulate (planar) polygons
6. Adjust z values

We now describe each step in detail.

Slice contour intersection removal

Our 2D contour intersection removal algorithm is a simple and efficient algorithm aimed at separating contours by a value $\delta$ (see figure 4). All contours in a given slice are first dilated by $\delta/2$ after which proper intersections between contours (proper, in this case, meaning intersections such that a segment from one contour touches both the interior and exterior of another contour) are found using a sweep line and marked. We are guaranteed that there are an even number of intersections as only proper intersections are marked. Intersection points are paired up such that the mid-point of the line segment defined between two intersections is in the interior of two contours. For each of these pairs, the intersecting
intersection point, a pointer to the two tiles and every edge the tiles are considered. These "intersection" points are a tile of another component (still in the xy projection of tiles touch. In addition, all tile vertices that are inside of another tile, or grazing, where only the boundaries of two proper, where a segment lies on the interior and exterior of tiles of different components. These intersections can be once all components between two slices are found, we use aponent reconstructions.

Determine conflict points

The single component reconstruction algorithm we use is conformity to all of the criteria as they apply to single com-
ponents reconstructions.

Figure 5 shows an interesting case in that if the induced edges is greater than two since every tile vertex touches at least two tiles.

Conflict points can occur only at these intersection points. We determine whether a point is conflicting or not by ex-
amining the z values $p_1^y$ and $p_2^y$ where $y$ and $y$ are the intersecting components. We compute $\mathcal{Z}(p_1^y)$ and $\mathcal{Z}(p_2^y)$ and compare their respective distances to $p_1^y$ and $p_2^y$ described in definition 4. Then, using the same data structure that stores the intersections, we mark each intersection as conflicting.

Figure 5(a) shows a yellow tile and three green tiles. The algorithm finds all intersection points (black dots) and then determines which of these are conflict points (red dots).

**Trace tile cuts**

The algorithm for tracing tile cuts is as follows:

```
Algorithm 1 TRACE_CUTS
1: cuts := empty array of polylines
2: for all points $p$ in conflict points do
3: \( p^y := \) projection of $p$ onto yellow component
4: \( t^y := \) yellow tile containing $p^y$
5: \( t^g := \) green tile containing $p^g$
6: polyline := empty array of points
7: push $p$ onto polyline
8: dir := direction to travel on $\partial t^g$ to go to interior of $r^y$
9: boundary := ordered intersections on $\partial t^g$
10: for all intersections $q^g$ in boundary do
11: if $q^g \in \partial t^g \cup \partial t^y$ then
12: \( q^y := \) projection of $q^g$ onto green component
13: \( q := q^g \)
14: push $q$ onto polyline
15: end if
16: end for
17: push polyline onto cuts
18: end for
19: return cuts
```

For each conflict point, we must trace out cut polylines that we will use to induce new polylines that will then be triangulated. The way this is done is to start at a conflict point $p$ and find its projection onto the yellow component to get $p^y$ (line 3). $p^y$ will be on a yellow tile’s boundary. Now follow the points on the boundary of the tile that have been marked as intersections from $p^y$ to the exit point where the yellow tile exits the green tile (lines 8-10). Each point encountered in this trace are added to the polyline (line 14).

This can be seen visually in figure 5(a). Point $p_1$ is a conflict point. The algorithm builds an ordered array of points following the yellow tile from $p_1$ all the way to the point $p_8$. Then it loops over each point in the array checking to see if the current point $p$ is inside of the projection of the green tile $t_1^g$. If it is, then the point is added to the cut. So the cut from point $p_1 = [p_1, p_2, p_3]$. All cuts can be shown in red in figure 5(b).

Cut polylines are specific to a tile. So cuts for the green component’s tiles are first found, and then cuts for the yellow component. These polylines are superimposed onto the tiles to generate a set of induced polylines (figure 5(c)). Even though the cuts are different for green tiles vs. yellow tiles, the projection of green tiles and cuts will be identical to that of the yellow.

Figure 5 shows an interesting case in that if the induced contours are clipped along this line segment. The contours are then eroded back to roughly their original shape minus the clipped areas.

This approach is simple and fast, but it does have its failings. For one, it doesn’t support intersections of more than two contours. This is, of course, possible, but generally doesn’t happen often in the data we have dealt with. Another side effect is a smoothing of the contours, which is dependent on $\delta$. In our case this is actually desirable, as the contours we generally deal with are rather noisy, which is why we have to perform the intersection removal in the first place.

**Single component reconstruction**

The single component reconstruction algorithm we use is found in [2]. It takes planar contours in adjacent slices as input and outputs a series of triangles, or “tiles”, forming a surface between the contours. It supports branching and conforms to all of the criteria as they apply to single component reconstructions.

Determine conflict points

Once all components between two slices are found, we use a sweep line to find all intersections in the xy plane between tiles of different components. These intersections can be proper, where a segment lies on the interior and exterior of another tile, or grazing, where only the boundaries of two tiles touch. In addition, all tile vertices that are inside of a tile of another component (still in the xy projection of the tiles) are considered. These “intersection” points are marked and stored in a data structure that contains the intersection point, a pointer to the two tiles and every edge passing through the point. There will be exactly two edges unless the intersection is grazing, in which case the number of edges is greater than two since every tile vertex touches at least two tiles.
polygons are triangulated naively, an illegal triangulation can result. Consider point $p_1$ in the figure. There will be a triangle vertex at this point due to the triangulation of tile $t_2$. If, then, it is not a vertex in the triangulation of tile $t_3$, then the triangulation will not be legal. Because of this, the algorithm checks for any point on a tile edge that is involved in the triangulation of any adjacent tile, and induces that point onto all adjacent tiles. This ensures legal triangulations.

**Triangulate polygons**

At this point we have polygons that are ready to be triangulated into a new induced surface. An intuitive approach would be to simply run a constrained Delaunay 2D triangulation on the xy projection of all induced points on the tile. This has two problems: first, the tiles can be vertical and thus the projections of the triangles are degenerate and second, inducing points on edges of triangles causes a large number of collinear points.

The first problem can be addressed by checking to see if the tile is vertical before triangulation. If it is, then rotate by 90 degrees and then triangulate. Happily, the tiles are still coplanar and so we can safely rotate the tile with induced points without worry of causing additional degeneracies.

The second problem is more troublesome. The combined problem of collinear points coupled with numerical error causing possible triangle edges outside of the original tile requires something more than a naive approach. Our solution is to maintain a data structure mapping points to the edges of the original tile from which they were induced. Now a numerically-prone check for collinearity is perfectly safe by simply doing a hash lookup for each point and comparing the original edges. If all three edges are the same then the points are collinear. If not, then they are not collinear, provided the original tiling algorithm returns non-degenerate tiles (which it does in our case).

We used a simple ear-cutting algorithm using this data structure to ensure legal triangulations. Even with the check, an additional modification is required to ensure numerical stability. That is to first generate triangles involving at least one unused collinear point. Without this, the result often includes very long thin triangles if at least one induced point is very close to an existing vertex.

**Adjust z values**

The result of triangulating induced polygons is an induced triangulated surface, which still has intersections between components. At this point we can adjust z values of all conflict points in the new triangulation to remove intersections. Each conflict point $p$ is checked and its two associated component points $p^y$ and $p^z$ are given new $z$ values as follows:

$$p^z = \frac{p^x + p^y}{2} + s^z \epsilon$$

where

$$s^z = \begin{cases} -1 & z^y > z^y \\ 1 & z^y < z^y \end{cases} \quad (3)$$

$p^y$ is calculated similarly.

Determining $\epsilon$ such that the desired minimum distance $\delta$ between components is done as follows. A conflict point $p^y$ is either an induced point on an edge or a vertex. Let $B$ be the vector $p^y - q^y$ where $q^y$ is either an induced point or vertex such that $p^y$ is between $q^y$ and $\mathcal{Z}(p^y)$. See figure 6. Further, let $A$ be the vector $p^y - q^y$. Now let $p^y = \{p^x, p^y, p^z + s^y \epsilon\}$ and $\bar{p}^y = \{p^x, p^y, p^z + s^y \epsilon\}$. And lastly, $\bar{B} = \bar{p}^y - q^y$ and $\bar{A} = p^y - q^y$. Distance $d$ is

$$d = \frac{|\bar{A} \times \bar{B}|}{|\bar{B}|} \quad (4)$$

Substituting for $\bar{A}$ and $\bar{B}$ and assuming that $s^y = 1$ we get

$$d = \frac{|(A_x(B_y - \epsilon) - (A_y + \epsilon)B_x, (A_z + \epsilon)B_y - A_z, A_z B_y - A_y B_z)|}{|(B_x, B_y, B_z + \epsilon)|} \quad (5)$$

Factoring $\epsilon$ yields a quadratic:

$$0 = \epsilon^2 ((A_x + B_x)^2 + (A_y + B_y)^2 - d^2) + \epsilon (2)((A_x + B_x)(A_x - A_y) - (A_y + B_y)(A_y - A_x) + (A_y B_x - A_x B_y)^2 - d^2 A_z) + (A_x B_y - A_y B_x)^2 - \epsilon^2 (B_x^2 + B_y^2 + B_z^2) \quad (6)$$

We then substitute $\delta$ in for $d$ and find the two roots for $\epsilon$. It is possible that both roots yield shifts in the direction of $\mathcal{Z}(p^y)$. This occurs when the surface intersections are gross enough to cause conflict points that are more than $\delta$ apart. So we choose the solution for $\epsilon$ with the greatest value and then multiply by $s^y$ (since we assumed that $s^y = 1$).

There is a denominator on the right hand side of (6) which we have omitted for brevity. But it should be clear from (5) that the denominator is zero only when $\bar{A}$ is degenerate which cannot happen since $p^y$ is moved in the direction of $\mathcal{Z}(p^y)$ for $\epsilon > 0$.

**Theorem 4.** $\epsilon < |p^y - \mathcal{Z}(p^y)|$ and $\epsilon < |p^y - \mathcal{Z}(p^y)|$.

This theorem bounds $\epsilon$ by the distance from the conflict points to the slices. In other words, no value of $\epsilon$ can cause a point shift in $z$ such that the point cross a slice boundary. This is important because any such shift would cause the reconstruction to violate criterion 3, not to mention all the criterion’s dependent guarantees.

**Computational complexity**

The computational complexity of our algorithm is bounded by the number of conflict points found. The maximum number of conflict points between the boundaries of two triangles is $12: 6$ for the xy-plane intersections and $6$ for the vertices. Thus, if a reconstruction generates $m$ components, each with $n_t$ triangles, then the maximum number of conflict points is $12nm$, where $n_t$ and $n_v$ are the two largest numbers of triangles in a single component. We state the computational complexity, then, as being $O(n^3)$ where $n$ is the largest number of triangles in a single component.
In practice we have found that there are far fewer conflict points, even in highly tortuous datasets. The intersection removal of figure 1 found \(???)\ conflict points.

**Smoothing**

One additional step in our pipeline is that of smoothing the surfaces. Initial surface reconstruction can introduce numerically-troublesome thin triangles, and the intersection of surfaces. Initial surface reconstruction can introduce numerically-troublesome thin triangles, and the intersection removal of figure 1 found \(???)\ conflict points. These tasks are done using our Level Set Boundary Interior and Exterior Mesher \([8]\) which is embedded in our Volume Rover software package \([9]\).

5. RESULTS AND DISCUSSION

We have implemented a fixed-\(\epsilon\) version of the intersection removal algorithm along with a slightly modified version of the single component reconstruction algorithm found in \([2]\). Figure 7 shows the results of our software on various cases of intersection between tiles (but is by not means exhaustive). Figure 7(a) shows intersection of a vertical tile. This requires the tile to be rotated by 90 degrees before being re-triangulated after new points are induced. 7(b) and (c) show conflict points at only tile vertices and not xy-plane intersections. As can be seen this case is handled since conflict points are checked at tile vertices in addition to xy-plane intersections. 7(d) shows a branching case. The yellow tile is branching from one contour above to two contours below, while the gray tile is branching opposite. The intersection is removed without treating the branching structure as a special case.

We have built up a suite of test cases similar to those shown in figure 7 to help verify correctness of the algorithm. In every case, intersections were removed in a single pass without causing additional intersections or moving any point further than \(\epsilon\) away from any point in another component.

Figure 1 shows results of intersection removal from a reconstruction of the hippocampal region of the brain. In the section of the dataset that is shown a large number of conflict points (~400) were found and removed, preserving extra-cellular spacing in non-intersection regions and completing without causing additional intersections.

This work has made progress on an important topic in 3D reconstruction. One improvement that needs to be made is the removal of criterion 5. This criterion prohibits intersection removal between nested components. In the case of neuronal modeling, so-called intracellular components such as endoplasmic reticulum and mitochondria are treated separately. But to get a truly accurate model, these will need to be included and thus, a 3D spatial model that is topologically correct, including geometries of both neuronal walls and intracellular components, will also be required.

**APPENDIX**

Our proofs rely on an additional theorem from \([2]\), which we restate here in a slightly weaker formulation, but sufficient for our purposes:

**Theorem 5.** \([2]\), Theorem 2) For every point \(p^o\) on the surface \(\partial C^9\) of the green component, \(1 \leq |\mathcal{C}^o(p^o)| \leq 2\).

**Proof of Theorem 1.** Since \(|\mathcal{C}^o(p^o)| \supset |\mathcal{C}^o(p^o)|\), then \(1 \leq |\mathcal{C}^o(p^o)|\) by theorem 5. We prove that \(|\mathcal{C}^o(p^o)| \leq 2\) by contradiction. Suppose \(|\mathcal{C}^o(p^o)| > 2\). Then by the pigeonhole principle, at least 2 contours must exist on some adjacent slice. These contours share the projection \(p_x^o\) onto the slice of point \(p_x^o\). This violates criterion 4.

**Proof of Lemma 2.** If the 2 containing contours exist on the same slice, then they share the projection \(p_x^o\) onto the slice of point \(p_x^o\). This violates criterion 4.

**Proof of Lemma 3.** This is a proof by contradiction. Consider a point \(p^o \in S^9\). Suppose that \(p^o\) is a conflict point. Then there exists some point \(p^s\) such that \(p^o = p^s\) (for the moment, set aside the fact that \(\mathcal{X}(p^o)\) is undefined). By lemma 1 we know that the projection of point \(p^o\) can have no more than 2 containing contours. By virtue of being in \(S^9\), \(p^o\) indeed has 2 containing contours, both of which belong to the green component by definition. Since \(p^o = p^s\), \(p^o\) has the same two containing contours. By theorem 5, \(p^o\) must have at least one containing contour that is yellow, which contradicts lemma 1. Thus no point \(p^o\) exists and \(p^o\) is not a conflict point.

**Proof of Lemma 4.** By lemma 3, \(p^o \in L^1\) and thus cannot be on a vertical tile edge. Then, by criterion 2, a vertical line passing through \(p^o\) passes through no other point.

**Proof of Corollary 1.** By definition of a conflict point, there exists some \(p^o\) such that \(p^o = p^o\). By lemma 4, \(p^o\) and \(p^o\) are the only points lying on the vertical line passing through them. In addition, these two points are unique, as the addition of a third point on the vertical line would violate theorem 5 and lemma 1.

**Proof of Theorem 1.** We first prove that if a conflict point exists, then there is an intersection. At the same time we show that a conflict point is inside the surface or on the boundary of another component. Let \(p^o\) be a conflict point. Let \(\Gamma\) be the vertical path from \(p^o\) to the slice at \(\mathcal{X}(p^o)\). By definition of a conflict point and by lemma 4, \(\Gamma\) will pass through exactly one point \(p^o\) in \(\partial C^9\) before reaching the slice. Since \(C^9\) is guaranteed to be a water tight surface with no holes, \(p^o\) and \(p^o\) are on opposite sides of \(C^9\) - one on the inside and one on the outside. By definition of \(\mathcal{X}(p)\), \(p^o\) is inside a green contour, and thus is inside the green component (note that \(p^o \notin \partial C^9\)). Therefore, \(p^o\) is inside the yellow component. There exists some path \(\Gamma\) from \(p^o\) to some other point \(q^o\) which is on a green contour and therefore outside of the yellow component. Therefore at some point \(\Gamma\) must pass through the yellow boundary, causing an intersection.

We now prove that if there is an intersection, there exists a conflict point. If \(\partial C^9 \cap \partial C^9 \neq \emptyset\) then they share some common point \(p^o = p^o\). This, by definition, is a conflict point.

**Proof of Theorem 2.** Introduction of a conflict point \(q^o\) requires that the left-hand side of relation 1 increase to exceed the right-hand side. Moving \(p^o\) towards \(\mathcal{X}(p^o)\) will only decrease the left-hand side and will thus cause no additional conflict points.

**Proof of Theorem 3.**

**Proof of Theorem 4.** By definition, the projection \(p^o\) of \(p^o\) onto the slice at \(z = \mathcal{X}(p^o)\) lies inside or on the boundary of a green contour. It can lie on a boundary only if
Figure 7: Results of examples showing various interesting cases of intersection. (a) shows a classic intersection except that the back green tile is vertical. (b) shows a case where conflict points occur at a tile vertex rather than intersections between tile edges. (c) is a slightly more complicated case containing conflict points both at tile edges and at vertices. (d) shows intersection removal of a branching topology.

$p^g \in S^g$, so by lemma 3 we know that $p''^g$ lies on the interior of a green contour. Since the contours are separated by $\delta$, and $q^y$ lies on a yellow contour, then $|p''^g - q^y| > \delta$. Thus there is some point $\bar{p}$ on the vertical line between $p^g$ and $p''^g$ such that $\frac{A \times B}{|B|} = \delta$ proving the first statement. The second follows with a similar argument.

A. REFERENCES

[18] J-M. Oliva, M. Perrin, and S. Coquillart. 3d


