The Thin Shell Tetrahedral Mesh

Kenny Erleben[*] and Henrik Dohlmann[†]
Department of Computer Science, University of Copenhagen, Denmark

Abstract

Tetrahedral meshes are often used for simulation of deformable objects. Unlike engineering disciplines graphics is biased towards stable, robust and fast methods, instead of accuracy. In that spirit we present in this paper an approach for building a thin inward shell of the surface of an object. The goal is to device a simple and fast algorithm yet capable of building a topologically sound tetrahedral mesh. The tetrahedral mesh can be used with several different simulation methods, such as the finite element method (FEM).

The main contribution of this paper is a novel tetrahedral mesh generation method, based on surface extrusion and prism tesselation.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: Tetrahedral Mesh, Erosion, Extrusion, Tesselation, Shell, Prism

1 Introduction

Given a 3D polygonal model created by a 3D artist, it is often a challenge to create a spatial structure for simulating a deformable object. Besides, polygonal models for visual pleasing pictures tends to be highly tessellated. Thus, even if they do not pose any "errors", creating a tetrahedral mesh directly from the polygonal model, tends to create an enormous amount of tetrahedra. To achieve real time performance, one seeks a more coarse tetrahedral mesh. These are the kind of problems we aim to solve in this paper.

Given a twofold boundary representation of an object, as a connected triangular mesh (a watertight surface), the tetrahedral mesh is build by extruding each triangle inward, that is in opposite direction of the triangle normal. Thus, for each triangle a prism is generated. The result is a volumetric mesh consisting of connected prisms. These prisms can now be tessellated into tetrahedra, thereby creating the first layer of the thin shell tetrahedral mesh. Succeeding layers can be created by recursively applying this approach. Figure 1 illustrates the basic idea. Although the overall idea is simple, the approach is not without problems. Polygonal models are seldom twofold, but suffers from all kind of degeneracies. The idea we have illustrated is obviously capable of handling an open boundary, but cases where edges share more than two neighboring faces, or self-loop edges, are clearly unwanted, since the generated prisms will overlap each other or degenerate into a zero-volume prism.

The prism generation is reminiscent of an erosion operation with a spherical strutural element (mathematical morphology). The radius of the sphere corresponds to the extrusion length. It is well known that working directly on the Brep [Sethian 1999] is fast and simple, but topological problems arises easily, such as swallow tails.

In case the given triangle mesh is not a proper mesh, one can apply a mesh reconstruction algorithm [Nooruddin and Turk 2003].

[*]kenny@diku.dk
[†]henrikd@diku.dk

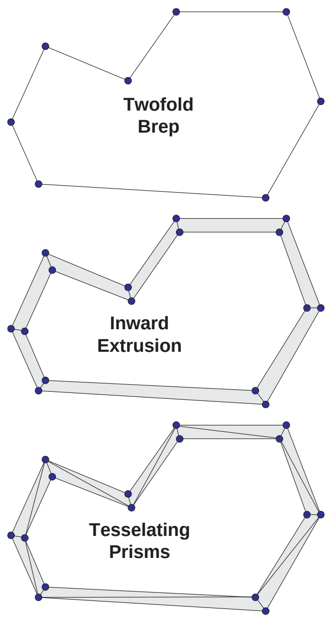

Figure 1: The basic algorithm for generating a thin tetrahedral shell from a boundary representation.

We will require the following properties of the prisms making up a thin shell layer

- No two prisms must be intersecting each other (Neighbors are allowed to touch each other at their common faces).
- No prism must collapse to zero volume or be turned inside out (equivalent to signed volume is always positive).
- All prisms must be convex

Unfortunately, even if we are given a perfect connected twofold triangle mesh, we can get into trouble if we make the inward extrusion too big. This is illustrated in Figure 2. Here, the large extrusion length causes prisms *B* and *C* to become non-convex. Furthermore, *A* and *D*, *B* and *D*, *C* and *A*, and *B* and *C* are overlapping. Fortunately, these degenerate and unwanted prisms can be avoided if the extrusion were made smaller. Thus, we seek an upper bound on how far we can extrude the triangle faces inward, without causing degenerate prisms.

A publicly available implementation of the described algorithm can be found in [OpenTissue 2004].

Existing tetrahedral mesh generation methods create an initial, blockified tetrahedral mesh from a voxelization or signed distance map. Afterwards, nodes are iteratively repositioned, while subsampling tetrahedra to improve mesh quality [Müller and Teschner 2003; Persson and Strang 2004; Molino et al. 2004]. Our approach differs mainly from these in being surface-based.

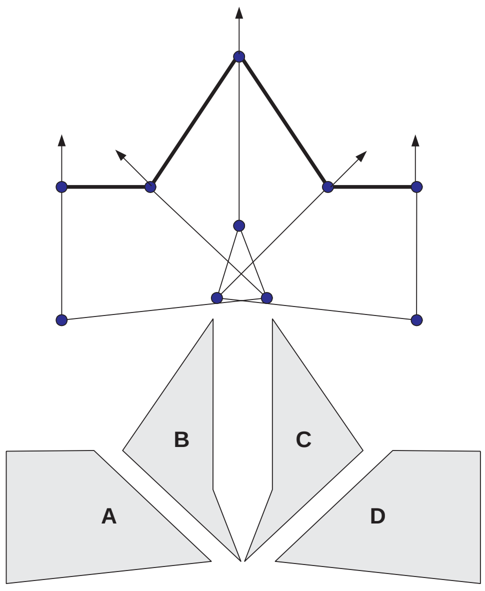

Figure 2: Degenerate prisms results from a too big inward extrusion.

2 Inward Extrusion

As a preprocessing step, we compute the pseudo normals for all vertices (the angle-weighted normals [Aanæs and Bærentzen 2003]). These will indicate the direction along which a vertex will be extruded.

Given a triangle consisting of three vertices $\vec{p}_1$, $\vec{p}_2$ and $\vec{p}_3$, with angle weighted normals $\vec{n}_1$, $\vec{n}_2$ and $\vec{n}_3$, the inward extruded prism is defined by the six corner points:

$$\begin{aligned}
&\vec{p}_1\\
&\vec{p}_2\\
&\vec{p}_3\\
\vec{q}_1(\varepsilon) &= \vec{p}_1 - \vec{n}_1\varepsilon\\
\vec{q}_2(\varepsilon) &= \vec{p}_2 - \vec{n}_2\varepsilon\\
\vec{q}_3(\varepsilon) &= \vec{p}_3 - \vec{n}_3\varepsilon.
\end{aligned}$$

The extrusion length is given by $\varepsilon > 0$. Notation is illustrated in Figure 3. By requiring ε to be strictly positive, all generated prisms must have non-zero volume, as long as prism is not turned inside out. We therefore seek a robust way to determine an upper bound on ε, such that all generated prisms will be valid.

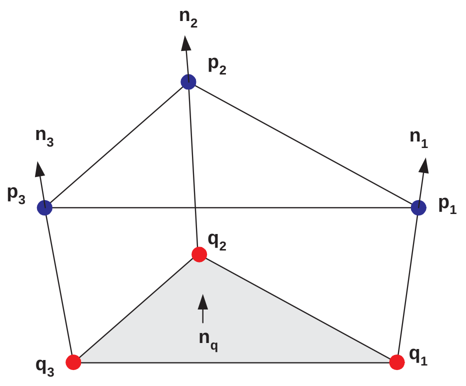

Figure 3: The six corner points defining a prism, and pseudo normals yielding extrusion directions.

The direction of the normal of the extruded face, $\vec{n}_q$, can be found from $\vec{q}_1$,$\vec{q}_2$, and $\vec{q}_3$, using the cross-product:

$$\vec{n}_q(\varepsilon) = (\vec{q}_2(\varepsilon) - \vec{q}_1(\varepsilon)) \times (\vec{q}_3(\varepsilon) - \vec{q}_1(\varepsilon)).$$

This is a second order polynomial in ε,

$$\begin{aligned}
\vec{n}_q(\varepsilon) &= (\vec{q}_2(\varepsilon) - \vec{q}_1(\varepsilon)) \times (\vec{q}_3(\varepsilon) - \vec{q}_1(\varepsilon))\\
&= ((\vec{p}_2 - \vec{n}_2\varepsilon) - (\vec{p}_1 - \vec{n}_1\varepsilon)) \times ((\vec{p}_3 - \vec{n}_3\varepsilon) - (\vec{p}_1 - \vec{n}_1\varepsilon))\\
&= ((\vec{p}_2 - \vec{p}_1) + (\vec{n}_1 - \vec{n}_2)\varepsilon) \times ((\vec{p}_3 - \vec{p}_1) + (\vec{n}_1 - \vec{n}_3)\varepsilon)\\
&= \underbrace{((\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1))}_{\vec{c}} +\\
&\quad \underbrace{((\vec{p}_2 - \vec{p}_1) \times (\vec{n}_1 - \vec{n}_3) + (\vec{n}_1 - \vec{n}_2) \times (\vec{p}_3 - \vec{p}_1))}_{\vec{b}}\varepsilon +\\
&\quad \underbrace{((\vec{n}_1 - \vec{n}_2) \times (\vec{n}_1 - \vec{n}_3))}_{\vec{a}}\varepsilon^2\\
&= \vec{a}\varepsilon^2 + \vec{b}\varepsilon + \vec{c}.
\end{aligned}$$

Observe that $\vec{c} \neq \vec{0}$, since its magnitude is equal to twice the area of the triangle being extruded.

To ensure convexity, the dot product of the direction of the normal of the extruded face, $\vec{n}_q$, with the pseudo normals, $\vec{n}_1$, $\vec{n}_2$, and $\vec{n}_3$, must always be positive. That is

$$\begin{aligned}
\vec{n}_1 \cdot \vec{n}_q(\varepsilon) &> 0\\
\vec{n}_2 \cdot \vec{n}_q(\varepsilon) &> 0\\
\vec{n}_3 \cdot \vec{n}_q(\varepsilon) &> 0.
\end{aligned}$$

This yields the following system of constraints,

$$\begin{bmatrix} \vec{n}_1 \cdot \vec{a} & \vec{n}_1 \cdot \vec{b} & \vec{n}_1 \cdot \vec{c} \\ \vec{n}_2 \cdot \vec{a} & \vec{n}_2 \cdot \vec{b} & \vec{n}_2 \cdot \vec{c} \\ \vec{n}_3 \cdot \vec{a} & \vec{n}_3 \cdot \vec{b} & \vec{n}_3 \cdot \vec{c} \end{bmatrix} \begin{bmatrix} \varepsilon^2 \\ \varepsilon \\ 1 \end{bmatrix} > 0.$$

We solve for the smallest positive ε fulfilling the system of constraints. That is, each row represents the coefficient of a second order polynomial in ε, thus for each row we find the two roots of the corresponding polynomial. The three rows yields a total of 6 roots. If no positive root exist, then $\varepsilon = \infty$, otherwise ε is set equal to the smallest positive root.

Observe that the third column of the coefficient matrix is always positive (by the property of the angle weighted normals). The first column can be interpreted as an indication of, whether the corresponding extrusion line is trying to “shrink” (< 0) or “enlarge” (> 0) the extruded face. The middle column is difficult to interpretate. As far as we can tell it resembles an indication of the skewness of the resulting prism.

In fact, the tree convexity constraints ensure that no neighboring prism will intersect each other, nor will the prism turn its inside out (ie. flipping the extruded face opposite the original face).

The maximum extrusion length for the entire layer can be found by iterating over each prism. For the i'th prism the extrusion length ε^i is computed. The maximum extrusion length of the layer is found as

$$\varepsilon = \min\left(\varepsilon^0, \ldots, \varepsilon^{n-1}\right).$$

Afterwards, it is a simple matter to compute the actual extrusion and generating the prisms.

3 Prism Generation

The technique in the previous section guarantee that prisms will be convex, and that no intersections occurs between neighboring prisms. However, degenerate prisms can still occur whenever the extruded face collapses to a line or to a point. These are shown in Figure 4. These degenerate cases must be marked, such that the following tessellation can take them into consideration. The problem is how to detect these degenerate cases. If no upper bound were found on the extrusion length for a prism, we can trivially reject the prism. However, if an upper extrusion bound were computed, the prism might degenerate. As a first step in marking the degenerate prisms, we iterate over all those prisms where an upper bound were computed. If the computed upper bound for a prism is equal to the extrusion length, then it might be a degenerate prism. For each of these possible degenerate prisms we first test whether

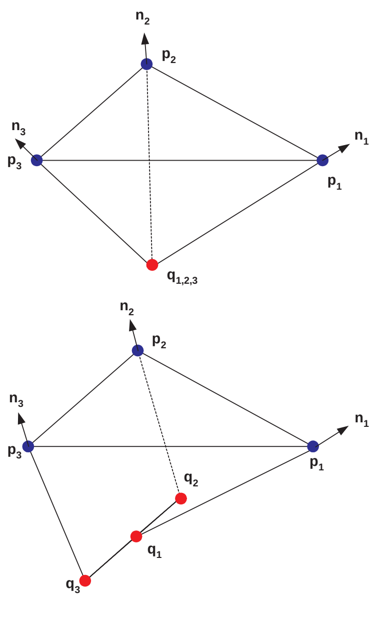

Figure 4: Point-degenerate and line-degenerate prisms.

$$||\vec{n}_q(\varepsilon)|| \leq \gamma,$$

where γ is a user specified threshold to counter numerical precision problems. If this criteria is fulfilled, we clearly know that we are dealing with a degenerate prism.

The degenerate prisms can be classified as either point-degenerate or line-degenerate, as shown in Figure 4. The line-degenerate case is given by the criteria

$$(\vec{q}_2(\varepsilon) - \vec{q}_1(\varepsilon)) \neq \vec{0} \quad \text{or} \quad (\vec{q}_3(\varepsilon) - \vec{q}_1(\varepsilon)) \neq \vec{0}.$$

If this criteria is not fulfilled we have a point-degenerate case. A pseudo code version of the algorithm is shown in Figure 5. The degenerate cases do not only influence the prism tessellation (which we will treat in the next section). If another thin shell layer is to be generated, then the original mesh faces can no longer be used. A simple 2D case is shown in Figure 6. Notice that the bold red faces were used when creating layer 0, but they vanish when layer 1 is created. Therefore, if another layer should be generated, a new connected triangular mesh must be formed from the extruded faces of the non-degenerate prisms.

In Figure 7 the surface mesh generation algorithm is shown in pseudo code.

```
algorithm markDegeneratePrism($\varepsilon,\gamma$)
  for each prism $p$ do
    if $\varepsilon^p = \varepsilon$ then
      if $||\vec{n}_q(\varepsilon)|| \leq \gamma$ then
         if $(\vec{q}_2(\varepsilon) - \vec{q}_1(\varepsilon)) \neq \vec{0}$
               or $(\vec{q}_3(\varepsilon) - \vec{q}_1(\varepsilon)) \neq \vec{0}$ then
        mark $p$ as line-degenerate
      else
        mark $p$ as point-degenerate
      end if
    end if
  next $p$
End algorithm
```

Figure 5: Pseudo code for marking degenerate prisms.

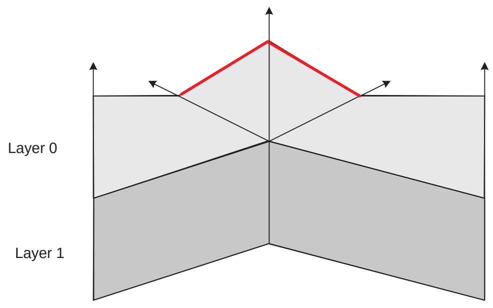

Figure 6: Degenerate cases affect succeeding layers.

4 Prism Tessellation

For non-degenerate prisms, having 6 corners, the minimum number of tetrahedra we can tesselate the prism into, is 3. This is shown in Figure 8. The problem with this approach is that the extruded sides of the prism will be triangulated. One therefore has to ensure, that the tessellation of neighboring prisms agree with each other. This is illustrated in Figure 9. As seen in Figure 9 it becomes a global combinatorial problem to match the tesselation of neighbors against each other.

If we use the centroid of the prism as the apex for creating tetrahedra then a prism can be tesselated into eight tetrahedra, as shown in Figure 10. With this approach the tesselation is no longer a global problem, since the tesselation of each prism side can be chosen in-

```
algorithm build-surface(Mesh : $M$)
  for each prism $p$ do
    for $i = 1,2,3$ do
      if $\vec{q}_i \notin M$ then
        add $\vec{q}_i$ to $M$
      end if
    next $i$
    if $p$ not degenerate then
      add face $\vec{q}_1, \vec{q}_2, \vec{q}_3$ to $M$
    end if
  next $p$
End algorithm
```

Figure 7: Pseudo code for generating surface mesh for next shell layer.

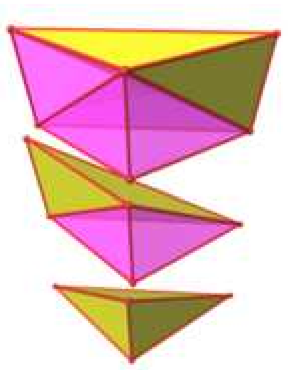
Figure 8: Prism tessellated into 3 tetrahedra.

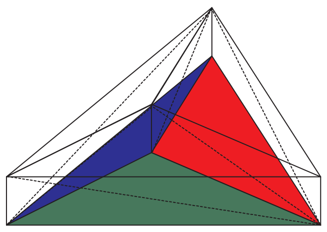
Figure 9: Tessellation of neighboring prisms must be consistent.

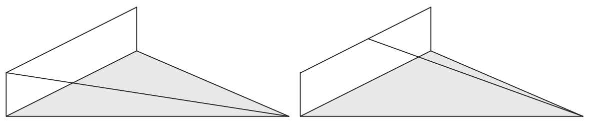
Figure 11: Nice vs. bad line case.

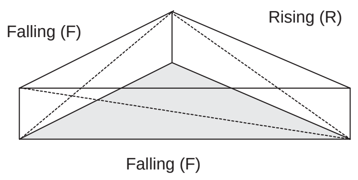

Figure 12: Classification of prism sides as falling (F) or rising (R).

dependently of each other.

Both approaches deals nicely with the point-degenerate case. Since a point-degenerate prism is already a tetrahedron, there is no need to tesselate it. However, since the point-degenerate case only have triangular sides, it can never be a direct neighbor with a non-degenerate prism. It can only be neighbor with other point-degenerate cases or line-degenerate cases, which have both rectangular and triangular sides, as shown in Figure 11.

In the following we disregard degeneracies and consider the three-tetrahdra tesselation strategy. This is because it is more attractive due to the lower tetrahedra count.

A prism can be tesselated into three tetrahedra in 6 different ways. In order to classify the 6 types of tesselation, we will mark the rectangular sides of a prism as falling (F) or rising (R). The edge type depends on wether the tesselation edge is falling or rising as we travel along the extruded prism face in counter clock wise order. See Figure 12. We observe that the three-tetrahedra tesselation strategy will always have two prism sides of the same type, and the last side of opposite type. Thus, we can only have 6 different patterns, as shown in Table 1. The consistency requirement implies that if one side of a prism is marked as F, then the neighboring prism will have marked the same side as R. In short, no neighboring prisms will have a side marked with the same type.

A simple tesselation example is shown in Figure 13. Here a tetrahedron mesh is being tesselated. The thetrahedron has been cut up and layed out in 2D. Triangle edges corresponds to rectangular sides of prisms. Let us apply a brute-force strategy to the combinatorial problem of the tesselation as follows: We start at a single prism and choose one of the 6 tesselation types. Then we visit the neighboring prisms and choose a tesselation type that agree with the immediate neighbor prisms, which have already been tesselated. This is a breadth first traversal over the prisms.

The method is not fail-safe, since inconsistency can arise, as shown in Figure 14. Here, the middle prism is the last prism to be visited by the traversal. Clearly, it is impossible to assign a tesselation type to the prism, since all three sides should have the same type. We can repair the inconsistency by picking one of the neighboring prisms and flipping the type of its shared edge. This action will not change the type of any of the edges marked with arrows in Figure 14. Therefore, the repairing action do not cause a rippling effect through the prisms, and inconsistencies are not introduced at other places.

Fixing inconsistency in this way is attractive, since it offers a local solution to a global problem. However, sometimes we might end up in a dead-lock where no local solution can be found, as is shown in the top of Figure 15.

This time the drawing in the figure resembles a small local view of a larger mesh. Notice that none of the edges shared with the inconsistent prism can be flipped, without creating an inconsistent neighboring prism. The problem is that all the edges marked with arrows are of the same type.

The solution to the problem is shown in the bottom of Figure 15. We let the inconsistency ripple as water waves over to neighboring prisms, in a search for a single prism, where an edge flip does not give rise to a new inconsistency. When such a prism is encountered, we track the trajectory of the ripple wave-front back to the originating inconsistent prism, and flip all shared edges lying on this path. In Figure 16 we have shown the result of the rippling. Notice that two edges are flipped. These are the edges lying on the path to the prism that could be flipped. Also notice that all edges marked with arrows are unaffected by the rippling action. This property ensures that the rippling action will not cause inconsistencies in any prisms elsewhere in the mesh.

A pseudo code of the tesselation-pattern-finding algorithm is shown in Figure 17. Our proposed tesselation pattern algorithm

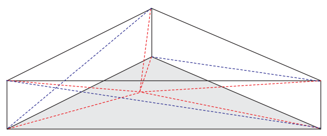
Figure 10: Prism tessellated into eight tetrahedra.

F	R	R
R	F	R
R	R	F
R	F	F
F	R	F
F	F	R

Table 1: The 6 Four-Tetrahedra Tesselation types.

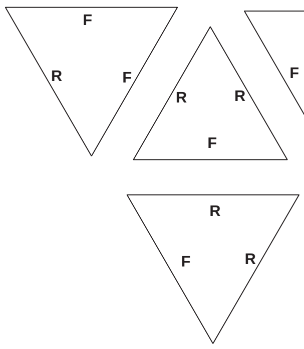

Figure 13: Tesselation Example. A simple 3D mesh (a tetrahedron) have been cut up and layed down in 2D. Triangles correspond to prisms in the thin shell.

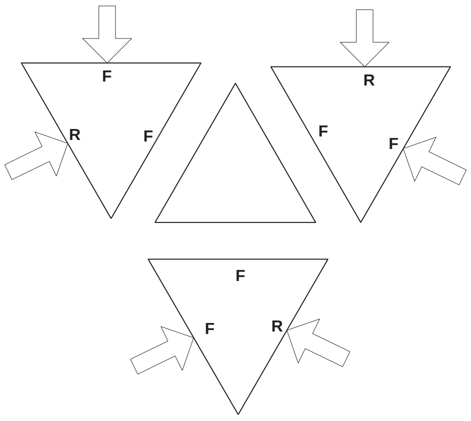

Figure 14: Inconsistent Tesselation Example. The middle prism will have the same type on all sides, which is illegal.

is an ad-hoc solution for the problem at hand. We do not have a formal proof, stating that it is always possible to find a consistent pattern of rising and falling tesselation edges.

5 Results

We have implemented the extrusion length computation and tesselation-pattern algorithm. Currently the implementation detects degenerate prisms, but do not tesselate these.

In our test cases we have chosen 14 meshes of increasing size, that were all scaled to be within the unit-cube. A single layer shell were computed, given a user specified maximum extrusion length. In Table 2, performance statistics are listed, together with polygon count, extrusion lengths and rippling action information. The timings for a single layer construction are cheap, and appears to scale lineary with mesh size. The resulting tetrahedral meshes are visualized in Figure 18. As seen in Table 2, the test-cases: diku, teapot, propeller, funnel, cow, and bend have a surpringsingly small extrusion limit. The remaining test cases show excellent extrusion limits. Figure 19 shows the prisms corresponding to the minimum extrusion limit. Notice that in all cases, where the limit is unexpected small, the limit is caused by small faces or long slivers on sharp ridges. Figure 20 shows cut-through views of the cylinder, pointy, tube, sphere, teapot, funnel, bowl, and torus meshes. Notice how thin the teapot and funnel are.

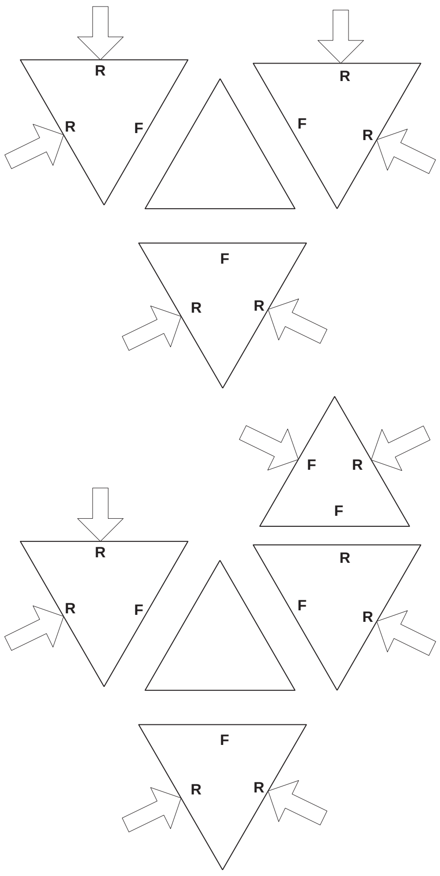

Figure 15: Top picture shows a dead-locked inconsistent tesselation. The bottom picture shows that inconsistency problem have been propagated to neighboring prisms further away.

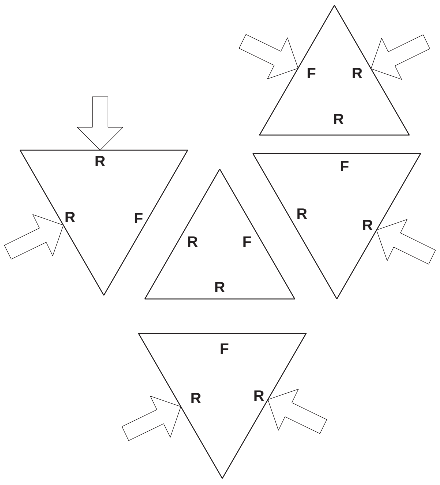

Figure 16: The rippling solution to the dead-locked case shown in Figure 15.

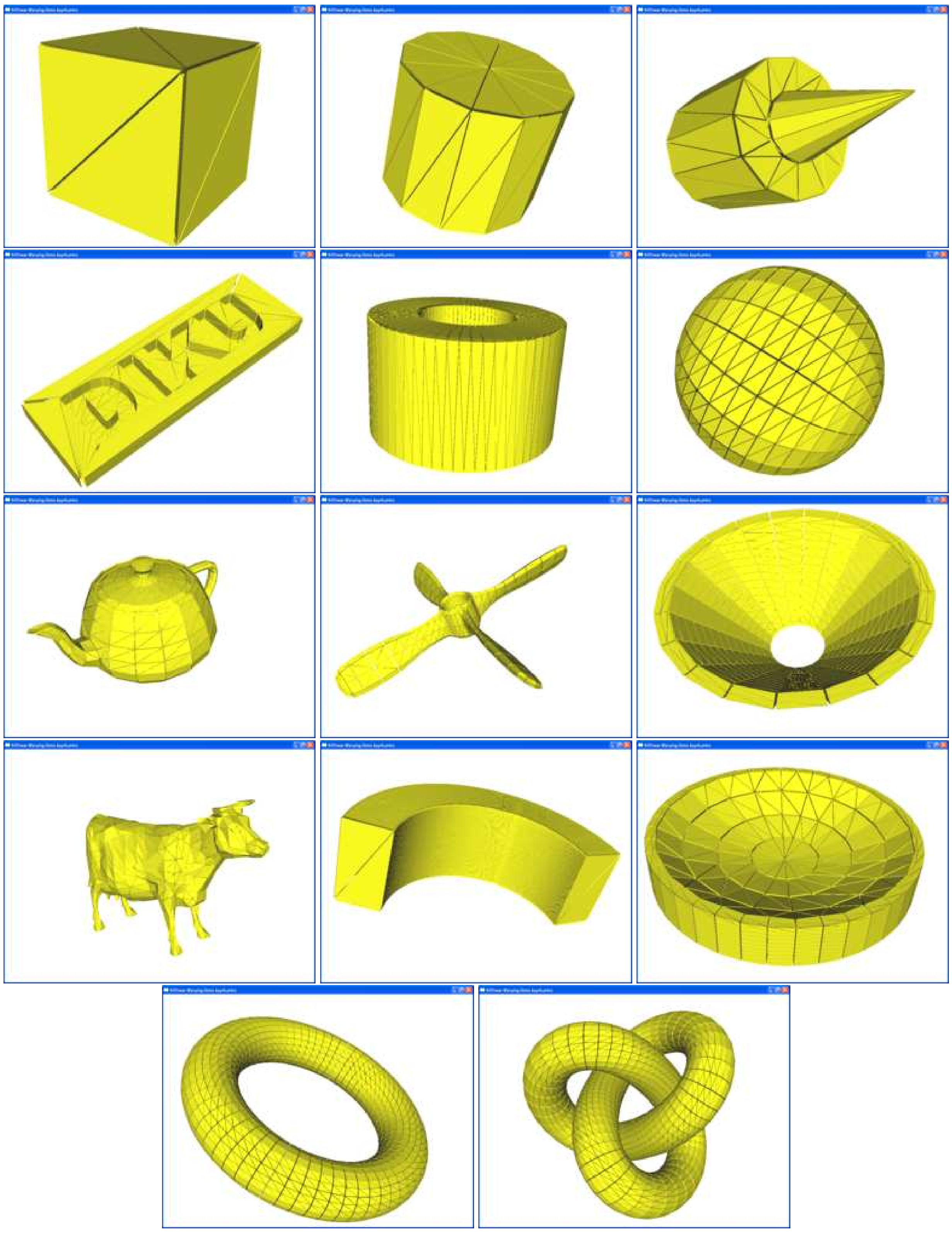

Figure 18: Tesselation results of the 14 meshes.

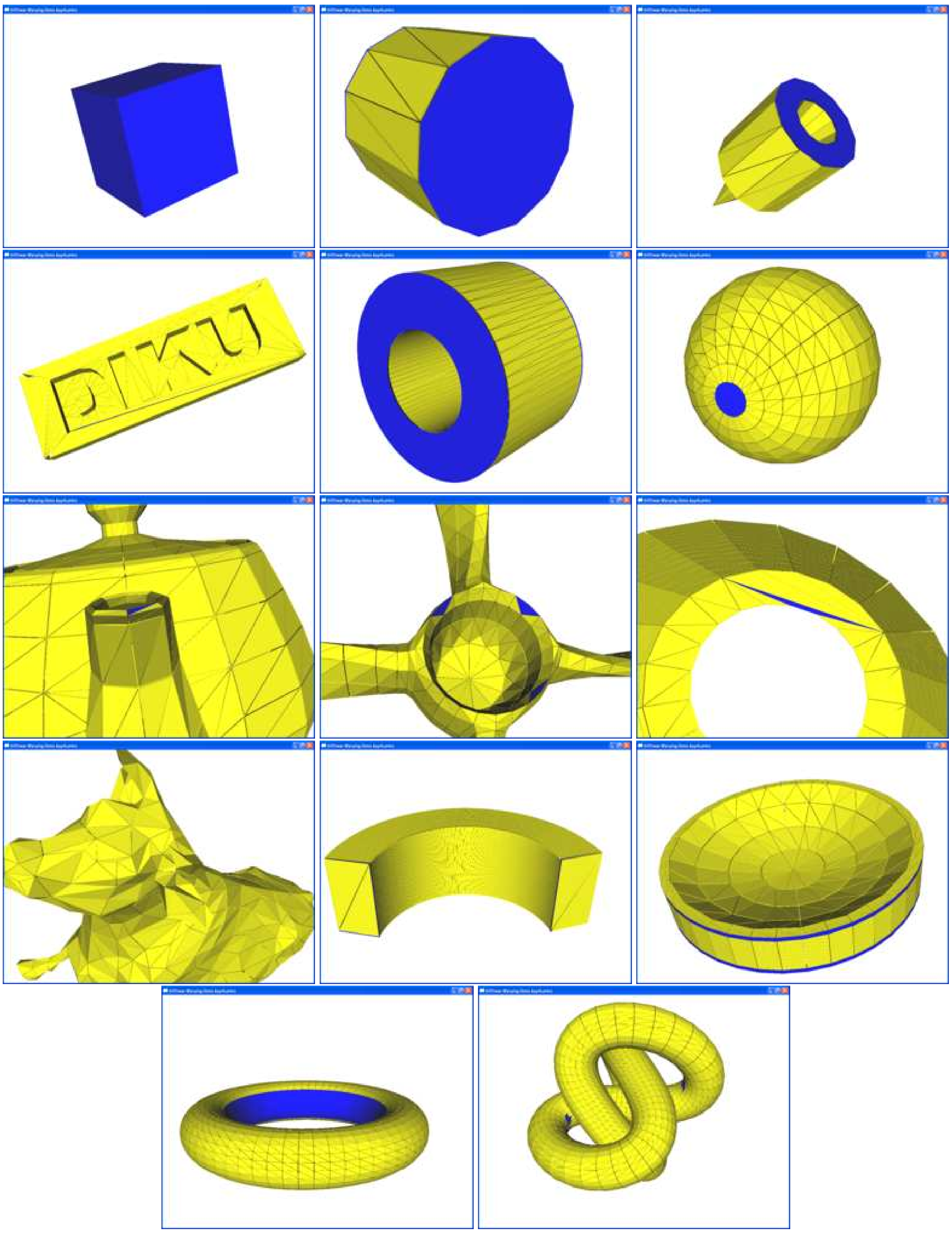

Figure 19: Prisms marked with blue have minimum extrusion limit.

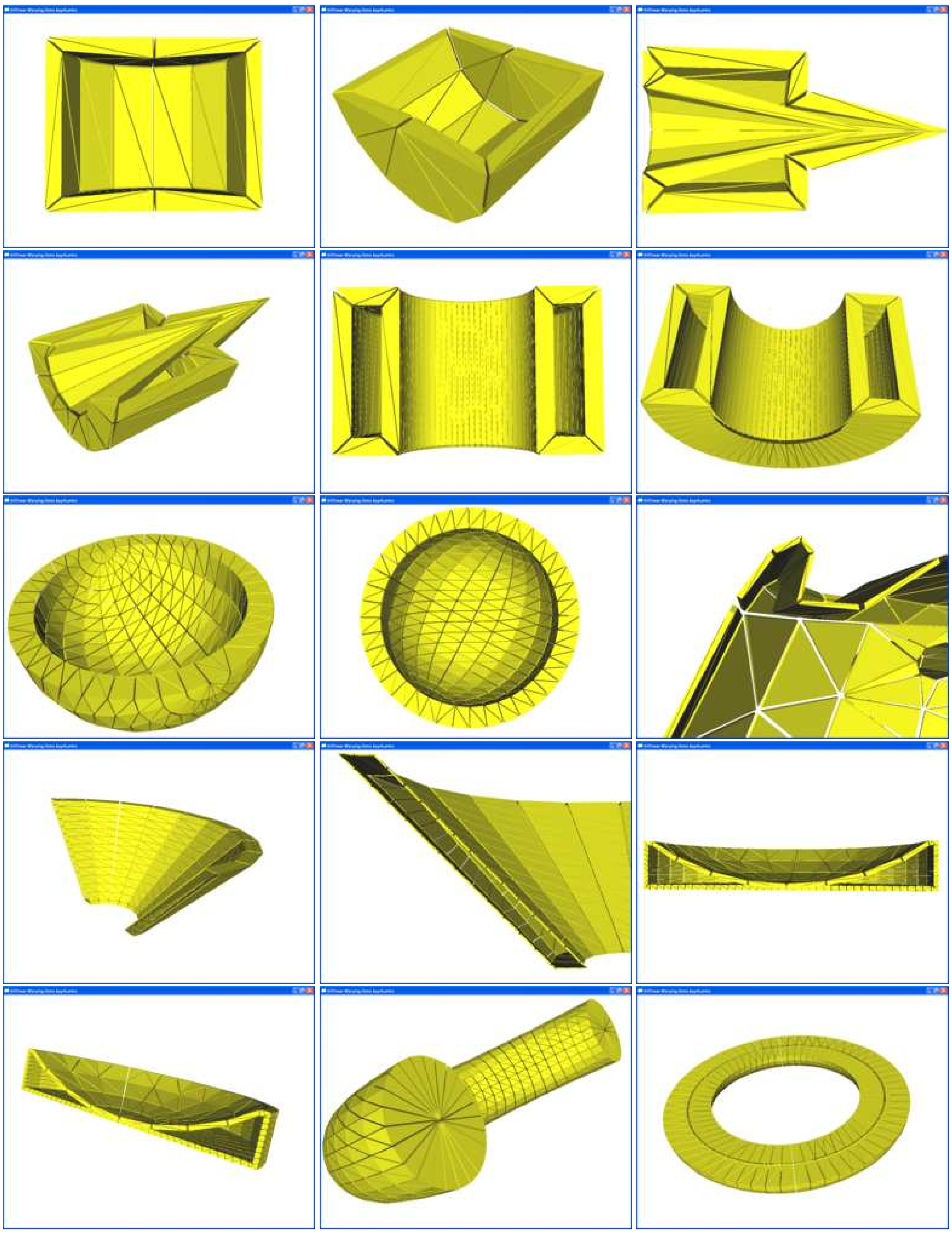

Figure 20: Various cut-through views of a few selected meshes, illustrating the thin shell.

```
algorithm tesselation-pattern()
  Queue $Q$
  Push first prism onto $Q$
  While $Q$ not empty do
    Prism $p = \text{pop}(Q)$
    mark $p$ as visited
    if neighbors is not tesselated then
      pick random pattern of p
    else if exist consistent pattern  with neighbors
      assign consistent pattern to p
    else
      if exist neighbor that can be flipped
        flip edge type of shared edge with p
        assign constent pattern to p
      else
        perform-rippling
      end if
    end if
    for all unvisited neighbors $n$ of $p$ do
      $\text{push}(Q,n)$
    next $n$
  End while
End algorithm
```

Figure 17: Pseudo code for determining tesselation pattern.

6 Discussion

We have omitted the problem of shell layers overlapping from opposite sides. In Figure 20 the problem is seen in the case of the bowl mesh. Our solution to the problem have been to ignore it. The user can always choose a smaller maximum extrusion length.

Degenerate prisms were ignored, in the sense that we are able to detect if they occur, but our tesselation pattern algorithm is not yet capable of handling them. No degenerate prisms were generated for any of the examples in our result section.

The global computation of the extrustion length work fairly well for some meshes, but for others a surpingsly small extrusion length is found. Long slivers and small faces lying close to sharp ridges are the reason for this phenemena, as can be seen in Figure 19. Thus, we conclude that, not surpringsingly, the algorithm is highly dependent on both the shape of the object, but also upon the tesselation of the object surface. It appears that a good uniform tesselation work best. Abrubt tesselation with large aspect ratios results in small extrusion lengths. Mesh reconstruction [Nooruddin and Turk 2003] could be used as a preprocessing step to create a more suitable tesselation.

Another avenue for circumventing the problem of a small global extrusion, might be to investigate the possibililty of having non-global extrusion lengths, i.e. a varying extrusion length over the mesh, adapting itself to take the local maximum length without causing degenerate prisms. We believe this is an interesting thought an leave it for future work.

Our results indicate that our tesselation pattern algorithm works: We have not yet encountered an unsolvable problem. We believe this shows, that the combinatorial problem of finding the tesselation pattern, is at least solvable in practice. From a theoretical viewpoint, a proof of existence would be very interesting, and we leave this for future work.

7 Conclusion

In this paper we have presented preliminary results, showing that it is possible to generate a thin shell, without any topological errors.

	$\|F\|$	$\|R\|$	δ	ε	time(secs.)
box	12	0	0.1	0.866	0
cylinder	48	0	0.1	0.431	0
pointy	96	0	0.083	0.083	0
diku	288	0	0.004	0.004	0
tube	512	0	0.1	0.174	0.01
sphere	760	0	0.1	0.476	0.01
teapot	1056	1	0.001	0.001	0.01
propeller	1200	2	0.004	0.004	0.02
funnel	1280	1	0.003	0.003	0.029
cow	1500	0	0.001	0.001	0.02
bend	1604	0	0.006	0.006	0.029
bowl	2680	0	0.017	0.017	0.04
torus	3072	0	0.099	0.099	0.059
knot	5760	0	0.1	0.102	0.089

Table 2: Performance Statistics on 14 different test cases. In all cases the end user requested a shell thickness of 0.1. The δ-column shows the actual shell thickness produced. The ε-column shows the extrusion limit. The $|F|$-column gives the face count of the meshes. The $|R|$-column gives the number of times the ripple action were invoked. The zero entires in the time column indicates that the duration were not measureable by the timing method.

As pointed out in the previous section, there are many unsolved issues to be dealt with.

Our motivation for this work were to create a volumetric mesh with low tetrahedra count for animation purpose. Due to the early stage of this work, we have not yet validated whether our approach is useful for animation.

References

AANÆS, H., AND BÆRENTZEN, J. A. 2003. Pseudo–normals for signed distance computation. In *Proceedings of VISION, MODELING, AND VISUALIZATION*.

MOLINO, N., BRIDSON, R., TERAN, J., AND FEDKIW, R. 2004. Adaptive physics based tetrahedral mesh generation using level sets. (in review).

MÜLLER, M., AND TESCHNER, M. 2003. Volumetric meshes for real-time medical simulations. In *Proc. BVM (Bildverarbeitung für die Medizin)*, 279–283.

NOORUDDIN, F. S., AND TURK, G. 2003. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics 9*, 2, 191–205.

OPENTISSUE, 2004. http://www.opentissue.org.

PERSSON, P.-O., AND STRANG, G. 2004. A simple mesh generator in matlab. *SIAM Review 46*, 2 (June), 329–345.

SETHIAN, J. A. 1999. *Level Set Methods and Fast Marching Methods. Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press. Cambridge Monograph on Applied and Computational Mathematics.