The Adaptive Thin Shell Tetrahedral Mesh

Kenny Erleben
Dept. of Computer Science,
University of Copenhagen,
Denmark
kenny@diku.dk

Henrik Dohlmann
3DLab, School of Dentistry,
University of Copenhagen,
Denmark
henrikd@lab3d.odont.ku.dk

Jon Sporring
Dept. of Computer Science,
University of Copenhagen,
Denmark
sporring@diku.dk

ABSTRACT

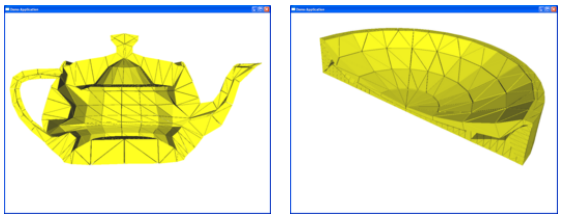
Tetrahedral meshes are often used for simulating deformable objects. Unlike engineering disciplines that often focuses on accuracy, computer graphics is biased towards stable, robust, and fast methods. In that spirit we present an approach for building an adaptive inward shell of the surface of an object. The goal is to device a simple and fast algorithm capable of building a topologically consistent tetrahedral mesh. The tetrahedral mesh can be used with several different simulation method, such as the finite element method (FEM), and the main contribution of this paper is a novel tetrahedral mesh generation method based on adaptive surface extrusion.

Keywords
Tetrahedral Mesh, Erosion, Extrusion, Tessellation, 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. Creating a tetrahedral mesh often results in an enormous number of tetrahedra, hence a more coarse tetrahedral mesh is often sought in order to achieve real-time performance. In this paper a mesh generation method is proposed, that works directly on the surface of a mesh, avoids a constrained Delaunay triangulation, is easy to implement, and yields a tetrahedral count, which is linearly proportional with the number of mesh faces.

Given a watertight twofold boundary representation of an object such as a connected triangular mesh, a prism is generated for each triangle by extruding the triangle inward. The result is a volumetric mesh consisting of connected prisms. These prisms can easily be tessellated into tetrahedra to create the first layer of the thin shell tetrahedral mesh. Succeeding layers can be created by recursively applying this approach. Figure 1 shows examples of thin shells from volumetric meshes.



(a) Teapot (b) Bowl

Figure 1: Cut-views showing the shell layers inside the volumetric meshes generated with our method.

Polygonal models are seldom twofold, but often suffers from several kinds 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 where edges self-loop, will generate prisms, which overlap or degenerate into a zero-volume prisms. In such cases a mesh reconstruction algorithm [NT03] must be applied first.

The suggested prism generation is reminiscent of an erosion operation with a spherical structural element on the polygonal model. The radius of the sphere corresponds to the extrusion length. It is well known that working directly on the boundary representation [Set99] is fast and simple, but topological problems arises easily such as shocks [KTZ95]. The counter-part to shocks are degenerated prisms, that is prisms with less than 6 vertices. These shocks turn out to be the limit on the extrusions lengths.

The Journal of WSCG, Vol.13, ISSN 1213-6964
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

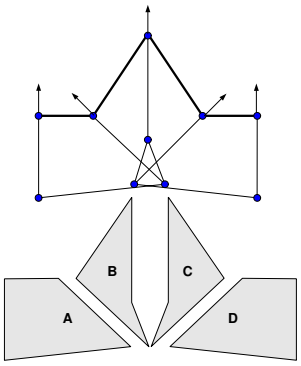


Figure 2: Degenerate prisms results from a too big inward extrusion. This is an example of a swallow tail.

Existing tetrahedral mesh generation methods in the literature typically create an initial, blockified tetrahedral mesh from a voxelization or signed distance map. Afterwards, nodes are iteratively repositioned, while tetrahedra are sub-sampled in-order to improve mesh quality [MT03, PS04, MBTF04]. In contrast to these methods, our is surface-based. An implementation of our method presented is available at [Ope04].

2 MINIMAX INWARD EXTRUSION

The thin shell layer is produced by extruding each triangle in the mesh inwardly, thus producing prisms. We will require the following three properties of the prisms: No two prisms must intersect each other, prisms must have volume larger than zero, and all prisms must be convex. Unfortunately, even for a perfectly connected twofold triangle mesh, too large inward extrusions will cause problems as illustrated in Figure 2. In the figure, 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 by reducing the extrusions. Thus, we must seek an upper bound on how far, we can extrude the triangle faces inwardly without causing degenerate prisms.
To make the inward extrusion, we first compute the outward, angle-weighted normals [AB03] for all vertices. Then for 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 points: $\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$, and

$$\vec{q}_1(\varepsilon) = \vec{p}_1 - \vec{n}_1\varepsilon, \quad (1)$$
$$\vec{q}_2(\varepsilon) = \vec{p}_2 - \vec{n}_2\varepsilon, \quad (2)$$
$$\vec{q}_3(\varepsilon) = \vec{p}_3 - \vec{n}_3\varepsilon, \quad (3)$$

where $\varepsilon > 0$ is the extrusion length. The notation is illustrated in Figure 3. Clearly ε must be strictly positive, however to further guarantee convexity and positive volume of the prisms, we will use the normal of the extruded faces to generate an upper bound on ε.

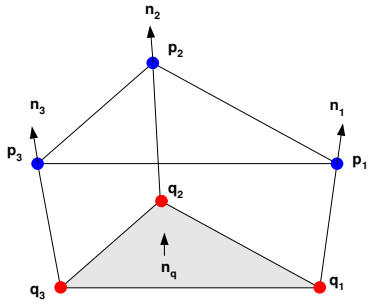


Figure 3: The six points and pseudo normals defining the prism and extrusion direction.

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:

$$\vec{n}_q(\varepsilon) = (\vec{q}_2(\varepsilon) - \vec{q}_1(\varepsilon)) \times (\vec{q}_3(\varepsilon) - \vec{q}_1(\varepsilon)). \quad (4)$$

By the distributive property of the cross product, we find a second order polynomial in ε,

$$\begin{aligned}\vec{n}_q(\varepsilon) = &\underbrace{((\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1))}_{\vec{c}} \\ &+ \underbrace{((\vec{p}_2 - \vec{p}_1) \times (\vec{n}_1 - \vec{n}_3) + (\vec{n}_1 - \vec{n}_2) \times (\vec{p}_3 - \vec{p}_1))}_{\vec{b}}\varepsilon \\ &+ \underbrace{((\vec{n}_1 - \vec{n}_2) \times (\vec{n}_1 - \vec{n}_3))}_{\vec{a}}\varepsilon^2 \\ = &\vec{a}\varepsilon^2 + \vec{b}\varepsilon + \vec{c}. \end{aligned} \quad (5)$$

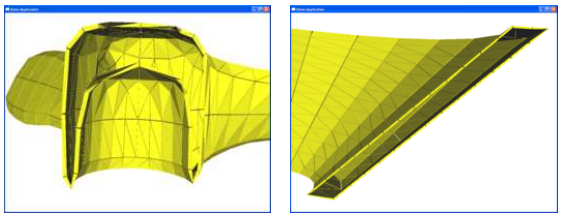
Observe that $\vec{c} \neq \vec{0}$, since its magnitude is equal to twice the area of the triangle being extruded.
To ensure positive volume and convexity, it is sufficient to ensure positivity of the dot product of the normal of the extruded face, $\vec{n}_q$, with the pseudo normals, $\vec{n}_1$, $\vec{n}_2$, and $\vec{n}_3$. I.e. we must have that,

$$\vec{n}_1 \cdot \vec{n}_q(\varepsilon) > 0, \quad (6)$$
$$\vec{n}_2 \cdot \vec{n}_q(\varepsilon) > 0, \quad (7)$$
$$\vec{n}_3 \cdot \vec{n}_q(\varepsilon) > 0. \quad (8)$$

Using (5), we may formulate the constraints as the following system of inequalities,

$$\begin{bmatrix} \vec{n}_1 \cdot \vec{a} & \vec{n}_1 \cdot \vec{b} & \vec{n}_1 \cdot \vec{c} \\ \vec{n}_2 \cdot \vec{a} & \vec{n}_2 \cdot \vec{b} & \vec{n}_2 \cdot \vec{c} \\ \vec{n}_3 \cdot \vec{a} & \vec{n}_3 \cdot \vec{b} & \vec{n}_3 \cdot \vec{c} \end{bmatrix} \begin{bmatrix} \varepsilon^2 \\ \varepsilon \\ 1 \end{bmatrix} > 0. \quad (9)$$

The largest positive ε fulfilling the system of constraints is the upper bound on the inward extrusion lengths. We find the upper bound by solving for each row the roots of the the second order polynomial in ε. The three rows



(a) Propeller (b) Funnel

Figure 4: Close-up cut-views showing how a global extrusion length causes thin shell layers.

yield a total of 6 roots. If no positive root exist, then $\varepsilon = \infty$, otherwise ε must be less than the smallest positive root.

The three convexity constraints ensure that no neighboring prism will intersect each other, nor will the prism turn its inside out, i.e. flipping the extruded face opposite the original face. A global extrusion length for the entire layer can be found by iterating over each prism. The global extrusion length of the layer is found as

$$\varepsilon = \min\left(\varepsilon^0, \ldots, \varepsilon^{n-1}\right), \tag{10}$$

where ε^i is the extrusion length for the i'th prism. Afterwards, it is a simple matter to compute the actual extrusion and generating the prisms. In some cases using the maximum possible extrusion length of a prism can degenerate it. The degenerated prism will have a zero-area extruded face and is easily detected and treated [ED04]. During triangulation degenerate prisms can be dealt with by insertion of an extra internal corner point.

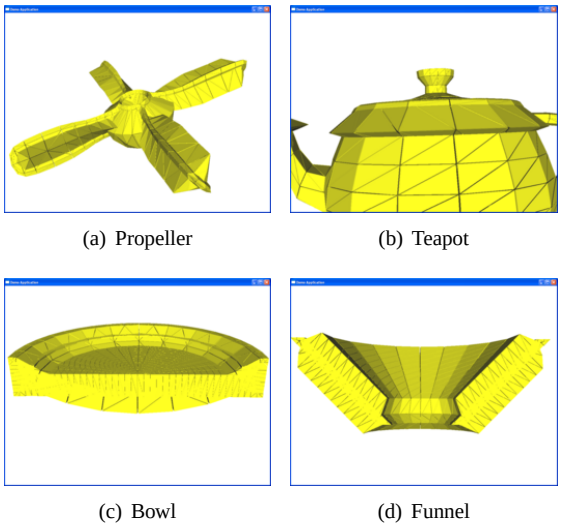
3 ADAPTIVE EXTRUSION

Badly shaped surface triangles can in some cases cause an inefficient small global extrusion length, as illustrated in Figure 4. This is caused by the global optimization in (10), in which a single prism near high curvature will dictate the thickness of the entire layer. Multiple layers will give an impression of a solid or dense object, but will introduce a large number of tetrahedra. The thin shell also causes the tetrahedra to turn into slivers, if the global extrusion length is too small. To overcome such inefficient thin shell layers, we propose to use an adaptive extrusion length.

To calculate the adaptive extrusion length, a surface vertex is assigned an extrusion length equal to the minimum extrusion length of the neighboring prisms, of which the vertex is part. Thus for vertex, v

$$\varepsilon_v = \min_{p \in \mathbf{P}(v)} \varepsilon^p, \tag{11}$$

where $\mathbf{P}(v)$ denotes the set of all prisms, of which v is part. Choosing the size of the adaptive extrusion length



(a) Propeller (b) Teapot

(c) Bowl (d) Funnel

Figure 5: Adaptive extrusion length using (11) causes self intersection of the thin shell layers.

in this way does not violate the convexity requirement to the prisms, since given a convex prism, one can always shrink any one of the extrusion lines without destroying convexity.

Nevertheless, this is a local solution, and as is shown in Figure 5, vertices with large extrusion lengths causes self-intersections with opposing faces. Our remedy is to use a root-search method to search for a smaller layer thickness without self-collisions. The idea is to re-cast the problem into a simulation formulation, where the extrusion length is thought of as a time parameter of an evolving surface. Thus we seek the point in time, where the evolving surface touches itself from opposite sides. The simulation loop consist simply of the following two steps:

- Perform collision detection,
- Update the extrusion length values.

These are repeated for a fixed number of iterations.

For each extrusion line we will keep a minimum value of the extrusion length $\varepsilon_{\min}$, a maximum value of the extrusion length $\varepsilon_{\max}$, and a current value of the extursion length ε. The interval $[\varepsilon_{\min}, \varepsilon_{\max}]$ represents the range of values, where we look for a solution for ε. Initially the value of the minimum, maximum, and current extrusion length are all set equal to (11). During the search for a solution, the minimum and current extrusion length values will be changed, but the maximum length value is left unchanged.

The parameter $\varepsilon_{\min}$ is the largest value that will not destroy the convexity requirement and is therefore always changed to a value that guarantees non-penetration. The parameter ε is the next guess for a non-penetrating extrusion line, and it is set to the half-way point between $\varepsilon_{\min}$ and $\varepsilon_{\max}$.

A spatial grid data structure [TBHPG03] is used to perform collision detection. During a first pass the axis aligned bounding boxes of all the extrusion lines are mapped into the spatial grid, and then in a second pass the axis aligned bounding boxes of the prisms are used to query for overlap with the extrusion lines. Whenever an overlap is found a penetration test is performed between the prism and extrusion line, The brute-force penetration test consist of testing, whether the extrusion line penetrates the five faces of the prism. This can be optimized to perform only penetration testing against the original surface face and the extruded face of the prism. If the extrusion line originates from a vertex shared with the prism, then the penetration test is ignored.

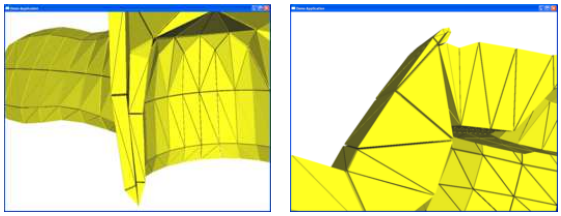
Upon having completed the collision query, a set of colliding extrusion lines and prisms are returned. Now we iterate over all these pairs, and mark each extrusion line, while finding the intersection point with the surface and extruded faces of the prism. If the distance to the intersection point from the originating surface vertex is lower than the minimum extrusion length, then the minimum extrusion length of the vertex is updated to this distance. If a non-penetrating extrusion line is found, then the current extrusion length ε yields a new possible value for the minimum extrusion length $\varepsilon_{\min}$. However instead of simply setting the minimum extrusion length equal to the current extrusion length, our experiments indicate that it is better to down-scale the value of current extrusion length, before assigning it to the minimum extrusion length. This is because, it is likely that the extrusion lines on the opposite side of the shell layer also will increase their minimum extrusion lengths. Down-scaling their values will reduce the chance for the growing extrusion lines to cause a self-collision.

Figure 6 shows a pseudo-code version of the simulation loop, which iteratively adjusts the extrusion lengths.

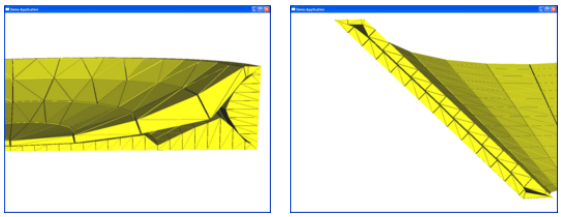
Upon completion of the last iteration of the simulation loop, the value of $\varepsilon_{\min}$ and not ε is used as the extrusion length, since only $\varepsilon_{\min}$ is guaranteed not to cause any self-collisions. Figure 7 shows close-ups of cut-views of volumetric meshes generated using the iteratively adjusted adaptive extrusion length method. Notice that the adaptive extrusion length is small near sharp rigdes, and at flat regions the adaptive extrusion length are increased to the point, where the extruded surface meets with prisms from the opposite of the object.

```
for i=1 to max iteration do
  Results R = collision(lines,prisms)
  for each (line,prism) in R do
    let p be originating point of line
    let v be intersection point of line
    εmin(line) = min(εmin(line),dist(p,v))
    mark(line) = true
  next (line,prism)
  for each line in lines do
    if not mark(line) then
      tmp  = ε(line)*0.9;
      if tmp > εmin(line) then
        εmin(line) = min(tmp,εmax(line))
      end if
    end if
    tmp = (ε(line) + εmin(line))/2
    ε(line) = min(εmax(line),tmp)
    q(line) = p(line) - n(line) * ε(line)
    mark(line) = false
  next line
next i
```

Figure 6: Pseudo-code for iterative adjustment of adaptive extrusion length.



(a) Propeller (b) Teapot



(c) Bowl (d) Funnel

Figure 7: Close-ups of volumetric mesh cut-views, showing the effect of the iteratively adjusted adaptive extrusion length, which gives a thick and non-overlapping layer.

4 PRISM TESSELLATION

For solid state simulations it is convenient to have objects on tetrahedra form, hence we will tesselate our prisms into tetrahedra. Due to space limitations, we will have to disregard degeneracies, these are however treated in details in [ED04]. For non-degenerate prisms having 6 corners, 3 is the minimum number of tetrahedra we can tesselate the prism into, and this tesselation

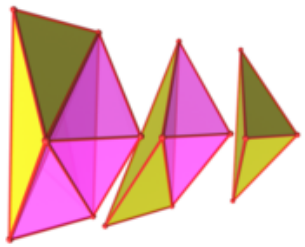


Figure 8: A Prism iteratively tessellated into 3 tetrahedra from left to right. It is helpful to imagine that the rightmost face has been inwardly extruded to produce the leftmost face. The yellow tetrahedra illustrate the iteratively produced tetrahedra.

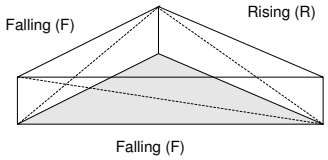


Figure 9: Classification of prism sides as falling (F) or rising (R).

is illustrated in Figure 8. For methods such as Finite Element Modelling (FEM) it is useful for neighbouring prisms to be tesselated such that the generated triangular faces match. We call this tesselation consistency, and it results in a global combinatorial problem.

A prism can be tesselated into three tetrahedra in 6 different ways. In order to classify the 6 types of tesselations, we will mark the rectangular sides of a prism as falling (F) or rising (R). The edge type depends on whether the tesselation edge is falling or rising as we travel along the extruded prism face in counter clock wise manner. This is illustrated in Figure 9. We observe that our 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 tabulated in Table 1. The consistency requirement implies that if one side of a prism is marked as F, then

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

Table 1: The 6 three-tetrahedra tesselation types.

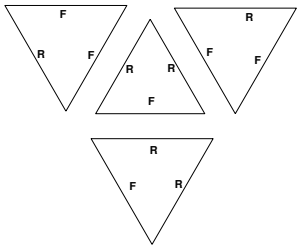


Figure 10: Tesselation example. A simple 3D surface mesh (a tetrahedron) as seen parallel to the surface normal. The letters R/F denotes the choice of Rising/Falling triangulation of the prism sides, which cannot be seen in this projection.

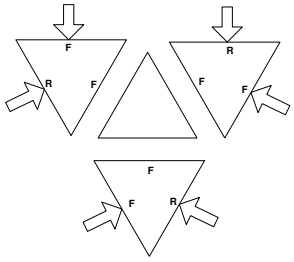


Figure 11: Inconsistent tesselation example. The middle prism will have the same type on all sides, which is not possible.

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 10.

Our algorithm for ensuring global consistency is 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, which is consistent with neighboring prisms already tesselated.

With this algorithm, inconsistency may arise as shown in Figure 11. Here, the middle prism is the last prism to be visited. Clearly, it is impossible to assign a tesselation type to the prism, since all three sides should have the same type. This can be repaired 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 11. In this case no further inconsistencies are introduced, and the repairing action of the example does not have large scale effect.

Fixing inconsistency locally is attractive, since it limits the computation time, but there is no guarantee that a local solution always can be found. An example of a non-local problem is the dead-lock shown in the top of Figure 12. In this example, none of the edges shared with the inconsistent prism can be flipped without creating an inconsistent neighboring prism, since all the edges marked with arrows are of the same type. The solu-

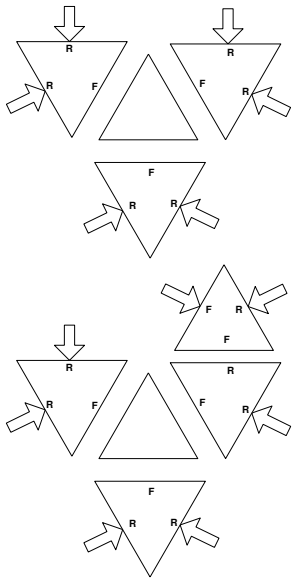


Figure 12: Top picture shows a inconsistent tesselation in a dead-lock. The bottom picture shows that inconsistency problem have been propagated to neighboring prisms further away by extended the region where we search for possible edge flips.

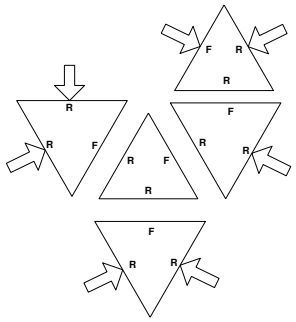


Figure 13: The rippling solution to the dead-locked case shown in Figure 12.

tion to the problem is shown in the bottom of Figure 12. 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. The result of the rippling for this specific case is shown in Figure 13. Notice that two edges are flipped, which are the edges lying on the path from the unassigned prism 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 14. It is our experience

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

Figure 14: Pseudo-code for determining tesselation pattern.

that the rippling distance rarely exceeds more than 1–2 faces [ED04]. The tesselation algorithm is thus a computational cheap and fast solution to an unpleasant problem.

5 RESULTS

The adaptive thin tetrahedral shell mesh method has been tested on several surface meshes, 14 of these surface meshes are shown in Figure 15, and number of iterations and total running time for these meshes are listed in Table 2. In [ED04] the time-complexity of the actual tesselation was shown to scale linearly with the number of faces in the original surface mesh. The running time is therefore clearly dependent on the shape of the original surface mesh. As can be seen from the table, surface meshes with small thin structures have the worst running time. This is because in every second iteration of the iterative adjustment of the extrusion length, extrusion lines in these thin regions are increased, leading to penetrations. In the succeding iteration the extrusion lines are shortened to remove the penetration. This oscillating behaviour only slowly converges towards a solution.

Figure 16 shows plots of the number of collisions per iteration. The plots indicate that the iterative adjustment of the adaptive extrusion lengths converges toward a solution, when given enough iterations. The plots however also indicate that the number of collisions is not a strictly monotone decreasing function. It is thus difficult to say anything conclusive about the convergence rate.

Cut-views of the generated tetrahedral meshes can be

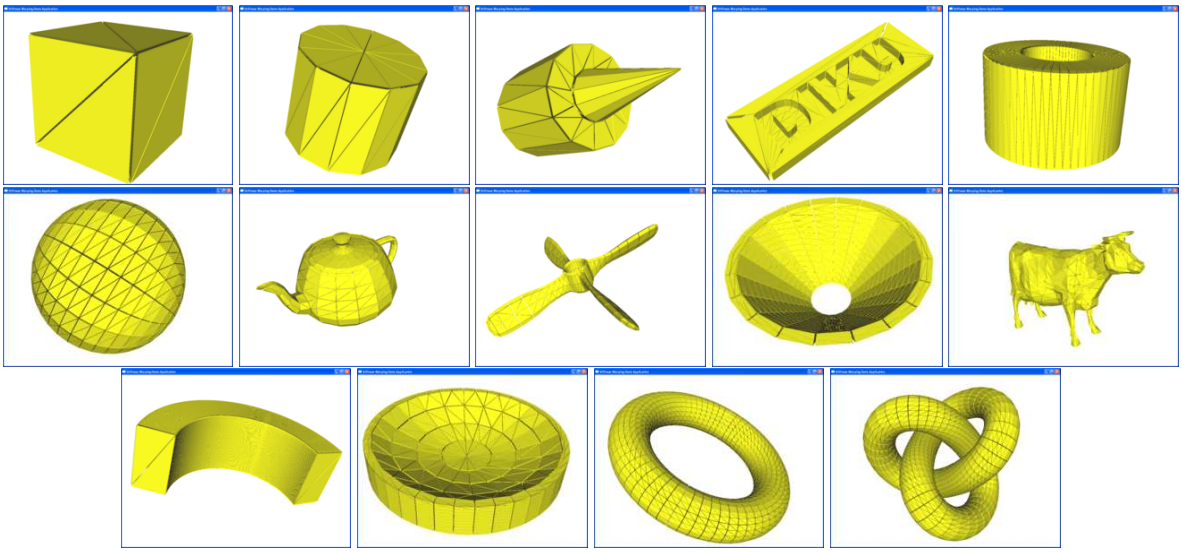


Figure 15: The 14 original surface meshes.

	$\|F\|$	$\|I\|$	time(secs.)
box	12	1	0.01
cylinder	48	1	0.01
pointy	96	100	0.42
diku	288	100	2.95
tube	512	1	0.15
sphere	760	1	0.26
teapot	1056	76	14.05
propeller	1200	72	19.89
funnel	1280	45	13.89
cow	1500	100	37.03
bend	1604	1	0.74
bowl	2680	85	136.75
torus	3072	1	1.49
knot	5760	1	5.94

Table 2: Performance Statistics using Iterative Adjustment. Maximum iteration count was set to 100 in all 14 test cases. The $|F|$-column gives the face count of the meshes, and the $|I|$-column shows the number of iterations.

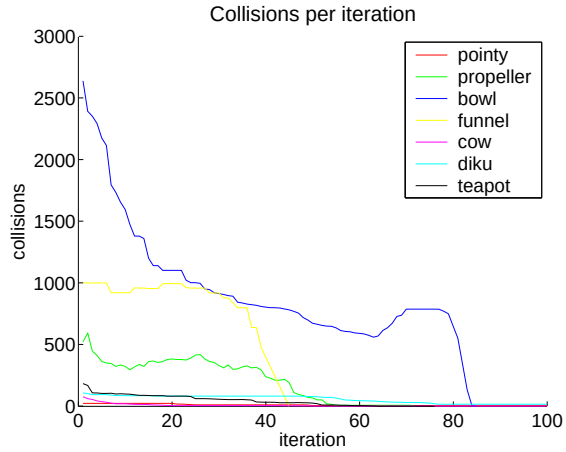


Figure 16: Collisions detected in each iteration. Collision-free test cases are not shown.

seen in Figure 1, 7, and 17. A user specified global maximum extrusion length was used, hence not all tetrahedral meshes fill out the inside void. The figures clearly show that the adaptive method is capable of filling out the inside of a surface mesh much more efficiently than the global extrusion length solution.

6 DISCUSSION

In this paper we have presented results showing that it is possible to generate a thin adaptive shell without topological errors. Our results show that the adaptive thin shell tetrahedral mesh generation method is versatile, robust, simple to implement, and yields useful results.
For the proposed consistent tesselation, we have not yet proven that it is always possible to find a consistent pattern of rising and falling tesselation edges. Neverthe-

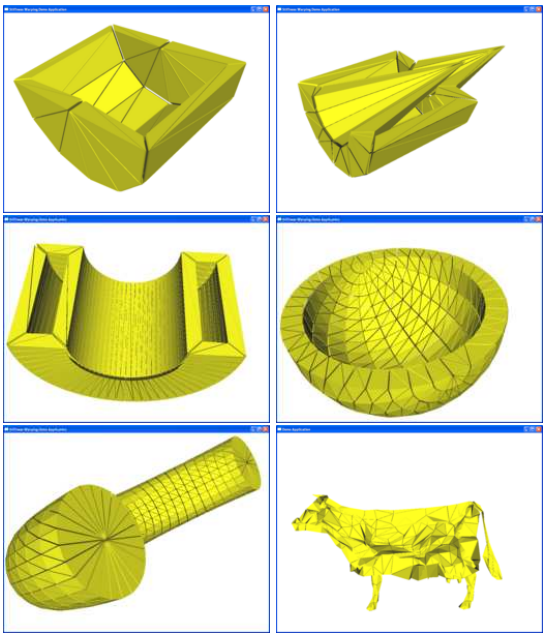


Figure 17: Cut-views of a few selected meshes.

less, we have not yet encountered difficulties with the method, and we believe that the combinatorial problem of finding a consistent tesselation pattern is tractable. We leave the proof for future work.

Lastly the iterative adjustment of the adaptive extrusion lengths can be improved in two ways. Firstly, the convergence rate could be improved to yield faster running times. Secondly, as seen in Figure 1, 7, and 17 some prisms seem to loose the race in increasing their extrusion lengths, before the algorithm terminates. The effect is most noticely seen in case of the bowl mesh, where some prisms could be extruded more to reduce empty space inside the mesh. We leave both these problems for future work.

References

[AB03] Henrik Aanæs and J. Andreas Bærentzen. Pseudo–normals for signed distance computation. In *Proceedings of VISION, MODELING, AND VISUALIZATION*, 2003.

[ED04] Kenny Erleben and Henrik Dohlmann. The thin shell tetrahedral mesh. In Søren Ingvor Olsen, editor, *Proceedings of DSAGM*, pages 94–102, August 2004.

[KTZ95] Benjamin B. Kimia, Allan R. Tannenbaum, and Steven W. Zucker. Shapes, shocks, and deformations I: The components of two-dimensional shape and reaction-diffusion space. *International Journal of Computer Vision*, 15:189–224, 1995.

[MBTF04] N. Molino, R. Bridson, J. Teran, and R. Fedkiw. Adaptive physics based tetrahedral mesh generation using level sets. (in review), 2004.

[MT03] M. Müller and M. Teschner. Volumetric meshes for real-time medical simulations. In *Proc. BVM (Bildverarbeitung für die Medizin)*, pages 279–283, Erlangen Germany, March 2003.

[NT03] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, 2003.

[Ope04] OpenTissue, 2004. http://www.opentissue.org.

[PS04] Per-Olof Persson and Gilbert Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, June 2004.

[Set99] James A. Sethian. *Level Set Methods and Fast Marching Methods. Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999. Cambridge Monograph on Applied and Computational Mathematics.

[TBHPG03] M. Teschner, M. Müller B. Heidelberger, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. Vision, Modeling, Visualization*, pages 47–54, Munich, Germany, November 2003.