Generation and Fracturing of Thick Shells

Denis Steinemann*

Computer Graphics Laboratory
ETH Zurich
Switzerland

Abstract

In this paper we present methods to generate and animate shells with a pre-defined thickness. Given a polygonal surface mesh, a thick shell is constructed by computing a second, extruded polygonal surface. We introduce methods to simulate deformation and fracture of thick shells in real-time. Mass-spring models are used in this context. A novel simulation framework to generate, animate and interact with shells in real-time has been created. The entire pipeline of shell animation may be handled, starting at the generation of shell, continuing on to deformation and fracture and ending with rigid body simulation of fractured shell fragments.

Keywords: Animation, Computer Graphics, Shells, Fracture, Physically-Based Modelling

1 Introduction

For the last decade, real-time computer graphics has been a very rapidly growing and active research field. With ever-increasing clock rates and more powerful graphics hardware, more and more realistic applications such as computer games and medical-, car- and flight simulators have become possible. The gaming industry, has become a larger market than the movie industry, indicating a huge demand for realistic and real-time 3D-graphics animations now and in the years to come. In early approaches, predefined animations were used to avoid time consuming computations that made it impossible to simulate in real-time. However, this is not enough, since realism not only depends on how things look but also how they act and how a user can interact with them. An important aspect is the physically plausible behavior of objects. This behavior is different for different types of materials and objects, which physically-based simulation must take into account. Real-time simulation of physically correct environments is still only in its early stages. Fields such as real-time rigid body physics and deformable objects have been extensively investigated in the last few years. Fracture and plastic deformation can be simulated in real-time using finite elements (FEM) . However, these methods become slow when animating stiff and brittle materials such as metal or stone, which are very common in virtual environments such as computer games.

*deniss@inf.ethz.ch

Objects in existing computer graphics applications such as simulators and games are usually represented by polygonal surfaces. However, physically-based simulations of deformation and fracture effects require volumetric representations such as tetrahedral meshes of the whole volume. Therefore, it is necessary to create a volumetric representation from a polygonal surface first [8]. However, in many applications objects such as cars, airplanes, bottles, etc. are actually hollow and therefore it is not efficient in terms of memory and time to simulate physics over the whole 3-D volume. A 2.5-D shell representation is enough.

1.1 Our contribution

We present a new iterative method to construct shells with a user-defined thickness, given a polygonal surface mesh. We then deform and fracture these shells using a fast and real-time approach based on a mass-spring-model. Finally, small fractured shell fragments are simulated as rigid bodies. We have implemented a framework to handle the entire pipeline of shell animation, consisting of generation, deformation, fracture and rigid body simulation. Our method features realistically looking results for brittle as well as ductile fracture of different types of materials.

2 Related Work

Shells have been analyzed using finite element methods [7, 10, 2] and membrane&flexural energies [13, 6]. [3] investigates how a triangle mesh can be simplified using so-called enveloping surfaces as error boundaries. An inner and an outer surface some user-defined distance ε from the original surface are constructed. Mesh simplification will then guarantee that the new surface lies between these two enveloping surfaces. We can use this approach by taking ε as the desired depth of our shell and using only one such surface, the inner one. In this paper, two algorithms to compute such surfaces are introduced, one of numerical and the other of more analytical nature. One of our methods is based on the first one. [1] uses the term ”fat surface” in connection with shells. They are constructed by the contours of trivariate function defined on prism scaffolds, for which two matched triangulation surfaces are needed. We will show methods how to obtain the second surface. In order to compute this second surface, collisions between polygonal surfaces must be detected. This problem is well

known in cloth-simulation [16] or in rigid-body dynamics [11].

Extensive research has been done in the field of fracturing. Müller [7] uses a static finite-element approach where an object is subdivided into volume elements called wedges. Strain and stress tensors can be computed from the deformation of these wedges, and when they become larger than some threshold, the object is fractured. O'Brien [10] uses an explicit dynamic FEM approach. This approach features extremely realistic results by simulating correctly the exact physics of a material. Unfortunately, it is computationally also very expensive and therefore not practical for our uses. Eberle et al. [4] present a procedural approach to modelling impact damage. They use very simple fracturing criteria, which makes the method very fast but also seems limited to a few cases. Deformation and fracture is only computed in some local region, and is based on lengths and strengths of edges in a triangle mesh. This approach seemed promising to us in the sense of being easy and fast. It is not physically correct, however. Rigid body dynamics have also been extensively investigated. There exist software development kits to simulate rigid bodies.

3 Shell Generation

3.1 Overview

Our goal is to create a shell with a user-defined thickness from a given triangle mesh surface. Our definition of a shell is a polygonal surface S, consisting of vertices and edges, which encloses a second, inner surface S'. S' does not intersect S nor itself, and is some distance ε from the original surface. Each vertex contains information about the thickness of the shell at that position (vertex depth d). In [1, 3] a shell is generated by starting at a vertex v, moving in the opposite direction of the vertex normal and after some distance creating another vertex v', resulting in a pair (v, v') which is called a shell vertex. When this is done for every vertex, a new inner surface S' is created. S' has the same topology as the original surface mesh. A shell element is a "thick triangle" consisting of three shell vertices connected by 3 pairs of edges. The two triangles t and t' are part of S and S', respectively. When we refer to an edge e, we usually mean a side of a shell element.

If the thickness is small enough, the described method works fine. A problem arises when the original object is too thin in some places so that S' intersects with itself or even with S. See 1(c). Intersections are visually unappealing and will cause problems later when computing collision responses. In such a case, corresponding shell vertices cannot have the desired shell thickness and must be shortened to avoid any intersections. On the other hand, the shell should still have maximum possible thickness in these places. Simply setting the vertex depth to zero for these vertices is not a good solution, because one would create degenerate shell elements.

A degenerate shell element has a shell vertex with depth $d = 0$ and/or vertex normals that point in akward directions, such that the two triangles t and t' of the element intersect. If vertex normals are not consistently oriented outward in the original surface mesh, it is impossible to make the depth of vertex v anything other than zero, because otherwise t and t' of the corresponding shell element(s) will intersect, which should be avoided in any case. Unevenly distributed shell thickness will influence the stability of the shell later when doing deformation and fracturing. Degenerate elements should be avoided if possible for later simulation of deformation or fracture.

3.2 Shell Generation Algorithms

We have implemented two algorithms that are based on the approach that we have described above. They are both of iterative nature in the sense that they try to build a shell in small steps, starting with low vertex depths and then growing it towards the desired thickness. When the step size per iteration is small enough, both algorithms produce a shell in a consistent state.

3.2.1 Requirements

There are several requirements that a shell construction algorithm working on a well-defined surface should fulfill. A *well-defined* triangle mesh surface in our context has no self intersections, is non-manifold, and has well defined vertex and triangle normals pointing outward (i.e. in consistent direction). It may have holes (it does not have to be a closed surface). Thus, a shell construction algorithm should

- guarantee that for a well-defined surface S, an extruded surface S' is created so that no degenerate shell elements are created. Then the shell is in a *consistent* state.
- maximize average shell thickness. It is undesirable to have a shell with maximal thickness in some places and a thickness close to zero in others. If possible, thickness should be distributed more evenly, so that we get as similar shell elements as possible.

A shell generation algorithm must do collision detection for the shell to be in a consistent state at all times during the algorithm. A good collision detection approach such as spatial hashing [15] is crucial in terms of computation time.

3.2.2 Single Vertex Propagation

The first algorithm is based on a numerical algorithm from [3]. Here, all shell vertices have some initial step size ε_i, some fraction of the desired shell thickness ε. Initially, $v'_i = v_i$ for all vertices (and therefore $t = t'$ for all shell elements). One single vertex v'_i per step is extruded by its step size along its opposite normal to create a new vertex v'_i. Then, all triangles t' of the shell elements that contain (v_i, v'_i) are tested for collision with the triangles t and t' of shell elements in some local neighborhood. If there is

no collision, the move is accepted and the algorithm moves on to the next vertex v'_{i+1}. If there is a collision, however, the new v'_i would put the shell into an inconsistent state. Therefore, the new vertex cannot be accepted and we must go back to the last position of v'_i and try to move it again with a smaller step size. In our implementation, the new ε_i is set to half the old step size. Once again, all elements containing (v_i, v'_i) are tested for collision with neighboring shell elements. If no collisions are detected, the move is committed, otherwise we must half the step size once more and try again until the new shell is in a consistent state. If ε_i becomes smaller than some ε_{min}, we know that v'_i is already very close to another part of the shell, so we stop trying to move it further. Then we can move on to v'_{i+1}. This is continued until each vertex has moved k steps (k a user-defined value) or its step size has become too small.

3.2.3 Front Propagation

The main drawback of the single-vertex propagation algorithm is the high number of collision detections that must be made. Since most of the time is used by collision detection, we have worked out and implemented a second algorithm that constructs a consistent shell while doing much less collision detection. Instead of moving only one vertex at a time, all vertices (or in other words, the whole *shell front*) at once are extruded by their ε_i. Then all shell elements are tested for collision against each other. Two shell elements e_1 and e_2 intersect, if one of e_1's triangles t_1 or t'_1 intersects t_2 or t'_2. If there is a collision, all six extruded vertices v' of the two elements are set back to their last position and their step sizes are halved. Unlike in the single-vertex propagation algorithm we do not try to move these vertices again right away. In the next step, all vertices v' are once again moved by their ε_i. See Figure 1 for a graphical explanation of the algorithm. As in the single-vertex propagation algorithm, one possibility is to set the initial step sizes to $\varepsilon_i = \varepsilon/k$. The main advantage of the front propagation algorithm is that it does less collision detection and is therefore faster. Per iteration over the whole shell, each shell element is tested exactly once against other shell elements. For single-vertex propagation, this is done three times for each shell element, once for each of its vertices. Per rejected move, all adjacent shell elements must be tested additionally.

However, the front propagation algorithm will more likely miss collisions, because whole fronts may overlap in one iteration. Thus, step sizes must be smaller, and so the speed gained by this new approach is partially lost.

3.2.4 Step sizes

We have done some simple improvements to speed up shell computation and enhance shell quality. In single-vertex propagation, time performance strongly depends on the number of times a vertex move is rejected, which is a direct consequence of a step size which is too large. For front propagation, correctness depends on the step size - if it is too large, the algorithm may not detect overlapping shell elements. It is therefore a good idea to pre-compute the initial step sizes for each vertex instead of using the same step size $\varepsilon_i = \varepsilon/k$ throughout the shell. To do this, we define a line segment s_i, starting at original vertex v_i and pointing in the opposite direction of the vertex normal (i.e. the direction of vertex extrusion). The length of s_i is $d_i = 2\varepsilon$. Using a regular grid or spatial hashing [15] data structure, we compute collision points of s_i with the triangles $t = t'$ of neighboring shell elements. If there are collisions, let c_0 be the closest collision point from v_i with an element e_0 and $d_i = \|c_0 - v_i\|$. Dividing d_i by 2 will then be the maximum extrusion distance for vertex v_i. This way, the element e_0 will have a chance to extrude its vertices also as far as possible, thus evenly distributing shell thickness. Step size is then simply $d_i/2k$. If there are no collisions, step size is $2\varepsilon/2k = \varepsilon/k$, which is what we had without initial step size computation and which will give maximum extrusion ε. When initially computing the step sizes, we can avoid most collisions during the propagation algorithms.

Instead of reducing step sizes, a different approach could be to use a continuous collision detection scheme [11]. This would allow us to use larger step sizes without decreasing them according to the surface geometry. In addition, problems in the front propagation algorithm, where whole fronts overlap at once and thus no collision is detected, could be better avoided with this approach.

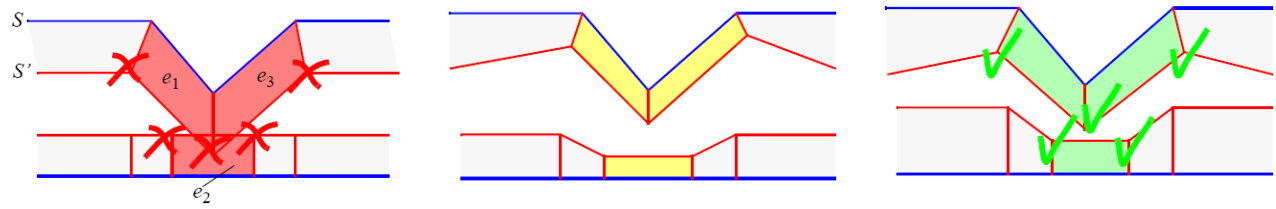
4 Deformation and Fracture

In a first part, we will describe our physical model to simulate deformation and fracture of a thick shell. In the second part, we will illustrate the different types of fracture that can be simulated with this model.

4.1 Physical Model

We have chosen mass-spring systems as our physical model to animate a thick shell. The reason for this choice is its simplicity: mass-spring models are both easy to implement and implicitly feature an easy fracture criterion: spring length. In a shell S, the shell vertices v_i and v'_i serve as points, while the edges e_i of the shell elements represent the springs.

For a mass-spring-model to be stable, however, having springs just on the object surface is not enough. There must be springs throughout the whole volume of the object. Since we basically use mass-spring models only to have some fracturing criteria, it is not a problem to use only one or two layers of springs on the object surface. If we used springs that span the whole volume, we would once again be simulating physics over the whole object, which is unappropriate when the object is hollow and does not have a volume. If the mass-spring dynamics cause a spring to become too long or too short (compared to some threshold percentages of the initial length), the shell surface mesh must be fractured. No stress or strain tensors must be computed. The spring lengths are implicitly given



(a) A collision is detected between elements e_1, e_2 and e_3.

(b) All vertices v' of these elements are moved back to their last position and their step sizes are halved.

(c) In the next iteration, the whole front is moved again with the new ε, creating a consistent shell without intersections.

Figure 1: Graphical explanation of the front propagation algorithm.

by the dynamics. We have implemented two approaches on how to use the mass-spring model.

One possibility is to use just one layer of springs on the surface. The advantage of this approach is its speed, and it is enough to simulate brittle fracture. However, it is not suitable to simulate deformation, because there are no springs or other extrinsic energy terms [6] that take surface bending into account. In addition, shell thickness is not considered at all in the physics. It is only needed for rendering.

For these reasons, it is necessary to use two layers of springs, one on surface S of the shell and one on the inner surface S', which are then connected to each other by additional springs (*internal* springs). See Figure 2. This approach is slower, because more springs must be simulated. However, in addition to distance-preserving forces, we can use surface area and volume preserving forces [14] to make our model more stable. Although surface area and volume preservation work well and are an important part of shell dynamics, they have one drawback: if a shell element (and thus the three tetrahedra it can be subdivided into) are too thin, volume preservation will not work. Having

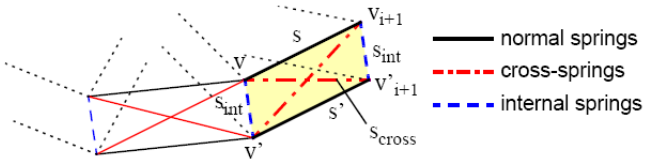


Figure 2: A two-layer Mass-Spring-Model must have cross-springs and internal springs to increase the stability.

shell elements of some minimal thickness is an unacceptable constraint, because our goal is to simulate stable shells independent of thickness. For best results, the length of an internal spring between v and v' of a shell vertex should have approximately the same length as the adjacent springs of the vertex. Such a shell element is thick enough for volume and area preserving forces to have the desired effects. We introduce a so-called *extrusion length factor* f. It allows the user to make the thickness of a shell vertex f times as large as the original vertex depth. The two-layer mass-spring model is based on these new vertex depths. This will be used behind the scenes just for computation of the dynamics, it is not rendered, so that user sees the same shell with the same thickness as before, the only difference being a shell that is more stable.

Figure 3: A thin shell is more easily fractured than a thick shell.

It is desirable to have some kind of relationship between shell strength and shell thickness. A thick shell should be harder to fracture than a thin shell. One possibility to model this behavior is to use the spring constants k by making k proportional to the vertex depths. The thicker a shell is, the stiffer all the springs are. In order to cause a fracture, the length of a spring s on the shell surface must exceed some threshold. The same force F will move a vertex connected to softer springs more than a vertex attached to stiff springs, increasing the probability that the fracture threshold is exceeded. In Figure 3, a massive object has been thrown at a wall. The thin wall breaks more easily than the thick one.

4.2 Fracturing

4.2.1 Mesh Operations

The basic fracture criterion in our model is spring length. When the length of a spring on surface S exceeds a threshold, a crack is initiated by splitting one of the spring's endpoints v or v_2. A crack propagates by repeated use of an elementary operation called *vertex split* [7].

There are two ways to choose this vertex, either select it randomly or depending on so-called crack tips [7]. Crack

tips state if a vertex is at the end of an existing crack or not. Vertices with a crack tip are preferably chosen to increase the probability that an existing crack is extended and not a new, small crack is created. This models realistic behavior where the material near a crack is weaker. In our model, the physics itself does not always guarantee this behavior. If both or none of the endpoints have crack tips, the point to be split must be chosen randomly.

Let us assume that vertex v has been chosen. A *splitting plane* E is defined. E contains vertex v and its normal is the vector $(v_2 - v)/||v_2 - v||$. Now, divide v's adjacent shell elements into two disjoint sets $E+$ and $E-$. All elements in $E+$ get the new vertex. Thus, when the vertex split operation is finished, spring s will return to its initial length, which creates a gap or a small crack in the shell.

Since we do not create new shell elements, it is necessary that the initial triangle mesh has some minimal resolution. If this is not the case in some parts (often this happens in planar regions), large edges are subdivided and new triangles are created in a preprocessing step [8].

4.2.2 Fracture Types

We simulate three types of fracture by controlling crack propagation and vertex splits.

Fine-grained fracture produces many small fragments. Modelling this behavior is quite easy, because unless the simulation time step is made very small, a mass-spring model is quite inert with respect to movements of shell vertices. Springs can be stretched strongly before forces take effect and bring the model back to its original shape. It is thus not just a few springs whose lengths exceed the threshold, it is actually most springs in some region of impact. Simply passing over all springs with no constraints will result in many vertex splits and thus many small shell fragments. See Figure 4 for an example of fine-grained fracture. In *coarse-grained* fracture, larger fragments are created by artificially keeping vertices together. The idea is to have a flag for each spring which determines if it can initiate a crack. The flag is set to true by default. When a vertex v is split, all its adjacent springs have their flag set to false. The same is done for the adjacent springs of all adjacent vertices of v, except the ones at the ends of the crack, which get crack tips. Again, leaving vertices at the end of the cracks separable forces longer cracks to appear and tends to suppress the creation of small and short cracks.

With fine-grained and coarse-grained fracture as described above, it is not possible to simulate *large-fragment* fracture. Large-fragment fracture is characterized by a shell being broken into a few large pieces. This is often the case in reality. For example, a vase or glass usually do not break apart into many small pieces when they fall down on the ground. Rather, a few large shards together with some smaller ones are created. This is because for hard materials such as porcellan or glass, once a crack has turned up, it propagates extremely quickly through the whole object or at least some part of it. The reason for this is that in the region of a crack, the material has weakened and so the probability that it breaks again is higher in regions of cracks than in other places. Because of the inert nature of mass spring models (and FEM models as well), it is not possible to simulate this kind of crack propagation. It is therefore necessary to artificially extend cracks along the splitting plane. Another possibility is to control crack frequency, i.e. limit the number of vertices that can be split in one time step. As a nice side effect, a small crack frequency will also speed up the simulation, since there are less new vertices or springs that pose an additional computational burden.

4.2.3 Crack Handling

For large-fragment fracture, cracks are artificially extended along the splitting plane E for some user-defined distance r, independent of any fracture criteria that must be satisfied. This way, long cracks and thus larger shards are created. Our implementation is based on [7].

The smoothness of cracks artificially extended depends strongly on the tessellation of the original shell mesh. Often, cracks look very jagged, which may be a desired effect, but often the tessellation of the shell has some pattern, making this jaggedness look regular and therefore unrealistic. To create smoother cracks, we have implemented a method that locally subdivides shell elements. The idea to this approach is from [10]. We insert new vertices and split shell elements that are cut by the splitting plane into two new elements before actually splitting the vertex. The crack near this vertex is then a straight instead of a jagged line. In addition, when propagating a crack along the splitting plane, the direction of the plane can be altered slightly at each new vertex to create bumpy, but not jagged cracks. Refer to Figure 5. The implementation of this local re-meshing scheme is quite tedious, because the local neighborhoods must be kept consistent at all times. Nonetheless, it greatly improves the appearance of fracture. The procedure may create many very slim new elements, which slows down the simulation and may cause problems with volume preservation. Therefore, shell elements are only split if the new elements are well-shaped enough. With a reasonable threshold, local re-meshing is still quite fast and features realistic results.

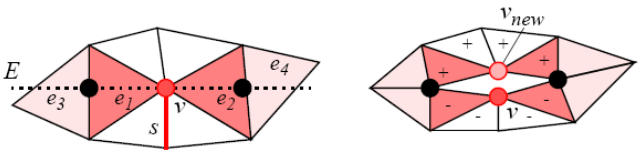


Figure 5: Local Re-Meshing: Instead of splitting shell elements along existing edges, new vertices and edges are created, forming straighter cracks.

4.2.4 Brittle vs. Ductile Fracture

Brittle fracture is characterized by stiff materials breaking up into inelastic parts. An example would be throwing a

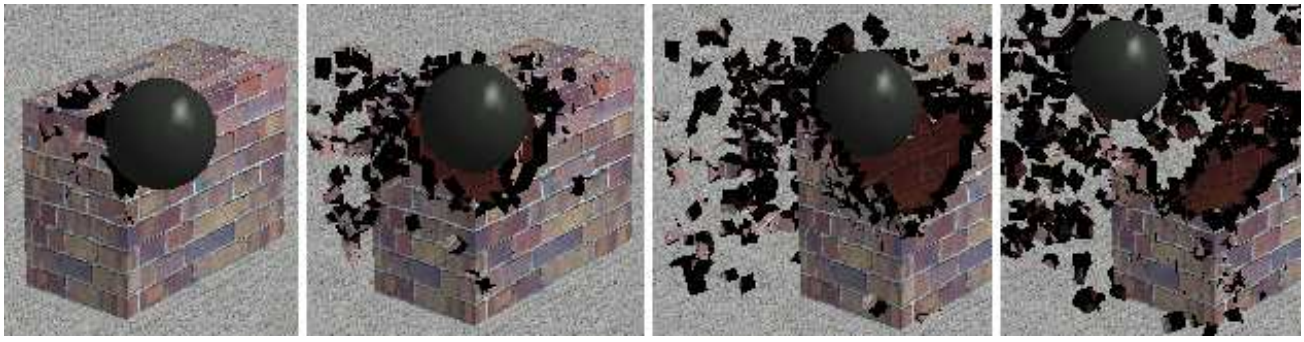


Figure 4: An example of fine-grained fracture.

massive object at a brick wall, causing it to collapse. See the images in the previous sections. These were all created using brittle fracture. Mass-spring-models are well suited to simulate deformation and fracture of objects. Some materials, however, are not really deformed at all because they are very inelastic and stiff. For such a material, porcellan for example, the deformation of the shell is an unwanted feature. We therefore render only the movement of shell fragments (which are treated as rigid bodies). The dynamics of the mass-spring model must be computed for the fracturing process. However, this deformation is only simulated behind the scenes and is not rendered. Even though this is a simple approach, it yields very good results.

To simulate ductile fracture [9], shell vertices are directly matched to the dynamics of the mass-spring model. However, this requires a stable mass-spring model. It must therefore be noted that for the above reasons and because we do not model plasticity, we get much better results doing only brittle fracture. See Figure 6 for an example of ductile fracture.

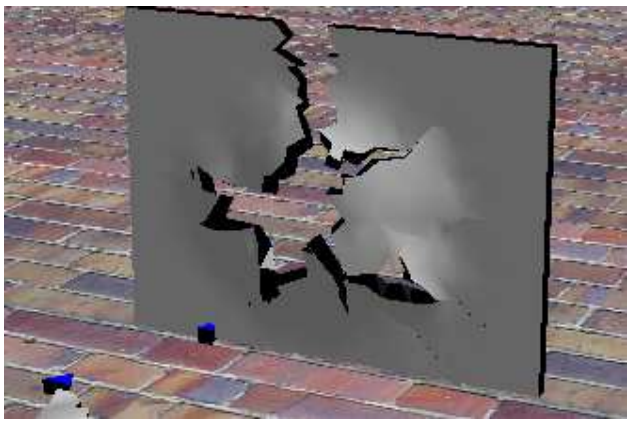


Figure 6: Ductile fracture: a piece of soft material is fractured.

4.3 Rigid Bodies

For a fracturing simulation to be complete, one probably cannot omit the concept of rigid bodies. To achieve most realistic results, new fragments should be treated as rigid bodies. There exist robust software development kits which handle rigid body movement as well as collision detection between the bodies, making it relatively little work to implement this feature. We use the commercial rigid body engine by NovodeX AG.

To detect new fragments, connectivity components in the shell must be computed first. This can be done in linear time $O(n)$, where n is the number of shell elements. In order to speed up the rigid body simulation, we approximate shell fragments by object-oriented bounding boxes (OBB). Positions of fragment vertices are updated according to the OBBs local coordinate system. To compute a fragment's minimal OBB, we use Principal Component Analysis [5], because it features satisfactory results and is easy to implement.

Initial values such as mass, linear velocity, angular velocity and the inertia tensor of a rigid body can be computed from the shell vertices contained by the body.

5 Results

5.1 Shell Generation

We have developed a new iterative algorithm to generate a thick shell from a triangle mesh surface. Single-vertex propagation produces consistent shells with only a few iterations. Our faster front propagation algorithm yields good results, although one must use more iterations. When step sizes are initially pre-computed, however, the quality of the generated shells is comparable for both algorithms at the same number of iterations. Table 1 shows computation times for two shell models. Shells were generated with initial step size computation. Note the lower number of shell element collisions for front propagation. The front propagation algorithm is about 3-4 times faster than single-vertex propagation. Nonetheless, quality (i.e. number of degenerated shell elements) of the shells is very good for both algorithms. Thus, the front propagation algorithm is to be used preferably if computation time is important. For our performance measurements, the pig (Figure 7) has 3500 vertices and 4000 triangles, while the castle (Figure 9) model consists of 6500 vertices and 12000 triangles. Tests were made on a standard PentiumIV 3GHz PC. For reasonable shell thickness a non-intersecting thick surface is generated. Using initial step size computation,

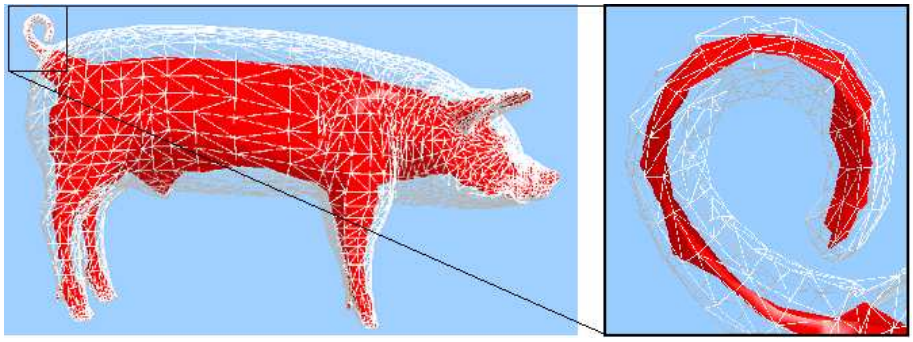


Figure 7: Shell model of a pig. The red surface is the inner surface S', while the white mesh is the original surface S. At the tail and the legs, shell thickness could not be made maximal. Nonetheless, there are no intersections between S and S'.

Figure 8: The horn and ear of a cow. Even in complex models, our shell generation algorithm produces good results.

Model	ε	1-Vertex Propagation		Front Prop.	
		[s]	# of collisions	[s]	# coll.
Pig	0.01	4.0	8196	1.1	696
	0.05	5.2	22106	1.4	1615
	0.1	6.4	28498	2.0	2310
Castle	0.01	6.4	31439	2.6	7314
	0.02	7.2	41147	3.1	7983
	0.05	8.4	47005	4.6	8416

Table 1: Shell generation computation times in seconds for different models and with various thicknesses. Number of iterations k=5.

desired shell thickness does not affect the correctness of either of the two algorithms much, because oversized vertex moves are already caught during pre-computation of the step sizes. See Fig. 7 for an example of a shells. The red surface represents the inner surface S', while the white mesh is the original surface S of the shell. For the pig, notice the very thin shell at the tail, while near the center of the body the actual thickness could be made close or equal to the desired thickness. Also notice the horn and ear of a cow, where the new inner surface fits nicely to the original surface.

5.2 Deformation and Fracturing

We have developed three different types of fracture containing many user-defined parameters that can be used to simulate different materials. Shell fragments are treated as rigid bodies, enhancing the realism of our simulation greatly. On a standard Pentium IV 3GHz PC with ATI RADEON X800 graphics hardware, the simulation is real-time for models up to 10000 vertices, such as the cow or the castle (11000 vertices, 22000 triangles) (Figures 9 and 10). Due to the approximation of shell fragments with bounding boxes, the rigid body simulation is fast, leaving the bulk of computation time with mass-spring dynamics. See Table 2 for a comparison of computation times for different fracture types for the models below (for brick walls refer to Figure 4). It must be noted that simulation of the rigid bodies accounts for about 20% of total computation time. Large-fragment fracture without local re-meshing generally features the lowest computation times because for low crack frequency, only a small part of the springs will be iterated over at all. The same may be said for coarse-grained fracture. When a spring is already split, it is omitted and no forces must be computed, resulting in a faster simulation. Only one layer of springs is simulated. To speed up the simulations, only the dynamics of springs in some impact neighborhood are simulated. This is a valid assumption because one would expect an object to fracture in a region of impact.

Model	# vertices/△s	Frame Rate [fps]		
		Fine	Coarse	Large-Frag.
Walls	600/1200	100	160	70
Cow	3000/5700	32	40	30
Castle	11000/22000	10	11	9

Table 2: Frame rates for different fracture types and models. Only one layer of springs is simulated.

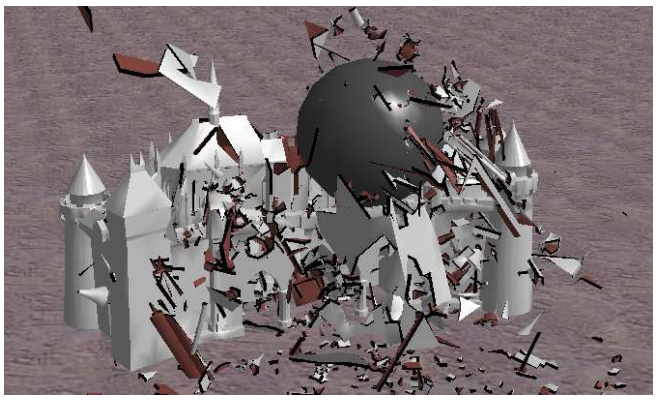


Figure 9: The Disney Castle fractured in real-time.

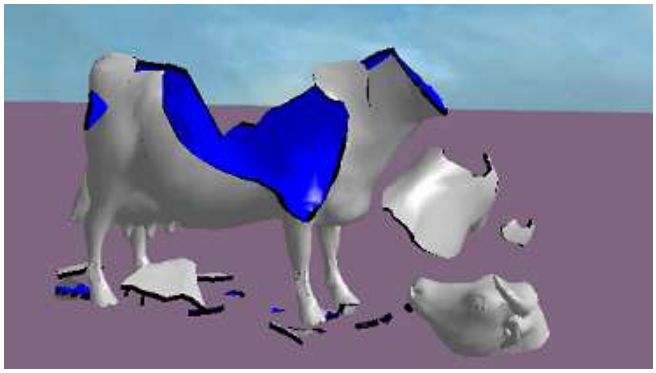


Figure 10: A fractured shell cow. Due to local re-meshing, cracks appear smooth.

6 Conclusions and Future Work

We have presented a fast and easy method to model realistic fracture of shells. We have shown methods to construct a shell with a user-defined thickness, given a polygonal surface mesh. We have implemented a framework to handle the entire pipeline of shell animation, consisting of generation, deformation, fracture and rigid body simulation.

While our approach is simple and fast, it is not really physical. Instead of using mass-spring models, we could also use finite elements to compute the deformations and fractures. This could easily be implemented in our framework. However, the simulation will become substantially slower. To make deformation more stable, bending energies could be additionally included in the physical model.

The real-time but somewhat non-physical nature of our approach makes it interesting for applications where a high frame-rate is crucial, for example computer games.

7 Acknowledgements

This research is part of a Diploma Thesis [12] submitted to ETH Zürich in 2004. The author would like to thank Dr. Matthias Müller and Prof. Dr. Markus Gross for their support in this work.

References

[1] C. Bajaj, G. Xu, R. Holt, and A. Netravali. *Hierarchical Multiresolution Reconstruction of Shell Surfaces*. Computer Aided Geometric Design, 2002. Volume 19. No. 2 Pages 89- 112.

[2] M. Bernadou and J.M. Boisserie. *The Finite Element Method in Thin Shell Theory: Application to Arch Dam Simulations*. Birkhäuser, 1982.

[3] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F.P. Brooks Jr., and W.V. Wright. *Simplification Envelopes*. Proceedings of SIGGRAPH '96 (New Orleans, LA), August 4-9, 1996.

[4] D. Eberle, O. Strunk, and R. O'Sullivan. *A Simple Hack for Breaking Stuff*. ACM SIGGRAPH Sketch, 2003.

[5] S. Gottschalk, M.C. Lin, and D. Manocha. *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. Proc. of ACM Siggraph, Pages 171-180, 1996.

[6] Eitan Grinspun, Anil N. Hirani, Mathieu Desbrun, and Peter Schröder. *Discrete Shells*. Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation, July 26-27, 2003, San Diego, California.

[7] M. Müller and M. Gross. *Interactive Virtual Materials*. Submitted to Graphics Interface (GI), 2004.

[8] M. Müller, M. Teschner, and M. Gross. *Physically-Based Simulation of Objects Represented by Surface Meshes*. Computer Graphics International (CGI), Crete, Greece, Jun 16-19 2004.

[9] J. O'Brien, A. Bargteil, and J. Hodgins. *Gaphical Modelling and Animation of Ductile Fracture*. Proceedings of SIGGRAPH'02, San Antonio, Texas, pp. 291-294, 2002.

[10] James F. O'Brien and Jessica K. Hodgins. *Graphical modeling and animation of brittle fracture*. Proceedings of SIGGRAPH 99, pages 137-146, 1999.

[11] S. Redon, A. Kheddar, and S. Coquillart. *Fast continuous collision detection between rigid bodies*. CGForum 21, 3, pp. 279-288, 2002.

[12] D. Steinemann, M. Müller, and M. Gross. *Generation and Animation of Shells*. Diploma Thesis, ETH Zürich, 2004.

[13] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. *Elastically Deformable Models*. Proceedings of SIGGRAPH, pages 205-214, 1987.

[14] M. Teschner, B. Heidelberger, M. Müller, and M. Gross. *A Versatile and Robust Model for Geometrically Complex Deformable Solids*. Computer Graphics International (CGI), Crete, Greece, Jun 16-19 2004.

[15] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. *Optimized Spatial Hashing for Collision Detection of Deformable Objects*. Proceedings of Vision, Modeling, and Visualization 2003, pp. 47-54.

[16] P. Volino and N. Magnenat-Thalmann. *Accurate Collision Response on Polygonal Meshes*. Proceedings of the 2000 Conference on Computer Animation, p.154, May 03-05, 2000.