Physically-Based Simulation of Objects Represented by Surface Meshes

Matthias Müller Matthias Teschner Markus Gross

Computer Graphics Laboratory
ETH Zurich
muellerm@inf.ethz.ch

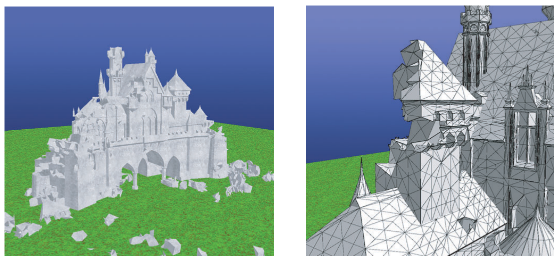

Figure 1: Interactively fractured surface mesh with dynamically generated closing surface.

Abstract

Objects and scenes in virtual worlds such as 3-d computer games are typically represented by polygonal surface meshes. On the other hand, physically-based simulations of deformations or fracture effects require volumetric representations such as tetrahedral meshes. In this paper we propose techniques to generate volumetric meshes dynamically for objects represented by surface meshes allowing the simulation of physical effects such as motion, deformation and fracture.

We use the Finite Element Method based on cubical elements of uniform size. Regular cube meshes have several advantages over geometrically more complex representations. Because of their simplicity, cube meshes can be generated quickly by voxelizing objects while neither geometry nor stiffness information needs to be stored explicitly. The low memory consumption makes physically-based animation possible for large scenes even on game consoles. We animate the original high resolution surface mesh by coupling it to the underlying volumetric mesh. This way, the regular structure of the volumetric mesh is hidden from the user. We also propose a technique to fracture the surface mesh along with the cube mesh which keeps the surface watertight and results in realistic fracture patterns.

Key words: Physically Based Animation, Finite Element Method, Deformation, Fracture

1 Introduction

1.1 Motivation

Virtual environments in which the user can interact with a virtual world have become very popular in recent years. Three dimensional computer games are an important example. They represent the largest market in this field. Besides computer games, there is a large number of other examples for interactive virtual environments such as medical-, car- and flight simulators or caves for virtual collaboration.

To give the user the impression of a physical world, it is important that objects behave like they do in the real world. Virtual scenes such as those encountered in computer games are composed of a large number of objects most of which are typically represented by surface meshes only. To give the impression of solid physical objects – especially when these objects are deformed or shattered into pieces – volumetric representations are needed. The generation of such representations is expensive and usually not part of today's game developing art pipelines.

In this paper we present a method to generate such volumetric representations automatically for given surface meshes. As representation we chose meshes of uniformly sized cubes. Because of their simplicity, their generation is very fast and their memory consumption low. The mesh geometry is obtained via voxelization of the surface mesh. We animate the resulting regular hexahedral mesh using the Finite Element Method (FEM) and/or a rigid body simulator. The original surface mesh is deformed and fractured according to the simulation performed on the underlying volumetric cube mesh.

1.2 Related Work

In recent years, physically-based animation for interactive systems has been an active research field in computer graphics with many important contributions. Among the physical phenomena that have been simulated *in real-time* are rigid-body-dynamics [1, 16]), deformable objects [7, 4, 17, 8, 2, 12], fluids [14], fracture [13, 9], just to mention a few contributors out of a much longer list. Most of these methods use volumetric representations to simulate volumetric effects. Exeptions are rigid body simulators and the boundary element method used by James and Pai.

MathEngine[1] and *Havok*[2], the market leaders for physically-based animation engines for computer games have included quite a number of these techniques into their physics engine. However, physically-based deformations or fracture effects for large scenes are not included yet and have not appeared in computer games so far. One of the reasons is the fact that models and methods used in computer graphics research become more and more sophisticated and complex. On the other hand the complexity of the geometry of scenes in computer games increases as well. To animate such complex scenes simple and efficient methods in terms of memory and time consumption are needed. The method we present in this paper tries to bridge this gap.

1.3 Contribution

We propose the use of regularly shaped cube meshes as volumetric representations for scenes and objects defined by surface meshes only. We also describe simulation methods for cube meshes and methods to animate the associated surfaces which hide the simplicity of the underlying mesh structure.

2 Cube Mesh

2.1 Mesh Generation

A cube mesh is a set of equally sized cubical elements that are linked via shared vertices. The cube size h is a global parameter of the simulation. Tables 1 and 2 show the attributes of cube elements and vertices respectively.

Attributes	Description
x, y, z	Integer position indices with respect to the bounding box of the surface
$v[8]$	Pointers to vertices
triangles	Pointer to the list of triangles that intersect the element

Table 1: Cube element attributes.

Attributes	Description
position	Position in the deformed mesh
velocity	Velocity for dynamic simulations
mass	Mass lumped from all adjacent cubes

Table 2: Vertex attributes.

After a surface mesh is loaded, the cube mesh is generated automatically. First, the axis aligned bounding box

[1]www.mathengine.com
[2]www.havok.com

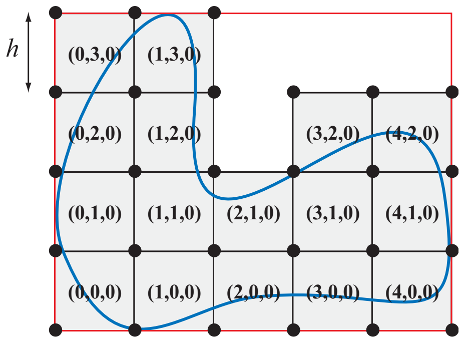

Figure 2: A 2-d cut through a cube mesh. The cube elements are generated inside the bounding box (red) of the surface mesh (blue). Three integer indices define their positions with respect to the bounding box.

of the surface mesh is divided into cubes of size h. Then, for all cubes that either contain part of the surface mesh or lie completely inside the surface, a cube element c_i is generated. For each of the eight corners, a vertex is generated in case no vertex exists at that location yet and a link to the new or existing vertex is stored in c_i. All cube elements store integer position indices x, y, z which number them along the coordinate axes. Together with the location of the bounding box of the surface mesh these indices determine the location of the element and the locations of its vertices in space (see Fig. 2).

We store all cube elements $c_1 \dots c_N$ in a hash table. With this table the element c_i that corresponds to a given index (x, y, z) can be found in $O(1)$ time. Since each cube element keeps a list with references to all triangles that intersect it, spatial queries concerning triangles can be answered in $O(1)$, too. This is important for the surface animation and surface fracturing procedures.

2.2 Mesh Animation

We use the linear Finite Element Method in connection with hexahedral elements for deformation and internal stress computations. To avoid visual artifacts in connection with large rotational deformations we apply the warped stiffness method [8] which is a corotational formulation [6] that computes forces in local rotated coordinate frames.

Since all elements have the same geometry, their stiffness matrices only depend on material properties, e.g. Young's Modulus and Poisson's ratio (see Appendix A). Thus, only one stiffness matrix *per material* rather than one per element needs to be stored which saves a substantial amount of memory for large scenes. In our implementation we use a single set of material parameters for the entire mesh and, thus, only one stiffness matrix. Even though implicit Euler integration is used, the global

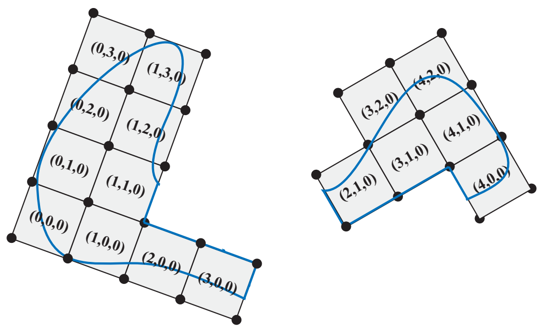

Figure 3: Connected parts of the cube mesh are animated as separate rigid bodies.

stiffness matrix of the mesh does not need to be stored explicitly either. The Conjugate Gradient solver [11] only needs to multiply the global stiffness matrix with a vector. This global multiplication is computed as the sum of multiplications with the element's stiffness matrices which is a common way to save space in connection with the finite element method.

In order to animate a small part of a scene, only a subset of vertices and cube elements are passed to the solver. The computational cost is then only dependent on the size of the active region and not on the size of the entire mesh.

2.3 Fracturing

The internal stress tensors of each cube provided by the finite element method can be used to fracture the mesh. Our fracture method is based on the ideas of Terzopoulos [15], O'Brien [10] and Müller [9]. The largest positive eigenvalue of the stress tensor represents the largest tensile stress and the corresponding eigenvector its direction. If the tensile stress exceeds a material threshold, the mesh is fractured perpendicular to the stress direction along element boundaries within a local neighborhood. We put a virtual plane through the center of the element of high stress perpendicular to the stress direction and label the elements in the neighborhood with a plus or minus sign depending on the side of the plane they lie on. Each vertex that is linked by cubes of both signs is duplicated and the cubes with a minus sign are relinked to the duplicated vertex. No other updates are necessary in connection with our FEM computations. Of course fracture operations within the cube mesh need to trigger updates in the surface mesh (see Section 3).

2.4 Rigid Body Animation

In virtual environments such as computer games, stiff and brittle materials (e.g. stone or metal) play an important role. FEM-based techniques are not very well suited to animate those types of materials in real-time since the corresponding equations get stiff and only small times steps can be taken. In case of rigid materials we animate the mesh as a series of rigid bodies based on the idea of Müller *et al.*[9]. Hereby each connected set of cubes forms its own rigid body (see Fig. 3). Only when external forces (e.g. collision forces) exceed a certain threshold, deformations are computed instantaneously and the mesh is fractured if necessary. Between those events, a rigid body simulator [16] animates the connected parts of the mesh. The fracture process can split bodies into two or more pieces, an event we detect by traversing connected sets of cubes after each fracture call.

3 Surface Animation

The way the cube mesh is generated guarantees that every surface vertex lies within a certain cube element. The deformed positions of surface vertices can, thus, be computed from the deformed positions of the eight vertices of the corresponding cube element via trilinear interpolation.

3.1 Surface Fracturing

In this section we present our algorithm to fracture the surface mesh and to generate new surface in order to keep objects watertight. We explain the process in detail because the algorithm constitutes one of the main contributions of this paper.

In Section 2.3 we described how the cube mesh is fractured when internal stresses exceed the material threshold. The basic event that needs to trigger surface mesh updates is the separation of two cubes c_1 and c_2 where their common face f gets exposed. This situation is depicted in Fig. 4. Two tasks have to be completed. First, the surface mesh needs to be fractured near face f and second, a new closing surface needs to be generated in order to keep the mesh watertight.

We do not cut any surface triangles during the fracture process. This can yield artifacts when only a few big triangles are used to represent parts of the surface. One way to solve this problem is to subdivide large triangles as a preprocessing step while the mesh is loaded (see Section 3.3). Not splitting surface triangles along f during the simulation has several advantages:

- Rendering is not slowed down by many new (sometimes tiny) triangles.
- No interpolation of positions and texture coordinates is needed which accelerates the fracture process.
- The artist can influence the fracture behavior through the design of the surface mesh.
- Since our elements are regularly shaped, they would produce regularly shaped fracture lines. Fracturing along triangle boundaries hides this regularity.

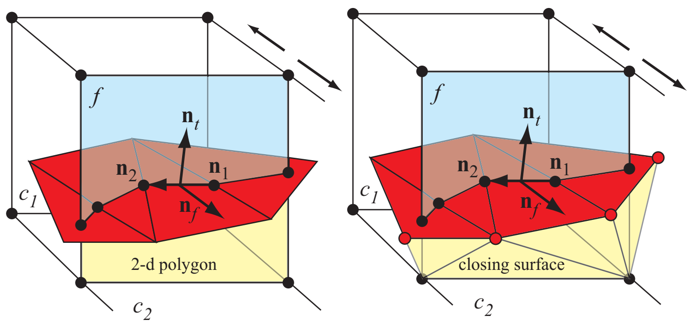

Figure 4: When two cubes c_1 and c_2 are separated along their common face f, a new closing surface needs to be generated. If the original surface was cut along f, the 2-d polygon shown on the left would be the closing surface. Since triangles are not cut, the closing surface is non planar as shown on the right.

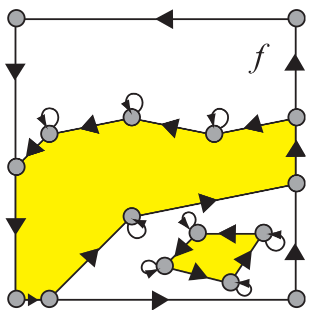

Figure 5: Front view of face f: The intersection of the surface mesh with f generates a series of oriented polygons. The positively oriented polygons surround the fraction of f that lies inside the surface mesh. Each inner intersection point is generated by two triangles resulting in two nodes at the same location both of which are needed later in the process.

In order to fracture along triangle boundaries, each of the surface triangles is assigned to the cube element its center lies in. When two cubes are separated, the surface is separated along edges that connect triangles assigned to two different cubes.

3.2 Surface Generation

The simple surface fracture process comes at a price. Closing the surface is more complex than it would be if we had cut the surface along the exposed common face f. In this case, the closing surface would be the subset of f that lies inside the surface. Now the closing surface is not planar anymore because its boundary needs to coincide with the surface fracture line (see Fig. 4). The procedure we describe in the remainder of this section can handle arbitrary fracture lines. The basic steps are:

- Compute the subset of f that lies inside the surface mesh. This 2-d region is surrounded by polygons. The polygons are cycles in a directed graph composed of nodes and directed edges in f (see Fig. 5).
- Assign each node $\mathbf{n}$ a vertex $\mathbf{v}$ of the surface mesh.
- Triangulate the polygons based on the positions of the nodes using a 2-d triangulation algorithm.
- For each triangle $(\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3)$ that is enumerated by the triangulation process, generate a surface triangle $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ using the assigned surface vertices.

We will now look at these steps in more detail: If f lies completely inside the surface the entire face becomes the closing surface. Otherwise the area of f that lies inside the surface is determined. In general this area is bounded by a set of simple non-convex polygons, possibly with holes. The line segments that form the polygonal boundaries are generated by triangles that intersect the face f (see Figs. 4 and 5).

To generate the bounding polygons, all triangles that intersect cube c_1 are tested against face f. Our data structure allows for fast enumeration of these triangles (see Section 2.1). All the geometric computations are done with original (non-deformed) coordinates. The fact that f – as a face of a cube element – is parallel to either the $x-y$, the $y-z$ or the $x-z$ plane speeds up the following computations significantly. Each triangle t that intersects face f generates two nodes $\mathbf{n}_1$ and $\mathbf{n}_2$ on f either when an edge of t intersects f or when an edge of f intersects t. A directed edge from node $\mathbf{n}_1$ to $\mathbf{n}_2$ is generated if

$$[\mathbf{n}_t \times (\mathbf{n}_2 - \mathbf{n}_1)] \cdot \mathbf{n}_f > 0 \qquad (1)$$

or from $\mathbf{n}_2$ to $\mathbf{n}_1$ otherwise, where $\mathbf{n}_t$ is the normal on triangle t and $\mathbf{n}_f$ the outward normal of face f with respect to cube c_1. This guarantees that polygons which are positively oriented with respect to $\mathbf{n}_f$ are the ones that surround part of the closing surface (see Fig. 4).

So far all intersecting triangles generate a directed edge. Because each intersecting triangle edge is adjacent to two intersecting triangles it generates two nodes, one without an outgoing edge and one with an outgoing edge. All these pairs are connected via a directed edge originating at the node without the outgoing edge.

To complete the polygons, four nodes at the corners of face f are generated. These four corner nodes and all nodes on the edges of f are now visited by walking along the boundary of f in the mathematically positive sense. Two consecutive nodes are connected by a directed edge if the first node has no outgoing edge yet (Fig. 5).

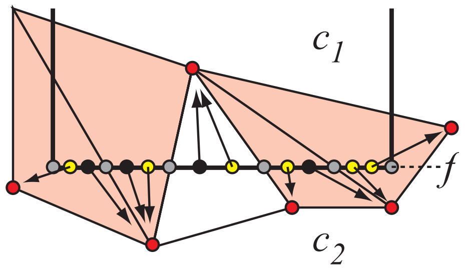

Figure 6: Top view on face f with the intersecting triangles in red (assigned to c_1) and white (assigned to other cubes): The original nodes at the intersection points (gray points) are moved towards each other (black and yellow points). Each node is associated with a vertex of the surface mesh (red points). Only one node per vertex is kept for triangulation (yellow points).

If we had cut the surface mesh along f, the polygons generated above would close the surface perfectly. To close the surface that is fractured along triangle boundaries, we assign to each node a vertex of the surface mesh. Then the original locations of the nodes on f are used to compute a 2-d triangulation of the polygons (using ear-cutting [5]) while the triangles that are generated during the triangulation are spanned between the assigned surface vertices. This is the key idea behind the algorithm. The 2-d triangulation process operates in the plane of f using the coordinates of the nodes but generates a non-planar triangulation because is outputs the assigned vertex positions instead of the nodes themselves.

Node $\mathbf{n}$ is assigned a vertex $\mathbf{v}$ of the surface mesh according to the following rules (see Fig 6):

- If $\mathbf{n}$ is generated by a triangle edge $e = (\mathbf{v}_1, \mathbf{v}_2)$ and the corresponding triangle t is assigned to cube c_1, then the vertex of e which does not lie on the c_1 side of f is selected. If triangle t is not assigned to cube c_1 then the vertex of e which lies on the c_1 side of f is selected.

- If $\mathbf{n}$ is generated by the intersection of an edge of face f with triangle $t = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$, we choose the vertex depending on the second node $\mathbf{n}'$ generated by t. If $\mathbf{n}'$ is generated by an edge $(\mathbf{v}_1, \mathbf{v}_2)$ of t, we select the third vertex $\mathbf{v}_3$ of t. If $\mathbf{n}'$ is also generated by the intersection of an edge of f with t, we choose the vertex $\mathbf{v}$ of t closest to $\mathbf{n}$.

As for regular nodes, it is also possible to assign separate coordinates to corner nodes. Here we use a random delta vector that is computed for every vertex in the cube mesh prior to the simulation. These random vectors are chosen such that they do not cross the surface mesh. This step makes sure that inner fracture faces look random, too.

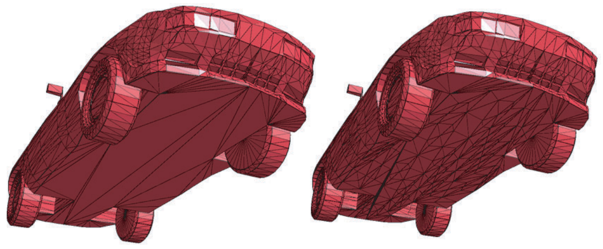

Figure 7: Planar parts of a surface are typically represented by a few large triangles. To get realistic results, the mesh is subdivided in a preprocessing step which yields evenly sized triangles.

There are three remaining issues:

First, if two adjacent triangles t_1 and t_2 generate the four nodes $\mathbf{n}_1, \mathbf{n}_2$ and $\mathbf{n}_3, \mathbf{n}_4$, the nodes $\mathbf{n}_2$ and $\mathbf{n}_3$ are generated by the common edge of t_1 and t_2 and, thus, have the same location on f (see Fig. 5). However, these two nodes might represent different surface vertices. To make sure the 2-d triangulation process generates all the necessary triangles, each pair of nodes $\mathbf{n}_1, \mathbf{n}_2$ of a triangle t is moved towards each other on f by one third of the segment length (see Fig. 6).

Second, several consecutive nodes on the connected paths that form the polygons on f may represent the same surface vertex $\mathbf{v}$. In this case, only one node is kept while all others are deleted or otherwise, zero area triangles would be generated (see Fig. 6).

Third, whenever a new triangle t is generated, a duplicate t' with opposite orientation is generated as well. Triangle t is assigned to cube c_1 while t' is assigned to c_2.

3.3 Surface Preprocessing

Planar regions of a surface mesh as the bottom of the car in Fig. 7 are usually represented by a few large triangles. For reasons of speed we do not subdivide triangles during the fracture process. In order to still get realistic results for arbitrary meshes we subdivide large triangles in a preprocessing step that guarantees that no edge is longer than the cube size h. To this end, we propose the following simple and fast algorithm that runs in $O(n \log n)$ time, where n is the number of triangles of the final mesh:

1. Generate a lexicographically sorted list of all possible triples (v_1, v_2, t) where (v_1, v_2) is an edge of a triangle with index t and $v_1 > v_2$ are vertex indices.

2. Traverse the list iteratively. If the current edge is shorter than cube size h skip the entry.

3. Otherwise there are two cases. Triple (v_1, v_2, t) is either followed by a triple (v_1, v_2, t') or not. In the first case the edge is shared by two triangles t and t' while in the second case the edge only belongs to t.

4. In both cases, a new vertex v_3 near the center of the edge is generated. Triangle t and, if present, triangle t' are each split in two new triangles containing v_3.

5. Finally the list of triples has to be updated. The edges of the newly generated triangles that contain v_3 are added at the end of the list. If the new entries are added in lexicographical order at the end of the list, the entire list remains sorted, because v_3 is the largest vertex index of the mesh.

6. The edges of the newly generated triangles that do no contain v_3 are already in the list but need to be updated since they contain the old triangles t and t'. They can be found in logarithmic time via a binary search on the sorted list.

7. Skip all entries containing v_1 and v_2 and go to step 2.

4 Results

All the animations described in this section were computed and rendered in real time on a 1.8 GHz Pentium IV PC with a GForce 4 graphics card. Fig. 8 shows an eagle surface mesh composed of $23,000$ triangles for which a cube mesh of about 800 elements was generated. The object deforms elastically under gravity and user applied forces at $10 - 15$ frames per second.

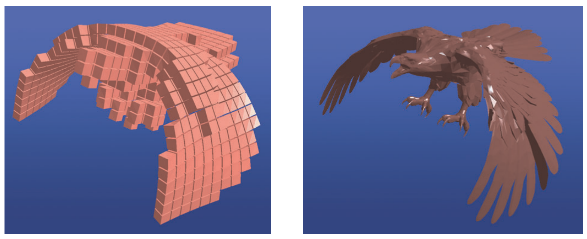

Figure 8: Eagle surface mesh (right) for which a cube mesh was generated (left) to simulate elastic deformation.

The pig shown in Fig. 9 is animated as a rigid body. When it hits the ground a static Finite Element step is computed which yields the internal stresses used to fracture the model. New rigid bodies are generated while the surfaces are closed dynamically. The simulation runs at over 40 frames per second.

The examples in Figures 1 and 10 show that the fracture procedure works robustly for more complex geometries as well (see also accompanying video sequences).

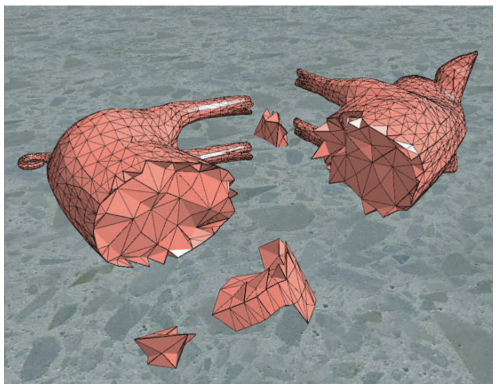

Figure 9: A pig model fractures as it hits the ground. New surface triangles are generated dynamically to keep the surface mesh closed.

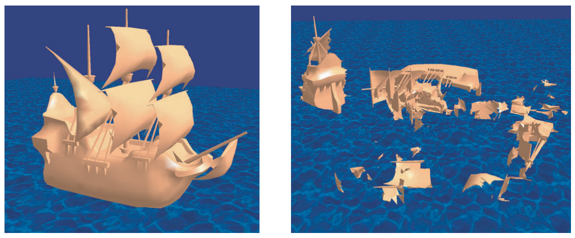

Figure 10: A more complex model shows the robustness of the fracture procedure.

5 Conclusions and Future Work

We have presented a method for the physically-based animation of scenes that are defined by their surfaces only. Objects are voxelized and the resulting regular hexahedral meshes are animated using Finite Element or rigid body simulations. The associated surface mesh is deformed and fractured according to the simulation performed on the underlying volumetric mesh.

At the time we only use one material per cube mesh even though the simulation can handle multiple materials. With a separate tool or process different materials could be applied to different parts of the cube mesh.

Another way to improve our method could be to use cube elements of more than one size. Instead of using cubes of uniform size h, a structure similar to an octtree could be used to reduce the number of elements or to subdivide the mesh in regions of high stress. Since all elements would still be cubes, properly scaled versions of the same stiffness matrix could be used.

Sometimes independent small features of the surface mesh are associated with the same cube element. Subdivision of elements could solve this problem as well.

6 Acknowledgements

This project was funded by the Swiss National Commission for Technology and Innovation (KTI) project no. 6310.1 KTS-ET.

References

[1] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34. ACM Press, 1994.

[2] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popovic. Interactive skeleton-driven dynamic deformations. In *Proceedings of SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 586–593. ACM, ACM Press / ACM SIGGRAPH, 2002.

[3] R. D. Cook. *Finite Element Modeling for Stress Analysis*. John Wiley & Sons, NY, 1995.

[4] G. Debunne, M. Desbrun, M. P. Cani, and A. H. Barr. Dynamic real-time deformations using space & time adaptive sampling. In *Computer Graphics Proceedings*, Annual Conference Series, pages 31–36. ACM SIGGRAPH 2001, August 2001.

[5] H. ElGindy, H. Everett, and G.T. Toussaint. Slicing an ear in linear time. *Pattern Recognition Letters*, pages 719–722, 1993.

[6] C. A. Felippa. A systematic approach to the element-independent corotational dynamics of finite elements. Technical Report Report CU-CAS-00-03, University of Colorado, 2000.

[7] D. James and D. K. Pai. Artdefo, accurate real time deformable objects. In *Computer Graphics Proceedings*, Annual Conference Series, pages 65–72. ACM SIGGRAPH 99, August 1999.

[8] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler. Stable real-time deformations. *Proceedings of 2002 ACM SIGGRAPH Symposium on Computer Animation*, pages 49–54, 2002.

[9] M. Müller, L. McMillan, J. Dorsey, and R. Jagnow. Real-time simulation of deformation and fracture of stiff materials. *EUROGRAPHICS 2001 Computer Animation and Simulation Workshop*, pages 27–34, 2001.

[10] J. F. O'Brien and J. K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of SIGGRAPH 1999*, Computer Graphics Proceedings, Annual Conference Series, pages 287–296. ACM, ACM Press / ACM SIGGRAPH, 1999.

[11] C. Pozrikidis. *Numerical Computation in Science and Engineering*. Oxford Univ. Press, NY, 1998.

[12] C. Shen, K.K. Hauser, C.M. Gatchalian, and J.F. O'Brien. Modal analysis for real-time viscoelastic deformation. *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, 2002.

[13] J. Smith, A. Witkin, and D. Baraff. Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Interface*, pages 27–34, May 2000.

[14] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[15] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 269–278. ACM Press, 1988.

[16] A. Witkin and D. Baraff. Physically based modeling: Principles and practice. *Siggraph Course Notes*, August 1997.

[17] X. Wu, M. S. Downes, T. Goktekin, and F. Tendick. Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes. *Eurographics*, pages 349–358, September 2001.

A Appendix Stiffness Matrix of a Cube Element

To ease the implementation of our method, we give the entries of the stiffness matrix for a cube element explicitly. The stiffness matrix of a finite element linearly relates the displacement vectors at the element's nodes to the force vectors they cause. In the case of a cube element, the stiffness matrix $\mathbf{K}$ relates eight three dimensional displacement vectors to eight three dimensional force vectors and is, thus, 24×24 dimensional. Once $\mathbf{K}$ is known, the forces can simply be computed as

$$\mathbf{f} = \mathbf{K} \cdot \Delta\mathbf{x}, \tag{2}$$

where $\mathbf{f} \in \mathbf{R}^{24}$ contains the eight force vectors at the corners and $\Delta\mathbf{x} \in \mathbf{R}^{24}$ the displacement vectors.

We compute the stiffness matrix for a cube analogous to the derivation of the stiffness matrix for constant strain triangles in [3] pp 46–47. First, we consider a cube centered at the origin with edge length h and $r = h/2$ (see Fig. 11).

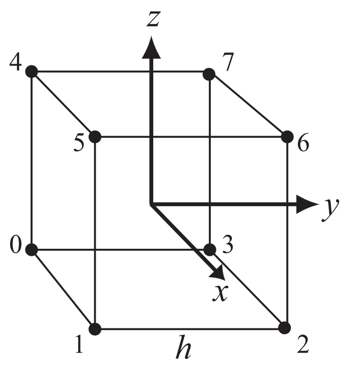

Figure 11: A cube element of size h centered at the origin.

The following shape functions

$$\begin{aligned}\phi_0 &= (r-x)(r-y)(r-z)/h^3\\ \phi_1 &= (r+x)(r-y)(r-z)/h^3\\ \phi_2 &= (r+x)(r+y)(r-z)/h^3\\ \phi_3 &= (r-x)(r+y)(r-z)/h^3\\ \phi_4 &= (r-x)(r-y)(r+z)/h^3\\ \phi_5 &= (r+x)(r-y)(r+z)/h^3\\ \phi_6 &= (r+x)(r+y)(r+z)/h^3\\ \phi_7 &= (r-x)(r+y)(r+z)/h^3\end{aligned}$$

have the property that $\phi_i = 1$ at corner i and zero at all other corners. Using these basis functions, Cauchy's linear strain tensor and assuming an isotropic Hookean material, we get a constant stiffness matrix $\mathbf{K}$. For this regular element, although the shape functions are non-linear, the integrals for the coefficients of $\mathbf{K}$ can be solved analytically and turn out to be simple expressions. We use the following auxiliary variables

$$\begin{aligned}a = &\quad h \cdot E \cdot \tfrac{1-\nu}{(1+\nu)(1-2\nu)}\\ b = &\quad h \cdot E \cdot \tfrac{\nu}{(1+\nu)(1-2\nu)}\\ c = &\quad h \cdot E \cdot \tfrac{1}{2(1+\nu)},\end{aligned}$$

where E is Young's Modulus and ν the Poisson ratio of the material to be modeled [3]. The stiffness matrix $\mathbf{K}$ can naturally be devided into 3×3-dimensional sub matrices K_{ij} with $i, j \in [0..7]$.

For the sub matrices corresponding to the nodes of the cube K_{00}, K_{11}, K_{22}, K_{33}, K_{44}, K_{55}, K_{66}, K_{77}, we get

$$\begin{bmatrix} d & o & o\\ o & d & o\\ o & o & d\end{bmatrix}, \begin{bmatrix} d & n & n\\ n & d & o\\ n & o & d\end{bmatrix}, \begin{bmatrix} d & o & n\\ o & d & n\\ n & n & d\end{bmatrix}, \begin{bmatrix} d & n & o\\ n & d & n\\ o & n & d\end{bmatrix},$$
$$\begin{bmatrix} d & o & n\\ o & d & n\\ n & n & d\end{bmatrix}, \begin{bmatrix} d & n & o\\ n & d & n\\ o & n & d\end{bmatrix}, \begin{bmatrix} d & o & o\\ o & d & o\\ o & o & d\end{bmatrix}, \begin{bmatrix} d & n & n\\ n & d & o\\ n & o & d\end{bmatrix},$$

where $d = \frac{a+2c}{9}$, $o = \frac{b+c}{12}$ and $n = -o$.

The submatrices corresponding to edges in x-direction $K_{01}, K_{23}, K_{45}, K_{67}$, y-direction $K_{03}, K_{12}, K_{47}, K_{56}$ and in z-direction $K_{04}, K_{15}, K_{26}, K_{37}$ are

$$\begin{bmatrix} d_1 & o_1 & o_1\\ n_1 & d_2 & o_2\\ n_1 & o_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & o_1 & n_1\\ n_1 & d_2 & n_2\\ o_1 & n_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & o_1 & n_1\\ n_1 & d_2 & n_2\\ o_1 & n_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & o_1 & o_1\\ n_1 & d_2 & o_2\\ n_1 & o_2 & d_2\end{bmatrix},$$
$$\begin{bmatrix} d_2 & n_1 & o_2\\ o_1 & d_1 & o_1\\ o_2 & n_1 & d_2\end{bmatrix}, \begin{bmatrix} d_2 & o_1 & n_2\\ n_1 & d_1 & o_1\\ n_2 & n_1 & d_2\end{bmatrix}, \begin{bmatrix} d_2 & n_1 & n_2\\ o_1 & d_1 & n_1\\ n_2 & o_1 & d_2\end{bmatrix}, \begin{bmatrix} d_2 & o_1 & o_2\\ n_1 & d_1 & n_1\\ o_2 & o_1 & d_2\end{bmatrix},$$
$$\begin{bmatrix} d_2 & o_2 & n_1\\ o_2 & d_2 & n_1\\ o_1 & o_1 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & n_2 & o_1\\ n_2 & d_2 & n_1\\ n_1 & o_1 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & o_2 & o_1\\ o_2 & d_2 & o_1\\ n_1 & n_1 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & n_2 & n_1\\ n_2 & d_2 & o_1\\ o_1 & n_1 & d_1\end{bmatrix},$$

where $d_1 = \frac{-a+c}{9}$, $d_2 = \frac{a-c}{18}$, $o_1 = \frac{b-c}{12}$, $o_2 = \frac{b+c}{24}$, $n_1 = -o_1$ and $n_2 = -o_2$.

The submatrices corresponding to face diagonals K_{05}, K_{14}, K_{16}, K_{25}, K_{27}, K_{36}, K_{07}, K_{34}, K_{02}, K_{13}, K_{46} and K_{57} are

$$\begin{bmatrix} d_1 & o_2 & n_1\\ n_2 & d_2 & n_2\\ n_1 & o_2 & d_1\end{bmatrix}, \begin{bmatrix} d_1 & n_2 & o_1\\ o_2 & d_2 & n_2\\ o_1 & o_2 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & o_2 & o_2\\ n_2 & d_1 & n_1\\ n_2 & n_1 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & n_2 & o_2\\ o_2 & d_1 & o_1\\ n_2 & o_1 & d_1\end{bmatrix},$$
$$\begin{bmatrix} d_1 & o_2 & o_1\\ n_2 & d_2 & o_2\\ o_1 & n_2 & d_1\end{bmatrix}, \begin{bmatrix} d_1 & n_2 & n_1\\ o_2 & d_2 & o_2\\ n_1 & n_2 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & n_2 & n_2\\ o_2 & d_1 & n_1\\ o_2 & n_1 & d_1\end{bmatrix}, \begin{bmatrix} d_2 & o_2 & n_2\\ n_2 & d_1 & o_1\\ o_2 & o_1 & d_1\end{bmatrix},$$
$$\begin{bmatrix} d_1 & n_1 & o_2\\ n_1 & d_1 & o_2\\ n_2 & n_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & o_1 & n_2\\ o_1 & d_1 & o_2\\ o_2 & n_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & n_1 & n_2\\ n_1 & d_1 & n_2\\ o_2 & o_2 & d_2\end{bmatrix}, \begin{bmatrix} d_1 & o_1 & o_2\\ o_1 & d_1 & n_2\\ n_2 & o_2 & d_2\end{bmatrix},$$

where in this case $d_1 = \frac{-2a-c}{36}$, $d_2 = \frac{a-4c}{36}$, $o_1 = \frac{b+c}{12}$, $o_2 = \frac{b-c}{24}$, $n_1 = -o_1$ and $n_2 = -o_2$.

Finally, the submatrices corresponding to cube diagonals K_{06}, K_{17}, K_{24} and K_{35} are

$$\begin{bmatrix} d & n & n\\ n & d & n\\ n & n & d\end{bmatrix}, \begin{bmatrix} d & o & o\\ o & d & n\\ o & n & d\end{bmatrix}, \begin{bmatrix} d & n & o\\ n & d & o\\ o & o & d\end{bmatrix}, \begin{bmatrix} d & o & n\\ o & d & o\\ n & o & d\end{bmatrix},$$

where $d = \frac{-a-2c}{36}$, $o = \frac{b+c}{24}$ and $n = -o$. Since $\mathbf{K}$ is symmetric, the remaining sub matrices can be computed as $\mathbf{K}_{ij} = \mathbf{K}_{ji}^T$.