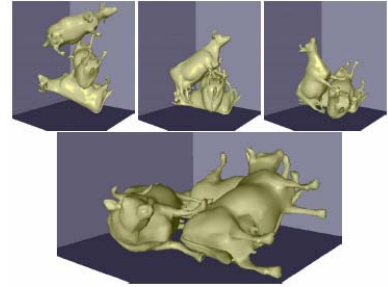


A Versatile and Robust Model for Geometrically Complex Deformable Solids

Matthias Teschner, Bruno Heidelberger, Matthias Muller,
Markus Gross

Summary by Dan Morris



This paper describes the authors' system for simulating deformable materials. It is largely an "architecture" paper; the individual components are discussed in other publications. This summary will highlight the major points, focusing on the tricks they used to really nail the state of the art in interactive deformation simulation.

Here's the super-short outline:

- Tetrahedral meshes for simulation, bound to triangle meshes for rendering
- Local force computation based on potential energies
- Explicit Verlet integration
- Super-clever collision detection (based on spatial hashing) and collision response (based on a novel penetration-depth-estimation scheme) that allows this to function at interactive speeds. These features are summarized here, but are actually described in two related papers.

-
- The meshes they *simulate* are tetrahedral meshes; a pre-processing step takes whatever objects you want to simulate and fits a tetrahedral mesh to it.
 - Their mesh-generation approach is described in:
<http://cg.informatik.uni-freiburg.de/publications/volumetricMeshesBVM2003.pdf>
 - They do discuss generating thin tetrahedral meshes to represent thin sheets of triangles (section 4). Basically they're pointing out that the tet-mesh approach is a superset of the triangle-mesh approach, so they can handle thin sheets pretty well.
 - The meshes they *render* are triangular meshes; another pre-processing step fits a surface mesh to the outer surface of the tetrahedral mesh. As the object deforms, the surface mesh is rendered via skinning (remember how we learned about skinning when we read the EigenSkin paper?)
 - This, incidentally, is the only topology-dependent component of their system, and the only reason they don't have topologically-changing demos
 - Like most mass-point-based simulation techniques, they compute a force on each element, then eventually turn those forces into accelerations. They formulate their forces as potential energies, under the typical model in which the universe generates forces to minimize potential energies. So each "force" in their system is presented as a function that "wants" to be zero.

- To derive actual forces, they take the spatial partial derivatives of the energies (eq 2). They don't actually *show* these derivatives in the paper, but they say they have *analytical* formulas for the derivative of each force at each mass point. Actually, eqs 1 and 2 come up a lot in simulation papers, so I'll include them here... eq 1 represents energy E in terms of a constraint C (something the universe wants to make zero), and eq 2 represents force in terms of energy:

$$E(\mathbf{p}_0, \dots, \mathbf{p}_{n-1}) = \frac{1}{2}kC^2 \quad (1)$$

$$\mathbf{F}^i(\mathbf{p}_0, \dots, \mathbf{p}_{n-1}) = -\frac{\partial}{\partial \mathbf{p}_i} E = -kC \frac{\partial C}{\partial \mathbf{p}_i} \quad (2)$$

- They present three forces:
 - Distance preservation (maintains distance between mass points)
 - Surface area preservation (maintains surface area of tetrahedral faces)
 - Volume preservation (maintains volume of tetrahedral
- Each of these forces is associated with a gain (a spring constant). These constants are the values you can tweak to make different materials behave differently. For example, an object with a high volume preservation constant but a low distance preservation constant will behave like a fluid-filled balloon.
 - These spring constants are not easily translated into real physical constants (e.g. Poisson's ratio, bulk modulus, etc.), which is a significant limitation of this approach relative to a continuum-mechanics-based approach.
 - There is no special reason these constants can't vary within an object, although they haven't yet modeled any objects with varying material properties.
- They use an explicit Verlet integration scheme to get from forces to new positions at each timestep. This is a popular integration scheme for physical simulation... in fact it's another useful take-away message from this paper, so I'll include the equations here:

$$\begin{aligned} \mathbf{x}(t+h) &= 2\mathbf{x}(t) - \mathbf{x}(t-h) + h^2 \frac{\mathbf{F}(t)}{m} + O(h^4) \\ \mathbf{v}(t+h) &= \frac{\mathbf{x}(t+h) - \mathbf{x}(t-h)}{2h} + O(h^2) \end{aligned} \quad (7)$$

- The major benefits of this approach are:
 - Only one force computation per time step (none of that $F(t+h/2)$ business)
 - $O(h^4)$ error
 - You *can* compute positions *without* computing velocities, which means a lot less computation. In practice, they do compute velocities so they can apply some damping.

- For collision detection, they use a novel scheme that tests whether each mass point in the scene is inside each tetrahedra in the scene. This is described in detail in :

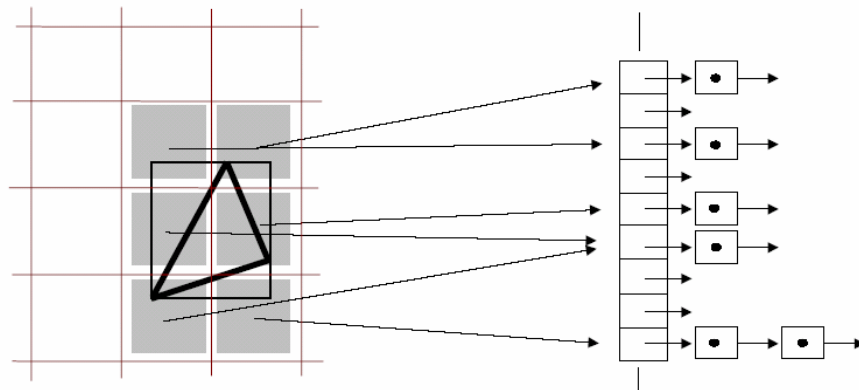
<http://cg.informatik.uni-freiburg.de/publications/collisionDetectionHashingVMV2003.pdf>

The basic summary is:

- Divide the world into grid cells
- At each time step:
 - Hash all vertices (mass points) into a spatial hash table, which is basically a list of all the grid cells that have vertices in them, where each element contains a list of all the vertices that live in that grid cell
 - For each tetrahedron in your scene:
 - Find all the grid cells in the AABB of the tet
 - See if each grid cell has an entry in the hash table
 - If it does, check for collisions

This is simple and fast. Very clever.

This is my favorite figure from this paper, showing the grid, a single tet's AABB, and the spatial hash table:



- For collision response, they use an equally novel scheme that actually computes *good* penetration depth values. Most approaches have problems when a mass point that's immersed in another object gets too close to a surface that *isn't* the surface it entered through. Their solution is described in detail in :

<http://cg.informatik.uni-freiburg.de/publications/penetrationDepthVMV2004.pdf>

The basic summary is:

- Find all points on each object that live inside another object (using the method described above)
- Find all “colliding” points on each object that are *adjacent to* “non-colliding” points

- The edges between these “border points” are the intersecting edges. When you know which edges are intersecting and the surface normals at those edges, you can compute the effective penetration direction and apply a consistent response force to all immersed points.

This is also simple and fast. And very clever.

This is my favorite figure from this paper, showing two colliding objects, the intersecting edges, and the resulting response forces:

