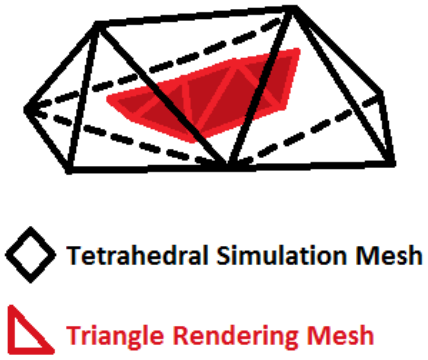
Real-time Deformation and Fracture in a Game Environment

Reference paper for the lecture:

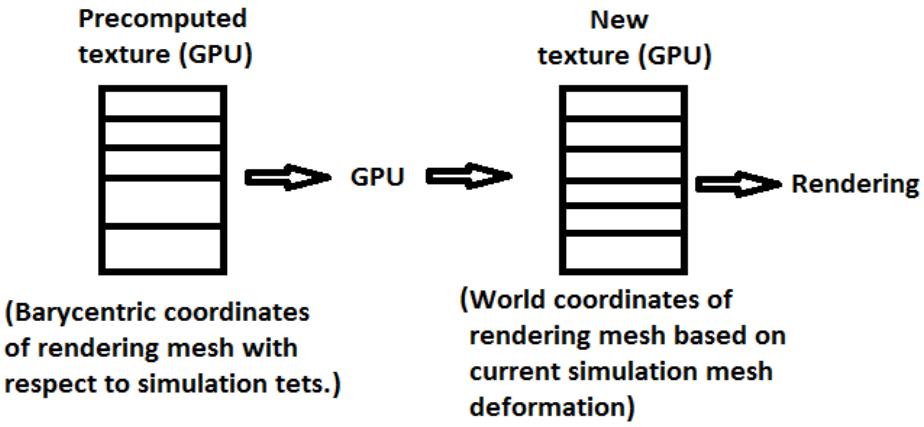
> E. G. Parker and J. F. O'Brien: Real-Time Deformation and Fracture in a Game Environment, ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2009.

Real-time Deformation

Start with two meshes: a simulation mesh (tetrahedral mesh) and a rendering mesh (polygonal mesh). The tetrahedral mesh should completely surround the rendering mesh:



The undeformed state of the rendering mesh with respect to the simulation mesh is precomputed and stored in a texture on the GPU. During a time step, the deformation of the simulation mesh is used along with the undeformed texture data to find the world coordinates of the rendering mesh:



Let us define an undeformed tetrahedron to have points X_0, X_1, X_2, and X_3, and the deformed tetrahedron to have points x_0, x_1, x_2, and x_3. Then the deformation gradient matrix F is given below, which transforms the vector $<X_0 – X_N>$ to $<x_0 – x_N>$ where X_N/x_N are points in the deformed/undeformed tetrahedra:

$$F_{3x3} = \left[x_1 - x_0, x_2 - x_0, x_3 - x_0\right]\left[X_1 - X_0, X_2 - X_0, X_3 - X_0\right]^{-1}$$

Now we can define strain:

$$\epsilon = \frac{1}{2}(F^T F - I) \qquad \text{(Green-Lagrange strain)}$$

This formulation is good, but leads to non-linear ODE's, which are harder to solve in real time

$$\epsilon = \frac{1}{2}(F^T + F) - I \qquad \text{(Linear Cauchy strain)}$$

This is only an approximation to strain, but leads to linear ODE's!

And define stress:

$$\sigma = \lambda (tr\,\epsilon) \cdot I + 2\mu\,\epsilon \qquad \text{(Cauchy stress)} \qquad \textit{(stress converts displacement to forces)}$$

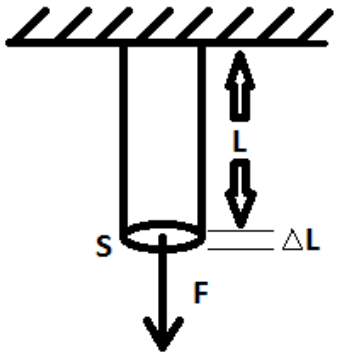
Here λ and μ are Lamé parameters given by:

$$\lambda = \frac{\nu \cdot E_0 (1 - 2\nu)}{1 + \nu} \qquad \mu = \frac{E_0}{2(1+\nu)}$$

E_0 and ν are known as Young's modulus and Poisson's ratio and have recorded values for many materials. Young's modulus describes a material's response to linear strain and is computed by:

$$E_0 \cdot \frac{\Delta L}{L} = \frac{F}{S}$$

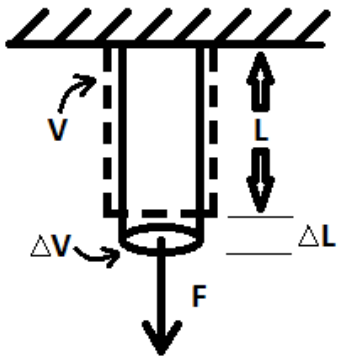
where L is the rod length, F is the force, and S is the surface area



Poisson's ratio is a measure of how much a material expands or contracts transversely when strained in some direction:

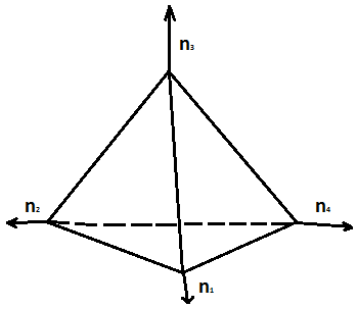
$$\frac{\Delta V}{V} = (1 - 2\nu) \cdot \frac{\Delta L}{L}$$

where V is the original volume, F is the force, and L is the original length



Now, let's go back to the simulation mesh. The elastic force applied by the tetrahedral element on vertex i is given by the following formula, provided that the normals are of a length equal to the area of the triangle opposite the vertex in the tetrahedron:

$$F_i = \sigma \cdot n_i$$



Cauchy stress works well for small deformations, but is poor for large deformations. We want to use it anyway because it is easy to compute from F, scales linearly with deformation, and has other nice properties, so we "fix" the method by using Polar Decomposition:

$$F_{3x3} = Q_{3x3} \cdot S_{3x3}$$

Q is a rotation matrix that represents the rotation closest to the transformation matrix F and S is a symmetric matrix. The idea here is to use Q^{-1} (also Q^T) to unrotate the deformed tetrahedron, then compute the forces in "undeformed" space, then rotate the force back to "deformed" space.

Plastic Deformation

Maintain a breakdown of strain into plastic strain and elastic strain for each tetrahedron in the simulation mesh.

$$\epsilon = \epsilon^P + \epsilon^E$$

Here, the plastic deformation term represents the "base" strain and should start at zero when the simulation begins. Each time step, compute the elastic deformation as usual using ε^E, and if the deformation is big enough, then store most of it in the plastic deformation term. More formally:

$$\text{if } \|\epsilon^E\| > y \text{ then } \epsilon^P := \epsilon^P + \frac{\|\epsilon^E\| - y}{\|\epsilon^E\|} \cdot c \cdot \epsilon^E$$

where "y" and "c" are the plastic yield and plastic creep constants

Real-time Fracture

See the notes from the previous lecture on fracture; the real-time method is much the same. The trick to making it run faster and still look good is to avoid re-meshing where a fracture occurs and simply fracture along the tetrahedron's edges. Normally, it would look bad if the rendering mesh fractured along the simulation mesh's edges, but by adding another level of indirection called "splinters", the problem is fixed.

Let a “splinter” be a single “piece” of the rendering mesh and build the rendering mesh from these pieces. For example, a brick wall would be built from brick “splinters”. Each “splinter” has an owner tetrahedron in the simulation mesh according to where the splinter's center point lies. If a fracture would separate the center of a “splinter”from its owner tetrahedron, then make the “splinter” part of the fracturing tetrahedron. This creates the illusion that the rendering mesh is fracturing in a more realistic way: