Fracture Modeling

Siddhartha Saha
CSE291 Project Report, Fall 2005, UCSD

June 10, 2005

Abstract

In this project, I have explored several published methods which try to do realistically model the fracture of brittle materials. In addition to that, I have also implemented a fracture modeling framework which simulates fracture of real world brittle objects by closely following the work of Brian et al [[**?**]].

Even though there were several difficulties in implementing the fracture simulator given the complexity of the undertaken work and the short time frame that I had - the final results looks quite good. All the images in this report are generated by using simple OpenGL primitives - as the focus was on modeling rather than the rendering part. I did not have sufficient time to render the resulting images using a ray tracer - that would have definitely improved the quality of the renderings.

1 Introduction

Using simulation to model material deformation is not a new thing - it is the primary method to predict failures in structural engineerings. For the last few years, it has also been used in computer graphics a lot. After reading a few approaches to modeling the fracture of a brittle material it seemed that the most promising approach is the one that is used by O'Brien et al [[**?**]]. So, I have tried to follow that approach in the implementation of this fracture simulation system.

The next section describes the methodology of the implementation - sub-categorized into several important sub-problems.

2 Methodology

In my project, I implemented the work by O'Brien et al [[**?**]][[**?**]]. The physical modeling and the deformation/fracture part was easy to follow in both the paper and the thesis. Even though the implementation sounded simple enough after going through the literature a couple of times, the main challenge which I faced was to handle cases where the system had very high chance of being unstable. I will briefly describe the techniques that I have used myself to handle those problems in subsequent sections.

2.1 Modeling the System

The modeling part was most simple. This simulation works on tetrahedral models. I used the freely available NETGEN Package to create some simple tetrahedral models - like a cube, a plane and a bowl. I wrote my own data structures to store the mesh, to handle the splitting and local re meshing and rendering the system in a simple OpenGL viewer. The simulator would periodically (typically 1 ms apart) store the snapshot of the state of the system - and this made it easy to resume the simulation from any point. This helped a lot - as for very stiff spring constants, the simulations usually would take several hours. Later in the project, I have also used these snapshots to render the pre-computed/simulated system states to create an animation-like demo system.

2.2 Deformation Model Formula

The physical equations describing the deformation states of an object assumes a continuous material and is based on continuum mechanics. The primary assumption in the continuum approach is that the scale of the effects being modeled is significantly greater than the scale of the materials composition.

Let $\vec{u} = [u, v, w]^T$ be a vector in R^3 that denotes a a location in the material coordinate frame. The deformation of the material is defined by the function $\vec{x}(\vec{u}) = [x, y, z]^T$ that maps locations in the material coordinate frame to locations in world coordinates. The local deformation of the material is given by Green's strain tensor:

$$\epsilon_{i,j} = \left(\frac{\delta \vec{x}}{\delta u_i} . \frac{\delta \vec{x}}{\delta u_j} \right) - \delta_{i,j} \tag{1}$$

In addition to the Strain tensor, we also use the strain rate tensor ν which measures the rate at which the strain is changing. It is defined by taking the time derivative of 1

$$\nu_{i,j} = \left(\frac{\delta \vec{x}}{\delta u_i} . \frac{\delta \dot{\vec{x}}}{\delta u_j} \right) + \left(\frac{\delta \dot{\vec{x}}}{\delta u_i} . \frac{\delta \vec{x}}{\delta u_j} \right) \tag{2}$$

The stress tensors σ that this model uses is derived making the assumption that the material is isotropic and is given by:

$$\sigma^e_{i,j} = \sum_{k=1}^{3} \lambda \epsilon_{k,k} \delta_{i,j} + 2\mu \epsilon_{i,j} \tag{3}$$

$$\sigma^v_{i,j} = \sum_{k=1}^{3} \phi \nu_{k,k} \delta_{i,j} + 2\psi \nu_{i,j} \tag{4}$$

The material rigidity ie determined by the value of μ and the resistance to changes in volume is controlled by λ. The parameters ϕ and ψ will control how quickly the material dissipates internal kinetic energy.

We need to compute the stress tensors to compute the deformation force. Given the tetrahedral model, and the linear shape functions, computing is them is pretty straightforward. They are described very well in

O'Brien's paper[[**?**]]. So there is no point in my reproducing them here once again. I will just rewrite the equations for completeness.

We define the quantity β for each tetrahedral element.

$$\beta = \begin{bmatrix} m_1, & m_2, & m_3, & m_4 \\ 1, & 1, & 1, & 1 \end{bmatrix}^{-1} \tag{5}$$

where, m_i are the material coordinates of the tetrahedron.

We compute $\frac{\delta \vec{x}}{\delta u_i}$ and $\frac{\delta \vec{x}}{\delta u_i}$ by

$$\frac{\delta \vec{x}}{\delta u_i} = P\beta\delta_i \tag{6}$$

$$\frac{\delta \vec{\dot{x}}}{\delta u_i} = V\beta\delta_i \tag{7}$$

where,

$$P = [p_1, p_2, p_3, p_4] \tag{8}$$

$$V = [v_1, v_2, v_3, v_4] \tag{9}$$

$$\delta_i = [\delta_{i1}, \delta_{i2}, \delta_{i3}, 0]^T \tag{10}$$

Finally, the force at a node is given by:

$$\vec{f}^{\epsilon}_{[i]} = \frac{\text{vol}}{2} \sum_{j=1}^{4} \vec{p}_j \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl}\beta_{ik}\sigma_{kl} \tag{11}$$

2.3 Collision Detection

The next important milestone was to get collision detection correctly. The collision detection system had to be numerically very stable - as it not only had to detect the collision between two tetrahedrons, or a tetrahedrons and a plane, - but it also had to compute the overlapped convex polyhedron that is formed by the collision. Computing this polyhedron in a stable manner proposed a bit problem. To get a proper realistic simulation, self collision detection is important. And often after a crack two adjacent tetrahedrons will collide in such a way that one or more vertices of one tetrahedron would lie on, or will be very near to, some face of the other tetrahedron. Problems would also occur when one edge of a tetrahedron intersects (or lies very close to) another edge of the other tetrahedron. After quite a bit of work, I got the collision detection system working in a reasonably stable manner.

The overall approach that I took to compute the polyhedron is that for each face of the colliding tetrahedrons, I check if that face will give rise to a face in the resulting polyhedron. If yes, then I collect the points on that tetrahedral face that would make a face in the polyhedron (making sure that I do not take duplicate or very near points - which would arise if we have an edge-edge intersection). Once I have collected the points that

would make a face in the polyhedron, I orient the points in a counter-clockwise manner - which is needed to make the volume/center of mass calculations correct.

After I computed the collision polyhedron, I needed to compute the forces that should be applied to the nodes such that the linear and angular momentum is conserved. I followed O'Brien's method for that - and it works pretty well.

2.4 Speeding Up Collision Detection

This is one part that *must* be implemented in order to make the system capable of handling large simulations - as the collision detection can quickly become the bottleneck of the system. I did not have time to implement any acceleration structure for collisions. All I did was that I use a list in which I store potentially colliding pairs of elements based on the current position and velocity. I traverse this list to detect collisions. This list is computed periodically. This simple technique reduced the collision detection bottleneck to some extent, but this is by no means complete. A hierarchical bounding box based system is described in [[**?**]]. Also, there are many more methods to do efficient collision detection pair selection algorithms in literature.

2.5 Splitting a Node and Local Re-meshing

This is also a problem area. A lot of time and effort has gone into making this work properly. The core of the algorithm is not complex - that is rather simple. Complexity increases many folds when one wants to employ some threshold to avoid ill-conditioned tetrahedron creation. This is essentially done by checking if any point lies close to the splitting plane - and then snapping the plane to the existing points. Finally I managed to do that.

One more thing to take care is to make sure that whenever a crack reaches the boundary of the material, the discontinuity should pass through the boundary and crate a crack. O'Brien's thesis [[**?**]] just touches on this. But with the snapping vertex to the plane going on, it is somewhat involved task to find when a crack reaches a boundary of the material. I did not get this right for many days - and I would get artifacts where two tetrahedrons will be connected by only a single node - and will dangle. This does not happen in practice. So, I had to include some checks to detect this kind of effects.

One thing that was confusing me for some time is that sometimes a splitting plane as calculated from the *separation tensor* [[**?**]] will not split the node - i.e. all the tetrahedrons connected to the node lie on the same side of the plane. At first I thought I did something wrong - but later, carefully examining the separation tensor and a few problem instances side by side - I realized this may indeed happen. The separation tensor methods works only by calculating the unbalanced tensile and compressive

forces - and this method can give rise to this spurious splits at some nodes which are in a convex corner of the object.

3 Shortcomings

There are few shortcomings in my implementation. The collision detection algorithm is slow - as I did not have time to implement a acceleration structure. The only acceleration structure (kind of) that I have used is a collision list - which stores the pairs of primitives which may potentially collide (based on a bounding box test). Moreover, this entire simulation runs based on explicit integration - which severely limits the stability and scalability of the system. Hence I am forced to run it with extremely small time-step (one or less microseconds). This constraint on time step and collision made it difficult to run the simulation for large models.

4 Results

The results looks moderately good. The meshes that I used were very coarse - and hence sometime the split runs in a long straight line. But that is not a problem with the simulator - just a bad model They are all rendered in OpenGL. It would have been much better if these were rendered using a ray tracer. I might do that sometime later in my own time. Here are some screen shots from some demo run of the system.

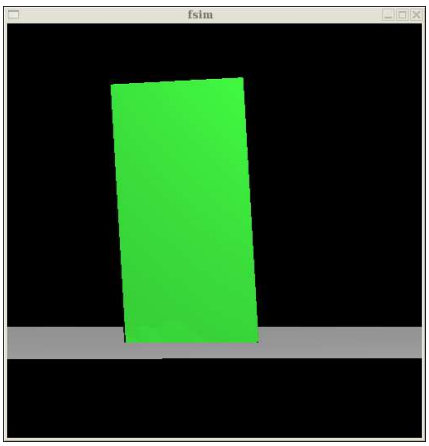

Figure 1: Breaking of a plate. [1]

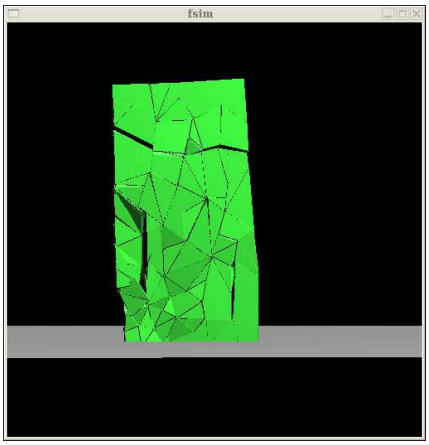

Figure 2: Breaking of a plate. [2]

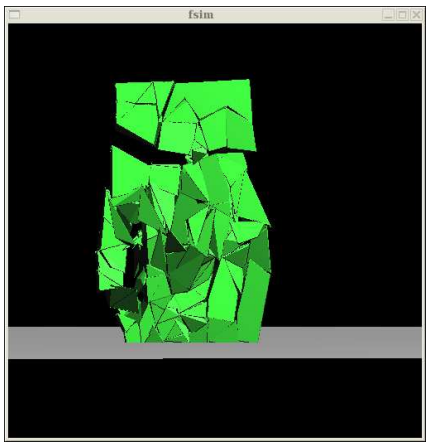

Figure 3: Breaking of a plate. [3]

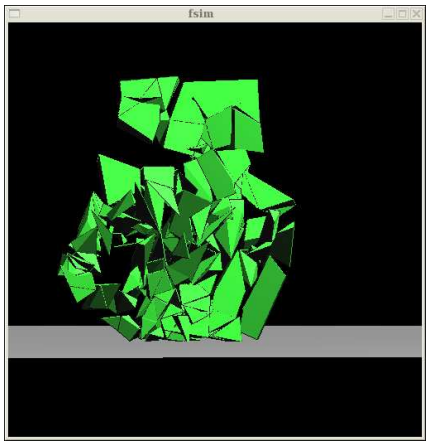

Figure 4: Breaking of a plate. [4]

Figure 5: Breaking of a plate. [1]

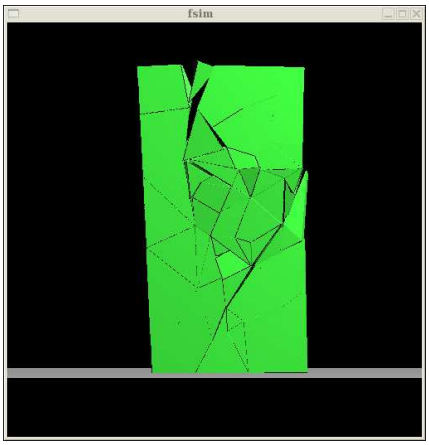

Figure 6: Breaking of a plate. [2]

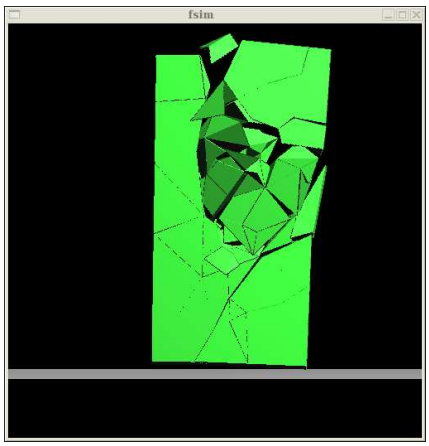

Figure 7: Breaking of a plate. [3]

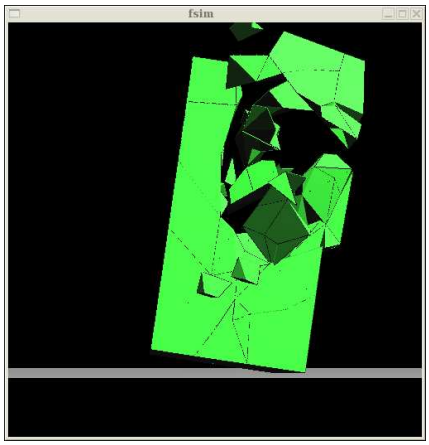

Figure 8: Breaking of a plate. [4]

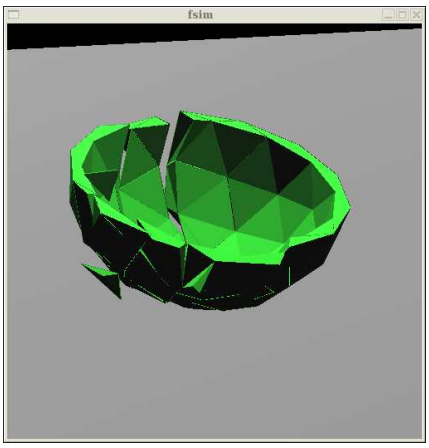

Figure 9: A bowl

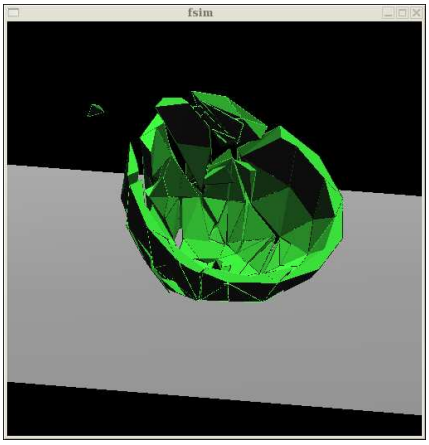

Figure 10: A fragile bowl

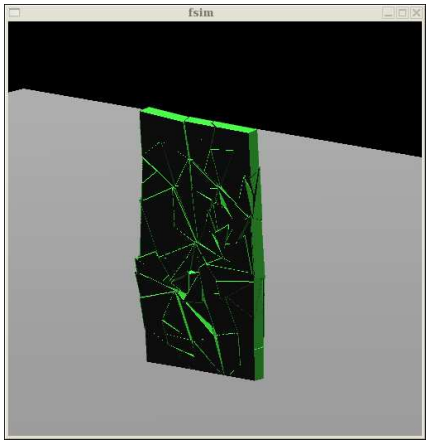

Figure 11: Light showing through cracks

5 Future Work

This is a very good method to simulate fractures. My implementation has some drawbacks as I have mentioned earlier. I would like to fix them in my own time in future. It would also be nice to have a implicit integration scheme. It would also be nice to improve this method to incorporate non-isotropic objects - like a mud and brick wall - where the fracture is more likely along certain planes in the object.