

**Dan Morris's Notes on :**  
**“Precomputing Interactive Dynamic Deformable Scenes” (James & Fatahalian)**

- The big picture
  - I want to apply forces to a deformable model in real-time, but that's hard.
  - If I can assume that only a limited set of forces will be applied to my model, you can do all your dynamic / deformable simulation offline, and play back your pre-computed simulations at interactive rates.
  
- I want to use this method to simulate a deformable scene. What do I do?
  - Get models for my scene from 3dcafe.com
  
  - Choose a small (<10) set of fundamental excitations (forces) my model might be subject to when I play with it in real-time. These are very model-specific entities that a user always has to pick by hand, e.g. “the only way the dinosaur gets moved is via someone poking his three feet”.
    - An excitation has four parameters that I need to specify :
      - An impulse (instantaneous force) ( $\alpha^I$ )
      - A constant force that persists after the initial force is applied ( $\alpha^F$ )
      - The amount of time this impulse should be modeled for (T)
    - An excitation includes a force at *every* vertex in my model, even though it might be zero at most vertices. In other words,  $\alpha^I$  and  $\alpha^F$  are vectors.
  
  - Download my favorite finite element simulator (or some other simulator), and give it my model.
  
  - Randomly apply the excitations I picked above and let the system run for a while. Every time I hit it with an impulse, record the position and velocity of the system right when I hit it with the impulse (the x component of the IRF in their paper) and the position of every vertex for a few time steps after I hit it with the impulse. I get to choose the number of time steps (the T component of the IRF in their paper).
    - Every combination of initial state (x), excitation ( $\alpha^I$  and  $\alpha^F$ ), runtime (T), and trajectory ( $x^1 \dots x^T$ ) is an IRF. Here a “trajectory” is the position and velocity of every vertex in my model at every timestep after an excitation.
    - Note that it's important *not* to let the system come to rest after every excitation, otherwise I would only know what the system does when I hit it with excitations when it's not moving and is probably at its rest position.
    - Also note that this simulation process may take days. That's the point.
  
  - Now I have my sampling of IRF's. The collection of IRF's that I sampled is called the “discrete phase portrait” P.
  
  - Load my model into an interactive simulator, and wait for user input.

- When the user hits me with some input (which *must* be one of the excitations I chose above, otherwise this method doesn't work), I look up the best pre-computed trajectory that I have. So I look at the current position and velocity of my vertices, and I look at which excitation mode I'm seeing, and I find the closest match in my library of IRF's.
- Now I just play back the IRF I found in my library until I get hit with another excitation.
  - Note that the current object state never *exactly* matches the initial state of the IRF I found in my library. So if I *literally* just played back the IRF, there would be a discontinuity when my object "jumped" from one IRF to the next. So I actually blend between them over some time to make the transition smoother.

Phil summarized the playback process really well :

---

### Playing back the model

At each instant, we are either playing back an IRF or stopped.

```
while (not stopped) {
  play (x')
  if (no new impulse)
    x' =  $\xi(x, a^I, a^F; T) + (x' - x)e^{\lambda t}$ 
  else if (t==T or new impulse)
    choose a new  $\xi$  by finding the IRF with the initial
    state closest to x'
  x' =  $\xi(x, aI, aF; T)$ 
}
```

---

- What did I leave out in this description?
  - This paper talks about using a very similar process for global illumination, which is also hard to compute in real-time. The concepts are the same, but I don't understand the lighting issues well enough to describe it here.
  - When storing an IRF in your "library", you wouldn't have enough space on a real computer to actually store the position and velocity of every vertex at every time step. So they do all their storage in a "reduced space". Basically you put all the positions and velocities for a given IRF into a matrix, do an SVD on that matrix, and throw away the big matrix (U). Conceptually, this is just a lossy compression algorithm that happens to be very easy to un-do in real-time. If you had a fast enough computer, you could use WinZip instead of SVD.
  - Looking up an IRF in my library is actually a bit tricky, since I never find an exact match for the current state of the system. Really I have to convert my system state into the reduced space and see how closely it matches the IRF's in my library. I

haven't figured out (a) whether there's a data structure that lets you do this in a non-brute-force way or (b) how much computation time this actually takes. I imagine this is a limiting factor... they do talk about caching the IRF's that are likely to come up soon, which would definitely speed things up.

- What are the limitations of this approach?
  - You have to pick a finite set of possible excitation modes ahead of time. This means I need to know the exact direction and magnitude of force that I'm going to apply at every vertex any time I interact with my model. Clearly this is *really* restrictive, but is perfect for video games.
  - My real-time playback is always following one of my pre-computed trajectories exactly, even if my initial state didn't quite match one of the IRF's in my library. So the real-time dynamics are not that accurate.
  - I can't take advantage of any linearity in my model to superimpose the deformations resulting from my excitations.