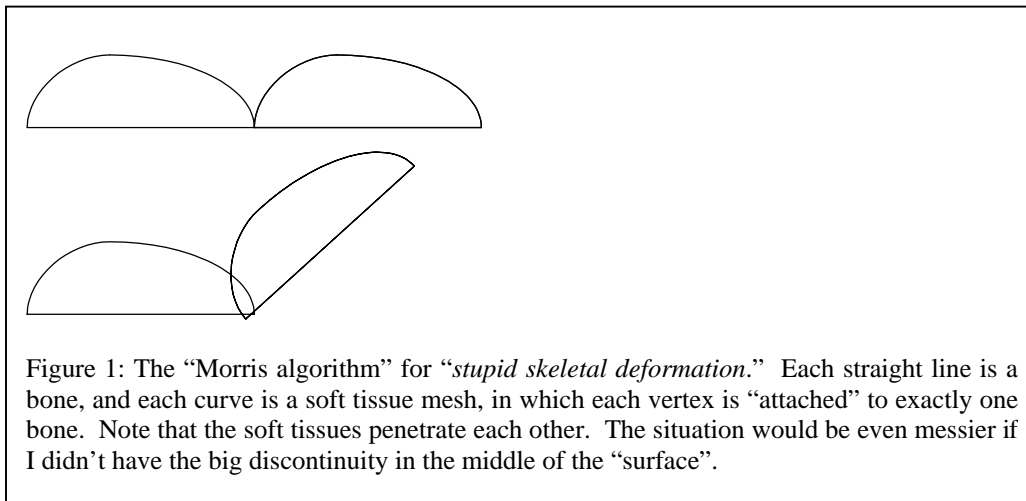


Dan Morris's notes on :

**EigenSkin: Real Time Large Deformation Character Skinning in Hardware**  
**Paul Kry, Doug James, Dinesh Pai**

- The big picture
  - We have some “skeleton” that we know how to move around. In their example, it’s actually the skeleton of a hand, but it could be any set of fixed rigid objects that are attached to each other. The whole paper assumes that moving these things around is a solved problem.
  - We have some “soft tissue” on top of the skeleton. We know where each vertex is at rest, but we want to put it at a nice-looking place as the bones move around. No topology changes, no new vertices or removed vertices.
- SSD (skeletal-subspace deformation)
  - What’s the simplest possible approach to moving vertices around with an underlying skeleton? The absolute simplest approach is to say that each vertex is just attached to one bone, and wherever that bone moves, the vertex moves with it. In other words, each vertex stores its position relative to one of the bones and is rendered in the local reference frame of that bone. I’ll call this “*stupid skeletal deformation*”.
    - What’s wrong with this? Typically tissues actually *deform* as an underlying bone moves around, especially near the intersection of two bones. This method doesn’t capture this (vertices never move relative to their neighbors), so tissue never deforms, and some tissue would just penetrate inside neighboring tissue when a joint moved.
    - This is exactly the same as using your favorite rigid mesh class to represent vertices and skipping the bone altogether. Just rotate and translate your CMesh’s instead of your bones.



- What's the next step up from this?

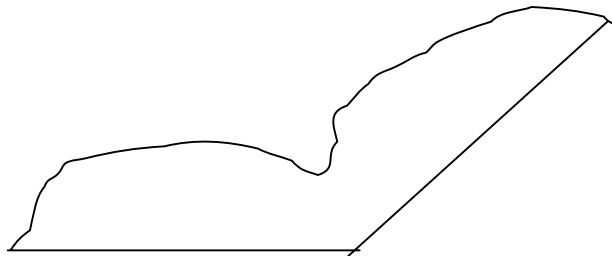
- Let's say each vertex stores a couple pieces of information... if I'm a vertex, I know which bones might affect my final position, how far I am from each of those bones at rest, and how "important" each bone is in deciding my final position (i.e. I store a "weight" for each bone).
- When I go to render myself, I let each bone "vote" for my current position, like this:

```
// My position in world space
position p = (0,0,0);

for each bone that affects me {
    find this bone's current transformation (position and rotation);
    find my own position relative to this bone at rest;
    transform this position into the global reference frame;
    multiply this global position by my "weight" for this bone;
    add this position to p;
};

glVertex3f(p.x,p.y,p.z);
```

- This (I think) is the "skeletal subspace deformation" algorithm (SSD)
- It's important that vertices near a joint have roughly equal weights for both bones they live near. Vertices in the middle of the bone are probably affected only by that bone.
- The effect is that there's no funny self-collision, since vertices near the joints are interpolated between joints.
- In fact, for this paper, their SSD weights are just derived based on distance to a bone... if I'm close to a bone in a nearest-point sense, it gets a high weight. As bones get farther away, they get lower weights. Above some threshold distance, a bone doesn't affect me at all.



- SSD is good enough for a lot of applications. It's easy to implement and avoids certain awkward self-collisions.

- Another nice property of SSD is that the only things I have to do online for each vertex are additions and multiplications of some known constants. The only things that change from vertex to vertex are the vertex position and the weight, which makes this really easy to do in a vertex shader (I can feed the weight in a texture or whatever). I put the relevant bone positions in global matrices, and send a whole bunch of vertices down the pipe.
- Eigenskin
  - But the authors are not happy with SSD... it has no physical basis and gives strange “bulging” deformations near joints.
  - So I’m going to propose a new approach, called “the Morris algorithm for stupid vertex superposition”. Forget SSD entirely, and try to do this :
    - Run a super-fancy FEM simulation of your skeleton offline, and track the position of each vertex in a whole bunch of different hand positions.
    - Try to learn some set of weights at each vertex that tells you the position of that vertex as a linear function of every bone position and rotation.
    - When you render in real-time, just multiply the weights by the bone positions/rotations and voila, you have vertex positions.
  - Conceptually, this is okay. In practice, it’s just too much information to represent with linear superposition and simple weights, and the learning problem is just too hard.
  - Instead, the authors realize that SSD is “pretty close”. So instead of trying to go offline and build a linear map from bone position to vertex position, they just plan to run SSD online, and use a fancy offline simulator to build a linear map from bone position to *SSD error*. Soooooooo clever.
  - Then when we run it online, we just have a few more multiplications to do, but we still basically have a simple linear function with limited per-vertex data that we can run in a vertex shader program.
  - In fact, using offline simulation to learn SSD error instead of absolute position is the key insight in this paper. The learning itself is basically just SVD :
    - For each joint, collect a bunch of sample “poses” (bone configurations) and all the associated vertex positions from your fancy FEM program (they cleverly only use the vertices that are affected noticeably by moving a given joint). Remember to write vertex positions as *offsets from what you would get if you used SSD*.
    - Put all the vertex positions for each pose in a big matrix
    - Take the SVD of that matrix. Now you have a set of “eigendisplacements” (fundamental displacements caused by moving this joint) and – for each

vertex – its exact displacement in each pose in terms of these eigendisplacements. Again, remember that displacements are all *relative to the SSD-based positions*.

- As we always do with SVD, throw out the low singular values and all their associated displacements. We usually do it to save space, but here we do it because we have a limited number of multiplications we can perform in the graphics hardware that we're ultimately going to use for rendering.
  - A footnote... note that I say "for each joint" at the beginning of this bulleted list. They choose to represent each vertex's displacement due to each joint independently, so this method wouldn't work if there were complex or non-linear effects involving multiple joints.
- When we render each vertex, we'll load up the eigendisplacement basis (which we got from our SVD) for that vertex and the current bone configuration for relevant bones, and let the hardware do the linear superposition. Then we'll run SSD and add the result to what we got from our eigenstuff.
  - One interesting point comes up that's related specifically to the implementation in graphics hardware. For each vertex, I can only fit 64 floats (at the time) of per-vertex data, plus some global data that doesn't change from vertex to vertex (like matrix transformations). They conclude that I can fit roughly 10 eigendisplacements per vertex. So how do I allocate that space? If a vertex was affected by just one single bone, I would use all 10 spots to store displacements related to that bone (meaning I could use the first 10 singular values from my SVD). If a vertex is affected by two bones, do I take five singular values (eigendisplacements) from each bone? I could, or I could do four and six, based on the magnitude of the relevant displacements or singular values. They don't really speculate on that much, but they do imply in their conclusion that they end up using just one or two singular values from each joint for a given vertex in some cases (probably because that vertex was affected by five or ten different joints).