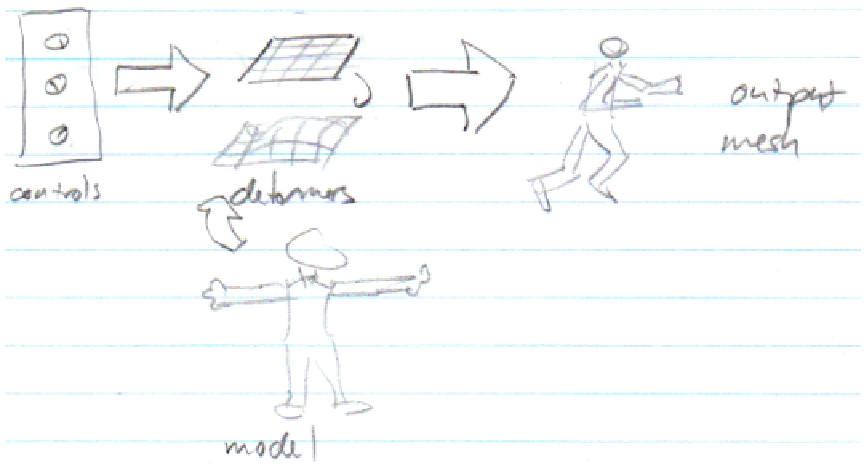
Real time geometric deformations

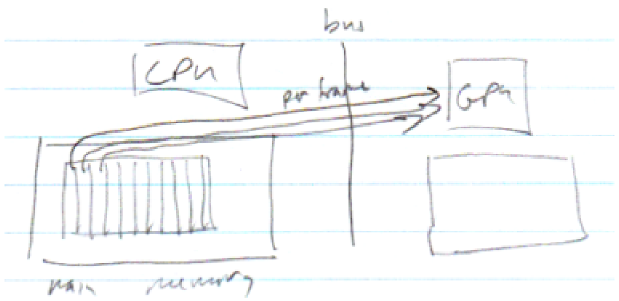
Steve Marschner
Cornell University
CS 569 Spring 2008, 6 March

We've discussed simulation as a way to get motion, but if you can't simulate the motion you want you'll need to be able to work with data that comes in as keyframed animation or motion capture data.

The tools for computer animation (for offline applications) have developed a variety of great ways to specify subtle and realistic motion, by using variants of a general *rigging* approach in which animator controls define the parameters of deformers that are then applied to a model to obtain the output model. I'll assume the output model is ultimately just a triangle mesh, although in practice the structure of the model (and its deformations) may be maintained into the renderer.



For offline rendering, the output meshes can just go, one per frame, to the rendering system. But for realtime applications, this creates a bandwidth problem: a fresh set of vertex positions per frame is a lot of data, both to store and to transmit to the GPU:



You could certainly do this, for a moderately sized mesh, but it does consume a lot of resources.

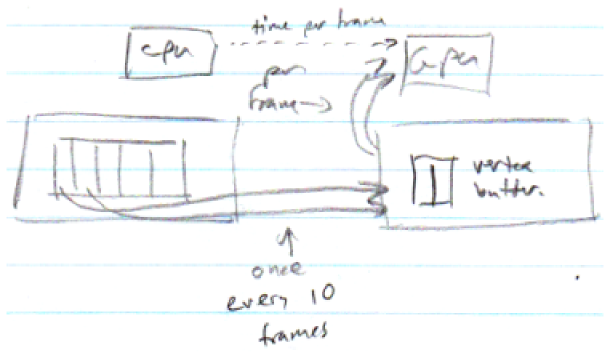
Frame interpolation and blend shapes

One might want to reduce bandwidth by somehow compressing the vertex position data stream. For instance, we could send just a subset of the frames, say every tenth frame, and linearly interpolate between them to approximate the frames in between. How could we implement this in the shader pipline?

A: previous and next vertex positions in attributes; time in uniform; interpolation in vertex shader.

Does this help? No! In fact, it's just doubled the bandwidth problem. (Although it does improve the storage requirements in application memory.)

In order to reap the benefits of expressing our deformations in terms of a smaller set of frames, we need to avoid redundantly transmitting the vertex positions. The way to do this is by using a *vertex buffer*, an OpenGL object that lives in GPU memory (like a texture or a render buffer) and stores vertex data. A buffer object can serve as the source of attributes for later drawing calls. Nothing changes from the shader's point of view, but the system diagram now looks like this:



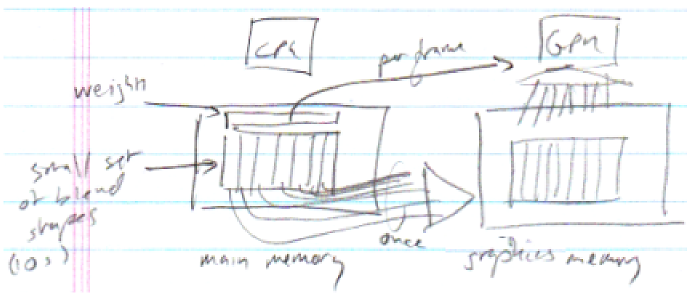
The quality of the resulting motion can be considerably improved if we choose the frames carefully, rather than just by regular sampling. Just like with keyframe animation, the best results will be had by interpolating between poses that are extremes of the motion, or *key frames*. These key frames could be selected based on the key frames that the animator used, or selected automatically. Also, there's no reason the blending weight has to be tied directly to the time; it could change at a variable rate to better match the original motion. None of this would change anything in the implementation; we just send the vertex data at different times and send different numbers for the interpolation weight.

To get smoother results we could use three or four frames in a higher-order interpolation. If the interpolation involves n frames, we'd need n sets of vertex data in graphics memory, which we'd pass into the vertex shader as n different attributes, and we'd send n (or $n - 1$) blending weights per frame. With this fancier interpolation we could hope to send fewer frames of data, at the cost of more complex vertex processing.

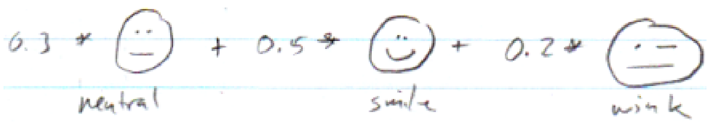
If we step back for a moment and look at what we're doing from a mathematical standpoint, we can see that we have a vertex program that basically computes a linear combination of vectors (thinking of all the vertex positions strung out as a long vector). What we are doing is sending a basis—a set of vectors that span a useful range of deformations—to the GPU, and then for each frame sending weights for a linear combination of those vectors—or, the coordinates of our deformed vertex positions in that basis. For an arbitrary animation where things change a lot, we need to keep updating this basis set, but for cases with a more limited range of motions we can come up with one basis that serves for the entire animation.

This is a limiting case of the frame blending idea, where we send all the basis shapes up front, then only send blending weights thereafter. This approach is known as *morph targets* or *blend shapes*, and is quite popular for facial animation. For example, see NVIDIA's various human character demos.

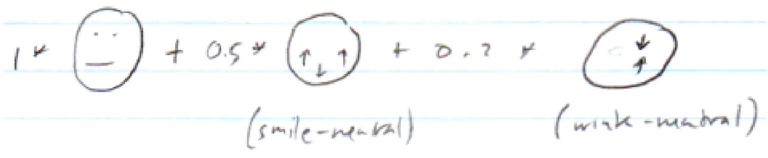
In the data flow diagram it now looks like this:



With a time-independent set of blend shapes, the blending shapes are also useful animation parameters, if we choose meaningful blend shapes. For faces the shapes are often classified as expressions (e.g. smile) that affect the whole shape or modifiers (e.g. eye closing) that are more local:



In the simple form where we store the actual vertex positions as the basis vectors, we have to ensure that the weights sum to 1. Another way to do it, which automatically ensures this, is to store each pose as an offset from the neutral pose:



Blend shapes can be quickly summarized with an equation:

$$P_{out} = \sum_i w_i P_i$$

$$P_{out} = P_{neutral} + \sum_i w_i (P_i - P_{neutral})$$

The second equation reflects the practice of using offsets from a canonical "neutral" pose, which results in better numerical behavior (the basis is better conditioned and you're less likely to see cancellation in low-precision storage formats).

Calculating space deformations on the GPU

Blend shapes are one approach to animating meshes, by taking linear combinations of basis meshes. The other major approach is to deform a mesh by using a remapping of the space in which the mesh lives. Here the idea is that we have a function ϕ that maps R^3 onto itself, and we define the deformation of a surface by mapping all the points of the surface through that function.

For the case of a mesh, we'll just transform the vertices through ϕ—and there had better be enough vertices or there will be artifacts.

Examples of sources for the function ϕ are free-form deformations, which define a deformation of space directly by using a 3D spline, parametric deformations like twisting, and arbitrary functions we cook up for special purposes, like the circular wave deformer everyone seems to like to use as an example (see slides).

As long as ϕ is simple enough to evaluate, it's trivial to implement the deformation of vertex positions. It starts to get more interesting when you need to compute the transformed normal. (Where have we seen this problem before?)

We all know how to compute transformed normals for the simplest deformation, which is an affine transformation. We use the inverse transpose matrix. (Recall why.)

If the deformation is nonlinear, though, we can't just have a matrix for the transformation. But we can have a matrix that approximates the transformation locally—it's called the derivative matrix. We can map tangents through the derivative matrix, and normals through its inverse transpose.

Implications of normal transformation in vertex shader. Do you really want to invert a matrix?

That's pretty much all there is to it. It's important to have enough triangles, so that when the deformation stretches them out they don't get too big.

Linear blend mesh skinning

One of the most popular ways of deforming meshes is called skinning. The idea is that you have a mesh that's supposed to deform pretty much rigidly over local areas, but not by the same transformation everywhere. The canonical example is a human figure.

An early approach was just to assign each vertex to one bone, and transform that vertex by its bone's transformation. This could be OK for very low-res meshes, but it will produce bad artifacts if you start with nice smooth meshes, because the sharp boundaries between surface regions affected by different transformations will become painfully obvious.

The idea behind skinning is to smooth out those discontinuities by blending between the transformations over some range rather than just switching from one to the next. Rather than assigning each vertex to one bone, you give each vertex a weight for every bone (many of which will be zero—the left pinky doesn't need to affect the right forearm). Then each vertex of the mesh is transformed by a linear combination of bone matrices:

$$p_i^{out} = \left[\sum_k w_{ik} M_k\right] p_i^{in}$$

Another way of seeing this is that we are blending the positions the vertex would receive under each individual bone transformation:

$$p_i^{out} = \sum_k w_{ik} (M_k p_i^{in})$$

To do linear skinning you need bone weights per vertex. They are the responsibility of the modeler—they could be automatic, or they could be painted manually. They do need to add to 1 in order to avoid changing the overall position of the mesh.

To match the form you often see this equation written in, we can think of each bone transformation as the difference between that bone's *bind pose transform*—its location in the model's neutral pose, where the original vertex positions are created—and its current transform. This amounts to taking the current frame transformation and multiplying by the "bind pose inverse" matrix:

$$p_i^{out} = \sum_k w_{ik} T_k B_k^{-1} p_i^{in}$$

where *T* is the current transform and *B* is the neutral (bind pose) transform.

Now, what about the normals to this surface? We already know how to transform the normals through each of the (affine) bone transformations:

$$n^{out} = (M_k^{-T}) n^{in}$$

so all we need to do is blend the transformed normals, right? Like this:

$$n_i^{out} = \sum_k w_{ik} (M_k^{-T}) n_i^{in}$$

Well, yes, this is how everyone does it. But is it right? Does skinning fit into the general deformation framework from the previous section, so that we can we use the same Jacobian rule we used for general deformations?

Further reading