Pose Space Deformation Notes

j.p.lewis

Pose Space Deformation is described in a Siggraph 2000 paper (a .pdf is probably located in the same place where you found this). This document gives some additional notes and implementation considerations.

Pose Space Deformation (PSD) is a simple algorithm (the core engine can be implemented in a few dozen lines of code) that combines aspects of skinning and blend-shape approaches to deformation, and offers improvements and additional control. The basic idea is to think of skin movement, not as a function of time, but as a function of the creature's pose.

PSD allows you to move the creature to any pose and resculpt the skin surface in that pose. The sculpted alteration is smoothly (rather than linearly) interpolated as the creature moves to and away from that particular pose. Because the edits can occur at any pose, the underlying interpolation problem is one of scattered interpolation, as opposed to interpolation schemes such as splines that assume the data are situated on a regular grid. There are a number of scattered interpolation algorithms that could be considered. Among these, Radial Basis Functions are simple, and have a variety of extensions that can be explored.

Lastly, the concept of "pose" can be more generally considered, to include abstract variables such as the amount of weight being carried or the character state (super-hero mode or Clark Kent), as well as the literal relative joint angle variables that define the pose.

Dimensionality

J. Blinn commented in one of his CG&A columns that humans are particularly bad at switching between coordinate systems. Pose space deformation can be confusing in that there are several different "spaces" and dimensionalities involved: the dimensionality of the pose space, the dimensionality of the model, and the number of training examples (as well as the "3D" space of the resulting model). In PSD these dimensionalities can all be different. This section will explain these several spaces, starting from simple examples.

The first step is to temporarily forget about the ideas of 2D and 3D. Mathematically, a "dimension" is anything that can be varied. A slider is one "dimension". A single vertex has three variables (x,y,z), and so is a point in 3 dimensions. Similarly, a triangle on a 2D plane is defined by three vertices each with two coordinates, and so such triangles can be considered as points in 6 dimensional space. In general, add up the number of variables, consider each as a dimension, and consider objects with that number of variables to be points in a space with that many dimensions.

Thus, if a face model has 10,000 vertices, each with 3 coordinates, then a set of 60 sculpted faces (e..g. blendshape targets) of this resolution will be considered as 60 points in 30,000 dimensional space.

As a first example, consider an animation of a face aging over time. Use 5 keyframes: baby, child, teen, adult, old, and interpolate over these five in the time direction. In this example, the "pose space" is one dimensional (time), the number of training examples is 5 (the keyframes), and the dimensionality of the models is 30,000.

This is a standard sort of setup in computer graphics and the interpolation can be accomplished with splines without requiring and scattered interpolation and pose space. To make it more interesting, consider changing the pose space to be two-dimensional. As dimensions, use two sliders, one being "happy-sad" and the other being "low/high energy". As shown in the circomplex psychology research showin in the paper, this 2D space is an appropriate space for interpolating emotions – for example, "bored" has the energy slider set to low, and has the happy-sad slider slightly on the sad side. To restate, in this example the "pose space" is two dimensional, and the dimensionality of the models is 30,000, and the number of training examples is the number of sculpted face model targets that you place in the space. The models can be placed at arbitrary points in the space – position the sliders, and put the model there. It will be smoothly interpolated.

Next consider a blendshape-like setup. Suppose there are 60 blendshape targets, each with 30k dimensions (as above). There are also 60 sliders. In PSD terms, this would be considered as a 60-dimensional pose space, with 60 training examples. With blendshapes, the number of targets dictates the dimensionality of the space in which they are interpolated, and the targets must be placed exactly along a single dimension, that is, the target corresponding to the 4th slider will be at the location 0,0,0,1,0,0,0,..... in the 60d (pose) space. Mathematically, the targets are placed at the vertices of a 60-D hypercube, and the neutral is at the origin. (Maya and probably other systems allow intermediate targets to be placed along a particular slider, so one could have an intermediate target at 0,0,0,0.5,0,0,0,.. as well as at the 1 position on that dimension).

Pose space generalizes this setup by breaking the link between the dimensionality imposed by the sliders (the dimensionality of the pose space) and the number of sculpted targets. There can be any number of sculpted targets, and they can be placed anywhere in the space. Further, they can be smoothly (rather than linearly or "hyper-linearly") interpolated. Thus, one might start off by doing 60 standard targets (61 counting the neutral). Then, supposing that sliders 2 and 5 "fight", and the interference is worse at position 0.7 and 0.3 on these sliders. Put those sliders at that position, resculpt the model, and associate that target with position 0,0.7,0,0,0.3,0,0,... in the pose space.

PSD

In the region near an elbow, the skin is affected by the relative angle between the upper and lower arm “bones”. The relative angle is one value, so this is a one-dimensional pose space. This is a bit too simple to be interesting, so we’ll talk about a shoulder instead.

In the shoulder case, the pose space has two dimensions, these being the angles between the upper arm (with 2 degrees of rotation) and the torso coordinate system. (Actually, for a very realistic pose-based deformation these two angles may not be enough; it might the upper arm, a “collar” type bone, and maybe some bones on the back. But let’s just consider the two angles).

First the modeler will pose the creature, and at one or more particular poses they decide to resculpt the surface. The resculpted surface (call it a *PSD target*) is saved, associated with that pose. Probably, the model has some underlying skinning or stitching applied that serves to keep it from cracking at the joints when the model is moved, but fails to produce a realistic deformation. Call this the *underlying* surface. When the modeler resulpts the surface, at each vertex the *displacement* from the underlying surface to the resulpted surface is saved. The deformation is expressed in the local coordinate system of the body part (upper arm or torso).

Suppose that the modeler decides to make PSD targets (adjustments) at three different poses. The vertices that are moved may be different in each of these three poses. Thus, vertices fall into several classes:

- vertices not changed by any of the PSD targets
- vertices changed by only one of the three PSD targets
- vertices changed by two of the three PSD targets
- vertices changed in all three of PSD resculpting operations

Thus, in the most general and flexible scheme, one should think of setting up and solving the PSD at the vertex level, not at the surface level. (2004 added note: the Weighted PSD scheme is actually a better approach, see the section later in this document).

(While the weights will be different at every vertex, the “PHI’ matrix is common to a particular class of vertices... so its inverse could be computed once rather than once at each vertex in the class. The inverse operation happens at modeling time rather than at animation time, and the inverse involved is small, so this isn’t a huge cost anyway - it’s probably easiest to just treat every vertex independently).

For a particular cv the modeler makes corrections at 3 different poses. Then the length of d is 3.

In the case of the 3d model there are 3 ”d” vectors, one for each of x,y,z.

so call Ri = PHI inverted, then

```
// solve for weights
  wx = Ri * dx weights for x, given x component of the N displacements
  wy = Ri * dy
  wz = Ri * dz

// synthesize
  dx = sum of wx[k] * PHI( |pose[current] - pose[k]| )      sum over k=1..N
  dy = sum of wy[k] * PHI( |pose[current] - pose[k]| )
  dz = ...
```

where pose is a 2d location pose(angle1, angle2), so —pose1-pose2— is

```
  sqrt( (pose1[angle1]-pose2[angle1])^2, (pose1[angle2]-pose2[angle2])^2 )

// lastly,
  point[X] += dx; point[Y] += dy; point[Z] += dz;
```

Kurihara and Miyata's weighted PSD

Kurihara and Miyata showed an incredible animated hand, produced semi-automatically from cat scans.

their video

`http://www.eg.org/EG/DL/WS/SCA/SCA04/357-365.pdf.abstract.pdf` paper abstract

Their Weighted PSD (WPSD) scheme introduced a major improvment: in forming ”distance” in the pose space, they consider the underlying ”SSD”/skinning weights - so if a vertex has only a small weight to a particular bone, that dimension does not contribute much to the distance.

They also normalize the RBF weights to sum to one. I do not follow the logic of this normalization, in that: far from any of the sculpted poses it would seem to cause whichever is the nearest (albeit far) example/sculpt/model to be fully on, rather than having the examples decay to zero far from their locations in pose space. However, given that the results look good, maybe there is some reason why the normalized RBF weights are a good thing.

Layering PSD on an existing skinning system

PSD can be used to improve upon an existing skinning system, by interpolating sculpted adjustments to the underlying skinning.

In this case, arguably the quality is improved if one switches the order of operations from

```
  PSD(SSD(model), corrections)
```

to

```
  SSD(model + PSD(corrections2))
```

where ”corrections” are what the animator sculpts, and ”corrections2” are some other displacements that produce the same effect when passed through SSD().

Here’s a description of how the Powell’s method could be used to effect this change of order

```
s = the sculpted (moved) vertex
p = the orignal vertes
d = the movement by the user

   s = SSD(p) + d        =        SSD(p+e)
```

we want to find e (e=corrections2). This can be setup as a minimization problem,

```
   minimize(e):   | s - SSD( p+e ) |^2
```

so, the function to minimize is

```
   f(e) = | s - SSD( p+e ) |^2
```

if this f() can be expressed as computer routine (that calls Maya to get the value of SSD()), then this f() can be given to the Powell method to find the minimum of e.

In the case of pure SSD() it is possible to find the inverse directly, however in the ”Maya” situation where

1. the skinning is a somewhat unknown algorithm,
2. the skinning process is user-adjustable and the particular rig may have other components such as blendshapes, lattices,...

this general approach makes more sense.

See Xiao Xian, et al, A Powell Optimization Approach for Example-Based Skinning in a Production Animation Environment, Computer Animation and Social Agents (CASA 2006) for further details and an illustration of why the SSD-last ordering is preferable.

Other related approaches

The origins of PSD are in an an approach I developed at ILM in 1995. That system was used briefly in Jumanji (in the elephant-walks-over-car hero shot), and in Casper (it was used to animate the dress in a scene where the fat ghost was dancing). ILM obtained a patent on this approach in 1999. Compared with PSD, the this earlier approach used a primitive form of scattered interpolation, and overall the interpolation was poorly conceived. On the other hand, it did "invert" the skinning to do the interpolation in the local space.

"Verbs and adverbs" interpolate motion of skeletons using RBFs (with added linear trends). This paper really should have been cited by our 2000 Siggraph paper – however I was working in production at the time and was not aware of it until later (in production one has trouble keeping up with the academic literature, or anything else in life...).

Shape-by-Example is a technique presented by Sloan, Rose, and Cohen at the 2001 I3D conference; their technique has some broad similarities to PSD. Shape by example interpolates shape using RBFs with linear added trends. However, they seem to be interpolating the shape itself, rather than a delta from an underlying skinning system.

A quick analysis suggests that their technique requires less computation at the authoring stage but is slower at runtime synthesis: Shape-by-example deforms a point as

$$x[j](p) = \sum_k \sum_m x_k[j] r_{m,k} R(p)$$

whereas PSD is

$$x[j](p) = \sum_k w_k R(|p - p_k|)$$

Since the sqrt involved in the distance can be folded into R, PSD deformation appears to be more efficient.

The shape-by-example paper has some tips on improving RBF interpolation. One of these improvements is arguable for the purpose of skinning however: they fit a polynomial (e.g. linear) trend under the RBF deformation (as is commonly done in RBF implementations of thin-plate splines). This is not always desirable, because it means that a sculpted deformation does not die off far away from the original pose, but instead continues to grow.

Regularization

If several of the sculpted poses are very similar then the matrix R can be close to singular. There is an easy and effective approach to dealing (known by several names including regularization, ridge regression, etc.). This technique is useful in other situations too.

As motivation/background,) When two rows of a matrix are similar, the data can be "explained" by setting the weight on one of the rows to zero, or by setting each to half of the needed weight, or by having one of the weights be huge and offset by a large negative weight on the other row. This last scenario is undesirable of course.

In general this would mean that your selection of basis functions is poor, and the model should be understood better. In the PSD case however the user chooses the "basis functions" (sculpted poses) and you (the programmer) has no control over what is chosen.

Regularization is an adequate solution in this case: Add a soft constraint that votes for the weights to be not too large. i.e., minimize

$$e = (Rw - d)^2 + \lambda w'w$$

(' = transpose, so $w'w = ||w||^2$) w=the weight vector, λ is a small user-defined scalar like 0.01.

Taking derivative with respect to the weights gives a nice result:

$$\begin{aligned} de/dw = 0 &= 2R'(Rw - d) + 2\lambda w \\ & R'Rw - R'd + \lambda w = 0 \\ & (R'R + \lambda I)w = R'd \\ & w = -(R'R + \lambda I)^{-1}R'd \end{aligned}$$

so in practice you can just add a small constant like 0.001 to the diagonal before inverting. There is some further analysis (which I forget for the moment) that points out that when a diagonal element of R'R is already large (corresponding to an element of w that will turn out small) then adding 0.001 to that diagonal element affects things little, but when the diagonal element is small (and the corresponding element of w would end up being huge), adding 0.001 will make a big difference. So this selectively affects the problematic weights much more than others.

The 0.01 should be a user-adjustable parameter. I'm not sure how to explain to the users what it means!

Different RBF kernel widths at different points

This can be implemented simply. In the matrix instead of having

```
[R(1,1), R(1,2), ...;
 R(2,1), R(2,2), ...
...]
```

where R(1,2) means the rbf kernel R() indexed by the distance between points 1,2, one would have

```
[R1(1,1), R2(1,2), ...;
 R1(2,1), R2(2,2), ...
...]
```

where R1() is the rbf kernel scaled for the width at point one.

Other Implementation extensions

Some of the various possible extensions (different basis shapes, kernels not located at data points, etc.) are useful in implementations. The Bishop book Neural Networks for Pattern Recognition (Oxford) cited in the paper is a good introduction; also see F. Girosi's paper "On Some Extensions of Radial Basis Functions and their Applications in Artificial Intelligence". The paper

> Reconstruction and Representation of 3D Objects with Radial Basis Functions J. C. Carr, R. K. Beatson, J.B. Cherrie T. J. Mitchell, W. R. Fright, B. C. McCallum and T. R. Evans ACM SIGGRAPH 2001, Los Angeles, CA, pp67-76, 12-17 August

is another good reference; it describes fitting RBFs to data sets containing millions of points.

There are a number of papers on techniques for adapting the kernel shapes and locations to the data, e.g.

> St. de Marchi, Schaback, and Wendland, Optimal Data-independent Point Locations for Radial Basis Function Interpolation.

Tutorial Java language libraries for two-dimensional scattered interpolation are available at

```
scribblethink.org/~zilla/Work/PSD/
```

(translation to C++ should be straightforward).

There are two routines. A Gaussian RBF routine has the constructor

```
public RbfScatterInterp(double[][] pts, double[] values, double width)
```

where `pts` are the 2-D locations with data, `values` are the values at those locations, and `width` is the width of the Gaussian kernel. A Thin-Plate RBF routine has the constructor

```
public RbfThinPlate(double[][] pts, double[] values)
```

This routine does a thin plate + affine fit to the data. The thin plate minimizes the integrated second derivative of the fitted surface (approximate curvature).

In fact any radial kernel different than a linear function will work, so one can choose a smooth piecewise polynomial for efficiency, or store the kernel in a precomputed table. It is often desirable that the interpolated function decay to zero far from the data, which is not true of the thin-plate interpolation. The affine component of the thin-plate code is useful; this should be incorporated in the Gaussian RbfScatterInterp routine.

Both routines implement the interface

```
public interface ScatterInterpSparse2
{
  float interp(float x, float y);
}
```

as the call to actually perform the interpolation.

For the character deformation application one needs M-dimensional interpolation of 3-dimensional CVS, for arbitrary M depending on the number of joints and other qualities. This can be accomplished using 3 separate M-to-1 dimensional interpolations. The extension of the interpolation code from 2-D to M-D requires changing the distance function from x^2+y^2 to an appropriate distance between points in M-dimensional space.

PSD versus Shape Interpolation

Blend shapes or shape interpolation (SI) has several drawbacks:

- *Shapes are not independent*. A major consideration in designing face models for shape interpolation is finding sets of shapes that do not "fight" with each other.

  Animators describe this common problem with shape interpolation: the model is adjusted to look as desired with two targets. Now a third target is added; it interferes with the other two, so the animator must go back and adjust the previous two sliders. And so on for the fourth and subsequent sliders. Sophisticated models (e.g. those on Disney's Dinosaur) can have as many as 100 blend shapes, so this is a lot of adjustment due to shape "fighting".

  Likewise, the authors of shape interpolation programs have described artists' complaints relating to lack of shape independence – with highly correlated shapes it is not clear which slider should be moved. Some shapes reinforce, others cancel, sometimes a small slider movement results in a large change, sometimes not.

- *Animation control is dictated by sculpting* Each slider controls one key shape, each key shape is controlled by one slider, as it has been for 15 years of facial animation.

- *Linear interpolation*

Pose space deformation (PSD) algorithms address these problems.

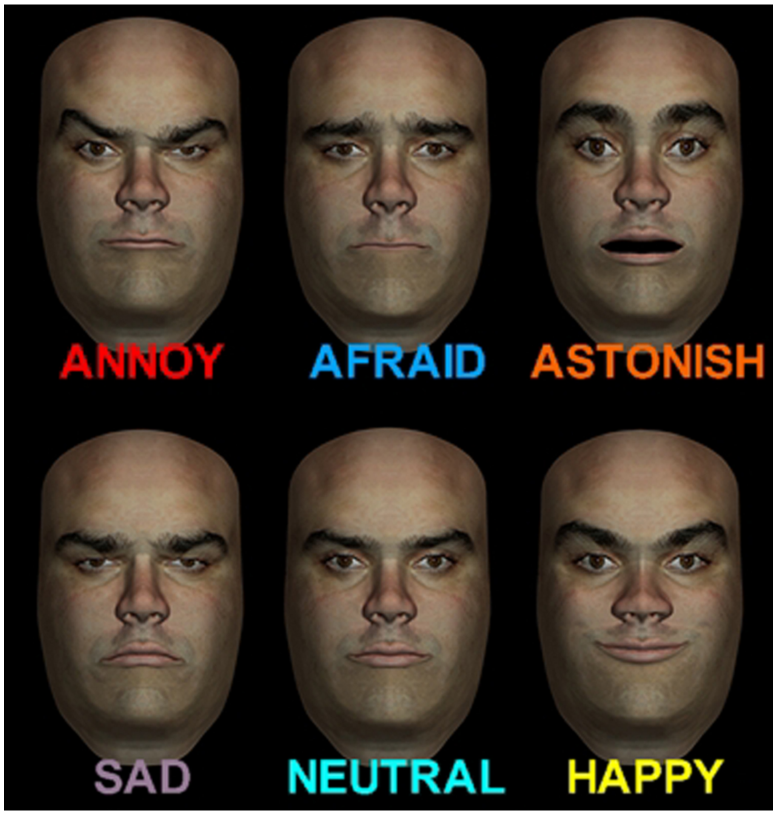

Figure 1: PSD Facial poses selected according to psychological research rather than traditional motion extremes. Model and figure by Nickson Fong.

Interpolation versus superposition

The problem of shapes “fighting” is because the shapes are simply added. PSD *interpolates,* so keyshapes do not interfere.

Modeling is decoupled from animation control

It is necessary to be able to control the influence of each keyshape, but the one-for-one mapping is not the only way to do this.

- Non-control shapes. Suppose “excited” and “happy” are two distinct target shapes, but in a direct crossfade the intermediate shape is not adequate and a new model is required. With SI one would need to introduce a new slider for the intermediate “half-excited-half-happy” model, and this simple crossfade then requires manipulating three sliders. Arguably this is complexity caused by the system rather than desired by the animator. With PSD, place the halfway shape halfway between the key shapes and it will automatically be interpolated during the crossfade.

- Higher order parameters. The decoupling of sculpting from animation control makes it possible to consider other sorts of control. Example: A character is sometimes possessed by the devil, sometimes not. Place the keyshapes in a PSD space with one extra dimension (‘possessed’), then a performance can be changed from possessed to not by just flipping a switch.

- Psychologically relevant poses. The decoupling of sculpting from animation control makes it possible to design facial poses by criteria other than motion extremes. Example: Psychologist J.A.Russel has studied human perception of emotion and found that emotional similarity is mostly explained by a two-dimensional space . This space may be the most appropriate one in which to interpolate emotional expressions. See Fig. 1.

- Regularization of parameters. If two sculpted shapes are similar, having one slider per shape does not reflect this, and thus the ‘slider movement may make small or large changes’ problem. PSD allows similar shapes to be placed as neighbors in a chosen control space. See the implementation section on regularization for more details.

Smooth rather than linear interpolation

PSD allows smooth interpolation if desired, whereas with shape interpolation, in going from shape A to B and then to C, an individual cv moves in a piecewise linear way – there is a kink at B. Easing in/out of the transition does not change this fact.

Linear Algebra View of Shape Interpolation

Linear algebra gives another viewpoint on the character of motion resulting from shape interpolation: Shape interpolation of n shapes each having m control vertices

$$S = \sum_k^n w_k S_k$$

can be written as a vector-matrix multiply with the keyshape vertices arranged in the columns of a $m \times n$ matrix.

$$\begin{bmatrix} c_{1x} \\ c_{1y} \\ c_{1z} \\ c_{2x} \\ c_{2y} \\ \vdots \\ \vdots \\ c_{nz} \end{bmatrix} = \begin{bmatrix} | & | & \cdots & | \\ | & | & \cdots & | \\ | & | & \cdots & | \\ | & | & \cdots & | \\ S_1 & S_2 & \cdots & S_n \\ | & | & \cdots & | \\ | & | & \cdots & | \\ | & | & \cdots & | \\ | & | & \cdots & | \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

The range of this matrix is at most of dimension n, so the animation is restricted to this subspace of dimension $n << 3m$ reflecting the fact that individual cvs cannot move independently. The ‘Bruton’ Dino model appears to have 60*52 + 4*21 + 4*18 + 4*21 + 15*35 + 16*35 + 11*35 + 18*35 + 17*21 + 17*16 + 17*21 + 11*21 = 6677 cvs and so can be represented in a 3 (x,y,z) * 2 (symmetry) * 6677 length vector. On the other hand it appears that there are under a hundred key shapes used to animate this head.

The preceding vector interpretation is valid; the next analogy is only that (an analogy). Consider the cv’s as “samples” representing the resolution of the model – so the Bruton model has 18k samples. Also consider the number of samples needed to represent an object in the subspace of possible movement: 100 or less. This ratio of 100/18k reflects a *movement deficiency* - it indicates how much modeling resolution is *not* used in the animated movement.

A similar vector space interpretation of PSD is more complex but indicates that the PSD motion is richer than that produced by shape interpolation. A single coordinate of a particular cv is deformed as

$$c = \sum w_k R(|\vec{\theta} - \vec{\theta_k}|)$$

where $\vec{\theta}$ is the vector of PSD parameters. The matrix R changes depending on $\vec{\theta}$, and w_k are different from one coordinate to the next, so the range is not a simple subspace – each cv has some amount of independent movement.

Efficiency: PSD vs SSD*

```
ssd:  v_final = sum w_k * T_k * v
```

```
   e.g.
      v_final[x] = w1 * (T1[0,0]*v[0] + T1[0,1]*v[1] + T1[0,2]*v2 + T1[0,3])
                 + w2 * (T2[0,0]*v[0] + T1[0,1]*v[1] + T1[0,2]*v2 + T1[0,3])

   very roughly = 3*J*4 multiply/adds, where J is the number
   of bones/transforms that the vertex is weighted to, and 3 because x,y,z.

psd:  v_psd = sum w_k table[ angle - angle_k ]
   e.g.
      v_psd[x] = w_1 * table[ (rx-rxp[1])^2 + (ry-ryp[1])^2 + rz-rzp[1])^2 ]
               + w_2 * table[ (rx-rxp[2])^2 + (ry-ryp[2])^2 + rz-rzp[2])^2 ]
               + w_3 * table[ (rx-rxp[3])^2 + (ry-ryp[3])^2 + rz-rzp[3])^2 ]
   rx,ry,rz is the current pose, assuming 3d case,
   and  rxp[k] is the x-rotation of the k'th example/correction

   very roughly = 3*E*(D*something) multiplies,
   where E is the number of example shapes, D is the number of dimensions,
   and "something" more than 1, maybe less that 2, to take into account
   that the sum of distances^2 involves an extra subraction,
   not just add-multiply as in the ssd case.
```

...psd is more expensive, but it does not seem hugely more expensive in terms of the number of multiplies, maybe a factor of 2-5x depending on the number of dimensions and poses?