

Run-Time Skin Deformation

Jason Weber

Intel Architecture Labs

jason.p.weber@intel.com

<http://www.intel.com/ial/3dsoftware/index.htm>

1 Introduction

This paper describes how to take data from common authoring systems and implement a skeletal based animation system. By using a skeletal system instead of saving vertex positions for every key frame, not only can you save storage space, but you have the opportunity to generate instantaneous new motions in your application such as with Inverse Kinematics.

To apply the skeletal motion to the mesh, a concise technique of vertex blending allows multiple bones to influence each vertex. This is used to produce smooth joint motion without excessive stretching. Additional techniques are shown to resolve troublesome animation artifacts.

Issues are discussed concerning integration with other technologies. Also, cache optimization and performance numbers are presented.

2 Quick Quaternion Primer

This paper assumes basic knowledge of quaternions, which are very briefly described here. For a proper explanation, see [Bobic98-2].

A quaternion is a four-dimensional extension to complex numbers. So, just as a complex number $x+yi$ can be used for some two-dimensional rotational computations, $w+xi+yj+zk$ ($i^2=j^2=k^2=-1, ij=k=-ij$) can nicely represent three-dimensional rotations otherwise commonly represented as a 3x3 matrix. Fortunately, all the common operations of quaternions useful in this context are all already worked out and readily available. This paper refers to right-handed unit quaternions of the form (w,x,y,z) , where $(1,0,0,0)$ represents an identity of no rotation. Be aware that some sources prefer (x,y,z,w) notation and there are a few rare examples with left handed quaternions which involve some sign differences.

The easiest way to get to a quaternion is from an angle/axis notation. Given an angle of rotation θ about an axis v , the quaternion is $(\cos(\theta/2), \sin(\theta/2)v)$. (Note that since v is vector, the second part represents the last three quaternion components). When looking at a quaternion, it may be helpful to picture it as "sort of" a normalized angle/axis notation.

Conversions to and from matrices contain quite a few math operations, but example code is easily found. Each conversion contains a couple of dozen add/multiplies. You would not want to continually convert back and forth, but you can easily make just-in-time conversions once per frame. Inverting a unit quaternion involves just a simple sign flip.

The primary advantage of quaternions is their interpolation qualities. Imagine two quaternions rotating a unit vector where the endpoints of the resultants are points on a unit sphere. There is an established method to find any quaternion in between where the resultant endpoint is on an interpolated arc path between those points and where that arc is the shortest path along surface of the sphere. This is referred to as Spherical Linear interERPolation (SLERP). If you want to take a longer path around the sphere, you have to interpolate through intermediary points. Similar operations with matrices or Euler angles can be costly and have a much less pleasing result.

Euler angles are a rotational representation given as a concatenation of rotations about orthogonal vectors, usually axes. Euler angles can perform very badly when some components are near 90 degrees since resultant axes can align and cause a situation called Gimbal Lock. Conversion to a quaternion from a set of Euler angles is fairly easy, but the reverse is difficult and can have many possible solutions. However, the conceptual simplicity of Euler angles can make the conversion worth the capability in certain author-driven situations such as defining the angular limits of joints.

3 Required Data

3.1 Mesh

The mesh represents the surface of the object we wish to animate. Although the discussed techniques are not highly dependent on the structure of the mesh, we will assume some details used in our performance-optimized format.

First of all, our implementation uses a purely triangular-faced mesh. However, most of the described methods do not really depend heavily on face structure.

More importantly, the mesh structure has exactly one independent normal for every vertex. Normals that would otherwise be shared are duplicated. Because of multi-resolution rendering algorithms, there may even be multiple co-located vertices that don't necessarily have the same normal values. This format facilitates the use of vertex array rendering calls, such as `glDrawElements()`, which draw a large number of polygons all at once instead of specifying each discrete polygon through multiple API calls.

3.2 Reference Bones

To control the animation of the mesh, we need to have a hierarchy of bone segments that are carefully aligned with the original mesh. Each bone can be thought of as a transform, with a displacement (translation) and rotation. By convention, each bone is aligned with its local X-axis. Displacements are stored as 3D vectors. Rotations are stored as quaternions. We store each bone transform relative to the endpoint of the parent bone. A bone's endpoint is located on its X-axis displaced by a fixed bone

length, also given for each bone. This convention means that the common case where each bone's base is attached to its parent tip (endpoint) results in zero displacement. Using relative rotations results in near linear chains of bones having near-identity rotations.

At this time, our system neglects the capability of scaling bone transforms for the sake of optimizing space and performance, but it is a trivial modification to add scaling to these resultant transforms if needed.

The bone data also contains constraint data representing the maximum angles of rotation around each axis. Constraints are used to limit an Inverse Kinematics solution to angles that the represented body can reach. Following the apparent convention of a common authoring package, these three angles are interpreted as a ZYX Euler angle relative to the absolute world space, not each bone's parent. This means that the Z angle is restrained first, and the Y angle is restrained based on the new orientation after the Z rotation. Likewise, the X rotation (which controls the twisting motion of the bone about its length) is restrained after the Z and Y rotations are already applied.

Additional information is stored to support weight smoothing and regeneration algorithms that will be described later in this paper. First, an approximate cross-section of the mesh at the base and tip of each bone is represented by a pair of potentially offset ellipses. Also, the number and length of bone links is given.

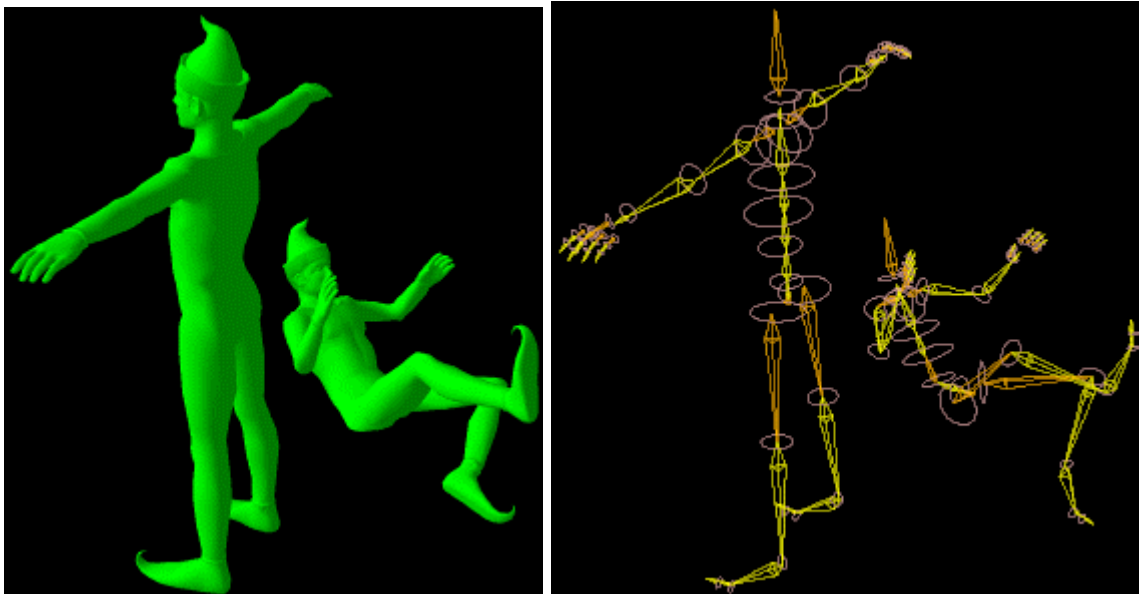


Figure 1 Reference and animated positions, show as (a) solid mesh and (b) with bones and cross-sections exposed.

3.3 Vertex Weights

Vertex weights are what ties the bones and mesh together. Each vertex in the mesh is tied to any number of bones. Each vertex-bone relationship has a scalar weight. The total weighting of any vertex should add up to exactly 100%. It is not necessarily an

error condition to have a vertex that is not tied to any bone, but it will be immune to the deformation effects.

If a vertex is tied to only one bone, that vertex will move with that bone as though it was connected to the bone by a rigid rod. For very coarse characters, this may be acceptable, but detailed characters will show excessive stretching and twisting at the joints since the effects will often occur on single faces. Tying a vertex to multiple bones will smooth out the transition between the transforms of those bones. This is explained under Vertex Blending.

3.4 Motion Data

The motion data represents the changes in displacement and rotation of each of the bones. We represent a motion as a list of named tracks, generally one for each bone. Each track has a list of displacements and rotations for specific times.

Since the times do not have to be at regular intervals nor do tracks have to follow the same set of time values, this does not necessarily have to represent either sampled or keyframed data. Track entries that can be accurately interpolated between their adjacent time points do not need to be listed.

4 Applying Motion to Bones

Every bone can be assigned one track. A specific skeleton can draw from multiple motion sets, such as applying a running motion to the lower body and applying a throwing motion to the upper body. Bones that are not assigned a motion track stay as they are, presumably at their reference. These bones can also be controlled manually or through Inverse Kinematics.

The assignment of a bone to a motion track maintains pointers to the linked list nodes in the track most nearly before and after the requested time. These values are retained to prevent searching the linked list for every frame since they only need to be updated when the requested time exceeds the time bounds of these nodes. Interpolation is used to compute the exact displacement and rotation for the requested instant. Currently, we use linear interpolation for the displacements and quaternion SLERP for the rotation.

5 Using Inverse Kinematics

Inverse Kinematics (IK) can be used to spontaneously generate motions based on the location of movable control points called effectors. Generally, a certain chain of bones, such as an arm, is directed to follow a specific effector. The IK system attempts to rotate the participating bones so that the end bone is located at the effector. Multiple iterations can improve the solution, potentially over multiple frames.

Solving large inverse kinematics solutions can be very complex and expensive. Fortunately, there is a reasonably simple iterative method that is quite fast and surprisingly robust, called Cyclic Coordinate Descent, nicely outlined in [Lander98-11]. Basically, an iteration starts with the last bone in the chain. The bone is rotated so that it points directly at the effector. Next, the parent of that bone is rotated so that an imaginary line from the base of that parent to the tip of the newly rotated child points

toward the effector. This is repeated for every bone in the chain so that each bone's imaginary line from its base to that same end bone's endpoint rotates toward the effector. Using quaternions to determine the change in angle makes the computation fairly straightforward. Multiple iterations refine the solution to a smoother distribution of angles. Weighting can be used to assert preference on which bones participate more in the solution.

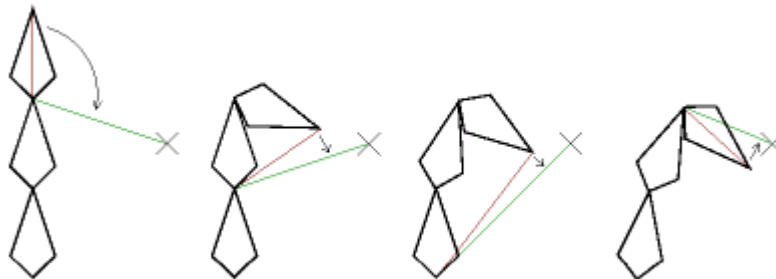


Figure 2 Cyclic Coordinate Descent

The figure shows steps in a descent. The X is the location of the effector. The red line represents the current base-to-tip angle. Each currently addressed bone is rotated to align the red line with the desired green line.

If a bone participates in multiple effected end bones, the participating bone can compute an angular change to satisfy each of the solutions, then use given weighting factors to work out a blended solution.

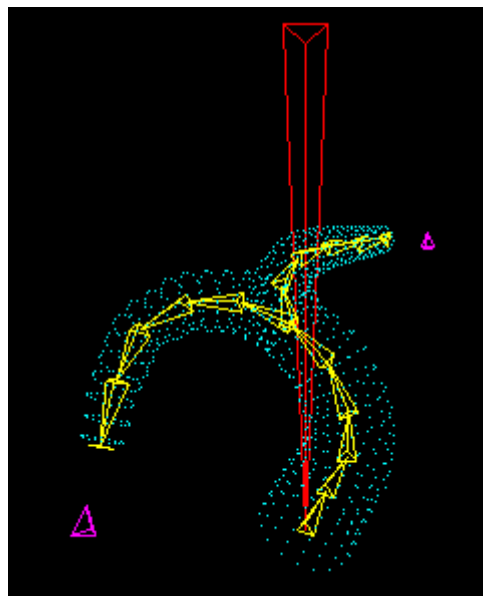


Figure 3 IK determining joint angles based on two effectors

See [Welman93] for detailed explanation of CCD and other methods.

5.1 Reference-based IK

One approach resets the participating bones to the reference position at the start of each frame. The solver must go through multiple iterations to reach a reasonable solution. If the root of the chain and the effector do not move, the solution will be identical for every frame. If the effector moves in a way that makes a radically different solution preferable, the entire limb can suddenly jump in a single frame.

5.2 Incremental IK

An alternate approach, which we will distinguish by calling “incremental” IK, is to retain the previous frame’s solution and use it to start the iterations on the new solution. As the effector moves gradually, a new solution can be found quickly, perhaps by using only one IK iteration per frame. Also, this method gives the solver a historical preference. If an arm is straining to reach one way around the back, the solver is less likely to fling the arm around the opposite way until doing so is significantly preferable. However, if the constraint mechanism is not carefully built, it is possible to form small looping motions that vibrate around a solution.

5.3 Velocity Limiting

To reduce the potential odd effect of sudden solution changes, a limit on angular velocity can distribute a large change over multiple frames. Comparisons are made using the relative difference in rotation from the last frame converted to angle/axis format. This difference angle is limited to the maximum angular velocity and converted back to the quaternion representation. Acceleration limits haven’t appeared to be very helpful since most organic creatures can reach their maximum comfortable angular velocity within the normal period of an animation frame.

5.4 Joint Constraints

Without restriction, IK would quickly cause contortions of limbs that would not be physically possible without appreciably injuring the character. As mentioned before, each joint has a set of angular constraints. After each angle is determined during the cyclic coordinate descent, it is restricted to constraining angles, given as differences from the reference position. To apply the constraint, the quaternion angle is first converted to Euler angles (a non-trivial operation). Each of the angles is then limited to the appropriate constraint. The resulting Euler is converted back to a quaternion, which replaces the original angle. These restrictions could probably have been more flexibly represented with a quaternion-based region, but authoring considerations and established convention prefer the conceptually easier Euler format.

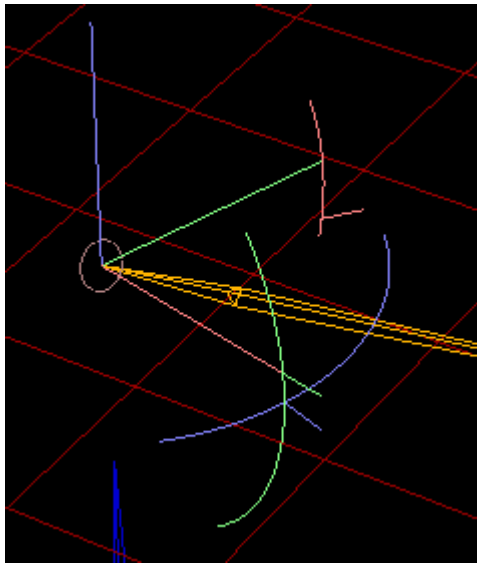


Figure 4 Arc Representation of Constraints

In the above figure, the green (Y) and blue (Z) axes and arcs represent bending limits and the red (X) axis and arc represents twisting limits. Note that in this method the angles are not independent. The restrictions are applied in the ZYX order.

By authoring constraints for a full chain of joints, the IK system can be applied to full limbs without reaching unnatural positions.

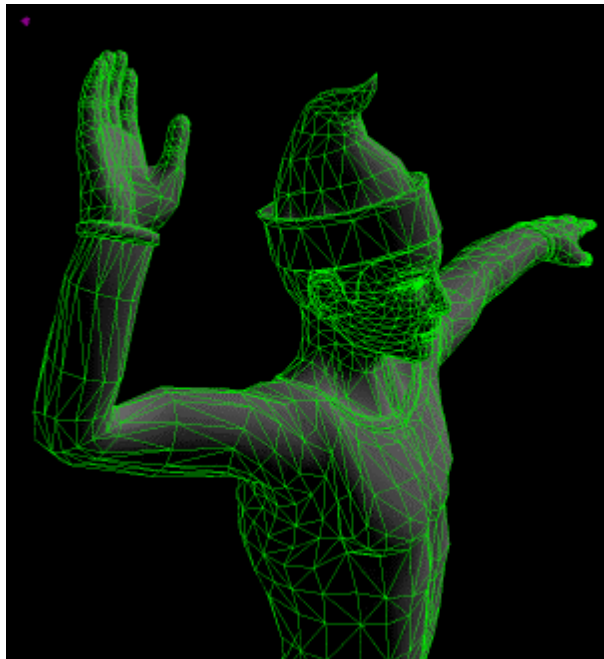


Figure 5 Joint constraints limiting IK solution of reaching back to effector (shown as tiny purple dot)

6 Basic Vertex Blending

Vertex blending utilizes the vertex weights we obtain from an authoring process. Recall that each vertex is potentially tied to several bones, where each association has a weight, all of which total 100%. We also need the position of the vertex relative to each applicable bone's reference transform. (We will alter this method later in the Hardware Compatibility section). To compute each blended (deformed) vertex position, we use the following equation:

y = deformed vector position of vertex
 b = number of bones
 w_n = scalar weight of vertex to bone n
 x_n = original vector position of vertex relative to bone n
 M_n = transform matrix of bone n

$$\bar{y} = \sum_{n=0}^{b-1} (w_n * \bar{x}_n M_n)$$

Note that since most of the w_n values for any specific vertex will be zero, you should really only sum over the few bones that actually contribute.

This method passes each bone-relative vertex position through that bone's current transformation matrix, scales each result by that bone's weight to the vertex, and then adds up all the results. This gives you the new vertex position. If the weights smoothly transition from one bone to the next along the joint, the resulting deformation should also be reasonably smooth.

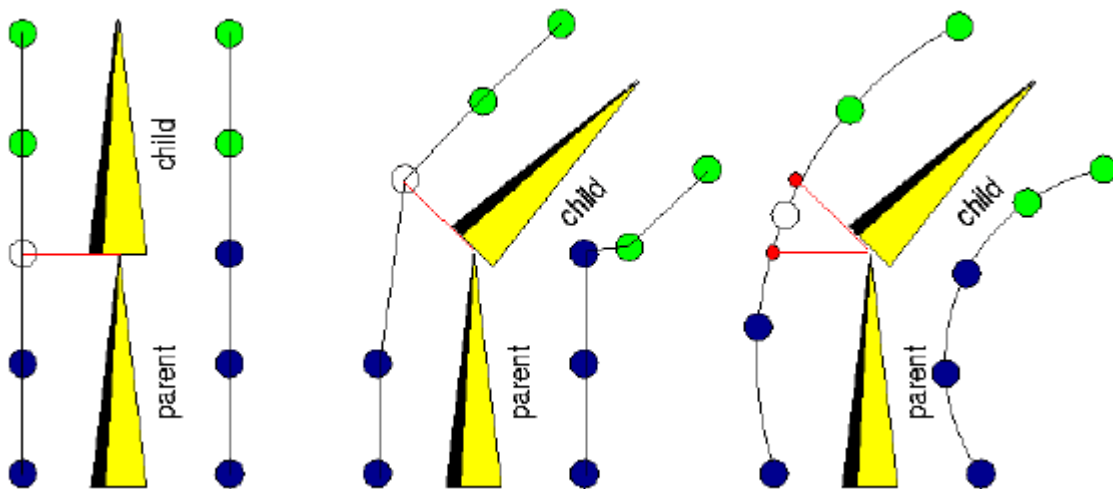


Figure 6 Vertex Blending (a) before rotation (b) without blending (c) with blending

The figure shows a simple joint where, at first, the dark blue vertices are fully weighted to the parent bone and the light green and white vertices are fully weighted to the child. The middle diagram shows the result using just the single transform per vertex. The rigid behavior causes excessive stretch and compression. The third diagram shows how the vertices can move if the middle vertices use a weighted blend of the two transforms. As an example, if you transform the white vertex by each of the two

transforms, the results would be the two small red vertices. The resultant position of the white vertex is the weighted average of the red results, in this case an equal weighting.

Normals can be treated much the same as vertices with a minor change. You only use the rotational component of the transform and ignore the translation. Note that the blending will likely reduce the magnitude below unity, but since the resulting magnitude will always be between 0 and 1, a small lookup table can avoid using square roots to restore unity.

See [Lander98-05] for further reading.

7 Shortcomings of Vertex Blending

Basic vertex blending can be very fast and works very well with reasonably small differences in angles. However, larger angles can cause serious artifacts. The averaging process tends to make joints shrink with both bending and twisting motions.

Mathematically, this problem is clearly demonstrated in an example with a twist of 180 degrees about a parent's axis. Consider a point weighted 50/50 located at $(0,1,0)$ relative to the child bone, such as the white vertex in the following figure. Examining the results relative to the tip of the parent bone, the 50% influence contributed from the parent bone results in $(0,1,0)$ weighted to $(0,0.5,0)$. The 50% influence contributed from the child rotated 180 about X results in $(0,-1,0)$ weighted to $(0,-0.5,0)$. The sum of these two results is $(0,0,0)$. Any 50/50 weighted vertex in the 180 rotation would received a Y and Z of 0.

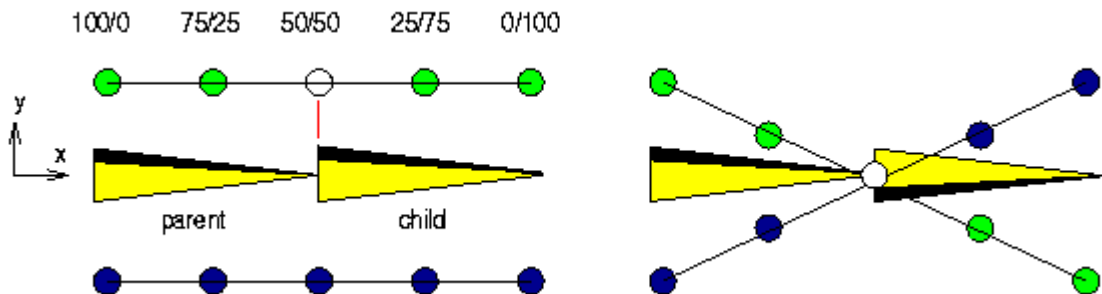


Figure 7 Effect of twisting child bone 180 degrees

So, instead of acting like a chain of incrementally rotating rings, the joint acts more like cardboard paper towel tube. An equally displeasing effect occurs when bending about the Y and/or Z axes. In this case, the joint flattens along the angle of rotation, again like a cardboard tube.

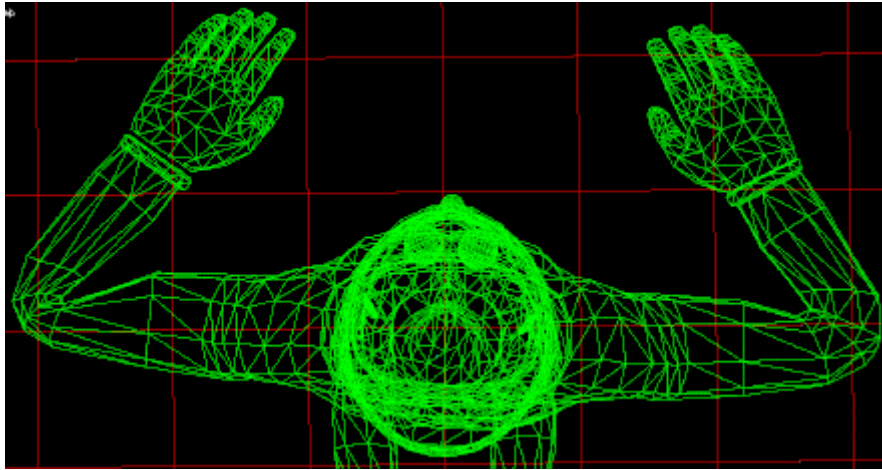


Figure 8 Bending Elbows (right side uses Bone Links, described later)

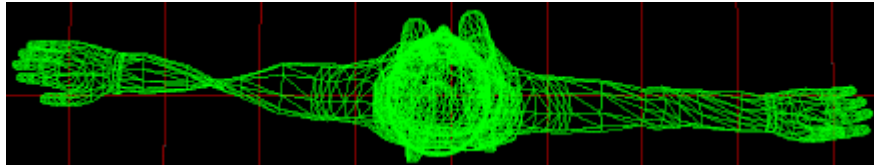


Figure 9 Twisting Elbows (right side uses Bone Links, described later)

These left sides of these figures show the simple blending effect applied to elbows of an actual model. The right side uses an additional technique explained later.

8 Cheating the Shortcomings

There are ways to adjust the weights to minimize the appearance of the problems. For example, the elbow is usually represented in reference position as fully outstretched. From there, it can bend almost 180 degrees in one direction. Simple vertex blending will do pretty good job of rounding the backside of the elbow. If you use single bone weighting on the front side of the elbow, the skin surfaces will intersect, but the result is fairly believable. However, this would not help if the elbow flexed both ways or twisted about the X-axis. This can be more difficult in a joint such as the shoulder whose ball and socket allows wide rotation about all three axes.

9 Bone Links

After a few attempts at alternate algorithms to vertex blending that retained radial information in general cases, we only ended up with results that were much slower and had frequent problem cases. Also, any algorithm that did not use simple vertex blending might not be compatible with emerging hardware support of simultaneously using multiple transforms. We needed a vertex blending approach without the artifacts.

Simple vertex blending works very well for small angles up to perhaps 60 to 90 degrees, but can fail badly at angles closer to 180 degrees. So if we can limit our skeleton to only blend between bones rotated less than around 60 degrees from each other, the simple vertex blending should work well.

Of course, an elbow, for example, bends nearly 180 degrees. One trick is to insert extra smaller bones near the troublesome joint. The skeletal system can automatically move these new bone “links” based on the motion of the original parent and child bones. The weights at that joint need to be algorithmically reassigned accounting for the bone links. This reassignment can be based on either the original proportion of weights between a parent and child or by the actual relative X location. The former only seems to work well when it resembles the latter, so the latter appears preferable.

The number and length of the bone links are important to the overall result. Using a single added link between bones usually results in a poor effect, but two or three links are just about as good as ten and cost much less in overhead. A reasonable link length can be generated from the child bone length and average joint radius. We have been using a default ($total_linklength=0.3*child_length+1.5*joint_radius$). In any case, both the link number and length should be adjustable as part of the authoring process.

A little additional attention needs to be made when there are more than just two bones involved, which is common in forking areas, such as the hands, shoulders, and hips. The problem is that you only want to reassign the amount of the parent’s weighting that is actually attributable to that side of the fork. For example, if a vertex is weighted to three bones, one of which is the parent of the other two, a naïve algorithm might first address the parent and one child. After the parent’s weight is potentially entirely reassigned, addressing the second child results in significantly less weight to redistribute than if it were the child that was addressed first. So during any reassignment, the algorithm must consider only a fraction of the parent’s weight. A reasonable method is to use the magnitude of each child’s original weight to split up the parent’s weight. This can become a little more complicated when you consider ancestors beyond the parent.

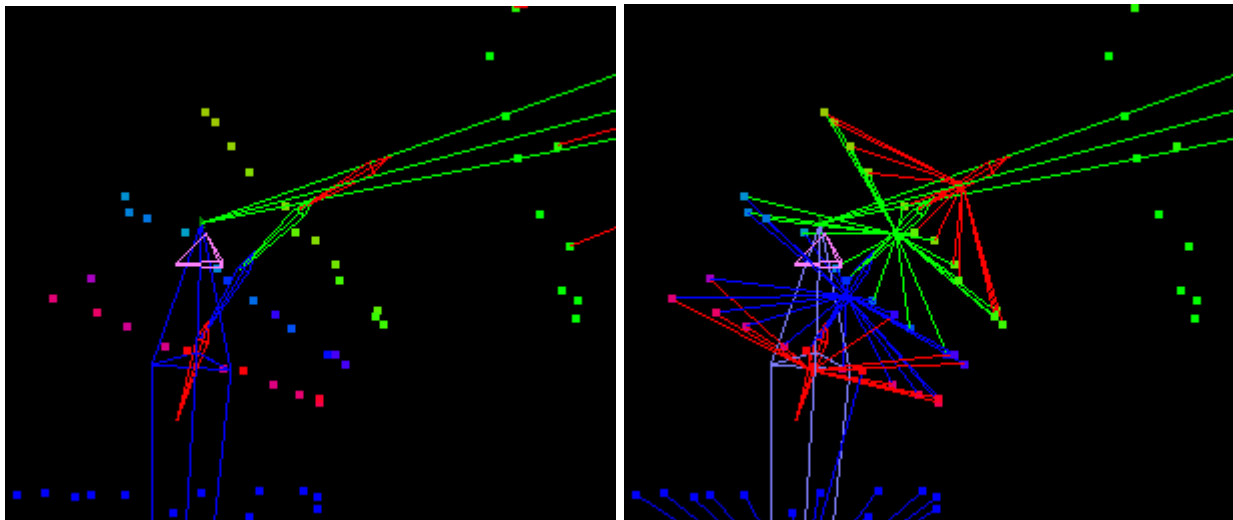


Figure 10 Bone Links in an Elbow (second image shows associations to bones)

The figure shows each bone in red, green, or blue and blends the colors of the deformed vertices based on their weighting. The second image also draws a line from each vertex to every bone with which it is weighted. (The purple tetrahedron is just a center of interest cursor in the viewing tool).

Another problem we can correct for is common in the lower armpits. Since the mid-chest can be partially weighted to the upper arm, it can be affected by raising and lowering the arms. But the transform for the upper arm tends to make area below the arm bulge in and out instead of slide up and down. To reduce this effect, some the weight intended for bone link redistribution is simply reassigned to the parent. This magnitude of this reassignment is determined by the dot product of the child bone's X-axis and the vertex normal. Vertices with normals perpendicular to the bone, such as near the elbow, have little change. Vertices with normals more parallel to the child bone, such as below the armpit, are appreciably affected.

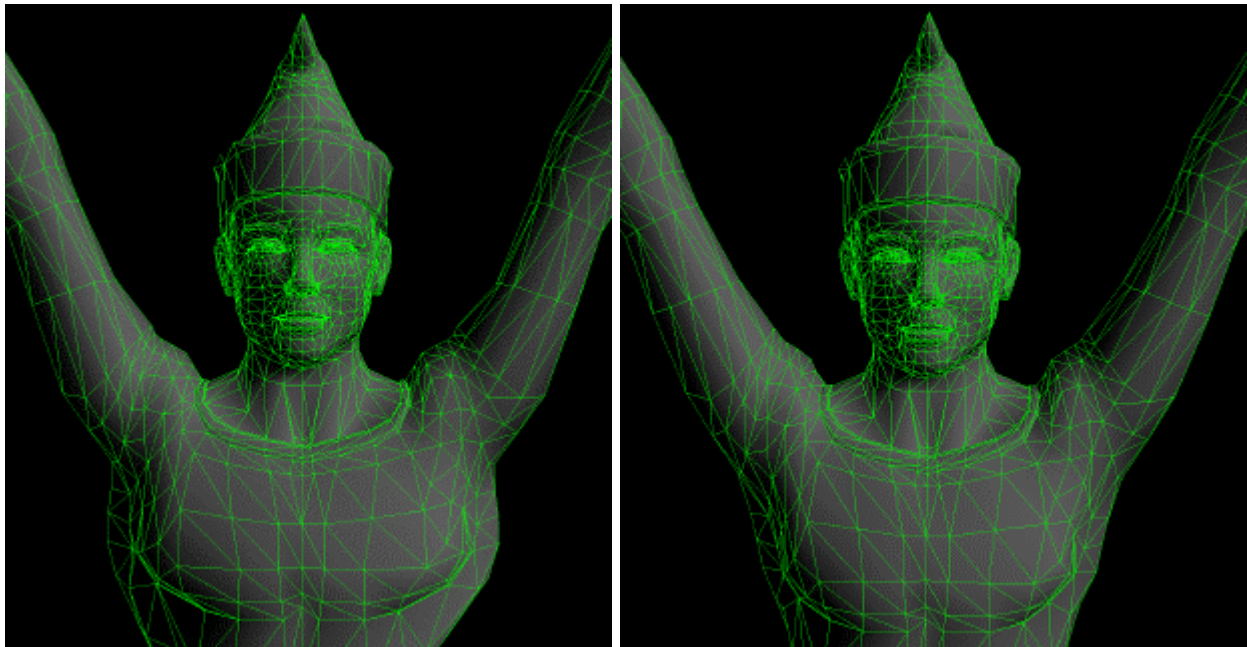


Figure 11 Normal-Derived Influence Fade (a) without and (b) with

10 Weight Smoothing

For vertex blending to have reasonable results, vertices near each other should only have mild differences in weighting. If radical weight changes occur on the mesh surface, excessive stretching and self-intersection are likely to occur.

This often happens with quick assignments through some semi-automated algorithm in the authoring package, but even apparently properly authored models can exhibit such problems.

To overcome this, a weight-smoothing algorithm can be used that limits these transitions. A base threshold specifies a maximum change in weight allowed proportional to distance between vertices. This limit is applied between all vertices tied to each bone. If the threshold is exceeded, the more heavily weighted vertex contributes some of its weight to the lesser. After each iteration, all the vertex weights are re-normalized to 100% total weighting. Multiple iterations can spread a radical change over a wide area.

Using this uniform threshold over the whole object is problematic. A threshold that is mild enough to correctly smooth the torso will have basically no effect in the details of the fingers. So, we adjust the threshold based on the thickness of the body area we are dealing with. To do this, we first automatically calculate an approximate elliptical cross-section at the base and tip of each bone. As each vertex is smoothed, the cross-sections of its influencing bones adjust the threshold for that comparison.

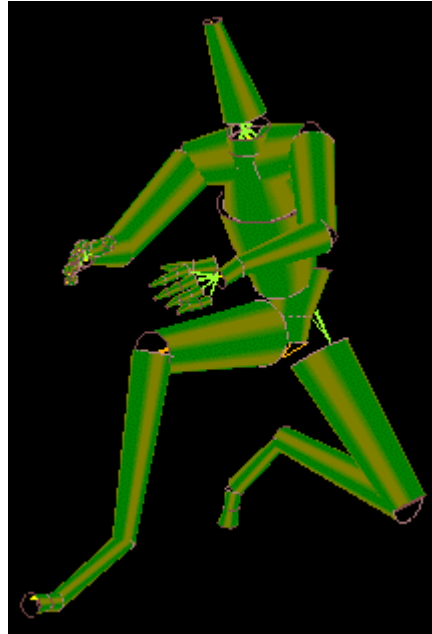


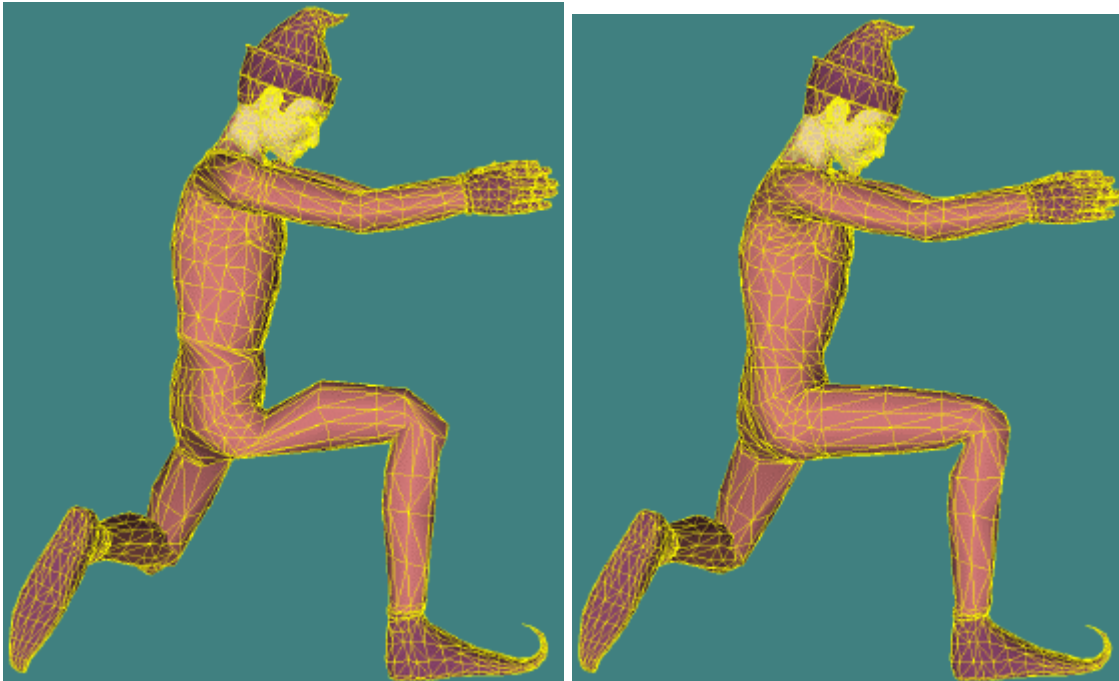
Figure 12 Estimated Cross-Sections connected with cylinders

Note that if the model is meant to be mechanical in nature, smoothing may likely produce an undesirable soft and flexible behavior.

11 Weight Regeneration

If weights do not exist or the given weights are particularly bad, it is often possible to generate new weights using a completely automatic algorithm. The first step is to assign every vertex to one bone. For this, we use the cross-sections mentioned in weight smoothing. These cross-sections represent tapered elliptical pseudo-cylinders encapsulating each bone. For every vertex, its location is compared to every one of these cylinders and assigned a score. A cylinder that contains the point obviously receives a very good score, better for how close to the bone the vertex is. The rules outside the cylinder are less obvious. Scores fade fairly quickly off the endpoints of the cylinder (where $x < 0$ or $x > \text{bone_length}$) but fade rather mildly radially from the bone (increasing magnitude of Y and Z). Most desirable distributions don't have many vertices attaching past the endpoints since the next or previous bone would probably be a better choice. But objects with a lot of features can easily have bumps or projections well beyond the radial limits of the appropriate cylinder. This slack also allows for the possibility that the automatic determination of joints can sometimes be smaller than desired, but the lengths of the cylinders are always the same as the bone lengths.

After this process is complete, the result is a single-weighted model that would apparently have all the artifacts of a non-vertex-blended model. But, if you use the aforementioned weight smoothing after the weight regeneration, you can often get a very pleasing result.



**Figure 13 (a) Original exported model authored using ellipsoidal regions
(b) Same model using Weight Regeneration, Proportional Auto-Smoothing, and Bone Links**

Note that carefully authored weights are almost certainly preferable to any automatic algorithm. This algorithm was originally designed to demonstrate what reasonable weights would look like, but ended up being more applicable than was expected.

12 Rogue Removal

Both the auto-regeneration algorithm described above and the common elliptical containment algorithms in authoring packages can result in what we will refer to as rogue patches. A specific example is a character whose large ears get partially weighted to the clavicle bones. Since the tips of the ears are closer to the clavicle than to the head, either of the algorithms can potentially choose the clavicle as the appropriate attach point. This can result in rather displeasing stretching of the ears as the head and shoulders move. We have seen similar effects near hands, feet, and beaks.

Fortunately, this is a fairly easy condition to detect and correct. First, each bone determines which of the vertices that it influences is the least likely to be incorrectly attached, such as the one that received the best score in weight regeneration. (A read-only pass at weight regeneration can determine these scores if the original weights are not being replaced). All vertices tied to this bone that are contiguously connected to this “best” vertex are marked valid for that bone. Each vertex’s validity is independently

determined for all the bones it is tied to. Any vertices that have weights to bones that are not validated have to be changed.

A simple correction method is to find the nearest valid vertex following the surface mesh and assign those invalid weights to the bone of the nearby valid vertex, thus making a larger valid patch around that nearby vertex.

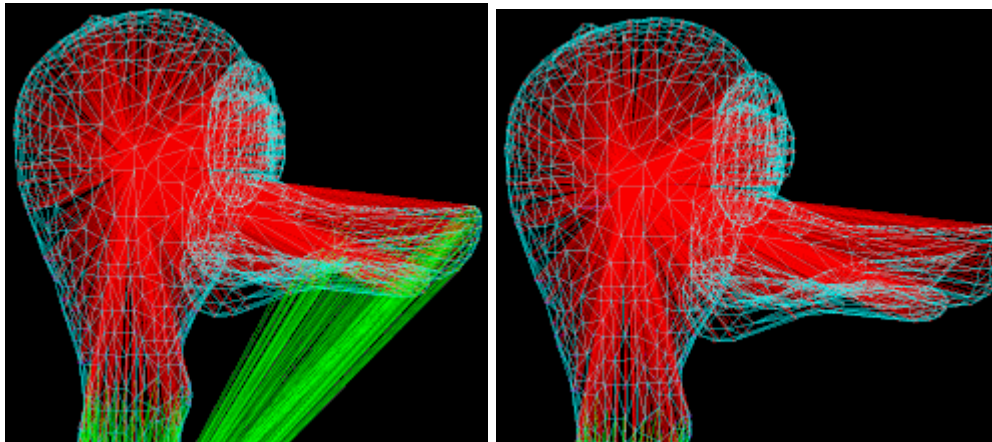


Figure 14 Duck Beak (a) without and (b) with Rogue Removal

The figure shows a duck whose beak was partially weighted to the neck (green) instead of entirely to the head (red). This would cause bend abnormally when the head moved relative to the body. Since this patch is detached from the primary patch on the neck, it was determined to be invalid. The nearest valid vertices along the surface were assigned to the head, so the neck component of each of these vertices was reassigned to the head.

13 Pipeline Integration

Character animation needs to operate in an animation pipeline with other technologies. Each of these other technologies has its own requirements and effects on the mesh. Our current pipeline can include:

Multi-Resolution Mesh (MRM) -> Animation -> Non-Photorealistic Rendering (NPR)

13.1 Multi-Resolution Mesh (MRM)

The purpose of the multi-resolution technology is to modify a mesh in place to continuously lower and restore the vertex and face count in response to changes such as distance from the viewer and system loading. MRM changes the quantity and connectivity of the faces. If different resolution levels require different normal or material data for a vertex, that vertex is duplicated. Only one "version" of the vertex is used at a time. The vertex array is sorted so that at any resolution setting, most of the currently unused vertices are above an adjustable index. This is very fortunate for the deformation algorithm since it makes it very easy not to apply deformation to unused vertices. Because of this, deformation processing time is nearly linearly proportional to the MRM resolution settings.

Another fortunate “coincidence” is that during run-time, MRM does not alter the actual vertex locations or normal directions, just the face structure. Animation changes just vertex locations and normal directions, not the face structure. Since the two technologies don’t overlap their changes, both technologies can operate on the original mesh without having to continuously copy data every frame. Note that this requires the animation system to cache the original vertex locations, but this is already done to optimize cache performance.

There was some concern that since the MRM algorithm was not accounting for possible deformations, the flexed areas would have artifacts. But, we have not seen any examples where this is an evident problem.

13.2 Alternate Rendering (such as NPR)

Alternate rendering techniques are easily integrated with vertex blending animation engines as long as the rendering system accounts for the fact that the normals and vertices will change. There are some issues raised in the Hardware Compatibility section.

13.3 Subdivision

We are not currently cleanly integrated with a subdivision surface technology. One open question is whether subdivision should precede animation or follow. If subdivision precedes animation, then it may be able to adaptively add faces at critical flex points. However, it would also have to continuously add and remove vertex weights for the ever-changing face structure. MRM does not have this problem since all the possible faces are known beforehand.

If subdivision follows animation, it will have to deal with continuously changing vertices and normals, which may spoil the ability to pre-compute or cache subdivision results. Further development will hopefully allow either or both of these options to work.

14 Caching and Performance

For run-time deformation to be useful it must be very fast. If a frame takes about 30 milliseconds, we can’t expect the animation system to consume more than 3 or 4 milliseconds. To optimize the use of caching, we use two structures.

The first structure is the bone transform cache. This is an array of 3x4 matrix transforms, one for each bone including the bone links. After all motion and IK is resolved for the frame each bone’s displacement and quaternion rotation are converted to 3x4 matrix. The fourth column of the conventional 4x4 matrix is discarded since the additional effects it can produce are rarely desirable. We could actually store the transform as a 3-vector and a quaternion (total of 7 floats) instead of the 3x4 matrix (12 floats). But it turns out that although quaternions multiply with each other efficiently and interpolate nicely, it takes considerable more operations to transform a vector using a quaternion than a matrix (for rotation only, a matrix uses 9 multiplies and 6 adds; a quaternion uses 32 multiplies and 24 adds).

The second structure is the vertex weight array. Each entry contains the submesh index, the vertex index, the bone index (which maps directly into the bone cache), the fractional weight, the vertex location, and the normal direction. By storing the vertex and normal data here, it not only gives us the single mesh capability mentioned in the MRM section, but it means we never read any mesh after initialization. The structure is sorted so that all entries related to a specific vertex are grouped together. Because of this we only need to write to the mesh once per vertex even if there are multiple contributions. Otherwise, we would have to reset all the vertex locations and normals at the start and then add a weighted contribution for every bone that influences each vertex. So, by accumulating the weighting contributions in local variables, we not only avoid the clearing stage, we reduce the number of mesh writes by the average number of influences per vertex.

Note that during the core deformation loop, both these structures are never written to and the mesh is never read from. So, we never have to wait on our own output.

Currently, on a test model with 3910 vertices/normals and 7100 faces, the full deformation process at full resolution takes about 3.2 ms per frame. Broken down, this includes:

- 0.07 ms to interpolate and apply the motion

- 0.11 ms to re-compute the bone link positions

- 0.09 ms to fill the bone transform cache including quaternion to matrix conversions

- 2.04 ms to apply and accumulate all the transformed vertices and normals

- 0.92 ms to re-normalize the normals

The re-normalization step, which takes up about 30% of the total animation time, can optionally be skipped. If you trace the math into the lighting equations, the result would presumably be slight dimming around bent joints. This effect has been verified through visual observation.

Also, as mentioned before, MRM de-resolution reduces animation deformation time. Therefore, the same algorithms that adjust mesh rendering performance also affect animation performance.

Performance was measured on a 500-MHz Intel® Pentium® III processor-based system.

15 Hardware Compatibility

There is support for vertex blending in recent PC graphics boards. However, we need to rearrange our math to accommodate these implementations. The main difference is how we present the transform matrices to the graphics API. In previous discussion, we discussed how we compute offsets of each vertex from each of its contributing bones' reference positions. Then, we run each vertex offset through each bones current transform and accumulate the results.

But for graphics API and hardware support, we can not have all these different offsets for each vertex. We just use the original world vertex location found in the reference mesh (we may prefer to cache them). The previously mentioned step of finding the bone-relative locations during initialization can be equivalently achieved by pre-multiplying the inverse of each bones reference transform to its current transform for every frame and storing the result in the bone transform cache used for deformation. See [Lander 99-10].

Another restriction is that you can only load a limited number of transforms into the system at once. The mesh has to be broken down into pieces that use the same subset of the total list of transforms. This can be quite restrictive if only a few number of transforms can be loaded at once.

It is debatable how much benefit will come from offloading these operations onto the graphics API and hardware. Following these strict guidelines may be acceptable, but the resultant location of a vertex position is now deep into a pipeline, which you probably can not access without an appreciable loss of performance, if you can read it at all. This eliminates the ability to make some aspect of your program dependent on one of these resultant vertex positions, such as attaching a non-integrated object to a specific vertex on the character. It can also prevent special rendering techniques such as Non-Photorealistic Rendering (NPR) that will need to know the locations of all the altered vertices and normals for silhouettes and alternate shading.

We have added and tested the matrix manipulation changes mentioned but have not tested hardware compatibility since we have not yet built the capability to divide up the mesh into transform groups at load time.

16 Future Possibilities

16.1 Multiple reference positions

It would appear that the algorithms could greatly benefit from data from multiple reference positions, presumably at some of the extreme angles. The systems can then additionally blend between vertex position of the meshes associated with the different reference positions.

16.2 Separate deformation for bending and twisting

One clear problem with defining vertex weights is that real bodies don't tend to deform the same in twisting and bending motions. For example, if you bend your hand up and down about your wrist, the deformation occurs only in a small region around the wrist. However, if you twist your hand by rotating it about the axis of your forearm, the deformation occurs all the way from the wrist back to the parent elbow. Similarly, a shoulder twists rather differently than it bends, but not like the wrist. It twists forwards to the elbow and not back to a parent.

It is apparent that a single set of overall vertex weights may be a bit naïve. Perhaps a set of twisting weights and a set of bending weights could be blended together. This would put more burden on the author but may achieve a more desirable effect.

16.3 Clothing

The ability to use physically based cloth on animated characters will help with overall realism. This effort will not only involve building a fast constrained particle system to move the cloth, but finding fast and reliable methods to apply the cloth to the characters.



Figure 15 Prototype Real-Time Cloth Patch

Acknowledgements

The “harley” model used in many of the images was provided by Discreet, a division of Autodesk**, makers of the 3D Studio MAX** software suite.

Thanks to Keith Feher for the duck model.

Bibliography

[Bobick98-2] Bobick, Nick, “Rotating Objects Using Quaternions”, Game Developer Magazine, Feb 1998, pp. 34-42.

[Lander98-05] Lander, Jeff, “Skin Them Bones: Game Programming for the Web Generation”, Game Developer Magazine, May 1998, pp. 11-16.

[Lander98-11] Lander, Jeff, “Making Kine More Flexible”, Game Developer Magazine, Nov 1998, pp. 15-22.

[Lander99-10] Lander, Jeff, “Over My Dead, Polygonal Body”, Game Developer Magazine, Oct 1999, pp. 17-22.

[Welman93] Welman, Chris, “Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation”, Masters Thesis, Simon Fraser University, Sept 1993.

Copyright © 2000 Intel Corporation. All rights reserved.

**Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner’s benefit, without intent to infringe.