# Slashing Through Real-Time Character Animation

Last time we left off, you were hanging out on a bridge in quaternion space. Quaternion space is like the space between the handout desk and the deal room at any booth of any publisher at E3. It's a concept you can grasp, but it's really tough to visualize being there.



As you'll recall, I finished up with an OpenGL application that converts Euler angle data to quaternions and then displays the results. Now I want to extend this application to allow for the interpolation of two keyframed positions. It struck me that I should create a custom 3D first-person interface that effectively demonstrates interpolation.

## The Task

As the project technical lead, I am asked to create an interface for a first-person fighting game. However, the design calls for allowing the player to create custom attacks in some sort of pregame editor. The player does this by manipulating an arm consisting of an upper arm, a lower arm, and a hand with a weapon. The player positions this arm into two poses. One pose is the beginning of the attack move and the other pose is the end of the attack move. During the game, this custom action is triggered and creates a smooth attack. The player's effective use of this interface determines the effectiveness of the move. Several of these moves are then combined to create a unique fighting experience.

Alright, so it's not revolutionary, but it's a well-defined task with a pretty clear path of attack. As technical lead, I like that. So how do I get started?

*When not bending the bones of some strange alien creature, Jeff can be found hanging out at his studio at the beach. See if you can smack some sense into him by writing to jeffl@darwin3d.com.*

Clearly, the problem revolves around the interpolation of the arm positions.

## Interpolation

As I discussed last month, one of the key benefits of using a quaternion representation is the ability to interpolate between keyframes. Nick Bobick, in his article in the February 1998 issue of *Game Developer*, discussed interpolation of quaternions ("Rotating Objects Using Quaternions," pp.38-39). Bobick described the use of Spherical Linear Interpolation (SLERP) to achieve smooth interpolation. For very small interpolation, he mentioned that it's a good idea to use simple Linear Interpolation (LERP). Being the hardcore game programmers that you are, you may ask, "Why not use LERPs all the time and avoid the expensive math that SLERPs require?" The reason is that unit length quaternions describe a 4D hypersphere. If I were simply to interpolate between the two keyframes in a straight line, I would be cutting across the arc of that sphere (Figure 1a). As you can see, in-betweens that are evenly spaced on the hypersphere create nonlinear positions on the LERP-line. Alternately, the effect of evenly spacing out in-betweens along the LERP-line would create an animation that would appear to move faster as it traveled across the middle of the interpolation (Figure 1b). This may not always be a bad thing, so it's quite easy to adjust between LERP and SLERP.

The code in Listing 1 gives me the basis for creating my 3D interface. By applying these routines to a three-bone hierarchy, I get a smooth attack from any two keyframed positions. To implement this in OpenGL, I only needed to modify my display routine a little bit. The critical section is in Listing 2, and you can see the results in Figures 2a-c.

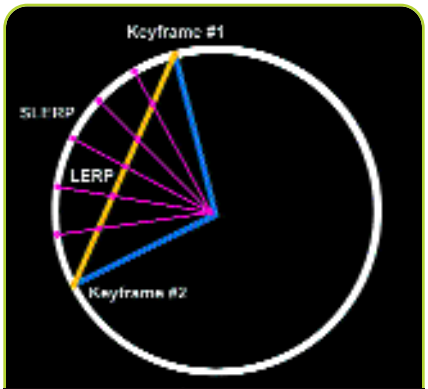



**FIGURE 1a.** *Spherical Linear Interpolation (SLERP) between quaternions.*

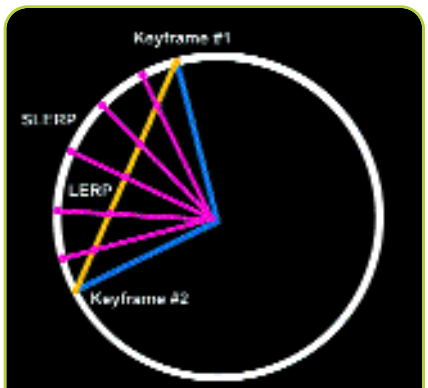


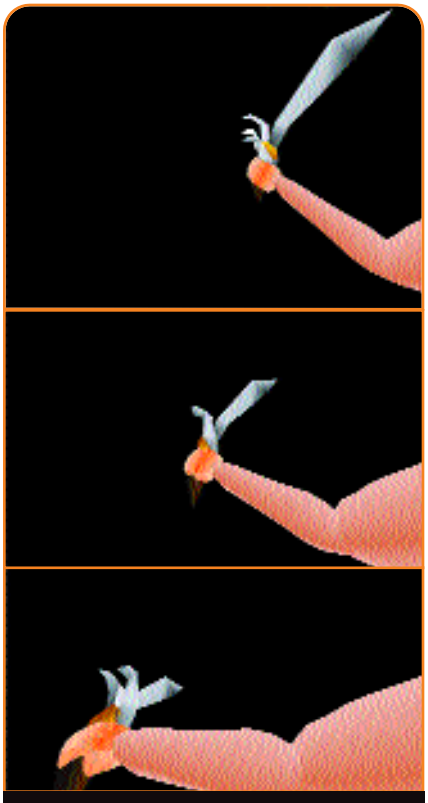

**FIGURE 1b.** *Linear Interpolation (LERP) between quaternions.*

## Let's Recap

Let me take a moment to recap what I've covered in my past two columns. I started by taking motion capture data and applying it to a skeletal system. I then created a method for converting Euler angles to quaternions. Then, by using quaternion interpolation, I created smooth in-betweens for the skeletal system. Lastly, to make things look a bit more interesting, I attached 3D objects to individual bones.

One remaining problem is that the arm is created from three separate objects — and it shows. The points at which the different objects are connected look a little rough. This problem has been plaguing real-time 3D graphics for some time now. In fact, many successful games live with this and get quite good results (VIRTUA FIGHTER, TOMB RAIDER, and JEDI KNIGHT come to mind). However, to combat this problem, many artists design their characters to disguise the fact that the bones are composed of separate objects. This is done through clever texturing or modeling, and explains why so many real-time 3D characters wear armor or tank top shirts.

If the character was created from one single mesh, we wouldn't have any of the problems that separate objects create. In a 3D graphics package such as Softimage, Alias, or 3D Studio MAX, I could create a mesh and animate it with the software's skeletal



**FIGURES 2a-c.** *Keyframe #1, 50 percent interpolation, and keyframe #2.*

**LISTING 1.** *Interpolation of Two Quaternions.*

```
////////////////////////////////////////////////////////////////////////////
// Function:        SlerpQuat
// Purpose:         Spherical Linear Interpolation Between two Quaternions
// Arguments:       Two Quaternions, blend factor, result quaternion
// Source:          Watt and Watt, Advanced Animation, p. 364
//                  NOTE: This fixes a bug in their code
////////////////////////////////////////////////////////////////////////////
void SlerpQuat(tQuaternion *quat1,tQuaternion *quat2,float slerp, tQuaternion *result)
{
/// Local Variables ///////////////////////////////////////////////////////
        tQuaternion quat1b;
        double omega,cosom,sinom,scale0,scale1;
////////////////////////////////////////////////////////////////////////////
        // USE THE DOT PRODUCT TO GET THE COSINE OF THE ANGLE BETWEEN THE QUATERNIONS
        cosom = quat1->x * quat2->x +
                        quat1->y * quat2->y +
                        quat1->z * quat2->z +
                        quat1->w * quat2->w;

        // MAKE SURE WE ARE TRAVELING ALONG THE SHORTER PATH
        if ((1.0 + cosom) > DELTA)
        {
                // IF THE ANGLE IS NOT TOO SMALL, USE A SLERP
                if ((1.0 - cosom) > DELTA) {
                        omega = acos(cosom);
                        sinom = sin(omega);
                        scale0 = sin((1.0 - slerp) * omega) / sinom;
                        scale1 = sin(slerp * omega) / sinom;
                } else {
                        // FOR SMALL ANGLES, USE A LERP
                        scale0 = 1.0 - slerp;
                        scale1 = slerp;
                }
                result->x = scale0 * quat1->x + scale1 * quat2->x;
                result->y = scale0 * quat1->y + scale1 * quat2->y;
                result->z = scale0 * quat1->z + scale1 * quat2->z;
                result->w = scale0 * quat1->w + scale1 * quat2->w;
        } else {
                // SINCE WE FOUND THE LONG WAY AROUND, USE THE SHORTER ROUTE
                result->x = -quat2->y;
                result->y = quat2->x;
                result->z = -quat2->w;
                result->w = quat2->z;
                scale0 = sin((1.0 - slerp) * (float)HALF_PI);
                scale1 = sin(slerp * (float)HALF_PI);
                // MULT BY THE SCALE
                result->x = scale0 * quat1->x + scale1 * result->x;
                result->y = scale0 * quat1->y + scale1 * result->y;
                result->z = scale0 * quat1->z + scale1 * result->z;
                result->w = scale0 * quat1->w + scale1 * result->w;
        }
}
// SlerpQuat ////////////////////////////////////////////////////////////////
```

**LISTING 2.** *Applying quaternion rotation in OpenGL.*

```
// CONVERT THE TWO KEYFRAME ROTATIONS TO QUATERNIONS
EulerToQuaternion(&curBone->p_rot,&primaryQuat);
EulerToQuaternion(&curBone->s_rot,&secondaryQuat);
// INTERPOLATE BETWEEN THEM BY A BLEND FACTOR 0.0 - 1.0
SlerpQuat(&primaryQuat,&secondaryQuat,m_AnimBlend,&curBone->quat);
// QUATERNION HAS TO BE CONVERTED TO AN AXIS/ANGLE REPRESENTATION
QuatToAxisAngle(&curBone->quat,&axisAngle);
// DO THE ROTATION
glRotatef(axisAngle.w, axisAngle.x, axisAngle.y, axisAngle.z);
```

deformation system. This would create very seamless characters that could be animated very quickly by a game engine. In fact, this method has been used by QUAKE (and its genetic offspring) quite successfully.

However, by predeforming the characters to animate them, I lose the key benefit of real-time 3D — flexibility. By sticking to a hierarchy of bones, I'm able to apply unique motion capture data, interpolate between keyframes, and incorporate many things that I haven't talked about, such as real-time inverse kinematics, dynamics, and motion blending. So how do I achieve the key benefits of a skeleton without having the ugly seams that come with it?

## Skin Them Bones

The answer is to stretch a single skin over the bones in the skeleton. While this is a fairly advanced feature in most 3D modeling packages, the technique behind it is really quite easy. As a good starting point, let's consider associating each vertex in the skin mesh with an individual bone. The influence of that bone directly effects its associated vertex . Thus, when you rotate a bone, it rotates the associated vertices about the root position of that bone. You can see the effect of this process in Figure 3. In this image, two bones define the hierarchy, and each bone has eight vertices associated with it. While this technique creates a seamless, deformable mesh with very little processor overhead, it has one drawback. At the point where the two bones meet, the skin is stretched a bit. While this may be fine for many applications, with extreme motion, this stretch is very unrealistic.

The solution to this problem is to add a few more vertices to the model and "weight" the individual vertices. This means that for each vertex in the model, you assign a certain percentage of its influence to each bone. While many vertices may be assigned 100 percent to an individual bone, some may be assigned 50/50 between two bones. By blending the influence of different bones, you can achieve a very smooth skin. In some extreme cases, you may even need to weight a vertex between three or more bones, but in general, two is sufficient.

Figure 4 shows how a fully weighted mesh could be applied to the same two-bone system. Because of the calculations needed to handle the weighting, this system is a bit more processor intensive then basic skinning. However, for a main character or opponent, the smooth results and flexibility are worth the increased processor load.

The remaining question is, How do I implement a fully weighted mesh applied to a hierarchical skeleton in an immediate mode API such as OpenGL? Well, that's going to be the topic for next time. However, some of you may be itching to get started. Since I have now covered all of the main pieces, for your homework, see if you can figure out an efficient way to calculate those vertex positions, and I will work it out next month. Try to wrap your brain around the underlying concept, and then we'll work out the details together next month.

The sample application for this month (on the *Game Developer* website at www.gdmag.com) allows you to keyframe two positions for a three-bone arm and interpolate smoothly between them. ■

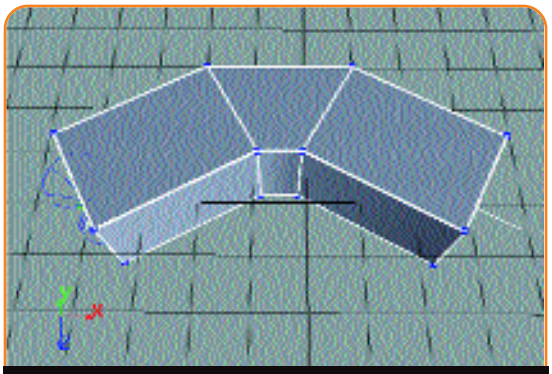


**FIGURE 3.** *Two bones weighted 100 percent to each vertex.*

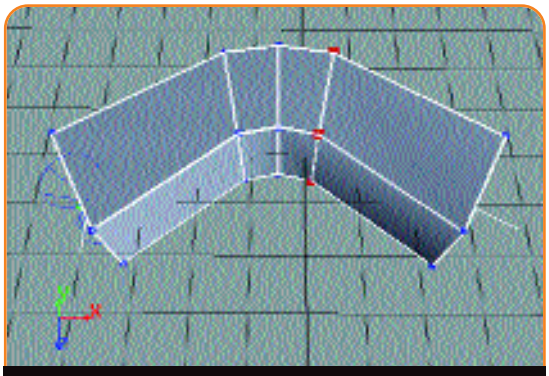


**FIGURE 4.** *Two bones weighted to each vertex 100-66/33-50/50-33/66-100.*

### REFERENCES

For my quaternion SLERP code, I have used *Advanced Animation and Rendering Techniques*,(ACM Press, 1992) by Watt and Watt as a starting point. This is a very good book that covers many topics not touched by any other text. However, as I've tried different concepts in the book, I have found a few errors. The code sample on interpolating quaternions is a case in point. When negating an input quaternion to find the shortest arc, they negate the wrong one. The source that accompanies this article corrects that error. Problems such as these are a good reason to study the underlying concepts and follow sources for any new technique. This can save you a great deal of frustration when things do not work out as they should.