MARCH 2002
Game Developer Mag

Hacking Quaternions

Quaternions are a nifty way to represent rotations in 3D space. You can find many introductions to quaternions out there on the Internet, so I'm going to assume you know the basics. For a refresher, see the papers by Shoemake or Eberly in For More Information. In this article we will look closely at the tasks of quaternion interpolation and normalization, and we'll develop some good tricks.

Interpolation

When game programmers want to interpolate between quaternions, they tend to copy Ken Shoemake's code without really understanding it (hey, that's what I did at first!). Ken uses a function called slerp that walks along the unit sphere in four-dimensional space from one quaternion to the other. Because it's navigating a sphere, it involves a fair amount of trigonometry, and is correspondingly slow.

Lacking a strong grasp of quaternions, most game developers just accept this: slerp is slow, and if you want something faster, maybe you should go back to Euler angles and all their nastiness. But the situation is not so bad. There's a cheap approximation to slerp that will work in most cases, and is so brain-dead simple and fast that it's shocking. Shocking, I tell you.

What Slerp Does

Slerp is desirable because of two main properties; any approximation we formulate would ideally have the same properties. The first, and perhaps most important, is that slerp produces the shortest path between the two orientations on that unit sphere in 4D; this is equivalent to finding the "minimum torque" rotation in 3D space, which you can think of as the smoothest transition between two orientations. The second property of slerp is that it travels this path at a constant speed, which basically means you have full control over the nature of the transition. (If you want to add some style, like starting slowly and then speeding up, you can just spline your time parameter before feeding it into slerp.)

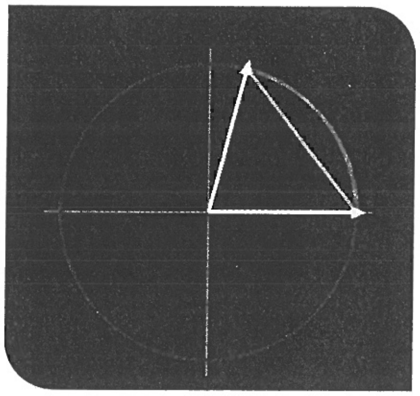


FIGURE 1. A two-dimensional picture of quaternion interpolation. The blue circle is the unit sphere; the two yellow vectors are the quaternions. The red arc represents the path traveled by slerp; the green chord shows the path taken by linear interpolation.

So here's our approximation: linearly interpolate the two quaternions componentwise. That is, if t is your time parameter from 0 to 1, then $x = x_0 + t(x_1 - x_0)$, and similarly for y, z, and w.

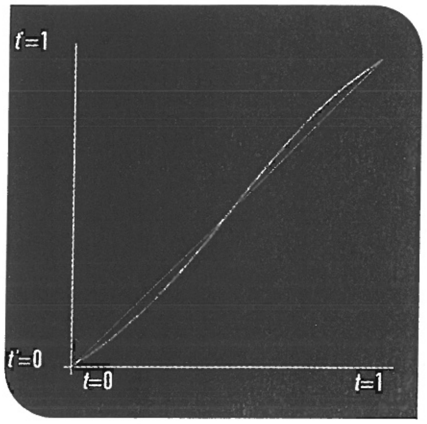
One might wonder how that could possibly be a worthwhile interpolation when the right answer is so much more complicated. Let's take a look at why that is.

Figure 1 shows a two-dimensional version of quaternion interpolation. Slerp walks around the edge of the unit circle, which is what we want. Linear interpolation results in a chord that cuts inside the circle. But here's the thing to realize: Normalizing all the points along the chord stretches them out to unit length, so that they lie along the slerp path. In other words, if you linearly interpolate two quaternions from $t = 0$ to $t = 1$ and then normalize the result, you get the same minimal-torque transition that slerp would have given you.

The linear algebra way to see this is that both the great circle and the chord lie in Span(q_0, q_1), which is a 2D subspace of the 4D embedding space. Adding the constraint that Length(Interpolate(q_0, q_1, t)) = 1 reduces the dimensionality to one, so both paths must lie along the same circle. And both forms of interpolation produce only a continuous path of points between q_0 and q_1, so they must be the same.

If q_0 and q_1 lie on opposing points of the sphere, the chord will pass through the origin and normalization will be undefined. But that's O.K. — unless you're doing something wacky, you don't

JONATHAN BLOW | *Jon (jon@bolt-action.com) would like you to know that the Slimelight in London is a horrible nightclub that really isn't any fun.*



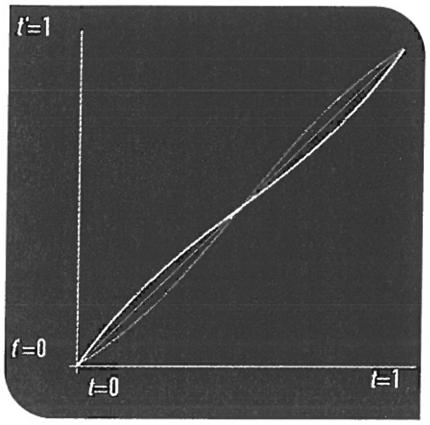


FIGURE 2 (left). Worst case of lerp speed variation. The green line represents the ideal result produced by slerp; the red line represents the distorted result produced by lerp. The error between these two functions should be measured vertically, so they are more different than they may appear at first. FIGURE 3 (right). The compensating cubic spline, $k = 0.45$, shown in yellow atop the graph of Figure 2.

want your quaternions to be more than 90 degrees apart in the first place (because every rotation has two quaternion representations on the unit sphere, and you want to pick the closest ones to interpolate between). So the normalization will always be well defined.

Thus the normalized linear interpolation and the slerp both trace out the same path. There is a difference between them, though: they travel at differing speeds. The linear interpolation will move quickly at the endpoints and slowly in the middle. Figure 2 shows a graph of the worst case, 90 degrees.

The function graphed in Figure 2 is roughly

$$\tan^{-1}\frac{t\sin\alpha}{1+t(\cos\alpha-1)}$$

where α is the original angle between the two quaternions. I figured this out just by drawing a 2D graph like Figure 1, where one of my vectors is the x-axis (1, 0) and the other one is ($\cos\alpha$, $\sin\alpha$). Then I just wrote an expression for linearly interpolating between them by t, and then finding the resulting angle by $\tan^{-1}$. This rather simplistic approach is valid for two reasons. First, since all the action happens in Span(q_0, q_1), we can just take that 2D cross section out of 4D space; studying it in isolation, we see the entirety of what is happening. Second, on the resulting 2D unit circle, because the set of all possibilities for the two unit vectors is redundant by rotational symmetry, we can choose one of the vectors to be anything we like; I chose (1, 0) to simplify the math.

Casey Muratori of RAD Game Tools is the first person I know of who considered linear interpolation of quaternions as a serious option. He investigated numerically and found linear interpolation, when properly employed, to be quite worthwhile. Casey has eradicated all slerps from his code for Granny 2.

Augmenting Linear Interpolation

The linear interpolation is monotonic from q_1 to q_2, so if you are doing an application where you're binary searching for a result that satisfies some constraint, using the linear interpolation works just fine. If your quaternions are very close together (less than 30 degrees, say), as you have when playing back a series of time-sampled animation data, linear interpolation works fine. And if you have some number of different character poses (like an enemy pointing a gun in several different directions), and you want to mix them based on a blending parameter, linear interpolation probably works fine.

Linear interpolation won't work if you need precise speed control and wide interpolation angles. But maybe we can fix that.

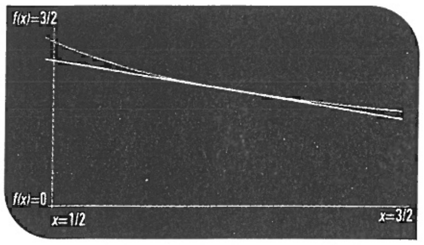
Perhaps we can make a spline that cancels most of the speed distortion. Looking at Figure 2, can we concoct a function that, when multiplied against the curve, causes it to lie much closer to the ideal line? The way I chose to visualize this was with a cubic spline that tries to pull the distortion function onto the diagonal. Figure 3 shows a cubic spline with the equation $y = 2kt^3 - 3kt^2 + (1 + k)t$, where the tuning parameter $k = 0.45$ has been graphed against the plot of Figure 2.

Because both the distortion curve and our compensating spline have an average value of t and are approximately complementary, when we multiply them together we get a function that is approximately $g(t) = t^2$. We want $g(t) = t$, so we'll divide the cubic spline by t. Fortunately, since the spline passes through the origin, it has no d coefficient; so dividing by t just turns it into a quadratic curve: $y = 2kt^2 - 3kt + 1 + k$.

So now, if we're linearly interpolating two splines that are 90 degrees apart, we find $t' = 2kt^2 - 3kt + 1 + k$, and use t' as our interpolation parameter. We get something very close to constant-speed interpolation (I will quantify how close in a little bit). However, if we reduce the angle between the input quaternions, we get something that's less accurate than the original t.

That's because, by defining its slope at $t = 0$ and $t = 1$, I concocted this spline specifically for the worst-case scenario. That's where the k parameter comes in: it's a slope-control mechanism. To get this spline to compensate for distortion across the full range of quaternion input angles, we want to adjust the tuning parameter as some easily computable function of the angle between the two quaternions.

Well, taking the dot product of two quaternions gives us $\cos\alpha$, the cosine of the angle between them. I started playing around with simple functions of $\cos\alpha$ until I found something reasonable. Basically, we want a function that is 1 when $\cos\alpha = 0$, and that is near 0 when $\cos\alpha = 1$. After some experimentation I



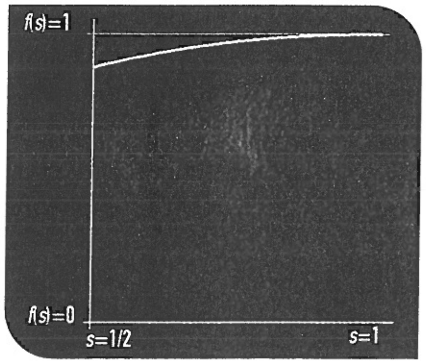


FIGURE 4 (top). In green, the function $f(x) = 1/\sqrt{x}$ in yellow, its tangent line at $x = 1$. FIGURE 5 (bottom). The length of an approximately normalized vector (yellow), versus the squared length of the input, when using the naive tangent line approximation. The green line indicates the ideal result.

landed on $k = 0.45(1 - s \cos \alpha)^2$, where $s = 0.9$ for now. To cursory visual inspection, this gives pretty good results across the full range of α from 0 to 90 degrees.

These numbers are in the right neighborhood, but because I just made them up, they're not going to be as close as we can get. So I wrote some code to do hill-climbing least-squares minimization. The initial distortion function has an RMS error of about 1.6×10^{-2} when averaged over all interpolation sizes (the worst case, graphed in Figure 2, has an RMS error of 3.234×10^{-2}). The minimizer gave me the following values: $k = 0.5069269$, $s = 0.7878088$, yielding an overall error of 2.07×10^{-3}, which is about eight times lower than we'd started with (see Listing 1).

But while I had been aligning things by eye, I noticed that if I gave k a high value, I got results that were close to exact from $t = 0$ to $t = 0.5$, but diverged after $t = 0.5$. So I wrote an interpolator that only needs to evaluate t from 0 to 0.5. If you pass in a t higher than 0.5, it just swaps the endpoints of interpolation. Running the optimizer on this, I got $k = 0.5855064$, $s = 0.8228677$, overall error 5.85×10^{-4} — a reduction of more than 27 from the original. We incur another small cost to gain this accuracy, an extra `if` statement.

You can probably do better than these numbers; my methods were ad hoc, and there are many possibilities I haven't explored. I should also give a few warnings. For example, the `if` statement I just mentioned introduces a slight discontinuity at $t = 0.5$; you can fix this discontinuity by shifting the midpoint away from 0.5, but this wasn't important for my needs.

So we can interpolate pretty quickly now, but we end up with non-unit quaternions. We probably want unit quaternions, so how do we normalize without doing a really slow inverse square root operation?

Normalization

To normalize any vector, quaternions included, we want to divide the vector by its length. The squared length of some vector v is cheap to compute — it's v·v — so we need to obtain $1/\sqrt{v \cdot v}$ and multiply the vector by that. Division and square-rooting are pretty expensive, though.

We can compute a fast $1/\sqrt{x}$ by using a tangent-line approximation to the function. This is like a really simple one-step Newton-Raphson iteration, and by tuning it for our specific case, we can achieve high accuracy for cheap. (A Newton-Raphson iteration is how specialized instruction sets like 3DNow and SSE compute fast inverse square root).

The basic idea is that we graph the function $1/\sqrt{x}$, locate some neighborhood that we're interested in, and pretend that the function is linear there. A linear function is cheap to evaluate.

So, we want to approximate $f(x) = 1\sqrt{x}$. We are interested in vectors whose lengths are somewhere near 1, meaning $f(x) = 1$, which means $x = 1$. So we are going to focus on the neighborhood $x = 1$, as you see in Figure 4. To get the line, we just take the derivative of f, $f'(x) = -\frac{1}{2}x^{-\frac{3}{2}}$, and evaluate it at 1:

$$f'(1) = -\frac{1}{2}$$

An equation that says "locally, a function is approximately its value at some point plus its first derivative extrapolated over distance" is:

$$f(x + \Delta x) \approx f(x) + \Delta x f'(x)$$

We evaluate this at $x = 1$ to get

$$f(1 + \Delta x) \approx f(1) + \Delta x f'(1) = 1 - \frac{1}{2}\Delta x$$

Now for the last trick: we want to represent the squared length of our input vector, which we'll call s, as a value in the neighborhood of 1, so we can plug it

LISTING 1. A function that splines t to compensate for the distortion induced by lerping.

```
float correction(float t, double alpha, double k, double attenuation) {
    double factor = 1 - attenuation * cos(alpha);
    factor *= factor;
    k *= factor;

    float b = 2 * k;
    float c = -3 * k;
    float d = 1 + k;

    double t_prime = t * (b*t + c) + d;

    return t_prime;
}
```

into our new linear function. We say $s = 1 + \Delta x$, and thus $\Delta x = s - 1$.

That is all we need. When we plug $\Delta x = s - 1$ into our approximation, we get

$$f(1+s-1) \approx 1 - \frac{1}{2}(s-1)$$

Simplified, this says:

$$f(s) \approx \frac{1}{2}(3-s)$$

For as wide of a neighborhood as the inverse square root is well approximated by a tangent line, this extremely fast computation will give us the factor to normalize a vector. Figure 5 graphs the vector lengths we get when we use this computation to normalize. As long as we start with a vector whose length is near 1, we get results that are fairly accurate.

For some applications, accuracy in a narrow range is all we need. If you are reconstructing quaternions from splines, as one might do in a skeletal animation system that stores animation data in a small memory footprint, you can ensure a maximum length deviation during the spline-fitting process (inserting extra keyframes to alleviate any problems). Then at run time you just evaluate the splines and pump the coefficients into this one-step normalizer, and you can be assured that the results are good.

On the other hand, this isn't good enough to use blindly on the results of quaternion linear interpolation. We can see that, during our worst-case interpolation from (1, 0) to (0, 1), the closest we get to the origin is (1/2, 1/2), which gives us a squared vector length $s = 1/2$. So for good results after lerping, we need a fast normalizer that produces good results all the way through the interval from $s = 1/2$ to $s = 1$.

Retuning the Tangent Line Approximation

When we linearly interpolate quaternions, we get a chord that cuts through the unit sphere; that is, the resulting length is always less than 1. So we don't need our linear approximation to be accurate above 1. We can, in effect, slide the graph of Figure 5 to the left a little bit, making our approximation more effective for shorter vectors.

Also, if we are going to permit some small amount of error ε in our result, it probably makes sense to allow results in the range $1 \pm \varepsilon$, instead of just $1 - \varepsilon$ as in Figure 4. So we can scale the approximation by some small factor. This roughly doubles the zone of good results.

But this still doesn't cover the full range from 1/2 to 1. A simple solution would be to just check the value of s, and if it is too low, just compute the answer the slow way. For most applications, wide-angle interpolations will be extremely rare, so the speed hit will be small. But if you need to be faster than that, there are some hackish things we can do.

I wrote some code that repeatedly applies the fast normalization, tuned by some optimization parameters, in order to achieve the least error across the interval we are interested in. Running the numerical optimizer on this yields $x_{\text{offset}} = 0.959066$, scale = 1.000311, and a root-mean-square error of 2.15×10^{-4}. This loop only iterates at most three times over the interval we care about, so you can rephrase the loop as a small series of nested `if` statements, which are mostly never descended into (see Listing 2).

LISTING 2. A fast normalizer.

```
inline float isqrt_approx_in_neighborhood(float s) {
    const float NEIGHBORHOOD = 0.959066;
    const float SCALE = 1.000311;
    const float ADDITIVE_CONSTANT = SCALE / sqrt(NEIGHBORHOOD);
    const float FACTOR = SCALE * (-0.5 / (NEIGHBORHOOD * sqrt(NEIGHBORHOOD)));

    return ADDITIVE_CONSTANT + FACTOR * (s - NEIGHBORHOOD);
}

inline void fast_normalize(float vector[3]) {
    float s = vector[0]*vector[0] + vector[1]*vector[1] + vector[2]*vector[2];
    float k = isqrt_approx_in_neighborhood(s);

    if (s < 0.83042395) {
        k *= isqrt_approx_in_neighborhood(s);

        if (s < 0.30174562) {
            k *= isqrt_approx_in_neighborhood(s);
        }
    }

    vector[0] *= k;
    vector[1] *= k;
    vector[2] *= k;
}
```

Sample Code

This month's sample code implements fast linear interpolation and renormalization, as well as the numerical optimization code that computes the best parameters. Download it from www.gdmag.com.

FOR MORE INFORMATION

Eberly, David. "Quaternion Algebra and Calculus." www.magic-software.com/Documentation/quat.pdf

Shoemake, Ken. "Animating Rotation with Quaternion Curves." *Computer Graphics* Vol. 19, No. 3 (July 1985).