David Goodhue
Advanced Technology Division
Square Enix, Co. LTD

Optimizing CPU Performance and Maximizing Data Precision for Velocity-based Animation Data

Problem

Simply implementing the necessary algorithms needed for playback of velocity-based animation in an intuitive manner would generally not result in very good CPU performance. Below is an explanation of two alternate solutions which can be used to bypass many of the heavier computations.

The Straightforward Solution

As most game animation systems are engineered to handle updating via variable framerate, velocities are repeatedly applied to the previous keyframe's pose as rotational transformations as such:

```
currentPose = prevKeyframePose * scaledVelocity;
```

Intuitively, this would be applied as a quaternion multiply, where scaledVelocity is a quaternion constructed from the velocity axis and an angle which is essentially the angular velocity's speed component scaled by elapsed time. Given our speed represented as a floating point value in units of radians/sec, or alternatively radians/frame, that leaves us with:

```
angle = speed*(currentTime – prevKeyframeTime);
```

and

```
scaledVelocity = BuildFromAxisAngle(velocityAxis, angle);
```

However, building quaternions from an axis and angle incurs a high CPU cost. The quaternion multiply to apply scaledVelocity to prevKeyFramePose adds to that. Therefore, while perhaps the most intuitive solution, this method will not yield the performance we hope to achieve.

Alternate Solution #1

Rather than multiplying some additive application of velocity with the previous keyframe's pose as is done above, we can extrapolate out our rotational trajectory to some future time and slerp, or perhaps instead elect to use our favorite efficient algorithm for interpolating between two rotations. And since most popular interpolation algorithms traverse the shortest path between two rotations,

we need to make sure that the extrapolated goal is no more than 180° distance along our rotational trajectory. In fact, using the full 180° can be convenient, maximizing how far out we can interpolate while also providing us a fairly cheap way to build our extrapolated value which will be cached and reused for future updates up until the next key is encountered. Building this value is a bit cheaper than an ordinary quaternion multiplication, as seen in this C++ example:

```
// This is a quaternion multiply where b.W()==0 because angle==180°
Quaternion a = prevKeyframePose;
Vector3 b = velocityAxis;

float aX = a.X(), aY = a.Y(), aZ = a.Z(), aW = a.W();
float bX = b.X(), bY = b.Y(),bZ = b.Z();

Quaternion e; // extrapolated rotation
e.X() = aW*bX + aY*bZ - aZ*bY;
e.Y() = aW*bY + aZ*bX - aX*bZ;
e.Z() = aW*bZ + aX*bY - aY*bX;
e.W() = -aX*bX - aY*bY - aZ*bZ;
```

A typical slerp algorithm can be simplified to assume that the two inputs are 180° apart, leaving only this for consecutive updates between the same two keys:

```
float halfAng= speed*(currentTime – prevKeyframeTime);
float sc1 = Sin(kHalfPI-halfAng);
float sc2 = Sin(halfAng)
Quaternion currentPose = sc1*prevKeyframePose + sc2*e;
```

Since Sin is a costly function to call, a polynomial approximation of Sin may be preferred. Or, instead of a slerp-based interpolation, something like OnLerp [Kapoulkine 2016] makes a nice solution. So long as the interpolated result is “close enough”, there is no penalty for not being perfect. Any pose error should be verified when compressing animation data offline, and more keys can always be inserted wherever the resultant pose error would otherwise be unacceptable. In general, one will wish to tune their interpolator such that the amount of extra keys required to compensate for approximation-induced error is of little significance.

Very often, in a typical dataset, we’ll be interpolating between the same two keys repeatedly for a number of frames before eventually moving on to the next pair of keys, so it’s also reasonable to compute some values whenever a new key is encountered, cache them in memory, and reuse those cached values for subsequent updates between those same two keys if it makes the interpolation cheaper.

Alternate Solution #2

This significantly different approach first requires that we first envision our pose rotations not as quaternions but as values stored in exponential map space [Grassia 1998]. Using exponential maps, approximate interpolation between poses can be achieved with a simple lerp. However, it becomes a very poor approximation when interpolating across large angles, so the idea of extrapolating far ahead as in Alternate Solution #1 is not practical. However, if in our offline data preparation, we perform our extrapolation out to only as far as the next keyframe while also ensuring that that keyframe is not so far away that the exponential map interpolation becomes too inaccurate, we can then rethink our "velocity delta" to instead be an "exponential map delta". As such, we would know that when applied it can yield an accurate pose at the time of our next keyframe. For example, if our velocity in solution #1 was stored in units of angle/time, the analogue to that here would be an exponential map delta computed as:

```
delta = (keyframePoseExpMap - prevKeyframePoseExpMap)/
        (keyframeTime – prevKeyframeTime);
```

And applying that delta in subsequent updates is as simple as

```
currentPoseExpMap = prevKeyframePoseExpMap +
                    (currentTime - prevKeyframeTime)*delta;
```

Using this solution, the entire animation decompression module can occur in exponential map space at a very low cost. It is very important that the original computation of each delta, which again, happens only during offline data preparation, be based on some future keyframe time, not just some arbitrary time or angle extrapolation into the future. This is because of how scaling and applying exponential map deltas using addition is not an accurate replacement for applying angular velocities as actual proper rotational transformations. An exponential map delta is, unlike a true angular velocity delta, for most purposes, only of value when applied between the two exponential map rotations from which it was originally computed. We must carefully consider the future point in our animation at which the application of the time-scaled exponential map delta to our base will give us the pose we hope for, thus confining the most significant pose inaccuracies to the timespan between our keyframes, just as they exist in the other solutions.

Keep in mind though, that before the results can be used to rotate anything in the game, generally a somewhat costly conversion back to ordinary quaternions or perhaps matrices will happen somewhere. Also, it is not possible to achieve the same level of precision when interpolating as in a precise implementation of solution #1, so it can potentially require a small number of additional keyframes since it is more often than not a slightly less accurate attempt at recreating what the original data happened to be on those missing keyframes.

Precise Quantization

Regardless of which solutions are chosen for the preceding issue, packing individual key values as tightly and accurately as possible is, obviously, very important for maximizing compression. It is widely known that quaternions lose substantial precision if simply quantized as four fixed point values, which is why methods such as “smallest three” are so popular. A good explanation of smallest three can be found in the “Gaffer On Games” blog [Fiedler 2015]. While a similar precision disparity exists for exponential map rotations if they are simply quantized as three fixed point values, it is less pronounced than it is with ordinary quaternions. For any rotation, if taken as an axis-angle value pair, by quantizing the axis separately using methods known to cover the desired range of possible axial values with even precision, then next simply quantizing the angle as a fixed point value with an appropriate number of bits, very precise quantization of the rotation can be realized. Despite yielding superior precision, this concept of quantizing a rotation’s angle separately from its axis is not normally done for games, most of which hope to have a quaternion result from decompression. This is because most developers wish to avoid the high CPU cost of building a quaternion from the unpacked axis-angle values. On the other hand, for exponential map values, it is very much worth considering, because the decompressed value is essentially just:

```
expMap = Vector3(normalizedAxis)*float(angle);
```

There are various methods for quantizing the normalized axis in a precise manner, as Krzysztof Narkowicz discusses in his blog [Narkowicz 2014]. While not the absolute most precise option, due to better CPU performance, the octahedron method appears to be the superior choice. Clearly, this is useful not only for quantizing exponential map values in implementations which choose to use them, but also for implementing velocity-based animation where the encoded velocities are keyed such that the axis of rotation is in a separate track from rotational speeds.

References

FIEDLER, G. 2014-03. Snapshot Compression. *Gaffer on Games*. Retrieved from http://gafferongames.com/networked-physics/snapshot-compression/

GRASSIA, F. S. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools,* v.3 n.3, p.29-48, 1 March 1998. DOI:http://doi.acm.org/10.1080/10867651.1998.10487493

KAPOULKINE, A. 2016-05-05. Optimizing slerp. *Bits, pixels, cycles, and more.* Retrieved from http://zeuxcg.org/2016/05/05/optimizing-slerp/

NARKOWICZ, K. 2014-04-16. Octahedron Normal Vector Encoding. *Krzysztof Narkowicz*. Retrieved from https://knarkowicz.wordpress.com/2014/04/16/octahedron-normal-vector-encoding/