Extracting Euler Angles from a Rotation Matrix

Mike Day, Insomniac Games
mday@insomniacgames.com

This article attempts to fix a problem which came up when implementing Ken Shoemake's Euler angle extraction in the context of a single-precision floating point library. The original Shoemake code uses double precision, which presumably maintains sufficient precision for the problem not to arise.

We'll follow the notational conventions of Shoemake's "Euler Angle Conversion", Graphics Gems IV, pp. 222-9, with the exception that our vectors are row vectors instead of column vectors. Thus, all our matrices are transposed relative to Shoemake's, and a sequence of rotations will be written from left to right. We'll simplify the discussion by ignoring the various possible axis permutations, and will instead focus on one particular order of applying rotations to illustrate the problem.

Consider the following sequence of rotations: θ_1 about the x-axis, then θ_2 about the y-axis, then θ_3 about the z-axis, each rotation being applied about one of the world axes as opposed to one of the body axes. This can be written

$$R_x(\theta_1)\,R_y(\theta_2)\,R_z(\theta_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{pmatrix} \begin{pmatrix} c_2 & 0 & -s_2 \\ 0 & 1 & 0 \\ s_2 & 0 & c_2 \end{pmatrix} \begin{pmatrix} c_3 & s_3 & 0 \\ -s_3 & c_3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} c_2 c_3 & c_2 s_3 & -s_2 \\ s_1 s_2 c_3 - c_1 s_3 & s_1 s_2 s_3 + c_1 c_3 & s_1 c_2 \\ c_1 s_2 c_3 + s_1 s_3 & c_1 s_2 s_3 - s_1 c_3 & c_1 c_2 \end{pmatrix}$$

with $c_1 = \cos\theta_1$, $s_1 = \sin\theta_1$, etc.

Now suppose we are given a matrix

$$M = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

and are required to extract Euler angles corresponding to the above rotation sequence, i.e. find angles $\theta_1, \theta_2, \theta_3$ which make the two matrices equal.

In the general case, Shoemake's code proceeds as follows. First θ_1 is extracted using

$$\theta_1 = \operatorname{atan2}(m_{12},\, m_{22})$$

$$= \operatorname{atan2}(s_1 c_2,\, c_1 c_2)$$

Next, c_2 is computed using using

$$c_2 = \sqrt{{m_{00}}^2 + {m_{01}}^2}$$

$$= \sqrt{{c_2}^2{c_3}^2 + {c_2}^2{s_3}^2}$$

and hence θ_2 using

$$\theta_2 = \operatorname{atan2}(-m_{02},\ c_2)$$

Finally θ_3 is obtained using

$$\theta_3 = \operatorname{atan2}(m_{01},\ m_{00})$$

$$= \operatorname{atan2}(c_2 s_3,\ c_2 c_3)$$

which is problematic when m_{00} and m_{01} are both very small or zero. We may note that this will also cause m_{12} and m_{22} to be very small or zero (since m_{02} will be near ± 1), making the extraction of θ_1 equally problematic. Shoemake's solution is to test the value computed for c_2 against a tiny threshold – if it falls below this threshold then the matrix elements reduce to approximately the following:

$$\begin{pmatrix} 0 & 0 & -(\pm 1) \\ \pm s_1 c_3 - c_1 s_3 & \pm s_1 s_3 + c_1 c_3 & 0 \\ \pm c_1 c_3 + s_1 s_3 & \pm c_1 s_3 - s_1 c_3 & 0 \end{pmatrix}$$

and in this case the angles θ_1 and θ_3 are extracted by a different code path. This matrix provides an example of the phenomenon of gimbal lock, in which the 1st and 3rd axes are brought into alignment by the 2nd rotation, effectively losing a degree of freedom because now θ_1 and θ_3 act in combination as though they were a single parameter. Shoemake handles this case by forcing θ_3 to zero – which is ok, because gimbal lock allows us to arbitrarily set one of θ_1 and θ_3 and then derive the other. This further reduces the matrix to the following form:

$$\begin{pmatrix} 0 & 0 & -(\pm 1) \\ \pm s_1 & c_1 & 0 \\ \pm c_1 & -s_1 & 0 \end{pmatrix}$$

from which θ_1 is easily extracted using

$$\theta_1 = \operatorname{atan2}(-m_{21},\ m_{11})$$

This works perfectly well for cases which fall within the small threshold, which is 16*FLT_EPSILON in Shoemake's code. (I wasn't able to tell where this magic value came from – maybe it's just a very long-lived fudge factor.) However, it can be dangerous to use in cases which fall *just outside* the threshold.

When the routine is passed a real-world single-precision matrix whose elements are subject to some typical rounding errors, θ_1 can be expected to take on a fairly chaotic set of values. This is not surprising, as we expect near-gimbal-lock orientations to produce wild jumps in the angle values, and any numerical errors can accentuate these jumps. This in itself is not a major issue because of the close relationship between θ_1 and θ_3 near the gimbal lock orientations. Any wobble in the value of θ_1 can be counteracted by a suitable anti-wobble in the value of θ_3.

However, we have a couple of major expectations of the extracted angles. First, they should approximately reproduce the original matrix when passed to the inverse function. Second, although we may not be able to extract an accurate θ_1 or θ_3 in isolation, we still expect their *resultant* angle to be correct: the extracted values should not be independent.

In Shoemake's version, in cases which fall outside the threshold, the angles θ_1 and θ_3, while algebraically dependent, are *numerically independent* in the sense that rounding errors in one are not compensated for in the other. In single-precision code these rounding errors can be very large, leading to completely erroneous results when the computed angles are used in an attempt to reconstruct the original matrix.

This theory is easily tested by passing to the Shoemake version a gimbal locked matrix with a couple of the zeros slightly jiggled by amounts just larger than 16*FLT_EPSILON – i.e. of the order 10^{-6}, which could easily arise from rounding errors in real-world single-precision matrices. Sure enough, the result of passing the extracted angles into the inverse function is often a completely different matrix. That's a bug, because we'd like it to be insensitive to rounding errors of order 10^{-6}. We might try substantially increasing the threshold, to make the results less sensitive to rounding error, but the trouble with this approach is that the approximation used in the gimbal lock cases would no longer be valid.

Have we all simply lived with this problem? Perhaps the errors in double precision matrices tend to be sufficiently small that the bad cases never arise, but this reliance doesn't seem like a robust approach, and is certainly not the way to go for a single-precision library.

Fortunately, there seems to be an easy fix: compute the rotation generated by the first and second extracted angles, and work out the rotation needed in the third angle to match the target matrix. This is easily derived by pre-multiplying the target matrix by the transpose of the reconstructed first-and-second-angle matrix. So we need to compute the following:

$$M' = \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{pmatrix} \begin{pmatrix} c_2 & 0 & -s_2 \\ 0 & 1 & 0 \\ s_2 & 0 & c_2 \end{pmatrix} \right)^T \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

$$= \begin{pmatrix} c_2 & 0 & -s_2 \\ s_1 s_2 & c_0 & s_1 c_2 \\ c_1 s_2 & -s_0 & c_1 c_2 \end{pmatrix}^T \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

$$= \begin{pmatrix} c_2 & s_1 s_2 & c_1 s_2 \\ 0 & c_1 & -s_1 \\ -s_2 & s_1 c_2 & c_1 c_2 \end{pmatrix} \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

If this product represents a pure rotation about the z-axis, it must be of the following form:

$$M' = \begin{pmatrix} c_3 & s_3 & 0 \\ -s_3 & c_3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and extraction of θ_3 should be straightforward. In practice we don't even need to compute a full matrix product. Reading across the middle row of M', we obtain

$$c_1 m_{10} - s_1 m_{20} = -s_3; \qquad c_1 m_{11} - s_1 m_{21} = c_3$$

and hence

$$\theta_3 = \text{atan2}(s_1 m_{20} - c_1 m_{10},\ c_1 m_{11} - s_1 m_{21})$$

which requires that we compute the sine and cosine of the value we extracted for θ_1. This way, any gimbal lock instability in the value of θ_1 is fed back into the extraction process, and will be counteracted in the value computed for θ_3. Note that this method of extracting the 3rd angle makes no assumptions about the other two angles, so that it can be applied in the non-gimbal lock cases too, and no conditional branches are needed.

The final calculation is

$$\theta_1 = \text{atan2}(m_{12},\ m_{22})$$

$$c_2 = \sqrt{{m_{00}}^2 + {m_{01}}^2}$$

$$\theta_2 = \text{atan2}(-m_{02},\ c_2)$$

$$s_1 = \sin(\theta_1),\ \ c_1 = \cos(\theta_1)$$

$$\theta_3 = \text{atan2}(s_1 m_{20} - c_1 m_{10},\ c_1 m_{11} - s_1 m_{21})$$