Converting a Rotation Matrix to a Quaternion

Mike Day, Insomniac Games
mday@insomniacgames.com

This article attempts to improve upon an existing method for extracting a unit quaternion from a rotation matrix.

Summary of the problem

We will use as our starting point the following correspondence. (The reader is referred to http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation for some background on how this correspondence arises.)

Suppose we are given a unit quaternion $q = (x, y, z, w)$, where w is the real part. Then the rotation matrix M corresponding to q takes the following form:

$$M = \begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2zw & 2xz - 2yw \\ 2xy - 2zw & 1 - 2x^2 - 2z^2 & 2yz + 2xw \\ 2xz + 2yw & 2yz - 2xw & 1 - 2x^2 - 2y^2 \end{pmatrix}$$

(Note that here we adopt the convention of expressing vectors as rows, so that a vector is rotated by post-multiplying it by a rotation matrix. This convention is opposite to the one used in the Wikipedia article, so the matrix will appear transposed. This is done to ensure consistency with what seems to be the most frequently-used form of the prior conversion code.)

The problem can now be stated as follows. Suppose we are given the values of the elements of the rotation matrix M:

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

Then our task is to recover the components of the corresponding quaternion q.

Equating the above two forms of the matrix, consider the following four expressions:

$$1 + m_{00} - m_{11} - m_{22} = 4x^2$$

$$m_{10} + m_{01} = 4xy$$

$$m_{20} + m_{02} = 4xz$$

$$m_{12} - m_{21} = 4xw$$

We can see that by forming a quaternion out of these 4 components in the order given we will generate $4x(x, y, z, w)$. We have almost recovered the desired quaternion (x, y, z, w), but it has been scaled by a factor $4x$, and we wish to remove this factor. One option would be to simply normalize the result, which would require generating the reciprocal length. A cheaper option is to simply generate the reciprocal of $|4x|$ from the known value of $4x^2$:

$$\frac{1}{|4x|} = \frac{1}{2\sqrt{4x^2}}$$

We can then scale all components by this factor.

This process immediately raises the question, what happens if $x = 0$? Then all four expressions will evaluate to zero, leading to a nonsense result. Even if the expressions are not actually zero but small in magnitude, then they may have suffered catastrophic cancellation which can lead to an unstable result.

We can solve this problem via the following observation. One thing we can definitely say about a unit quaternion is that there will always exist at least one component which is *not* close to zero – indeed there must exist a component whose magnitude is at least $1/2$. The expressions we used generated a quaternion which had been scaled by $4x$. But there are other similar forms available which lead to scaling factors of $4y$, $4z$, and $4w$. If we are prepared to branch in one of four ways, we can always pick a branch where numerical instability is not an issue. Indeed this is the rationale behind the commonly used conversion code.

Previous code

A web-search turns up a fairly standard piece of code, with minor variations here and there, which can be paraphrased as follows:

```
trace = m00 + m11 + m22;

if (trace > 0.0f)
{
  k = 0.5f / sqrt(1.0f + trace);
  q = quat( k * (m12 - m21), k * (m20 - m02), k * (m01 - m10), 0.25f / k );
}
else if ((m00 > m11) && (m00 > m22))
{
  k = 0.5f / sqrt(1.0f + m00 - m11 - m22);
  q = quat( 0.25f / k, k * (m10 + m01), k * (m20 + m02), k * (m12 - m21) );
}
else if (m11 > m22)
{
  k = 0.5f / sqrt(1.0f + m11 - m00 - m22);
  q = quat( k * (m10 + m01), 0.25f / k, k * (m21 + m12), k * (m20 - m02) );
}
else
{
  k = 0.5f / sqrt(1.0f + m22 - m00 - m11);
  q = quat( k * (m20 + m02), k * (m21 + m12), 0.25f / k, k * (m01 - m10) );
}
```

This seems to be less than ideal in a few ways:

1. It computes the sum $m_{00} + m_{11} + m_{22}$ upfront, but throws it away in three out of the four cases.
2. There is an asymmetrical treatment of the 4 components: the first path is taken if $|w| > \frac{1}{2}$, but if this condition fails then it will pick the path corresponding to the largest of $|x|, |y|, |z|$.
3. A divide is always performed for one component.

New treatment

We shall look at how to address these objections. First, we ask: when is each of the diagonal elements negative?

$$m_{00} < 0 \quad \Leftrightarrow \quad 1 - 2y^2 - 2z^2 < 0 \quad \Leftrightarrow \quad y^2 + z^2 > \frac{1}{2}$$

$$m_{11} < 0 \quad \Leftrightarrow \quad 1 - 2x^2 - 2z^2 < 0 \quad \Leftrightarrow \quad x^2 + z^2 > \frac{1}{2}$$

$$m_{22} < 0 \quad \Leftrightarrow \quad 1 - 2x^2 - 2y^2 < 0 \quad \Leftrightarrow \quad x^2 + y^2 > \frac{1}{2}$$

But we also have $x^2 + y^2 + z^2 + w^2 = 1$ because q is a unit quaternion. So, for example, m_{22} is negative when x and y together contribute more than half the magnitude of the quaternion, which is enough to ensure that either x or y is big enough to avoid numerical instability; we just need a test to determine which one. (Quite likely both are, but we need a guarantee.)

Now consider the pairwise differences of diagonal elements:

$$m_{00} - m_{11} = 1 - 2y^2 - 2z^2 - 1 + 2x^2 + 2z^2 = 2(x^2 - y^2)$$

$$m_{11} - m_{22} = 1 - 2x^2 - 2z^2 - 1 + 2x^2 + 2y^2 = 2(y^2 - z^2)$$

$$m_{22} - m_{00} = 1 - 2x^2 - 2y^2 - 1 + 2y^2 + 2z^2 = 2(z^2 - x^2)$$

And finally the pairwise sums:

$$m_{00} + m_{11} = 1 - 2y^2 - 2z^2 + 1 - 2x^2 - 2z^2 = 2(1 - x^2 - y^2 - 2z^2) = 2(w^2 - z^2)$$

$$m_{11} + m_{22} = 1 - 2x^2 - 2z^2 + 1 - 2x^2 - 2y^2 = 2(1 - 2x^2 - y^2 - z^2) = 2(w^2 - x^2)$$

$$m_{22} + m_{00} = 1 - 2x^2 - 2y^2 + 1 - 2y^2 - 2z^2 = 2(1 - x^2 - 2y^2 - z^2) = 2(w^2 - y^2)$$

Thus, for any specific choice of two components, we can determine which one is greater in magnitude just by comparing one particular diagonal element with another, or with its negative. For example,

$$|x| > |y| \quad \Leftrightarrow \quad x^2 - y^2 > 0 \quad \Leftrightarrow \quad m_{00} > m_{11}$$

$$|x| > |w| \quad \Leftrightarrow \quad w^2 - x^2 < 0 \quad \Leftrightarrow \quad m_{11} < -m_{22}$$

Revised framework

Together, these observations suggest a divide-and-conquer strategy to select one of the four code paths. Furthermore, the selection can be made solely by comparing diagonal elements with each other (possibly after a negation) and with zero. Here is one possibility for implementing such a strategy:

```
if (m22 < 0)        // is |(x,y)| bigger than |(z,w)|?
{
  if (m00 > m11)    // is |x| bigger than |y|?
    // use x-form
  else
    // use y-form
}
else
{
  if (m00 < -m11)   // is |z| bigger than |w|?
    // use z-form
  else
    // use w-form
}
```

This framework addresses the criticism over the lack of symmetry in the code. Also, since we are no longer required to compute the trace of the matrix upfront, we will not be throwing away the computed value – the trace will only be computed in one out of the four branches (the 'w-form' one). In each of the other three branches the corresponding expression needed by just that branch will be computed.

Finally, the objection about the use of a divide operation can be addressed as follows. The original code computes

```
k = 0.5f / sqrt(t);
```

for each branch, where t is some sum of terms, and then sets one component equal to $0.25/k$. (It is the second of these divisions we are concerned with; the first is a reciprocal-square-root operation which is normally performed directly without the use of a divide.) But note that

$$\frac{0.25}{k} = \frac{0.25}{0.5/\sqrt{t}}$$

$$= 0.5\sqrt{t}$$

$$= kt$$

Thus the divide is replaced with a multiply. It is possible that the compiler may perform this optimization, but this way we can ensure it.

New version

The revised code becomes:

```
if (m22 < 0)
{
  if (m00 > m11)
  {
    t = 1 + m00 - m11 - m22;
    q = quat( t, m01+m10, m20+m02, m12-m21 );
  }
  else
  {
    t = 1 - m00 + m11 - m22;
    q = quat( m01+m10, t, m12+m21, m20-m02 );
  }
}
else
{
  if (m00 < -m11)
  {
    t = 1 - m00 - m11 + m22;
    q = quat( m20+m02, m12+m21, t, m01-m10 );
  }
  else
  {
    t = 1 + m00 + m11 + m22;
    q = quat( m12-m21, m20-m02, m01-m10, t );
  }
}

q *= 0.5 / Sqrt(t);
```