Using Quaternion to Perform 3D rotations

By confuted

The title of this tutorial is short and sweet. The tutorial itself will not be. First, an introduction. Quaternions are the things that scare all manner of mice and men. They are the things that go bump in the night. They are the reason your math teacher gave you an F. They are all that you have come to fear, and more. Quaternions are your worst nightmare.

Okay, not really. They aren't actually that hard. I just wanted to scare you. Quaternions have become a popular tool in 3d game development - and for a good reason. Once you understand them, quaternions are really pretty amazing. Also, unlike the other tutorials, I'm going to more or less be assuming that you *know nothing* about quaternion math in this tutorial. Here are the basics of a quaternion:

A quaternion represents two things. It has an x, y, and z component, which represents the axis about which a rotation will occur. It also has a w component, which represents the amount of rotation which will occur about this axis. In short, a vector, and a float. With these four numbers, it is possible to build a matrix which will represent all the rotations perfectly, with no chance of gimbal lock. (I actually managed to encounter gimbal lock with quaternions when I was first coding them, but it was because I did something incorrectly. I'll cover that later). So far, quaternions should seem *a lot* like the axis angle representation. However, there are some large differences, which start....now.

A quaternion is *technically* four numbers, three of which have an imaginary component. As many of you probably know from math class, i is defined as sqrt(-1). Well, with quaternions, $i = j = k =$ sqrt(-1). The quaternion itself is defined as $q = w + xi + yj + zk$. w, x, y, and z are all real numbers. The imaginary components are important if you ever have a math class with quaternions, but they aren't particularly important in the programming. Here's why: we'll be storing a quaternion in a class with four member variables: float w, x, y, z;. We'll be ignoring *i*, *j*, and *k*, because we never liked them anyway. Okay, so we're actually just ignoring them because we don't need them. We'll define our quaternions (w, x, y, z).

You may have guessed by now that w is the amount of rotation about the axis defined by <x, y, z>. (Warning: the math is going to start getting pretty heavy. I'll explain the quaternion specific stuff, though). Much like unit vectors are necessary for much of what is done in a 3d engine, with lighting, back-face culling, and the like, unit quaternions are needed to perform the operations we'll be doing below. Luckily, normalizing a quaternion isn't much harder than normalizing a vector. The magnitude of a quaternion is given by the formula magnitude = sqrt($w^2 + x^2 + y^2 + z^2$). For the unit quaternions, the magnitude is one. Which means that, already, an optimization is in order. Floating-point numbers aren't perfect. Eventually, you'll lose accuracy with them, and you may want to check to see if you need to re-normalize your unit quaternion, to prevent math errors. This can be done by checking the magnitude, which will need to be one... but if you're checking to see if it's one, there is no need for the sqrt() at all (sqrt(1) = 1). So, if ($w^2 + x^2 + y^2 + z^2$) is more than a certain tolerance away from 1, you'll know that your quaternion needs to be normalized, and you will have saved taking a square root in the cases when it is within tolerance. Every few clock cycles count. But if you ever need to actually normalize a quaternion, here's how it would be done:

> magnitude = sqrt($w^2 + x^2 + y^2 + z^2$)
> w = w / magnitude
> x = x / magnitude
> y = y / magnitude
> z = z / magnitude

(Yes, I'm familiar with how to do that in C++, including the /= operator. I'm trying to keep this easy to read mathematically.) Performing the above operations will ensure that the quaternion is a unit quaternion.

However, they are (somewhat) expensive in CPU time and should not be called unless needed - which they usually aren't. There are many different possible unit quaternions - they actually describe a *hyper-sphere*, a four dimensional sphere. Don't try to visualize it; your head will explode. But because the end points for unit quaternions all lay on a hyper-sphere, multiplying one unit quaternion by another unit quaternion will result in a third unit quaternion. I guess now it's time for me to describe quaternion multiplication.

One of the most important operations with a quaternion is multiplication. If you are using C++ and coding your own quaternion class, I would highly suggest overloading the * operator to perform multiplications between quaternions. Here is how the multiplication itself is performed: (sorry about the HTML subscripts, I know they suck)

Let Q_1 and Q_2 be two quaternions, which are defined, respectively, as (w_1, x_1, y_1, z_1) and (w_2, x_2, y_2, z_2).
$(Q_1 * Q_2).w = (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2)$
$(Q_1 * Q_2).x = (w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)$
$(Q_1 * Q_2).y = (w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)$
$(Q_1 * Q_2).z = (w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2$

It is very, very, very important for you to note that $Q_1 * Q_2$ is NOT equal to $Q_2 * Q_1$. Quaternion multiplication is not commutative. If you don't remember this, it will give you trouble later, and it won't be easy to spot the cause. So do yourself a favor and remember it. I didn't the first time I read it. I'm speaking from experience here.

What does the quaternion multiplication *mean*? To tell you the truth, this is where quaternions start to become beautiful. Until now, they've been a bunch of math which wasn't difficult, but might have been annoying. Now, quaternions will become *useful*. Remember that a quaternion stores an axis and the amount of rotation about the axis. So, with that, after I give you the matrix for rotations with quaternions, you would be able to rotate an object over some arbitrarily defined axis by some arbitrary amount, without fear of gimbal lock. However, changing the rotation would be a trickier manner. To change the rotation represented by a quaternion, a few steps are necessary. First, you must generate a temporary quaternion, which will simply represent how you're changing the rotation. If you're changing the current rotation by rotating backwards over the X-axis a little bit, this temporary quaternion will represent that. By multiplying the two quaternions (the temporary and permanent quaternions) together, we will generate a *new* permanent quaternion, which *has been changed by the rotation described in the temporary quaternion*. At this point, it's time for a good healthy dose of pseudo-code, before you get so confused we have to bring in the EMTs to resuscitate you.

```
/*to keep with the conventions I've followed in some posts on cprogramming.com, I
will call the temporary quaternion described above local_rotation. I'll be calling the
permanent quaternion described above total.*/

// generate local_rotation (details later)
total = local_rotation * total  //multiplication order matters on this line
```

Before I try to explain that any more, I need to teach you how to generate local_rotation. You'll need to have the axis and angle prepared, and this will convert them to a quaternion. Here's the formula for generating the *local_rotation* quaternion.

```
//axis is a unit vector
local_rotation.w  = cosf( fAngle/2)
local_rotation.x = axis.x * sinf( fAngle/2 )
local_rotation.y = axis.y * sinf( fAngle/2 )
local_rotation.z = axis.z * sinf( fAngle/2 )
```

Then, just multiply *local_rotation* by *total* as shown above. Since you'll be multiplying two unit quaternions together, the result will be a unit quaternion. You won't need to normalize it. At this point, I feel the need to once again point out that quaternion multiplication is not commutative, and the order matters.

Hang in there; we're almost done. All that is left for this tutorial is generating the matrix from the quaternion to rotate our points.

$$\begin{matrix} w^2+x^2-y^2-z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & w^2-x^2+y^2-z^2 & 2yz+2wx & 0 \\ 2xz-2wy & 2yz-2wx & w^2-x^2-y^2+z^2 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

And, since we're only dealing with unit quaternions, that matrix can be optimized a bit down to this:

$$\begin{matrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz+2wx & 0 \\ 2xz-2wy & 2yz-2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Amazing. Well, maybe not amazing, but pretty cool. You'll regenerate these matrices each frame. Most importantly, you *won't get gimbal lock* if you follow my directions. One thing that I forgot to mention earlier - you'll need to initialize your *total* quaternion to the value (1,0,0,0). This represents the "initial" state of your object - no rotation.

Interpolating between two orientations using quaternions is also the smoothest way to interpolate angles. It's done via a method known as SLERP, or Spherical Linear intERPolation. I haven't encountered a need for this yet, so I haven't researched it, but perhaps someday I'll research it and write a tutorial about it to add to this series. If anyone needs the information, feel free to contact me, and clue me in that someone actually read this and wants more. That's all for now, have fun coding Quake. Or Doom. Make sure to give me a copy of it when you're done.

Previous: Uses for What You've Learned
Back to Graphics tutorials index