Better 3D:

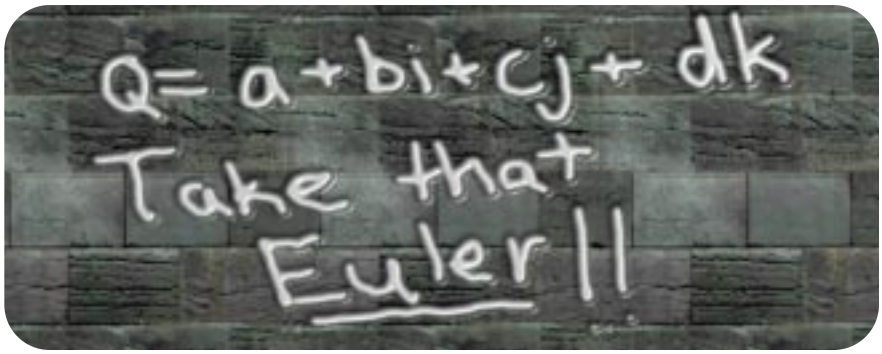
The Writing Is on the Wall

Have you ever come up with the solution to a problem at an unfortunate time? It can strike so suddenly. You know that if you don't write it down immediately, the solution may be lost forever. There are times when I wake from a sound sleep with the answer to a problem that's been bugging me for days. Occasionally, I have to jump up and crank on the computer to try it out right away. Most times though, I just roll over and go back to sleep. Epiphany or not, a guy needs his rest.

This phenomenon didn't begin with the computer age. In fact, long ago, the mathematician Sir William Hamilton found himself in the exact same situation. In the early 1830s, he was trying to extend complex number theory from 2D to 3D. Hamilton believed that a complex volume could be defined by one real and two imaginary axes. He worked on this problem for over a decade with no luck, and I might add, without the aid of a computer. Then, on October 16, 1843, while walking past the Broome Bridge to a meeting at the Royal Irish Academy in Dublin, the answer came to him. Hamilton realized that he needed three imaginary units with some special properties to describe the volume. He was so excited and so fearful of losing the answer for all time, that he carved the formula into the side of the bridge with a knife. It's a good thing he didn't roll over and go to sleep.

He called his solution a "quaternion," and it looks like this:

q = a + bi + cj + dk



Although I've never come up with something so good that I needed to carve it into the side of a bridge, I do have some ideas for animating characters that I've wanted to implement for years. Until now, I've had problems making these ideas come to life. Some of my problems are solved by better hardware and some are solved by just plain hard work. Today's problem has been solved by a combination of both. But what, you ask, do quaternions and my problems with animating characters have to do with game programming? Read on.

Now that we have the ability to create complex real-time characters, the challenge lies in bringing these characters to life. So how do I do that successfully? It really comes down to a problem of data storage and playback. What I need to store is the position and orientation of each "bone" in the character that I'm animating.

Last month, Nick Bobick wrote about using quaternions for camera animation and physical simulation ("Rotating Objects Using Quaternions," February 1998). We are going to apply the same concepts to character animation in OpenGL.

Who is this Jeff Lander dude, anyway? Jeff's a maniacal research nerd who loves the challenge of computer graphics programming when not mountain biking by the beach. He is always willing to learn more, and in fact, thrives on it. If you have anything you want to learn more about or something you would like to discuss, contact Jeff at jeffl@darwin3d.com.

Character Animation

Animation data is normally stored in keyframes. These keyframes are recorded periodically. For instance, motion capture data is often recorded at 30 frames per second (fps), the standard NTSC video playback rate. Obviously, such a high frame rate generates a lot of data. In order to save precious RAM, many game applications store animation data at a much lower rate, say 10 fps.

However, modern 3D action games try to run quite a bit faster than 10 fps. A game running at 30 fps will show three times as many frames as a 10 fps

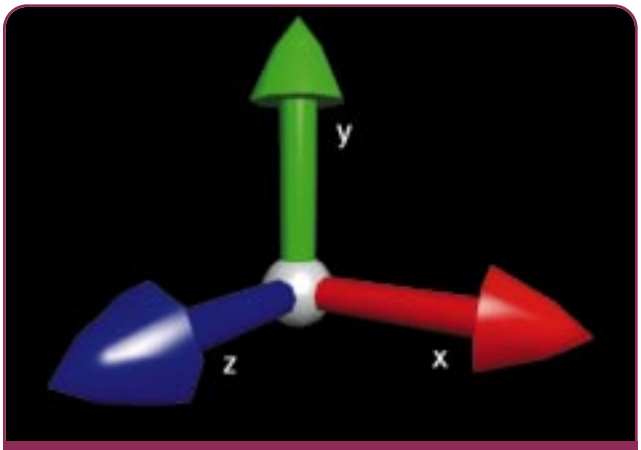


FIGURE 1A.

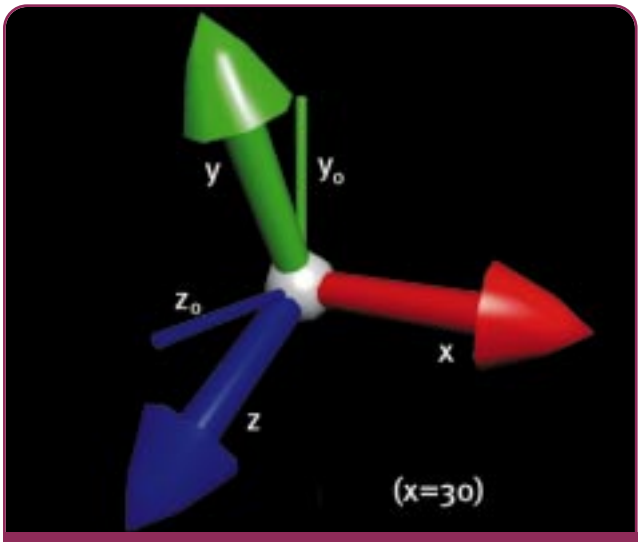


FIGURE 1B.

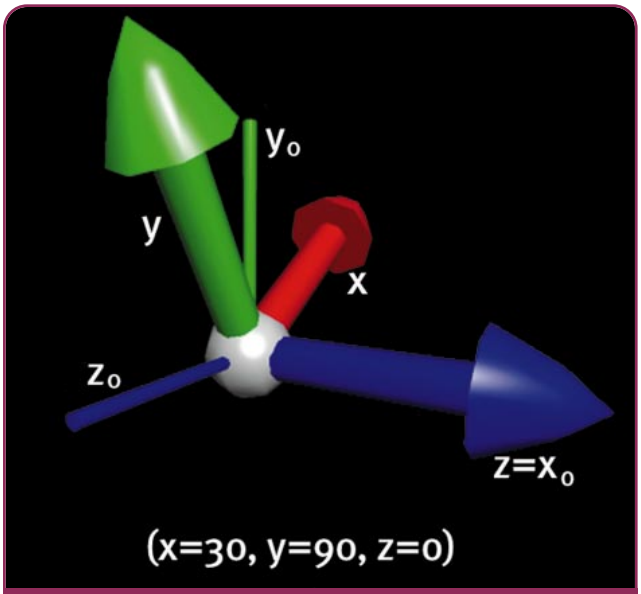


FIGURE 1C.

animation. This leaves you with a couple of options. The first option is to repeat or hold the same keyframe over three game cycles until the next animation frame arrives. This approach, however, can make your animation look a bit jerky.

The second option — and I consider this to be the better solution — is to create frames in real time that are in-between the existing keyframes. You generate these in-betweens by interpolating where the bone is at a given time between your keyframed data. Coming up with a method by which to calculate this in-between position is where I start hitting the wall (or bridge).

Euler Angles

The most common method of representing the current orientation of an object is through the use of Euler angles. This method represents the absolute orientation of an object as a series of three rotations about three mutually orthogonal axes, normally labeled x, y, and z. These Euler angles describe the three angular degrees of freedom. It's important to remember that by applying these rotations in different orders, you'll get different final orientations. So it's best to stay consistent.

If you saw the movie *Apollo 13*, you were probably amused by those guys whipping out their slide rules and complaining about something called "gimbal lock." Well, as strange as it sounds, gimbal lock can be a real problem. It's named after a situation that occurs in a mechanical gyroscope consisting of three concentric rings. Under certain rotations, the support for the gyroscope, called a gimbal, could lose of a degree of freedom.

This problem has a parallel in computer graphics. Bobick explained the situation in his February article, but it may help us to visualize it. Because Euler angles do not act independently of each other, it's possible to lose a degree of freedom. Therefore, a change to one of the angles affects the entire system. Let's look at an example. If I take the object in Figure 1a and rotate it 30 degrees about the x axis, I'll get the image in Figure 1b. I now rotate the object 90 degrees about the y axis, resulting in Figure 1c. As you can see, the current z axis is in line with the x_o axis. What we have now is gimbal lock. Any further rotation about the z axis affects the same degree of freedom as rotating about the x axis. I have completely lost the ability to rotate around the third degree of freedom. In many 3D modeling packages, it's possible to avoid this problem by creating a parent to this object and rotating the parent to gain back the additional degree of freedom. However, in a real-time 3D game, this isn't really possible.

The second problem with the use of Euler angles involves generating an in-between. In order to animate my character smoothly, I need to interpolate a new position out of the existing keyframes in my animation data. The fact that the Euler angles don't act independently of each other again raises a problem. I could represent the orientation of an object as (0,180,0) degrees about (x,y,z) respectively. This same orientation could also be described as (180,0,180). Now, what if I want to generate a position halfway between (0,0,0) and both orientations? In the first case, I would get (0,90,0) (Figure 2a), but in the second case I would get (90,0,90) (Figure 2b). Clearly, these two orientations are not

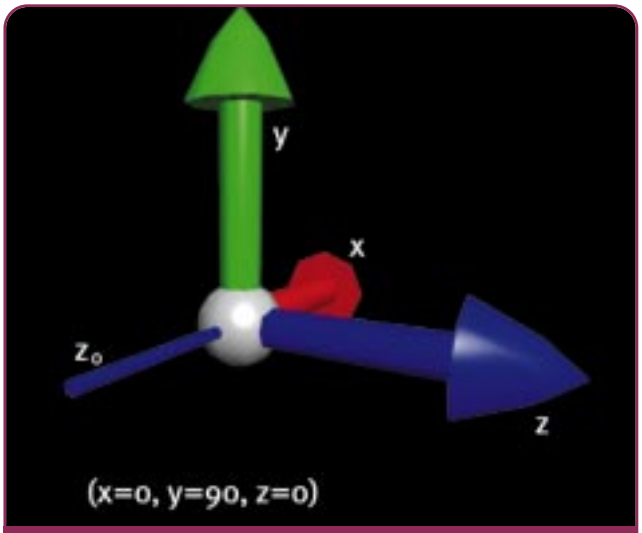


FIGURE 2A.

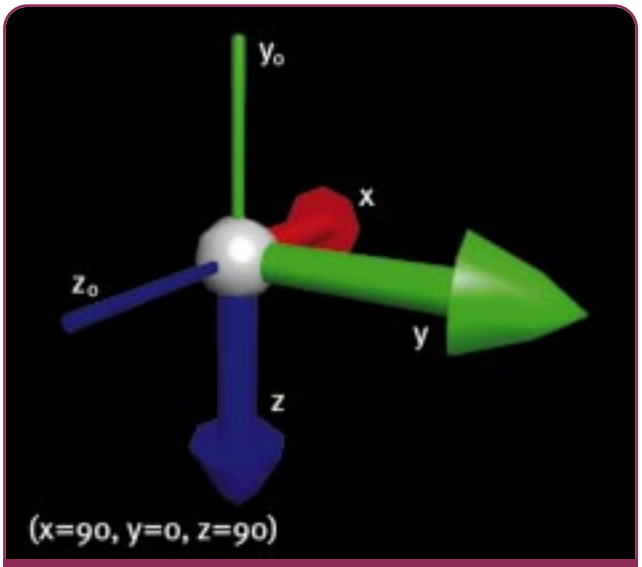


FIGURE 2B.

equivalent. These two problems with animating Euler angles lead me to look for a better way to store orientations for my animation.

Quaternions

By using quaternions to represent the orientation of an object, I can avoid the drawbacks of using Euler angles. A quaternion, when used to represent the orientation of a rigid body in space, is defined by two parts: a vector that describes the axis of rotation and a scalar value. When two quaternions of unit magnitude are multiplied together, they generate a single quaternion of unit magnitude — this is crucial to the use of quaternions for representing rotations. So, to use this system of animation, I have to convert my orientations from Euler angles to quaternions. For details on how quaternions are multiplied together I would suggest the Shoemake article that is referenced at the end of this column.

Conversion from Euler Angles to Quaternions

Rotations are defined as the following quaternions using [(x,y,z),w] notation, where (x,y,z) is a vector and w is a scalar value.

Euler ($x = \psi$, $y = \theta$, $z = \phi$)
$Q_x = [(\sin(\psi/2),0,0), \cos(\psi/2)]$
$Q_y = [(0,\sin(\theta/2),0), \cos(\theta/2)]$
$Q_z = [(0,0, \sin(\phi/2)), \cos(\phi/2)]$

LISTING 1. *Conversion from Euler angles to quaternions.*

```
////////////////////////////////////////////////////////////////////////
// Function:        EulerToQuaternion
// Purpose:         Convert a set of Euler angles to a quaternion
// Arguments:       A rotation set of 3 angles, a quaternion to set
// Discussion:      This creates a Series of quaternions and multiplies them together
//                  in the X Y Z order.
////////////////////////////////////////////////////////////////////////
void EulerToQuaternion(tVector *rot, tQuaternion *quat)
{
/// Local Variables ///////////////////////////////////////////////////
          float rx,ry,rz,ti,tj,tk;
          tQuaternion qx,qy,qz,qf;
////////////////////////////////////////////////////////////////////////
          // FIRST STEP, CONVERT ANGLES TO RADIANS
          rx =  (rot->x * M_PI) / (360 / 2);
          ry =  (rot->y * M_PI) / (360 / 2);
          rz =  (rot->z * M_PI) / (360 / 2);
          // GET THE HALF ANGLES
          ti = rx * 0.5;
          tj = ry * 0.5;
          tk = rz * 0.5;

          qx.x = sin(ti); qx.y = 0.0; qx.z = 0.0; qx.w = cos(ti);
          qy.x = 0.0; qy.y = sin(tj); qy.z = 0.0; qy.w = cos(tj);
          qz.x = 0.0; qz.y = 0.0; qz.z = sin(tk); qz.w = cos(tk);

          MultQuaternions(&qx,&qy,&qf);
          MultQuaternions(&qf,&qz,&qf);

          quat->x = qf.x;
          quat->y = qf.y;
          quat->z = qf.z;
          quat->w = qf.w;

}
// EulerToQuaternion ////////////////////////////////////////////////////
```

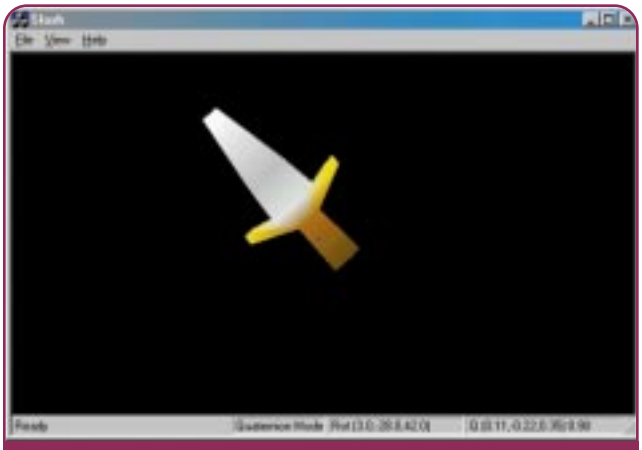


FIGURE 3.

By multiplying these quaternions together, I get a single quaternion representing the orientation of the object. The code for this conversion is shown in Listing 1.

This routine is set up to handle rotations in the order xyz. If your rotations are stored in a different order, you may need to be make some adjustments. By combining functions, it is possible to speed this up a bit. In the code that accompanies this article, I reworked it as a secondary conversion routine.

OpenGL Implementation

Since my eventual output is through OpenGL, I need to convert the quaternion back into something useful. The **glRotatef** command takes an axis to rotate around and the angle of rotation about that axis. This axis-angle representation was explored briefly in Chris Hecker's column "Physics, Part 4: The Third Dimension" (*Game Developer*, June 1997, pp.17-18). This representation is very similar to the quaternion form. The quaternion is defined as a rotation of ϕ about the axis (x, y, z) such that

$$q = \cos(\phi/2) + x \sin(\phi/2)i + y \sin(\phi/2)j + z \sin(\phi/2)k$$

Therefore, the axis and angle of rotation can be retrieved from a quaternion by simple algebra.

$$\phi = \text{acos}(Q_w)*2$$
$$x = Q_x/\sin(\phi/2)$$
$$y = Q_y/\sin(\phi/2)$$
$$z = Q_z/\sin(\phi/2)$$

This gives us the axis and angle that we need. To use it in OpenGL, the last thing to do is convert angle ϕ from radians to degrees, because OpenGL is expecting degrees (Listing 2). Since the data is now in the correct format for OpenGL, the actual display code is very straightforward (Listing 3).

Where Does That Bring Us?

The question is, "If quaternions are so great, why not use them all the time?" You certainly could. However, most people find it difficult to visualize the orientation of an object as a quaternion. People (at least 3D programming dweebs) are often more comfortable visualizing an orientation as Euler angles. This is an important consideration when you're creating a tool that will be used by artists and programmers. I'm sure people confronted with a dialog box asking them to enter the quaternion for a rocket ship would have some problems. At least Euler angles are a little more friendly.

The sample application uses the previously mentioned conversion routines to display an object using either Euler angles or quaternions. By switching between these display methods, you can see that both representations result in the same final display (Figure 3).

Using these routines, I now have a method for converting my Euler angle rotations into quaternions and then displaying them. Whew! This has been a lot of difficult stuff. But now we have the foundation. From here, I need to start animating these orientations. Next time, I'll add the interpolation code and build an application to display the in-betweens. ■

LISTING 2. *Converting the angle ϕ to degrees.*

```
////////////////////////////////////////////////////////////////////////////
// Function:        QuatToAxisAngle
// Purpose:         Convert a quaternion to Axis Angle representation
// Arguments:       A quaternion to convert, a axisAngle to set
////////////////////////////////////////////////////////////////////////////
void QuatToAxisAngle(tQuaternion *quat,tQuaternion *axisAngle)
{
/// Local Variables ///////////////////////////////////////////////////////
          float scale,tw;
////////////////////////////////////////////////////////////////////////////
          tw = (float)acos(quat->w) * 2;
          scale = (float)sin(tw / 2.0);
          axisAngle->x = quat->x / scale;
          axisAngle->y = quat->y / scale;
          axisAngle->z = quat->z / scale;

          // NOW CONVERT THE ANGLE OF ROTATION BACK TO DEGREES
          axisAngle->w = (tw * (360 / 2)) / (float)M_PI;
}
// QuatToAxisAngle  ////////////////////////////////////////////////////////
```

LISTING 3. *Using quaternions in OpenGL.*

```
// TAKE THE BONE EULER ANGLES AND CONVERT THEM TO A QUATERNION
EulerToQuaternion(&curBone->rot,&curBone->quat);
// QUATERNION HAS TO BE CONVERTED TO AN AXIS/ANGLE REPRESENTATION
QuatToAxisAngle(&curBone->quat,&axisAngle);
// DO THE ROTATION
glRotatef(axisAngle.w, axisAngle.x, axisAngle.y, axisAngle.z);
drawModel(curBone);
```

Hardware for OpenGL Developer Tools

When developing an advanced real-time 3D game engine and production tools, it's important to work on decent hardware. Likewise, when artists and designers work on those tools, performance means productivity. Now obviously, the 3D artists, modelers, and animators should have great systems. Any of the fully OpenGL-certified high-end graphics cards would also work well for production tools.

But what about the programmers, level designers, balance testers, and so on? I would like a great production environment for everyone. Luckily for me, certain "market pressures" have pushed some consumer graphics hardware towards OpenGL support. For me, OpenGL support means that the images that I see in my modeling and animation program and in my toolset are identical. The material and display parameters match exactly. This is very important at the design level no matter which final game application API is used (OpenGL, Direct3D, or proprietary). Low-cost OpenGL acceleration allows companies to get high-performance game production stations for all the team members.

To be a good system for game production, a graphics card should support acceleration in multiple windows, because most 3D development tools require more then one viewpoint. It should support most, if not all, display features of the target platform. It should also have decent performance in the tool, as well as being able to run the actual game application at a decent rate.

The 3Dlabs Permedia 2 chipset is an excellent choice for a low-cost OpenGL tool platform. 3Dlabs has much experience working with OpenGL through its Glint family of cards, and it shows. Its OpenGL drivers are stable and robust. The Permedia 2 adds acceleration of glLines, which really can help a development tool. The lack of some blending modes may be an issue when previewing the application. However, the price/performance ratio is great.

The new Nvidia Riva 128 chipset is a great performer. It has excellent potential as an all around perfect production card, as well as great speed and image quality and fast Windows acceleration. Right now, the OpenGL drivers are in testing, but I have high hopes for this card.

The 3Dfx Voodoo Rush chipset is very interesting. It's certainly the king of the hill (right now) as far as actual game performance. However, the OpenGL support beyond full-screen game applications is questionable. The alpha driver that was released was unusable in a development environment, and I have yet to see whether the new beta OpenGL driver will accelerate graphics in a window. That said, a Voodoo card to run the game application combined with either of the other boards for the production tools is a great environment.

The recent announcement that Microsoft is working with SGI to speed up and improve OpenGL driver development could change things dramatically. I will be watching closely.

—Jeff Lander

REFERENCES

I've only scratched the surface of the math behind quaternions. But as you can see, a whole lot of history rests behind them. People interested in reading more about quaternions and their use in computer graphics should read these sources.

Hamilton, Sir William Rowan. "On quaternions; or On a New System of Imaginaries in algebra." ***Philosophical Magazine*****, XXV:10-13, 1844.** This is the original paper describing the actual discovery. However, it's very mathematical and probably not necessary unless you have the desire to learn the underlying theory.

Shoemake, Ken. "Animating Rotation with Quaternion Curves." ***Computer Graphics (Proceedings of Siggraph 1985)*** **1:245-254, 1985.**

Shoemake, Ken. "Euler Angle Conversion." ***Graphic Gems II.*** **Academic Press: 222-229, 1994.** These papers were essential to my understanding of the conversion process. The paper in *Graphic Gems II* actually handles the general case of different rotation orders.

Watt, Alan and Mark Watt. ***Advanced Animation and Rendering Techniques.*** **New York: ACM Press, 1992.** This is a very good book that helped me with the background and comparisons between Euler angles and quaternions.

Laura Downs has written a proof that quaternions really do represent rotations. It's available at http://http.cs.Berkeley.edu/~laura/cs184/quat/quatproof.html. Thanks to Ian Wakelin of Rhythm and Hues Studios for turning me on to that site.

Also be sure to read Nick Bobick's article "Rotating Objects Using Quaternions," in the Feb. 1998 issue of *Game Developer* for other uses and a matrix-based approach.