为什么使用四元数

为了回答这个问题，先来看看一般关于旋转（面向）的描述方法－欧拉描述法。它使用最简单的 x,y,z 值来分别表示在 x，y，z 轴上的旋转角度，其取值为 0-360(或者 0-2pi)，一般使用 roll，pitch，yaw 来表示这些分量的旋转值。需要注意的是，这里的旋转是针对世界坐标系说的，这意味着第一次的旋转不会影响第二、三次的转轴，简单的说，三角度系统无法表现任意轴的旋转，只要一开始旋转，物体本身就失去了任意轴的自主性，这也就导致了万向轴锁（Gimbal Lock）的问题。

还有一种是轴角的描述方法(即我一直以为的四元数的表示法)，这种方法比欧拉描述要好，它避免了 Gimbal Lock，它使用一个 3 维向量表示转轴和一个角度分量表示绕此转轴的旋转角度，即(x,y,z,angle)，一般表示为(x,y,z,w)或者(v,w)。但这种描述法却不适合插值。

那到底什么是 Gimbal Lock 呢？正如前面所说，因为欧拉描述中针对 x,y,z 的旋转描述是世界坐标系下的值，所以当任意一轴旋转 90°的时候会导致该轴同其他轴重合，此时旋转被重合的轴可能没有任何效果，这就是 Gimbal Lock，这里有个例子演示了 Gimbal Lock，点击这里下载。运行这个例子，使用左右箭头改变 yaw 为 90°，此时不管是使用上下箭头还是 Insert、Page Up 键都无法改变 Pitch，而都是改变了模型的 roll。

那么轴、角的描述方法又有什么问题呢？虽然轴、角的描述解决了 Gimbal Lock，但这样的描述方法会导致差值不平滑，差值结果可能跳跃，欧拉描述同样有这样的问题。

什么是四元数

四元数一般定义如下：

q=w+xi+yj+zk

其中 w 是实数，x,y,z 是虚数，其中:

i*i=-1

j*j=-1

k*k=-1

也可以表示为：

q=[w,v]

其中 v=(x,y,z)是矢量，w 是标量，虽然 v 是矢量，但不能简单的理解为 3D 空间的矢量，它是 4 维空间中的的矢量，也是非常不容易想像的。

四元数也是可以归一化的，并且只有单位化的四元数才用来描述旋转（面向），四元数的单位化与 Vector

类似，

首先||q|| = Norm(q)=sqrt(w2 + x2 + y2 + z2)

因为 w2 + x2 + y2 + z2=1

所以 Normlize(q)=q/Norm(q)=q / sqrt(w2 + x2 + y2 + z2)

说了这么多，那么四元数与旋转到底有什么关系？我以前一直认为轴、角的描述就是四元数，如果是那样其与旋转的关系也不言而喻，但并不是这么简单，轴、角描述到四元数的转化：

w = cos(theta/2)

x = ax * sin(theta/2)

y = ay * sin(theta/2)

z = az * sin(theta/2)

其中(ax,ay,az)表示轴的矢量，theta 表示绕此轴的旋转角度，为什么是这样？和轴、角描述到底有什么不同？这是因为轴角描述的“四元组”并不是一个空间下的东西，首先（ax,ay,az）是一个 3 维坐标下的矢量，而 theta 则是级坐标下的角度，简单的将他们组合到一起并不能保证他们插值结果的稳定性，因为他们无法归一化，所以不能保证最终插值后得到的矢量长度（经过旋转变换后两点之间的距离）相等，而四元数在是在一个统一的 4 维空间中，方便归一化来插值，又能方便的得到轴、角这样用于 3D 图像的信息数据，所以用四元数再合适不过了。

关于四元数的运算法则和推导这里有篇详细的文章介绍，重要的是一点，类似与 Matrix 的四元数的乘法是不可交换的，四元数的乘法的意义也类似于 Matrix 的乘法－可以将两个旋转合并，例如：

Q=Q1*Q2

表示 Q 的是先做 Q2 的旋转，再做 Q1 的旋转的结果，而多个四元数的旋转也是可以合并的，根据四元数乘法的定义，可以算出两个四元数做一次乘法需要 16 次乘法和加法，而 3x3 的矩阵则需要 27 运算，所以当有多次旋转操作时，使用四元数可以获得更高的计算效率。

为什么四元数可以避免 Gimbal Lock

在欧拉描述中，之所以会产生 Gimbal Lock 是因为使用的三角度系统是依次、顺序变换的，如果在 OGL 中，代码可能这样：

glRotatef(angleX, 1, 0, 0)

glRotatef(angleY, 0, 1, 0)

glRotatef(angleZ, 0, 0, 1)

注意：以上代码是顺序执行，而使用的又是统一的世界坐标，这样当首先旋转了 Y 轴后，Z 轴将不再是原来的 Z 轴，而可能变成 X 轴，这样针对 Z 的变化可能失效。

而四元数描述的旋转代码可能是这样：

TempQ = From Eula(x,y,z)

FinalQ =CameraQ * NewQ

theta, ax, ay, az = From (FinalQ)

glRotatef(theta, ax, ay, az);

其中(ax,ay,az)描述一条任意轴，theta 描述了绕此任意轴旋转的角度，而所有的参数都来自于所有描述旋转的四元数做乘法之后得到的值，可以看出这样一次性的旋转不会带来问题。这里有个例子演示了使用四元数不会产生 Gimbal Lock 的问题。

关于插值

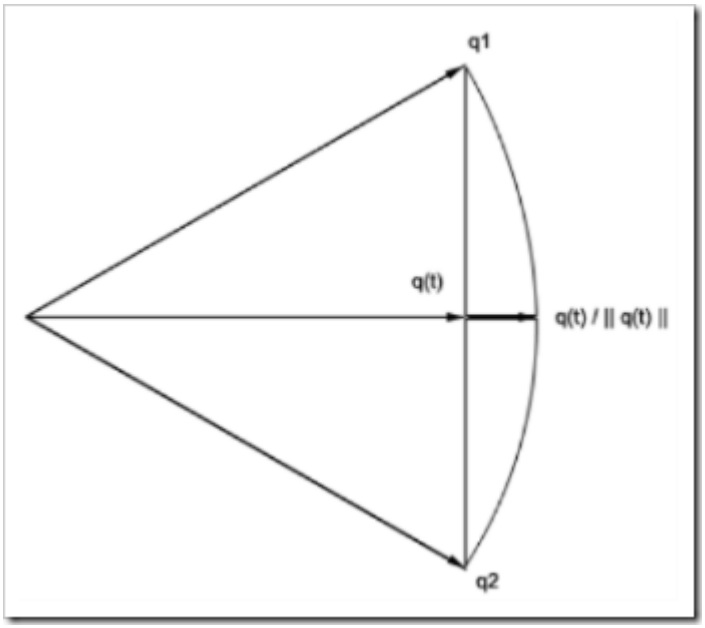
使用四元数的原因就是在于它非常适合插值，这是因为他是一个可以规格化的 4 维向量，最简单的插值算法就是线性插值，公式如：

q(t)=(1-t)q1+t q2

但这个结果是需要规格化的，否则 q(t)的单位长度会发生变化，所以

q(t)=(1-t)q1+t q2 / ‖ (1-t)q1+t q2 ‖

如图:



尽管线性插值很有效，但不能以恒定的速率描述 q1 到 q2 之间的曲线，这也是其弊端，我们需要找到一种插值方法使得 q1->q(t)之间的夹角θ是线性的，即θ(t)=(1-t)θ1+t*θ2，这样我们得到了球形线性插值函数 q(t)，如下：

$$q(t)=q1 * \sin\theta(1-t)/\sin\theta + q2 * \sin\theta t/\sin e\theta$$

如果使用 D3D，可以直接使用 D3DXQuaternionSlerp 函数就可以完成这个插值过程。

第二篇:

四元数入门

四元数常常可以在 3D 的书上看到。

但我的那本 3D 图形学书上，在没讲四元数是干什么的之前，就列了几张纸的公式，

大概因为自己还在上高中，不知道的太多，看了半天没看懂。。。

终于，在 gameres 上看到了某强人翻译的一个"4 元数宝典 "（原文是日本人写的。。。），感觉很好，分享下。

★旋转篇：

我将说明使用了四元数（si yuan shu, quaternion）的旋转的操作步骤

（１）四元数的虚部，实部和写法

所谓四元数，就是把 4 个实数组合起来的东西。

4 个元素中，一个是实部，其余 3 个是虚部。

比如，叫做 Q 的四元数，实部 t 而虚部是 x,y,z 构成，则像下面这样写。

Q = (t; x, y, z)

又，使用向量 V=(x,y,z)，

Q = (t; V)

也可以这么写。

正规地用虚数单位 i,j,k 的写法的话，

Q = t + xi + yj + zk

也这样写，不过，我不大使用

（２）四元数之间的乘法

虚数单位之间的乘法

ii = -1, ij = -ji = k (其他的组合也是循环地以下同文)

有这么一种规则。（我总觉得，这就像是向量积（外积），对吧）

用这个规则一点点地计算很麻烦，所以请用像下面这样的公式计算。

A = (a; U)

B = (b; V)

AB = (ab - U·V; aV + bU + U×V)

不过，“U·V”是内积，「U×V」是外积的意思。

注意：一般 AB<>BA 所以乘法的左右要注意！

（3）3 次元的坐标的四元数表示

如要将某坐标(x,y,z)用四元数表示，

P = (0; x, y, z)

则要这么写。

另外，即使实部是零以外的值，下文的结果也一样。用零的话省事所以我推荐。

（４）旋转的四元数表示

以原点为旋转中心，旋转的轴是(α, β, γ)

（但 α^2 + β^2 + γ^2 = 1），

（右手系的坐标定义的话，望向向量(α, β, γ)的前进方向反时针地）

转θ角的旋转，用四元数表示就是，

Q = (cos(θ/2); α sin(θ/2), β sin(θ/2), γ sin(θ/2))

R = (cos(θ/2); -α sin(θ/2), -β sin(θ/2), -γ sin(θ/2))

(另外 R 叫 Q 的共轭四元数。）

那么，如要实行旋转，

则 R P Q = (0; 答案)

请像这样三明治式地计算。这个值的虚部就是旋转之后的点的坐标值。

（另外，实部应该为零。请验算看看）

例子代码

```
/// Quaternion.cpp
/// (C) Toru Nakata, toru-nakata@aist.go.jp
/// 2004 Dec 29

#include <math.h>
#include <iostream.h>

/// Define Data type
typedef struct
{
        double t; // real-component
        double x; // x-component
```

```
        double y; // y-component
        double z; // z-component
} quaternion;



//// Bill 注：Kakezan 在日语里是“乘法”的意思
quaternion Kakezan(quaternion left, quaternion right)
{
        quaternion ans;
        double d1, d2, d3, d4;

        d1 =  left.t * right.t;
        d2 = -left.x * right.x;
        d3 = -left.y * right.y;
        d4 = -left.z * right.z;
        ans.t = d1+ d2+ d3+ d4;

        d1 =  left.t * right.x;
        d2 =  right.t * left.x;
        d3 =  left.y * right.z;
        d4 = -left.z * right.y;
        ans.x =  d1+ d2+ d3+ d4;

        d1 =  left.t * right.y;
        d2 =  right.t * left.y;
        d3 =  left.z * right.x;
        d4 = -left.x * right.z;
        ans.y =  d1+ d2+ d3+ d4;

        d1 =  left.t * right.z;
```

```
	d2 =  right.t * left.z;
	d3 =  left.x * right.y;
	d4 = -left.y * right.x;
	ans.z =  d1+ d2+ d3+ d4;

	return ans;
}

//// Make Rotational quaternion
quaternion MakeRotationalQuaternion(double radian, double AxisX, double AxisY, double Axis
Z)
{
	quaternion ans;
	double norm;
	double ccc, sss;

	ans.t = ans.x = ans.y = ans.z = 0.0;

	norm = AxisX *  AxisX +  AxisY *  AxisY +  AxisZ *  AxisZ;
	if(norm <= 0.0) return ans;

	norm = 1.0 / sqrt(norm);
	AxisX *= norm;
	AxisY *= norm;
	AxisZ *= norm;

	ccc = cos(0.5 * radian);
	sss = sin(0.5 * radian);

	ans.t = ccc;
```

```
		ans.x = sss * AxisX;
		ans.y = sss * AxisY;
		ans.z = sss * AxisZ;

		return ans;
}

//// Put XYZ into  quaternion
quaternion PutXYZToQuaternion(double PosX, double PosY, double PosZ)
{
		quaternion ans;

		ans.t = 0.0;
		ans.x = PosX;
		ans.y = PosY;
		ans.z = PosZ;

		return ans;
}

///// main
int main()
{
		double px, py, pz;
		double ax, ay, az, th;
		quaternion ppp, qqq, rrr;

		cout << "Point Position (x, y, z) " << endl;
		cout << "  x = ";
		cin >> px;
```

```
		cout << "  y = ";
		cin >> py;
		cout << "  z = ";
		cin >> pz;
		ppp = PutXYZToQuaternion(px, py, pz);

		while(1) {
			cout << "\nRotation Degree ? (Enter 0 to Quit) " << endl;
			cout << "  angle = ";
			cin >> th;
			if(th == 0.0) break;

			cout << "Rotation Axis Direction ? (x, y, z) " << endl;
			cout << "  x = ";
			cin >> ax;
			cout << "  y = ";
			cin >> ay;
			cout << "  z = ";
			cin >> az;


			th *= 3.14159265358979323846264338327795 / 180.0; /// Degree -> radi
an;

			qqq = MakeRotationalQuaternion(th, ax, ay, az);
			rrr = MakeRotationalQuaternion(-th, ax, ay, az);

			ppp = Kakezan(rrr, ppp);
			ppp = Kakezan(ppp, qqq);
```

```
                cout << "\nAnser X = " << ppp.x
                        <<  "\n      Y = " << ppp.y
                        <<  "\n      Z = " << ppp.z << endl;


        }


        return 0;
}
```

又一篇

在以前涉及的程序中，处理物体的旋转通常是用的矩阵的形式。由于硬件在纹理映射和光栅化上的加强，程序员可以将更多的 CPU 周期用于物理模拟等工作，这样将使得程序更为逼真。

一，原文出处：
http://www.gamasutra.com/features/19980703/quaternions_01.htm

二，摘录：

有三种办法表示旋转：矩阵表示，欧拉角表示，以及四元组表示。矩阵，欧拉角表示法在处理插值的时候会遇到麻烦。
There are many ways to represent the orientation of an object. Most programmers use 3x3 rotation matrices or three Euler angles to store this information. Each of these solutions works fine until you try to smoothly interpolate between two orientations of an object.

矩阵表示法不适合于进行插值(在转动的前后两个朝向之间取得瞬时朝向)。矩阵有 9 个自由度（3×3 矩阵），而实际上表示一个旋转只需要 3 个自由度（旋转轴 3）。
Rotations involve only three degrees of freedom (DOF), around the x, y, and z coordinate axes. However, nine DOF (assuming 3x3 matrices) are required to constrain the rotation - clearly more than we need.

Another shortcoming of rotation matrices is that they are extremely hard to use for interpolating rotations between two orientations. The resulting interpolations are also visually very jerky, which simply is not acceptable in games any more.

欧拉角表示法
You can also use angles to represent rotations around three coordinate axes. You can write this as (q, c, f); simply stated, "Transform a point by rotating it counterclockwise about the z axis by q degrees, followed by a rotation about the y axis by c degrees, followed by a rotation about the x axis by f degrees."

欧拉角表示法的缺陷：把一个旋转变成了一系列的旋转。并且，进行插值计算也不方便。
However, there is no easy way to represent a single rotation with Euler angles that corresponds to a series of concatenated rotations.
Furthermore, the smooth interpolation between two orientations involves numerical integration

目录

[隐藏]

[编辑]

四元组表示法

[w, v]其中 v 是矢量，表示旋转轴。w 标量，表示旋转角度。所以，一个四元组即表示一个完整的旋转。
There are several notations that we can use to represent quaternions. The two most popular notations are complex number notation (Eq. 1) and 4D vector notation (Eq. 2).

w + xi + yj + zk (where i2 = j2 = k2 = -1 and ij = k = -ji with real w, x, y, z) (Eq. 1)

[w, v] (where v = (x, y, z) is called a "vector" and w is called a "scalar") (Eq. 2)

4D 空间以及单元四元组：
Each quaternion can be plotted in 4D space (since each quaternion is comprised of four parts), and this space is called quaternion space. Unit quaternions have the property that their magnitude is one and they form a subspace, S3, of the quaternion space. This subspace can be represented as a 4D sphere. (those that have a one-unit normal), since this reduces the number of necessary operations that you have to perform.

[编辑]

四元组的基本运算法则

Table 1. Basic operations using quaternions.

Addition: q + q′ = [w + w′, v + v′]

Multiplication: qq′ = [ww′ − v • v′, v x v′ + wv′ +w′v] (• is vector dot product and x is vector cross product); Note: qq′ ? q′q
//为什么是这样？——定义成这样滴，木有道理可以讲
Conjugate: q* = [w, −v] 共轭

Norm: N(q) = w2 + x2 + y2 + z2 模

Inverse: q-1 = q* / N(q)

Unit Quaternion: q is a unit quaternion if N(q)= 1 and then q-1 = q*

Identity: [1, (0, 0, 0)] (when involving multiplication) and [0, (0, 0, 0)] (when involving addition)

[编辑]

用四元组进行旋转的公式

重要!!!!：只有单元四元组才表示旋转。**为什么????——已解决（shoemake 有详细证明）**
It is extremely important to note that only unit quaternions represent

rotations, and you can assume that when I talk about quaternions, I'm talking about unit quaternions unless otherwise specified.

Since you've just seen how other methods represent rotations, let's see how we can specify rotations using quaternions. It can be proven (and the proof isn't that hard) that the rotation of a vector v by a unit quaternion q can be represented as

v′ = q v q-1 (where v = [0, v]) (Eq. 3)

//

四元组和旋转参数的变换一个四元组由(x,y,z,w)四个变量表达,假设给定一个旋转轴 axis 和角度 angle,那么这个四元组的计算公式为:

```
void Quaternion::fromAxisAngle(const Vector3 &axis, Real angle)
{
	Vector3 u = axis;
	u.normalize();
	Real s = Math::rSin(angle/2.f);
	x = s*u.x;
	y = s*u.y;
	z = s*u.z;
	w = Math::rCos(angle/2.f);
}
```

由四元组反算回旋转轴 axis 和角度 angle 的公式为:

```
/
void Quaternion::toAxisAngle(Vector3 &axis, Real &angle) const
{
	angle = acos(w)*2
	axis.x = x
	axis.y = y
	axis.z = z
}
```

英文版:

Rotating Objects Using Quaternions

Last year may go down in history as The Year of the Hardware Acceleration. Much of the work rasterizing and texture-mapping polygons was off-loaded to dedicated hardware. As a result, we game developers now have a lot of CPU cycles to spare for physics simulation and other features. Some of those extra cycles can be applied to tasks such as smoothing rotations and animations, thanks to quaternions.

Many game programmers have already discovered the wonderful world of quaternions and have started to use them extensively. Several third-person games, including both TOMB RAIDER titles, use quaternion rotations to animate all of their camera movements. Every third-person game has a virtual camera placed at some distance behind or to the side of the player's character. Because this camera goes through different motions (that is, through arcs of a different lengths) than the character, camera motion can appear unnatural and too "jerky" for the player to follow the action. This is one area where quaternions come to rescue.

Another common use for quaternions is in military and commercial flight simulators. Instead of manipulating a plane's orientation using three angles (roll, pitch, and yaw) representing rotations about the x, y, and z axes, respectively, it is much simpler to use a single quaternion.

Many games coming out this year will also feature real-world physics, allowing amazing game play and immersion. If you store orientations as quaternions, it is computationally less expensive to add angular velocities to quaternions than to matrices.

Both *Tomb Raider* titles use quaternion rotations to animate camera movements.

There are many ways to represent the orientation of an object. Most programmers use 3x3 rotation matrices or three Euler angles to store this information. Each of these solutions works

fine until you try to smoothly interpolate between two orientations of an object. Imagine an object that is not user controlled, but which simply rotates freely in space (for example, a revolving door). If you chose to store the door's orientations as rotation matrices or Euler angles, you'd find that smoothly interpolating between the rotation matrices' values would be computationally costly and certainly wouldn't appear as smooth to a player's eye as quaternion interpolation.

Trying to correct this problem using matrices or Euler angles, an animator might simply increase the number of predefined (keyed) orientations. However, one can never be sure how many such orientations are enough, since the games run at different frame rates on different computers, thereby affecting the smoothness of the rotation. This is a good time to use quaternions, a method that requires only two or three orientations to represent a simple rotation of an object, such as our revolving door. You can also dynamically adjust the number of interpolated positions to correspond to a particular frame rate.

Before we begin with quaternion theory and applications, let's look at how rotations can be represented. I'll touch upon methods such as rotation matrices, Euler angles, and axis and angle representations and explain their shortcomings and their relationships to quaternions. If you are not familiar with some of these techniques, I recommend picking up a graphics book and studying them.

To date, I haven't seen a single 3D graphics book that doesn't talk about rotations using 4x4 or 3x3 matrices. Therefore, I will assume that most game programmers are very familiar with this technique and I'll just comment on its shortcomings. I also highly recommend that you re-read Chris Hecker's article in the June 1997 issue of the *Game Developer* ("Physics, Part 4: The Third Dimension," pp.15-26), since it tackles the problem of orienting 3D objects.

Rotations involve only three degrees of freedom (DOF), around the x, y, and z coordinate axes. However, nine DOF (assuming 3x3 matrices) are required to constrain the rotation - clearly more than we need. Additionally, matrices are prone to "drifting," a situation that arises when one of the six constraints is violated and the matrix introduces rotations around an arbitrary axis. Combatting this problem requires keeping a matrix orthonormalized - making sure that it obeys constraints. However, doing so is not computationally cheap. A common way of solving matrix drift relies on the Gram-Schmidt algorithm for conversion of an arbitrary basis into an orthogonal basis. Using the Gram-Schmidt algorithm or calculating a correction matrix to solve matrix drifting can take a lot of CPU cycles, and it has to be done very often, even when using floating point math.

Another shortcoming of rotation matrices is that they are extremely hard to use for interpolating rotations between two orientations. The resulting interpolations are also visually very jerky, which simply is not acceptable in games any more.

You can also use angles to represent rotations around three coordinate axes. You can write this as (q, c, f); simply stated, "Transform a point by rotating it counterclockwise about the z axis

by q degrees, followed by a rotation about the y axis by c degrees, followed by a rotation about the x axis by f degrees." There are 12 different conventions that you can use to represent rotations using Euler angles, since you can use any combination of axes to represent rotations (XYZ, XYX, XYY…). We will assume the first convention (XYZ) for all of the presented examples. I will assume that all of the positive rotations are counterclockwise (Figure 1).

Euler angle representation is very efficient because it uses only three variables to represent three DOF. Euler angles also don't have to obey any constraints, so they're not prone to drifting and don't have to be readjusted.

However, there is no easy way to represent a single rotation with Euler angles that corresponds to a series of concatenated rotations. Furthermore, the smooth interpolation between two orientations involves numerical integration, which can be computationally expensive. Euler angles also introduce the problem of "Gimbal lock" or a loss of one degree of rotational freedom. Gimbal lock happens when a series of rotations at 90 degrees is performed; suddenly, the rotation doesn't occur due to the alignment of the axes.

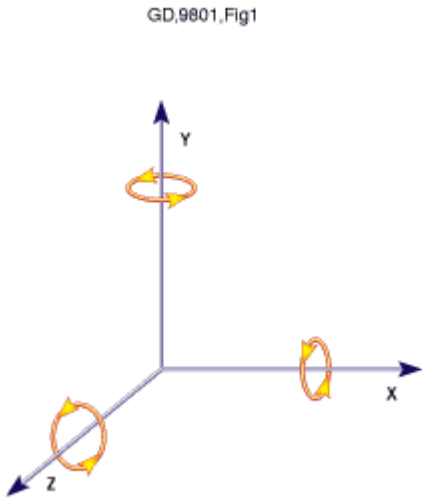


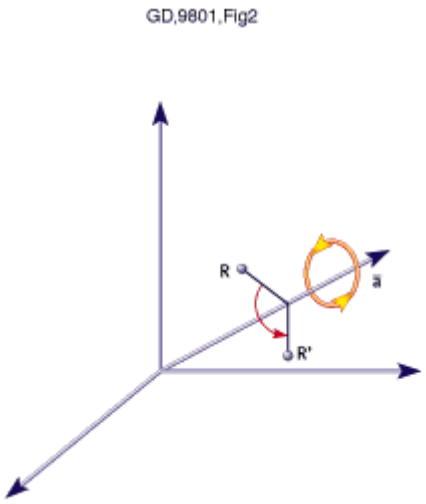
Figure 1: Euler angle representation.

For example, imagine that a series of rotations to be performed by a flight simulator. You specify the first rotation to be Q1 around the x axis, the second rotation to be 90 degrees around the y axis, and Q3 to be the rotation around the z axis. If you perform specified rotations in succession, you will discover that Q3 rotation around the z axis has the same effect as the rotation around the initial x axis. The y axis rotation has caused the x and z axes to get aligned, and you have just lost a DOF because rotation around one axis is equivalent to opposite rotation around the other axis. I highly recommend Advanced *Animation and Rendering Techniques: Theory and Practice* by Alan and Mark Watt (Addison Wesley, 1992) for a detailed discussion of the Gimbal lock problem.

Using an axis and angle representation is another way of representing rotations. You specify an arbitrary axis and an angle (positive if in a counterclockwise direction), as illustrated in Figure

2.

Even though this is an efficient way of representing a rotation, it suffers from the same problems that I described for Euler angle representation (with the exception of the Gimbal lock problem).



In the eighteenth century, W. R. Hamilton devised quaternions as a four-dimensional extension to complex numbers. Soon after this, it was proven that quaternions could also represent rotations and orientations in three dimensions. There are several notations that we can use to represent quaternions. The two most popular notations are complex number notation (Eq. 1) and 4D vector notation (Eq. 2).

$w + xi + yj + zk$ (where $i^2 = j^2 = k^2 = -1$ and $ij = k = -ji$ with real w, x, y, z)
(Eq. 1)

[w, v] (where v = (x, y, z) is called a "vector" and w is called a "scalar")
(Eq. 2)

I will use the second notation throughout this article. Now that you know how quaternions are represented, let's start with some basic operations that use them.

If q and q´ are two orientations represented as quaternions, you can define the operations in Table 1 on these quaternions.

All other operations can be easily derived from these basic ones, and they are fully documented in the accompanying library, which you can find **here**. I will also only deal with unit quaternions. Each quaternion can be plotted in 4D space (since each quaternion is comprised of four parts), and this space is called quaternion space. Unit quaternions have the property that their magnitude is one and they form a subspace, S3, of the quaternion space. This subspace can be represented as a 4D sphere. (those that have a one-unit normal), since this

reduces the number of necessary operations that you have to perform.

It is extremely important to note that only unit quaternions represent rotations, and you can assume that when I talk about quaternions, I'm talking about unit quaternions unless otherwise specified.

Since you've just seen how other methods represent rotations, let's see how we can specify rotations using quaternions. It can be proven (and the proof isn't that hard) that the rotation of a vector v by a unit quaternion q can be represented as

$v' = q\ v\ q^{-1}$ (where $v = [0, v]$)
(Eq. 3)

The result, a rotated vector v´, will always have a 0 scalar value for w (recall Eq. 2 earlier), so you can omit it from your computations.

Table 1. Basic operations using quaternions.

Addition: $q + q' = [w + w', v + v']$

Multiplication: $qq' = [ww' - v \cdot v', v \times v' + wv' + w'v]$ (· is vector dot product and x is vector cross product); Note: $qq' \neq q'q$

Conjugate: $q^* = [w, -v]$

Norm: $N(q) = w^2 + x^2 + y^2 + z^2$

Inverse: $q^{-1} = q^* / N(q)$

Unit Quaternion: q is a unit quaternion if $N(q) = 1$ and then $q^{-1} = q^*$

Identity: [1, (0, 0, 0)] (when involving multiplication) and [0, (0, 0, 0)] (when involving addition)

Today's most widely supported APIs, Direct3D immediate mode (retained mode does have a limited set of quaternion rotations) and OpenGL, do not support quaternions directly. As a result, you have to convert quaternion orientations in order to pass this information to your favorite API. Both OpenGL and Direct3D give you ways to specify rotations as matrices, so a quaternion-to-matrix conversion routine is useful. Also, if you want to import scene information from a graphics package that doesn't store its rotations as a series of quaternions (such as NewTek's LightWave), you need a way to convert to and from quaternion space.

ANGLE AND AXIS. Converting from angle and axis notation to quaternion notation involves two trigonometric operations, as well as several multiplies and divisions. It can be represented

as

q = [cos(Q/2), sin(Q /2)v] (where Q is an angle and v is an axis)
(Eq. 4)

EULER ANGLES. Converting Euler angles into quaternions is a similar process - you just have to be careful that you perform the operations in the correct order. For example, let's say that a plane in a flight simulator first performs a yaw, then a pitch, and finally a roll. You can represent this combined quaternion rotation as

q = qyaw qpitch qroll where:
qroll = [cos (y/2), (sin(y/2), 0, 0)]
qpitch = [cos (q/2), (0, sin(q/2), 0)]
qyaw = [cos(f /2), (0, 0, sin(f /2)]
(Eq. 5)

The order in which you perform the multiplications is important. Quaternion multiplication is not commutative (due to the vector cross product that's involved). In other words, changing the order in which you rotate an object around various axes can produce different resulting orientations, and therefore, the order is important.

ROTATION MATRIX. Converting from a rotation matrix to a quaternion representation is a bit more involved, and its implementation can be seen in Listing 1.

Conversion between a unit quaternion and a rotation matrix can be specified as

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

(Eq. 6)

It's very difficult to specify a rotation directly using quaternions. It's best to store your character's or object's orientation as a Euler angle and convert it to quaternions before you start interpolating. It's much easier to increment rotation around an angle, after getting the user's input, using Euler angles (that is, roll = roll + 1), than to directly recalculate a quaternion.

Since converting between quaternions and rotation matrices and Euler angles is performed often, it's important to optimize the conversion process. Very fast conversion (involving only nine muls) between a unit quaternion and a matrix is presented in Listing 2. Please note that the code assumes that a matrix is in a right-hand coordinate system and that matrix rotation is represented in a column major format (for example, OpenGL compatible).

Listing 1: Matrix to quaternion code.

```
MatToQuat(float m[4][4], QUAT * quat)
{
  float  tr, s, q[4];
  int    i, j, k;


  int nxt[3] = {1, 2, 0};


  tr = m[0][0] + m[1][1] + m[2][2];


  // check the diagonal
  if (tr > 0.0) {
    s = sqrt (tr + 1.0);
    quat->w = s / 2.0;
    s = 0.5 / s;
    quat->x = (m[1][2] - m[2][1]) * s;
    quat->y = (m[2][0] - m[0][2]) * s;
    quat->z = (m[0][1] - m[1][0]) * s;
} else {
        // diagonal is negative
         i = 0;
         if (m[1][1] > m[0][0]) i = 1;
            if (m[2][2] > m[i][i]) i = 2;
           j = nxt[i];
           k = nxt[j];


           s = sqrt ((m[i][i] - (m[j][j] + m[k][k])) + 1.0);

           q[i] = s * 0.5;

           if (s != 0.0) s = 0.5 / s;


          q[3] = (m[j][k] - m[k][j]) * s;
          q[j] = (m[i][j] + m[j][i]) * s;
          q[k] = (m[i][k] + m[k][i]) * s;
```

```
        quat->x = q[0];
        quat->y = q[1];
        quat->z = q[2];
        quat->w = q[3];
  }
}
```

If you aren't dealing with unit quaternions, additional multiplications and a division are required. Euler angle to quaternion conversion can be coded as shown in Listing 3.

One of the most useful aspects of quaternions that we game programmers are concerned with is the fact that it's easy to interpolate between two quaternion orientations and achieve smooth animation. To demonstrate why this is so, let's look at an example using spherical rotations. Spherical quaternion interpolations follow the shortest path (arc) on a four-dimensional, unit quaternion sphere. Since 4D spheres are difficult to imagine, I'll use a 3D sphere (Figure 3) to help you visualize quaternion rotations and interpolations.

Let's assume that the initial orientation of a vector emanating from the center of the sphere can be represented by q1 and the final orientation of the vector is q3. The arc between q1 and q3 is the path that the interpolation would follow. Figure 3 also shows that if we have an intermediate position q2, the interpolation from q1 -> q2 -> q3 will not necessarily follow the same path as the q1 ->q3 interpolation. The initial and final orientations are the same, but the arcs are not.

Quaternions simplify the calculations required when compositing rotations. For example, if you have two or more orientations represented as matrices, it is easy to combine them by multiplying two intermediate rotations.

R = R2R1 (rotation R1 followed by a rotation R2)
(Eq. 7)

Listing 2: Quaternion-to-matrix conversion. *([07.30.02] Editor's Note: the following `QuatToMatrix` function originally was published with a bug -- it reversed the row/column ordering. This is the correct version. Thanks to John Ratcliff and Eric Haines for pointing this out.)*

```
QuatToMatrix(QUAT * quat, float m[4][4]){

float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;


// calculate coefficients
x2 = quat->x + quat->x; y2 = quat->y + quat->y;
```

```
z2 = quat->z + quat->z;
xx = quat->x * x2; xy = quat->x * y2; xz = quat->x * z2;
yy = quat->y * y2; yz = quat->y * z2; zz = quat->z * z2;
wx = quat->w * x2; wy = quat->w * y2; wz = quat->w * z2;


m[0][0] = 1.0 - (yy + zz); m[1][0] = xy - wz;
m[2][0] = xz + wy; m[3][0] = 0.0;

m[0][1] = xy + wz; m[1][1] = 1.0 - (xx + zz);
m[2][1] = yz - wx; m[3][1] = 0.0;


m[0][2] = xz - wy; m[1][2] = yz + wx;
m[2][2] = 1.0 - (xx + yy); m[3][2] = 0.0;


m[0][3] = 0; m[1][3] = 0;
m[2][3] = 0; m[3][3] = 1;

}
```

This composition involves 27 multiplications and 18 additions, assuming 3x3 matrices. On the other hand, a quaternion composition can be represented as

q = q2q1 (rotation q1 followed by a rotation q2)
(Eq. 8)

As you can see, the quaternion method is analogous to the matrix composition. However, the quaternion method requires only eight multiplications and four divides (Listing 4), so compositing quaternions is computationally cheap compared to matrix composition. Savings such as this are especially important when working with hierarchical object representations and inverse kinematics.

Now that you have an efficient multiplication routine, see how can you interpolate between two quaternion rotations along the shortest arc. Spherical Linear intERPolation (SLERP) achieves this and can be written as

$$\mathrm{SLERP}(p,q,t) = \frac{p\sin((1-t)\theta) + q\sin(t\theta)}{\sin(\theta)}$$

(Eq. 9)

where pq = cos(q) and parameter t goes from 0 to 1. The implementation of this equation is

presented in Listing 5. If two orientations are too close, you can use linear interpolation to avoid any divisions by zero.

GD,9801,Fig3

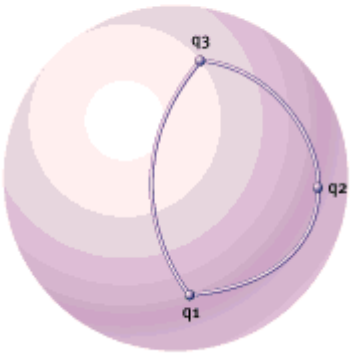


Figure 3. Quaternion rotations.

Listing 3: Euler-to-quaternion conversion.

```
EulerToQuat(float roll, float pitch, float yaw, QUAT * quat)
{
        float cr, cp, cy, sr, sp, sy, cpcy, spsy;


// calculate trig identities
cr = cos(roll/2);

        cp = cos(pitch/2);
        cy = cos(yaw/2);


        sr = sin(roll/2);
        sp = sin(pitch/2);
        sy = sin(yaw/2);

        cpcy = cp * cy;
        spsy = sp * sy;


        quat->w = cr * cpcy + sr * spsy;
        quat->x = sr * cpcy - cr * spsy;
        quat->y = cr * sp * cy + sr * cp * sy;
        quat->z = cr * cp * sy - sr * sp * cy;
```

```
}
```

The basic SLERP rotation algorithm is shown in Listing 6. Note that you have to be careful that your quaternion represents an absolute and not a relative rotation. You can think of a relative rotation as a rotation from the previous (intermediate) orientation and an absolute rotation as the rotation from the initial orientation. This becomes clearer if you think of the q2 quaternion orientation in Figure 3 as a relative rotation, since it moved with respect to the q1 orientation. To get an absolute rotation of a given quaternion, you can just multiply the current relative orientation by a previous absolute one. The initial orientation of an object can be represented as a multiplication identity [1, (0, 0, 0)]. This means that the first orientation is always an absolute one, because

q = qidentity q
(Eq. 10)

Listing 4: Efficient quaternion multiplication.

```
QuatMul(QUAT *q1, QUAT *q2, QUAT *res){


float A, B, C, D, E, F, G, H;


A = (q1->w + q1->x)*(q2->w + q2->x);
B = (q1->z - q1->y)*(q2->y - q2->z);
C = (q1->w - q1->x)*(q2->y + q2->z);
D = (q1->y + q1->z)*(q2->w - q2->x);
E = (q1->x + q1->z)*(q2->x + q2->y);
F = (q1->x - q1->z)*(q2->x - q2->y);
G = (q1->w + q1->y)*(q2->w - q2->z);
H = (q1->w - q1->y)*(q2->w + q2->z);


res->w = B + (-E - F + G + H) /2;
res->x = A - (E + F + G + H)/2;
res->y = C + (E - F + G - H)/2;
res->z = D + (E - F - G + H)/2;
}
```

As I stated earlier, a practical use for quaternions involves camera rotations in third-person-perspective games. Ever since I saw the camera implementation in TOMB RAIDER, I've wanted to implement something similar. So let's implement a third-person camera (Figure 4).

To start off, let's create a camera that is always positioned above the head of our character and that points at a spot that is always slightly above the character's head. The camera is also positioned d units behind our main character. We can also implement it so that we can vary the roll (angle q in Figure 4) by rotating around the x axis.

As soon as a player changes the orientation of the character, you rotate the character instantly and use SLERP to reorient the camera behind the character (Figure 5). This has the dual benefit of providing smooth camera rotations and making players feel as though the game responded instantly to their input.

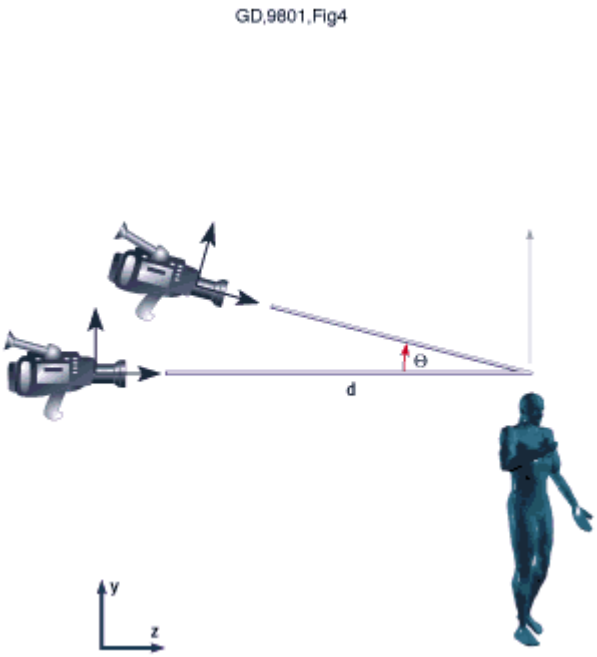


Figure 4. Third-person camera.

GD,9801,Fig4

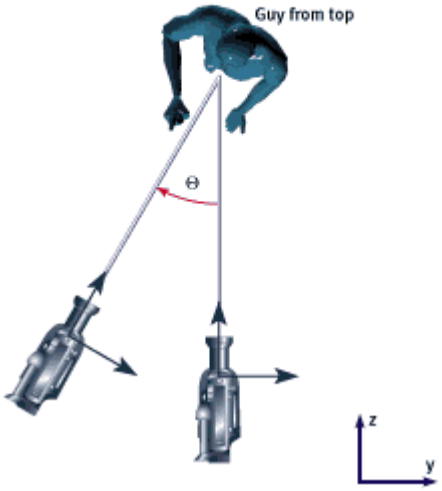


Figure 5. Camera from top.

You can set the camera's center of rotation (pivot point) as the center of the object it is tracking. This allows you to piggyback on the calculations that the game already makes when the character moves within the game world.

Note that I do not recommend using quaternion interpolation for first-person action games since these games typically require instant response to player actions, and SLERP does take time.

However, we can use it for some special scenes. For instance, assume that you're writing a tank simulation. Every tank has a scope or similar targeting mechanism, and you'd like to simulate it as realistically as possible. The scoping mechanism and the tank's barrel are controlled by a series of motors that players control. Depending on the zoom power of the scope and the distance to a target object, even a small movement of a motor could cause a large change in the viewing angle, resulting in a series of huge, seemingly disconnected jumps between individual frames. To eliminate this unwanted effect, you could interpolate the orientation according to the zoom and distance of object. This type of interpolation between two positions over several frames helps dampen the rapid movement and keeps players from becoming disoriented.

Another useful application of quaternions is for prerecorded (but not prerendered) animations. Instead of recording camera movements by playing the game (as many games do today), you could prerecord camera movements and rotations using a commercial package such as Softimage 3D or 3D Studio MAX. Then, using an SDK, export all of the keyframed camera/object quaternion rotations. This would save both space and rendering time. Then you could just play the keyframed camera motions whenever the script calls for cinematic scenes.

Listing 5: SLERP implementation.

```
QuatSlerp(QUAT * from, QUAT * to, float t, QUAT * res)
     {
       float          to1[4];
       double       omega, cosom, sinom, scale0, scale1;


       // calc cosine
       cosom = from->x * to->x + from->y * to->y + from->z * to->z
                             + from->w * to->w;


       // adjust signs (if necessary)
       if ( cosom <0.0 ){ cosom = -cosom; to1[0] = - to->x;
                to1[1] = - to->y;
                to1[2] = - to->z;
                to1[3] = - to->w;
       } else  {
                to1[0] = to->x;
                to1[1] = to->y;
                to1[2] = to->z;
                to1[3] = to->w;
       }


       // calculate coefficients


      if ( (1.0 - cosom) > DELTA ) {
              // standard case (slerp)
              omega = acos(cosom);
              sinom = sin(omega);
              scale0 = sin((1.0 - t) * omega) / sinom;
              scale1 = sin(t * omega) / sinom;


       } else {
    // "from" and "to" quaternions are very close
            //  ... so we can do a linear interpolation
              scale0 = 1.0 - t;
              scale1 = t;
       }
```

```
        // calculate final values
        res->x = scale0 * from->x + scale1 * to1[0];
        res->y = scale0 * from->y + scale1 * to1[1];
        res->z = scale0 * from->z + scale1 * to1[2];
        res->w = scale0 * from->w + scale1 * to1[3];
}
```

After reading Chris Hecker's columns on physics last year, I wanted to add angular velocity to a game engine on which I was working. Chris dealt mainly with matrix math, and because I wanted to eliminate quaternion-to-matrix and matrix-to-quaternion conversions (since our game engine is based on quaternion math), I did some research and found out that it is easy to add angular velocity (represented as a vector) to a quaternion orientation. The solution (Eq. 11) can be represented as a differential equation.

$$\frac{dQ}{dt} + 0.5 * \text{quat}(\text{angular}) * Q$$

(Eq. 11)

where quat(angular) is a quaternion with a zero scalar part (that is, w = 0) and a vector part equal to the angular velocity vector. Q is our original quaternion orientation.

To integrate the above equation (Q + dQ/dt), I recommend using the Runge-Kutta order four method. If you are using matrices, the Runge-Kutta order five method achieves better results within a game. (The Runge-Kutta method is a way of integrating differential equations. A complete description of the method can be found in any elementary numerical algorithm book, such as Numerical Recipes in C. It has a complete section devoted to numerical, differential integration.) For a complete derivation of angular velocity integration, consult Dave Baraff's SIGGRAPH tutorials.

Quaternions can be a very efficient and extremely useful method of storing and performing rotations, and they offer many advantages over other methods. Unfortunately, they are also impossible to visualize and completely unintuitive. However, if you use quaternions to represent rotations internally, and use some other method (for example, angle-axis or Euler angles) as an immediate representation, you won't have to visualize them.

Nick Bobick is a game developer at Caged Entertainment Inc. and he is currently working on a cool 3D game. He can be contacted at nb@netcom.ca. ***The author would like to thank Ken Shoemake for his research and publications. Without him, this article would not have been possible.***