ADVANCED ANIMATION AND RENDERING TECHNIQUES

Theory and Practice

Alan Watt
University of Sheffield

Mark Watt
Xaos, San Francisco

ACM Press
New York, New York

Addison-Wesley
Harlow, England · Reading, Massachusetts · Menlo Park, California
New York · Don Mills, Ontario · Amsterdam · Bonn · Sydney · Singapore
Tokyo · Madrid · San Juan · Milan · Mexico City · Seoul · Taipei

These complications arise because the motion specification is not properly divorced from the path specification. Moreover, as we have seen, driving the motion via u directly has a disadvantage in that it is impossible to get an object moving with constant speed along the path. Only the rigorous approach of arclength reparametrization, which treats the specification of path and motion completely separately, can provide this.

15.3.8 Parametrization of orientation

This section deals with the problems that are encountered when parametrizing the space of all possible orientations of an object, where all orientations, or rotations, take place about a point fixed in space with respect to that object. We begin by looking at a common, but as we shall see somewhat inadequate, method for animating rotation – Euler angles.

Euler angles

Historically, the most popular parametrization of orientation space, well established through appearing in standard maths and physics textbooks, has been in terms of Euler angles, where a general rotation is described as a sequence of rotations about three mutually orthogonal coordinate axes fixed in space. (Note that the rotations are applied to the space and not to the axes.) This has led to animators setting up general orientation as a composite of these axis rotations which we will call 'rolls': x-roll for rotation about the x-axis, y-roll for rotation about the y-axis and z-roll for rotation about the z-axis. These rolls, in homogeneous matrix notation, give rise to the principal rotation matrices shown in Figure 15.15.

The precise order in which these rolls are applied lead to different definitions of the parametrization of orientation in terms of Euler angles. These considerations do not concern us here. In general, an angular displacement has three degrees of freedom, and since each principle rotation matrix has but one degree of freedom, a minimum of three principle rotations must be combined to represent a general angular displacement. Let us choose an x-roll, followed by a y-roll, followed by a z-roll. Our parametrization of orientation space is thus a general rotation matrix $R(\Theta_1, \Theta_2, \Theta_3)$ in terms of the Euler angles Θ_1, Θ_2, Θ_3 given by:

$$\begin{bmatrix} c_2c_3 & c_2s_3 & -s_2 & 0 \\ s_1s_2c_3 - c_1s_3 & s_1s_2s_3 + c_1c_3 & s_1c_2 & 0 \\ c_1s_2c_3 + s_1s_3 & c_1s_2s_3 - s_1c_3 & c_1c_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

$s_i = \sin \Theta_i$ and $c_i = \cos \Theta_i$

In general there are 12 possible ways in which to define a rotation in terms of Euler angles, each one resulting in a different form for the above rotation matrix.

Because of its historical popularity, computer animation systems were quick to use Euler angles as parameters for animating orientation. There are two major drawbacks to this approach, however. The first is a practical

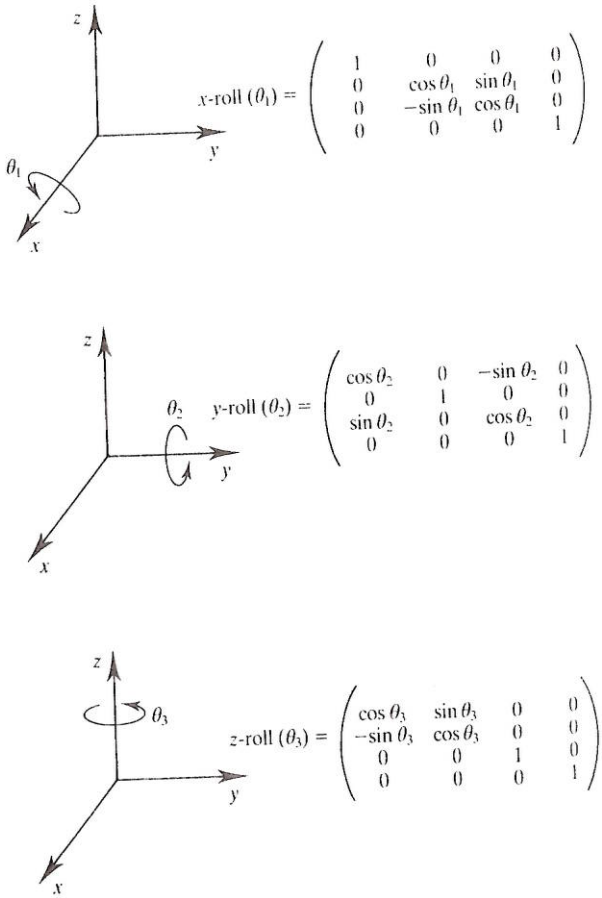

Figure 15.15 The principal rotation matrices.

problem often encountered by animators trying to set up an arbitrary orientation using Euler angles, and the second is a mathematically deep objection to their use when interpolating orientation. Both of these problems occur because Euler angles ignore the interaction of the rolls about the separate axes. As we shall see these rolls are not independent of each other.

Euler angles and Gimbal lock

'Gimbal lock' is a term derived from a mechanical problem that arises in the gimbal mechanism used to support a compass or gyroscope. These generally consist of three concentric frames or rings and under certain rotations a degree of freedom is lost – the mechanism exhibits gimbal lock.

Suppose an animator uses the above parametrization to set up an arbitrary orientation. First, the animator applies an x-roll, then a y-roll and finally a z-roll in order to move an object into a required orientation. Suppose further that during this process the animator innocently specifies a y-roll of $\pi/2$. In dismay he will discover that the subsequent rotation about the z-axis has an effect that is no different to rotating about the x-axis initially. In order to understand this consider the effect a y-roll has on the x-axis, about which we have already performed a rotation of amount Θ_1. Although, as you will remember, the rolls act on points in the space – we are not rotating the coordinate axes which remain fixed – we can still talk about the *effect* on the x-axis. This is because the rolls are applied in a fixed order and subsequent rolls have the *effect* of rotating in space the axes about which the preceding rolls have been applied. We track the effect a y-roll of $\pi/2$ has on the preceding x-roll by rotating the x-axis as if it were embedded in the object. Thus the effect of a y-roll of $\pi/2$ is to rotate the x-axis to x' (Figure 15.16), which is in alignment with the z-axis. Consequently any z-roll of Θ_3 could have been achieved by an x-roll of $-\Theta_3$. Effectively, now that we are in this configuration with the x- and z-axes aligned, it is impossible to rotate the object about the x-axis.

This sudden loss of a degree of freedom is extremely irritating to the animator. Mathematically, the animator has unwittingly stumbled upon a singularity in the parametrization, where Θ_1 and Θ_3 become associated with the same degree of freedom. To see the reason for this mathematically, we set $s_2 = 1$ and $c_2 = 0$ into the rotation matrix, reducing it to $R(\Theta_1, \pi/2, \Theta_3)$ given by:

$$\begin{bmatrix} 0 & 0 & -1 & 0 \\ \sin(\Theta_1 - \Theta_3) & \cos(\Theta_1 - \Theta_3) & 0 & 0 \\ \cos(\Theta_1 - \Theta_3) & -\sin(\Theta_1 - \Theta_3) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using different formulations of Euler angles in the general rotation matrix does not remove this singularity.

Euler angles and interpolation

We now consider the problem of interpolation when Euler angles are used. Suppose the three Euler angles are

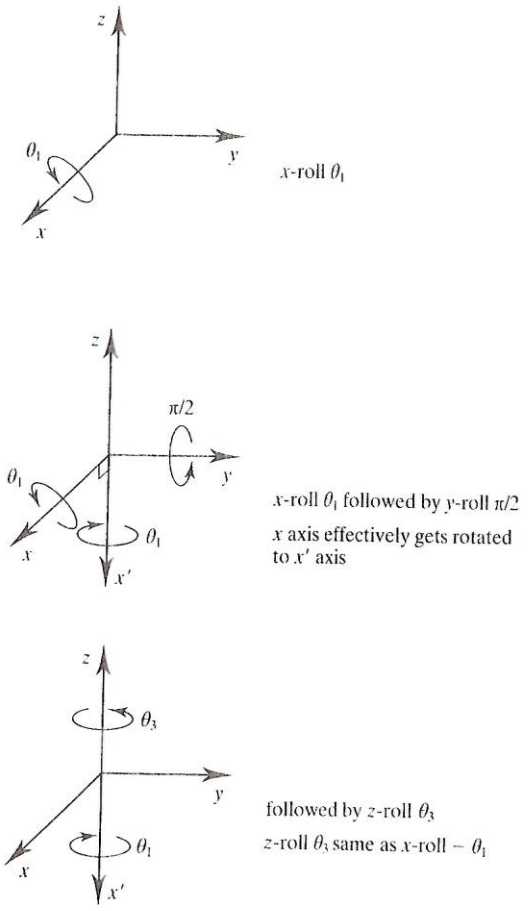

Figure 15.16 Illustrating the loss of one degree of freedom – gimbal lock.

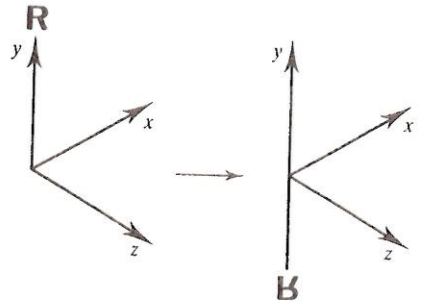

Figure 15.17 The start and finish positions for the animation of the block letter 'R'.

used as key parameters in an interpolating system. Suppose also that key orientation i is described by the triple $(\Theta_{1i}, \Theta_{2i}, \Theta_{3i})$ and interpolation is carried out by interpolating through the three Euler angles separately in a manner identical to interpolating translation, that is, by separately interpolating through the x, y and z components of key positions (x_i, y_i, z_i). This means that at a certain frame t, the interpolated values $(\Theta_1(t), \Theta_2(t), \Theta_3(t))$ are combined to produce the rotation matrix $R(\Theta_1(t), \Theta_2(t), \Theta_3(t))$ which is applied to the object. There is a problem with this approach, however. The hidden assumption behind such a scheme is that rotations act just like translations – but they do not. That this is so should be apparent from the fact that rotation involves multiplication, whereas translations only involve addition. Moreover, as is well known, rotation matrices do not commute in multiplication, whereas translation matrices do under addition.

Consideration of a specific example will reveal the inadequacy of the Euler angle parametrization more clearly. Let our object be a block letter 'R' and let it be initially offset from the origin along the y-axis by a nonzero amount. The final orientation is the reflection of the object in the x, z-plane as shown in Figure 15.17. The animator's task is to set up an animation that rotates the letter from the start to the final orientation. There is more than one way to achieve this movement. One way would be a single rotation of π about the x-axis (Figure 15.18(a)). An alternative is to first perform a y-roll of π followed by a z-roll of π (Figure 15.18(b)). Both routes reach the end position but get there in different ways. Generating inbetweens via linear interpolation to give rotation matrices for the intermediate frames gives us the sequence of rotation matrices:

$$R(0, 0, 0), \ldots, R(\pi t, 0, 0), \ldots, R(\pi, 0, 0) \quad t \in [0, 1]$$

for the first route, and:

$$R(0, 0, 0), \ldots, R(0, \pi t, \pi t), \ldots, R(0, \pi, \pi)$$

for the second route. The effect of these two sequences for our example is shown in Figures 15.19(a) and (b). Clearly, the two moves are very different; the first produces a simple steady rotation, whereas the second both rotates about the y-axis and simultaneously twists about the z-axis. In general, specifying orientation moves in this manner can easily produce such contorted movements, as the object is only allowed to twist about separate coordinate axes.

Although this example is somewhat contrived, it should be clear that it represents a dilemma for the animator. Depending on the choice of principal rotations there is more than one way to get from one key orientation to another. Compare this to interpolating translation between successive key positions in cartesian coordinates. The movement is always the same. If linear interpolation is employed, for example, the move is always along a straight line from one key position to the other. Using Euler angles for interpolation to get from

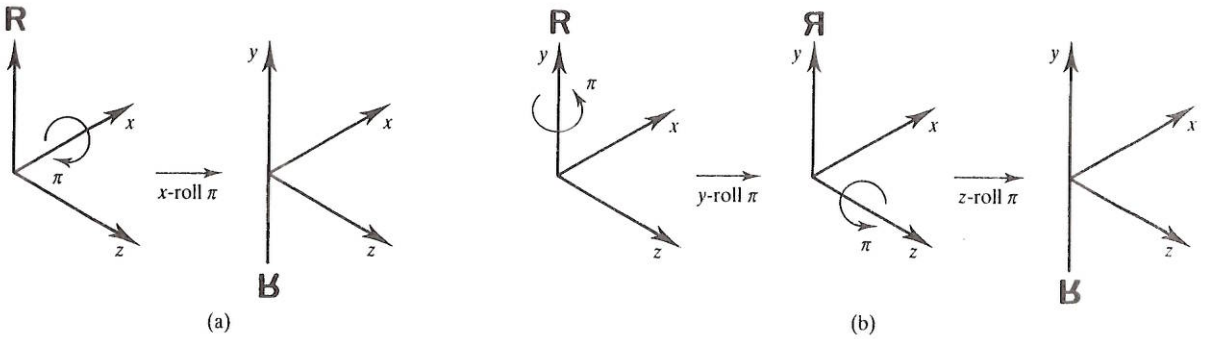

Figure 15.18 The two routes for the animation of the block letter 'R'.

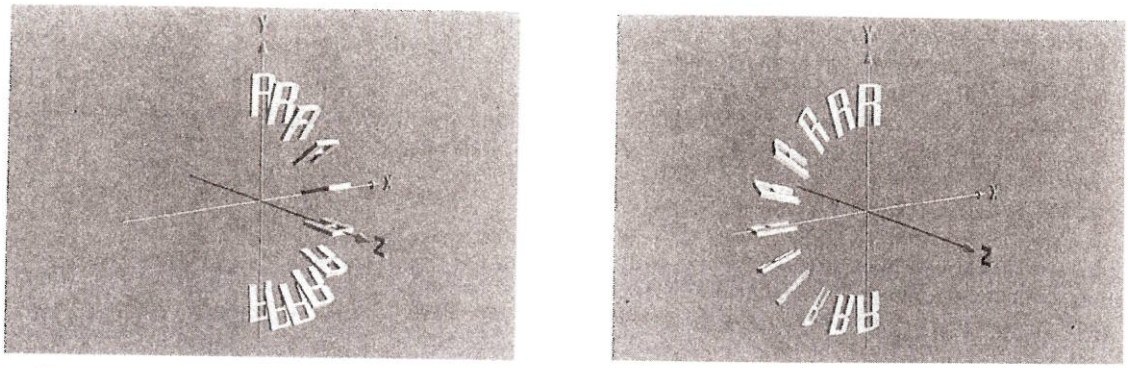

Figure 15.19 Euler angle parametrization. (a) A single x-roll of π. (b) A y-roll of π followed by a z-roll of π.

one orientation to another is not unique. Why is there this difference? The answer lies in the fact that the components of cartesian coordinates are truly independent of each other, whereas Euler angles are not. Interpolating Euler angles treats them as if they were independent of each other and completely ignores the effect they have on each other. The animator is forced into specifying orientation as a composition of rotations about three separate axes, the order of which must be strictly observed. Moreover, different coordinate systems will produce different moves through identical key orientations. We conclude that Euler angle interpolation produces a motion as inappropriate as that obtained by interpolating position through key positions specified in spherical polar coordinates as opposed to cartesian coordinates.

Euler's theorem tells us that it is possible to get from one orientation to any other by a simple steady rotation about a single axis. Interpolation between two key orientations should produce precisely this simple rotation. Euler rotation is inadequate because, as we have seen, given two successive rotations the notation does not provide close expressions to determine the angle and the axis of the resultant rotation. What we seek is a parametrization of orientation that can accommodate the interaction of rotations within its working, thereby enabling us to:

1. guarantee a simple steady rotation between any two key orientations, which we know must exist, and
2. define moves that are independent of the choice of the coordinate system.

Luckily such a parametrization exists, but in order to discuss it we need to introduce a notation implied by Euler's theorem – angular displacement.

Angular displacement

We define orientation as an angular displacement given by (Θ, n) of an amount about an axis n. Just as we did for Euler angle notation, we shall derive the rotational matrix in terms of this new notation, so instead of $R(\Theta_1, \Theta_2, \Theta_3)$ we write $R(\Theta, n)$. Consider the angular displacement acting on a vector r taking it to position Rr as shown in Figure 15.20.

The problem can be decomposed by resolving r into components parallel to n, $r_\parallel$, which by definition remains unchanged after rotation, and perpendicular to n, $r_\perp$ in the plane passing through r and Rr.

$$r_\parallel = (n \cdot r)n$$
$$r_\perp = r - (n \cdot r)n$$

$r_\perp$ is rotated into position $Rr_\perp$. We construct a vector perpendicular to $r_\perp$ and lying in the plane. In order to evaluate this rotation, we write:

$$V = n \times r_\perp = n \times r$$

So

$$Rr_\perp = (\cos\Theta)r_\perp + (\sin\Theta)V$$

hence

$$\begin{aligned} Rr &= Rr_\parallel + Rr_\perp \\ &= Rr_\parallel + (\cos\Theta)r_\perp + (\sin\Theta)V \\ &= (n \cdot r)n + \cos\Theta(r - (n \cdot r)n) + (\sin\Theta)n \times r \\ &= (\cos\Theta)r + (1 - \cos\Theta)n(n \cdot r) + (\sin\Theta)n \times r \end{aligned} \quad (15.5)$$

And now, we beg the reader's indulgence for an apparent *non sequitur*, the relevance of which will be revealed at the end of the digression.

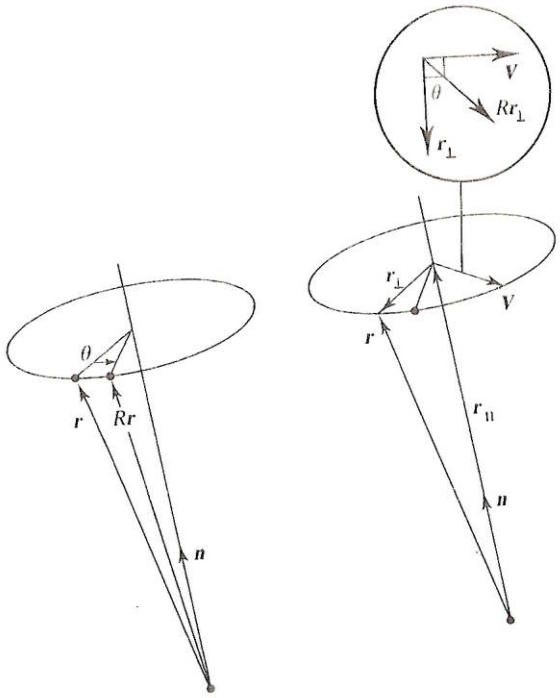

Figure 15.20 Angular displacement (Θ, n) of r.

Quaternions

The great mathematician Sir William Hamilton had been interested in complex numbers since the early 1830s. Complex numbers have the form:

$$a + ib$$

where a and b are real and the multiplication rules are:

$$1^2 = 1 \quad \text{and} \quad i^2 = -1$$

These complex numbers define a plane – the complex plane – where one axis is real and the other imaginary. For over 10 years Hamilton tried to extend this concept in order to define a complex volume by searching for a second imaginary axis. Just such a number would have three components: one real and two imaginary. This, however, he could not do. Then, on 16 October 1843, when walking past Broome Bridge in Dublin towards the Royal Irish Academy, where he was to preside over a meeting, Hamilton, in a flash of inspiration, realized that three rather than two imaginary units were needed, with the following properties:

$$i^2 = j^2 = k^2 = -1$$
$$ij = k \quad \text{and} \quad ji = -k$$

with the cyclic permutation $i \to j \to k \to i$. Such was his elation, Hamilton carved these formulae on the side of the bridge and called the number:

$$q = a + bi + cj + dk$$

a 'quaternion'.

For our purposes we shall use the condensed notation:

$$q = (s, v)$$

where:

$$(s, v) = s + v_x i + v_y j + v_z k$$

s is thought of as the scalar part of the quaternion and v the vector part with axes i, j and k. Using the above rules it is easy to derive the following properties. The multiplication of two quaternions:

$$q_1 = (s_1, v_1) \quad \text{and} \quad q_2 = (s_2, v_2)$$

is given by:

$$q_1 q_2 = (s_1 s_2 - v_1 \cdot v_2,\ s_1 v_2 + s_2 v_1 + v_1 \times v_2)$$

The multiplication of two quaternions is thus a quaternion. Mathematically, we have defined a group. Stated somewhat simplistically, a group is just a set of elements with a rule defining their multiplication such that the result of this multiplication is itself an element of that group. Groups can be constructed completely arbitrarily, though a surprising number of groups are relevant to the physical world. We shall see that a subgroup of the quaternion group is closely related to the group of rotations or, more precisely, the group of rotation matrices.

Note that except for the cross product term at the end of the previous equation, it bears a strong similarity to the law of complex multiplication:

$$(a_1 + ib_1)\ (a_2 + ib_2) = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + a_2 b_1)$$

The cross product term has the effect of making quaternion multiplication noncommutative.

We define the conjugate of the quaternion:

$$q = (s,\ v) \text{ to be } \bar{q} = (s,\ -v)$$

The product of the quaternion with its conjugate defines its magnitude:

$$q\bar{q} = s^2 + |v|^2 = |q|^2$$

Finally, as promised, we come to the point of all this, which is contained in the following properties. Take a pure quaternion (one that has no scalar part):

$p = (0, r)$

and a unit quaternion

$q = (s, v)$ where $q\bar{q} = 1$

and define

$R_q(p) = qpq^{-1}$

Using our multiplication rule, and the fact that $q^{-1} = \bar{q}$ for q of unit magnitude, this expands to:

$$R_q(p) = (0, (s^2 - v \cdot v)r + 2v(v \cdot r) + 2sv \times r) \qquad (15.6)$$

This can be simplified further since q is of unit magnitude and we can write:

$q = (\cos\theta, \sin\theta n)$ $\quad |n| = 1$

Substituting into Equation (15.6) gives:

$$\begin{aligned} R_q(p) &= (0, (\cos^2\theta - \sin^2\theta)r + 2\sin^2\theta\, n(n \cdot r) \\ &\quad + 2\cos\theta \sin\theta\, n \times r) \\ &= (0, \cos 2\theta\, r + (1 - \cos 2\theta)\, n(n \cdot r) \\ &\quad + \sin 2\theta\, n \times r) \end{aligned} \qquad (15.7)$$

Now compare this with Equation (15.5). You will notice that aside from a factor of 2 appearing in the angle they are identical in form. What can we conclude from this? The act of rotating a vector r by an angular displacement (θ, n) is the same as taking this angular displacement, 'lifting' it into quaternion space, by representing it as the unit quaternion $(\cos(\theta/2), \sin(\theta/2)\, n)$ and performing the operation $q\,(\,)\,\bar{q}$ on the quaternion $(0,r)$. We could therefore parametrize orientation in terms of the four parameters:

$\cos(\theta/2)$, $\sin(\theta/2)\, n_x$, $\sin(\theta/2)\, n_y$, $\sin(\theta/2)\, n_z$

using quaternion algebra to manipulate the components.

In practice this would seem an extremely perverse way of going about things were it not for one very important advantage afforded by the quaternion parametrization. Two quaternions multiplied together, each of unit magnitude, will result in a single quaternion of unit magnitude. If we use unit quaternions to represent rotations then this translates to two successive rotations producing a single rotation. Now a variation of Euler's theorem states that two successive rotations is equivalent to one rotation. So we can see that inherent in the algebra of the quaternion group is Euler's theorem. The single steady rotation between successive keyframes that we seek is provided for us automatically by the rules particular to the parametrization and contained in the statement:

$R_{q''} = R_q R_{q'}$ where $q'' = qq'$

Let us now return to our example of Figure 15.17 to see how this works in practice. The first single x-roll of π is represented by the quaternion:

$(\cos(\pi/2), \sin(\pi/2)\ (1,0,0)) = (0, (1,0,0))$

Similarly, a y-roll of π and a z-roll of π are given by $(0,(0,1,0))$ and $(0,(0,0,1))$ respectively. Now the effect of a y-roll of π followed by a z-roll of π can be represented by the single quaternion formed by multiplying these two quaternions together:

$$\begin{aligned} (0,(0,1,0))\ (0,(0,0,1)) &= (0,(0,1,0) \times (0,0,1)) \\ &= (0,(1,0,0)) \end{aligned}$$

which is the single x-roll of π. From this we can see that the cross product term in (15.7) can be thought of as correcting for the interdependence of the separate axes that is ignored by Euler's angle notation.

An additional advantage afforded by using quaternions is that the gimbal lock singularity, which is a consequence of using three parameters to parametrize orientation, disappears.

Much of what now follows is based on the work of the researcher who brought quaternions to the attention of the computer graphics community. The interested reader is referred to [SHOE85] and [SHOE87] for further detail. The latter reference concerns itself more with the practical details of an implementation.

Interpolating using quaternions

Given the superiority of quaternion parametrization over Euler angle parametrization, this section covers the issue of interpolating rotation in quaternion space. Consider an animator sitting at a workstation and interactively setting up a sequence of key orientations by whatever method is appropriate. This is usually done with the principal rotation operations, but now the restrictions that were placed on the animator when using Euler angles, namely using a fixed number of principal rotations in a fixed order for each key, can be removed. In general, each key will be represented as a single rotation matrix. This sequence of matrices will then be converted into a sequence of quaternions. Interpolation between key quaternions is performed and this produces a sequence of inbetween quaternions, which are then converted back into rotation matrices. The matrices are then applied to the object. The fact that a quaternion interpolation is being used is transparent to the animator.

Moving into and out of quaternion space

The implementation of such a scheme requires us to move into and out of quaternion space, that is, to go from a general rotation matrix to a quaternion and vice versa. It can be shown that the effect of taking a unit quaternion:

$$q = (\cos(\theta/2), \sin(\theta/2)\, n)$$

and performing the operation $q(\ )q^{-1}$ on a vector is the same as applying the following rotation matrix to that vector:

$$\begin{bmatrix} 1-2Y^2-2Z^2 & 2XY-2WZ & 2XZ+2WY & 0 \\ 2XY+2WZ & 1-2X^2-2Z^2 & 2YZ-2WX & 0 \\ 2XZ-2WY & 2YZ+2WX & 1-2X^2-2Y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the quaternion $(\cos(\theta/2), \sin(\theta/2)\, n)$ is written as (W, (X,Y,Z)), the notation used in Listing 15.3. By these means then, we can move from quaternion space to rotation matrices. Listing 15.3 gives the conversion from quaternion space to rotation matrix in the routine *quattomat(q, mat)*.

The inverse mapping from a rotation matrix to a quaternion is only slightly more involved. All that is required is to convert a general rotation matrix:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

into the matrix format directly above. The resulting quaternion is trivially $(W, (X, Y, Z))$. Given a general rotation matrix the first thing to do is to examine the sum of its diagonal components M_{ii} where $0 \leqslant i \leqslant 3$. This is called the trace of the matrix. From the above format we know:

$$\begin{aligned} \text{trace} &= 1-2Y^2-2Z^2+1-2X^2-2Z^2+1-2X^2-2Y^2+1 \\ &= 4-4(X^2+Y^2+Z^2) \end{aligned}$$

Since the matrix represents a rotation we know that the corresponding quaternion must be of unit magnitude, that is:

$$X^2+Y^2+Z^2+W^2=1$$

and so the trace reduces to $4W^2$. Thus for a 4×4 homogeneous matrix we have:

$$W = (\text{trace})^{1/2}$$

The remaining components of the quaternion (X, Y, Z) which, as you will recall, is the axis of rotation scaled by half the sine of the angle of rotation, are obtained by combining diagonally opposite elements of the matrix M_{ij} and M_{ji} where $0 \leqslant i,j \leqslant 2$. We have:

$$X = \frac{M_{21}-M_{12}}{4W} \qquad Y = \frac{M_{02}-M_{20}}{4W} \qquad Z = \frac{M_{10}-M_{01}}{4W}$$

For zero W these equations are undefined and so other combinations of the matrix components, along with the fact that the quaternion is of unit magnitude, are used to determine the axis of rotation. Listing 15.3 gives the code in full for moving from rotation matrices to quaternions in the routine *mattoquat(mat,q)*.

Having outlined our scheme we now discuss how to interpolate in quaternion space. Since a rotation maps onto a quaternion of unit magnitude, the entire group of rotations maps onto the surface of the four-dimensional unit hypersphere in quaternion space. Curves interpolating through key orientations should therefore lie on the surface of this sphere. Consider the simplest case of interpolating between just two key quaternions. A naive, straightforward, linear interpolation between the two keys results in a motion that speeds up in the middle. This is because we are not moving along the surface of the hypersphere but cutting across it. In order to ensure a steady rotation we must employ spherical linear interpolation, where we move along an arc of the geodesic that passes through the two keys. (Figure 15.1 showing the differences between interpolating position and interpolating rotation angle is entirely analogous to this situation.) Technically, the metric of the hypersphere's surface is said to be the same as the angular metric of the rotation group.

The formula for spherical linear interpolation is easy to derive geometrically. Consider the two-dimensional case of two vectors A and B separated by angle Ω and vector P which makes an angle θ with A as shown in Figure 15.21. P is derived from spherical interpolation between A and B and we write:

$$P = \alpha A + \beta B$$

Trivially, we can solve for α and β given:

$$|P| = 1$$
$$A \cdot B = \cos\Omega$$
$$A \cdot P = \cos\theta$$

to give:

$$P = A\frac{\sin(\Omega-\theta)}{\sin\Omega} + B\frac{\sin\theta}{\sin\Omega}$$

Listing 15.3 quatlib.c

```
#define X 0
#define Y 1
#define Z 2
#define W 3
#define EPSILON 0.00001
#define HALFPI 1.570796326794895

        int nxt[3] = {Y,Z,X};

void quattomat(q,mat)
float q[4];
float mat[4][4];
{
        double s,xs,ys,zs,wx,wy,wz,xx,xy,xz,yy,yz,zz;

        s = 2.0/(q[X]*q[X] + q[Y]*q[Y] + q[Z]*q[Z] + q[W]*q[W]);

        xs = q[X]*s; ys = q[Y]*s; zs = q[Z]*s;
        wx = q[W]*xs; wy = q[W]*ys; wz = q[W]*zs;
        xx = q[X]*xs; xy = q[X]*ys; xz = q[X]*zs;
        yy = q[Y]*ys; yz = q[Y]*zs; zz = q[Z]*zs;

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

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

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

        mat[0][3] = 0.; mat[1][3] = 0.; mat[2][3] = 0.; mat[3][3] = 1.;
        mat[3][0] = 0.; mat[3][1] = 0.; mat[3][2] = 0.;

}
void mattoquat(mat,q)
float mat[4][4];
float q[4];
{

        double tr,s;
        int i,j,k;

        tr = mat[0][0] + mat[1][1] + mat[2][2];
        if (tr > 0.0) {

                s = sqrt(tr + 1.0);
                q[W] = s*0.5;
                s = 0.5/s;

                q[X] = (mat[1][2] - mat[2][1])*s;
                q[Y] = (mat[2][0] - mat[0][2])*s;
                q[Z] = (mat[0][1] - mat[1][0])*s;
        }
```

Listing 15.3 (*cont.*)

```
        else {

                i = X;
                if (mat[Y][Y] > mat[X][X]) i = Y;
                if (mat[Z][Z] > mat[i][i]) i = Z;
                j = nxt[i] ; k = nxt[j];

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

                q[i] = s*0.5;
                s = 0.5/s;
                q[W] = (mat[j][k] - mat[k][j])*s;
                q[j] = (mat[i][j] + mat[j][i])*s;
                q[k] = (mat[i][k] + mat[k][i])*s;

        }

}
void slerp(p,q,t,qt)
float p[4],q[4];
float t;
float qt[4];
{
        double omega,cosom,sinom,sclp,sclq;
        int i;

        cosom = p[X]*q[X] + p[Y]*q[Y] + p[Z]*q[Z] + p[W]*q[W];

        if ( (1.0 + cosom) > EPSILON ) {

                if ( (1.0 - cosom) > EPSILON ) {

                        omega = acos(cosom);
                        sinom = sin(omega);
                        sclp  = sin( (1.0 - t)*omega )/sinom;
                        sclq  = sin( t*omega )/sinom;

                }
                else {

                        sclp = 1.0 - t;
                        sclq = t;

                }
                for (i=0;i<4;i++) qt[i] = sclp*p[i] + sclq*q[i];

        }
        else {

                qt[X] = -p[Y]; qt[Y] = p[X];
                qt[Z] = -p[W]; qt[W] = p[Z];
                sclp = sin((1.0 - t)*HALFPI);
                sclq = sin(t*HALFPI);
                for (i = 0; i< 3; i++) qt[i] = sclp*p[i] + sclq*qt[i];

        }
}
```

Spherical linear interpolation between two unit quaternions q_1 and q_2, where:

$$q_1 \cdot q_2 = \cos \Omega$$

is obtained by generalizing the above to four dimensions and replacing Θ by Ωu where $u \in [0,1]$. We write:

$$\text{slerp}(q_1, q_2, u) = q_1 \frac{\sin(1-u)\,\Omega}{\sin \Omega} + q_2 \frac{\sin \Omega u}{\sin \Omega}$$

Listing 15.3 gives a code fragment for this. *slerp (p,q,t,qt)* returns the interpolated quaternion qt, for t between p and q. The routine caters for the special cases where the keys are very close together, in which case we approximate using the more economical linear interpolation and avoid divisions by very small numbers since

$$\sin \Omega \to 0 \qquad \text{as } \Omega \to 0$$

The case where p and q are diametrically opposite, or nearly so, also requires special attention.

Now, given any two key quaternions, p and q, there exist two possible arcs along which one can move, corresponding to alternative starting directions on the geodesic that connects them. One of them goes around the long way and this is the one that we wish to avoid. Naively, one might assume that this reduces to either spherically interpolating between p and q by the angle Ω, where:

$$p \cdot q = \cos \Omega$$

or interpolating in the opposite direction by the angle $2\pi - \Omega$. This, however, will not produce the desired effect. The reason is that the topology of the hypersphere of orientation is not just a straightforward extension of the three-dimensional Euclidean sphere. To appreciate this, it is sufficient to consider the fact that every rotation has two representations in quaternion space, namely q and $-q$, that is, the effect of q and $-q$ is the same. That this is so is because algebraically the operator $q(\,)q^{-1}$ has exactly the same effect as $(-q)\,(\,)\,(-q)^{-1}$. Thus, points diametrically opposed represent the same rotation. Because of this topological oddity care must be taken when determining the shorter arc. A strategy that works is to choose interpolating between either the quaternion pairs p and q or p and $-q$. Given two key orientations p and q find the magnitude of their difference, that is $(p-q)\cdot(p-q)$, and compare this to the magnitude of the difference when the second key is negated, that is $(p+q)\cdot(p+q)$. If the former is smaller then we are already moving along the smaller arc and nothing needs to be done. If, however, the second is smaller, then we replace q by $-q$ and proceed. These considerations are shown schematically in Figure 15.22.

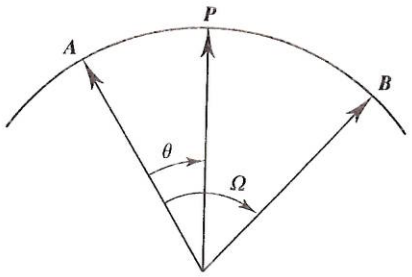

Figure 15.21 Spherical linear interpolation.

So far we have described the spherical equivalent of linear interpolation between two key orientations, and, just as was the case for linear interpolation, spherical linear interpolation between more than two key orientations will produce jerky, sharply changing motion across the keys. What is required for higher order continuity is the spherical equivalent of the cubic spline. Unfortunately, because we are now working on the surface of a four-dimensional hypersphere, the problem is far more complex than constructing splines in three-dimensional Euclidean space. [DUFF86] and [SHOE87] have tackled this problem. We shall describe the approach made in [SHOE87] since it pays greatest lip service to implementation points.

The following construction enables us to think of a cubic spline as a series of three linear interpolations. By extension [SHOE87] takes three spherical linear interpolations and defines a cubic spline on the surface of a sphere. Consider four points (S_0, S_1, S_2, S_3) at the corners of the rectangle shown in Figure 15.23. We linearly interpolate by an amount $u \in [0,1]$, along the horizontal edges to get the intermediate points S_α, S_β, where:

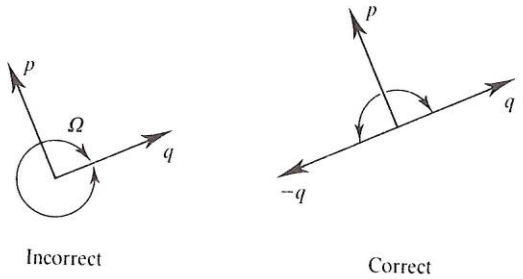

Figure 15.22 Shortest arc determination on quaternion hypersphere.

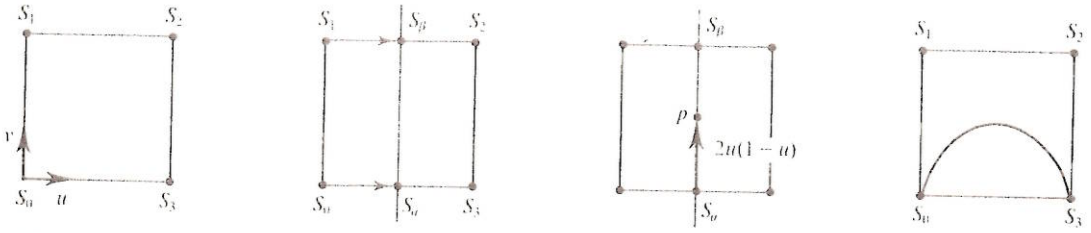

Figure 15.23 The quadrangle construction for a parabola.

$$S_\alpha = S_0(1-u) + S_3 u$$
$$S_\beta = S_1(1-u) + S_2 u$$

Now we perform a vertical linear interpolation by an amount

$$v = 2u(1-u)$$

to get the point

$$p = S_\alpha(1-v) + S_\beta v$$

As u varies from 0 to 1, the locus of p will trace out a parabola. This process of bilinear interpolation, where the second interpolation is thus restricted, is called 'parabolic blending'. Böhm [BÖHM82] shows how, given a Bézier curve segment (b_0, b_1, b_2, b_3) one can derive the quadrangle points (b_0, S_1, S_2, b_3) of the above construction. This has the geometric significance of enabling us to visualize the cubic as a parabola whose quadrangle points are not necessarily parallel or coplanar. The cubic can be thought of as a warped parabola as shown in Figure 15.24.

The mathematical significance of this construction is that it shows how to construct a cubic as a series of three linear interpolations of the quadrangle points. [SHOE87] takes this construction onto the surface of the four-dimensional hypersphere by constructing a spherical curve, using three spherical linear interpolations of a quadrangle of unit quaternions. This he defines as squad( ), where:

$$\text{squad}(b_0, S_1, S_2, b_3, ut) = \text{slerp}(\text{slerp}(b_0, b_3, ut), \text{slerp}(S_1, S_2, ut), 2u(1-u))$$

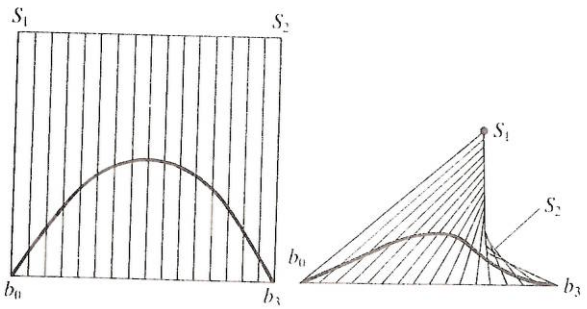

Figure 15.24 A warped parabola is a cubic.

Given a series of quaternion keys one can construct a cubic segment across keys q_i and q_{i+1} by constructing a quadrangle of quaternions $(q_i, a_i, b_{i+1}, q_{i+1})$ where a_i, b_{i+1} have to be determined. These inner quadrangle points are chosen in such a way to ensure that continuity of tangents across adjacent cubic segments is guaranteed. The derivation for the inner quadrangle points is difficult, involving as it does the calculus and exponentiation of quaternions and we will just quote the results, referring the interested reader to [SHOE87]:

$$a_i = b_i = q_i \exp\left(-\frac{\ln(q_i^{-1}q_{i+1}) + \ln(q_i^{-1}q_{i-1})}{4}\right)$$

where, for the unit quaternion:

$$q = (\cos\Theta, \sin\Theta\, v)$$
$$|v| = 1$$
$$\ln(q) = (0, \Theta v)$$

and, inversely for the pure quaternion (zero scalar part):

$$q = (0, \Theta v)$$
$$\exp(q) = (\cos\Theta, \sin\Theta v)$$

Finally, in order to illustrate the principles underlying this discussion on the parametrization of orientation, we return to our block letter 'R' and apply the various interpolation techniques that we have discussed. Because all orientations take place about a fixed point, R effectively moves over the surface of a sphere (Figure 15.25).

Using the principal rotations of an x-roll followed by a y-roll followed by a z-roll, the animator sets up an orientation key represented by the rotation matrix $R(\Theta_{1i}, \Theta_{2i}, \Theta_{3i})$. This rotation matrix is then lifted into quaternion space to give the quaternion key q_i. The animator specifies three such keys. As we have discussed, interpolation to generate the inbetweens can be carried out by:

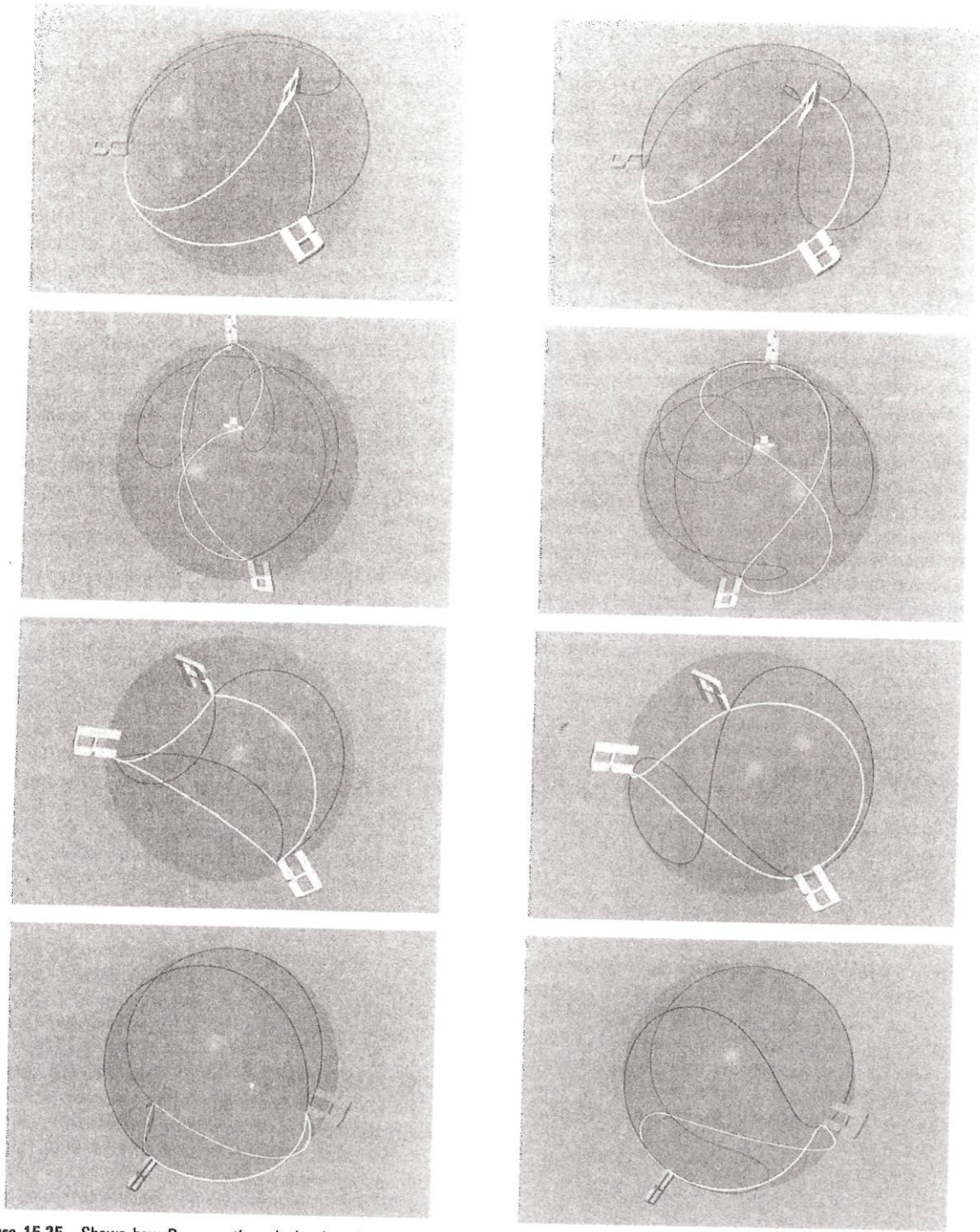

Figure 15.25 Shows how R moves through the three keys. In all cases the white line tracks the motion of R when the interpolation is carried out in quaternion space; the black line tracks the motion of R when Euler angles are interpolated. In each row the left illustration compares linear interpolation of Euler angles with spherical linear interpolation of quaternions. In each row the right illustration compares a cubic spline interpolation of Euler angles to the spherical cubic spline interpolation of quaternions (using squad( )).

1. Using the Euler angle keys $(\Theta_{1i}, \Theta_{2i}, \Theta_{3i})$, $i = 0,1,2$, to produce an interpolated Euler angle $(\Theta_1(t), \Theta_2(t), \Theta_3(t))$ at time t which is used to generate the rotation matrix $R(\Theta_1(t), \Theta_2(t), \Theta_3(t))$ for that frame.
2. Moving into quaternion space and using the quaternion keys q_i, $i = 0,1,2$, to produce an interpolated quaternion key $q(t)$ at time t, which is then converted to a rotation matrix $R_q(t)$ for that frame.

In all cases periodic interpolation modulo 3 is employed to give a closed loop. Figure 15.25 shows how R moves through the three keys. In all cases the white line tracks the motion of R when the interpolation is carried out in quaternion space; the black line tracks the motion of R when Euler angles are interpolated. In each row the left illustration compares linear interpolation of Euler angles with spherical linear interpolation of quaternions. In each row the right illustration compares a cubic spline interpolation of Euler angles to the spherical cubic spline interpolation of quaternions (using squad( )). In both columns quaternions come out better than Euler angles, the motions being far more direct and less convoluted. Note that if we change the coordinate axes and regenerate the image, the black lines will change shape whereas the white lines will stay constant. Going down the rows, the initial set of key orientations is varied in each case.

Finally, we mention a potential difficulty when applying quaternions. Quaternion interpolation is indiscriminate in that it does not prefer any one direction to any other. Interpolating between two keys produces a move that depends on the orientations of the keys and nothing else. This is inconvenient when choreographing the virtual camera. Normally when moving a camera the film plane is always required to be upright – this is usually specified by an 'up' vector. By its very nature, the notion of a preferred direction cannot easily be built into the quaternion representation. Thus the advantages afforded by quaternions as applied to objects cannot be exploited when setting up camera moves. In fact, specification of an up vector is problematic whatever way we choose to parametrize orientation.