The Visual Computer

Quaternion calculus as a basic tool in computer graphics

Daniel Pletinckx

R & D Department,
Barco-Industries Creative Systems,
Th. Sevenslaan 106, B-8500 Kortrijk, Belgium

Quaternions, although not well known, provide a fundamental and solid base to describe the orientation of an object or a vector. They are efficient and well suited to solve rotation and orientation problems in computer graphics and animation. This paper describes two new methods for splining quaternions so that they can be used within a keyframe animation system. We also show that quaternions, although up to now solely used for animation purposes, can be used successfully in the field of modelling and rendering and we prove that we can speed up the rendering algorithm by using quaternions.

Key words: Quaternions – Spline subdivision – Animation – Propagation control graphs – Phong shading

Direction and orientation are two important notions in computer graphics. One property although makes them somewhat difficult to handle: there is no good way to describe them in Cartesian coordinates. You can experience this in the following way: if you hold your arm extended and point in different directions, you will find out that your fingertip moves on the surface of a three-dimensional sphere with your arm as radius. So directions in a 3D space are organised like a sphere, and there is no way to depict the surface of a sphere in Cartesian coordinates without distorsion (every map of the Earth demonstrates this). Orientations have one more degree of freedom: while pointing in some direction, you can still rotate your wrist. As this additional degree of freedom is again a rotation, orientation is organised like a 4-dimensional sphere, which equally cannot be represented properly in Cartesian coordinates.

Computer graphics people are somewhat sloppy with regard to directions (normal vectors for instance) and orientations. Most people treat them in a Cartesian way but overlook the consequences. Normal vectors are interpolated linearly in Cartesian space: the price to pay for this is a time consuming renormalisation to get a unit vector again. Orientations are obtained through rotations about 3 axes, fixed to the object, but this doesn't work well because these rotations are treated as if they behave like Cartesian coordinates. Most computer programs, which deal with directions and orientations, are spangled with cosines and sines and other trigonometric stuff, which make the code slow and complex.

Quaternions were discovered by Sir William Rowan Hamilton in October of 1843, while trying to extend the complex plane to the 3-dimensional space. This act is well recorded as Hamilton considered quaternions to be the most important discovery of his entire career. After failing for 14 years to find a way to multiply triples (vectors) so that the norm was preserved, Hamilton suddenly realised that quadruples would work (see Hankins 1980). We know by now that, by an odd quirk of mathematics, only systems of 2, 4 or 8 components are norm preserving. Soon after their discovery in February 1845, A. Cayley showed that quaternions can represent orientation. James Clerk Maxwell, in his classic "Treatise on Electricity and Magnetism" used quaternion differentials. Dot and cross products of vectors were discovered as part of the quaternion product.

The Visual Computer (1989) 5:2–13

1 Origin and properties of quaternions

As may seem from the introduction, quaternions are surprisingly old and surprisingly unknown. About a hundred years ago, they were widely spread and frequently used, but for some mysterious reason, they disappeared in history although they were already rediscovered some years ago in the world of aerospace engineering and robotics. Some universities start teaching and exploring them anew, because of their powerful and fundamental properties.

Hamilton defined a quaternion q in the following way:

$$q = w + ix + jy + kz \tag{1}$$

where i, j, k are the three imaginary units. This definition looks quite similar to the definition of a vector as a complex number

$$\bar{v} = x + iy \tag{2}$$

in the complex plane. So a quaternion can be seen as the complex representation of a point (w, x, y, z) in some 4-dimensional space, which is governed by the following fundamental rules:

$$i^2 = j^2 = k^2 = -1 \tag{3}$$

$$jk = -kj = i \tag{4}$$

$$ki = -ik = j \tag{5}$$

$$ij = -ji = k \tag{6}$$

A lot of interesting things can be said about the mathematical properties of quaternions as defined by Eq. (1) (see for instance Shoemake 1987) but, as we want to deal with quaternions as a representation of orientation, we will narrow our scope to the subclass of quaternions of unit length. Explaining the reason for this would take us too far in mathematics, but think of the following analogy: trigonometry (which has to do with directions and angles in a 2-dimensional space) is also based on unit vectors, like the one in Eq. (2). All these unit vectors lie on a circle, and when multiplied with each other, they can be regarded as rotations in a plane.

Let's approach the quaternion world from an animator's point of view. Positioning a rigid body in 3D space, as we all know, can be done with 6 degrees of freedom: some fixed point of the object (for instance the center of gravity) can be placed anywhere in space with a translation, while every orientation of the object can be achieved by one rotation about that fixed point (Euler proved this in 1752). When rotating the object, one can observe that one line of points remains fixed, namely the "rotation axis" (which passes through the fixed point).

So one way to describe orientation is a rotation about an axis $\bar{a} = (X, Y, Z)$ over a certain angle 2θ (Fig. 1). The unit quaternion Q which represents this rotation is

$$Q = (w, x, y, z) = (\cos\theta, X \sin\theta, Y \sin\theta, Z \sin\theta) \tag{7}$$

or written in some handier way

$$Q = (\cos\theta, \sin\theta(X, Y, Z)) \tag{8}$$

with

$$\|Q\|^2 = w^2 + x^2 + y^2 + z^2 = 1 \tag{9}$$

The axis $\bar{a}$ is represented by a normalised vector, i.e.

$$\|\bar{a}\|^2 = X^2 + Y^2 + Z^2 = 1 \tag{10}$$

In fact, another way to write a unit quaternion is

$$Q = (w, \bar{v}) = (\cos\theta, \bar{a} \sin\theta) \tag{11}$$

where $\bar{v}$ is a scaled version of $\bar{a}$ and holds the complex part of Eq. (1).

Note that a unit quaternion only has three degrees of freedom (because of its normalisation). Note also from Eq. (10) that all possible rotation axes $\bar{a}$ occupy a sphere in the 3-dimensional space.

One can show (see Shoemake 1985) that there is a great deal of analogy between unit quaternions and rotation matrices (which are orthonormal matrices). Both form a non-communicative group under their multiplication. Both are easily converted one into the other (see Shoemake 1985, 1987). Although similar, composing (multiplying) quaternions is much more efficient than composing rotation matrices. This comes from the fact that quaternions are a non-redundant way to describe rotations (4 floating point numbers), while rotation matrices are redundant (9 floating point numbers). As unit quaternions form a group under multiplication, the product of two unit quaternions remains a unit quaternion, i.e. the multiplication of quaternions is norm preserving (as intended by Hamilton). A unit quaternion describes the rotation an object should make from its initial orientation to its new orientation. When a second quaternion

is applied on this last position, the object rotates towards another new orientation. The overall quaternion, being product of the two quaternions, depicts the rotation from the initial orientation to the last orientation (see Fig. 2).

Figure 2 shows us very clearly that rotations do not behave like translations do: the rotation axis of the composite rotation $Q_1 \cdot Q_2$ is not a linear composition of the rotation axes of Q_1 and Q_2. This phenomenon stems from the spherical nature (see Eq. (10)) of orientation and is called "confounding of the axes". It is easy to derive the quaternion product from (1) and (3) through (6):

$$\begin{aligned} q_1 \cdot q_2 &= (w_1 + i x_1 + j y_1 + k z_1) \\ &\quad \cdot (w_2 + i x_2 + j y_2 + k z_2) \\ &= w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \\ &\quad + i(x_1 w_2 + w_1 x_2 - z_1 y_2 + y_1 z_2) \\ &\quad + j(y_1 w_2 + z_1 x_2 + w_1 y_2 - x_1 z_2) \\ &\quad + k(z_1 w_2 - y_1 x_2 + x_1 y_2 + w_1 z_2) \end{aligned} \tag{12}$$

If we use the vector notation of Eq. (11), we can write this product in the following way:

$$\begin{aligned} q_1 \cdot q_2 &= (w_1, \bar{v}_1) \cdot (w_2, \bar{v}_2) \\ &= [(w_1 w_2 - \bar{v}_1 \cdot \bar{v}_2), \\ &\quad (w_1 \bar{v}_2 + w_2 \bar{v}_1 + \bar{v}_1 \times \bar{v}_2)] \end{aligned} \tag{13}$$

This is the very origin of the dot and cross product, where the latter is responsible for the confounding of the axes. It is useful to try Eq. (13) on the example shown in Fig. 2.

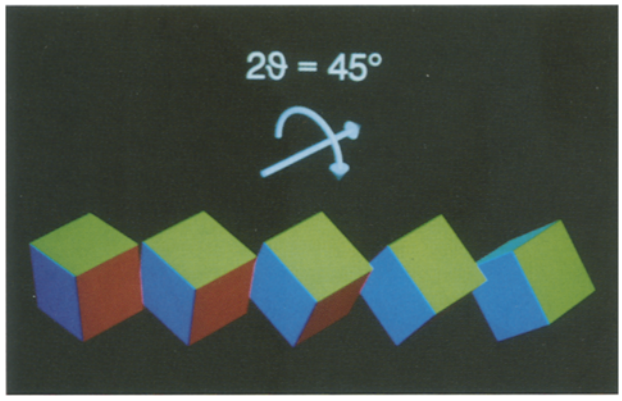
Normalised vectors $\bar{n}$ are a subclass of the unit quaternion space. They occupy the great circle on the hypersphere where there is no rotational part (see Blake 1987), i.e. $\cos\theta = 0$ and $\sin\theta = 1$:

$$\bar{n} = (0, (X, Y, Z)) \tag{14}$$

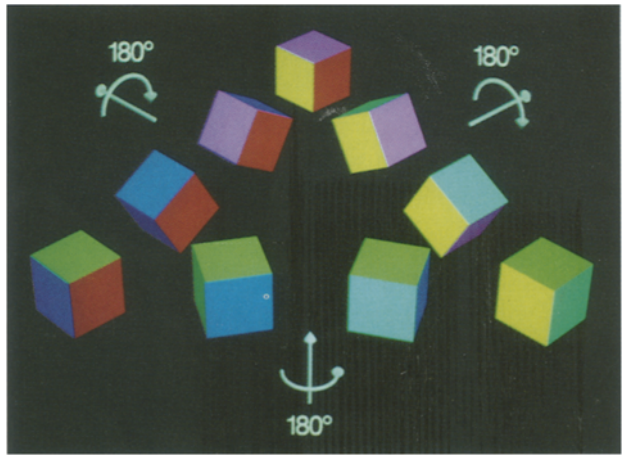
2 Quaternions versus Euler angles

Nearly every computer animation system nowadays uses Euler angles. And the users are bothered with it! One has to apply the Euler rotations in a particular order (because rotations do not commute like translations do!). Changing the order alters the whole animation (and there are about a dozen ways to specify Euler rotations). Euler angle animation also suffers from non-uniformity (a fixed change in Euler angles does not always give the same amount of rotation change). This means that Euler angles do not describe orientation in a rotation independent way. Euler angles also suffer from "gimbal lock": at certain orientations, you loose a rotational degree of freedom. For instance, if you rotate your camera 90 degrees to the left or the right, it's possible that you can't look up any more. If you have hierarchical motion in your animation system, you can solve this problem by putting your camera on a chariot: the chariot does the rotation about the vertical axis while the camera looks up and down. Remark that this works well because the composition of the two rotations is done through a rotation matrix.

In their defense, it must be said that it is easy to define an interactive user interface for Euler angles, because you need to change only one parameter at the time (and they are apt to solve differential equations – which is what Euler designed them



1



2

Fig. 1. Quaternion acting upon a cube

Fig. 2. Composition of quaternions (Q_1: upper left, Q_2: upper right)

for). But, we're lucky: Euler angles are easily converted to quaternions.
Quaternions on the other hand do not suffer from singularities, like gimbal lock, and provide a uniform and rotation independent description of all possible orientations. This is the major reason why quaternions are used in the robotics and aerospace field. An interactive user interface for specifying unit quaternions is certainly not obvious. Maybe a joystick (for the rotation axis) plus a dial (for the rotation angle) will do.
Equation (9) shows us that all unit quaternions occupy the surface of a sphere in the 4-dimensional space. It is easy to see that each possible orientation in 3D space is represented twice on that hypersphere, namely by (w, x, y, z) and $(-w, -x, -y, -z)$. In other words, a rotation about the opposite axis over the opposite angle gives the same orientation. Unit quaternions with w negative in fact incorporate rotation over a rotation angle

$$\theta' = \theta + n\pi; -\pi \leq \theta \leq \pi;$$
$$n \in \{\ldots, -10, -6, -2, 2, 6, 10, \ldots\} \quad (15)$$

while a positive w represents rotation over

$$\theta' = \theta + m\pi; -\pi \leq \theta \leq \pi;$$
$$m \in \{\ldots, -8, -4, 0, 4, 8, \ldots\} \quad (16)$$

This means that quaternions are not able to represent spins of more than 2π with respect to the reference position if we use the whole hypersphere, and of more than π if we limit ourselves to the hemisphere where w is positive. This will be important when using quaternions for animation.

3 Linear interpolation of quaternions

We noted already that all unit quaternions occupy a hypersphere in the 4-dimensional space. It is not trivial to do linear interpolation on such a surface. Therefore we will develop quaternion interpolation intuitively from an analogy with the 3D vector space.
3D vectors form an additive group, quaternions form a multiplicative group. The inverse of a vector $\bar{v}$ is $-\bar{v}$ while the inverse of a quaternion Q is Q^{-1} with

$$Q^{-1} = (w, -x, -y, -z) \quad (17)$$

Scaling a vector $\bar{v}$ is done by $\alpha \cdot \bar{v}$ (α being a scalar) while scaling a quaternion Q goes like

$$Q' = Q^{\alpha} = (\cos\theta, \sin\theta(X, Y, Z))^{\alpha}$$
$$= (\cos\alpha\theta, \sin\alpha\theta(X, Y, Z)) \quad (18)$$

The vector $\Delta\bar{v}$ which brings $\bar{v}_1$ to $\bar{v}_2$ is

$$\Delta\bar{v} = \bar{v}_2 - \bar{v}_1 = -\bar{v}_1 + \bar{v}_2 \quad (19)$$

while the quaternion ΔQ which transforms Q_1 into Q_2 is by analogy

$$\Delta Q = Q_1^{-1} \cdot Q_2 \quad (20)$$

Linear interpolation between $\bar{v}_1$ and $\bar{v}_2$ can be written as

$$\bar{v} = \bar{v}_1 + \alpha \cdot (\bar{v}_2 - \bar{v}_1); 0 \leq \alpha \leq 1 \quad (21)$$

where $\bar{v}$ goes from $\bar{v}_1$ to $\bar{v}_2$ as α goes from 0 to 1. As this is a frequently used function, we define a function "*lerp*" (from "Linear intERPolation"):

$$\bar{v} = lerp(\bar{v}_1, \bar{v}_2, \alpha) = \bar{v}_1 + \alpha \cdot (\bar{v}_2 - \bar{v}_1) \quad (22)$$

From the analogy we can define a "*spherical linear interpolation*" function or "*slerp*" in the unit quaternion space (remember this is a sphere) like this

$$Q = slerp(Q_1, Q_2, \alpha) = Q_1 \cdot (Q_1^{-1} \cdot Q_2)^{\alpha}; 0 \leq \alpha \leq 1 \quad (23)$$

The interpolated vector $\bar{v}$ goes along a straight line and the interpolated quaternion Q goes along a great arc on the 4D sphere, which is the shortest way between Q_1 and Q_2. For $\alpha = 0.5$, we get a special case for *lerp*( ) and *slerp*( ), because the result is the midpoint between $\bar{v}_1$ and $\bar{v}_2$ or Q_1 and Q_2 (the appendix shows how easily *slerp*( ) and *smid*( ) are put into code, see Shoemake 1987):

$$mid(\bar{v}_1, \bar{v}_2) = lerp(\bar{v}_1, \bar{v}_2, 0.5) \quad (24)$$
$$smid(Q_1, Q_2) = slerp(Q_1, Q_2, 0.5) \quad (25)$$

If α lies outside the range $[0, 1]$, we extrapolate $\bar{v}$ or Q on a straight line or great arc. Extrapolation doesn't give any problems in vector space and it is easy to see it doesn't give problems in quaternion space, as long as there is no ambiguity between extrapolated quaternions and interpolated quaternions. This is the case if we restrict ourselves to the hemisphere where w is positive, and if $\alpha < 2$, because an extrapolated quaternion can get – in the worst case – only on the back of the hypersphere (where w is negative).

4 Spline subdivision

Most cubic splines used nowadays can be written in the form

$$S_i(t)=[t^3\,t^2\,t\,1]\cdot P\cdot\begin{bmatrix}D_i\\D_{i+1}\\D_{i+2}\\D_{i+3}\end{bmatrix} \tag{26}$$

where $S_i(t)$, $(0\le t\le 1)$ is the i-th polynomial segment of the spline and D_i, $(0\le i\le m)$ are the control points (there are $m+1$ of them) (see Duff 1986). P is a matrix holding the polynomial coefficients and incorporates the properties of the spline (Foley 1984).

The class of splines, covered by the above representation, is local, linear and uniform. A local spline is one for which changing the value of a single control point D_j, $(0\le j\le m)$ affects only a bounded number (in this case 4) of spline segments in the control point's vicinity. A linear spline is one for which linear transformations of the spline can be reduced to (the same) linear transformation of the control points. A uniform spline is one for which each polynomial segment is defined along a parameter interval of length 1.

Generalisation to non-uniform control point spacing is quite straightforward (Duff 1986). The properties of the spline are fully controlled by the matrix P. For instance the family of uniform cardinal splines (which have first order continuity) are represented by (Clark 1981):

$$P_C=\begin{bmatrix}-c & 2-c & c-2 & c\\ 2c & c-3 & 3-2c & -c\\ -c & 0 & c & 0\\ 0 & 1 & 0 & 0\end{bmatrix};\ 0\le c\le 1 \tag{27}$$

An interesting member of the cardinal spline family is the Catmull-Rom cubic spline for which $c=0.5$ (Catmull 1974). Most computer animation systems use cardinal splines for interpolation between keyframes.

B-splines on the other hand have already been very popular for years because they generate extremely smooth interpolations between a set of control points (Prenter 1975; Foley 1984). They have second order continuity but don't pass through the control points. Tensioned B-splines (which are identical to Beta2-splines, see Barsky 1985) have the same properties but give something in between ordinary B-splines and straight lines (Duff 1986). The more tension is applied, the closer the spline approximates the control points. The tensioned B-spline matrix is:

$$P_{TB}=\begin{bmatrix}-c & 12-9c & 9c-12 & c\\ 3c & 12c-18 & 18-15c & 0\\ -3c & 0 & 3c & 0\\ c & 6-2c & c & 0\end{bmatrix};\ 0\le c\le 1 \tag{28}$$

Tensioned B-splines reduce to normal B-splines for $c=1$ (no tension) and to straight lines for $c=0$ (infinite tension). The behaviour of cardinal splines and tensioned B-splines for varying c can be found in Duff 1986.

Given the above formulation, it is easy to find the beginpoint ($t=0.0$), the midpoint ($t=0.5$) and the endpoint ($t=1.0$) of a polynomial segment of a spline. In doing so, we subdivide each spline segment in two halves. For the cardinal spline there comes:

$$\textit{beginpoint}\ B_i(t=0.0)=D_{i+1} \tag{29}$$

$$\textit{midpoint}\ M_i(t=0.5)=-\frac{c}{8}D_i+\left(\frac{1}{2}+\frac{c}{8}\right)D_{i+1}+\left(\frac{1}{2}+\frac{c}{8}\right)D_{i+2}-\frac{c}{8}D_{i+3} \tag{30}$$

$$\textit{endpoint}\ E_i(t=1.0)=D_{i+2}=B_{i+1} \tag{31}$$

A crude approximation for that spline segment can be obtained by drawing straight lines from B_i to M_i and from M_i to E_i (see bold line in Fig. 3), where M_i is the only point we need to construct. It is easy to see that

$$M_i=lerp\left(mid(D_i,D_{i+3}),\,mid(D_{i+1},D_{i+2}),\,1+\frac{c}{4}\right);\quad 0\le c\le 1 \tag{32}$$

In Fig. 3, we constructed this midpoint M_i graphically with $c=0.5$ (Catmull-Rom spline).

If we construct these points for each segment i and draw straight lines in between them, we get a piecewise linear approximation of the spline we want to construct. We try to refine this approximation in the following way. We use the original set of control points $\{D_i\}$ plus the set of constructed points $\{M_i\}$ as new set of control points on which we apply the same subdivision scheme (29) through

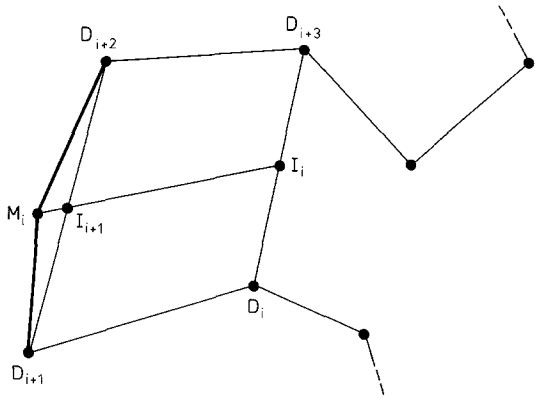


Fig. 3. Graphical construction of the Catmull-Rom spline

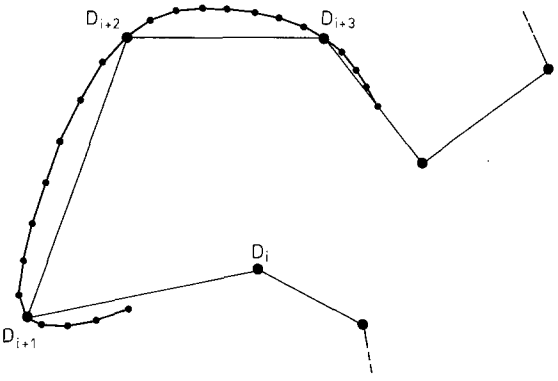


Fig. 4. Approximated cardinal spline after 3 subdivisions

(31). After a limited number of subdivisions we get a fairly good approximation of the spline we want to construct (Fig. 4).

Although this way of working is very attractive, we have to point out that the constructed points of the second or more generation are not guaranteed to lie on the spline itself. This stems from the fact that in most cases the boundary conditions, imposed on $S_i(t)$ in $t=0.0$ and $t=1.0$ do not hold exactly in $t=0.5$, but they do approximately for most splines (Pletinckx 1987), so the result is a good approximation of the spline itself. If we would continue this subdivision process an infinite number of times, we would end up with a curve of a degree, much higher than 3 (in fact of an infinite degree), but of which the coefficients of higher order drop dramatically $-O(h^4)$ to be precise – to zero.

The subdivision scheme we proposed here (also published in Pletinckx 1988) was also independently discovered by Dyn et al. 1987, but they don't mention the connection with cardinal splines. They give an indepth overview of the mathematical properties of this scheme.

The same cooking recipe can be applied to tensioned B-splines: out of Eq. (26) and (28) we calculate that

$$\textit{beginpoint } B_i(t=0.0) = \frac{c}{6}(D_i + D_{i+2}) + \left(1-\frac{c}{3}\right) D_{i+1} \tag{33}$$

$$\textit{midpoint } M_i(t=0.5) = \frac{c}{48}(D_i + D_{i+3}) + \left(\frac{1}{2}-\frac{c}{48}\right)(D_{i+1} + D_{i+2}) \tag{34}$$

$$\textit{endpoint } E_i(t=1.0) = B_{i+1} = \frac{c}{6}(D_{i+1} + D_{i+3}) + \left(1-\frac{c}{3}\right) D_{i+2} \tag{35}$$

Equation (34) can be approximated by

$$\textit{midpoint } M_i(t=0.5) = \frac{1}{2}(D_{i+1} + D_{+2}) = \textit{mid}(D_{i+1}, D_{i+2}) \tag{36}$$

Equation (33) can be rewritten in terms of *lerp*( ) and *mid*( ) functions:

$$B_i(t=0.0) = \textit{lerp}\left(D_{i+1}, \textit{mid}(\textit{mid}(D_i, D_{i+1}), \textit{mid}(D_{i+1}, D_{i+2})), \frac{2}{3}c\right) \tag{37}$$

$$= \textit{lerp}\left(D_{i+1}, \textit{mid}(M_{i-1}, M_i), \frac{2}{3}c\right) \tag{38}$$

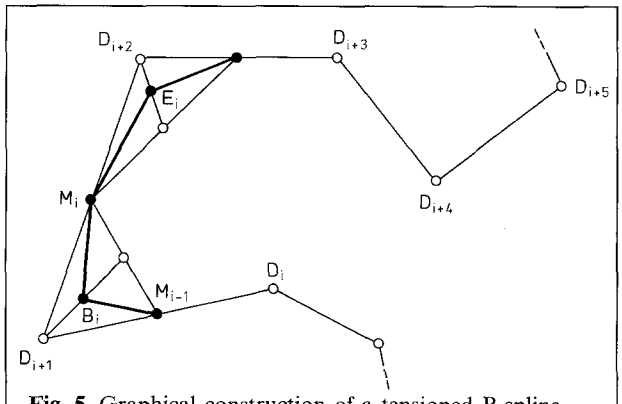


Fig. 5. Graphical construction of a tensioned B-spline for $c=\frac{3}{4}$

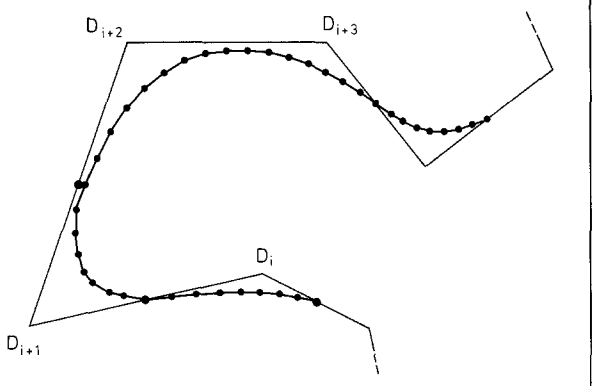


Fig. 6. Approximated tensioned B-spline after 3 subdivisions

An interesting case occurs when $c=3/4$ where the *lerp*( ) of Eq. (37) reduces to a *mid*( ):

$$B_i\left(t=0.0; c=\frac{3}{4}\right)=mid(D_{i+1}, mid(M_{i-1}, M_i)) \qquad (39)$$

The subdivision scheme, based on (36) and (39), and mentioned by Tom Duff (1986), was used to construct B_i, M_i and E_i in Fig. 5. Successively applying this scheme three times yields the piecewise linear approximation of Fig. 6.

The two schemes we proposed here, together with analogue schemes for other types of splines (Pletinckx 1987) are successfully used in the high resolution digital paint system CREATOR, developed by Barco-Industries Creative Systems. In the next section, we will also use these subdivision methods for splining quaternions in a computer animation system.

5 Splining quaternions

The piecewise linear approximation of a spline can also be applied in the quaternion space by using the simularity of "*lerp*" in the Cartesian space with "*slerp*" in the quaternion space. This means that a straight line in Cartesian space is replaced by a great arc in quaternion space. Each control point D_i will correspond to a keyframe quaternion. To find the quaternion for a certain in-between frame, we locate the proper linear segment and use linear interpolation to obtain the corresponding quaternion.

Already three algorithms have been proposed to spline quaternions: a Bezier interpolation scheme by Ken Shoemake (1985), a B-spline interpolation scheme by Tom Duff (1986) and a Boehm quadrangle spline by Ken Shoemake (1987). The scheme by Tom Duff has the advantage of (extreme) smoothness because B-splines have a continuity of second degree but consequently the disadvantage of not passing through the control points. It also lacks local control of behaviour of the spline. The schemes by Ken Shoemake have the advantage of local control of the spline, but as this is accomplished by placing and/or moving quaternion control points, it can be cumbersome to do this interactively and it is not transparent to the user.

As cardinal splines pass through the control points they don't need additional control points which are not associated with keyframes, and have spline behaviour control via the c-parameter (which can differ for each spline segment). Therefore they are most suited for interpolating quaternions in a keyframe animation system. Out of a set $\{Q_i;\ i=0, 1, \ldots, m\}$ of keyframes quaternions, one can generate a more elaborate set $\{Q'_j;\ j=0,1,\ldots,\ 2m\}$ by cardinal spline subdivision with

$$Q'_j=slerp\left(smid(Q_i, Q_{i+3}),\ smid(Q_{i+1}, Q_{i+2}), 1+\frac{c}{4}\right)$$
$$with\ j=2i+1;\ 0\le i<m;\ 0\le c\le 1 \qquad (40)$$

$$Q'_j=Q_i \quad with \quad j=2i;\ 0\le i\le m \qquad (41)$$

If we restrict the keyframe quaternions Q_i to the hemisphere where w is non-negative, and as the extrapolation factor α doesn't exceed 1.25, no ambiguities occur and the algorithm is robust, which was proved by several tests.

If a higher degree of smoothness is required or if extrapolation cannot be allowed, we can use the tensioned B-spline scheme (Eqs. (36), (38)). For quaternions, this looks like

$$Q'_j=smid(Q_{i+1}, Q_{i+2}) \quad with \quad j=2i+1;\ 0\le i<m \qquad (42)$$

$$Q'_j=slerp(Q_{i+1}, smid(Q'_{j-1}, Q'_{j+1}))$$
$$with \quad j=2i; \quad 0\le i\le m; \quad 0\le c\le 1 \qquad (43)$$

As no extrapolation is involved, this works on the whole quaternion hypersphere. No ambiguities are introduced if spins of maximum $\pm 2\pi$ with regard to the reference position are allowed. This can be

a slight drawback when animating objects (extensive spins need more keyframes) but is more than sufficient for animating cameras and light-sources.

In practice, the subdivision operation has to be repeated only 3 or 4 times. Out of the resulting set, the quaternions for each frame can be calculated by a single *slerp*( ). As the transformations from Euler angles to quaternions and from quaternions to rotation matrices (for use in the visualisation software or hardware) are well defined, the algorithm can easily be fitted in existing software. Table 1 gives an overview of the computational cost of the proposed algorithms in terms of the basic functions *slerp*( ) and *smid*( ) while Table 2 shows timings of those functions for three different computers.

Table 1. Number of function calls/segment

Scheme	Subdivision level		
	1	2	3
Bezier	6 S	18 S	42 S
B-spline	1 S+2 M	3 S+6 M	7 S+14 M
Boehm	3 S	9 S	21 S
Cardinal	1 S+2 M	3 S+6 M	7 S+14 M

Table 2. Time/function call for different computers

Computer	slerp( )	smid( )
Celerity C1230	98 μs	61 μs
Iris 4 D/70 G	31 μs	20 μs
Iris 3130	396 μs	85 μs

($S = slerp(\ ), M = smid(\ )$)

6 Quaternions for modeling

Some modelling tasks become quite simple when working with quaternions. As an example, we take the "propagation control graph" (PCG) of the MIRALAB modelling program (Magnenat-Thalmann and Thalmann 1985, 1986). A PCG is a set of control polygons, which defines a mesh of control points. With this mesh, one can construct a spline surface with beta-splines, Bezier splines or some other spline types. A PCG is constructed by

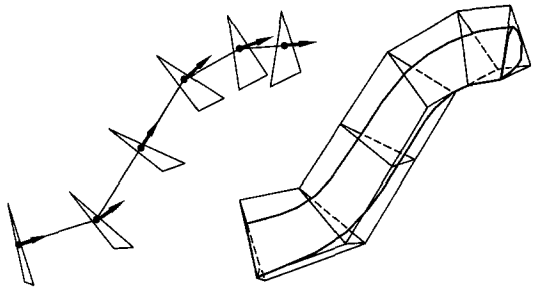


Fig. 7. Propagation control graph and resulting surface (β-spline)

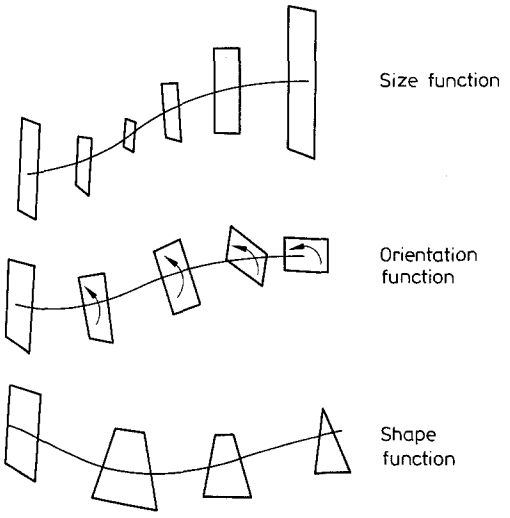


Fig. 8. Scaling, rotating and interpolating the control polygon (Magnenat-Thalmann and Thalmann 1985)

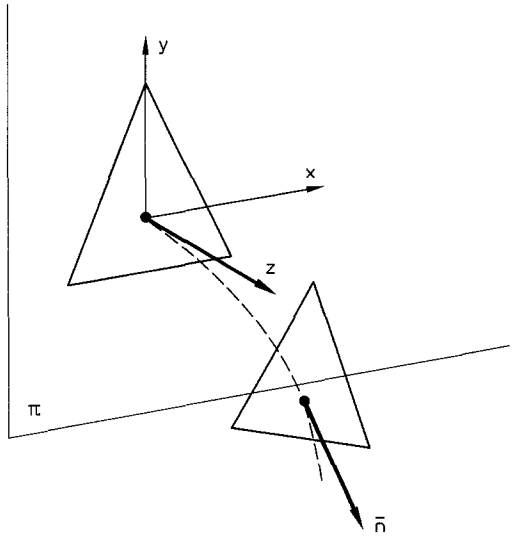
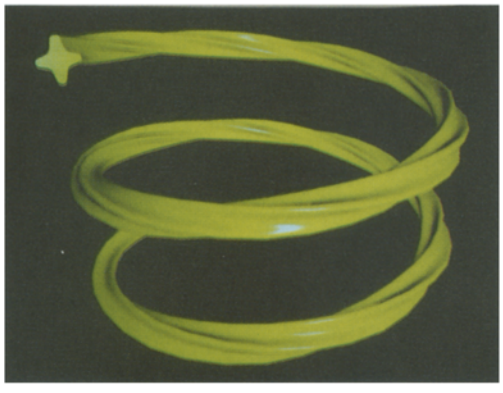
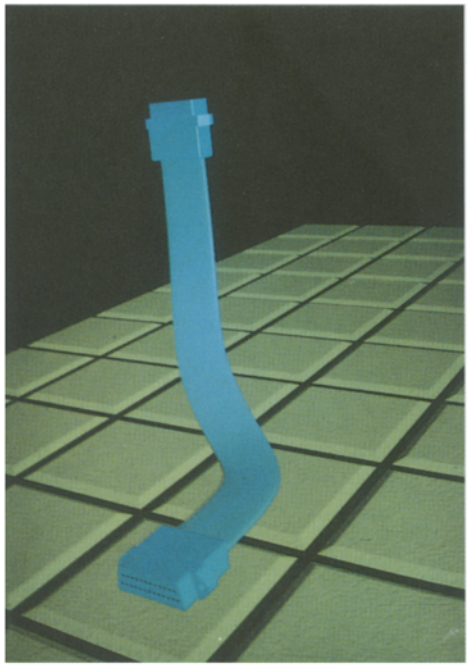


Fig. 9. Construction of a 3D propagation control graph

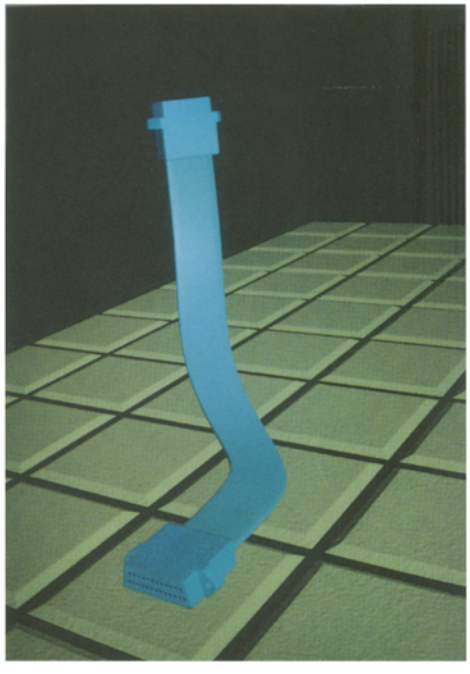
10



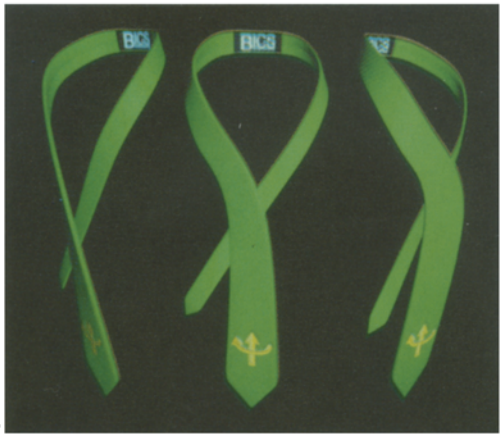
11



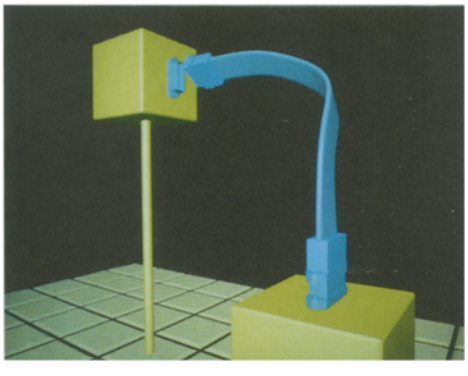
12



13



15



14

16

repeating a control polygon along a path (Fig. 7a) which generates a mesh of points (Fig. 7b). While repeating, the control polygon can be rotated, scaled or interpolated to another control polygon (Fig. 8).

The control polygon is supposed to be a 2-dimensional figure. The path or "backbone" of the PCG is perpendicular to the plane of the control polygon at the point of intersection of both (Fig. 7). If the backbone is a 2-dimensional curve, there is no problem in finding the 3-dimensional orientation of the control curve. If the backbone is a 3-dimensional curve, we can make the normal of the control polygon coincide with the tangent of the backbone, but we need a criterion to decide on the orientation of the control polygon. Most authors simply omit this problem (see for instance Magnenat-Thalmann and Thalmann 1985, 1986; Coquillart 1987). A good criterion is that, when no rotation around the backbone is specified, the orientation of the control polygons should vary as little as possible with respect to the orientation of the first control polygon.

We can solve the problem with quaternions. Suppose we use the orientation of the first control polygon (at the beginning of the backbone) as reference position, to which we connect coordinate axes xyz, z being the normal of the first control polygon and π being the plane of the xy axes. As quaternions are a rotation independent description, the solution of our problem will not depend on that reference position. Rotating the control curve, so that the normal $\bar{n}$ of the control polygon matches the tangent of the backbone at some point, can be done with a quaternion whose rotation axis lies in the plane π of the first control polygon (Fig. 9). This leaves one degree of freedom to specify the rotation around the backbone (rotation around z). As the quaternion we use to match the tangent, is situated on a minimal distance from the reference position (1, 0, 0, 0), the orientation variation is minimal, which proves our criterion.

Figures 10 and 11 show three-dimensional rendered propagation control graphs, with a cross shaped control polygon. In Fig. 10, no rotation around the backbone is specified. In Fig. 11, a rotation of 2π per turn of the spiral makes the resulting surface twist around the backbone. Figure 12 shows the modelling of a flatcable, where no twisting around the backbone was specified, while Fig. 13 shows the same for a twisting of 60 degrees. Figures 15 and 16 show a tie and a plant which were modelled with 3D PCGs with twisting and scaling. Rendering was done on the PRODUCER system of Barco-Industries Creative Systems, with the use of the Wavefront IMAGE software. Texture maps where painted on the CREATOR paint system.

Fig. 10. Spiral without torsion

Fig. 11. Spiral with torsion

Fig. 12. Flactable without torsion

Fig. 13. Flactable with 60 degrees torsion

Fig. 14. A frame out of "Cable & Co.", where the flactable is animated as a flexible object (by animating the skeleton and the torsion)

Fig. 15. One of my ties

Fig. 16. Plant

7 Quaternions for rendering

Most polygon renderers use normal interpolation, i.e. they interpolate the normal across each polygon to give the impression that the surface has a continuous curvature and is not build out of facets. Phong shading for instance needs normal interpolation to render highlights properly.

In Eq. (14) we showed that unit vectors also can be written in quaternion format. So each vertex normal can be written as a vertex quaternion. Figure 17 shows a polygon where each vertex has its corresponding quaternion. If we go along the edge AB, $Q_A^{-1}Q_B$ will be the quaternion which transforms Q_A into Q_B. If we go along the edge in $n+1$ steps,

$$\Delta Q_{AB}=(Q_A^{-1}\cdot Q_B)^{1/n} \tag{44}$$

will be the quaternion who transforms Q_D into the Q_D of the next scanline. The same holds for the quaternion Q_E along the edge AC where

$$\Delta Q_{AC}=(Q_A^{-1}\cdot Q_C)^{1/m} \tag{45}$$

and where there are $m+1$ scanlines from A to C. Interpolation along the scanline DE (over $k+1$ pixels) gives

$$\Delta Q_{DE}=(Q_D^{-1}\cdot Q_E)^{1/k} \tag{46}$$

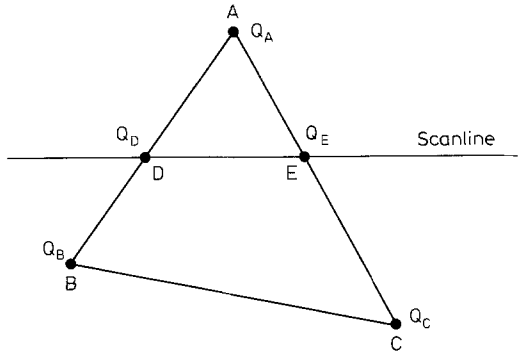


Fig. 17. Vertex quaternions

So one can calculate the appropriate quaternion Q for each pixel, which represents the interpolated normal in that pixel. The following piece of pseudocode gives the structure of a rendering program with quaternion interpolation:

$\Delta Q_{AB} = (Q_A^{-1} \cdot Q_B)^{1/n}$
$\Delta Q_{AC} = (Q_A^{-1} \cdot Q_C)^{1/m}$
$Q_D = Q_E = Q_A$
for each scanline over $k+1$ pixels
{
$Q = Q_D$
$\Delta Q_{DE} = (Q_D^{-1} \cdot Q_E)^{1/k}$
for each pixel on the scanline
{
shading calculations
$Q = Q \cdot \Delta Q_{DE}$
}
$Q_D = Q_D \cdot \Delta Q_{AB}$
$Q_E = Q_E \cdot \Delta Q_{AC}$
}

One can see that there is no need for normalising the quaternions (as intended by Hamilton), they stay normalised all the time. As normalising and incrementing a normal along a scanline can take up to 50% of the time spent in the inner loop of a normal renderer (a great deal of the time is spent in calculating a square root, see Earnshaw 1987), and as quaternion multiplication is quite efficient and easy, we can save time per pixel we render. Table 3 shows how much faster the quaternion incrementation works in comparison with incrementation and normalisation of normals (code was written in C)

Table 3. Speed up comparisons

Computer	Incrementation Speedup	Max. rendering Speedup
Iris 3130	1.81	1.29
Iris 4 D/70 G	2.05	1.35
Celerity C1230	1.25	1.11

8 Conclusion

In this paper we showed the advantages of using quaternions in animation, modelling and rendering. We proposed two new subdivision schemes for approximating parameterised splines and used this in a keyframe animation system, which uses quaternions internally for better orientation representation and efficiency. We indicated that orientation problems in modelling can be solved by the use of quaternions and we developed a new solution to the normalization problem of normals in rendering and showed the speed up which goes with this.

Acknowledgements. I would like to thank Jean Despaey, Jef Vandenberghe, David Eldridge and Peter De Mangelaere for reading and commenting this paper. Thanks also to Marianne Gryson, Hilde Verthe and Veerle Delange for the text and the illustrations. I would like to acknowledge Luc De Simpelaere for creating the opportunity and the environment to explore computer graphics and for stimulating the development of new ideas in this field.

References

Barsky BA, DeRose TD (1985) The beta2-spline: a special case of the beta-spline curve and surface representation. IEEE Comput Graph Appl (September 1985), pp 46–58

Blake EH (1987) A metric for computing adaptive detail in animated scenes using object oriented programming. Proc Eurographics 87, pp 295–307

Boehm W, Farin G, Kahmann J (1984) A survey of curve and surface methods in CAGD. Computer Aided Geometric Design 1, pp 1–60

Catmull E, Rom R (1974) A class of local interpolating splines. In: Barnhill RE, Riesenfeld RF (eds) Computer Aided Geometric Design, Academic Press, San Francisco, pp 317–326

Clark J (1981) Parametric curves, surfaces and volumes in computer graphics and computer-aided geometric design. Tech Rep 221, Comput Syst Lab, Stanford Univ, Palo Alto, California (November 1981)

Coquillart S (1987) A control-point-based sweeping technique. IEEE Comput Graph Appl (November 1987), pp 36–45

Duff T (1986) Splines in animation and modelling. Siggraph 86 course # 15: State of the art in image synthesis

Dyn N, Levin D, Gregory JA (1987) A 4-point interpolatory subdivision scheme for curve design. Computer Aided Geometric Design 4, pp 257–268
Earnshaw RA (1987) The mathematics of computer graphics, The Visual Computer 3:115–124
Foley JD, Van Dam A (1984) Fundamentals of interactive computer graphics. Addison Wesley
Hankins TL (1980) Sir William Rowan Hamilton. The John Hopkins University Press
Magnenat-Thalmann N, Thalmann D (1985) Area, spline-based and structural models for generating and animating 3D characters and logos. The Visual Computer 1:15–23
Magnenat-Thalmann N, Thalmann D (1986) Building complex bodies: combining computer animation with CAD. Computers in Mechanical Engineering (May 1986), pp 26–33
Pletinckx D (1987) The use of spline subdivision in computer animation and digital painting. Barco Industries Creative Systems technical memo # TM.03DEC87.DP
Pletinckx D (1988) The use of quaternions for animation, modelling and rendering. In: Magnenat-Thalmann N, Thalmann D (eds) New Trends in Computer Graphics (Proc CG International '88), Springer, Berlin Heidelberg New York, pp 44–53
Prenter PM (1975) Splines and variational methods. Wiley-Interscience
Shoemake K (1985) Animating rotation with quaternion curves. Comput Graph (Proc Siggraph 85) 19(3):245–254
Shoemake K (1987) Quaternion calculus and fast animation, Siggraph 87 course # 10: Computer Animation: 3D Motion specification and control, pp 101–121

Appendix

C code to implement quaternion interpolation

```
# define W 0
# define X 1
# define Y 2
# define Z 3
typedef double quaternion;
smit(q1, q2, q)
quaternion q1[ ], q2[ ], q[ ];
{
    double sqrt( ), sum, factor;
    sum = q1[W]*q2[W] + q1[X]*q2[X]
    + q1[Y]*q2[Y] + q1[Z]*q2[Z];
    factor = 1.0/sqrt(2.0*(1.0 + sum));
    q[W] = (q1[W] + q2[W])*factor;
    q[X] = (q1[X] + q2[X])*factor;
    q[Y] = (q1[Y] + q2[Y])*factor;
    q[Z] = (q1[Z] + q2[Z])*factor;
      return;
}
slerp(q1, q2, alfa, q)
quaternion q1[ ], q2[ ], q[ ];
double alfa;
{
  double acos( ), sin( ), sum;
  double beta1, beta2, teta;
  sum = q1[W]*q2[W] + q1[X]*q2[X]
  + q1[Y]*q2[Y] + q1[Z]*q2[Z];
  teta = acos(sum);
  if (teta ≤ EPSILON)
    {
    beta1 = 1.0 − alfa;
    beta2 = alfa;
    }
  else
    {
    beta1 = sin((1.0 − alfa)* teta)/sin(teta);
    beta2 = sin(alfa* teta)/sin(teta);
    }
  q[W] = beta1*q1[W] + beta2*q2[W];
  q[X] = beta1*q1[X] + beta2*q2[X];
  q[Y] = beta1*q1[Y] + beta2*q2[Y];
  q[Z] = beta1*q1[Z] + beta2*q2[Z];
  return;
}
```

DANIEL PLETINCKX received a M.Sc. degree in electrical engineering in 1984 from the University of Ghent, Belgium. Since then, he is a software engineer at Barco-Industries in Belgium. After carrying out research in image processing and medical imaging, he joined the Creative Systems division in 1985, where the is involved in the design and implementation of 2D and 3D computer graphics systems.