3D Transformations and Interpolations Based on Quaternions

Kari Peisa

Abstract

Peisa, Kari (2005). 3D Transformations and Interpolations Based on Quaternions. In *Proceedings of the Algorithmic Information Theory Conference, Vaasa 2005.* Proceedings of the University of Vaasa, Reports 124, 141-152. Eds S.Hassi,V.Keränen, C.-G. Källman, M. Laaksonen, and M. Linna.

Quaternion technique has been effectively used in the recent development of computer graphics and game programming. The potential of quaternions as a general and powerful rotation operator has been widely recognized. Many recent graphics APIs, for instance Java-3D API and DirectX, provide functions for quaternion operations. In this paper we provide a mathematical summary and an implementation of the quaternion algebra and calculus as a set of rewrite rules of *Mathematica* for four dimensional vector space over the real number field $\mathbb{R}$. The properties of *Mathematica* as a rewrite language are also discussed. By using quaternion technique and symbolic computation we introduce some general functions that perform various transformations and interpolations. As an example we show how these functions can be used for illustrating rotations and curves constructed by using unit quaternions. The motivation for this paper originates from some obstacles that arose when the add-on `Algebra`‘`Quaternions`‘ package of *Mathematica* was experimented in connection with computer graphics.

Kari Peisa, School of Technology, Rovaniemi Polytechnic, E-mail: Kari.Peisa@ramk.fi

Keywords: quaternion, quaternion spline interpolation, computer graphics

1. Implementing the Quaternion Algebra in *Mathematica*

In this paper we denote quaternion by

$$quat[w, x, y, z] = w\,\mathrm{E} + x\,\mathrm{I} + y\,\mathrm{J} + z\,\mathrm{K}, \tag{1}$$

where the coefficients $w, x, y, z \in \mathbb{R}$. Elements E, I, J and K represent *unit quaternions* that obey the following Hamilton's rules

$$\mathrm{I}.\mathrm{I} \;=\; \mathrm{J}.\mathrm{J} \;=\; \mathrm{K}.\mathrm{K} \;=\; \mathrm{I}.\mathrm{J}.\mathrm{K} \;=\; -1. \tag{2}$$

These primitive elements are also called *imaginary principals* according to William Rowan Hamilton (1805-1865) who invented quaternions when he tried to generalize the meaning of complex numbers for rotations in the plane to rotations in three dimensional space through ordered number triplets with two imaginary components and one as a real number. However this proved to be impossible and four numbers, that is quaternions, are needed to specify 3D rotations.
Define $\mathrm{q}_0 = quat[w_0, x_0, y_0, z_0]$ and $\mathrm{q}_1 = quat[w_1, x_1, y_1, z_1]$. The dot ($\cdot$) for the multiplication of quaternions is defined by

$$\begin{aligned}(3)\qquad \mathrm{q}_0 \cdot \mathrm{q}_1 = quat[&w_0w_1 - x_0x_1 - y_0y_1 - z_0z_1,\\ &w_0x_1 - x_0x_1 - y_0y_1 - z_0z_1,\\ &w_0y_1 - x_0z_1 + y_0w_1 + z_0x_1,\\ &w_0z_1 + x_0y_1 - y_0x_1 + z_0w_1].\end{aligned}$$

The following definitions complete the four dimensional vector space H over the real number field $\mathbb{R}$:

$$(4)\qquad \mathrm{q}_0 \pm \mathrm{q}_1 = quat[w_0 \pm w_1, x_0 \pm x_1, y_0 \pm y_1, z_0 \pm z_1]$$

$$(5)\qquad \mathrm{q}_0^* = quat[w_0, -x_0, -y_0, -z_0]$$

$$(6)\qquad norm(\mathrm{q}_0) = w_0^2 + x_0^2 + y_0^2 + z_0^2$$

$$(7)\qquad \mathrm{q}_0^{-1} = q_0^*/norm(\mathrm{q}_0)$$

Multiplication of quaternions is associative and distributes across addition but is not generally commutative. Scalar multiplication obeys the ordinary componentwise multiplication and division as with vectors. For the complete presentation of the quaternion mathematics the reader is referred to Damm et al (1998).
As Hamilton introduced, quaternions can be represented by

$$(8)\qquad \mathrm{q} \;=\; w + \mathbf{v},$$

where $w \in \mathbb{R}$ is called the *real part* and $\mathbf{v} = (x, y, z) \in \mathbb{R}^3$ is called the *vector part* of the quaternion. The set of *pure quaternions* denoted by $\dot{\mathrm{H}}$, i.e, quaternions with real part equal to zero, is isomorphic with the 3D vector space. It is known that the representation (8) makes it possible also to represent the quaternion product by using the ordinary dot product and cross product defined for 3D vectors.

$$(9)\qquad \mathrm{q}_0 \cdot \mathrm{q}_1 \;=\; w_0w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1 + w_0\mathbf{v}_1 + w_1\mathbf{v}_0 + \mathbf{v}_0 \times \mathbf{v}_1$$

It is also known that unit quaternions in the expression (1) can be represented by using special complex valued $\mathbb{C}_{2\times 2}$ matrices (10) called Pauli matrices:

$$\text{(10)} \qquad \mathrm{E} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathrm{I} = \begin{pmatrix} i & 0 \\ 0 & i \end{pmatrix}, \mathrm{J} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \mathrm{K} = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}.$$

In the matrix representation multiplication (3), addition and subtraction (4) of quaternions and also the scalar multiplication obey the standard rules of matrix algebra. Instead of $\mathbb{C}_{2\times 2}$ matrices we can also use special real valued $\mathbb{R}_{4\times 4}$ matrices

Mathematica does not include quaternion algebra as standard algebra. There is an add-on package named `Algebra‘Quaternions‘`, where the Hamilton's quaternion algebra is implemented as an extension of the complex number field $\mathbb{C}$. Unfortunately, some important extracting functions and the algebraic functions defined in this package do not support symbolic computation. This is a defect for experimenting quaternion technics in connection with computer graphics.

The basic quaternion algebra (1) - (7) for vector space H with support of symbolic computations can be implemented in *Mathematica* quite shortly. The quaternion multiplication can be defined by three equivalent ways, namely by a vector representation, a matrix representation and a 4-tuple representation. In the vector representation we use equation (9) together with the standard operations of 3D vectors of *Mathematica*. With the $\mathbb{C}_{2\times 2}$ or $\mathbb{R}_{4\times 4}$ matrix representations we can use the built-in matrix algebra as we explained above.

We introduce here an implementation of the quaternion algebra based on a set of rewrite rules of *Mathematica* and using the 4-tuple representation (3) of the quaternion product. The rewrite rules in Mathematica are also function definitions. Some special properties of these definitions are very convenient, especially for programming mathematical rules of the form

$$arg1 \quad op \quad arg2 \quad = \quad definition.$$

Above the left hand side (*lhs*) has the internal form $op[arg1, arg2]$ and the *op* is some, maybe protected, built-in operator or a function. In *Mathematica* we are able to associate the definition of a rule either with the head of *lhs* or with the heads of the arguments appearing in the level one of *lhs*. Furthermore, user defined rules are automatically compiled into a special code based on a form of hashing which allows for rapid pattern matching. The following rules implement basic arithmetic for quaternions. Note that the head of *lhs* of each rule is a protected built-in function:

```
quat /: (Times[quat[a_, b_, c_, d_], exp_] /; Head[exp] =!= quat):=
  quat[exp a, exp b, exp c, exp d];
quat /: Plus[quat[w_, x_, y_, z_], q__quat] :=
  Thread[Unevaluated[Plus[quat[w, x, y, z], q]], quat];
quat /: Dot[quat[w0_, x0_, y0_, z0_], quat[w1_, x1_, y1_, z1_]] :=
    quat[w0 w1 - x0 x1 - y0 y1 - z0 z1,
    w0 x1 + x0 w1 + y0 z1 - z0 y1,
    w0 y1 - x0 z1 + y0 w1 + z0 x1,
    w0 z1 + x0 y1 - y0 x1 + z0 w1];
```

In the scalar multiplication we have the condition not to apply `Times` to multiplication of quaternions by accident. The attribute `Orderless` of the built-in function `Times` makes the rule work properly in spite of the order of the arguments of the function. The head `quat` is `Thread`*ed* over the sum for a more efficient computation in long sums. Function `Unevaluated` is used to avoid an infinite recursion.
The algebra for quaternion space H is completed by

```
quat/: Norm[quat[w_, x_, y_, z_]] := w^2 + x^2 + y^2 + z^2;
quat/: Conjugate[quat[w_, x_, y_, z_]] := quat[w, -x, -y, -z];
quat/: Power[quat[w_, x_, y_, z_], -1] :=
    Conjugate[quat[w, x, y, z]]/Norm[quat[w, x, y, z]];
```

The following rules define the extracting functions:

```
quat /: realPart[quat[w_, x_, y_, z_]] := w;
quat /: vectorPart[quat[w_, x_, y_, z_]] := {x, y, z};
quat /: normalize[quat[w_, x_, y_, z_]] :=
    quat[w, x, y, z]/Power[Norm[quat[w, x, y, z]], 1/2];
```

2. Rotations and Unit Quaternions

Let q be a quaternion and $\dot{\mathrm{v}}$ a pure quaternion representing a vector $\mathbf{v}$ in the quaternion space $\dot{\mathrm{H}}$. It can be proven that quaternion multiplication

$$\mathrm{q}.\dot{\mathrm{v}}.\mathrm{q}^{-1} \tag{11}$$

yields always a pure quaternion $\dot{\mathrm{w}}$ of the same norm as $\dot{\mathrm{v}}$. The expression (11) represents a rotation of the vector $\mathbf{v}$ by the quaternion q. Another fundamental property of (11) is that a scalar multiplication of the quaternion does not affect the result:

$$(\alpha\,\mathrm{q}).\mathrm{v}.(\alpha\,\mathrm{q})^{-1} = \alpha\,\mathrm{q}.\mathrm{v}.\frac{(\alpha\,\mathrm{q}^*)}{\alpha^2\|\mathrm{q}\|} = \mathrm{q}.\mathrm{v}.\frac{\mathrm{q}^*}{\|\mathrm{q}\|} = \mathrm{q}.\mathrm{v}.\mathrm{q}^{-1}. \tag{12}$$

Above $\|\mathrm{q}\|$ is the quaternion norm (6). From equation (12) it follows that with all the rotations about the origin in 3-space we need only consider unit quaternions. The inverse of a unit quaternion $\hat{\mathrm{q}}$ is simply the conjugate quaternion

$$\hat{\mathrm{q}}^{-1} = \hat{\mathrm{q}}^*. \tag{13}$$

Multiplication of two unit quaternions yields always a unit quaternion. Therefore the quaternion product $\hat{q}_2.\hat{q}_1$ defines a new quaternion rotation operator that can be generalized for the composition of any number of rotations. The fact $\|q\| = 1$ for unit quaternions is a constraint over the values of the components. Thus, a rotation about origin has three degrees of freedom. Furthermore, there is always a unique $\theta \in [0, \pi]$ such that each unit quaternion can be represented in the form

$$\hat{q} = \cos\theta + \sin\theta\ \hat{\mathbf{u}} \tag{14}$$

where $\hat{\mathbf{u}}$ is a unit vector. By the quaternion rotation (11) it can be proven that in the representation (14) vector $\hat{\mathbf{u}}$ (or $\mathbf{u} = sin\theta\ \hat{\mathbf{u}}$) represents the axis of the rotation and the angle 2θ is the rotation angle.

We denote $S^3 = \{q \in \mathbb{R}^4 \mid \|q\| = 1\}$ for quaternion rotations. The set of *special orthogonal* 3×3 matrices that provides a mathematical structure for rotations in 3-space about an axis through an origin is denoted by $SO(3) = \{M \in \mathbb{R}^3 \mid M.M^T = M^T.M = I \wedge detM = 1\}$. The correspondence between S^3 and $SO(3)$ is not quite an isomorphism since unit quaternions $\hat{q}$ and $-\hat{q}$ represent the same rotation and correspond to the same orthogonal matrix. The rotation through an angle $-\theta$ about an axis $-\mathbf{u}$ equals to the rotation through an angle θ about an axis $\mathbf{u}$.

$$(-\hat{q}).v.(-\hat{q})^* = \hat{q}.v.\hat{q}^*$$

The equation (9) can be used to obtain the elements of the corresponding rotation matrix from a given unit quaternion $\hat{q} = quat[w, x, y, z]$ with $x^2 + y^2 + z^2 = 1 - w^2$.

```
rotationMatrix3D[quat[w_,x_,y_,z_]] :=
        {{-1 + 2 w^2 + 2 x^2,2 x y + 2 w z, -2 w y + 2 x z},
          {2 x y - 2 w z, -1 + 2 w^2 + 2 y^2, 2 w x + 2 y z},
          {2 w y + 2 x z, -2 w x + 2 y z, -1 + 2 w^2 + 2 z^2}};
```

For convenience, it is useful to add a new definition to obtain the rotation matrix from a given rotation angle and a vector that represents the rotation axis:

```
rotationMatrix3D[ang_, axis_] /; (N[axis] =!= {0., 0.,0.}) :=
    Module[ {u, w, x, y, z},
    u = axis/Power[Dot[axis, axis], 1/2];
    w =Cos[ang/2];
    {x=Sin[ang/2] u[[1]], y=Sin[ang/2] u[[2]], z=Sin[ang/2] u[[3]]};
    rotationMatrix3D[quat[w, x, y, z]]];
```

Next we apply quaternion technique to produce a function that rotates quite a general graphics objects through a given angle and about any rotation axis. Actually we do not use quaternion algebra here but only the `rotationMatrix3D` function. We first define a general transformation function that applies any given function to desired

parts of a graphics object. In this version the desired parts are 3D points $\{x, y, z\}$ which may appear in special graphics primitives in the complicated structure of a graphics object.

```
transform3DObject[grObject_, myTransformation_] :=
    grObject /. {Point[p_] :> Point[myTransformation[p]],
                 Text[e_, p_] :> Text[e, myTransformation[p]],
                 head_[list_] /; MemberQ[{Line, Polygon}, head] :>
                     head[Map[myTransformation, list]]};
```

Note that the given argument `myTransformation` must be a function (usually a pure function). The motivation for the design of this function originates from a numeric version of the function made by **Veikko Keränen** from Rovaniemi Polytechnic:

```
transform3DObject[grObject_, myTransformation_] :=
    ReplaceAll[grObject, {x_?NumericQ, y_?NumericQ, z_?NumericQ} ->
        myTransformation[{x, y, z}]];
```

The following functions or their compositions can be given as an argument for applying matrix multiplication and translation.

```
linearMapping[matrix_, {x_,y_,z_}] := matrix.{x,y,z};
translate[{x_,y_,z_},{a_,b_,c_}] := {x+a,y+b,z+c};
```

For example, the pure function `linearMapping[matrix,translate[#,vector}]]&` as the argument of the transformation function provides an *affine mapping* in which the `matrix` argument represents a linear transformation and the `vector` argument represents a translation.

The general rotation function, that makes rotations about an axis passing through origin, is defined by

```
rotateShape[qrObject_,ang_,axis_] := Module[{rotMat},
    rotMat = rotationMatrix3D[ang, axis];
    transform3DObject[qrObject,linearMapping[rotMat,#]&]];
```

The more complicated case of rotating about an axis passing through any given point is defined shortly:

```
rotateShape[grObject_,ang_,axis_,pnt_] := Module[{t,r},
    t = transform3DObject[grObject,translate[#,-pnt]&];
    r = rotateShape[t,ang,axis];
    transform3DObject[r,translate[#,pnt]&]];
```

All the functions defined above can be executed using symbolic arguments, too. If one wishes to rotate plotted graphics objects, they should be in the `Graphics3D` representation form. This form is, for example, returned by the `ParametricPlot3D` function (see Figure 1).

```
obj1 =
  ParametricPlot3D[{Cos[t], Sin[t], t / (2 π), {Hue[0], Thickness[0.01]}},
   {t, 0, 2 π}, AxesLabel → {"x", "y", "z"}, ViewPoint -> {0.8, -1.1, 1.5},
   DisplayFunction → Identity];
Show[obj1, rotateShape[obj1, π / 2, {0, 0, 1}, {1, 0, 0}] /. {Hue[0] → Hue[0.15]},
  DisplayFunction → $DisplayFunction];
```

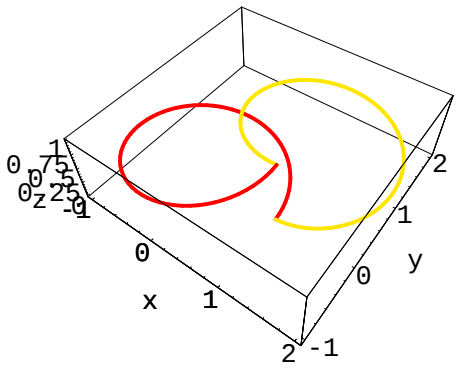

FIGURE 1. Rotating a graphics object

To extract the corresponding quaternion from a matrix representation we need to reverse the engineering of previous `rotationMatrix3D` function. However, there are some square roots with which we should be able to choose either the positive or the negative root. There is a strategy suggested by **Ken Shoemake** to solve this problem numerically. We write here only the solution code and omit the details. The reader is referred, for example, to Dunn et al. (2002).

```
fromRotationMatrix3D[m_] :=
Module[{s,biggestIndex,biggestVal,eff},
    s[1] = m[[1, 1]] + m[[2, 2]] + m[[3, 3]];
    s[2] = m[[1, 1]] - m[[2, 2]] - m[[3, 3]];
    s[3] = m[[2, 2]] - m[[1, 1]] - m[[3, 3]];
    s[4] = m[[3, 3]] - m[[1, 1]] - m[[2, 2]];
    biggestIndex = Ordering[{s[1], s[2], s[3], s[4]}, -1][[1]];
    biggestVal = 1/2 Power[1+ s[biggestIndex],1/2];
    eff = -1/(4 biggestVal);
    Which[biggestIndex == 1, {biggestVal, eff (m[[2, 3]] - m[[3, 2]]),
            eff (m[[3, 1]] - m[[1, 3]]), eff (m[[1, 2]] - m[[2, 1]])},
        biggestIndex == 2, {eff (m[[2, 3]] - m[[3, 2]]), biggestVal,
            eff (m[[1, 2]] + m[[2, 1]]), eff (m[[3, 1]] + m[[1, 3]])},
        biggestIndex == 3, {eff (m[[3, 1]] - m[[1, 3]]),
            eff (m[[1, 2]] + m[[2, 1]]), biggestVal,
            eff (m[[2, 3]] + m[[3, 2]])},
        biggestIndex == 4, {eff (m[[1, 2]] - m[[2, 1]]),
            eff (m[[3, 1]] + m[[1, 3]]), eff (m[[2, 3]] + m[[3, 2]]),
            biggestVal}];
```

3. Quaternion Calculus and Interpolation Algorithms

In this section we introduce the calculus of unit quaternions that is needed in two useful interpolation algorithms for rotations called *slerp* and *squad*. We implement the algebraic functions power, exponential, and the natural logarithm of a unit quaternion needed in the calculus. We also implement the differentiation rule for a unit quaternion raised to a power of a real-valued function, which is used in the derivation of *squad* algorithm.

Algebraic functions for unit quaternions can be defined by generalizing the Euler's identity for complex numbers

$$\exp(\theta i) = \sum_{n=0}^{\infty} \frac{(\theta i)^n}{n\,!} = \sum_{n=0}^{\infty} (-1)^n \frac{(\theta)^{2n}}{(2n)\,!} + i \sum_{n=0}^{\infty} (-1)^n \frac{(\theta)^{2n+1}}{(2n+1)\,!} = \cos\theta + i\,\sin\theta$$

to quaternions. We obtain the exponential of a pure quaternion

$$\dot{\mathrm{q}} = quat[0, x, y, z] = \|\dot{\mathrm{q}}\|\,\hat{\mathrm{q}} = \theta\,\hat{\mathbf{v}}$$

by substituting symbolically $\theta\,\hat{\mathbf{v}}$ for all occurrences of θi in the power series expansion of $\exp(\theta i)$ and using the identity $\hat{\mathbf{v}}.\hat{\mathbf{v}} = -1$ in the quaternion space $\dot{\mathrm{H}}$:

$$\exp(\theta\,\hat{\mathbf{v}}) = \cos\theta + \hat{\mathbf{v}}\,\sin\theta. \tag{15}$$

Note that the exponential function returns a unit quaternion. The inverse function, i.e., the natural logarithm of a unit quaternion $\hat{\mathrm{q}}$, is defined by

$$\ln\,\hat{\mathrm{q}} = \ln(\cos\theta + \hat{\mathbf{v}}\,\sin\theta) = \theta\,\hat{\mathbf{v}}. \tag{16}$$

One must be very careful when using exponential and logarithm functions as the corresponding real versions. For example, the real number rule $\ln(p\,q) = \ln p + \ln q$ does not hold for quaternions because of non-commutative multiplication. The power of a unit quaternion is defined by

$$\hat{\mathrm{q}}^t = (\cos\theta + \hat{\mathbf{v}}\,\sin\theta)^t = \exp(\theta t\,\hat{\mathbf{v}}) = \cos\theta t + \hat{\mathbf{v}}\,\sin\theta t. \tag{17}$$

Definitions (15), (16) and (17) can be implemented in *Mathematica* by the rules

```
quat /: Exp[quat[0,x_,y_,z_]] :=
  Block[{n = Power[x^2+y^2+z^2,1/2],s},
    s = Sin[n]/n;
    quat[Cos[n],s x,s y,s z]];
quat /: Log[quat[w_,x_,y_,z_]] :=
  Block[{s = ArcCos[w]}, s pureUnitXYZ[quat[w,x,y,z]]];
quat /: Power[quat[w_,x_,y_,z_],t_] :=
  Block[{s = ArcCos[w]},
    quat[Cos[t s],0,0,0] + Sin[t s] pureUnitXYZ[quat[w,x,y,z]]];
```

where the auxiliary function

```
quat /: pureUnitXYZ[quat[w_, x_, y_, z_]] :=
  Block[{s = 1/Power[1-w^2,1/2]}, quat[0, s x, s y, s z]];
```

returns a pure unit quaternion whose vector part is the normalized vector part of the unit quaternion given in the argument.

A unit quaternion $\hat{\mathrm{q}} = \cos\theta + \sin\theta\ \hat{\mathbf{u}}$ can be illustrated as the element of the unit sphere. The vector part of a unit quaternion represents the direction of the rotation axis. The length $\sin\theta$ of the vector part is determined by the rotation angle. In the interpolation of a rotation we need to find the shortest path between two unit quaternions. The shortest path can be projected onto the unit sphere as the *great arc* visualizing rotation matrices $SO(3)$. Spherical interpolation algorithm, *slerp* in short, is a mathematical formulation for the shortest path in S^3 that spans, in fact, the shortest great arc interpolation on $SO(3)$ (see Dam et al. (1998) pp. 45-46). Function *slerp* is defined by

$$slerp(\hat{\mathrm{q}}_1, \hat{\mathrm{q}}_2, t) = \hat{\mathrm{q}}_1 . (\hat{\mathrm{q}}_1^* . \hat{\mathrm{q}}_2)^t. \tag{18}$$

Term $\hat{\mathrm{q}}_1^* . \hat{\mathrm{q}}_2 = \cos\alpha + \hat{\mathbf{v}}\sin\alpha$, where α is the angle between $\hat{\mathrm{q}}_1$ and $\hat{\mathrm{q}}_2$. The angle α is, in fact, the angle of the rotation in 3D space from the orientation $\hat{\mathrm{q}}_1$ along the great arc between $\hat{\mathrm{q}}_1$ and $\hat{\mathrm{q}}_2$ to the orientation $\hat{\mathrm{q}}_2$. Multiplying $\hat{\mathrm{q}}_1$ by term $(\hat{\mathrm{q}}_1^* . \hat{\mathrm{q}}_2)^t = (\cos\alpha + \hat{\mathbf{v}}\sin\alpha)^t = \cos\alpha t + \hat{\mathbf{v}}\sin\alpha t$, where the argument t varies between 0 and 1, represents a rotation from the orientation $\hat{\mathrm{q}}_1$ along the great arc between $\hat{\mathrm{q}}_1$ and $\hat{\mathrm{q}}_2$ through a fraction of the angle α. Thus $slerp(0, \hat{\mathrm{q}}_1, \hat{\mathrm{q}}_2) = \hat{\mathrm{q}}_1$ and $slerp(1, \hat{\mathrm{q}}_1, \hat{\mathrm{q}}_2) = \hat{\mathrm{q}}_2$. Spherical quadrangle interpolation algorithm, *squad* in short, produce a spline, i.e., a cubic interpolation function, which gives a "smooth" rotation function for given sequence of unit quaternions that represent control orientations of a 3D graphics object. "Smooth" means here the conditions that the spline passes through the control points and that in the control points two consecutive spline segments are connected smoothly, i.e., the derivatives are continuous.

Algorithm *squad* can be defined using *slerp* in a similar way as *slerp* uses unit quaternions. In *squad* for each 4 consecutive control points there are two auxiliary control points for satisfying the derivative condition. With given sequence of N unit quaternions $\{\mathrm{q}_1, \mathrm{q}_2, ..., \mathrm{q}_n, ..., \mathrm{q}_N\}$ the *squad* segments can be defined by

$$squad_n(\mathrm{q}_n, \mathrm{s}_n, \mathrm{s}_{n+1}, \mathrm{q}_{n+1}, t) = \tag{19}$$

$$slerp(\mathrm{q}_n, \mathrm{q}_{n+1}, t) . (slerp(\mathrm{q}_n, \mathrm{q}_{n+1}, t)^* . slerp(\mathrm{s}_n, \mathrm{s}_{n+1}, t))^{2t(1-t)},$$

where $n = 2, 3, ..., N-1$, and the auxiliary points are defined by

$$\mathrm{s}_n = \mathrm{q}_n . exp(-\frac{ln(\mathrm{q}_n^{-1}.\mathrm{q}_{n+1}) + ln(\mathrm{q}_n^{-1}.\mathrm{q}_{n-1})}{4}).$$

There exists also an efficient code for *slerp* based on a geometrical derivation, which does not use quaternion algebra at all. Algorithms *slerp* and *squad* can be implemented in *Mathematic*a by the following way:

```
slerp[q0_,q1_,t_] := q0.(Conjugate[q0].q1)^t;

squad[q0_,q1_,q2_,q3_,t_] := Module[{s1,s2},
    s1 = q1.Exp[-(Log[Conjugate[q1].q2] + Log[Conjugate[q1].q0])/4];
    s2 = q2.Exp[-(Log[Conjugate[q2].q3] + Log[Conjugate[q2].q1])/4];
    slerp[slerp[q1,q2,t],slerp[s1,s2,t],2t(1-t)]];
```

The correctness of *squad* can be proved by showing that *squad* function is continuously differentiable. The complete proof can be found in Dam et al. (1998) pp. 52-54. The general differentiation of unit quaternion curves are out of the scope of this paper. The reader is referred, for example, to the papers of Kim et al. or to Liefke (1998). However, we introduce here an implementation of the differential formula (see Dam et al. (1998) p.23) for a unit quaternion raised to a power of a real valued function, which can be used in the differentiation of the *slerp* function:

$$\frac{d}{dt}\hat{\mathrm{q}}^{f(t)} = f'(t)\,\hat{\mathrm{q}}^{f(t)} . ln(\hat{\mathrm{q}}). \tag{20}$$

When implementing this rule in *Mathematica* we have to prevent evaluation of the given argument. This can be done by the attribute `HoldFirst`, which specifies that the first argument of the function is to be maintained in an unevaluated form.

```
SetAttributes[quatDiff, HoldFirst];
quatDiff[q_^exp_,t_Symbol]:=D[exp, t] Dot[q^exp, Log[q]]
        /; Head[q]===quat && FreeQ[q, t];
```

The condition at the end specifies the rule to be applied only to quaternions whose components do not include the independent variable.

Finally, we visualize in Figure 2. the *slerp* and *squad* functions as the following parametric plots:

```
ParametricPlot3D[vectorPart[slerp[q1,q2,t]],{t,0,1}];
ParametricPlot3D[vectorPart[squad[q1,q2,q3,q4,t]],{t,0,1}];
```

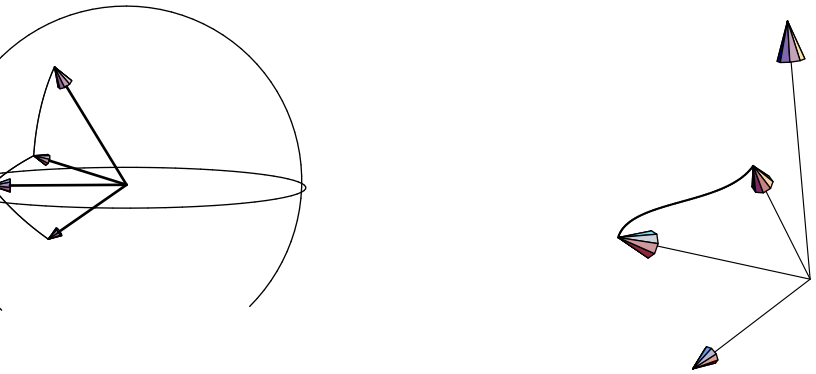

FIGURE 2. *Slerps* between unit quaternions and a *squad* segment

4. Conclusion and Future Work

We have presented a brief mathematical summary and an implementation of quaternion algebra and calculus by using the symbolic computing power of *Mathematica*. This work contributes to experimentation of quaternion technique with *Mathematica* in various areas of computer graphics, for example, in visualizing 3D curves and surfaces (see Hanson (1998)). By modifying the differentiation rule (20) it is possible to investigate more general quaternion curves which play an important role in modeling rigid body animations (see Kim et al. (1995) and Liefke (1998)).

Acknowledgements

Special thanks are due to Dr. Veikko Keränen for his cooperation during the process of programming and for editing this paper in the final phase of writing.

References

Dam, E.B., M. Koch & M. Lillholm (1998). Quaternions, Interpolation and Animation. *Technical Report DIKU-TR-98/5*. Department of Computer Science University of Copenhagen.

Dunn, F. & I. Parberry (2002). *3D Math Primer for Graphics and Game Development*. Wordware Publishing, Inc.

Hanson, A.J. (1998). Quaternion Gauss Maps and Optimal Framings of Curves and Surfaces. *Technical Report 518*. Computer Science Department of Indiana University.

Kim, M-J., M-S. Kim & S.Y. Shin (1995). A General Construction Scheme for Unit Quaternion Curves with Simple High Order Derivatives. *Computer Graphics (Annual Conference Series)* 29, 369–376.

Kim, M-J., M-S. & and S.Y. Shin (1996). A compact differential formula for the first derivative of a unit quaternion. *Journal of Visualization and Computer Animation*, 7(1), 43–57.

Liefke, H. (1998). Quaternion Calculus for Modeling Rotations in 3D Space. *http://www.cis.upenn.edu/∼li*