Dual-Quaternions

From Classical Mechanics to Computer Graphics and Beyond

Ben Kenwright
www.xbdev.net
bkenwright@xbdev.net

Abstract

This paper presents an overview of the analytical advantages of dual-quaternions and their potential in the areas of robotics, graphics, and animation. While quaternions have proven themselves as providing an unambiguous, un-cumbersome, computationally efficient method of representing rotational information, we hope after reading this paper the reader will take a parallel view on dual-quaternions. Despite the fact that the most popular method of describing rigid transforms is with homogeneous transformation matrices they can suffer from several downsides in comparison to dual-quaternions. For example, dual-quaternions offer increased computational efficiency, reduced overhead, and coordinate invariance. We also demonstrate and explain how, dual-quaternions can be used to generate constant smooth interpolation between transforms. Hence, this paper aims to provide a comprehensive step-by-step explanation of dual-quaternions, and it comprising parts (i.e., quaternions and dual-numbers) in a straightforward approach using practical real-world examples and uncomplicated implementation information. While there is a large amount of literature on the theoretical aspects of dual-quaternions there is little on the practical details. So, while giving a clear no-nonsense introduction to the theory, this paper also explains and demonstrates numerous workable aspect using real-world examples with statistical results that illustrate the power and potential of dual-quaternions.

Keywords: *dual-quaternion, transformation, blending, interpolation, quaternion, dual-number*

Introduction (Why should we use dual-quaternions?)

Dual-quaternions are a neat mathematical tool that breaks away from the norm. Probably one of their most important properties is in classical mechanics since they can represent complex problems in a unified compact way. A dual-quaternion combines the linear and rotational components together into a single variable that can be interpolated, concatenated and transformed using a single set of algebraic rules. While it has been demonstrated that quaternions are the best general solution for rotations **[1]** they can only represent half the rigid transformation. Since, a full 3D rigid transformation is composed of a translational and rotational component, which is traditionally managed as a 4x4 homogenous matrix. However, the matrix contains a great deal of overhead and is difficult to interpolate between transforms. Alternatively, the transformations can be managed using two independent components (e.g., translation vector and a quaternion). Therefore, dual-quaternions take us in a different direction and present us with a unified component that presents us with a huge number of advantages.

In a nutshell:

- ✓ They combine rotation and translation into a unified state variable
- ✓ They are a compact representation (8 scalars)
- ✓ They are easily converted to other forms (e.g., matrices)
- ✓ They can be interpolated easily without ambiguity or gimbals' lock
- ✓ They are computationally efficient (comparable with matrices and quaternions) **[2][3]**
- ✓ They can be integrated into a current system with little disruption (i.e., matrix alternative)
- ✓ They present a single invariant coordinate frame to representation rigid transforms **[4]**

Dual-quaternions are an algorithmically simple and computationally efficient approach of representing rigid transforms (i.e., rotation and translation). They are used in the same way as quaternions but provide the added advantage of encapsulating both translation and rotation into a unified state that can be concatenated and interpolated effortlessly. In fact, we believe that the reader after reading this paper will be sufficiently familiar with how dual-quaternion algebra works, and how it can be used in practical situations, to begin to appreciate the enormous potential dual-quaternions can offer, both for the graphical community but also in other areas of research.

Overview (What we need to know)

Dual-quaternions are a combination of dual-number theory and quaternion mathematics. Whereby, to have a good understanding of how we can exploit dual-quaternions to our advantage, we need to understand the basics of quaternions and dual-number theory. Hence, this paper begins by explaining the fundamental components of dual-quaternions to help establish a common ground for readers, after which, we then focus on dual-quaternions and the applicability for representing transformations both computationally and dynamically (e.g., calculating differences and interpolating).

Basically, a dual-quaternion is the concatenation of quaternion and dual-number theory (see **Figure 1**).

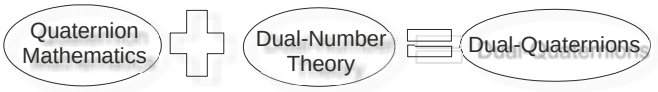

Figure 1: Dual-Quaternions Components.

To avoid confusion and enable the reader to easily distinguish a quaternion from a dual-quaternion we use two discernible symbols to identify them (see Equation **1**).

$$\begin{aligned} &Quaternion(q) \\ &Dual-Quaternion(\varsigma) \end{aligned} \qquad \mathbf{1}$$

it is essential to have a good understand of its underpinned parts work (i.e., quaternions and dual-numbers). Furthermore, once the reader understands how quaternions work, it should be trouble-free and straightforward to see how dual-quaternions operate due to their likeness.

A quaternion is represented by two fundamental parts, a scalar *real* part (w) and an *imaginary* vector part ($\mathbf{v} = x, y, z$). In practice we are only concerned with a unit-quaternion since they offer the most benefits and represent the rotation on a 4D unit-hypersphere. While the majority of people are familiar with the decomposition and principles of quaternions, there can, however, be a deficiency in the practical considerations.

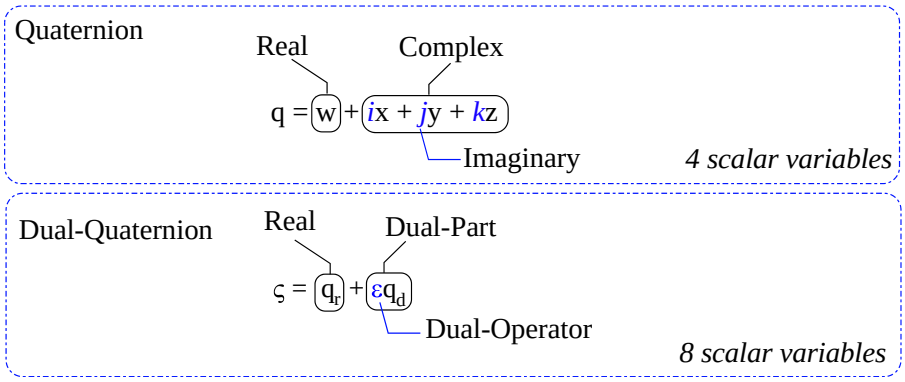

Figure 2: Visual Overview of Quaternion and Dual-Quaternion Components.

An overview of both the quaternion and dual-quaternion components is shown in **Figure 2**. While a quaternion consists of four scalar values, a dual-quaternion consists of eight scalar values. However, a quaternion can only represent rotation, while a dual-quaternion can represent both rotation and translation.

Dual-quaternions are a valuable tool that can be added to an individual's library to achieve a particular task, e.g., rigid hierarchy concatenation, interpolation, character skinning. They operate similar to existing methods (i.e., matrices) and can be transformed to and from other forms easily (i.e., quaternions, matrices) which enables them to be integrated or exchanged with little disruption into a system to gain their rewards. For a beginners introduction to dual-quaternions with an emphasis on comparison between diverse methods (e.g., matrices and Euler angles) and how to go about implementing a straightforward library we refer the reader to the paper by Kenwright [3].

Quaternion Algebra

While walking with his wife in 1843, Sir William Hamilton [5] gave birth to a revolutionary new concept that later became known as *Quaternions*. While it took some time for quaternions to be accepted, they eventually demonstrated themselves as being the most competent, memory efficient, ambiguity-free method of representing rotations. Furthermore, since quaternions are the foundation upon which dual-quaternions are built it comes as no shock, and is quite understandable, that these properties are inherited. Nevertheless, to ensure the reader is truly able to understand the potential of dual-quaternions

$$q=(w,x,y,z)=(w,\mathbf{v}) \qquad \mathbf{2}$$

The fundamental mathematical operations are defined for quaternions (i.e., addition and multiplication of quaternions and the multiplication of a quaternion by a scalar).

Quaternion from Axis-Angle

Given an angle and axis of rotation, we can construct a quaternion using Equation **3**.

$$\begin{aligned} &\hat{q} = \left(\cos\left(\frac{\theta}{2}\right),\ \hat{\mathbf{n}}\sin\left(\frac{\theta}{2}\right)\right) \\ &or \\ &q_w = \cos\left(\frac{\theta}{2}\right),\quad q_x = n_x \sin\left(\frac{\theta}{2}\right),\quad q_y = n_y \sin\left(\frac{\theta}{2}\right),\quad q_z = n_z \sin\left(\frac{\theta}{2}\right) \end{aligned} \qquad \mathbf{3}$$

where θ is the angle and $\hat{\mathbf{n}}$ is a unit-vector representing the axis of rotation.

While it is recommended that you consistently use quaternions for rotation, we can, however, rewrite Equation **3** to give us the axis-angle from the quaternion to aid in visualizing angle-axis differences as shown in Equation **4**.

$$\begin{aligned} &\theta = 2\cos^{-1}(q_w) \\ &n_x = \frac{q_x}{\sin\left(\frac{\theta}{2}\right)},\ n_y = \frac{q_y}{\sin\left(\frac{\theta}{2}\right)},\ n_z = \frac{q_z}{\sin\left(\frac{\theta}{2}\right)} \end{aligned} \qquad \mathbf{4}$$

In practice, if you do decide to convert to the axis-angle representation, you should ensure the quaternion is always a unit-quaternion and be aware of the divide by zero causality that may occur (i.e., $\sin\left(\frac{\theta}{2}\right)$ is zero).

Quaternion Vector Transformation

The quaternions transformation can be applied to a 3D vector coordinate by means of multiplication. Whereby, to transform a vector position by a quaternion we simply convert the vector to a quaternion (i.e., the imaginary part is the vector position, and the scalar real part zero) and multiply it by the quaternion transform and its conjugate, as shown in Equation **5**. Optionally, we can convert the quaternion transform to a matrix with little or no extra work for systems that operate with matrices (e.g., transforms are done on the GPU using matrices).

$$p' = \hat{q}p\hat{q}^{-1} \quad \textbf{5}$$

where

- $\hat{q}$ is a unit-quaternion representing the rotation transform
- $\hat{q}^{-1}$ is a unit-quaternion that represents the inverse of the rotation quaternion
- p is the 3D vector point in quaternion form (i.e., $p = (0, \mathbf{v})$ with $\mathbf{v} = (v_x, v_y, v_z)$)
- p' is the 3D transformed vector point in quaternion form (i.e., $p' = (0, \mathbf{v})$)

However, it is extremely important to note that for a unit-quaternion the inverse is the same as the conjugate. This is due to the mathematical and computational efficiency by which the conjugate is calculated. The conjugate of a quaternion is simply the negation of the vector component (shown in Equation **6**).

$$q^{-1} = q^{*} = (w, -\mathbf{v}) \quad \textbf{6}$$

Quaternion to Matrix

Due to the popularity of matrices, it is vital to be able to transform a quaternion to matrix form and vice-versa. A quaternion can be transformed to a matrix using little more than multiplications and additions as shown in Equation **7**.

$$\mathbf{M}_q = \begin{bmatrix} 1\text{-}2(y^2+z^2) & 2(xy+zw) & 2(xz+yw) \\ 2(xy+zw) & 1\text{-}2(x^2+z^2) & 2(yz+xw) \\ 2(xy+yw) & 2(yz+xw) & 1\text{-}2(x^2+y^2) \end{bmatrix} \quad \textbf{7}$$

where $\mathbf{M}_q$ is a matrix equivalent of the quaternion q, and x, y, z and w represent the elements of the quaternion.

Quaternion Addition

Adding two quaternions together is accomplished by simply summing the individual components together as shown in Equation **8**.

$$q_0 + q_1 = (q_{0w} + q_{1w},\ q_{0x} + q_{1x},\ q_{0y} + q_{1y},\ q_{0z} + q_{1z}) \quad \textbf{8}$$

Quaternion Multiplication

Quaternion multiplication is analogous to matrix multiplication; whereby, multiplying quaternions together is equivalent to combining their transforms. For example, when two quaternions are multiplied together it is equivalent to the first quaternion being rotated by the axis and angle of the second quaternion. However, quaternion multiplication is non-commutative (i.e., order of multiplication matters) but can be simplified by being represented using the dot and cross product (shown in Equation **9**).

$$\begin{aligned} q_0 q_1 &= (q_{w0} + \mathbf{q}_{\mathbf{v}0})(q_{w1} + \mathbf{q}_{\mathbf{v}1}) \\ &= (q_{w0}q_{w1} - \mathbf{q}_{\mathbf{v}0} \cdot \mathbf{q}_{\mathbf{v}1}) + (q_{w0}\mathbf{q}_{\mathbf{v}1} + q_{w1}\mathbf{q}_{\mathbf{v}0} + \mathbf{q}_{\mathbf{v}0} \times \mathbf{q}_{\mathbf{v}1}) \end{aligned} \quad \textbf{9}$$

where q_{w0} and q_{w1} represent the real scalar components of each quaternion and, $\mathbf{q}_{\mathbf{v}0}$ and $\mathbf{q}_{\mathbf{v}1}$ represent the vector component of each quaternion.

Quaternion Difference

Since each quaternion represents an axis-angle, then multiplying two quaternions together is equivalent to transforming one quaternion by another. Hence, it should be obvious, that we can use this to determine differences between quaternions. If both quaternions are the same, and we multiply one by the inverse of itself, it will cancel out (see Equation **10**) and give us an identity quaternion.

$$\hat{q}\hat{q}^{-1} = 1 \quad \textbf{10}$$

So if we have two quaternions, we simply multiply one by the inverse to get the difference between them (Equation **11**). It is vital to remember that the inverse of a unit-quaternion is the same as the conjugate.

$$\hat{q}_{diff} = \hat{q}_A \hat{q}_B^{-1} \quad \textbf{11}$$

For example, a simplified numerical example of the difference between two quaternions is shown in Equation **12**.

$$\begin{aligned} &A: \theta = 0,\ \hat{n} = <0,0,1> \\ &B: \theta = \pi,\ \hat{n} = <0,0,1> \\ &\hat{q}_A = \left(\cos\left(\frac{0}{2}\right),\ <0,0,1>\sin\left(\frac{0}{2}\right)\right) = <1,0,0,0> \\ &\hat{q}_B = \left(\cos\left(\frac{\pi}{2}\right),\ <0,0,1>\sin\left(\frac{\pi}{2}\right)\right) = <0,0,0,1> \\ &\hat{q}_B^{*} = \hat{q}_B^{-1} = <0,0,0,-1> \\ &\hat{q}_{diff} = \hat{q}_A \hat{q}_B^{-1} = <(1)(0) - (0,0,0)\cdot(0,0,-1), \\ &\qquad (1)(0,0,-1) + (0)(0,0,0) + (0,0,0)\times(0,0,-1)> \\ &\qquad = <0,0,0,-1> \\ &\hat{q}_{diff}: \theta = \pi,\ \hat{n} = <0,0,-1> \end{aligned} \quad \textbf{12}$$

To help visualize the result for the example in Equation **12**, imagine comparing the difference between two scalar numbers A and B (e.g., 0 and n). Then the difference, A-B

= (0 - n) = -n, which is analogous to what we calculated in the example.

Quaternion Spherical Linear Interpolation (SLERP)

Quaternion spherical linear interpolation is the transformation along the surface of the 4D unit-hypersphere.

Starting with the well known exponential function from complex numbers it can be shown that in Equation **13**.

$$e^{i\theta} = \cos\theta + i\sin\theta \qquad \mathbf{13}$$

Then we can equate our quaternion and represent it as an exponential given by Equation **14**.

$$q = e^{\mathbf{v}\Omega} = \cos\Omega + \mathbf{v}\sin\Omega \qquad \mathbf{14}$$

where $\Omega = \frac{\theta}{2}$ and $\mathbf{v}$ is a unit vector (noting that $\mathbf{v}^2 = -1$).

We can then write the quaternion in the form (Equation **15**):

$$q^t = \cos(t\Omega) + \hat{\mathbf{v}}\sin(t\Omega) \qquad \mathbf{15}$$

Then the Slerp expression is given by Equation **16**.

$$SLERP(q_0, q_1 : t) = q_0(q_0^{-1}q_1)^t \qquad \mathbf{16}$$

For example, let us consider two very simple cases when t=0 and t=1

$$t = 0$$
$$q^t = \cos(t\Omega) + \mathbf{v}\sin(t\Omega)$$
$$q^0 = \cos(0) + \mathbf{v}\sin(0)$$
$$q^0 = 1$$
$$q_0(q_0^{-1}q_1)^t = q_0(q_0^{-1}q_1)^0 = q_0(1) = q_0$$

and

$$t = 1$$
$$q^t = \cos(t\Omega) + \mathbf{v}\sin(t\Omega)$$
$$q^1 = \cos(\Omega) + \mathbf{v}\sin(\Omega)$$
$$q_0(q_0^{-1}q_1)^t = q_0(q_0^{-1}q_1)^1 = (q_0q_0^{-1})q_1 = (1)q_1 = q_1$$

An alternative, and more popular, representation of Equation **16** can be calculated using a geometric approach and is shown in Equation **17** (for a more detailed description see Shoemake **[1]**).

$$SLERP(q_0, q_1 : t) = \frac{\sin\left((1-t)\theta\right)}{\sin(\theta)}q_0 + \frac{\sin(t\theta)}{\sin(\theta)}q_1 \qquad \mathbf{17}$$

Dual-Number Theory

Clifford [6] published his intriguing work on dual-numbers in 1873, and provided us with a powerful tool for facilitating the analysis of complex systems (e.g., mechanical, geometric). In fact, it was not long before they found a place in the movement of rigid bodies [7][8] and later in geometry [9]. The relevant formalism that was developed and what we primarily make use of in this paper is the screw calculus that allows the unification of translation and rotation.

The definition and properties of a dual-number are given in Equation **18**. Dual-numbers are akin to complex numbers. However, whereas complex numbers have a real-part and an imaginary-part and dual-numbers have a real-part and a dual-part.

$$z = r + \varepsilon d \quad with \quad \varepsilon^2 = 0 \quad but \quad \varepsilon \neq 0 \qquad \mathbf{18}$$

where ε is known as the dual-operator, r is the real-part and, d the dual-part.

Dual-Number Addition

$$(r_A + \varepsilon d_A) + (r_B + \varepsilon d_B) = (r_A + r_B) + \varepsilon(d_A + d_B) \qquad \mathbf{19}$$

Dual-Number Multiplication

$$\begin{aligned}(r_A + \varepsilon d_A)(r_B + \varepsilon d_B) &= r_A r_B + \varepsilon r_A d_B + \varepsilon r_B d_A + \varepsilon^2 d_A d_B \\ &= r_A r_B + \varepsilon(r_A d_B + r_B d_A) \qquad (remember\ \varepsilon^2 = 0)\end{aligned} \qquad \mathbf{20}$$

Dual-Number Division

$$\begin{aligned}\frac{(r_A + \varepsilon d_A)}{(r_B + \varepsilon d_B)} &= \frac{(r_A + \varepsilon d_A)}{(r_B + \varepsilon d_B)}\frac{(r_B - \varepsilon d_B)}{(r_B - \varepsilon d_B)} \\ &= \frac{r_A r_B + (r_B d_A - r_A d_B)\varepsilon}{(r_B)^2} \\ &= \frac{r_A r_B}{r_B^{\,2}} + \frac{r_B d_A - r_A d_B}{r_B^{\,2}}\varepsilon\end{aligned} \qquad \mathbf{21}$$

Dual-Number Differentiation

From elementary calculus principles shown in Equation **22**.

$$\frac{d}{dx}\mathbf{s}(x) = \lim_{\delta x \to 0}\frac{\mathbf{s}(x + \delta x) - \mathbf{s}(x)}{\delta x} \qquad \mathbf{22}$$

We use Taylor series to find the differentiable (Equation **23**).

$$\begin{aligned}f(r_A + d_A\varepsilon) &= f(r_A) + \frac{f'(r_A)}{1!}d_A\varepsilon + \frac{f''(r_A)}{2!}(d_A\varepsilon)^2 + \frac{f'''(r_A)}{3!}(d_A\varepsilon)^3 + \ldots \\ &= f(r_A) + \frac{f'(r_A)}{1!}d_A\varepsilon + 0 + 0 + \ldots. \qquad (as,\ \varepsilon^2 = 0) \\ &= f(r_A) + f'(r_A)d_A\varepsilon\end{aligned} \qquad \mathbf{23}$$

Remarkably, due to the condition $\varepsilon^2 = 0$, we end up with an extremely elegant solution.

For a more in-depth explanation of the rationale behind dual-number theory see Keler [10] or Pennestr et al [11].

Dual-Quaternion Algebra

The dual-quaternion is an extension of dual-number theory whereby the numbers for the dual-number equation are represented by quaternions. Remarkably, the dual-quaternion algebra that results is very straightforward and elegant and provides us an algebraically compact and

efficient system for solving otherwise complex problems. For example, we can represent a rigid transforms with eight scalar variables; we can combine transforms effortlessly through concatenation, and we are able to produce smooth constant interpolation between rigid transformations. As shown in Figure 2, the dual-quaternion is decomposed into two parts the real part and the dual-part.

Dual-Quaternion Identity

The identity of a dual-quaternion is shown in Equation **24** and is analogous to a quaternion identity. Therefore, any dual-quaternion that is multiplied with an identity dual-quaternion remains unchanged. To define an identity dual-quaternion we set the first scalar value to 1 and the other seven scalar values are all 0.

$$\varsigma = [1,0,0,0][0,0,0,0] \quad \textbf{24}$$

Dual-Quaternion from Position and Rotation

To construct a unit-dual quaternion from a rotation and a translation we use Equation **25**. We construct the dual-quaternion from a pair of quaternions that represent the rotation and translation.

$$\begin{aligned} \varsigma &= q_r + q_d \\ q_r &= \mathbf{r} \\ q_d &= \frac{1}{2}\mathbf{t}\,\mathbf{r} \end{aligned} \quad \textbf{25}$$

where $\mathbf{r}$ is a unit quaternion representing the rotation and $\mathbf{t}$ is a quaternion describing the translation. The individual elements of the two quaternions from Equation **25** are shown in Equation **26**.

$$\begin{aligned} \mathbf{r} &= [\cos(\frac{\theta}{2}),\ \mathbf{n}_x \sin(\frac{\theta}{2}),\ \mathbf{n}_y \sin(\frac{\theta}{2}),\ \mathbf{n}_z \sin(\frac{\theta}{2})] \\ \mathbf{t} &= [0, \mathbf{t}_x, \mathbf{t}_y, \mathbf{t}_z] \end{aligned} \quad \textbf{26}$$

where $\mathbf{n}$ is the axis of rotation, θ is the angle of rotation, and $\mathbf{t}_x, \mathbf{t}_y, \mathbf{t}_z$ is the position in Cartesian coordinates.

For example, if we want to construct a dual-quaternion that only has a rotation we have:

$$\varsigma = [\cos(\frac{\theta}{2}),\ \mathbf{n}_x \sin(\frac{\theta}{2}),\ \mathbf{n}_y \sin(\frac{\theta}{2}),\ \mathbf{n}_z \sin(\frac{\theta}{2})][0,0,0,0] \quad \textbf{27}$$

and, if we want to construct a dual-quaternion that only has a translation we have:

$$\varsigma = [1,0,0,0][0, \frac{\mathbf{t}_x}{2}, \frac{\mathbf{t}_y}{2}, \frac{\mathbf{t}_z}{2}] \quad \textbf{28}$$

Comparable to matrices and quaternions we can concatenate dual-quaternion transformations using multiplication. Hence, you can create a pure rotation dual-quaternion and a pure translation dual-quaternion and multiply them together to form a combined dual-quaternion that possesses both the translation and rotation components; however, be aware that the multiplication order is important.

Dual-Quaternion to Position and Rotation

We can extract the position and rotation from a dual-quaternion. In reverse to Equation **25** that created a dual-quaternion from a position and rotation, we conversely extract the position and rotation using Equation **29**.

$$\begin{aligned} \varsigma &= q_r + q_d \\ \mathbf{r} &= q_r \\ \mathbf{t} &= 2\, q_d\, q_r^* \end{aligned} \quad \textbf{29}$$

Dual-Quaternion Addition

The addition of dual-quaternions is one of the simplest operations since we only need to add each individual component together (see Equation **30**).

$$\begin{aligned} \varsigma_A &= (a_0 + a_1 i + a_2 j + a_3 k) + (a_4 + a_5 i + a_6 j + a_7 k)\varepsilon \\ \varsigma_B &= (b_0 + b_1 i + b_2 j + b_3 k) + (b_4 + b_5 i + b_6 j + b_7 k)\varepsilon \\ \varsigma_A + \varsigma_B &= ((a_0 + b_0) + (a_1 + b_1) + (a_2 + b_2) + (a_3 + b_3)) + \\ &\quad ((a_4 + b_4) + (a_5 + b_5) + (a_6 + b_6) + (a_7 + b_7))\varepsilon \end{aligned} \quad \textbf{30}$$

Dual-Quaternion Multiplication

Due to dual-numbers requiring $\varepsilon^2 = 0$ results in the multiplication of dual-quaternions being a very neat and tidy operation (see Equation **31**). Hence, the resulting dual-quaternion multiplication can be broken down into three quaternion multiplications and a quaternion addition operation.

$$\begin{aligned} \varsigma_A &= q_0 + q_1\varepsilon \\ \varsigma_B &= q_2 + q_3\varepsilon \\ \varsigma_A + \varsigma_B &= (q_0 + q_1\varepsilon)(q_2 + q_3\varepsilon) \\ &= q_0 q_2 + (q_0 q_3 + q_1 q_2)\varepsilon \end{aligned} \quad \textbf{31}$$

Dual-Quaternion Conjugate

The dual-quaternion conjugate is essentially an extension of the quaternion conjugate, and is given by Equation **32**.

$$\varsigma^* = q^* + \varepsilon q^* \quad \textbf{32}$$

Dual-Quaternion Magnitude

A dual-quaternion multiplied by its conjugate gives the magnitude squared and hence the square root of this is the scalar magnitude length (see Equation **33**).

$$\| \varsigma \| = \sqrt{\varsigma\varsigma^*} \quad \textbf{33}$$

It is crucial to note that a unit dual-quaternion has a magnitude of 1. Hence, we can say that the magnitude of a unit dual-quaternion multiplied by its conjugate must equal 1.

$$\| \hat{\varsigma} \| = \| \hat{\varsigma}\hat{\varsigma}^* \| = 1 \quad \textbf{34}$$

Dual-Quaternion Vector Transformation

Equivalent to a quaternion a dual-quaternion can transform a 3D vector coordinate as shown in Equation **34**. Note that for a unit-quaternion the inverse is the same as the conjugate.

$$p' = \hat{\varsigma} p \hat{\varsigma}^{-1} \quad \textbf{35}$$

where

- ς is a dual-quaternion representing the transform
- ς^{-1} is a dual-quaternion that is the inverse of the dual-quaternion ς
- p is a dual-quaternion representing the rigid transform (e.g., 3D vector point $p = (1,0,0,0) + \varepsilon(0, v_x, v_y v_z)$)
- p' is a dual-quaternion with the resulting transform.

Plücker Coordinates

Plücker coordinates [12] are used to create Screw coordinates which are an essential technique of representing lines. We need the Screw coordinates so that we can re-write dual-quaternions in a more elegant form to aid us in formulating a neater and less complex interpolation method that is comparable with spherical linear interpolation for classical quaternions.

The Definition of Plücker Coordinates:

1. $\mathbf{p}$ is a point anywhere on a given line
2. $\vec{\mathbf{l}}$ is the direction vector
3. $\vec{\mathbf{m}} = \mathbf{p} \times \vec{\mathbf{l}}$ is the moment vector
4. $(\vec{\mathbf{l}}, \vec{\mathbf{m}})$ are the six Plücker coordinate

We can convert the eight dual-quaternions parameters to an equivalent set of eight screw coordinates and vice-versa. The definition of the parameters are given in Equation **36**.

$$\begin{aligned} &screw\ parameters = (\theta, d, \vec{\mathbf{l}}, \vec{\mathbf{m}}) \\ &dual-quaternion = q_r + \varepsilon q_d \\ &\qquad = (w_r + \mathbf{v}_r) + \varepsilon(w_d + \mathbf{v}_d) \end{aligned} \quad \textbf{36}$$

where in addition to $\vec{\mathbf{l}}$ representing the vector line direction and $\vec{\mathbf{m}}$ the line moment, we also have d representing the translation along the axis (i.e., pitch) and the angle of rotation θ.

Converting to and from a dual-quaternion and its screw parameters is shown in Equation **37** and Equation **38** (see Daniilidis [13] for details).

$$\begin{aligned} &dual-quaternion \rightarrow screw\ parameters \\ &\theta = 2\cos^{-1}(w_r) \\ &d = -2w_d \frac{1}{\sqrt{\mathbf{v}_r \bullet \mathbf{v}_r}} \\ &\vec{\mathbf{l}} = \mathbf{v}_r \left(\frac{1}{\sqrt{\mathbf{v}_r \bullet \mathbf{v}_r}} \right) \\ &\vec{\mathbf{m}} = \left(\mathbf{v}_d - \vec{\mathbf{l}} \frac{d\, w_r}{2} \right) \frac{1}{\sqrt{\mathbf{v}_r \bullet \mathbf{v}_r}} \end{aligned} \quad \textbf{37}$$

and

$$\begin{aligned} &screw\ parameters \rightarrow dual-quaternion \\ &w_r = \cos\left(\frac{\theta}{2}\right) \\ &\mathbf{v}_r = \vec{\mathbf{l}} \sin\left(\frac{\theta}{2}\right) \\ &w_d = -\frac{d}{2}\sin\left(\frac{\theta}{2}\right) \\ &\mathbf{v}_d = \sin\left(\frac{\theta}{2}\right)\vec{\mathbf{m}} + \frac{d}{2}\cos\left(\frac{\theta}{2}\right)\vec{\mathbf{l}} \end{aligned} \quad \textbf{38}$$

Dual-Quaternion Power

We can write the dual-quaternion representation in the form given in Equation **39** (see Daniilidis [14] for details).

$$\begin{aligned} \hat{\varsigma} &= \cos\left(\frac{\theta + \varepsilon d}{2}\right) + (\vec{\mathbf{l}} + \varepsilon\vec{\mathbf{m}}) \sin\left(\frac{\theta + \varepsilon d}{2}\right) \\ &= \cos\left(\frac{\hat{\theta}}{2}\right) + \hat{\mathbf{v}} \sin\left(\frac{\hat{\theta}}{2}\right) \end{aligned} \quad \textbf{39}$$

where

- $\hat{\varsigma}$ is a unit dual-quaternion
- $\hat{\mathbf{v}}$ is a unit dual-vector $\hat{\mathbf{v}} = \vec{\mathbf{l}} + \varepsilon\vec{\mathbf{m}}$
- $\hat{\theta}$ is a dual-angle $\hat{\theta} = \theta + \varepsilon d$

The dual-quaternion in this form is exceptionally interesting and valuable as it allows us to calculate a dual-quaternion to a power. Calculating a dual-quaternion to a power is essential for us to be able to easily calculate spherical linear interpolation. However, instead of purely rotation as with classical quaternions, we are instead now able to interpolate full rigid transformations (i.e., rotation and translation) by using dual-quaternions.

$$\hat{\varsigma}^t = \cos\left(t\frac{\hat{\theta}}{2}\right) + \hat{\mathbf{v}} \sin\left(t\frac{\hat{\theta}}{2}\right) \quad \textbf{40}$$

Dual-Quaternion Screw Linear Interpolation (ScLERP)

ScLERP is an extension of the quaternion SLERP technique, and allows us to create constant smooth interpolation between dual-quaternions. Similar to

quaternion SLERP we use the power function to calculate the interpolation values for ScLERP shown in Equation **41**.

$$ScLERP(\hat{\varsigma}_A, \hat{\varsigma}_B : t) = \hat{\varsigma}_A(\hat{\varsigma}_A^{-1}\hat{\varsigma}_B)^t \quad \textbf{41}$$

where $\hat{\varsigma}_A$ and $\hat{\varsigma}_B$ are the start and end unit dual-quaternion and t is the interpolation amount from 0.0 to 1.0.

The implementation of ScLERP involves first using Equation **37** to convert the dual-quaternion parameters to screw parameters, so we can calculate the power function with Equation **40.** Afterwards, we use Equation **38** to convert back to a dual-quaternion to complete the calculation and give the resulting interpolated result.

$$\begin{aligned} screw\ parameters &= (\theta, d, \vec{\mathbf{l}}, \vec{\mathbf{m}}) \\ dual-quaternion &= q_r + \varepsilon q_d \\ &= (w_r + \mathbf{v}_r) + \varepsilon(w_d + \mathbf{v}_d) \end{aligned} \quad \textbf{42}$$

Basic Un-Optimized Implementation Steps of ScLERP (for Equation **41**):

1. Calculate Inverse of A (i.e., Conjugate of A)
2. Multiply Inv(A) and B
3. Calculate Screw Parameters for result Inv(A)B
4. Calculate to the power of
5. Convert screw parameters form back to the classical dual-quaternion form
6. Multiply with A to get the answer

(For example, see **Listing 1** for a practical implementation example).

Alternatively, a fast approximate alternative to ScLERP was presented by Kavan et al. [15] called Dual-Quaternion Linear Blending (DLB). Furthermore, dual-quaternions have gained a great deal of attention in the area of character-based skinning. Since, a skinned surface approximation using a weighted dual-quaternion approach produces less kinking and reduced visual anomalies compared to linear methods by ensuring the surface keeps its volume (for example, see **Figure 3**).

Dual-quaternions eliminate skin collapsing artefacts and while they are slightly slower than the linear blended skinning method they are, however, graphical processor unit (GPU) friendly. Furthermore, they are simple to integrate into a 3D engine and cause very little disruption since the same rigging as standard linear blending skinning can be used.

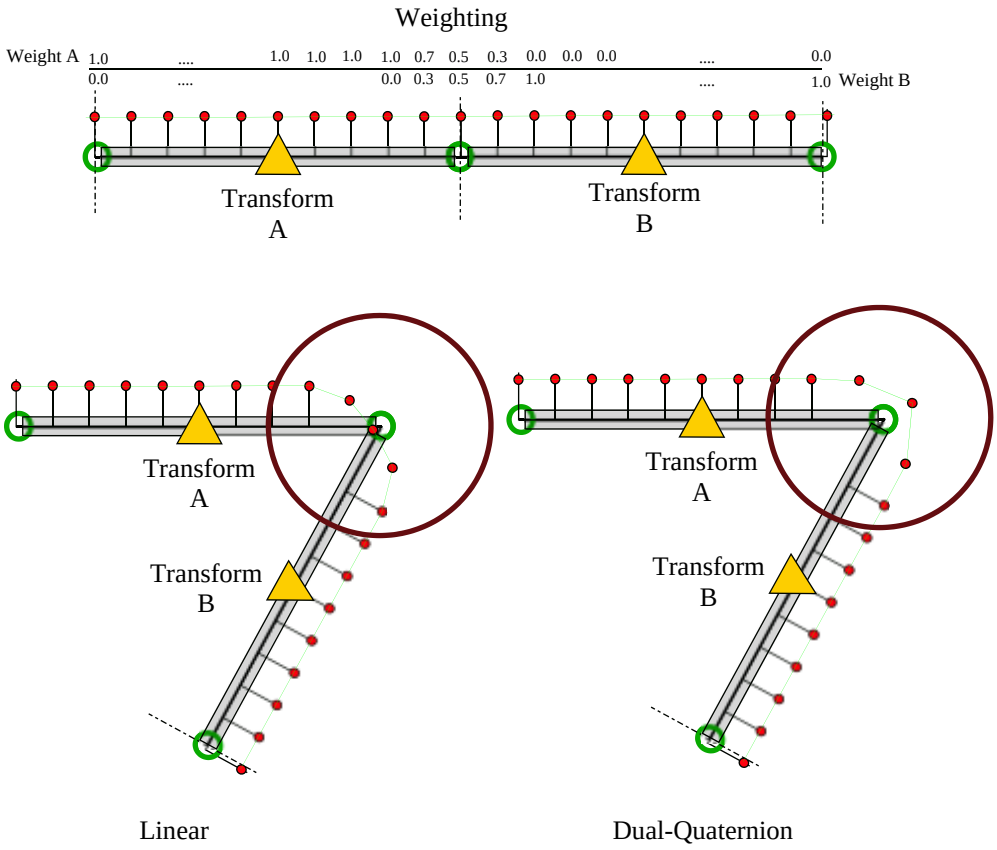

Figure 3: Visual comparison between linear and dual-quaternion weighting for vertex skinning.

Interpolation

In general, one of the greatest advantages of using quaternions and dual-quaternions over any other method is their ability to interpolate smoothly between transforms. Naively, two values can represent the start and end, and a scalar constant represents the interpolation amount (scalar ratio is from 0.0 to 1.0). For a straight-line vector we can treat each component separately and use a parametric equation shown in Equation **43**. This has the added advantage of being computationally fast and simple.

$$LERP(a,b:t) = b-(a-b)t \quad \textbf{43}$$

where a and b represent the start and end value and t the in-between ratio.

In fact, for small changes we can use Equation **43** to interpolate between quaternions and dual-quaternions. However, as the quaternion and dual-quaternion become more dissimilar there is a greater error and the intermediate steps become less smooth and less correct. The intermediate steps between the start and end do not represent a unit-quaternion rotation or dual-quaternion rotation. Hence, we need to re-normalize the value at each step to ensure it falls on the unit-hypersphere. Most importantly, though, is that the interpolation rate is not constant. We can reduce the error and make the linear interpolation approximation more tolerable by normalizing the values between steps. This is known as Normalized Linear Interpolation (NLERP) and has the added advantage of ensuring that the intermediate values are always of unit-length (see Equation **44**). Again, it should be stressed that the linear interpolation approximation is only suitable for small changes.

$$NLERP(q_A,q_B:t) = \frac{q_A+(q_B-q_A)t}{\| q_A+(q_B-q_A)t \|} \quad \textbf{44}$$

where a is the start, b is the end and t is the interpolation amount (i.e., 0.0 to 1.0).

While it has numerous problems for both quaternions and dual-quaternions, it is computationally fast and easy to implement and can, however, give reasonably good approximations for small interpolations. The trouble is, quaternions and dual-quaternions do not travel along straight-line trajectories. However, we can use an alternative interpolation method that follows the unit-hypersphere sphere. This is accomplished by interpolation along the unit-hypersphere to produce a constant and smooth rate of change. Dual-quaternions can use the exponential representation similar to quaternions to generate an interpolation scheme to produce constant smooth interpolation.

Shortest or Longest Interpolation Path

Contrary to popular belief, a quaternion and dual-quaternion by default will not take the shortest path between points when interpolated. This is because a quaternion can represent the same orientation using two different representations, and consequently a dual-quaternion. This means that both quaternions and dual-quaternions do not offer a unique representation of an orientation or transformation (i.e., there are two). The difference between the two representation becomes apparent during interpolation and provide a method for determining the shortest or longest path to be taken during interpolation.

The interpolation direction can be calculated by examining the angle between the two transforms. If the angle between the two quaternions (or dual-quaternions) is greater than $\frac{\pi}{2}$ then the interpolation will take the "longest path". We can detected easily in practice by taking the dot produce of the two quaternions (for a dual-quaternion we use the quaternion for the rotation). If the dot product is less than zero then the longest path will be taken. However, if we want to prevent the longest path from being taken we simply negate all the elements for the quaternion or dual-quaternion before interpolating. Likewise, if we desire the longest path we can check that the dot product is greater than zero before negating the quaternion or dual-quaternion.

Catmull-Rom Spline-Based Interpolation

For irregular spaced key-frame data, we can exploit the Catmull-Rom spline-based vector interpolation function and dual-quaternions algebra as a method for generating a unified, smooth, continuous trajectory path.

Eradication of the Square Root

We can optimize some operations by eradicating the square root overhead. Since both quaternions and dual-quaternions are normalized the same way as vectors (see Equation **45**), we can identify cases whereby an element is multiplication with another element to cancel out the square root.

$$\hat{q} = \frac{q}{\| q \|} = \frac{q}{\sqrt{q \cdot q}} \quad \textbf{45}$$

However, the multiplication of two quaternion elements results in the square root being redundant. For example, when we construct a matrix from a quaternion (as shown in Equation **7**) we multiply pairs of elements. This can be used to cancel out the necessity to normalize the result as shown in Equation **46**.

$$\hat{q}_x \hat{q}_y = \frac{q_x}{\sqrt{q \cdot q}} \frac{q_y}{\sqrt{q \cdot q}} = \frac{q_x q_y}{q \cdot q} \quad \textbf{46}$$

Performance Comparison

It can be shown without difficulty that in general a dual-quaternion takes less operations to compute a general transform concatenation compared to a matrix (see **Table 1**).

Matrix4x4	: 64mult + 48adds
Matrix4x3	: 48mult + 32adds
DualQuaternion	: 42mult + 38adds

Table 1: Computational cost of combining matrices and dual-quaternions.

Furthermore, for rigid skeletal animations, the computation of world space transforms in addition to the overhead cost of transferring the data to the graphics processing unit (GPU) can be noticeably better. For example, to transfer the transforms to the GPU each frame a dual-quaternions requires only eight floats compared to a 3x4 matrix that requires twelve per joint.

Inverse Kinematics

The conventional method for representing and concatenating links together in hierarchical systems is the Denavit-Hartenberg **[16]** matrix convention, and while Wang and Ravani **[17]** proposed an alternative more efficient forward recursion method for kinematic equations, we propose using dual-quaternions, since they offer an analogous alternative that is numerically stable and computationally efficient. Dual-quaternions have shown promising results for providing singularity-free solutions for inverse kinematic (IK) problems with nonlinearities [18]. It is clearly an advantage to use dual-quaternions for rigid hierarchies since each dual-quaternion can be concatenated easily, interpolated smoothly and provide rigid transform comparisons effortlessly.

Porting to Dual-Quaternion

Converting an exiting matrix scheme to a dual-quaternion system is straightforward since much of the operations (i.e., concatenation of transforms) are done the same way. For example, the concatenation of transforms with a matrices and dual-quaternions:

Matrix

$$M_{03} = M_0 M_1 M_2 M_3$$

Dual-Quaternion

$$\varsigma_{03} = \varsigma_0 \varsigma_1 \varsigma_2 \varsigma_3$$

where the subscript represents the transform, while matrix transform M_0 corresponds the dual-quaternion transform ς_0. However, unlike matrices, dual-quaternions provide an additional repertoire of valuable functions to easily compare and interpolate between transforms.

Conclusion and Final Thoughts

This paper has attempted to introduction the reader to the practical potential of dual-quaternions and their advantages in solving kinematic problems (i.e., systems with rotational and translational properties). The fundamental features and workings of dual-quaternions have been outlined. It has also been shown, that in general, they provide a compact and efficient tool for representing rigid transformation (i.e., *simultaneously* rotation and translation).

In practicality, a dual-quaternion is a tool like any other tool to be used to solve a problem. It is a novel and fresh alternative to the de-facto method of matrices with numerous benefits that can be integrated into a system with little disruption or complication. It is hoped that the reader after reading this paper will go forwards and implement a straightforward dual-quaternion class to enable them to explore the potential and decide for themselves if they are the right tool for the job.

References

[1] K. Shoemake, "Animating rotation with quaternion curves," In Proceedings of the 12th annual conference on Computer graphics and interactive techniques. ACM Press, pp. 245–254, 1985.

[2] M. Schilling, "Universally manipulable body models—dual quaternion representations in layered and dynamic MMCs," Autonomous Robots, vol. 30, no. 4, pp. 399–425, 2011.

[3] B. Kenwright, "A Beginners Guide to Dual-Quaternions: What They Are , How They Work, and How to Use Them for 3D Character Hierarchies," The 20th International Conference on Computer Graphics, Visualization and Computer Vision, no. June 26–28, pp. 1–10, 2012.

[4] Q. Ge, A. Varshney, J. P. Menon, and C. F. Chang, "Double quaternions for motion interpolation," in Proceedings of the ASME Design Engineering Technical Conference, 1998.

[5] S. W. R. Hamilton, "On quaternions; or on a new system of imaginaries in algebra," Philosophical Magazine and Journal of Science, no. July, pp. 10–13, 1844.

[6] W. Clifford, Mathematical Papers. London, Macmillan, 1882.

[7] A. P. Kotelnikov, "Screw calculus and some of its applications in geometry and mechanics," Kazan (in Russia), 1895.

[8] Leipzig, "Geometrie der Dynamen," E. Study, 1903.

[9] I. M. Yaglom, "A simple non-Euclidean geometry and its physical basis," Springer Verlag, vol. New York, 1979.

[10] M. L. Keler, "On the theory of screws and the dual method," In Proceedings of A Symposium Commemorating the Legacy, Works, and Life of Sir Robert Stawell Ball Upon the 100th Anniversary of "A Treatise on the Theory of Screws," vol. July 9–11, 2000.

[11] E. Pennestr and R. Stefanelli, "Linear Algebra and Numerical Algorithms Using Dual," Multibody System Dynamics, vol. 18, no. 3, pp. 323–344, 2007.

[12] J. Plùcker, "On a new geometry of space," Philosophical Transactions of the Royal Society of London, vol. 155, no. 1865, pp. 725–791, 1865.

[13] K. Daniilidis, "Hand-Eye Calibration Using Dual Quaternions," The International Journal of Robotics Research, vol. 18, no. 3, pp. 286–298, Mar. 1999.

[14] K. Daniilidis and B.-C. Eduardo, "The dual quaternion approach to hand-eye calibration," Proceedings of the 13th International Conference on Pattern Recognition, vol. 1, pp. 318–322, 1996.

[15] L. Kavan, S. Collins, J. Žára, and C. O'Sullivan, "Skinning with dual quaternions," In 2007 ACM SIGGRAPH symposium on interactive 3D graphics and games, vol. ACM Press, no. April/May, pp. 39–46, 2007.

[16] J. Denavit and R. S. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," Journal of Applied Mechanics, vol. 22, no. June, pp. 215–221, 1955.

[17] L. T. Wang and B. Ravani, "Recursive computations of kinematic and dynamic equations for mechanical manipulators," IEEE Journal of Robotics and Automation, vol. September, no. 3, pp. 124–131, 1985.

[18] Y. Aydın and S. Kucuk, "Quaternion Based Inverse Kinematics for Industrial Robot Manipulators with Euler Wrist," IEEE International Conference on Mechatronics, vol. July 3–5, pp. 581–586, 2006.

Appendix

Sample Dual-Quaternion Class Implementation

```
public class DualQuaternion_c
{
public Quaternion   m_real;
public Quaternion   m_dual;

public static readonly DualQuaternion_c Identity = new DualQuaternion_c();

public DualQuaternion_c()
{
  m_real = new Quaternion(0,0,0,1);
  m_dual = new Quaternion(0,0,0,0);
}
public DualQuaternion_c( Quaternion r, Quaternion d )
{
  m_real = Quaternion.Normalize( r );
  m_dual = d;
}
public DualQuaternion_c( Quaternion r, Vector3 t )
{
  m_real     = Quaternion.Normalize( r );
  m_dual  = ( new Quaternion( t, 0 ) * m_real ) * 0.5f;
}
public static float Dot( DualQuaternion_c a, DualQuaternion_c b )
{
  return Quaternion.Dot( a.m_real, b.m_real );
}
public static DualQuaternion_c operator* (DualQuaternion_c q, float scale)
{
  DualQuaternion_c ret = q;
  ret.m_real *= scale;
  ret.m_dual *= scale;
  return ret;
}
public static DualQuaternion_c Normalize( DualQuaternion_c q )
{
  float mag = Quaternion.Dot( q.m_real, q.m_real );
  Debug_c.Assert( mag > 0.000001f );
  DualQuaternion_c ret = q;
  ret.m_real *= 1.0f / mag;
  ret.m_dual *= 1.0f / mag;
  return ret;
}
public static DualQuaternion_c operator +(DualQuaternion_c lhs, DualQuaternion_c rhs)
{
  return new DualQuaternion_c(lhs.m_real + rhs.m_real, lhs.m_dual + rhs.m_dual);
}
// Multiplication order - left to right
public static DualQuaternion_c operator *(DualQuaternion_c lhs, DualQuaternion_c rhs)
{
  lhs = DualQuaternion_c.Normalize( lhs );
  rhs = DualQuaternion_c.Normalize( rhs );

  return new DualQuaternion_c( rhs.m_real * lhs.m_real,
                rhs.m_dual * lhs.m_real + rhs.m_real * lhs.m_dual);
}
public static DualQuaternion_c Conjugate( DualQuaternion_c q )
{
  return new DualQuaternion_c( Quaternion.Conjugate( q.m_real ),
                               Quaternion.Conjugate( q.m_dual ) );
}
public static Quaternion GetRotation( DualQuaternion_c q )
{
  return q.m_real;
}
public static Vector3 GetTranslation( DualQuaternion_c q )
{
  Quaternion t = ( q.m_dual * 2.0f ) * Quaternion.Conjugate( q.m_real );
  return new Vector3( t.X, t.Y, t.Z );
}
public static Matrix DualQuaternionToMatrix( DualQuaternion_c q )
{
  q = DualQuaternion_c.Normalize( q );

  Matrix M = Matrix.Identity;
  float w  = q.m_real.W;
  float x  = q.m_real.X;
  float y  = q.m_real.Y;
  float z  = q.m_real.Z;

  // Extract rotational information
  M.M11 = w*w + x*x - y*y - z*z;
```

```
  M.M12 = 2*x*y + 2*w*z;
  M.M13 = 2*x*z - 2*w*y;

  M.M21 = 2*x*y - 2*w*z;
  M.M22 = w*w + y*y - x*x - z*z;
  M.M23 = 2*y*z + 2*w*x;

  M.M31 = 2*x*z + 2*w*y;
  M.M32 = 2*y*z - 2*w*x;
  M.M33 = w*w + z*z - x*x - y*y;

  // Extract translation information
  Quaternion t = ( q.m_dual ) * Quaternion.Conjugate( q.m_real ) * 2.0f;
  M.M41 = t.X;
  M.M42 = t.Y;
  M.M43 = t.Z;
  return M;
}

public static
DualQuaternion_c ScLERP( DualQuaternion_c from, DualQuaternion_c to, float t )
{
  // Shortest path
  float dot = Quaternion.Dot(from.m_real, to.m_real);
  if ( dot < 0 ) to = to * -1.0f;

  // ScLERP = qa(qa^-1 qb)^t
  DualQuaternion_c diff = DualQuaternion_c.Conjugate(from) * to;

  Vector3 vr    = new Vector3(diff.m_real.X, diff.m_real.Y, diff.m_real.Z);
  Vector3 vd    = new Vector3(diff.m_dual.X, diff.m_dual.Y, diff.m_dual.Z);
  float   invr  = 1 / (float)Math.Sqrt( Vector3.Dot(vr, vr) );

  // Screw parameters
  float angle       =  2 * (float)Math.Acos( diff.m_real.W );
  float pitch       = -2 * diff.m_dual.W * invr;
  Vector3 direction =  vr * invr;
  Vector3 moment    =  (vd - direction*pitch*diff.m_real.W*0.5f)*invr;

  // Exponential power
  angle *= t;
  pitch *= t;

  // Convert back to dual-quaternion
  float sinAngle = Sin(0.5f*angle);
  float cosAngle = Cos(0.5f*angle);
  Quaternion real = new Quaternion( direction* sinAngle,
                                    cosAngle );
  Quaternion dual = new Quaternion( sinAngle*moment+pitch*0.5f* cosAngle *direction,
                                    -pitch*0.5f*sinAngle );

  // Complete the multiplication and return the interpolated value
  return from * new DualQuaternion_c( real, dual );
}

#if false
public static void SimpleTest()
{
  DualQuaternion_c dq0 = new DualQuaternion_c( Quaternion.CreateFromYawPitchRoll(1,2,3),
                                               new Vector3(10,30,90) );
  DualQuaternion_c dq1 = new DualQuaternion_c( Quaternion.CreateFromYawPitchRoll(-1,3,2),
                                               new Vector3(30,40,190) );
  DualQuaternion_c dq2 = new DualQuaternion_c( Quaternion.CreateFromYawPitchRoll(2,3,1.5f),
                                               new Vector3(5,20,66) );
  DualQuaternion_c dq = dq0 * dq1 * dq2;

  Matrix dqToMatrix = DualQuaternion_c.DualQuaternionToMatrix( dq );

  Matrix m0 = Matrix.CreateFromYawPitchRoll(1,2,3) * Matrix.CreateTranslation(10,30,90);
  Matrix m1 = Matrix.CreateFromYawPitchRoll(-1,3,2) * Matrix.CreateTranslation(30,40,190);
  Matrix m2 = Matrix.CreateFromYawPitchRoll(2,3,1.5f) * Matrix.CreateTranslation(5,20,66);
  Matrix m = m0 * m1 * m2;
}
#endif
} // End DualQuaternion_c
```

Listing 1: Dual-Quaternion Implementation Class (*note, this version of the class was written for clarity a production ready version could be optimised and made more compact*).