Journal of Applied Mathematics and Computational Mechanics 2019, 18(1), 77-87
www.amcm.pcz.pl p-ISSN 2299-9965
DOI: 10.17512/jamcm.2019.1.07 e-ISSN 2353-0588

AN INSIGHT INTO THE EVOLUTION OF ROTATION OPERATOR TO QUATERNION'S. COMPUTER GRAPHICS PERSPECTIVE

Saad Bin Sami, Humera Tariq

Department of Computer Science, University of Karachi
Karachi, Pakistan
saadbinsami121@gmail.com, humera@uok.edu.pk

Received: 8 April 2019; Accepted: 24 April 2019

Abstract. Rotations are an integral part of various computational techniques and mechanics. The objective in this paper is twofold: first to have a classical insight into the history of quaternions, a problem that Hamilton faced for over a decade and secondly to look at into its applications from computer graphics perspective. Thorough revision of quaternion algebra and its use case as a rotation operator has been presented. A quaternion simulation algorithm has been written and practiced to generate simulation results. Results show that though quaternions supersede Euler angles technically but are tricky to use and control for e.g. when same quaternion is applied on a different vector axis, the particle is not able to reach its initial position and an incomplete rotation effect has been recorded and observed.

MSC 2010: *65D18, 97R60*
Keywords: *rotation, affine, transformation OpenGL, visualization*

1. Introduction

Translation, scaling and rotation are three primitive Affine Operations in Computer Graphics for modeling, animation, viewing, and creating special techniques for e.g. Frenet frame and Bill boarding [1]. The transition of all these operations from an equation form into their counterpart matrices is itself an interesting phenomena for early stage computer scientists. If a programmer is asked to develop a routine for rotation, he or she might consider this problem as a computation of new coordinates x, y and z with older object coordinates as input through equations as a key process. An experienced developer, on the other hand, knows very well that composition of transformations is computationally efficient which involves representing each primitive transformation by a 4×4 homogenous matrix. The homogenous representation not only facilitate the discrimination between a point and vector in the same Euclidean space (world space) but also allows merging of translation with other transformations through matrix multiplication [2]. The scheme also allows to maintain a transformation stack for manipulation of articulated objects and supports the splendid concept of coordinate system transformation (top down machine thinking) in contrast to object transformation (bottom up human mind thinking) [1, 3]. The nutshell behind all this is the Euler's rotation theorem which states that: "Any rotation (or sequence of rotations) about a fixed point is equivalent to a single rotation about some axis through that point" [3, 4]. The theorem led the most simplest definition of rotation in Open GL as glRotated (angle, u_x, u_y, u_z) which means that a graphics developer simply provides the angle and specific axis **'u'** about which rotation is required. The complex matrix that works behind; establishes a 2D coordinate system with the help of two orthogonal vectors taking the following form:

$$R_u(\beta) = \begin{bmatrix} c + (1-c)u_x^2 & (1-c)u_y u_x - s u_z & (1-c)u_z u_x + s u_y & 0 \\ (1-c)u_x u_y + s u_z & c + (1-c)u_y^2 & (1-c)u_z u_y - s u_x & 0 \\ (1-c)u_x u_y - s u_y & (1-c)u_x u_z + s u_x & c + (1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

where $c = \cos(\beta)$, $s = \sin(\beta)$, and (u_x, u_y, u_z) are the components of the unit vector u. Quaternions are computationally more efficient operators to describe the orientation of an object in a world space. They do not get stuck in a Gimbal Lock Situation nor cause an interpolation problem and are thus more suited for a machine oriented paradigm to approach rotation in comparison to constructive Euler angles base ambiguous rotation operator definition as described in (1) [5]. The recent decade has witnessed the interest of researchers into quaternion for a variety of applications including but not limited to modeling, animation and quaternion based estimation [6, 7]. The rest of the paper is organized as follows: Section 2 provides a historical overview of quaternions along with key algebraic manipulations and Rotation. The subsequent Section 3 and Section 4 will contain the detailed description of a rotation operator and its transformation into the matrix form for computing applications is presented in the end of Section 5. Simulation in C++ with OpenGL and Python with Jupyter Notebook is presented in Section 6. Finally conclusion and future work is described in the end.

2. Historical overview of quaternion's

In the early 19th century, mathematicians were interested in the question "can we represent complex numbers in a three dimensional space?". The answer to this question was not obvious and many mathematicians, including Gauss, Möbius, Grassmann, and Hamilton had been searching for the answer. Hamilton was working under the assumption that new algebra would be a super set of the already established Complex Number Algebra [8, 9]. However the problem that puzzled

Sir Hamilton over a decade was multiplication of two triplets. The multiplication of $z_1 = a_1 + b_1 i + c_1 j$ with $z_2 = a_2 + b_2 i + c_2 j$ violates Morgan's Law of commutation as $(ij)^2 = \pm 1$ showing that $ij \neq ji$ which in turn violates the law of modulus causing the length of the product of two vectors to not be the same as the product of the lengths of two original vectors as demonstrated via example in Table 1 for:

$$z_1 z_2 = (1 + 2i + 6j)(-4 + 3i + 7j).$$

Table 1. Violation of complex triplet multiplication and modulus law when $\boldsymbol{ij} = \mathbf{1}$

$(1 + 2i + 6j)(-4 + 3i + 7j) = \{-4 - 6 - 42 + 14 + 18\} + \{3 - 8\}i + \{7 - 24\}j$
$= -20 - 5i - 17j = x^2 + y^2 + z^2$
Product of length of two original vectors $= (a^2 + b^2 + c^2)\,(d^2 + e^2 + f^2)$
$= (1 + 4 + 36)(16 + 9 + 49)$
$= (41)(74) = 3034$
Product of two vectors $= -20 - 5i - 17j$
Length of product of two vectors $= 20^2 + 5^2 + 17^2 = 200 + 25 + 289 = 514$
Clearly the law of modulus is violating as $3034 \neq 514$ and thus $(a^2 + b^2 + c^2)(d^2 + e^2 + f^2) \neq x^2 + y^2 + z^2$

The same reasoning of Table 1 can be done for the case when $ij = -1$. The solution to the aforementioned problem is treating the vector as a quadruple rather than a triplet, and Hamilton led the definition of a quaternion q and its associated rules as follows:

$$Q = q_0 + q_1 i + q_2 j + q_3 k = [q_0 + \vec{q}] \tag{2}$$

where $\vec{q} = q_1 i + q_2 j + q_3 k$ and $i^2 = j^2 = k^2 = ijk = -1$.

Hamilton showed courage to deny Morgan's Law of Commutation for Algebra and turned the imaginary terms i, j, k into unit Cartesian vectors i, j, k, but Simon Altman's suggestion is to replace the imaginaries by the ordered pairs: $i = [0, i]$ $j = [0, j]$ $k = [0, k]$ which are themselves quaternions, and called quaternion units [8, 10].

2.1. Quaternion Algebra

Let $P = p_0 + ip_1 + jp_2 + kp_3$ and $Q = q_0 + iq_1 + jq_2 + kq_3$. Addition will be done in component wise manner with both commutativity and associativity intact

$$P + Q = (p_0 + q_0) + i(p_1 + q_1) + j(p_2 + q_2) + k(p_3 + q_3) \tag{3}$$

It is important to look at details of multiplying two quaternions as follows:

$$PQ = (p_0q_0 + ip_0q_1 + jp_0q_2 + kp_0q_3) + (ip_1q_0 + i^2p_1q_1 + ijp_1q_2 + ikp_1q_3) + (jp_2q_0 + jip_2q_2 + jkp_2q_2 + jkp_2q_3) + (kp_3q_0 + kip_3q_1 + kjp_3q_2 + k^2p_3q_3$$

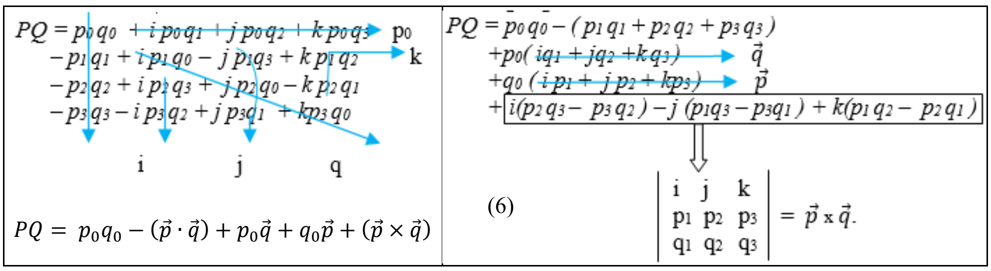
Using interaction properties of vectors from (4), equation (3) can be transformed into Eq. (5) by the steps described below:

$$\begin{gathered} i^2 = j^2 = k^2 = ijk = -1; \\ ij = k, jk = i, ki = j; \\ ji = -k, kj = -i, ik = -j \end{gathered} \tag{4}$$

$$\begin{aligned} PQ = {} & (p_0q_0 + ip_0q_1 + jp_0q_2 + kp_0q_3) + (-p_1q_1 + ip_1q_0 - jp_1q_3 + kp_1q_2) \\ & +(-p_2q_2 + ip_2q_3 + jp_2q_0 - kp_2q_1) + (-p_3q_3 - ip_3q_2 + jp_3q_1 + kp_3q_0) \end{aligned} \tag{5}$$

Table 2 represents steps to complete multiplication in equation (6).

Table 2. Intermediate and final steps for Quaternion Multiplication



$PQ = p_0q_0 + ip_0q_1 + jp_0q_2 + kp_0q_3 \rightarrow p_0$ $-p_1q_1 + ip_1q_0 - jp_1q_3 + kp_1q_2 \rightarrow k$ $-p_2q_2 + ip_2q_3 + jp_2q_0 - kp_2q_1$ $-p_3q_3 - ip_3q_2 + jp_3q_1 + kp_3q_0$ $\downarrow i \quad \downarrow j \quad \searrow q$ $PQ = p_0q_0 - (\vec{p}\cdot\vec{q}) + p_0\vec{q} + q_0\vec{p} + (\vec{p}\times\vec{q})$	$PQ = p_0q_0 - (p_1q_1 + p_2q_2 + p_3q_3)$ $+p_0(iq_1 + jq_2 + kq_3) \rightarrow \vec{q}$ $+q_0(ip_1 + jp_2 + kp_3) \rightarrow \vec{p}$ $+\boxed{i(p_2q_3 - p_3q_2) - j(p_1q_3 - p_3q_1) + k(p_1q_2 - p_2q_1)}$ $\Downarrow$ (6) $\begin{vmatrix} i & j & k \\ p_1 & p_2 & p_3 \\ q_1 & q_2 & q_3 \end{vmatrix} = \vec{p}\times\vec{q}.$

In equation (6), the term $p_0q_0 - (\vec{p}\cdot\vec{q})$ is scalar and the term $p_0\vec{q} + q_0\vec{p} + (\vec{p}\times\vec{q})$ is a vector but since the term $\vec{p}\times\vec{q}$ is non commutative so the quaternion product also inherently becomes non-commutative.

2.2. Complex conjugate, norm and inverse

The complex conjugate of quaternion ($Q = q_0 + \vec{q} = q_0 + iq_1 + jq_2 + kq_3$) is:

$$Q^* = q_0 + \vec{\boldsymbol{q}}^* = q_0 - iq_1 - jq_2 - kq_3 \tag{7}$$

The norm of the quaternion q is denoted by $N(q)$ or$|N|$ and sometimes called the length of the quaternion and is given by equation (8):

$$\begin{aligned} & N(Q) = \sqrt{\mathrm{Q} * \mathrm{Q}} \\ & N^2(q) = (q_0 - \vec{q})(q_0 + \vec{q}) = 0 \\ & = q_0q_0 - (-\vec{q})\cdot\vec{q} + q_0(\vec{q}) + q_0(-\vec{q}) + (\vec{q}\times\vec{q}) \text{ using Eq. (6)} \end{aligned}$$

$$N^2(Q) = q_0^2 + (\vec{q}\cdot\vec{q}) = q_0^2 + q_1^2 + q_2^2 + q_3^2 = q_0^2 + |q|^2 \tag{8}$$

For every non-quaternion, we can calculate the inverse. By definition of inverse:

$$q^{-1}q = qq^{-1} = 1$$

By pre and post multiplication of complex conjugate:

$$q^*qq^{-1} = qq^{-1}q^* = q^*$$

Since $q * q = N^2(Q)$, we get:

$$q^{-1} = q^*/N^2(q)$$

The quaternion with the norm = 1 is called a **Unit Quaternion**.

3. Quaternion rotation operator

The quaternion operator q can be applied on a given vector $\vec{v}$ to get the image $\vec{q}$which may serve as desired rotated vector $w = q\vec{v}$. Expressing $\vec{w}$ and $\vec{v}$ into its real and imaginary components:

$$\begin{aligned}\vec{w} &= (q_0 + \vec{q})(0 + \vec{v}) \\ &= q_0(0) - \vec{q} \cdot \vec{v} + 0(\vec{q}) + q_0\vec{v} + \vec{q} \times \vec{v}\end{aligned}$$

This computation shows that the image $\vec{w}$ is not necessarily in Q_0. As it is not guaranteed that the term $(-\vec{q} \cdot \vec{v})$ is always zero. The product $\vec{v}q$ will not work either as commuting the factors will not change the real part either. Thus we cannot expect our quaternion rotation operator to simply consist of a single quaternion. These observations led to the thinking that desired rotation operator may involve triplets or perhaps even a higher order product. Let Q and R be two quaternion and $\vec{p}$ be the pure quaternion (representing the desired vector to be rotated). Then there are 6 possibilities of these factors [11]: $\vec{p}$QR, $\vec{p}RQ$, QR$\vec{p}$, RQ$\vec{p}$, R$\vec{p}$Q, Q$\vec{p}$R. Now we know that the set of quaternions except pure quaternions (a quaternion $Q \in R^4$ whose real part is zero is called pure quaternion) are closed under multiplication. Thus the product of the first four combinations will produce the product of a quaternion and a pure quaternion as demonstrated recently in this section. The last hope will be R$\vec{p}$Q and Q$\vec{p}$R; both of them are similar [11].

Let's solve Q$\vec{p}$R. Let Q = $q_0 + \vec{q}$; $\vec{p} = 0 + \vec{p}$ and R = $r_0 + \vec{r}$. Then the real part of the product Q$\vec{p}$R will be:

$$Q\vec{P}R = (q_0 + \vec{q})\ (0 + \vec{p})(r_0 + \vec{r})$$
$$Q\vec{P}R = (q_0(0) - \vec{q} \cdot \vec{p} + \vec{q}(0) + q_0\vec{p} + \vec{q}_x\vec{p})(r_0 + \vec{r})$$
$$Q\vec{P}R = (-\vec{q} \cdot \vec{p} + q_0\vec{p} + \vec{q}_x\vec{p})(r_0 + \vec{r})$$

Let us suppose for our simplicity that: $x_0 = -\vec{q} \cdot \vec{p}$ and $\vec{x} = q_0\vec{p} + \vec{q}_x\vec{p}$.

So, Q$\vec{p}$R $= (x_0 + \vec{x})(r_0 + \vec{r})$

Now Q$\vec{p}$R $= x_0r_0 - \vec{x} \cdot \vec{r} + \vec{x}r_0 + \vec{r}x_0 + \vec{x}_x\vec{r}$

But $x_0 = -\vec{q} \cdot \vec{p}$ and $\vec{x} = q_0\vec{p} + \vec{q}_x\vec{p}$

$$Q\vec{p}R = (-\vec{q} \cdot \vec{p})r_0 - (q_0\vec{p} + \vec{q}_x\vec{p}) \cdot \vec{r} + (q_0\vec{p} + \vec{q}_x\vec{p})r_0 + \vec{r}(-\vec{q} \cdot \vec{p}) + (q_0\vec{p} + \vec{q}_x\vec{p})_x\vec{r}$$

Here, the real part is $= (-\vec{q}\cdot\vec{p})r_0 - (q_0\vec{p} + \vec{q}_x\vec{p})\cdot\vec{r}$
$= -r_0(\vec{q}\cdot\vec{p}) - q_0(\vec{p}\cdot\vec{r}) - (\vec{q}_x\vec{p})\cdot\vec{r}$
$= -r_0(\vec{q}\cdot\vec{p}) - q_0(\vec{p}\cdot\vec{r}) - (-\vec{p}_x\vec{q})\cdot\vec{r}$
$= -r_0(\vec{q}\cdot\vec{p}) - q_0(\vec{p}\cdot\vec{r}) + (\vec{p}_x\vec{q})\cdot\vec{r}$

Here $(\vec{p}_x\vec{q})\cdot\vec{r}$ is a vector triple product. And by vector algebra $(\vec{p}_x\vec{q})\cdot\vec{r}$ $= (\vec{p}_x\vec{r})\cdot\vec{q}$. So, the real part is $= -r_0(\vec{q}\cdot\vec{p}) - q_0(\vec{p}\cdot\vec{r}) + (\vec{p}_x\vec{r})\cdot\vec{q}$

Let $r_0 = q_0$; is $= -q_0(\vec{q}\cdot\vec{p}) - q_0(\vec{p}\cdot\vec{r}) + (\vec{p}_x\vec{r})\cdot\vec{q}$
$= -q_0(\vec{q}\cdot\vec{p} - \vec{p}\cdot\vec{r}) + (\vec{p}_x\vec{r})\cdot\vec{q}$
$= -q_0(\vec{q} - \vec{r})\cdot\vec{p} + (\vec{p}_x\vec{r})\cdot\vec{q}$

Clearly the real Part will be zero, when $\vec{q} = -\vec{r}$. And we have already supposed that $r_0 = q_0$. So this means that

$R = r_0 + \vec{r} = q_0 + \vec{q} = Q*$

So, Q = R* and R = Q*

So our quaternion rotation operator will be $Q\vec{p}Q*$ or $Q*\vec{p}Q$. Both of these operators will produce the required pure quaternion i.e. $\vec{w}_1 = Q\vec{p}Q*$ or $\vec{w}_1 = Q*\vec{p}Q$.

4. Trigonometric encoding and geometry of quaternion's rotation

Let the quaternion $Q = q_0 + \vec{q}$ be a unit quaternion. We know that: $N^2(Q) =$ $= {q_0}^2 + |q|^2 = 1$. We also know that, $Cos^2\Theta + Sin^2\Theta = 1$.

Comparing the two facts: $Cos^2\Theta = {q_0}^2$ and $Sin^2\Theta = |q|^2$. Let $\vec{u}$ be a unit vector. The unit quaternion can be encoded using trigonometric identities as follows:

$$Q = Cos\Theta + \vec{u}Sin\Theta.$$

If we substitute angle to be $-\Theta$. Then

$$Q* = Cos\Theta - \vec{u}Sin\Theta \tag{9}$$

Let $Q = Cos\Theta + \vec{u}Sin\Theta$ then $Q* = Cos\Theta - \vec{u}Sin\Theta$ and $\vec{v} = 0 + j$ (the vector to be rotated) and let $\Theta = 45 = \pi/4$ and $\vec{u} = i$.

Then, $Q\vec{v}Q^* = (Cos\Theta + \vec{u}Sin\Theta)(0 + j)(Cos\Theta - \vec{u}Sin\Theta)$
$= (Cos\frac{\pi}{4} + iSin\frac{\pi}{4})(0 + j)(Cos\frac{\pi}{4} - iSin\frac{\pi}{4})$

$$= \left(\frac{\sqrt{2}}{2} + \mathrm{i}\frac{\sqrt{2}}{2}\right)(0 + \mathrm{j})\left(\frac{\sqrt{2}}{2} - \mathrm{i}\frac{\sqrt{2}}{2}\right)$$

$$= \left\{0 - 0 + \mathrm{j}\frac{\sqrt{2}}{2} + 0 + \frac{\sqrt{2}}{2}(\mathrm{i}_x\mathrm{j})\right\}\left(\frac{\sqrt{2}}{2} - \mathrm{i}\frac{\sqrt{2}}{2}\right)$$

$$= \left\{0 + \mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k}\right\}\left(\frac{\sqrt{2}}{2} - \mathrm{i}\frac{\sqrt{2}}{2}\right)$$

$$= 0 - \left(\mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k}\right).\left(\mathrm{i}\frac{\sqrt{2}}{2}\right) + \frac{\sqrt{2}}{2}\left(\mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k}\right) + 0 + \left(\mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k}\right)_x\left(\mathrm{i}\frac{\sqrt{2}}{2}\right)$$

$$
\begin{aligned}
&= \mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k} + \begin{vmatrix} i & j & k \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & 0 & 0 \end{vmatrix} \\
&= \mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k} - \mathrm{j}\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}\mathrm{k} \\
&= \mathrm{k}
\end{aligned}
$$

The insight interpretation of the result is as follows. Given a unit quaternion, $Q = Cos\theta + \vec{u}Sin\theta$ and $\vec{v} = 0 + j$ and $\theta = 45 = \pi/4$ and $\vec{u} = i$, the vector j is rotated 90° (twice the given angle) about the axis $\vec{u} = i$ in a counterclockwise direction, and similarly $Q^*\vec{v}Q$ will rotate the vector in a clockwise direction.

5. Generalized formula and matrix form of quaternion rotation

$$
\begin{aligned}
\vec{w} = Q\vec{v}Q^* &= (q_0 + \vec{q})(0 + \vec{v})(q_0 - \vec{q}) \\
&= (0 - \vec{q}\cdot\vec{v} + q_0\vec{v} + 0 + \vec{q}\mathrm{x}\vec{v})(q_0 - \vec{q}) \\
&= (-\vec{q}\cdot\vec{v} + q_0\vec{v} + \vec{q}\mathrm{x}\vec{v})(q_0 - \vec{q}) \\
&= -(\vec{q}\cdot\vec{v})q_0 - (q_0\vec{v} + \vec{q}\mathrm{x}\vec{v})\cdot(-\vec{q}) + (-\vec{q}\cdot\vec{v})(-\vec{q}) + q_0(q_0\vec{v} + \vec{q}\mathrm{x}\vec{v}) + \\
&\quad (q_0\vec{v} + \vec{q}\mathrm{x}\vec{v})\mathrm{x}(-\vec{q}) \\
&= -\cancel{(\vec{q}\cdot\vec{v})q_0} + \cancel{q_0(\vec{v}\cdot\vec{q})} + \cancel{(\vec{q}x\vec{v})}\cdot(\vec{q}) + (\vec{q}\cdot\vec{v})(\vec{q}) + q_0{}^2\vec{v} + q_0(\vec{q}x\vec{v}) \\
&\qquad - q_0(\vec{v}x\vec{q}) - (\vec{q}x\vec{v})x(\vec{q}) \\
&= (\vec{q}\cdot\vec{v})(\vec{q}) + q_0{}^2\vec{v} + q_0(\vec{q}\mathrm{x}\vec{v}) - q_0(\vec{v}\mathrm{x}\vec{q}) - (\vec{q}\mathrm{x}\vec{v})\mathrm{x}(\vec{q}) \\
&= (\vec{q}\cdot\vec{v})(\vec{q}) + q_0{}^2\vec{v} - q_0(\vec{v}\mathrm{x}\vec{q}) - q_0(\vec{v}\mathrm{x}\vec{q}) - (\vec{q}\mathrm{x}\vec{v})\mathrm{x}(\vec{q}) \\
&= (\vec{q}\cdot\vec{v})(\vec{q}) + q_0{}^2\vec{v} - 2q_0(\vec{v}x\vec{q}) - (\vec{q}x\vec{v})x(\vec{q}) \qquad (10)
\end{aligned}
$$

Here: $-(\vec{q}x\vec{v})x(\vec{q}) = -[\vec{v}(\vec{q}\cdot\vec{q}) - \vec{q}(\vec{q}\cdot\vec{v})] = -[\vec{v}|\vec{q}|^2 - \vec{q}(\vec{q}\cdot\vec{v})]$ because $a \times b \times c = b(a\cdot c) - c(a\cdot b)$.

Let $N^2(Q) = q_0{}^2 + |q|^2 = 1$, so $|\vec{q}|^2 = 1 - q_0{}^2$. This assumption clarifies the use of a unit quaternion in a rotation operator.
So Equation (10) becomes,

$$
\begin{aligned}
\vec{w} &= (\vec{q}\cdot\vec{v})(\vec{q}) + q_0{}^2\vec{v} - 2q_0(\vec{v}\mathrm{x}\vec{q}) - \vec{v} + \vec{v}q_0{}^2 + \vec{q}(\vec{q}\cdot\vec{v}) \\
&= (\vec{q}\cdot\vec{v})(\vec{q}) + 2q_0{}^2\vec{v} - 2q_0(\vec{v}x\vec{q}) - \vec{v} + \vec{q}(\vec{q}\cdot\vec{v}) \\
&= 2(\vec{q}\cdot\vec{v})(\vec{q}) + 2q_0{}^2\vec{v} - 2q_0(\vec{v}x\vec{q}) - \vec{v} \\
\vec{w} &= Q\vec{v}Q^* = \vec{v}(2q_0{}^2 - 1) + 2(\vec{q}\cdot\vec{v})(\vec{q}) - 2q_0(\vec{v}x\vec{q}) \qquad (11)
\end{aligned}
$$

Matrix form of quaternion for rotation can be formulated as follows:

$$\vec{v}(2{q_0}^2 - 1) = \begin{bmatrix} (2{q_0}^2 - 1) & 0 & (2{q_0}^2 - 1) \\ 0 & (2{q_0}^2 - 1) & 0 \\ 0 & 0 & (2{q_0}^2 - 1) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$2(\vec{q} \cdot \vec{v})(\vec{q}) = \begin{bmatrix} 2{q_1}^2 & 2q_1q_2 & 2q_1q_3 \\ 2q_1q_2 & 2{q_2}^2 & 2q_2q_3 \\ 2q_1q_3 & 2q_2q_3 & 2q_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$2{q_0}^2 \left(\overrightarrow{\vec{v}_x q}\right) = \begin{bmatrix} 0 & -2q_0q_3 & 2q_0q_2 \\ 2q_0q_3 & 0 & -2q_0q_1 \\ -2q_0q_2 & 2q_0q_1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

The sum of these components may be written as:

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 2{q_0}^2 - 1 + 2{q_1}^2 & 2q_1q_2 - 2q_0q_3 & 2{q_0}^2 - 1 \\ 2q_1q_2 + 2q_0q_3 & 2{q_0}^2 - 1 + 2{q_2}^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 2{q_0}^2 - 1 + 2{q_3}^2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

So, $\vec{w} = Q\vec{v}Q^* = Q\vec{v}$ will give the counterclockwise rotation and $\vec{w} = Q^*\vec{v}Q = = Q^t\vec{v}$ will give the clockwise rotation.

6. Simulation and results

To experiment with quaternions from the computer graphics perspective, we modify existing Quaternion Code for Intuition Building and simulation on C++ and OpenGL [12]. The existing quaternion implementation treat quaternion as a composition of 3D vector as shown in Figure 1. To Use Quaternion in a program, the Simulation Algorithm of Table 1 can be used. The Algorithm has been tested for a point that resides on a circle as: (cos(t), sin(t),0) and the axis of rotation is set to (0,0,1), (0,1,0) and (1,0,0) respectively. It has been observed that for rotation about (0,0,1) i.e. z-axis the point rotates from 0° to 360° but for rotation about the y-axis (0,1,0) it doesn't complete the cycle. Rotation about the x-axis (1,0,0) has been observed as a particle is moving on straight line and depth effect from front to back is not observable. Quaternion's simulation results about *z* and *y* axis are shown in Figures 2 and 3 respectively. We also test Quaternion matrix in Python on Jupyter notebook to rotate Blue vector (image matrix) **B** around green vector (green matrix) **G** where both **B** and **G** are of size (600, 400). It is important to note that Python and OpenCV image library treated RGB as BGR space. Result in Figure 4 shows that Blue Image is successfully turned into Red through quaternion operation. The simulation Code is presented in Table 4 for reader convenience.

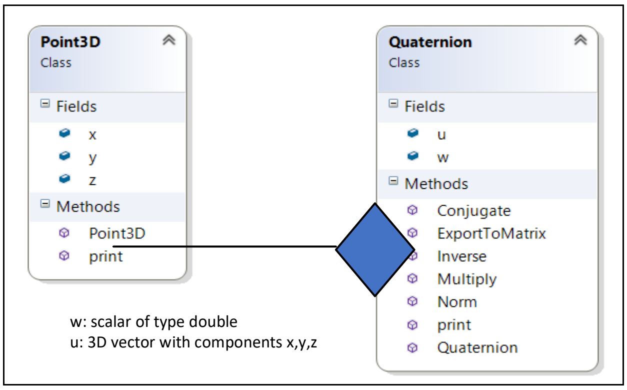


Fig. 1. C++ implementation of quaternion as composite object [12]

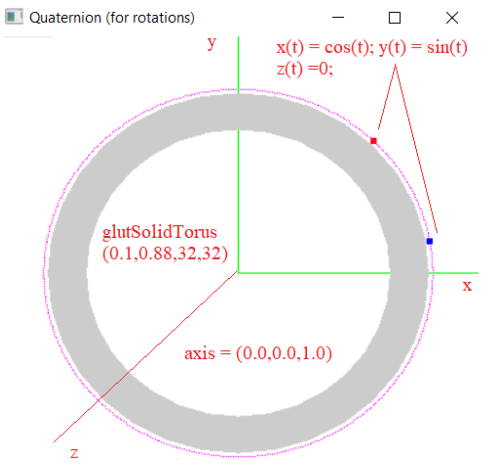


Fig. 2. Quaternion rotation about Z-axis

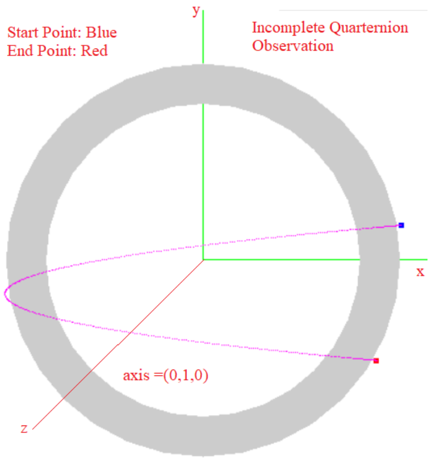


Fig. 3. Rotation about Y-axis

Table 3. Algorithm to simulate quaternion

```
Procedure  QuaternionSimultion (function x(u,v: real): real;
           function y(u,v:  real); function z(u,v: real): real;
           angle: double; var $\vec{v}$,axis: Point3D): q: Quarternion,
           mat: array[1...4,1...4] of real; var
           i: integer; numPoints: integer;        incr: integer; Q: Quaternion, p: Point3D
begin
      for i:=1 to numPoints do
      begin
        Q.w =  cos(angle/2);
         Q.u.x = sin(angle/2)*axis.x;
         Q.u.y = sin(angle/2)*axis.y;
         Q.u.z = sin(angle/2)*axis.z;
        q :=    Q$\vec{v}$Q*
        mat: = ExportToMatrix(q);  mat:= GL_MODELVIEW * mat;
        drawRotatedPoint(v.x,v.y,v.z);
        end
end
```

Table 4. Python and OpenCV Simulation

(1)
`import numpy as np;`
`import cv2 ;`
`import math`
	(2)
	`size = (600,400,3)`
	
	`blue = np.zeros(size,dtype=float)`
	`blue[:,:,0] = 1`
	`Input_color = blue`
	`green = np.zeros(size,dtype=float)`
	`green[:,:,1] = 1`

(3)
`q_not = np.cos(math.pi/4)  90 deg rotation`
 	
`q = np.zeros(size,dtype=float)`
`q[:,:,0] = green[:,:,0]*np.sin(math.pi/4)# q1`
`q[:,:,1] = green[:,:,1]*np.sin(math.pi/4)# q2`
`q[:,:,2] = green[:,:,2]*np.sin(math.pi/4)# q3`
	(4)
	`Q11 = 2*q_not**2 - 1 + 2*q[:,:,0]**2`
	`Q12 = 2*q[:,:,0]*q[:,:,1] - 2*q_not*q[:,:,2]`
	`Q13 = 2*q[:,:,0]*q[:,:,2] + 2*q_not*q[:,:,1]`
	
	`Q21 = 2*q[:,:,0]*q[:,:,1] + 2*q_not*q[:,:,2]`
	`Q22 = 2*q_not**2 - 1 + 2*q[:,:,1]**2`
	`Q23 = 2*q[:,:,1]*q[:,:,2] - 2*q_not*q[:,:,0]`

(5)
`Q31 = 2*q[:,:,0]*q[:,:,2] - 2*q_not*q[:,:,1]`
`Q32 = 2*q[:,:,1]*q[:,:,2] + 2*q_not*q[:,:,0]`
`Q33 = 2*q_not**2 - 1 + 2*q[:,:,2]**2`
	(6)
	`#clock wise`
	`w11 = Q11*blue[:,:,0] + Q21*blue[:,:,1] + Q31*blue[:,:,2]`
	`w12 = Q12*blue[:,:,0] + Q22*blue[:,:,1] + Q32*blue[:,:,2]`
	`w13 = Q13*blue[:,:,0] + Q23*blue[:,:,1] + Q33*blue[:,:,2]`

(7)
`Output_color = np.zeros(size,dtype=float)`
`Output_color[:,:,0] = w11`
`Output_color[:,:,1] = w12`
`Output_color[:,:,2] = w13`
	(8)
	`cv2.imshow('Output_color',Output_color)`
	`cv2.imshow('Rotated Around',green)`
	`cv2.imshow('Input_color',Input_color)`
	`cv2.waitKey(0)`
	`cv2.destroyAllWindows()`

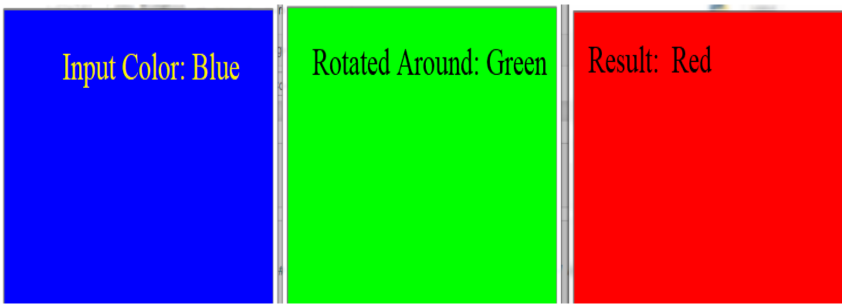


Fig. 4. Blue Matrix Rotated Around Green through Quaternion's

7. Conclusions

In the paper, quaternions as an alternative to a rotation operator have been studied for simulation from computer graphics perspective. Thorough study of quaternion algebra has been turned into pseudocode and Python code to be used for variety of applications. Special cases has been demonstrated for rotation of a point on circle or sphere around *x*, *y* and z-axis. C++, OpenGL and OpenCV has been used for presenting simulation results. Future work will emphasize on higher level modeling and animation e.g surface of revolution, bill boarding and Frenets.

References

[1] Hill, F.S. (1999). *Computer Graphics using Open GL* (Second Edition). Prentice Hall.

[2] Paeth, A.W. (1995). *Graphics Gems V*. Academic Press.

[3] Shriener, D., Sellers, G., Kessinish, J., & Licea, K.B. (2013). *Open GL Programming Guide: The Official Guide to Learning*. (Eighth Edition). Addison Wesley.

[4] Gellert, W., Kiistner, H., Hellwich, M., & Kastner, H. (1975). *The VNR Concise Encyclopedia of Mathematics*. Van Nostrand Reinhold.

[5] Pletinckx, D. (1989). Quaternion calculus as a basic tool in computer graphics. *The Visual Computer*, 5(1), 2-13.

[6] Pavllo, D., Feichtenhofer, C., Auli, M., & Grangier, D. (2019). Modeling Human Motion with Quaternion-based Neural Networks. Pre print arXiv.

[7] Wei-Hsu, H. et al. (2019). Quaternion-based head pose estimation with multiregression loss. *IEEE Transactions on Multimedia*, *21*(4), 1035-1046.

[8] Familton, J.C. (2015). Quaternions: A History Of Complex Noncommutative Rotation Groups In Theoretical Physics. Pre print arXiv.

[9] Van Der Waerden, B.L. (1976). Hamilton's discovery of quaternions. *Mathematics Magazine*, *49*(5), 227-234.

[10] Vince, J. (2011). *Quaternions for Computer Graphics*. London: Springer-Verlag.

[11] Kuipers, J.B. (1999). *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton University Press.

[12] Nielson, F., & River, C. (2005). *Visual Computing: Geometry, Graphics and Vision*. Charler Medial Press.