X.3
USING QUATERNIONS FOR CODING 3D TRANSFORMATIONS

Patrick-Gilles Maillot
Sun Microsystems, Inc.
Mountain View, California

Introduction

This paper presents some fundamentals of a mathematical element called a *quaternion*. A theoretical study emphasizes the characteristic properties of these elements and the analogy that can be made between the geometrical interpretation of 3D transformations and the subgroup of *unit* quaternions.

We also give a practical application of quaternions for the coding of 3D transformations and some basic algorithms for orthogonal coordinate systems displacements.

Quaternions were introduced in 1843 by Sir William R. Hamilton. They are useful primarily in the coding of natural movements in a 3D space, as it has been realized in the *PATK3D* software (Maillot, 1983), or to perform movements along parametric curves (Shoemake, 1985). Quaternions may also be used when coding transformations in Constructive Solid Geometry (CSG) trees as used in MCAD systems or applications, as well as when coding the hierarchical transformation trees of visualization software, such as proposed by the PHIGS (PHIGS, 1987) standard.

This paper describes a way to code and evaluate 3D transformations using a different method than coding 4×4 matrices. The initial researches on the material presented here have been conducted at the University Claude Bernard, Lyon I, France. In 1983, the graphic group was working on formalization of scene composition and natural movements coding in space. Some mathematical demonstrations presented

here are due to E. Tosan (Tosan, 1982), who is a mathematician researcher of the computing sciences group.

A mathematical definition of the quaternion division ring, **Q**, is proposed to demonstrate some of the particularities and properties of the quaternions, as well as the definition of the operations that can be applied to them.

The analogy that can be made between this algebraic element and the geometry of 3D transformations will be described. This is constructed over a subgroup of quaternions having a particular property.

Some examples of code are proposed that show two basic transformations involving only some simple algebraic computations.

Definition of the Quaternions

A quaternion is a mathematical element noted

$$q = c + xi + yj + zk,$$

with c, x, y, z: real numbers, and i, j, K: imaginary numbers. It is also written in condensed notation:

$$q = c + u,$$

with $u = xi + yj + zk$ called the *pure* part of the quaternion, and c the *real* part of the quaternion.

Let us supply **Q**, the set of quaternions, with two operations: addition,

$$q + q' = (c + c') + (x + x')i + (y + y')j + (z + z')k$$

and multiplication, defined on the base $\{1, i, j, k\}$:

$$i^2 = j^2 = k^2 = -1$$

$$ij = k,\ ji = -k;\ jk = i,\ kj = -i;\ ki = j,\ ik = -j.$$

If we develop the multiplication, we obtain:

$$qq' = (c + xi + yj + zk)(c' + x'i + y'j + z'k)$$

$$= (cc' - xx' - yy' - zz') + (yz' - y'z + cx' + c'x)i$$

$$+ (zx' - z'x + cy' + c'y)\, j + (xy' - x'y + cz' + c'z)k.$$

Or, if we use the condensed notation,

$$qq' = (c + u)(c' + u')$$

$$= (cc' - u \cdot u') + (u \times u' + \langle cu' \rangle + \langle c'u \rangle),$$

with

$u \cdot u' = xx' + yy' + zz'$: inner product.

$\langle cu \rangle = cxi + cyj + czk$

$u \times u' = (yz' - zy', zx' - xz', xy' - yz')$: cross product.

Notes

- If we write $u = s\mathbf{N}$ and $u' = s'\mathbf{N}$, with s and s' being real numbers, and $\mathbf{N} = (\mathbf{N}_x, \mathbf{N}_y, \mathbf{N}_z)$ a unit vector of $\mathbf{R}^3$, so that $|\mathbf{N}|^2 = 1$, we find the product of complex numbers, with $\mathbf{N}$ instead of the imaginary number i.

  Then

$$qq' = (c + s\mathbf{N})(c' + s'\mathbf{N})$$

$$= (cc' - ss'\mathbf{N} \cdot \mathbf{N}) + (\mathbf{N} \times \mathbf{N} + \langle cs'\mathbf{N} \rangle + \langle c's\mathbf{N} \rangle)$$

$$= (cc' - s's) + \langle (cs' + c's)\mathbf{N} \rangle.$$

• If we write $q = q' = \mathbf{N}$, or $c = c' = 0$ and $s = s' = 1$, we finally find

$$\mathbf{N}^2 = -ss'\mathbf{N} \cdot \mathbf{N} = -1.$$

Properties of the Quaternions

The addition of quaternions has an *identity* element,

$$0 = 0 + 0i + 0j + 0k,$$

and an *inverse* element,

$$-q = -c - xi - yj - zk.$$

The multiplication of quaternions has an *identity* element,

$$1 = 1 + 0i + 0j + 0k.$$

The quaternions have some *conjugate* elements,

$$q = c + u;\ \bar{q} = c - u.$$

It is possible to define a magnitude for q by applying the *hermitian* inner product of q into $\bar{q}$:

$$\begin{aligned} q\bar{q} &= (c^2 + u \cdot u) + (u \times u - \langle cu \rangle + \langle cu \rangle) \\ &= c^2 + x^2 + y^2 + z^2 \\ &= \bar{q}q = |q|^2, \text{ magnitude of } q. \end{aligned}$$

The quaternions have some *inverse* elements:

$$q^{-1} = \frac{1}{|q|^2}\bar{q}.$$

The multiplication is *not commutative.* If it was commutative, we would

have had

$$qq' = q'q.$$

Then,

$$(c + u)(c' + u') = (c' + u')(c + u)$$

$$\Rightarrow u \times u' = u' \times u \Rightarrow u \times u' = 0.$$

The multiplication is *distributive over* the addition:

$$q(v + w) = qv + qw \qquad (v + w)q = vq + wq.$$

In conclusion, we may write that

- **Q** is closed under quaternion addition.
- **Q** is closed under quaternion multiplication.
- (**Q**, +, ·) is a division ring.

Properties of the Set of Unit Quaternions

This paragraph focuses on the subset quaternions of unit magnitude. These are the particular quaternions that are of interest in this application.

Let us consider the quaternions so that $|q| = 1$, or $c^2 + u \cdot u = 1$, and $c^2 + s^2 = 1$, (if $u = s\mathbf{N}$); then we can write any *unit* quaternion as $q = \cos(\theta) + \sin(\theta)\mathbf{N}$.

Then, for this particular set of quaternions,

- The multiplication of the conjugate of two quaternions is the conjugate of the multiplication of the quaternions.

$$\overline{qq'} = (cc' - u \cdot u') - (u \times u' + \langle cu' \rangle + \langle c'u \rangle)$$

$$\bar{q}'\bar{q} = (c' - u')(c - u)$$

$$= (cc' - u' \cdot u) + ((-u') \times (-u) - \langle cu' \rangle - \langle c'u \rangle)$$

$$= (cc' - u \cdot u') - (u \times u' + \langle cu' \rangle + \langle c'u \rangle)$$

$$= \overline{qq'}$$

- The magnitude of the multiplication of two quaternions is the multiplication of the magnitudes of the quaternions.

$$\begin{aligned} |qq'|^2 &= (qq')(\overline{qq'}) \\ &= qq'\,\overline{q}'\overline{q} = q|q'|^2\,\overline{q} \\ &= |q|^2|q'|^2 \end{aligned}$$

- If two quaternions have a magnitude of 1, then the product of these two quaternions will also have a magnitude of 1.

$$|q| = |q'| = 1 \Rightarrow |qq'| = 1$$

$$(\cos(\theta) + \sin(\theta)\mathbf{N})(\cos(\phi) + \sin(\phi)\mathbf{N}) = \cos(\theta + \phi) + \sin(\theta + \phi)\mathbf{N}$$

- The conjugate of a unit quaternion is its inverse.

$$|q| = 1 \Rightarrow q^{-1} = q$$

The set of unit quaternions is a multiplicative subgroup of the non-null quaternions.

Rotations in a 3D Space

Rotations can be expressed using a geometrical formulation. Figure 1 graphically presents the rotation of a vector around a given axis.

Let $P' = Rot_{(a,\mathbf{N})}(P)$ be the transformation of point P by the rotation of angle θ around the axis $\mathbf{N}$. We can write:

$$\mathbf{H} = \overrightarrow{\mathrm{OH}}$$

$$\mathbf{P}' = \overrightarrow{OP'} = \overrightarrow{OH} + \overrightarrow{HP'} = \mathbf{H} + \mathbf{U}'$$

$$\mathbf{P}' = \overrightarrow{OP'} = \overrightarrow{OH} + \overrightarrow{HP'} = \mathbf{H} + \mathbf{U},$$

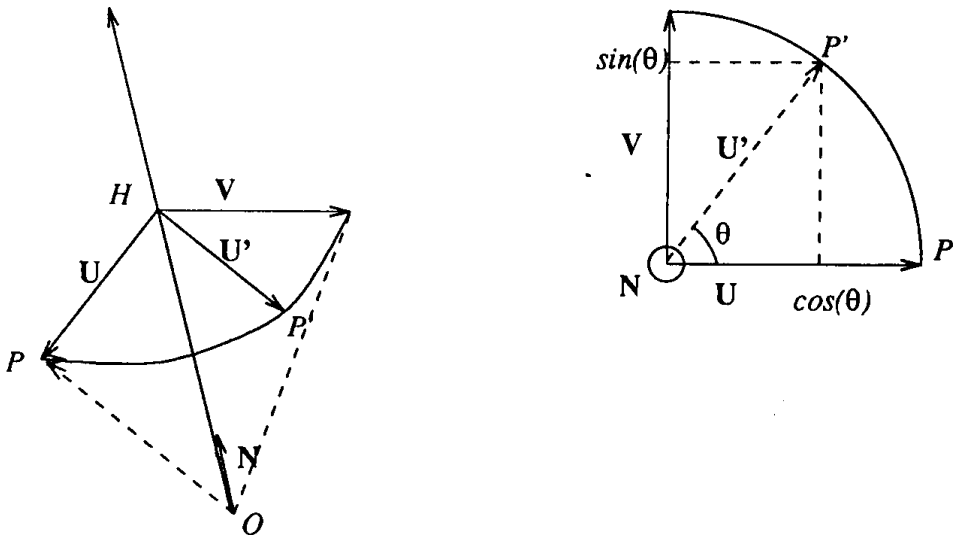

Figure 1. Geometrical representation of a rotation.

and then,

$$\mathbf{U} = \mathbf{P} - \mathbf{H}.$$

In the plane formed by (PHP'), we can set

$$\mathbf{U}' = \cos(\theta)\mathbf{U} + \sin(\theta)\mathbf{V},$$

with $\mathbf{U} \perp \mathbf{V}$, $\mathbf{V} \perp \mathbf{N}$, and $|\mathbf{V}| = |\mathbf{U}|$, that is to say, $\mathbf{V} = \mathbf{N} \times \mathbf{U} = \mathbf{N} \times (\mathbf{P} - \mathbf{H}) = \mathbf{N} \times \mathbf{P}$. On the $(O, \mathbf{N})$ axis, $\overrightarrow{OH}$ is the projection of $\overrightarrow{OP}$ on $\mathbf{N}$, and then $\overrightarrow{OH} = \mathbf{H} = \mathbf{N}\,(\mathbf{N} \cdot \mathbf{P})$, which gives the formulas

$$\overrightarrow{OP'} = \mathbf{H} + \mathbf{U}' = \mathbf{N}(\mathbf{N} \cdot \mathbf{P}) + (\cos(\theta)\mathbf{U} + \sin(\theta)\mathbf{V})$$

$$= \mathbf{N}(\mathbf{N} \cdot \mathbf{P}) + \cos(\theta)(\mathbf{P} - \mathbf{N}(\mathbf{N} \cdot \mathbf{P}) + \sin(\theta)\mathbf{N} \times \mathbf{P}$$

$$= \cos(\theta)\mathbf{P} + (1 - \cos(\theta))\mathbf{N}(\mathbf{N} \cdot \mathbf{P}) + \sin(\theta)\mathbf{N} \times \mathbf{P}.$$

This can be written in a matrix formulation:

$$Rot_{(\theta,\mathbf{N})} = \cos(\theta)\mathbf{I}_3 + (1 - \cos(\theta))\mathbf{N}^{\mathbf{T}}\mathbf{N} + \sin(\theta)\mathbf{A}_{\mathbf{N}},$$

with

$$\mathbf{N} = [N_1, N_2, N_3], \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{A}_{\mathbf{N}} = \begin{bmatrix} 0 & N_3 & -N_2 \\ -N_3 & 0 & N_1 \\ N_2 & -N_1 & 0 \end{bmatrix}$$

The geometry formulation presented above can be expressed in an algebraic formulation:

Let $p = (0, v) = xi + yj + zk$ be a *pure* quaternion.

Let $q = (c, u)$ be a *unit* quaternion.

We can define

$$\begin{aligned} R_q(p) &= qp\bar{q} \\ &= (c, u)p(c, -u) \\ &= (c, u)(0, v)(c, -u) \\ &= (c, u)(v \cdot u, -v \times u + \langle cv \rangle) \\ &= (c(v \cdot u) - u \cdot (-v \times u) - c(u \cdot v), u \times (-v \times u) \\ &\qquad + \langle c(u \times v) \rangle + \langle c(-v \times u + \langle cv \rangle) \rangle + (v \cdot u)u) \\ &= (0, -u \times (u \times v) + \langle 2c(u \times v) \rangle + \langle c^2 v \rangle + (v \cdot u)u) \\ &= (0, (u \cdot v)u - \langle (u \cdot u)v \rangle + \langle 2c(u \times v) \rangle \\ &\qquad + \langle c^2 v \rangle + \langle (v \cdot u)u \rangle) \\ &= (0, \langle (c^2 - u \cdot u)v \rangle + \langle 2(v \cdot u)u \rangle + \langle 2c(u \times v) \rangle). \end{aligned}$$

And because q is a unit quaternion, that is, $q = (c, u) = (c, s\mathbf{N})$ with $c = \cos(\theta)$ and $s = \sin(\theta)$,

$$R_q(p) = (0,\langle(c^2 - s^2)v\rangle + (2s^2(\mathbf{N} \cdot \mathrm{v})\mathbf{N},) + \langle 2sc(\mathbf{N} \times \mathrm{v})\rangle)$$

$$= (0, \langle\cos(2\theta)v\rangle + \langle(1 - \cos(2\theta))(\mathbf{N} \cdot v)\mathbf{N}\rangle + \langle\sin(2\theta)(\mathbf{N} \times v)\rangle)$$

R_q can be interpreted as the rotation of angle 2θ around the axis $\mathbf{N} = \mathrm{N}_1 i + \mathrm{N}_2 j + \mathrm{N}_3 k$. As a reciprocity, $Rot_{(q,\mathbf{N})}$ is represented by the quaternion

$$q = \cos(\theta/2) + \sin(\theta/2)\mathbf{N}$$

$$\mathbf{N} = \mathrm{N}_1 i + \mathrm{N}_2 j + \mathrm{N}_3 k$$

Notes

$$R_{qq'} = R_q \circ R_{q'}$$

$$R_{qq'}(p) = (qq')p(\overline{qq'})$$

$$= q(q'p\bar{q})\bar{q}'$$

$$= R_q(R_{q'}(p))$$

$$R_{qq'} = R_q \circ R_{q'}.$$

If we state that $-\pi \le \theta \le \pi$, then $-\pi/2 \le \theta/2 \le \pi/2$, and then

$$-q = -\cos(\theta/2) - \sin(\theta/2)\mathbf{N} = \cos(-\theta/2) + \sin(-\theta/2)(-\mathbf{N})$$

$$R_{-q} = Rot_{(-\theta,-\mathbf{N})} = Rot_{(\theta,\mathbf{N})}.$$

The algebraic formulation can be directly used in computing the displacement of orthonormal (orthogonal and normalized) coordinate systems.

If we make an analogy between the points $P = [x, y, z]$ of a nonhomogeneous space with the pure quaternions $p = xi + yj + zk$, it is possible to represent the displacements $Tr_{(\mathbf{U})} \circ + Rot_{(\theta,\mathbf{N})}$ of the space with the following functions:

$$p \rightarrow p' = M_{(u,r)}(p) = u + rp\bar{r},$$

with p, p', u being pure quaternions, and r a unit quaternion.

We can write

$$Tr_{(\mathbf{U})} = M_{(u,1)}: p' = p' + u$$

$$Rot_{(\theta,\mathbf{N})} = M_{(0,r)} = M_{(0,\cos(\theta/2) + \sin(\theta/2)\mathbf{N})}$$

$$Id = M_{(0,1)}: \textit{Identity transform.}$$

We can also define the property

$$M_{(u,r)} \circ M_{(u',r')}(p) = u + r(u' + r'p\bar{r}')\bar{r}$$

$$M_{(u,r)} \circ M_{(u',r')}(p) = (u + ru'\bar{r}) + (rr')p(\overline{rr'})$$

$$M_{(u,r)} \circ M_{(u',r')}(p) = M_{(u + ru'\bar{r}, rr')}.$$

We have defined a multiplication * over the elements (u, r):

$$(u, r) * (u', r') = (u + ru'\bar{r}, rr')$$

$$M_{(u,r)} * M_{(u',r')} = M_{(u,r)} \circ M_{(u',r')}.$$

We can now evaluate displacements in a 3D space using only algebraic elements (u, r).

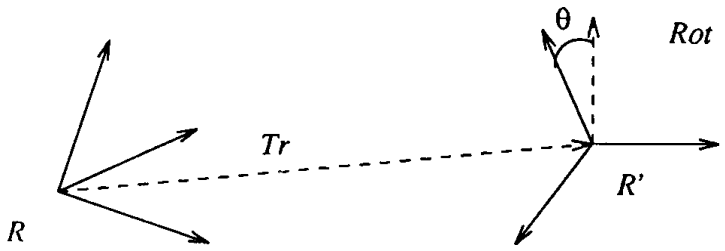

Figure 2. Movement from one orthonormal base to another.

By setting the initial coordinate system to *Id*, we can represent the movements of orthonormal bases (see Fig. 2).

$$R' = R \circ Tr_{(\mathbf{V})} \circ Rot_{(\theta,\mathbf{N})}$$

$$(u',r') = (u,r) * (v,\cos(\theta/2) + \sin(\theta/2)\mathbf{N}$$

We can also represent series of displacements as explained in Fig. 3.

$$(u,\ r) = (u_0,\ r_0) * (v_1,\ s_1) * (v_2,\ s_2) *. \ . .* (v_n,\ s_n),$$

where $(v_i,\ s_i) = (v_i,\ \cos(\theta_i/2) + \sin(\theta_i/2)\mathbf{N})$ represents the transformation $Tr_{(\mathbf{V})} \circ Rot_{(\theta,\mathbf{N})}$.

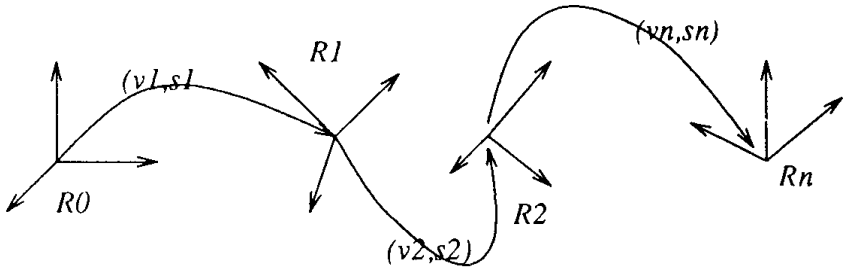

Figure 3. Series of movements from one orthonormal base to another.

Notes

Let R_0 and R_1 be two given orthonormal bases. The transformation from R_0 to R_1 has an algebraic representation:

$$(u_1,r_1) = (u_0,r_0) * (v,s)$$

$$(v,s) = (u_0,r_0)^{-1} * (u_1,r_1)$$

$$(v,s) = (-\bar{r}_0 u_0 r_0, \bar{r}_0) * (u_1,r_1)$$

$$(v,s) = (-\bar{r}_0 (u_1 - u_0) r_0, \bar{r}_0, r_1)$$

Algorithmic Implementation

The use of quaternions is a good way of coding 3D transformations. It requires less space than a 4×4 matrix (only 7 elements instead of 16) and the functions that are necessary for the implementation of a pre-/post-translation or a pre-/post-rotation are short and simple to code.

The pseudo-code examples that follow have been implemented in the *PATK3D* software, and have already been presented in a FORTRAN-like code (Maillot, 1986). A C language version is given in the appendix section. In the *PATK3D* package, the observer's position is initially [1,0,0], looking at the point $[0, 0, 0]$. There are three primitives that control the position and visualization direction of the observer.

- The first one sets the observer at a given position, $[x, y, z]$, and initializes the visualization direction to be in the X decreasing, that is, *looking in the X decreasing direction.*

- Another primitive can be used to set the visualization direction while not changing the observer position. It only needs a point in 3D, which is equivalent to saying: *look at the point* $[x, y, z]$.

- The last primitive is more general and gives to the *PATK3D* user the capability of moving in the 3D space using a displacement descriptor.

This descriptor is coded as a string where the user specifies a series of movements such as: *forward 10, turn right 20 degrees, pitch 30 degrees,* and so on.

Once all the observer's movements are described by the user, a 4×4 matrix is calculated using the observer's resulting quaternion. This matrix is accumulated with the other matrices of the visualization pipeline to produce the final matrix used in the single model-coordinate-to-device-coordinate transformation.

The functions presented here deal only with the kernel part of the observer's position and visualization direction primitives. We define the following structures:

P: **point** ← [– 1.0, 0.0, 0.0] is used to keep trace of the observer's position. It actually codes the orthonormal world coordinate base in respect to the observer.

Q: **array** [0..3] **of real** ← [1.0, 0.0, 0.0, 0.0] is the quaternion itself. The first element, *Q*[0], keeps the real part of the quaternion, while the three other elements represent the components of the pure part of the quaternion.

M: matrix4 is the matrix that will be calculated from *P* and *Q*. **M** will be coded like this:

$$M = \begin{bmatrix} 1 & tx & ty & tz \\ 0 & m_{1,1} & m_{1,2} & m_{1,3} \\ 0 & m_{2,1} & m_{2,2} & m_{2,3} \\ 0 & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$$

The *set_obs_position* primitive sets new values for the observer's eye's position in the space.

```
name: set_obs_position(position: point)
begin
    Set the values of the eye's position.
    The position here represents the position of the orthonormal base
    in respect to the observer.
    P ← –position;
    Set the visualization to be in the decreasing x-axis.
    Q ← [1.0, 0.0, 0.0, 0.0];
    end;
```

P and *Q* (Fig. 4) have the following values:

P: [– 2.0,0.0,0.0] ,

Q: [1.0,0.0,0.0,0.0].

The *translate_quarternion* function computes translation movements for a given quaternion. *i* is the axis of the translation, while *x* is the parameter that characterizes the translation for the quaternion. *w* should be set to –1 if the observer moves in respect to the scene, or to 1 if the scene moves in respect to the observer.

```
name: translate_quaternion(x: real, i,w: integer)
begin

    j, k: integer;
    A,B,D,E,F:real;

    Does the observer move in respect to the scene?
    if w < 0 then P[i – 1] ← P[i – 1] – x;
    else begin

        The scene moves in respect to the observer.
        Compute the successor axis of i [1, 2, 3];
        and then the successor axis of j [1, 2, 3];
        j ← i + 1;
        if j > 3 then j ← 1;
        k ← j + 1;
        if k > 3 then k ← 1;
        A ← Q[j]; B ← Q[k]; F ← Q[0]; E ← Q[i];
        P[i – 1] ← P[i – 1] + x*(E*E + F*F – A*A – B*B);
        D ← x + x;
        P[j – 1] ← P[j – 1] + D*(E*A + F*B);
        P[k – 1] ← P[k – 1] + D*(E*B + F*A);
        end;
end;
```

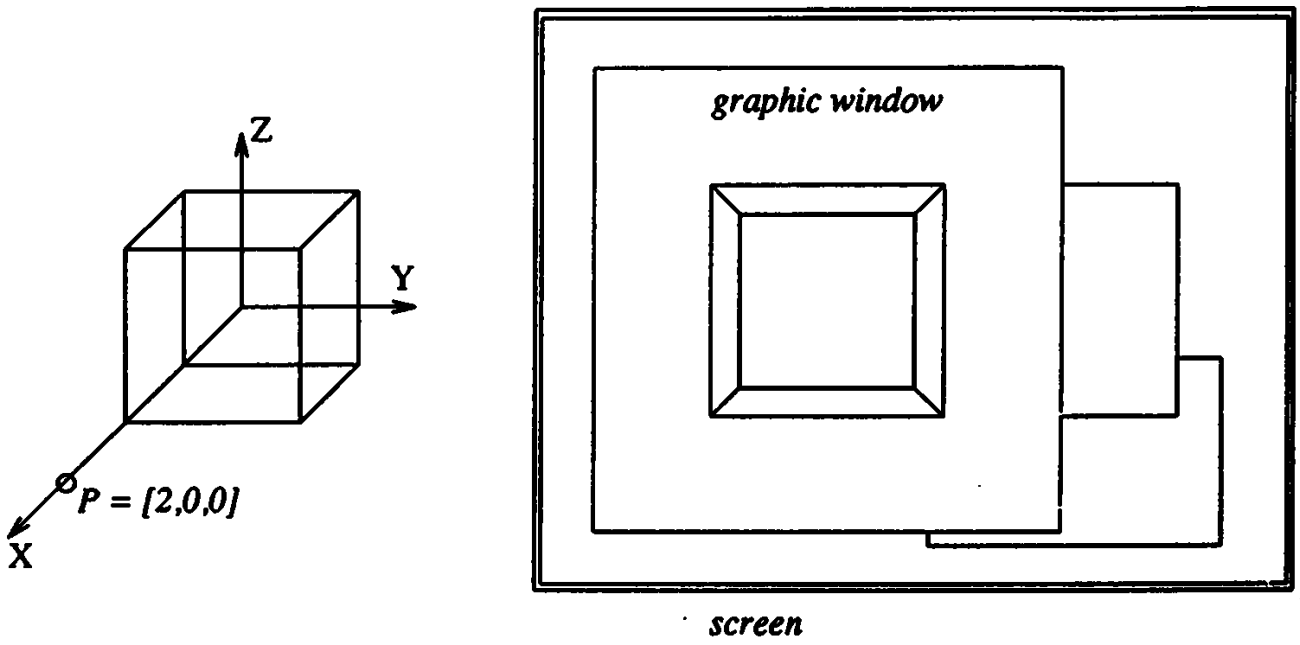

Figure 4. A simple graphic scene, and the user's screen.

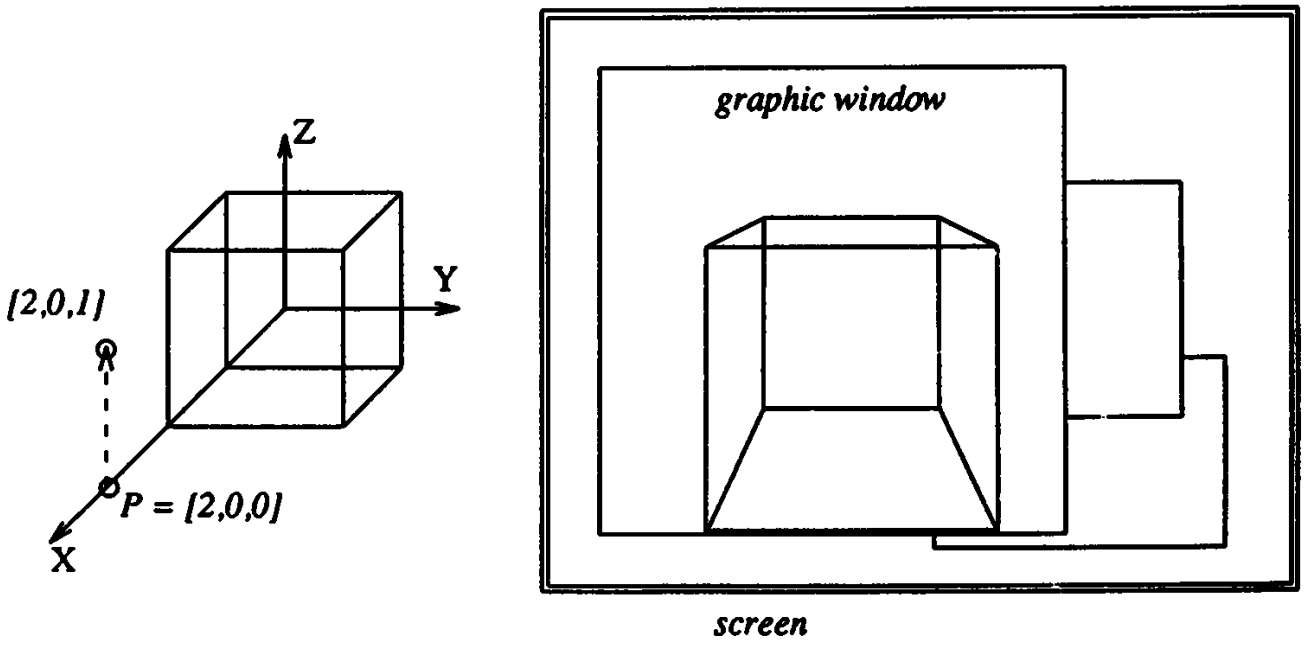

Figure 5. The result of a translation of vector [0, 0, 1].

P and *Q* (Fig. 5) get the following values:

P: [– 2.0, 0.0, – 1.0] ,

Q: [1.0, 0.0, 0.0, 0.0].

The *rotate_quaternion* function computes rotation based movements for a given quaternion. *i* is the axis number of the rotation, while *x* and *y* are the parameters that characterize the rotation for the quaternion. *w* follows the same rule as for *translate_quaternion*. *x* and *y* are typically the cosine and sine of the half-rotation angle.

```
name: rotate_quaternion(x, y: real, i, w: integer)
begin
    j, k: integer;
    E, F, R1 : real;
    Compute the successor axis of i [1, 2, 3,] and j [1, 2, 3];
    j ← i + 1;
    if j ← 3 then j ← 1;
    k ← j + 1;
    if k > 3 then k ← 1;
    E ← Q[i];
    Q[i] ← E*x + w*y*Q[0];
    Q[0] ← Q[0]*x – w*y*E;
    E ← Q[j];
    Q[j] ← E*x + y*Q[k];
    Q[k] ← Q[k]*x – y*E;
    if w < 0 then begin
            Compute a new position if the observer moves in respect to the scene.
            j ← j – 1; k ← k – 1;
            R1 ← x*x – y*y;
            F ← 2.*x*y;
            E ← P[j];
            P[j] ← E*R1 + F*P[k];
            P[k] ← P[k]*R – F*E;
            end;
    end;
```

P and *Q* (Fig. 6) get the following values:

P: [–2.24, 0.0, 0.0],

Q: [0.97, 0.0, 0.23, 0.0].

The *evaluate_matrix* primitive (re)computes the matrix corresponding to the observer's position and visualization direction given by *P* and *Q*. The method presented here is the direct application of the mathematical formulae. Faster ways to evaluate the matrix can be found.

```
name: evaluate_matrix()
begin
    e, f: real;
    r:array [0..3] of real;
    i, j, k: integer;
    We will need some square values!
    for i: integer ← 0, i < 4 do
     r[i] ← Q[i]*Q[i];
     i ← i + l;
     endloop;
    Compute each element of the matrix.
    j is the successor of i (in 1, 2, 3), where k is the successor of j.
    for i: integer ← 1, i < 4 dO
     j ← i + l;
     if j > 3 then j ← 1;
     k ← j + l;
     if k > 3 then k ← 1;
     e ← 2.*Q[i]*Q[j];
     f ← 2.*Q[k]*Q[0];
     M[j][i] ← e – f;
     M[i][j] ← e + f;
     M[i][i] ← r[i] + r[0] – r[j] – r[k];
     M[0][i] ← P[i – 1];
     M[i][0] ← 0.0;
     endloop;
    M[0][0] ← 1.0;
    end;
```

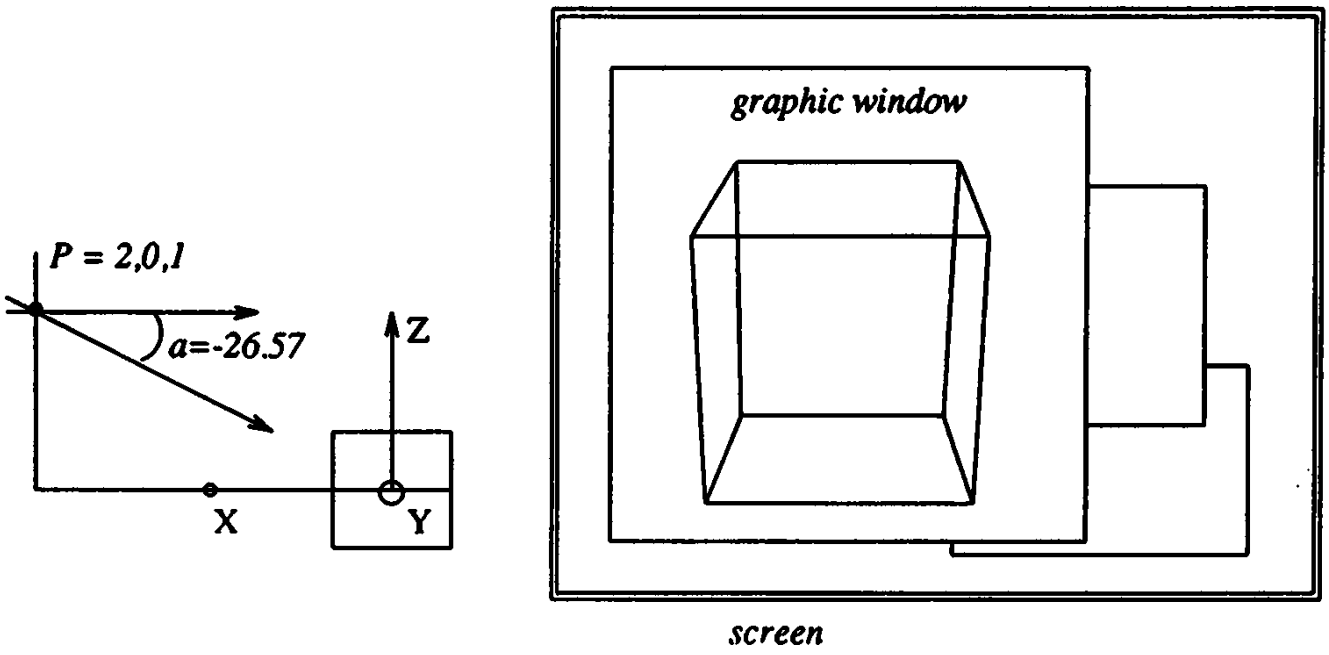

Figure 6. The result of a rotation angle –26.57 degrees around the y–axis to look at the point [0, 0, 0].

As an example, the matrix resulting from P and *Q* values of Fig. 6 is given below.

$$M = \begin{bmatrix} 1.0 & -2.24 & 0.0 & 0.0 \\ 0.0 & 0.89 & 0.0 & 0.45 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & -0.45 & 0.0 & 0.89 \end{bmatrix}$$

See also Rotation Matrix Methods Summary (455); Bit Patterns for Encoding Angles (442); Fast 2D–3D Rotation (440); Fixed-Point Trigonometry with CORDIC Iterations (494)

See Appendix 2 for C Implementation (775)