Béatrice Frey et Cyril Crassin **Automne 2003**

Projet Quaternions

Sommaire

Introduction

Le but ce projet d'*IA41* était de découvrir les quaternions, d'en implémenter les fonctions de calcul en *LISP* et de développer une application pour en illustrer l'utilisation.

Ne connaissant pas cette notion mathématique, notre première démarche fut de nous documenter à ce sujet afin d'en comprendre les intérêts et le fonctionnement. Cette documentation s'est dans un premier temps dirigée vers une recherche sur le web puis nous avons eu la chance d'avoir droit à un cours simplifié sur le sujet de la part de M. Martin, professeur de Mathématiques à *l'UTBM*.

A partir des connaissances ainsi acquises, nous avons pu répertorier les principales primitives nécessaires à la manipulation des quaternions. De là, nous avons établi les profils ainsi que les définitions formelles de ces fonctions que nous avons ensuite implémenté en LISP.

Cette étape terminée, nous avons cherché à comprendre l'intérêt et les applications des quaternions dans différents domaines. Il en est ressorti que cet outil est principalement utilisé en mécanique et en infographie pour représenter des rotations. L'image présente en page de garde est issue d'une autre application plus mathématique et artistique des quaternions : la création d'images fractales.

1. Présentation des Quaternions

Les quaternions furent découverts en *1843* par l'Irlandais William Rowan *HAMILTON* (1805-1865). En faisant des calculs avec les nombres complexes, il s'intéressa à l'interprétation géométrique de l'addition et de la multiplication dans le plan à 2 dimensions. Il se demanda si avec des nombres hypercomplexes, on ne pourrait pas obtenir des résultats analogues dans l'espace à 3 dimensions.

Ses recherches sont bien décrites par la célèbre anecdote avec son fils : Durant de nombreuses années, Hamilton avait espéré trouver une forme de multiplication satisfaisante pour les triplets de nombres réels avec de bonnes propriétés. En particulier: conservation de la multiplication des modules.

Peu de temps avant sa mort en 1865, il écrivit à son fils:

"Tous les matins, alors que tu descendais pour prendre le petit déjeuner, tu me demandais : 'Eh bien, papa, est-ce que tu peux multiplier les triplets ?' J'étais toujours obligé de répondre, avec un triste hochement de tête : 'Non, je peux seulement les ajouter et les soustraire'."

Finalement, Hamilton a l'idée géniale de passer à un paramètre de plus :

$a + ib + jc$	$a + ib + jc + kd$
Ne semble pas marcher	Essayons avec une composante de plus Il passe dans la 4^e dimension pour calculer les triplets

"Ainsi naquit l'idée en moi d'admettre une quatrième dimension pour calculer avec des triplets."

Il aboutit aux quaternions :

- en imposant de respecter la multiplication des modules,
- en conservant l'associativité,
- mais, hélas, il est obliger d'abandonner la commutativité

2. Définitions mathématiques

Les quaternions sont formés d'une quantité (un scalaire) et d'un vecteur (un point dans l'espace).

Un quaternion q est donc un nombre complexe formé de 4 composantes :

q = a + ib + jc + kd avec a, b, c, d des nombres réels et i, j, k des coefficients imaginaires.

- ***Egalité des quaternions***

  Deux quaternions q = a + ib + jc + kd et q' = a' + ib' + jc' + kd' sont égaux si et seulement si on a :

  a = a': b = b'; c = c'; d = d'.

- ***Propriétés des coefficients***

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

  ij = k = -ji

  jk = i = -kj

  ki = j = -ik

Tableau de multiplication des coefficients entre eux :
(minuscules et majuscules par commodité, car non commutatif)

Non commutatif

xX =	**I**	**J**	**K**
i	-1	k	-j
j	-k	-1	i
k	j	-i	-1

Exemples de lecture:

i I = -1

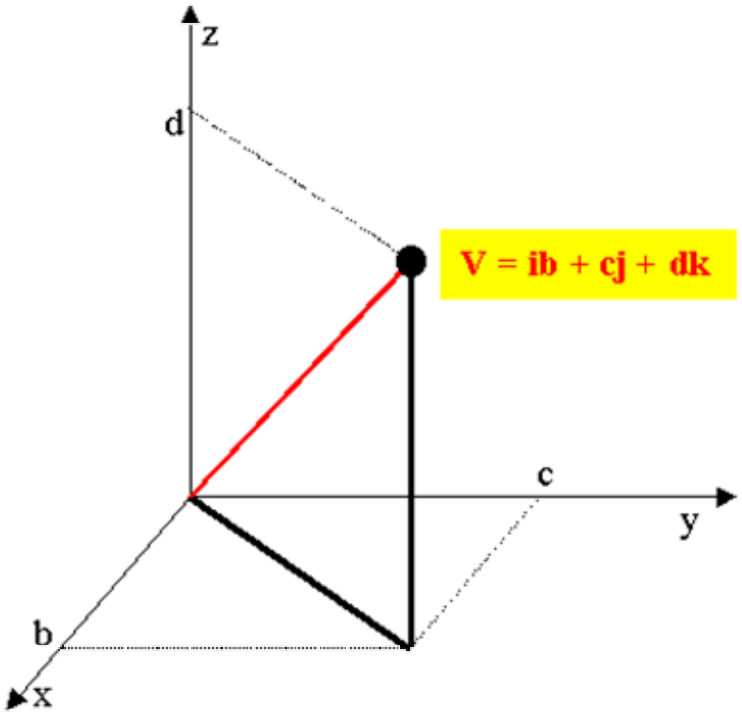
I i = -1

Mais,

i J = k

I j = - k

- ***Représentation d'un quaternion***



- ***Rotations***

Les rotations dans l'espace peuvent être décrites par les quaternions. Formulation générale : (ib + jc + kd) ® q^{-1} (ib + jc + kd) q avec q le quaternion adéquate pour effectuer la rotation désirée.

Exemples

Rotation de	Multiplication par
180°	q = k
90°	q = (1 + k) / √ 2

Rotation particulière d'un angle 2*o* alors q = cos*o* + (bi + cj + dk) sin*o*

- ***Multiplication des quaternions***

Etant donné deux quaternions q = t + xi + yj + zk et q' = t' + x'i + y'j + z'k, leur produit qq' est donné par :

qq' = tt' - xx' - yy' - zz'
+ (tx' + t'x + yz' - y'z) i
+ (ty' + t'y + x'z - xz') j
+ (tz' + t'z + xy' + x'y) k

Remarque : la multiplication n'est pas commutative !

- ***Propriétés des quaternions***

CONJUGUE

$q^* = a - bi - cj - dk$

On peut vérifier que $qq^* = a^2 + b^2 + c^2 + d^2$

NORME

$||q|| = \text{sqrt}(qq^*)$

INVERSE

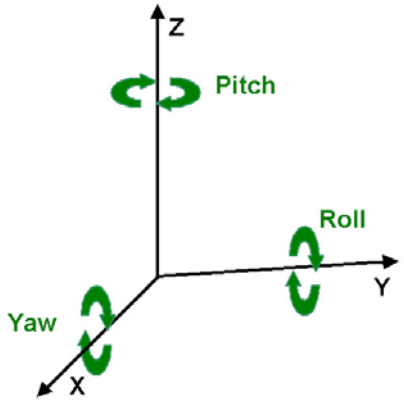
$q^{-1} = q^*/ (||q||^2)$

3. Applications en Infographie

3.1 Rotations dans l'espace

Comme nous l'avons signalé précédemment les *Quaternions* trouvent leur principale application en *infographie,* et plus précisément dans le domaine de l'animation *3D,* pour la représentation de rotations dans l'espace. Seulement ce n'est pas la seule technique qui permet de manipuler des rotations et nous allons voir en quoi les Quaternions sont meilleurs pour réaliser ce genre d'opérations.

3.2 Les angles d'Euler



Les angles d'*Euler* sont le nom donné à un ensemble d'angles qui spécifient chacun une rotation autour des axes X, Y et Z qui combinés définissent une rotation arbitraire dans l'espace. Il s'agit de la représentation la plus couramment utilisée car elle se manipule très simplement à l'aide d'outils extrêmement bien maîtrisés telles que les matrices que nous aborderons dans la partie suivante. De plus, c'est la représentation la plus intuitive quand on veut générer des rotations à partir d'entrées utilisateur, que ce soit au clavier ou à la souris. Il suffit en effet de faire correspondre des touches ou des directions de souris à chaque axe et de faire varier les valeurs des angles en fonctions de ces interactions.

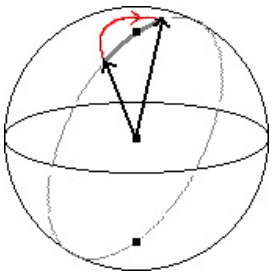
Ces angles sont spécifiés sous la forme d'un vecteur |x y z| et peuvent être stockés comme une structure de type *Vecteur* ce qui est économique en terme d'espace de stockage.

Voici quelques exemples d'angles d'*Euler*:

| 0 0 0 | Rotation nulle (générera la matrice identité).
| 90 0 0 | Rotation de 90° autour de l'axe X
| 0 90 0 | Rotation de 90° autour de l'axe Y
| 10 160 90 | Rotation de 10° autour de l'axe X, 160 autour de Y et 90° autour de Z.

Le problème de cette technique est que pour calculer la position finale d'un objet dans l'espace après application de la rotation, il faut appliquer successivement (dans un

ordre prédéterminé) les rotations autour de chaque axe. Cette opération est à la fois relativement coûteuse en calcul puisqu'elle implique plusieurs opérations trigonométriques et entraîne un résultat imprécis. De plus, elle rend les interpolations entres angles extrêmement compliquées car elle entraîne des trajets fantaisistes entre deux orientations.



Trajet (en rouge) suivi par l'interpolation de deux rotations par les angles d'Euler

Un autre problème de cette méthode est la perte d'un degré de liberté ou *Gimbal Lock* en anglais. Comme la position finale d'un objet dépend de l'ordre d'application des rotations suivant les 3 axes, il arrive que parfois l'une des rotations autour d'un axe se confonde avec un autre axe de rotation ce qui entraîne l'impossibilité de tourner l'objet selon le troisième axe ainsi perdu.

Par exemple, supposons que l'on oriente un objet à l'aide d'angles d'Euler en appliquant dans l'ordre les rotations selon les axes X, Y puis Z. Dans ce cas, la rotation selon X se fait correctement, puis viens la rotation selon Y qui est de 90° de telle sorte que l'axe Z ainsi généré vienne se confondre avec l'axe X autour duquel on a effectué la première rotation. Dans ce cas, la troisième rotation selon l'axe Z s'effectue en réalité autour du même axe que la première rotation, c'est-à-dire l'axe X ! On a bel et bien perdu la possibilité d'effectuer une rotation selon l'axe Z. La seule solution de remédier à ce problème est d'utiliser les quaternions.

3.3 Les matrices

La matrice de rotation est l'outil le plus couramment utilisé en animation 3D pour représenter des rotations. Les matrices permettent en effet de représenter et de combiner toutes les transformations (Rotation, Translation et Mise à l'échelle) applicables aux vecteurs dans l'espace. Elles sont intégrées à toutes les *API* graphiques et leur manipulation est même accélérée par certains composants hardware (sur *PC* en tout cas) tels que les *cartes 3D* ou les instructions avancées (*SSE*, *3Dnow*) de certains processeurs. Leur grand avantage est qu'elles unifient toutes les transformations dans l'espace mais les matrices de rotation sont soumises aux limitations liées aux angles d'Euler auxquels elles se rapportent. De plus les matrices sont extrêmement coûteuses en espace mémoire vu qu'elles sont stockées sous la forme de tableaux bidimensionnels de taille 3x3 ou 4x4 (Coordonnées homogènes).

A la base, les matrices de rotations sont un moyen simple et efficace d'appliquer des rotations d'angle d'Euler à un vecteur de l'espace. On définie une matrice par axe de rotation (X, Y et Z) et toutes les rotations sont définies à l'aide des fonctions

trigonométriques sinus et cosinus dont on enregistre le résultat de l'application sur l'angle impliqué dans la matrice.

Par exemple, pour un système de coordonnées à deux dimensions, la matrice de rotation est la suivante:

$$\begin{vmatrix} \cos(A) & -\sin(A) \\ \sin(A) & \cos(A) \end{vmatrix}$$

Quand l'angle A est nul, on génère la matrice identité:

$$I = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

Si A vaut +90 degrés, on a la matrice:

$$M = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

Et si l'angle vaut -90 degrés:

$$M = \begin{vmatrix} 0 & 1 \\ -1 & 0 \end{vmatrix}$$

En 3D, on obtient par exemple pour l'axe X la matrice 3x3 :

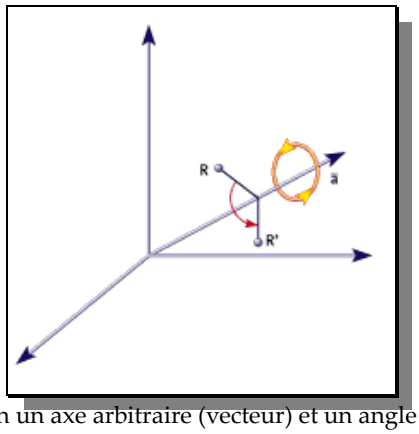
$$Rx = \begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos(A) & -\sin(A) \\ 0 & \sin(A) & \cos(A) \end{vmatrix}$$

On combine ensuite les matrices définies pour chaque axe en une unique matrice de rotation par simples produits matriciels (Il existe tout de mêmes des formules pour simplifier les calculs et trouver directement la matrice résultat). Mais attention car ici aussi l'ordre de combinaison des matrices est importante et influe sur le résultat final. Le G*imbal lock* sera donc bel et bien présent ici aussi !

Pour appliquer la rotation à un vecteur, il suffit ensuite de multiplier ce dernier par la matrice de rotation et obtenir ainsi la position rotée.

3.4 Les Quaternions

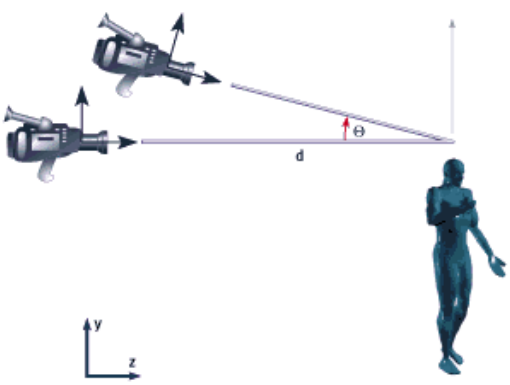
Le principe de représentation des rotations utilisé avec les quaternions est à opposer totalement avec les angles d'Euler. Pour stocker une rotation dans un *Quaternion*, on utilise non plus une série de rotation successives autours de chaque axe mais un vecteur arbitraire et un angle quelconque de rotation autour de ce vecteur. La partie réelle du *Quaternion* contenant l'angle, la partie imaginaire le vecteur 3D et le *Quaternion* étant obligatoirement unitaire, c'est à dire de norme 1.



Rotation selon un axe arbitraire (vecteur) et un angle quelconque

Les Quaternions offrent d'énormes avantages, le premier est la place mémoire relativement faible qu'ils occupent à comparer avec celle que nécessitent les matrices. Leur second avantage est la facilité qu'ils offrent pour la combinaison de rotations. Tout comme les matrices, les Quaternions de rotation se combinent par multiplication mais cette fois se sont bien des rotations arbitraires qui sont combinées et le nombre d'opérations nécessaires à cette combinaison est bien inférieur à celui requis pour combiner des matrices. Une telle économie de calculs est extrêmement importante lorsque l'on travaille par exemple avec la représentation d'objets hiérarchiques (L'animation d'un *squelette*) ou de l'*inverse kinematics*. De plus, la représentation *angle-plus-axe* ne souffre pas du problème du *Gimble Lock* et l'utilisation des *Quaternions* permet ainsi d'en venir à bout.

Un autre avantage des *Quaternion* est la facilitée et la précision qu'ils offrent en terme d'interpolation entre deux rotations. Il existe en effet des techniques telles que *SLERP* (*Spherical Linear intERPolation*) qui permettent d'obtenir des interpolations de rotations extrêmement réalistes. De telles interpolations peuvent êtres utilisées par exemple dans un jeu vidéo pour faire suivre un personnage par une camera en réponse à des actions du joueur. Imaginons que le joueur se retourne d'1/2 tours en appuyant sur une touche, on calcul la rotation finale de la camera pour la recentrer sur le personnage puis on effectue un interpolation afin que la camera prenne sa place de manière fluide et progressive.



Pour les calculs de positions à l'aide de Quaternions, on stocke la position initiale de l'objet dans un quaternion imaginaire pur (sans partie réelle) de la forme :

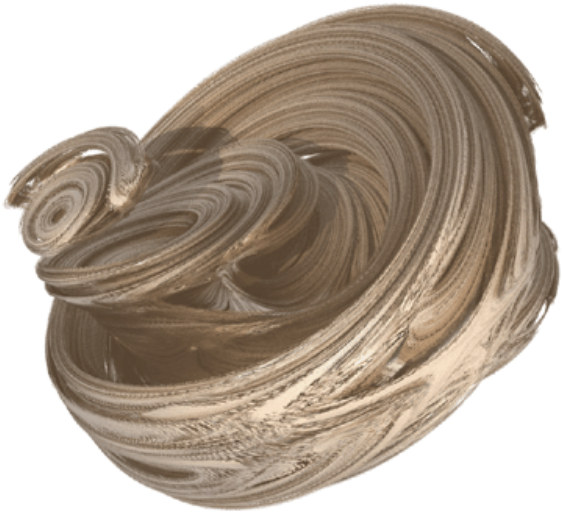
qPos=[0, v] ou qPos={0, v.x, v.y, v.z} avec v vecteur de position de l'objet.

L'application de la rotation se fait ensuite par la formule suivante :

qPos'= qRot * qPos * qRot^-1, dans laquelle on peut remplacer qRot^-1 par conj(qRot), le conjugué car un Quaternion de rotation est toujours unitaire.

Il suffit ensuite d'extraire les composantes imaginaires de qPos' qui représentent la position rotée de l'objet.

3.5 Les Fractales



Même si il s'agit d'une utilisation des *Quaternions* beaucoup moins courante et qui n'offre pour l'instant que peu d'applications en dehors de l'intérêt visuel de la chose (mais qui sait, peut être qu'un jour…), la création d'images fractales tridimensionnelles à l'aide de Quaternions fournit tout de même des résultats extrêmement impressionnants.

Le principe est le même que pour le création de fractales de type *Julia* ou *Mandelbrot* en 2D seulement on utilise des *Quaternion* à la place des *Complexes*. Les Quaternions possédant quatre composantes et comme nous ne savons représenter facilement que des images en trois dimensions, la quatrième composante du *Quaternion* est soit fixée à une constante soit faite variée au cours du temps ce qui crées des animations assez intéressantes.

Pour rappel, la création d'une image fractale se fait par itération d'une suite, soit *Complexe* soit sur les *Quaternions* dans notre cas, pour chaque pixel de l'image. Si cette suite diverge, c'est qu'elle n'appartient pas à l'ensemble testé et si elle converge c'est que le pixel de l'image correspondant forme la fractale. On attribue ensuite les couleurs en fonction du nombre d'itération qu'il a été nécessaire d'effectuer pour conclure.

Les fonctions non linéaires sont de loin les plus intéressantes et on utilisera par exemple pour l'ensemble de Julia la fonction $Zn+1 = (Zn)^2 + C$, avec Z0 point de l'espace correspondant au pixel testé et C une constante qui identifie le *Quaternion* particulier.

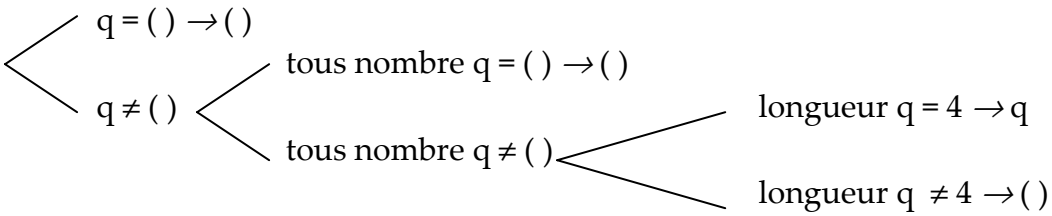
4. Primitives sur les quaternions

4.1 quat : (a b c d)

profil : List → List
q

spécifications : Teste si la liste est un quaternion. Retourne la liste q si elle est non vide, si sa longueur vaut 4 et si tous ses éléments sont des nombres, retourne nil sinon.

définition formelle :



code lisp associé :

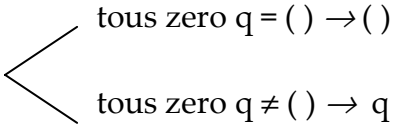
```
(defun quat (q)
  (if (or (null q) (null (every 'numberp q)) (not (equal (length q)
4)))
      nil
      q
  ))
```

4.2 quatNul : (0 0 0 0)

profil : List → List
q

spécifications : Retourne la liste q si c'est le quaternion nul (0 0 0 0), rend nil sinon.

définition formelle :



code lisp associé :

```
(defun quatNul (q)
(if (every 'zerop q)
      q
      nil))
```

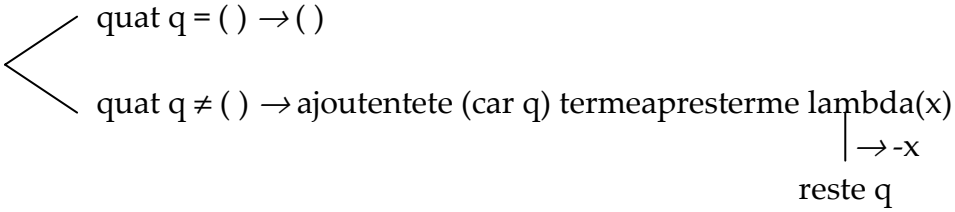
4.3 conj : (a -b -c -d)

profil : List → List

q

spécifications : Retourne le conjugué du quaternion si la liste q est un quaternion, rend nil sinon.

définition formelle :



code lisp associé :

```
(defun conj (q)
  (if (null (quat q))
      nil
      (cons (car q) (mapcar (lambda (x)
                        (- 0 x)
                        )
                        (cdr q)))))
```

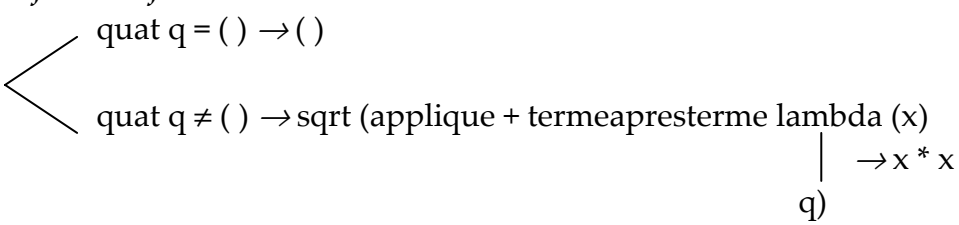
4.4 norme

profil : List → Nombre

q

spécifications : Retourne la norme du quaternion si la liste q est un quaternion, rend nil sinon.

définition formelle :



code lisp associé :

Sous lisp, pour calculer la norme, nous sommes passé par une fonction qui donne la norme au carré pour ensuite faire un sqrt sur le résultat.

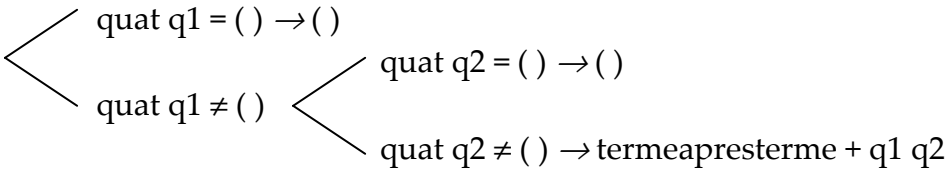
```
;norme * norme
(defun squareNorme (q)
  (if (null (quat q))
      nil
      (apply '+ (mapcar (lambda (x)
                          (* x x)
                        )
                  q))))

;norme
(defun norme (q)
  (sqrt (squareNorme q))
)
```

4.5 add : q1 = a1 + ib1 + jc1 + kd1 ; q2 = a 2+ ib2 + jc2 + kd2
q1 + q2 = a1 + a2 + (b1 + b2)i + (c1 + c2)j + (d1 + d2)k

profil : List * List → List
q1 q2

spécifications : Retourne l'addition de deux quaternions si les deux listes q1 et q2 sont des quaternions, rend nil sinon.

définition formelle :



code lisp associé :

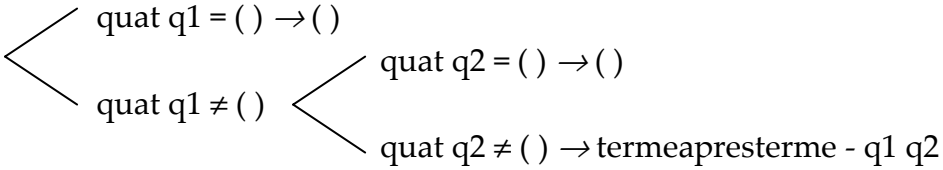
```
(defun add (q1 q2)
  (if (or (null q1) (null q2))
      nil
      (mapcar '+ q1 q2)
   ))
```

4.6 sub : q1 = a1 + ib1 + jc1 + kd1 ; q2 = a 2+ ib2 + jc2 + kd2 q1 - q2 = a1 - a2 + (b1 - b2)i + (c1 - c2)j + (d1 - d2)k

profil : List * List → List
q1 q2

spécifications : Retourne la soustraction de deux quaternions si les deux listes q1 et q2 sont des quaternions, rend nil sinon.

définition formelle :



code lisp associé :

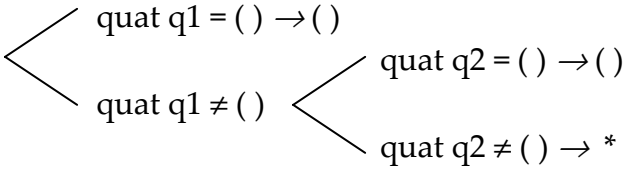
```
(defun sub (q1 q2)
  (if (or (null q1) (null q2))
      nil
      (mapcar '- q1 q2)
   ))
```

4.7 mult : multiplication de deux quaternions

profil : List * List → List
q1 q2

spécifications : Retourne la multiplication de deux quaternions si les deux listes q1 et q2 sont des quaternions, rend nil sinon.

définition formelle :



* miseenliste (magic q1 q2 '(1 -1 -1 -1))
(magic q1 (permut (permut q2 0 1) 2 3) '(1 1 1 -1))
(magic q1 (permut (permut q2 0 2) 1 3) '(1 -1 1 1))
(magic q1 (permut (permut q2 1 2) 0 3) '(1 1 -1 1))

Pour effectuer la multiplication on utilise une fonction "magic" qui retourne applique + (termeapresterme * q1 q2 n) avec q1, q2 et n des listes.

code lisp associé :

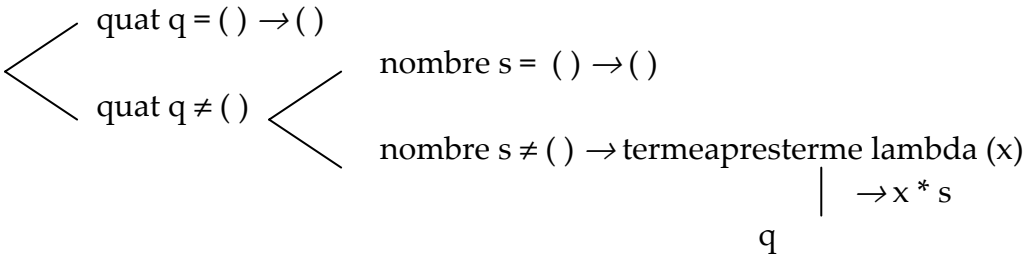
```
(defun magic (q1 q2 l)
  (apply '+ (mapcar '* q1 q2 l)))


(defun mult (q1 q2)
  (if (not (and (quat q1) (quat q2) ) ) ;prise de tete de reste
      nil
      (list (magic q1 q2 '(1 -1 -1 -1))
    (magic q1 (permut (permut q2 0 1) 2 3) '(1 1 1 -1))
    (magic q1 (permut (permut q2 0 2) 1 3) '(1 -1 1 1))
    (magic q1 (permut (permut q2 1 2) 0 3) '(1 1 -1 1))
      )
  ))
```

4.8 multScal : multiplication d'un quaternion par un scalaire

profil : List * Nombre → List
q s

spécifications : Retourne le résultat de la multiplication du quaternion par le scalaire si la liste q est un quaternion et si s est un nombre, rend nil sinon.

définition formelle :



code lisp associé :

```
(defun multScal(q s)
  (if (or (not (numberp s)) (not (quat q)))
      nil
      (mapcar (lambda(x)
                (* x s))
              q
      )))
```

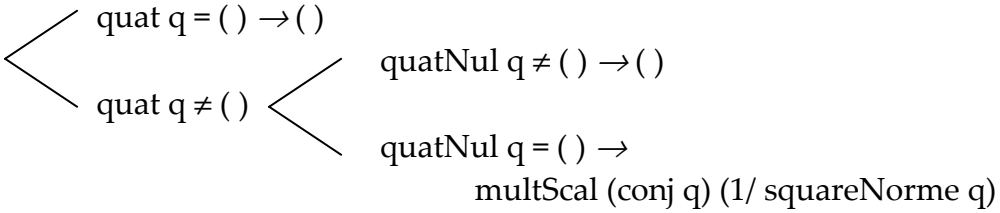
4.9 inv : $q^{-1} = q^*/ (||q||^2)$

profil : List → List

q

spécifications : Retourne l'inverse du quaternion si la liste q est bien un quaternion et si ce n'est pas le quaternion nul, rend nil sinon

définition formelle :



code lisp associé :

```
(defun inv (q)
  (if (or (null (quat q)) (not (null (quatNul q))))
      nil
      (multScal (conj q) (/ 1 (squareNorme q)))
  ))
```

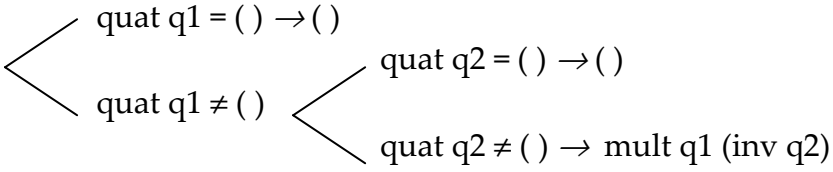
4.10 div : division de deux quaternions

profil : List * List → List

q1 q2

spécifications : Retourne la division de deux quaternions si les listes q1 et q2 sont des quaternions, rend nil sinon.

définition formelle :



code lisp associé :

```
(defun div (q1 q2)
  (mult q1 (inv q2))
)
```

4.11 normalise : ||q|| = 1

profil : List → List

q

spécifications : Retourne un quaternion de norme 1.

définition formelle :

→ divScal q (norme q)

avec divScal (q s) → multScal q (1/s)

code lisp associé :

```
(defun divScal (q s)
  (multScal q (/ 1 s)))

(defun normalise (q)
  (divScal q (norme q )))
```

4.12 quat2mat : passage d'un quaternion à une matrice 4*4

formule :

$$M = \begin{pmatrix} 1 - 2Y^2 - 2Z^2 & 2XY - 2ZW & 2XZ + 2YW \\ 2XY + 2ZW & 1 - 2X^2 - 2Z^2 & 2YZ - 2XW \\ 2XZ - 2YW & 2YZ + 2XW & 1 - 2X^2 - 2Y^2 \end{pmatrix}$$

profil : List → List

q

spécifications : Transforme un quaternion de rotation en matrice 4x4 de rotation si la liste q est un quaternion, rend nil sinon.

définition formelle :

quat q = ( ) → ( )

quat q ≠ ( ) → *voir formule*

code lisp associé :

```
(defun quat2mat (q)
 (if (null (quat q))
   nil
   (list
   (- 1 (* 2 (* (nth 2 q) (nth 2 q))) (* 2 (* (nth 3 q) (nth 3 q))))
   (+ (* 2 (nth 1 q) (nth 2 q)) (* 2 (nth 3 q) (nth 0 q)))
   (- (* 2 (nth 1 q) (nth 3 q)) (* 2 (nth 2 q) (nth 0 q)))
   0.0
   (- (* 2 (nth 1 q) (nth 2 q)) (* 2 (nth 3 q) (nth 0 q)))
   (- 1 (* 2 (nth 1 q) (nth 1 q)) (* 2 (nth 3 q) (nth 3 q)))
   (+ (* 2 (nth 2 q) (nth 3 q)) (* 2 (nth 1 q) (nth 0 q)))
   0.0
   (+ (* 2 (nth 1 q) (nth 3 q)) (* 2 (nth 2 q) (nth 0 q)))
   (- (* 2 (nth 2 q) (nth 3 q)) (* 2 (nth 1 q) (nth 0 q)))
   (- 1 (* 2 (nth 1 q) (nth 1 q)) (* 2 (nth 2 q) (nth 2 q)))
   0.0
   0.0 0.0 0.0 1.0
 )
))
```

4.13 angle2quat

formule :

```
    sin_a = sin( angle / 2 )
    cos_a = cos( angle / 2 )

   q -> x    = axis -> x * sin_a
   q -> y    = axis -> y * sin_a
   q -> z    = axis -> z * sin_a
   q -> w    = cos_a

   normalise( q );
```

profil : Nombre^4 $\rightarrow$ List

angle * aX * aY * aZ

spécifications : Retourne un quaternion formé à partir d'un axe et de l'angle de rotation autour de l'axe.

définition formelle : Voir formule (abracadabra !?)

code lisp associé :

```
(defun angle2quat (angle aX aY aZ)
  (list (cos (/ angle 2))
        (* aX (sin (/ angle 2)))
        (* aY (sin (/ angle 2)))
        (* aZ (sin (/ angle 2)))
  )
)
```

4.14 : euler2quat

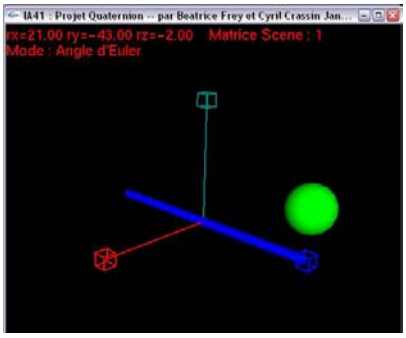
profil : Nombre^3 → List
eX * eY * eZ

spécifications : Convertit un triplet d'angles d'Euler en quaternion.

code lisp associé :

```
(defun euler2quat_calc (eX eY eZ)
  (list (+ (* (cos ex)(cos ey)(cos ez))(* (sin ex)(sin ey)(sin ez)))
  (- (* (sin ex) (cos ey) (cos ez)) (* (cos ex) (sin ey) (sin ez)))
  (+ (* (cos ex) (sin ey) (cos ez)) (* (sin ex) (cos ey) (sin ez)) )
  (- (* (cos ex) (cos ey) (sin ez)) (* (sin ex) (sin ey) (cos ez)) )
  )
)

(defun euler2quat (aX aY aZ)
    (euler2quat_calc (/ aX 2) (/ aY 2) (/ aZ 2))
)
```

5. Notre application



5.1 Présentation

Notre programme de démonstration montre l'intérêt des *Quaternions* pour la manipulation de rotations dans l'espace. Il permet de manipuler un bâton en 3D en lui appliquant des rotations à la souris et en choisissant la méthode de calcul de la rotation, soit par angles d'Euler, soit par *Quaternions*. On observe ainsi parfaitement la différence entre les deux méthodes et particulièrement le problème du *Gimbal Lock*. Une deuxième rotation par *Quaternion* est également mise en œuvre par l'affichage d'une sphère tournant autour du bâton et que l'on peut déplacer au clavier.

5.2 L'interpréteur LISP

L'interpréteur LISP que nous avons utilisé pour notre projet ce nome CLISP. Il s'agit d'une implémentation de l'ANSI *Common Lisp* en Open Source sous licence *GNU*. Cet interpréteur à l'avantage d'être libre, multi-plateforme et offre un grand nombre de fonctionnalités tel que la pré-compilation du code LISP pour de meilleures performances. Mais si nous avons choisit de l'utiliser c'est surtout parce qu'il propose une fonctionnalité qui nous intéressait particulièrement pour ce projet : les *FFI* pour *Foreign Function Interfaces*. Ce mécanisme donne en effet la possibilité d'appeler du code C directement à l'intérieur d'un programme LISP et, réciproquement, du code LISP à partir du C et ce de manière très simple. Il existe d'autres implémentations LISP telle que CMUCL qui proposent ce type de fonctionnalité mais celle-ci nous a parue être la plus simple à mettre en œuvre.

5.3 Le programme

Notre but était de créer une application capable d'utiliser une API 3D, afin de ne pas avoir à tout recréer nous même, tout en conservant la librairie de manipulation de *Quaternion* en LISP que nous avions écrite. Nous avons ainsi fait le choix de nous orienter vers une application écrite en C++ pour l'affichage et l'interaction avec l'utilisateur et capable d'interagir avec du code écris en LISP pour la manipulation des *Quaternions*. Pour cela, nous avons fait le choix technique de créer une classe *Quaternion*, manipulée par notre application en *C++* et qui fait directement appel pour ses traitements au code LISP via les FFI de CLISP.

Pour l'affichage, notre application utilise l'API (*Application Programming Interface*) 3D *OpenGL* qui a l'avantage d'être multi-plateforme et dont les primitives sont accélérées par les matériels la supportant. Cette bibliothèque offre des fonctions de bases pour l'affichage d'objets tridimensionnels.

Notre classe C++/LISP nous a également permis de réaliser une application de création et de visualisation de fractales basées sur les *Quaternions* ainsi qu'un petit exemple d'interpolation SLERP adapté d'un exemple trouvé sur Internet. Notre objectif a été tout au long du développement de créer un code évolutif et réutilisable afin de disposer d'un outil efficace pour exposer plusieurs utilisations des Quaternions.

5.4 Organisation des fichiers et utilisation

Notre application est constituée de 3 fichiers LISP :

- **QuatAppli.lisp** : Point de départ du programme, initialisation et appel C
- **c_quat.lisp** : Chargé dans *QuatDll.lisp*, déclare les fonctions intermédiaires appelées à partir du C, voir plus loin.
- **quat.lisp** : Contient toutes les fonctions de manipulation des quaternions.

Ainsi que de 6 fichiers C++ :

- **QuatMain.cpp** : La fonction de départ du programme C (*quatMain*).
- **QuatAppli.cpp** : Contient les fonctions liées à la démo elle-même.
- **Quaternion.h/Quaternion.cpp** : Contient la déclaration de classe et l'implémentation des méthodes de la classe *Quaternion*.
- **QuatAppli.h** : Prototypes des fonctions de la démo
- **Common.h** : Diverses choses nécessaires à la démo

Voici les commandes pour compiler une librairie C++ partagée sous *Linux* et *Solaris* (SunRays). Attention car l'argument pour créer une librairie partagée n'est pas le même sur les deux plateformes :

```
Linux:
g++ QuatAppli.cpp Quaternion.cpp QuatMain.cpp -o quatAppli.so -shared –lglut

Solaris:
g++ QuatAppli.cpp Quaternion.cpp QuatMain.cpp /usr/local/glut-
3.7/lib/libglut.a -I/usr/local/glut-3.7/include/ -o quatAppli.so –G -lGLU -lGL
-lXmu -lX11 -lm
```

L'application nécessite bien entendu l'installation de CLISP, d'une implémentation d'OpenGL (MESA par exemple) et de GLUT (tout est maintenant présent sur Sunserv2). Toutes les librairies que nous utilisons étant portables, notre programme fonctionne aussi bien sous *Linux*, *Solaris* ou *Windows*. Nous avons fournit un script d'exécution appelé *quatAppli.sh* qui permet de compiler et de lancer le programme sous UNIX.

La manipulation de l'application se fait au clavier et à la souris et voici les commandes à utiliser :

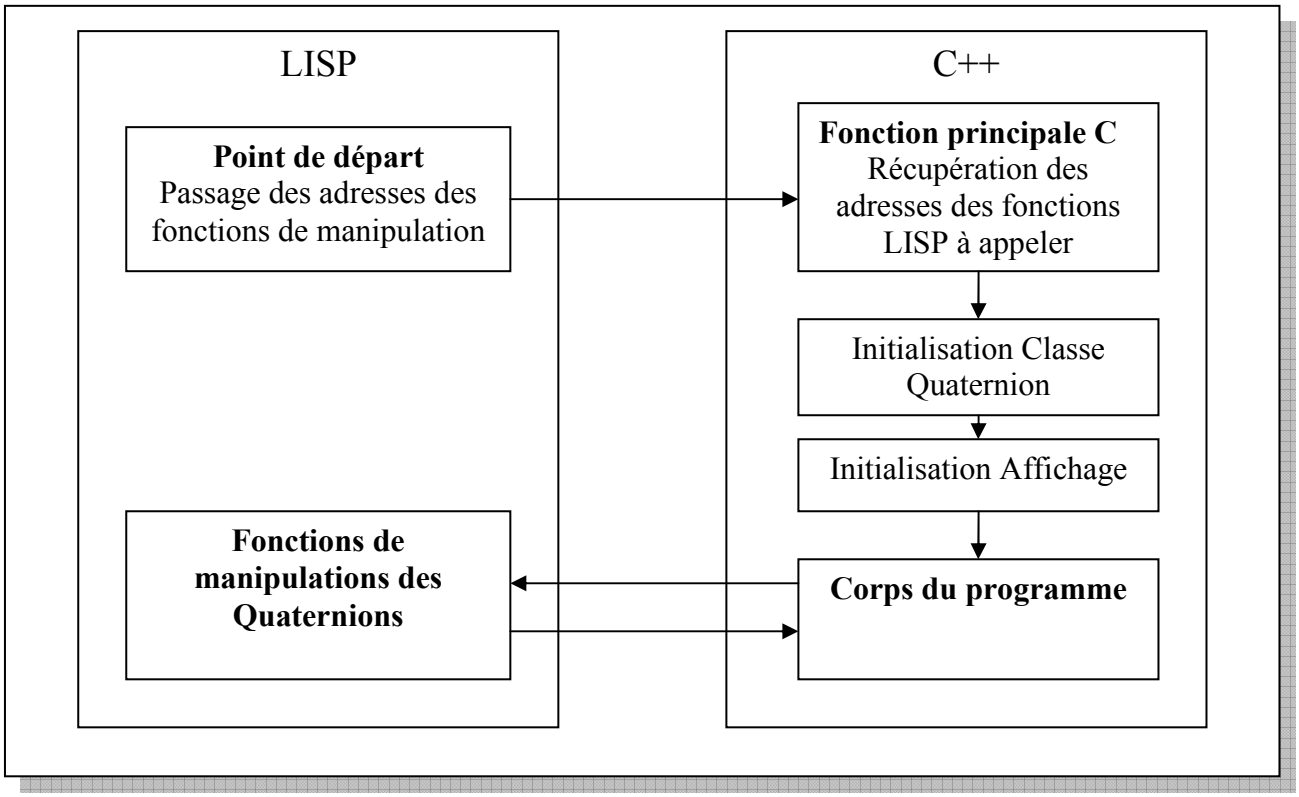
- **Mouvement Souris + Bouton gauche** : Rotation de la scène autour de X et de Z
- **Touches *4* et *6* du pavé numérique** : Rotation de la scène autour de Y
- **Touches *q*, *d*, *z* et *s*** : Déplacement de la sphère le long du bâton - **Touche *w*** : Quitter
- **Touche *m*** : Mode Matrice - **Touche *e*** : Mode angles d'Euler/Quaternions pour rotation scène
- **Touche *r*** : Activation/Désactivation rotation de la sphère - **Touche *a*** : Affichage des axes du repère
- **Touches + et -** : Modification de la vitesse de rotation de la sphère

5.5 Principe général

Pour l'appel de code C, les FFI imposent que le point de départ de l'application soit du code LISP pour initialiser l'interpréteur et l'environnement d'exécution LISP. Les FFI proposent différentes méthodes pour appeler du code C, il est possible de pré-compiler le code LISP et de le linker statiquement avec les objets C mais la méthode est relativement compliquée à mettre en œuvre. L'autre méthode que nous avons adoptée est plus simple à implémenter et consiste simplement à compiler le code C en tant que librairie dynamique et il suffit ensuite de préciser le nom de la librairie aux FFI pour que le mécanisme soit capable d'appeler les fonctions qu'elle exporte.

Le principe général de notre application est donc le suivant : L'application LISP une fois lancée appelle, via FFI, la fonction principale en C (en fait en C++ mais sous convention de nomage C, les FFI n'étant capable de manipuler uniquement des fonctions C) en lui passant en paramètre les adresses de toutes les fonctions LISP nécessaires à la manipulation des *Quaternions*. La fonction C les récupères pour initialiser la classe *Quaternion* puis démarre la démo. La démo utilise ainsi des Objets *Quaternions* pour effectuer ses rotations dans l'espace qui eux-mêmes font appel aux fonctions LISP pour leurs calculs.

Le schéma de principe général de notre application est le suivant :



5.6 Initialisation

Quelques explications sur l'utilisation des FFI de CLISP avec une librairie partagée C/C++. Pour appeler une fonction C, il faut commencer par en déclarer l'existence à CLISP avec la fonction « ffi:def-call-out » dont voici la syntaxe :

```
(ffi:def-call-out NOM_LISP
  (:name "NOM_C")
  (:library "NOM_LIBRARY")
  (:arguments
   (NOM_ARGUMENT
    (TYPE_ARGUMENT)
   )
  (:return-type TYPE_RETOUR)
  (:language LANGUAGE_UTILISE)
)
```

Par exemple la déclaration de la fonction principale C dans notre programme :

```
(ffi:def-call-out cmain      ;Fonction principale C
  (:name "quatMain")         ;Nomee quatMain dans la library
  (:library "quatdll.dll")   ;Nom de la library
  (:arguments                ;Liste d'arguments
   (conj             ;Le premier argument passé, adresse de la fct conjugé
    (ffi:c-function  ;Converti en pointeur sur fonction pour C
     (:arguments      ;de profile
                      ;Un argument de type tableau de 4 floats
      (quaternion (ffi:c-ptr (ffi:c-array single-float 4)) :in-out) )
                     ;retourne un tableau de 4 floats
     (:return-type (ffi:c-ptr (ffi:c-array single-float 4)) :malloc-free)
                     ;Language C
     (:language :stdc)
     )
    )

   ; ...
   ; Autres functions LISP de manipulation des quaternions
  )
  (:return-type ffi:int)   ;La fonction renvoie un int en cas d'erreur
  (:language :stdc)        ;Fonction écrite en C
)
```

Puis son appel avec passage de l'adresse de chaque fonction LISP :

```
(cmain #'c_conj #'c_norme #'c_add #'c_sub #'c_mult #'c_multScal #'c_inv
#'c_div #'c_divScal #'c_normalise #'c_quat2mat #'c_angle2quat #'c_euler2quat)
```

Et la fonction correspondante en C++ (*QuatMain.cpp*) :

```
//Quaternion.h
//Types pour le passage des pointeurs sur fonction
typedef float* (*Quat_Quat)(float[4]);
typedef float (*Scal_Quat)(float[4]);
typedef float* (*Quat_QuatQuat)(float[4], float[4]);
typedef float* (*Quat_QuatScal)(float[4], float);
typedef float* (*Quat_QuatScal)(float[4], float);
typedef float* (*Quat_Scal4)(float, float, float, float);
typedef float* (*Quat_Scal3)(float, float, float);

//QuatMain.cpp
extern "C"        //Force le compilateur a utiliser la convention C pour CLISP
#ifdef WIN32      //Pour exporter une fct d'une dll sous Visual C++
__declspec(dllexport)
#endif
int quatMain(     //Fonction quatMain, recupere les pointeurs sur fcts LISP
            Quat_Quat conj, Scal_Quat norme,
            Quat_QuatQuat add, Quat_QuatQuat sub,
            Quat_QuatQuat mult, Quat_QuatScal multScal,
            Quat_Quat inv,
            Quat_QuatQuat div, Quat_QuatScal divScal,
            Quat_Quat normalise,
            Quat_Quat quat2mat, Quat_Scal4 angle2quat,
            Quat_Scal3 euler2quat){

      ...
      //Corps de fonction, initialisation de la classe Quaternion avec les
pointeurs sur fonctions LISP
```

5.7 Conversion des types de donnés

Le fichier *c_quat.lisp* contient des fonctions LISP appelées par le C à la place des fonctions de manipulation elles mêmes pour convertir le types données de leurs arguments et valeur renvoyée. Ces fonctions sont dues au fait que les FFI ne renvoient pas des listes standard à la réception de tableaux C mais des types propres à *Common Lisp* appelés *simple-vector* (sous type de *Array*) qu'il faut convertir en liste. Il faut ensuite refaire la conversion dans le sens inverse pour renvoyer les listes résultat à C.

Pour effectuer la conversion, on utilise la fonction *coerce* de *Common Lisp*. Voici un exemple de fonction d'interfaçage pour la fonction *conj* (conjugué d'un quaternion) :

```
(defun c_conj (q)
  (coerce (conj (coerce q 'list)) 'simple-vector)
)
```

5.8 La classe Quaternion

Cette classe est définie dans le fichier *Quaternion.h* et ses méthodes sont implémentées dans le fichier *Quaternion.cpp*. Son rôle est de fournir un type de donnée *Quaternion* facilement manipulable et qui utilise les fonctions LISP pour ses calculs.

5.9 La fonction de calcul et d'affichage

La fonction de rafraîchissement de la démo s'appelle *RenderScene* (*QuatAppli.cpp*) et est chargée de calculer la position des objets et de les afficher. Cette fonction est appelée a intervalle régulier par la bibliothèque de fenêtrage (GLUT) pour redessiner la fenêtre d'affichage *OpenGL*. Cette fonction est assez longue et mélange *OpenGL* avec calculs sur les *Quaternion* mais nous avons essayé de la commenter au maximum pour qu'elle soit compréhensible, nous conseillons donc d'aller la consulter pour en voir le fonctionnement. Les parties spécifiquement liées à l'affichage sont isolées par des commentaires du type : //// <OpenGL> //// …code… //// </OpenGL> ////.

Les calculs sur les *Quaternions* sont également séparés en deux parties :

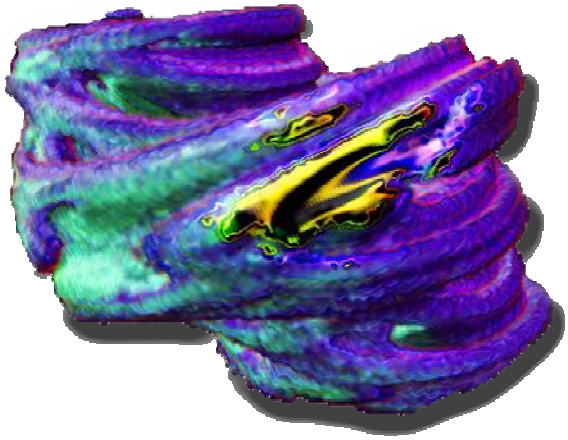
La première partie concerne la rotation de la scène contrôlée par la souris et les touches 4 et 6 du pavé numérique. Cette rotation est calculée, en mode *Quaternion*, par combinaison de la rotation ajoutée au moment *t* avec la rotation totale de la scène en multipliant les deux *Quaternions*. Les angles de rotation sont récupères sous formes d'angles d'Euler à partir de la souris et du clavier et sont convertis en Quaternion via la méthode *EulerToQuat*. Une fois les calculs terminés, le quaternion de rotation est converti, selon le mode (matrice scène, touche m), en couple axe+angle ou en matrice 4x4. Selon les cas on fait appel soit à la fonction *glRotate* soit *glMultMatrix* d'OpenGL pour roter la scène.

La seconde partie concerne la rotation de la sphère autour du bâton. Cette fois, le calcul de la position n'est plus laissé à OpenGL mais effectué par calcul sur les quaternions avec la formule *Quat_pos_finale=Quat_Rotation * Quat_pos_initiale * Quat_Rotation^-1* . *Quat_Rotation^-1* pouvant être remplace par le conjugué de *Quat_Rotation* les *Quaternions* de rotation étant normalisés. On positionne ensuite simplement la sphère par la fonction *glTranslate* d'OpenGL.

5.10 Autres démos

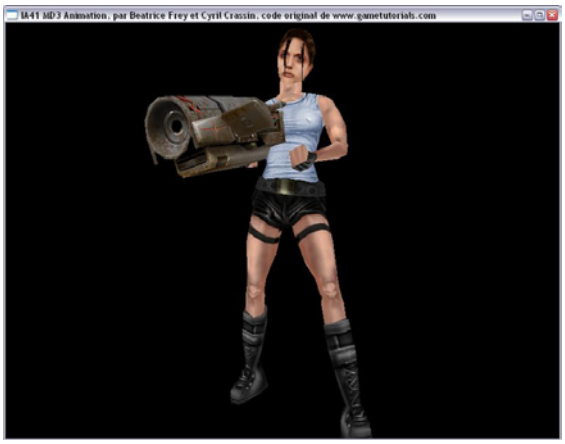
En plus d'avoir crée notre programme de démonstration, nous avons utilisé notre système LISP/Classe C++ dans deux autres applications qui montrent d'autres utilisations des fractales en infographie.

La première est un programme de création et de visualisation de fractales 3D que nous avons écris entièrement en utilisant notre système ainsi qu'un moteur de *Volume Rendering* développé il y a quelques mois par l'un d'entre nous pour la visualisation directe en 3D de données volumiques. Le système de visualisation a la particularité de ne pas recourir à une détermination de surface pour le rendu de données d'un champ de scalaires tridimensionnel mais d'utiliser les capacités graphiques des nouveaux matériels 3D (*shaders*) pour en faire l'affichage direct ainsi que l'éclairage.



Exemple de rendu d'une fractale de Julia obtenu avec notre application

La seconde application montre l'utilisation des fractales pour l'animation hiérarchique de personnages à l'aide d'interpolations de rotation SLERP. Ce programme est issu d'un exemple récupéré sur Internet (www.gametutorials.com) que nous avons adapté à notre système. Seules les primitives de bases sont calculées en LISP et les interpolations SLERP sont pour l'instant laissées au C.



Conclusion

Ce projet nous a permis de découvrir les intérêts majeurs des Quaternions et d'en implémenter les applications principales en infographie. Notre démarche a été tout au long de ce projet de tenter de comprendre les avantages réels de cette notion mathématique qui date tout de même de plus d'un siècle mais qui à trouvé ces dernières années un nouveau souffle en animation 3D par informatique. Il s'agit en effet d'un domaine en forte expansion dans lequel cette notion est de plus en plus utilisée de par sa grande souplesse et facilité de manipulation. On peut noter l'existence d'un autre ensemble de complexes formés de huit composantes : les octonions ou octaves de Cayley, qui lui par contre n'est pas associatif.