

# Chapitre 3 : Repères et positionnement 3D

Modélisation 3D et Synthèse

Fabrice Aubert  
fabrice.aubert@univ-lille1.fr

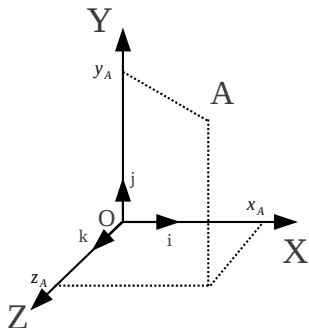


Master Informatique

2015-2016

# 1 Repères d'une scène 3D

# Repère 3D



► Repère noté  $(O, i, j, k)$  (origine et vecteurs de base).

► Un point :  $A = \overrightarrow{OA} = \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix}$  ou

$$A = (A_x, A_y, A_z).$$

►  $A = O + xi + yj + zk.$

► Les repères considérés seront généralement directs :

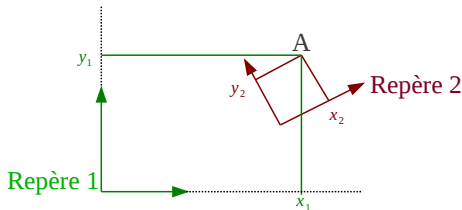
- Règle de la main **droite** : (Pouce, Index, Majeur) =  $(X, Y, Z)$

► Un repère est dit orthonormé si :

- Vecteurs  $(i, j, k)$  deux à deux orthogonaux.
- $(i, j, k)$  de même normes 1.

# Plusieurs repères

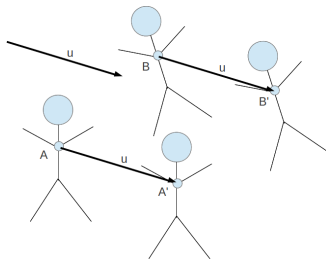
- ▶ On travaillera avec plusieurs repères : **toujours** préciser le repère en indice.
- ▶ Sur l'exemple, un même point  $A$  a les coordonnées  $(x_1, y_1, z_1)$  dans le repère 1 et  $(x_2, y_2, z_2)$  dans le repère 2.



⇒ Noter  $A_1$  ou  $A_2$  selon le repère considéré (même point mais coordonnées différentes)

# Points et directions

- Distinguer **les positions** (i.e. les points) et **les directions** (i.e. les vecteurs).
- Exemple :
  - Le point  $A$  est déplacé au point  $A'$  par le vecteur  $u$ . Le point  $B$  est déplacé au point  $B'$  par le même vecteur  $u$ .
  - Par le calcul :  $A' = A + u$  et  $B' = B + u$ .
  - Remarque sur les notations : pas de "flèche" sur  $u$ . Par contre  $u = \overrightarrow{AA'} = \overrightarrow{BB'}$ . Pour les calculs, on utilisera souvent la relation :  $u = A' - A = B' - B$ .



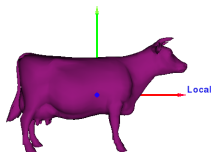
- **Attention** : dans le code on représente généralement les positions et les directions par une même classe (`Vector3` pour les tps).

# Scène 3D et repères

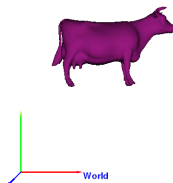
- ▶ Pour la conception d'une scène 3D et de ses objets 3D, nous considérons les repères suivants :
  - Le **repère de l'observateur** (ou **repère de la caméra** ; noté `Eye` dans la suite) : c'est dans ce repère qu'on définit le volume de visualisation (i.e. placement de l'écran et définition des paramètres de la projection).
  - Le **repère local** (ou **repère objet** ; noté `Local` dans la suite) : on conçoit un objet indépendamment du reste de la scène. On prendra le repère le plus naturel pour définir les points de l'objet.
  - Le **repère du monde** (ou **repère de scène** ; noté `World` dans la suite) : il s'agit du repère de référence de tout positionnement (les repères locaux et la caméra seront placés directement ou indirectement par rapport à `World`).
- ▶ Nous placerons les repères les uns par rapport aux autres avec des changements de repères (en général par translations, rotations, changements d'échelle). Le repère "initial" étant le repère de scène `World`.
- ▶ Les positions  $(x, y, z)$  des sommets des objets 3D sont données dans le repère `Local` de l'objet qu'ils définissent, et l'objet est placé dans la scène en déplaçant son repère local (et non directement/explicitement ses points)..

# Exemple

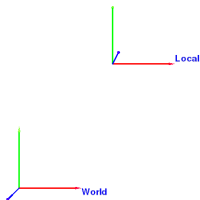
1) on dispose d'un objet défini dans son repère local



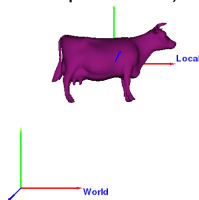
2) on souhaite



3) on place `Local` par rapport à `World`



4) on obtient (points toujours exprimés dans le repère local)



- ▶ **Chapitre précédent** : donner au vertex shader les positions  $P_{Eye} = (x, y, z)$  (i.e. définies dans le **repère de l'observateur**).
  - Obtenir  $P_{ClipCoordinates}$  à partir de  $P_{Eye}$  avec la matrice de projection (cf chapitre précédent sur le volume de visualisation).
- ▶ **Dans la suite** : donner au vertex shader des positions  $P_{Local} = (x, y, z)$  (i.e. définies dans le **repère de l'objet** à tracer).
  - 1 Obtenir  $P_{Eye}$  à partir de  $P_{Local}$  : pour cela on indiquera comment est placé le repère  $Local$  par rapport au repère  $Eye$  **avec une matrice de changement de repères** (cf la suite du cours).
  - 2 Puis obtenir  $P_{ClipCoordinates}$  à partir de  $P_{Eye}$  avec la matrice de projection (comme précédemment).



## 2 Transformations et changement de repères

# Coordonnées homogènes (point)

- **Un point** (une position) 3D en coordonnées homogènes :

$$P_{3D} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \iff P_H = \begin{pmatrix} xw \\ yw \\ zw \\ w \end{pmatrix} \text{ avec } w \in \mathbb{R}^* (w \neq 0)$$

- Passer des coordonnées homogènes en coordonnées 3D :

$$P_H = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \iff P_{3D} = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

# Coordonnées homogènes (vecteur)

- **Un vecteur** (une direction) 3D en coordonnées homogènes :

$$u_{3D} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \iff u_H = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \text{ noter le 0}$$

- Passer des coordonnées homogènes en coordonnées 3D :

$$u_H = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \iff u_{3D} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

# Attention aux calculs !

- ▶  $A_H + B_H$  (points en coordonnées homogènes) n'a pas de sens et ne correspond pas à  $A_{3D} + B_{3D}$  (calcul en 3D).
- ▶  $u_H + v_H$  (vecteurs en coordonnées homogènes) correspond bien par contre à  $u_{3D} + v_{3D}$
- ▶  $A_H + u_H$  (1 point + 1 vecteur) ne correspond pas à  $A_{3D} + u_{3D}$  sauf si  $A_H$  est normalisé (i.e.  $w = 1$ ).

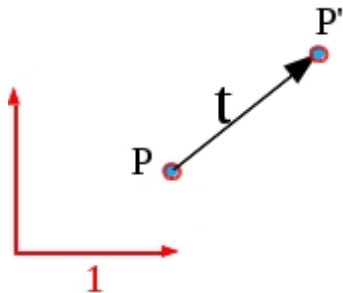
# Exemple de la translation

- Rappel (cf introduction des coordonnées homogènes dans le chapitre précédent) : la translation de vecteur  $t = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$  peut se traduire par une matrice en coordonnées homogènes :

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- On donne, dans les transparents suivants, 3 interprétations de cette translation (avec les mêmes coordonnées  $P(x, y, z)$  et la même translation  $t$  dans les 3 cas).

# Interprétation par transformation



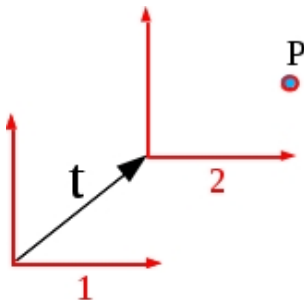
- On connaît le point  $P(x, y, z)$  donné dans un repère 1 et on déplace le point  $P$  en  $P'$  par la translation  $t$ .

$$P' = TP$$

T appelée matrice de transformation.

# Interprétation par changement de repère

- ▶ On connaît le point  $P(x, y, z)$  exprimé dans un repère 2 et on déplace le repère 2 en partant du repère 1 par la translation  $t$ .

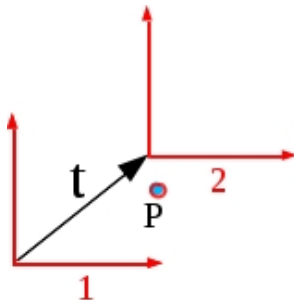


- ▶ Dans cette interprétation,  $T$  est appelée **la matrice de passage du repère 1 au repère 2**, et on la note  $M_{1 \rightarrow 2}$ .
- ▶ On connaît  $P_2 = P(x, y, z)$ . On peut connaître  $P_1$  par :

$$P_1 = M_{1 \rightarrow 2} P_2 \text{ avec } P_2 = P \text{ et } M_{1 \rightarrow 2} = T$$

- ▶ Remarque :  $P_1$  correspond à  $P'$  du transparent précédent (c'est le même calcul).

# Interprétation inverse



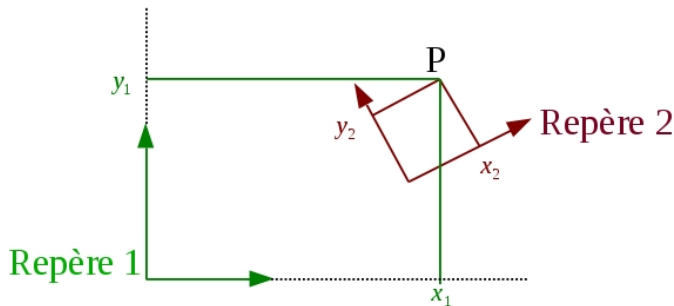
- ▶ On connaît le point  $P(x, y, z)$  donné dans le repère 1 et on se donne un repère 2 par rapport au repère 1 par la translation  $t$ .
- ▶ On a toujours la relation :

$$P_1 = M_{1 \rightarrow 2} P_2 \text{ mais, cette fois, c'est } P_1 \text{ qu'on connaît : } P_1 = P$$

- ▶ Si on souhaite  $P_2$ , on peut l'obtenir par  $P_2 = M_{2 \rightarrow 1} P_1$ .
- ▶  $M_{2 \rightarrow 1}$  ? obtenue par **la translation opposée**  $-t$ .



# Changement de repère (généralisation)



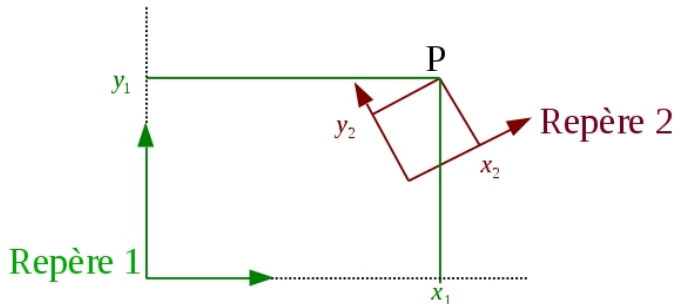
$$P_1 = M_{1 \rightarrow 2} P_2$$

(bien noter la position et l'ordre des indices dans la relation).

►  $M_{1 \rightarrow 2}$  dans la relation indique :

- comment le repère 2 est obtenu à partir du repère 1
- permet d'exprimer les coordonnées du point  $P$  dans le repère 1 à partir des coordonnées dans le repère 2.

# Attention à la confusion !!!

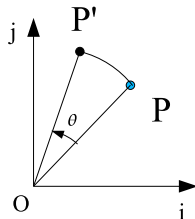


$$P_1 = M_{1 \rightarrow 2} P_2$$

- ▶ "le passage du repère 1 au repère 2" signifie (i.e. noté  $M_{1 \rightarrow 2}$ ) :
  - le "déplacement" du repère 1 vers le repère 2 .
  - **Et non pas** "passage" des coordonnées d'un point  $P$  connues dans le repère 1 vers les coordonnées de  $P$  dans le repère 2 (c'est le contraire : notez bien la position des indices).

# Rotation en 2D

Rotation d'angle  $\theta$  autour de l'origine.



$$\begin{cases} x' &= x \cos(\theta) - y \sin(\theta) \\ y' &= x \sin(\theta) + y \cos(\theta) \end{cases}$$

Comment retrouver ? Soit  $\alpha$  l'angle  $(i, OP)$  et  $r = \|OP'\| = \|OP\|$ . Alors

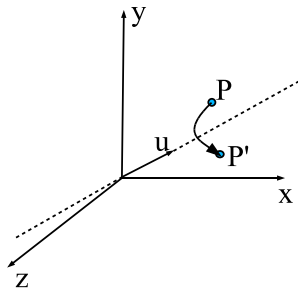
$$\begin{cases} x' &= r \cos(\alpha + \theta) \\ y' &= r \sin(\alpha + \theta) \end{cases}$$

$$\begin{cases} x' &= r \cos(\alpha) \cos(\theta) - r \sin(\alpha) \sin(\theta) \\ y' &= r \cos(\alpha) \sin(\theta) + r \sin(\alpha) \cos(\theta) \end{cases}$$

or  $x = r \cos(\alpha)$  et  $y = r \sin(\alpha)$

$$P' = RP \text{ avec } R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

# Rotation en 3D



- ▶ La rotation en 3D s'effectue autour **d'un axe** dont on donne un vecteur directeur  $u$  (rotations considérées : axe passant par l'origine).
- ▶ Le sens de rotation est le sens trigonométrique par rapport à l'axe (« tourne » dans le sens direct quand le vecteur de l'axe pointe vers vous).
- ▶ Exemple : autour de l'axe  $z$  (droite de vecteur directeur  $(0, 0, 1)$ ).

$$R_{OZ} = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

# Rotation 3D : autres axes

$$R_{OX} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix}$$

Substituer  $Y$  à  $X$  et  $Z$  à  $Y$

$$R_{OY} = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}$$

Substituer  $Z$  à  $X$  et  $X$  à  $Y$  (attention aux signes)

- La rotation autour d'un axe de vecteur  $u$  quelconque est un peu plus laborieuse à exprimer.

Avec  $c = \cos(\theta)$ ,  $s = \sin(\theta)$  et  $u = (u_x, u_y, u_z)$  **normé**.

$$\begin{pmatrix} u_x^2 + (1 - u_x^2)c & u_x u_y (1 - c) - u_z s & u_x u_z (1 - c) + u_y s \\ u_x u_y (1 - c) + u_z s & u_y^2 + (1 - u_y^2)c & u_y u_z (1 - c) - u_x s \\ u_x u_z (1 - c) - u_y s & u_y u_z (1 - c) + u_x s & u_z^2 + (1 - u_z^2)c \end{pmatrix}$$

# Rotation en coordonnées homogènes

- ▶ On ajoute une ligne et une colonne par rapport à la matrice  $3D$  :

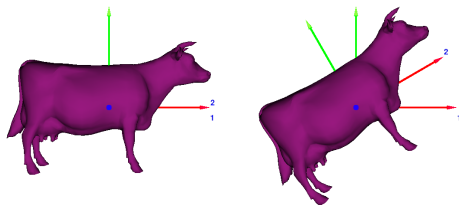
$$R_H = \left( \begin{array}{c|c} R_{3D} & \begin{matrix} 0 \\ 0 \end{matrix} \\ \hline \begin{matrix} 0 & 1 \end{matrix} \end{array} \right)$$

- ▶  $P' = RP$

# Rotation 2D (changement de repères)

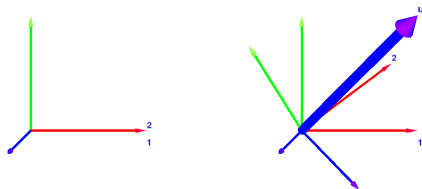


- ▶ Avec un objet défini localement dans le repère 2 :

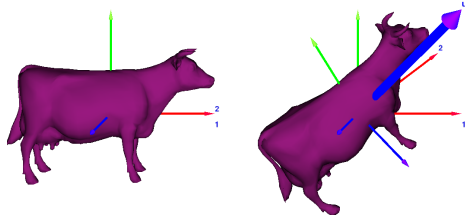


$P_1 = M_{1 \rightarrow 2} P_2$  avec  $M_{1 \rightarrow 2} = R$  (matrice de rotation homogène du transparent précédent)

# Rotation 3D (changement de repères)



► Avec un objet défini localement dans le repère 2 :

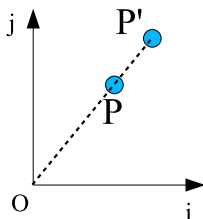


$P_1 = M_{1 \rightarrow 2} P_2$  avec  $M_{1 \rightarrow 2} = R$  (matrice de rotation homogène du transparent précédent)



# Scale (transformation)

Changement d'échelle de rapport  $k$ .



$$\begin{cases} x' = kx \\ y' = ky \\ z' = kz \end{cases} \text{ peut être défini pour chaque coordonnée : } \begin{cases} x' = k_x x \\ y' = k_y y \\ z' = k_z z \end{cases}$$

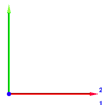
$$P' = SP \text{ avec } S = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{pmatrix}$$

# Scale en coordonnées homogènes

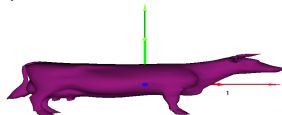
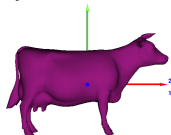
$$S = \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P' = SP$$

# Scale (changement de repère)



- Avec un objet défini localement dans le repère 2 :



$$P_1 = M_{1 \rightarrow 2} P_2 \text{ avec } M_{1 \rightarrow 2} = S \text{ (matrice de scale homogène du transparent précédent)}$$

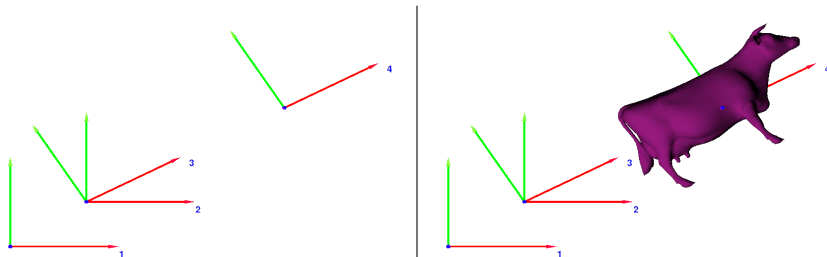
- La composition de changements de repères successifs se traduit par une unique matrice obtenue par le produit des matrices de chaque changement de repère.

$$M_{i \rightarrow j} = M_{i \rightarrow k} M_{k \rightarrow j}$$

# Composition (exemple)

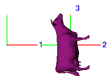
- ▶  $M_{1 \rightarrow 2}$  = translation en x et y (2 placé par rapport à 1)
- ▶  $M_{2 \rightarrow 3}$  = rotation (bien noter : 3 placé par rapport à 2 ; donc de centre l'origine de 2).
- ▶  $M_{3 \rightarrow 4}$  = translation de vecteur (2,0,0) (4 placé par rapport à 3 ; donc déplacement sur l'axe x de 3).

$$M_{1 \rightarrow 4} = M_{1 \rightarrow 2} M_{2 \rightarrow 3} M_{3 \rightarrow 4}$$



# Commutation

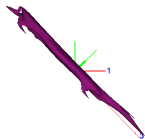
$$M_{1 \rightarrow 3} = TR$$



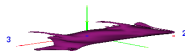
$$M_{1 \rightarrow 3} = RT$$



$$M_{1 \rightarrow 3} = RS$$



$$M_{1 \rightarrow 3} = SR$$



(Attention au scale !)

- ▶ Un objet 3D (sans changement d'échelle) peut être placé par une composition quelconque de translations et de rotations (n'importe quel ordre).
- ▶ Cette composition est équivalent à une simple composition  $TR$  ( $T$  = une translation,  $R$  = une rotation)
- ▶ La matrice d'une telle composition sera alors sous la forme :

$$TR = \left( \begin{array}{c|c} R_{3D} & T_{3D} \\ \hline 0 & 1 \end{array} \right)$$

(i.e. la rotation sur la partie 3x3 supérieure gauche et la translation en dernière colonne).

- ▶ un placement  $TR$  est appelé **transformation rigide**.

- ▶ **Attention !!!** une direction ne subit pas **les translations**.
- ▶ En coordonnées homogènes : soit  $u = \overrightarrow{AB}$  (coordonnée  $w = 0$ ) et  $M_{1 \rightarrow 2}$  une translation alors :

$$u_1 = M_{1 \rightarrow 2} u_2$$

et on obtient bien  $u_2 = u_1$

- ▶ **mais** attention au code qui manipule les données 3D (non homogènes). Exemple :

```
Vector3 a,b;  
Transform m; // peu importe le code : matrice homogène ou autre  
b=m.transform(a) // VECTEUR ou POINT ????? (résultat différent).  
// la librairie 3D devrait différencier, par exemple : m.transformPosition(a) et m.transformDirection(a)  
// mais n'offre souvent que m.transform(a) pour les POINTS 3D (comment alors obtenir une transformation de vecteur 3D ?)
```



# Changements inverses

- ▶ L'inverse de  $M_{1 \rightarrow 2}$  (notée  $M_{1 \rightarrow 2}^{-1}$ ) permet d'obtenir  $M_{2 \rightarrow 1}$
- ▶ Inverse d'un produit de matrices :  $(M_1 M_2 M_3)^{-1} = M_3^{-1} M_2^{-1} M_1^{-1}$  (attention à l'ordre !)
- ▶ Inverse d'une composition de changements de repères :  
 $(M_{1 \rightarrow 2} M_{2 \rightarrow 3} M_{3 \rightarrow 4})^{-1} = M_{4 \rightarrow 3} M_{3 \rightarrow 2} M_{2 \rightarrow 1}$
- ▶ Dans le cas général l'inversion consiste à résoudre  $MM^{-1} = I$  (pivot de gauss, par calcul de déterminant, ...),
- ▶ Mais l'inverse s'obtient aisément pour les transformations usuelles :
  - La translation inverse de  $T$  est la translation de vecteur opposé  $-T$ .
  - La rotation inverse de  $R_\theta$  est la rotation d'angle opposé  $R_{-\theta}$
  - Le changement d'échelle inverse de  $S(k_x, k_y, k_z)$  est  $S(\frac{1}{k_x}, \frac{1}{k_y}, \frac{1}{k_z})$ .
  - L'inverse de toute matrice orthonormale (matrice  $3 \times 3$  de rotation par exemple) s'obtient en la transposant :

$$\begin{pmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{pmatrix}^{-1} = \begin{pmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{pmatrix} \text{ si orthonormal}$$

# Changements de repère et application 3D

- ▶ Dans l'application on gère :
  - Le placement des objets dans la scène se traduit par  $M_{World \rightarrow Local}$  (on place `Local` par rapport à `World`).
  - Le placement de la caméra dans la scène se traduit par  $M_{World \rightarrow Eye}$  (on place `Eye` par rapport à `World`).
  - Les objets pour lesquels les coordonnées des points  $P_{Local}$  sont données dans `Local`.
- ▶ Pour la visualisation (OpenGL), il faut calculer  $P_{Eye}$  (pour ensuite calculer  $P_{ClipCoordinates}$ ) :
  - Il suffit d'appliquer  $P_{Eye} = M_{Eye \rightarrow Local} P_{Local}$
  - En décomposant :  $M_{Eye \rightarrow Local} = M_{Eye \rightarrow World} M_{World \rightarrow Local}$
  - On dispose déjà de  $M_{World \rightarrow Local}$ , mais il faut **inverser**  $M_{World \rightarrow Eye}$  pour avoir  $M_{Eye \rightarrow World}$  : il reste alors à faire le produit.
- ▶ Ce calcul de  $M_{Eye \rightarrow Local}$  est généralement assurée par l'application (et non par le vertex shader), et on donne à OpenGL (i.e. au vertex shader) cette matrice  $M_{Eye \rightarrow Local}$ .
- ▶ La matrice  $M_{Eye \rightarrow Local}$  est appelée `MODELVIEW`. (traditionnellement).

► On fournira au vertex shader :

- l'attribut position exprimé dans le repère local (i.e.  $P_{Local}$ ).
- la `modelview` (i.e.  $M_{Eye \rightarrow Local}$ ).
- la projection (i.e.  $M_{ClipCoordinate \rightarrow Eye}$ ).

► Vertex Shader :

```
in vec3 position;  
...  
uniform mat4 projection;  
uniform mat4 modelview;  
...  
void main() {  
...  
    gl_Position = projection*modelview*vec4(position,1.0);  
}
```

► Application :

```
_shader.uniform("modelview",p3d::modelview); // p3d::modelview gérée par l'application (variable "globale")  
_shader.uniform("projection",p3d::projection); // p3d::projection gérée par l'application ("globale")  
glDraw...
```

- Souvent le produit `projection*modelview` est directement calculé par l'application et donné au vertex shader ("traditionnellement" l'uniform correspondant est noté `mvp`)

# Classe pour les matrices

- Nécessité d'une classe pour gérer les matrices homogènes dans l'application (dans les tps : `Matrix4`).

```
Matrix4 m1,m2,m3;  
Vector3 p1,p2,u1,u2;  
...  
m1.setIdentity(); // initialisation  
m1.mul(m2); // Attention ! multiplication à droite ! (pour traduire l'approche par changement de repère).  
m1.translate(x,y,z); // compose avec la matrice de translation (à droite...)  
m1.rotate(angle, axisX, axisY, axisZ); // compose avec la matrice de rotation (angle en degré).  
m1.invert(); // inverse la matrice  
m1=m2*m3; // produit  
m1*=m2; // produit (i.e. m1=m1*m2)  
p1 = m1.transformPoint(p2); // transformation d'un point  
u1 = m1.transformDirection(u2); // transformation d'une direction (i.e. vecteur).
```

# Attention aux notations !

- ▶ Dans le code, la matrice de passage  $M_{Rep1 \rightarrow Rep2}$  sera (généralement) représentée par la variable `rep1Rep2` (par exemple : `worldLocal`, `eyeWorld`, etc).
- ▶ Le nom de la variable indique bien le changement de repère **et non** la transformation.
- ▶ Par exemple, si vous connaissez un point `P_World` (en 3D) et que vous souhaitez le connaître dans le repère `Eye` (3D), il faut appliquer  
`P_Eye=eyeWorld.transformPoint(P_World)`
- ▶ **Attention** donc à la confusion (changements de repères).
- ▶ Notez le `transformPoint` qui distingue points 3D et directions 3D (il doit y avoir aussi une méthode `transformDirection`)

# Exemple de code avec la modelview

- ▶ En décomposant avec world, eye et local :

```
worldLocal.setIdentity();
worldEye.setIdentity();

// placement de l'objet par rapport à world : world->local
worldLocal.rotate(angle,1,0,0);

// placement de la caméra par rapport à world : world->eye
worldEye.translate(0,0,10);
worldEye.rotate(5,0,1,0);

p3d::modelview = worldEye.inverse()*worldLocal; // noter l'inversion
drawObject(); // p3d::modelview est passée au shader
```

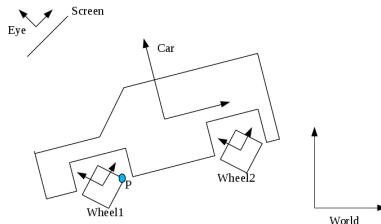
- ▶ Avec uniquement la modelview (attention aux interprétations pour la caméra ! l'objet est placé directement par rapport à la caméra ; à proscrire pour des positionnements plus complexes) :

```
p3d::modelview.setIdentity();
// placement de la caméra : eye->world
p3d::modelview.rotate(-5,0,1,0);
p3d::modelview.translate(0,0,-10);
// traduction du placement de l'objet : world->local
p3d::modelview.rotate(angle,0,1,0);
drawObject(); // p3d::modelview est passée au shader
```

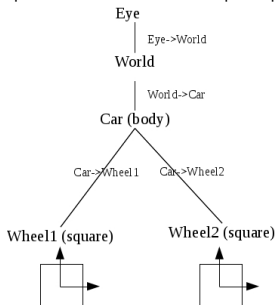
# 3 Conception hiérarchique

# Conception hiérarchique

- On peut généraliser l'approche de placer les objets par rapport au monde, en les plaçant les uns par rapport aux autres :



- Le positionnement relatif des repères peut se représenter sous forme d'arbre :



- Chaque branche se traduit par un changement de repère  $M_{parentNode \rightarrow childNode}$
- Chaque sous-arbre est conçu indépendamment de son parent (i.e. on associe à chaque nœud un `drawNode` sans se préoccuper du nœud parent).
- On remarque que les deux roues sont identiques dans leurs repères locaux (ce sera donc la même procédure qui tracera les deux roues).



- ▶ La matrice `p3d::modelview = M_{Eye \rightarrow Local}` indique dans quel repère `Local` on se trouve (i.e. le repère courant).
- ▶ Descendre dans l'arbre consiste à changer le repère courant (on applique le passage du noeud à son fils).
- ▶ Lorsqu'un noeud a plusieurs enfants (tous placés par rapport au parent) : on doit mémoriser le repère du parent.
- ▶ Eviter de faire des inversions  $\Rightarrow$  préférer mémoriser les repères.
- ▶  $\Rightarrow$  utilisation d'une pile pour mémoriser `p3d::modelview` (i.e. `p3d::modelview.push()` ; `p3d::modelview.pop()`).

# L'exemple

`p3d::modelview.apply(R1,R2)` est du pseudo code : correspond à une suite d'instructions (translate, rotate, etc) qui modifie la modelview pour traduire le changement de repère de R1 à R2 :

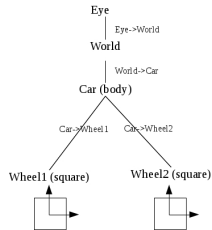
```
void drawWheel() { // repère local à une roue
    p3d::modelview.push(); // par précaution (inutile ici).
    drawSquare();
    p3d::modelview.pop();
}

void drawCar() { // repère local à une voiture
    p3d::modelview.push();

    // MODELVIEW = M_Eye_Car (Repere courant = Voiture)
    drawBody();

    p3d::modelview.push();
    p3d::modelview.apply(Car,Wheel1); // MODELVIEW *= M_Car_Wheel1
    // i.e. Repère courant = Wheel1
    drawWheel();
    p3d::modelview.pop(); // on revient à Car : MODELVIEW = M_Eye_Car

    p3d::modelview.apply(Car,Wheel2); // MODELVIEW *= M_Car_Wheel2;
    // i.e. Repère courant = Roue2
    drawWheel();
    p3d::modelview.pop();
}
```



```
void drawScene() {
    p3d::modelview.setIdentity();
    p3d::modelview.apply(Eye,World); // caméra
    p3d::modelview.apply(World,Car); // objet
    drawCar();
}
```

- ▶ Remarquez que c'est le noeud père (le `draw` appelant) qui place ses fils (les `draw` appelés). Par exemple : c'est la voiture qui place les roues (et non les roues qui se placent elle mêmes par rapport à la voiture) ; c'est le monde qui place la voiture ; etc.
- ▶ Remarquez les push/pop systématiques à l'entrée/sortie des `draw` : permet d'éviter tout effet de bord (aucun noeud fils doit modifier le repère courant d'un noeud après son appel).

# Attention au Scale !

- Comparez le résultat de :

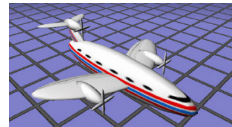
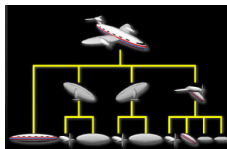
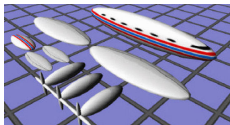
```
p3d::modelview.setIdentity();  
p3d::modelview.rotate(angle,0,0,1);  
p3d::modelview.scale(2,1,1);  
drawSquare();
```

- et

```
p3d::modelview.setIdentity();  
p3d::modelview.scale(2,1,1);  
p3d::modelview.rotate(angle,0,0,1);  
drawSquare();
```

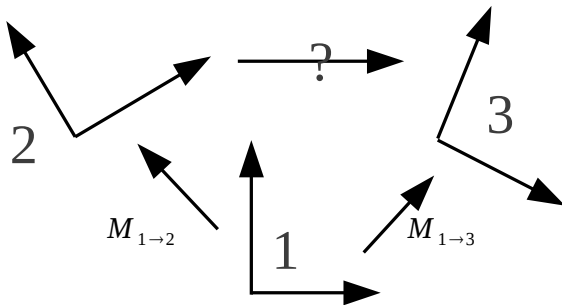
# Conception hiérarchique : résumé

- ▶ Concevoir les composants dans des repères locaux les plus simples (ou intuitifs) possibles.
- ▶ Assembler les composants hiérarchiquement (sous forme d'arbres), en les positionnant relativement les uns par rapport aux autres (on place les enfants par rapport au parent par un changement de repère  $M_{parentNode \rightarrow childNode}$ ).



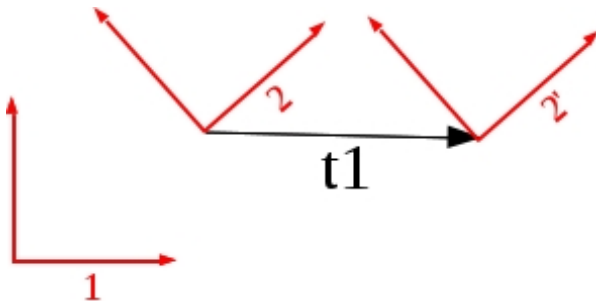
- ▶ Le principe de la conception hiérarchique est fondamentale et constitue la fondation de la majeure partie des bibliothèques 3D (graphes de scène).

## 4 Quelques changements de repères usuels



- ▶ On connaît  $M_{1 \rightarrow 2}$  et  $M_{1 \rightarrow 3}$ , quelle est la matrice  $M_{2 \rightarrow 3}$  ?
- ▶ Solution :  $M_{2 \rightarrow 3} = M_{2 \rightarrow 1} M_{1 \rightarrow 3}$  (il reste à inverser  $M_{1 \rightarrow 2}$  pour avoir  $M_{2 \rightarrow 1}$ ).

# Transformation par rapport au repère "précédent"

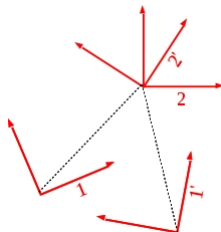


- ▶ On connaît  $M_{1 \rightarrow 2}$ , et on connaît la translation  $t$  exprimée **dans 1** (de matrice  $T_1$ ). On souhaite appliquer  $t_1$  sur 2.
- ▶ Expliquez pourquoi  $M_{1 \rightarrow 2'} = M_{1 \rightarrow 2} T_1$  ne donnera pas le bon résultat.
- ▶ Plusieurs approches :
  - Exprimez le **vecteur**  $t_1$  dans 2 :  $t_2 = M_{1 \rightarrow 2} t_1$  puis  $M_{1 \rightarrow 2'} = M_{1 \rightarrow 2} T_2$ .
  - On "attache" 2 à 1, puis on "bouge" 1 de  $t_1$  sur 1' :  $M_{2 \rightarrow 2'} = M_{2 \rightarrow 1} M_{1 \rightarrow 1'} M_{1' \rightarrow 2'}$  (même raisonnement que précédent).
  - $\Rightarrow M_{1 \rightarrow 2'} = t_1 M_{1 \rightarrow 2}$  (ce qui donne une autre interprétation : on applique la translation avant  $M_{1 \rightarrow 2}$ ).



# Transformation exprimée dans un autre repère

- ▶ Exemple : rotation  $R_2$  du repère 1 pour obtenir  $1'$  (rotation exprimée dans 2).
- ▶ Interprétation possible : “attacher” le repère 1 au repère 2.



- ▶  $\Rightarrow M_{1 \rightarrow 1'} = M_{1 \rightarrow 2} M_{2 \rightarrow 2'} M_{2' \rightarrow 1'}$
- ▶ avec  $M_{2 \rightarrow 2'} = R$  et  $M_{2' \rightarrow 1'} = M_{2 \rightarrow 1}$ .
- ▶  $\Rightarrow M_{1 \rightarrow 1'} = M_{1 \rightarrow 2} R M_{2 \rightarrow 1}$
- ▶ Rotation de 1 autour d'un point quelconque : prendre  $M_{1 \rightarrow 2}$  la plus simple possible (translation).

- Soient  $1 = (O^{(1)}, i^{(1)}, j^{(1)}, k^{(1)})$  et  $2 = (O^{(2)}, i^{(2)}, j^{(2)}, k^{(2)})$ , deux repères orthonormés. On connaît les expressions de l'origine et de la base de 2 dans le repère 1.

Alors :

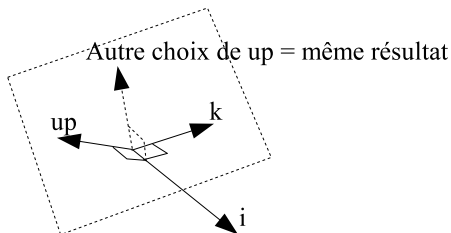
$$M_{1 \rightarrow 2} = \left( \begin{array}{ccc|c} i_1^{(2)} & j_1^{(2)} & k_1^{(2)} & O_1^{(2)} \\ \hline 0 & 0 & 0 & 1 \end{array} \right)$$

- L'inverse  $M_{2 \rightarrow 1}$  ? il suffit de faire comme pour  $(TR)^{-1}$  : transposer le bloc haut-gauche (c'est une rotation) et d'opposer le bloc haut-droit (c'est une translation) et faire le produit des inverses.

Placer la caméra par rapport à  $World$  en donnant sa position  $A_{World}$ , quel point elle regarde (point  $At_{World}$ ) et son roulis par un vecteur  $Up_{World}$ .

► Construire  $(i, j, k)$  :

- $k$  est donné par  $k = O - At$ , et on le normalise.
- $i$  est tel que  $i = up \times k$
- enfin  $j$  est calculé par  $j = k \times i$
- La matrice  $M_{World \rightarrow Eye}$  est donnée en mettant en colonne  $i, j, k$  et  $O$  (repère étant orthonormé).



# 5 Changement de repères et éclairage

# Calcul dans le repère Eye

- Pour un calcul cohérent, il faut que toutes les données nécessaires (N, position des sources, L, ...) soient exprimées dans le même repère.
- $\Rightarrow$  transformation des normales :

```
void main() {  
    vec4 positionEye=modelviewMatrix*vec4(position,1.0);  
    vec4 normalEye=modelviewMatrix*vec4(normal,0.0); // calcul FAUX en général  
  
    L=lightPosition-positionEye.xyz/positionEye.w;  
    N=normalEye.xyz;  
    ...  
  
    gl_Position = projection*positionEye;  
}
```

# Attention au Scale ! (encore)

- Transformation des normales incorrecte lors d'un changement d'échelle.



avant transformation

après transformation par `modelview`

- La matrice correcte pour transformer les normales est  $(M^{-1})^t$  où  $M$  est la sous-matrice 3x3 (3 premières lignes, 3 premières colonnes) de `modelview`.
- Il faut donc passer une matrice supplémentaire :

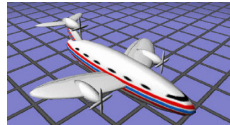
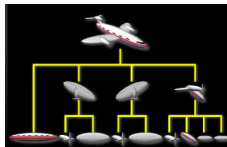
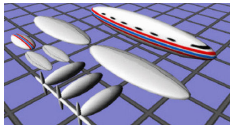
```
vec4 positionEye=modelviewMatrix*vec4(position,1.0);
vec3 normalEye=normalMatrix*normal;

L=lightPosition-positionEye.xyz/positionEye.w;
N=normalEye;
...

gl_Position = projection*positionEye;
```

## 6 Représentation des changements de repères par TRS - Quaternions

# Représenter la conception hiérarchique



- Représenter explicitement la hiérarchie par un arbre (graphe de scène).

```
class Object3D {  
    Matrix4 _parentChild; // changement de repère  
  
    Object3D *_parent;    // référence sur le noeud parent  
    std::vector<Object3D *> _child; // noeuds enfants  
public:  
    ...  
    virtual void draw()=0; // interface : trace dans le repère local  
    ...  
};
```

- Le visualiseur ("render") se charge de parcourir tout l'arbre en composant la modelview et en traçant les géométries à chaque noeud (+ gestion des shaders/textures/etc).
- Offrir toutes les méthodes de positionnement et de changements de repères nécessaires.  
Ex : translate (par rapport au local ou au parent) ; rotate (idem) ; matrice worldLocal ; localWorld ; pointToWorld( $P_{Local}$ ) ; directionToWorld( $U_{Local}$ ) ; etc ; etc.
- On fera plus simple lors des TPs : uniquement un niveau.



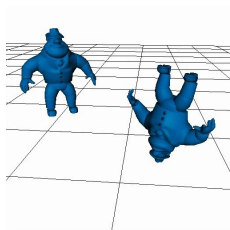
- ▶ Si les matrices homogènes s'imposent (composition simple par produit, librairie OpenGL), la représentation du placement est souvent couplée à une représentation Translation/Rotation.
- ▶ Tout placement d'un objet rigide (composition de translations et rotations) peut se réduire à  $TR$  : c'est à dire une translation (c'est la **position** de l'objet) et une rotation (c'est l'**orientation** de l'objet).
- ▶ On peut éventuellement compléter par un changement d'échelle  $S$  pour contrôler les dimensions d'un objet.

```
class Object3D {  
    Vector3 _position; // T  
    Quaternion _orientation; // R  
    Vector3 _scale; // S  
    ...  
};
```

- ▶ Quaternion ?

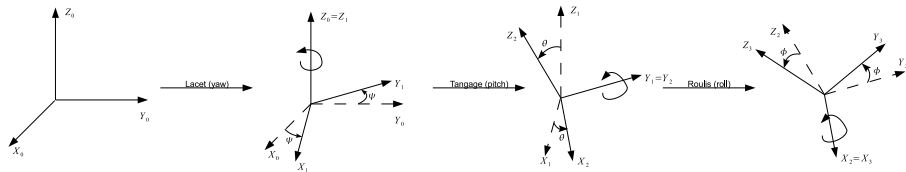
# Représentation d'une orientation

- ▶ Plusieurs choix pour représenter l'orientation :
  - matrices (quel est l'axe ? l'angle ? ; erreurs numériques lors de compositions multiples ; interpolation  $M = (1 - \lambda)M1 + \lambda M2$ ).

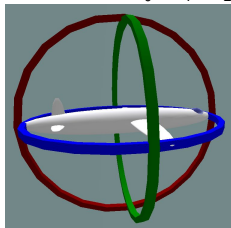


- angle/axe : intuitif mais pas nécessairement évident à manipuler (composition)
- angles de Cardan ("abusivement" appelés angles d'Euler en informatique graphique).
- quaternions.

# Remarques sur angles d'Euler



$\Rightarrow M_{0 \rightarrow 3} = R_{Z_0} R_{Y_1} R_{X_2}$  (ordre  $Z - Y - X$ ).



```
rotation(bleu); // lacet (yaw)
rotation(rouge); // tangage (pitch)
rotation(vert); // roulis (roll)
```

Remarque : en informatique graphique, l'ordre des axes est généralement  $Y - X - Z$  : autour de  $y$  (= lacet), puis autour de  $x$  (= tangage), puis autour de  $z$  (= roulis).

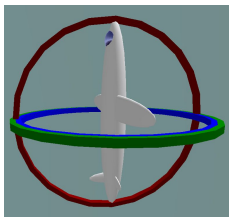
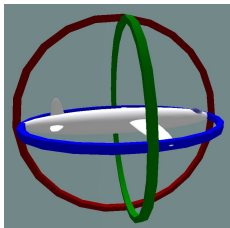
# Représentation par angles d'Euler (2)

```
class Object3D {
    Vector3 _position; // position en translation
    double _ay,_ax,_az; // orientation (angles d'Euler)
    ...
public:
    ...
}

Draw :
p3d::modelview.translate(_position.x(),_position.y(),_position.z());
p3d::modelview.rotate(_ay,0,1,0); // yaw : autour de y
p3d::modelview.rotate(_ax,1,0,0); // pitch : autour de x
p3d::modelview.rotate(_az,0,0,1); // roll : autour de z
...
```

- ▶ Représentation peut sembler simple : 3 angles, et l'orientation totale est alors donnée par  $M = R_{Y_0} R_{X_1} R_{Z_2}$ , mais la composition pose un véritable problème de conception :
  - Comment composer par rapport à l'orientation actuelle ? (exemple de l'interaction à la souris : comment faire un yaw **après** un pitch ??).
  - Gimbal lock : possible de perdre un degré de liberté.

# Gimbal Lock



rouge = 90 degrés

```
rotation(bleu); // yaw  
rotation(rouge); // pitch  
rotation(vert); // roll
```

# Représentation d'une orientation par quaternion (1)

- ▶ Définition :  $q = (a, u)$  avec  $a$  un scalaire de  $\mathbf{R}$ , et  $u$  un vecteur de  $\mathbf{R}^3$  (donc 4 composantes en tout).
- ▶ Un vecteur  $u$  peut s'écrire avec le quaternion  $q = (0, u)$ .
- ▶ Somme :  $q_1 + q_2 = (a, u) + (b, v) = (a + b, u + v)$ .
- ▶ Multiplication :  $q_1 q_2 = (a, u)(b, v) = (ab - u \cdot v, u \times v + av + bu)$
- ▶ Conjugué :  $(a, u)^* = (a, -u)$
- ▶ Multiplication par un scalaire :  $kq = k(a, u) = (ka, ku)$
- ▶ Norme :  $\|q\| = \sqrt{a^2 + u \cdot u}$

# Représentation par quaternion (2)

- ▶ Calcul à partir de 2 vecteurs :  $u$  et  $v$  **unitaires**, et formant un angle  $\theta$  alors le quaternion  $q = (u \cdot v, u \times v)$  est une représentation d'une rotation de vecteur  $u \times v$  et d'angle  $2\theta$ .
- ▶ Tout quaternion **normé** s'écrit  $q = (\cos\alpha, \sin\alpha u)$  avec  $u$  **normé** : représente une rotation d'angle  $2\alpha$  et de vecteur  $u$  (axe passant par l'origine).
- ▶ Soit  $q$  un quaternion représentant une rotation, alors l'image  $w'$  du vecteur  $w$  s'obtient par  $w' = qwq^*$
- ▶ Soient  $q_1$  et  $q_2$  deux rotations, alors la rotation composée est  $q_1 q_2$ .

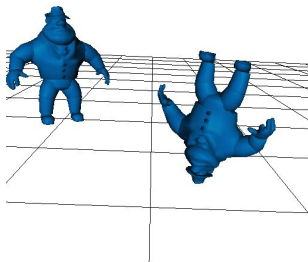
- Soit  $q = (a, u) = (a, (x, y, z))$  une rotation, alors la matrice homogène de rotation est :

$$\begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2za & 2xz + 2ya & 0 \\ 2xy + 2za & 1 - 2x^2 - 2z^2 & 2yz - 2xa & 0 \\ 2xz - 2ya & 2yz + 2xa & 1 - 2x^2 - 2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



# Intérêt des quaternions

- ▶ représentation de la composition moins gourmande en temps de calcul (comparer à la multiplication entre matrice).
- ▶ composition des rotations plus robuste : il suffit de normaliser le quaternion pour s'assurer que nous avons toujours une rotation.
- ▶ interpolation : l'interpolation linéaire de deux quaternions donne un résultat plus naturel qu'avec les matrices (i.e.  $q_3 = (1 - \lambda)q_1 + \lambda q_2$ , mais il faut toutefois normaliser  $q_3$ ).



**Remarque :** l'attribut `_position` est également interpolé linéairement.

- ▶ La classe `Quaternion` est disponible pour les tps.
- ▶ Les méthodes utiles sont :
  - Initialisation à la rotation identité : `q.setIdentity()`
  - Conversion d'une représentation (angle,axe) à un quaternion :  
`q.setFromAngleAxis(angle,axe)` (axe de type `Vector3`)
  - Conversion d'un quaternion à une représentation (angle,axe) :  
`q.copyToAngleAxis(&angle,&axe)`
  - Composition de 2 rotations : `q3=q1*q2`
  - Composition avec une rotation représentée par (angle,axe) : `q.rotate(angle,axe)`  
(q est modifié)
  - Rotation d'un vecteur : `v=q*u` (avec u et v de type `Vector3`)

- Il faut différencier les points  $P(x, y, z)$  et les directions  $u(x, y, z)$  lors des transformations. Une direction **ne doit pas subir la translation** :
- Avec les matrices homogènes : appliquer une coordonnée homogène 0 pour les vecteurs suffit.
  - Avec une représentation translation/rotation (quaternion par exemple) : la direction subit uniquement la rotation (il faut donc explicitement ignorer la translation).