Chapitre 2 : représentation Position/Orientation 3D

Modélisation 3D et Synthèse

Fabrice Aubert
fabrice.aubert@lifl.fr

IEEA - Master Info - Parcours IVI

2012-2013

Découpler modèle/vue

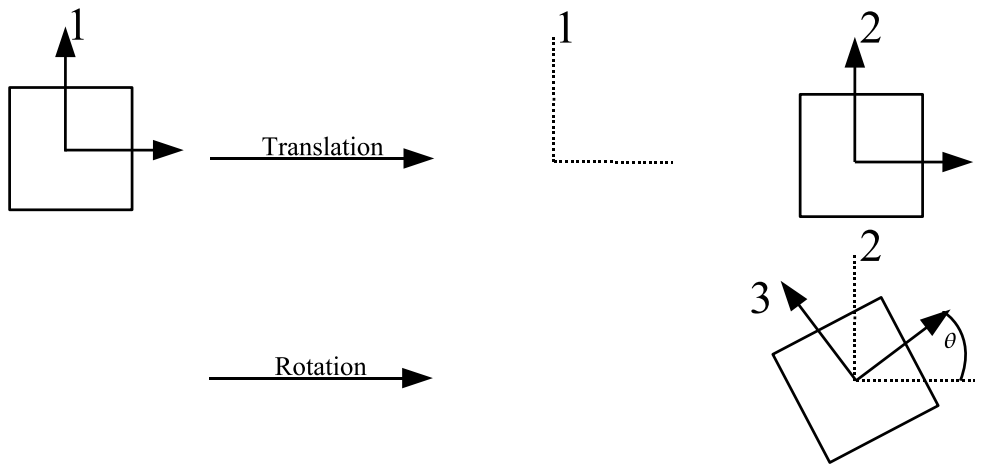
- Matrices homogènes : utilisées par la majorité des librairies graphiques (i.e. visualisation, ou vue)
- Cependant, pour représenter les données d'une scène (i.e. **le modèle**), les matrices homogènes ne sont pas nécessairement les plus pertinentes.
- $\Rightarrow$ représentation par translation/quaternion.

Représentation des données d'une scène

- Une scène est constituée d'objets (`class Entity`), et d'une caméra (`class Camera`).
- On considère 3 repères fondamentaux pour la scène (i.e. les données) :
  - Le repère global : `World`.
  - Le repère de la caméra : `Eye`.
    - La caméra est placée par rapport au repère global.
    - Ce placement correspond à $M_{World \rightarrow Camera}$
  - Le repère local d'un objet : `Object`. C'est le repère dans lequel sont définies les coordonnées des points de l'objet.
    - L'objet est placé par rapport au repère global.
    - Ce placement correspond à $M_{World \rightarrow Object}$

Représentation d'un placement rigide

- Toute combinaison de translations et rotations peut se réduire à une translation suivie d'une rotation (i.e. passage en TR).
- La translation T est appelée **position**
- La rotation R est appelée **orientation**
- Ce type de placement est dit placement à composante rigide.
- Pour représenter le placement d'un objet (ou d'une caméra) par rapport au repère global, on peut donc se contenter d'une translation et d'une rotation :



Proposition de code

```
class Movable {
  // placement par rapport au repère World
  Vector3 _t; // position en translation
  Orientation _r; // orientation (par exemple un axe et un angle)
public:
  ... (* inclure toute méthode de transformation : translate, rotate, etc *)
}

class Entity : public Movable { // 3D object
  ...
  public:
    drawWorld(); // i.e. tracé dans le repère World
    drawLocal(); // i.e. tracé dans le repère Object
};

class Camera : public Movable { // camera
  ...
  public:
  ...
};

class Scene {
  Entity _objet;
  Camera _camera;
  ...
  public:
  ...
  void drawScene();
};
```

Tracé par OpenGL

```
class Entity : public Movable {
  ...
  public:
    drawWorld();
    drawLocal();
};

void Entity::drawWorld() {
  glPushMatrix();

  glTranslated(_t.x(),_t.y(),_t.z());
  glRotated(_r.angle(),_r.axe().x(),_r.axe().y(),_r.axe.z());
  drawLocal();

  glPopMatrix();
}
```

```
class Scene {
  ...
  public:
  ...
  void drawScene();
};

void Scene::drawScene() {
  glPushMatrix();
  // ATTENTION : en OpenGL on part du repère Camera !

  // Eye->World : visualisation depuis la caméra
  glRotated(-_camera.r().angle(),_camera.r().axe().x(),_camera.r().y(),_camera.r().z());
  glTranslated(-_camera.t().x(),-_camera.t().y(),-_camera.t().z());

  // tracé dans le repère world des objets
  _objet.drawWorld();
  glPopMatrix();
}
```

Représentation d'une orientation

- Plusieurs choix pour représenter l'orientation :
  - angle/axe : intuitif mais pas nécessairement évident à manipuler lors de compositions.
  - matrices homogènes.
  - angles de Cardan.
  - quaternions.

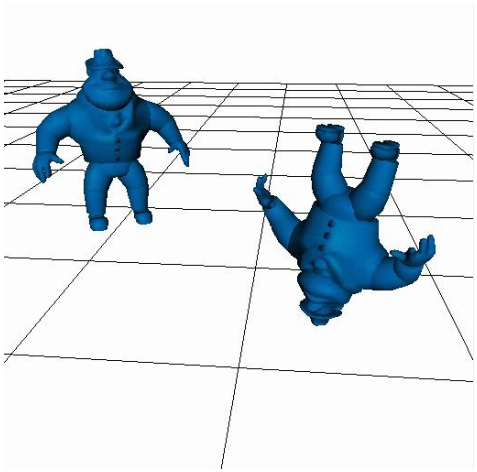
Représentation par matrices homogènes (1)

- Représentation générique. Inclut la translation.

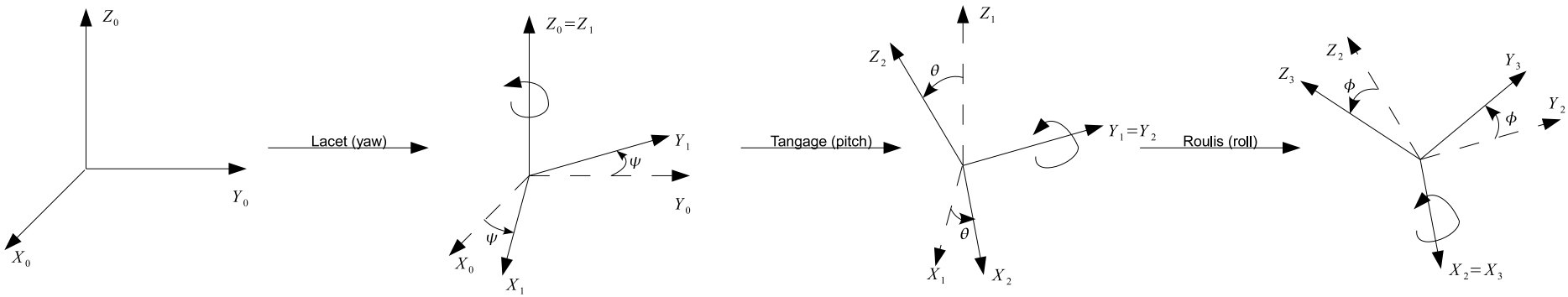
```
class Movable {
  Matrix4d _placement; // placement en translation et orientation
  ...
  public:
  ...
};

void Entity::drawWorld() {
  glPushMatrix();

  glMultMatrixd(_placement.dv()); // .dv() retourne le tableau des 16 coefficients (type double *) de la matrice

  drawLocal();

  glPopMatrix();
}
```

Représentation par matrices homogènes (2)

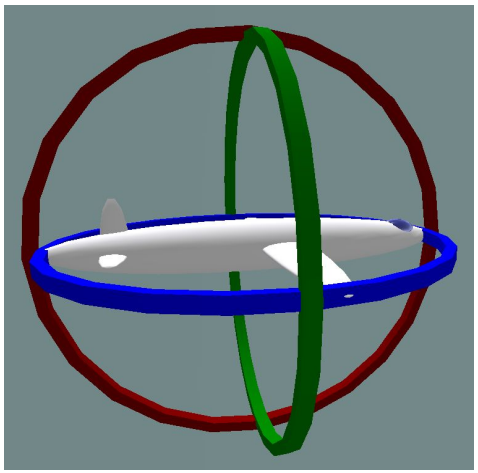
- Composition par simples produits de matrices.
- Inconvénients :
  - Les valeurs de la matrice peuvent s'avérer non intuitive (la décomposition TR est plus intuitive).
  - Les compositions (i.e. multiplication des matrices) peuvent induire des erreurs numériques : matrice de rotation non stable.
  - Ne s'interpole pas naturellement (i.e. le calcul $M = (1 - \lambda)M1 + \lambda M2$ donne un résultat non naturel).



Représentation par angles de Cardan



$\Rightarrow M_{0\to 3} = R_{Z_0} R_{Y_1} R_{X_2}$



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

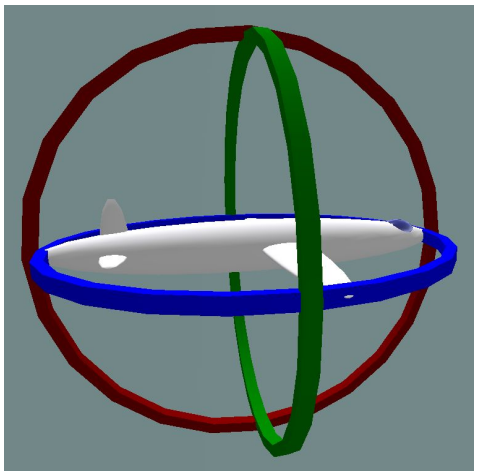
Remarque : en informatique graphique, l'ordre des axes est interprété selon le contexte ; par exemple pour le mouvement de caméra : autour de y (= lacet), puis autour de x (= tangage), puis autour de z (= roulis).

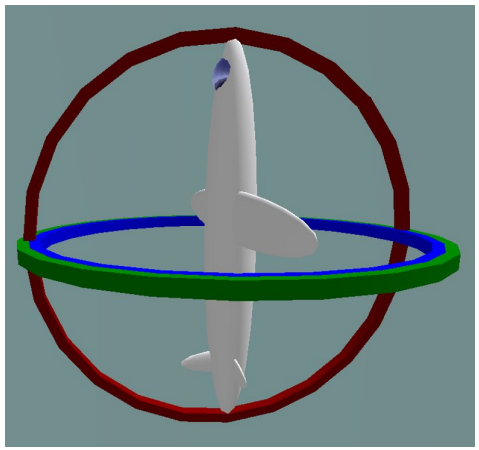
Représentation par angles de Cardan (2)

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

void Entity::drawWorld() {
  glPushMatrix();

  glTranslatef(_position.x(),_position.y(),_position.z());
  glRotatef(_ay,0,1,0); // yaw : autour de y
  glRotatef(_ax,1,0,0); // pitch : autour de x
  glRotatef(_az,0,0,1); // roll : autour de z
  drawLocal();

  glPopMatrix();
}
```

- 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 plusieurs orientations ?
  - Gimbal lock : possible de perdre un degré de liberté.

Gimbal Lock





rouge = 90 degrés

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

Représentation d'une orientation par quaternion (1)

- Définition : $q = (a, u)$ avec a un scalaire de $\boldsymbol{R}$, et u un vecteur de $\boldsymbol{R}^3$ (donc 4 composantes en tout).
- On représente un vecteur u par 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)

- Soient u et v deux vecteurs **unitaires**, et formant un angle θ alors le quaternion $q = (u \cdot v, u \times v)$ est une représentation d'une rotation de vecteur $u \times v$ et d'angle 2θ.
- Tout quaternion **normé** peut s'écrire $q = (cos\alpha, sin\alpha u)$ avec u **normé** et peut représenter une rotation d'angle 2α 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 $q1$ et $q2$ deux rotations, alors la rotation composée est $q1q2$.

Placement par quaternion

```
class Entity {
  Vector3 _position; // position en translation
  Quaternion _orientation // orientation (quaternion)
  ...
  public:
    drawWorld();
    drawLocal();
}

void Entity::drawWorld() {
  glPushMatrix();

  glTranslated(_position.x(),_position.y(),_position.y());

  // conversion du quaternion en (angle/axe)
  double angle;
  Vector3 axe;
  _orientation.copyToAngleAxis(&angle,&axe);

  glRotated(angle,axe.x(),axe.y(),axe.z());

  drawLocal();

  glPopMatrix();
}
```

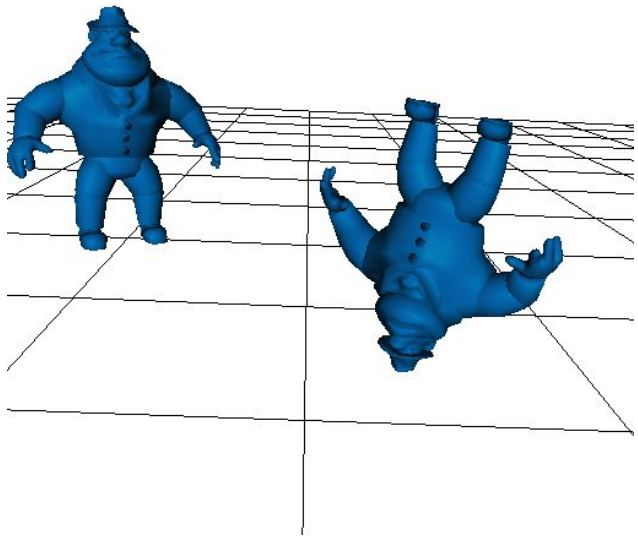
Conversion en matrice

- 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 (comparer avec les matrices : il faudrait orthonormaliser la matrice...).
- interpolation : l'interpolation linéaire de deux quaternions donne un résultat plus naturel qu'avec les matrices (i.e. $q3 = (1 - \lambda)q1 + \lambda q2$, mais il faut toutefois normaliser $q3$).



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

Classe Quaternion

- 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`)

Transformation d'un point/direction

- On connait le changement de repère $M_{1\to 2}$ représenté par une position et un quaternion (champs `_position` et `_orientation` d'une classe `Movable` par exemple) :
- Transformation d'un point : (Attention à l'ordre ! $M_{1\to 2} = TR \Rightarrow P_1 = TRP_2$)

```
void Movable::transformPoint(Vector3 *p) { // *p donné dans repère 2
  _orientation->transform(p); // applique la rotation à p
  p->add(_position); // applique la translation
}
```

- Transformation d'une direction : (subit uniquement la rotation !)

```
void Movable::transformDirection(Vector3 *u) { // *u donné dans repère 2
  _orientation->transform(u); // applique la rotation
}
```