UNIVERSITÉ DU LITTORAL CÔTE D'OPALE

TP OpenGL 3
Création d'objets

Licence Informatique 3ème année

Année 2010-2011

Préliminaires

- Créez un dossier nommé Tp3 dans le dossier qui ocntient vos TP d'informatique graphique ;
- Y recopier les sources développées lors du TP2.

1 Objets prédéfinis par `Glut`

La librairie `Glut` fournit quelques fonctions permettant de construire des objets « classiques » , tels que des sphères, des cônes, des tores et ... des théières !!! Ces objets peuvent être construits soit en *fil de fer*, soit avec des faces pleines. Nous donnons ci-après quelques-unes des fonctions qui peuvent être utilisées.

1.1 La sphère

Les fonctions permettant de créer une sphère sont les suivantes :

- `glutSolidSphere(GLdouble r, GLuint tranches, GLuint quartier)` : permet de créer une sphère de rayon `r` avec des faces pleines, l'approximation de la sphère pouvant être réglée en précisant le nombre de « tranches » et de « quartiers » à utiliser pour générer les facettes approchant le contour de la sphère ;
- `glutWireSphere(GLdouble r, GLuint tranches, GLuint quartier)` : même principe que la fonction précédente, sauf que la sphère est générée en fil de fer.

A noter que le centre de la sphère créée se trouve à l'origine du repère global.

Application : remplacez, dans votre fonction `dessiner`, l'appel à la fonction `cube` du TP précédent, par la création d'une sphère en fil de fer. Compilez et testez.

1.2 Le cône

On retrouve ici le même type de fonction que pour la sphère :

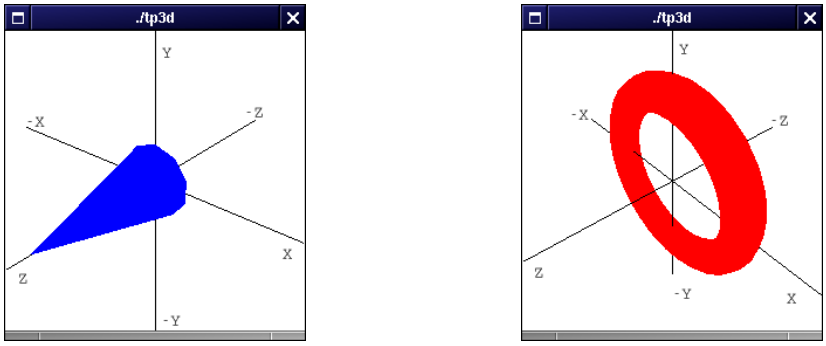
- `glutSolidCone(GLdouble r, GLdouble h, GLuint t, GLuint q)`

– `glutWireCone(GLdouble r, GLdouble h, GLuint t, GLuint q)`

avec :

– `r` : rayon du cercle à la base du cône ;
– `h` : la hauteur du cône ;
– `t` : le nombre de quartiers autour du cône ;
– `q` : le nombre de tranches le long de la hauteur.

Le cône créé est aligné le long de l'axe des Z positifs (voir figure FIG 1.a), le centre de sa base étant situé à l'origine.



(a) (b)

FIGURE 1 – Géométrie et position initiale du cône (a) et du tore (b)

Application : remplacez, dans votre fonction `dessiner`, la sphère par un cône solide. Compilez et testez.

1.3 Le tore

Les deux fonction suivantes permettent de créer un tore d'axe Oz centré à l'origine (voir figure FIG 1.b) :

– `glutSolidTorus(GLdouble ri, GLdouble ro, GLuint c, GLuint a)`
– `glutWireTorus(GLdouble ri, GLdouble ro, GLuint c, GLuint a)`

On peut considérer qu'il est obtenu par rotation d'un anneau autour de l'axe Oz.

Les paramètres sont les suivants :

– `ri` : rayon de l'anneau ;
– `ro` : rayon du cercle décrit par le centre de l'anneau dans sa rotation autour de l'axe Oz ;
– `c` : nombre de segments approximant l'anneau ;
– `a` : nombre de segments utilisés pour approximer le grand cercle parcouru par l'anneau.

Application : remplacez, dans votre fonction `dessiner`, le cône par un tore solide. Compilez et testez.

1.4 La théière

Les deux fonctions suivantes permettent de créer **la** teapot, centrée à l'origine du repère, le paramètre permettant de spécifier la taille de l'objet.

- `glutSolidTeapot(GLdouble taille)`
- `glutWireTeapot(GLdouble taille)`

Application : remplacez, dans votre fonction `dessiner`, le tore par une théière solide. Compilez et testez.

2 Construction d'objets polygonaux

Nous avons vu dans le premier Tp sur OpenGL, comment tracer des points et des lignes en deux dimensions. Nous allons à présent examiner les primitives permettant de construire des objets polygonaux en 3D.

2.1 Définir un point

Comme en 2D, la définition d'un point en 3D se fait par l'intermédiaire de l'une des fonctions `glVertex`. Comme il s'agit ici de points en trois dimensions et que les coordonnées sont a priori réelles, la fonction à utiliser sera donc nommée `glVertex3f`. On lui fournira alors comme paramètres les trois coordonnées x, y et z du point. On peut aussi utiliser la forme vectorielle de la méthode : `glVertex3fv`.

Exemple :

```
// on peut écrire
glVertex3f(1.2f, 3.8f,-5.8f);
// ou bien
float unPoint[] = {1.2f, 3.8f,-5.8f};
glVertex3fv(unPoint);
```

2.2 Définir des primitives 3D

Pour définir une primitive 3D, il est nécessaire :

- de préciser le type de primitive que l'on souhaite construire ;
- d'énumérer les points qui permettent de construire cette primitive.

La spécification de la primitive à construire se fait par l'intermédiaire de la fonction `glBegin()`, dont la paramètre précise à OpenGL comment il doit interpréter les points qui suivent. Parmi les différentes possibilités offertes par OpenGL, nous en citons quelques unes ci-dessous :

- `GL_LINES` : définition d'une suite de segments, chaque segment étant définie par un couple de points successifs ;

– `GL_TRIANGLES` : définition d'une suite de triangles, chaque triangle étant défini par trois points successifs ;
– `GL_QUADS` : définition d'une suite de quadrilatères ;
– `GL_POLYGON` : définition d'un polygone construit avec tous les points qui suivent. A noter que le polygone doit être convexe ;
– `GL_QUAD_STRIP` : définition d'une suite de quadrilatères à partir des points $V_0, V_1, V_2, V_3, V_4, V_5, \ldots$; les quadrilatères construits sont $(V_0, V_1, V_3, V_2), (V_2, V_3, V_5, V_4) \ldots$
– `GL_TRIANGLE_FAN` : définition d'une suite de triangles à partir des points $V_0, V_1, V_2, V_3, V_4, \ldots$; les triangles construits sont $(V_0, V_1, V_2), (V_0, V_2, V_3), (V_0, V_3, V_4) \ldots$

Enfin, la fonction `glEnd()` doit être appelée dès que la ou les primitives construites à partir d'un appel à `glBegin` sont terminées. Vous trouverez un exemple d'utilisation dans les fonctions `repere cube` de `graphique.c` du `tp2`.

3 Amélioration de l'affichage

Vous devez vous être rendu compte que l'affichage obtenu n'est pas toujours très fluide, en particulier lorsque l'on applique des transformations rapides sur les objets en laissant, par exemple, le doigt appuyé sur l'une des touches autorisées du clavier ... Cela provient du fait qu'OpenGL dessine directement dans le buffer image ; on a alors une impression désagréable de clignotement à chaque fois que l'image est effacée et pas beaucoup plus agréable lorsque l'on voit [1] les facettes s'afficher les unes après les autres.

Pour contourner ce problème, il est possible d'utiliser une technique connue sous le nom de **double buffer image** : un premier buffer image est supposé contenir une image « finale » et c'est celui qui est utilisé pour lire l'image et l'afficher à l'écran ; le second est utilisé par OpenGL pour le calcul de l'image suivante. Lorsque l'image est terminée, les deux buffers sont intervertis, l'image qui vient d'être calculée pouvant alors être affichée, tandis que le calcul de l'image suivante peut être commencé dans l'autre buffer.

Par défaut, OpenGL utilise un seul buffer qui sert alors à la fois pour lire l'image et la calculer. Ceci explique le fait que l'on peut parfois distinguer la construction de la scène, mais aussi percevoir les effacements successifs du contenu du buffer par la couleur de fond.

Il est cependant possible d'utiliser la technique du double buffer via la librairie Glut, au travers de deux appels de fonctions :

– dans la fonction `main`, la fonction `glutInitDisplayMode` sert à initialiser différents paramètres liés à l'affichage. Son appel actuel est le suivant :

```
glutInitDisplayMode(GLUT_DEPTH);
```

Sous cette forme, cette fonction active le Z-buffer. Elle admet un certain nombre d'autres valeurs possibles comme paramètres, ces valeurs étant toutes définies comme une valeur entière dont **un seul bit** est activé. De ce fait, il est possible de **composer**

1. La vision fugitive de l'affichage successif des facettes dépend beaucoup à la fois des capacités de la carte graphique et du nombre de facettes composant la scène. Sur une scène de faible complexité, ce phénomène n'apparaîtra que sur des cartes graphiques de faibles capacités. Par contre, sur des scènes très complexes, il peut apparaître même sur des cartes puissantes ...

les valeurs à activer par l'intermédiaire d'un **ou-logique**. Sachant que la valeur associée au double buffer est symbolisée par la constante `GLUT_DOUBLE`, on peut activer (aussi) ce mécanisme par l'appel suivant :

```
glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE);
```

– lorsque toutes les instructions nécessaires à la construction d'une image ont été exécutées, vous devez **explicitement** demander à OpenGL d'effectuer l'interversion des deux buffers. Ceci se fait par un appel à la fonction `glutSwapBuffers()`. Dans le cas de votre application, cet appel doit être le dernier de la fonction `dessiner`.

4 Définir des objets complexes

Vous allez dans ce Tp créer une application qui modélise un « avion » (figure Fig 2) et le déplace en utilisant les touches du clavier (comme le cube était déplacé dans le Tp précédent). Pour cela vous allez créer et assembler, petit à petit, des morceaux plus ou moins complexes de cet objet.

(a) (b)

Figure 2 – Avion vu de profil (a) et vu de face (b)

4.1 Ajout d'une primitive cylindre

Vous allez commencer par créer une nouvelle fonction qui permet de représenter les faces latérales d'un cylindre (on ne s'intéresse pas aux deux faces circulaires des extrémités).

1. Écrivez, dans le fichier `graphique.c`, la fonction de signature

```
void cylindre(float r, float h, int nb)
```

qui réalise la facettisation de la surface latérale du cylindre en `nb` rectangles dont les grands côtés sont des méridiens du cylindre. Le cylindre admet l'axe Oz comme axe

de symétrie et il a ses faces circulaires de rayon `r` situées dans les plans d'équations respectives $z = 0$ et $z = h$.

2. Testez cette fonction dans votre fonction `dessiner`.

4.2 Création du fuselage de l'avion

Le fuselage de l'avion est constituée d'un cylindre terminé à l'avant par une demi-sphère et à l'arrière par un cône.

1. Créez, dans le fichier `graphique.c`, la fonction de signature

   ```
   void fuselage();
   ```

   dont le rôle est de modéliser le fuselage de l'avion avec les primitives requises. Pensez à préciser une couleur pour votre fuselage ...

2. Modifiez la fonction `dessiner`, compilez votre application et testez-la.

5 La pile des matrices

De manière à pouvoir gérer facilement la multiplication de transformations géoémtriques qui apparaissent lors de la modelisation d'objets complexes, OpenGL offre un mécanisme de pile pour les matrices de transformation [2]. Le principe en est très simple : lorsque la matrice de transformation courante va être modifiée par une transformation, mais que l'on souhaite pouvoir réutiliser la matrice courante ultérieurement, on empile la matrice. Elle est ainsi sauvegardée dans la pile, d'où on la dépilera lorsque cela sera nécessaire. Deux fonctions sont fournies :

- `glPushMatrix()` : empile une copie de la matrice de transformation courante ;
- `glPopMatrix()` : dépile la matrice de transformation qui se trouve en sommet de pile ; elle devient alors la matrice de transformation courante.

Exemple :

```
void un_objet()
{
    glMatrixMode(GL_MODELVIEW);
    /* dessin d'une partie */
    glPushMatrix(); /* sauvegarde de la matrice courante */
    glRotatef(90.0, 1.0, 0.0, 0.0);
    /* dessin d'une deuxième partie */
    glPopMatrix(); /* restauration de la matrice courante */
}
```

2. En fait OpenGL offre ce mécanisme aussi pour les matrices de projection. Nous n'utiliserons dans ce Tp que la pile des matrices de tranformation, sachant que la gestion des matrices de projection est similaire.

5.1 Création d'une hélice

Vous allez rajouter une hélice à votre fuselage. Cette hélice sera composée d'un cône et de deux pales, une pale étant représentée par un polygone à quatre côtés. Les deux pales seront positionnées vers l'extrêmité pointue du cône (voir la figure Fig 3). Il est fortement conseillé, dans ce qui suit, d'utiliser le mécanisme de pile de matrice, afin de vous faciliter la tâche ...

1. Écrivez une fonction `pale` qui permet de construire l'une des pales de l'hélice, le polygone correspondant devant être défini dans le plan Oxy pour plus de simplicité.
2. En utilisant cette fonction, écrivez la fonction `helice` qui permet d'obtenir l'hélice telle qu'elle apparaît sur la figure Fig 3. Il est suggéré de tester la construction de l'hélice indépendamment du fuselage, par exemple en commentant le dessin de celui-ci ...
3. Modifiez la fonction `dessiner` pour placer votre hélice à l'avant du fuselage. Compilez et testez votre application.

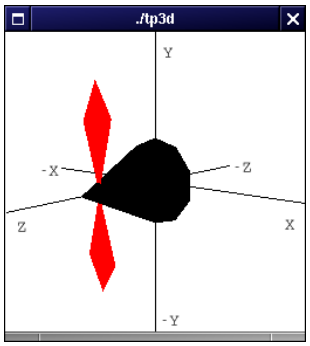


Figure 3 – Géométrie de l'hélice

6 Les « Display List »

Jusqu'à présent les objets 2D ou 3D étaient tracés en **mode immédiat**, c'est à dire qu'à chaque nouvelle image, toutes les primitives à tracer figurant entre un appel à `glBegin` et à `glEnd` étaient relues et ré-exécutées totalement. Il existe une autre façon d'effectuer les affichages sous OpenGL, par l'intermédiaire de structures de données appelées `Display List`, qui vont être détaillées dans ce qui suit.

6.1 Présentation

Les `Display List` sont des structures gérées entièrement par OpenGL. L'utilisateur doit simplement préciser comment construire chaque `Display List` et laquelle (ou lesquelles) appeler pour construire l'image.

D'un point de vue pratique, les `Display List` offrent quelques avantages :

- lors de la construction d'une `Display List`, OpenGL optimise la représentation mémoire de l'objet qui y est défini ; ceci peut conduire à de meilleures performances en terme d'affichage ;
- lorsque le même objet doit être utilisé plusieurs fois dans la scène, il n'est construit qu'une seule fois ; seules les éventuelles transformations géométriques servant à le positionner seront recalculées ;
- en mode client-serveur (application sur une machine, affichage sur une autre), la `Display List` n'est transmise qu'une seule fois et mémorisée sur le client, ce qui réduit les débits nécessaires et améliore d'autant la « fluidité » de l'affichage (réduction des temps d'attente).

En contrepartie, une `Display List` ne peut pas être modifiée de quelque manière que ce soit ; si un attribut quelconque figurant dans la `Display List` (géométrie, couleur, texture, ...) doit être modifié, il est nécessaire de supprimer la `Display List` et d'en reconstruire une nouvelle ...

6.2 Méthodes OpenGL

6.2.1 Identification

OpenGL gère ses `Display List` par l'intermédiaire d'un **numéro** identifiant chaque liste de manière unique. L'obtention d'un tel numéro se fait par l'intermédiaire de la méthode suivante :

```
GLuint glGenLists(GLsizei nb)
```

avec :

- `nb` : le nombre de numéros souhaités ; on peut effectivement demander à obtenir plusieurs numéros de `Display List` d'un seul coup, les numéros fournis étant forcément consécutifs ;
- valeur de retour :
  - un numéro de `Display List` (si `nb=1`) ;
  - le premier numéro d'une série de `Display List` (si `nb > 1`) ;
  - 0 si aucun numéro ne peut être attribué.

6.3 Création

Lorsqu'un numéro valide de `Display List` a pu être obtenu, il est possible de créer une nouvelle `Display List` (ayant cet identifiant) par l'intermédiaire de deux méthodes, l'une précisant le début de la `Display List`, l'autre la fin :

- `void glNewList(GLuint identifiant, GLenum mode)` : spécifie le début de la création d'une `Display List` portant le numéro `identifiant`. Le paramètre mode peut prendre les deux valeurs suivantes :
  - `GL_COMPILE` : construit la `Display List` et la range dans la liste d'affichage d'OpenGL ;

– `GL_COMPILE_AND_EXECUTE` : actions identiques, mais provoque en plus l'affichage immédiat du contenu de la liste.

De manière générale, c'est la valeur `GL_COMPILE` qui est la plus fréquemment utilisée, puisque l'on construit habituellement toutes les `Display List` avant de lancer le processus d'affichage.

– `void glEndList(void)` : termine la création de la `Display List`.

Entre ces deux fonctions doivent se trouver toutes les commandes OpenGL permettant de construire un objet. À noter cependant que toutes les fonctions OpenGL ne peuvent pas être appelées dans le cadre d'une `Display List`. Ces cas particuliers dépendent de la version d'OpenGL, la liste pouvant en être trouvée dans tout manuel de référence [3].

6.4 Exemple

```
GLuint monObjet; // servira à stocker le numéro d'une display list
monObjet = glGenLists(1);// demande d'un numéro de display list
if(monObjet==0) /* gérer l'erreur */
else{
  // création de la display list
  glNewList(monObjet, GL_COMPILE);
    glBegin(GL_POLYGON);
      glColor3f(1.0f, 0.0f, 0.0f); /* rouge */
      glVertex3f(0.0f, 0.0f, 0.0f);
      glVertex3f(1.0f, 0.0f, 0.0f);
      glVertex3f(1.0f, 1.0f, 0.0f);
    glEnd();
    glBegin(GL_POLYGON);
      glColor3f(0.0f, 1.0f, 0.0f); /* vert */
      glVertex3f(0.0f, 0.0f, 1.0f);
      glVertex3f(1.0f, 0.0f, 1.0f);
      glVertex3f(1.0f, 1.0f, 1.0f);
    glEnd();
  glEndList();
}
```

6.5 Utilisation

Lorsque les `Display List` ont été construites, elles sont rangées dans la liste d'affichage d'OpenGL. Il est cependant nécessaire de les appeler explicitement lorsque l'on souhaite déclencher leur visualisation. Ceci se fait par l'intermédiaire de la méthode suivante :

```
void glCallList(Gluint identifiant)
```

3. Aucune des fonctions utilisées dans ce Tp ne posera de problème dans le cadre des `Display List`.

qui déclenche l'affichage de la `Display List` portant le numéro `identifiant`.

Bien entendu, il est possible de modifier tous les états souhaités d'OpenGL (matrice de transformation, épaisseur des lignes, ...) avant d'appeler une `Display List`, ces états influençant alors l'affichage de la liste.

6.6 Suppression

Lorsqu'une `Display List` n'a plus d'utilité au sein d'une application, il est conseillé de l'effacer, à la fois pour des raisons évidentes d'occupation mémoire, mais aussi pour pouvoir récupérer l'identifiant qui lui est affecté. Ceci peut se faire avec la méthode suivante :

```
void glDeleteLists(GLuint identifiant, GLsizei nombre)
```

qui efface les `nombre` listes consécutives dont la première correspond à `identifiant`.

6.7 Application

Comme application directe de ce qui vient d'être vu, vous allez modifier le code de création et d'affichage de votre avion dans son état actuel en utilisant les `Display List`.

1. Déclarez les nouvelles variables globales suivantes dans le fichier `graphique.c` (pensez à mettre à jour le fichier `graphique.h`) :
   - `GLuint leFuselage` : contiendra le numéro de la `Display List` contenant les instructions permettant de dessiner le fuselage ;
   - `GLuint uneHelice` : idem mais pour une hélice ;
2. Ajoutez au fichier `graphique.c` une fonction de prototype :

   ```
   int initDisplayLists()
   ```

   dont le rôle est de créer les `Display List` qui représentent les différents objets constituant l'avion. Vous utiliserez les fonctions `fuselage()` et `helice()`.

   Cette méthode devra retourner la valeur `1` si les `Display List` ont pu être créées, et la valeur `0` sinon.
3. Modifiez la fonction `main` en y ajoutant un appel de la fonction `initDisplayLists()`. Si l'appel ne réalise pas l'initialisation souhaitée alors le programme devra être arrêté. À noter que vous ne pouvez créer de géométrie OpenGL que lorsque le contexte graphique de la librairie a lui-même été créée.
4. Modifiez la fonction `dessiner` de telle sorte qu'elle appelle les différentes `Display List` pour dessiner l'avion.

7 Amélioration de l'avion

7.1 Création des roues

Dans ce qui suit, vous allez être amenés à ajouter deux roues à votre carlingue ; une roue sera représentée par un tore et quatre cônes, le tore représentant le pneu et les cônes

représentant les rayons de la roue.

7.1.1 Création d'un tore

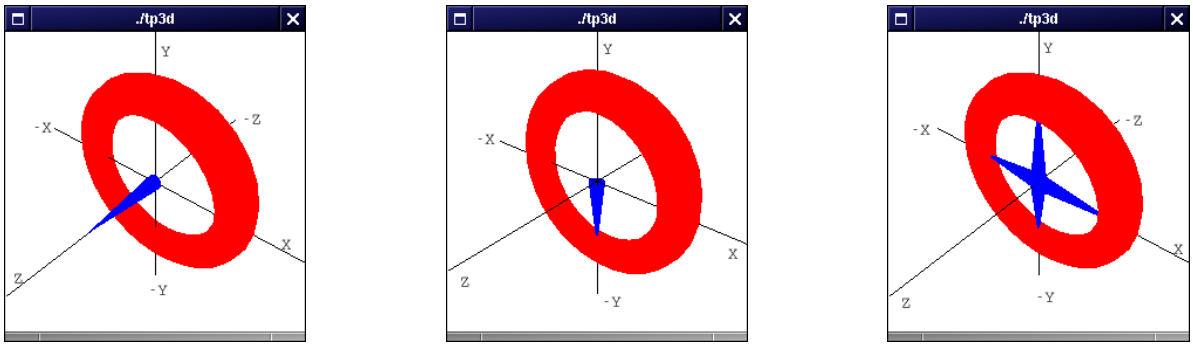
1. Créez, dans le fichier `graphique.c`, la fonction de signature

   ```
   void roue(float rayonRoue, float rayonPneu);
   ```

   dont l'objectif est de créer une roue, de rayon `rayonRoue`, à partir d'un tore. Le second paramètre correspondra au rayon du pneu, c'est à dire au premier paramètre de la fonction `glutSolidTorus`. Pensez à préciser une couleur pour votre tore ...
2. Ajoutez, dans `graphique.c`, une variable globale permettant de conserver l'index de la *liste d'affichage* qui représentera la roue.
3. Modifiez la fonction `initDisplayLists()` de manière à lui rajouter la création d'une *liste d'affichage* représentant la roue. Cette *liste d'affichage* devra, temporairement, inclure le dessin du repère ...
4. Modifiez la fonction `dessiner` de manière à ce que seule la roue apparaisse pour le moment (commenter les lignes correspondant au fuselage et à l'hélice).
5. Compilez votre application et testez-la.

7.1.2 Ajout des rayons

1. Modifiez la fonction `roue` de telle sorte qu'elle dessine en plus un cône dont le rayon soit inférieur ou égal au rayon du pneu et la hauteur égale au rayon interne de la roue. Vous devez obtenir ce qui apparaît sur la figure Fig 4.a.



(a) (b) (c)

Figure 4 – Géométrie et position initiale du rayon de la roue (a) et position finale (b)

2. Compilez votre application et testez-la.
3. Complétez la fonction `roue` de manière à y rajouter les quatre rayons à la bonne position (figure FIG 4.b et 4.c) ;

7.1.3 Positionnement des roues

1. Complétez la fonction `dessiner` de telle sorte qu'elle positionne la roue sur l'un des côtés du fuselage.
2. Faites de même pour positionner une seconde roue de l'autre côté.
3. Complétez le train d'atterrissage en y ajoutant une connexion entre les roues et le fuselage (par exemple en utilisant un ou plusieurs cylindres).

7.2 Création des ailes

Ajoutez des ailes à votre avion. Avant d'être positionnées les deux ailes seront obtenues en une seule instruction déformant une sphère. Il ne vous reste plus, si vous le souhaitez, qu'à compléter votre avion à votre convenance ...