Géométrie pour la 3D

Modélisation 2D et 3D

Arash HABIBI

On cherche à créer des polygones dans le plan et dans l'espace.

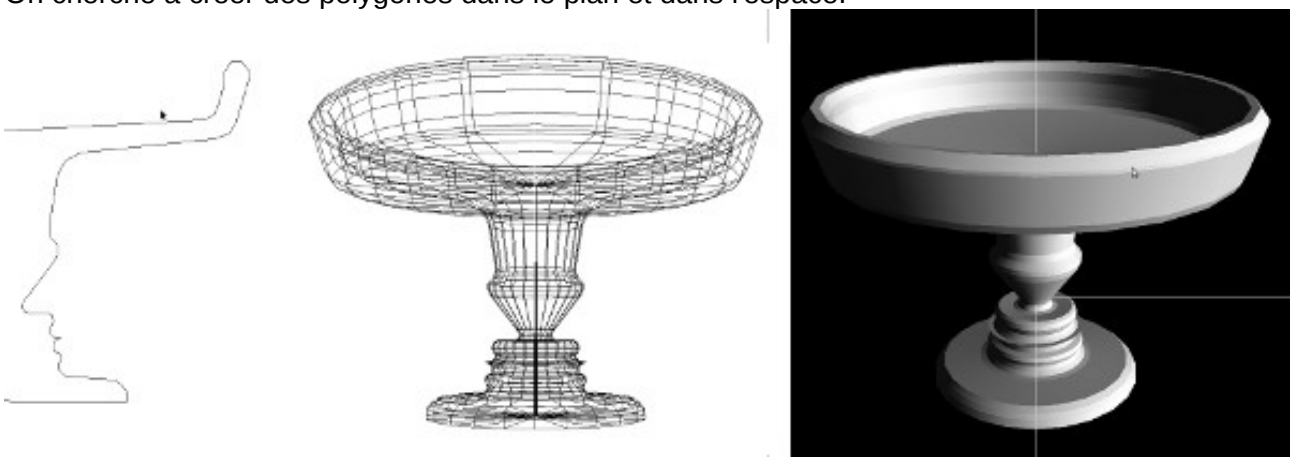

Figure 1. Modélisation d'un polygone en 2D (à gauche) et en 3D (au milieu en filaire et à droite avec des ombres). Les polygones en 3D ont été obtenus à partir du polygone de gauche par révolution.

Pour ce faire, nous allons partir du canevas TP2.tgz que vous pourrez télécharger depuis Moodle.

1. Bibliothèque sur les vecteurs

Pour commencer vous travaillerez sur les fichiers Vector.h et Vector.c qui contiennent la définition du type vecteur et les profils de quelques fonctions de traitement des vecteurs. C'est à vous de les compléter :

```
Vector V_add(Vector v1, Vector v2) ;
// retourne le résultat v1 + v2

Vector V_substract(Vector v1, Vector v2) ;
// retourne le résultat v1 – v2

Vector V_multiply(double lambda, Vector v) ;
// retourne le résultat lambda v

Vector V_cross(Vector v1, Vector v2) ;
// retourne le produit vectoriel : v1 ^ v2

Vector V_dot(Vector v1, Vector v2) ;
// retourne le produit scalaire : v1.v2

int V_isOnTheRight(Vector M, Vector A, Vector B) ;
// retourne 1 si M est à la droite de la droite (AB) et 0 sinon

int V_segmentsIntersect(Vector p1, Vector p2, Vector q1, Vector q2) ;
// retourne 1 si le segment [p1p2] intersecte le segment [q1q2] et 0 sinon.

int V_rayIntersectsSegment(Vector M, Vector u_ray,
Vector p1, Vector p2) ;
// retourne 1 si la demie-droite d'origine M et de direction u_ray
// coupe le segment de droite [p1p2] et 0 sinon.
```

Cette dernière fonction est plus difficile que les autres. Précisons qu'elle ne nécessite pas de trouver

le point d'intersection. Dans un premier temps, trouver l'algorithme pour déterminer si la demie-droite intersecte la droite (p1p2). Ensuite déterminer si l'intersection a lieu sur le segment [p1p2].

2. Dessin élémentaire 2D

A présent, nous allons créer des polygones plans. Vous aurez peut-être remarqué que, contrairement au TP précédent, la projection n'est pas réalisée par la fonction `gluPerspective`, mais par la fonction `glOrtho`. Cela signifie qu'il s'agit d'une projection orthogonale. Les paramètres de `glOrtho`, c'est à dire 6 flottants, (xmin, xmax, ymin, ymax, znear, zfar) qualifient la projection. znear et zfar ont la même signification que dans `gluPerspective` : tout ce qui est plus proche que znear et plus loin que zfar n'est pas visualisé. Les autres paramètres sont propres à `glOrtho` mais ont une signification semblable : tout ce qui est plus à gauche que xmin, plus à droite que xmax, plus bas que ymin et plus haut que ymax n'est pas visualisé. Ainsi l'appel à `glOrtho` spécifie que si nous dessinons un point dont l'abscisse vaut xmin (resp xmax), alors ce point apparaîtra sur le bord gauche (resp. droit) de la fenêtre. Il en va de même pour ymin et ymax. Attention !!! Par contre les gestionnaires des événements souris continuent à représenter la position de la souris par deux entiers compris respectivement entre [0,largeur_fenetre] et [0,hauteur_fenetre].
Dessiner un quadrilatère dont les sommets touchent les bords de la fenêtre en leur milieu.

3. Polygones plans

Nous représenterons un polygone un ensemble de sommets, chaque sommet étant représenté par un vecteur dans le plan (x,y). Le nombre de sommets peut varier pendant la construction. Plutôt que d'utiliser une liste chaînée, nous représenteront un polygone plan par un tableau de 1000 vecteurs et un entier `nb_sommets` représentant le nombre (variable) de sommets du polygone. Définir le type `polygon` et écrire les fonctions suivantes : .

- **`void P_init(Polygon *p) ;`** // initialise un polygone (0 sommets)
- **`void P_copy(Polygon *original, Polygon *copie) ;`**
  // original et copie sont deux polygones déjà alloués. Cette fonction copie les donnée
  // depuis original vers copie de façon à ce que les deux polygones soient identiques.
- **`void P_addVertex(Polygon *P, Vector pos) ;`**
  // ajoute un sommet au polygone P. Ce nouveau sommet est situé en pos.
- **`void P_removeLastVertex(Polygon *P) ;`** // enlève le dernier sommet de P
- **`void P_draw(Polygon *P) ;`** // dessine le polygone P
- **`void P_print(Polygon *P, char *message) ;`**
  // Ecrit sur une console les données de P à des fins de debuggage.

4. Saisie de polygones plans

- Écrire un programme qui à chaque clic gauche de souris ajoute ce point comme un sommet du polygone P et affiche l'ensemble du polygone (utiliser la fonction `drawline) ;`
- faire de façon à ce qu'à chaque clic droit le dernier sommet soit éliminé ;
- faire de façon à ce qu'en frappant 'c', le polygone affiché soit un polygone fermé ;
- écrire une fonction **`P_isConvex`** qui retourne 1 si le polygone donné en paramètre est convexe et 0 sinon ;
- faire de façon à ce qu'à chaque sommet ajouté, on vérifie si le polygone saisi est convexe ou non ; afficher le polygone en rouge s'il est convexe et en bleu s'il ne l'est pas ;
- faire de façon à ce qu'il ne soit pas possible de créer un polygone non-simple ;
- faire de façon à ce qu'un clic avec le bouton du milieu arrête la saisie du polygone (autrement dit, les clics suivants ne créent pas de nouveaux sommets).

5. Intérieur-extérieur d'un polygone

- Écrire une fonction qui détermine, pour un point M et pour un polygone convexe P, si M appartient à l'intérieur de P ou non.
- Même chose pour un polygone non-convexe.
- Écrire une fonction **`P_isInside`** qui détermine pour un point M et pour un polygone P

quelconque si M appartient à l'intérieur de P ou non. Faire de façon à ce que, lorsque la saisie du polygone est terminé, chaque clic de souris affiche un message à l'écran indiquant si on a cliqué à l'intérieur ou à l'extérieur du polygone.

6. Maillage polygonal (polygon mesh) dans l'espace

Un maillage polygonal est un assemblage de polygones qui permet de décrire des formes complexes en 3D. De la même façon qu'on considère un polygone comme un tableau de sommets, on peut considérer un maillage polygonal comme un tableau de polygones. Or la structure `Polygon` que nous avons choisie est assez lourde (puisqu'elle contient par défaut 1000 sommets). Et dans le cas des extrusions et révolutions, on n'aura affaire qu'à des quadrilatères. Donc plutôt que de manipuler des tableaux de tableaux de 1000 vecteurs, nous allons définir un nouveau type `quad` qui sera un tableau statique de 4 Vecteurs représentant les 4 sommets du quadrilatère. Un maillage polygonal sera alors un tableau de `quads`.
Définir le type quad et écrire la fonction :

```
Quad Q_new(Vector v1, Vector v2, Vector v3, Vector v4) ;
```

qui, à partir de 4 vecteurs, retourne un quad dont les sommets sont ces 4 vecteurs.

- **`void M_init(Maillage *M) ;`** // initialise un maillage (0 quads)
- **`void M_addQuad(Maillage *M, Quad q) ;`**
  // ajoute au maillage le quadrilatère q
- **`void M_draw(Maillage *M) ;`**
  // dessine le maillage M
- **`void M_print(Maillage *M, char *message) ;`**
  // Ecrit sur une console les données de M à des fins de debuggage.

Tester ces fonctions en dessinant quelques quadrilatères (un cube par exemple) définis en dur dans le programme (et non saisis à la souris). Utiliser les fonctions vues au TP précédent (gluPerspective et les opérations matricielles) pour tourner autour de cet objet 3D.

7. Révolution

Nous avons besoin de quelques fonctions supplémentaires traitant les vecteurs …

- **`double V_length(Vector v) ;`**
  // retourne le module du vecteur v
- **`Vector V_unit(Vector v) ;`**
  // retourne un vecteur colinéaire à v mais de longueur 1
- **`Vector V_turnAroundY(Vector p, double radians) ;`**
  // Tourne d'un angle de radians radians le point p autour de l'axe Y et retourne le résultat.
  // Donnez-vous le temps de réfléchir. Ca n'a rien d'immédiat. Dans un premier temps,
  // vous pouvez considérer que p est sur l'axe des abscisses.

… les polygones.

```
void  P_turnAroundY(Polygon *P, double radians) ;
```

// tourne tous les points de P d'un angle de radians radians autour de l'axe Y.

Elles nous permettent de réaliser les opérations suivantes :

- **`void M_addSlice(Maillage *M, Polygon *P1, Polygon *P2) ;`**
  // P1 et P2 sont supposés être des polygones ayant le même nombre N de sommets.
  // Cette fonction ajoute à M les N quads obtenus en reliant deux à deux les sommets de P1
  // à ceux de P2.

- **`void M_revolution(Maillage *M, Polygon *P, int nb_tranches) ;`**
  // A partir d'un polygone P, cette fonction divise 2π en nb_tranches angles et ajoute à M les
  // quads nécessaires pour réaliser une révolution de P autour de l'axe Y (cf figure 1).

Écrire un programme qui, à partir d'un polygone P dans le plan (x,y) défini en dur dans le programme (par exemple un carré) réalise une révolution de cette figure.

8. Synthèse (facultative)

Faire évoluer le programme de la question 4 (permettant de saisir un polygone à la souris) pour que, une fois la saisie terminée, le fait de frapper la touche 'r' (comme *révolution*) réalise la révolution de la figure saisie et passe d'une visualisation 2D (glOrtho) à une visualisation 3D (gluPerspective) permettant d'examiner votre œuvre sous tous les angles.

Compléter encore le programme pour qu'en frappant la touche 'a' (comme *affichage*) on puisse passer successivement d'un affichage en fil de fer à un affichage à plat puis à un affichage ombré. Pour cela, vous pourrez vous inspirer du programme affichage.tgz que vous trouverez également sur Moodle. Ce programme affiche un cube avec les trois visualisations (toujours en pressant la touche 'a'). On peut tourner autour du cube avec les touches flèches et changement de page. On peut aussi déplacer la source de lumière avec les touches flèches + Ctrl.

A titre de question de réflexion : dans le programme `affichage.c`, vous pouvez voir les fonctions **glNormal3f(x,y,z)** qui (vous l'aurez deviné) spécifient les vecteurs normaux à chaque sommet. Deux questions s'imposent :

- Pourquoi spécifie-t-on les vecteurs normaux aux sommets et non aux faces ?
- Pourquoi a-t-on besoin de spécifier les vecteurs normaux ? Pourquoi est-ce qu'OpenGL ne les calcule-t-il pas automatiquement ?

Pour nourrir votre réflexion, je voudrais vous montrer de nouveau la figure 1. Sur la figure du milieu on peut voir que la résolution de l'objet n'est pas très grand. Pour la révolution, on a divisé 2π en 16 tranches seulement et chaque tranche est bien visible. La figure de droite représente exactement le même maillage avec le même nombre de polygones. Pourtant la surface paraît lisse et on ne voit pas la décomposition en polygones. Comment est-ce possible ?