Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Master Informatique 1

TD Infographie n°2

Routines d'affichage

Routines d'affichage OpenGl *à partir de points, normales, facettes, en mode* `POINTS`, `TRIANGLES`, `QUADS`...

---

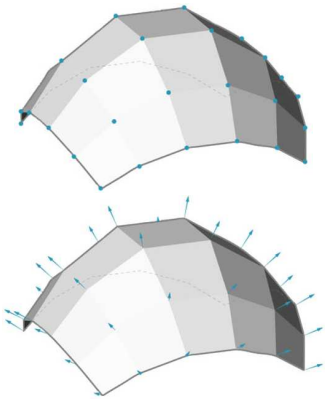
① Représentation par couple (vertex/normale)→faces.

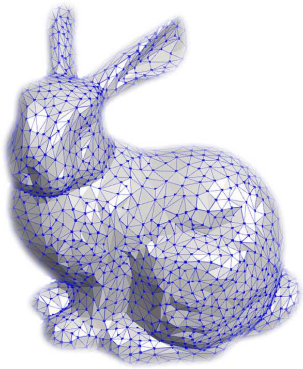
Le mode de visualisation que nous utiliserons ici est basée sur un approximation polygonale des formes (comme une courbe 2D représentée par une suite de segments). Ce mode de représentation (*Mesh-Based*) constitue 99,9% des usages de la "3D". D'autres modes de visualisation (le RayTracer en particulier) fonctionnent différemment, mais s'appuient généralement sur des objets de type *mesh*.

Une forme (surface) sera donc représentée par un ensemble de points (on les appelle vertex) disposés judicieusement sur sa surface.

Afin de modéliser les interactions lumière/matière, il sera indispensable d'associer une normale à chacun de ces vertex(cf TD4-Illum.)

Ces points seront ensuite "liés" dans un ordre bien précis pour former des faces.





Un objet sera donc stocké sous la forme de deux tableaux, propres à l'objet, de nb vertex et nb normales.

On pourra éventuellement leur associer un troisième tableau représentant les faces (triplets ou quadruplets d'indices indiquant comment les vertex sont liés). Cette indication est indispensable dans le cas de données issues de fichiers de points (le lapin ci-dessus), mais pas dans le cas de formes procédurales, définies à partir d'équations.

Il faut bien distinguer ce qui relève de la modélisation (construction des vertex/normales/faces) de ce qui relève du "rendu" et de l'affichage.
☞ les tableaux seront créés lors de l'étape (unique) de *chargement* (la fonction init()).
☞ les routines d'affichage sont à appeler **exclusivement** dans le *moteur de rendu* (fonction draw()).

La libg3x travaille avec des structures (cf. <g3x_geom.h>) alors que OpenGl travaille avec des tableaux.
☞ la libg3x fourni quelques fonctions d'adaption en respectant la 'nomenclature' OpenGl (cf. man OpenGl ...).

	OpenGl	libg3x
points/vecteurs	`double P[3],V[3];`	`typedef struct {double x,y,z;} G3Xpoint, G3Xvector;`
appel vertex	`glVertex3d(x,y,z);`	
	`glVertex3dv(double P[3]);`	`g3x_Vertex3dv(G3Xpoint P);`
appel normale	`glNormal3d(x,y,z);`	
	`glNormal3dv(double V[3]);`	`g3x_Normal3dv(G3Xvector V);`
couleurs RGBA	`float c[4];`	`typedef struct {float r,g,b,a;} G3Xcolor`
appel couleur	`glColor4f(r,g,b,a);`	
	`glColor4fv(float c[4]);`	`g3x_Color4dv(G3Xcolor c);`

Dans la suite on considère que l'on a créé (peu importe comment pour l'instant) deux tableaux, de même taille `nb` :

```
#define nb /* nombre de points */
G3Xpoint  P[nb];
G3Xvector N[nb];
```

ⓐ **rendu par points** : ça consiste à n'afficher que les vertex (avec, bien sûr, leur normales).

```
01| glPointSize(s);                          /* fixe la taille du point (float s=1. par défaut) */
02| g3x_Material(col,ambi,diff,spec,shin,0.); /* fixe les attibuts couleur/matière               */
03| glBegin(GL_POINTS);                       /* balise ouvrante : affichage en mode POINT       */
04|   for (i=0; i<nb; i++)                    /* parcours les tableaux                           */
05|   {
06|     g3x_Normal3dv(N[i]);                 /* d'abord la normale                              */
07|     g3x_Vertex3dv(P[i]);                 /* ensuite le vertex                               */
08|   }
09| glEnd();                                 /* balise fermante : fin de l'affichage            */
```

→ **quelques explications :**

01| La taille du "point" (exprimée en pixels, même si le paramètre est un `float`) est une *variable d'état* (cf. plus loin)

02| La gestion des attributs couleur/matière (`g3x_Material`) sera présentée ultérieurement[1] mais elle est **indispensable** (sans elle, pas de '3D').
Ses paramètres sont, par exemple[2] :

```
G3Xcolor col=(G3Xcolor){1.0,0.7,0.4,0.0}; /* un 'orange' clair */
float ambi=0.2,diff=0.6,spec=0.8,shin=0.99;
```

03| la commande `glBegin(GL_ATTRIBUTE)` (et son dual `glEnd()`) est la plus fondamentale à maîtriser. Elle définit le mode d'affichage, et conditionne la façon donc OpenGl va gérer les vertex et les normales (individuellement ou en les groupant).

Dans ce premier cas, le plus simple, l'attribut `GL_POINTS` indique que les vertex seront traités individuellement (pas d'association en facette). Cette *balise ouvrante* déclenche l'affichage : tant qu'il ne rencontre pas un appel à la *balise fermante* `glEnd()`, le système affiche tels quels tous les vertex qu'il rencontre, avec les paramètres couleur/matière définis par `g3x_Material`.

06| 07| l'**indispensable** association (vertex/normale)
Les fonctions `"g3x_Normal3dv"` et `"g3x_Vertex3dv"` sont les adaptations libg3x des fonctions OpenGl `"glNormal3dv"` et `"glVertex3dv"` (cf. manuel OpenGl ...).

☞ le vertex donne la position, mais c'est la normale qui permet à OpenGl de gérer l'illumination (cf. TD4-illum), c'est à dire l'interaction lumière/surface.

☞ attention : on donne d'**abord** la normale !

☞ si plusieurs points ont la même normale, il n'est pas nécessaire de la répéter : c'est une vaiable d'état, comme la couleur, elle garde sa valeur tant qu'on ne la change pas :

```
00| /* affiche une face carrée orientée vers le haut                                  */
01| glNormal3d(0.,0.,+1);           /* la normale, commune à tous les vertex suivants */
02| glBegin(GL_QUADS);              /* balise ouvrante : affichage en mode QUAD (cf. plus loin) */
03|   glVertex3dv((-1,-1,0);        /* 4 vertex formant un carré (attention à l'ordre !!!) */
04|   glVertex3dv((-1,+1,0);
05|   glVertex3dv((+1,+1,0);
06|   glVertex3dv((-1,+1,0);
07| glEnd();                        /* balise fermante : fin de l'affichage           */
```

09| la balise fermante marquant la fin de ce 'bloc' d'affichage
☞ mais tous les `GL_ATTRIBUTS` restent les mêmes (c'est des variables d'état d'OpenGl)

[1] cf. TD.4-illum.
[2] cf. également le programme d'exemple g3x_demo.c qui vous a été fourni avec la libg3x

ⓑ **rendu par triangle** : ça consiste grouper les points par 3 pour former des facettes triangulaires.

```
01|                                          /* plus besoin de régler la taille du 'point'      */
02| g3x_Material(col,ambi,diff,spec,shin,0.); /* fixe les attibuts couleur/matière               */
03| glBegin(GL_TRIANGLES);                    /* balise ouvrante : affichage en mode TRIANGLE    */
04|   for (i=0; i<nb; i++)                    /* parcours les tableaux                           */
05|   {
06|     g3x_Normal3dv(N[i]);                 /* d'abord la normale                              */
07|     g3x_Vertex3dv(P[i]);                 /* ensuite le vertex                               */
08|   }
09| glEnd();                                 /* balise fermante : fin de l'affichage            */
```

→ **quelques explications :**

03| On retrouve la balise ouvrante, avec cette fois l'attribut `GL_TRIANGLES` : OpenGl va traiter les points (et leurs normales) par paquets de 3 dans l'ordre où ils arrivent et va les associer pour former des triangles.

06| 07| Dans cet exemple (peu pertinent), les triangles formés seront donc, bêtement, `<P[0],P[1],P[2]>` puis `<P[3],P[4],P[5]>`, etc... jusqu'au dernier qui sera `<P[nb/3-3],P[nb/3-2],P[nb/3-1]>`. Si il reste des points (2 maxi.), ils seront ignorés.

Une façon beaucoup plus pertinente de fonctionner consisterait à créer, en plus des tableaux de vertex et normales, un tableau de `nf` facettes `int F[nf][3];`.

☞ une face est définie par 3 indices `(i,j,k)` renvoyant vers un triplet de vertex/normale.

☞ ce tableau est lui aussi construit[3] dans l'étape de "chargement" (init()).

La routine d'affichage deviendrait alors quelque chose comme :

```
01| g3x_Material(col,ambi,diff,spec,shin,0.);
02| glBegin(GL_TRIANGLES);
03|   for (f=0; f<nf; f++)                  /* parcours le tableau des FACETTES               */
04|     for (i=0; i<3; i++)                  /* pour chaque face F[f]...                       */
05|     {
06|       g3x_Normal3dv(N[F[f][i]]);        /* ... on va chercher les bons vertex/normales    */
07|       g3x_Vertex3dv(P[F[f][i]]);
08|     }
09| glEnd();
```

☞ Nous verrons par la suite qu'il est possible de s'affranchir de ce tableau de facettes lorsque la structure interne des tableaux `P` et `N` est bien concue à la base (ce que nous ferons...)

ⓒ **rendu par quadrilatères** : ça consiste grouper les points par 4.

☞ On utilise cette fois l'attribut d'affichage `GL_QUADS`.

☞ En réalité, OpenGl se débrouille pour construire 2 facettes triangulaires à partir de ces 4 points.

☞ Là encore, le recours à un tableau de facettes prédéfinies (à 4 indices : `int F[nf][4];`) rend les choses plus claires :

```
01| g3x_Material(col,ambi,diff,spec,shin,0.);
02| glBegin(GL_QUADS);
03|   for (f=0; f<nf; f++)                  /* parcours le tableau des FACETTES               */
04|     for (i=0; i<4; i++)                  /* pour chaque face F[f] (4 indices) ...          */
05|     {
06|       g3x_Normal3dv(N[F[f][i]]);        /* ... on va chercher les bons vertex/normales    */
07|       g3x_Vertex3dv(P[F[f][i]]);
08|     }
09| glEnd();
```

☞ Là encore, une bonne conception de la structure interne des tableaux `P` et `N` permet de s'affranchir de ce tableau de facettes

[3] pour des objets complexes (le lapin du début), ces données sont nécessairement fournies avec les coordonnées des vertex (les normales peuvent être reconstruites donc elles ne sont pas fournies en général)

② Une question critique : l'ordre d'association des vertex formant les facettes

Deux considérations évidentes mais fondamentales :

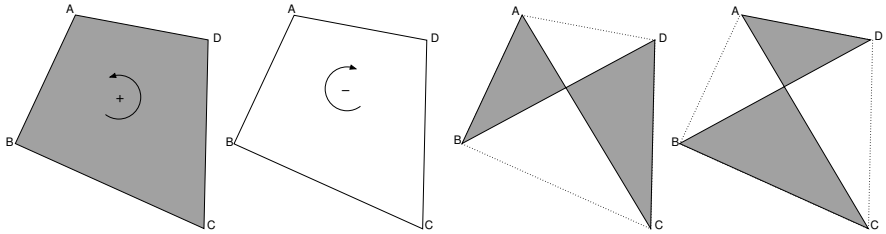
① dans l'espace 3d Infographique **tout** est ordonné et, en conséquence, orienté.

② OpenGl ne sait pas ce que vous voulez faire donc fait exactement ce que vous lui demandez.

Cela implique que, pour construire des facettes triangulaires, et plus encore des `GL_QUADS`, il est impératif de bien ordonner les vertex. En effet avec 3 vertex A,B et C, on peut former deux triangles (<ABC> et <ACB>). L'un sera d'orientation positive, l'autre négative.

OpenGl faisant une distinction entre la face positive (`GL_FRONT`) et négative (`GL_BACK`) cela peut produire des effets indésirables. Cela sera rédiscuté ultérieurement si nécessaire....

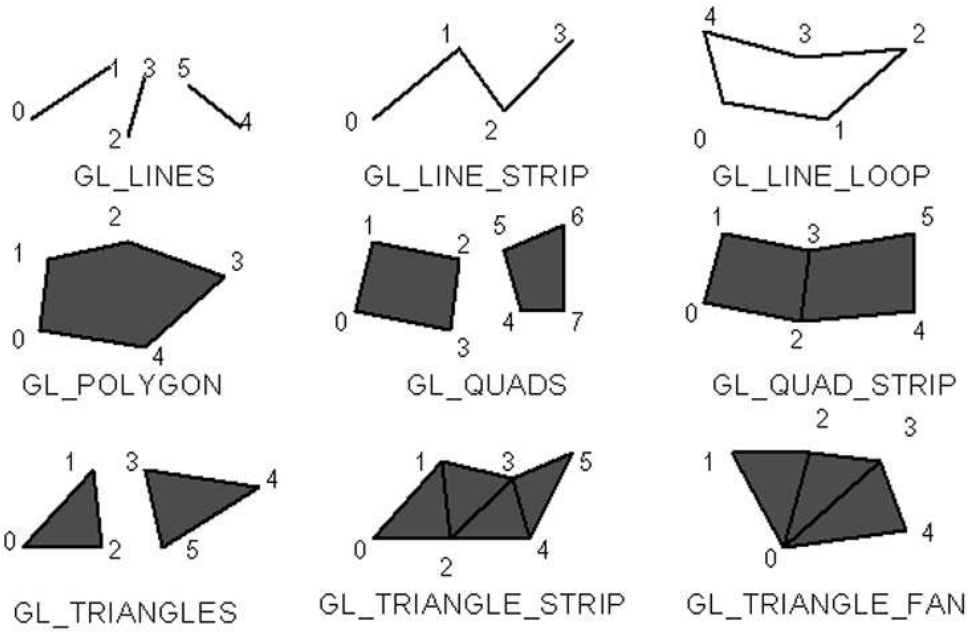
Avec les `GL_QUADS`, outre la question de l'orientation, il y a une notion d'ordre bien précise : <ABCD> et <ADCB> sont simplement d'orientations opposées, mais <ABDC> ou <ADBC> sont totalement différentes (croisements).



③ les autres `GL_ATTRIBUTS` d'affichage

OpenGl propose, en plus des modes `GL_POINTS`, `GL_TRIANGLES` et `GL_QUADS`, d'autres façons d'associer les vertex pour tracer des formes : en voici le résumé en image.

☞ Le plus important est de bien ordonner les vertex en fonction du mode choisi



☞ à voir également pour affiner sa compréhension du fonctionnement d'OpenGl pour le traitement des faces visibles/invisibles :

- `void glPolygonMode(GLenum face,GLenum mode)`
  `face`$\in \{$`GL_FRONT,GL_BACK,GL_FRONT_AND_BACK`$\}$
  `mode`$\in \{$`GL_POINT,GL_LINE,GL_FILL`$\}$
- `glEnable(GL_CULLING)`
- `void glCullFace(GLenum face)` `face`$\in \{$`GL_FRONT,GL_BACK,GL_FRONT_AND_BACK`$\}$
- https://www.khronos.org/opengl/wiki/Face_Culling
- https://learnopengl.com/Advanced-OpenGL/Face-culling