Sommaire intermédiaire

L'analyse des maillages obtenus nécessitent une structure de données efficace. On cherche :

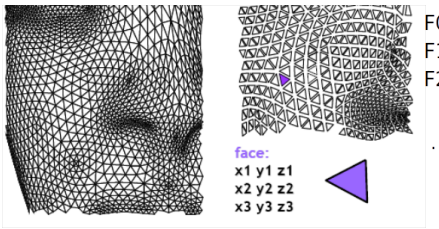
- une représentation compacte (taille mémoire)
- une représentation optimisée pour le parcours (voisinages)

Duplication des sommets...

pour chaque triangle, on donne la liste de sommets (c'est l'approche la plus naïve).

- 4 octets par coordonnée (un flottant simple)
- 9 coordonnées par faces
- 2 fois plus de faces que de sommets

pour un maillage composé de S sommets, on a donc besoin de $4 \times 9 \times 2 \times S = 72 \times S$ octets.

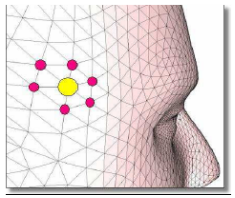


F0 : x_0,y_0,z_0 x_1,y_1,z_1 x_2,y_2,z_2
F1 : x_5,y_5,z_5 x_1,y_1,z_1 x_{12},y_{12},z_{12}
F2 : x_9,y_9,z_9 x_0,y_0,z_0
$x_{1025},y_{1025},z_{1025}$
...

Liste indexée de sommets

On d'abord stocke la liste des sommets, puis les faces sont définies en donnant les index des sommets qui la compose. Combien d'octets par sommets ? 36.

Liste des sommets	Liste des facettes
x_0,y_0,z_0	0 1 2
x_1,y_1,z_1	5 1 12
x_2,y_2,z_2	9 0 1025
...	...

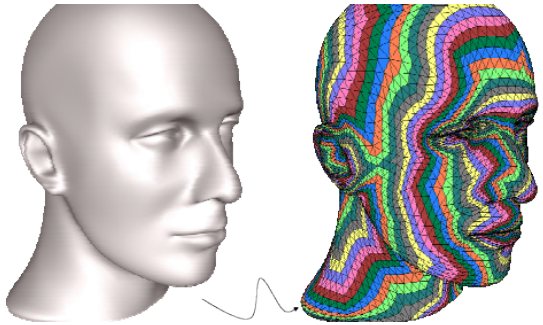


e.g. OFF / VRML

Sommets (géométrie)	Faces (connectivité)
v1 (x1;y1;z1)	f1 (v1;v3;v2)
v2 (x2;y2;z2)	f2 (v4;v3;v1)
v3 (x3;y3;z3)	f3 (v4;v1;v5)
v4 (x4;y4;z4)	f4 (v1;v6;v5)
v5 (x5;y5;z5)	f5 (v6;v1;v7)
v6 (x6;y6;z6)	f6 (v2;v7;v1)
v7 (x7;y7;z7)	f7 (...)

Triangles strip

On assemble les triangles par adjacence d'arête, on stocke (et on communique via les bus mémoire) moins d'information. Cela apporte également une accélération sur la recherche de voisinage (moyennes de normales pour l'éclairement par ex.).



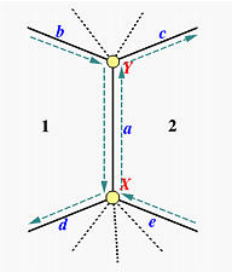
Triangles strip 2

Le calcul est long, difficilement optimal, adapté au rendu et pas aux calculs.

- Belmonte Fernández, Oscar & Ribelles, J & Remolar, Inmaculada & Chover, Miguel. Searching Triangle Strips Guided By Simplification Criterion.2001
- Francine Evans ; Steven Skiena & Amitabh Varshney. Optimizing triangle strips for fast rendering. Visualization 1996. IEEE. pp. 319-326.
- Regina Estkowski, Joseph S. B. Mitchell, Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. In Proceedings of Symposium on Computational Geometry'2002. pp. 254-263

Winged-edges & consorts

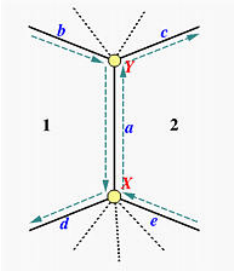
Winged-edge : comme une arête partage 2 faces, et est composée de 2 sommets. On stocke pour chaque arête les références vers les sommets "source" et "destination" (donc un sens de parcours), ainsi que les faces voisines ou les Winged-edge voisins (4).



Edge	Vertices		Faces		Left Traverse		Right Traverse	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
a	X	Y	1	2	b	d	e	c

Figure – `http://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html`

Winged-edges & consorts



Edge	Vertices		Faces		Left Traverse		Right Traverse	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
a	X	Y	1	2	b	d	e	c

Figure – `http://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html`

Cette structure recouvre la précédente (tableau de faces) qui peut être déduite. On peut se passer du tableau de faces, qui peut être déduit en parcourant les winged-edges.

Winged-edges & consorts2

Half-edge : similaire au winged-edge précédent, en décrivant chaque arête par 2 half-edges (un dans chaque sens de parcours). La moitié des informations est stockée, l'autre moitié étant déduite du second half-edge.

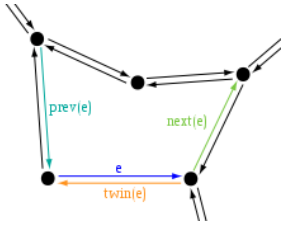


Figure – `https://en.wikipedia.org/wiki/Doubly_connected_edge_list`

S'appelle également DCEL (Doubly Connected Edge List).

Sommaire intermédiaire

Dans une boucle de rendu OpenGL, il serait contre-productif de :

1. transmettre les données géométriques et de voisinages (i.e. le tableau des points et celui des faces + la primitive à utiliser pour le tracé)
2. donner les ordres de tracé

à chaque image.

Le rendu peut être appelé aussi vite que possible (i.e. le *framerate* n'est pas fixe, dès qu'une image est prête elle est affichée) ou après un temps fixe (évenement d'horloge), ce qui fixe également le *framerate* mais impose une borne sup. sur la durée des calculs.

Depuis OpenGL 1.5 on peut se servir des `VertexBufferObject`. L'intérêt de ces structures est que l'on peut stocker directement la géométrie sur la carte graphique.

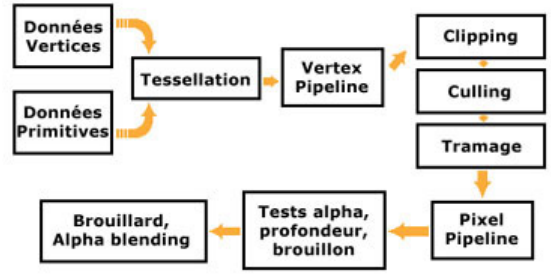
On sépare transmission des données géométriques et ordre de tracé. Les données sont transmises en amont (si leur géométrie n'est pas modifiée, elle n'est transmise qu'une fois).
Plusieurs techniques sont possibles : Vertex Arrays, Vertex Buffer Objects, Buffer Objects selon les version d'OpenGL.
L'utilisation se fait en 4 phases :

1. on remplit des tableaux pour stocker la géométrie d'un objet,
2. on lie ensuite ces tableaux avec des zones mémoires de la carte graphique,
3. on copie les données,
4. et hop, c'est prêt pour le rendu !

Du coup on peut désallouer la mémoire en RAM du CPU puisque la géométrie est sur la carte graphique. Il devient donc inutile de conserver la géométrie à la fois dans le CPU et le GPU, d'où un gain de mémoire en plus du gain de transfert.

Pipeline détaillé

Le pipeline graphique effectue une opération après l'autre (lorsque les données sont prêtes).



Le fragment en sortie de la projection 2D -> 3D (avant le `Pixel Pipeline`) contient pour chaque pixel :

- les coordonnées 2D entières,
- sa couleur,
- sa pseudo-profondeur (profondeur relative dans le volume de vue).

Tableaux vers le GPU

Comment fait-on ? On crée des tableaux qui vont contenir les données de l'objet (des tableaux de `float`), on met dedans toutes les données de l'objet chargé.

Préparation des données :

1. on crée les tableaux de géométrie (identique pour les Vertex Arrays)

```
float lcolors[nbsommets*3];
float lnormales[nbsommets*3];
float lsommets[nbsommets*3];
```

2. on crée les buffers GPU qui contiendront ces données (`glBindBuffer`, `glBufferData`)

À la création

```
//VBO pour les sommets
glGenBuffers((GLsizei) 1, &VBOSommets);
glBindBuffer(GL_ARRAY_BUFFER, VBOSommets);

glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float),
 lsommets, GL_STATIC_DRAW);

//VBO pour les couleurs
glGenBuffers((GLsizei) 1, &VBOCouleurs);
  glBindBuffer(GL_ARRAY_BUFFER, VBOCouleurs);
  glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float),lcolors, GL_STATIC_DRAW);

//VBO pour les normales
glGenBuffers((GLsizei) 1, &VBONormales);
  glBindBuffer(GL_ARRAY_BUFFER, VBONormales);
  glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float) , lnormales, GL_STATIC_DRAW);
```


Au rendu

Pour les sommets seulement :

```
glEnableClientState(GL_VERTEX_ARRAY);


  glBindBuffer( GL_ARRAY_BUFFER, VBOSommets);
  glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );

    glDrawArrays(GL_TRIANGLES, 0, nbfaces * 3);

glDisableClientState(GL_VERTEX_ARRAY);
```

Au rendu2

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

  glBindBuffer( GL_ARRAY_BUFFER, VBOSommets);
  glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );

  glBindBuffer( GL_ARRAY_BUFFER, VBOCouleurs);
  glColorPointer( 3, GL_FLOAT, 0, (char *) NULL );

  glBindBuffer( GL_ARRAY_BUFFER, VBONormales);
  glNormalPointer(GL_FLOAT, 0, (char *) NULL );

    glDrawArrays(GL_TRIANGLES, 0, nbfaces * 3);

glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```


Le mode des VBO

On peut gérer les VBO sous 3 modes :

- `GL_STREAM_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque rendu = VertexArrays. On envoie les tableaux à chaque image.
- `GL_DYNAMIC_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque frame. On utilise ce mode pour laisser la carte graphique gérer les emplacements mémoires des données, pour le rendu multi-passe notamment.
- `GL_STATIC_DRAW` lorsque les informations sur les sommets ne sont pas mises à jour - ce qui redonne le fonctionnement d'une DisplayList.
- Se rajoute à ces modes, des modes d'accès aux données : `READ`, `WRITE`, `COPY`, `READ_WRITE`. Ce qui permet également de copier des parties d'un objet dans un autre, et même de faire des interactions avec le CPU.

note : attention aux différents modes, cela ne fonctionne pas sur toutes les cartes graphiques.

Au fait les FBO ! ?

Un `Frame Buffer Object` (FBO) est l'espace de stockage qu'utilise OpenGL pour mettre, au fur et à mesure de son avancement, toutes les données issues d'algorithmes de l'espace image (lié à la résolution du rendu, en pixels). Par exemple les pixels de la scène, mais aussi le Z-buffer. Dans un fonctionnement statique, vous devez utilisez des stencils ou des masques pour interagir avec le Z-buffer, et vous ne pouvez pas modifier les pixels de l'image 2D rendue.

À partir d'OpenGL 2, on peut adresser ces espaces de stockages comme des textures (`Texture Object`) ou des buffers (`RenderBuffer Object`). Pour schématiser, les premiers sont faciles d'accès (par le programme principal ou un shader) et les seconds sont plus rapides (stockés en mémoire vive de la carte graphique, pas dans l'espace des textures).

Rendu d'un VBO

On active ensuite le FBO dans la boucle de rendu par :

`void glBindFramebufferEXT(GLenum target, GLuint id)`

avec target = `GL_FRAMEBUFFER_EXT`. Si on met target = 0, le FBO est désactivé.

Note : faire du rendu (`off-screen`) directement dans une image ou un fichier AVI -> c'est le `pbuffer`, et c'est différent.

On peut imaginer diverses utilisation des FBO, par exemple :

- effectuer un flou d'une scène selon la profondeur (comme dans les déserts par exemple) -> c'est faire 2 FBO (couleurs et profondeur) + 1 fragment shader qui parcourt la texture de profondeur et modifie la texture de couleur par un masque de cœfficients pris dans la profondeur (comme une convolution) ;
- appliquer des effets d'ombrage ou d'éclairement. On peut mettre une caméra à la place de la lumière, prendre le tampon de profondeur et l'utiliser pour faire un éclairement des sommets visibles et un ombrage des autres (cf `shadowmap`).