

Introduction à OpenGL



Johan Montagnat
I3S, CNRS
johan@i3s.unice.fr

Plan du cours

Partie I: Rudiments d'OpenGL

1. OGL : Open Graphic Language
2. références
3. Pipeline OpenGL
4. Conventions de l'API
5. Matrices homogènes
6. Buffers OpenGL
7. Primitives géométriques
8. Couleurs et matériaux
9. Sources lumineuses
10. Interaction entre OpenGL et X windows

1. OGL : Open Graphic Language

- Qu'est-ce qu'OpenGL ?
- Références
- La philosophie GL
- Organisation des librairies
- Possibilités et limitations

Qu'est-ce qu'OpenGL

- Histoire - OpenGL est un standard récent
 - 1989: GL (Graphic Language) a été développé par Silicon Graphics pour utiliser le hardware spécialisé de ses stations graphiques
 - 1993: OpenGL est une extension de GL portable sur d'autres architectures
 - La version actuelle est 1.2
- OpenGL est devenu un standard de fait en infographie 3D
 - Existe sur toutes les architectures
 - En langage C
 - Indépendant du système graphique
 - Performant et souple d'utilisation
- Mesa3D est une implémentation logicielle (libre) d'OpenGL
 - Sans hardware spécialisé : dépendant du processeur de la machine
 - Néanmoins bien optimisé
 - Indépendant du hardware : a contribué à la généralisation d'OpenGL

Qu'est-ce qu'OpenGL

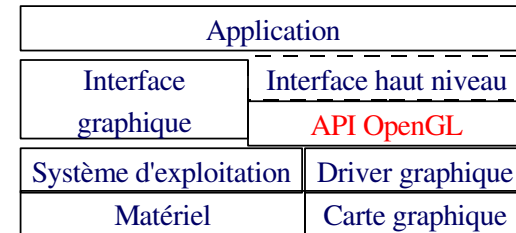
- Que trouve-t-on dans OpenGL ?

- Une API (bas niveau) en C permettant de tirer l'interface entre le logiciel et le driver matériel
- Spécialisé pour le rendu réaliste 2D et 3D
- Des matrices homogènes pour la projection de coordonnées 3D
- Des primitives graphiques (points, lignes, triangles...)
- Des primitives image
- Des buffers (image, Z-buffer, ...)
- Des matériaux
- Des lumières
- Des algorithmes de rendu simples à accélérer en hardware (faces cachées, lissage de Gouraud, placage de texture)
- Des fonctions d'optimisation
- Des bibliothèques plus haut niveau simplifiant l'utilisation des primitives

Philosophie GL

- API: Application Programmer's Interface

- 150 fonctions
- Librairie logicielle
- Interface bas niveau entre l'application et le hardware
- Pas de structures de données: OpenGL fonctionne comme une machine à états.



Philosophie GL

- Notion de contexte

- À chaque fenêtre graphique 3D est associé un contexte OpenGL
- Un seul contexte actif à un instant donné
- Toutes les commandes GL ont lieu dans le contexte actif

- Architecture en pipeline

- Deux types de primitives:
 - Géométriques (points, lignes...)
 - Pixels (images, textures...)
- Traitement des primitives en série (pipeline) jusqu'à l'étape de rasterization (discretisation et écriture dans le buffer d'affichage)

- Superposition de buffers comme autant de couches transparentes

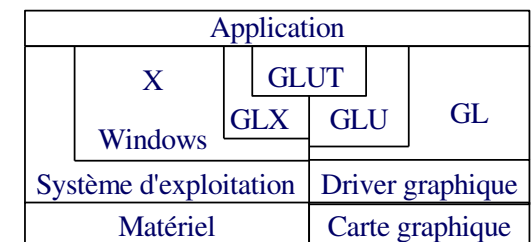
- Le buffer d'affichage (framebuffer), généralement sur 24 ou 32 bits
- Mais aussi le Z-buffer, stencil-buffer, double buffer...
 - Autant de bits par pixel!

Organisation de bibliothèques

- Les principales bibliothèques

- GL: API de base
- GLU (GL Utility): API haut niveau (primitive graphiques complexes, position de la caméra, projections, ...)
- GLX: interface avec X windows (sous UNIX)
- GLUT (GL Utility Toolkit): API haut niveau (initialisation, boucle d'événements X windows, ...)

- Organisation



Possibilités

- Les fonctionnalités de base
 - Transformations géométriques de primitives et pixels
 - Calculs de l'éclairage fonction des lumières et matériaux
 - Gestion des faces cachées
 - Rendu surfacique par facette ou de Phong
 - Anti-crénelage
 - Transparence
 - Plans de coupe
 - Placage de textures...
- Les fonctionnalités dérivées
 - Reflets
 - Ombres
 - Rendu volumique
 - Rendus mixtes surfaciques / volumiques...

Limitations

- Certains algorithmes se prêtent mal à un câblage en hardware
 - le lancé de rayons
 - la radiosité
 - ...
- OpenGL apporte un compromis en performance et puissance du rendu
- La performance effective dépend de la qualité de la carte graphique
 - Beaucoup de cartes sur PC sont optimisées pour le jeux (placage de texture intensif) mais négligent les aspects géométriques (reportés sur le processeur).
 - Ordre de grandeur: SGI Onyx2 Infinite Reality ~ 10^6 triangles rendus / s
 - Les cartes sur PCs se rapprochent rapidement de ces performances...

10. Références

Références électroniques

- Le site officiel
 - <http://www.opengl.org>
- Mesa3D
 - <http://www.mesa3d.org>
- SGI
 - <http://www.sgi.com/software/opengl/>
 - <http://www.sgi.com/software/opengl/examples/>
 - <http://www.sgi.com/software/opengl/advanced97/notes/notes.html>
- Tutorial et exemples
 - <http://nehe.gamedev.net/>

Livres de référence

- Le « livre rouge »
The OpenGL programming guide, 3rd edition, the official guide to learning OpenGL version 1.2
By Mason Woo et al., ISBN 0-201-60458-2
- Le « livre bleu »
OpenGL Reference Manual, 3rd edition
ISBN 0-201-65675-1
- The OpenGL Graphic System: A specification
Marc Segal, Kurt Akeley
- The OpenGL Graphics System Utility Library
Norman Chien et al.
<http://www.opengl.org/developers/documentation/specs.html>

3. Conventions de l'API OpenGL

- Conventions de nommage
- Changer l'état courant
- Paradigme `glBegin()` / `glEnd()`

Conventions de nommage

- Les types
Il n'y a ni types, ni structures définies en OpenGL. Les seuls types manipulés sont les types primitifs du C ou des tableaux de type primitifs
- Les constantes
`GL_...` suivi de mots en majuscules séparés par des `_`
`GL_PROJECTION`, `GL_TRIANGLE_FAN`
- Les fonctions de l'API
`gl...` suivi de mots en minuscule dont la première lettre est majuscule:
`glBegin(...)`, `glGetError(...)`, etc.
Les primitives ont 2, 3 ou 4 dimensions mais L'API en C ne permet pas la surcharge des noms de fonction
`glVertex{234}{ifd}{v}(...)`
└─ integer, float, double, vector (tableau)
└─ 2, 3 ou 4 paramètres
ex: `glVertex2i(int, int)`, `glVertex4fv(float *)...`

Changer l'état courant

- OpenGL fonctionne comme une machine à états
Les commandes ne sont souvent pas suffisantes par elle même, elles s'exécutent dans un *contexte*.
Ex: à un sommet est attaché une couleur courant, une normale courante, etc.
Le changement d'état se fait à travers de très nombreuses fonctions de l'API:
`gl{Enable,Disable}({GL_LIGHTING, GL_FOG, GL_DEPTH_TEST...})`
`glColor{34}{ifd}(...)`
`glNormal{34}{fd}(...)`
- Il est possible d'interroger l'état courant avec `glGet/isEnabled`:
`glGet{Boolean,Integer,Float,Double}v(enum flag, T *)`
`float matrix[16];`
`glGetFloatv(GL_MODELVIEW_MATRIX, matrix);`
`isEnabled(enum flag)`
`isEnabled(GL_LIGHTING);`

Paradigme glBegin() / glEnd()

- Dessin des object géométrique en regroupant les primitives graphique dans une "paire" glBegin/glEnd

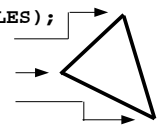
- Synthaxe:

```
glBegin(GL_TRIANGLES)
... glVertex ...
glEnd()
```

Pas d'entrelacement des paires glBegin/glEnd!

- Utilisation:

```
glBegin(GL_TRIANGLES);
glVertex3d(...);
glVertex3d(...);
glVertex3d(...);
glEnd();
```

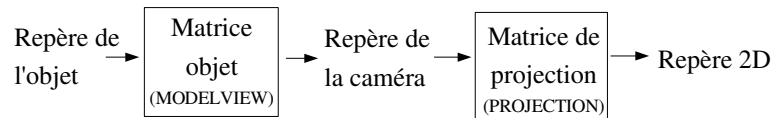


4. Matrices Homogènes

- Piles de matrices
- Représentation des matrices
- Calcul matriciel
- Matrices de projection
- Étapes de transformation des primitives

Piles de matrices

- Toutes les coordonnées (sommet, normales, directions...) sont multipliées par des matrices de transformation et de projection.
- OpenGL définit la chaîne de transformations suivante:



- OpenGL utilise deux types de matrices

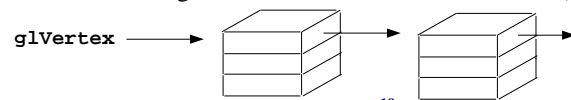
GL_MODELVIEW: associée à chaque objet

GL_PROJECTION: paramétrant la projection 3D vers 2D

- Les matrices sont stockées sur des piles

Toutes les primitives géométriques sont multipliées par la tête de pile

Possibilité de changer de sauveur/restaurer des matrices (gl{Push,Pop}Matrix)



Piles de matrices

- Matrices représentées en rangées dominantes

Attention, ce n'est pas la représentation usuelle mais sa transposée!

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

- Écriture d'une matrice sur une pile

glMatrixMode(GL_{PROJECTION,MODELVIEW}) permet de choisir la pile concernée

glLoadIdentity() charge la matrice identité en tête de pile

glLoadMatrix{fd}v(...) permet de spécifier une nouvelle matrice de tête de pile (16 composantes)

- Lecture de la valeur d'une matrice

glGet{Float,Double}v(GL_{PROJECTION,MODELVIEW}, ...)

Pour obtenir une matrice en lignes dominantes, utiliser

glGet{Float,Double}v(GL_TRANSPOSE_{PROJECTION,MODELVIEW}_MATRIX, ...)

Calcul matriciel

- Matrices homogènes (4x4):

Rotation	Translation
Echelle	
(3x3)	
projection 1	

- Translations: `glTranslate`

```
glTranslate{fd}(T x, T y, T z);
```

1	0	0	x
0	1	0	y
0	0	1	z
0	0	0	1

- Facteur d'échelle: `glScale`

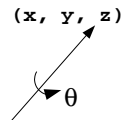
```
glScale{fd}(T x, T y, T z);
```

x	0	0	0
0	y	0	0
0	0	z	0
0	0	0	1

- Rotation: `glRotate`

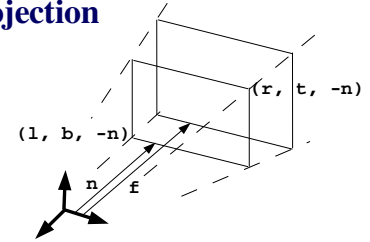
```
glRotate{fd}(T theta, T x, T y, T z);
```

	0
R	0
	0
0	0
0	0
0	1



- Multiplication à droite: `glMultMatrix{fd}(...)`

Matrices de projection



- Projection orthogonale

```
glOrtho(double l, double r, double b, double t,
double n, double f)
```

$\frac{2}{r-l}$	0	0	$-\frac{r+l}{r-l}$
0	$\frac{2}{t-b}$	0	$-\frac{t+b}{t-b}$
0	0	$-\frac{2}{f-n}$	$-\frac{f+n}{f-n}$
0	0	0	1

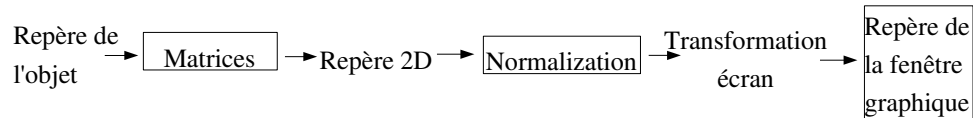
- Projection perspective

```
glFrustum(double l, double r, double b, double t,
double n, double f)
```

$\frac{2n}{r-l}$	0	$\frac{r+l}{r-l}$	0
0	$\frac{2n}{t-b}$	$\frac{t+b}{t-b}$	0
0	0	$-\frac{f+n}{f-n}$	$-\frac{2fn}{f-n}$
0	0	-1	0

Le rapport n/f influence l'aspect projectif

Étapes de transformation des primitives



- Normalization

division par la composante d'homogénéité

- Contrôle de la transformée écran

dimensions $w \times h$, position (x, y)

contrôle des bornes de profondeur (n, f) (pour optimiser la précision du Zbuffer)

```
glDepthRange(float n, float f)
```

```
glViewport(int x, int y, int w, int h)
```

5. Buffers OpenGL

- Les différents buffers
- Écriture dans les buffers
- Utilisation des buffers
- Un premier programme fonctionnel

Les différents buffers

- Il existe de nombreux buffers utilisés pendant l'étape de rendu. Le *frame buffer* se décompose en:

Color buffer: couleur (R,G,B,A) des pixels

se divise en double buffer et buffers droit et gauche pour la stéréo

Depth ou Z-buffer: profondeur de chaque pixel

Accumulation: composition d'opérations (+,*...) sur chaque pixel

Stencil: superposition d'un dessin sur le résultat du rendu

- Les buffers ne sont pas utilisés que pour le rendu mais aussi pour la sélection:

```
glRenderMode(GL_{RENDER,SELECTION,FEEDBACK})
```

GL_RENDER: rendu des primitives

glSelectBuffer + glClipPlane + GL_SELECTION: identification des primitives rendues en un point de la fenêtre.

GL_FEEDBACK: informations sur les primitives qui seraient rendues en un point de la fenêtre.

Écriture dans les buffers

- Il existe plusieurs buffers de couleur: deux pour le double buffer (un buffer d'affichage et un buffer de travail) chacun dédoublé en cas de mode stéréo (un pour l'oeil droit, un pour l'oeil gauche).

glBegin(GL_{FRONT,BACK,FRONT_LEFT...}) permet de sélectionner le buffer courant

glXSwapBuffers() échange les buffers d'affichage et de travail

- Manipulation des buffers:

glClear(GL...): réinitialisation d'un buffer

glClearColor(float r, float g, float b, float a): couleur de réinitialisation

glLogicOp(...): composition des valeurs du buffer de rendu et des couleurs des primitives qui se superposent

glDrawPixels(...): dessiner une région d'un buffer

glReadPixels(...): lire une région d'un buffer

Un premier programme fonctionnel

```
// vide le color buffer
```

```
glClearColor(0.0, 0.0, 0.0, 0.0);
```

```
glClear(GL_COLOR_BUFFER_BIT);
```

```
// applique une matrice de projection
```

```
glMatrixMode(GL_PROJECTION);
```

```
glPushMatrix();
```

```
glOrtho(...);
```

```
glMatrixMode(GL_MODELVIEW);
```

```
// ... réalise le rendu des objets
```

```
// restore l'ancienne matrice de projection
```

```
glMatrixMode(GL_PROJECTION);
```

```
glPopMatrix();
```

6. Primitives géométriques

- Points, lignes, triangles et quadrilatères
- Normales
- Polygones
- Rendu des primitives

Points, lignes, triangles et quadrilatères

- Toutes les primitives géométriques sont construites par une succession de sommets (`glVertex`). Le paramètre de `glBegin` indique le type de primitive construite.

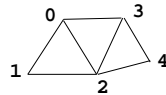
```
glBegin(GL_POINTS, LINE{S,_STRIP,_LOOP}, TRIANGLE{S,_STRIP,_FAN},
        QUAD{S,_STRIP}, POLYGON)
... glVertex, glNormal, glColor, glTexCoord ...
glEnd()
```

- Primitives: Simples

GL_POINTS
GL_LINES
GL_TRIANGLES
GL_QUADS
GL_POLYGON

Strips

GL_LINE_STRIP
GL_TRIANGLE_STRIP
GL_QUAD_STRIP



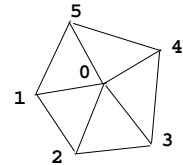
Loop

GL_LINE_LOOP



Fan

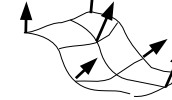
GL_TRIANGLE_FAN



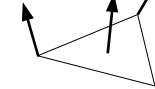
- Tous les sommets ont un certain nombre de "paramètres courants": couleur, normale, coordonnées de texture...

Normales

- Pour le rendu des surfaces, il est nécessaire de connaître leur orientation. En OpenGL, une direction normale est associée à chaque sommet.



- La normale est propre aux sommets et pas aux polygones



- La normale à un sommet est un vecteur normalisé. Il doit être indiquée avant le sommet lui-même:

```
glNormal
glVertex
glNormal
glVertex
...
```

Rendu des primitives

- Rendu "fil de fer" ou "faces pleines:

fil de fer: `glDisable(GL_LIGHTING)`
`glBegin(GL_POINTS, LINES, ...)`

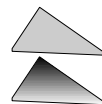
faces pleines: `glEnable(GL_LIGHTING)`
`glBegin(GL_TRIANGLES, POLYGON, ...)`

- Rendu plat ou lissé:

contrôle par `glShadeModel(GL_FLAT, SMOOTH)`

`GL_FLAT`: éclairage constant par face

`GL_SMOOTH`: rendu Phong (extrapolation de l'orientation des normales)



- Une ou deux faces?

Dans de nombreuses applications, il n'est pas nécessaire de faire un rendu de la face interne des objets (objets convexes...). Ceci permet de réduire le temps du rendu:

```
gl{Enable,Disable}(GL_CULL_FACE);
glCullFace(GL_FRONT, BACK, FRONT_AND_BACK);
```

7. Couleurs et matériaux

- Couleurs
- Matériaux
- Éclairer une scène

Couleurs

- Les couleurs sont définies comme des quadruplets (R,G,B,A)
chaque composante réelle entre 0 et 1 (pour les calculs d'éclairage)
discrétisation de chaque composante sur 8 bits à l'affichage (cartes graphiques
truecolor 24 bits = 3 * 8 bits)
la composante Alpha peut être utilisée comme une constante d'opacité
- La couleur est un état de la machine OpenGL
`glColor3f(red, green, blue)` ou `glColor4f(red, green, blue, alpha)`
Toutes les primitives affichées par la suite prennent la couleur fixée
- La couleur est composée en fonction des calculs d'éclairage et de transparence
- `glColor` permet de colorer les primitives points ou lignes

Matériaux

- L'interaction réaliste d'une source lumineuse et d'un matériau fait intervenir plusieurs paramètres de couleur
- La couleur des primitives de type triangle, quadrilatère ou polygone est définie par la fonction `glMaterial`
- Une surface présente deux faces qui peuvent correspondre à deux matériaux différents

`GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK`

- Définition des composantes d'un matériau

`glMaterialfv(GL_FRONT, GL_SPECULAR, tab_4_float)` composante spéculaire

`glMaterialfv(GL_FRONT, GL_DIFFUSE, tab_4_float)` composante diffuse

`glMaterialfv(GL_FRONT, GL_AMBIENT, tab_4_float)` composante ambiante

`glMaterialfv(GL_FRONT, GL_EMISSION, tab_4_float)` composante émissive

`glMaterialf(GL_FRONT, GL_SHININESS, float)` coefficient de brillance dans $[0, \infty[$

Lien entre Couleur et Matériaux

- Pour simplifier la définition d'un matériau
`glColorMaterial(GL_{FRONT, BACK, FRONT_AND_BACK},
GL_{AMBIENT, DIFFUSE, EMISSION, SPECULAR, AMBIENT_AND_DIFFUSE})`
La composante désignée du matériau prend la valeur courante de couleur
- Ne permet pas de rendre des surfaces de tous types (plastique brillant, bois mat, verre translucide...)
Pas de réflexions
Pas de transparence
La définition d'un matériau en termes de composante ambiante, diffuse, émissive et spéculaire est un modèle pour représenter une réalité plus complexe.

8. Sources lumineuses

- Sources lumineuses
- Couleur des sources lumineuses
- Atténuation de la lumière
- Spots

Sources lumineuses

- Activer les calculs d'éclairage
`glEnable(GL_LIGHTING)`
- Spécifier une source lumineuse
8 sources prédéfinies: `GL_LIGHT{0,7}`
activation individuelle des sources: `glEnable(GL_LIGHT0)`
- Paramètres des sources
`glLightfv(GL_LIGHT0, enum param, GLfloat param)`
- La position des sources lumineuses est transformée par la matrice `GL_MODELVIEW`
- Modèle d'éclairage
`glLightModel{if}[v](enum p, T value)`
`p=LIGHT_MODEL_AMBIENT, value = couleur (4 float) de la lumière résiduelle`
`p=LIGHT_MODEL_LOCAL_VIEWER, LIGHT_MODEL_TWO_SIDE,`
`LIGHT_MODEL_COLOR_CONTROL, value = booléen`

Couleur des sources lumineuses

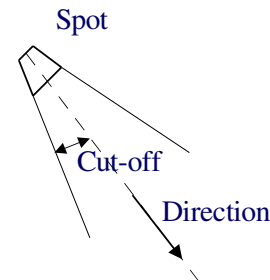
- Couleur définie par composante
`glLightfv(enum p, T value)`
composante diffuse `param = GL_DIFFUSE`
composante spéculaire `param = GL_SPECULAR`
composante ambiante `param = GL_AMBIENT`
`value = 4 float`
- La couleur d'une source se compose avec celle du matériau
- Après calcul d'éclairement, les valeurs de couleur sont tronquées au besoin dans l'intervalle [0, 1] (saturation de l'intensité), puis arrondie sur l'intervalle entier [0, 255] (discrétisation).
- Les couleurs sont estimées aux sommets, puis interpolées sur les polygones en cas de rendu de Phong.

Atténuation de la lumière

- Différentes lois d'atténuation de l'intensité
`glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, value)` atténuation indépendante de la distance à l'objet (lumière directionnelle à l'infini)
`glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, value)` atténuation linéaire en la distance à l'objet (lumière ponctuelle)
`glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, value)` atténuation quadratique en la distance à l'objet (lumière ponctuelle)
- Position de la lumière soumise à la matrice de transformation
`glLightfv(GL_LIGHT0, GL_POSITION, value)`
`value = 4 float`

Spots

- En plus des paramètres précédents...
- Une direction
`glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, value)`
`value = 3 float`
soumise aux transformations géométriques
- Un angle d'ouverture
`glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, value)`
en degrés
- Un coefficient d'atténuation exponentielle à l'intérieur du cône
`glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, value)`
La valeur par défaut est 0 (pas d'atténuation = bords nets du cône de lumière).



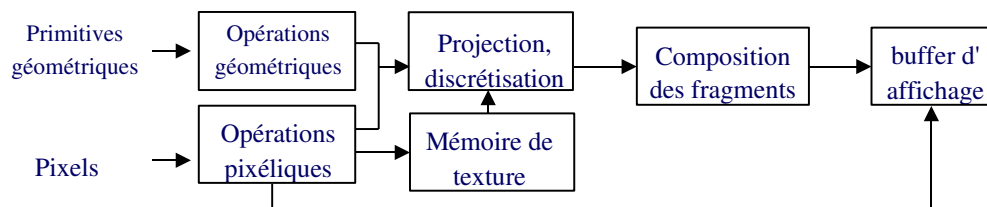
9. Pipeline OpenGL

- Primitives géométriques et pixeliques
- Pipeline OpenGL
- Étapes de traitement des primitives

Primitives géométriques et pixeliques

- Primitives géométriques
 - Décrites par des ensemble de sommets (Vertex)
 - Points
 - Segments de droites
 - Polygônes
 - Les courbes et surfaces doivent être approximées par ces primitives
- Primitives pixeliques
 - Bitmaps (8, 16, 24, 32 bits...)
 - Textures
- Autres Primitives
 - Couleurs
 - Normales
 - ...

Pipeline OpenGL



glFlush indique que toutes les commandes envoyées dans le pipeline doivent se terminer en temps fini.

glFinish force la terminaison de toutes les commandes envoyées dans le pipeline et ne retourne qu'ensuite.

2. Interaction entre OpenGL et X windows

- Interfaces graphiques: programmation par événements
- Rappels (?) sur X windows
- Création d'une fenêtre X
- Création d'un visual pour OpenGL
- Création d'un contexte OpenGL
- Exemple de programme

Interfaces graphiques: programmation par événements

- Tout programme graphique est basé sur une architecture événementielle

```
#include <WhateverIsNeeded.h>
void main() {
    CreateWindowsAndOtherGraphicWidgets();
    While(true) {
        event = WaitForAnEvent();
        ProcessEvent(event);
    }
}
```

- **ProcessEvent** fait appel à des fonctions (callbacks) de gestions des événements: la programmation n'est plus linéaire
- Java cache la boucle d'événements (elle est non apparente dans **main**) en l'exécutant dans un nouveau thread
- Le paradigme objet est bien adapté à la gestion événementielle et la définition de callbacks.

Rappels (?) sur X windows

- X windows est une API très bas niveau pour la gestion de fenêtres graphiques
 - Principaux fichiers d'entête: **X.h**, **Xutils.h**
 - Principale librairie: **libX11**
- Un serveur X windows tourne sur toute machine permettant l'exécution de requêtes et l'envoi de messages à distance

Ouverture d'une connexion avec un serveur X (un display):

```
Display *dpy = XOpenDisplay();
```

Gestion des événements:

XNextEvent(): attend un événement et retour une structure **XEvent**

XEvent.type: définit le type d'événement reçu

Xevent est un en-tête de structure commune à tous les types d'événements et se décline en **XExposeEvent**, **XKeyEvent**, **XMotionEvent**...

Les actions utilisateur (clavier, souris...) génèrent des événements

Création d'une fenêtre X

- Création d'une nouvelle fenêtre X en deux étapes:

XCreateWindow: création de la fenêtre (retourne l'identifiant X)

XMapWindow: affichage de la fenêtre à l'écran (provoque un l'envoi d'un événement de type **XExposeEvent** au serveur X)

- Les paramètres de la création

```
XCreateWindow(Display *dpy, Window parent, int x, int y, int width,
               int height, int border, int depth, int class, Visual *visual, long
               mask, XSetWindowAttributes attributes);
```

depth : nombre de bits par pixel

visual : capacités de la fenêtre

Les valeurs possibles pour ces paramètres sont dépendantes de la carte graphique et conditionnent les capacités de rendu.

- Les capacités du serveur X sont paramétrables mais limitées par la carte graphique pilotée

Création d'un visual adapté à OpenGL

- Interface entre X et OpenGL : GLX

Toutes les fonctions (définies dans **glx.h**) sont préfixées par **glX**

- Disponibilité de l'extension GLX du serveur X

```
Bool glXQueryExtension(Display *dpy, int *errcode1, int *errcode2)
```

- Détermination d'un visual adapté

```
XvisualInfo *glXChooseVisual(Display *, Screen, int *parameters)
```

parameters est un tableau de paramètres (terminé par None) qui indique les capacités demandées parmi lesquels: **GLX_USE_GL** (visual qui peut être rendu avec GL), **GLX_RGBA** (visual truecolor), **GLX_DOUBLEBUFFER** (capacité pour un double buffer), **GLX_DEPTH_SIZE** (taille du Z-buffer), etc.

man glXChooseVisual!

Plusieurs tentatives peuvent être nécessaire pour obtenir un visual supporté par la carte graphique (si les paramètres transmis ne sont pas supportés, la fonction retourne 0).

Création d'un contexte OpenGL

- Création d'un contexte OpenGL avec un visual valide
`GLXContext glXCreateContext(Display *, XVisualInfo *, None, GL_TRUE)`
Le contexte OpenGL n'est a priori pas attaché à une fenêtre particulière
Il contrôle simplement à quel buffer doivent être envoyées les instructions GL
- Activation d'un contexte particulier
`glXMakeCurrent(Display *dpy, GLXDrawable window, GLXContext ctx)`
`window` est un identificateur X de fenêtre qui crée le lien entre une fenêtre donnée et les instruction OpenGL produites
- Désactivation de tout contexte
`glXMakeCurrent(dpy, None, NULL)`
- Destruction d'un contexte
`glXDestroyContext(Display *, GLXContext)`
- Affichage final du rendu OpenGL dans la fenêtre (en cas de double buffering)
`glXSwapBuffers()`

Un code typique

```
// Ouverture de la connexion avec le serveur
Display *dpy = XOpenDisplay(":0"); if(!dpy) error(...);
if(!glXQueryExtension(dpy, ...)) error(...);

// Création du contexte GL
XVisualInfo *vinfo = glXChooseVisual(dpy, ...); if(!vinfo) error(...);
GLXContext context = glXCreateContext(dpy, vinfo, ...);

// creation de la fenêtre X
Window win = XCreateWindow(dpy, ..., vinfo->depth, ..., vinfo->visual, ...);
XMapWindow(dpy, win);

// boucle d'événements
do {
    Xevent evt; XNextEvent(dpy, &evt);
    if(evt.type == Expose) {
        glXMakeCurrent(dpy, win, context);
        // instructions X et GL
        glXSwapBuffers(dpy, win);
    }
    ...
} while(true);
```

Une vue plus haut niveau avec GLUT

```
#include <GL/glut.h>

int main(int argc, char **argv) {
    // Ouverture de la connexion avec le serveur
    glutInit(&argc, argv);
    // Obtention d'un contexte
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    // Creation d'une fenêtre
    glutCreateWindow("title");
    // Assignment de la fonction de réaffichage
    glutDisplayFunc(display);
    // Boucle de gestion des événements
    glutMainLoop();
}

// callback de réaffichage
void display() {
    // instructions GL...
}
```

Plan du cours

Partie II: Aperçu des fonctionnalités

1. La bibliothèque GLU
2. Tableaux de pixels
3. Placage de textures
4. Rendu avancé
5. Optimisation

1. La bibliothèque GLU

- Présentations
- Projections
- Position de la caméra
- Quadriques
- NURBS (Non-Uniform B-Splines)

Présentation

- GLU: GL Utility Library
 - Partie intégrante d'OpenGL
 - Fonctionnalités plus haut niveau dans tous les domaines (primitives géométriques, textures, matrices, ...)
- Uniquement construite sur la librairie GL
- Fonctionnalités
 - Manipulations matricielles (→ repose sur `glMultMatrix()`)
 - Positionnement de la camera (→ `glMultMatrix()`)
 - Quadriques (→ primitives géométriques)
 - NURBS (→ primitives géométriques)
 - Simplification de polygones

Projections

- Chaque méthode crée une matrice puis appelle `glMultMatrix`

`gluProject()` / `gluUnproject()`

Projection des coordonnées de l'espace objet à l'espace écran et réciproquement

Paramètres = coordonnées + tous les paramètres de transformation (matrices, viewport)

Les paramètres de la transformation finale écran (viewport), établis par appel à `glViewport()`, peuvent être relus par `glGetDoublev(GL_VIEWPORT, tableau_4_double)`.

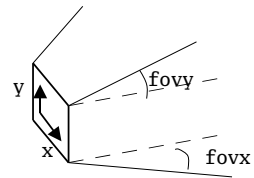
- Projection orthogonale 2D

`gluOrtho2D(double left, double right, double bottom, double top)`
`= gluOrtho(left, right, bottom, top, -1, 1)`

Projections

- Projection perspective 3D: spécification intuitive
 - `gluPerspective(double fovy, double aspect, double near, double far)`
 - fovy** = angle de vue en y (en degrés)
 - aspect** = rapport de l'angle de vue en x sur l'angle en y (généralement 1)
 - near** = distance au plan de coupe proche
 - far** = distance au plan de coupe éloigné
 - Par simple appel à `glFrustum()`:

```
ymax = - ymin = near * tan(fovy * PI / 360)
xmax = ymax * aspect
xmin = ymin * aspect
glFrustum(xmin, ymin, xmax, ymax, near, far)
```



Position de la camera

- Position de la caméra

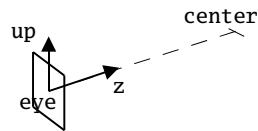
```
gluLookAt(double eyeX, double eyeY, double eyeZ,
          double centerX, double centerY, double centerZ,
          double upX, double upY, double upZ)
```

$z = (\text{center} - \text{eye}) / \|\text{center} - \text{eye}\|$

(x, up, z) repère orthonormé

M = [x1 y1 z1 0] (changement de repère) + translation -eye

```
[x2 y2 z2 0]
[x3 y3 z3 0]
[0 0 0 1]
```



Sélection

- Sélection à la souris

gluPickMatrix crée une matrice de rendu dans une petite région autour d'un point particulier

→ utilisation: faire un rendu de taille réduite autour du curseur

réaliser le rendu en mode sélection activé: tous les objets rendus sont listés dans un tableau de sélection défini par l'utilisateur

= dériver le hardware pour accélérer un calcul géométrique

l'un des (multiples!) détournement des fonctionnalités OpenGL

- Utilisation

glSelectBuffer(...)	allouer un tableau de sélection
glRenderMode(GL_SELECT)	passer en mode sélection
gluPickMatrix(...)	créer la matrice de rendu locale, autour du pointeur souris
glInitNames()	vider la pile de labels
...glPushName / glPopName...	primitives OpenGL + labels associés
...	lire les valeurs écrites dans le tableau de sélection
glRenderMode(GL_RENDER)	repasser en rendu standard

Quadriques

- Primitives géométriques de haut niveau

Quadriques = sphère, cylindres et disques

Structure commune `GLUquadricObj *quad = gluNewQuadric();`

- Paramètres de rendu

gluQuadricDrawStyle: GLU_POINT, GLU_LINE, GLU_FILL, GLU_SILHOUETTE

primitives utilisées pour le rendu

gluQuadricNormals: GLU_NONE, GLU_FLAT, GLU_SMOOTH

utilisation des normales: aucune, une par face, une par sommet

gluQuadricTexture: GL_TRUE, GL_FALSE

génération des coordonnées de texture

gluQuadricOrientation: GL_INSIDE, GL_OUTSIDE

direction des normales

Quadriques

- Exécution des primitives GL

Création de facettes quadrilatérales

Orientation par défaut (selon l'axe Y)

gluSphere(quad, radius, slices, stack): sphère (méridiens et parallèles)

gluCylinder(quad, r1, r2, height, slices, stack): cylindre conique

gluDisk(quad, r1, r2, slices, loops): disque dans le plan z=0

gluPartialDisk(quad, r1, r2, slices, loops, a1, a2): arc de disque

- Callback associé

gluQuadricCallback(): gestion des erreurs

- Destruction

gluDeleteQuadric(quad)

NURBS: Non-Uniform B-Splines

- NURBS = Courbes ou surfaces lisses

Les courbes et surfaces sont approximées par des lignes ou des polygones (cf sphères, cylindres et disques...)

Pour faciliter la définition de formes courbes plus complexes que les quadriques, GLU génère des B-Splines discrétisées par des primitives OpenGL.

- Implémentation OpenGL

Permet le rendu direct ou la génération d'une triangulation

- B-Splines

Une B-Spline ou une courbe de Bézier est une courbe/surface continue définie par un nombre fini de *points de contrôle*

La courbe/surface est définie comme la composition d'un ensemble de fonctions polynomiales de base

Les polynômes de base sont paramétrés par un ensemble fini de *noeuds* (suite croissante de réels).

Leur *degré* peut varier.

B-Splines (cas des surfaces)

- $n+d$ Noeuds en x et $m+d$ noeuds en y

$$0 < t_0 < t_1 < \dots < t_{n+d} \text{ en } x$$

$$0 < t_0 < t_1 < \dots < t_{m+d} \text{ en } y$$

- 2 ensembles de n et m Polynômes de degré d

Définition récursive:

A l'ordre 0: fonctions escaliers $B_i^0(x) = 1$ si $t_i \leq x \leq t_{i+1}$ ou 0 sinon

A l'ordre d : $B_i^d(x) = (x - t_i)/(t_{i+d} - t_i) B_{i-1}^{d-1}(x) + (t_{i+d+1} - x)/(t_{i+d+1} - t_{i+1}) B_{i+1}^{d-1}(x)$

- nm points de contrôle

$$P_{0,0}, P_{0,1}, \dots, P_{0,m-1}$$

...

$$P_{n-1,0}, P_{n-1,1}, \dots, P_{n-1,m-1}$$

- Point (x, y) de la surface B-Spline défini comme:

$$S(x, y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} P_{nm}^{x,y} B_{i,x}^d(x) B_{j,y}^d(y)$$

NURBS

- Comme pour les quadriques: objet NURBS

```
GLUnurbsObj *nurbs = gluNewNurbsRenderer();
```

- Callbacks (pour connaître la triangulation générée)

```
gluNurbsCallback(nurbs, GLU_NURBS_{BEGIN_EXT, VERTEX_EXT,
NORMAL_EXT, COLOR_EXT, TEXTURE_COORD_EXT, ...}, function);
```

permet un contrôle fin des NURBS et de connaître la position des points, normales, points de texture... générés par GLU

- Paramètres d'affichage

```
gluNurbsProperty(nurbs, GLenum property, GLfloat value)
```

`property = GLU_{SAMPLING_METHOD, DISPLAY_MODE, ...}` pour contrôler la discrétisation, l'affichage, ...

- Création d'une courbe ou d'une surface

```
gluNurbsCurve() / gluNurbsSurface()
```

- Destruction

```
gluDeleteNurbsRenderer(nurbs);
```

NURBS

- Création d'une courbe

```
gluBeginCurve(nurbs);
```

// autant de fois que désiré

```
gluNurbsCurve(nurbs, int #nœuds, float *nœuds, int offset, float
*ctlpoints, int order, GL_MAP1_VERTEX{3,4});
```

// où **#nœuds = n+d**, **order = d**, **nœuds** est un tableau de **n+d** points et

// **ctlpoints** est un tableau de **3*n** coordonnées

```
gluEndCurve(nurbs);
```

- Création d'une surface de manière similaire

```
gluBeginSurface(nurbs);
```

// autant de fois que désiré

```
gluNurbsCurve(nurbs, int #nœudsx, float *nœudsx, int #nœudsy, float
*nœudsy, int offset, float *ctlpoints, int orderx, int ordery
GL_MAP2_VERTEX{3,4});
```

```
gluEndSurface(nurbs);
```


2. Tableau de pixels

- Importation de tableau de pixels
- Transformations
- Ecriture d'un tableau de pixels

Importation de tableaux de pixels

- Tableaux de pixels utilisé pour la manipulation de textures et des buffer image
- `glPixelStore`, `glPixelTransfer` et `glPixelMap` contrôlent les paramètres d'importation des tableaux de pixel
- Les fonctions manipulant des tableaux de pixel (`glTexture[1-3]D`, `glDrawPixels...`) sont affectés par `glPixelStore`
- Importation de tableaux de pixels
`glPixelStore{if}(GL_UNPACK_{SWAP_BYTES, LSB_FIRST, ROW_LENGTH, SKIP_{ROWS, PIXELS, IMAGES}, ALIGNEMENT, IMAGE_HEIGHT}, value)`
SWAP_BYTES, LSB_FIRST: inverser les octets, little endian / big endian
ALIGNEMENT: nombre d'octets par pixels
SKIP_{ROWS, PIXELS, IMAGES}: offsets entre rangées/pixels/images si les données sont discontinues

Transformations

- Les tableaux de pixels manipulés peuvent être soumis à une série de transformations
voir pipeline de `glDrawPixels`, (GLSPEC p 96)
- Exemple: augmenter l'intensité du plan rouge
`glPixelTransferf(GL_RED_SCALE, 2.0);`
- Exemple: convolution du tableau avec un filtre
`glConvolutionFilter2D(GL_CONVOLUTION_2D, GLenum outputFormat, int width, int height, GLenum inputFormat, GLenum type, void *data)`
inputFormat = format du buffer d'entrée (e.g. `GL_RGB`)
outputFormat = format du tableau de pixels généré (e.g. `GL_RGBA`)
width, height = taille de l'image
type = type du buffer de pixels (e.g. `GL_UNSIGNED_BYTE`)
data = image composant le filtre de convolution
`glConvolutionParameter{if}v(GL_CONVOLUTION_2D, GL_CONVOLUTION_FILTER_{SCALE, BIAS}, 4_float_values)`: paramètres à appliquer au filtre de convolution

Ecriture d'un tableau de pixels

- Ecriture dans le buffer d'affichage
`glDrawPixels()`
- Ecriture dans la mémoire de texture
`glTexImage[1-3]D()`
- Copie d'une région du buffer d'affichage dans une autre
`glCopyPixels()`
- Lecture d'une région du buffer d'affichage
`glReadPixels()`
cf pipeline des transformations en lecture (GLSPEC p 181), inverse des transformations en écriture
- Lecture de tout un buffer
`glReadBuffer()`

3. Placage de textures

- Création de textures
- Coordonnées texture
- Textures planes
- Textures sphériques
- Textures environnementales

Création de textures

- Chargement d'un tableau de pixels dans la mémoire de texture
`glTexImage2D(GL_TEXTURE_2D, int level, int outputFormat, int width, int height, int border, enum inputFormat, enum type, void *data)`
`level = 0` Niveau de détail (pour textures hiérarchiques)
`outputFormat` = format interne (e.g. `GL_RGBA`)
`inputFormat` = format des données (e.g. `GL_RGB`)
`type` = type des composantes R,G,B,A (e.g. `GL_UNSIGNED_CHAR`)
`data` = buffer de données
`width, height` = Taille de l'image de texture (toujours puissance de 2)
`border = 0` épaisseur du bord
- OpenGL gère des texture 1D, 2D et 3D
`glTexImage[1-3]D`

Coordonnées texture

- OpenGL prévoit un espace de texture à 4 dimensions, normalisées entre 0 et 1, et désignées par `GL_T`, `GL_S`, `GL_R`, `GL_Q`
- Paramètres de texture: `glTexParameter`
`glTexParameteri(GL_TEXTURE_{1,2,3}D, GL_TEXTURE_WRAP_{S,T,R}, GL_{CLAMP, REPEAT})`: Troncature ou répétition au bord
`glTexParameteri(GL_TEXTURE_{1,2,3}D, GL_TEXTURE_{MIN,MAG}_FILTER, GL_{NEAREST,LINEAR,...})`: Filtre de sous- ou sur-échantillonnage
- Association des points de texture avec les sommets géométriques
`glTexCoord2f(float x, float y)`: x et y dans [0,1]
Alternance coordonnée sommet - coordonnée texture:
`glTexCoord2f(0.0, 0.0)`
`glVertex3f(0, 0, 0)`
`glTexCoordef(1.0, 0.0)`
`glVertex3f(100.0, 0, 0)`

Paramètres de texture

- Couleur de base de la texture
`glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, 4_floats)`
Par défaut : (0, 0, 0, 0)
- Mode de composition des pixels de la texture avec la couleur de base et le contenu du buffer d'affichage
`glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_{REPLACE, MODULATE, DECAL, BLEND})`
`GL_REPLACE` = ignorer le buffer d'affichage
`GL_MODULATE` = moduler avec la couleur de base
`GL_DECAL, GL_BLEND` = moduler avec la couleur de base et le buffer d'affichage

Placage de textures

- Différentes transformations des coordonnées texture:

```
glTexGen(GL_{S,T,R,Q}, GL_TEXTURE_GEN_MODE, GL_{OBJECT_LINEAR,  
EYE_LINEAR, SPHERE_MAP})
```

GL_OBJECT_LINEAR

(défaut)



GL_SPHERE_MAP



GL_EYE_LINEAR

- GL_OBJECT_LINEAR = défaut
interpolation linéaire dans l'espace de l'objet
- GL_EYE_LINEAR
interpolation linéaire dans l'espace de la caméra
- GL_SPHERE_MAP
placage d'une texture fish-eye sur l'objet (=reflet de l'environnement sur l'objet)

4. Rendu avancé

- Clipping
- Brouillard
- Transparence
- Anti-crénelage
- Rendu volumique
- Stéréo

Clipping

- Supprimer du pipeline de rendu toutes les primitives à l'extérieur du/des plan(s) de coupe
- Pour éviter les aberrations (projection d'objet derrière la caméra, etc), il existe toujours deux plans parallèles à la caméra (near et far) mis en place avec `glFrustum()`
- Activation de plans de coupe
`glEnable(GL_CLIP_PLANEi)`
`glClipPlane(GL_CLIP_PLANEi, 4_floats)` : équation de plan

Brouillard

- Modifier la couleur fonction de la distance pour donner une impression de profondeur (pour le rendu fil de fer ou les effets de brouillard)
`glEnable(GL_FOG)`
- Choisir pour couleur de brouillard celle du fond
`glFogfv(GL_FOG_COLOR, 4_floats)`
- Fonctions de transformation linéaire, **exp** ou **exp²**
`glFogi(GL_FOG_MODE, GL_{LINEAR, EXP, EXP2})`
- Coefficients d'atténuation (cf GLSPEC p 139)
`glFogf(GL_FOG_DENSITY, value)` en mode EXP ou EXP2
`glFogf(GL_FOG_{START,END}, value)` en mode LINEAR

Transparence

- Transparence réaliste

composer la couleurs de n primitives appartenant à des surfaces différentes et se superposant en un pixel de l'écran

composition = opacification avec l'ajout de couches transparentes

réalisme = tenir compte de la distance de chaque primitive...

- très coûteux en espace mémoire et en temps

- Compromis: utilisation de la composante Alpha de couleur pour simuler la transparence

Composer les couleurs (R,G,B,A) 2 à 2

Garder un seul plan des valeurs A pour chaque pixel de l'écran

ex: pixel (i,j) de couleur (R1,G1,B1,A1) à composer avec (R2,G2,B2,A2):

$$(R,G,B,A) = (A1*R1+A2*R2, A1*G1+A2*G2, A1*B1+A2*B2, A1*A2)$$

éclaircissement progressif par superposition

ou bien une autre forme de composition ?

Transparence en OpenGL

- Test Alpha: ignorer un fragment qui ne passe pas le test

```
glEnable(GL_ALPHA_TEST)
```

```
glAlphaFunc(GL_{NEVER, ALWAYS, LESS, LEQUAL, GEQUAL, GREATER, EQUAL, NOTEQUAL}, value)
```

spécifier le type de test et la valeur seuil de test

- Moduler la couleur en fonction du coefficient Alpha:

```
glEnable(GL_BLEND)
```

La couleur d'un fragment peut être composée avec une couleur de référence:

```
glBlendColor(float red, float green, float blue, float alpha)
```

et le contenu du buffer d'affichage:

```
glBlendEquation() et glBlendFunc()
```

Transparence en OpenGL

- glBlendEquation :

```
glBlendEquation(GL_FUNC_{ADD, SUBTRACT, MIN, MAX, REVERSE_SUBTRACT})
```

GL_FUNC_ADD: Couleur = src * CouleurFragment + dst * CouleurBuffer

...

- glBlendFunc :

```
glBlendFunc(enum src, enum dst)
```

```
src = GL_{ZERO, ONE, DST_COLOR, ONE_MINUS_DST_COLOR, CONSTANT_COLOR...}
```

```
dest = GL_{ZERO, ONE, SRC_COLOR, ONE_MINUS_SRC_COLOR, CONSTANT_COLOR...}
```

exemple de transparence:

```
glBlendEquation(GL_FUNC_ADD)
```

```
glBlendFunc(GL_DST_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

Anti-crênelage (antialiasing)

- Affectation d'une composante Alpha dans [0,1] proportionnelle à la surface de pixel couverte par chaque primitive

- Activation de l'anti-crênelage

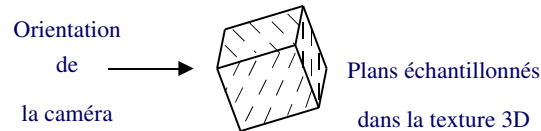
```
glEnable(GL_{POINT, LINE, POLYGON}_SMOOTH)
```

- Contrôle qualité / coût

```
glHint(GL_{POINT, LINE, POLYGON}_SMOOTH_HINT, GL_{FASTEST, NICEST, DONT_CARE})
```

Rendu volumique

- Utilisation d'une texture 3D représentant le volume
- Echantillonnage de plans perpendiculaires à la direction du regard dans la texture 3D



- Superposition transparente de plans
- Définition d'une fonction de transfert pour contrôler l'opacité de chaque pixel traversé

Stéréo

- Nécessite 2 buffers
 - Un buffer pour dessiner le point de vue de chaque oeil
 - Dans le cas de double buffering: 4 buffers
 - Allocation: dans `glXChooseVisual`, spécifier le paramètre `GLX_STEREO`
- Technique
 - Réaliser 2 rendus sous 2 points de vu légèrement décalés
 - Afficher alternativement chaque buffer de rendu
 - Utiliser des lunettes polarisées synchronisées sur la fréquence d'affichage de l'écran

- Sélection du buffer de rendu
`glDrawBuffer(GL_BACK_{LEFT,RIGHT})`

5. Optimisation

- Double buffers
- Listes précompilées
- Textures hiérarchisées
- Simplification de triangulations

Double buffers

- Eviter le scintillement en dessinant hors de l'écran puis en rafraîchissant la fenêtre « instantanément »
 - Initialisation: paramètre `GLX_DOUBLEBUFFER` de `glXChooseVisual()`
 - Forcer à rafraichir: `glXSwapBuffers()`
 - Forcer l'écriture dans l'un des buffers
`glDrawBuffer(GL_{BACK,FRONT})`
 - Normalement, on travaille toujours dans `GL_BACK`, `GL_FRONT` est affiché et `glXSwapBuffers()` provoque une inversion des deux

Listes précompilées

- Précalculer certaines étapes du pipeline et mémoriser le résultat dans une « liste d'affichage » qui pourra être exécutée à nouveau plus rapidement
- Précalcul certaines étapes du pipeline

```
int list = glGenLists(1); // obtenir un nouvel identifiant de liste
glNewList(list, GL_{COMPILE,COMPILE_AND_EXECUTE}); // débiter la liste
...gl...
glEndList(); // terminer la liste
```
- Réexécute les instructions gl en accélérant

```
glCallList(list);
```
- Exemple d'utilisation: dessiner 2 objets de couleurs et positions différentes

```
glColor4f(...);
int list = glGenLists(1); glNewList(list, GL_COMPILE_AND_EXECUTE);
... dessiner un cube...
glEndList();
glColor4f(...); glTranslated(...);
glCallList(list); glDeleteLists(list, 1);
```

Textures hiérarchisées

- Textures multi-résolution (MipMaps)
Affichage de textures adaptées à la distance de visualisation
- Chargement de texture hiérarchisées

```
glTexImage[1-3]D(..., int level, ...)
```
- Création:

```
gluScaleImage()
gluBuild[1-3]DMipMaps()
```

Simplification de triangulations

- Simplification = réduction d'une triangulation (on parle de décimation: 1 sur 10)
 - Conserver la topologie
 - Préserver au mieux la géométrie (concentrer les triangles dans les zones de forte courbure)
- Optimisation des performances
 - Valable pour les objets complexes
 - Réduire le nombre de primitives géométriques à afficher en minimisant la perte de qualité
 - La simplification peut être pré-calculée...
- Construction de modèles hiérarchiques
 - Utiliser des modèles de plus en plus grossier lorsque la distance augmente