

**Open GL**

# Table des matières

Introduction.....	5
Généralités.....	5
La librairie Glut.....	6
Installation de l'environnement.....	6
Installation de glut, open gl et code::block sous ubuntu.....	6
Installation de glut sous Windows.....	6
Compiler avec la glut .....	6
Hello World.....	6
Les principales fonctionnalités de la Glut.....	8
glutInit(int* argc, char** argv).....	8
Void glutInitDisplayMode(unsigned int mode).....	8
void glutInitWindowSize ( int width, int height).....	9
void glutInitWindowPosition ( int x, int y);.....	9
int glutCreateWindow ( char * name );.....	9
void glutMainLoop ( void ).....	9
void glutDisplayFunc ( void (*func) (void));.....	10
void glutIdleFunc ( void (*func) ( void ));.....	10
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha).....	11
void glClear(GLbitfield mask);.....	11
void glColor3f(GLfloat red, GLfloat green, GLfloat blue).....	11
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2).....	11
void glutSwapBuffers ( void );.....	11
void glutPostRedisplay ( void );.....	12
Évènements de la glut .....	12
L'affichage .....	12
Les frappes de touches clavier .....	12
Gestion de la souris .....	14
Événement Idle : pour le animations et les taches de fond.....	15
Dessiner des formes de base .....	16
2.1 Les primitives géométriques .....	16
2.2 Variantes de glVertex*() et représentations des sommets .....	18
2.3 Modes d'affichage de polygones .....	19
2.4 Représentation d'un maillage .....	21
2.4.1 Définition d'un maillage .....	21
2.4.2 Exemple .....	21
Le vecteur normal en un point.....	21
Le modèle d'ombrage.....	22
Elimination des surfaces cachées.....	22
L'architecture OpenGL.....	23
OpenGL est basé sur des Etats.....	23
Pipe-Line de Rendu simplifié.....	23
Modélisation hiérarchique.....	24
Description hiérarchique d'une scène.....	24
Construction.....	24
Pile de transformations.....	25
Manipulation des matrices de transformation.....	25
La transformation de modélisation.....	26
La vision.....	27
Principe de la vision.....	27
La transformation de vision.....	27

La transformation de modélisation .....	29
La transformation de projection.....	29
Projection en perspective.....	29
Projection orthographique.....	30
Clipping.....	30
La transformation de cadrage.....	30
Eclairage.....	31
Modèle d'éclairage.....	31
Lumière émise (Ne concerne que les objets).....	31
Lumière ambiante (Concerne les objets et les lampes).....	31
Lumière diffuse (Concerne les objets et les lampes).....	31
Lumière spéculaire (Concerne les objets et les lampes).....	31
Brillance (Ne concerne que les objets).....	31
Les lampes.....	31
Nombre de lampes.....	31
Couleur des lampes.....	32
Lampes directionnelles.....	32
Lampes positionnelles.....	32
Modèle d'éclairage.....	33
Atténuation de la lumière.....	33
Lampes omnidirectionnelles et spots.....	33
Couleur d'un matériau.....	34
Propriétés matérielles d'un objet.....	34
Combinaison des coefficients.....	34
Listes d'affichage.....	34
Listes d'affichage hiérarchiques.....	35
Création, suppression des listes d'affichage.....	35
Les Textures.....	36
Introduction.....	36
Mécanisme général.....	36
Coordonnées de texture.....	36
Répétition de la texture.....	37
Les Objets-Textures.....	37
Filtrage.....	37
Les niveaux de détail.....	38
Filtrage.....	38
Lecture d'une image de texture dans un fichier.....	38
Tableaux de sommets.....	39
Mécanisme général.....	39
Activer les tableaux.....	39
Spécifier les données de tableaux.....	39
Fonctions de spécification des tableaux.....	40
Exemple 1 : données dans différentes tables.....	40
Exemple 2 : données entrelacées.....	40
Dé-référencer le contenu des tableaux.....	41
Accès à un élément unique.....	41
Accès à une liste d'éléments.....	41
Accès à une séquence d'éléments.....	42
Mélange de couleurs : blending.....	43
Activer/désactiver le blending.....	43
Facteurs de blending source et destination.....	43
Définition.....	43

Spécifier les facteurs.....	43
Exemples.....	44
Mélange homogène de deux objets.....	44
Importance de l'ordre d'affichage.....	44
Du bon usage du tampon de profondeur.....	45
Lissage (antialiasing), fog et décalage de polygones.....	46
Définition et principe.....	46
Exemple.....	46
Contrôle de qualité.....	46
Fog, le brouillard.....	47
Mise en œuvre.....	47
Couleur et équation du fog.....	47
Décalage de polygone.....	48
Définition.....	48
Modes de rendu des polygones.....	48
Types de décalage.....	48
Valeur du décalage.....	49
Exemple.....	49
Les tampons d'image.....	50
Introduction.....	50
Définitions.....	50
Les tampons et leur utilisation.....	50
Tampons chromatiques.....	50
Sélectionner les tampons chromatiques en écriture et en lecture.....	51
Le tampon d'accumulation.....	51
Lisser une scène.....	51
Jittering.....	52
Fondu enchaîné.....	52
Profondeur de champ.....	52
Masquer les tampons.....	52
Vider les tampons.....	52
Sélection et picking.....	54
Sélection.....	54
Principales étapes.....	54
Détails de ces étapes.....	54
L'enregistrement des hits.....	55
Exemple de sélection.....	55
Utilisation de la pile des noms.....	56
Picking.....	56

# Introduction

## **Généralités**

OpenGL est un moteur de rendu graphique qui permet de visualiser des scènes 2D et 3D avec une accélération matérielle.

La librairie GL a été créée par Silicon Graphics en 1989, puis portée sur d'autres architectures à partir de 1993 (OpenGL)

OpenGL est une interface comprenant plus de 120 commandes, couvrant le spectre fonctionnelle nécessaire à la description et la manipulation des objets 3D.

OpenGL est disponible sur la plupart des architectures et des systèmes d'exploitation dont (Linux, Windows NT+, os2, Amiga..)

La librairie est une interface, plusieurs implémentations existent dont : SGI OpenGL, 3dfx OpenGL, Mesa, Microsoft OpenGL)

Il existe une version libre de droits d'OpenGL développée par Brian martin et disponible sur <http://www.mesa3d.org>.

OpenGL s'accompagne de bibliothèques supplémentaires (GLU, GLUT, GLX, OpenInventor) qui facilitent sa mise en oeuvre.

OpenGL est aujourd'hui géré par un consortium parmi lesquels (3DLabs, Nvidia, ATI, Sun, Apple, Intel, ...)

OpenGL est résolument orienté rendu temps réel

# La librairie Glut

## Installation de l'environnement

Dans ce cours nous utiliserons la librairie glut pour interfacer OpenGL avec le système d'exploitation. Glut permet de créer une fenêtre graphique et de gérer les événements tels que le click de souris ou le redimensionnement de la fenêtre par l'utilisateur.

## Installation de glut, open gl et code::block sous ubuntu

- Installation des dépendances
  - Installation de g++
    - `sudo apt-get install g++`
  - Installation de GLUT via apt-get
    - `sudo apt-get install freeglut3 freeglut3-dev`
  - Installation de la librairie : libXxf86vm-dev
    - `sudo apt-get install libXxf86vm-dev`
- Installation de Code::Block via l'interface Ubuntu.
- Créer un nouveau projet GLUT project, sélectionner le répertoire d'install de glut pour définir la variable \$(#glut) de code::block. (/usr)

## Installation de glut sous Windows

- Installation de la librairie Glut pour Windows
  - **Télécharger la dernière version sur Glut sur <http://www.xmission.com/~nate/glut.html>**
- Copier les fichiers dans les répertoires suivants
  - glut32.dll dans c:\windows\system,
  - glut32.lib dans c:\program files\mingw\lib,
  - glut.h dans c:\program files\mingw\include\GL.
- Ajouter au début du programme l'include suivante:
  - `#include <windows.h>`

## Compiler avec la glut

Pour pouvoir compiler un programme C ou C ++ avec la glut, il faut inclure la bibliothèque glut.h :

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

Pour compiler, il faut utiliser la librairie lglut. Sous linux ou unix avec le compilateur g++, la ligne de commande est :

```
$ g++ programme.c -lglut -o programme.out
```

## Hello World

Pour tester l'installation de l'environnement, nous allons mettre en œuvre le traditionnel "Hello World".

L'objectif est d'obtenir un résultat en un minimum de lignes.

- Inclure le minimum de librairie pour compiler, exécuter et visualiser le programme.
  - Faire appel aux minimum de fonction pour produire un résultat.
  - Disposer d'un point d'entrée et d'un point de sortie dans le framework
  - Réaliser un cas d'utilisation simple et utile (affichage)
- 
- Créer sous code block un projet Glut, s'il vous demande de sélectionner à un répertoire pour définir la variable d'environnement Glut, choisissez votre répertoire d'installation de minGw.

```
1  /*****
2  * OpenGL/Glut HelloWorld
3  * Michaël THOMAS @ Natysphère
4  * Initialize & display a rectangle on an OpenGL surface via
5  * Glut.
6  *****/
7
8  #include <GL/glut.h>
9
10 void display(){
11     ...glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
12     ...glColor3f(0.0f, 0.0f, 0.0f);
13     ...glRectf(-0.75f, 0.75f, 0.75f, -0.75f);
14     ...glutSwapBuffers();
15 }
16
17 void idle(){
18     ...glutPostRedisplay();
19 }
20
21 void setup(){
22     ...glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
23 }
24
25
26 int main(int argc, char **argv)
27 {
28     ...glutInit(&argc, argv);
29     ...glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
30     ...glutInitWindowSize(800, 600);
31
32     ...glutCreateWindow("Hello World");
33     ...setup();
34     ...glutDisplayFunc(display);
35     ...glutIdleFunc(idle);
36     ...glutMainLoop();
37     ...return 0;
38 }
39
```

## Les principales fonctionnalités de la Glut

### glutInit(int\* argc, char\*\* argv)

La fonction glutInit initialise la bibliothèque GLUT et négocie une session avec le système de fenêtrage. Elle traite également les lignes de commandes qui sont propres à chaque système de fenêtrage. Si une erreur survient, la fonction peut se terminer

#### Paramètres

- **argc** : Un pointeur à la variable non modifiée argc du programme main. argc est modifiée par glutInit qui extrait les options de la ligne de commande
- **argv** : Un pointeur à la variable non modifiée argv du programme main. argv est mis à jour et les options de la ligne de commande utilisées par GLUT sont extraites.

-display DISPLAY	Spécifie l'adresse du serveur X auquel se connecter. Si ce n'est spécifié, la variable d'environnement est utilisée.
-geometry WxH+W+Y	Détermine la position de la fenêtre sur l'écran. Le paramètre de geometry doit être formaté selon la spécification standard de X.
-iconic	Indique de que la fenêtre sera créée dans un état iconisé.
-indirect	Force l'utilisation du contexte indirect de rendu réaliste d'OpenGL.
-direct	Force l'utilisation du contexte direct de rendu réaliste d'OpenGL. Une erreur fatale est rencontrée si le système ne supporte pas ce mode. Par défaut, le mode direct est utilisé, sinon c'est le mode indirect.
-gldebug	Après le traitement des fonctions de rappel ou des événements, vérifier s'il y a des erreurs d'OpenGL en appelant glGetError. S'il y a une erreur, imprimer un avertissement obtenu par la fonction gluErrorString.
-sync	Utiliser le protocole X synchronisé. Plus facile pour retracer les erreurs potentielles du protocole X

### Void glutInitDisplayMode(unsigned int mode)

Le mode d'affichage est utilisé pour créer les fenêtres et les sous-fenêtres. Le mode GLUT\_RGBA permet d'obtenir une fenêtre utilisant le modèle de couleur RGB, c'est le mode par défaut si aucun n'est spécifié.

GLUT_RGBA, GLUT_RGB	Masque de bits pour choisir une fenêtre en mode RGBA. C'est la valeur par défaut si GLUT_RGBA ou GLUT_INDEX ne sont spécifiés.	GLUT_DEPTH	Masque de bits pour choisir une fenêtre avec un tampon de profondeur.
GLUT_INDEX	Masque de bits pour choisir une fenêtre en mode index de couleur. Ceci l'emporte si GLUT_RGBA est	GLUT_STENCIL	Masque de bits pour choisir une fenêtre avec un tampon de pochoir.



	spécifié.		
GLUT_SINGLE	Masque de bits pour spécifier un tampon simple pour la fenêtre. Ceci est la valeur par défaut.	GLUT_STEREO	Masque de bits pour choisir une fenêtre stéréo.
GLUT_DOUBLE	Masque de bit pour spécifier une fenêtre avec un double tampon. Cette valeur l'emporte sur GLUT_SINGLE.		
GLUT_ACCUM	Masque de bits pour choisir une fenêtre avec un tampon d'accumulation.		

**void glutInitWindowSize ( int width, int height)**

**void glutInitWindowPosition ( int x, int y);**

width	Largeur de la fenêtre en pixels.
height	Hauteur de la fenêtre en pixels.
x	Position en x du coin gauche supérieur de la fenêtre.
y	Position en y du coin gauche supérieur de la fenêtre.

**int glutCreateWindow ( char \* name );**

name	Chaîne de caractères identifiant la fenêtre
------	---

Cette fonction crée une fenêtre. Le nom de la fenêtre dans la barre de titre de la fenêtre prend la valeur de la chaîne de caractères spécifiée par name. Cette fonction retourne un entier positif identifiant le numéro de la fenêtre. Cet entier peut par la suite être utilisé par la fonction glutSetWindow.

Chaque fenêtre possède un contexte unique d'OpenGL. Un changement d'état de la fenêtre associée au contexte d'OpenGL peut être effectué une fois la fenêtre créée. L'état d'affichage de la fenêtre à afficher n'est pas actualisé tant que l'application n'est pas entrée dans la fonction glutMainLoop. Ce qui signifie qu'aucun objet graphique ne peut être affiché dans la fenêtre, parce que la fenêtre n'est pas encore affichée.

**void glutMainLoop ( void )**

Cette fonction permet d'entrer dans la boucle de GLUT de traitement des événements. Elle est appelée seulement une fois dans une application. Dans cette boucle, les fonctions de rappel (backends) qui ont été enregistrées sont appelées à tour de rôle.

## **void glutDisplayFunc ( void (\*func) (void));**

func	Identifie la nouvelle fonction de rappel d'affichage (callback)
------	---

La fonction glutDisplayFunc établit la fonction de rappel pour la fenêtre courante. Quand GLUT détermine que le plan normal de la fenêtre doit être réaffiché, la fonction de rappel d'affichage est appelée. Avant l'appel, la fenêtre courante devient la fenêtre qui doit être réaffichée et (si aucune fonction de rappel d'affichage du plan de superposition [overlay] n'est inscrite) le plan normal devient la couche en utilisation. La fonction de rappel d'affichage ne possède aucun paramètre. La région du plan normal entier doit être réaffichée en réponse à la fonction de rappel (incluant les tampons auxiliaires si le programme dépend de leur état).

GLUT détermine quand la fonction de rappel doit être déclenchée en se basant sur l'état d'affichage de la fenêtre. L'état d'affichage peut être modifié explicitement en faisant appel à la fonction glutPostRedisplay ou lorsque le système de fenêtrage rapporte des dommages à la fenêtre. Si plusieurs requêtes d'affichage en différé ont été enregistrées, elles sont regroupées afin de minimiser le nombre d'appel aux fonctions de rappel d'affichage.

Quand un plan de recouvrement (superposition) est établi pour une fenêtre, et qu'aucune fonction de rappel d'affichage du plan de recouvrement de la fenêtre n'est inscrite, la fonction de rappel d'affichage est utilisée pour réafficher les plans normal et de recouvrement de la fenêtre ( la fonction est appelée si l'état du plan normal ou du plan de recouvrement l'exige). Dans ce cas, la couche en utilisation n'est pas implicitement changée à l'entrée de la fonction de rappel d'affichage.

Il faut se référer à la documentation de la fonction glutOverlayDisplayFunc pour comprendre la distinction entre les fonctions de rappel des plans normal et de recouvrement d'une fenêtre.

Lorsqu'une fenêtre (ou sous-fenêtre) est créée, aucune fonction de rappel d'affichage n'est inscrite pour cette fenêtre. Chaque fenêtre doit avoir une fonction de rappel inscrite. Une erreur fatale se produit si une tentative d'affichage d'une fenêtre est effectuée sans qu'une fonction de rappel n'ait été inscrite. C'est donc une erreur avec GLUT 3.0 de faire appel à la fonction glutDisplayFunc avec le paramètre NULL.

Au retour de la fonction de rappel, l'état endommagement normal de la fenêtre (retourné par la fonction glutLayerGet ( GLUT\_NORMAL\_DAMAGED)) est effacé. Si aucune fonction de rappel pour le plan de superposition de la fenêtre n'est inscrite, l'état endommagement superposition (retourné par une appel à la fonction glutLayerGet ( GLUT\_OVERLAY\_DAMAGED))) est également effacé.

## **void glutIdleFunc ( void (\*func) ( void ));**

func	Identifie la nouvelle fonction de rappel d'affichage (callback)
------	---

La fonction glutIdleFunc établit la fonction de rappel au repos de telle sorte que GLUT peut effectuer des tâches de traitement en arrière-plan ou effectuer une animation continue lorsque aucun événement n'est reçu. La fonction de rappel n'a aucun paramètre. Cette fonction est continuellement appelée lorsque aucun événement n'est reçu. La fenêtre courante et le menu courant ne sont pas changés avant l'appel à la fonction de rappel. Les applications utilisant plusieurs fenêtres ou menus doivent explicitement établir fenêtre courante et le menu courant, et ne pas se fier à l'état courant.

On doit éviter les calculs dans une fonction de rappel pour le repos afin de minimiser les effets sur le temps de réponse interactif.

### **void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)**

- Détermine les valeurs de remplissage pour nettoyer le buffer. (rouge, vert, bleu, alpha)
- Les valeurs sont comprises bornés dans l'intervalle [0,1]
- Les valeurs sont ensuite utilisées par glClear() pour nettoyer le buffer.

### **void glClear(GLbitfield mask);**

Les valeurs de masque sont composables par "ou" (| en C) pour effectuer plusieurs opérations d'effacement en une seule instruction glClear et rendre possible l'optimisation de ces opérations.

mask	Masque spécifiant le buffer à effacer
GL_COLOR_BUFFER_BIT	Effacement des pixels du buffer d'affichage
GL_DEPTH_BUFFER_BIT	Effacement de l'information de profondeur associée à chaque pixel du buffer d'affichage (information utilisée pour l'élimination des parties cachées)
GL_ACCUM_BUFFER_BIT	Effacement du tampon accumulation utilisé pour composer des images
GL_STENCIL_BUFFER_BIT	Effacement du tampon pinceau

### **void glColor3f(GLfloat red, GLfloat green, GLfloat blue)**

Spécifie les valeurs rouges, vertes, bleues pour définir la couleur courante.

### **void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)**

Dessine un rectangle ayant pour coin supérieur gauche x1, y1 et inférieur droit x2, y2.

### **void glutSwapBuffers ( void );**

Cette fonction échange les tampons de la couche en utilisation de la fenêtre courante. En fait, le contenu du tampon arrière de la couche en utilisation de la fenêtre courante devient le contenu du tampon avant. Le contenu du tampon arrière devient indéfini.

La fonction glFlush est appelée implicitement par glutSwapBuffers. On peut exécuter des commandes d'OpenGL immédiatement après glutSwapBuffers, mais elles prennent effet lorsque l'échange de tampon est complété. Si le mode double tamponnage n'est pas activé, cette fonction n'a aucun effet.

## **void glutPostRedisplay ( void );**

Cette fonction indique que le plan normal de la fenêtre courante doit être réaffiché. Lors de la prochaine itération dans la boucle principale de glutMainLoop, la fonction de rappel d'affichage est appelée et le plan normal est affiché. Plusieurs appels à la fonction glutPostRedisplay n'engendrent qu'un seul rafraîchissement.

Logiquement, une fenêtre endommagée est marquée comme devant être rafraîchie, ce qui est équivalent à faire appel la fonction glutPostRedisplay.

## **Évènements de la glut**

Les événements dans un programme informatique sont les interventions de l'utilisateur par le biais de la souris, du clavier, d'un joystick, ect...

## **L'affichage**

L'événement d'affichage a lieu à chaque fois que la vue doit être rafraîchie. La fonction d'affichage (ou display function) doit alors redessiner les objets. L'affichage a lieu notamment dans les circonstances suivantes :

- Création de la fenêtre graphique ;
- Passage de la fenêtre au premier plan ;
- Appel explicite du programmeur (voir la fonction glutPostRedisplay) lorsqu'il le juge nécessaire.

Avec la glut, on doit déclarer la fonction d'affichage en utilisant la fonction glutDisplayFunc, qui prend en paramètre un pointeur de fonctions.

La fonction glutDisplayFunc a pour prototype :

`void glutDisplayFunc(void (*func)(void));`

c'est à dire qu'elle prend en paramètre une fonction d'affichage qui, elle, ne prend aucun paramètre.

**Exemple GlutDisplayFunc.** La fonction d'affichage de l'exemple dessine tout simplement le background de la fenêtre en Rouge.

```
void display(void )
{
    /* coef. RGB + A of background-color*/
    glClearColor(1,0,0,0);
    /* effaçage */
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    /* Send calculated image to screen */
    glutSwapBuffers();
}
```

## **Les frappes de touches clavier**

Les événements liés aux touches du clavier peuvent être déclarer grace aux fonctions glutKeyboardFunc (pour les caractères usuelles) et glutSpecialFunc (pour les touches spéciales telles que F 1, F 2, les flèches, etc...).

**Exemple GlutKeyboardFunc.** Dans le programme suivant, la couleur du fond est grise de plus en plus claire lorsqu'on appuie sur la flèche droite au clavier, et de plus en plus foncée lorsqu'on appuie sur la flèche gauche. Le programme se termine lorsqu'on appuie sur la touche 'q'.

```

void specialKeyboard(int touche, int x, int y)
{
    switch(touche)
    {
        case GLUT_KEY_LEFT:
            grayLevel -= 0.05;
            if(grayLevel < 0)
                grayLevel = 0;
            break;
        case GLUT_KEY_RIGHT:
            grayLevel += 0.05;
            if(grayLevel > 1)
                grayLevel = 1;
            break;
        default:
            fprintf(stdout,"Touche non gérée\n");
    }
    glutPostRedisplay(); /* rafraîchissement de l'affichage */
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 'q':
            exit(0);
        default:
            fprintf(stderr,"Touche non gérée\n");
            break;
    }
    glutPostRedisplay(); /* rafraîchissement de l'affichage */
}

```

```

glutDisplayFunc(display);
/* Setting th keyboard callback */
glutKeyboardFunc(keyboard);
glutSpecialFunc(specialKeyboard);

```

## Gestion de la souris

La fonction gérant l'événement de pression sur un bouton de la souris est déclarée par la fonction `glutMouseFunc`. La fonction gérant le mouvement de la souris avec un bouton pressé est déclarée par la fonction `glutMotionFunc`.

**Exemple *GlutMouseFunct*.** Dans le programme suivant, lorsqu'on déplace la souris vers la droite, la couleur du fond devient plus bleue. Lorsqu'on déplace la souris vers la gauche la couleur du fond devient moins bleue. Lorsqu'on déplace la souris vers le haut, la couleur du fond devient plus verte. Lorsqu'on déplace la souris vers le bas, la couleur du fond devient moins verte.

...

```
void mouseButtonPressed(int button, int state, int x, int y)
{
    if (GLUT_LEFT_BUTTON == button)
    {
        /* Saving mouse position on left mouse button pressed */
        if (GLUT_DOWN == state)
        {
            leftButtonDown = 1;
            mousex = x;
            mousey = y;
        }
        if (GLUT_UP == state)
            leftButtonDown = 0;
    }
}

void mouseMove(int x, int y)
{
    if (leftButtonDown)
    {
        coef_b += 0.01*(x-mousex);
        if(1 < coef_b )
            coef_b = 1;
        if(0 > coef_b )
            coef_b = 0;
        coef_g += 0.01*(y-mousey);
        if(1 < coef_g )
            coef_g = 1;
        if(0 > coef_g )
            coef_g = 0;
        mousex = x;      /* enregistrement des nouvelles */
        mousey = y;      /* coordonnées de la souris */
        glutPostRedisplay();
    }
}
```

```
glutDisplayFunc(display);
glutMouseFunc(mouseButtonPressed);
glutMotionFunc(mouseMove);
```

## Événement Idle : pour les animations et les tâches de fond.

L'événement Idle est un événement qui se produit régulièrement lorsqu'aucun autre événement ne survient. L'événement Idle permet de faire des animations en modifiant l'état de la vue pour créer du mouvement. Dans le programme suivant, la couleur du fond de l'image varie en fonction du temps.

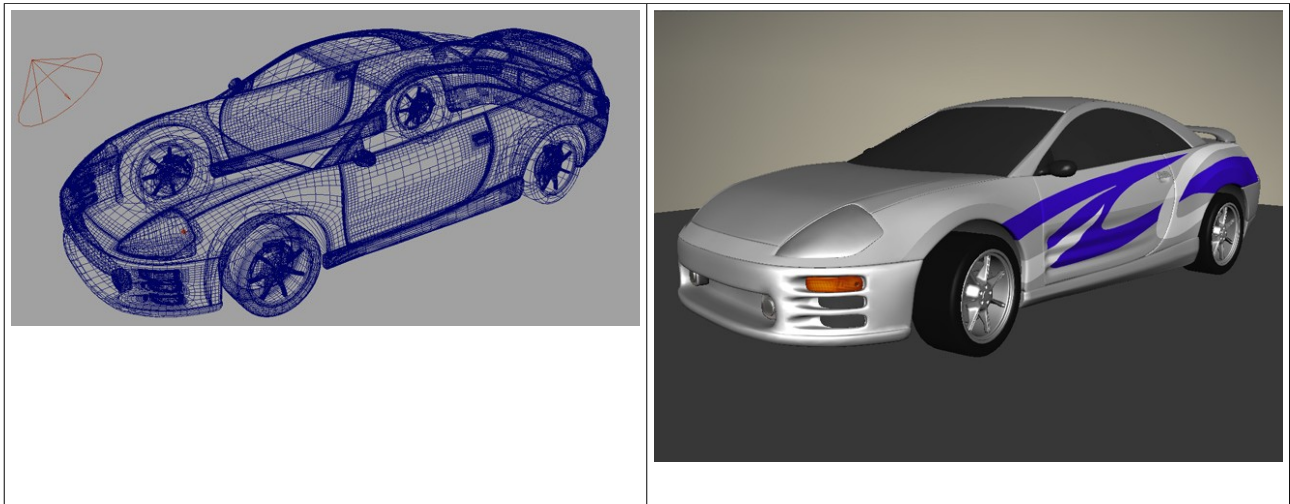
```
void display(void )
{
    glClearColor(0, coef_g, coef_b, 0);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}

void IdleFunction(void )
{
    coef_b = sin(speed*parameter);
    parameter = parameter + 1;
    glutPostRedisplay();
}
```

```
glutDisplayFunc(display);
glutIdleFunc(idleFunction);    /* appel de glutIdleFunc */
```

## Dessiner des formes de base

Toute forme 3D est une composition de polygones (triangle, quadrilatère, ...). En effet, la modélisation d'une forme consiste en la décomposition de cette dernière en un maillage 3D, constitué de polygones.



### 2.1 Les primitives géométriques

En OpenGL, pour décrire un polygone on doit mettre ses sommets entre les fonctions `glBegin()` et `glEnd()`. C'est la fonction `glBegin` qui permet de déterminer la primitive géométrique à dessiner selon l'argument qui lui est passé en paramètre.

Exemple 1 : dessiner un triangle

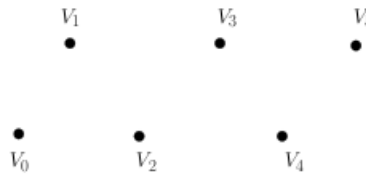
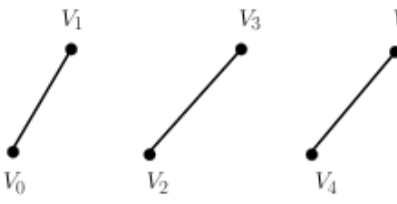
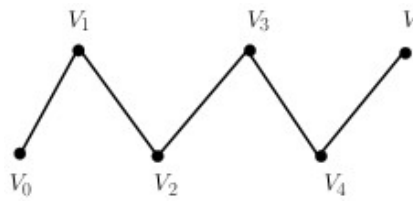
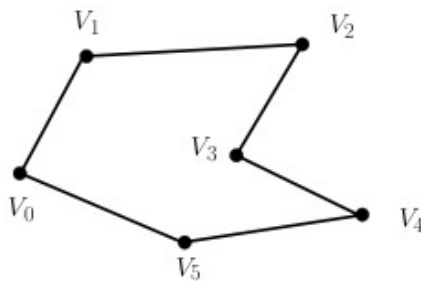
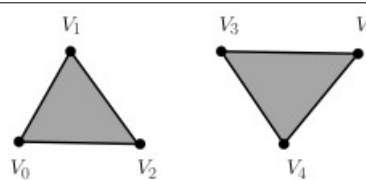
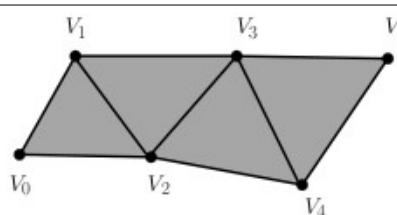
```
glBegin(GL_TRIANGLES);
glVertex2f(0.0, 0.0);
glVertex2f(1.0, 0.0);
glVertex2f(0.0, 1.0);
glEnd();
```

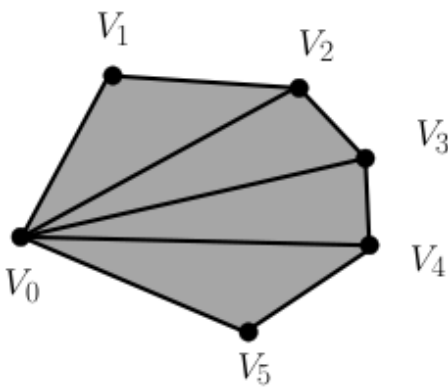
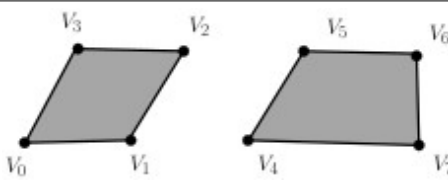
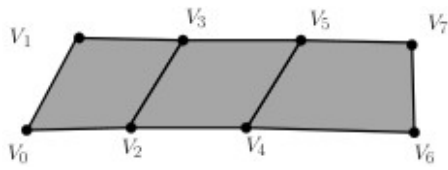
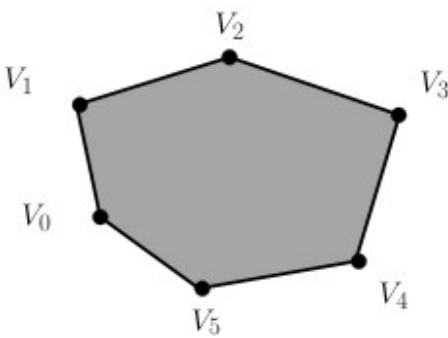
Exemple 2 : dessiner un polygone (utilisation de `math.h`)

```
glBegin(GL_POLYGON);
for (GLint i=0 ; i<6 ; i++)
    glVertex2f(cos(2*i*M_PI/6), sin(2*i*M_PI/6));
glEnd();
```

et plus généralement, voici les différents arguments correspondant aux primitives géométriques pouvant être passés à la fonction `glBegin(arg)`.



<b>GL_POINTS</b>		seuls des points sont dessinés. (L'épaisseur des points peut être réglée avec la fonction <code>glPointSize()</code> ).
<b>GL_LINES</b>		Des segments de droites sont dessinées entre V0 et V1 , entre V2 et V3 , etc., les $V_i$ étant les sommets successifs définis par des appels à <code>glVertex*()</code> . (L'épaisseur des droites peut être réglée avec la fonction <code>glLineWidth()</code> ).
<b>GL_LINE_STRIP</b>		Une ligne polygonale est tracée
<b>GL_LINE_LOOP</b>		Une ligne polygonale fermée est tracée.
<b>GL_TRIANGLES</b>		Des triangles sont dessinés entre les trois premiers sommets, entre les trois suivants, etc... Si le nombre de sommets n'est pas multiple de 3, les 1 ou 2 derniers sommets sont ignorés.
<b>GL_TRIANGLE_STRIP</b>		Des triangles sont dessinés entre V0 , V1 , V2 , puis entre V1 , V2 , V3 , entre V2 , V3 , V4 , etc.

<b>GL_TRIANGLE_FAN</b>		Un éventail est dessiné formé des triangles V0 , V1 , V2 , puis V0 , V2 , V3 , etc.
<b>GL_QUADS</b>		Un quadrilatère est un polygone à 4 sommets. Des quadrilatères sont dessinés entre les quatre premiers sommets, puis entre les 4 suivants, etc.
<b>GL_QUAD_STRIP</b>		Une série de quadrilatères sont dessinés entre V0 , V1 , V3 , V2 , puis, V2 , V3 , V5 , V4 , puis V4 , V5 , V7 , V6 , etc.
<b>GL_POLYGON</b>		Un polygone fermé rempli est dessiné. Le polygone doit être convexe faute de quoi le comportement est indéfini (dépend de l'implémentation).

## 2.2 Variantes de *glVertex\*()* et représentations des sommets

Un sommet est toujours représenté dans l'espace R3. On peut le spécifier en utilisant 2, 3 ou même 4 coordonnées.

- Lorsqu'on utilise 2 coordonnées, la troisième vaut 0
- Lorsqu'on utilise 4 coordonnées, il s'agit d'une représentation en coordonnées homogènes. (voir plus bas)

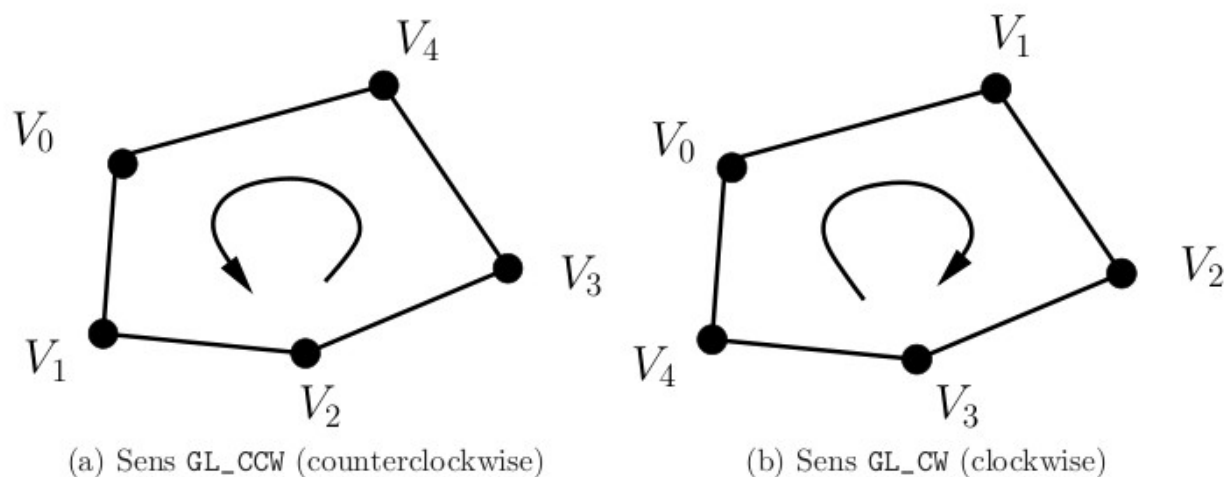
Les coordonnées peuvent être spécifiées soit par une liste de paramètres, soit par un vecteur de coordonnées.

A chacune de ces possibilités correspond une fonction *glVertex* :

<i>glVertex2f</i>	Deux paramètres flottants en simple précision de type GLfloat (la troisième
-------------------	---

	coordonnée vaut 0) ;
glVertex3f	Trois paramètres flottants en simple précision de type GLfloat
glVertex2fv	un paramètre de type tableau de 2 GLfloats
glVertex3fv	un paramètre de type tableau de 3 GLfloats.
glVertex2d, glVertex3d, glVertex2dv, glVertex3dv	similaires aux précédents mais avec des floats en double précision de type GLdouble.
glVertex2i, glVertex3i, glVertex2di, glVertex3di	similaires aux précédents mais avec des entiers 32 bits de type GLint.

## 2.3 Modes d'affichage de polygones



Les polygones en 3D ont deux cotés : le devant et le derrière. Par défaut, un polygone est vu de face si sa projection dans la fenêtre graphique voit la liste de ses sommets dans l'ordre positif trigonométrique (on peut changer ce comportement avec la fonction **glFrontFace**).

On peut spécifier un mode d'affichage différents pour les polygones vus de face et pour les polygones vus de dos par la fonction **glPolygonMode**.

void glPolygonMode(GLenum face, GLenum mode);	
face = {GL_FRONT, GL_BACK, GL_FRONT_AND_BACK }	mode = { - GL_POINT, dessine uniquement des sommets du polygone - GL_LINE, dessine des contours du polygone - GL_FILL, dessin du polygone avec remplissage }

On peut également éliminer les faces (par exemple) qui sont vues de dos pour accélérer l'affichage,

(si l'objet est fermé et vu de l'extérieur, on sait qu'aucune face vue de dos ne sera visible car elles seront cachées par d'autres faces. Pour cela, il faut activer le culling par ***glEnable(GL\_CULL\_FACE)*** et indiquer le type de polygones à éliminer par l'usage de la fonction void ***glCullFace(GLenum mode)***.

- mode = {GL\_FRONT, GL\_BACK ou GL\_FRONT\_AND\_BACK}

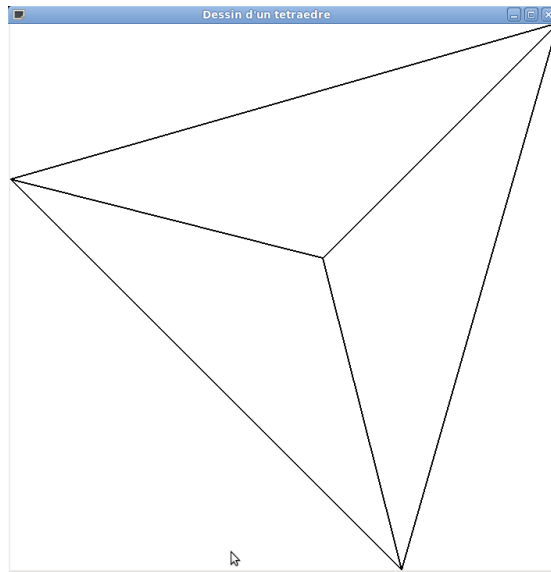
## 2.4 Représentation d'un maillage

### 2.4.1 Définition d'un maillage

Un maillage  $P$  dans l'espace tridimensionnel  $R^3$  est la donnée de :

- Une suite de **points**  $P_0, P_1, \dots, P_{n-1}$  de  $R^3$  appelés sommets du maillage ;
- Un ensemble de **faces**, chaque face étant une suite de numéros de sommets dans  $\{0, \dots, n-1\}$ .

### 2.4.2 Exemple



Par exemple, considérons le tétraèdre construit sur les quatre points

$A = (400, 400, -10)$ ,

$B = (700, 700, -10)$ ,

$C = (0, 500, -10)$

et  $D = (500, 0, -500)$

Le maillage correspondant est la donnée de :

1. Les sommets  $P_0 = A$ ,  $P_1 = B$ ,  $P_2 = C$ , et  $P_3 = D$  ;
2. Les quatre faces qui sont :
  - La face numéro 0 représentant le triangle OAB : (0, 1, 2) ;
  - La face numéro 1 représentant le triangle OAC : (0, 1, 3) ;
  - La face numéro 2 représentant le triangle OBC : (0, 2, 3) ;
  - La face numéro 3 représentant le triangle ABC : (1, 2, 3).

### **Le vecteur normal en un point**

Un vecteur normal (aussi appelé normale ) à une surface en un point de cette surface est un vecteur dont la direction est perpendiculaire à la surface.

Pour une surface plane, les vecteurs normaux en tous points de la surface ont la même direction. Ce n'est pas le cas pour une surface quelconque.

C'est grâce au vecteur normal que l'on peut spécifier l'orientation de la surface dans l'espace, et en particulier l'orientation par rapport aux sources de lumière. L'appel à `glNormal*()` donne une valeur

à la normale courante. Elle sera associée aux points spécifiés par les appels suivants à glVertex\*().

## ***Le modèle d'ombrage***

Chaque facette d'un objet peut être affichée d'une unique couleur (ombrage plat) ou à l'aide de plusieurs couleurs (ombrage lissé). OpenGL implémente une technique de lissage appelée Ombrage de Gouraud. Ce choix s'effectue avec glShadeModel(GLenum mode).

***Exemple GLShade***, changer de mode de rendu avec les touches 's' et 'S'.

## ***Elimination des surfaces cachées***

OpenGL utilise la technique du Z buffer (ou buffer de profondeur) pour éviter l'affichage des surfaces cachées. On ne voit d'un objet que les parties qui sont devant et pas celles qui se trouvent derrière.

Pour chaque élément de la scène la contribution qu'il aurait à l'image s'il était visible est calculée, et est stockée dans le Z buffer avec la distance de cet élément à la caméra (c'est cette distance qui est la profondeur).

Chaque pixel de l'image garde donc la couleur de l'élément qui est le plus proche de la caméra. Dans le cas plus complexe d'un objet transparent, il y a une combinaison des couleurs de plusieurs éléments.

### ***Fonctions utilisées***

glutInitDisplayMode(GLUT\_DEPTH | ... )

glEnable(GL\_DEPTH\_TEST)

glClear (GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) Avant l'affichage d'une scène.

# L'architecture OpenGL

## ***OpenGL est basé sur des Etats***

C'est le principe général d'OpenGL : On positionne un état interne, et sa valeur est ensuite utilisée comme valeur courante : les ordres de dessin suivants utiliseront cette valeur-ci.

Si ce n'est pas clair pour vous, prenons un exemple : Pour dessiner un sommet rouge, on positionne d'abord l'état correspondant à la couleur courante à la valeur rouge, et ensuite on demande le dessin d'un sommet. Tous les sommets qui seront ensuite dessinés seront rouges, tant que l'on n'a pas modifié la couleur courante.

Et ce principe que nous avons illustré à partir de l'état "couleur" s'applique à tous les états, comme l'épaisseur des traits, la transformation courante, l'éclairage, etc.

## ***Pipe-Line de Rendu simplifié***

[Données]->[Évaluateurs]->[Opérations sur les sommets et assemblage de primitives]->[Discrétisation]->[Opérations sur les fragments]->[Image]

- Les **Évaluateurs** produisent une description des objets à l'aide de sommets et de facettes.
- Les **Opérations** sur les sommets sont les transformations spatiales (rotations et translations) qui sont appliquées aux sommets.
- L' **assemblage de primitive** regroupe les opérations de clipping : élimination des primitives qui sont en dehors d'un certain espace et de transformation perspective .
- La **discrétisation** (Rasterization) est la transformation des primitives géométriques en fragments correspondant aux pixels de l'image.
- Les **Opérations sur les fragments** vont calculer chaque pixel de l'image en combinant les fragments qui se trouvent à l'emplacement du pixel. On trouve entre autres la gestion de la transparence, et le Z-buffer (pour l'élimination des surfaces cachées).

# Modélisation hiérarchique

## Description hiérarchique d'une scène

Un modèle hiérarchique permet de décrire facilement des objets complexes composés d'objets simples. La scène est organisée dans un arbre tel que les objets ne sont plus définis par leur transformation absolue par rapport au repère de toute la scène, mais par leur transformation relative dans cet arbre.

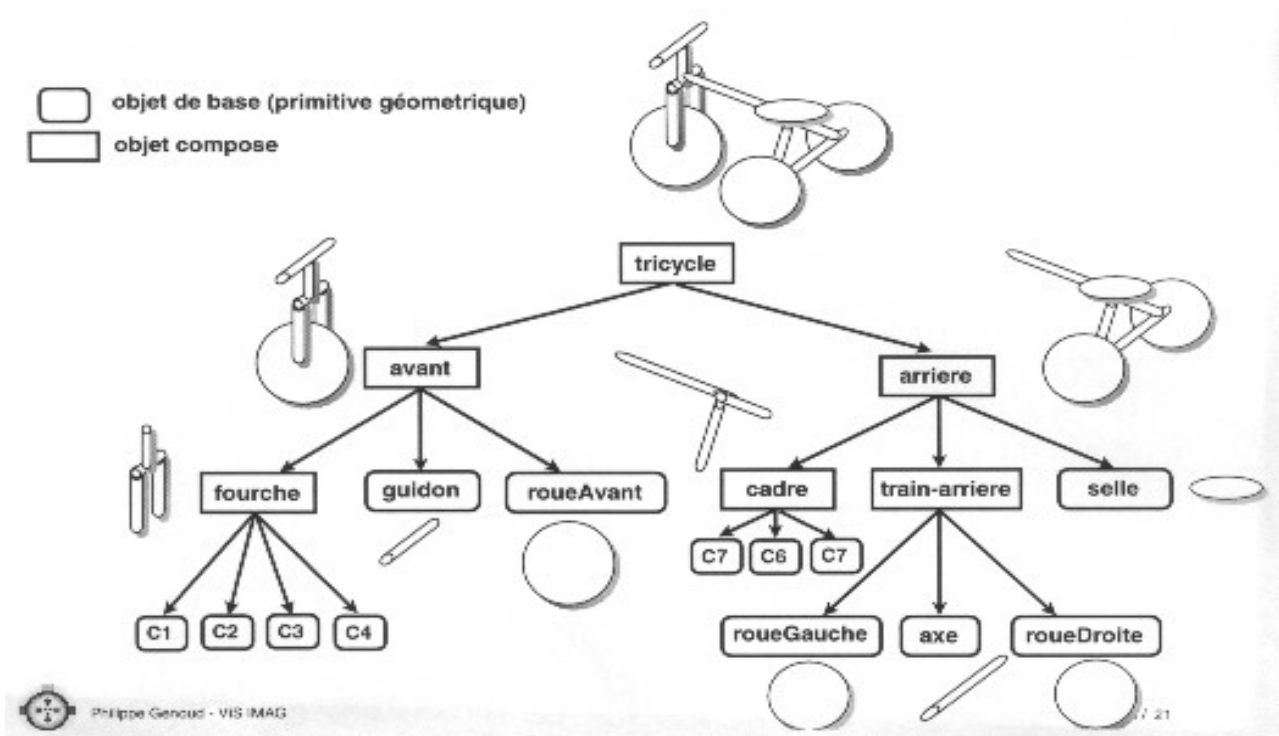
Comme un même objet peut être inclus plusieurs fois dans la hiérarchie, la structure de données est un Graphe Orienté Acyclique (DAG). A chaque noeud est associé un repère. Le repère associé à la racine est le repère de la scène. A chaque arc est associée une transformation géométrique qui positionne l'objet fils dans le repère de son père.

## Construction

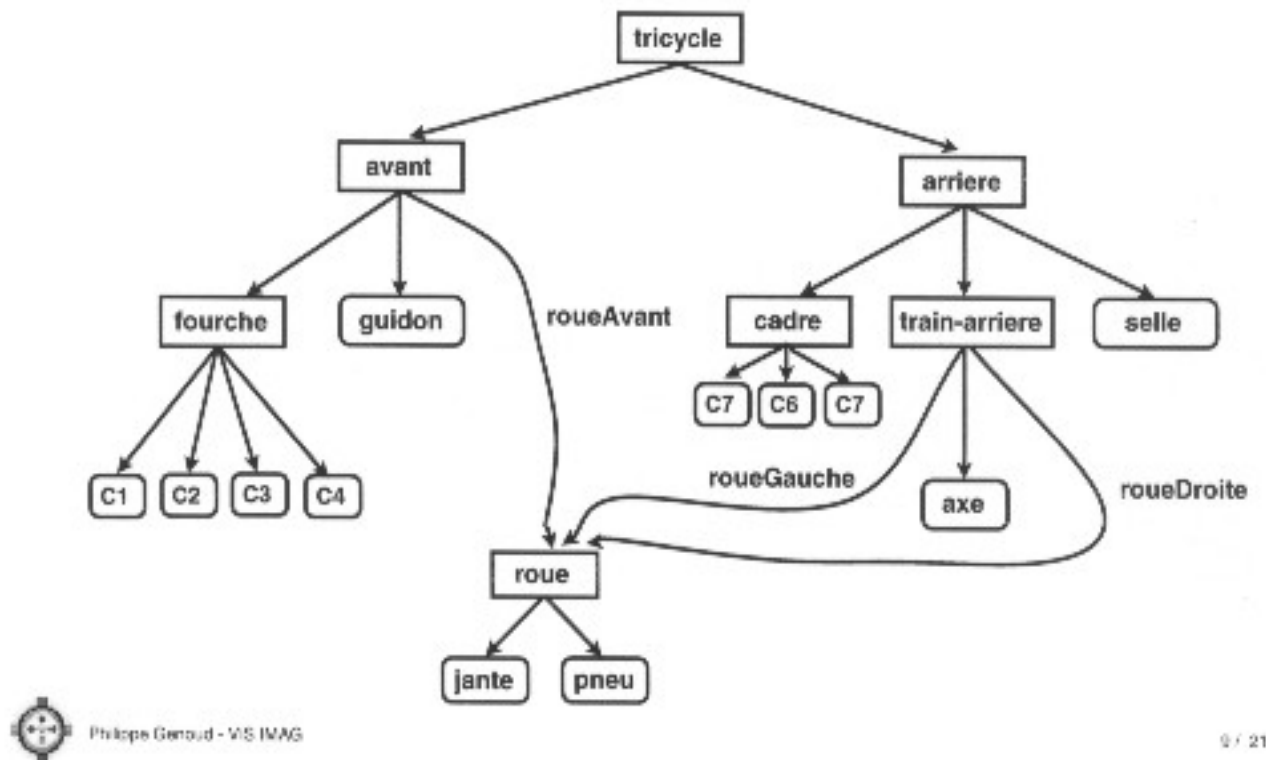
La structure hiérarchique du modèle peut être induite par :

Un processus de construction ascendant ("bottom-up") dans lequel les composants de base (primitives géométriques) sont utilisés comme des blocs de construction et assemblés pour créer des entités de niveau plus élevé, et ainsi de suite.

Un processus de construction descendant ("top-down") dans lequel on effectue une décomposition récursive d'un modèle géométrique en objets plus simples jusqu'à aboutir à des objets élémentaires (primitives géométriques)







## Pile de transformations

OpenGL utilise les coordonnées homogènes pour manipuler ses objets (cf Cours de Math). Il maintient trois matrices 4x4 distinctes pour contenir les différentes transformations.

Pour coder l'arbre de description de la scène, il faut utiliser la pile de transformation, en empilant la matrice de transformation courante (sauvegarde des caractéristiques du repère local associé) avant de descendre dans chaque nœud de l'arbre, et en dépilant la matrice en remontant (récupération du repère local associé).

`glPushMatrix()` Empile la matrice courante pour sauvegarder la transformation courante.

`glPopMatrix()` Dépile la matrice courante (La matrice du haut de la pile est supprimée de la pile, et elle devient la matrice courante)

## Manipulation des matrices de transformation

`glMatrixMode(GLenum mode)` spécifie quelle matrice sera affectée par les commandes suivantes de manipulation de transformations : la matrice de modélisation-vision, de projection ou de texture.

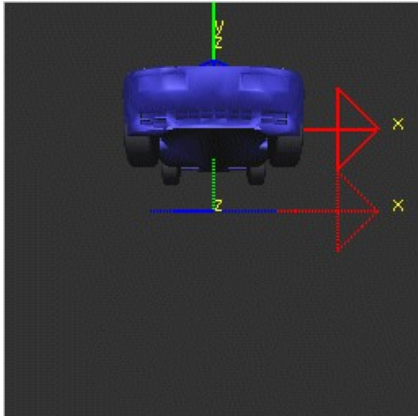
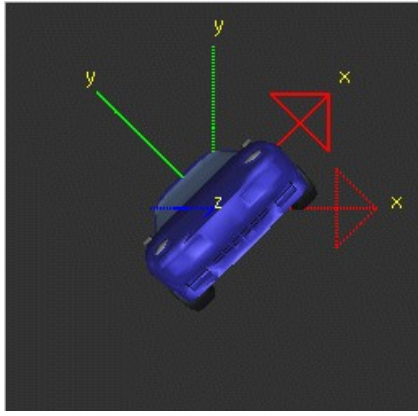
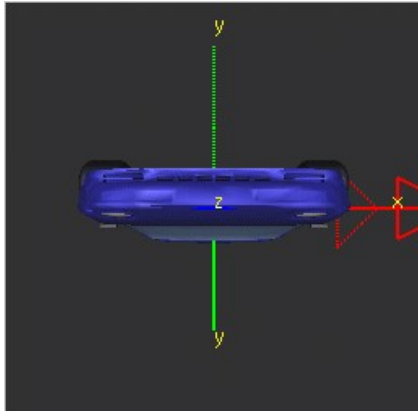
`glLoadIdentity()` donne à la Matrice courante la valeur identité. (comme déjà indiqué)

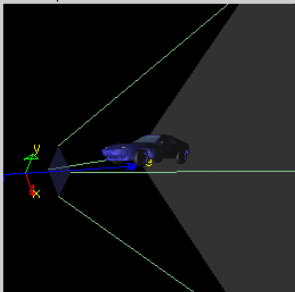
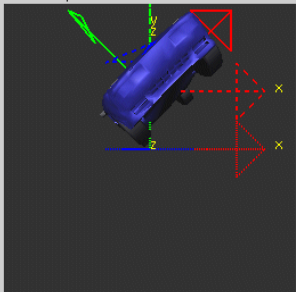
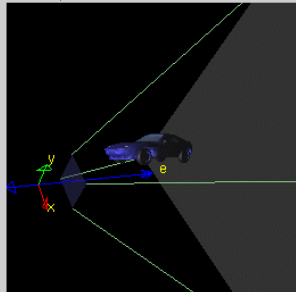
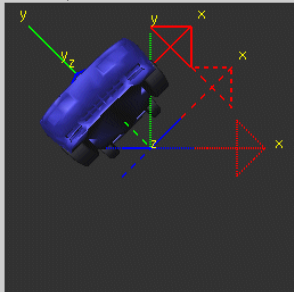
`glLoadMatrix*(const TYPE * m)` donne à la Matrice courante la valeur de la matrice `m`.

`glMultMatrix*(const TYPE * m)` multiplie la Matrice courante par la matrice `m`.

## La transformation de modélisation

La manipulation d'objets 3D peut mener à des réflexions complexes pour bien positionner et orienter les objets. En effet, l'application successive d'une translation puis d'une rotation, ou l'inverse mènent à des résultats différents.

<p><code>glTranslate*(TYPE x, TYPE y, TYPE z)</code> translate le repère local de l'objet du vecteur (x,y,z).</p>	<p><code>glRotate*(TYPE angle, TYPE x, TYPE y, TYPE z)</code> opère une rotation de l'objet autour du vecteur (x,y,z).</p>	<p><code>glScale*(TYPE a, TYPE b, TYPE c)</code> opère un changement d'échelle sur 3 axes. Les coordonnées en x sont multipliées par a, en y par b et en z par c.</p>
		
<p><code>glTranslatef( 0.0, 0.5, 0.0)</code></p>	<p><code>glRotatef(45.0 , 0.0 , 0.0 , 1.0)</code></p>	<p><code>glScalef(1.5 , -0.5 , 1.0)</code></p>

<p>World-space view</p>  <p>Screen-space view</p>  <p>Command manipulation window</p> <pre>glTranslatef( 0.00 , 0.50 , 0.00 ); glRotatef( 45.0 , 0.00 , 0.00 , 1.00 );</pre>	<p>World-space view</p>  <p>Screen-space view</p>  <p>Command manipulation window</p> <pre>glRotatef( 45.0 , 0.00 , 0.00 , 1.00 ); glTranslatef( 0.00 , 0.50 , 0.00 );</pre>
--	---

Pour s'y retrouver, il suffit de considérer que la transformation est appliquée au repère local de l'objet.

# La vision

## Principe de la vision

Le processus de transformation qui produit une image à partir d'un modèle de scène 3D est analogue à celui qui permet d'obtenir une photographie d'une scène réelle à l'aide d'un appareil photo. Il comprend quatre étapes :

Etape	Appareil Photographique	Transposition OpenGL	Type de transformation
1	Positionner l'appareil photo	Placer la caméra virtuelle	transformation de vision
2	Arranger les éléments d'une scène à photographier	Composer une scène virtuelle à représenter	transformation de modélisation
3	Choisir la focale de l'appareil photo	choisir une projection	transformation de projection
4	Choisir la taille de la photographie au développement	choisir les caractéristiques de l'image	transformation de cadrage

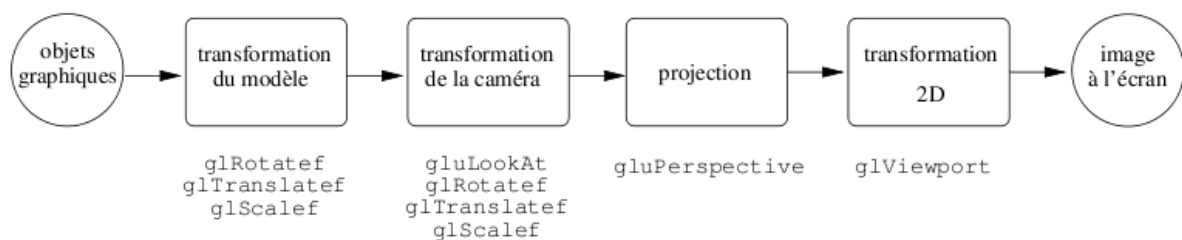
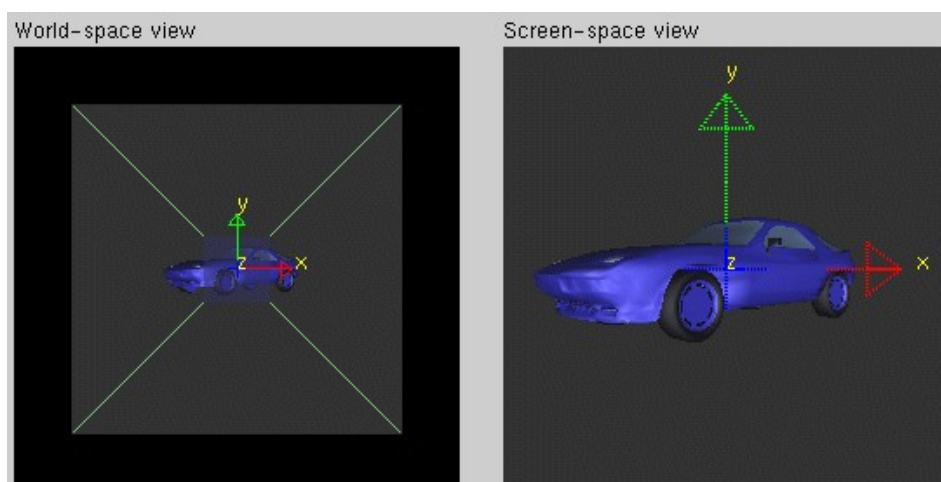


FIG. 3.6: Les transformations géométriques subies par les objets avant affichage

## La transformation de vision

Elle modifie la position et l'orientation de la caméra virtuelle. La position par défaut de la caméra est à l'origine du repère de la scène, orientée vers les z négatifs, et la verticale de la caméra est alignée avec l'axe des y positifs :



En fait, ce qui compte pour la visualisation c'est la position relative de la caméra par rapport aux objets. Il est équivalent par exemple de translater la caméra d'un vecteur  $T$  ou de translater tous les objets du vecteur  $-T$ . On utilise ainsi les procédures OpenGL de translation et de rotation appliquées aux objets, `glTranslate*()` et `glRotate*()` pour changer le point de vue de la caméra.

On peut également utiliser (et c'est plus simple pour commencer ) La fonction `gluLookAt` permet de modifier du `GL_MODELVIEW` pour régler la transformation de la caméra.

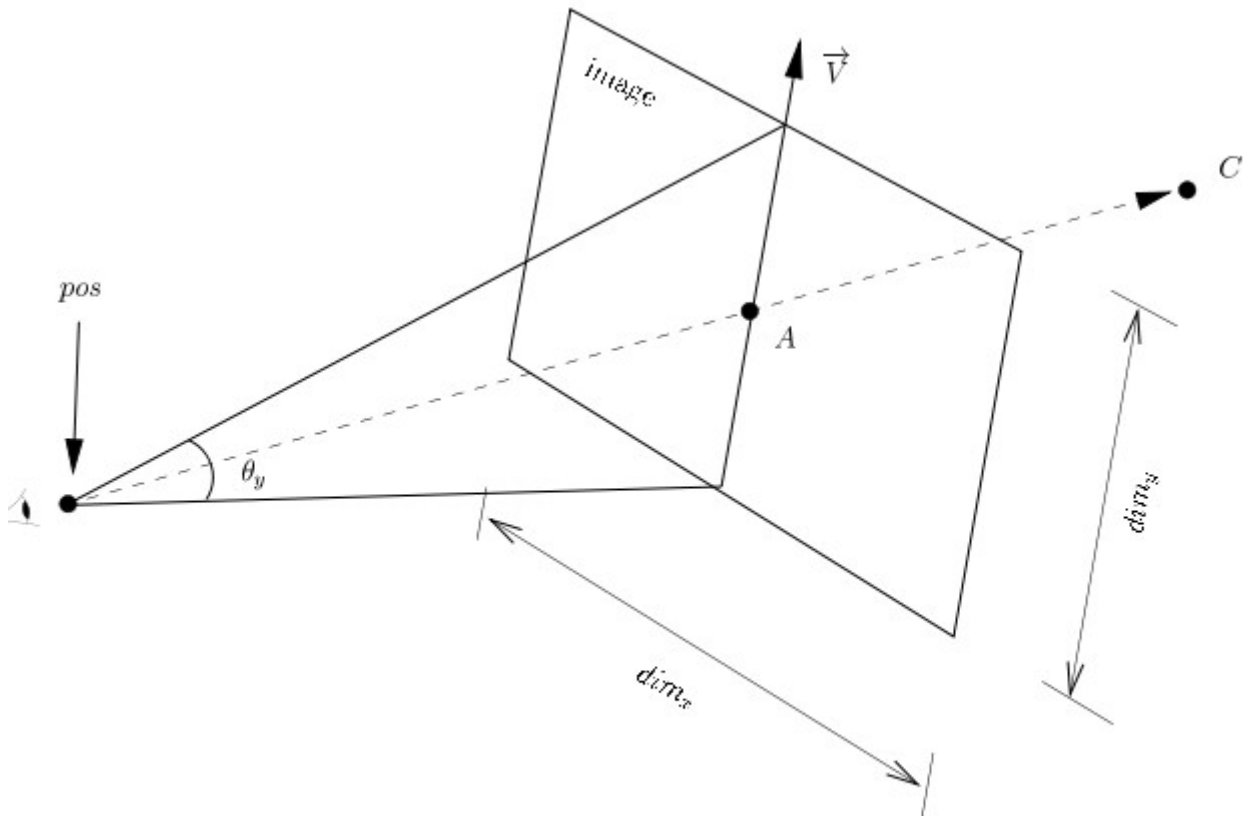


FIG. 3.9: Les paramètres de `gluLookAt`

Les paramètres de `gluLookAt` sont :

1. les coordonnées de la position (`posx` , `posy` , `posz` ) de la caméra ;
2. Les coordonnées (`Cx` , `Cy` , `Cz` ) d'un point dans la direction de visée (ce point est aussi appelé le centre) ;
3. Les coordonnées d'un vecteur  $V$  qui doit apparaître vertical dans l'image obtenue par projection.

Le prototype de `gluLookAt` est :

```
void gluLookAt(GLdouble posx, GLdouble posy, GLdouble posz,
               GLdouble Cx, GLdouble Cy, GLdouble Cz,
               GLdouble Vx, GLdouble Vy, GLdouble Vz);
```

La fonction `gluLookAt` doit être utilisée dans le mode `GL_MODELVIEW`.

## La transformation de modélisation

vue précédemment.

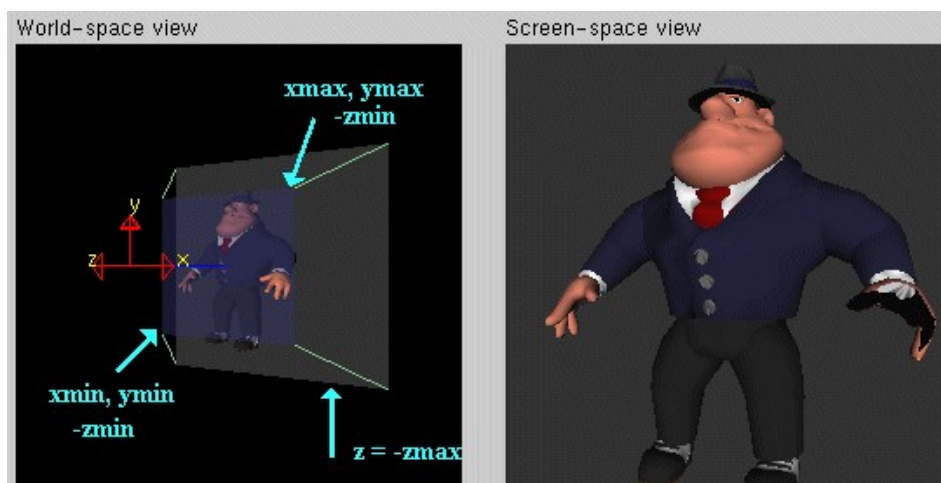
## La transformation de projection

Avant de pouvoir effectuer toute commande liée à la projection, il est nécessaire de changer d'état en exécutant les commandes suivantes

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();
```

### Projection en perspective.

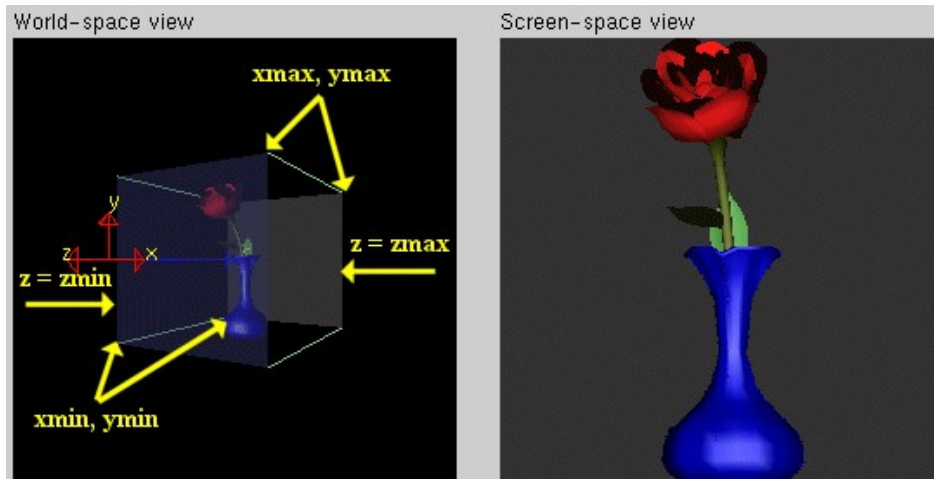
`gluPerspective( GLdouble angle_ouverture, GLdouble ratio_XY, GLdouble zmin, GLdouble zmax)` ou `glFrustum(GLdouble xmin, GLdouble xmax, GLdouble ymin, GLdouble ymax, GLdouble zmin, GLdouble zmax )` peuvent être utilisées. Ces fonctions définissent le volume de vision sous la forme d'une pyramide tronquée.



notez que la main gauche et le pied gauche d'Al Capone sortent du volume de vision Dans la fonction `glFrustum( ... )`, les coins bas-gauche et haut-droite du plan de clipping proche ont pour coordonnées respectives  $(x_{min}, y_{min}, -z_{min})$  et  $(x_{max}, y_{max}, -z_{min})$ .  $z_{min}$  et  $z_{max}$  indiquent la distance de la caméra aux plans de clipping en  $z$ , ils doivent être positifs.

## Projection orthographique

`glOrtho( GLdouble xmin, GLdouble xmax, GLdouble ymin, GLdouble ymax, GLdouble zmin, GLdouble zmax )` sera utilisée. Le volume de vision est un pavé : un parallélépipède rectangle. Il est représenté en perspective dans l'image de gauche ci-dessous, mais tous ses angles sont bien droits.



## Clipping

Les 6 plans de définition du volume de vision sont des plans de clipping : toutes les primitives qui se trouvent en dehors de ce volume sont supprimées.

## La transformation de cadrage

`glViewport(GLint x, GLint y, GLint largeur, GLint hauteur)` sera utilisée pour définir une zone rectangulaire de pixels dans la fenêtre dans laquelle l'image finale sera affichée. Par défaut, les valeurs initiales du cadrage sont (0,0, largeurFenêtre, hauteurFenêtre)

# Eclairage

## **Modèle d'éclairage**

La perception de la couleur de la surface d'un objet du monde réel dépend de la distribution de l'énergie des photons qui partent de cette surface et qui arrivent aux cellules de la rétine de l'oeil. Chaque objet réagit à la lumière en fonction des propriétés matérielles de sa surface.

Le modèle d'éclairage d'OpenGL considère qu'un objet peut émettre une lumière propre, renvoyer dans toutes les directions la lumière qu'il reçoit, ou réfléchir une partie de la lumière dans une direction particulière, comme un miroir ou une surface brillante.

Les lampes, elles, vont envoyer une lumière dont les caractéristiques seront décrites par leurs trois composantes : ambiante, diffuse ou spéculaire. OpenGL distingue quatre types de lumières :

## **Lumière émise (Ne concerne que les objets)**

Les objets peuvent émettre une lumière propre, qui augmentera leur intensité, mais n'affectera pas les autres objets de la scène.

## **Lumière ambiante (Concerne les objets et les lampes)**

C'est la lumière qui a tellement été dispersée et renvoyée par l'environnement qu'il est impossible de déterminer la direction d'où elle émane. Elle semble venir de toutes les directions. Quand une lumière ambiante rencontre une surface, elle est renvoyée dans toutes les directions.

## **Lumière diffuse (Concerne les objets et les lampes)**

C'est la lumière qui vient d'une direction particulière, et qui va être plus brillante si elle arrive perpendiculairement à la surface que si elle est rasante. Par contre, après avoir rencontré la surface, elle est renvoyée uniformément dans toutes les directions.

## **Lumière spéculaire (Concerne les objets et les lampes)**

La lumière spéculaire vient d'une direction particulière et est renvoyée par la surface dans une direction particulière. Par exemple un rayon laser réfléchi par un miroir.

## **Brillance (Ne concerne que les objets)**

Cette valeur entre 0.0 et 128.0 détermine la taille et l'intensité de la tâche de réflexion spéculaire. Plus la valeur est grande, et plus la tâche est petite et l'intensité importante.

## **Les lampes**

### **Nombre de lampes**

OpenGL offre d'une part une lampe qui génère uniquement une lumière ambiante (lampe d'ambiance), et d'autre part au moins 8 lampes (GL\_LIGHT0, ... , GL\_LIGHT7) que l'on peut

placer dans la scène et dont on peut spécifier toutes les composantes.

La lampe GL\_LIGHT0 a par défaut une couleur blanche, les autres sont noires par défaut.

La lampe GL\_LIGHTi est allumée par un glEnable(GL\_LIGHTi)

Il faut également placer l'instruction glEnable(GL\_LIGHTING) pour indiquer à OpenGL qu'il devra prendre en compte l'éclairage.

## Couleur des lampes

Dans le modèle d'OpenGL, la couleur d'une composante de lumière d'une lampe est définie par les pourcentages de couleur rouge, verte, bleue qu'elle émet.

Par exemple voici une lampe qui génère une lumière ambiante bleue :

```
GLfloat bleu[4] = { 0.0, 0.0, 1.0, 1.0 };  
glLightfv(GL_LIGHT0, GL_AMBIENT, bleu);
```

Il faut donc définir pour chaque lampe la couleur et l'intensité des trois composantes ambiante, diffuse et spéculaire.

Voici un exemple complet de définition de la lumière d'une lampe :

```
GLfloat bleu[4] = { 0.0, 0.0, 1.0, 1.0 };  
glLightfv(GL_LIGHT0, GL_AMBIENT, bleu);  
glLightfv(GL_LIGHT0, GL_DIFFUSE, bleu);  
glLightfv(GL_LIGHT0, GL_SPECULAR, bleu);
```

## Lampes directionnelles

Il s'agit d'une lumière qui vient de l'infini avec une direction particulière. La direction est spécifiée par un vecteur (x,y,z)

```
GLfloat direction[4];  
direction[0]=x; direction[1]=y; direction[2]=z;  
direction[3]=0.0; /* notez le zéro ici */  
glLightfv(GL_LIGHT0, GL_POSITION, direction);
```

## Lampes positionnelles

La lampe se situe dans la scène au point de coordonnées (x,y,z)

```
GLfloat position[4];  
position[0]=x; position[1]=y; position[2]=z;  
position[3]=1.0; /* notez le un ici */  
glLightfv(GL_LIGHT0, GL_POSITION, position);
```



## Modèle d'éclairage

Il faut indiquer avec `glLightModel*()` si les calculs d'éclairage se font de la même façon ou non sur l'envers et l'endroit des faces des objets.

Il faut également indiquer si OpenGL doit considérer pour ses calculs d'éclairage que l'œil est à l'infini ou dans la scène. Dans ce dernier cas, il faut calculer l'angle entre le point de vue et chacun des objets, alors que dans le premier cas, cet angle est ignoré ce qui est moins réaliste, mais moins coûteux en calculs. C'est le choix par défaut. Il est modifié par l'instruction

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE).
```

C'est avec cette même fonction que l'on définit la couleur de la lampe d'ambiance

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, couleur)
```

## Atténuation de la lumière

Dans le monde réel, l'intensité de la lumière décroît quand la distance à la source de lumière augmente. Par défaut le facteur d'atténuation d'OpenGL vaut un (pas d'atténuation), mais vous pouvez le modifier pour atténuer la lumière des lampes positionnelles.

La formule est :  $\text{facteur\_d\_atténuation} = 1.0 / (k_c + k_l * d + k_q * d * d)$ , où  $d$  est la distance entre la position de la lampe et le sommet.

$k_c = \text{GL\_CONSTANT\_ATTENUATION}$

$k_l = \text{GL\_LINEAR\_ATTENUATION}$

$k_q = \text{GL\_QUADRATIC\_ATTENUATION}$

Exemple de modification du coefficient d'atténuation linéaire pour la lampe 0 :

```
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 2.0);
```

## Lampes omnidirectionnelles et spots

Par défaut, une lampe illumine l'espace dans toutes les directions. L'autre type de lampe proposé par OpenGL est le spot. Un spot est caractérisé, en plus de sa position, par sa direction, le demi angle du cône de lumière et l'atténuation angulaire de la lumière.

La direction par défaut est  $\{0.0, 0.0, -1.0\}$ , c'est le demi-axe -z.

```
GLfloat direction[]={-1.0, -1.0, 0.0};
```

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0); /* ce spot éclairera jusqu'à 45°
```

```
autour de son axe */
```

```
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction);
```

```
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 0.5); /* coefficient d'atténuation angulaire
```

```
*/
```

## Couleur d'un matériau

Dans le modèle d'OpenGL, la couleur que l'on perçoit d'un objet dépend de la couleur propre de l'objet et de la couleur de la lumière qu'il reçoit.

Par exemple, un objet rouge renvoie toute la lumière rouge qu'il reçoit et absorbe toute la lumière verte et bleue qu'il reçoit.

- Si cet objet est éclairé par une lumière blanche (composé en quantités égales de rouge, vert et bleu), il ne renverra que la lumière rouge et apparaîtra donc rouge.
- Si cet objet est éclairé par une lumière verte, il apparaîtra noir, puisqu'il absorbe le vert et n'a pas de lumière rouge à réfléchir.

## Propriétés matérielles d'un objet

Les propriétés matérielles d'un objet sont celles qui ont été évoquées dans la partie Modèle d'éclairage OpenGL : la lumière émise, la réflexion ambiante, diffuse et spéculaire du matériau dont est fait l'objet.

Elles sont déterminées par des instructions :

`glMaterial*(GLenum face, GLenum pname, TYPE param)`

- Où `pname` vaut `GL_AMBIENT`, `GL_DIFFUSE`, `GL_AMBIENT_AND_DIFFUSE`, `GL_SPECULAR`, `GL_SHININESS`, ou `GL_EMISSION`.

## Combinaison des coefficients

- Pour une lampe, les coefficients RVB correspondent à un pourcentage de l'intensité totale pour chaque couleur. Par exemple  $R=1.0$ ,  $V=1.0$ ,  $B=1.0$  correspond au blanc de plus grande intensité, alors que  $R=0.5$ ,  $V=0.5$ ,  $B=0.5$  correspond à un blanc d'intensité moitié moins grande, qui est donc un gris.
- Pour un objet, les nombres correspondent à la proportion de la lumière renvoyée pour chaque couleur. Si une lampe qui a comme coefficients ( $LR$ ,  $LV$ ,  $LB$ ) éclaire un objet ( $OR$ ,  $OV$ ,  $OB$ ), la couleur perçue sera ( $LR*OR$ ,  $LV*OV$ ,  $LB*OB$ ).
- La combinaison de deux lampes de coefficients ( $R1$ ,  $V1$ ,  $B1$ ) et ( $R2$ ,  $V2$ ,  $B2$ ) produit une lumière ( $R1+R2$ ,  $V1+V2$ ,  $B1+B2$ ) (et les valeurs supérieures à 1 sont ramenées à 1).

## Listes d'affichage

Il s'agit d'un mécanisme pour stocker des commandes OpenGL pour une exécution ultérieure, qui est utile pour dessiner rapidement un même objet à différents endroits.

- Les instructions pour stocker les éléments d'une liste d'affichage sont regroupées entre une instruction `glNewList(GLuint numList, GLenum mode)` et une instruction `glEndList()`. Le paramètre `numList` est un numéro unique (index) qui est généré par `glGenLists(GLsizei range)` et qui identifie la liste.
- Le paramètre `mode` vaut `GL_COMPILE` ou `GL_COMPILE_AND_EXECUTE`. Dans le deuxième cas, les commandes sont non seulement compilées dans la liste d'affichage, mais aussi exécutées immédiatement pour obtenir un affichage.

- Les éléments stockés dans la liste d'affichage sont dessinés par l'instruction `glCallList(GLuint numList)`
- Une fois qu'une liste a été définie, il est impossible de la modifier à part en la détruisant et en la redéfinissant.

## Listes d'affichage hiérarchiques

Il est possible de créer une liste d'affichage qui exécute une autre liste d'affichage, en appelant `glCallList` à l'intérieur d'une paire `glNewList` et `glEndList()`.

Il y a un nombre maximal d'imbrications qui dépend des implémentations d'OpenGL.

## Création, suppression des listes d'affichage

L'instruction `glGenLists(GLsizei range)` génère `range` numéros de liste qui n'ont pas déjà été employés, mais il est également possible d'utiliser un numéro spécifique, en testant par `glIsList(GLuint numListe)`.

En effet, cette fonction renvoie `GL_TRUE` si la liste `numListe` existe déjà, et `GL_FALSE` sinon. L'instruction `glDeleteLists(GLuint list, GLsizei range)` permet de supprimer `range` listes en commençant par la numéro `list`. Une tentative de suppression d'une liste qui n'existe pas est simplement ignorée.

# Les Textures

## Introduction

L'instruction `glGenLists(GLsizei range)` génère `range` numéros de liste qui n'ont pas déjà été employés, mais il est également possible d'utiliser un numéro spécifique, en testant par `glIsList(GLuint numListe)`.

En effet, cette fonction renvoie `GL_TRUE` si la liste `numListe` existe déjà, et `GL_FALSE` sinon. L'instruction `glDeleteLists(GLuint list, GLsizei range)` permet de supprimer `range` listes en commençant par la numéro `list`. Une tentative de suppression d'une liste qui n'existe pas est simplement ignorée.

## Mécanisme général

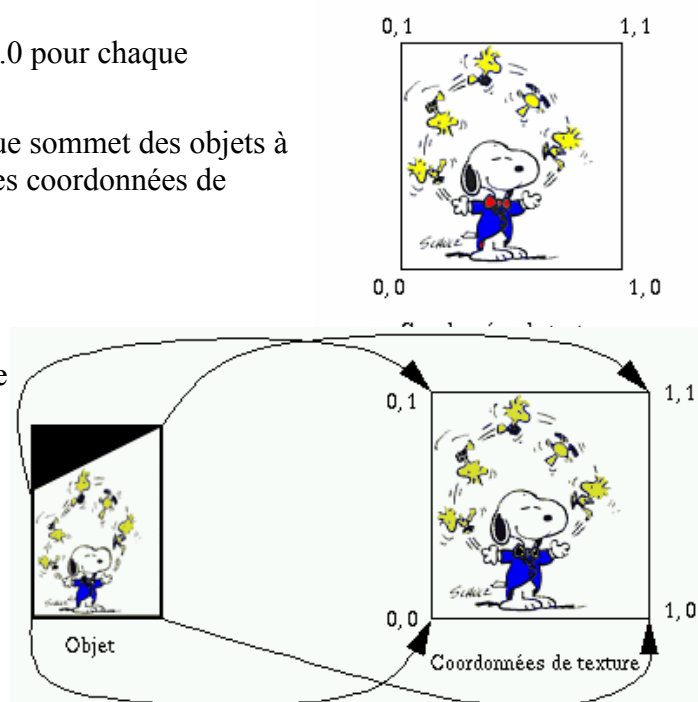
- Il s'agit d'abord de créer un Objet-Texture et de spécifier la texture que l'on va utiliser.
- Ensuite choisir le mode de placage de la texture avec `glTexEnv[f,i,fv,iv]()`
  - Le mode `decal` décalque la texture sur l'objet : la couleur de l'objet est remplacée par celle de la texture.
  - Le mode `modulate` utilise la valeur de la texture en modulant la couleur précédente de l'objet.
  - Le mode `blend` mélange la couleur de la texture et la couleur précédente de l'objet.
- Puis autoriser le placage de textures avec `glEnable(GL_TEXTURE_2D)` si la texture a deux dimensions. (ou `GL_TEXTURE_1D` si la texture est en 1D).
- Il est nécessaire de spécifier les Les coordonnées de texture en plus des coordonnées géométriques pour les objets à texturer.

## Coordonnées de texture

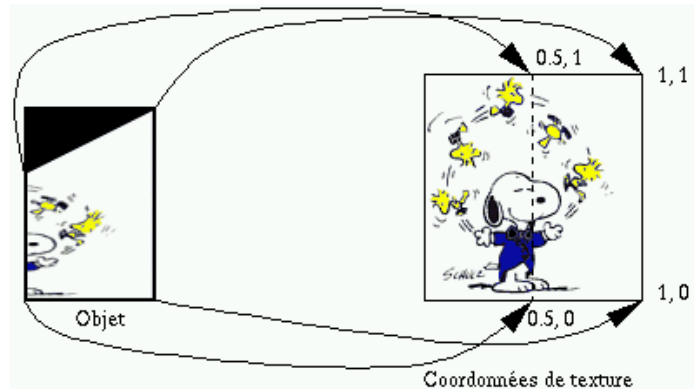
Les coordonnées de texture vont de 0.0 à 1.0 pour chaque dimension.

On assigne avec `glTexCoord*()` pour chaque sommet des objets à texturer un couple de valeurs qui indique les coordonnées de texture de ce sommet.

Si l'on désire que toute l'image soit affichée sur un objet de type quadrilatère, on assigne les valeurs de texture (0.0, 0.0) (1.0, 0.0) (1.0, 1.0) et (0.0, 1.0) aux coins du quadrilatère.



Si l'on désire mapper uniquement la moitié droite de l'image sur cet objet, on assigne les valeurs de texture (0.5, 0.0) (1.0, 0.0) (1.0, 1.0) et (0.5,1.0) aux coins du quadrilatère.



## Répétition de la texture

Il faut indiquer comment doivent être traitées les coordonnées de texture en dehors de l'intervalle [0.0, 1.0]. Est-ce que la texture est répétée pour recouvrir l'objet ou au contraire "clampée" ?

Pour ce faire, utilisez la commande `glTexParameter()` pour positionner les paramètres

`GL_TEXTURE_WRAP_S` pour la dimension horizontale de la texture ou  
`GL_TEXTURE_WRAP_T` pour la

dimension verticale à `GL_CLAMP` ou `GL_REPEAT`.

## Les Objets-Textures

Depuis la version 1.1 d'OpenGL, il est possible de déclarer, de nommer et de rappeler simplement des Objets-Textures.

- `glGenTextures(GLsizei n, GLuint *textureNames)` renvoie n noms (des numéros, en fait) de textures dans le tableau `textureNames[]`
- `glBindTexture(GL_TEXTURE_2D, GLuint textureNames)` a trois effets différents : o la première fois qu'il est appelé avec une valeur `textureNames` non nulle, un nouvel Objet-Texture est créé, et le nom `textureNames` lui est attribué.
  - avec une valeur `textureNames` déjà liée à une objet-Texture, cet objet devient actif
  - avec la valeur zéro, OpenGL arrête d'utiliser des objets-textures et retourne à la texture par défaut, qui elle n'a pas de nom.
- `glDeleteTextures(GLsizei n, const GLuint *textureNames)` efface n objets-textures nommés par les éléments du tableau `textureNames`

## Filtrage

Notez que les pixels de l'image de texture s'appellent des texels (au lieu de picture-elements, on a des texture-elements).

- Lorsque la partie de l'image de texture qui est mappée sur un pixel d'un objet est plus petite qu'un texel, il doit y avoir agrandissement.
- Lorsqu'au contraire la partie de l'image de texture qui est mappée sur un pixel d'un objet contient plus d'un texel, il doit y avoir une réduction.

La commande `glTexParameter()` permet de spécifier les méthodes d'agrandissement (`GL_TEXTURE_MAG_FILTER`) et de réduction (`GL_TEXTURE_MIN_FILTER`) utilisées :

- `GL_NEAREST` choisit le texel le plus proche
- `GL_LINEAR` calcule une moyenne sur les 2x2 texels les plus proches.

L'interpolation produit des images plus lisses, mais prend plus de temps à calculer. Le compromis valable pour chaque application doit être choisi.

## ***Les niveaux de détail***

Les objets texturés sont visualisés comme tous les objets de la scène à différentes distances de la caméra. Plus les objets sont petits, moins il est nécessaire d'utiliser une taille importante pour l'image de texture utilisée. En particulier, l'utilisation d'images de tailles adaptées à la taille de l'objet peut accélérer le rendu de l'image et éviter certains artéfacts visuels lors d'animations.

- OpenGL utilise une technique de mipmaping pour utiliser des images de taille appropriée. Les images du quart, du seizième, du soixante-quatrième, etc de la taille de l'image initiale sont stockées, jusqu'à l'image d'un pixel de côté, et en fonction de la taille de l'objet dans la scène, la texture de la taille adaptée est choisie.
- Vous pouvez fournir vous-mêmes les images de texture aux différentes tailles avec plusieurs appels à `glTexImage2D()` à différentes résolutions, ou bien utiliser une routine de l'OpenGL Utility Library : `gluBuild2DMipmaps()`, qui calcule les images réduites.

## **Filtrage**

Lors d'utilisation de mipmaps, il y a quatre autres filtres de réduction :

- `GL_NEAREST_MIPMAP_NEAREST` et `GL_LINEAR_MIPMAP_NEAREST` utilisent l'image mipmap la plus proche, et dans cette image, le premier choisit le texel le plus proche, et le deuxième réalise une interpolation sur les quatre texels les plus proches.
- `GL_NEAREST_MIPMAP_LINEAR` et `GL_LINEAR_MIPMAP_LINEAR` travaillent à partir de l'interpolation de 2 mipmaps.

## **Lecture d'une image de texture dans un fichier**

Le fichier `loadppm.c` permet de lire une image au format PPM (avec les données en binaire et éventuellement des lignes de commentaires). `glTexImage2D()` que vous trouvez dans l'exemple `texture1.c` permet de spécifier quelle image vous utilisez comme texture. Sa limitation est de nécessiter une image dont la hauteur et la largeur sont des puissances (éventuellement différentes) de 2 : votre image peut avoir comme taille 256x64, mais pas 100x100.

- La routine `gluBuild2DMipmaps()`, par contre accepte les images de toutes tailles.

## Tableaux de sommets

La motivation pour l'utilisation de tableaux de sommets est double : il s'agit d'une part de réduire le nombre d'appels de fonctions, et d'autre part d'éviter la description redondante de sommets partagés par des polygones adjacents.

- Par exemple un cube est formé de 6 faces et 8 sommets. Si les faces sont décrites indépendamment, chaque sommet est traité 3 fois, une fois pour chaque face à laquelle il appartient.
- Les tableaux de sommets sont standards depuis la version 1.1 d'OpenGL.

### **Mécanisme général**

- 1. Il s'agit d'abord d'activer jusqu'à six tableaux, stockant chacun un des six types de données suivantes :
  - Coordonnées des sommets
  - Couleurs RVBA
  - Index de couleurs
  - Normales de surfaces
  - Coordonnées de texture
  - Indicateurs de contour des polygones
- 2. Ensuite spécifier les données du ou des tableaux.
- 3. Puis dé-référencer le contenu des tableaux ( accéder aux éléments ) pour tracer une forme géométrique avec les données. Il y a trois méthodes différentes pour le faire :
  - Accéder aux éléments du tableau un par un : accès aléatoire
  - Créer une liste d'éléments du tableau : accès méthodique
  - Traiter les éléments du tableau de manière séquentielle : accès séquentiel ou systématique

### **Activer les tableaux**

Cela se fait en appelant `glEnableClientState(GLenum array)` avec comme paramètre : `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, ou `GL_EDGE_FLAG_ARRAY`.

La désactivation d'un état se fait en appelant `glDisableClientState(GLenum array)` avec les mêmes paramètres.

### **Spécifier les données de tableaux**

Il faut indiquer l'emplacement où se trouvent les données et la manière dont elles sont organisées.

Les différents types de données (coordonnées des sommets, couleurs, normales, ...) peuvent avoir été placées dans différentes tables, ou être entrelacées dans une même table.

## Fonctions de spécification des tableaux

Six routines permettent de spécifier des tableaux en fonction de leur type :

- void glVertexPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid \*pointeur);
- void glColorPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid \*pointeur);
- void glIndexPointer(GLenum type, GLsizei stride, const GLvoid \*pointeur);
- void glNormalPointer(GLenum type, GLsizei stride, const GLvoid \*pointeur);
- void glTexCoordPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid \*pointeur);
- void glEdgeFlagPointer(GLsizei stride, const GLvoid \*pointeur);

Le paramètre pointeur indique l'adresse mémoire de la première valeur pour le premier sommet du tableau.

Le paramètre type indique le type de données

Le paramètre taille indique le nombre de valeurs par sommets, et peut prendre suivant les fonctions les valeurs 1, 2, 3 ou 4.

Le paramètre stride indique le décalage en octets entre deux sommets successifs. Si les sommets sont consécutifs, il vaut zéro.

N.B. Il existe également une routine glInterleavedArrays(GLenum format, GLsizei stride, const GLvoid \*pointeur); qui permet de spécifier plusieurs tableaux de sommets en une seule opération, et d'activer et désactiver les tableaux appropriés. Les données doivent être stockées dans une table suivant un des 14 modes d'entrelacement prévus.

### Exemple 1 : données dans différentes tables

```
static GLint sommets[]={25, 25,
100, 325,
175, 25,
175, 325,
250, 25};
static GLfloat couleurs[]={1.0, 0.2, 0.2,
0.2, 0.2, 1.0,
0.8, 1.0, 0.2,
0.5, 1.0, 0.5,
0.8, 0.8, 0.8};
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(2, GL_INT, 0, sommets);
glColorPointer(3, GL_FLOAT, 0, couleurs);
```

### Exemple 2 : données entrelacées

```
static GLfloat sommetsEtcouleurs[]={ 25.0, 25.0, 1.0, 0.2, 0.2,
100.0, 325.0, 0.2, 0.2, 1.0,
175.0, 25.0, 0.8, 1.0, 0.2,
175.0, 325.0, 0.5, 1.0, 0.5,
250.0, 25.0, 0.8, 0.8, 0.8};
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(2, GL_FLOAT, 5*sizeof(GLfloat), &sommetsEtcouleurs[0]);
glColorPointer(3, GL_FLOAT, 5*sizeof(GLfloat), &sommetsEtcouleurs[2]);
```



## Dé-référencer le contenu des tableaux

L'accès aux éléments des tableaux peut se faire pour un élément unique (accès aléatoire), ou pour une liste ordonnée d'éléments (accès méthodique), ou pour une séquence d'éléments (accès séquentiel).

### Accès à un élément unique

La méthode void glVertexElement(GLint indice) trouve les données du sommet d'indice indice pour tous les tableaux activés. Cette méthode est généralement appelée entre glBegin() et glEnd().

Exemple d'accès à un élément des tableaux construits dans l'exemple 1 :

```
glBegin(GL_LINES);  
    glVertexElement(1);  
    glVertexElement(4);  
glEnd();
```

ces lignes ont le même effet que

```
glBegin(GL_LINES);  
    glColor3fv(couleurs + (1*3));  
    glVertex2iv(sommets + (1*2));  
    glColor3fv(couleurs + (4*3));  
    glVertex2iv(sommets + (4*2));  
glEnd();
```

### Accès à une liste d'éléments

La méthode void glDrawElements(GLenum mode, GLsizei nombre, GLenum type, void \*indices) permet d'utiliser le tableau indices pour stocker les indices des éléments à afficher.

Le nombre d'éléments dans le tableau d'indices est nombre Le type de données du tableau d'indices est type, qui doit être GL\_UNSIGNED\_BYTE, GL\_UNSIGNED\_SHORT, ou GL\_UNSIGNED\_INT

Le type de primitive géométrique est indiqué par mode de la même manière que dans glBegin().

- glVertexElements() ne doit pas être encapsulé dans une paire glBegin()/ glEnd().
- glVertexElements() vérifie que les paramètres mode, nombre, et type sont valides, et effectue ensuite un traitement semblable à la séquence de commandes :

```
glBegin(mode);  
    for (i = 0, i < nombre ; i++)  
        glVertexElement(indices[i]);  
glEnd();
```

Exemple d'accès à un ensemble d'éléments des tableaux construits dans l'exemple 1 :

```
static GLubyte mesIndices[] = {1, 4};  
glDrawElements(GL_LINES, 2, GL_UNSIGNED_BYTE, mesIndices);
```

## Accès à une séquence d'éléments

La méthode `void glDrawArrays(GLenum mode, GLint premier, GLsizei nombre)` construit une séquence de primitives géométriques contenant les éléments des tableaux activés de `premier` à `premier + nombre - 1`. Le type de primitive géométrique est indiqué par `mode` de la même manière que dans `glBegin()`.

L'effet de `glDrawArrays()` est semblable à celui de la séquence de commandes :

```
int i;
glBegin(mode);
for (i=0 ; i < nombre ; i++)
    glArrayElement(premier + i);
glEnd();
```

## Mélange de couleurs : blending

La valeur alpha (le A de RGBA) correspond à l'opacité d'un fragment. Comment intervient-elle dans la composition des couleurs ? Elle est utilisée avec `glColor()`, et avec `glClearColor()` pour spécifier une couleur de vidage, et pour spécifier les propriétés matérielles d'un objet ou l'intensité d'une source de lumière.

En l'absence de blending, chaque nouveau fragment écrase les valeurs chromatiques antérieures de la mémoire tampon, comme si le fragment était opaque. Lorsque le mélange des couleurs (blending) est activé, la valeur alpha est utilisée pour combiner la couleur du fragment en cours de traitement avec la valeur du pixel déjà stocké dans la mémoire tampon.

Le mélange des couleurs intervient après que la scène a été rasterisée et convertie en fragments, et avant que les pixels définitifs aient été stockés dans la mémoire tampon.

### **Activer/désactiver le blending**

L'activation est nécessaire pour tous les calculs de mélange de couleurs, elle se fait par `glEnable(GL_BLEND)`. Pour désactiver le blending, utilisez `glDisable(GL_BLEND)`.

### **Facteurs de blending source et destination**

#### **Définition**

Au cours du blending, les valeurs chromatiques du fragment en cours de traitement (la source) sont combinées avec les valeurs du pixel déjà stocké dans la mémoire tampon correspondant (la destination).

Pour cela, un coefficient appelé facteur de blending source est appliqué à la valeur de la source, et un coefficient appelé facteur de blending destination est appliqué à la valeur de la source. Ces deux valeurs sont ensuite ajoutées pour composer la nouvelle valeur du pixel.

Les facteurs de blending sont des quadruplets RVBA, que l'on notera ( $S_r, S_v, S_b, S_a$ ) pour le facteur de blending source, et ( $D_r, D_v, D_b, D_a$ ) pour le facteur de blending destination.

Si l'on note ( $R_s, V_s, B_s, A_s$ ) la couleur de la source, et ( $R_d, V_d, B_d, A_d$ ) la couleur de la destination, la nouvelle valeur du pixel sera :

$(R_s.S_r + R_d.D_r, V_s.S_v + V_d.D_v, B_s.S_b + B_d.D_b, A_s.S_a + A_d.D_a)$

Chaque composant de ce quadruplé est finalement arrondi à l'intervalle  $[0, 1]$ .

### **Spécifier les facteurs**

La fonction `glBlendFunc(GLenum facteur_source, GLenum facteur_destination)` est utilisée pour spécifier les facteurs de blending source et destination. Ses deux paramètres sont des constantes dont le tableau suivant indique 11 des 15 valeurs possibles : (les 4 autres sont utilisées avec le sous-ensemble de traitement d'images).

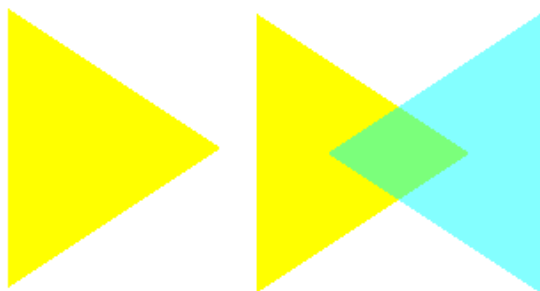
Constante	Facteur concerné		Valeur du facteur de blending
GL_ZERO	Source	destination	(0, 0, 0, 0)
GL_ONE	Source	destination	(1, 1, 1, 1)
GL_DST_COLOR	source		( <b>Rd</b> , <b>Vd</b> , <b>Bd</b> , <b>Ad</b> )
GL_SRC_COLOR		destination	( <b>Rs</b> , <b>Vs</b> , <b>Bs</b> , <b>As</b> )
GL_ONE_MINUS_DST_COLOR	source		(1, 1, 1, 1) - ( <b>Rd</b> , <b>Vd</b> , <b>Bd</b> , <b>Ad</b> )
GL_ONE_MINUS_SRC_COLOR		destination	(1, 1, 1, 1) - ( <b>Rs</b> , <b>Vs</b> , <b>Bs</b> , <b>As</b> )
GL_SRC_ALPHA	Source	destination	( <b>As</b> , <b>As</b> , <b>As</b> , <b>As</b> )
GL_ONE_MINUS_SRC_ALPHA	Source	destination	(1, 1, 1, 1) - ( <b>As</b> , <b>As</b> , <b>As</b> , <b>As</b> )
GL_DST_ALPHA	Source	destination	( <b>Ad</b> , <b>Ad</b> , <b>Ad</b> , <b>Ad</b> )
GL_ONE_MINUS_DST_ALPHA	Source	destination	(1, 1, 1, 1) - ( <b>Ad</b> , <b>Ad</b> , <b>Ad</b> , <b>Ad</b> )
GL_SRC_ALPHA_SATURATE	source		(f, f, f, 1) avec $f = \min(\mathbf{As}, 1 - \mathbf{Ad})$

## Exemples

### Mélange homogène de deux objets

Pour dessiner une image composée de deux objets à parts égales, on peut procéder comme suit :

- Positionner le facteur de blending source à GL\_ONE et le facteur de destination à GL\_ZERO : `glBlendfunc(GL_ONE, GL_ZERO);`
- Dessiner le premier objet (le triangle jaune ici)
- Positionner le facteur de blending source à GL\_SRC\_ALPHA et le facteur de destination à GL\_ONE\_MINUS\_SRC\_ALPHA : `glBlendfunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
- Dessiner le deuxième objet avec un alpha de 0.5 (le triangle cyan ici)

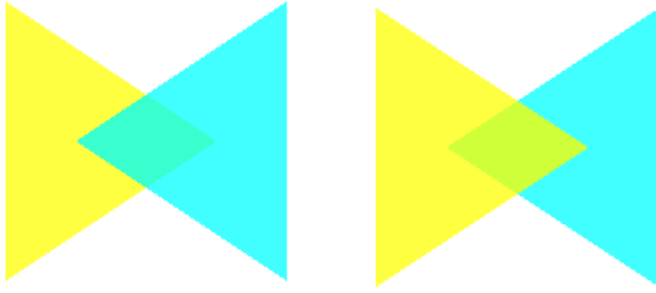


### Importance de l'ordre d'affichage

Dans l'exemple suivant,

- le facteur de blending source est positionné à GL\_SRC\_ALPHA et le facteur de destination à GL\_ONE\_MINUS\_SRC\_ALPHA : `glBlendfunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`

- chacun des deux objets a un paramètre  $\alpha = 0.75$



A gauche, l'image est obtenue en dessinant d'abord le triangle jaune, puis le cyan. A droite, l'ordre d'affichage est inversé.

### ***Du bon usage du tampon de profondeur***

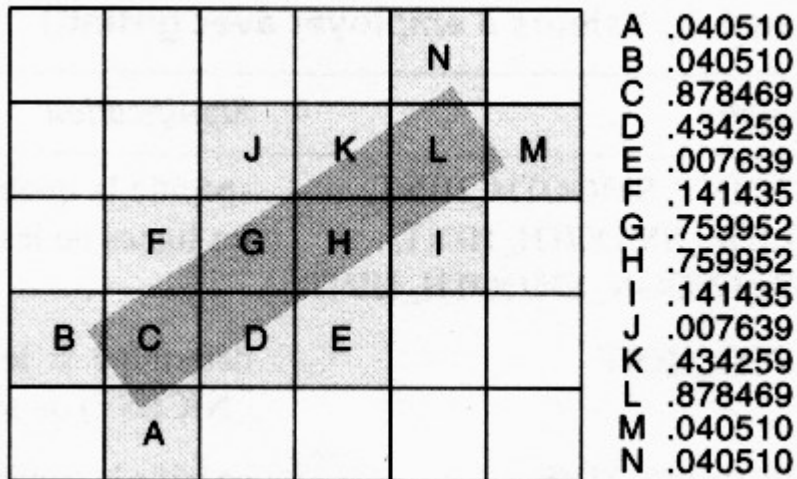
Le tampon de profondeur stocke la distance entre le point de vue et la portion de l'objet occupant un pixel donné. Lorsqu'un autre objet doit s'ajouter au même pixel, il ne sera dessiné que s'il est plus proche du point de vue, et dans ce cas, c'est sa distance qui sera stockée dans le tampon de profondeur.

Si une même scène contient à la fois des objets opaques et des objets translucides, il y a un problème si on laisse les objets translucides masquer des objets opaques. Il faut donc d'abord dessiner les objets opaques, en activant le tampon de profondeur en lecture / écriture (par `glDepthMask(GL_TRUE)` qui correspond à son fonctionnement par défaut ) pour le mettre à jour au fur et à mesure de l'ajout des objets. Il faut ensuite activer le tampon de profondeur en lecture seule (par `glDepthMask(GL_FALSE)`) pour ajouter les objets translucides seulement s'ils ne sont pas cachés par des objets opaques.

# Lissage (antialiasing), fog et décalage de polygones

## Définition et principe

Les lignes qui ne sont ni horizontales ni verticales sont crénelées. Ce phénomène est appelé aliasing, et nous allons voir comment la technique de lissage (ou antialiasing) permet de l'estomper.

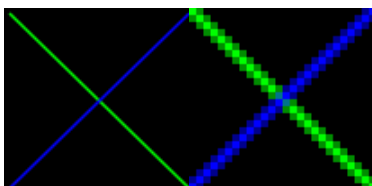


Si l'on agrandit un segment de droite comme sur l'image ci-contre, on s'aperçoit que chaque pixel est plus ou moins recouvert par le segment.

L'affichage des pixels traversés par le segment génère une ligne crénelée.

La technique de lissage consiste à affecter à chaque pixel traversé par le segment une opacité dépendant du taux de couverture.

## Exemple



Ainsi sur l'exemple à gauche, les lignes verte et bleues sont lissées, et l'on voit sur l'agrandissement du croisement entre les deux segments (à droite), que les intensités des pixels sont variables.

## Contrôle de qualité

La fonction `glHint(GLenum cible, GLenum hint)` permet d'indiquer les préférences que l'on a entre qualité de l'image et rapidité de l'affichage. Néanmoins, toutes les implémentations ne tiennent pas compte de ces préférences.

Le paramètre cible peut prendre les valeurs suivantes :

GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT	qualité d'échantillonnage souhaitée pour les points, les lignes ou les polygones au cours des opérations de lissage
GL_FOG_HINT	détermine si les calculs de fog se font pixel par pixel

	pixel (meilleure qualité) ou sommet par sommet (plus rapide)
GL_PERSPECTIVE_CORRECTION_HINT	spécifie si l'on souhaite ou non tenir compte de la perspective pour l'interpolation des couleurs et des textures.

Le paramètre hint peut prendre la valeur GL\_FASTEST, pour privilégier la vitesse, ou GL\_NICEST pour privilégier la qualité, ou GL\_DONT\_CARE pour indiquer l'absence de préférence.

## ***Fog, le brouillard***

Le terme "Fog" désigne le brouillard et les effets atmosphériques du même type : brume, buée, fumée, pollution. C'est un effet qui permet de simuler une visibilité limitée. Lorsque le fog est activé, les objets se mélangent avec la couleur du fog en fonction de leur distance au point de vue.

Comme par temps de brouillard, les objets les plus lointains sont les moins visibles.

## **Mise en œuvre**

Il suffit d'activer le fog à l'aide de l'appel à glEnable(GL\_FOG), et de choisir la couleur du fog, et l'équation qui va contrôler la densité à l'aide d'appels à glFog\*(GLenum pname, TYPE param) et à glFog\*v(GLenum pname, TYPE \*params).

## **Couleur et équation du fog**

Soit Cf la couleur du fog. Cette couleur est définie par un appel à glFogfv(GL\_FOG\_COLOR, params), où params est un tableau de 4 nombres réels de type GLfloat, qui spécifient les valeurs chromatiques RVBA du fog.

Soit Cs la couleur d'un fragment entrant. On parle aussi de fragment source, c'est un fragment d'un objet qui va être soumis au fog.

La couleur finale C sera calculée en utilisant une fonction de mélange f(z) qui est une fonction décroissante de la distance z entre le point de vue et le centre du fragment :

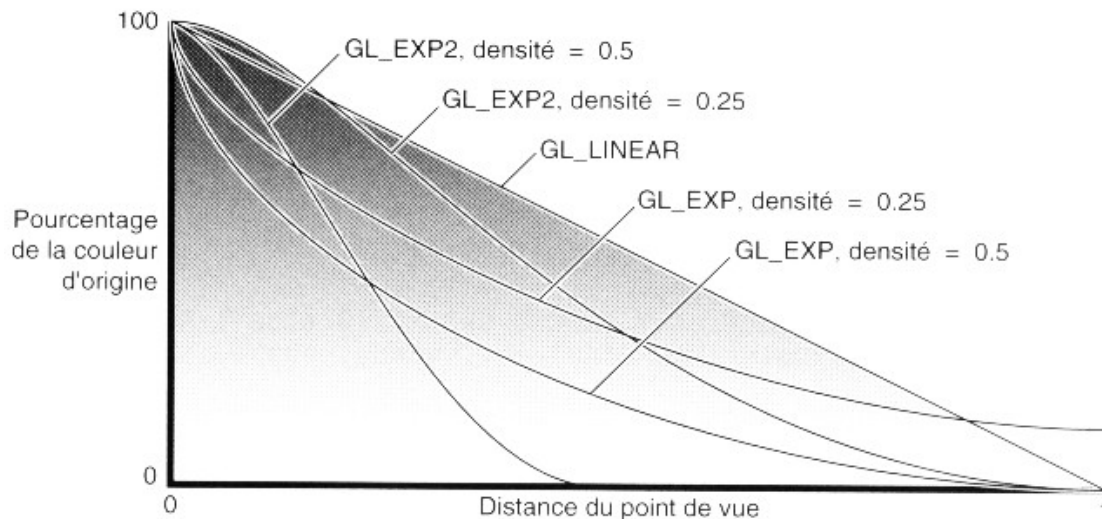
$$C = f(z).Cs + (1 - f(z)).Cf$$

L'équation de cette fonction f dépend du mode de fog choisi. Elle est soit exponentielle, soit exponentielle au carré, soit linéaire.

Si l'on choisit une décroissance exponentielle ou exponentielle au carré, on doit le déclarer par un appel à glFogi(GL\_FOG\_MODE, GL\_EXP) (c'est le mode par défaut), ou glFogi(GL\_FOG\_MODE, GL\_EXP2), et spécifier la densité du fog à l'aide de glFogf(GL\_FOG\_DENSITY, densite). La valeur par défaut de la densité est 1.

On a alors  $f = \exp -(densite.z)$  pour la décroissance exponentielle, ou  $f = \exp -(densite.z)^2$  pour une décroissance exponentielle au carré.

Si l'on choisit une fonction linéaire, l'équation est  $f = (end - z) / (end - start)$ , et on doit indiquer ce choix à l'aide de glFogi(GL\_FOG\_MODE, GL\_LINEAR). Les paramètres end et start sont spécifiés par glFogf(GL\_FOG\_END, end) et glFogf(GL\_FOG\_START, start). Par défaut, start vaut 0 et end vaut 1.



La figure ci-contre représente le comportement des équations de densité en fonction de la distance au point de vue

## Décalage de polygone

### Définition

Pour souligner les contours d'un objet solide, on peut être amené à le dessiner deux fois, une fois en remplissant les faces (mode `GL_FILL`), et une fois en redessinant les lignes (mode `GL_LINE`) d'une autre couleur.

Mais la rasterisation des lignes et des polygones pleins ne se fait pas de la même manière, et le résultat contient un effet désagréable de chevauchement (stitching), que l'on va supprimer avec le décalage de polygone.

Il s'agit d'ajouter un décalage (offset) en z aux lignes pour les rapprocher de l'observateur, afin de les séparer des polygones.

Cet effet de décalage est aussi utilisé pour rendre des images avec suppression des lignes cachées.

## Modes de rendu des polygones

On peut visualiser les polygones en mode point ( `glPolygonMode(GL_POINT)` ), en mode ligne ( `glPolygonMode(GL_LINE)` ), ou en mode plein ( `glPolygonMode(GL_FILL)` ).

## Types de décalage

Il y a trois types de décalages, un par mode de rendu. On peut décaler les points (`glEnable(GL_POLYGON_OFFSET_POINT)`), les lignes (`glEnable(GL_POLYGON_OFFSET_LINE)`), ou les faces pleines (`glEnable(GL_POLYGON_OFFSET_FILL)`).



## Valeur du décalage

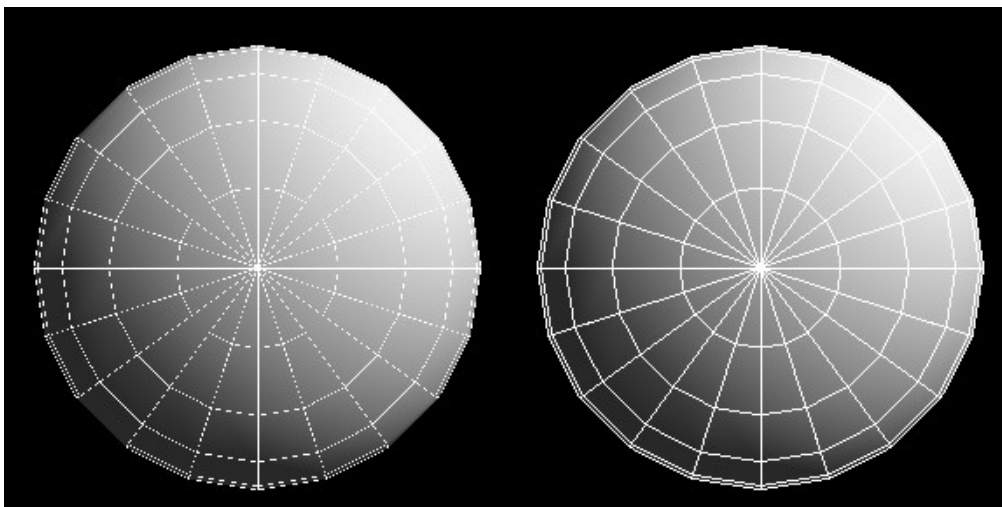
La commande `glPolygonOffset(GLfloat facteur, GLfloat unites)` ajoute à la profondeur de chaque fragment un décalage  $o$  :

$$o = m \cdot \text{facteur} + r \cdot \text{unites}$$

Dans cette formule,  $m$  est l'inclinaison de profondeur maximale du polygone, ( c'est à dire la variation en  $z$  du polygone divisée par la variation en  $x$  ou  $y$  correspondante) et  $r$  une constante spécifique à l'implémentation.

En général, on obtient un décalage satisfaisant en utilisant les valeurs suivantes : `facteur = unites = 1.0` . On note que  $m$  et  $r$  étant positifs, avec ces valeurs, le décalage  $o$  est positif : les objets sont bien rapprochés du point de vue.

## Exemple



l'image de gauche, sans décalage des lignes, dans l'image de droite, avec :

# Les tampons d'image

## ***Introduction***

Après la rasterisation (y compris le texturage et le fog), les données sont des fragments qui contiennent des coordonnées correspondant à un pixel, ainsi que des valeurs chromatiques et de profondeur.

Ces fragments subissent une série de tests ( le test de scission, le test alpha, le test de stencil et de profondeur) et d'opérations (logiques et de fondu) avant de devenir des pixels.

## **Définitions**

- Une zone mémoire permettant de stocker une certaine quantité de données fixe par pixel est appelée tampon (buffer).
- Un tampon stockant un seul bit par pixel est appelé bitplan.
- Le tampon servant à stocker la couleur de chaque pixel est appelé Tampon chromatique ou Frame-Buffer.
- Le tampon servant à stocker la profondeur de chaque pixel est appelé Tampon de profondeur ou Zbuffer.
- Le tampon servant à restreindre le rendu à certaines portions de l'écran est appelé Tampon Stencil ou Stencil-Buffer.
- Le tampon servant à accumuler une série d'images dans une image finale est appelé Tampon d'Accumulation ou Accumulation-Buffer.

Le Tampon d'Image est le système OpenGL qui contient tous les tampons :

- les tampons chromatiques : (selon l'implémentation) avant-gauche, avant-droit, arrière-gauche, arrière-droit et un nombre quelconque de tampons chromatiques auxiliaires
- le tampon de profondeur
- le tampon stencil
- le tampon d'accumulation

## ***Les tampons et leur utilisation***

### **Tampons chromatiques**

Les tampons chromatiques contiennent des données en mode couleur indexée ou RVB, et peuvent également accueillir des valeurs alpha.

#### *Implémentations*

Une implémentation OpenGL supportant la stéréo comprend des tampons chromatiques gauche et droit. Si la stéréo n'est pas prise en charge, seuls les tampons de gauche sont utilisés.

De même, si le double-buffering est supporté, le système comprend des tampons avant et arrière. En simple buffering, seuls les tampons avant sont utilisés.

#### *Tampon d'accumulation*

Il contient des données chromatiques RVBA. Le résultat de l'utilisation du tampon d'accumulation en mode couleurs indexées n'est pas défini. On ne dessine pas directement dans le tampon d'accumulation, les opérations d'accumulation se font par des transferts de données depuis ou vers un tampon chromatique.

### **Sélectionner les tampons chromatiques en écriture et en lecture**

Les opérations de lecture peuvent s'effectuer depuis n'importe quel tampon chromatique. Celui-ci sera spécifié par `glReadBuffer()`.

De même, le résultat des opérations de dessin ou d'écriture peut s'effectuer vers n'importe quel tampon chromatique, spécifié par `glDrawBuffer()`.

### **Le tampon d'accumulation**

Son utilisation est la suivante : une série d'images, générées chacune dans l'un des tampons chromatiques, sont accumulées une à une dans le tampon d'accumulation. Lorsque l'accumulation est terminée, le résultat est copié dans un tampon chromatique pour affichage.

Pour réduire les erreurs d'arrondis, le tampon d'accumulation peut disposer d'une précision plus élevée (plus de bits par pixels par exemple) que les tampons chromatiques standards.

Plusieurs restitutions d'une scène prennent plus de temps qu'une seule, mais le résultat est de meilleure qualité.

`void glAccum(GLenum op, GLfloat value);` contrôle le tampon d'accumulation. Le paramètre `op` sélectionne l'opération, et peut avoir pour valeur `GL_ACCUM`, `GL_LOAD`, `GL_RETURN`, `GL_ADD` ou `GL_MULT`.

`value` est un nombre à utiliser dans cette opération.

- `GL_ACCUM` lit chaque pixel du tampon sélectionné avec `glReadBuffer()`, multiplie les valeurs RVBA par `value` et ajoute les valeurs obtenues au tampon d'accumulation.
- `GL_LOAD` est semblable à `GL_ACCUM`, mais les valeurs calculées remplacent le contenu du tampon d'accumulation.
- `GL_RETURN` prend les valeurs du tampon d'accumulation, les multiplie par `value`, et place le résultat dans le ou les tampons chromatiques activés en écriture.
- `GL_ADD` ajoute (resp. `GL_MULT` multiplie ) `value` à (resp. par ) la valeur de chaque pixel du tampon d'accumulation et retourne le résultat dans le tampon d'accumulation. Pour `GL_MULT`, `value` est arrondi à l'intervalle  $[-1.0, 1.0]$ , `GL_ADD` il n'est pas arrondi.

### **Lisser une scène**

Pour effectuer l'antialiasing d'une scène, le principe est d'accumuler `n` versions de la même scène, légèrement décalées :

- Commencer par vider le tampon d'accumulation
- Accumuler chaque image par : `glAccum(GL_ACCUM, 1.0/n);`
- Afficher l'image finale par `glAccum(GL_RETURN, 1.0);`

Noter que cette méthode est un peu plus rapide si la première image est chargée dans le tampon d'accumulation sans le vider, en utilisant `GL_LOAD` à la place de `GL_ACCUM`.

Pour éviter que les `n` images intermédiaires de la scène ne soient affichées, dessinez-les dans un tampon chromatique qui n'est pas affiché.

## ***Jittering***

Le procédé selon lequel on applique un léger décalage aux versions intermédiaire la scène s'appelle jittering.

Le décalage doit avoir une valeur de moins d'un pixel en x et y par rapport à l'image de référence.

Les routines `accPerspective()` et `accFrustum()` de l'exemple `accpersp.c` peuvent être employées à la place de `gluPerspective()` et `glFrustum()`.

Si vous avez choisi une perspective cavalière, comme dans l'exemple `accanti.c`, utilisez `glTranslate*()` pour décaler la scène, en vous rappelant que le décalage doit être inférieur à 1 pixel mesuré en coordonnées écran.

Le fichier `jitter.h` donne des valeurs de décalages utilisables pour  $n=2,3,4,8,15,24$  et  $66$ .

## ***Fondu enchaîné***

Pour créer un effet de fondu-enchaîné, ou une impression de mouvement, copiez la scène obtenue au temps précédent dans le tampon d'accumulation, et appelez `glAccum(GL_MULT, coeffAttenuation)`; avec un coefficient compris strictement entre 0.0 et 1.0, afin d'atténuer l'image. Calculez ensuite la nouvelle scène,

## ***Profondeur de champ***

La mise au point d'une photographie n'est parfaite que pour les éléments se trouvant à une certaine de l'appareil photographique. Pour simuler cet effet avec OpenGL, le tampon d'accumulation va servir à accumuler différentes images dans lesquelles seul un plan de mise au point parfaite va rester stable.

## ***Masquer les tampons***

Avant qu'OpenGL n'écrive des données dans les tampons chromatique, de profondeur ou stencil actifs, une opération de masquage est appliquée aux données. Un ET logique est effectué entre le masque et les données à écrire.

Les fonctions correspondant sont `glIndexMask`, `glColorMask`, `glDepthMask`, et `glStencilMask`.

En particulier, pour désactiver le tampon de profondeur en écriture, ce qui est utile lors de l'affichage d'objets translucides, appeler `glDepthMask(false)`.

## ***Vider les tampons***

Selon la taille du tampon, le vider peut constituer une des opérations les plus longues de l'application graphique. Certaines implémentations d'OpenGL permettent de vider plusieurs tampons en même temps.

Pour en bénéficier, il faut commencer par spécifier les valeurs à écrire dans chaque tampon à vider, puis effectuer le vidage en appelant la fonction `glClear()` à laquelle on passe la liste de tous les tampons à vider.

Pour définir les valeurs de vidage des tampons, utiliser les commandes suivantes :

- `void glClearColor( GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha );`

- void glClearIndex( GLfloat index );
- void glClearDepth( GLclampd depth );
- void glClearStencil( GLint s);
- void glClearAccum( GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha );

Ensuite, pour procéder au vidage, utiliser : glClear( GLbitfield mask ); avec comme valeur demask, un OU logique parmi GL\_COLOR\_BUFFER\_BIT, GL\_DEPTH\_BUFFER\_BIT, GL\_STENCIL\_BUFFER\_BIT et GL\_ACCUM\_BUFFER\_BIT.

Lors du vidage du tampon chromatique, tous les tampons chromatiques activés en écriture sont vidés.

Le test d'appartenance du pixel, le test de scission et le dithering sont appliqués à l'opération de vidage, s'ils sont actifs. Les opérations de masquage, telles que glColorMask() et glIndexMask() sont également opérationnelles.

## Sélection et picking

Dans les modes de calcul de la scène qui s'appellent Sélection et Feed-Back, les informations de dessin sont retournées à l'application, au lieu d'être envoyées dans la mémoire tampon pour y créer une image comme dans le mode normal qui s'appelle Restitution (ou Rendu).

Le mode de Sélection permet de récupérer pour chaque objet qui est dans le volume de vision une information simple, appelée hit, accompagnée du nom qu'on aura donné à cet objet.

Le mode de Feed-Back permet de récupérer en plus les coordonnées, couleur, et coordonnées de texture des objets du volume de vision.

Ces deux modes traitent tous les objets du volume de vision. Or leur utilisation la plus courante est de cliquer à la souris sur un objet pour le choisir.

C'est le mécanisme de picking, qui consiste à restreindre le dessin à une région réduite du cadrage, généralement autour du pointeur de la souris, qui va permettre de retrouver par Sélection ou Feed-Back uniquement les objets qui sont "sous" le pointeur de la souris.

### Sélection

Lorsque la scène est dessinée en mode Sélection, le contenu de la mémoire tampon n'est pas modifié. Lorsqu'on quitte le mode Sélection, OpenGL retourne la liste des primitives qui intersectent le volume de vision. Ces primitives sont caractérisées par des noms, qu'il faut associer aux objets lors de la description de la scène.

### Principales étapes

- 1. Spécifier à l'aide de `glSelectBuffer(...)` le tableau qui va recevoir les enregistrements de hits.
- 2. Entrer en mode Sélection, par `glRenderMode(GL_SELECT)`
- 3. Initialiser la pile des noms par `glInitNames()` et `glPushName(...)`.
- 4. Définir le volume de vision concerné par la sélection, qui peut être différent du volume de vision pour le rendu de la scène.
- 5. Dessiner la scène, en ajoutant les commandes pour nommer les primitives significatives.
- 6. Quitter le mode Sélection et traiter les données de sélection retournées (les enregistrements de hits).

### Détails de ces étapes

1. Spécifier le tableau qui va recevoir les enregistrements de hits  
`glSelectBuffer(GLsizei taille, GLuint * tab)` indique que l'on veut récupérer les données de sélection dans le tableau `tab` d'entiers non signés, qui peut contenir jusqu'à `taille` valeurs. Ce tableau doit être alloué au préalable.
2. Entrer en mode Sélection  
L'appel à `glRenderMode(GL_SELECT)` permet d'entrer en mode Sélection.
3. Initialiser la pile des noms  
Les noms sont des entiers non signés que l'on associe à des primitives. La structure de

données permettant de nommer les objets est une pile. C'est à dire que l'on peut créer une représentation hiérarchique d'une scène, et récupérer non seulement le nom de l'objet final, mais tous les noms de la hiérarchie menant à cet objet. Voyons pour l'instant comment donner un nom à un objet sans utiliser pleinement le mécanisme de la pile : pour un usage courant, il suffit de placer un nom dans la pile, et de le remplacer par un autre nom pour un autre objet.

La pile des noms est initialisée par `glInitNames()`.

Un nom `name` est placé dans la pile en appelant `glPushName(GLuint name)`.

#### 4. Définir le volume de vision

Si le volume de vision du mode de Sélection est différent de celui du mode de Rendu (comme c'est le cas pour le picking), c'est à cette étape qu'il faut le modifier.

#### 5. Dessiner la scène en nommant les primitives significatives

L'appel à `glLoadName(GLuint name)` permet de remplacer le nom courant dans la pile par `name`. Si la pile est vide, cet appel génère l'erreur `GL_INVALID_OPERATION`.

#### 6. Quitter le mode Sélection et traiter les données retournées.

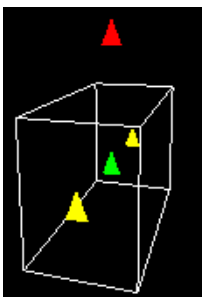
L'appel à `hits = glRenderMode(GL_RENDER)` ; permet de quitter le mode Sélection pour retourner en mode de Rendu. La valeur `hits` retournée est le nombre de hits de sélection. Une valeur négative signifie que le tableau de sélection a récupéré plus de données qu'il ne peut en accueillir. Les enregistrements de hits sont accessibles dans le tableau `tab` de la première étape.

## L'enregistrement des hits

Chaque enregistrement de hit se compose des éléments suivants :

Nombre de noms dans la pile au moment du hit Valeurs minimale et maximale de la coordonnée `z` de tous les sommets des primitives qui sont entrées en intersection avec le volume de vision depuis le dernier hit enregistré. Ces valeurs, situées dans la plage `[0, 1]`, sont multipliées par  $2^3 - 1$ , et arrondies à l'entier non signé le plus proche Contenu de la pile de noms au moment du hit, de bas en haut de la pile.

## Exemple de sélection



Le programme dont est issu cette image dessine quatre triangle (un vert, un rouge et deux jaunes), et représente une boîte en fil de fer correspondant au volume de vision du mode de sélection. Le premier triangle génère un hit, ce qui n'est pas le cas du second, et les troisième et quatrième triangles génèrent un hit unique.

`hits = 2`

`number of names for hit = 1`

`z1 is 0.999999; z2 is 0.999999`

`the name is 1`

`number of names for hit = 1`

`z1 is 0; z2 is 2`

`the name is 3`

## Utilisation de la pile des noms

Pour manipuler la pile des noms, on utilisera `glPushName(GLuint name)` pour ajouter le nom `name` au sommet de la pile, `glLoadName(GLuint name)` pour remplacer le nom situé au sommet de la pile par `name`, et `glPopName()` pour enlever le nom situé au sommet de la pile.

`glPushName(GLuint name)` génère l'erreur `GL_STACK_OVERFLOW` si la capacité de la pile est dépassée. La profondeur de la pile est au minimum de 64, et sa valeur pour une implémentation donnée est indiquée par `glGetIntegerv(GL_NAME_STACK_DEPTH)`.

`glLoadName(GLuint name)` génère l'erreur `GL_INVALID_OPERATION` si la pile est vide.

`glPopName()` génère l'erreur `GL_STACK_UNDERFLOW` si la pile est vide.

## *Picking*

Il s'agit de passer en mode de Sélection, avec un volume de vision restreint à une région autour du pointeur de la souris.

Notez que le pointeur de la souris est en coordonnées image, et qu'il faut définir un volume de vision dans l'espace de la scène. Cette opération est réalisée par la routine `gluPickMatrix(GLdouble x, GLdouble y, GLdouble largeur, GLdouble hauteur, GLint viewport[4])`.

Dans cette fonction, `(x,y)` est le centre de la région de picking en coordonnées de la fenêtre. `largeur` et `hauteur` définissent la taille de la région de picking en coordonnées d'écran. `viewport` indique les frontières du cadrage actif, dont les valeurs sont obtenues par `glGetIntegerv(GL_VIEWPORT, viewport)`.

Généralement, le picking est déclenché par un clic de la souris, et les coordonnées `(x, y)` sont celles du pointeur. Dans le programme `picksquare.c`, par exemple, c'est