

# OpenGL

1	
I.	<b>Documentation ..... 4</b>
Et des exemples ?	4
Et en Français ?	4
II.	<b>Présentation générale ..... 5</b>
OpenGL est basé sur des Etats	5
Pipe-Line de Rendu simplifié	5
Les Bibliothèques co-existant avec OpenGL	5
Utilisation de GLUT	5
Les couleurs	7
Les primitives géométriques	7
Le vecteur normal en un point	8
Le modèle d'ombrage	8
Elimination des surfaces cachées	8
III.	<b>Vision ..... 11</b>
Principe de la vision	11
La transformation de vision	11
La transformation de modélisation	12
La transformation de projection	13
La transformation de cadrage	13
Compréhension par l'exemple	14
IV.	<b>Modélisation hiérarchique ..... 15</b>
Description hiérarchique d'une scène	15
Pile de transformations	16
Manipulation des matrices de transformation	16
Exercices	16
V.	<b>Eclairage ..... 18</b>
Modèle d'éclairage OpenGL	18
Les lampes	19
Couleur d'un matériau	20
Exercice 1	21
Listes d'affichage	21
Quadriques	22
VI.	<b>Textures ..... 25</b>
Introduction	25
Les coordonnées de texture	25
Les Objets-Textures	27
Filtrage	27

Les niveaux de détail.....	27
Lecture d'une image de texture dans un fichier.....	28
Exercice.....	28
<b>VII. Tableaux de Sommets .....</b>	<b>30</b>
Introduction.....	30
Activer les tableaux.....	30
Spécifier les données des tableaux.....	30
Déréférencer le contenu des tableaux.....	32
Exercices.....	33
<b>VIII.Mélange de couleurs : blending .....</b>	<b>35</b>
Introduction.....	35
Activer/désactiver le blending.....	35
Facteurs de blending source et destination.....	35
<b>Définition .....</b>	<b>35</b>
<b>Spécifier les facteurs .....</b>	<b>35</b>
Exemples.....	36
<b>Mélange homogène de deux objets .....</b>	<b>36</b>
<b>Importance de l'ordre d'affichage .....</b>	<b>36</b>
Exercice.....	37
Du bon usage du tampon de profondeur.....	37
<b>IX. Lissage, fog et décalage de polygone.....</b>	<b>39</b>
Lissage (antialiasing).....	39
<b>Définition et principe .....</b>	<b>39</b>
<b>Exemple.....</b>	<b>39</b>
<b>Contrôle de la qualité : glHint(...)</b> .....	<b>39</b>
Fog.....	40
<b>Mise en oeuvre .....</b>	<b>41</b>
<b>Couleur et équation du fog.....</b>	<b>41</b>
Décalage de polygone.....	41
<b>Définition .....</b>	<b>42</b>
<b>Modes de rendu des polygones .....</b>	<b>42</b>
<b>Types de décalage .....</b>	<b>42</b>
<b>Valeur du décalage.....</b>	<b>42</b>
<b>Exemple.....</b>	<b>42</b>
<b>X. Les tampons d'image .....</b>	<b>45</b>
Introduction.....	45
Les tampons et leurs utilisations.....	45
Le tampon d'accumulation.....	47
Exercice.....	48

XI. Sélection et Picking.....	49
Introduction.....	49
Sélection.....	49
Principales étapes.....	49
Détail de ces étapes :.....	49
L'enregistrement des hits.....	50
Exemple de sélection.....	50
Utilisation de la pile des noms.....	51
Picking.....	51
Exercice.....	51

# I. Documentation



- Le premier réflexe est le site [www.opengl.org](http://www.opengl.org), c'est là que vous trouverez tout, tout, et des liens sur tout le reste. Allez d'abord visiter la rubrique [developers](#), il y a de la [documentation](#), des [exemples de code et des cours en ligne](#) (tutoriaux), des [FAQs](#) et des [Forums de discussion](#).
- Le livre OpenGL Programming Guide (Addison-Wesley Publishing Company) en anglais est disponible au format HTML. Il s'agit de la seconde édition, qui concerne OpenGL 1.1. ( <http://fly.cc.fer.hr/~unreal/theredbook/> )
- Voici l'ouvrage que je vous **recommande chaudement**. *OpenGL 1.2, guide officiel 3ème édition* (Woo, Neider, Davis & Shreiner chez CampusPress) . C'est de ce livre qu'est tirée la majeure partie de ce cours.

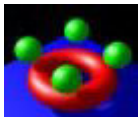


## Et des exemples ?



Les tutoriaux de Nate Robins sont **pédagogiques et interactifs**. Allez les télécharger et vous comprendrez par l'exemple toutes les bases d'OpenGL.

<http://www.xmission.com/~nate/tutors.html>



Les projets de Paul Baker sont des tutoriaux fort bien réalisés avec du code pour des fonctionnalités plus avancées. <http://www.paulsprojects.net/>



Les tutoriaux de NeHe sont intéressants pour les aspects plus avancés également. <http://nehe.gamedev.net/>



Le site CodeSampler présente des démos pour de nombreuses fonctionnalités d'OpenGL, et il indique l'équivalent Direct3D, pour DirectX8.1 et DirectX9.0.

<http://www.codesampler.com/>

## Et en Français ?

Bien sûr, il n'y a pas que dans le seul monde anglo-saxon qu'on utilise OpenGL. Dans le monde francophone, je vous recommande :

- Le site de Nicolas Janey, à l'université de Franche-Comté, il est **excellent** ! <http://raphaello.univ-fcomte.fr/IG/>



## II. *Présentation générale*

**OpenGL** ( <http://www.opengl.org> ) est un système graphique qui permet de visualiser une scène 3D (et aussi 2D). Les objets de cette scène peuvent être composés de points, de lignes, de polygones, de quadriques, de nurbs. Ils possèdent des attributs graphiques : paramètres de réflexion, couleur, texture. L'éclairage de la scène est réalisé par des lumières de différents types (spot, lumière à l'infini).

La bibliothèque OpenGL a été créée par [Silicon Graphics](#) et bénéficie sur ces machines de l'accélération matérielle. Une implémentation libre de droits d'OpenGL très performante a été écrite par Brian Paul : <http://www.mesa3d.org> . Et tout son code source est accessible !

### 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]->[Evalueurs]->[Opérations sur les sommets et assemblage de primitives]->[Discrétisation]->[Opérations sur les fragments]->[Image]

Les **Evalueurs** 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. Nous les décrirons ultérieurement en détail.

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).

### Les Bibliothèques co-existant avec OpenGL

GLU : OpenGL Utility Library contient les routines de bas niveau pour gérer les matrices de transformation et de projection, la facettisation des polygones et le rendu de surface.

Les bibliothèques d'extension du système de fenêtres permettent l'utilisation du rendu OpenGL. Il s'agit de GLX pour X Windows (fonctions ayant pour préfixe **glX**) et de WGL pour Microsoft Windows (fonctions ayant pour préfixe **wgl**).

GLUT : OpenGL Utility Toolkit est une boîte à outils indépendante du système de fenêtrage, écrite par Mark Kilgard pour simplifier la tâche d'utiliser des systèmes différents (fonctions ayant pour préfixe **glut**).

### Utilisation de GLUT

Pour la compilation, copiez le [Makefile](#) et tapez `make`

Examinez et compilez le programme [hello.c](#) Comprenez l'utilité des différentes procédures.

faites de même pour [double.c](#).

### *Les fonctions de gestion d'une fenêtre*

**glutInit**(int \* argc, char \*\*argv) initialise GLUT et traite les arguments de la ligne de commande.

**glutInitDisplayMode**(unsigned int mode) permet de choisir entre couleurs RVB et couleurs indexées, et de déclarer les buffers utilisés.

**glutInitWindowPosition**(int x, int y) spécifie la localisation dans l'écran du coin haut gauche de la fenêtre.

**glutInitWindowSize**(int width, int size) spécifie la taille en pixels de la fenêtre.

**int glutCreateWindow**(char \* string) crée une fenêtre contenant un contexte OpenGL et renvoie l'identificateur de la fenêtre. La fenêtre ne sera affichée que lors du premier **glutMainLoop()**

### *La fonction d'affichage*

**glutDisplayFunc**(void (\*func)(void)) spécifie la fonction à appeler pour l'affichage

**glutPostRedisplay**(void) permet de forcer le réaffichage de la fenêtre.

### *Lancement de la boucle d'événements*

1. Attente d'un événement
2. Traitement de l'événement reçu
3. Retour en 1.

**glutMainLoop**(void) lance la boucle principale qui tourne pendant tout le programme, attend et appelle les fonctions spécifiques pour traiter les événements.

### *Traitement des événements*

**glutReshapeFunc**(void (\*func)(int w, int h)) spécifie la fonction à appeler lorsque la fenêtre est retaillée.

**glutKeyboardFunc**(void (\*func)(unsigned char key, int x, int y)) spécifie la fonction à appeler lorsqu'une touche est pressée

**glutMouseFunc**(void (\*func)(int button, int state, int x, int y)) spécifie la fonction à appeler lorsqu'un bouton de la souris est pressé.

**glutMotionFunc**(void (\*func)(int x, int y)) spécifie la fonction à appeler lorsque la souris est déplacée tout en gardant un bouton appuyé.

### *Exécution en tâche de fond*

**glutIdleFunc**(void (\*func)(void)) spécifie la fonction à appeler lorsqu'il n'y a pas d'autre événement.

### *Dessin d'objets Tridimensionnels*

GLUT possède des routines pour afficher les objets suivants :

cone, cube, tétraèdre, octaèdre, icosaèdre, dodécaèdre, sphère, tore, théière (!)

### *Animation*

Une animation peut se réaliser simplement en utilisant la technique du **double buffer** : montrer à l'écran une image correspondant à une première zone mémoire (buffer) et dessiner les objets dans une deuxième zone mémoire qui n'est pas encore à l'écran, et qui sera affiché lorsque la scène entière y sera calculée, et à une fréquence régulière. L'échange des buffers peut se faire par la commande **glutSwapBuffers**(void).

## Les couleurs

Les couleurs sont définies en OpenGL de deux manières :

Couleurs indexées : 256 couleurs sont choisies, et on se réfère au numéro de la couleur (son index). C'est un mode qui était intéressant lorsque les écrans d'ordinateurs ne savaient afficher que 256 couleurs simultanées. Glissons allégrement à l'époque moderne

Couleurs RVBA : une couleur est définie par son intensité sur 3 composantes Rouge, Vert, Bleu. La quatrième composante est appelée canal Alpha, et code l'opacité.

La couleur d'un objet est spécifiée par l'appel à [glColor\(...\)](#).

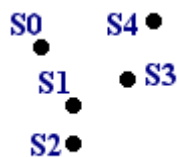
## Les primitives géométriques

### Déclaration

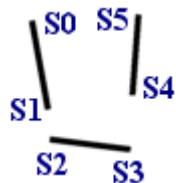
[glBegin](#)(GLenum mode) ouvre la déclaration des sommets de la primitive

[glEnd](#)(void) termine cette déclaration

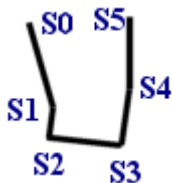
Il y a dix types de primitives différents, qui correspondent à des points, des lignes, des triangles, des quadrilatères, et des polygones convexes.



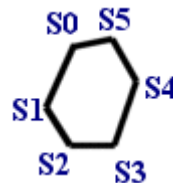
GL\_POINTS



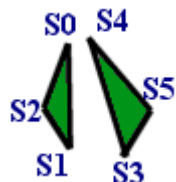
GL\_LINES



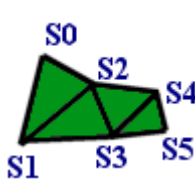
GL\_LINE\_STRIP



GL\_LINE\_LOOP



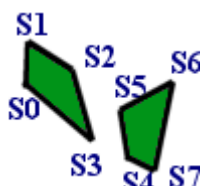
GL\_TRIANGLE



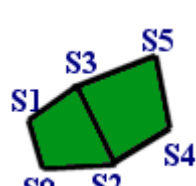
GL\_TRIANGLE\_STRIP



GL\_TRIANGLE\_FAN



GL\_QUADS



GL\_QUAD\_STRIP



GL\_POLYGON

## 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\(GLenum mode\)](#). Compilez [scene.c](#) et exécutez le programme obtenu. Changez 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.

---



```

/*
 * hello.cpp
 * This is a simple, introductory OpenGL program.
 */
#include <GL/glut.h>

void display(void)
{
/* clear all pixels */
glClear (GL_COLOR_BUFFER_BIT);

/* draw white polygon (rectangle) with corners at
 * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
glBegin(GL_POLYGON);
glColor3f (1.0, 1.0, 1.0);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();

/* Swap the buffers to show the one
 * on which we writed
 */
glutSwapBuffers();
}

```

```

void init (void)
{
/* select clearing color */
glClearColor (0.0, 1.0, 0.0, 0.0);

/* initialize viewing values */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

/*
 * Declare initial window size, position, and display
 mode
 * (double buffer and RGBA). Open window with
 "hello"
 * in its title bar. Call initialization routines.
 * Register callback function to display graphics.
 * Enter main loop and process events.
 */
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE |
GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition (100, 100);
glutCreateWindow ("hello");
init ();
glutDisplayFunc(display);
glutMainLoop();
return 0; /* ANSI C requires main to return int. */
}

```

```

/*
 * double.cpp
 * This is a simple double buffered program.
 * Pressing the left mouse button rotates the rectangle.
 * Pressing the middle mouse button stops the
rotation.
 */
#include <GL/glut.h>
#include <stdlib.h>

static GLfloat spin = 0.0;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(spin, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glPopMatrix();

    glutSwapBuffers();
}

void spinDisplay(void) {
    spin = spin + 2.0;
    if (spin > 360.0)
        spin = spin - 360.0;
    glutPostRedisplay();
}

void init(void) {
    glClearColor (0.0, 1.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
}

void reshape(int w, int h) {
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouse(int button, int state, int x, int y) {
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(spinDisplay);
            break;
        case GLUT_MIDDLE_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(NULL);
            break;
        default:
            break;
    }
}

/*
 * Request double buffer display mode.
 * Register mouse input callback functions
 */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE |
        GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}

```

# III. 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 :

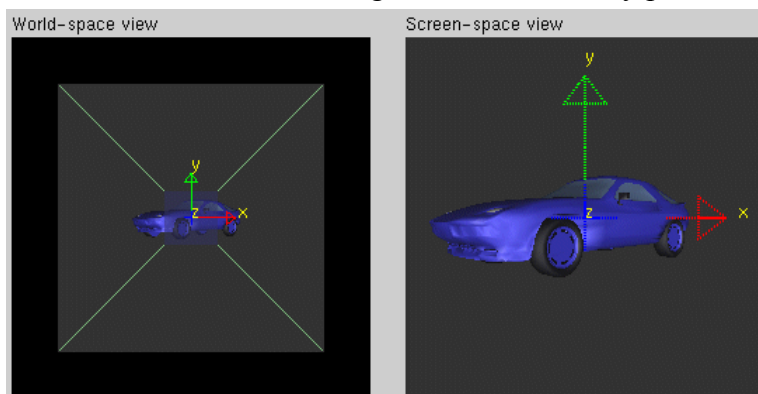
	Avec un appareil photo	Avec un ordinateur	type de transformation
1	Positionner l'appareil photo	Placer la caméra virtuelle	transformation de vision
2	Arranger les éléments d'une scène à photographier	Composer une scène virtuelle à représenter	transformation de modélisation
3	Choisir la focale de l'appareil 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

[Sommets]	[Matrice de Modélisation-Vision]	[Matrice de Projection]	[Division perspective]	[Transformation de cadrage]
	Coordonnées de l'oeil	Coordonnées de clipping	Coordonnées normalisées	Coordonnées de la fenêtre

## 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 :



le point de vue de la caméra.

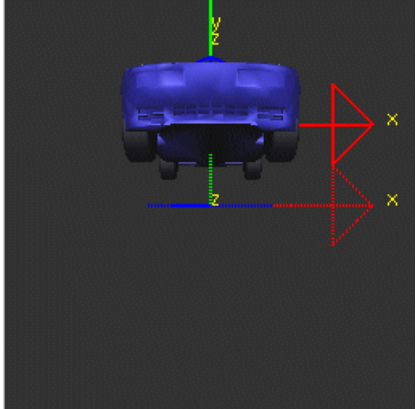
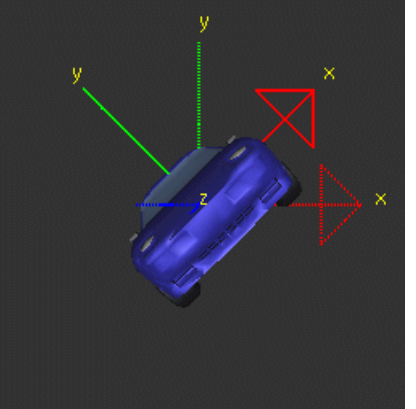
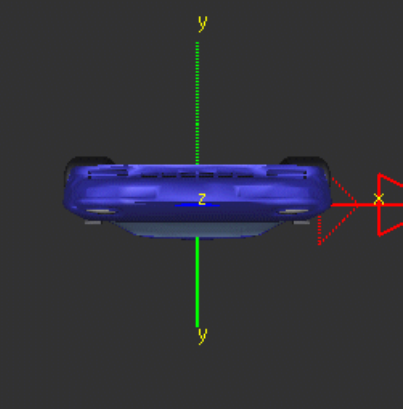
On peut également utiliser (et c'est plus simple pour commencer) la procédure [gluLookAt](#)(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz).

Vous retrouverez la Porsche sur le [site de Nate Robins](#).

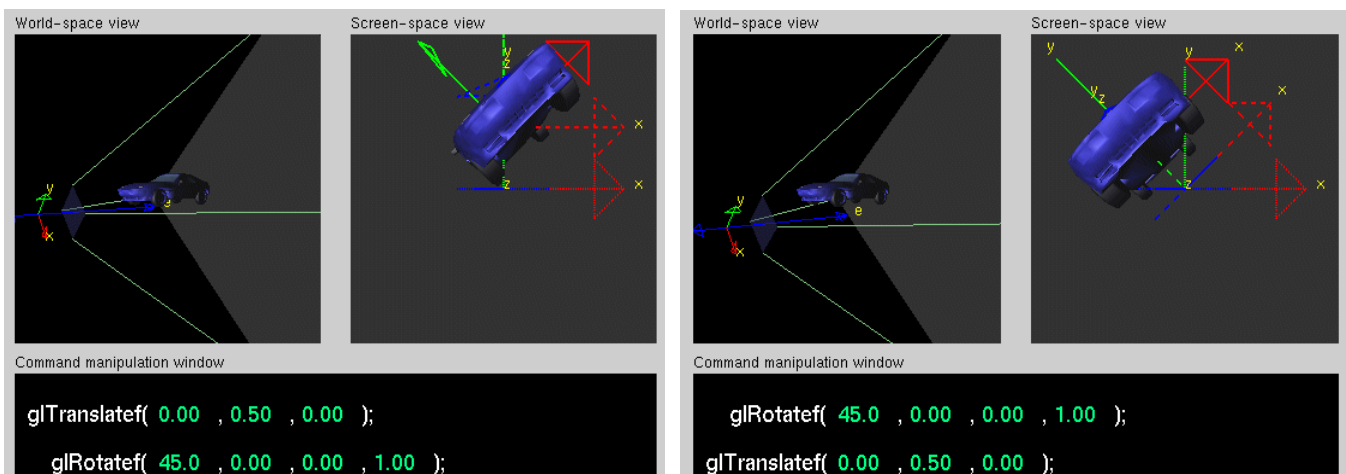
En fait, ce qui compte pour la visualisation est la position *relative* de la caméra par rapport aux objets. Il est équivalent par exemple de traduire la caméra d'un vecteur  $T$  ou de traduire tous les objets du vecteur  $-T$ .

On utilise ainsi les procédures OpenGL de translation et de rotation appliquées aux objets, [glTranslate\\*](#)() et [glRotate\\*](#)() pour changer

## La transformation de modélisation

<p><b>glTranslate*</b>(TYPE x, TYPE y, TYPE z) translate le repère local de l'objet du vecteur (x,y,z).</p>	<p><b>glRotate*</b>(TYPE angle, TYPE x, TYPE y, TYPE z) opère une rotation de l'objet autour du vecteur (x,y,z).</p>	<p><b>glScale*</b>(TYPE a, TYPE b, TYPE c) 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>

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.



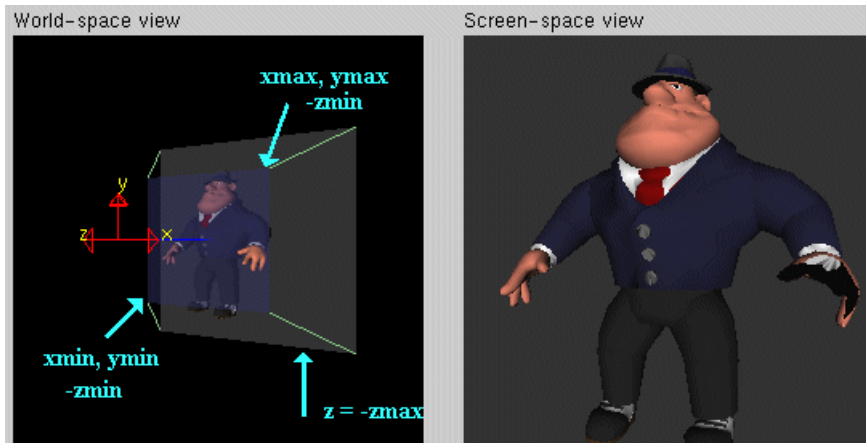
Pour s'y retrouver, il suffit de considérer que la transformation est appliquée au *repère local* de l'objet.

## La transformation de projection

N'oubliez pas d'abord d'écrire `glMatrixMode(GL_PROJECTION); glLoadIdentity();` avant les commandes de transformation.

### Projection perspective

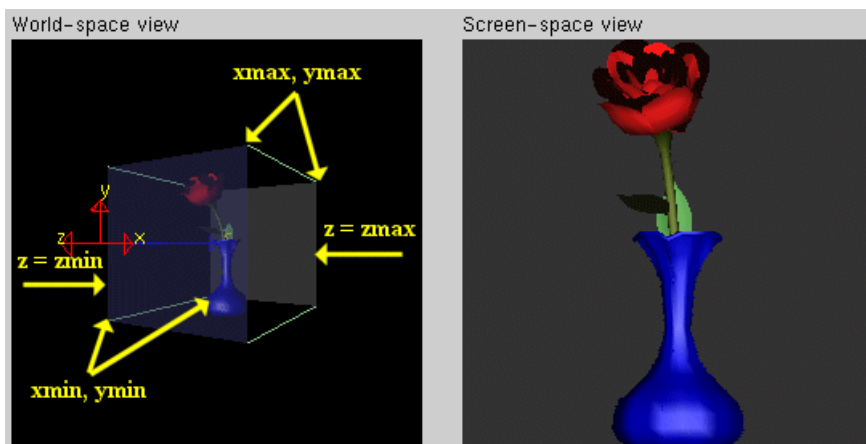
[`gluPerspective`](#)( `GLdouble angle_ouverture`, `GLdouble ratio_XY`, `GLdouble zmin`, `GLdouble zmax`) ou [`glFrustum`](#)(`GLdouble xmin`,`GLdouble xmax`, `GLdouble ymin`,`GLdouble ymax`, `GLdouble zmin`, `GLdouble zmax` ) peuvent être utilisées. Ces fonctions définissent le **volume de vision** 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 (`xmin`, `ymin`, `-zmin`) et (`xmax`, `ymax`, `-zmin`). `zmin` et `zmax` 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 parallépipède 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)

## Compréhension par l'exemple

Examinez le programme [cube.c](#). Modifiez dans le fichier Makefile la ligne commençant par `PROGS =` en `PROGS = cube`. Compilez le programme (`make`).

### *La transformation de vision*

[glLoadIdentity\(\)](#) donne à la *Matrice courante* la valeur identité. En effet, les opérations de transformation multiplient la matrice courante par leur matrice de transformation associée, alors autant partir du bon pied.

[gluLookAt\(\)](#) spécifie la matrice de vision, en positionnant l'oeil de la caméra virtuelle, le point visé, et la direction du vecteur *verticale* de la caméra (ce qui détermine l'orientation de celle-ci).

### *La transformation de modélisation*

Le modèle est positionné ou orienté, ou encore mis à l'échelle si nécessaire, ou toute combinaison de ces opérations. Dans l'exemple `cube.c`, le cube est mis à l'échelle par [glScalef\(\)](#). [glLoadIdentity\(\)](#) donne à la *Matrice courante* la valeur identité. En effet, les opérations de transformation multiplient la matrice courante par leur matrice de transformation associée, alors autant partir du bon pied.

[gluLookAt\(\)](#) spécifie la matrice de vision, en positionnant l'oeil de la caméra virtuelle, le point visé, et la direction du vecteur *verticale* de la caméra (ce qui détermine l'orientation de celle-ci).

### *Exercice*

Positionnez la caméra en (0,0,10) et visionnez le cube.

Comment peut-on obtenir la même image en laissant la caméra en (0,0,5) ?

### *La transformation de projection*

Elle comprend le choix du champ de vision (comme sur un appareil photo on peut adapter un grand angle, un objectif normal ou un téléobjectif).

Elle comprend également le choix du type de projection : perspective ou orthographique.

La projection *perspective* fait apparaître les objets lointains plus petits que ceux qui sont proches. Dans l'exemple, elle est spécifiée par [glFrustum\(\)](#).

La projection *orthographique* n'affecte pas la taille relative des objets.

### *Exercice*

Remplacez [glFrustum\(\)](#) par [gluPerspective\(\)](#) avec les paramètres (60.0,1.0,1.5,20.0).

Expérimentez différentes valeurs des paramètres de la projection. En particulier *fovy* et *aspect*.

### *La transformation de cadrage*

[glViewport\(\)](#) spécifie la taille et la position de l'image sur l'écran de l'ordinateur.

# IV. Modélisation hiérarchique

## Description hiérarchique d'une scène

### Définition

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 **Grphe 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 associé une transformation géométrique qui positionne l'objet fils dans le repère de son père.

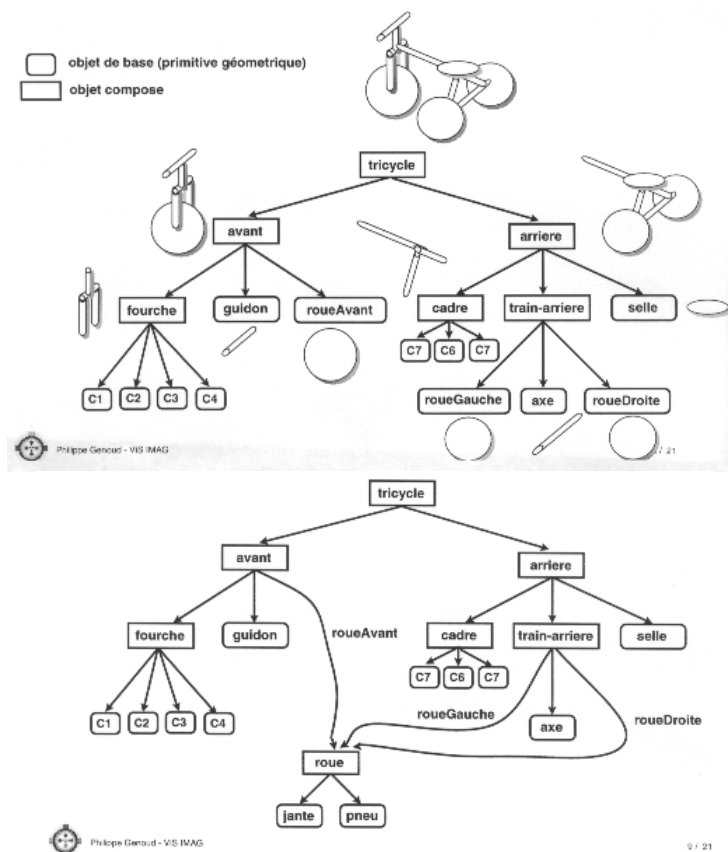
### Construction

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)

### Exemple





## Pile de transformations

OpenGL utilise les *coordonnées homogènes* pour manipuler ses objets (cf Cours de Math). Il maintient trois matrices  $4 \times 4$  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 noeud 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.

## Exercices

1. Repartez de la scène du TD précédent (scène comprenant une sphère rouge de rayon unité en (2,0,0), ([glutSolidSphere](#)(GLdouble radius, GLint slices, GLint stacks)), un cube bleu d'arête 2, tourné de  $20^\circ$  autour de l'axe x du repère de toute la scène, positionné en (0,3,0), et un cube jaune tourné de  $-20^\circ$  autour de l'axe z du repère de la scène, positionné en (3,3,3) ([glutSolidCube](#)(GLdouble size), caméra positionnée de manière à voir l'ensemble des objets.

Ecrivez le graphe de la scène (ou plutôt UN graphe pour la scène)

Ecrivez le programme qui affiche cette scène en utilisant la gestion de la pile des matrices.

2. Copiez le programme [robot.c](#) et compilez-le chez vous.

Faites le dessin du graphe de la scène avec toutes les transformations.

Faites un schéma du bras du robot en indiquant les repères locaux de l'épaule (shoulder) et du coude (elbow)

Modifiez la scène en ajoutant trois doigts (index, majeur, et pouce en opposition) au bras du robot, chaque doigt étant composé de deux segments. Faites le dessin du nouveau graphe, et le schéma du bras complet.

Modifiez le programme en conséquent. Chaque élément doit être mobile de manière indépendante et doit être animable par l'utilisateur au travers du clavier.

Quels types de transformation utilisez-vous ?

Combien de degrés de libertés possède alors le bras ?

Ajoutez un degré de rotation du poignet : suivant l'axe du bras.



```

/*
 * robot.cpp
 * This program shows how to composite modeling transformations
 * to draw translated and rotated hierarchical models.
 * Interaction: pressing the s and e keys (shoulder and elbow)
 * alters the rotation of the robot arm.
 */
#include <GL/glut.h>
#include <stdlib.h>

static int shoulder = 0, elbow = 0;

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glTranslatef (-1.0, 0.0, 0.0);
    glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    glPushMatrix();
    glScalef (2.0, 0.4, 1.0);
    glutWireCube (1.0);
    glPopMatrix();

    glTranslatef (1.0, 0.0, 0.0);
    glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
    glTranslatef (1.0, 0.0, 0.0);
    glPushMatrix();
    glScalef (2.0, 0.4, 1.0);
    glutWireCube (1.0);
    glPopMatrix();

    glPopMatrix();
    glutSwapBuffers();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0,
20.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 's':
            shoulder = (shoulder + 5) % 360;
            glutPostRedisplay();
            break;
        case 'S':
            shoulder = (shoulder - 5) % 360;
            glutPostRedisplay();
            break;
        case 'e':
            elbow = (elbow + 5) % 360;
            glutPostRedisplay();
            break;
        case 'E':
            elbow = (elbow - 5) % 360;
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE |
GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

# V. *Eclairage*

## Modèle d'éclairage OpenGL

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 taille 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\_LIGHT*i* est allumée par un [glEnable\(GL\\_LIGHT\*i\*\)](#). 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'oeil 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).

### Transparence

La transparence est obtenue en indiquant pour un objet une couleur diffuse RVBA ou la valeur A sur le canal alpha est strictement plus petite que un.

## Exercice 1

1. Copiez et compilez le programme [colormat.c](#). Il permet de modifier indépendamment les quatre composants du matériau de la sphère.
2. Ajoutez une lampe en  $(-1,1,1)$  qui émet une lumière ambiante verte, une lumière diffuse bleue, et une lumière spéculaire rouge.

Utilisez les touches '0' (resp. '1') pour allumer et éteindre la lampe 0 (resp. 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 et 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.

## Quadriques

La bibliothèque GLU permet de créer certains objets définis par une équation quadratique : des sphères, des cylindres, des disques et des disques partiels. Pour cela il y a cinq étapes :

1. Utiliser [gluNewQuadric\(\)](#) pour créer une quadrique.
  2. Spécifier les attributs de rendu :
    - [gluQuadricOrientation\(\)](#) indique la direction des normales vers l'extérieur ou l'intérieur.
    - [gluQuadricDrawStyle\(\)](#) indique le style de rendu : avec des points, des lignes ou des polygones pleins.
    - [gluQuadricNormals\(\)](#) permet de générer des normales pour chaque face ou pour chaque sommet.
    - [gluQuadricTexture\(\)](#) permet de générer les coordonnées de texture.
  3. Indiquer la fonction qui traite les erreurs dans l'affichage de la quadrique par [gluQuadricCallback\(\)](#)
  4. Afficher la quadrique désirée avec [gluSphere\(\)](#), [gluCylinder\(\)](#), [gluDisk\(\)](#), ou [gluPartialDisk\(\)](#). Pour une meilleure efficacité, il est recommandé d'utiliser des listes d'affichage.
  5. Enfin, pour détruire l'objet, appeler [gluDeleteQuadric\(\)](#).
-

```

/*
 * colormat.cpp
 * After initialization, the program will be in
 * ColorMaterial mode. Interaction: pressing the
 * mouse buttons will change the diffuse reflection
 values.
 */
#include <GL/glut.h>
#include <stdlib.h>

GLfloat ambientMaterial[4] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat diffuseMaterial[4] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat specularMaterial[4] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat shininessMaterial = 0.0;
GLfloat emissionMaterial[4] = { 0.0, 0.0, 0.0, 1.0 };

GLfloat white[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat red[] = { 1.0, 0.0, 0.0, 0.0 };
GLfloat green[] = { 0.0, 1.0, 0.0, 0.0 };
GLfloat blue[] = { 0.0, 0.0, 1.0, 0.0 };
GLfloat black[] = { 0.0, 0.0, 0.0, 0.0 };

/* Initialize material property, light source, lighting
 model, and depth buffer.
 */
void init(void) {
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);

    glLightfv(GL_LIGHT0, GL_AMBIENT, red);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, green);
    glLightfv(GL_LIGHT0, GL_SPECULAR, blue);
    glLightfv(GL_LIGHT0, GL_POSITION,
light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
    glMaterialfv(GL_FRONT, GL_AMBIENT,
ambientMaterial);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,
diffuseMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR,
specularMaterial);
    glMaterialf(GL_FRONT, GL_SHININESS,
shininessMaterial);
    glMaterialfv(GL_FRONT, GL_EMISSION,
emissionMaterial);
    /* glutSolidSphere(1.0, 20, 16);*/
    glutSolidSphere(1.0, 40, 40);
    glutSwapBuffers();
}

void reshape (int w, int h) {
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0,
10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void incrementMaterial (GLfloat material[4], GLfloat
delta) {
    material[0] +=delta;
    if (material[0] > 1.0) material[0] = 1.0;
    if (material[0] < 0.0) material[0] = 0.0;

    material[1] +=delta;
    if (material[1] > 1.0) material[1] = 1.0;
    if (material[1] < 0.0) material[1] = 0.0;

    material[2] +=delta;
    if (material[2] > 1.0) material[2] = 1.0;
    if (material[2] < 0.0) material[2] = 0.0;
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'r':

            ambientMaterial[0]=ambientMaterial[1]=ambientMate
            rial[2]=0.0;
            glMaterialfv(GL_FRONT, GL_AMBIENT,
            ambientMaterial);

            diffuseMaterial[0]=diffuseMaterial[1]=diffuseMaterial
            [2]=0.0;
            glMaterialfv(GL_FRONT, GL_DIFFUSE,
            diffuseMaterial);

            specularMaterial[0]=specularMaterial[1]=specularMat
            erial[2]=0.0;
            glMaterialfv(GL_FRONT, GL_EMISSION,
            emissionMaterial);

```

```

    shininessMaterial=0.0;
    glMaterialf(GL_FRONT, GL_SHININESS,
shininessMaterial);

emissionMaterial[0]=emissionMaterial[1]=emissionM
aterial[2]=0.0;
    glMaterialfv(GL_FRONT, GL_EMISSION,
emissionMaterial);
    glutPostRedisplay();
    break;

case 'a':
    incrementMaterial(ambientMaterial, 0.1);
    glMaterialfv(GL_FRONT, GL_AMBIENT,
ambientMaterial);
    glutPostRedisplay();
    break;
case 'A':
    incrementMaterial(ambientMaterial, -0.1);
    glMaterialfv(GL_FRONT, GL_AMBIENT,
ambientMaterial);
    glutPostRedisplay();
    break;

case 'd':
    incrementMaterial(diffuseMaterial, 0.1);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,
diffuseMaterial);
    glutPostRedisplay();
    break;
case 'D':
    incrementMaterial(diffuseMaterial, -0.1);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,
diffuseMaterial);
    glutPostRedisplay();
    break;

case 's':
    incrementMaterial(specularMaterial, 0.1);
    /*specularMaterial[0]=specularMaterial[1]=0.0;*/
    glMaterialfv(GL_FRONT, GL_SPECULAR,
specularMaterial);
    glutPostRedisplay();
    break;
case 'S':
    incrementMaterial(specularMaterial, -0.1);
    /*specularMaterial[0]=specularMaterial[1]=0.0;*/
    glMaterialfv(GL_FRONT, GL_SPECULAR,
specularMaterial);
    glutPostRedisplay();

break;

case 'b':
    shininessMaterial +=2.0; if
(shininessMaterial>128.0) shininessMaterial=128.0;
    glMaterialf(GL_FRONT, GL_SHININESS,
shininessMaterial);
    glutPostRedisplay();
    break;
case 'B':
    shininessMaterial -=2.0; if (shininessMaterial<0.0)
shininessMaterial=0.0;
    glMaterialf(GL_FRONT, GL_SHININESS,
shininessMaterial);
    glutPostRedisplay();
    break;

case 'e':
    incrementMaterial(emissionMaterial, 0.1);
    glMaterialfv(GL_FRONT, GL_EMISSION,
emissionMaterial);
    glutPostRedisplay();
    break;
case 'E':
    incrementMaterial(emissionMaterial, -0.1);
    glMaterialfv(GL_FRONT, GL_EMISSION,
emissionMaterial);
    glutPostRedisplay();
    break;

case 27:
    exit(0);
    break;
}
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE |
GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```



# VI. Textures

## Introduction

Le placage de texture consiste à placer une image (la texture) sur un objet. Cette opération se comprend aisément lorsqu'il s'agit de plaquer une image rectangulaire sur une face rectangulaire, tout au plus imagine-t-on au premier abord qu'il y a un changement d'échelle à effectuer sur chacune des dimensions.

Mais peut-être au contraire le choix est de répéter l'image un certain nombre de fois sur la face.

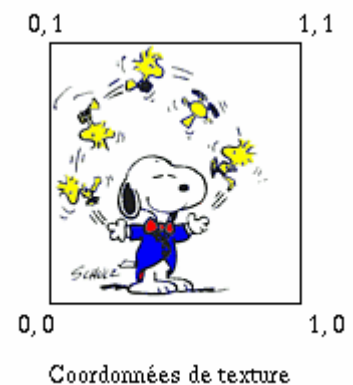
Se pose ensuite la question d'objets qui ne sont pas des faces rectangulaires : comment définir la portion d'image et la manière de la plaquer ?

### Comment procéder de manière générale

- 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.

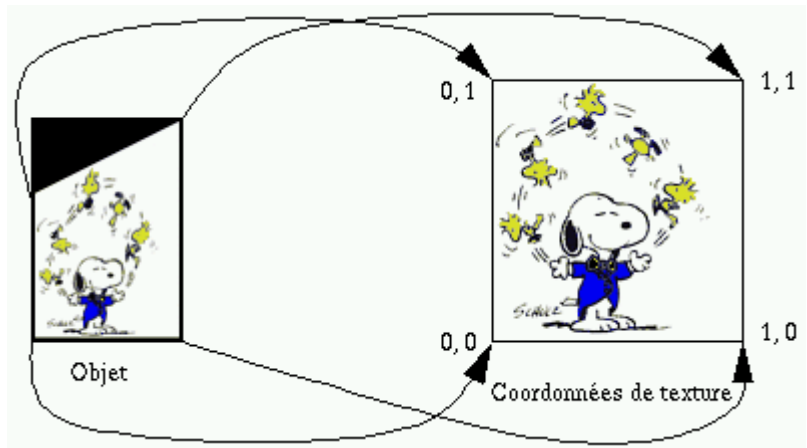
## Les coordonnées de texture

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

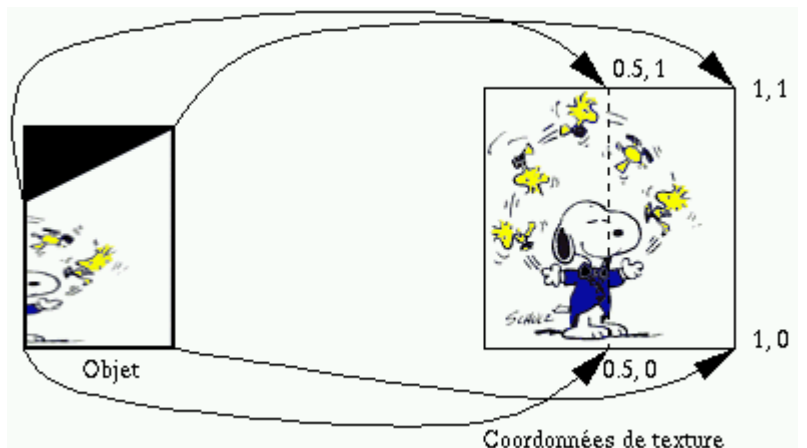


On assigne avec `glTexCoord*()` pour chaque sommet des objets à texturer un couple de valeurs qui indique les coordonnées de texture de ce sommet.

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.



### Exercice

Examinez le programme [texture1.c](#) (il utilise l'image [snoopy2.ppm](#)) pour comprendre l'utilisation des coordonnées de texture.

Plaquez uniquement la tête de snoopy sur le quadrilatère grâce aux coordonnées de texture.

### 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`.

### Exercice

Dans la moitié inférieure du quadrilatère, plaquez trois fois l'image du snoopy.

## 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 :
  - 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`

### Exercice

Examinez le programme [texturebind.c](#) qui utilise les images [tex1.ppm](#) et [tex2.ppm](#)

## 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 **mipmapping** 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 : suite

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.

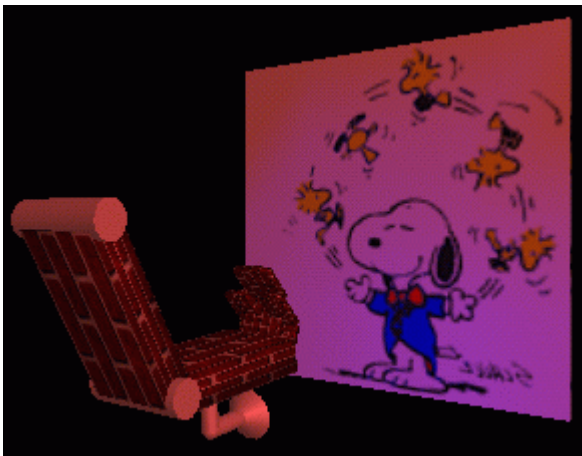
## Lecture d'une image de texture dans un fichier

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.

## Exercice

Récupérez deux images de textures (images qui peuvent se répéter) et plaquez l'une sur le bras de votre robot, l'autre sur l'avant-bras et les doigts.



Enjoy !

```

/* texturebind.cpp */
#include <GL/glut.h>
#include "loadppm.c"

char texFileName1[] = "tex1.ppm";
char texFileName2[] = "tex2.ppm";
PPMImage *image1,*image2;
GLuint texName[2];
GLfloat alpha=0.0;

void myinit(void) {
    image1 = LoadPPM(texFileName1);
    image2 = LoadPPM(texFileName2);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glGenTextures(2, texName);

    glBindTexture(GL_TEXTURE_2D, texName[0]);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3,
image1->sizeX, image1->sizeY, GL_RGB,
GL_UNSIGNED_BYTE, image1->data);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexEnvf(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE, GL_DECAL);

    glBindTexture(GL_TEXTURE_2D, texName[1]);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3,
image2->sizeX, image2->sizeY, GL_RGB,
GL_UNSIGNED_BYTE, image2->data);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexEnvf(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

    glRotatef(alpha,0.0,1.0,0.0);

    glBindTexture(GL_TEXTURE_2D, texName[0]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
    glEnd();
    glBindTexture(GL_TEXTURE_2D, texName[1]);
    glBegin(GL_QUADS);
    glTexCoord2f(0, 0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0, 3); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(3, 3); glVertex3f(2.414, 1, -1.414);
    glTexCoord2f(3, 0); glVertex3f(2.414, -1, -1.414);
    glEnd();

    glutSwapBuffers();
}

void myReshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h,
1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

static void Key(unsigned char key, int x, int y) {
    switch (key) {
        case 27: exit(1);
        case 'a': alpha+=5.0; break;
        case 'A': alpha-=5.0; break;
        case 'r': alpha=0.0;break;
    }
    glutPostRedisplay();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("texturebind");
    myinit();
    glutReshapeFunc (myReshape);
    glutKeyboardFunc(Key);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}

```

# VII. Tableaux de Sommets

## Introduction

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.

### *Comment procéder de manière générale*

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 **déréférencer** 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 des 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.

## *Routines 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]);
```

## Déréférencer 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 **glArrayElement**(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);
  glArrayElement(1);
  glArrayElement(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\_BYTES**, **GL\_UNSIGNED\_SHORT**, ou **GL\_UNSIGNED\_INT**. Le type de primitive géométrique est indiqué par *mode* de la même manière que dans **glBegin()**.

**glDrawElements()** ne doit pas être encapsulé dans une paire **glBegin()/ glEnd()**.

**glDrawElements()** vérifie que les paramètres *mode*, *nombre*, et *type* sont valides, et effectue ensuite un traitement semblable à la séquence de commandes :

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

## Exercices

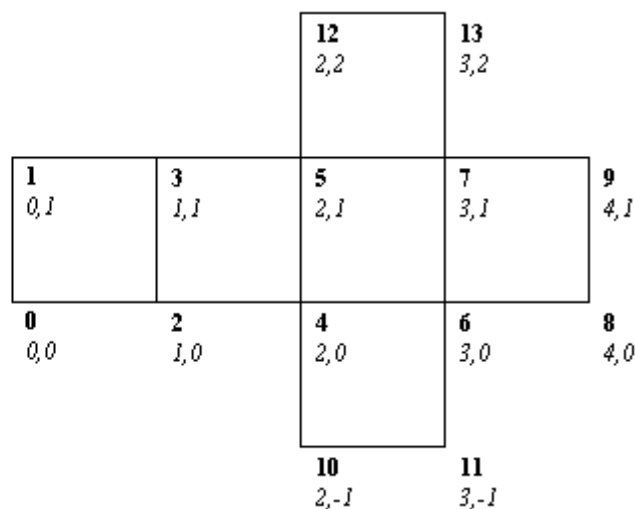
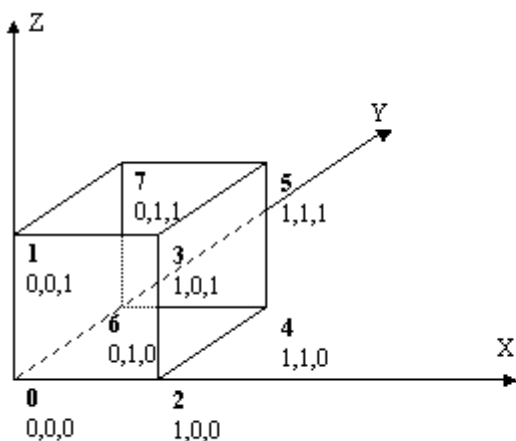
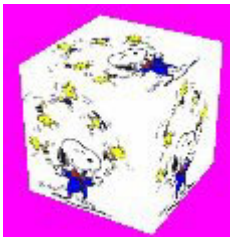
### Réécrire la méthode `ParalleX()` en utilisant des tableaux

Pour cet exercice, vous affecterez simplement une couleur à chaque sommet, en utilisant un tableau pour les sommets, et un pour les couleurs.



### Ensuite, texturez les faces au lieu de les colorer

Attention, chaque sommet de chaque face doit avoir ses coordonnées de texture propres : on ne peut pas se contenter de 8 sommets.



### *Lecture d'un objet dans un fichier*

Chargez et affichez l'objet [Generic.ply](#) qui est au format ply suivant :

- nombre de sommets
- nombre de facettes
- coordonnées de tous les sommets
- nombre de sommets et indices des sommets de toutes les facettes

Comment calculer les normales ?

### *Et la texture ?*

Chargez et affichez l'objet [MAD.ply](#) qui est au format plyTex suivant :

- nombre de sommets
- nombre de facettes
- coordonnées de tous les sommets
- nombre de sommets et indices des sommets de toutes les facettes
- coordonnées de texture u et v de tous les sommets

Affichez-le avec l'image de texture [MAD.tex.ppm](#)

---

# VIII. Mélange de couleurs : *blending*

## Introduction

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).

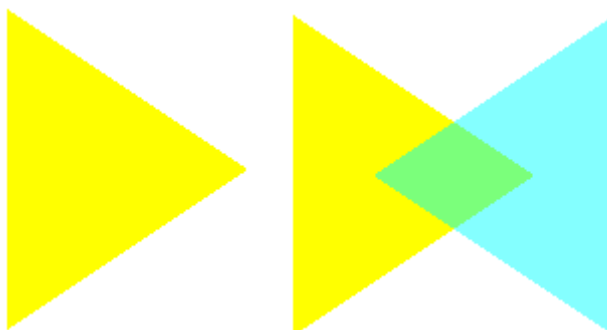
Constante	Facteur concerné		Valeur du facteur de blinding
GL_ZERO	source ou	destination	(0, 0, 0, 0)
GL_ONE	source ou	destination	(1, 1, 1, 1)
GL_DST_COLOR	source		<b>(Rd, Vd, Bd, Ad)</b>
GL_SRC_COLOR		destination	<b>(Rs, Vs, Bs, As)</b>
GL_ONE_MINUS_DST_COLOR	source		(1, 1, 1, 1) - <b>(Rd, Vd, Bd, Ad)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

## 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 blinding 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 blinding 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 [alpha.c](#),

- le facteur de blinding 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.

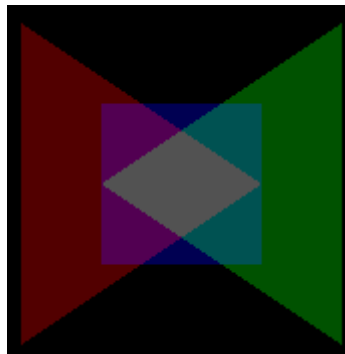
A droite, l'ordre d'affichage est inversé.

Expliquez la différence entre les deux images.



## Exercice

Affichez sur fond noir le mélange à parts égales de deux triangles rouges et verts correspondants à ceux du programme [alpha.c](#), et d'un carré bleu de coins opposés (0.3, 0.3) et (0.7, 0.7) :



Quelle est l'importance de l'ordre d'affichage ?

## 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.

Voir l'exemple [alpha3D.c](#).

```

/*
 * alpha.cpp
 * This program draws several overlapping filled
 polygons to demonstrate the effect order has on alpha
 blending results.
 * Use the 't' key to toggle the order of drawing
 polygons.
 */
#include <GL/glut.h>
#include <stdlib.h>

static int leftFirst = GL_TRUE;

/* Initialize alpha blending function. */
static void init(void) {
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
    glShadeModel (GL_FLAT);
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

static void drawLeftTriangle(void) {
    /* draw yellow triangle on LHS of screen */

    glBegin (GL_TRIANGLES);
        glColor4f(1.0, 1.0, 0.0, 0.75);
        glVertex3f(0.1, 0.9, 0.0);
        glVertex3f(0.1, 0.1, 0.0);
        glVertex3f(0.7, 0.5, 0.0);
    glEnd();
}

static void drawRightTriangle(void) {
    /* draw cyan triangle on RHS of screen */

    glBegin (GL_TRIANGLES);
        glColor4f(0.0, 1.0, 1.0, 0.75);
        glVertex3f(0.9, 0.9, 0.0);
        glVertex3f(0.3, 0.5, 0.0);
        glVertex3f(0.9, 0.1, 0.0);
    glEnd();
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);

    if (leftFirst) {
        drawLeftTriangle();
        drawRightTriangle();
    }
    else {
        drawRightTriangle();
        drawLeftTriangle();
    }

    glFlush();
}

void reshape(int w, int h) {
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (0.0, 1.0, 0.0,
1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0,
1.0);
}

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 't':
        case 'T':
            leftFirst = !leftFirst;
            glutPostRedisplay();
            break;
        case 27: /* Escape key */
            exit(0);
            break;
        default:
            break;
    }
}

/* Main Loop
 * Open window with initial window size, title bar,
 * RGBA display mode, and handle input events.
 */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE |
GLUT_RGB);
    glutInitWindowSize (200, 200);
    glutCreateWindow (argv[0]);
    init();
    glutReshapeFunc (reshape);
    glutKeyboardFunc (keyboard);
    glutDisplayFunc (display);
    glutMainLoop();
    return 0;
}

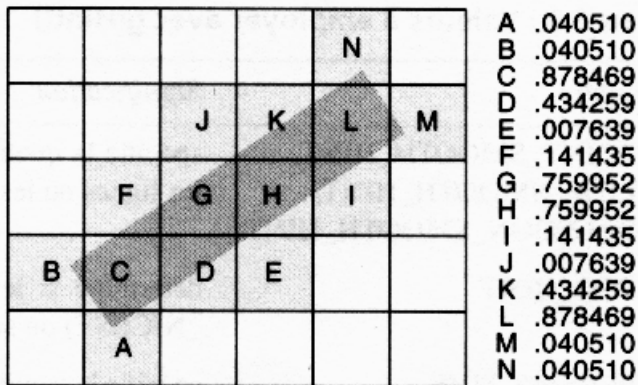
```

# IX. Lissage, fog et décalage de polygone

## Lissage (antialiasing)

### Définition et principe

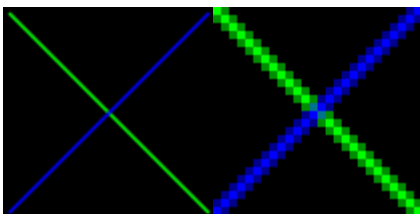
Les lignes qui ne sont ni horizontales ni verticales sont crénelées. Ce phénomène est appelé *aliasing*, et nous allons voir comment la technique de lissage (ou antialiasing) permet de l'estomper.



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 [aargb.c](#), 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 la qualité : `glHint(...)`

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 :

<code>GL_POINT_SMOOTH_HINT</code> , <code>GL_LINE_SMOOTH_HINT</code> , <code>GL_POLYGON_SMOOTH_HINT</code>	qualité d'échantillonnage souhaitée pour les points, les lignes ou les polygones au cours des opérations de lissage
<code>GL_FOG_HINT</code>	détermine si les calculs de fog se font pixel par pixel (meilleure qualité) ou sommet par sommet (plus rapide)
<code>GL_PERSPECTIVE_CORRECTION_HINT</code>	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.

```

/*
 * aargb.cpp
 * This program draws shows how to draw anti-
 * aliased lines. It draws two diagonal lines to form an
 * X; when 'r' is typed in the window, the lines are
 * rotated in opposite directions.
 */
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

static float rotAngle = 0.;

/* Initialize antialiasing for RGBA mode, including
 * alpha blending, hint, and line width. Print out
 * implementation specific info on line width granularity
 * and width. */
void init(void) {
    GLfloat values[2];
    glGetFloatv (GL_LINE_WIDTH_GRANULARITY,
    values);
    printf ("GL_LINE_WIDTH_GRANULARITY value
    is %3.1f\n", values[0]);

    glGetFloatv (GL_LINE_WIDTH_RANGE, values);
    printf ("GL_LINE_WIDTH_RANGE values are
    %3.1f %3.1f\n",
    values[0], values[1]);

    glEnable (GL_LINE_SMOOTH);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA);
    glHint (GL_LINE_SMOOTH_HINT,
    GL_DONT_CARE);
    glLineWidth (1.5);

    glClearColor(0.0, 0.0, 0.0, 0.0);
}

/* Draw 2 diagonal lines to form an X */
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f (0.0, 1.0, 0.0);
    glPushMatrix();
    glRotatef(-rotAngle, 0.0, 0.0, 0.1);
    glBegin (GL_LINES);
        glVertex2f (-0.5, 0.5);
        glVertex2f (0.5, -0.5);
    glEnd ();
    glPopMatrix();

    glColor3f (0.0, 0.0, 1.0);
    glPushMatrix();
    glRotatef(rotAngle, 0.0, 0.0, 0.1);
    glBegin (GL_LINES);
        glVertex2f (0.5, 0.5);
        glVertex2f (-0.5, -0.5);
    glEnd ();
    glPopMatrix();

    glFlush();
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (-1.0, 1.0,
        -1.0*(GLfloat)h/(GLfloat)w,
        1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
        1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 'r':
        case 'R':
            rotAngle += 20.;
            if (rotAngle >= 360.) rotAngle = 0.;
            glutPostRedisplay();
            break;
        case 27: /* Escape Key */
            exit(0);
            break;
        default:
            break;
    }
}

/* Main Loop
 * Open window with initial window size, title bar,
 * RGBA display mode, and handle input events.
 */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE |
    GLUT_RGB);
    glutInitWindowSize (200, 200);
    glutCreateWindow (argv[0]);
    init();
    glutReshapeFunc (reshape);
    glutKeyboardFunc (keyboard);
    glutDisplayFunc (display);
    glutMainLoop();
    return 0;
}

```



## Fog

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 oeuvre

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\(GLenum pname, TYPE \\*params\)](#). Un exemple est donné dans le programme [fog.c](#).

### Couleur et équation du fog

Soit  $C_f$  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  $C_s$  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 :

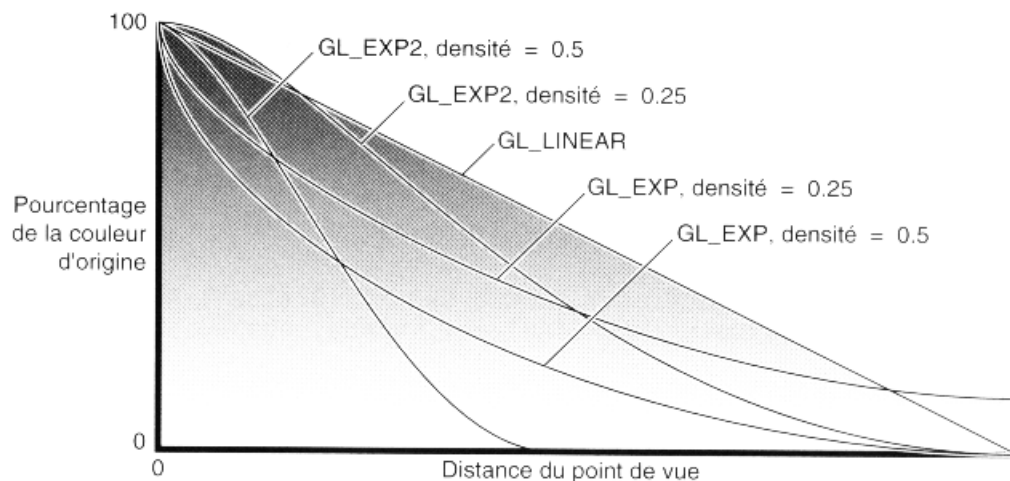
$$C = f(z).C_s + (1 - f(z)).C_f$$

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  $o$  :

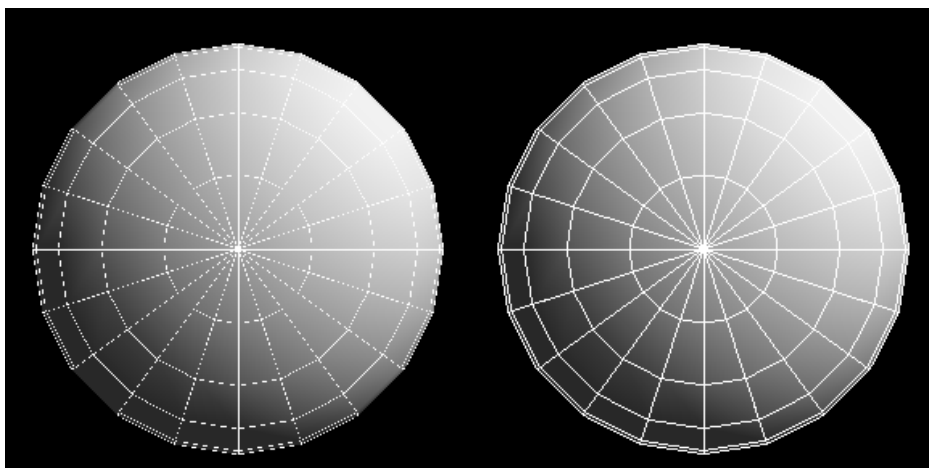
$$o = m \cdot \text{facteur} + r \cdot \text{unites}$$

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 :  $\text{facteur} = \text{unites} = 1.0$  . 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

C'est la démonstration classique, tirée du programme [polyoff.c](#) : dans l'image de gauche, sans décalage des lignes, dans l'image de droite, avec :



```

/*
 * polyoff.cpp
 * This program demonstrates polygon offset to draw
 a shaded polygon and its wireframe counterpart
 without ugly visual artifacts ("stitching").
 */
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>

GLuint list;
GLint spinx = 0;
GLint spiny = 0;
GLfloat tdist = 0.0;
GLfloat polyfactor = 1.0;
GLfloat polyunits = 1.0;

/* display() draws two spheres, one with a gray,
diffuse material,
 * the other sphere with a magenta material with a
specular highlight.
 */
void display (void) {
    GLfloat gray[] = { 0.8, 0.8, 0.8, 1.0 };
    GLfloat black[] = { 0.0, 0.0, 0.0, 1.0 };

    glClear (GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, tdist);
    glRotatef ((GLfloat) spinx, 1.0, 0.0, 0.0);
    glRotatef ((GLfloat) spiny, 0.0, 1.0, 0.0);

    glMaterialfv(GL_FRONT,
GL_AMBIENT_AND_DIFFUSE, gray);
    glMaterialfv(GL_FRONT, GL_SPECULAR, black);
    glMaterialf(GL_FRONT, GL_SHININESS, 0.0);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(polyfactor, polyunits);
    glCallList (list);
    glDisable(GL_POLYGON_OFFSET_FILL);

    glDisable(GL_LIGHTING);
    glDisable(GL_LIGHT0);
    glColor3f (1.0, 1.0, 1.0);
    glPolygonMode(GL_FRONT_AND_BACK,
GL_LINE);
    glCallList (list);
    glPolygonMode(GL_FRONT_AND_BACK,
GL_FILL);

    glPopMatrix ();
    glFlush ();
}

/* specify initial properties
 * create display list with sphere
 * initialize lighting and depth buffer
 */
void gfxinit (void) {
    GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    GLfloat global_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glClearColor (0.0, 0.0, 0.0, 1.0);

    list = glGenLists(1);
    glNewList (list, GL_COMPILE);
        glutSolidSphere(1.0, 20, 12);
    glEndList ();

    glEnable(GL_DEPTH_TEST);

    glLightfv (GL_LIGHT0, GL_AMBIENT,
light_ambient);
    glLightfv (GL_LIGHT0, GL_DIFFUSE,
light_diffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR,
light_specular);
    glLightfv (GL_LIGHT0, GL_POSITION,
light_position);
    glLightModelfv (GL_LIGHT_MODEL_AMBIENT,
global_ambient);
}

/* call when window is resized */
void reshape(int width, int height) {
    glViewport (0, 0, width, height);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(45.0,
(GLdouble)width/(GLdouble)height,
1.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

/* call when mouse button is pressed */
void mouse(int button, int state, int x, int y) {
    switch (button) {
        case GLUT_LEFT_BUTTON:
            switch (state) {
                case GLUT_DOWN:
                    spinx = (spinx + 5) % 360;
                    glutPostRedisplay();
                    break;
                default:

```

```

        break;
    }
    break;
    case GLUT_MIDDLE_BUTTON:
        switch (state) {
            case GLUT_DOWN:
                spiny = (spiny + 5) % 360;
                glutPostRedisplay();
                break;
            default:
                break;
        }
    break;
    case GLUT_RIGHT_BUTTON:
        switch (state) {
            case GLUT_UP:
                exit(0);
                break;
            default:
                break;
        }
    break;
    default:
        break;
}
}

void keyboard (unsigned char key, int x, int y) {
    switch (key) {
        case 't':
            if (tdist < 4.0) {
                tdist = (tdist + 0.5);
                glutPostRedisplay();
            }
            break;
        case 'T':
            if (tdist > -5.0) {
                tdist = (tdist - 0.5);
                glutPostRedisplay();
            }
            break;
        case 'F':
            polyfactor = polyfactor + 0.1;
            printf ("polyfactor is %f\n", polyfactor);
            glutPostRedisplay();
            break;
        case 'f':
            polyfactor = polyfactor - 0.1;
            printf ("polyfactor is %f\n", polyfactor);
            glutPostRedisplay();
            break;
        case 'U':
            polyunits = polyunits + 1.0;
            printf ("polyunits is %f\n", polyunits);
            glutPostRedisplay();
            break;
        case 'u':
            polyunits = polyunits - 1.0;
            printf ("polyunits is %f\n", polyunits);
            glutPostRedisplay();
            break;
        default:
            break;
    }
}

/* Main Loop
 * Open window with initial window size, title bar,
 * RGBA display mode, and handle input events.
 */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE |
        GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow(argv[0]);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutKeyboardFunc(keyboard);
    gfxinit();
    glutMainLoop();
    return 0;
}

```

# X. 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 *Z-Buffer*.
- 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 leurs utilisations

### *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.

### *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 de mask, 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.

### *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`](#)().

### *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).

## 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. Le fichier [dof.c](#) est un exemple de simulation de la profondeur de champ.

## Exercice

Simulez un effet de fondu-enchaîné sur une théière en rotation autour de son axe vertical.





# XI. Sélection et Picking

## Introduction

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).

## Détail 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

recupérer 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 :

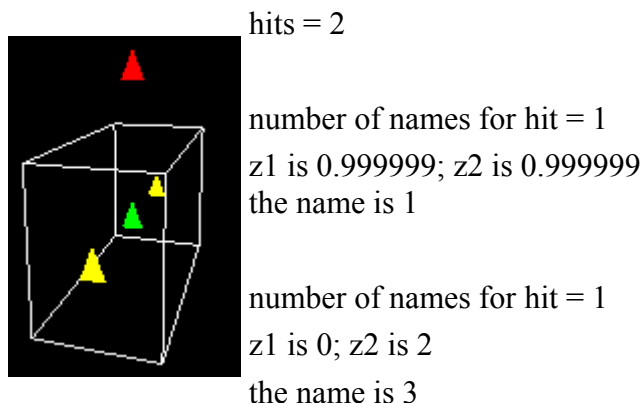
Nombre de noms dans la pile au moment du hit

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 [select.c](#) 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.



## 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 la fonction **pickSquares** qui est la fonction de rappel associée au clic souris qui déclenche le picking.

## Exercice

Créez un jeu dans lequel vous faites apparaître et disparaître rapidement des objets dans la scène à des positions, des dimensions et des orientations aléatoires. Grâce au picking, comptez un point pour chaque objet cliqué, et affichez le score au bout de 50 objets affichés.

```

/*
 * picksquare.cpp
 * Use of multiple names and picking are demonstrated.
 * A 3x3 grid of squares is drawn. When the left mouse
 * button is pressed, all squares under the cursor position have
 * their color changed. */
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

int board[3][3]; // amount of color for each square

/* Clear color value for every square on the board */
void init(void) {
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            for (j = 0; j < 3; j++)
                board[i][j] = 0;
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

/* The nine squares are drawn. In selection mode,
 * each square is given two names: one for the row and
 * the other for the column on the grid. The color of each
 * square is determined by its position on the grid, and
 * the value in the board[][] array. */
void drawSquares(GLenum mode) {
    GLuint i, j;
    for (i = 0; i < 3; i++) {
        if (mode == GL_SELECT)
            glLoadName (i);
        for (j = 0; j < 3; j++) {

```

```

    if (mode == GL_SELECT)
        glPushName (j);
    glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
              (GLfloat) board[i][j]/3.0);
    glRecti (i, j, i+1, j+1);
    if (mode == GL_SELECT) glPopName ();
}
}
}

/* processHits prints out the contents of the
 * selection array. */
void processHits (GLint hits, GLuint buffer[]) {
    unsigned int i, j;
    GLuint ii, jj, names, *ptr;

    printf ("hits = %d\n", hits);
    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++) { /* for each hit */
        names = *ptr;
        printf (" number of names for this hit = %d\n",
names); ptr++;
        printf (" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
        printf (" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
        printf (" names are ");
        for (j = 0; j < names; j++) { /* for each name */
            printf ("%d ", *ptr);
            if (j == 0) /* set row and column */
                ii = *ptr;
            else if (j == 1)
                jj = *ptr;
            ptr++;
        }
        printf ("\n");
        board[ii][jj] = (board[ii][jj] + 1) % 3;
    }
}

/* pickSquares() sets up selection mode, name stack,
and projection matrix for picking. Then the objects are
drawn. */
#define BUFSIZE 512

void pickSquares(int button, int state, int x, int y) {
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];

    if (button != GLUT_LEFT_BUTTON || state !=
GLUT_DOWN) return;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();

    glPushName(0);

    glMatrixMode (GL_PROJECTION);
    glPushMatrix ();
    glLoadIdentity ();
    // create 5x5 pixel picking region near cursor location
    gluPickMatrix ((GLdouble) x, (GLdouble)
viewport[3] - y), 5.0, 5.0, viewport);
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    drawSquares (GL_SELECT);

    glMatrixMode (GL_PROJECTION);
    glPopMatrix ();
    glutSwapBuffers ();

    hits = glRenderMode (GL_RENDER);
    processHits (hits, selectBuf);
    glutPostRedisplay();
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    drawSquares (GL_RENDER);
    glFlush();
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 27: exit(0); break;
    }
}

/* Main Loop */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE |
GLUT_RGB);
    glutInitWindowSize (100, 100);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutReshapeFunc (reshape);
    glutDisplayFunc(display);
    glutMouseFunc (pickSquares);
    glutKeyboardFunc (keyboard);
    glutMainLoop();
    return 0;
}

```