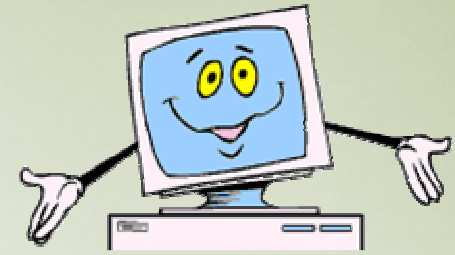


OpenGL



S P A C E W A R

Plan du cours



1. Introduction et objectifs.
2. Les bases d'OpenGL et glut.
3. Processus de visualisation en OpenGL
4. Un cube tournant.
5. Les transfo géométriques.
6. Eclairage et matériaux.
7. Les textures.
8. Génération de terrains.

1

Introduction et objectifs.

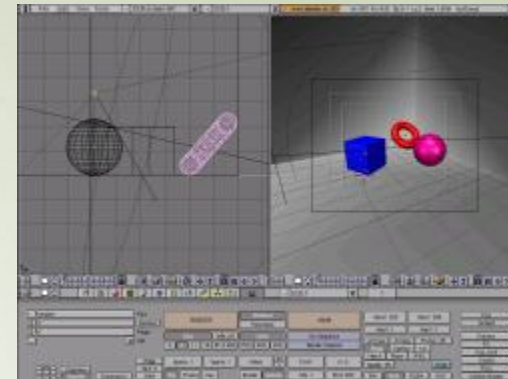


Introduction

Aujourd'hui, lorsqu'on parle de bibliothèque de développement 3D, les mots qui reviennent le plus souvent sont **Direct3D** et **OpenGL**. Direct3D est une API (Application Programming Interface) développée par Microsoft pour son OS Windows. Propriétaire et non portable, Direct3D fait partie de l'ensemble DirectX et est relativement orienté 'Jeux vidéo'. Son principal avantage est d'être bien supporté par les constructeurs de cartes graphiques grand public. Pour sa part, **OpenGL** (Open Graphic Library, ce que se traduit en bon français par 'Bibliothèque Graphique Ouverte') est une API développée depuis 1992 par Silicon Graphics (<http://www.sgi.com>), un des grands ténors de l'informatique graphique professionnelle.

Introduction

OpenGL en action dans Blender



Originellement développé pour les stations graphiques haut de gamme Silicon Graphics, l'aspect ouvert d'**OpenGL** lui a valu d'être porté sur la plupart des plateformes modernes, et sur la plupart des systèmes d'exploitation. Aujourd'hui, OpenGL est reconnu par les professionnels comme un standard, et est utilisé par tous les logiciels professionnels de synthèse d'images (3DSMax, Maya, Softimage...), de CAO (ProEngineer, Catia...), mais aussi dans le domaine des jeux vidéos 3D (Quake (1,2,3), Unreal...)

Fonctionnalités d'OpenGL La première question qui vient évidemment à l'esprit est la suivante : **Que peut on faire avec OpenGL ?** La réponse est la suivante : OpenGL permet de représenter à l'écran des scènes tridimensionnelles réalistes. Le terme 'réaliste' est ici tout a fait relatif : on peut considérer comme réaliste une image de synthèse ou tous les objets représentés sont correctement placés au niveau de la perspective sans que les couleurs correspondent à ce qu'on observe dans la nature. Le réalisme ultime peut être atteint lorsqu'il est impossible à première vue de savoir si une image est issue d'un calcul ou si elle simplement une photo numérisée (on parle alors de photoréalisme). Au risque d'en décevoir certains, OpenGL ne vous permettra pas de générer des images photoréalistes.

OpenGL est résolument orienté vers l'interaction 'temps réel', c'est à dire qu'un programme conçu avec OpenGL permet en général à l'utilisateur d'agir par l'intermédiaires de périphériques d'entrée (clavier, souris, trackball...) sur la scène (déplacement de caméra, changement de focale, déplacement des objets....), et d'en visualiser immédiatement l'effet. Ainsi, lorsque vous jouez à Quake, chaque fois que vous déplacez le joueur le programme recalcule la nouvelle image et l'affiche à l'écran. Pour que le jeu vidéo soit 'fluide', c'est-à-dire que les déplacements du joueur ne paraissent pas trop saccadés, le programme doit être capable de calculer et d'afficher une image en moins d'un dixième de secondes. Dans le cas d'un jeu vidéo on peut considérer que si l'image n'est pas recalculée une quarantaine de fois par secondes, le jeu ne sera pas fluide.

Bien que non orienté photoréalisme, OpenGL permet d'obtenir des rendus tout à fait satisfaisants. Voici une liste non exhaustive des principales fonctionnalités d'OpenGL. Ne vous inquiétez pas si vous ne comprenez pas tous les termes énoncés, la plupart des termes seront abordés dans les didacticiels.

- Affichage de maillages polygonaux.
- Rendu par projection orthogonale ou perspective.
- Placage de texture.
- Simulation d'éclairages réalistes
- Ombre portées
- Insertion d'objets bitmaps en avant plan (images, polices)
- Effet de brouillard, de flou
- Gestion des courbes et surfaces gauches (Splines, Nurbs...)
- ...

Nous avons vu précédemment qu'OpenGL a été porté sur toutes les grandes plates-formes actuelles, et bien évidemment, linux en fait partie. Il existe plusieurs implémentations d'OpenGL sous linux, certaines sont libres, d'autres sont commerciales. La plus connue est **Mesa**, libre, le code source est disponible sur le site <http://www.mesa3d.org>. Bien que Mesa ne soit pas une implémentation officielle d'OpenGL, dans le sens où l'auteur de Mesa n'a pas payé la licence OpenGL à SGI, cette API reste une très bonne alternative à un OpenGL 'Officiel', et en reprend les noms de commandes avec l'autorisation de Silicon Graphics. Seul quelques rares fonction de la version 1.2 ne sont pas disponibles dans Mesa.

Il existe une ombre qui plane sur OpenGL sous linux.

Sous environnement Windows tout se passe bien. Chaque constructeur fournit avec la carte les pilotes permettant de bénéficier de l'accélération matérielle. Mais sous linux, rares sont ceux qui le font. Bien sûr, de nombreuses âmes généreuses seraient prêtes à écrire elles même ces pilotes pour linux, mais pour cela, il leur faut disposer des spécifications techniques de la carte, ceux que les constructeurs rechignent à faire, de peur de voir leur technologie pillée.

Cependant, il existe quelque projets tentant de fournir des drivers pour certaines cartes. Ainsi, les possesseurs de carte 3DFX pourront trouver leur bonheur sur le site <http://www-hmw.caribel.pisa.it/fxmesa/index.shtml>

Les processeurs de carte Matrox G200 et G400, ATI RagePro, S3Virge et Savage3D, Intel i810, Sis 6326 se référeront au projet Utah-GLX : <http://utah-glx.sourceforge.net>

Félicitons le constructeur NVidia, qui fournit les pilotes linux pour ses cartes graphiques TNT, TNT2, TNT Ultra, GeForce, et Quadro : <http://www.nvidia.com/Products/Drivers.nsf/Linux.html>

Enfin, notons que depuis sa version 4, le serveur X Xfree86 intègre des fonctions d'accélération matérielle qui vont, espérons le, enfin nous faire voir le bout du tunnel. <http://www.xfree86.org>

Glut

Pour créer un programme OpenGL dans un environnement fenêtré comme X-Window, il faut être en mesure de créer une fenêtre pour y afficher nos images, de la détruire, et de gérer les interactions avec l'utilisateur par le clavier, la souris et tous les autres types de périphériques d'entrée. OpenGL se voulant indépendant de toute plate-forme matérielle, l'API ne fournit pas de telles fonctions. Heureusement, il existe une bibliothèque annexe, nommée **Glut** (GL Utility Toolkit) qui fournit ces fonctions. Elle est développée par Mark Kilgard et est disponible sur le site <http://reality.sgi.com/mjk/glut3>. Glut est fourni avec les dernières moutures de Mesa.

Voici une liste de toutes les fonctionnalités proposées par Glut :

- Gestion de fenêtres.
- Gestion des événements par fonctions de rappel.
- Gestion de périphériques d'entrée exotiques (spaceballs...).
- Gestion des polices de caractères.
- Fonctions de création de menus.

Bibliothèques d'OpenGL et la commence la SF

(1) OpenGL Library (GL)

Librairie standard proposant les fonctions de base pour l'affichage en OpenGL

Pas de fonction pour la construction d'une interface utilisateur (fenêtres, souris, clavier, ...)

Préfixe des fonctions: gl

2) OpenGL Utility Library (GLU)

Librairie standard proposant des commandes bas-niveau écrites en GL:

certaines transformations géométriques

triangulation des polygones

rendu des surfaces paramétriques et quadriques

...

Préfixe des fonctions: glu

(3) OpenGL extension to X-Windows (GLX)

Utilisation d'OpenGL en association avec X Windows pour l'interface utilisateur

Préfixe des fonctions: glX

(4) Auxiliary Library

Bibliothèque écrite pour rendre simple la programmation de petites applications dans le cadre d'interfaces graphiques interactives simples:

- gestion d'une fenêtre d'affichage
- gestion de la souris
- gestion du clavier

...

Préfixe des fonctions: aux

(5) OpenGL Tk Library (GLTk)

Équivalent à Aux mais avec l'utilisation de TclTk pour l'interface graphique

Préfixe des fonctions: tk

(6) OpenGL Utility Toolkit (GLUT)

Équivalent à Aux

Interface utilisateur programmable plus élaborée (multifenêtrage, menus popup) -> plus grande complexité

Préfixe des fonctions: glut

L'Auxiliary Library: Aux

Facilite la programmation d'OpenGL au moyen d'un jeu de fonctions d'interfaçages vis à vis du système d'exploitation de l'ordinateur (Interface graphique).

Bibliothèque parfois « capricieuse »

L'Utility Toolkit: GLUT

Programmation presque identique à celle réalisée au moyen de Aux.

Fonctionnalités supplémentaires:

- gestion des menus,
- gestion des environnements multifenêtres,
- existence de polices de caractères bitmap et vectorielles intégrées,
- gestion de périphériques d'entrée supplémentaires

Conclusion

Vous voilà armés pour vous attaquer à la programmation OpenGL. Vous savez maintenant tout ce que vous deviez savoir avant de rentrer dans le vif du sujet. En ce qui concerne les prérequis mathématiques, si vous avez quelques notions de géométrie dans l'espace, vous ne devriez pas avoir trop de problèmes. Cependant, il se peut que quelques notions de calcul matriciel vous soient utiles.

Il ne vous reste maintenant plus qu'à vous attaquer au cours. Bon courage.

2

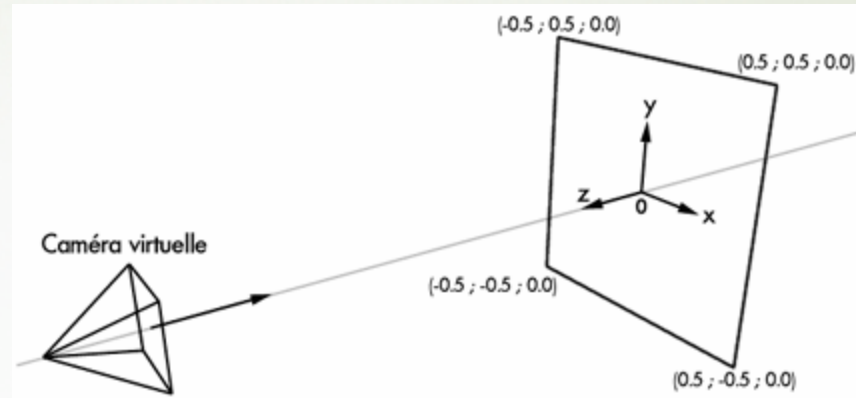
Les bases d'OpenGL



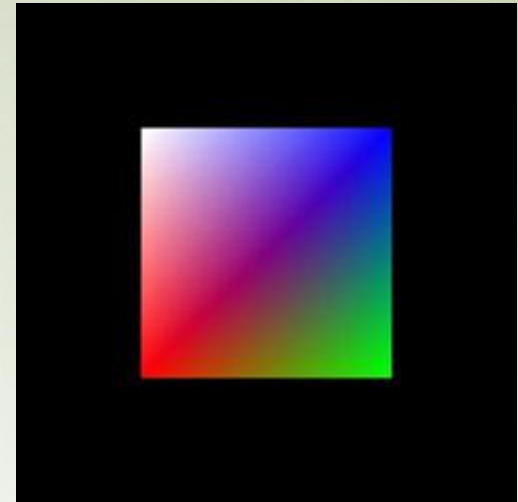
Après notre présentation d'OpenGL, passons à la pratique, avec l'étude d'un exemple classique qui peut être considéré comme le "Hello World" d'OpenGL. Le but est d'aborder les mécanismes standards de la programmation OpenGL en C : création de fenêtre avec glut, gestion des événements, description de la scène 3D.

L'exemple que nous allons étudier est relativement simple. Notre but est d'afficher à l'écran un carré coloré. En guise de petit plus, et afin d'aborder la gestion des événements, nous donnerons à l'utilisateur la possibilité de modifier la méthode d'affichage du carré (carré plein, fil de fer, ou sommets seuls) par l'intermédiaire du clavier. Pour l'instant, nous nous contenterons d'un carré en deux dimensions, car un passage à la troisième dimension pour l'affichage d'un cube nous obligerait à considérer des notions de projection perspective et de placement de la caméra virtuelle qui feront l'objet d'autres chapitres.

Avant de commencer à coder, je vous propose de visualiser sur la figure suivante la scène que devra représenter notre petit programme.



La scène finale sera.



Préparatifs :

A la manière d'un modèleur 3D, nous utiliserons en OpenGL le concept de caméra virtuelle. Virtuelle parce qu'elle n'est pas à proprement parler un objet de la scène 3D, mais elle nous indique le point de vue de l'observateur, la direction du regard et l'inclinaison de la caméra. Nous utiliserons dans un premier temps la caméra par défaut d'OpenGL.

Conventions de noms de fonctions

Les fonction OpenGL sont régies par un certain nombre de règles :

Tous les noms de fonctions OpenGL sont préfixés : les noms de fonction OpenGL commencent par 'gl'. Pour les bibliothèques annexes telles que GLU et GLX, les préfixes respectifs sont 'glu' et 'glX'. En guise d'exercice, je vous laisse déterminer le préfixe pour les fonctions glut :-).

Certaines commandes OpenGL existent en plusieurs variantes, suivant le nombre et le type de paramètres qu'admet la variante. Ainsi, la fonction glVertex3f() prend trois paramètres ('3') de type Float ('f'), alors que glVertex2i() fonctionne avec 2 entiers.

Certaines fonctions possèdent une version suffixée par un 'v'. Les paramètres sont passés à ces fonctions par l'intermédiaire d'un pointeur sur un tableau.

OpenGL redéfinit des types de données numériques : GLint correspond aux entiers, GLfloat aux flottants, GLbyte aux caractères non signés... Il est préférable d'utiliser ces types de données plutôt que les types standards du C.

A l'attaque !

Votre session devC++ est ouverte ? Allons-y !

Première chose à faire : inclure les fichiers d'en-tête OpenGL. Il en existe plusieurs, mais heureusement glut les appelle pour nous. On se contentera donc d'un simple

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

La bibliothèque OpenGL est conçue comme une machine à états. Par conséquent, une des premières choses à faire est d'initialiser cette machine. Le nombre de variables d'état accessibles est assez impressionnant. Le mode de dessin, la couleur de fond et le type d'éclairage en sont des exemples. Nous introduirons progressivement les états les plus couramment utilisés. Bien sûr, OpenGL dispose d'un état par défaut, et nous nous contenterons donc dans la phase d'initialisation de régler 2 états : la couleur de fond, et la taille d'un point. Les fonctions à utiliser sont les suivantes:

```
void glClearColor(GLclampf rouge, GLclampf vert, GLclampf bleu,  
GLclampf alpha);
```

```
void glPointSize(GLfloat taille);
```


`glClearColor()` permet de spécifier la couleur de remplissage utilisée lors d'un effacement de la scène. On peut donc l'assimiler à la couleur du fond. Le mode de spécification de couleur est le classique RGB avec une composante supplémentaire alpha, utilisée pour la gestion de la transparence des objets. Nous ne l'utiliserons pas ici et lui affecterons systématiquement la valeur 0. Chacune des 4 composantes doit être comprise entre 0 et 1. Ainsi `glClearColor(1.0,1.0,1.0,0.0)` nous donne un fond blanc, et `glClearColor(0.0,0.0,0.0,0.0)` un fond noir.

Par défaut, lorsqu'on choisit de représenter les objets simplement par leurs sommets, la taille des sommets à l'écran est de 1 pixel. Afin de les rendre plus visible, nous réglerons la taille des sommets à 2 pixels :

`glPointSize(2.0);`


```
int main(int argc, char** argv)
{
    /* Initialisation d'OpenGL */
    glClearColor(0.0,0.0,0.0,0.0);
    /* On passe à 2 pixels pour des raisons de clarté */
    glPointSize(2.0);
}
```



Initialisation de glut et création de la fenêtre :

Comme vous nous l'avons vu c'est glut qui gère les interactions entre OpenGL et le système. La phase d'initialisation va nous permettre de créer notre fenêtre. Un des atouts de glut est sa grande simplicité. L'initialisation de glut se fait de la manière suivante :

```
glutInit(&argc,argv);  
glutInitDisplayMode(GLUT_RGB);  
glutInitWindowPosition(200,200);  
glutInitWindowSize(250,250);  
glutCreateWindow(« Exemple 1");
```

```
int main(int argc, char** argv)
{
    /* initialisation de glut et création de la fenêtre */

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowPosition(200,200);
    glutInitWindowSize(250,250);
    glutCreateWindow("exemple 1");

    /* Initialisation d'OpenGL */
    glClearColor(0.0,0.0,0.0,0.0);
    /* On passe à une taille de 2 pixels pour des raisons de clarté */
    glPointSize(2.0);
}
```

glutInit analyse les données passées en paramètres au programme et initialise la bibliothèque glut, notamment en établissant la communication avec le système.

La fonction **glutInitDisplayMode** est celle qui vous paraîtra sans doute la plus obscure. Elle permet de régler les paramètres liés à l'affichage : type d'image (palette indexée ou RVB), utilisation d'un tampon de profondeur (plus connu sous le nom de Z-buffer), utilisation du double buffering... Aujourd'hui un simple tampon d'image en mode RVB défini par la constante GLUT_RGB nous suffira.

Bien que cela soit évident, je vais quand même préciser que les fonctions `glutInitWindowPosition()` et `glutInitWindowSize()` permettent respectivement de définir la position du coin supérieur gauche de la fenêtre OpenGL par rapport au coin supérieur gauche de l'écran, et de spécifier la largeur et la hauteur de la fenêtre.

La fenêtre est créée grâce à l'appel à `glutCreateWindow()` et son nom doit être transmis sous forme de chaîne de caractères. Il s'agit du nom qui apparaîtra notamment dans le bandeau de la fenêtre, gérée par le window manager. Si la fenêtre est bel et bien créée, elle ne sera affichée à l'écran que lors de l'entrée dans la boucle de gestion des événements.

Mise en place des fonctions de rappel :

Le système de gestion des événements offert par glut est relativement simple. Des fonctions sont associées aux différents types d'événements envoyés par le système, puis une boucle d'attente est lancée. A chaque fois qu'un événement est émis par le système, la fonction de rappel associée à l'événement est appelée. Vous devez associer au moins une fonction de rappel, la fonction d'affichage, qui est celle dans laquelle vous devez décrire votre scène 3D. Vous pouvez associer des fonctions de rappel aux événements liés :

- au clavier
- à la souris
- à des périphériques d'entrée répandus en infographie (spaceballs, boîtes de potentiomètres, tablettes graphiques...)
- à la modification de la configuration de la fenêtre

Il existe par ailleurs deux fonctions de rappel spéciales qui permettent d'enregistrer des fonctions qui seront appelées à intervalles de temps réguliers ou pendant les temps d'oisiveté (idle), c'est-à-dire lorsque le gestionnaire n'a aucun événement à traiter. Ces deux fonctions sont extrêmement pratiques pour la création de scènes animées.

Dans notre programme, en plus de la fonction d'affichage obligatoire, nous allons mettre en place une fonction de rappel pour le clavier. Les deux appels suivants permettent de spécifier quelles seront les fonctions respectivement associées à l'affichage et au clavier :

```
glutDisplayFunc(affichage);    glutKeyboardFunc(clavier);
```

Après cela, il ne nous reste plus qu'à lancer la boucle d'attente des événements : `glutMainLoop();`


```
int main(int argc, char** argv)
{
/* initialisation de glut et création de la fenêtre */

glutInit(&argc,argv);
glutInitDisplayMode(GLUT_RGB);
glutInitWindowPosition(200,200);
glutInitWindowSize(250,250);
glutCreateWindow("ogl1");

/* Initialisation d'OpenGL */
glClearColor(0.0,0.0,0.0,0.0);
/* On passe à une taille de 2 pixel pour des raisons de clarté */
glPointSize(2.0);

/* enregistrement des fonctions de rappel */
glutDisplayFunc(affichage);
glutKeyboardFunc(clavier);

/* entrée dans la boucle principale de glut */
glutMainLoop();
}
```

La fonction d'affichage :

La fonction d'affichage constitue le cœur du programme. C'est ici que nous allons décrire la scène. Pour être plus exact, la fonction d'affichage contient la procédure à appliquer pour redessiner notre scène dans la fenêtre, en commençant par un remplissage de la fenêtre avec la couleur de fond que nous avons défini dans la phase d'initialisation. On utilise pour cela

void glClear(GLbitfield masque);

OpenGL travaille avec plusieurs zone tampons en plus de la zone d'image (tampon de profondeur, d'accumulation...). Le paramètre masque permet de spécifier les tampons que l'on souhaite effacer. Nous n'utilisons aujourd'hui que le tampon d'image. Notre masque vaudra **GL_COLOR_BUFFER_BIT**.

Il nous faut ensuite décrire la scène 3D. La méthode de représentation d'une scène d'OpenGL est la plus classique qui soit :

- Une scène est constituée d'objets.
- Un objet est défini par un ensemble de polygones.
- Un polygone est un ensemble de points de l'espace reliés entre eux par des arêtes.

Pour spécifier notre scène 3D, nous allons devoir énumérer la liste des polygones. La méthode est la suivante : on indique le début de la description du polygone, on explicite chaque point du polygone, puis on indique la fin de l'énumération. À chaque déclaration de points, on peut modifier certaines variables d'état, comme la couleur active. Notre scène ne comportant qu'un polygone, la description sera vite faite.

```
glBegin(GL_POLYGON);
```

```
    glColor3f(1.0,0.0,0.0);
```

```
    glColor3f(0.0,1.0,0.0);
```

```
    glColor3f(0.0,0.0,1.0);
```

```
    glColor3f(1.0,1.0,1.0);
```

```
glEnd();
```

```
    glVertex2f(-0.5,-0.5);
```

```
    glVertex2f(0.5,-0.5);
```

```
    glVertex2f(0.5,0.5);
```

```
    glVertex2f(-0.5,0.5);
```



Chaque description de polygone commence par un appel à `void glBegin(GLenum mode);`

Le paramètre mode auquel nous donnons la valeur `GL_POLYGON` indique la technique utilisée pour relier par des arêtes les différents points du polygone. En mode `GL_POLYGON`, chaque sommet est relié à son prédécesseur, et pour fermer le polygone, le dernier point est relié au premier.

La primitive de spécification d'un sommet du polygone est `glVertex()`. comme nous travaillons dans le plan d'équation $z=0$, nous utiliserons la variante à deux arguments de type `GLfloat` :

`void glVertex2f(GLfloat x, GLfloat y);` Bien entendu, `x` et `y` sont les coordonnées cartésiennes du sommet. La commande `glColor3f()` modifie la couleur dite active. Son prototype est :

`void glColor3f(GLfloat r, GLfloat v, GLfloat b);`

La fin de la description d'un polygone est marquée par un simple **glEnd()**. Dans le cadre d'un cube, il faudrait réitérer 5 fois l'opération de description de polygone. Pour nous simplifier la tâche, OpenGL dispose d'un certain nombre des fonctionnalités comme des liste d'affichage.

Pour terminer notre fonction d'affichage, un **glFlush()** permet de s'assurer que toutes les commandes ont bien été transmises au serveur.

```
void affichage(void)
{
    /* initialisation des pixels */
    glClear (GL_COLOR_BUFFER_BIT);
    /* dessin d'un polygone (rectangle) avec les sommets en (0.25, 0.25, 0.0) et
       (0.75, 0.75, 0.0) */
    glBegin(GL_POLYGON);
        glColor3f(1.0,0.0,0.0);
        glVertex2f(-0.5,-0.5);
        glColor3f(0.0,1.0,0.0);
        glVertex2f(0.5,-0.5);
        glColor3f(0.0,0.0,1.0);
        glVertex2f(0.5,0.5);
        glColor3f(1.0,1.0,1.0);
        glVertex2f(-0.5,0.5);
    glEnd();

    /* On force l'affichage du résultat */
    glFlush();
}
```


La fonction de rappel du clavier :

```
void clavier(unsigned char touche,int x,int y);
```

Lorsque l'utilisateur appuiera sur une touche du clavier, la boucle de gestion effectuera un appel à `clavier()`. La variable `touche` contiendra le caractère correspondant à la touche pressée, `x` et `y` indiqueront la position de la souris lors de la frappe sur le clavier. `x` et `y` sont données relativement à la fenêtre OpenGL. Dans la plupart des cas, la fonction de rappel pour les événements de type clavier est constituée d'une structure `switch` dont la clause de test porte sur le caractère passé en paramètre.

```
void clavier(unsigned char touche,int x,int y)
{
switch (touche)
{
case 'p': /* affichage du carre plein */
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glutPostRedisplay(); break;
case 'f': /* affichage en mode fil de fer */
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glutPostRedisplay(); break;
case 's' : /* Affichage en mode sommets seuls */
    glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
    glutPostRedisplay(); break;
case 'q' : /*la touche 'q' pour quitter le programme */
    exit(0);
}
}
```

Chaque polygone possède une face avant et une face arrière définie par l'ordre de saisie des sommets du polygone. Il est possible d'associer un mode de dessin différent pour la face avant et arrière d'un polygone. Le paramètre face indique la face dont on veut changer le mode de dessin :

GL_FRONT : face avant

GL_BACK : face arrière

GL_FRONT_AND_BACK : les deux faces

Le paramètre mode définit le mode de dessin souhaité pour la ou les faces choisies :

GL_FILL : le polygone est entièrement rempli.

GL_LINE : seuls les contours du polygone sont dessinés (mode fil de fer)

GL_POINT : seuls les sommets du polygone sont affichés.

Voici le source complet [ici](#)

glutglutons

L'Utility Toolkit: GLUT

Elle facilite la programmation d'applications utilisant OpenGL au moyen d'un jeu de fonctions d'interfaçage vis à vis du système d'exploitation de l'ordinateur (Interface graphique).

Programmation événementielle: Une boucle infinie avec gestion par le dispositif GLUT d'une file d'attente d'événements et exécution de fonctions prédéfinies ou paramétrables par le programmeur en réaction à ces événements.

Fonctionnalités principales

- Définition et ouverture d'une ou plusieurs fenêtres d'affichage.
- Définition de fonctions devant s'exécuter en réaction à des événements du type:
 - rafraîchissement de l'image,
 - modification de la taille de la fenêtre d'affichage,
 - utilisation de la souris dans la zone d'affichage,
 - utilisation du clavier,
 - ...
- Création de menus et définition de fonctions exécutées en association avec des entrées dans ces menus.

Glut: Initialisation et ouverture d'une fenêtre d'affichage

void glutInit(int *argc, char **argv);

Initialise la bibliothèque GLUT. Traite les options de ligne de commande (celles reçues en entête de fonction main) (options non décrites ici).

void glutInitDisplayMode(unsigned int b);

Définit le type de fenêtre d'affichage par composition des constantes suivantes:

- GLUT_RGBA ou GLUT_INDEX (vraies couleurs ou couleurs indexées),
- GLUT_SINGLE ou GLUT_DOUBLE (simple ou double buffer),
- GLUT_DEPTH, GLUT_STENCIL et GLUT_ACCUM (utilisation de tampons profondeur, stencil et/ou accumulation).

void glutInitWindowPosition(int x, GLint y);

Définit la position de la fenêtre sur l'écran.

void glutInitWindowSize(int larg, int haut);

Définit la taille de la fenêtre à l'écran.

int glutCreateWindow(char *t);

Crée une fenêtre avec le titre t selon les paramètres définis lors de l'exécution des fonctions précédentes.

La fenêtre n'est affichée qu'ultérieurement lors de l'exécution de glutMainLoop.

La fonction rend un entier désignant la fenêtre ouverte.

Glut: Affichage principal

void glutDisplayFunc(void (*displayFunc)(void));

Définit la fonction exécutée automatiquement par GLUT quand un événement intervient entraînant le rafraîchissement de l'image affichée (création initiale de la fenêtre d'affichage, déplacement, agrandissement, réduction, réactivation en avant-plan, ordre par programme, ...).

La fonction display passée en paramètre ne doit assurer l'affichage que d'une seule image. Son prototype doit obligatoirement correspondre à `void (*display)(void)`.

void glutMainLoop(void);

Fonction assurant la boucle principale d'affichage. La fonction glutMainLoop est bloquante. Elle gère les événements liés au fonctionnement du programme et exécute les fonctions associées par le programmeur à ces événements -> simili programmation événementielle.

void glutPostRedisplay(void);

Fonction plaçant en file d'attente d'événements un ordre de rafraîchissement de la fenêtre d'affichage active.

void glutPostWindowRedisplay(int handle);

Fonction plaçant en file d'attente d'événements un ordre de rafraîchissement de la fenêtre d'affichage désignée par le handle.

void glutSwapBuffers(void);

Fonction assurant le swap entre les deux buffers d'affichage dans le cas où le double buffering est utilisé.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
/* attention à l'ordre de déclaration des inclusions sous windows */

void display(void) { /* dessin de la scène */ }

void myinit(void)
{ /* initialisation du programme, */
  /* appels de fonctions OpenGL */
  /* réalisés une fois et une seule */
}

int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowPosition(10, 10);
  glutInitWindowSize(500, 500);
  glutCreateWindow(argv[0]);
  glutDisplayFunc(display);
  myinit();
  glutMainLoop();
  return(0);
}
```

ceci n'est que du pseudo code

Le programme est [ici](#)

Gestion d'un processus en tâche de fond

void glutIdleFunc(void *func(void));

Définit une fonction à exécuter par GLUT s'il n'y a pas d'événement en attente et donc que l'application est disponible pour la réalisation d'une tâche annexe
-> exécution d'actions en tâche de fond.

Cette fonction est fréquemment utilisée pour programmer des animations. Dans ce cadre, elle est généralement dédiée à la modification de variables globales employées dans la fonction display et ne contient pas d'appel de fonction OpenGL. Il lui est possible de demander l'affichage d'une nouvelle image via glutPostRedisplay ou glutPostWindowRedisplay générant ainsi l'animation car chaque idle entraîne un display et idle est lancé de manière récurrente quand rien d'autre n'est à réaliser.

Le prototype de la fonction en paramètre doit obligatoirement correspondre à void (*idle)(void).

Le programme est [ici](#)

Gestion des redimensionnements fenêtre

```
void glutReshapeFunc(void (*fonct)(int larg,int haut));
```

Définit la fonction exécutée automatiquement par GLUT avant le rafraîchissement (lui aussi automatique) d'une fenêtre dont la taille est modifiée.

Cette fonction "reshape" est habituellement employée pour configurer deux caractéristiques liées à la visualisation:

- la zone de la fenêtre d'affichage dans laquelle l'image est affichée,
- le mode de représentation (projection parallèle ou en perspective) et ses paramètres, et éventuellement pour amorcer le dessin de la scène en fixant un point de vue. Elle doit obligatoirement avoir en entête deux paramètres de type entier qui représentent les tailles en x et en y de la fenêtre après changement de dimension.

Ces paramètres permettront éventuellement à la fonction reshape d'adapter son action aux nouvelles dimensions de la fenêtre.

Le programme est [ici](#)

Gestion des événements clavier

void glutKeyboardFunc(void (*fonct) (unsigned char key,int x,int y))

Définit la fonction exécutée automatiquement par GLUT lorsqu'une touche ASCII du clavier est frappée. Au moment de son exécution, cette fonction recevra les paramètres key, x et y contenant respectivement le code ASCII de la touche de clavier et les positions instantanées en x et en y de la souris relativement à la fenêtre au moment de la frappe clavier.

void glutSpecialFunc(void (*fonct)(int key,int x,int y))

Définit la fonction exécutée automatiquement par GLUT lorsqu'une touche de fonction ou une touche de curseur du clavier est frappée. Au moment de son exécution, cette fonction recevra les paramètres key, x et y contenant respectivement le code de la touche de clavier (GLUT_KEY_F1 à GLUT_KEY_F12, GLUT_KEY_LEFT, GLUT_KEY_RIGHT, GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_PAGE_DOWN, GLUT_KEY_PAGE_UP, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT) et les positions instantanées en x et en y de la souris relativement à la fenêtre.

Le programme est [ici](#)

Gestion des événements souris

```
void glutMouseFunc(void (*fonct)(int bouton,int etat,int  
x,int y));
```

Définit la liaison entre les clics de boutons de la souris et une fonction -> fonction exécutée automatiquement par GLUT quand les boutons de la souris sont utilisés. Cette fonction est exécutée avec un jeu de paramètres fonction de l'utilisation réalisée de la souris:

- bouton: GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON ou GLUT_RIGHT_BUTTON
- etat: GLUT_UP ou GLUT_DOWN
- x,y: position de la souris dans la fenêtre au moment de l'événement

void glutMotionFunc(void (*fonct)(int x,int y));

Définit la fonction exécutée automatiquement par GLUT quand la souris se déplace devant la fenêtre avec un bouton appuyé.

x et y sont les positions de la souris dans la fenêtre au moment de l'appel de la fonction.

void glutPassiveMotionFunc(void (*fonct)(int x,int y));

Définit la fonction exécutée automatiquement par GLUT quand la souris se déplace devant la fenêtre sans bouton appuyé.

x et y sont les positions de la souris dans la fenêtre au moment de l'appel de la fonction.

Le programme est [ici](#) et [la](#)

Dessin d'objets 3D en fil de fer ou en volumique

OpenGL permet possède de base un certain nombre de forme prédéfinies. Nous allons rapidement lister l'ensemble de ces modèles puis nous écrirons un programme qui les affichera tous.

Vous aurez sans doute besoin de ces différentes [biblio](#)

Objet

Sphère

Rayon
Nombre sections
Nombre coupes

```
glutWireSphere(double r,  
                int ns,  
                int nc);  
glutSolidSphere(double r),  
                int ns,  
                int nc);
```

Cube

Coté

```
glutWireCube(double c);  
glutSolidCube(double c);
```

Tore

Rayon intérieur
Rayon extérieur
Nombre côtés
Nombre anneaux

```
glutWireTorus(double rint,  
              double rext,  
              int ns,  
              int nr);  
glutSolidTorus(double rint,  
              double rext,  
              int ns,  
              int nr));
```

Cône

Rayon
Hauteur
Nombre sections
Nombre coupes

```
glutWireCone(double r,  
             double h,  
             int ns,  
             int nc);  
glutSolidCone(double r,  
             double h,  
             int ns,  
             int nc);
```

Tétraèdre
4 faces

```
glutWireTetrahedron(void);  
glutSolidTetrahedron(void);
```

Octaèdre
8 faces

```
glutWireOctahedron(void);  
glutSolidOctahedron(void);
```

Icosaèdre
12 faces

```
glutWireIcosahedron(void);  
glutSolidIcosahedron(void);
```

Dodécaèdre
20 faces

```
glutWireDodecahedron(void);  
glutSolidDodecahedron(void);
```

Teapot
Rayon

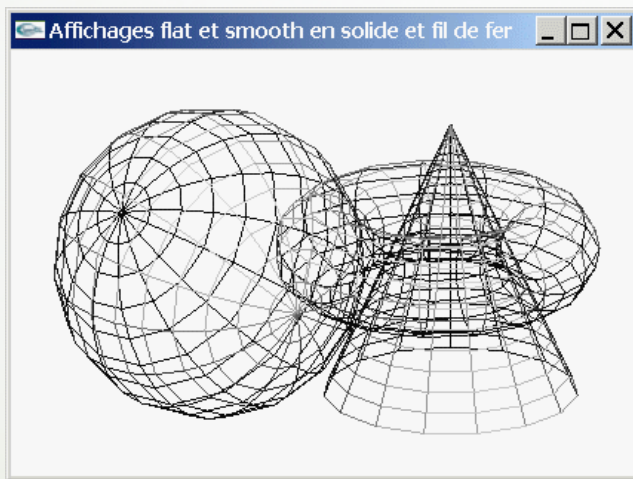
```
glutWireTeapot(double s);  
glutSolidTeapot(double s);
```

Vers un programme qui présente tous les objets : [ici](#)

Fonctions génériques:

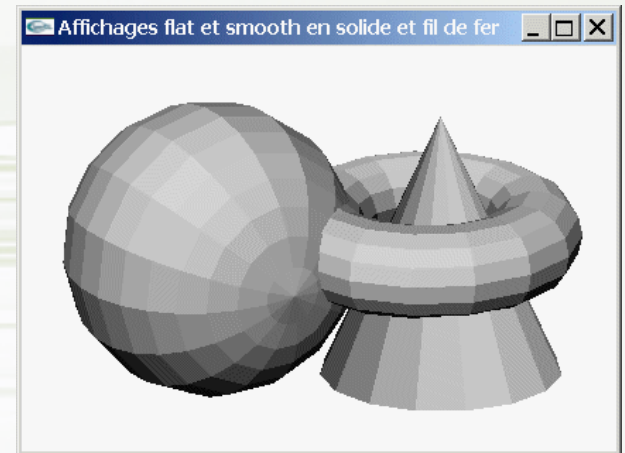
- void glutWireObjet() ;
- void glutSolidObjet() ;

Dans le mode de dessin en fil de fer (wireframe), les objets sont représentés par dessin de segments de droite. Dans le mode de dessin en surfacique (solid), les objets sont représentés par dessin de surfaces.



Enter: solide/facettes

Programme [ici](#)



Les différents types de projection

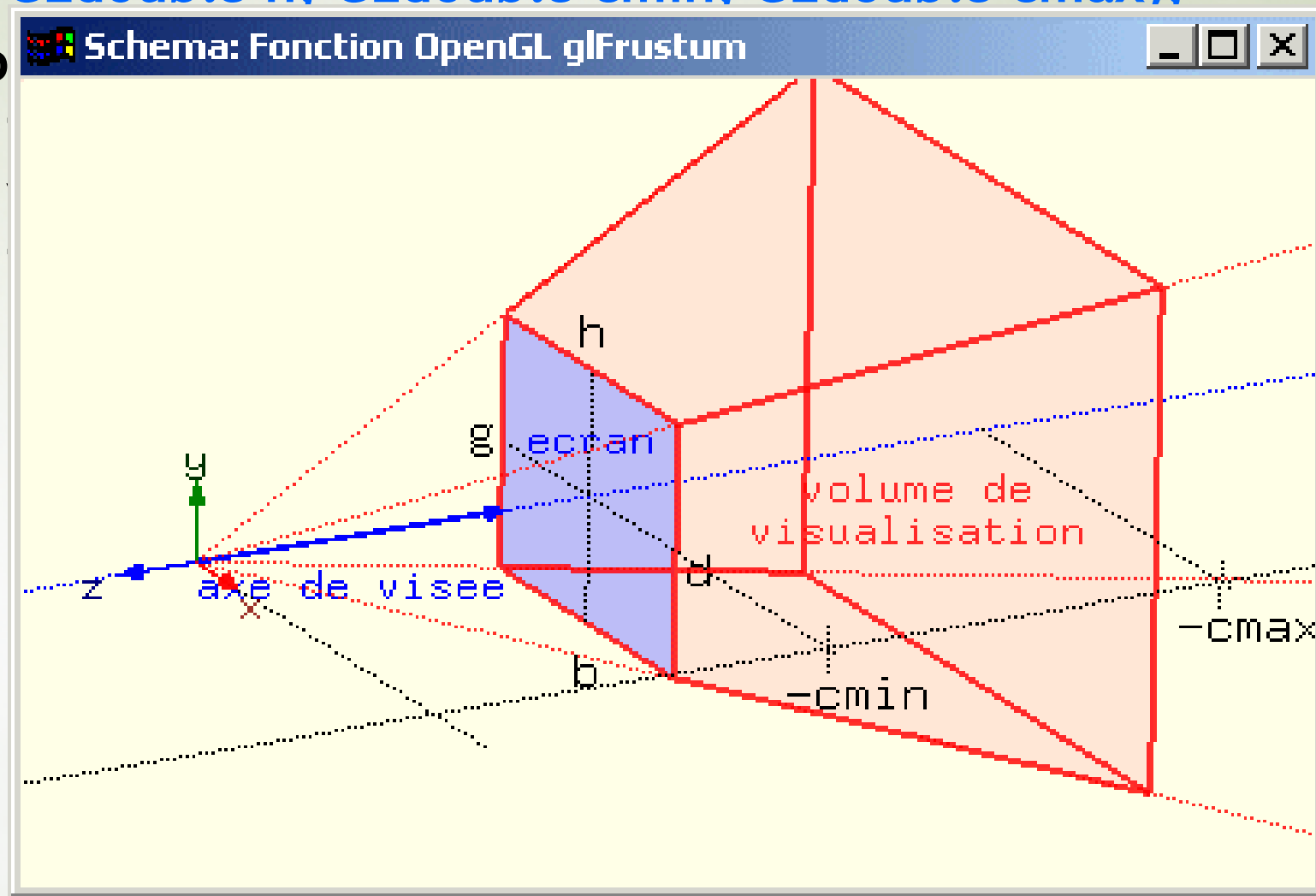
OpenGL permet de choisir le mode d'affichage des scènes dessinées. Deux choix sont offerts:

- (1) la projection parallèle orthographique,
- (2) deux projections en perspective.




```
void glFrustum(GLdouble g, GLdouble d, GLdouble b,
              GLdouble h, GLdouble cmin, GLdouble cmax):
```

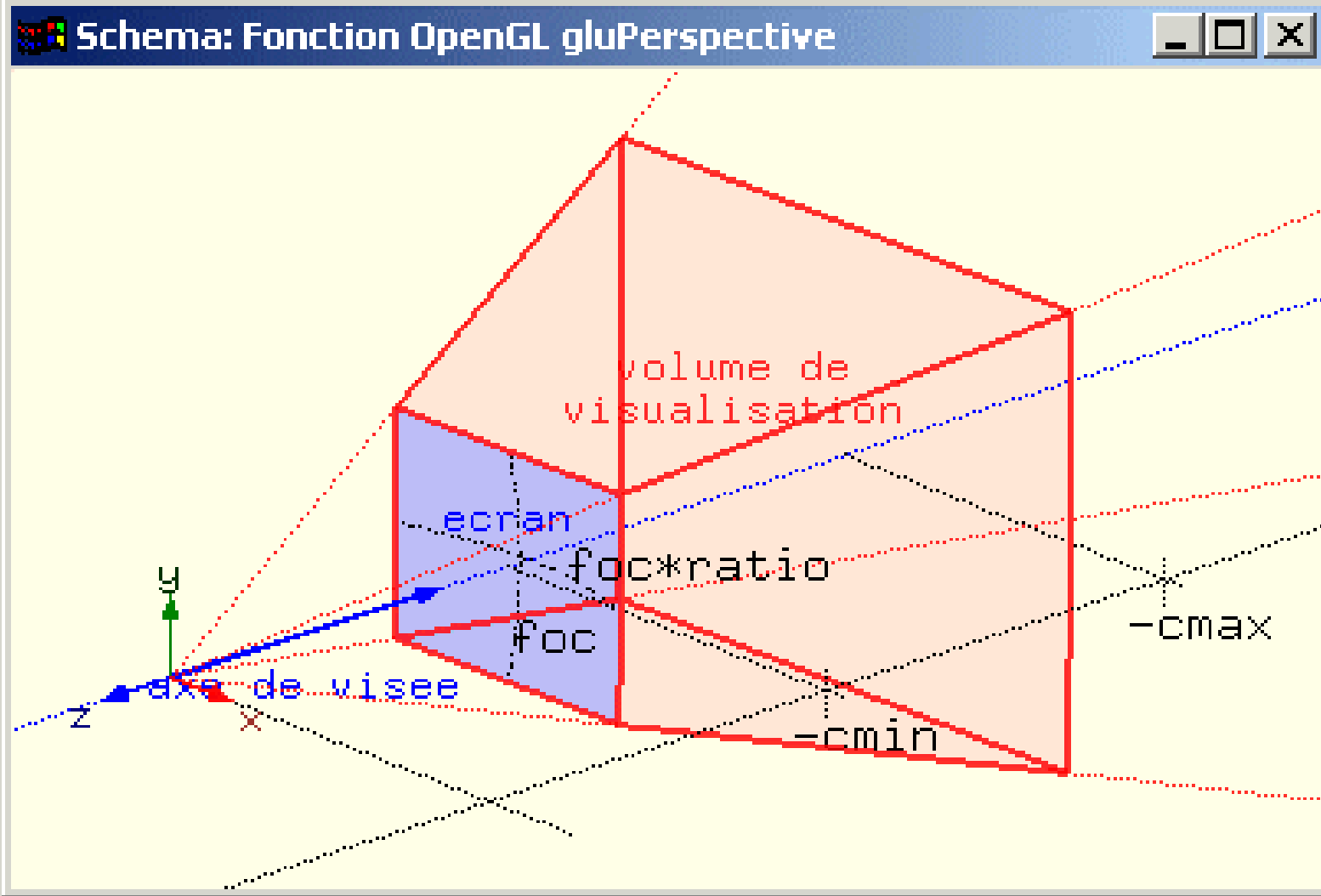
Co



la
de
de
xe
o,-
n).
les
(-
ur
de

```
void gluPerspective(GLdouble foc, GLdouble  
ratio, GLdouble cmin, GLdouble cmax);
```

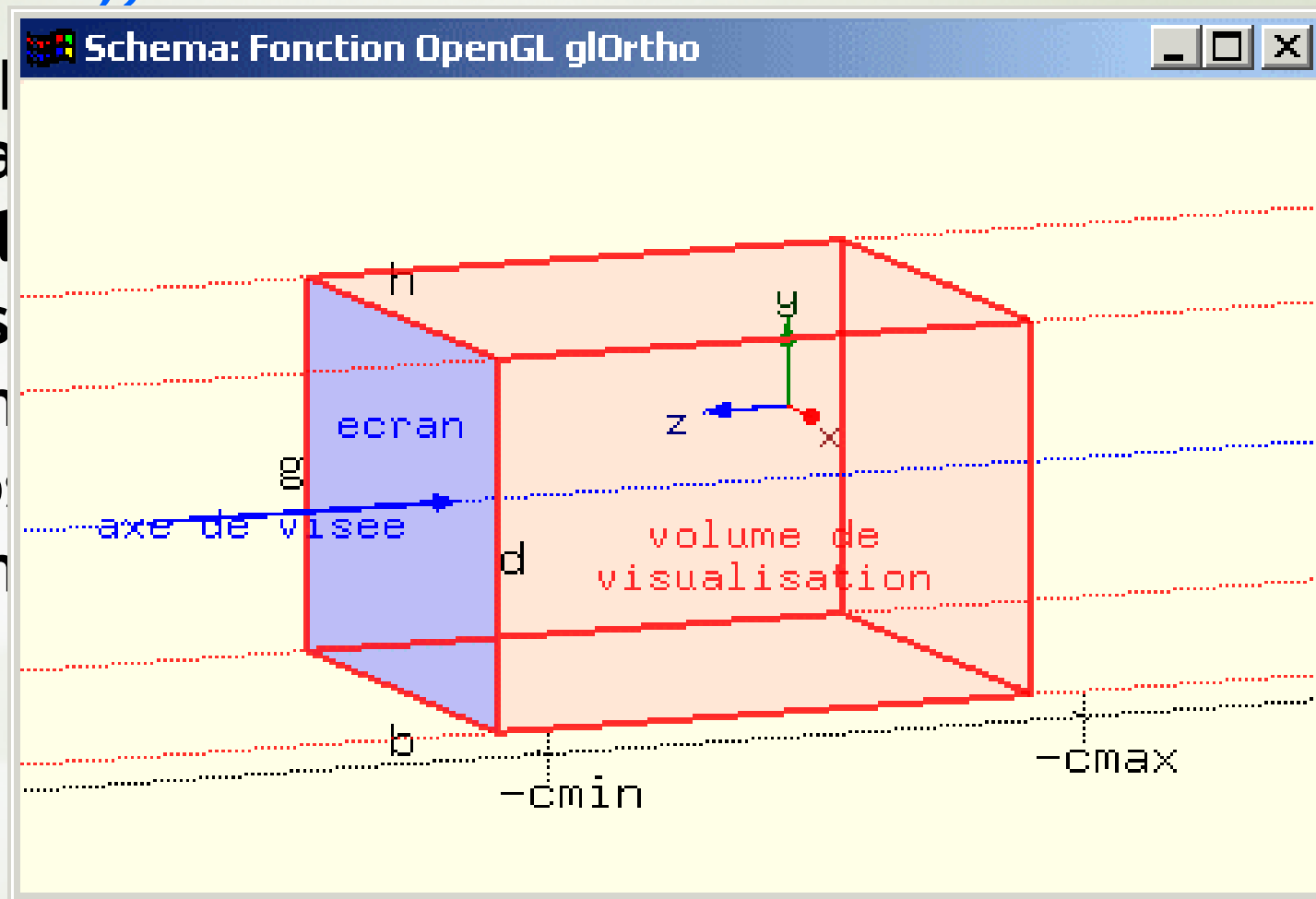
C



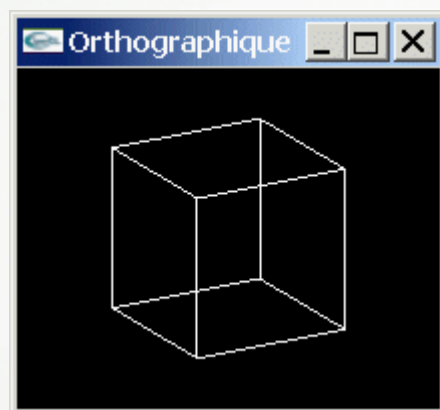
la
ve
la
D,
oc
t-
es
-
ur
it

```
void glOrtho(GLdouble g, GLdouble d, GLdouble
             b, GLdouble h, GLdouble cmin, GLdouble
             cmax);
```

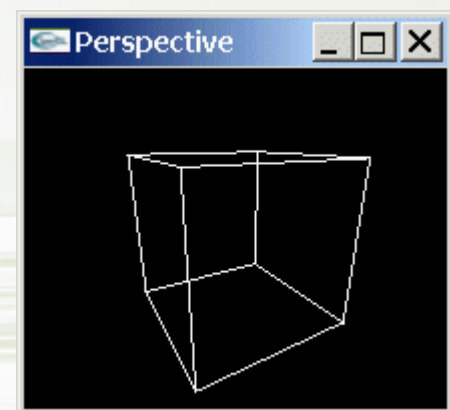
Com
tra
sel
vis
cm
po
cm



la
ue
de
h,-
tre
ter



Programme [ici](#)



Affichages OpenGL en ombrage plat et lissé (Gouraud)

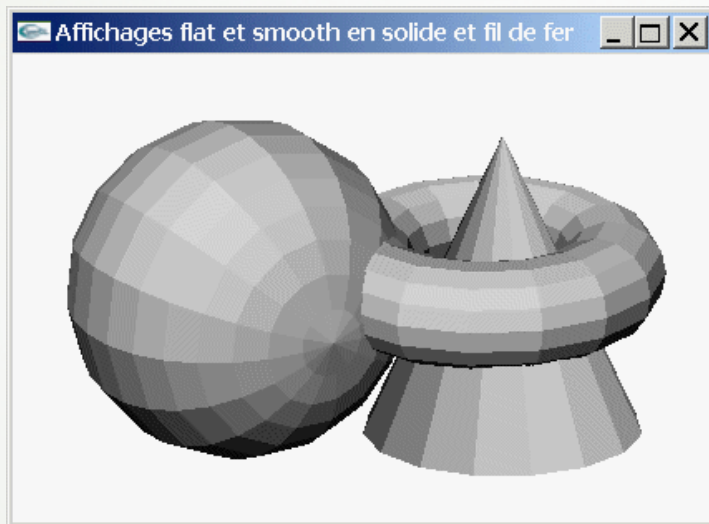
OpenGL permet la création de lumières dans les scènes et l'affectation de matériaux aux surfaces. Lorsqu'un ombrage plat est géré, chaque surface plane adopte une couleur unie fonction de son orientation et des sources lumineuses présentes dans la scène (ombrage de Lambert).

L'ombrage de Gouraud (lissé) crée un dégradé entre les trois valeurs de couleur calculées aux extrémités d'une facette triangulaire munie d'une teinte de base et soumise à un éclairage. Les caractéristiques restituées permettent de:

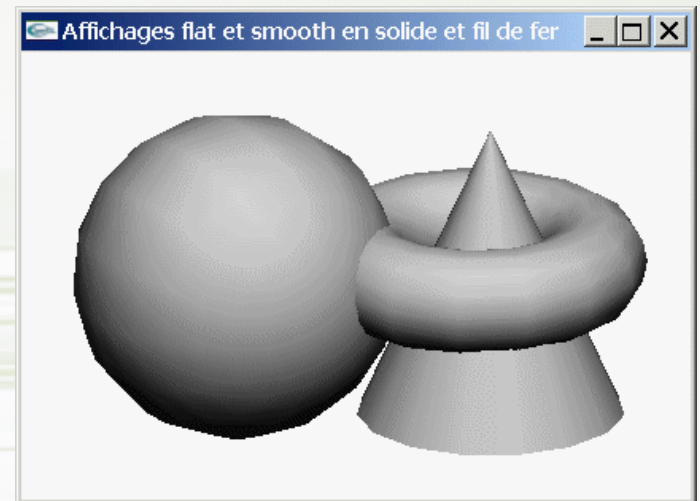
- donner une impression de relief,
- masquer la facettisation des objets liée à l'utilisation d'un ombrage plat sur des ensembles de facettes planes adjacentes modélisant une surface courbe.

Une explication plus complète [ici](#)

L'illumination est calculée en un sommet au moyen de la formule de Lambert et tiendra compte de l'orientation de l'ensemble des facettes adjacentes à ce sommet pour être représentative du relief créé par ces facettes en ce sommet. Ce sommet étant partagé par plusieurs facettes, il sera affecté pour chacune d'elles de la même illumination et aura donc la même couleur.

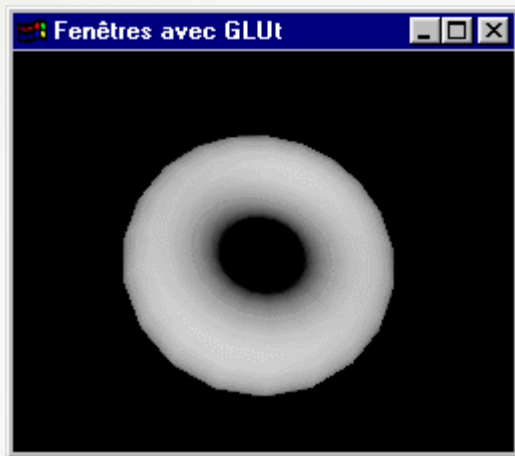


Exemple de
Lissage [ici](#)

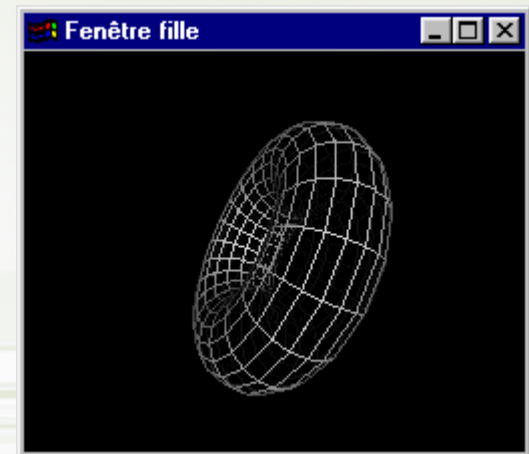


Environnement multifenêtre

Vous pouvez gérer en opengl plusieurs fenêtres.



Le programme est [ici](#)



Affichage OpenGL avec depth cueing

Un affichage avec depth cueing rendra les objets d'autant moins contrastés qu'ils sont éloignés de l'observateur. On rend ainsi compte du phénomène de diffusion de la lumière dans l'atmosphère.



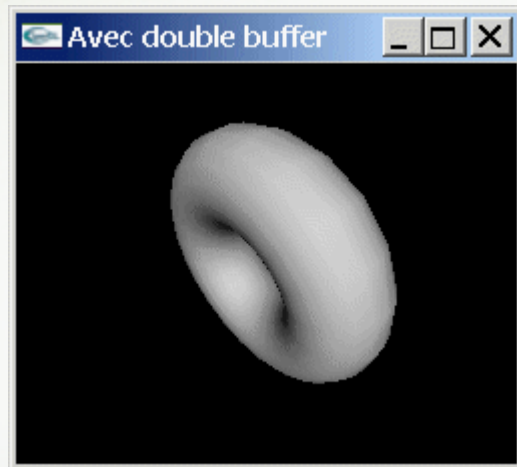
Le programme est [ici](#)

Affichage OpenGL avec double buffer

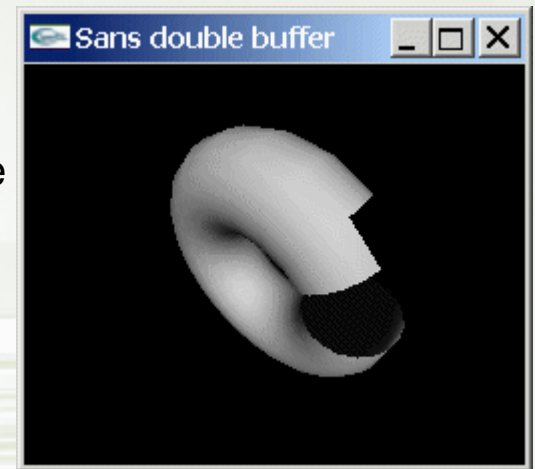
Le double buffer est une technique ayant pour but l'amélioration de la qualité et de la vitesse d'affichage lors de la réalisation animations.

En l'absence du double-buffering, une image en cours de calcul peut être affichée au cours d'une animation faisant alors clairement apparaître la génération progressive du résultat final. Avec le double-buffering, l'image est calculée dans un premier tampon mémoire, tandis qu'un second tampon est envoyé à l'écran. Lorsque l'image est terminée, les deux tampons sont inversés de manière que l'image finie soit envoyée à l'écran et que l'ancien tampon écran puisse être utilisé pour calculer l'image suivante de l'animation. Ainsi, à aucun moment, une image incomplète n'est affichée. L'opération d'inversion (swap) des deux tampons est réalisée en synchronisation avec le balayage vidéo pour être totalement masqué.

L'opération d'inversion (swap) des deux tampons est réalisée en synchronisation avec le balayage vidéo pour être totalement masqué.

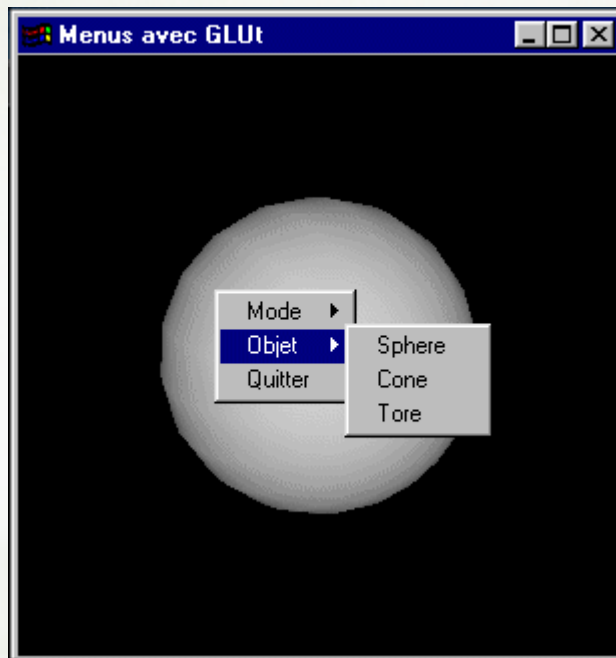


Le programme est [ici](#) mais vue
La vitesse d'affichage, la différence
N'est plus visible.



Utilisation de menus

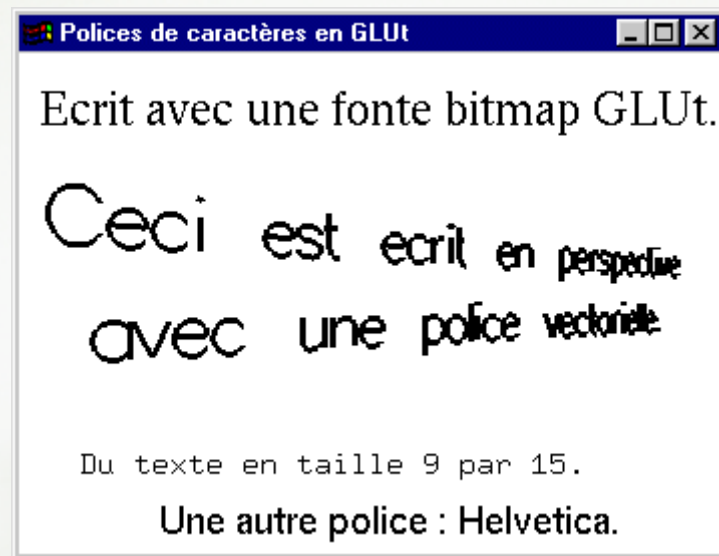
Vous avez la possibilité de faire apparaître dans l'application des menus contextuels qui vont vous permettre de manipuler certaines variables.



Le source du programme est [ici](#)

Les polices de caractères bitmap

Vous avez la possibilité d'inscrire du texte dans une fenêtre opengl en fixant la police, la taille mais aussi des paramètres comme la perspective.



Le programme est [ici](#)

3

Le processus de visualisation en opengl



Processus de visualisation

Quatre transformations successives utilisées au cours du processus de création d'une image:

(1) Transformation de modélisation (Model)

Permet de créer la scène à afficher par création, placement et orientation des objets qui la composent.

(2) Transformation de visualisation (View)

Permet de fixer la position et l'orientation de la caméra de visualisation.

(3) Transformation de projection (Projection)

Permet de fixer les caractéristiques optiques de la caméra de visualisation (type de projection, ouverture, ...).

(4) Transformation d'affichage (Viewport)

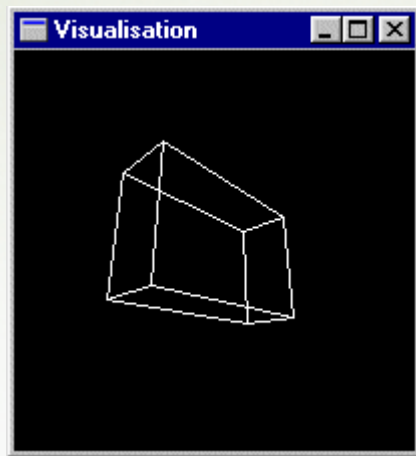
Permet de fixer la taille et la position de l'image sur la fenêtre d'affichage.

Les transformations *de visualisation* et de *modélisation* n'en forment qu'une pour OpenGL (transformation *modelview*). Cette transformation fait partie de l'environnement OpenGL.

La transformation de projection existe en tant que telle dans OpenGL, et fait elle aussi partie de l'environnement OpenGL.

Chacune de ces deux transformations peut être modifiée indépendamment de l'autre ce qui permet d'obtenir une indépendance des scènes modélisées vis à vis des caractéristiques de la "caméra" de visualisation.

La transformation d'affichage est elle aussi paramétrable en OpenGL.



Le programme qui va [avec](#)



Choix de la transformation OpenGL de travail

void glMatrixMode(GLenum mode);

mode: transformation sur laquelle (GL_MODELVIEW ou GL_PROJECTION) les transformations géométriques à venir vont être composées de manière incrémentale.

Pour réaliser un affichage, glMatrixMode est généralement appelé successivement une fois sur chacun des deux paramètres de manière à établir les matrices modelview et projection. Ces appels sont habituellement réalisés au sein de la fonction reshape si les librairies GLUT sont utilisées. Dans la fonction display de GLUT, l'habitude veut que l'on ne travaille qu'en modelview (sauf exception).

Transformations géométriques

Transformations d'utilité générale

void glLoadIdentity(void);

Affecte la transformation courante avec la transformation identité.

void glLoadMatrix {f d} (const TYPE *m);

Affecte la transformation courante avec la transformation caractérisée mathématiquement par la matrice m (16 valeurs en coordonnées homogènes).

void glMultMatrix {f d} (const TYPE *m);

Compose la transformation courante par la transformation de matrice m (16 valeurs en coordonnées homogènes).

{f d} signifie que le calcul s'effectuera en simple (f) ou double (d) précision. On pourra donc appeler les fonction glLoadMatrixf ou glLoadMatrixd

void glTranslate {f d} (TYPE x,TYPE y,TYPE z);

Compose la transformation courante par la translation de vecteur (x,y,z). Très utilisé en modélisation.

void glRotate{f d}(TYPE a,TYPE dx,TYPE dy,TYPE dz);

Compose la transformation courante par la rotation d'angle a degrés autour de l'axe (dx,dy,dz) passant par l'origine. Très utilisé en modélisation.

void glScale {f d} (TYPE rx,TYPE ry,TYPE rz);

Compose la matrice courante par la transformation composition des affinités d'axe x, y et z, de rapports respectifs rx, ry et rz selon ces axes. Très utilisé en modélisation.

Exemple d'un bras de robot [ici](#) et la terre [là](#)

Transformation spécifique à la visualisation (view)

```
void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez,  
               GLdouble cx, GLdouble cy, GLdouble cz,  
               GLdouble upx, GLdouble upy, GLdouble upz);
```

Compose la transformation courante (généralement MODELVIEW) par la transformation donnant un point de vue depuis (ex,ey,ez) avec une direction de visualisation passant par (cx,cy,cz). (upx,upy,upz) indique quelle est la direction du repère courant (fréquemment le repère global) qui devient la direction y (0.0, 1.0, 0.0) dans le repère écran.

gluLookAt est une fonction de la bibliothèque GLU.

Exemple:

```
gluLookAt(10.0,15.0,10.0,3.0,5.0,-2.0,0.0,1.0,0.0)
```

place la caméra en position (10.0,15.0,10.0), l'oriente pour qu'elle vise le point (3.0,5.0,-2.0) et visualisera la direction (0.0,1.0,0.0) de telle manière qu'elle apparaisse verticale dans la fenêtre de visualisation.

Ces valeurs sont considérées dans le repère global.

Dans la pratique, gluLookAt place et oriente la scène devant la caméra de telle manière que la caméra semble placée et orientée selon les paramètres d'entête.

-> gluLookAt doit être exécuté en mode MODELVIEW et doit être placé avant la création de la scène.

Transformations de projection (déjà vues)

```
void glFrustum(GLdouble g, GLdouble d, GLdouble b, GLdouble h,  
GLdouble cmin, GLdouble cmax);
```

Compose la transformation courante par la transformation de projection en perspective de volume de visualisation défini par la pyramide tronquée de sommet l'origine O, orientée selon l'axe -z et de plan supérieur défini par la diagonale (g,b,-cmin) (d,h,-cmin).

```
void gluPerspective(GLdouble foc, GLdouble ratio, GLdouble cmin,  
GLdouble cmax);
```

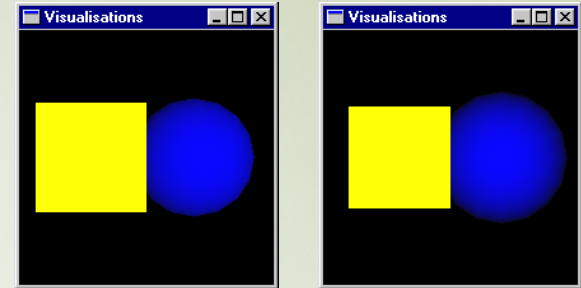
Compose la transformation courante par la transformation de projection en perspective de volume de visualisation défini par la pyramide tronquée de sommet l'origine O, orientée selon l'axe -z, possédant l'angle foc comme angle d'ouverture verticale, l'aspect-ratio ratio (rapport largeur/hauteur) et les plans de clipping proche et éloignés -cmin et -cmax.

```
void glOrtho(GLdouble g, GLdouble d, GLdouble b, GLdouble h, GLdouble  
cmin, GLdouble cmax);
```

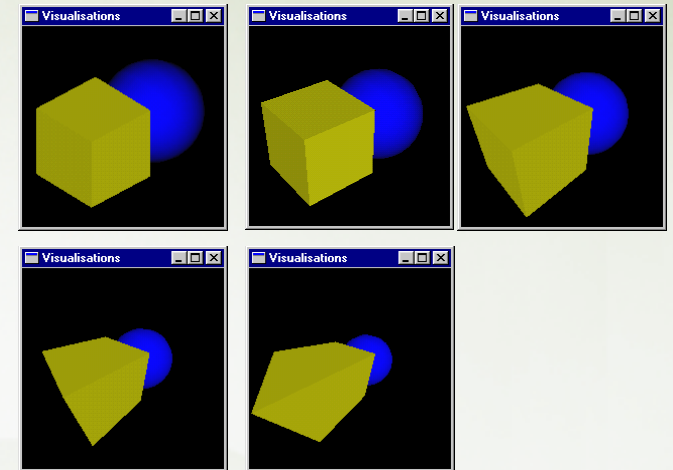
Compose la transformation courante par la transformation de projection orthographique selon l'axe -z et définie par le volume de visualisation parallélépipédique (g,d,b,h,-cmin,-cmax). cmin et cmax peuvent être positifs ou négatifs et mais doivent respecter $cmin < cmax$.

Exemples de projections

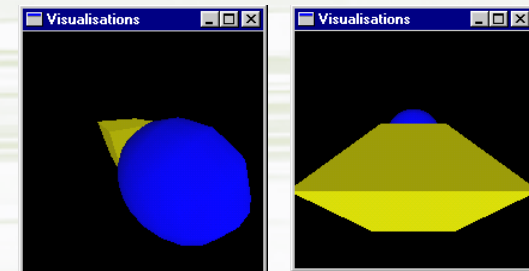
Projections orthogonales et perspectives (sans angle, pas de différence)



Projection en perspective avec rapprochement de scène



projection en perspective: très grosses déformations

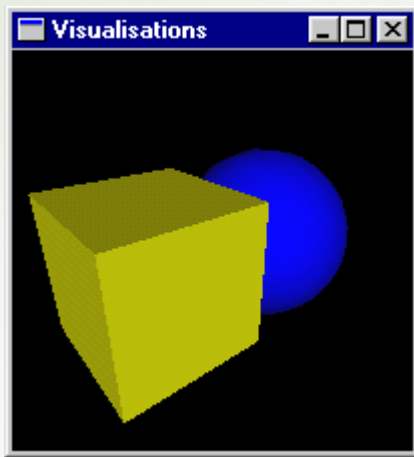


Le programme est [ici](#)

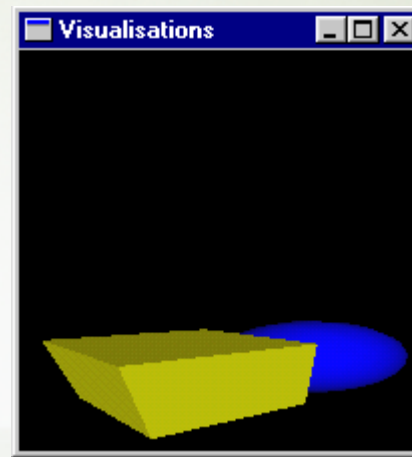
Transformation d'affichage

```
void glViewport(GLint x,GLint y,GLsizei l,GLsizei h);
```

Définit le rectangle de pixels dans la fenêtre d'affichage dans lequel l'image calculée sera affichée.



Toute la fenêtre



Une partie seulement

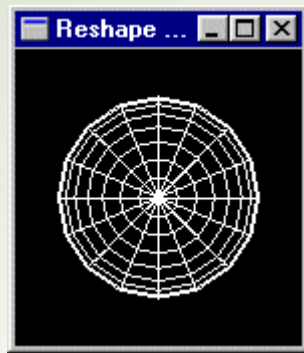
Le programme est [ici](#)

Problème

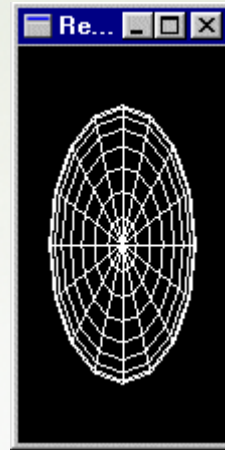
L'affichage des scènes fait appel à un repère virtuel associé au viewport de visualisation. Les dimensions fournies au moment de l'exécution de la fonction de projection définissent ce repère qui peut donc ne pas être homothétique au viewport.

Exemple: `glFrustum(-1.,1.,-1.,1.,1.5,20.)`
associe le repère virtuel $(-1,-1) - (1,1)$ au viewport d'affichage. Si celui-ci n'a pas des résolutions en x et y identiques, les objets dessinés sont déformés.

Utilisation du reshape : non homothétique



original



Après déformation de la fenêtre

Solution: Adapter aux dimensions du viewport les dimensions données pour la définition de la projection.

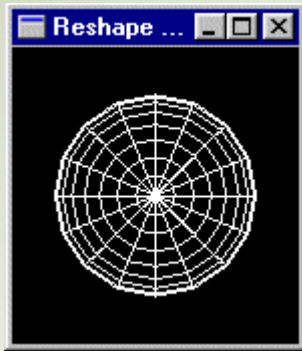
Exemple: Pour un viewport défini par

glViewport(0,0,w,h)

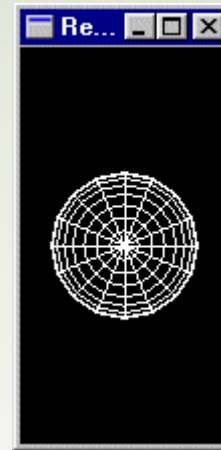
on pourra utiliser

gluPerspective(65.,(double) w/h,1.0,20.0)

ce qui revient à adapter le ratio de gluPerspective au ratio du viewport.



original



Après déformation de
la fenêtre

Le programme est [ici](#)

Les listes d'affichage

Une liste d'affichage est une suite de commandes OpenGL stockées pour une utilisation future.

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.

Quand une liste d'affichage est invoquée, les commandes qu'elle contient sont exécutées dans l'ordre où elles ont été stockées.

Les commandes sont non seulement compilées dans la liste d'affichage, mais peuvent aussi être exécutées immédiatement pour obtenir un affichage.

Les commandes d'une liste d'affichage sont les mêmes que les commandes d'affichage immédiat.

Une fois qu'une liste a été définie, il est impossible de la modifier à part en la détruisant et en la redéfinissant.

Les listes d'affichages sont des macros et, à ce titre, ne peuvent pas être modifiées ou paramétrées.

Les listes d'affichage optimisent les temps de calcul et d'affichage car elles permettent la suppression de calculs redondants sur:

- les opérations matricielles,
- les conversions de formats d'image bitmap,
- les calculs de lumière,
- les calculs de propriétés des matériaux de surface,
- les calculs de modèle d'éclairage,
- les textures,
- les motifs de tracé.

L'inconvénient principal des listes d'affichage est qu'elles peuvent occuper beaucoup de place en mémoire.

Les listes d'affichage Commandes principales

void glNewList(GLuint liste, GLenum mode);

Indique le début de la définition d'une liste d'affichage.

liste: entier positif unique qui identifiera la liste.

mode: GL_COMPILE ou GL_COMPILE_AND_EXECUTE suivant que l'on désire seulement stocker la liste ou la stocker et l'exécuter simultanément.

void glEndList(void);

Indique la fin d'une liste d'affichage.

Toutes les commandes depuis le dernier glNewList ont été placées dans la liste d'affichage (quelques exceptions).

void glCallList(GLuint liste);

Exécute la liste d'affichage référencée par liste.

Commandes auxiliaires

GLuint glGenLists(GLsizei nb);

Alloue nb listes d'affichage contiguës, non allouées auparavant.

L'entier retourné est l'indice qui donne le début de la zone libre de références.

GLboolean glIsList(GLuint liste);

Retourne vrai si liste est utilisé pour une liste d'affichage, faux sinon.

void glDeleteLists(GLuint l, GLsizei nb);

Détruit les nb listes à partir de la liste l.

Exemples de liste d'affichage

Dessin d'un polygone régulier de 100 sommets.

```
void cercle(){
    GLint i ;
    GLfloat cosinus,sinus ;
    glBegin(GL_POLYGON) ;
    for ( i = 0 ; i < 100 ; i++ ) {
        cosinus = cos(i*2*PI/100.0);
        sinus = sin(i*2*PI/100.0);
        glVertex2f(cosinus,sinus) ; }
    glEnd() ;
}
```

Cette méthode sans liste d'affichage est inefficace car tout appel à cercle() entraîne l'exécution des calculs de sinus et cosinus.

```
void cercle() {
    GLint i ;
    GLfloat cosinus,sinus ;
    glNewList(1,GL_COMPILE) ;
    glBegin(GL_POLYGON) ;
    for ( i = 0 ; i < 100 ; i++ ) {
        cosinus = cos(i*2*PI/100.0);
        sinus = sin(i*2*PI/100.0);
        glVertex2f(cosinus,sinus) ; }
    glEnd() ;
    glEndList() ;
}
```

glCallList(CERCLE) ;

Le programme des cercles est [ici](#)

Les bitmaps

OpenGL permet la gestion de bitmaps en autorisant les opérations de transfert entre le buffer couleur et la mémoire centrale.

Buts

- gestion de fichiers
- gestion de polices de caractères (non traité)
- gestion de textures (voir plus loin)

Commandes

```
void glReadPixels(GLint x, GLint y, GLsizei l,  
    GLsizei h, GLenum format, GLenum type,  
    GLvoid *pixels);
```

Lecture des pixels d'un frame-buffer et stockage en mémoire.

x et y: position du coin supérieur gauche du rectangle de pixels

l et w: dimensions du rectangle de pixels

format: information à lire sur les pixels

type: type du résultat à fournir

pixels: tableau destiné à recevoir les résultats

Format & Type de données pour les pixels

GL_COLOR_INDEX Indexe de couleurs

GL_RGB Couleurs RVB

GL_RGBA Couleurs RVBA

GL_RED Composante rouge

GL_GREEN Composante verte

GL_BLUE Composante bleue

GL_ALPHA Composante alpha

GL_LUMINANCE Luminance

GL_LUMINANCE_ALPHA Luminance puis composante alpha

GL_STENCIL_INDEX Stencil

GL_DEPTH_COMPONENT Composante de profondeur

type Type C

GL_UNSIGNED_BYTE Entier 8 bits non signé

GL_BYTE Entier 8 bits signé

GL_BITMAP Bits dans des entiers 8 bits non signés

GL_UNSIGNED_SHORT Entier 16 bits non signé

GL_SHORT Entier 16 bits signé

GL_UNSIGNED_INT Entier 32 bits non signé

GL_INT Entier 32 bits signé

GL_FLOAT Réel simple précision


```
void glDrawPixels(GLsizei l,GLsizei h,GLenum f,GLenum type,GLvoid *pixels);
```

Écriture dans un frame-buffer de pixels stockés en mémoire. Le rectangle de pixels est affiché avec son coin supérieur gauche en position raster courante

l et w: dimensions du rectangle de pixels

f: information à écrire sur les pixels

type: type des données transmises

pixels: tableau destiné à transmettre les pixels

```
void glRasterPos{sifd}{v}(TYPE x,TYPE y,TYPE z,TYPE w);
```

Fixe la position raster courante

Si 2 est utilisé z est fixé à 0 et w à 1. Si c'est 3, w est fixé à 1

```
void glCopyPixels(GLint x,GLint y,GLsizei l,GLsizei h,GLenum type);
```

Copie depuis un buffer dans lui-même

x et y: position du coin supérieur gauche du rectangle source (la position du rectangle destination donnée par la position raster courante)

l et w: dimensions du rectangle source

type: buffer sur lequel agir (GL_COLOR, GL_STENCIL ou GL_DEPTH)

Le programme est [ici](#)

4

Un cube tournant



I n t r o d u c t i o n

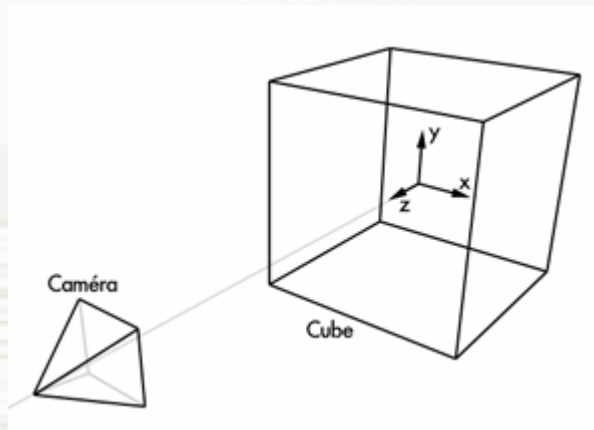
Dans le précédent chapitre, vous avez eu l'occasion d'écrire vos premiers programmes OpenGL et de voir la bibliothèque glut (partiellement). Ce programme était sans doute un peu décevant (le premier carré), car il se contentait d'afficher un carré à l'écran. Maintenant, je vous propose de passer à la vitesse et à la dimension supérieures avec 'LE' classique de la 3D, celui avec lequel ont débuté bon nombre de demomakers : le cube tournant.

Pour être plus précis, nous allons partir du premier programme que nous avons écrit, et lui ajouter de nouvelles fonctionnalités : nous allons remplacer le carré par un cube et offrir à l'utilisateur la possibilité de faire tourner le cube sur lui-même par l'intermédiaire de la souris. Ce programme a pour but d'aborder :

- les méthodes de stockage des scènes tridimensionnelles,
- les grands principes de l'animation avec OpenGL,
- De voir de nouvelles fonctions de rappel (pour la gestion de la souris et des redimensionnements).

Le cube : stockage :

Dans le premier programme, nous n'avons pas rencontré de problème concernant le stockage de la scène, puisqu'elle ne contenait que 4 sommets. La figure ci-dessous représente la scène que nous souhaitons afficher.



Notre cube comporte 6 faces et 8 sommets. La scène est encore assez simple et on pourrait tout à fait se contenter, comme nous l'avons fait dans le précédent chapitre, décrire linéairement chaque polygone. Nous allons cependant mettre en place une structure de données pour stocker notre scène. OpenGL fournit des fonctions permettant la mise en place de structures de stockage, mais nous allons voir qu'il est tout à fait possible de le faire soi-même.

Première chose, nous allons définir un type de données pour stocker chaque point : il s'agit d'une structure composée de 6 nombres de type flottant qui contient les coordonnées cartésiennes du point (x, y et z), ainsi que la couleur du sommet (r, g et b).

```
typedef struct {  
    float x;  
    float y;  
    float z;  
    float r;  
    float g;  
    float b;  
} point;
```

Pour stocker tous les sommets de la scène, on définit un tableau de points, et on en profite pour le remplir avec les coordonnées des points de notre cube et leurs couleurs :

```
point p[8]=  
{  
    {-0.5,-0.5, 0.5,1.0,0.0,0.0},  
    {-0.5, 0.5, 0.5,0.0,1.0,0.0},  
    { 0.5, 0.5, 0.5,0.0,0.0,1.0},  
    {0.5,-0.5,0.5,1.0,1.0,1.0},  
    {-0.5,-0.5,-0.5,1.0,0.0,0.0},  
    {-0.5, 0.5,-0.5,0.0,1.0,0.0},  
    { 0.5, 0.5,-0.5,0.0,0.0,1.0},  
    { 0.5,-0.5,-0.5,1.0,1.0,1.0}  
};
```

Reste maintenant à trouver une structure permettant de stocker les 6 faces de notre cube. Sachant qu'une face comporte exactement 4 sommets, nous pouvons utiliser un tableau à 2 dimensions contenant les indices (dans le tableau p que nous venons de créer) de chacun des points de la face :

```
int f[6][4]={ {0,1,2,3}, {3,2,6,7}, {4,5,6,7},  
              {0,1,5,4}, {1,5,6,2}, {0,4,7,3}};
```


Ce tableau peut à première vue paraître obscure, aussi voici de quoi éclairer votre lanterne : $f[i][j]$ contient l'indice du j -ième sommet de la face numéro i . Ainsi, la face numéro 4 est définie par les sommet $p[1]$, $p[5]$, $p[6]$ et $p[2]$ (je vous rappelle qu'en C, les indices de tableaux commencent à 0, et non à 1).



Le cube : affichage

Voyons maintenant comment utiliser cette structure dans notre fonction `affichage()`. Nous avons vu que la description d'un polygone se fait par énumération des sommets (avec la fonction `glVertex()`), le tout encadré par un `glBegin()` et un `glEnd()`. Avec notre structure de données, nous allons utiliser 2 boucles `for` imbriquées. La première permet de parcourir chacune des faces, la seconde permet d'énumérer les 4 points de chaque face :

```
for (i=0;i<6;i++) {  
  glBegin(GL_POLYGON);  
    for (j=0;j<4;j++)  
    {    glColor3f(p[f[i][j]].r,p[f[i][j]].g,p[f[i][j]].b);  
        glVertex3f(p[f[i][j]].x,p[f[i][j]].y,p[f[i][j]].z);  
    }  
  glEnd();  
}
```

Animation

Nous souhaitons donner du mouvement à notre cube grâce à la souris. Aussi, il faut essayer de répondre à la question "Comment faire tourner notre cube ? ". Sachant que le sujet des transformations d'objets (translation, rotation, mise à l'échelle) est vaste, nous allons comme d'habitude utiliser les appels classiques aux fonctions de transformation.


```
glLoadIdentity();  
glRotatef(-angley,1.0,0.0,0.0);  
glRotatef(-anglex,0.0,1.0,0.0);
```

Ces lignes sont à placer dans la fonction `affichage()`, juste avant la description des faces. Les transformations d'objets sont stockées dans une matrice. La fonction `GLLoadIdentity(angle,x,y,z)` permet de réinitialiser la matrice de transformation, `glRotatef()` ajoute à la matrice de transformation une rotation dont l'axe passe par l'origine et est porté par le vecteur (x,y,z) . Ici, nous avons défini une rotation d'angle `'-angley'` autour de l'axe x, et une rotation d'angle `'-anglex'` autour de l'axe y. La notation utilisée peut vous paraître bizarre, mais elle est dictée par la souris : un déplacement horizontal (suivant x) de la souris correspond intuitivement à une rotation du cube autour de l'axe y.

Il est important de noter que les rotations sont appliquées dans le sens **inverse** de leur description. Ainsi, nous avons défini ci-dessus une rotation autour de l'axe y suivie d'une rotation autour de x (dont le résultat est en général différent d'une rotation autour de x suivie d'une rotation autour de y).

Il nous faut maintenant déterminer les valeurs des variables `anglex` et `angley` en fonction de la position de la souris. Le comportement souhaité est le suivant : *pour faire tourner le cube, l'utilisateur doit cliquer sur la fenêtre OpenGL, maintenir le bouton enfoncé et déplacer la souris.* Nous allons utiliser deux nouvelles fonctions de rappel offertes par glut.

Les fonctions de rappel liées à la souris :

Glut permet de définir une fonction de rappel pour les boutons de la souris. On met en place le rappel avec `glutMouseFunc()`. La fonction enregistrée sera appelée à chaque fois que l'utilisateur appuiera sur un des boutons de la souris ou le relâchera. Enregistrons une fonction appelée `mouse()` :

`glutMouseFunc(mouse);`

Le prototype de la fonction `mouse()` doit être le suivant :

```
void mouse(int bouton,int etat,int x,int y)
```


Lorsque la fonction est lancée par le gestionnaire d'événements de glut, 'bouton' contient le nom du bouton qui a été pressé ou relâché (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON), 'etat' indique si le bouton a été pressé (GLUT_DOWN) ou relâché (GLUT_UP) et 'x' et 'y' contiennent les coordonnées de la souris au moment de l'événement.

Nous souhaitons que le mouvement de la souris ne soit appliqué au cube que lorsque le bouton gauche est enfoncé. Nous allons mettre en place une variable globale 'pressé' (de type char) qui vaudra 1 lorsque le bouton est pressé, et 0 lorsque le bouton est relâché. Nous allons également introduire deux entiers x_old et y_old (défini comme variable globale afin qu'ils soient accessibles depuis l'autre fonction de rappel pour la souris). Ils vont servir à stocker la position de la souris lors de l'appui sur le bouton de gauche.


```
void mouse(int button, int state,int x,int y)
{ /* si on appuie sur le bouton gauche */
if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    { presse = 1; /* le booleen presse passe a 1 (vrai) */
    xold = x; /* on sauvegarde la position de la souris */
    yold=y;
    }
    /* si on relache le bouton gauche */
    if (button == GLUT_LEFT_BUTTON && state == GLUT_UP) presse=0;
    /* le booleen presse passe a 0 (faux) */
}
```

Le second rappel lié à la souris génère un événement lorsque la souris est déplacée alors qu'au moins un des boutons est pressé. L'enregistrement de la fonction de rappel se fait avec `glutMotionFunc()`, et le prototype de la fonction enregistrée doit être le suivant :

```
void mousemotion(int x,int y)
```

'x' et 'y' sont les coordonnées de la souris par rapport au coin supérieur gauche de la fenêtre au moment de la génération de l'événement. Notre fonction `mousemotion` a pour but de mettre à jour les valeurs de 'anglex' et 'angley' en fonction du déplacement relatif de la souris si le bouton gauche est enfoncé (c'est-à-dire si 'presse' vaut 1) :

```
void mousemotion(int x,int y)
{
if (presse) /* si le bouton gauche est presse */
{ /* on modifie les angles de rotation de l'objet en fonction de la position
actuelle de la souris et de la dernière position sauvegardée */
    anglex=anglex+(x-xold);
    angley=angley+(y-yold);
    glutPostRedisplay();

/* on demande un rafraichissement de l'affichage */
}
    xold=x; yold=y;
/* sauvegarde des valeurs courante de le position de la souris */
}
```

Le tampon de profondeur :

Le passage du carré 2D au cube 3D fait surgir un nouveau problème : celui de l'ordre d'affichage des polygones de la scène. La dernière fois, la scène ne comportait qu'un polygone et donc la question ne se posait pas. Aujourd'hui, avec nos 6 faces, il est important d'ordonner correctement l'affichage.



Pour résoudre ce problème de faces cachées, OpenGL propose une technique extrêmement répandue en synthèse d'images : le tampon de profondeur, plus connu sous son nom anglais : Z-buffer. Le tampon de profondeur est une technique simple et très puissante. L'idée principale est de créer en plus de notre image un tampon de même taille. Ce tampon va servir à stocker pour chaque pixel une profondeur, c'est-à-dire la distance entre le point de vue et l'objet auquel appartient le pixel considéré.

A l'origine, le tampon est rempli avec une valeur dite "de profondeur maximale" : l'image est vide. A chaque fois qu'on dessine un polygone, pour chaque pixel qui le constitue, on calcule sa profondeur et on la compare avec celle qui est déjà stockée dans le tampon. Si la profondeur stockée dans le tampon est supérieure à celle du polygone qu'on est en train de traiter, alors le polygone est plus proche (au point considéré) de la caméra que les objets qui ont déjà été affichés. Le point du polygone est affiché sur l'image, et la profondeur du pixel est stockée dans le tampon de profondeur. Dans le cas inverse, le point que l'on cherche à afficher est masqué par un autre point déjà placé dans l'image, donc on ne l'affiche pas. On peut afficher les polygones dans un ordre quelconque.

Nous allons mettre en place un tampon de profondeur pour notre programme. Première chose à faire, il faut modifier l'appel à la fonction glut d'initialisation de l'affichage pour lui indiquer qu'il faut allouer de l'espace pour le tampon de profondeur :

```
glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
```

L'utilisation ou non du tampon de profondeur est gérée par une variable d'état OpenGL : `GL_DEPTH_TEST`.

L'activation du tampon de profondeur se fait en plaçant la ligne suivante dans la phase d'initialisation du programme :

```
glEnable(GL_DEPTH_TEST);
```

Tout comme le tampon image, il faut effacer le tampon de profondeur à chaque fois que la scène est redessinée. Il suffit de modifier l'appel `glClear()` de la fonction d'affichage :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```



Afin de vous permettre de visualiser l'intérêt du tampon de profondeur, je vous propose de vous donner la possibilité d'activer et de désactiver le tampon de profondeur avec les combinaisons de touches 'd' et 'Shift'+ 'd'. Pour cela, ajoutons ces quelques lignes a notre fonction de rappel clavier() :

case 'd':

```
glEnable(GL_DEPTH_TEST);
```

```
glutPostRedisplay(); break;
```

case 'D':

```
glDisable(GL_DEPTH_TEST);
```

```
glutPostRedisplay(); break;
```

Le double tampon :

Si vous compilez et exécutez le programme tel que nous l'avons modifié jusqu'à présent, vous vous rendrez compte qu'un problème subsiste. On peut voir les polygones se dessiner les uns après les autres. Evidemment, moins votre configuration sera puissante, plus l'effet sera marqué. Pour remédier à cet effet désagréable, nous allons mettre en place un système de double tampon d'image (double-buffering). Pour comprendre ce système, raisonnons par analogie : imaginons une personne en train de faire un exposé en utilisant comme support un rétroprojecteur. Si la personne n'a qu'un transparent, elle va dessiner sur le celui-ci alors qu'il est posé sur le rétroprojecteur éclairé. Le public va voir sur l'écran le crayon effectuer le tracé. Si en revanche l'orateur dispose de deux transparents, il peut projeter un transparent, dessiner le transparent suivant sur une table, échanger les deux transparents.

Glut permet de gérer un double tampon de façon extrêmement simple. Tout d'abord, comme pour le tampon de profondeur, il faut indiquer au système que l'on souhaite utiliser un double tampon d'image. Nous modifions donc à nouveau l'appel à `glutInitDisplayMode()` :

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

Il ne reste plus qu'à indiquer à glut le moment opportun pour l'échange des buffers : la fin du dessin de la scène. On place à cet effet un appel à `glutSwapBuffers()` à la fin de notre fonction d'affichage.

La fonction de rappel 'redimensionnement' :

Un problème se pose lorsque qu'on redimensionne la fenêtre OpenGL. Par défaut, la scène OpenGL occupe toute la fenêtre, et la scène représente tous les points dont les coordonnées en x et y sont comprises entre -1 et 1. Si on redimensionne la fenêtre et que le nouveau rapport hauteur/largeur de la fenêtre ne vaut pas 1, l'image subit une déformation. Une technique pour éviter ceci consiste à limiter la zone utilisée pour le dessin à la plus grande sous partie carrée de la fenêtre OpenGL, qu'on prendra soin de centrer. Si on redimensionne la fenêtre à une taille 'w' par 'h', la plus grande sous partie carré de la fenêtre mesure w pixels si $w < h$ et h pixels si $w > h$.

glut propose une fonction de rappel pour les redimensionnements de la fenêtre. Cette fonction doit être enregistrée grâce à `glutReshapeFunc()`.

Voici une fonction de rappel possible :

```
void reshape(int x,int y)
{
    if (x<y)
        glViewport(0,(y-x)/2,x,x);
    else
        glViewport((x-y)/2,0,y,y);
}
```

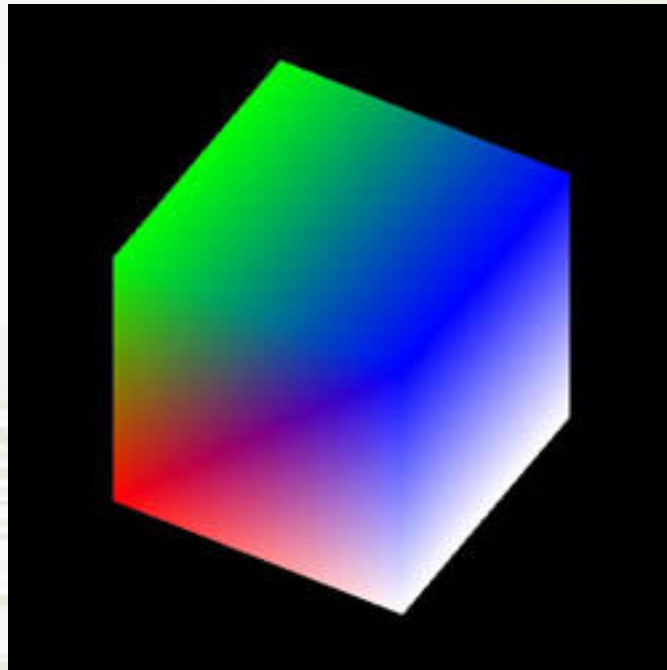
Lors de l'appel à `reshape()`, 'x' contient la nouvelle largeur de la fenêtre, et 'y' la nouvelle hauteur. La fonction `glViewport()` permet de limiter la zone de dessin à une portion de la fenêtre. Son prototype est :

```
void glViewport(GLint x, GLint y, GLsizei largeur, GLsizei hauteur)
```

'x' et 'y' sont les coordonnées du coin supérieur gauche de la sous-fenêtre. Je vous laisse le soin de deviner ce que sont les paramètres 'largeur' et 'hauteur'.

Le programme complet

Cliquez [ici](#) pour obtenir le source.



5

Transformations géométriques



Des maths, des maths !!!

Les algorithmes 3D utilisent le calcul matriciel, car il permet de rassembler tous les calculs nécessaires sous une forme compacte. Dans la suite du didacticiel, je suppose que vous savez ce qu'est une matrice et un produit matriciel. Si ce n'est pas le cas, je vous conseille de jeter un coup d'œil dans un ouvrage traitant du sujet avant de lire ce qui suit. Mon but n'est pas de faire un cours de mathématiques, mais de vous faire comprendre ce qui se passe "à l'intérieur" d'une application 3D.

Il semblerait logique de penser que puisqu'on travaille en 3 dimensions, les matrices utilisées sont elles aussi de dimension 3. En fait ce n'est pas le cas. OpenGL travaille avec des matrices 4x4, dans un système de coordonnées dit "homogène". L'avantage de ce système est qu'il permet de représenter les transformations par des matrices et de composer toutes ces transformations par un simple produit matriciel.

Les points de l'espace sont stockés dans des vecteurs de 4 lignes. Les 3 premières lignes contiennent respectivement les coordonnées x , y et z du point. La quatrième ligne contient un coefficient noté w , appelé facteur d'échelle, qui vaut souvent 1. Considérons un point P en coordonnées homogènes, pour lequel le facteur d'échelle ne vaut pas 1. Pour retrouver les coordonnées cartésiennes du point, il suffit de diviser toutes les composantes par w .

$$P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1.5 \\ 10 \\ 2.1 \\ 1 \end{bmatrix}$$

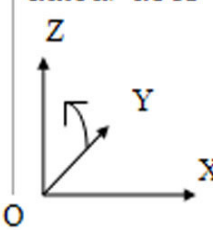
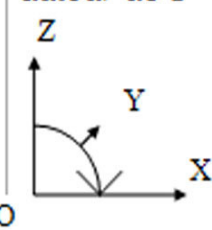
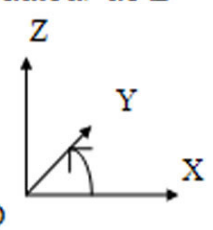
Vous remarquerez qu'avec ce système, un même point de l'espace a une infinité de représentations : les coordonnées $(2,10,8,2)$ et $(1,5,4,1)$ correspondent au même point. Il est également intéressant de noter que les coordonnées homogènes permettent de définir des points situés à l'infini, en choisissant $w=0$.

Je pense que vous savez ce que sont une rotation et une translation, mais l'homothétie est peut-être plus mystérieuse. Une homothétie est un changement d'échelle de l'objet. Une homothétie uniforme de facteur 2 va faire doubler la taille d'un objet, une homothétie de facteur 0,5 va la diminuer de moitié, et bien sûr, une homothétie de facteur 1 va laisser l'objet à sa taille d'origine. Dans la matrice d'homothétie on trouve 3 facteurs h_x , h_y et h_z . Si ces trois coefficients sont égaux, on est dans le cas d'une homothétie uniforme. Si les facteurs ne sont pas identiques, on a une homothétie non uniforme, qui vous donne la possibilité d'étirer et de rétrécir différemment suivant les axes X, Y et Z.

Les matrices

Les matrices de rotation:

Si R2 est une rotation d'angle α par rapport à R1, on a $M_{12} =$

autour de X	autour de Y	autour de Z
		
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Les matrices de translation:

Si R2 est une translation de vecteur $T(T_x, T_y, T_z)$ par rapport à R1, on a

$$M_{12} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Application à un point

$$P' = M_r . M_t . P$$

$$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1.5 \\ 10 \\ 2.1 \\ 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} 3.5 \\ -6.1 \\ 11 \\ 1 \end{bmatrix}$$

Retour à OpenGL :

Revenons maintenant à OpenGL. Si vous vous rappelez le didacticiel précédent, je vous ai indiqué comment faire tourner le cube avec la fonction `glRotatef()`. Grâce au paragraphe précédent, vous allez pouvoir comprendre ce que font réellement la fonction `glRotatef()` et les autres fonctions de transformation que nous allons voir.

La bibliothèque OpenGL met à la disposition du programmeur trois matrices : une matrice de transformation-visualisation (celle qui nous intéresse aujourd'hui), une matrice de projection, et une matrice de texture. Ces trois matrices sont stockées dans des structures de données internes à la bibliothèque, et en général, l'utilisateur n'y accède directement.

Afin de travailler sur ces matrices, OpenGL définit une matrice "active", c'est-à-dire la matrice sur laquelle vont être effectuées les transformations qui suivront. Pour choisir la matrice active, on utilise:

`glMatrixMode(GLenum mode)`

Le paramètre "mode" désigne la matrice que l'on souhaite activer. Il peut prendre comme valeur `GL_MODELVIEW`, `GL_PROJECTION`, et `GL_TEXTURE`. Pour effectuer des transformations sur les objets de la scène, il faut modifier la matrice de transformation-visualisation (`GL_MODELVIEW`). Nous reviendrons sur les deux autres matrices dans les prochains chapitres. Par défaut, la matrice de modélisation-visualisation est la matrice active, ce qui explique pourquoi nous n'avons pas eu besoin d'utiliser `glMatrixMode()` sur le cube tournant.

Définir une transformation avec OpenGL consiste à placer la matrice correspondant à cette transformation dans la matrice de modélisation-visualisation. Bien qu'il soit possible de spécifier directement les 16 coefficients de la matrice active grâce à la fonction `glLoadMatrix()`, on procède en général autrement, en utilisant les fonctions:

`glLoadIdentity(),`
`glRotate(),`
`glTranslate,`
`glScale().`

`void glLoadIdentity();`

cette fonction a pour effet de placer dans la matrice active la matrice dite d'identité. Cette matrice correspond à une transformation nulle : les points ne sont pas déplacés. Quel en est l'intérêt ? Cette fonction permet de "remettre à zéro" la matrice de transformation.

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void glTranslatef(float x,float y,float z);
```

Cette fonction (et sa variante `glTranslated()` dont les paramètres sont de type 'double') multiplie (à droite) la matrice active par une matrice de translation de vecteur (x,y,z).

```
void glRotatef(float theta,float x,float y,float z);
```

Cette fonction multiplie la matrice active par une matrice de rotation d'angle θ autour de l'axe passant par l'origine et porté par le vecteur (x,y,z).

```
void glScalef(float hx,float hy,float hz);
```

Cette fonction multiplie la matrice active par une matrice d'homothétie dont les facteurs suivant les axes X, Y et Z sont respectivement hx, hy et hz.

Prenons un exemple concret : on souhaite placer dans la matrice de modélisation-visualisation une matrice correspondant à une translation de vecteur $(0,1,0)$ suivie d'une rotation d'angle 45° autour de l'axe Z.

En notant M_t la matrice de translation et M_r la matrice de rotation, on désire placer dans la matrice active le résultat du produit $M_r.M_t$ (rappelez vous que l'ordre d'écriture des matrices est inversé par rapport à l'ordre d'application des transformations). La portion de code OpenGL pour accomplir cette tâche est la suivante :

```
glLoadIdentity();  
glRotatef(45.0,0.0,0.0,1.0);  
glTranslatef(0.0,1.0,0.0);
```

A priori, on ne sait pas ce qui se trouve dans la matrice, et donc avant d'effectuer la moindre opération, il convient de la réinitialiser.

Le programme exemple :

L'exemple que je vous propose illustre le principe de composition de transformations. Intéressons-nous tout d'abord à la fonction d'affichage. L'objectif est de dessiner un carré 2D centré en l'origine, auquel on a appliqué une rotation d'angle a autour de l'axe Z , suivie d'une translation de vecteur $(0.5, 0.0, 0.0)$ et d'une rotation d'angle b autour de Z . En appliquant la procédure que l'on vient d'énoncer, voici ce que nous devons mettre dans la fonction d'affichage :

```
glLoadIdentity();  
glRotatef(b,0.0,0.0,1.0);  
glTranslatef(0.5,0.0,0.0);  
glRotatef(a,0.0,0.0,1.0);
```

```
glBegin(GL_POLYGON);  
glVertex3f(-0.2,-0.2, 0.0);  
glVertex3f( 0.2,-0.2, 0.0);  
glVertex3f( 0.2, 0.2, 0.0);  
glVertex3f(-0.2, 0.2, 0.0);  
glEnd();
```

Pour animer notre cube, nous allons utiliser une nouvelle fonction de rappel : la fonction d'oisiveté (idle en anglais). Cette fonction est appelée chaque fois que le gestionnaire d'évènement n'a aucun autre évènement à traiter.

```
void idle()
{
    a+=inca;
    if (a>360) a-=360;
    b+=incb;
    if (b>360) b-=360;
    glutPostRedisplay();
}
```


Dans la fonction idle, les valeurs des angles de rotation a et b sont incrémentées, puis une demande de rafraîchissement est faite. Le cube est alors redessiné à l'écran avec prise en compte des nouvelles valeurs des angles de rotation. En tenant compte du fait qu'une rotation de 360 degrés nous ramène au point de départ, on prend soin d'éviter les dépassements de capacité en ôtant 360 degrés aux angles dès qu'ils dépassent cette valeur.

Afin de vous permettre de visualiser l'effet de chacune des rotations, le programme vous donne la possibilité de modifier les incréments appliqués aux angles a et b à chaque pas d'animation. La touche 'a' augmente l'incrément de la rotation d'angle a. 'Shift+a' le diminue. De la même manière, les combinaisons de touches 'b' et 'Shift+b' modifient l'incrément de la rotation d'angle b.

Vous avez peut-être remarqué que `glRotate()` ne permet de générer que des rotations dont l'axe passe par l'origine. Comment faire si on souhaite obtenir une rotation dont l'axe passe par un point P de coordonnées (P_x, P_y, P_z) ? C'est très simple : il suffit de ramener le point P à l'origine par une translation de vecteur $(-P_x, -P_y, -P_z)$, d'appliquer la rotation, puis de ramener l'objet à sa position initiale avec une translation de vecteur (P_x, P_y, P_z) , ce qui donne en OpenGL :

```
glTranslatef(Px,Py,Pz);
```

```
glRotatef(a,Ax,Ay,Az);
```

```
glTranslatef(-Px,-Py,-Pz);
```

Le programme complet

Cliquez [ici](#) pour obtenir le source.



6

Eclairage et matériaux



Nous allons faire en sorte d'avoir des rendus plus vraisemblables, en mettant en œuvre les moyens offerts par OpenGL pour simuler un éclairage réaliste. Comme d'habitude, nous commencerons par quelques éléments théoriques avant de nous appuyer sur un exemple concret pour aborder la pratique.



Eclairage et matériaux réalistes :

Jusqu'à présent nous nous sommes contentés d'affecter à chacun des sommets de nos polygones une couleur fixe. Dans la réalité, un point de couleur apparaîtra différemment suivant l'éclairage auquel il est soumis et l'angle sous lequel vous l'observez. OpenGL permet d'effectuer un rendu prenant en compte l'illumination des objets. Pour simuler de manière réaliste une scène tridimensionnelle il est nécessaire de reproduire les lois physiques qui régissent la lumière. Ces lois sont complexes. On se contente donc de choisir un modèle mathématique aussi proche que possible de la réalité. En fonction de l'application à laquelle il est destiné, le modèle d'illumination choisi permettra d'obtenir des résultats plus ou moins réalistes et rapides. Le modèle utilisé par le moteur de rendu de Blender donne des résultats beaucoup plus réalistes que celui d'OpenGL. En revanche, le rendu d'une scène OpenGL est bien plus rapide.

Pour étudier le modèle d'illumination utilisé par OpenGL, il nous faut aborder 3 points :

- **Les sources lumineuses** : quels sont les paramètres permettant de définir une source de lumière ?
- **Les matériaux** : Une brique ne réfléchit pas la lumière de la même manière qu'une plaque d'aluminium : le matériau est un facteur primordial pour l'éclairage d'une scène.
- **L'algorithme de remplissage** : calculer l'éclairage en chaque pixel de l'image est trop coûteux en temps. OpenGL utilise une astuce pour accélérer le rendu.

Sources lumineuses :

Une source de lumière est caractérisée par 10 paramètres qu'on peut classer en 4 catégories :

1 - Les paramètres de lumière

La lumière émise par une source est formée de trois composantes : la plus importante, la composante **diffuse**, est réfléchiée par un objet dans toutes les directions. La composante **spéculaire** correspond à la lumière qui est réfléchiée dans une direction privilégiée (et qui est donc à l'origine de l'effet de brillance). La composante **ambiante** qui est une lumière non directionnelle. OpenGL vous permet d'affecter une lumière ambiante pour toute la scène. La composante ambiante d'une source ajoute une contribution à cette lumière ambiante globale. Vous avez la possibilité d'affecter une couleur à chacune des trois composantes.

Les paramètres de lumière diffuse, spéculaire et ambiante sont difficiles à appréhender pour le novice. Comme rien ne vaut la pratique, la meilleure chose à faire est d'expérimenter. Je vous conseille de modifier le programme exemple (à la fin) de façon à pouvoir faire varier les paramètres de lumière d'une des sources par l'intermédiaire du clavier et observez l'influence de chacune des composantes sur le rendu de la scène.

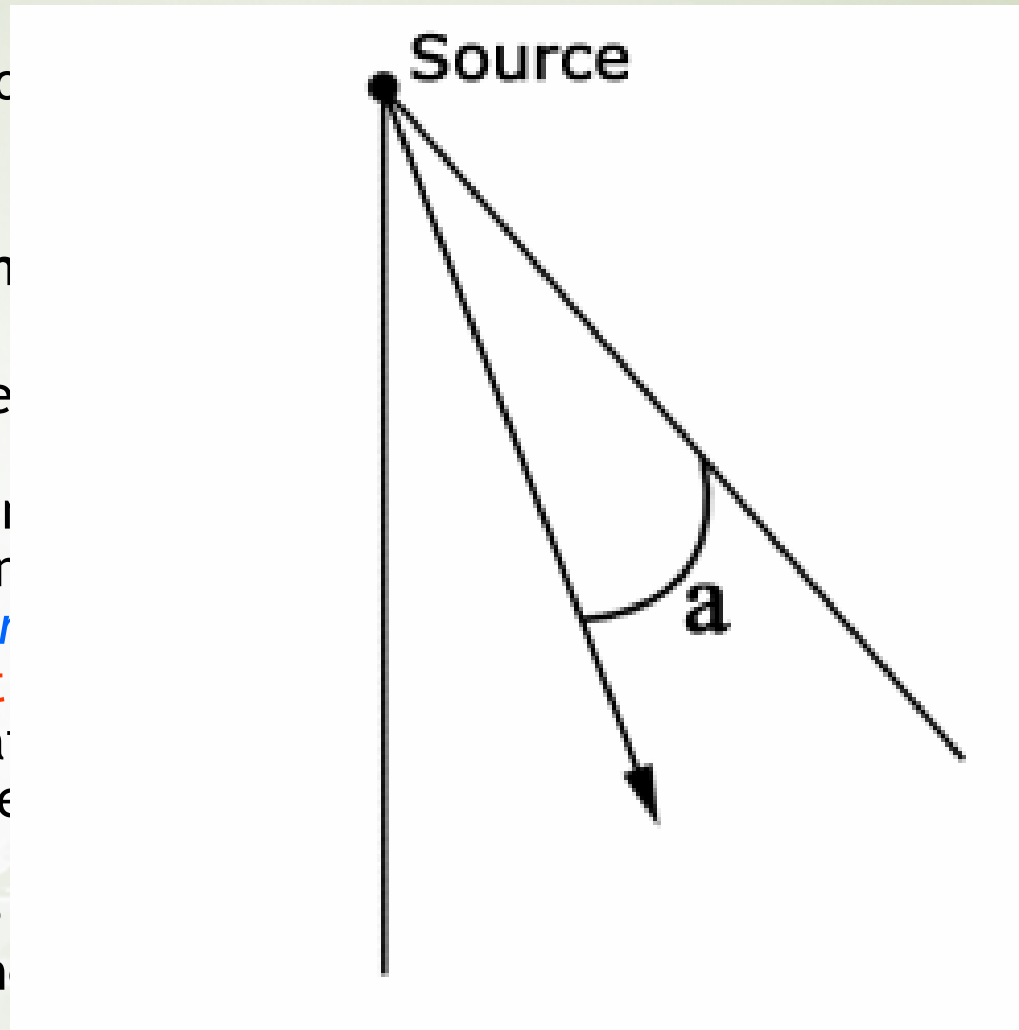
2 - Le paramètre de position

Comme source

position de la scène.

3 - Les param

L'angle de
souhaitez
lumière ou
vous défin
autres par
l'exposant
concentra
spot. Si l'e
toutes les
valeur de
l'axe donn



urce que vous
une source de
ntre 0 et 90°,
lifier les deux
le il pointe et
faire varier la
définissant le
galement dans
aites croître la
rée autour de

4 - Les paramètres d'atténuation

Ils permettent de prendre en compte le phénomène physique suivant : plus un objet est éloigné d'une source de lumière, moins il est éclairé par cette dernière. Les paramètres d'atténuation sont au nombre de 3 : le **facteur d'atténuation constante**, le **facteur d'atténuation linéaire** et le **facteur d'atténuation quadratique**. Si on note respectivement A_c , A_l et A_q ces trois coefficients, l'intensité reçue par un point P situé à une distance d de la source lumineuse est divisée par $A_c + A_l * d + A_q * d * d$. Par défaut, $A_c = 1$ et $A_l = A_q = 0$, ce qui correspond à une atténuation nulle (division par 1).

En informatique, une couleur est représentée par un triplet de composantes rouge, verte, et bleue. Pour définir un le comportement d'un objet vis-à-vis d'une couleur RVB, il suffit de dire quel pourcentage de chaque composante est réfléchi. Ainsi, si on affecte à un matériau les pourcentages ($R=1$, $V=0.5$, $B=0$), soit ($R=100\%$, $V=50\%$, $B=0\%$), et si on l'éclaire avec une lumière blanche ($R=1, V=1, B=1$), le matériau va réfléchir un rayon de couleur ($R=1, V=0.5, B=0$), et votre objet vous paraîtra orange. Vous remarquerez que si vous éclairez cet objet avec une lumière bleue ($R=0, V=0, B=1$), le matériau va renvoyer une rayon de couleur noire ($R=0, V=0, B=0$) qui correspond à une absence de lumière.

Le paramètre de couleur émise correspond à une composante de lumière supplémentaire, qui permet de prendre en compte le fait que certains objets peuvent émettre eux-mêmes de la lumière. Si vous modélisez une ampoule allumée, vous pourrez attribuer du blanc ou du jaune à la couleur émise par le matériau de l'objet. Pour les matériaux classiques, la couleur émise est noire. Cependant, cette couleur émise n'est pas considérée comme une source de lumière pour les autres objets de la scène.

Pour expliquer le dernier paramètre, le coefficient de brillance, revenons à ce que nous avons vu concernant la lumière spéculaire : elle est à l'origine du phénomène de brillance qui crée des tâches de lumière intense sur les objets. La composante spéculaire de la lumière est réfléchie dans une direction privilégiée. Dans la réalité, cette réflexion ne se fait jamais de manière parfaite, et le coefficient de brillance permet de modéliser cette imperfection. Sa signification physique est un étalement des taches spéculaires sur les objets lorsqu'on diminue la valeur du coefficient.

L'algorithme d'éclairage :

Pour des raisons d'efficacité, OpenGL ne calcule pas la couleur de chaque pixel d'un polygone : soit il remplit chaque polygone avec un couleur unie (mode de remplissage 'Flat'), soit il utilise un algorithme de Gouraud (mode 'Smooth'), dont le principe est une interpolation de la couleur de chacun des sommets. Pour calculer correctement la réflexion des rayon lumineux en un point, OpenGL a besoin de connaître la perpendiculaire à la surface de l'objet au point considéré. On appelle cette donnée une normale. Nous aurons l'occasion de revenir longuement sur la question des normales dans un prochain chapitre, et dans le programme exemple, nous utiliserons une thèière générée avec ses normales par glut.

L'exemple :

Passons maintenant au programme exemple. On affiche à l'écran une théière générée par glut et éclairée par 2 sources lumineuses différentes. Vous avez la possibilité de tourner autour de la théière avec les touches 'a' et 'z', et vous pouvez faire varier certains paramètres d'éclairage avec d'autres touches (jetez un coup d'œil à la fonction de rappel `clavier()` pour connaître toutes les variables modifiables).

Paramètres d'éclairage :

L'architecture du programme est classique et les seules nouveautés concernent l'utilisation du modèle d'illumination. La phase d'initialisation de l'éclairage commence par la spécification du mode remplissage des polygones avec :

```
glShadeModel(GL_SMOOTH);
```

Ensuite on indique à OpenGL qu'on souhaite utiliser le calcul d'éclairage, en activant la variable d'état GL_LIGHTING :

```
glEnable(GL_LIGHTING);
```

OpenGL permet d'utiliser jusqu'à huit sources de lumière. Ces huit lampes sont indexées par les constantes `GL_LIGHT0` à `GL_LIGHT7`. Il faut activer chacune des sources qu'on souhaite utiliser (2 dans notre cas) :

```
glEnable(GL_LIGHT0);
```

```
glEnable(GL_LIGHT1);
```

Vient ensuite le paramétrage des lampes. Il se fait avec une unique fonction, `glLightf()`, dont le prototype est le suivant :

```
void glLight{i,f}GLenum lampe, GLenum  
nomparam, GLType param)
```

'param' désigne la valeur à affecter au paramètre choisi. Vous remarquerez que les paramètres sont passés sous forme de tableaux.

La définition de la position des sources de lumières se trouve dans la fonction d'affichage. En effet, tout comme les sommets des polygones, les paramètres de position et de direction d'une source subissent les transformations contenues dans la matrice de modélisation-visualisation. Il faut donc placer judicieusement la déclaration de ces deux paramètres. Les deux spots que nous utilisons sont omnidirectionnels (car nous ne modifions pas la valeur par défaut de `GL_SPOT_CUTOFF` qui vaut 180), et donc le paramètre direction ne nous est pas utile.

Voici l'exemple de la théière




```
void glLight{i f}[v](GLenum nb, GLenum pname, TYPE p);
```

Pname

GL_AMBIENT(R,G,B,A): couleur de la lumière ambiante ajoutée à la scène

GL_DIFFUSE(R,G,B,A): couleur de la lumière diffuse liée à la source lumineuse

GL_SPECULAR(R,G,B,A): couleur de la lumière spéculaire liée à la source lumineuse

GL_POSITION(x,y,z,w): position de la lumière

si $w = 0$, lumière placée à l'infini, (x,y,z) donne la direction vers la source de lumière

si $w \neq 0$, lumière non placée à l'infini, (x,y,z) donne la position

GL_SPOT_DIRECTION orientation du cône d'ouverture dans le cas d'un spot

GL_SPOT_EXPONENT exposant d'un spot (plus l'exposant est grand plus le spot est focalisé sur son axe)

GL_SPOT_CUTOFF angle d'ouverture d'un spot

GL_CONSTANT_ATTENUATION

facteur d'atténuation constant: K_c

GL_LINEAR_ATTENUATION

facteur d'atténuation linéaire: K_l

GL_QUADRATIC_ATTENUATION

facteur d'atténuation quadratique: K_q

Facteur d'atténuation en fonction de la distance d
entre la lumière et le point considéré:

$$1 / (K_c + d * K_l + d^2 * K_q)$$

Valeurs par défaut

GL_AMBIENT(0.0,0.0,0.0,1.0)

GL_DIFFUSE

GL_LIGHT0:(1.0,1.0,1.0,1.0)

autres:(0.0,0.0,0.0,1.0)

GL_SPECULAR

GL_LIGHT0:(1.0,1.0,1.0,1.0)

autres: (0.0,0.0,0.0,1.0)

GL_POSITION (0.0,0.0,1.0,0.0)

GL_SPOT_DIRECTION (0.0,0.0,-1.0)

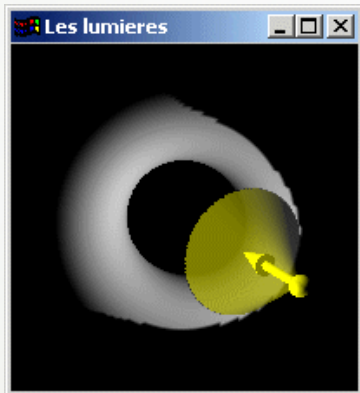
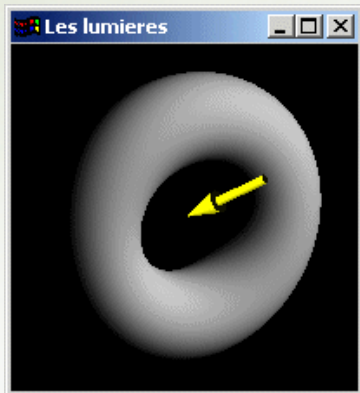
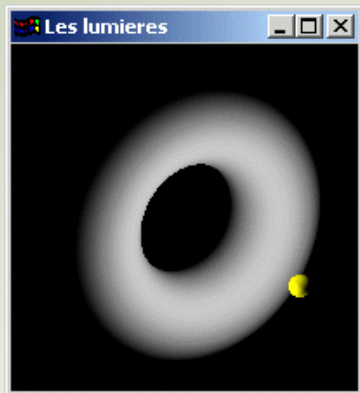
GL_SPOT_EXPONENT 0.0

GL_SPOT_CUTOFF 180.0

GL_CONSTANT_ATTENUATION 1.0

GL_LINEAR_ATTENUATION 0.0

GL_QUADRATIC_ATTENUATION 0.0



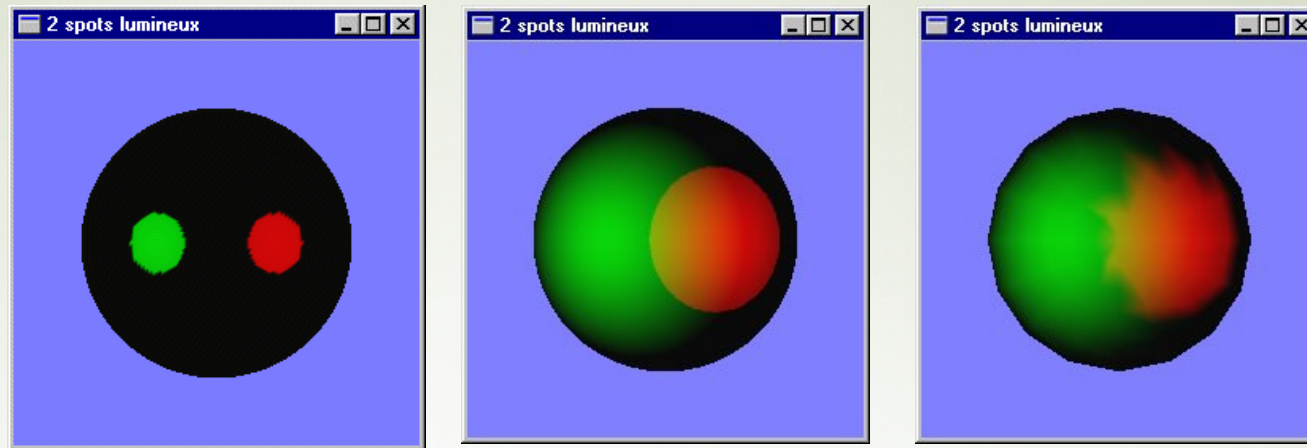
Valeurs				
DIFFUSE	0.80	0.80	0.80	
AMBIENT	0.00	0.00	0.00	
SPECULAR	0.00	0.00	0.00	
POSITION	0.00	0.00	2.30	1.00
SPOT CUTOFF	180.00			
SPOT DIR	0.00	0.00	-1.00	

Valeurs				
DIFFUSE	0.80	0.80	0.80	
AMBIENT	0.00	0.00	0.00	
SPECULAR	0.00	0.00	0.00	
POSITION	0.40	0.50	0.70	0.00
SPOT CUTOFF	180.00			
SPOT DIR	0.00	0.00	-1.00	

Valeurs				
DIFFUSE	0.80	0.80	0.80	
AMBIENT	0.00	0.00	0.00	
SPECULAR	0.00	0.00	0.00	
POSITION	0.00	0.00	2.00	1.00
SPOT CUTOFF	30.00			
SPOT DIR	0.00	0.00	-1.00	

Le programme est [ici](#) (prog en cpp)

Exemple de définition de spots



Le programme est [ici](#)

Lecture de la configuration des sources lumineuses

```
void glGetLight{i f}v(GLenum nb, GLenum pname, TYPE *p);
```

Lecture de la configuration d'une des sources lumineuses

nb: GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7

pname: caractéristique affectée

p: tableau destiné à la récupération de la caractéristique pname.

Autorisation des calculs d'éclairage

Les calculs d'éclairage ne sont réalisés qu'après exécution de:

```
glEnable(GL_LIGHTING);
```

Faute de cette autorisation, ce sont les informations de couleurs spécifiées par les fonctions glColor* qui sont prises en compte.

Chaque lumière (8 au maximum) doit être elle-aussi autorisée:

```
glEnable(GL_LIGHTi);
```

Ces options peuvent être annulées au moyen de `glDisable(GL_LIGHTi)`.

Choix d'un modèle d'illumination

Le modèle d'illumination OpenGL gère les trois composantes suivantes:

- l'intensité de lumière ambiante globale permettant de fixer l'éclairage minimum au sein de la scène via les composantes ambiantes des matériaux,
- la position du point de vue (position de l'observateur local, ou à l'infini) utilisé pour l'évaluation des réflexions spéculaires,
- si les calculs d'éclairage doivent être effectués différemment pour les deux faces des facettes modélisant les objets ou non.

```
void glLightModel{if}[v](GLenum md,TYPE v);
```

Configuration du modèle d'illumination

md: GL_LIGHT_MODEL_AMBIENT,
GL_LIGHT_MODEL_LOCAL_VIEWER,
GL_LIGHT_MODEL_TWO_SIDE

v: valeur affectée à md



Le blending (transparence)

OpenGL permet d'afficher un objet avec composition de la couleur de chacun de ses pixels avec la couleur du pixel déjà présent dans l'image à cette place -> transparence.

glEnable(GL_BLEND);

Activation du blending.

glBlendFunc(GLenum sfactor, GLenum dfactor);

Choix de l'arithmétique de composition des pixels sources et destination (sfactor pour les pixels de l'objet affiché, dfactor pour les pixels déjà présents).

Pour le blending les valeurs habituelles de sfactor et dfactor sont

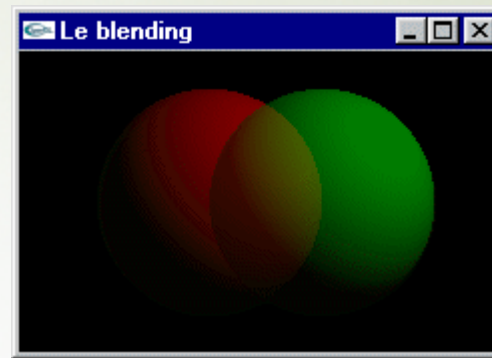
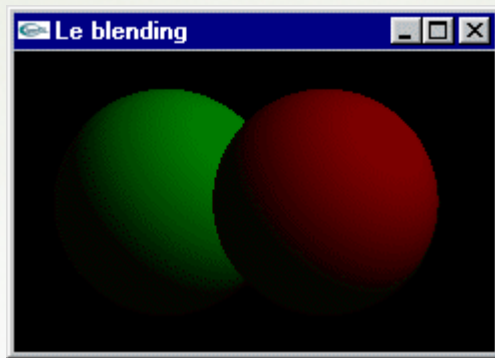
GL_SRC_ALPHA et GL_ONE_MINUS_SRC_ALPHA

-> on garde 1-alpha du pixel présent et on ajoute alpha du pixel calculé

-> objet non transparent avec alpha = 1

-> objet transparent avec alpha = 0.

ATTENTION: La composition entre les pixels présents et placés implique que l'ordre de dessin des objets graphiques a une influence sur l'image finale.



Le programme est [ici](#)

Matériau

La spécification d'un matériau pour un objet se fait par l'intermédiaire de 5 paramètres :

- La couleur diffuse
- La couleur spéculaire
- La couleur ambiante
- La couleur émise
- Le coefficient de brillance

Paramètres de matériaux :

Le système d'affectation des propriétés de matériau utilise le principe de machine à états. OpenGL gère un matériau courant. Lorsqu'un polygone est décrit, il se voit affecter le matériau courant. La modification du matériau courant se fait avec la fonction `glMaterialfv()` :

Void `glMaterialfv(GLenum face, GLenum nomparam, Gltype param)` ;

'face' indique la face (avant ou arrière) dont on souhaite modifier les paramètres. Nous n'avons pas encore abordé les considérations de face, et nous nous satisferons de la valeur `GL_FRONT_AND_BACK`. Tout comme pour `glLightfv()`, `nomparam` désigne la propriété qu'on souhaite changer, et 'param' est un tableau contenant la nouvelle valeur à affecter à 'nomparam'. Les valeurs de 'nomparam' possibles sont :

- `GL_AMBIENT`
- `GL_DIFFUSE`
- `GL_SPECULAR`
- `GL_EMISSION`
- `GL_SHININESS` (i.e. coefficient de brillance)

7

Les textures



Le placage de texture

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 ?

Etapes de l'utilisation d'une texture:

- (1) spécification de la texture,
- (2) spécification de la technique à utiliser pour appliquer la texture à chaque pixel,
- (3) autorisation du plaquage,
- (4) dessin de la scène qui sera implicitement texturée au moyen des paramètres définis en (1) et (2).

Voici un exemple de texture pour la formula 1 [ici](#)



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 *replace* remplace la couleur de l'objet 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.
- Le mode *decal* avec une texture RGBA utilise la composante alpha de la texture pour combiner 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 coordonnées de texture en plus des coordonnées géométriques pour les objets à texturer.

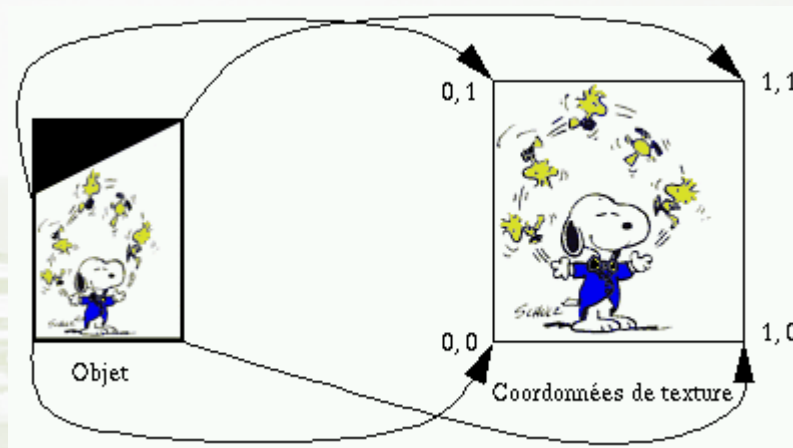
Les coordonnées de texture

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

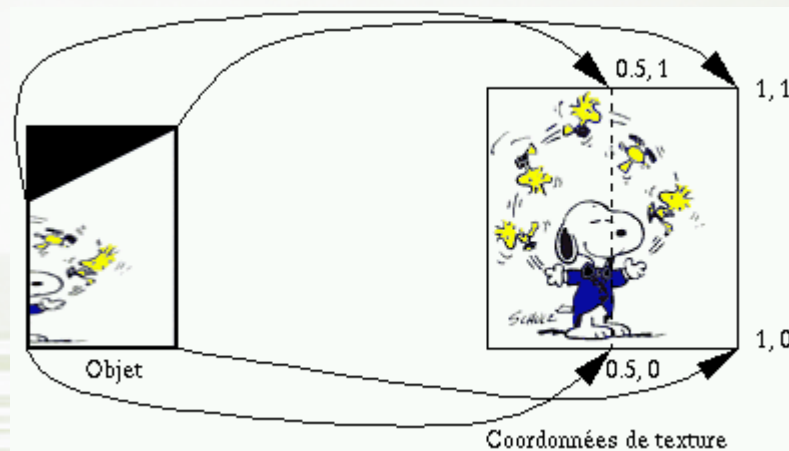
On assigne avec [glTexCoord*\(\)](#) pour chaque sommet des objets à texturer un couple de valeurs qui indique les coordonnées de texture de ce sommet.



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 texture de $(0.5, 0.0)$ $(1.0, 0.0)$ $(1.0, 1.0)$ et $(0.5, 1.0)$ aux coins du quadrilatère.



Exercice

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

(il utilise l'image [snoopy2.ppm](#). Vous pouvez aussi télécharger [l'archive du projet Dev-Cpp](#))

Quelles coordonnées de texture permettent de plaquer uniquement la tête de snoopy sur le quadrilatère ?

Répétition de la texture

Il faut indiquer comment doivent être traitées les coordonnées de texture en dehors de l'intervalle [0.0, 1.0]. Est-ce que la texture est répétée pour recouvrir l'objet ou au contraire "clampée" ? Pour ce faire, utilisez la commande [glTexParameter\(\)](#) pour positionner les paramètres GL_TEXTURE_WRAP_S pour la dimension horizontale de la texture ou GL_TEXTURE_WRAP_T pour la dimension verticale à GL_CLAMP ou GL_REPEAT.

Exercice

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

(1) Spécification de la texture

```
void glTexImage2D(GLenum target, GLint level, GLint  
    component, GLsizei l, GLsizei h, GLint b, GLenum format, GLenum  
    type, const GLvoid *pix);
```

Définit une texture 2D.

target: GL_TEXTURE_2D

level: 0 pour une texture à 1 niveau

component: nombre de composantes à gérer

1 -> rouge

2 -> R et A

3 -> R, V et B

4 -> R, V, B et A

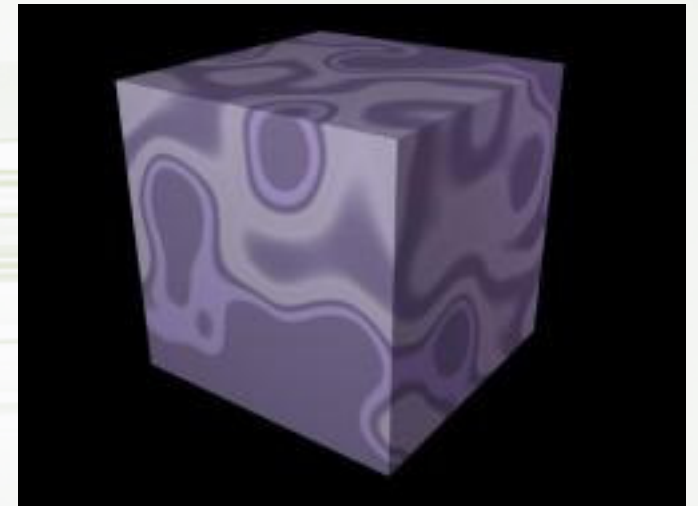
l et h: dimension de l'image représentant la texture. l et h sont obligatoirement des puissances de 2.

b: largeur du bord (usuellement 0). Si $b \neq 0$ alors l et h sont incrémentés de $2 \times b$.

format et type: format et type de données de l'image (voir glReadPixels)

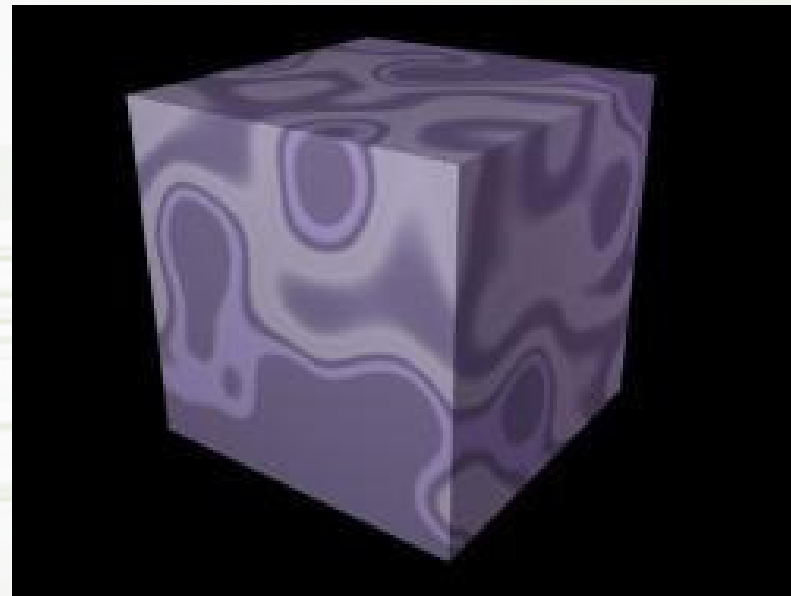
pix: image à placer stockée dans un tableau

Il est intéressant de noter que si dans notre cas l'image utilisée est chargée depuis un fichier au format JPEG, il est tout à fait possible d'utiliser des images calculées à partir de fonctions mathématiques. On parle alors de textures procédurales. Les textures procédurales les plus utilisées sont sans doute les textures de type « nuage » et « bruit de Perlin »



Contrairement à ce qu'on pourrait penser, les textures ne se bornent pas aux images en 2 dimensions. OpenGL permet également d'utiliser des textures 1D, peu intéressantes, et des textures 3D. S'il paraît difficile d'obtenir des images en 3 dimensions (bien que ce soit possible), les textures procédurales 3D sont courantes, et elles possèdent de nombreux avantages par rapport aux textures 2D, notamment en ce qui concerne la continuité des textures aux arêtes.

Pour s'en convaincre, il suffit de comparer les figures 1 et 2. L'image collée sur chacune des faces du cube de la figure 1 est la même, et le sens des veinures du bois n'est pas cohérent. En revanche, sur la texture procédurale 3D de la figure 2, la continuité de la texture est assurée sur chacune des faces.



OpenGL ne prend pas en charge la lecture de fichier au format JPEG, et il serait illusoire de vouloir écrire aujourd'hui une routine adéquate, l'algorithme de compression étant très complexe. Nous allons donc appeler à la rescousse la bibliothèque C JPEG. Cette bibliothèque est livrée avec toutes les distributions standard. Cependant, les fichiers d'entête ne sont pas forcément installés par défaut (par exemple, sur les distributions Mandrake, la bibliothèque se trouve dans l'archive libjpeg.rpm et les fichiers d'entête sont dans libjpeg-devel.rpm). Si vous ne parvenez pas à compiler un programme utilisant la bibliothèque JPEG, installez le package devel correspondant

Pour utiliser la bibliothèque JPEG, il faut commencer par inclure les fichiers d'entête :

```
#include <jpeglib.h>
```

```
#include <jerror.h>
```

Le processus de décompression nécessite la mise en œuvre de 2 structures de données nommées `cinfo` et `jerr`. La première va contenir les informations concernant la structure de l'image (ses dimensions, son type, ses paramètres de compression...). On initialise cette structure grâce à la fonction `jpeg_create_decompress()`. La seconde structure, `jerr`, va servir au traitement des erreurs qui surviendront éventuellement au cours de la décompression. Par un appel à `jpeg_stdio_src()`, on indique que les messages d'erreur doivent être envoyés sur la sortie standard. Il convient ensuite d'ouvrir le fichier contenant l'image grâce à un classique `fopen()`. On prendra soin de vérifier que l'ouverture du fichier s'est bien passée en testant la valeur de retour de `fopen()`. On indique ensuite que les données concernant l'image à décompresser seront lues depuis le fichier que l'on vient d'ouvrir, avec un appel à `jpeg_stdio_src()`.

A présent, il ne reste plus qu'à lire l'entête du fichier avec `jpeg_read_header()`. Les informations concernant l'image sont alors placées dans la structure `cinfo`. On peut donc tester si celles-ci sont conformes à nos souhaits. Nous voulons une image de taille 256x256 en mode RGB. Si les tests sont passés avec succès, on peut commencer la décompression de l'image par un appel à `jpeg_start_decompress()`. L'image sera décompressée dans un tableau que nous aurons pris soin de déclarer. Sachant que nous souhaitons lire une image de 256x256 et qu'un pixel contient trois composantes (R, V et B) stockées chacune sur un octet, nous avons besoin d'un tableau de 256x256x3 octets (le type correspondant en langage C est `unsigned char` qui peut prendre 256 valeurs différentes, de 0 à 255).

```
unsigned char image[256*256*3];
```


Vous remarquerez que nous stockons l'image dans un tableau unidimensionnel. La bibliothèque JPEG ne permet pas de décompresser l'image dans un format qu'accepte OpenGL (il lui faut un tableau à trois dimension où la troisième dimension correspond à la composante couleur R,V,B). Nous serons donc obligés de réorganiser le tableau après la fin de la lecture de l'image. La procédure de décompression se fait par un balayage de ligne avec une boucle « tant que » et un pointeur nommé « ligne » qui indique l'adresse à laquelle doivent être placées les données décompressées. La fonction `jpeg_read_scanlines()` provoque la décompression d'une ligne de l'image. Une fois la décompression de l'image achevée, il ne reste plus qu'à terminer le processus avec `jpeg_finish_decompress()` et à libérer la structure cinfo avec `jpeg_destroy_decompress()`.

La double boucle imbriquée qui termine la fonction [LoadJpegImage\(\)](#) copie les données du tableau `image[]` dans un nouveau tableau à 3 dimensions qui pourra être exploité comme une texture par OpenGL. Le tableau `image[]` contient la suite des composantes R,V et B de chacun des pixels, en parcourant l'image de gauche à droite et de haut en bas. Avec des indices démarrant à 0, la composante rouge du pixel de la ligne d'indice 4 et de la colonne d'indice 12 de l'image se trouve dans `image[256*4+12]`, la composante verte de ce même pixel se trouve dans `image[256*4+12+1]` et sa composante bleue est dans `image[256*4+12+2]`. Dans le tableau texture, ces mêmes composantes se trouvent respectivement dans `texture[4][12][0]`, `texture[4][12][1]`, `texture[4][12][2]`.

Utilisation de la texture

L'utilisation du placage de texture faite dans le programme est quasiment la plus simple qui soit : nous n'utilisons qu'une texture, et nous la plaquons sur des faces carrées.

La première étape nécessaire au placage de la texture est évidemment ... le chargement de l'image avec la fonction `LoadJpegImage()`. Nous allons ensuite paramétrer la manière dont OpenGL devra filtrer la texture lors de l'application de celle-ci sur un objet. Lors du rendu, la texture va être déformée par la perspective. A certains endroits elle va être étirée, à d'autre elle va être rétrécie. Le filtrage définit la méthode de calcul final de la texture déformée. OpenGL propose les méthodes « nearest » (la plus rapide) et « linear » (la plus jolie). Afin de vous permettre de comparer les deux méthodes, le programme vous permet de basculer d'une méthode à l'autre grâce aux touches 'n' et 'l'. Le paramétrage de la méthode de placage de texture se fait grâce à la fonction

```
void glTexParameteri(GLenum cible, GLenum nomparam, GL valeur)
```


'cible' définit le type de texture que l'on veut paramétrer (GL_TEXTURE_1D, GL_TEXTURE_2D ou GL_TEXTURE_3D). nomparam désigne le paramètre que l'on souhaite modifier. Nous ne modifierons aujourd'hui que la méthode de filtrage lors d'un étirement (GL_TEXTURE_MAG_FILTER) ou d'un rétrécissement (GL_TEXTURE_MIN_FILTER). 'valeur' correspond à la nouvelle valeur à affecter au paramètre. Dans notre cas, il s'agit de GL_LINEAR ou GL_NEAREST.

L'étape suivante consiste à définir la texture 2D que nous souhaitons utiliser avec :

```
void glTexImage2D(GLenum cible, GLint niveau, GLint  
formatinterne, GLsizei largeur, GLsizei hauteur, GLint  
bord, GLenum format, GLenum type, const GLvoid  
*texture)
```

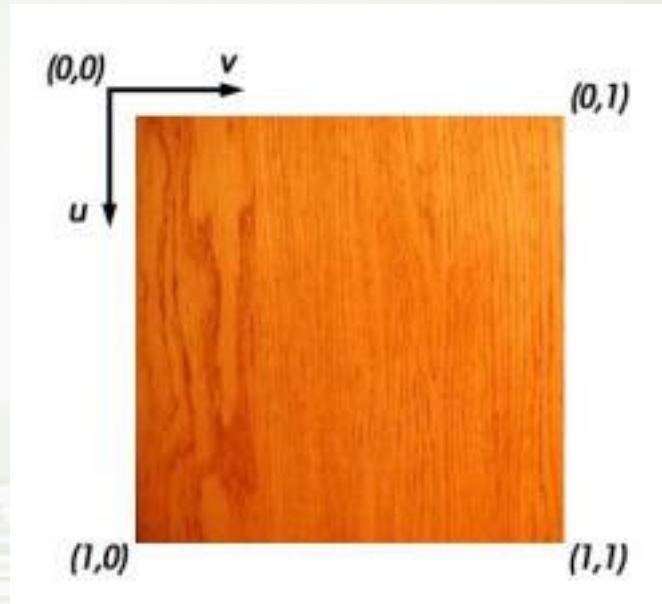
La cible sera pour nous GL_TEXTURE_2D. le paramètre de niveau n'est utile que si vous utilisez le mipmapping (textures multirésolution). Nous lui donnerons la valeur 0. Le format interne de stockage de la texture sera GL_RGB. La hauteur et la largeur de la texture valent toutes les deux 256. La texture ne possède pas de bord (valeur 0). L'image fournie est en mode RGB et chaque composante est codée sur un caractère non signé, donc on affectera respectivement GL_RGB et GL_UNSIGNED_BYTE aux paramètres de format et de type.

Il est également nécessaire d'activer l'utilisation de la texture2D grâce à l'appel

`glEnable(GL_TEXTURE_2D)`

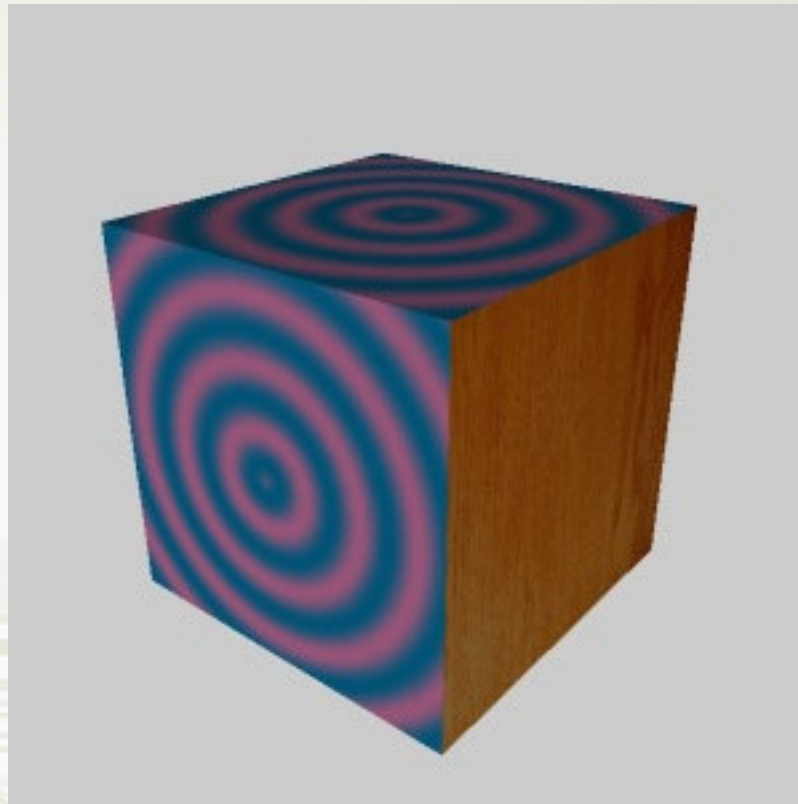
Enfin la dernière étape consiste à positionner la texture sur le polygone. En anglais, on parle de «UV mapping». Le principe est simple : on affecte un système de coordonnées (u,v) à la texture suivant l'illustration de la figure 4. Positionner la texture consiste à affecter à chaque sommet d'un polygone la coordonnée de la texture en ce point. Dans notre cas, puisque les polygones sont des carrés, la tâche est simple, les coordonnées de textures aux sommets des points des faces sont simplement (0,0), (1,0), (1,1) et (0,1). A l'instar de la couleur des sommets, les coordonnées de texture se spécifient lors de la déclaration des polygones entre un `glBegin()` et un `glEnd()`. Le principe est toujours le même : lorsqu'un sommet est déclaré avec `glVertex()`, la coordonnée de texture courante lui est affectée.

La modification de la coordonnée de texture courante se faite par un appel à `glTexCoord2f(GLfloat u,GLfloat v)`



Pour accéder au source (attention à l'inclusion de bibliothèques) : [ici](#)

textures procédurales, l'animation, éclairage de textures



Utilisation de plusieurs textures

Nous avons plaqué sur chacune des faces du cube la même image, nous allons plaquer une image de bois sur 3 faces du cube, et une texture procédurale sur les 3 autres faces du cube. Il nous faut pour cela mettre en œuvre le mécanisme d'objets de texture qu'OpenGL met à notre disposition.



Lorsqu'on souhaite travailler avec plusieurs textures, il faut leur assigner un identifiant, c'est-à-dire un nombre entier. OpenGL dispose d'un système interne de gestion des identifiants de texture ('texture ID' en anglais). Lorsque vous voulez créer une texture, il vous faut demander à OpenGL de vous attribuer un identifiant en faisant un appel à la fonction

Void glGenTextures(GLsizei nombre,GLuint *idtextures)

'nombre' indique le nombre de textures que vous souhaitez créer, et 'idtextures' est un pointeur vers un tableau d'entiers qui contiendra au retour de la fonction les 'nombres' identifiants de texture que vous avez demandés.

Par la suite, on retrouve encore le mécanisme de machine à état d'OpenGL : vous définissez une texture active, et toutes les opérations sur les textures qui seront faites par la suite seront appliquées à la texture active. La fonction que vous permet de définir la texture active est `glBindTexture()`, dont le prototype est le suivant :

```
void glBindTexture(GLenum cible, GLuint idtexture)
```

‘cible’ correspond au type de texture utilisé (GL_TEXTURE_1D, GL_TEXTURE_2D ou GL_TEXTURE_3D). Ce paramètre est nécessaire car l'allocation d'espace pour la texture se fait lors du premier appel à `glBindTexture()` pour un numéro de texture donné, et non lors de l'attribution de l'identifiant. Le paramètre ‘idtexture’ désigne l'identifiant de la texture qu'on souhaite rendre active.

L'utilisation des textures dans un programme OpenGL se fait donc en général de la façon suivante :

- Lors de l'initialisation, on réserve les identifiants de texture et on crée les textures :

```
glGenTextures(2,IdTex);
glBindTexture(GL_TEXTURE_2D,IdTex[0])      ;
/* Création de la première texture (avec
glTexParameteri(),      glTexImage2D...) */
...
glBindTexture(GL_TEXTURE_2D,IdTex[1])      ;
/* Création de la deuxième texture */
...
```


Au cours de la fonction d'affichage, on change éventuellement la texture active avant la description de chaque face :

```
void affichage()  
{  
    ...  
    glBindTexture(GL_TEXTURE_2D, IdTex[0]);  
    glBegin(GL_POLYGON); /* description de la face 1 */  
    glEnd();  
    glBindTexture(GL_TEXTURE_2D, IdTex[1]);  
    glBegin(GL_POLYGON); /* description de la face 2 */  
    glEnd();  
    ...  
}
```

Chargement d'image TIFF

La dernière fois, nous avons chargé une image JPEG grâce à la bibliothèque JPEG. Aujourd'hui, nous allons placer dans notre première texture une image lue dans un fichier TIFF, avec la bibliothèque libtiff. Le format TIFF (Tag Image File Format) est un format de fichier sans perte de qualité et qui offre la possibilité de stocker une couche alpha pour la transparence des images. Je ne vais pas détailler les prototypes des fonctions utilisées pour la lecture du fichier TIFF, vous verrez qu'on trouve bon nombre de similitudes avec la bibliothèque JPEG.

Dans un premier temps, on ouvre le fichier image avec `TIFFOpen()`. Si l'ouverture se passe bien, on récupère la taille de l'image avec la fonction `TIFFGetField()`, puis on alloue un tableau mono-dimensionnel de la taille de l'image avec `TIFFmalloc()`. Un pixel d'image correspond à un quadruplet RGBA, stocké sur 4 octets, ce qui correspond à un entier (type `uint32`). Si l'allocation se passe correctement, on peut lancer la lecture de l'image grâce à la fonction `TIFFReadRGBAImage()`. Ici, il n'y a pas besoin de faire un balayage comme nous l'avons fait avec la `libjpeg`. Au final, l'image est placée dans le tableau 'raster'. Il ne reste plus qu'à réorganiser les données de 'raster' dans un tableau à 3 dimensions nommé 'image', compatible avec la fonction `glTexImage2D()`. Pour faire les choses dans les règles, on libère les données contenues dans 'raster' et dont nous n'avons plus besoin avec un appel à `_TIFFfree()`, puis on ferme le fichier avec `TIFFClose()`.

```

TIFF* tif = TIFFOpen(fichier, "r");
if (tif) {
    TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &l);
    TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &h);
    npixels = l * h;
    raster = (uint32*) _TIFFmalloc(npixels * sizeof (uint32));
    if (raster != NULL) {
        /* lecture de l'image */
        if (TIFFReadRGBAImage(tif, l, h, raster, 1)) {
            /* transfert de l'image vers le tableau 'image' */
            for (i=0;i<256;i++)
                for (j=0;j<256;j++) {
                    image[i][j][0]=((unsigned char *)raster)[i*256*4+j*4+0];
                    image[i][j][1]=((unsigned char *)raster)[i*256*4+j*4+1];
                    image[i][j][2]=((unsigned char *)raster)[i*256*4+j*4+2];
                }
        }
    }
    else {
        printf("erreur de chargement du fichier %s\n",fichier);
        exit(0);
    }
    TIFFfree(raster);
}
TIFFClose(tif);

```


Création d'une texture procédurale

Cela consiste à utiliser une fonction mathématique au lieu d'une image pour habiller un objet 3D. En guise de seconde texture pour notre cube, nous allons utiliser une texture procédurale basée sur la fonction mathématique cosinus. Dans un moteur 3D, le gros avantage des textures procédurales est qu'elles prennent peu de place en mémoire, contrairement à une image. Avec OpenGL nous n'allons pas tirer profit de cet avantage puisque nous devons passer par un tableau pour spécifier une texture avec `glTexImage2D()` : nous remplissons un tableau 2D avec les valeurs d'une fonction mathématique. Dans le programme exemple, le chargement de la texture procédurale se fait dans `chargeTextureProc()`.

```
int fonctionTexture(int x,int y)
{
    float dx=(128.0-(float)x)/255.0*40.0;
    float dy=(128.0-(float)y)/255.0*40.0;
    float a=cos(sqrt(dx*dx+dy*dy)+decalage);
    return (int)((a+1.0)/2.0*255);
}

for (i=0;i<256;i++)
    for (j=0;j<256;j++) {
        a=fonctionTexture(i,j);
        image[i][j][0]=a;
        image[i][j][1]=128;
        image[i][j][2]=128;
    }
```

Animation de la texture procédurale

Nous souhaitons animer la texture que nous venons de créer en faisant bouger les cercles concentriques générés par la fonction. On utilise pour cela une variable 'décalage' introduite dans le cosinus de la fonction procédurale, et la fonction de rappel d'oisiveté ('idle'). Cette fonction est appelée chaque fois que le gestionnaire d'événements n'a rien à faire (pas d'événement clavier, souris,... à traiter). La définition d'une fonction de rappel d'oisiveté se fait grâce à :

```
void glutIdleFunc(void (*func)(void))
```

Ce décalage va se traduire à l'écran par un léger déplacement des cercles constituant la texture. La répétition de ce décalage chaque fois que le gestionnaire de déplacement n'a rien à faire va produire l'effet de mouvement sur la texture.

Eclairage des textures

L'éclairage des textures se fait de la même manière que pour les objets non texturés, nous y avons déjà consacré un didacticiel. La seule différence est que l'utilisation de textures a pour effet d'atténuer fortement l'éclairage spéculaire (les tâches lumineuses donnant un aspect brillant aux objets). Pour remédier à ce problème, OpenGL permet de séparer le calcul de l'éclairage spéculaire grâce à l'appel suivant :

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

Pour désactiver cette fonctionnalité, remplacez `GL_SEPARATE_SPECULAR_COLOR` par `GL_SINGLE_COLOR`. Dans le programme exemple, la touche 's' permet de basculer entre ces deux modes de calcul de l'éclairage.

Brouillard

Il est relativement facile de créer un effet de brouillard avec OpenGL. En effet, le rendu utilise la notion de profondeur (distance d'un objet à l'observateur) pour le rendu. En 'mixant' la couleur d'un objet avec une couleur 'du brouillard' en fonction de la distance (plus l'objet est éloigné, plus la couleur de fond est prépondérante), on crée un joli brouillard.

L'activation et la désactivation du brouillard se fait respectivement avec `glEnable(GL_FOG)` et `glDisable(GL_FOG)`. La modification des paramètres du brouillard se fait avec la fonction

`glFogf(GLenum nomparam, GLfloat valeur)`

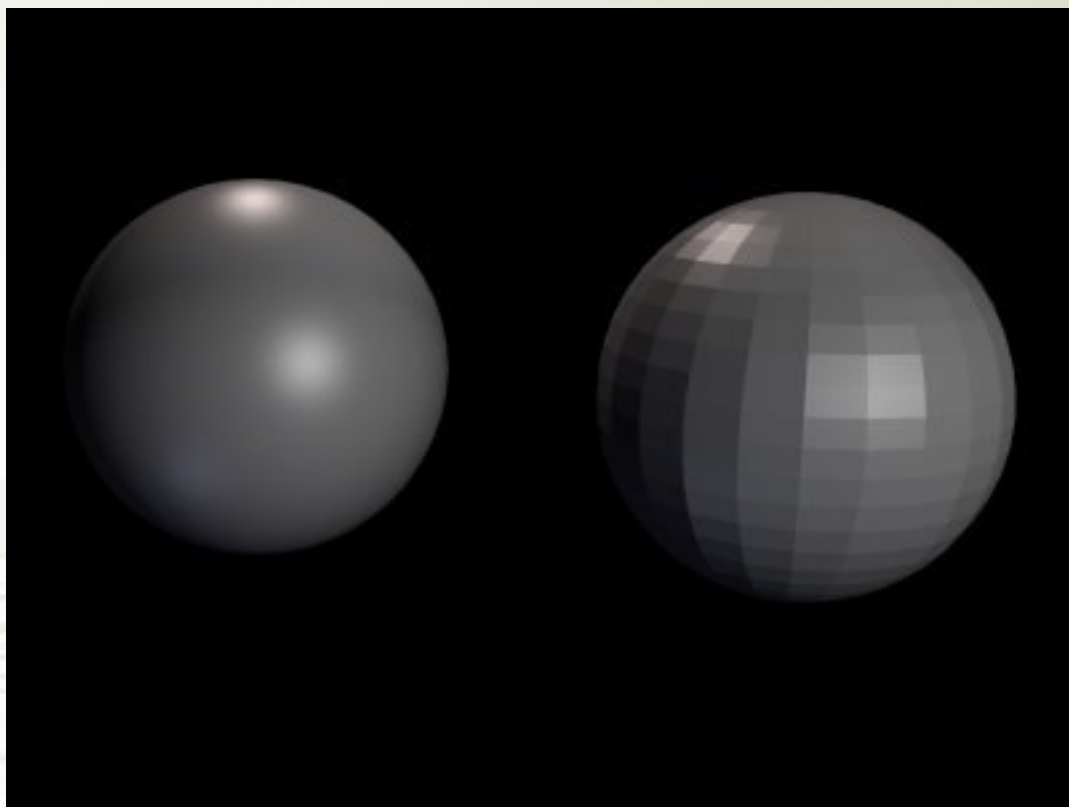
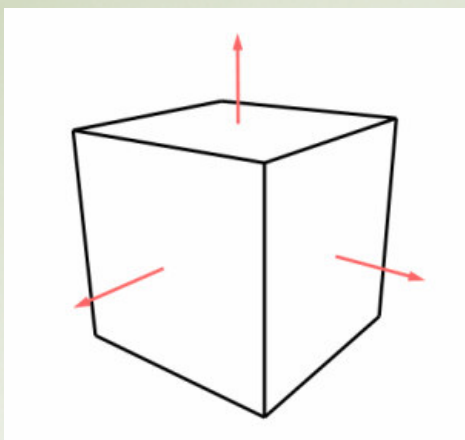
‘paramètre’ est le nom du paramètre à modifier, et ‘valeur’ est évidemment la nouvelle valeur du paramètre. Il existe plusieurs modes de brouillard, correspondant à différentes fonctions de mélange de la couleur de l’objet et de la couleur du brouillard. Je ne vais pas détailler tous les modes aujourd’hui (vous trouverez tout ça dans le livre [1]), je vous livre simplement les paramètres utilisés pour le programme exemple. Il s’agit d’un brouillard de type exponentiel :

```
glFogi(GL_FOG_MODE, GL_EXP); /* Réglage du type de brouillard */
glFogfv(GL_FOG_COLOR, couleurAP); /* Couleur du brouillard,
généralement la identique à la couleur de fond définie avec
glClearColor() */
glFogf(GL_FOG_START, 0); /* distance de début du brouillard
(relativement à l’observateur) */
glFogf(GL_FOG_END, 15); /* distance de fin du brouillard */
glFogf(GL_FOG_DENSITY, 0.35); /* Densité du brouillard */
```

Normales Normales

Nous avons déjà parlé des normales au cours du tutoriel concernant l'éclairage, mais nous avons en fait détourné le problème en utilisant comme objets 3D des primitives glut dont les normales étaient déjà mises en place. Aujourd'hui nous allons voir en détail comment définir une normale.

Les algorithmes d'éclairage ont besoin de connaître l'orientation des faces d'un objet. Pour cela, ils mettent en jeu la notion de normale, un vecteur perpendiculaire au polygone considéré. Bien sûr, en connaissant les points constituant un polygone, il est possible de calculer un vecteur normal à ce polygone, mais ce calcul est coûteux, et sauf convention implicite dans l'ordre des sommets, il ne permet pas de déterminer automatiquement les faces avant et arrière du polygone. C'est pourquoi il nous faut spécifier les vecteurs normaux.



En OpenGL, on n'associe pas les normales aux polygones mais aux sommets, car cela permet de créer des effets de lissage sur les surfaces courbes. La spécification des normales se fait comme d'habitude grâce à un système de normale active définie avec la fonction

```
glNormal3f(float x,float y,float z);
```

Dans notre cas, chacun des sommets d'une face se voit affecter la même normale, et donc la définition d'une face de notre cube se fait grâce à une portion de code du type :

```
glBegin(GL_POLYGON);  
glNormal3f(0.0,0.0,1.0);  
glTexCoord2f(0.0,1.0); glVertex3f(-0.5, 0.5, 0.5);  
glTexCoord2f(0.0,0.0); glVertex3f(-0.5,-0.5, 0.5);  
glTexCoord2f(1.0,0.0); glVertex3f( 0.5,-0.5, 0.5);  
glTexCoord2f(1.0,1.0); glVertex3f( 0.5, 0.5, 0.5);  
glEnd();
```