TP LO12

- Aide à l'utilisation d'OpenGL
- Aide à l'utilisation de GLUT
- Objectifs des TP
- Description du travail à faire

Ce document est destiné aux étudiants. Il apporte un support aux explications de l'assistant de TP. Les fonctions données dans ce document sont décrites précisément dans la documentation OpenGL. Les sujets de TP seront précisés à chaque séance par l'assistant.

Rappel : la présence aux séances de TP est obligatoire.

TP 1 : Introduction à OpenGL

1 OpenGL

OpenGL est une librairie graphique totalement portable offrant de nombreuses ressources aux programmeurs cherchant à élaborer un moteur 3D. OpenGL a été développé à partir de GL (graphic library pour Silicon Graphic).

OpenGL offre les mêmes fonctionnalités que GL, mais la gestion des fenêtres et de l'affichage propre au système d'exploitation est assurée par les fonctions définies dans la librairie GLUT.

2 GLUT (OpenGL Utility Toolkit)

Les programmes OpenGL sont indépendants du système de gestion des fenêtres. GLUT est une interface logicielle permettant de gérer le système de fenêtre.

GLUT peut être utilisé avec les systèmes de fenêtrages tels que Windows x/NT, OS/2, X window, Mac OS X

Structure principale d'un programme GLUT

Initialisation de GLUT	**glutInit**
Initialisation du mode d'affichage	**glutInitDisplayMode**
Création d'une ou plusieurs fenêtres	**glutInitWindowPosition** **glutInitWindowSize** **glutCreateWindow**
Définir la fonction de rappel d'affichage	**glutDisplayFunc**
Effectuer ses propres initialisations initialisation des lumières, du point de vue, de la perspective ……	**fonction Init( )**
Boucle principale	**glutMainLoop**

2.1. Initialisation

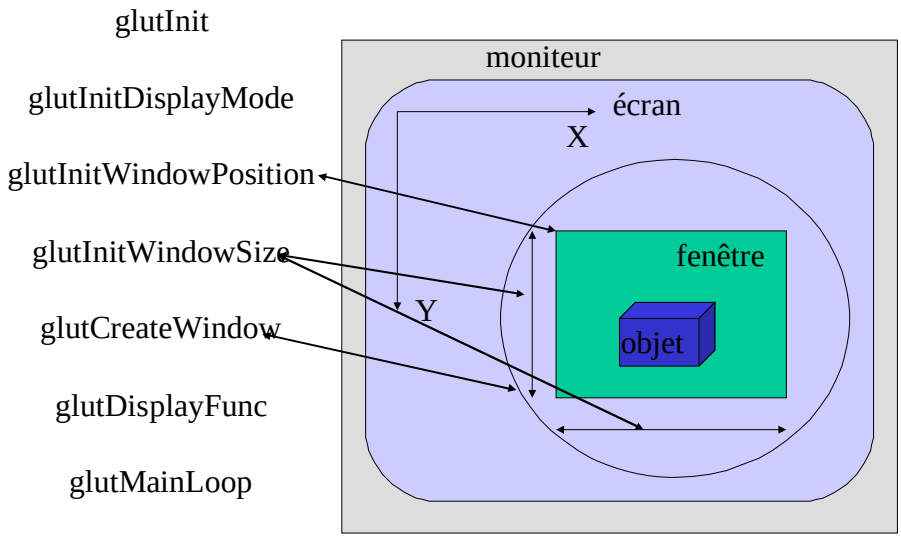
Initialisation et ouverture de session avec le système de fenêtre.

```
void glutInit( int *argcp, char **argv) ;
```

Initialisation des paramètres de la fenêtre

```
void glutInitWindowSize ( int largeur, int hauteur ) ;
void glutInitWindowPosition( int x, int y) ;
```

largeur, hauteur, x, y sont en pixels



Initialisation du mode d'affichage

```
void glutInitDisplayMode( unsigned int mode) ;
```

Cette fonction spécifie les mode d'affichage et gestion des buffers gérés par OpenGL de la fenêtre.

Les masques de modes pouvant être utilisés sont :

couleurs

GLUT_RGBA :	sélection d'un mode RGBA
GLUT_RGB :	mode RGB
GLUT_INDEX :	mode de couleur basé sur les index de couleurs.

buffers

GLUT_SINGLE :	un seul buffer d'affichage.
GLUT_DOUBLE :	double buffer d'affichage.
GLUT_ACCUM :	buffer d'accumulation.
GLUT_ALPHA :	buffer de transparence.
GLUT_DEPTH :	buffer de profondeur.

2.2. Début du traitement des événements

```
void glutMainLoop(void)
```

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

2.3. Gestion des fenêtres

GLUT supporte deux types de fenêtres : les fenêtres principales et les sous fenêtres.

- les fenêtres principales

```
void glutCreateWindow( char *nom) ;
```

Cette fonction crée une fenêtre principale. Implicitement, la fenêtre courante est la dernière fenêtre créée. Chaque fenêtre a un contexte OpenGL associé unique. Les changements d'états sur le contexte d'une fenêtre peuvent être faits juste après sa création, mais ils ne seront validés qu'au lancement de la fonction *glutMainLoop*.

- les sous fenêtres

```
void glutCreateSubWindow( int win, int x, int y, int largeur, int hauteur) ;
```

avec

win : identificateur de la fenêtre mère.
x : localisation du pixel en x relative à l'origine de la fenêtre mère.
y : localisation du pixel en y relative à l'origine de la fenêtre mère.

- Identification et accès

```
void glutSetWindow( int win) ;
```

choisir la fenêtre courante

```
int glutGetWindow( void) ;
```

donne l'identificateur de la fenêtre courante

- destruction d'une fenêtre

```
void glutDestroyWindow( int win) ;
```

- fonctions diverses

```
void glutPostRedisplay(void) ;
```

Marque la fenêtre courante qui doit être réaffichée. A la prochaine itération de la boucle principale de *glutMainLoop*, la fonction de rappel d'affichage est appelée et le plan normal est affiché, plusieurs appels n'engendrent qu'un seul rafraîchissement.

```
void glutSwapBuffer(void) ;
```

Cette fonction a pour but d'échanger la fenêtre active et la fenêtre de travail en mode double buffering.

```
void glutPositionWindow( int x, int y) ;
```

Changement de position pour la fenêtre courante.

```
void glutReshapeWindow( int largeur, int hauteur) ;
void glutFullScreen(void) ;
```

Changement de taille de la fenêtre.

```
void glutPopWindow(void) ;
```

Change l'ordre d'empilement de la fenêtre courante avec ses frères.

```
void glutPushWindow(void) ;
```

Change l'ordre d'empilement de la fenêtre courante avec ses frères.

```
void glutShowWindow(void) ;
void glutHideWindow(void) ;
void glutIconifyWindow(void) ;
void glutSetWindowTitle(char *name) ;
void glutSetIconTitle(char *name) ;
void glutSetCursor(int curseur) ;
```

Change l'image du curseur de la fenêtre courante :

GLUT_CURSOR_RIGHT_ARROW
GLUT_CURSOR_INFO
GLUT_CURSOR_DESTROY
GLUT_CURSOR_HELP
GLUT_CURSOR_CYCLE
GLUT_CURSOR_SPRAY
GLUT_CURSOR_WAIT
GLUT_CURSOR_TEXT
GLUT_CURSOR_CROSSHAIR
GLUT_CURSOR_UP_DOWN

2.4. Définitions de fonctions de type callback

GLUT possède plusieurs fonctions de rappel (callback). La fonction passée en paramètre est appelée automatiquement en fonction de l'événement détecté (modification de la taille, masquage de la fenêtre, déplacement de la souris, appui sur une touche…)

- les principales sont

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

Cette fonction appelle la fonction d'affichage à exécuter dans la fenêtre courante.

```
void glutReshapeFunc(void (*func) (int largeur, int hauteur)) ;
```

Cette fonction appelle la fonction de réorganisation à exécuter dans la fenêtre courante. *largeur* et *hauteur* contiennent la taille de la fenêtre courante.

Pour l'interaction clavier et souris (TP5) :

```
void glutKeyboardFunc ();
void glutMouseFunc ();
void glutMotionFunc ();
void glutPassiveMotionFunc ();
void glutVisibilityFunc ();
void glutSpecialFunc ();
```

2.5. Exemple d'ouverture de fenêtre

```
#include <stdio.h>
#include <stdlib.h>

#if defined(__APPLE__) && defined(__MACH__)
 #include <GLUT/glut.h>
 #include <OpenGL/gl.h>
 #include <OpenGL/glu.h>
#else
 #include <GL/glut.h>
 #include <GL/gl.h>
 #include <GL/glu.h>
#endif

void dessine_scene(void)
{…}

void init(void)
{… }

void main(int argc, char**argv)
{
   glutInit(&argc, argv);
   glutInitWindowSize(300, 300);    /* taille de la fenetre */

   glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
                                   /* modes de gestion des buffers */
   glutCreateWindow("TP1 : cube");

   glutDisplayFunc(dessine_scene);
   /* définition de la fonction d'affichage */

   init();
   glutMainLoop();
}
```

TP1

1. Introduction aux scènes utilisées en TP.
   - description statique de la scène avec syntaxe précise dans le fichier de données, ex *cube1.dat*
   - description de la scène en mémoire: voir la structure `scene` décrite dans le fichier *scene.h*

2. Lecture du fichier de données et stockage dans la structure : *lecture_fich.c*
- Affichage de la scène : utiliser la fonction *dessine_scene()*définie dans le fichier *affiche.c*

3. Création d'un projet avec Visual C++

Travail de l'étudiant :
- Ecrire un fichier de données avec 2 objets. La taille des objets est de l'ordre de 3 et leur position dans la scène est autour du centre du repère (éloignement de 10 max). Compiler le code donné dans TP1 et visualiser la scène.
- Afin de visualiser l'ordre d'affichage des facettes, vous pouvez insérer judicieusement une tempo (que vous enlèverez par la suite).
- Modifier éventuellement les paramètres des fonctions `glut` utilisées.

TP 2 : Affichage avec les fonctions OpenGL

1. descriptions des points, des lignes et des polygones

OpenGL travaille avec des coordonnées homogènes. Ainsi pour les calculs internes, chaque sommet est représenté à l'aide de 2 (*x*,*y*) 3 (*x*,*y*,*z*) ou 4 coordonnées (*x*,*y*,*z*,*w*). Si *w* est différent de 0 alors ces coordonnées correspondent à un point en 3D dans un repère euclidien (*x/w*, *y/w* ,*z/w*). Si *w* n'est pas spécifié alors il est considéré comme étant égal à 1. Si *w=0* alors le point est considéré comme étant à l'infini dans la direction donnée par le vecteur (*x*,*y*,*z*). Les coordonnées sont des entiers (s ou i), ou des flottants (f ou d).

Avec OpenGL, tous les objets géométriques sont décrits comme un ensemble de sommets ordonnés. Afin de spécifier un sommet, on utilise la commande :

```
void glVertex{234}{sifd}[v](TYPE coords) ;
```

L'appel d'un *glvertex* se fait toujours entre un *glBegin* et un *glEnd*.

```
Void glBegin(Glenum mode) ;
```

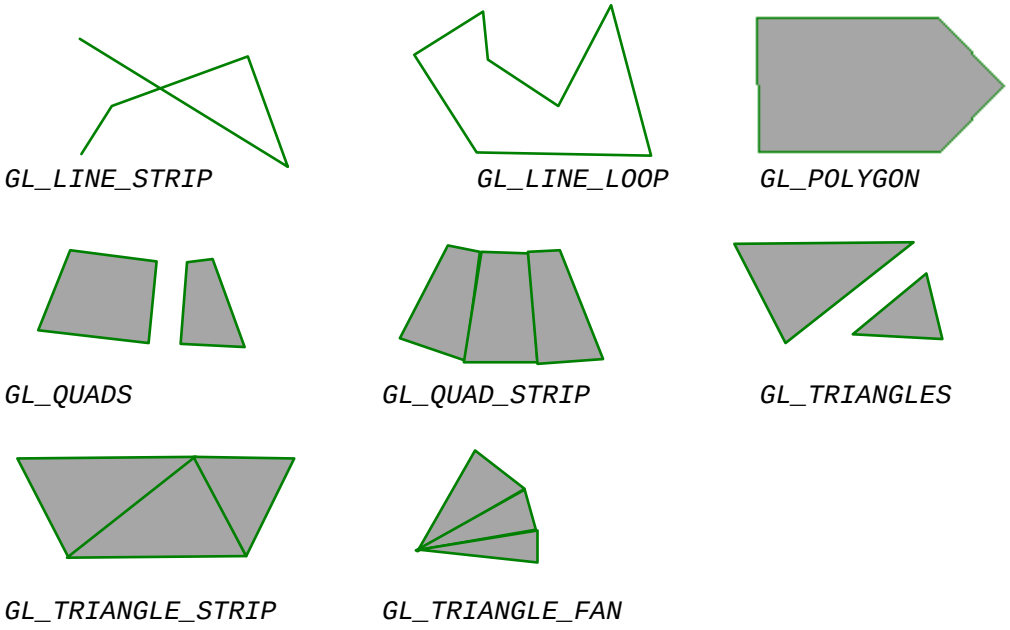
Les modes disponibles sont :

GL_POINTS	points individuels
GL_LINES	segment de ligne individuel (deux sommets)
GL_POLYGON	polygone simple convexe
GL_TRIANGLES	triangle (trois sommets)
GL_QUADS	polygone a quatre cotés (carré, rectangle, trapèze)
GL_LINE_STRIP	suite de segments connectés
GL_LINE_LOOP	comme précédemment mais avec le premier et le dernier sommet joint
GL_TRIANGLE_STRIP	triangles joints par un de leurs cotés
GL_TRIANGLE_FAN	éventail de triangles
GL_QUAD_STRIP	quadrilatères joints par un de leurs cotés

Exemple :

définition d'un polygone par définition de ses sommets en coordonnées réelles 2D:

```
glBegin (GL_POLYGON) ;
    glVertex2f(0.0, 0.0) ;
    glVertex2f(0.0, 3.0) ;
    glVertex2f(3.0, 3.0) ;
    glVertex2f(4.0, 1.5) ;
    glVertex2f(3.0, 0.0) ;
glEnd() ;
```



Pour chaque sommet, on peut ajouter des données spécifiques à l'aide des commandes suivantes :

`glVertex*()`	définir les coordonnées d'un sommet
`glColor*()`	définir la couleur courante(mode RGB)
`glIndex*()`	définir l'index de la couleur courante (mode index)
`glNormal*()`	définir les coordonnées de la normale
`glEvalCoord*()`	créer des coordonnées
`glCallList(), glCallLists()`	exécuter la liste(s) de display
`glTexCoord*()`	définir les paramètres de la texture
`glEdgeFlag*()`	contrôler l'affichage des arêtes
`glMaterial*()`	définir les propriétés d'une matière

2. Descriptions des couleurs

- couleur de fond

Sur un ordinateur, la mémoire garde le dessin chargé lors du dernier affichage. Il faut donc effacer cet affichage si on veut afficher un nouveau dessin. Pour cela, on remplit la fenêtre courante avec une couleur de fond. Ceci se fait à l'aide des commandes *glClearIndex( )*(mode INDEX) ou *glClearColor* (mode RGB) et *glClear()* .

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

Définit la couleur courante de fond d'écran (en mode RGB)

```
void glClearIndex(float);
```

Définit la couleur courante de fond d'écran (en mode INDEX)

```
void glClear(Glbitfield masque);
```

Applique la valeur définie par *glClearColor()* ou *glClearIndex()* à un ou plusieurs des buffers spécifiés. Ex : applique la couleur de fond à toute la fenêtre courante (mode RGB).

exemple :

```
glClearColor(0.0, 0.0, 0.0, 0.0, 0.0) ;
glClear(GL_COLOR_BUFFER_BIT) ;
```

la première commande définit la couleur de fond à noir et la seconde commande efface toute la fenêtre avec la couleur de fond.

glClear peut être utilisé pour effacer un ensemble de buffer dont :

GL_COLOR_BUFFER_BIT	buffer de couleur
GL_DEPTH_BUFFER_BIT	buffer de profondeur (z buffer)
GL_ACCUM_BUFFER_BIT	buffer d'accumulation
GL_STENCIL_BUFFER_BIT	buffer du stencil

- fixer une couleur

La description de la forme d'un objet à dessiner est indépendante de la description de ses couleurs. Pour définir une couleur, la commande est :

void glColor3() ;* (mode RGB)
void glIndex() ;* (mode INDEX)

3. L'affichage

Pour forcer l'affichage de tous les objets et de tous leurs paramètres définis jusqu'à maintenant, on utilise la commande *glFlush()*. Cette fonction sera utile lorsque nous utiliserons plusieurs buffers.

```
void glFlush(void) ;
```

TP2

1) Utilisation des couleurs

Pour ce TP, nous utiliserons le mode filaire (*GL_LINE_LOOP* pour la fonction *gl_begin()* dans *affiche.c*)

1– modifier la couleur de fond

2– donner une couleur aux 2 objets.

3– Donner une couleur différente à chaque face du cube.

4– Donner une couleur différente à chaque sommet (vertex) du cube.

5– Afficher le repère avec une couleur différente par axe.

(ne pas toucher au fichier de description de la scène, il faut créer une fonction *dessine_repere()*)

2) Définition d'une fonction « reshape »

1- Commentaires sur les fonctions *gluLookAt()* et *gluPerspective()* (à faire avec l'assistant de TP).
   Mettre la fonction *gluLookAt()* en commentaires ; que se passe-t-il ?
   Modifier la valeur du rapport (2eme paramètre) de la fonction *gluPerspective()*

2- Modifier la taille de la fenêtre à l'écran (avec la souris). Si l'affichage est incorrect proposer une fonction callback « reshape »

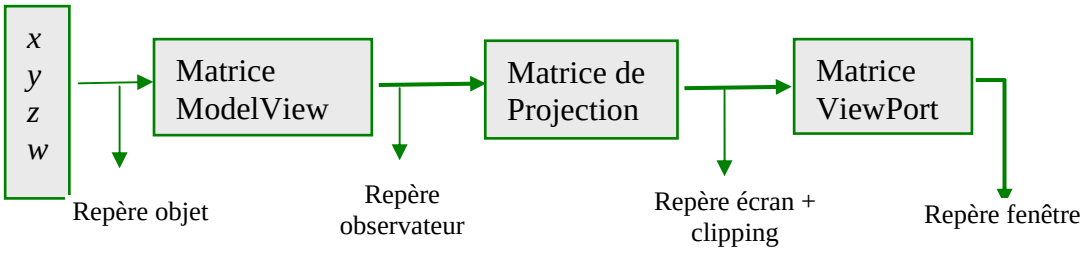
TP3 : Les transformations

1. Introduction

Cette partie va permettre de :

- Manipuler les piles de matrices qui contrôlent les transformations appliquées à la scène.
- Simuler des déplacements d'objets.

La chaîne de traitement des coordonnées d'un objet peut être représentée par :



Etapes de transformation d'un sommet

2. Les transformations

Les structures maintenues par OpenGL pour gérer les repères et les transformations sont :

- La matrice **ModelView** Matrice de transformation de l'objet vers l'observateur
- La matrice **Projection** Matrice de transformation de l'observateur vers le plan image
- La matrice **Texture** Matrice de transformation de texture des coordonnées objet vers le plan image

Pour spécifier le type de la matrice utilisée (ModelView, Projection ou Texture) dans un programme écrit sous OpenGL, on utilise la commande :

```
void glMatrixMode(Glenum mode);
```

Indique sur quelle matrice vont être effectuées les transformations.

Il existe trois modes pour cette commande:

GL_MODELVIEW	(pile de 32 matrices 4x4)
GL_PROJECTION	(pile de 2 matrices 4x4)
GL_TEXTURE	

La manipulation de ces matrices est faite à l'aide de l'ensemble des fonctions suivantes :

```
void glLoadIdentity(void);
```

Place dans la matrice courante une matrice identité 4x4.

```
void glLoadMatrix{fd}(const TYPE *M);
```

Place dans la matrice courante les 16 valeurs de M.

```
void glMultMatrix{fd}(const TYPE *M);
```

Multiplie la matrice courante par M et stocke le résultat dans la matrice courante.

L'argument des fonctions *glLoadMatrix()* et *glMultMatrix()* est un vecteur de 16 valeurs (m_0, m_1,…m_{15}) représentant la matrice suivante.

$$M = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

OpenGL met également à disposition des fonctions de transformations qui sont :

```
void glTranslate{fd}(TYPEx, TYPEy, TYPEz) ;
```

La matrice courante **C** est remplacée par **C** * **T** avec **T** = $\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
void glRotate{fd}(TYPEangle, TYPEx, TYPEy, TYPEz) ;
```

Cette fonction calcule une matrice **R** de rotation de *angle* degrés autour du vecteur $\overrightarrow{OP}$, $O(0,0,0)$ $P(x,y,z)$

La matrice courante **C** est remplacée par **M** * **R**

```
void glScale{fd}(TYPEx, TYPEy, TYPEz) ;
```

La matrice courante *M* est remplacée par **C** * **S** avec **S** = $\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

3 Manipulation de la pile des matrices de transformation

Pour la manipulation de la pile de matrice de transformation, nous avons à disposition trois fonctions principales :

```
glGetDoublev(GL_MODELVIEW_MATRIX, Mat);
```

Avec: *GLdouble Mat [16] ;*

Cette fonction permet de récupérer les valeurs de la matrice courante dans un vecteur.

```
void glPushMatrix(void) ;
```

Cette fonction duplique la matrice courante dans la matrice de tête.

```
void glPopMatrix(void) ;
```

Cette fonction permet de dépiler la dernière matrice.

Comme nous l'avons vu auparavant, une scène est définie par un ensemble d'objets et chaque objet par des faces et chaque face par des sommets (vertex). A chaque sommet est associé un certain nombre de caractéristiques et de paramètres tels que ses coordonnées et sa couleur. Lors de l'affichage, la transformation contenue dans la matrice de transformation courante sera appliquée à chacun des sommets. Si 2 objets ne doivent pas subir les mêmes transformations alors il faut manipuler la matrice concernée (ModelView, Projection, …) en utilisant si possible la pile associée.

Soit un sommet **P**, **P'** le transformé de **P** par la transformation **ModelView** sera égal à

P' = ModelView * P

Ainsi, tous les objets de la scène subiront toutes les transformations faites depuis le début du programme (voir fig 1).

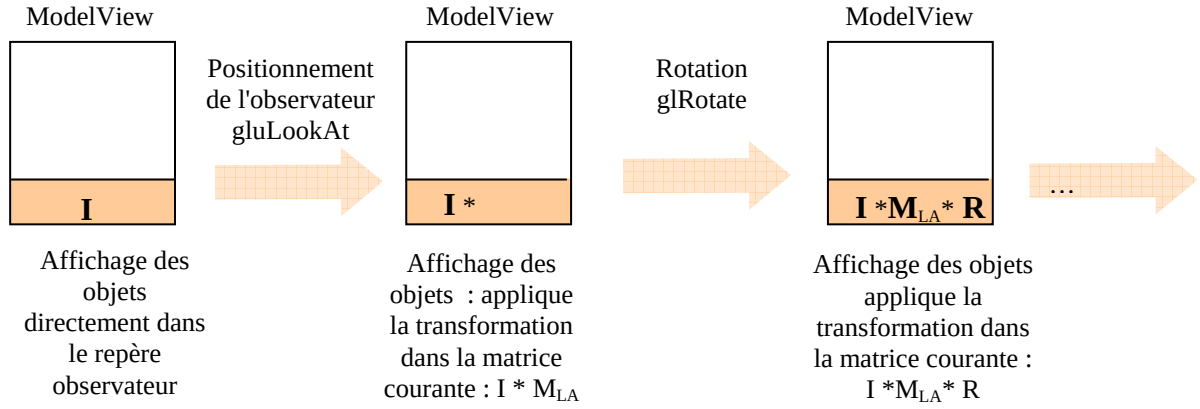


Figure 1 : Transformations successives sans gestion de la pile

Par contre si avant la déclaration des sommets d'un objet on empile une nouvelle matrice alors toutes les transformations faites à partir de ce moment seront appliquées jusqu'à ce que la matrice soit dépilée. Une fois la manipulation de l'objet finie, on peut dépiler la matrice de transformation courante pour se retrouver dans l'état initial (voir fig 2).

Le positionnement de l'observateur (par exemple la fonction `gluLookAt()`) se traduit par une modification de la matrice courante **ModelView** avec les valeurs de la transformation du repère du monde vers le repère observateur.

Remarques très importantes :

La multiplication des matrices se faisant à droite il faut faire attention à l'enchaînement des transformations. Par exemple, pour réaliser la transformation composée d'une rotation puis une translation sur **P**, soit **P'** = **T**.**R**.**P**, il faut réaliser les appels suivants :

```
glTranslate*(…);
glRotate*(…);
afficheP(…)
```

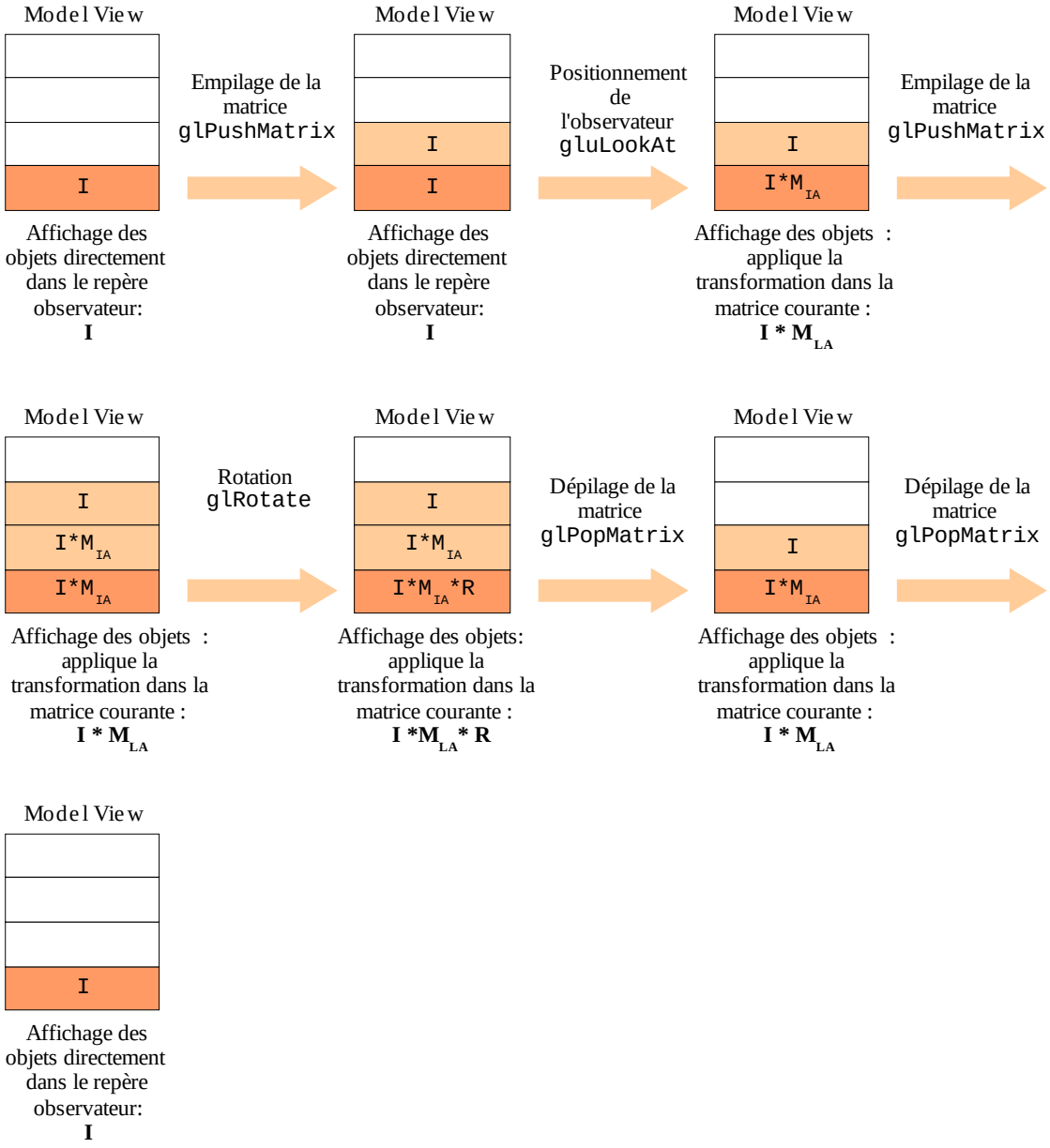


Figure 2 : Gestion de la pile pour les transformations

4. Balayage Écran et Double Buffering

Afin d'éviter l'effet de clignotement dû à la désynchronisation entre le calul de l'image et le balayage de l'écran, on peut utiliser le mode double buffering (le buffer de travail est différent de celui qui est affiché). Dans la définition des modes d'affichage, ajouter GLUT_DOUBLE et à la place de la fonction d'affichage (*glFlush*) mettre la fonction *glutSwapBuffers( )* qui échange la fenêtre active et la fenêtre de travail.

Utiliser également la fonction *glutPostRedisplay()* qui indique que la fenêtre courante doit être rafraîchie.

Pour avoir un mouvement en continu, il est également nécessaire d'utiliser une fonction de rappel de fonction d'affichage en mode repos. En effet pour l'instant le programme appelle une et une seule fois la fonction d'affichage de la scène. Le seul moyen de rafraîchir cette fenêtre est donc d'appliquer d'éventuelles modifications ou de générer un événement (par exemple en bougeant la fenêtre).

La fonction de rappel pour le mode repos est :

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

TP 3

.

1. Faire tourner la scène (2 objets) autour d'un des axes de la scène.

   Par la suite, utiliser la structure SCENE (et plus particulièrement le champ transfo) pour stocker les transformations appliquées aux objets.
2. Déplacer le cube en $\begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$.
   2.1. Faire tourner le cube autour de son axe y.
   2.2. Faire tourner le cube autour l'axe y de la scène.
   2.3. Faire tourner le cube autour de son axe y et l'autre objet autour de son axe z (différent de celui de la scène)
3. Faire tourner en sens inverse 2 cubes de couleurs différentes et mis à des positions différentes.
4. Activer le mode double-buffering et comparer avec le mode simple

Remarque : les questions 1, 2 et 3 peuvent se faire en modifiant directement la fonction *dessine_scene()*.

TP 4: Les points de vues et projections

1. L'observateur

Nous allons décrire dans cette partie les transformations pour placer l'observateur dans la scène.

1.1 Le LookAt

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centrex,
GLdouble centrey, GLdouble centrez, GLdouble upx, GLdouble upy, GLdouble upz)
```

Cette commande permet définir la façon dont l'observateur se place par rapport à la scène.

eyex, eyey et *eyez* spécifie le point de vue de l'observateur (la position de l'observateur)

centrex, centrey et *centrez* donne un point sur la ligne de vue de l'observateur. C'est le point vers lequel l'observateur regarde (vecteur $-z_0$).

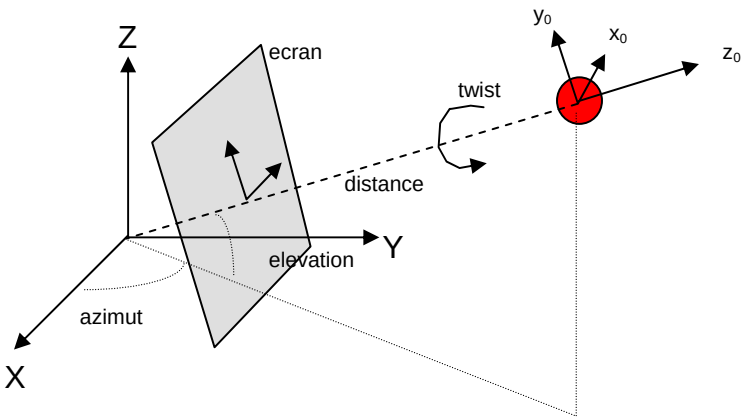
upx, upy et upz décrit le vecteur y_0 (twist).

1.2 Le PolarView

Le polarView n'existe pas en OpenGL. On peut le simuler par la composition des transformations vues en cours (attention à la translation qui doit être effectuée à la fin sur l'axe z du repère résultant des différentes rotations).

```
void polarView(GLdouble distance, GLdouble azimut, GLdouble elevation, GLdouble
twist)
{
// sujet du TP4
}
```

distance est la distance à l'origine.
azimut décrit l'angle de position de l'observateur par rapport au plan 0xz
elevation est l'angle d'élévation de l'observateur par rapport au plan 0xy.



twist représente la rotation du volume de vue autour de sa ligne de vue.

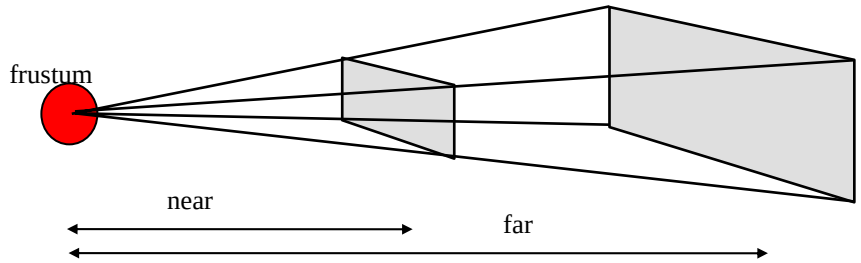
1.3 Le PilotView

Cette fonction reproduit la vue que pourrait avoir un pilote sur le monde de son avion. Elle utilise les différents angles de rotations de l'avion par rapport à son centre de gravité.

```
void pilotView(GLdouble planex, GLdouble planey, GLdouble planez, GLdouble
roulis, GLdouble tangage, GLdouble lacet)
   {
   // sujet du TP4
   }
```

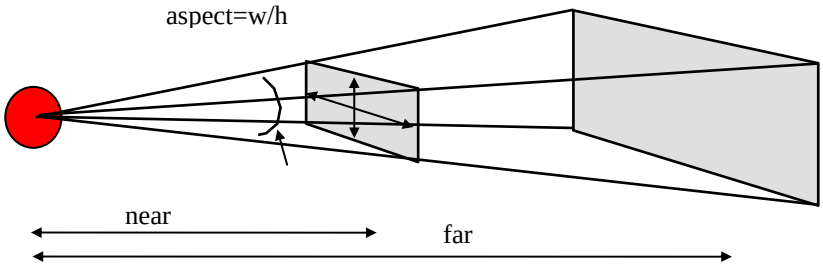
2. Les projections

2.1. Les perspectives




```
void glFrustum( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far) ;
```

near et *far* doivent toujours avoir des valeurs positives.



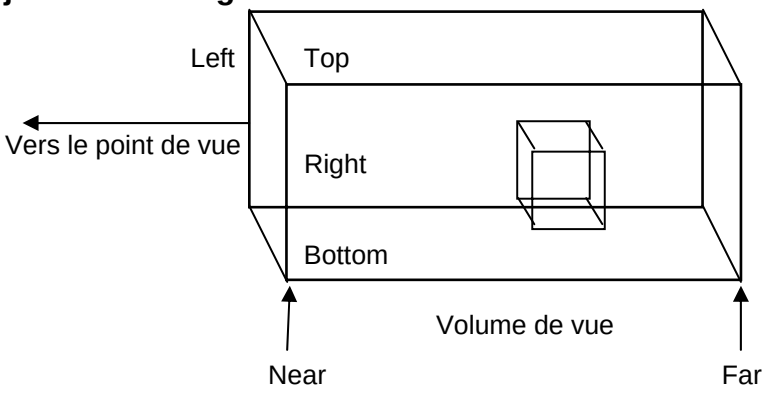

```
void gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble znear, GLdouble
zfar) ;
```

fovy est l'angle du champ de vue dans le plan 0*xy*. Sa valeur doit être comprise entre [0.0,180.0].

aspect est le rapport entre la largeur et la hauteur de la pyramide tronquée qui sert de fenêtre de vue pour la projection en perspective. Ce volume de clipping est également appelé un frustum en OpenGL.

znear et *zfar* sont les distances entre le point de vue et les plans de clipping avant et arrière, elles doivent toujours être positives.

2.2 Les projections orthogonales




```
void glOrtho( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far) ;
```

Cette fonction permet de créer un volume de fenêtrage parallélépipédique avec une projection parallèle orthogonale. Pour définir un plan, on a la fonction en 2D :

```
void glOrtho2D( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top) ;
```

2.3 Réduction de la région à afficher

glViewport permet d'établir le lien entre la fenêtre d'affichage et la fenêtre du volume de vision.

```
void glViewport (GLint x, GLint y, GLsizei largeur, GLsizei hauteur) ;
void glDepthRange( GLclampd near, GLclampd far) ;
```

Cette commande modifie les coordonnées en z durant la transformation du Viewport.

Le *near* et le *far* représentent les ajustements des valeurs mini et maxi pouvant être stockées dans le z-buffer (utilisé dans le TP suivant). Par défaut, ils sont à 0.0 et 1.0.

Remarques importantes :

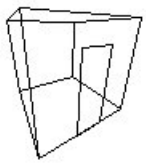
OpenGL n'utilise qu'un seul repère (celui de l'observateur). Toutes les transformations sur la scène et toutes les transformations liées au positionnement de l'observateur sont composées et stockées dans la matrice MODELVIEW vue au TP3.

Sans instruction de positionnement de l'observateur (et sans déplacement du cube) l'observateur est au centre du cube et regarde vers les z négatifs.

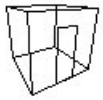
TP4

1. Utiliser la fonction *gluLookAt()* pour positionner l'observateur et essayer les différentes projections et fenêtrages associés.
2. Programmer et essayer les points de vue *polarView()* et *pilotView()*
3. Retrouver les figures suivantes avec le *polarView()*.
4. Programmer 2 points de vues différents et de passer de l'un à l'autre à chaque fois que l'on change la taille de la fenêtre par exemple. (les interactions sont vues dans le TP5)

Images du cube avec des valeurs de *distance*, *azimut*, *elevation* et *fovy* suivantes (*twist* = 0)

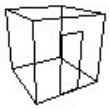


2,5 -30 20 114

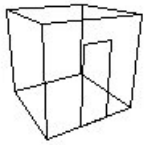


3,5 -30 20 114

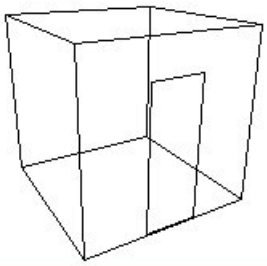
4,5 -30 20 114



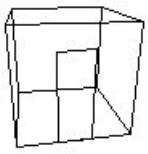
4,5 -30 20 114



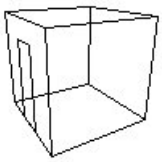
4,5 -30 20 76



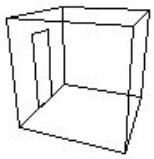
4,5 -30 20 45



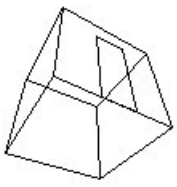
4,5 -10 20 70



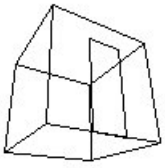
4,5 60 20 70



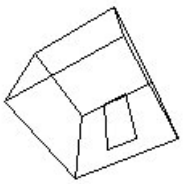
4,5 60 20 70



4,5 -30 -45 70



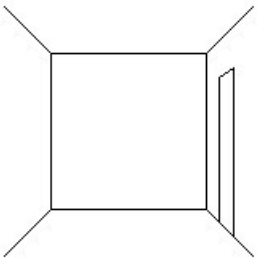
4,5 -30 -20 70



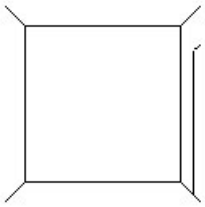
4,5 -30 120 70

Effets du fenêtrage des plans avant et arrière

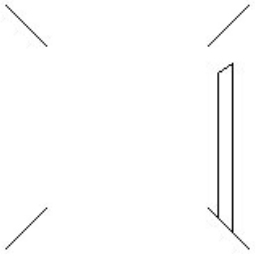
distance = 4, *azimut* = -90, *élévation* = 0, *twist* = 0 *fovy* = 45



avant = 3.1 arrière = 10

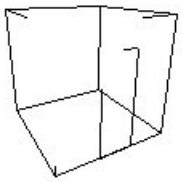


avant = 4 arrière = 10

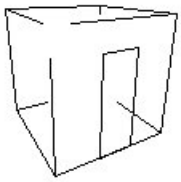


avant = 3.2 arrière = 4.8

distance = 4, *azimut* = -30, *élévation* = 20, *twist* = 0 *fovy* = 70



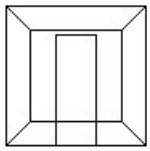
avant = 3.1 arrière = 10

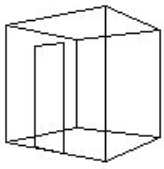


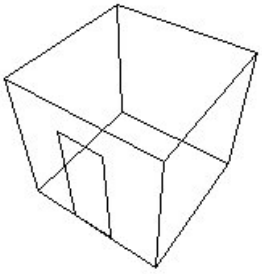
avant = 2.5 arrière = 5

avant = 0.1 arrière = 3

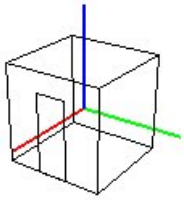
points de fuite



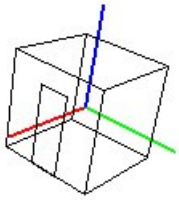




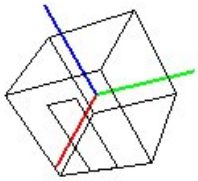
maintien de la tête : le twist (angle entre l'axe y0 de l'observateur avec le z initial)



twist = 0



twist = 10



twist = -30

TP 5 : Interaction et animation

Ce TP a pour objet de combiner les fonctions vues dans les TP précédents en animant la scène à partir du clavier et de la souris.

1. Gestion des événements : le clavier et la souris

Les fonctions permettant la gestion des événements sont les suivantes:

```
void glutKeyboardFunc( void (*func) (unsigned char key, int x, int y)) ;
```

Appel de la fonction de gestion des touches.

`key` contient le code de la touche.

`x` et `y` contiennent la position en pixel de la souris lors de l'appui sur une touche.

```
void glutSpecialFunc( void (*func) (int key, int x, int y)) ;
```

Appel de la fonction de gestion des touches spéciales.

key contient le code de la touche spéciale. Ceux-ci sont donnés ci-dessous.

x et *y* contiennent la position en pixel de la souris lors de l'appui sur une touche.

```
GLUT_KEY_F1
.....
GLUT_KEY_F12
GLUT_KEY_LEFT
GLUT_KEY_RIGHT
GLUT_KEY_UP
GLUT_KEY_DOWN
GLUT_KEY_PAGE_UP
GLUT_KEY_PAGE_DOWN
GLUT_KEY_HOME
GLUT_KEY_END
GLUT_KEY_INSERT
```

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

Appel de la fonction de gestion de la souris lorsqu'un bouton est activé.

`bouton` contient le code de la touche appuyée

```
GLUT_LEFT_BUTTON
GLUT_MIDDLE_BUTTON
GLUT_RIGHT_ BUTTON
```

`état` indique si le bouton est appuyé ou relâché:

```
GLUT_DOWN
GLUT_UP
```

`x` et `y` contiennent la position en pixel de la souris lors de l'appui sur une touche.

```
void glutMotionFunc( void (*func) ( int x, int y)) ;
```

Appel de la fonction de gestion de déplacement de la souris, cette fonction est activée si la souris se déplace et qu'un bouton est activé.

x et *y* contiennent la position en pixel de la souris lors de l'appui sur une touche.

```
void glutPassiveMotionFunc( void (*func) ( int x, int y)) ;
```

Appel de la fonction de gestion de déplacement de la souris, cette fonction est activée si la souris se déplace et qu'aucun bouton n'est activé.

x et *y* contiennent la position en pixel de la souris lors de l'appui sur une touche.

Ces fonctions sont à appeler à l'initialisation avant la fonction *glutMainloop()*.

2. Quelques fonctions utiles

```
int glutGetModifiers(void) ;
```

Retourne l'état de certaines touches qui ont pu être activée lors de l'appel des fonctions de gestion d'événements

GLUT_ACTIVE_SHIFT
GLUT_ ACTIVE _CTRL
GLUT_ ACTIVE _ALT

```
void glutPostRedisplay(void) ;
```

Marque la fenêtre courante qui doit être réaffichée. A la prochaine itération de la boucle principale de *glutMainLoop*, la fonction de rappel d'affichage est appelée et le plan normal est affiché, plusieurs appels n'engendrent qu'un seul rafraîchissement.

3. Exemples d'utilisation pour le TP.

1 Arrêt du programme par l'appui sur la touche ESC :

```
/* définition de la fonction activée par glutKeybordFunc */
void touche_clavier(unsigned char key, int x, int y)
{
   switch (key) {
       case 27:
       exit(0);
   }
}
```

2 Gestion des touches spéciales : déplacement d'un objet

```
/* définition de la fonction activée par glutSpecialFunc */
void touche_special_appuye(int key, int x, int y)
{
   int sens; /* sens positif ou négatif */
   if ( glutGetModifiers() == GLUT_ACTIVE_SHIFT )
       sens=-1;
   else
       sens=1;
   switch (key) {
       case GLUT_KEY_F1:/* avance selon direction axe x */
             move_obj_i(…);
             break;
       case GLUT_KEY_F2:/* avance selon direction axe y */
             move_obj_i(…);
             break;
       …
   }
}
```

Gestion de la souris : déplacement de l'observateur

```
/* définition de la fonction activée par glutMouseFunc */
void click_souris(int button, int state, int x, int y)
{
 …
 if (button== GLUT_LEFT_BUTTON || button== GLUT_RIGHT_BUTTON)
 {
 if (state==GLUT_DOWN)
 {
 /* memorisation de la position
 et choix du type de deplacement selon le bouton*/
 }
 else /*bouton releve*/
 {
 /* annuler le déplacement  */
 }
}

/* definition de la fonction activee par glutMotionFunc */
void click_dep_souris(int x, int y)
{
/* calcul du déplacement de la souris et
 définition des paramètres de la fonction qui déplace l'observateur
 en fonction du type de déplacement
 appel de la fonction move_obs() */
…
}
```

Dans le main()

```
void main(…)
{
   …
   glutSpecialFunc(touche_special_appuye);
   glutKeyboardFunc(touche_clavier);
   glutMouseFunc(click_souris);
   glutMotionFunc(click_dep_souris);
   …
}
```

TP5

Nous proposons dans ce TP de modifier de façon interactive soit la position (orientation) d'un objet, soit la position de l'observateur. Pour valider le déplacement d'un seul objet, nous conseillons d'utiliser une scène avec 2 objets.

Déplacement d'un objet (clavier uniquement) :

Chacune de ces touches combinée avec `shift` produit un mouvement inverse
`F1:` avance l'objet selon direction axe x
`F2:` avance l'objet selon direction axe y
`F3:` avance l'objet selon direction axe z
`F4:` tourne l'objet selon axe x
`F5:` tourne l'objet selon axe y
`F6:` tourne l'objet selon axe z

Déplacement de l'observateur (clavier et souris) :

- Avec le clavier :

`F10:` l'observateur tourne autour de l'axe z de la scène
`LEFT::` observateur tourne la tête à droite
`RIGHT:` ou à gauche
`UP:` observateur lève la tête vers le haut
`DOWN:` ou baisse la tête

- Avec la souris :

`clic gauche:` haut-bas: lever, baisser la tête
gauche-droite: tourner la tête à gauche, à droite
`clic droit:` haut-bas: monter-descendre, parallèlement à l'axe x (axe x vers le haut, y à gauche, z vers l'arrière)
gauche-droite: aller à gauche, à droite, parallèlement à l'axe y
`clic milieu:` haut-bas: se rapprocher, s'éloigner (effet de zoom)
gauche-droite: tourner autour de l'axe z de la scène

Conseils de réalisation:
Définir 2 fonctions qui appliquent les déplacements : une pour l'objet, l'autre pour l'observateur :

move_obj()
move_obs()

La fonction *move_obj()* va modifier directement la matrice de transformation de l'objet stockée dans la structure SCENE pour l'objet considéré. Dans ce cas il faut vérifier que la fonction *dessine_objet()* charge la transformation de l'objet avant son affichage

La fonction *move_obs()* va agir directement (et judicieusement!) sur la matrice MODELVIEW. Revoir les fonctions de récupération, multiplication … de cette matrice dans le TP3.

Améliorations possibles:
Ajouter dans la fonction *touche_clavier()* des options permettant de choisir l'objet à déplacer et de revenir à la position initiale par exemple.

TP 6 : Remplissage et élimination des parties cachées

Jusqu'à maintenant, la scène a été affichée en filaire. Ce TP à pour objet d'appliquer les algorithmes de remplissage de OpenGL et d'appliquer la méthode du Z-buffer pour la gestion des parties cachées.

1. Remplissage des polygones

Il suffit de modifier le mode d'affichage des polygones dans `glBegin()`.

2. Elimination des parties cachées.

L'activation et l'utilisation du buffer de profondeur se fait en rajoutant le mode `GLUT_DEPTH` à l'initialisation ainsi qu'en ajoutant le type de test associé au z-buffer *glEnable(GL_DEPTH_TEST).*

```
void glutInitDisplayMode( unsigned int mode) ;
```

Rappel des modes concernant le buffering :

`GLUT_SINGLE :`	fenêtre avec simple buffering.
`GLUT_DOUBLE :`	fenêtre avec double buffering.
`GLUT_DEPTH :`	fenêtre avec un buffer de profondeur.

L'activation et l'utilisation du buffer de profondeur se fait en rajoutant le mode `GLUT_DEPTH.`

3 Parties cachées et Animation.

Entre 2 affichages il faut re-initialiser le z-buffer avec la valeur de z_{min} (plus eloigné). Pour cela on ajoute l'option `GL_DEPTH_BUFFER_BIT`) à la fonction `glClear()`.
Afin d'éviter l'effet de clignotement dû au remplissage du frame buffer combiné au Z-buffer, on peut utiliser le mode double buffering (le buffer de travail est différent de celui qui est affiché). Dans la définition des modes d'affichage, ajouter `GLUT_DOUBLE` et à la place de la fonction d'affichage (`glFlush`) mettre la fonction `glutSwapBuffers( )` qui échange la fenêtre active et la fenêtre de travail.

Utiliser également la fonction `glutPostRedisplay()` qui indique que la fenêtre courante doit être rafraîchie.

Pour avoir un mouvement en continu, il est également nécessaire d'utiliser une fonction de rappel de fonction d'affichage en mode repos. En effet pour l'instant le programme appelle une et une seule fois la fonction d'affichage de la scène. Le seul moyen de rafraîchir cette fenêtre est donc d'appliquer d'éventuelles modifications ou de générer un événement (par exemple en bougeant la fenêtre).

La fonction de rappel pour le mode repos est :

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

TP 6

1. Afficher le cube (face pleine) en bleu et sa porte en jaune.
2. Se déplacer pour vérifier que les toutes faces sont correctes et constater le manque de gestion des parties cachées.
3. Activer le z-buffer et observer la cohérence de la scène.
4. Modifier la scène de manière à ce qu'il n'y ait pas d'effet de bord dû à la porte.
5. Animer la scène et activer le double-buffering pour éviter les effets de scintillement.

TP 7 : Eclairage

OpenGL propose 2 rendus : le mode flat (*GL_FLAT*) ou le rendu de Gouraud (*GL_SMOOTH*) : les calculs d'éclairage se font à chaque sommet et l'éclairage pour un pixel est calculé par interpolation. Cette option se détermine par la fonction *glShadeModel()*.

En OpenGL pour calculer une couleur en 1 point, il faut définir la matière et ses propriétés de réflexion de la lumière, le modèle (propriétés liées à la scène) et la définition de chaque source lumineuse. Ces définitions se font à l'aide des fonctions *glLight()*, *glLightModel()* et *glMateria()l*. Pour activer la plupart de ces définitions on utilise la fonction *glEnable()*.

3 fonctions de définitions ont été développées à partir de: *def_matieres()* *def_modele()* et *def_sources()*. Elles sont disponibles dans le fichier *eclairage.c*

1. Illumination

Pour activer le rendu avec le modèle d'illumination il faut le spécifier par l'appel de la fonction *glEnable(GL_LIGHTING);*

Le mode d'affichage doit être RGB ou RGBA et les coefficients prennent des valeurs entre 0.0 et 1.0.

2. Définition des matières

Pour chaque matière il faut donner les coefficients RGB pour chacun des coefficients de reflexion :

- réflexion diffuse
- réflexion spéculaire
- réflexion ambiante
- lumière émise

La réflexion spéculaire est calculée aussi en fonction du coefficient de brillance. La fonction *def_matieres()* réalise la définition d'une matière à partir de la structure contenant la scène.

3. Définition des sources

Chaque source est décrite par :

- la couleur de la lumière ambiante associée (RGB)
- la couleur de la source (RGB)
- la position de la source en coordonnées homogènes. (si w = 0, source à l'infini)
- l'allure du spot (coeff. k et angle)
- la direction du spot

La fonction *def_sources()* réalise la définition des sources à partir de la structure contenant la scène.

4. Définition du modèle

Le modèle d'éclairage attribue les propriétés d'éclairage applicable à toute la scène :

- la lumière ambiante globale (RGB).
- les coefficients des facteurs d'atténuation.
- la position locale ou à l'infini de l'observateur.
- l'option double face.

La fonction *def_modele()* donne un exemple de définition de modèle.

5. Activation des définitions.

Pour activer chacune de ces définitions, il faut utiliser la fonction `glEnable()`. On peut activer plusieurs sources de lumière en même temps mais seulement un seul modèle d'éclairage. Les positions sont données dans le repère scène, la matrice MODELVIEW est utilisée pour la transformation dans le repère observateur. La démarche est la suivante :

activation du modèle : une seule fois
activation des sources: après chaque positionnement de l'observateur .
activation de la matière : pour chaque face.

En plus de la matière, OpenGL a besoin de la normale à la face considérée. Il faut donc calculer les normales pour chacune des faces de la scène.

Pour dessiner une face, il faut :

activer la matière ;
récupérer la normale → N (tableau) ;

```
glBegin(GL_POLYGON);
   glNormal3fv(N);     /* N est la normale du polygone */
   …
glEnd();
```

On optimise les temps de calculs en donnant des normales unitaires, c'est le mode par défaut. Sinon, il faut utiliser la normalisation en donnant en argument `GL_NORMALIZE` à la fonction `glEnable()` : moins performant car il normalise tous les vecteurs.

6. Désactivations ou modifications

Pour désactiver l'action des sources, il faut utiliser la fonction `glDisable()`

Pour modifier une définition, il suffit de définir un nouveau tableau de description, puis de rappeler la fonction `glLight()` avec un index déjà existant. Les changements sont immédiatement actifs.

Modèle d'éclairage utilisé par OpenGL (simplifié):

$$I = I_{a-global} \cdot k_a + \sum_{sourcei} contribution_i$$

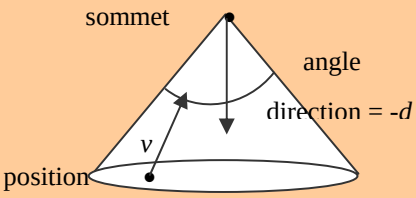
$$contribution_i = f_{atti} \cdot spoteffect_i . [ambiant_i + diffus_i + speculaire_i]$$

$$f_{atti} = \frac{1}{k_0 + d_i k_1 + d_{i^2} k_2}$$

$$spoteffect_i = \begin{cases} 1 & \text{si } angle = 180° \text{ (omnidirectionnelle)} \\ 0 & \text{si } angle \in [0,90°] \text{ (spot) et si le sommet n'est pas dans le cône d'éclairage} \\ \max(v.d,0)^k & \text{si } angle \in [0,90°] \text{ (spot) et si le sommet appartient au cône d'éclairage} \end{cases}$$

$k \in [0,128]$ exposant de concentration de la source

Description d'une source:



TP 7

1. Modifier le fichier de données et votre programme pour afficher le cube avec une couleur bleue et la porte en jaune. La matière de la porte est réfléchissante. Cette scène est éclairée avec une source devant la porte et une source au dessus du cube.

2. Définir des interrupteurs des spots lumineux à l'aide des touches F8 et F9.

3. Essayer de modifier les caractéristiques des sources (positions, cône…) par l'intermédiaire de touches du clavier.

4. Observer les effets du rendu de Gouraud liés à l'interpolation des intensités: discontinuité aux arêtes, calculs à partir des sommets…

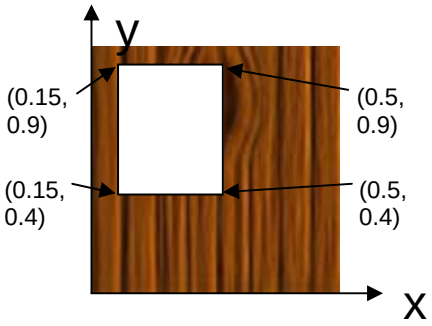
TP 8 : Textures

OpenGL permet de plaquer des textures sur des surfaces.

L'affichage de l'objet sur lequel on a placé une texture dépend de la couleur de la matière.

Pour être activée, la texture doit être chargée en mémoire, puis appelée avant de dessiner chaque face. Il faut alors indiquer, à l'appel de chaque point de la face, l'endroit de la texture correspondant.

Exemple :



Le rendu final est donc une combinaison entre la couleur de la matière et la texture que l'on a plaqué dessus. Une texture ne remplace donc pas une couleur de matière, mais la complète.

L'exemple fourni ci-dessous permet de charger des textures de type bitmap (fichier bmp) carrées, ayant pour taille une puissance de 2 (exemple : 2*2, 4*4, …, 256*256, …).

Procédure à suivre pour intégrer des textures en deux dimensions à la scène

Les textures doivent être chargées en mémoire, pour pouvoir être appelées au moment opportun.

Il n'existe pas de fonction dans OpenGL pour charger un fichier image en mémoire avant de l'utiliser. Nous vous fournissons une bibliothèque permettant de charger un fichier png en mémoire, sa fonction de lecture utilise la structure suivante:

```
typedef struct
{
   char *filename;
   GLint width;
   GLint height;
   GLenum format;
   GLint internalFormat;
   GLubyte *texels;
   GLuint glnum;
}
MTEX;
```

Pour mettre en mémoire des textures, nous devons tout d'abord demander à OpenGL où mémoriser les textures par la fonction:

```
void glGenTextures (GLsizei n, GLuint *textures)
```

avec n le nombre de textures que l'on veut utiliser et le deuxième argument les valeurs renvoyées. *textures* est un tableau de *n* composantes, contenant les indices des textures que nous allons charger.

Quand nous avons les indices des textures à charger, nous pouvons alors charger les textures en mémoire, par le code suivant :

```
//On charge la texture en mémoire
texture->filename="mon nom de fichier.png"
if(ReadPNGFromFile (texture)) {
   printf("Je n'arrive pas à lire %s\n",texture->filename);
   return -1;
}

//Mettre dans texture->glnum un numero de texture
//affecter par glGenTextures
texture->glnum=textures[k];

//Paramètres de gestion des textures
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
// Paramètres pour charger la texture
glBindTexture(GL_TEXTURE_2D,texture->glnum);
glTexImage2D (GL_TEXTURE_2D, 0, texture->internalFormat,
   texture-width, texture->height, 0, texture->format,
   GL_UNSIGNED_BYTE, texture->texels);
```

Les textures, tout comme l'éclairage ou le z-buffer, doivent être activées, par la fonction :

```
glEnable(GL_TEXTURE_2D) ;
```

Au chargement de la matière, la couleur de la matière est appelée, ainsi que la texture, à l'aide de :

```
glBindTexture(GL_TEXTURE_2D, GLuint texture);
// « texture » est le numéro de la texture à lier à la matière
// Si l'on ne veut lier aucune texture, on peut appeler
// la fonction suivante glDisable(GL_TEXTURE_2D)
```

À chaque point appelé quand on dessine un polygone, on doit indiquer le point correspondant pour l'image de la texture, à l'aide de :

```
glTexCoord2d(GLdouble coordX, GLdouble coordY) ;
// coordX et coordY varient de 0 à 1,
// comme indiqué sur le dessin
```

TP8

1. Modifier le code de manière à mettre une texture représentant du bois (fichier « bois.png ») sur une des faces du cube.
2. Introduire la texture précédente sur toutes les faces carrées de la scène.
3. Traiter le cas des faces triangulaires.
4. Proposer une modification de la structure « scene », pour pouvoir associer une texture à chaque face.
5. Proposer une modification du fichier « .dat » et de la fonction associée, pour pouvoir introduire une texture sur les faces voulues, avec les fichiers associés.

Remarque:

Le chargeur d'image utlise deux bibliothèques précompilées: n'oubliez pas d'ajouter aux chemins de vos fichiers d'en-tête (« include path ») le dossier *include* fourni ainsi que d'ajouter aux chemins de vos fichiers de bibliothèques (« lib path ») le dossier *lib* fourni.

P.S. :

Un tutorial interactif sur les textures très bien fait est présent sur le site de Nate Robins :

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

Le code du chargeur d'images PNG repose sur le code de David Henry:

http://tfcduke.developpez.com/tutoriel/format/png/

TP final

L'objectif de ce TP est d'appliquer tout ce que l'on a vu jusqu'à présent avec le cube à une scène plus complexe et partiellement décrite dans le fichier `TP_FINAL` accessible sur le site de l'UV.

1 Travail demandé (au minimum)

- Attribuer à la scène un rendu réaliste (couleur des objets, réflexion de la lumière…)
- Rajouter quelques arbres lors de la visualisation de la scène.
- Ajouter un objet mobile de votre choix (2 couleurs minimum) que l'on pourra déplacer avec les touches définies en TP.
- Eclairer la scène avec 3 sources lumineuses : l'une est située à l'infini, les 2 autres sont des sources de types spot. On peut éteindre et allumer chaque source à l'aide des touches suivantes :

  `F9` : source à l'infini
  `F10` : spot 1
  `F11` : spot 2

- Proposer une modification interactive de la position et de la direction du second spot.
- Proposer le déplacement de l'observateur dans la scène à l'aide des touches définies en TP et ajouter la possibilité de gérer 2 points de vues (avec mémoire des déplacements pour chacun des points de vue)

2 Evaluation

La soutenance du TP (démo + questions) aura lieu la dernière semaine de cours du semestre.

Le travail sera noté sur les points suivants

- le fonctionnement de la démo
- les réponses aux questions.
- l'aspect visuel du rendu réaliste.

3 Remarques :

Vous pouvez si cela vous intéresse faire plus que ce qui est demandé (lissage par calcul de moyenne de normales, mapping de textures, calculs d'ombre, etc…). Ce travail supplémentaire sera apprécié dans l'aspect visuel de la scène. Toutefois, sachant qu'il y a beaucoup de programmes existants dans le domaine, l'utilisation de ces fonctions supplémentaires doit être parfaitement comprise et ne doit pas masquer un manque d'analyse des résultats observés.