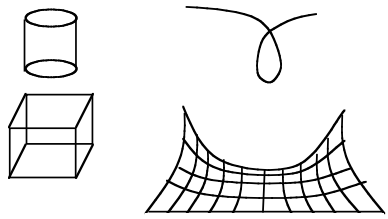
Initiation à OpenGL

Avril - 2003

Rémy Bulot – Polytech'Marseille



I. Introduction

1. L'infographie

L'œil reçoit une projection 2D (une photographie) de l'environnement dans lequel nous évoluons, et transmet cette image à notre cerveau. A partir de cette représentation partielle, le cerveau reconstruit l'environnement 3D en compensant par de nombreux « à priori » la perte d'information induite par cette projection (distance aux objets, partie cachée, …).

L'infographie consiste, par calcul, à construire une scène virtuelle (généralement 3D, mais parfois 2D) et à reproduire cette projection pour l'afficher sous forme d'une matrice rectangulaire de pixels : l'écran de l'ordinateur. L'œil capte alors cette image 2D et nous effectuons une reconstruction mentale plus ou moins réussie de la scène en fonction du degré de réalisme et du niveau des informations produites (certaines images sont d'interprétation ambiguë comme le montre le petit cube dessiné en tête de cette page).

2. Qu'est-ce qu'OpenGL

OpenGL est une librairie graphique 3D disponible sur de nombreuses plate-formes (portabilité) qui est devenu un standard en infographie.

C'est un **langage procédural** (environ 200 fonctions) qui permet de donner des ordres de tracé de primitives graphiques (segments, facettes, etc.) directement en 3D. C'est aussi une **machine à états** qui permet de définir un contexte de tracé : position de caméra, projection 2D, couleurs, lumières, matériaux... OpenGL se charge de faire les changements de repère, la projection à l'écran, le « clipping » (limites de visualisation), l'élimination des parties cachées, l'interpolation les couleurs, et de la « rasterisation » (tracer ligne à ligne) des faces pour en faire des pixels.

OpenGL s'appuie sur le hardware disponible (selon la carte graphique). Toutes les opérations de base sont a priori accessibles sur toute machine, simplement elles iront plus ou moins vite selon qu'elles sont implémentées au niveau matériel ou logiciel.

Ce que ça fait :

- sélection d'un point de vue,
- tracé de polygones,
- organisation des objets dans une scène
- effets de profondeur (luminosité décroissante, brouillard, flou de type photographie),
- placage de texture,
- éclairage des surfaces (en 3D)
- lissage ou anti-crénelage (anti-aliasing) à la « rasterisation ».

Ce que ça ne fait pas:

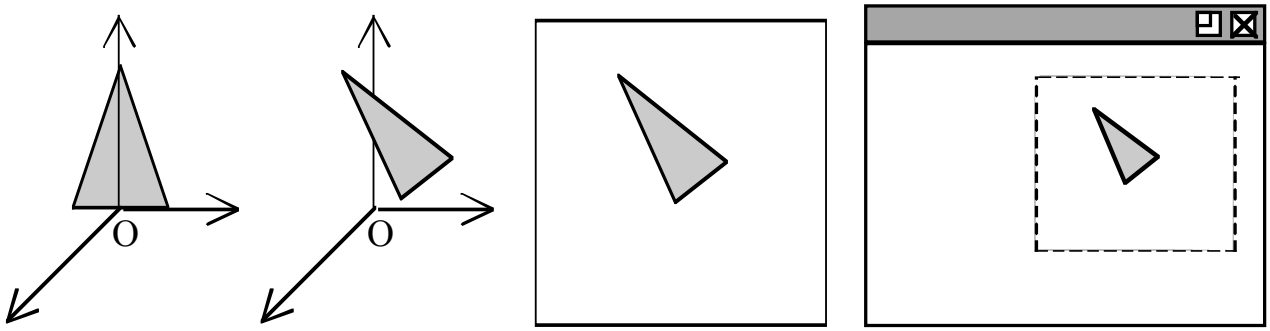
- ne gère pas le fenêtrage,
- ne gère pas les interruptions (interactions utilisateurs).

Remarque :

Une scène Virtuelle 3D n'est jamais mémorisée dans sa totalité et chaque primitive énoncée est immédiatement projetée « à l'écran » après un calcul de **rendu** plus ou moins complexe. Ce calcul se fait d'après un contexte prédéfini précisant la couleur, l'éclairage, le point de vue, le type de projection, … La simple modification de la scène ou du contexte de visualisation impose une reconstruction complète de la scène 3D car aucun résultat intermédiaire n'a été conservé.

Aussi, on retiendra tout au long de ce cours qu'un objet dans une scène 3D est défini par un **ensemble de primitives** graphiques (point, segment, triangle) et un **contexte de visualisation**. En simplifiant, le tracé d'une primitive déclenche immédiatement la séquence d'opérations suivantes :

1. définition de la primitive dans le repère de l'objet
2. transformation géométrique sur la primitive (positionnement de l'objet dans la scène)
3. projection de la primitive sur le plan image
4. cadrage dans la fenêtre de visualisation



1: primitive *2: positionnement* *3: projection* *4: cadrage*

Le plan de cet exposé ne respecte pas l'ordre des opérations effectuées par OpenGL. C'est simplement une manière d'aborder la construction d'une image. Nous parlerons de :

- GLUT : un système simple de fenêtrage
- Couleur et primitives de tracé (lignes et polygones)
- Transformations géométriques pour la construction d'une scène 3D
- Visualisation d'une scène 3D (z-buffer et projection 2D)
- Amélioration du rendu (ou rendu réaliste) : brouillard, éclairage, …
- Le cas des images 2D

Nous proposons de présenter ici un sous-ensemble des fonctionnalités d'OpenGL qui permettra une programmation déjà efficace. Certaines restrictions seront faites pour aller à l'essentiel. La première concerne le mode d'affichage que nous limiterons au **mode couleur RGB** pour la totalité de ce cours.

3. Philosophie des identificateurs OpenGL

OpenGL a pris la précaution de redéfinir les types de base pour assurer la **portabilité des applications** quelque soit l'implémentation. Si la conversion entre les types de base du C et les types GL est bien assurée (par exemple, pour les arguments passés par valeur dans une fonction OpenGL lors d'un appel), il est fortement conseillé d'utiliser les types OpenGL pour les tableaux qui sont passés par adresse.

Par ailleurs, OpenGL s'est imposé certaines règles pour créer ses identificateurs : ceux-ci disposent presque systématiquement d'un préfixe et d'un suffixe :

- Préfixe : gl*Nom*
- Suffixe : précise parfois le nombre d'arguments et leur type. La même fonction est souvent disponible pour différents type d'arguments. Par exemple, la définition d'une couleur peut se faire avec les fonctions suivantes :

```
glColor4f(rouge, vert, bleu, transparence)
glColor3iv (tableDeTroisEntiers)
```

Suffixe	**type**	**taille**	**nature**
b	`GLbyte`	8	octet signé
s	`GLshort`	16	entier court
i	`GLint`	32	entier
f	`GLfloat`	32	flottant
d	`GLdouble`	64	flottant double précision
ub	`GLubyte`	8	octet non signé
us	`GLushort`	16	entier court non signé
ui	`GLuint`	32	entier long non signé
*type*v			adr d'un vecteur de ce *type*

II. La librairie GLUT

Plan :

1. Structure d'une application GLUT
2. Initialisation d'une session GLUT
3. La boucle de traitement des événements
4. Gestion des fenêtres
5. Gestion de menus
6. Inscription des fonctions de rappel
7. Quelques variables d'état de GLUT
8. Rendu des polices de caractères

OpenGL a été conçu pour être indépendant du gestionnaire de fenêtres qui est intimement lié au système d'exploitation. Il existe toutefois un système de fenêtrage « élémentaire » qui permet de développer des applications graphiques dans un cadre simple tout en garantissant une très bonne portabilité sur de très nombreuses plate-formes : openGL Utility Toolkit (GLUT).

Les fonctionnalités de cette bibliothèque permettent principalement de :
- créer et gérer plusieurs fenêtres d'affichage,
- gérer les interruptions (click souris, touches clavier, …),
- disposer de menus déroulant,
- connaître la valeur d'un certain nombre de paramètres systèmes,

Quelques fonctions supplémentaires permettent de créer simplement un certain nombre d'objets 3D (cube, sphère, tore, …).

Cette bibliothèque s'enrichie régulièrement d'outils simples et pratiques (on trouve maintenant sur les sites OpenGL des boutons, des affichages de répertoires, …) sans devenir un « monstre » dont la maîtrise demande une longue pratique.

La philosophie générale de ce système de fenêtrage est basée sur la « programmation événementielle » (ce que l'on pourra regretter …), ce qui impose une structuration assez particulière de l'application.

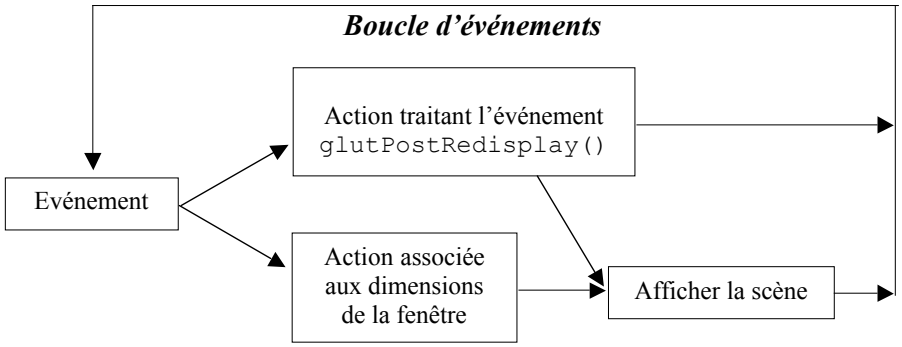
1. Structure d'une application GLUT

Une application GLUT lance une session graphique qui ne sera plus contrôlée que par des interruptions (click souris, touche clavier, …). On trouve dans le « main » les actions suivantes :
- initialisation du fenêtrage,
- désignation de la fonction d'affichage (1) dans la fenêtre courante,
- désignation de la fonction (2) déclenchée par un redimensionnant la fenêtre courante,
- association d'une fonction (3) à chaque type d'interruption,
- la boucle d'événements.

Avec les remarques suivantes :
- Toute opération de tracé est interdite en dehors de la fonction déclarée pour cette tâche (1).
- La boucle d'événement est la dernière action du programme principal et échappe totalement au contrôle du programmeur. Elle prend la main de façon définitive (jusqu'à la fin l'application) : elle réactualise régulièrement l'affichage d'une part et capte d'autre part les interruptions pour déclencher les procédures associées (3).
- C'est le système qui déclenche la fonction d'affichage (1),

- Le programmeur peut demander au système l'exécution de la fonction d'affichage au moyen de l'instruction `glutPostRedisplay()`

Cette gestion indépendante des différents processus impose l'utilisation de variables globales d'état pour contrôler l'affichage en fonction des interruptions (dialogue entre la fonction d'affichage et les fonctions traitants les interruptions).



Exemple simplifié de la structuration d'un programme GLUT :

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

void afficheMaScene(void)
{ « effacer l'écran »
  positionner la caméra
  construction (tracé) de la scène
  glutSwapBuffers(); /* glFlush() */
}

void monCadrage(int largeur, int hauteur)
{ redéfinition du cadre d'affichage après redimensionnement de la fenêtre
  definition de la projection 3D->2D
}

void maFctClavier (unsigned char key, int x, int y)
{ modification du contexte d'affichage sur un événement clavier
  glutPostRedisplay() ;
}

void maFctTouchesSpeciales(unsigned char key, int x, int y)
{ action déclenchée sur une touche F1, …, F10, flèches
  glutPostRedisplay() ;
}

void maFctSouris (int bouton, int etat, int x, int y)
{ modification du contexte d'affichage sur un événement souris
  glutPostRedisplay() ;
}

void monInitScène()
{ initialisation eventuelle de parametres propres à l'application (eclairages,
…)
}

int main (int argc, char **argv)
{     /* initialisation d'une session GLUT */
      glutInit(argc, argv); /* initialise la bibliothèque GLUT */
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(500, 500);
      glutInitWindowPosition(100, 100);
```

```
        glutCreateWindow(argv [0]);
        /* initialisation éventuelle de parametres
        monInitScène();
        /* ftcs définissant la scène3D et sa projection */
        glutDisplayFunc(afficheMaScene);          /* (1) */
        glutReshapeFunc(monCadrage);              /* (2) */
        /* ftcs liees aux iterruptions */
        glutKeyboardFunc(maFctClavier);           /* (3) */
        glutSpecialFunc(maFctTouchesSpeciales);   /* (3) */
        glutMouseFunc(maFctSouris);               /* (3) */
        /* boucle d'événements */
        glutMainLoop();
        return 0;
}
```

2. Initialisation d'une session GLUT

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

La fonction **glutInit** initialise la bibliothèque *GLUT* et négocie une session avec le système de fenêtrage. Elle traite également les lignes de commandes qui sont propres à chaque système de fenêtrage.

Paramètres pour le système X

Les paramètres de la ligne de commande qui sont compris par la bibliothèque *GLUT* sont par exemple:

- **display** *DISPLAY* Spécifie l'adresse du serveur X auquel se connecter. Si ce n'est spécifié, la variable d'environnement est utilisée.
- **geometry** *WxH+X+Y* Détermine la position de la fenêtre sur l'écran. Le paramètre de geometry doit être formaté selon la spécification standard de X.
- **gldebug** Après le traitement des fonctions de rappel ou des événements, vérifier s'il y a des erreurs *d'OpenGL* en appelant **glGetError**. S'il y a une erreur, imprimer un avertissement obtenu par la fonction **gluErrorString**.

void glutInitWindowSize (int width, int height);

void glutInitWindowPosition (int x, int y);

Les fonctions **glutInitWindowSize** et **glutInitWindowPosition** permettent de créer une fenêtre, de la positionner sur l'écran et d'en spécifier la taille.

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

void glutInitDisplayMode (unsigned int mode);

Cette fonction spécifie le mode d'affichage de la fenêtre. Le mode d'affichage est utilisé pour créer les fenêtres et les sous-fenêtres. Le mode GLUT_RGBA permet

d'obtenir une fenêtre utilisant le modèle de couleur RGB avec une composante de transparence. C'est le mode de base de la plupart des applications.

mode Mode d'affichage qui est en général une opération or bit à bit de masque de bits. Les valeurs permises sont :

- GLUT_RGBA Masque de bits pour choisir une fenêtre en mode RGBA. C'est la valeur par défaut si GLUT_RGBA ou GLUT_INDEX ne sont spécifiés.
- GLUT_RGB Un alias de GLUT_RGBA.
- GLUT_INDEX Masque de bits pour choisir une fenêtre en mode index de couleur. Ceci l'emporte si GLUT_RGBA est spécifié.
- GLUT_SINGLE Masque de bits pour spécifier un tampon simple pour la fenêtre. Ceci est la valeur par défaut.
- GLUT_DOUBLE Masque de bit pour spécifier une fenêtre avec un double tampon. Cette valeur l'emporte sur GLUT_SINGLE.
- GLUT_RGBA Masque de bits pour choisir une fenêtre avec une composante alpha pour le tampon de couleur.
- GLUT_DEPTH Masque de bits pour choisir une fenêtre avec un tampon de profondeur.
- ...

3. La boucle de traitement des événements

void glutMainLoop (void);

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

4. Gestion des fenêtres

int glutCreateWindow (char * name);

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

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

void glutSetWindow (int win);

Cette fonction établit que la fenêtre identifiée par **win** devient la fenêtre courante.

int glutGetWindow (void);

Cette fonction retourne le numéro de la fenêtre courante. Si la fenêtre courante a été détruite, alors le numéro retourné est 0.

void glutDestroyWindow (int win);

glutDestroyWindow détruit la fenêtre identifiée par le paramètre **win**. Elle détruit également le contexte OpenGL associée à la fenêtre. Si win identifie la fenêtre courante, alors la fenêtre courante devient invalide (**glutGetWindow** retourne la valeur 0).

void glutPostRedisplay (void);

Cette fonction indique que la fenêtre courante doit être réaffiché. Lors de la prochaine itération dans la boucle principale de **glutMainLoop**, la fonction de rappel d'affichage est appelée. Plusieurs appels à la fonction **glutPostRedisplay** n'engendrent qu'un seul rafraîchissement. Logiquement, une fenêtre endommagée est marquée comme devant être rafraîchie, ce qui est équivalent à faire appel la fonction **glutPostRedisplay**.

Cette fonction est principalement employée dans les procédures attachées à une interruption. La modification du contexte demande généralement une réactualisation de l'affichage.

void glutSwapBuffers (void);

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

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

void glutPositionWindow (int x, int y);

Demande un changement de position de la fenêtre courante. Les coordonnées x et y sont des décalages par rapport à l'origine de l'écran. Le changement de position n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne dans la boucle principale

Pour une fenêtre de base, le système de fenêtrage est libre d'appliquer face à la requête sa propre politique pour le positionnement de la fenêtre.

glutPositionWindow désactive le mode plein écran s'il est activé.

void glutReshapeWindow (int width, int height);

Demande un changement de dimensions de la fenêtre courante. Les paramètres **width** et **height** sont les nouvelles dimensions de la fenêtre et doivent être des entiers positifs. Le changement de dimension n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne à la boucle principale.

Pour une fenêtre de base, le système de fenêtrage est libre d'appliquer face à la requête sa propre politique pour le dimensionnement de la fenêtre.

glutReshapeWindow désactive le mode plein écran s'il est activé.

void glutFullScreen (void);

Demande que la fenêtre courante soit en plein écran. La sémantique de plein écran peut varier d'un système de fenêtrage à l'autre. Le but est d'obtenir la fenêtre la plus grande possible en la libérant des bordures et des barres de titre. Les dimensions de la fenêtre ne correspondent pas nécessairement aux dimensions de l'écran. Le changement de dimension n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne à la boucle principale.

Les appels aux fonctions **glutReshapeWindow** et **glutPositionWindow** désactivent le mode plein écran.

void glutPopWindow (void);

void glutPushWindow (void);

La fonction **glutShowWindow** affiche la fenêtre courante (elle pourrait ne pas être visible si elle est occultée par une autre fenêtre). La fonction **glutHideWindow** cache la fenêtre courante. Les actions de cacher ou afficher une fenêtre ne sont pas effectués immédiatement. Les requêtes sont conservées pour exécution future lors du retour à la boucle principale des événements. Les effets d'afficher ou masquer une fenêtre dépendent de la politique d'affichage du système de fenêtrage.

void glutSetWindowTitle (char *name);

Cette fonction s'applique à la fenêtre. Le nom d'une fenêtre est établi lorsque de la création de la fenêtre par la fonction **glutCreateWindow**. Par la suite, le nom d'une fenêtre peut être changé respectivement par un appel à la fonction **glutSetWindowTitle**.

void glutSetCursor (int cursor);

Change l'apparence du curseur pour la fenêtre courante. Valeur du curseur :

- **GLUT_CURSOR_RIGHT_ARROW** Flèche pointant vers le haut et la droite.
- **GLUT_CURSOR_LEFT_ARROW** Flèche pointant vers le haut et la gauche.
- **GLUT_CURSOR_INFO** Main directionnelle.
- **GLUT_CURSOR_DESTROY** Crâne et os (tête de mort).
- **GLUT_CURSOR_HELP** Point d'interrogation.

- **GLUT_CURSOR_CYCLE** Flèche tournant en cercle.
- **GLUT_CURSOR_SPRAY** Aérosol.
- **GLUT_CURSOR_WAIT** Montre bracelet.
- **GLUT_CURSOR_TEXT** Point d'insertion pour le texte.
- **GLUT_CURSOR_CROSSHAIR** Croix.
- **GLUT_CURSOR_UP_DOWN** Curseur bidirectionnel pointant vers le haut et le bas.
- **GLUT_CURSOR_LEFT_RIGHT** Curseur bidirectionnel pointant vers la gauche et la droite.
- **GLUT_CURSOR_TOP_SIDE** Flèche pointant vers le côté supérieur.
- **GLUT_CURSOR_BOTTOM_SIDE** Flèche pointant vers le côté inférieur.
- **GLUT_CURSOR_LEFT_SIDE** Flèche pointant vers le côté gauche.
- **GLUT_CURSOR_RIGHT_SIDE** Flèche pointant vers le côté droit.
- **GLUT_CURSOR_TOP_LEFT_CORNER** Flèche pointant vers le coin supérieur gauche.
- **GLUT_CURSOR_TOP_RIGHT_CORNER** Flèche pointant vers le coin supérieur droit.
- **GLUT_CURSOR_BOTTOM_LEFT_CORNER** Flèche vers le coin inférieur gauche.
- **GLUT_CURSOR_BOTTOM_RIGHT_CORNER** Flèche vers le coin inférieur droit .
- **GLUT_CURSOR_FULL_CROSSHAIR** Grande croix.
- **GLUT_CURSOR_NONE** Curseur invisible.

5. Gestion de menus

La bibliothèque *GLUT* supporte des menus déroulants en cascades. La fonctionnalité est simple et minimale. La bibliothèque GLUT n'a pas la même fonctionnalité que X-Windows ou WindowsXX; mais elle a l'avantage d'être portable sur plusieurs plates-formes. Il est illégal de créer ou éliminer des menus, ou de changer, ajouter ou retirer des éléments d'un menu pendant qu'il est en cours d'utilisation.

int glutCreateMenu (void (*func) (int value));

La fonction **glutCreateMenu** crée un nouveau menu déroulant et retourne un entier identifiant ce menu. La plage du numéro de menu commence à 1. Implicitement, le menu courant correspond au nouveau menu créé. L'identificateur de menu peut être

utilisé par la suite par la fonction **glutSetMenu**. Lorsque la fonction de rappel est appelée parce qu'un élément du menu a été sélectionné, la valeur du menu courant devient le menu sélectionné. La valeur de la fonction de rappel correspond à l'élément du menu sélectionné.

void glutSetMenu (int menu);

int glutGetMenu (void);

La fonction **glutSetMenu** permet d'établir le menu courant; la fonction **glutGetMenu** retourne la valeur du menu courant. Si le menu n'existe pas, ou si le menu courant précédent a été détruit, **glutGetMenu** retourne la valeur 0.

void glutDestroyMenu (int menu);

La fonction **glutDestroyMenu** détruit le menu identifié par menu. Si menu identifie le menu courant, la valeur du menu courant devient invalide ou 0.

void glutAddMenuEntry (char * name, int value);

La fonction **glutAddMenuEntry** ajoute un élément au bas du menu courant. La chaîne de caractères est affichée dans le menu déroulant. Si un élément du menu est sélectionné par un utilisateur, la valeur **value** est la valeur transmise à la fonction de rappel correspondant au menu courant.

void glutAddSubMenu (char * name, int menu);

La fonction **glutAddSubMenu** ajoute un sous-menu pour cet élément de menu. Lors de la sélection de cet élément, un sous-menu menu est ouvert en cascade pour le menu courant. Un élément de ce sous-menu peut être par la suite sélectionné.

void glutChangeToMenuEntry (int entry, char *name, int value);

La fonction **glutChangeToMenuEntry** permet de changer un élément du menu courant en une entrée du menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être changé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et **glutGet(GLUT_MENU_NUM_ITEMS)** inclusivement. La chaîne de caractères **name** est affichée pour l'entrée du menu modifiée. Si un élément du menu est sélectionné par un utilisateur, la valeur value est la valeur transmise à la fonction de rappel correspondant au menu courant.

void glutChangeToSubMenu (int entry, char *name, int menu);

La fonction **glutChangeToSubMenu** permet de changer l'élément du menu du menu courant en un élément déclenchant un sous-menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être changé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et **glutGet(GLUT_MENU_NUM_ITEMS)** inclusivement. L'identificateur **menu** nomme le menu qui est ouvert en cascade lorsque cet élément est sélectionné.

void glutRemoveMenuItem (int entry);

La fonction **glutRemoveMenuItem** élimine un élément du menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être éliminé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et

glutGet(GLUT_MENU_NUM_ITEMS) inclusivement. Les éléments du menu en dessous sont renumérotés.

void glutAttachMenu (int button);

void glutDetachMenu (int button);

Ces fonctions attachent ou détachent respectivement le menu courant à un des boutons de la souris.

6. Inscription des fonctions de rappel

La bibliothèque *GLUT* supporte un certain nombre de fonctions de rappel dont le but est d'attacher une réponse (une fonction programmeur) à différents types d'événement. Il y a trois types de fonctions de rappel:

- **fenêtre** : les fonctions de rappel concernant les fenêtres indiquent quand réafficher ou redimensionner la fenêtre, quand la visibilité de la fenêtre change et quand une entrée est disponible pour la fenêtre;
- **menu** : une fonction de rappel concernant un menu indique la fonction à rappeler lorsqu'un élément du menu est sélectionné;
- **globale** : les fonctions de rappel globales gère le temps et l'utilisation des menus

Les fonctions de rappel attachées à des événements d'entrée doivent être traitées pour les fenêtres pour lesquelles l'événement a été effectué.

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

La fonction **glutDisplayFunc** établit la fonction de rappel pour la fenêtre courante. Quand *GLUT* détermine que la fenêtre doit être réafficher, la fonction de rappel d'affichage est appelée.

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

Chaque fenêtre doit avoir une fonction de rappel inscrite. Une erreur fatale se produit si une tentative d'affichage d'une fenêtre est effectuée sans qu'une fonction de rappel n'ait été inscrite. C'est donc une erreur de faire appel à la fonction glutDisplayFunc avec le paramètre NULL.

void glutReshapeFunc (void (*func) (int width, int height));

La fonction **glutReshapeFunc** établit la fonction de rappel de redimensionnement de la fenêtre courante. La fonction de rappel de redimensionnement est déclenchée lorsque la fenêtre est refaçonnée. La fonction de rappel est aussi déclenchée immédiatement avant le premier appel à la fonction de rappel d'affichage après la

création de la fenêtre. Les paramètres **width** et **height** de la fonction de rappel de redimensionnement spécifient les dimensions en pixels de la nouvelle fenêtre.

Si aucune fonction de rappel de redimensionnement n'est inscrite ou qu'on fait appel à la fonction **glutReshapeFunc** avec la valeur **NULL**, la fonction de rappel de redimensionnement implicite est appelée. Cette fonction implicite fait simplement appel à la fonction **glViewport(0, 0, width, height)** pour le plan normal de la fenêtre courante.

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

La fonction **glutKeyboardFunc** établit la fonction de rappel du clavier pour la fenêtre courante. Lorsqu'un utilisateur tape au clavier (dans une fenêtre), chaque touche génère un appel à la fonction de rappel du clavier. Le paramètre **key** est le code ASCII de la touche. L'état d'une touche modificatrice telle majuscule [**Shift**] ne peut être connu directement; son effet se reflète cependant sur le caractère ASCII.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement de l'événement (frappe d'une touche).

Lors de la création d'une nouvelle fenêtre, aucun fonction de rappel du clavier n'est enregistrée implicitement et les touches du clavier sont ignorées. La valeur NULL pour la fonction **glutKeyboardFunc** désactive la génération de fonction de rappel pour le clavier.

Pendant le traitement d'un événement clavier, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (par exemple, la touche majuscule ou **Ctrl** ou **Alt**) lors du déclenchement d'un événement au clavier. Il faut se référer à la fonction **glutSpecialFunc** pour le traitement de caractères non-ASCII, par exemple les touches de fonction ou les touches fléchées.

void glutMouseFunc (void (*func) (int button, int state, int x, int y);

La fonction **glutMouseFunc** établit la fonction de rappel de la souris pour la fenêtre courante. Lorsqu'un utilisateur appuie ou relâche un des boutons de la souris, chaque action (appui ou relâchement d'un bouton) engendre un appel à la fonction de rappel de la souris.

Le paramètre **button** peut prendre les valeurs : **GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON**, ou **GLUT_RIGHT_BUTTON**.

Le paramètre **state** indique si la fonction de rappel a été rappelée suite à l'appui ou au relâchement d'un bouton de la souris et les valeurs permises sont : **GLUT_UP** et **GLUT_DOWN**.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement de l'événement.

Si un menu est attaché à un bouton de la souris, aucun rappel de la fonction de la souris n'est effectué pour ce bouton.

Pendant le traitement d'un événement de la souris, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (**Shift** ou **Ctrl** ou **Alt**).

La valeur **NULL** pour la fonction **glutMouseFunc** désactive la génération de fonction de rappel pour la souris.

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

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

Les fonctions **glutMotionFunc** et **glutPassiveMotionFunc** établissent les fonctions de rappel pour la fenêtre courante pour un déplacement de la souris. La fonction de rappel spécifiée par **glutMotionFunc** est appelée lors du déplacement de la souris avec un ou plusieurs boutons appuyés. La fonction de rappel spécifiée par **glutPassiveMotionFunc** est appelée lors du déplacement de la souris dans la fenêtre avec aucun bouton appuyé.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels.

La valeur **NULL** pour les fonctions **glutMotionFunc** ou **glutPassiveMotionFunc** désactive la génération de fonction de rappel lors du déplacement de la souris.

void glutVisibilityFunc (void (*func) (int state));

La fonction **glutVisibilityFunc** établit la fonction de rappel de visibilité pour la fenêtre courante . Cette fonction de rappel est appelée lorsque la visibilité de la fenêtre change. Le paramètre **state** peut prendre les valeurs **GLUT_VISIBLE** ou **GLUT_NOT_VISIBLE** selon la visibilité de la fenêtre. L'état **GLUT_VISIBLE** est valable pour une fenêtre partiellement ou totalement visible, i.e. à moins que la visibilité ne change, aucun rafraîchissement de la fenêtre n'est effectué. **GLUT_NOT_VISIBLE** signifie donc qu'aucun pixel de la fenêtre n'est visible.

La valeur **NULL** pour les fonctions glutVisibilityFunc désactive la fonction de rappel de visibilité. Si la fonction de rappel de visibilité est désactivée, l'état de la fenêtre devient indéfinie. Tout changement à la visibilité de la fenêtre est rapporté. Donc la réactivation de la fonction de rappel de visibilité garantit qu'un changement de visibilité est rapporté.

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

La fonction **glutSpecialFunc** établit la fonction de rappel du clavier pour les caractères non-ASCII pour la fenêtre courante. Des caractères non-ASCII sont générés du clavier lorsqu'une des touches de fonction (F1 à F12) ou une des touches de direction est utilisée. Le paramètre **key** est une constante correspondant à une touche spéciale (**GLUT_KEY_***). Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement d'un événement clavier. Pendant le traitement d'un événement du clavier, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (**Shift ou Ctrl ou Alt**).

La valeur **NULL** pour la fonction **glutSpecialFunc** désactive la génération de fonction de rappel pour le clavier (touches spéciales).

Les valeurs correspondant aux touches spéciales sont les suivantes:

- **GLUT_KEY_F1** Touche F1.

- **GLUT_KEY_F2** Touche F2.
- **GLUT_KEY_F3** Touche F3.
- **GLUT_KEY_F4** Touche F4.
- **...**
- **GLUT_KEY_LEFT** Touche fléchée vers la gauche.
- **GLUT_KEY_UP** Touche fléchée vers le haut.
- **GLUT_KEY_RIGHT** Touche fléchée vers la droite.
- **GLUT_KEY_DOWN** Touche fléchée vers le bas.
- **GLUT_KEY_PAGE_UP** Touche page précédente (Page up).
- **GLUT_KEY_PAGE_DOWN** Touche page suivante (Page down).
- **GLUT_KEY_HOME** Touche Home.
- **GLUT_KEY_END** Touche End.
- **GLUT_KEY_INSERT** Touche d'insertion (ins)

Il est à noter que les touches d'échappement [Escape], de recul [Backspace] et d'élimination [delete] génèrent des caractères ASCII. Voici quelques valeurs importantes de caractères ASCII:

Backspace	8
Tabulation	9
Return	13
Escape	27
Delete	127

void glutMenuStatusFunc (void (*func) (int status, int x, int y));

La fonction **glutMenuStatusFunc** établit une fonction de rappel pour l'état du menu de sorte qu'une application utilisant *GLUT* puisse déterminer si le menu est en utilisation ou non. Quand une fonction de rappel d'état du menu est inscrite, un appel est effectué avec la valeur **GLUT_MENU_IN_USE** pour le paramètre **status** quand les menus déroulants sont utilisés; la valeur **GLUT_MENU_NOT_IN_USE** pour le paramètre **status** est utilisée lorsque les menus ne sont pas en utilisation. Les paramètres **x** et **y** indique la position, en coordonnées de fenêtre, lorsque le menu a été déclenché par un bouton de la souris. Le paramètre **func** représente la fonction de rappel.

Les autres fonctions de rappel (excepté les fonctions de rappel pour le déplacement de la souris) continuent à être actives pendant l'utilisation des menus, de sorte que la fonction de rappel pour l'état du menu peut suspendre une animation ou d'autres tâches lorsque le menu est en cours d'utilisation. Une cascade de sous-menus pour un menu initial déroulant ne génère pas d'appel à la fonction de rappel pour l'état du menu. Il y a une seule fonction de rappel pour l'état du menu dans *GLUT*.

La valeur **NULL** pour la fonction **glutMenuStatusFunc** désactive la génération de fonction de rappel pour l'état du menu.

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

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

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

void glutTimerFunc (unsigned int msecs, void (*func) (int value), value);

La fonction **glutTimerFunc** établit une fonction de rappel de minuterie qui est appelée dans un nombre déterminé de millisecondes. La valeur du paramètre **value** de la fonction de rappel est la valeur du paramètre de la fonction **glutTimerFunc**. Plusieurs appels de la fonction de rappel à la même heure ou à des heures différentes peuvent être inscrits simultanément.

Le nombre de millisecondes constitue une borne inférieure avant qu'un appel à la fonction de rappel soit effectué. *GLUT* essaie d'effectuer l'appel à la fonction de rappel aussitôt que possible après l'expiration du délai. Il n'y a aucun moyen pour annuler une inscription d'une fonction de rappel de minuterie. Il faut plutôt ignorer l'appel en se basant sur la valeur du paramètre value.

7. Quelques variables d'état de GLUT

La bibliothèque *GLUT* contient un grand nombre de variables d'état dont un certain nombre (pas tous) peut être interrogé directement.

int glutGet (GLenum state);

Les principaux états de *GLUT* sont (il y en a un bon nombre) :

- **GLUT_WINDOW_X** Position en x en pixels relative à l'origine de la fenêtre courante.
- **GLUT_WINDOW_Y** Position en y en pixels relative à l'origine de la fenêtre courante.
- **GLUT_WINDOW_WIDTH** Largeur en pixels de la fenêtre courante.
- **GLUT_WINDOW_HEIGHT** Hauteur en pixels de la fenêtre courante.
- **GLUT_WINDOW_DEPTH_SIZE** Nombre total de bits du tampon de profondeur de la fenêtre courante.

- **GLUT_WINDOW_CURSOR** Le curseur courante de la fenêtre courante.
- **GLUT_SCREEN_WIDTH** Indique la largeur de l'écran en pixels; 0 indique que la largeur est inconnue ou non disponible.
- **GLUT_SCREEN_HEIGHT** Indique la hauteur de l'écran en pixels; 0 indique que la hauteur est inconnue ou non disponible.
- **GLUT_INIT_WINDOW_X** Position initiale en x en pixels relative à l'origine de la fenêtre courante.
- **GLUT_INIT_WINDOW_Y** Position initiale en y en pixels relative à l'origine de la fenêtre courante.
- **GLUT_INIT_WINDOW_WIDTH** Largeur initiale en pixels de la fenêtre courante.
- **GLUT_INIT_WINDOW_HEIGHT** Hauteur initiale en pixels de la fenêtre courante.
- **GLUT_ELPASED_TIME** Nombre de millisecondes depuis l'appel à glutIinit ou depuis le premier appel à glutGet(GLUT_ELAPSED_TIME).

La fonction **glutGet** interroge les variables d'état représenté par des entiers de la bibliothèque *GLUT*. Le paramètre **state** détermine quel état doit être retourné. Les variables d'état dont le nom commence par **GLUT_WINDOW** retournent des valeurs correspondant à la fenêtre courante. Les variables d'état dont le nom commence par **GLUT_MENU** retourne des valeurs concernant le menu courant. Les autres variables correspondent à des états globaux. Si une requête est incorrecte, la valeur -1 est retournée.

int glutGetModifiers (void);

Les valeurs retournées par cette fonction sont:

- **GLUT_ACTIVE_SHIFT** Une des touches modificatrices Shift ou CapsLock.
- **GLUT_ACTIVE_CTRL** La touche modificatrice Ctrl.
- **GLUT_ACTIVE_ALT** La touche modificatrice Alt.

La fonction **glutGetModifiers** retourne la valeur d'une des touches modificatrices lorsqu'un événement d'entrée est généré à partir du clavier, d'une touche spéciale ou de la souris. On ne doit faire appel à cette fonction que lors du traitement d'une fonction de rappel du clavier, des touches spéciales ou de la souris. Le système de fenêtrage peut intercepter certaines touches modificatrices; dans ce cas, aucun appel à des fonctions de rappel n'est effectué.

8. Rendu des polices de caractères

La bibliothèque *GLUT* supporte deux types de polices de caractères: les polices haute **qualité [stroke fonts**] pour lesquelles chaque caractère est construit à l'aide de segments de lignes et les polices de basse qualité [**bitmap fonts**] qui sont formées d'un ensemble de pixels et affichées avec la fonction **glbitmap**. Les polices haute qualité ont l'avantage de pouvoir être mises à l'échelle. Les polices basse qualité sont moins flexibles mais habituellement plus rapide à afficher.

void glutBitmapCharacter (void *font, int character);

Sans aucune liste d'affichage, la fonction **glutBitmapCharacter** affiche le caractère **character** selon la police de caractères **font**. Les polices de caractères disponibles sont:

GLUT_BITMAP_8_BY_13, GLUT_BITMAP_9_BY_15, GLUT_BITMAP_TIMES_ROMAN_10, GLUT_BITMAP_TIMES_ROMAN_24, GLUT_BITMAP_HELVETICA_10, GLUT_BITMAP_HELVETICA_12, GLUT_BITMAP_HELVETICA_18

Pour une chaîne de caractères, on utilise la fonction **glutBitmapCharacter** dans une boucle pour la longueur de la chaîne. Pour se positionner pour le premier caractère de la chaîne, on utilise la fonction **glRasterPos2f**.

int glutBitmapWidth (GLUTbitmapFont font, int character);

La fonction **glutBitmapWidth** retourne en pixels, la largeur d'un caractère dans une police de caractères supportée. Pendant que la largeur d'une police de caractères peut varier (la largeur d'une police fixe ne varie pas), la taille maximum d'une police est toujours fixe.

void glutStrokeCharacter (void * font, int character);

En n'utilisant aucune liste d'affichage, le caractère **character** est affiché selon la police de caractères **font**. Les polices de caractères sont: **GLUT_STROKE_ROMAN** et **GLUT_STROKE_MONO_ ROMAN** (pour les caractères ASCII de 32 à 127).

La fonction **glTranslatef** est utilisée pour positionner le premier caractère d'une chaîne de texte.

void glutStrokeWidth (GLUTstrokeFont font, int character);

La fonction **glutStrokeWidth** retourne en pixels, la largeur d'un caractère dans une police de caractères supportée. Pendant que la largeur d'une police de caractères peut varier (la largeur d'une police fixe ne varie pas), la taille maximum d'une police est toujours fixe.

9. Rendu d'objets géométriques

Bien que cela ne soit pas le rôle principal de GLUT, il existe quelques fonctions permettant de construire des objets géométriques 3D de base.

void glutSolidSphere (GLdouble radius, GLint slices, GLint stacks);

void glutWireSphere (GLdouble radius, GLint slices, GLint stacks);

Affichent une sphère centrée à l'origine de rayon **radius**. La sphère est subdivisée en **tranches** et en **pile** autour et le long de l'axe des z.

void glutSolidCube (GLdouble size);

void glutWireCube (GLdouble size);

Affichent un cube plein ou en fil de fer centré à l'origine. La largeur du côté est donnée par **Size**.

void glutSolidCone (GLdouble base, GLdouble height, GLint slices, GLint stacks);

void glutWireCone (GLdouble base, GLdouble height, GLint slices, GLint stacks);

Affichent un cône plein ou en fil de fer. La **base** est à **z=0** et le **sommet** du cône est à **z=height**. Le cône est subdivisé en **tranches** autour de l'axe des z et en **pile** le long de l'axe des z.

void glutSolidTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void glutWireTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

Affichent un tore plein ou en fil de fer. Le rayon intérieur **innerRadius** est utilisé pour calculer une section de cercle qui tourne autour du rayon extérieur **outerRadius**. Le tore est composé de **rings** anneaux subdivisées en **nsides** côtés.

void glutSolidDodecahedron (void);

void glutWireDodecahedron (void);

Affichent un dodécaèdre (12 côtés réguliers) plein ou en fil de fer centré à l'origine de rayon 3 en coordonnées de modélisation.

void glutSolidOctahedron (void);

void glutWireOctahedron (void);

Affichent un octaèdre plein ou en fil de fer centré à l'origine de rayon 1 en coordonnées de modélisation.

void glutSolidTetrahedron (void);

void glutWireTetrahedron (void);

Affichent un tétraèdre plein ou en file de fer centré à l'origine de rayon 3 en coordonnées de modélisation.

void glutSolidIcosahedron (void);

void glutWireIcosahedron (void);

Affichent un icosaèdre plein ou en file de fer centré à l'origine de rayon 1.0 en coordonnées de modélisation.

void glutSolidTeapot (void);

void glutWireTeapot (void);

Les fonctions **glutSolidTeapot** et **glutWireTeapot** affichent une théière pleine ou en fil de fer.

III. Les primitives graphiques

Toute primitive surfacique 3D est décomposée en triangles par OpenGL.
Le triangle, la ligne et le point sont donc les seules primitives géométriques traitées par le hardware, ce qui permet de ramener toutes les interpolations au cas linéaire (facile à traiter de façon incrémentale au niveau hardware).

Bien que cela ne rentre pas dans la catégorie des primitives, nous allons tout d'abord présenter comment définir la couleur (voir annexe) qui sera employée pour le tracé (on choisit en quelque sorte son crayon avant de dessiner) . Cette couleur est mémorisée au moyen d'une variable d'état que l'on peut modifier à tout moment. Les primitives qui seront tracées par la suite recevront cette couleur jusqu'à la prochaine modification de cette variable.

1. La couleur

Une couleur est généralement caractérisée par 3 valeurs réelles (dans l'ordre : le rouge, le vert et le bleu) où chaque composante doit varier dans `[0.0, 1.0]`. Cette représentation flottante est privilégiée au niveau hardware, notamment pour les calculs de rendu.

Une quatrième valeur, dite *composante alpha*, peut être spécifiée. Il s'agit d'un coefficient *d'opacité* qui vaut 1 par défaut. Une valeur plus faible permettra de définir une certaine transparence pour une face et de « voir » les objets qui se trouvent derrière. La gestion de ce coefficient pose un certain nombre de problèmes que nous préférons ne pas aborder pour une initiation à OpenGL. La composante alpha sera forcée à 1 lorsqu'une fonction OpenGL la réclame.

2. La couleur du fond

Un dessin commence sur une feuille dont il faut définir la couleur (le fond). Cette opération pourrait consister à tracer un rectangle de cette couleur, mais :

- ce fond n'est pas simple à définir dans le cas d'une scène 3D projetée dans la fenêtre,
- il est plus efficace d'utiliser une commande spéciale (cablée),

Il est à noter que colorier le fond consiste aussi à effacer ce qu'il y avait dans la fenêtre de visualisation en passant une nouvelle couche de « peinture ».

```
glClearColor(0.0, 0.0, 0.0, 1.0) ; /* définit la couleur d'effacement, ici noire */
```

La couleur du fond est affectée à une variable d'état et sera utilisée pour chaque appel de
`glClear(GL_COLOR_BUFFER_BIT) ; /* effacement du contenu de la fenêtre */`
`GL_COLOR_BUFFER_BIT` est une constante GL qui désigne les pixels de la fenêtre d'affichage.

Remarque : on spécifie la composante alpha qui vaut généralement `1.0` (fond opaque !).

3. La couleur des primitives

`glColor3f(`***`rouge, vert, bleu`***`)` permet de spécifier la couleur pour toutes les primitives graphiques qui vont suivre. Il est possible de rajouter un quatrième paramètre pour caractériser l'opacité (paramètre *alpha*) mais sa gestion n'est pas simple sous OpenGL et nous préfèrerons l'ignorer ici.
Quelques exemples de couleurs :

```
glColor3f(0., 0., 0.) ; /* noir */
glColor3f(1., 1., 1.) ; /* blanc */
glColor3f(0.5, 0.5, 0.5) ; /* gris moyen */
glColor3f(1., 0., 0.) ; /* rouge */
```

On peut aussi utiliser la notation vectorielle :

```
GLfloat rouge[3] = {1., 0., 0.};
GLfloat jaune[3] = {1., 1., 0.};
glColor3fv(rouge);
```

On conseille de séparer, quand on le peut, l'affectation d'une couleur de la construction géométrique d'un objet. On favorise ainsi la conception modulaire. Cela permet par exemple de construire plusieurs « clônes » de couleurs différentes :

```
glColor3fv(rouge);
dessineBicyclette(position, direction);
glColor3fv(jaune);
dessineBicyclette(autrePosition, autreDirection);
```

Une couleur peut être employée « brutalement » sans faire référence à aucun modèle de lissage ou d'éclairage. On spécifie simplement une fois avant le tracé :

```
glShadeModel(GL_FLAT)
```

OpenGL propose aussi des rendus plus « sophistiqués » avec l'option `glShadeModel(GL_SMOOTH)` qui permet d'obtenir des dégradés de couleur ou de prendre en compte l'orientation des faces par rapport aux éclairages (cette partie sera développée dans la partie *VI. Amélioration du rendu* de ce cours).

4. Primitives graphiques

Les fonctions **`glBegin(…)`** et **`glEnd()`** délimitent la suite de sommets associés au tracé.
Cette suite de sommets pourra aussi bien définir des lignes brisées (contours) que des polygones (éléments surfaciques). Un polygone doit être obligatoirement plan.

Un sommet est défini par la fonction **`glVertex*(…)`**

Par exemple, on dessine un triangle « plein » de la façon suivante:

```
glColor3f(1., 0., 0.) ; /* crayon rouge */
glBegin(GL_TRIANGLES);
/* un triangle */
glVertex3f(x1,y1,z1);
glVertex3f(x2,y2,z2);
glVertex3f(x3,y3,z3);
/* un autre triangle */
glVertex3f(x4,y4,z4);
glVertex3f(x5,y5,z5);
glVertex3f(x6,y6,z6);
/* etc. */
glEnd();
```

L'argument de `glBegin spécifie le type de primitive (OpenGL en propose 10 ), notamment` des plus complexes (qui seront décomposées en triangles...) :

- des quadrilatères (convexes et **plans**) avec GL_QUADS,
- des polygones (convexes et **plans**) avec GL_POLYGON.

On peut également ne tracer que les sommets :

```
glBegin(GL_POINTS);
glVertex2i(x1,y1);
glVertex2i(x2,y2);
glVertex2i(x3,y3);
glVertex2i(x4,y4);
glEnd();
```

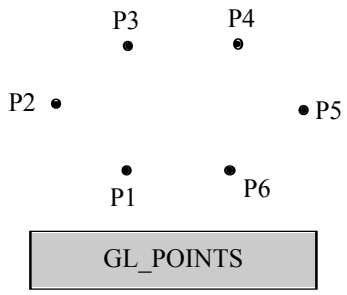
On notera au passage que `glVertex` est polymorphe :

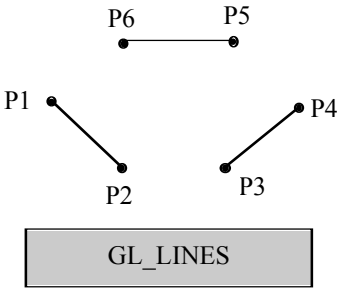
- on peut fournir 2 à 4 composantes (on peut se passer de la 3ième coordonnée si l'on fait des tracés 2D, le 4ième paramètre correspond à la coordonnée homogène...);
- on peut utiliser des `GLfloat`, des `GLdouble`, des `GLshort`, des `GLint` pour les coordonnées (suffixe `f`,`d`,`s` ou `i`);
- on peut citer explicitement les coordonnées, ou passer par un vecteur (sur-suffixe `v`).

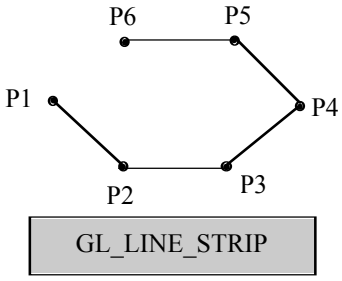
```
GLfloat P1[3] = {0, 0, 0} ;
GLfloat P2[3] = {4, 0, 0} ;
GLfloat P3[3] = {0, 2, 0} ;
glBegin(GL_LINE_LOOP);
glVertex3fv(P1);
glVertex3fv(P2);
glVertex3fv(P3);
glEnd();
```

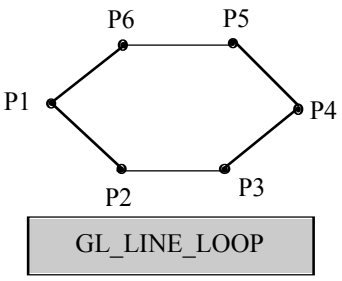
Valeurs du paramètre de `glBegin` :

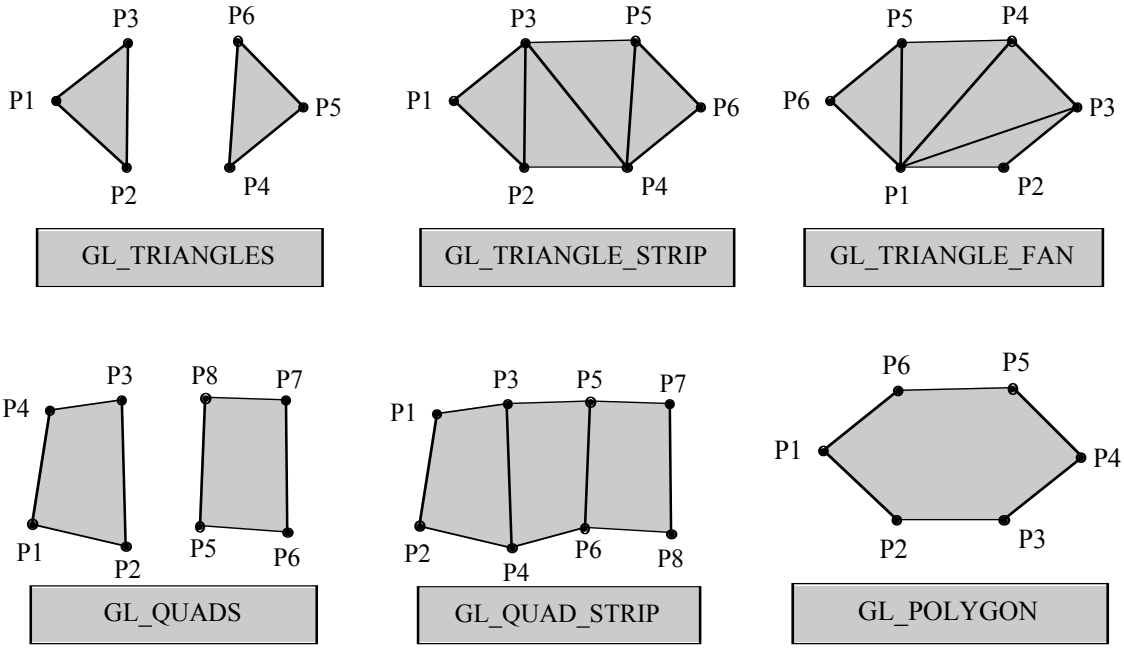
```
glBegin(GL_primitive);
  glVertex3fv(P1);
  glVertex3fv(P2);
  glVertex3fv(P3);
  glVertex3fv(P4);
  glVertex3fv(P5);
  glVertex3fv(P6);
glEnd();
```











Remarques sur l'optimisation du code GL :

- GL_TRIANGLES est plus rapide que GL_POLYGON
- Il est plus efficace de regrouper le maximum de primitives entre `glBegin()` et `glEnd()`.
- La notation vectorielle (suffixe `v`) est généralement plus rapide.

5. Faces avant et arrière

L'orientation d'un polygone est définie par l'ordre dans lequel on parcourt ses sommets lorsqu'on le dessine : traditionnellement, on définit sa **face avant** vers nous lorsque l'on parcourt ses sommets dans le sens trigonométrique. C'est cette règle qui est aussi appliquée par OpenGL pour définir l'avant et l'arrière d'une face.

OpenGL permet de traiter différemment les faces avant et arrière d'un polygone, soit par un rendu différent pour mieux les distinguer, soit pour optimiser les calculs en « oubliant » de dessiner un coté d'une f ace (par exemple la face intérieure du coté d'un cube). Cette distinction s'opère à l'aide de :

```
GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.
```

On peut obtenir des polygones pleins, ou seulement leurs contours :

```
glPolygonMode (GL_FRONT, GL_FILL); /* faces */
glPolygonMode (GL_BACK, GL_LINE);  /* contours */
```

Ou encore, supprimer une des deux faces (gain de temps !) :

```
glCullFace (GL_BACK);
glEnable (GL_CULL_FACE);
```

Dans le cas d'une scène éclairée, il faudra préciser:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE) ;
```

si l'on veut éclairer les deux côtés d'une face.

IV Construction d'une scène 3D

Il s'agit de définir ici les différents objets qui composent une scène, et de les positionner dans l'espace à l'aide de transformations géométriques 3D.

Sous GLUT, la scène est construite dans la fonction déclarée par `glutDisplayFunc` avant de lancer la boucle d'événement (`glutMainLoop`). On rappelle que cette fonction d'affichage est déclenchée par le système :
- soit parce que la fenêtre a été modifiée (redimensionnement),
- soit parce que le programmeur l'a demandé par l'intermédiaire de la fonction `glutPostRedisplay()`

OpenGL appliquera automatiquement la matrice de projection aux objets que l'on a construit pour obtenir une image 2D qui sera affichée dans la fenêtre.

1. Transformations géométriques de bases

La construction d'un objet et son positionnement dans la scène vont se faire à partir de trois transformations géométriques de bases :
- la **translation**,
- la **rotation** (en degré) autour d'un axe porté par un vecteur,
- l'**homothétie** suivant les trois axes X, Y et Z.

Ces transformations sont représentées par des matrices de dimension 4 dans l'**espace projectif**. Appliquer une de ces transformations consiste à opérer sa matrice sur les coordonnées des différents sommets de l'objet considéré.

2. Préliminaire : les Espaces Projectifs

a. Rappel sur les transformations 2D

a) Translation 2D

Soit un vecteur T(dx, dy) et un point P(x, y), alors le translaté P'(x', y') de P par T est donné par :

$$P' = P + T$$

b) Homothétie 2D

elle se fait par rapport à l'origine.

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} h & 0 \\ 0 & k \end{vmatrix} . \begin{vmatrix} x \\ y \end{vmatrix}$$

l'homothétie est dite **uniforme** si h.x = k.y et **différentielle** sinon

c) Rotation 2D

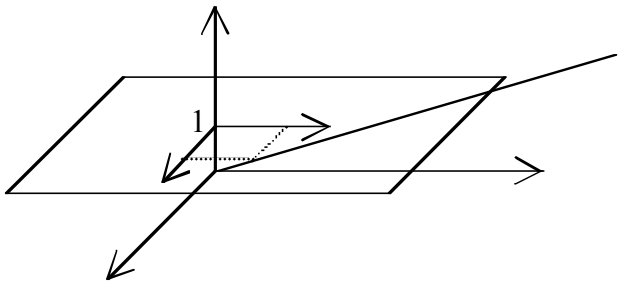
Une rotation d'angle □ autour de l'origine est définie par :

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} \cos\square & -\sin\square \\ \sin\square & \cos\square \end{vmatrix} . \begin{vmatrix} x \\ y \end{vmatrix}$$

b. Coordonnées homogènes en 2D

Si l'enchaînement d'homothétie et de rotation s'exprime sous la forme d'un produit matriciel, la translation est une opération de nature différente. Celle-ci peut toutefois être aussi définie comme un produit matriciel si les objets sont exprimés en **coordonnées homogènes**.

Les coordonnées homogènes d'un point sont obtenues en ajoutant une coordonnée supplémentaire égale à 1. On considère que tout point (x, y, w) dans **l'espace projectif** est un représentant du point (x/w, y/w) dans l'image initiale.



On remarquera que :
- un point image est associé à une droite dans l'espace projectif,
- les points pour lesquels w=0 sont des points à l'infini,
- l'enchaînement des trois transformations précédentes s'expriment sous la forme d'un produit matriciel.

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \qquad \begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} hx & 0 & 0 \\ 0 & hy & 0 \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\square & -\sin\square & 0 \\ \sin\square & \cos\square & 0 \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

c. Composition de transformations 2D

Rappel : le produit matriciel est associatif mais non commutatif.

Etudions la rotation autour d'un point C (cx, cy). Cette opération se décompose en trois transformations élémentaires :
- une translation de C vers O
- la rotation de □ autour de O
- une translation de O vers C

et qui se traduit par une formule de la forme : P' = T(cx,cy) . R(□) . T(-cx,-cy) . P

Cette opération étant effectuée sur tout les points P de l'image, on peut précalculer la composition des transformations :

$$\begin{vmatrix} 1 & 0 & cx \\ 0 & 1 & cy \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & -cx \\ 0 & 1 & -cy \\ 0 & 0 & 1 \end{vmatrix}$$

$$\begin{vmatrix} 1 & 0 & cx \\ 0 & 1 & cy \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & -\sin\theta & -\cos\theta.cx+\sin\theta.cy \\ \sin\theta & \cos\theta & -\sin\theta.cx-\cos\theta.cy \\ 0 & 0 & 1 \end{vmatrix}$$

$$\begin{vmatrix} \cos\theta & -\sin\theta & (1-\cos\theta).cx+\sin\theta.cy \\ \sin\theta & \cos\theta & -\sin\theta.cx+(1-\cos\theta).cy \\ 0 & 0 & 1 \end{vmatrix}$$

On démontre que la combinaison de ces trois transformations géométriques de base donne toujours une matrice de la forme :

$$\begin{vmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{vmatrix}$$

d. Extension au 3D

Nous nous plaçons dans un repère orthonormé direct où les rotations positives s'effectue dans le « sens inverse des aiguilles d'une montre », à savoir :

Axes de rotation	Direction d'une rotation positive
x	y à z
y	z à x
z	x à y

L'utilisation de coordonnées homogènes est naturellement applicable au 3D et les transformation géométrique de bases sont représentées par des matrices 4x4.

Un point image (x, y, z) est représenté par (x.w, y.w, z.w, w), (x, y, z, 1) étant les coordonnées homogènes.

Les matrices associées aux translations et aux homothéties sont respectivement de la forme :

$$\begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 1 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} hx & 0 & 0 & 0 \\ 0 & hy & 0 & 0 \\ 0 & 0 & hz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Les rotations se décomposent facilement suivant les axes du repère.
Les rotations autour de Oz, Ox, et Oy sont respectivement de la forme :

$$\begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

3. Transformations géométriques sous OpenGL

OpenGL dispose d'une matrice de transformation courante pour la modélisation, matrice qui est rendue active par la fonction :

```
glMatrixMode(GL_MODELVIEW)
```

Cette **matrice de modélisation** est appliquée automatiquement à tous les objets qui vont être tracés.

Le positionnement d'un objet dans une scène est décomposé comme une succession de transformations de base qu'OpenGL traduit par un produit matriciel (cf les *espaces projectifs*). La matrice de transformation courante est construite :

- à partir de la matrice identité,
- et par produits successifs avec des matrices d'opérations de base.

La matrice de modélisation est initialisée avec l'identité en appelant la fonction :

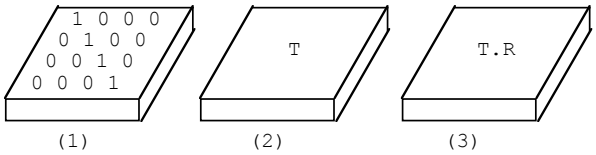
```
glLoadIdentity()
```

Il est possible de gérer soit même les opérations de base mais OpenGL propose des fonctions simples où le produit matriciel avec la matrice de transformation courante est implicite :

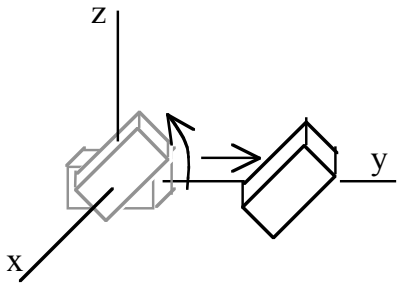
```
glTranslatef(GLfloat x, GLfloat y, GLfloat z) ;
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z) ;
glScalef(GLfloat x, GLfloat y, GLfloat z) ;
```

Les appels successifs de ces fonctions composent donc une seule transformation.
Remarque : il faut lire les opérations effectuées sur l'objet dans l'ordre inverse de leur apparition dans le code.

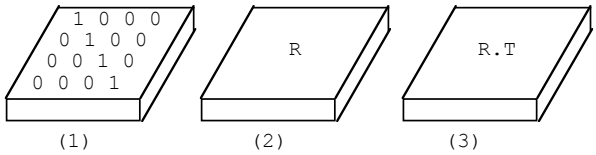
```
/* matrice de transformation A */
glLoadIdentity() ;      /* 1 */
glTranslatef(0, 5, 0);  /* 2 */
glRotatef(45, 1, 0, 0); /* 3 */
/* objet qui subira la transformation*/
dessineBoite();
```



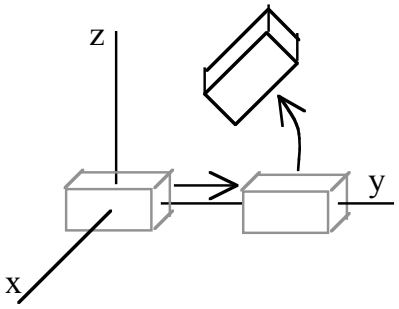
Evolution de la matrice de modélisation (A)




```
/* matrice de transformation B */
glLoadIdentity() ;      /* 1 */
glRotatef(45, 1, 0, 0); /* 2 */
glTranslatef(0, 5, 0);  /* 3 */
/* objet qui subira la transformation*/
dessineBoite();
```



Evolution de la matrice de modélisation (B)



4. Gestion des transformations

OpenGL dispose en fait d'une **pile de matrices de modélisation** qui va faciliter la description hiérarchique d'un objet complexe. La matrice courante (la seule active) est celle se trouvant au sommet de pile, mais les jeux d'empilage et dépilage permettront d'appliquer la même transformation à plusieurs assemblages ayant eux mêmes nécessité des transformations spécifiques pour leur construction.

`glLoadIdentity()` : mettre l'identité au sommet de pile
`glPushMatrix()` : empiler
`glPopMatrix()` : dépiler

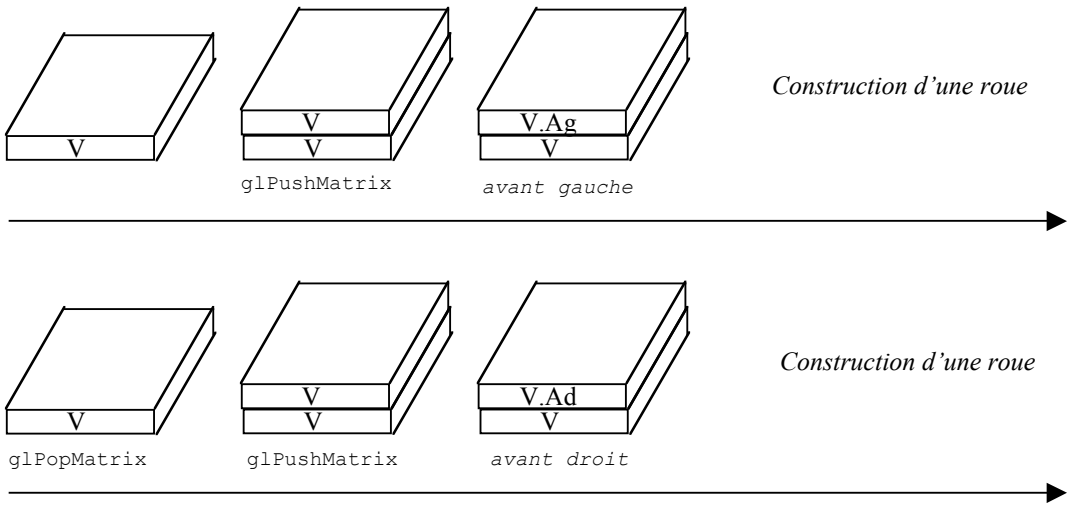
Si l'opération de dépilage ne présente pas de difficulté particulière (on retrouve la transformation précédente), l'opération d'empilage réclame quelques précisions : il s'agit en fait d'une duplication de la matrice se trouvant au sommet de pile. Toutes les opérations qui seront effectuées par la suite seront combinées à la transformation initiale (dupliquée dans le sommet de pile) de sorte que le sous-objet que l'on est en train de construire « hérite » de la transformation appliquée globalement à l'objet.
Bien sûr, il est toujours possible de faire suivre un `glPushMatrix()` par `glLoadIdentity() pour oublier temporairement une matrice de modélisation`.

Par exemple, pour dessiner une voiture, on définira ce qu'est la construction d'une roue dans le repère absolu, et on se positionnera successivement aux quatre coins de la voiture (repère voiture) avant d'appeler cette procédure. Si la voiture a été positionnée à un endroit spécifique de la scène, ses roues subiront aussi la transformation.

```
void dessineRoueEtBoulons()
{ int i ;
  dessineRoue() ;
  for (i=0; i<3; i++)
  { glPushMatrix() ;
      glRotatef(120*i, 0, 0, 1) ;
      glTranslatef(2, 0, 0) ;
      dessineBoulon() ;
    glPopMatrix() ;
  }
}

void dessineVoiture()
{ int i, posx[4]={20, 20, -20, -20}, posz[4]={8, -8, 8, -8};
  dessineCarrosserie() ;
  for (i=0; i<4; i++)
  { glPushMatrix() ;
      glTranslatef(posx[i], 5, posz[i]) ;
      dessineRoueEtBoulons() ;
    glPopMatrix() ;
  }
}
```

V : matrice positionnant la voiture dans la scène (repère « scène »)
Ag : matrice définissant l'avant gauche de la voiture dans le repère « voiture »
Ad : matrice définissant l'avant droit de la voiture dans le repère « voiture »



Evolution de la pile de modélisation dans la construction hiérarchique d'une voiture

Remarques :

a. La pile est initialisée par OpenGL avec la matrice identité.
b. Il est intéressant de conserver la matrice identité en bas de pile pour éviter de rappeler systématiquement `glLoadIdentity()` (=> une construction débute toujours par un *push*) .

5. Listes d'affichage

Il est possible de stocker une suite de routines OpenGL (à l'exception de quelques fonctions...) dans une liste qui pourra être réutilisée plusieurs fois. Il y a alors une précompilation des instructions GL et cette opération sera particulièrement rentable lorsqu'un objet « définitif » est dessiné plusieurs fois, soit parce qu'il constitue une primitive employée à plusieurs reprises (roue d'une voiture), soit parce qu'il se déplace dans la scène (animation).

c. une liste est identifiée par un numéro (`GLint`) strictement positif,
d. la création et la suppression des listes est gérée par OpenGL : **`glGenLists(`*`nbIndex`*`)`** attribue *`nbIndex`* numéros de listes consécutifs, le premier numéro est retourné par cette fonction (0 si échec d'allocation),
e. **`glDeleteLists(`*`numListe, nbre`*`)`** restitue au système le *nbre* de listes indiqué à partir de *`numListe`*.
f. **`glNewList(`*`numListe, mode`*`)`** et **`glEndList()`** permettent de créer une liste,
g. **`glCallList(`*`numListe`*`)`** exécute une liste d'affichage,

Par exemple, pour dessiner un tricycle, on pourra stocker la construction d'une roue dans une liste et appeler 3 fois cette liste après avoir définie les spécificités de chaque roue (position, taille).

```
int listeRoue ;
...
listeRoue  = glGenLists(1);
...
glNewList(listeRoue, GL_COMPILE) ; // ou encore GL_COMPILE_AND_EXECUTE
  suite d'instructions pour dessiner une roue de tricycle
glEndList() ;


void dessinerTricycle()
{ ...
  transformations pour positionner la roue arrière gauche
  glCallList(listeRoue);
  transformations pour positionner la roue arrière droite
  glCallList(listeRoue);
  transformations pour positionner la roue avant
  glScalef(1.2, 1.2, 1.) ; // roue avant 20% plus grande mais même largeur
  glCallList(listeRoue);
  ...
}
```

Remarque : Une liste peut-être construite avant de déclencher la boucle d'événements dans le `main()` (`glutMainLoop()`). Il faut toutefois que le processus graphique soit déjà initialisé (i.e. après `glutInit()` et `glutInitDisplayMode()`) et que l'on choisisse l'option `GL_COMPILE` .

V. Visualisation d'une scène

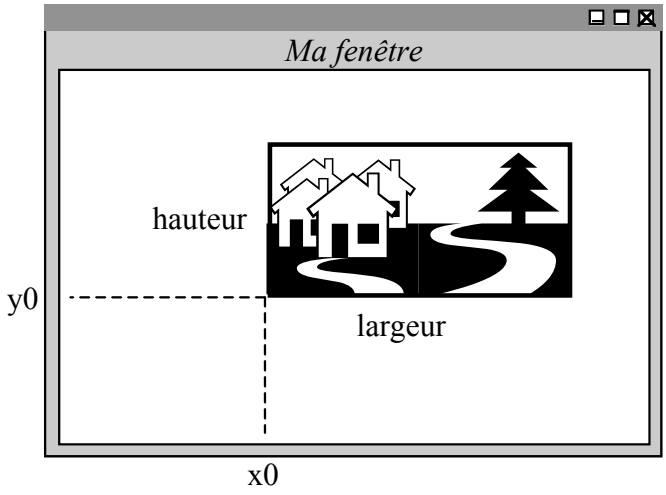
1. Cadrage

Il ne faut pas confondre la fenêtre d'affichage (définie par le système de fenêtrage qui est indépendant d'OpenGL) et le cadre (partie de fenêtre) dans lequel on veut visualiser la scène. De même que l'on colle une photo sur un poster, on va positionner l'image de la scène dans la fenêtre en la déformant éventuellement pour la faire rentrer dans un cadre (il faut conserver le ratio largeur/hauteur pour obtenir une image non « déformée »). C'est le rôle de la fonction `glViewport` .

`glViewport(GLint x0, GLint y0, GLint largeur, GLint hauteur)`

`x0` et `y0` précisent le coin inférieur gauche du cadre

`largeur` et `hauteur` précisent ses dimensions en pixels



GLUT envoie un événement pour dire que la fenêtre courante a été créée, déplacée ou redimensionnée. Cet événement est intercepté par la fonction désignée dans le main par *glutReshapeFunc* (monCadrage dans notre exemple) et l'affichage de la scène est automatiquement relancée. Le programmeur pourra, à cette occasion, décider d'adapter ou non la projection de la scène aux dimensions de la fenêtre. Exemple :

```
void monCadrage(int large, int haut)
/* les arguments sont fournis par GLUT : nouvelle taille de la fenêtre */
{ /* taille du cadre d'affichage dans la fenetre */
  glViewport(0, 0, large, haut) ;
  /* On peut a cette occasion redéfinir la projection de la scene */
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  gluPerspective(90., (float)large/(float)haut, 5, 20) ;
  glMatrixMode(GL_MODELVIEW) ;
  /* la fonction afficheMaScene() va etre automatiquement
     declenchee par le systeme */
}
```

2. Le mode *projection*

Lorsqu'une scène est construite, sa visualisation nécessite deux types de transformation :
- des transformations dans l'espace 3D qui permettent de positionner le point de vue et les éventuels éclairages si l'on veut obtenir un rendu réaliste,
- la transformation qui consiste à projeter cette scène 3D sur une fenêtre 2D et qui caractérise les propriétés de la prise de vue.

Pour le premier type de transformation, on utilise encore la **matrice de modélisation** (`GL_MODELVIEW`) : les lumières et le point de vue sont positionnés comme les autres acteurs de la scène.

Pour la projection 2D, openGL dispose d'une autre matrice spécifique : la **matrice de projection** .

Une seule de ces deux matrices est active à un moment donné et l'on bascule de l'une à l'autre avec :

```
glMatrixMode(GL_PROJECTION)
```

ou bien

```
glMatrixMode(GL_MODELVIEW)
```

Dans la pratique, on pourra définir la projection soit :
- dans la fonction de cadrage (cf la fonction `cadrage` ci-dessus),
- dans la fonction d'affichage déclarée par `glutDisplayFunc(afficheMaScene)`.

Dans ce deuxième cas, la fonction `afficheMaScene` a la structure suivante :

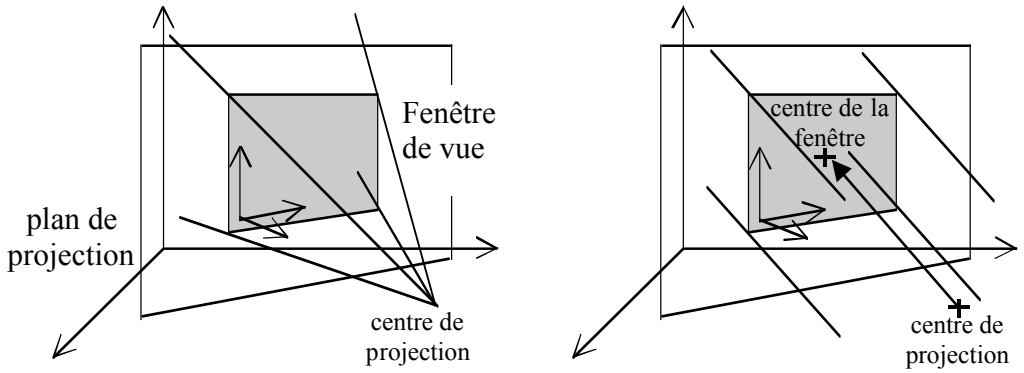
```
void afficheMaScene(void)
{ glMatrixMode(GL_PROJECTION) ;
  definir la projection
  glMatrixMode(GL_MODELVIEW) ;
  glClear(GL_COLOR_BUFFER_BIT); /* définit le fond de la scène */
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}


int main (int argc, char **argv)
{
      …
      glutReshapeFunc (monCadrage);
      glutDisplayFunc (afficheMaScene);
      …
      glutMainLoop ();
      return 0;
}
```

Remarque : dans les exemples de ce document, le cadrage et la projection sont definis dans la même fonction `monCadrage(int l, int h)`

3. Caractéristiques de l'appareil photo

OpenGL propose deux types de projection : la projection en **perspective** et la projection **parallèle**.



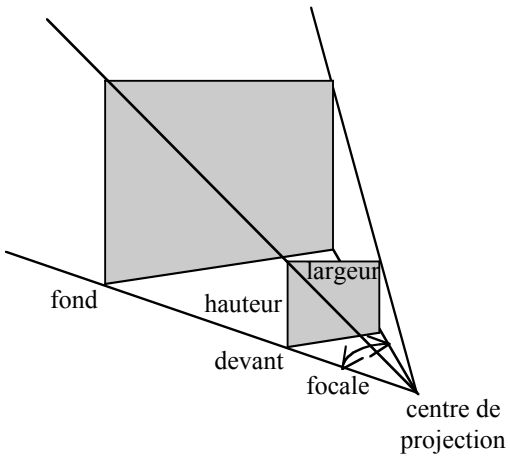
Pour un volume visionné en perspective conique :

`gluPerspective(GLdouble focale, GLdouble aspect, GLdouble devant, GLdouble fond)`

`focale` : angle du champ de vision (dans [0°, 180°])

`aspect` : rapport largeur/hauteur du plan de devant

`devant`, `fond` : distances (valeurs positives) du point de vue aux plans de clipping.



Pour un volume visionné en perspective cavalière (projection parallèle) :

`glOrtho(GLdouble gauche, GLdouble droite, GLdouble bas, GLdouble haut,`
`GLdouble devant, GLdouble fond)`

définit la « boite » de visualisation où (`gauche, bas, devant`) sont les coordonnées du point avant-inférieur-gauche et `(droite, haut, fond)` sont les coordonnées du point arrière-supérieur-droit.

Remarques :

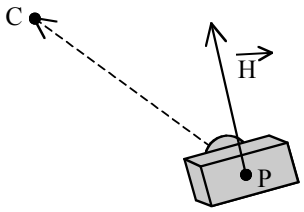
- Bien que cela soit possible, on ne compose généralement pas les projections.
- Il est important de minimiser au mieux la distance entre le plan de clipping avant et le plan de clipping arrière. En effet, OpenGL dispose d'une précision limitée pour représenter l'intervalle des profondeurs et une mauvaise gestion peut positionner des sommets artificiellement dans le même plan par effet d'arrondi, au risque de créer des artefacts de rendu. Ce phénomène peut se manifester occasionnellement lorsque l'on fait tourner un objet.

4. Positionnement de l'appareil photo

Le point de vue se situe par défaut à l'origine en regardant vers l'axe des z négatifs. Pour visualiser une scène, on peut
- soit la reculer pour la mettre dans le champ de vision,
- soit déplacer le point de vue avec `gluLookAt` .

Bien que le positionnement du point de vue soit déterminant pour effectuer la projection, on comprend pourquoi celui-ci doit être défini en mode **`GL_MODELVIEW`** : c'est un déplacement relatif de la scène.

```
gluLookAt( GLdouble Px, GLdouble Py, GLdouble Pz,      // position de l'appareil
           GLdouble Cx, GLdouble Cy, GLdouble Cz,      // point visé dans la scène
           GLdouble Hx, GLdouble Hy, GLdouble Hz)      // haut de l'appareil
```



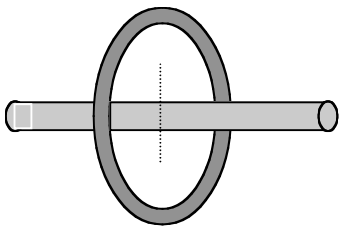
On peut maintenant donner un schéma un peu plus précis de la fonction d'affichage :

```
void afficheMaScene(void)
{ /* on recule de 5 dans la scène */
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity() ;
  gluLookAt(0,0,5, 0,0,0, 0,1,0) ;
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}
```

5. Z-buffer

Lorsque deux objets sont positionnés dans une scène, il est possible que l'un des deux soit partiellement ou totalement caché par l'autre en fonction de la position de l'observateur. Or, en pratique, le dernier dessiné écrase une partie du premier, et ceci indépendamment du point de vue qui peut changer.

L'algorithme du peintre est une solution qui n'est plus guère utilisée. Il consiste à trier les objets suivant l'ordre décroissant de leur distance au point de vue et à les dessiner dans cet ordre. Les objets en premier plan seront dessinés en dernier et « écraseront » les parties cachées des objets en arrière plan.



Découpage d'objets pour disposer d'une relation d'ordre totale.

Bien qu'un peu plus coûteux en espace mémoire, on préfère maintenant utiliser un tampon de profondeur (distance au point de vue) qui permet de s'affranchir de l'ordre de construction des objets : le **Z-buffer**. Le Z-buffer a la taille de la fenêtre de projection et est initialisé avec la plus grande profondeur possible (généralement le plan de clipping arrière). Lorsqu'un objet est dessiné, les pixels qui lui correspondent dans le plan de projection ont une valeur de profondeur qui est comparée à celle stockée dans le Z-buffer. Si un pixel est plus éloigné, il est abandonné. Sinon, le Z-buffer reçoit sa profondeur et la fenêtre reçoit ses valeurs chromatiques.

Mise en œuvre :

On déclare l'utilisation du Z-buffer à l'initialisation avant de rentrer dans la boucle d'événements avec la constante **`GLUT_DEPTH`** :

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

On active ou désactive le mode Z-buffer avec :

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
```

Ces deux opérations peuvent se faire à tout moment et permettent par exemple de rajouter des tracés « par dessus » la représentation d'une scène.

De même que l'on (re)définit la couleur du fond avec `glClearColor(r, v, b, a)`, on (re)définit au moins une fois la distance maximum de représentation d'un pixel :

```
glClearDepth(10.0) ; /* distance max visible au point de vue */
```

On définit ainsi la position d'un plan arrière de clipping lié au point de vue (ce qui est derrière sera occulté par ce plan).

Enfin, le dessin d'une scène sera toujours débuté par une réinitialisation du Z-buffer à la profondeur maximum :

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Cette opération est généralement associée à l'effacement de la fenêtre avec la couleur du fond :

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

Notre fonction d'affichage aura donc la forme suivante :

```
void afficheMaScene(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  /* on s'écarte de 5 dans la scène */
  glLoadIdentity() ;
  gluLookAt(0,0,5, 0,0,0, 0,1,0) ;
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}
```

6. La face cachée d'OpenGL

6.1 Ordre des opérations de construction d'une scène

Nous avons décrit l'ensemble des opérations qui permettent d'afficher une scène 3D en suivant un enchaînement « naturel » de la construction : construction des objets, positionnement dans la scène 3D, calcul des éclairages et des parties visibles, projection à l'écran.

Cette présentation pourrait laisser penser que la scène 3D est mémorisée quelque part et que l'on pourrait intervenir localement dessus avant d'effectuer une nouvelle projection. Il n'en est rien ! et il aurait fallu pour cela une capacité mémoire phénoménale.

On remarquera que le contexte de visualisation d'une primitive doit toujours être défini au préalable. Dans la pratique, chaque fois qu'une instruction de tracé d'un point, segment ou triangle est exécutée, son rendu et sa projection sont immédiatement déclenchés pour mettre à jour la mémoire écran (seule information conservée !) .

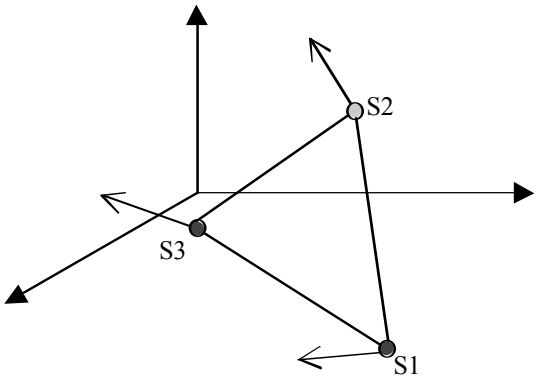
Une petite modification dans une scène implique donc un nouveau tracé complet de cette dernière. On pourra toutefois optimiser les calculs en découpant cette scène en plusieurs plans de profondeur que l'on mémorise. Par exemple, dans le cas d'une animation de personnage, on calcule et mémorise l'arrière-plan supposé ne pas évoluer ; la visualisation de la scène consiste alors à afficher l'arrière-plan (image 2D) et à superposer le personnage dans sa nouvelle position.

6.2 Détail du tracé d'une facette

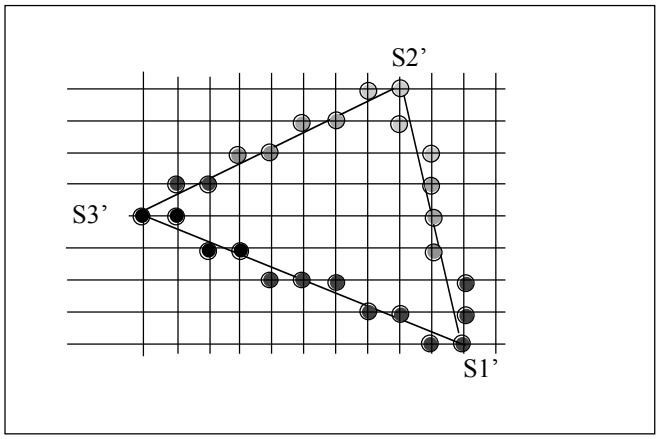
On peut s'intéresser un peu plus au fonctionnement de la « boîte noire » et étudier l'enchaînement des algorithmes implicitement mis en oeuvre par OpenGL lors du tracé d'une simple facette triangulaire. On pourra ainsi mieux « apprécier » les performances des matériels actuels lorsque l'on visualise des surfaces composées de plusieurs milliers de triangles…

Ainsi donc, la désignation de 3 sommets (`glVertex`) entre les deux instructions `glBegin(GL_TRIANGLES)` et `glEnd()` a pour effet de déclencher la séquence d'opérations suivante :

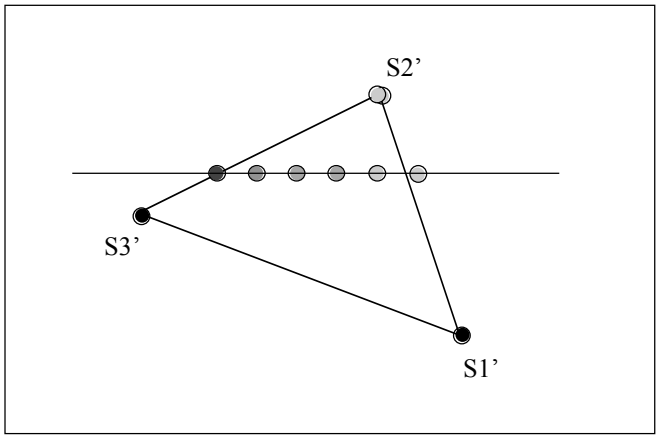
- produit des sommets par la matrice de modélisation (`GL_MODELVIEW`)
- évaluation de la couleur de chaque sommet en fonction du type de rendu (couleur brute ou simulation d'un éclairage avec gestion des normales, des lumières et du point de vue)



- évaluation de la distance des sommets au point de vue (profondeur)
- projection des sommets sur le plan image
- calcul des **pixels** constituant les 3 arêtes (algorithme de Bresenham) du triangle projeté
- extrapolation de la couleur et de la profondeur de chacun de ces pixels à partir des 3 sommets



- pour chaque ligne horizontale reliant deux arêtes
    - pour chaque pixel d’une ligne
        - extrapoler sa couleur
        - extrapoler sa profondeur
        - si sa profondeur est plus faible que son équivalent dans le z-buffer, alors tracer le point et affecter sa profondeur dans le z-buffer



VI. Amélioration du rendu

On parle de rendu réaliste lorsqu'une image contient la plupart des effets de lumière en interaction avec des objets physiques réels. Les travaux de recherche en ce domaine sont très nombreux et les solutions proposées sont parfois fort coûteuses suivant les effets recherchés.

Il ne faut toutefois pas perdre de vue que si le but principal est de communiquer une information, alors une image simplifiée peut être plus réussie qu'une image approchant la perfection d'une photographie : l'information n'est pas noyée dans un contexte peu pertinent pour l'observateur.

La réalité peut même parfois être intentionnellement altérée, voire même faussée, dans le but de faire encore mieux émerger le message que l'on veut transmettre : les films de science-fiction en sont un exemple flagrant lorsque les explosions dans l'espace sont accompagnées d'un effet sonore...

1. Le brouillard (fog)

La représentation 2D d'une scène 3D génère une perte d'information que l'observateur doit pouvoir reconstruire mentalement. Cette opération peut être rendue quasiment inconsciente si on utilise quelques « astuces » de rendu. La projection en perspective génère déjà la sensation de profondeur ; on peut renforcer cet effet en simulant un brouillard qui estompe les objets en fonction de leurs distances respectives au point de vue.

Le brouillard est activé (respectivement désactivé) avec :

```
glEnable(GL_FOG)
glDesable(GL_FOG)
```

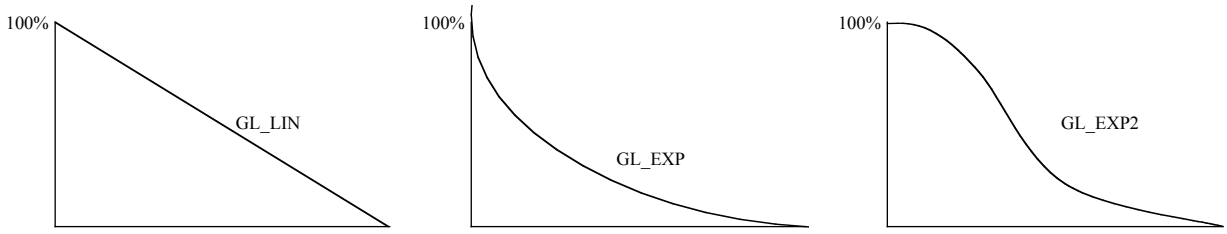
On lui associe une couleur vers laquelle tend un objet si on éloigne ce dernier du point de vue.

```
GLfloat fogColor[4] = {0.5, 0.5, 0.3, 1.} ; /* brouillard type Sirocco */
```

Cette couleur sera généralement utilisée pour le fond de la scène :

```
glClearColor(0.5, 0.5, 0.3, 1. ) ;
```

OpenGL propose trois types de courbe de mélange entre la couleur de l'objet et le brouillard : GL_LINEAR, GL_EXP et GL_EXP2



Pourcentage de la couleur d'origine d'après la distance au point de vue.

Les caractéristiques du brouillard sont définies à l'aide de la fonction `glFog`*`type`*`()` :

```
/* profil de la fonction brouillard */
glFogi(GL_FOG_MODE, GL_EXP2) ;
/* extrémités de la fonction brouillard */
glFogf(GL_FOG_START, 1.) ;
glFogf(GL_FOG_END, 5.) ;
/* coefficient de « cintrage » pour les profils EXP(2)*/
glFogf(GL_FOG_DENSITY, 0.35) ;
```

```
/* couleur du brouillard */
glFogfv(GL_FOG_COLOR, fogColor) ;
```

2. L'éclairage

OpenGL propose aussi un rendu plus réaliste avec un modèle d'illumination qui permet de prendre en compte l'orientation des surfaces par rapport aux lumières (voir annexe sur la couleur).

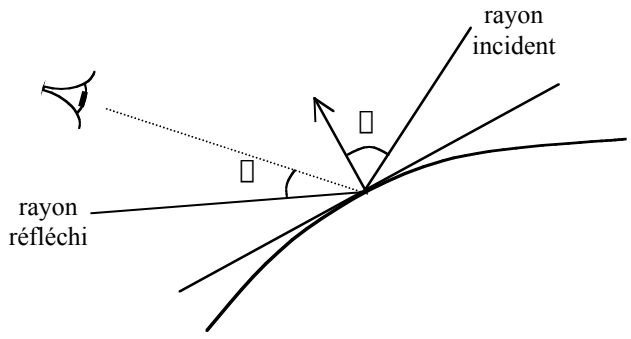
Ce rendu dépendra de :
- la position et des propriétés des éclairages,
- des propriétés optiques des matériaux employés pour la construction des objets,
- de l'orientation des surfaces vis-à-vis des éclairages et de l'observateur.

a) Un modèle physique simplifié

La lumière est représentée par la composition de trois valeurs : le rouge, le vert et le bleu. Les proportions entre ces trois valeurs vont définir une couleur que l'oeil est apte à percevoir. Nous sommes habitués à vivre avec une lumière blanche (fournie par le soleil), mais les sources peuvent être multiples (ex : éclairage d'un terrain de football générant 4 ombres sur chaque joueur) et de couleurs variées (batterie de projecteurs pour un spectacle).

Lorsqu'un rayon lumineux frappe une surface, une partie est absorbée (filtrage), une autre est réfléchie suivant les lois de la normale à la surface, le reste est restitué dans toutes les directions. OpenGL simule ces propriétés à l'aide de quatre composants (simulation d'après le modèle Lambertien, 19ème siècle) :

- l'**intensité ambiante** L_a simule une lumière qui a été dispersée par l'environnement : elle n'a pas de direction et sera réfléchie par une surface dans toutes les directions. Ainsi, une surface dans une zone d'ombre n'apparaîtra pas noire car éclairée par cette lumière ambiante.
- la **réflexion diffuse** L_d caractérise les surfaces mats. Aussi, lorsqu'un rayon lumineux provenant d'une direction particulière frappe cette surface, il sera filtré et dispersé dans toutes les directions avec une même intensité ($L_d = k_d \cos \theta$). L'effet sera donc lié à la position de la source lumineuse.
- la **réflexion spéculaire** L_s caractérise, quant à elle, le cône de réflexion de la lumière pour les surfaces brillantes. C'est elle qui va définir la brillance d'une surface lorsque l'oeil est dans l'axe symétrique de celui de la source par rapport à la normale. L'effet sera donc lié à la fois à la position de la source lumineuse et à la position de l'observateur.
- Les objets peuvent avoir une lumière **émissive** qui ajoute de l'intensité à l'objet. Par simplification, cette lumière n'ajoute pas d'éclairage supplémentaire à la scène.



$$L = L_a + k_d \cos \theta + k_s \cos^n \theta$$

Le terme n dans l'expression de la réflexion spéculaire est appelé "coefficient de surbrillance". C'est lui qui va déterminer l'étendue du reflet (saturation sur une portion de surface) que l'on peut observer sur une surface brillante lorsque certains rayons réfléchis se rapprochent de l'axe d'observation (proche de 0). Cette valeur caractérise les propriétés physiques de la surface éclairée. Un miroir parfait sera caractérisé par une valeur de n égal à l'infini : l'observateur n'est ébloui que si le rayon réfléchi coïncide avec l'axe d'observation.

b) Eclairage sous OpenGL

On passe du modèle couleur « simple » (`glColor3f(r, v, b )` au modèle d'éclairage avec :

```
glEnable (GL_LIGHTING);
glDisable (GL_LIGHTING);
```

Pour tracer une facette, il faudra définir :

- les propriétés des « lumières » (au plus 8)
- les positions des lumières
- l'interrupteur des lumières (allumer/éteindre)

- les propriétés de réflexion des matériaux (brillant, mat, …)

- le choix des faces « visibles » (avant, arrière, avant&arrière)

- le choix d'un rendu lisse où à facettes
- la normale pour chaque sommet

c) Les lumières

On peut mettre en place jusqu'à 8 lumières (`GL_LIGHT0, ..., GL_LIGHT7)` dont on peut spécifier de nombreux attributs par :

```
glLightfv(GL_LIGHT0, attribut, vecteur de float)
```

la position : `GL_POSITION . Sous OpenGL, une lumière est assimilée à un objet de la scène et subit les transformations géométriques définit pour GL_MODELVIEW. On peut donc la rendre fixe, la lier éventuellement à un objet, à la scène ou au point de vue : tout dépend de l'instant où on positionne cette lumière.`

```
GLfloat Lposition1 [4] = {-5.0, 0.0, 3.0, 0.0}; /* lumière a l'infini */
GLfloat Lposition2 [4] = {-5.0, 0.0, 3.0, 1.0}; /* position réelle */
glLightfv (GL_LIGHT0, GL_POSITION, Lposition1);
```

On notera dans cet exemple qu'une lumière peut être positionnée à l'infini (rayons parallèles) en mettant sa 4ème coordonnée à 0 (cf. les espaces projectifs). Les trois premières coordonnées définissent alors une direction et non plus une position.

la couleur : `GL_AMBIENT , GL_DIFFUSE` et `GL_SPECULAR` . On donne séparément les composantes ambiante, diffuse et spéculaire, ce qui permet de faire des choses peu physiques, comme des sources qui ne génèrent pas de reflets, ou que des reflets, ou qui ne contrôlent que la lumière ambiante. Chaque composante est un tableau de `float` définissant les 4 coefficients de base RVBA.

```
GLfloat Lambiant [4] = {0.4, 0.4, 0.4, 1.0};
GLfloat Lblanche [4] = {1.0, 1.0, 1.0, 1.0};

glLightfv (GL_LIGHT0, GL_AMBIENT, Lambiant);
glLightfv (GL_LIGHT0, GL_DIFFUSE, Lblanche);
```

```
glLightfv (GL_LIGHT0, GL_ SPECULAR, Lblanche);
```

Une lumière est activée et désactivée (« interrupteur ») par :

```
glEnable(GL_LIGHT0)
glDisable(GL_LIGHT0)
```

d) Matériau d'un objet

Dans un modèle d'éclairage, le rendu d'un objet n'est plus défini par une couleur brute (`glColor3f(r, v, b )`, mais par un matériau avec des propriétés de réflexion de la lumière qui sont spécifiques (cuivre, argent, peinture brillante, …). Sous OpenGL, les polygones qui seront construits recevront des propriétés optiques définies pour les 4 composantes RVBA :

- l'émission (cas d'un objet lumineux),
- la diffusion,
- la réflexion spéculaire.

Pour cette dernière, on dispose d'un coefficient qui précise la taille du reflet et son intensité (étroit et intense, ou faible et étalé).

```
GLfloat Lnoire [4] = {0.0, 0.0, 0.0, 1.0};
GLfloat mat_diffuse [4] = {0.057, 0.441, 0.361, 1.0};
GLfloat mat_specular [4] = {0.1, 0.1, 0.5, 1.0};
GLfloat mat_shininess [1] = {50.0};

glMaterialfv (GL_FRONT_AND_BACK, GL_EMISSION, Lnoire);
glMaterialfv (GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
glMaterialfv (GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
```

En pratique, on définit des fonctions qui définissent un matériau donné en regroupant ces propriétés :

```
void bronze() ;
void argent() ;
void peintureMetallisée(int couleur[3]) ;
```

Ces fonctions sont appelées juste avant le dessin des surfaces (à la place de `glColor`) pour définir la « couleur » des facettes.

e) La normale aux sommets

Le dernier élément déterminant pour la perception visuelle d'une surface est son orientation vis à vis de la source lumineuse et du point d'observation. Sous OpenGL, cette orientation est évaluée à partir du vecteur normal qui doit être de longueur 1 : cette dernière contrainte peut être gérée directement par le programmeur ou par OpenGL (généralement plus coûteux) en activant :

```
glEnable (GL_NORMALIZE);
```

La normale doit être spécifiée avant le tracé d'un polygone au moyen de :

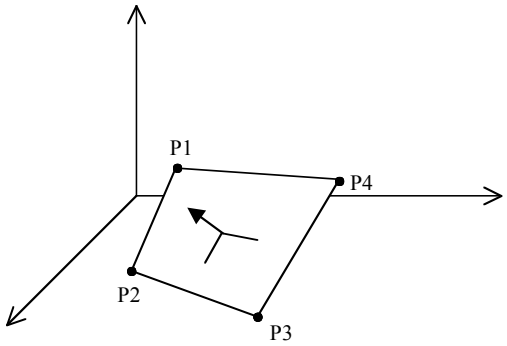
```
glNormal3f (x, y, z);
glNormal3fv(tab);
```

exemple :

```
glBegin(GL_TRIANGLES);
  glNormal3f(0., 0., 1.) ;
  glVertex3f(0., 0., 0.);
  glVertex3f(5., 0., 0.);
  glVertex3f(2.5, 5., 0.);
glEnd();
```

Si la normale n'est pas explicitement connue au moment de la programmation, il faut la calculer. Si on a pris soin de construire la liste des sommets dans l'ordre trigonométrique, les trois premiers sommets non alignés donnent deux vecteurs *u* et *v* dont le produit vectoriel donne la direction de la normale (face avant).

Rappel : $u \wedge v = [(y_u z_v - z_u y_v), -(x_u z_v - z_u x_v), (x_u y_v - y_u x_v)]$



On peut vouloir (ou non) éclairer les deux faces d'un polygones, il faut alors activer :

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE)`

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE)` *(valeur par défaut)*

Remarque : On se souvient que l'ordre dans lequel on dessine les sommets permet de définir la face avant et la face arrière d'un polygone (cf. chapitre sur les primitives). On pourrait penser que cette notion est redondante vis-à-vis de la normale. En fait, cela permet d'attribuer des propriétés spécifiques à chaque face (face pleine, face vide, couleur, matériau) alors que la normale ne servira qu'aux calculs d'éclairage comme le précise le paragraphe suivant.

f) Surfaces lisses ou « à facettes »

Bien que les surfaces « complexes » soient approximées par des données polygonales, OpenGL permet de donner à celles-ci un aspect « lisse » (modèle de Gouraud), même avec une discrétisation grossière. Une variable d'état permet de préciser si l'on souhaite un modèle de surface à facettes ou avec dégradé de couleur (plus lourd en calcul !) :

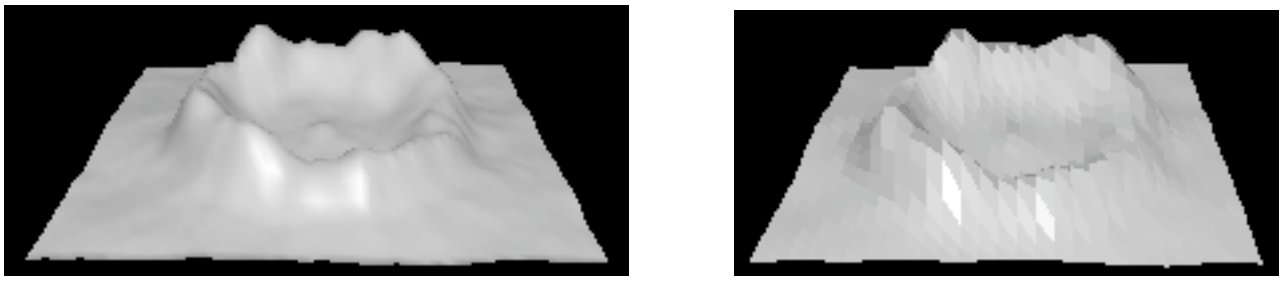
`glShadeModel(GL_FLAT)` *(plus rapide)*

`glShadeModel(GL_SMOOTH)` *(plus joli)*

En fait , les normales sont affectées aux sommets et non pas au polygone. Dans l'exemple ci-dessous, les trois sommets du triangles reçoivent la même normale `(0., 0., 1.)` : le triangle aura une couleur uniforme.

```
glBegin(GL_TRIANGLES);
/* définitions de la normale pour tous les sommets qui suivent : */
glNormal3f(0., 0., 1.) ;
glVertex3f(0., 0., 0.);
glVertex3f(5., 0., 0.);
glVertex3f(2.5, 5., 0.);
glEnd();
```

Pour obtenir un aspect non facetté, on affecte à chaque sommet la moyenne des normales des facettes voisines. OpenGL effectuera une extrapolation de la lumière réfléchie pour chaque pixel à partir des valeurs calculées aux sommets.



Eclairage d'un « cratère » avec et sans lissage

Schéma type du tracé d'une surface lissée à partir d'un maillage de points (`glFloat Point3D[3]`) :

```
glBegin(GL_TRIANGLES);
for (i=0 ; i<Longueur-1 ; i++)
 for (j=0 ; j<Largeur-1 ; j++)
 { /* 1 carreau = 2 triangles */
   glNormal3fv(Nsurface[i][j]);
   glVertex3fv(Surface[i][j]);
   glNormal3fv(Nsurface[i][j+1]);
   glVertex3fv(Surface[i][j+1]);
   glNormal3fv(Nsurface[i+1][j+1]);
   glVertex3fv(Surface[i+1][j+1]);

   glNormal3fv(Nsurface[i][j]);
   glVertex3fv(Surface[i][j]);
   glNormal3fv(Nsurface[i+1][j]);
   glVertex3fv(Surface[i+1][j]);
   glNormal3fv(Nsurface[i+1][j+1]);
   glVertex3fv(Surface[i+1][j+1]);
 }
}
glEnd();
```

Remarques :

- Lorsqu'une surface est représentée par un treillis de points, les sommets et les normales sont généralement rangés dans un tableau : cela facilite grandement l'évaluation des normales moyennes avant de dessiner la surface.
- Pour affiner le rendu, on pourra pondérer les normales de chaque facette par les secteurs angulaires respectifs.

VII. Images 2D

1. L'initialisation du 2D

Une image 2D est un tableau rectangulaire de pixels ayant chacun une « valeur » spécifique. Son affichage ne nécessitera pas de tampon de profondeur. Une initialisation courante se traduira dans la fonction *main* par un des deux appels ci-dessous :

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB) ;
```

ou

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

Bien que l'affichage se fasse en mode RGB (tampon image), on pourra disposer de plusieurs types de codage d'image en mémoire centrale. Nous nous limiterons ici aux :

- images RGB où un pixel est codé sur 3 octets consécutifs,
- images en niveaux de gris où un pixel est codé sur un octet.

On notera au passage que les données initiale d'une image peuvent avoir une dynamique bien supérieure au capacité d'affichage d'une carte graphique. Par exemple, une image IRM en imagerie médicale aura souvent des valeurs dans [-2048, 2047]. L'affichage d'une telle image demande un rééchelonnage préalable des amplitudes entre [0, 255] .

2. Matrice de projection

Il peut paraître étrange de définir une matrice de projection lorsque l'on manipule uniquement du 2D. Cette opération est pourtant nécessaire pour que la position d'un pixel image coïncide avec une coordonnée écran (entière). On utilise une matrice de projection spécifique au 2D :

```
gluOrtho2D(xmin, xmax, ymin, ymax)
```

Aussi, le cadrage d'une image aura la forme suivante :

```
void monCadrage(int large, int haut)
{ glViewport(0, 0, large, haut) ;
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  gluOrtho2D(0, large, 0, haut) ;
  glMatrixMode(GL_MODELVIEW) ;
}
```

3. Affichage d'une image 2D

Le prochain dessin d'une image (matrice rectangulaire de pixel) sera positionné dans la fenêtre en définissant l'angle inférieur gauche *(x0, y0)* :

```
glRasterPos2i(x0, y0) ;
```

OpenGL propose trois commandes de base pour manipuler des images :

- `glDrawPixels` qui recopie un tableau de pixels dans le tampon image (cf. `glRasterPos2i`),
- `glReadPixels` qui recopie une partie du tampon image dans un tableau,
- `glCopyPixels` qui recopie une zone à l'intérieur du tampon image (sans passer par la `CPU !`).

glDrawPixels(*largeur, hauteur, format, type, tableau*)
glReadPixels(*largeur, hauteur, format, type, tableau*)

`largeur, hauteur` : définissent la taille de l'image en terme de pixels,
`format` : format d'un pixel en mémoire centrale. On se limitera à **`GL_RGB`** ou **`GL_LUMINANCE`**,
`type` : on se limitera à **`GL_UNSIGNED_BYTE`** (un octet non signé),
`tableau` : adresse du tableau respectant le type précédemment définit.

Exemple :

```
#define hauteur 200
#define largeur 256
GLubyte imageRGB[hauteur][largeur][3] ;
GLubyte imageGRIS[hauteur][largeur] ;
```

affichage de `imageRGB`

```
glRasterPos2i(0, 0) ;
glDrawPixels(largeur, hauteur, GL_RGB, GL_UNSIGNED_BYTE, imageRGB) ;
glutSwapBuffers() ;
```

affichage de `imageGRIS`

```
glRasterPos2i(0, 0) ;
glDrawPixels(largeur, hauteur, GL_LUMINANCE, GL_UNSIGNED_BYTE, imageGRIS) ;
glutSwapBuffers() ;
```

glCopyPixels(*x0, y0, largeur, hauteur,* GL_COLOR)

La recopie s'effectue à partir de la trame active (cf. `glRasterPos2i`).
`x0 et y0` : Précise le coin inférieur gauche du rectangle (`largeur, hauteur`) de pixels à recopier.
Le dernier argument `p`précise le buffer sur lequel l'opération est effectuée. Nous nous limiterons ici à **`GL_COLOR`**.

4. Image et processeur…

a. Alignement des octets

Les processeurs sont performants pour manipuler des mots machines : Pentium et G4 sont des processeurs 32 bits « grand public », mais nul doute sur l'arrivée prochaine des 64 bits…

Les fonctions `glDrawPixels` et `glReadPixels` tiennent compte de l'architecture du processeur et effectuent, par défaut, les transferts par paquets d'octets correspondants au mot machine. En particulier, elles démarrent chaque ligne image à la première adresse multiple d'un mot machine qui suit la fin de la précédente ligne. En conséquence, la largeur d'une image doit occuper un nombre

d'octets multiple d'un mot machine, au risque d'obtenir un affichage avec une déformation latérale de l'image parce que le début de chaque ligne aura été artificiellement décalé.

Le contrôle des transferts se fait avec :

`glPixelStorei(GL_UNPACK_ALIGNMENT, taille)` pour `glDrawPixels`
`glPixelStorei(GL_PACK_ALIGNMENT, taille)` pour `glReadPixels`

où `taille` définit le nombre d'octets dans un paquet (1, 2, 4 ou 8).

Une solution sûre mais peu efficace est de forcer la taille des paquets à 1 pour qu'une ligne image contienne nécessairement un nombre entier de paquets d'octets.

Une autre solution consiste à définir une matrice image en mémoire dont la taille des lignes est un multiple de 4 (taille d'un mot machine pour les processeurs actuels), quitte à ne pas remplir les derniers octets de chaque ligne.

b. Ordre des octets

Là encore, la diversité existe et on constate qu'un mot machine peut ordonner ses octets dans un sens ou dans l'autre. Par exemple, l'octet de poids fort pour un G4 se trouve à la place de l'octet de poids faible pour un Pentium et réciproquement. L'exploitation sur une machine d'une image construite sur une autre machine peut demander au préalable quelques opérations de permutation.

5. Primitives graphiques 2D

Dans certaines applications graphiques, on peut souhaiter effectuer un tracé par dessus une image que l'on vient d'afficher. Par exemple, on peut tracer en rouge une route sur une image satellitaire pour la mettre en évidence.

Dans ce cas, on affichera l'image en premier et on utilisera ensuite les primitives de tracé 2D avec le suffixe 2i, principalement **`glVertex2i(x, y)`** .

Il sera aussi parfois plus commode de définir les couleurs de tracé en entier :
`glColor3ub(rouge, vert, bleu)`

Annexe : Couleurs et niveaux de gris

Sujet vaste et complexe : domaines de la physique, de la physiologie, de la psychologie, de l'art…

1. La lumière

1.1 Généralités

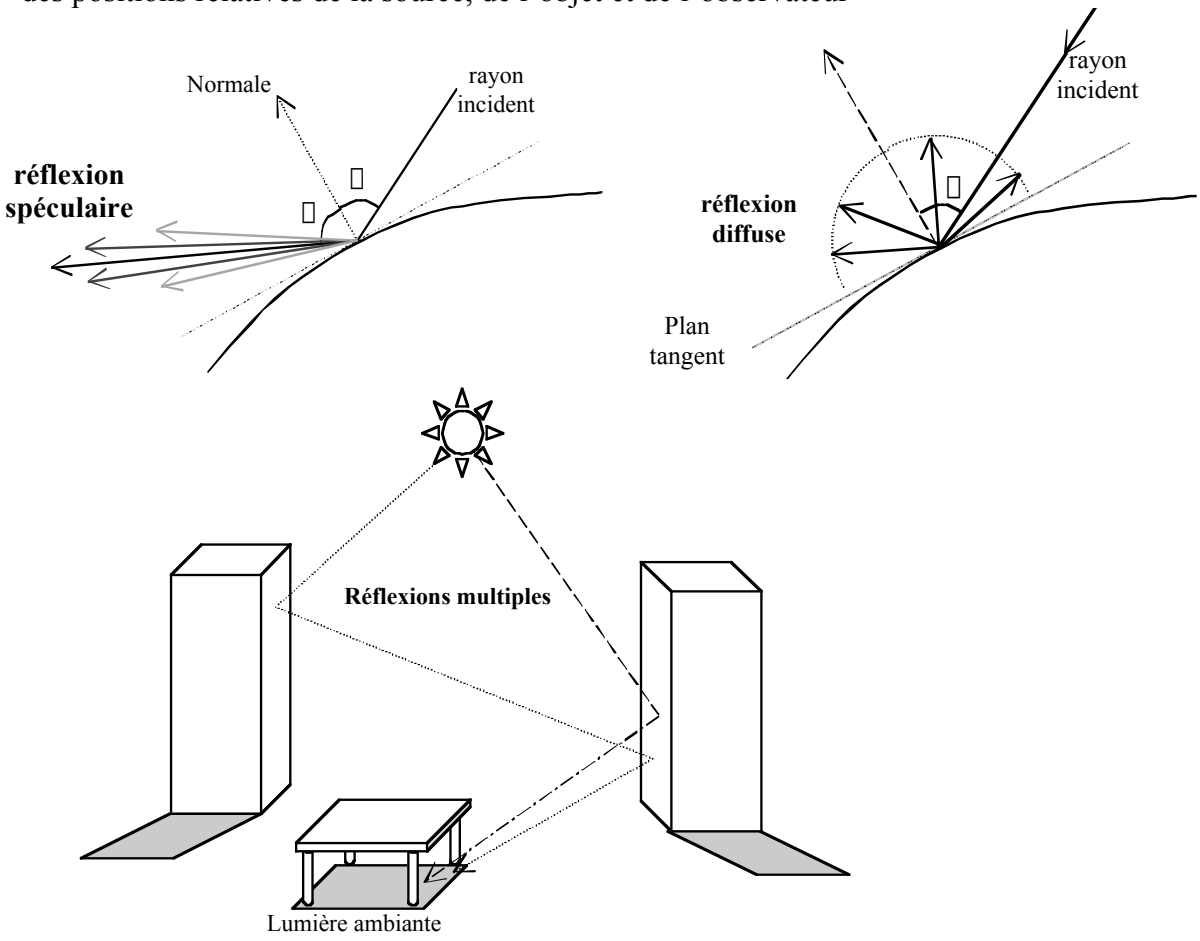
La lumière est composée de fréquences élémentaires (séparable par un prisme). La lumière visible se situe entre 400 et 700nm. Elle est issue :

- soit d'une source active (lampe, soleil, …),
- soit d'une source passive qui restitue une partie de la lumière reçue.

Un objet est miroir imparfait qui ne réfléchit qu'une partie de la lumière reçue :
- un objet blanc éclairé par de la lumière verte apparaît vert.
- un objet rouge apparaîtra noir sous une lumière verte.

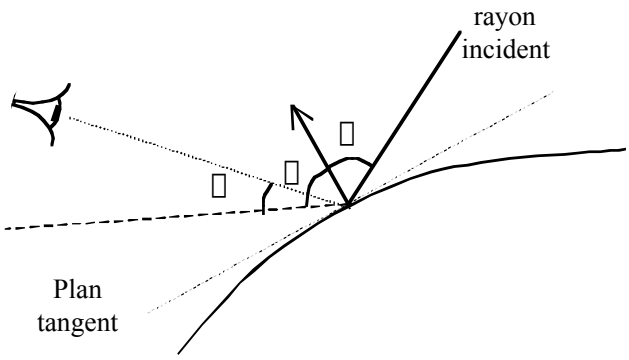
La perception de la couleur d'un objet dépend :

- de la distribution des longueurs d'onde de la source lumineuse,
- des caractéristiques physiques de l'objet (matériau, état de surface) :
- des positions relatives de la source, de l'objet et de l'observateur



1.2 Modèle d'éclairage simplifié

La lumière réfléchie par une surface et reçue par un observateur est décomposée en trois quantité :
- intensité ambiante,
- réflexion diffuse,
- réflexion spéculaire.



$L = L_a + k_d \cos \theta + k_s \cos^n \alpha$

2. Système achromatique

Littéralement « sans couleur ».

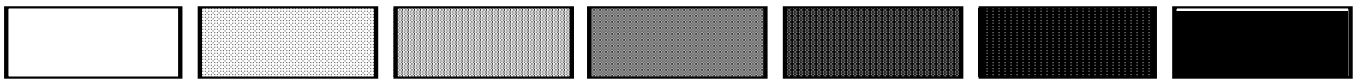
La seule information véhiculée : quantité de lumière
- **intensité** ou **luminance** dans un cadre physique,
- **luminosité** d'un point de vue perceptif (cadre psychologique).

Notre système visuel permet d'analyser une scène en niveaux de gris

Notre perception de la lumière :
- incapable de faire une mesure ponctuelle de la lumière
- intègre une quantité de lumière (surface et durée)
- sensible à une échelle logarithmique de l'énergie lumineuse (100W-50W ≠ 150W-100W).

Nuances de gris :
- niveaux d'énergie compris entre le **noir** (absence de lumière)
- et le **blanc** (saturation de lumière).
- en pratique : 256 niveaux de gris (un octet)

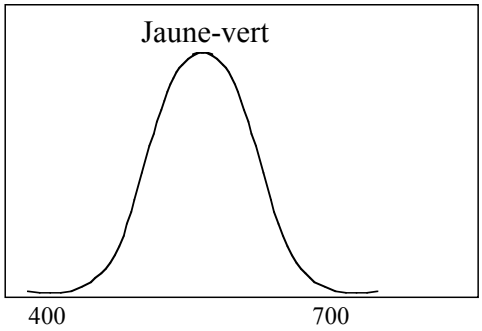
Deux types de matériels concernés :
- matériels pouvant représenter localement toute une échelle de luminance
- dispositifs bicolores (imprimante, …). Possibilité d'approximation par demi-ton.



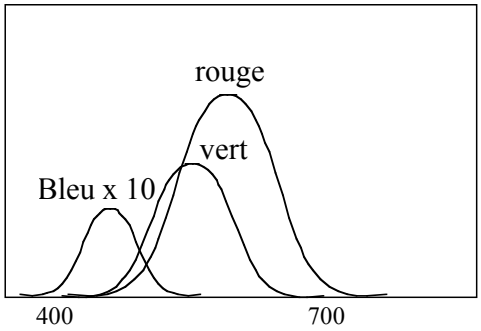
3. La couleur

3.1 Psychophysique

Cellules spécialisées de la rétine : les cellules à cône.



Sensibilté de l'oeil



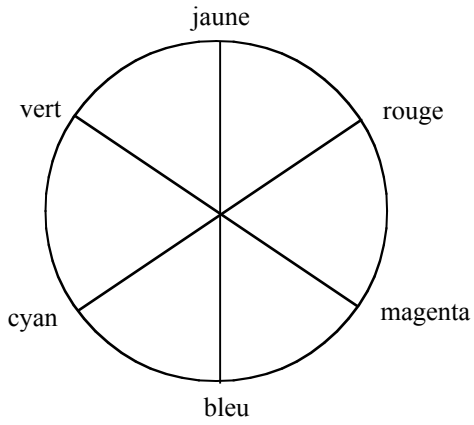
Sensibilté des cellules à cônes

C'est le couple œil-cerveau qui interprète un phénomène physique qui procure la sensation de couleur.

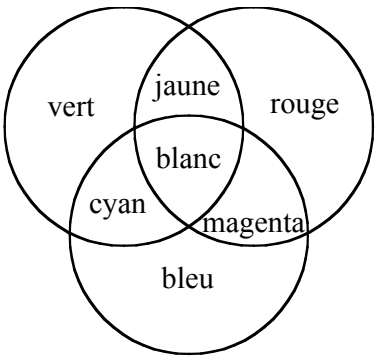
3.2 Les systèmes soustractifs et additifs

Construction d'une **couleur perçue** à partir de **trois couleurs primaires**.

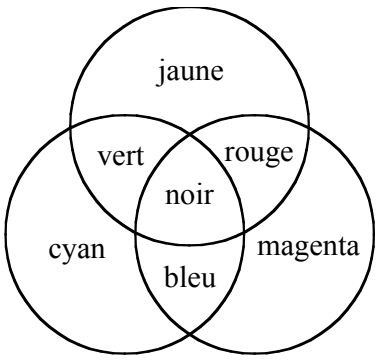
- ➔ mélange **additif** : **le rouge, le vert et le bleu** (écran cathodique, projecteurs d'un spectacle).
- ➔ mélange **soustractif** : **le cyan, le magenta et le jaune** (peinture, imprimantes…).



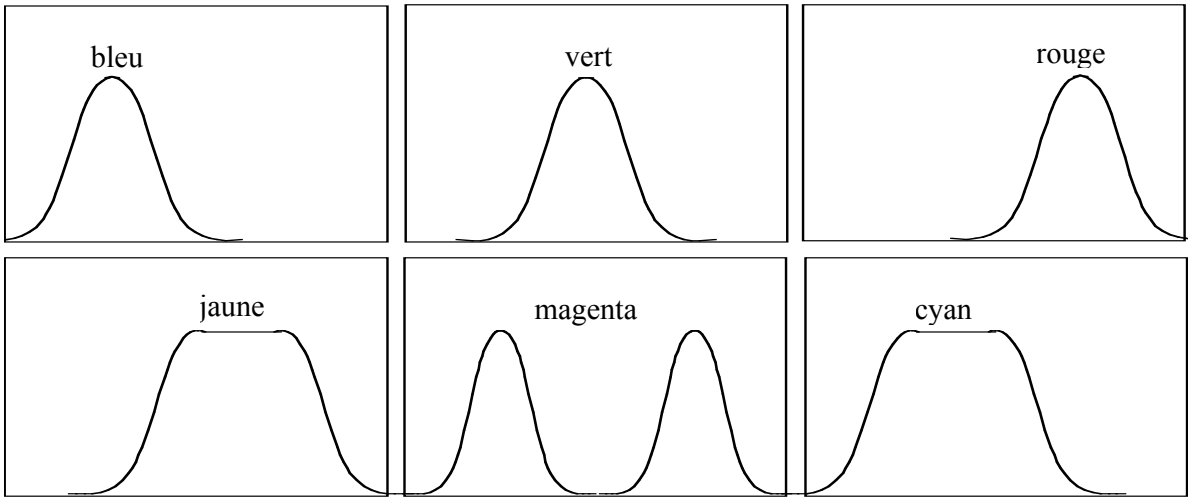
Catégorisation des couleurs



Système additif



Système soustractif



Spectres des couleurs de base

3.3 Modèles de couleurs

Il existe trois modèles de couleurs orientés vers les matériels :

Modèle RVB (rouge, vert, bleu)
- primaires additives
- l'oeil intègre la quantité de lumière
- l'information couleur est codée par 3 scalaires

Modèle CMJ (Cyan, magenta et jaune)
- filtres sur de la lumière blanche, ce sont des primaires soustractives.
- employé pour les imprimantes à jet d'encre.

On passe facilement d'un modèle RVB à un modèle CMJ
(R, V, B) = (255, 255, 255) – (C, M, J)

Modèle YIQ
- TV couleur aux USA,
- Y représente la luminance (seule composante pour les postes noir et blanc)
- La projection de RVB dans YIQ est donnée par :

$$\begin{bmatrix} \mathbf{Y} \\ I \\ Q \end{bmatrix} = \begin{bmatrix} \mathbf{0.299} & \mathbf{0.587} & \mathbf{0.114} \\ 0.596 & -0.275 & -0.321 \\ 0.212 & 0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ V \\ B \end{bmatrix}$$

II. La librairie GLUT

Plan :
1. Structure d'une application GLUT
2. Initialisation d'une session GLUT
3. La boucle de traitement des événements
4. Gestion des fenêtres
5. Gestion de menus
6. Inscription des fonctions de rappel
7. Quelques variables d'état de GLUT
8. Rendu des polices de caractères

OpenGL a été conçu pour être indépendant du gestionnaire de fenêtres qui est intimement lié au système d'exploitation. Il existe toutefois un système de fenêtrage « élémentaire » qui permet de développer des applications graphiques dans un cadre simple tout en garantissant une très bonne portabilité sur de très nombreuses plate-formes : openGL Utility Toolkit (GLUT).

Les fonctionnalités de cette bibliothèque permettent principalement de :
- créer et gérer plusieurs fenêtres d'affichage,
- gérer les interruptions (click souris, touches clavier, …),
- disposer de menus déroulant,
- connaître la valeur d'un certain nombre de paramètres systèmes,

Quelques fonctions supplémentaires permettent de créer simplement un certain nombre d'objets 3D (cube, sphère, tore, …).

Cette bibliothèque s'enrichie régulièrement d'outils simples et pratiques (on trouve maintenant sur les sites OpenGL des boutons, des affichages de répertoires, …) sans devenir un « monstre » dont la maîtrise demande une longue pratique.

La philosophie générale de ce système de fenêtrage est basée sur la « programmation événementielle » (ce que l'on pourra regretter …), ce qui impose une structuration assez particulière de l'application.

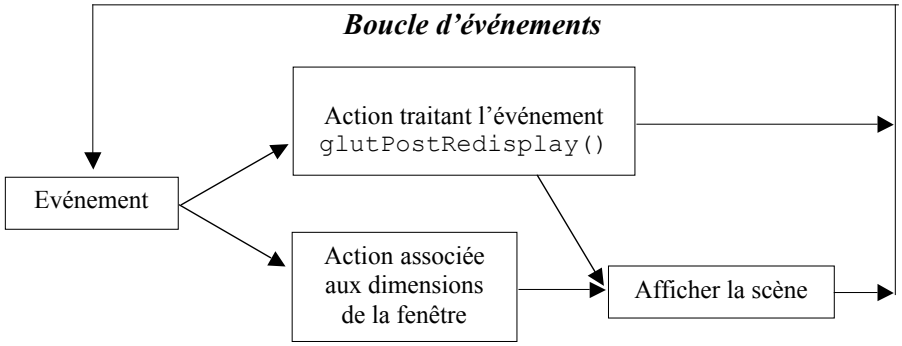
10. Structure d'une application GLUT

Une application GLUT lance une session graphique qui ne sera plus contrôlée que par des interruptions (click souris, touche clavier, …). On trouve dans le « main » les actions suivantes :
- initialisation du fenêtrage,
- désignation de la fonction d'affichage (1) dans la fenêtre courante,
- désignation de la fonction (2) déclenchée par un redimensionnant la fenêtre courante,
- association d'une fonction (3) à chaque type d'interruption,
- la boucle d'événements.

Avec les remarques suivantes :
- Toute opération de tracé est interdite en dehors de la fonction déclarée pour cette tâche (1).
- La boucle d'événement est la dernière action du programme principal et échappe totalement au contrôle du programmeur. Elle prend la main de façon définitive (jusqu'à la fin l'application) : elle réactualise régulièrement l'affichage d'une part et capte d'autre part les interruptions pour déclencher les procédures associées (3).
- C'est le système qui déclenche la fonction d'affichage (1),

- Le programmeur peut demander au système l'exécution de la fonction d'affichage au moyen de l'instruction `glutPostRedisplay()`

Cette gestion indépendante des différents processus impose l'utilisation de variables globales d'état pour contrôler l'affichage en fonction des interruptions (dialogue entre la fonction d'affichage et les fonctions traitants les interruptions).



Exemple simplifié de la structuration d'un programme GLUT :

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

void afficheMaScene(void)
{ « effacer l'écran »
  positionner la caméra
  construction (tracé) de la scène
  glutSwapBuffers(); /* glFlush() */
}

void monCadrage(int largeur, int hauteur)
{ redéfinition du cadre d'affichage après redimensionnement de la fenêtre
  definition de la projection 3D->2D
}

void maFctClavier (unsigned char key, int x, int y)
{ modification du contexte d'affichage sur un événement clavier
  glutPostRedisplay() ;
}

void maFctTouchesSpeciales(unsigned char key, int x, int y)
{ action déclenchée sur une touche F1, …, F10, flèches
  glutPostRedisplay() ;
}

void maFctSouris (int bouton, int etat, int x, int y)
{ modification du contexte d'affichage sur un événement souris
  glutPostRedisplay() ;
}

void monInitScène()
{ initialisation eventuelle de parametres propres à l'application (eclairages, …)
}

int main (int argc, char **argv)
{     /* initialisation d'une session GLUT */
      glutInit(argc, argv); /* initialise la bibliothèque GLUT */
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(500, 500);
      glutInitWindowPosition(100, 100);
```

```
        glutCreateWindow(argv [0]);
        /* initialisation éventuelle de parametres
        monInitScène();
        /* ftcs définissant la scène3D et sa projection */
        glutDisplayFunc(afficheMaScene);          /* (1) */
        glutReshapeFunc(monCadrage);              /* (2) */
        /* ftcs liees aux iterruptions */
        glutKeyboardFunc(maFctClavier);           /* (3) */
        glutSpecialFunc(maFctTouchesSpeciales);   /* (3) */
        glutMouseFunc(maFctSouris);               /* (3) */
        /* boucle d'événements */
        glutMainLoop();
        return 0;
}
```

11. Initialisation d'une session GLUT

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

La fonction **glutInit** initialise la bibliothèque *GLUT* et négocie une session avec le système de fenêtrage. Elle traite également les lignes de commandes qui sont propres à chaque système de fenêtrage.

Paramètres pour le système X

Les paramètres de la ligne de commande qui sont compris par la bibliothèque *GLUT* sont par exemple:

- **display** *DISPLAY* Spécifie l'adresse du serveur X auquel se connecter. Si ce n'est spécifié, la variable d'environnement est utilisée.
- **geometry** *WxH+X+Y* Détermine la position de la fenêtre sur l'écran. Le paramètre de geometry doit être formaté selon la spécification standard de X.
- **gldebug** Après le traitement des fonctions de rappel ou des événements, vérifier s'il y a des erreurs *d'OpenGL* en appelant **glGetError**. S'il y a une erreur, imprimer un avertissement obtenu par la fonction **gluErrorString**.

void glutInitWindowSize (int width, int height);

void glutInitWindowPosition (int x, int y);

Les fonctions **glutInitWindowSize** et **glutInitWindowPosition** permettent de créer une fenêtre, de la positionner sur l'écran et d'en spécifier la taille.

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

void glutInitDisplayMode (unsigned int mode);

Cette fonction spécifie le mode d'affichage de la fenêtre. Le mode d'affichage est utilisé pour créer les fenêtres et les sous-fenêtres. Le mode GLUT_RGBA permet d'obtenir une

fenêtre utilisant le modèle de couleur RGB avec une composante de transparence. C'est le mode de base de la plupart des applications.

mode Mode d'affichage qui est en général une opération or bit à bit de masque de bits. Les valeurs permises sont :

- GLUT_RGBA Masque de bits pour choisir une fenêtre en mode RGBA. C'est la valeur par défaut si GLUT_RGBA ou GLUT_INDEX ne sont spécifiés.
- GLUT_RGB Un alias de GLUT_RGBA.
- GLUT_INDEX Masque de bits pour choisir une fenêtre en mode index de couleur. Ceci l'emporte si GLUT_RGBA est spécifié.
- GLUT_SINGLE Masque de bits pour spécifier un tampon simple pour la fenêtre. Ceci est la valeur par défaut.
- GLUT_DOUBLE Masque de bit pour spécifier une fenêtre avec un double tampon. Cette valeur l'emporte sur GLUT_SINGLE.
- GLUT_RGBA Masque de bits pour choisir une fenêtre avec une composante alpha pour le tampon de couleur.
- GLUT_DEPTH Masque de bits pour choisir une fenêtre avec un tampon de profondeur.
- ...

12. La boucle de traitement des événements

void glutMainLoop (void);

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

13. Gestion des fenêtres

int glutCreateWindow (char * name);

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

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

void glutSetWindow (int win);

Cette fonction établit que la fenêtre identifiée par **win** devient la fenêtre courante.

int glutGetWindow (void);

Cette fonction retourne le numéro de la fenêtre courante. Si la fenêtre courante a été détruite, alors le numéro retourné est 0.

void glutDestroyWindow (int win);

glutDestroyWindow détruit la fenêtre identifiée par le paramètre **win**. Elle détruit également le contexte OpenGL associée à la fenêtre. Si win identifie la fenêtre courante, alors la fenêtre courante devient invalide (**glutGetWindow** retourne la valeur 0).

void glutPostRedisplay (void);

Cette fonction indique que la fenêtre courante doit être réaffiché. Lors de la prochaine itération dans la boucle principale de **glutMainLoop**, la fonction de rappel d'affichage est appelée. Plusieurs appels à la fonction **glutPostRedisplay** n'engendrent qu'un seul rafraîchissement. Logiquement, une fenêtre endommagée est marquée comme devant être rafraîchie, ce qui est équivalent à faire appel la fonction **glutPostRedisplay**.

Cette fonction est principalement employée dans les procédures attachées à une interruption. La modification du contexte demande généralement une réactualisation de l'affichage.

void glutSwapBuffers (void);

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

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

void glutPositionWindow (int x, int y);

Demande un changement de position de la fenêtre courante. Les coordonnées x et y sont des décalages par rapport à l'origine de l'écran. Le changement de position n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne dans la boucle principale

Pour une fenêtre de base, le système de fenêtrage est libre d'appliquer face à la requête sa propre politique pour le positionnement de la fenêtre.

glutPositionWindow désactive le mode plein écran s'il est activé.

void glutReshapeWindow (int width, int height);

Demande un changement de dimensions de la fenêtre courante. Les paramètres **width** et **height** sont les nouvelles dimensions de la fenêtre et doivent être des entiers positifs. Le changement de dimension n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne à la boucle principale.

Pour une fenêtre de base, le système de fenêtrage est libre d'appliquer face à la requête sa propre politique pour le dimensionnement de la fenêtre.

glutReshapeWindow désactive le mode plein écran s'il est activé.

void glutFullScreen (void);

Demande que la fenêtre courante soit en plein écran. La sémantique de plein écran peut varier d'un système de fenêtrage à l'autre. Le but est d'obtenir la fenêtre la plus grande possible en la libérant des bordures et des barres de titre. Les dimensions de la fenêtre ne correspondent pas nécessairement aux dimensions de l'écran. Le changement de dimension n'est pas immédiatement effectué, mais le changement est effectué lorsque l'application retourne à la boucle principale.

Les appels aux fonctions **glutReshapeWindow** et **glutPositionWindow** désactivent le mode plein écran.

void glutPopWindow (void);

void glutPushWindow (void);

La fonction **glutShowWindow** affiche la fenêtre courante (elle pourrait ne pas être visible si elle est occultée par une autre fenêtre). La fonction **glutHideWindow** cache la fenêtre courante. Les actions de cacher ou afficher une fenêtre ne sont pas effectués immédiatement. Les requêtes sont conservées pour exécution future lors du retour à la boucle principale des événements. Les effets d'afficher ou masquer une fenêtre dépendent de la politique d'affichage du système de fenêtrage.

void glutSetWindowTitle (char *name);

Cette fonction s'applique à la fenêtre. Le nom d'une fenêtre est établi lorsque de la création de la fenêtre par la fonction **glutCreateWindow**. Par la suite, le nom d'une fenêtre peut être changé respectivement par un appel à la fonction **glutSetWindowTitle**.

void glutSetCursor (int cursor);

Change l'apparence du curseur pour la fenêtre courante. Valeur du curseur :

- **GLUT_CURSOR_RIGHT_ARROW** Flèche pointant vers le haut et la droite.
- **GLUT_CURSOR_LEFT_ARROW** Flèche pointant vers le haut et la gauche.
- **GLUT_CURSOR_INFO** Main directionnelle.
- **GLUT_CURSOR_DESTROY** Crâne et os (tête de mort).
- **GLUT_CURSOR_HELP** Point d'interrogation.
- **GLUT_CURSOR_CYCLE** Flèche tournant en cercle.
- **GLUT_CURSOR_SPRAY** Aérosol.
- **GLUT_CURSOR_WAIT** Montre bracelet.

- **GLUT_CURSOR_TEXT** Point d'insertion pour le texte.
- **GLUT_CURSOR_CROSSHAIR** Croix.
- **GLUT_CURSOR_UP_DOWN** Curseur bidirectionnel pointant vers le haut et le bas.
- **GLUT_CURSOR_LEFT_RIGHT** Curseur bidirectionnel pointant vers la gauche et la droite.
- **GLUT_CURSOR_TOP_SIDE** Flèche pointant vers le côté supérieur.
- **GLUT_CURSOR_BOTTOM_SIDE** Flèche pointant vers le côté inférieur.
- **GLUT_CURSOR_LEFT_SIDE** Flèche pointant vers le côté gauche.
- **GLUT_CURSOR_RIGHT_SIDE** Flèche pointant vers le côté droit.
- **GLUT_CURSOR_TOP_LEFT_CORNER** Flèche pointant vers le coin supérieur gauche.
- **GLUT_CURSOR_TOP_RIGHT_CORNER** Flèche pointant vers le coin supérieur droit.
- **GLUT_CURSOR_BOTTOM_LEFT_CORNER** Flèche vers le coin inférieur gauche.
- **GLUT_CURSOR_BOTTOM_RIGHT_CORNER** Flèche vers le coin inférieur droit .
- **GLUT_CURSOR_FULL_CROSSHAIR** Grande croix.
- **GLUT_CURSOR_NONE** Curseur invisible.

14. Gestion de menus

La bibliothèque *GLUT* supporte des menus déroulants en cascades. La fonctionnalité est simple et minimale. La bibliothèque GLUT n'a pas la même fonctionnalité que X-Windows ou WindowsXX; mais elle a l'avantage d'être portable sur plusieurs plates-formes. Il est illégal de créer ou éliminer des menus, ou de changer, ajouter ou retirer des éléments d'un menu pendant qu'il est en cours d'utilisation.

int glutCreateMenu (void (*func) (int value));

La fonction **glutCreateMenu** crée un nouveau menu déroulant et retourne un entier identifiant ce menu. La plage du numéro de menu commence à 1. Implicitement, le menu courant correspond au nouveau menu créé. L'identificateur de menu peut être utilisé par la suite par la fonction **glutSetMenu**. Lorsque la fonction de rappel est appelée parce qu'un élément du menu a été sélectionné, la valeur du menu courant devient le menu sélectionné. La valeur de la fonction de rappel correspond à l'élément du menu sélectionné.

void glutSetMenu (int menu);

int glutGetMenu (void);

La fonction **glutSetMenu** permet d'établir le menu courant; la fonction **glutGetMenu** retourne la valeur du menu courant. Si le menu n'existe pas, ou si le menu courant précédent a été détruit, **glutGetMenu** retourne la valeur 0.

void glutDestroyMenu (int menu);

La fonction **glutDestroyMenu** détruit le menu identifié par menu. Si menu identifie le menu courant, la valeur du menu courant devient invalide ou 0.

void glutAddMenuEntry (char * name, int value);

La fonction **glutAddMenuEntry** ajoute un élément au bas du menu courant. La chaîne de caractères est affichée dans le menu déroulant. Si un élément du menu est sélectionné par un utilisateur, la valeur **value** est la valeur transmise à la fonction de rappel correspondant au menu courant.

void glutAddSubMenu (char * name, int menu);

La fonction **glutAddSubMenu** ajoute un sous-menu pour cet élément de menu. Lors de la sélection de cet élément, un sous-menu menu est ouvert en cascade pour le menu courant. Un élément de ce sous-menu peut être par la suite sélectionné.

void glutChangeToMenuEntry (int entry, char *name, int value);

La fonction **glutChangeToMenuEntry** permet de changer un élément du menu courant en une entrée du menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être changé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et **glutGet(GLUT_MENU_NUM_ITEMS)** inclusivement. La chaîne de caractères **name** est affichée pour l'entrée du menu modifiée. Si un élément du menu est sélectionné par un utilisateur, la valeur value est la valeur transmise à la fonction de rappel correspondant au menu courant.

void glutChangeToSubMenu (int entry, char *name, int menu);

La fonction **glutChangeToSubMenu** permet de changer l'élément du menu du menu courant en un élément déclenchant un sous-menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être changé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et **glutGet(GLUT_MENU_NUM_ITEMS)** inclusivement. L'identificateur **menu** nomme le menu qui est ouvert en cascade lorsque cet élément est sélectionné.

void glutRemoveMenuItem (int entry);

La fonction **glutRemoveMenuItem** élimine un élément du menu. Le paramètre **entry** indique quel est l'élément du menu qui doit être éliminé; 1 correspond à l'élément du haut et **entry** doit être entre **1** et **glutGet(GLUT_MENU_NUM_ITEMS)** inclusivement. Les éléments du menu en dessous sont renumérotés.

void glutAttachMenu (int button);

void glutDetachMenu (int button);

Ces fonctions attachent ou détachent respectivement le menu courant à un des boutons de la souris.

15. Inscription des fonctions de rappel

La bibliothèque *GLUT* supporte un certain nombre de fonctions de rappel dont le but est d'attacher une réponse (une fonction programmeur) à différents types d'événement. Il y a trois types de fonctions de rappel:

- **fenêtre** : les fonctions de rappel concernant les fenêtres indiquent quand réafficher ou redimensionner la fenêtre, quand la visibilité de la fenêtre change et quand une entrée est disponible pour la fenêtre;
- **menu** : une fonction de rappel concernant un menu indique la fonction à rappeler lorsqu'un élément du menu est sélectionné;
- **globale** : les fonctions de rappel globales gère le temps et l'utilisation des menus

Les fonctions de rappel attachées à des événements d'entrée doivent être traitées pour les fenêtres pour lesquelles l'événement a été effectué.

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

La fonction **glutDisplayFunc** établit la fonction de rappel pour la fenêtre courante. Quand *GLUT* détermine que la fenêtre doit être réafficher, la fonction de rappel d'affichage est appelée.

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

Chaque fenêtre doit avoir une fonction de rappel inscrite. Une erreur fatale se produit si une tentative d'affichage d'une fenêtre est effectuée sans qu'une fonction de rappel n'ait été inscrite. C'est donc une erreur de faire appel à la fonction glutDisplayFunc avec le paramètre NULL.

void glutReshapeFunc (void (*func) (int width, int height));

La fonction **glutReshapeFunc** établit la fonction de rappel de redimensionnement de la fenêtre courante. La fonction de rappel de redimensionnement est déclenchée lorsque la fenêtre est refaçonnée. La fonction de rappel est aussi déclenchée immédiatement avant le premier appel à la fonction de rappel d'affichage après la création de la fenêtre. Les paramètres **width** et **height** de la fonction de rappel de redimensionnement spécifient les dimensions en pixels de la nouvelle fenêtre.

Si aucune fonction de rappel de redimensionnement n'est inscrite ou qu'on fait appel à la fonction **glutReshapeFunc** avec la valeur **NULL**, la fonction de rappel de redimensionnement implicite est appelée. Cette fonction implicite fait simplement appel à la fonction **glViewport(0, 0, width, height)** pour le plan normal de la fenêtre courante.

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

La fonction **glutKeyboardFunc** établit la fonction de rappel du clavier pour la fenêtre courante. Lorsqu'un utilisateur tape au clavier (dans une fenêtre), chaque touche génère un appel à la fonction de rappel du clavier. Le paramètre **key** est le code ASCII de la touche. L'état d'une touche modificatrice telle majuscule [**Shift**] ne peut être connu directement; son effet se reflète cependant sur le caractère ASCII.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement de l'événement (frappe d'une touche).

Lors de la création d'une nouvelle fenêtre, aucun fonction de rappel du clavier n'est enregistrée implicitement et les touches du clavier sont ignorées. La valeur NULL pour la fonction **glutKeyboardFunc** désactive la génération de fonction de rappel pour le clavier.

Pendant le traitement d'un événement clavier, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (par exemple, la touche majuscule ou **Ctrl** ou **Alt**) lors du déclenchement d'un événement au clavier. Il faut se référer à la fonction **glutSpecialFunc** pour le traitement de caractères non-ASCII, par exemple les touches de fonction ou les touches fléchées.

void glutMouseFunc (void (*func) (int button, int state, int x, int y);

La fonction **glutMouseFunc** établit la fonction de rappel de la souris pour la fenêtre courante. Lorsqu'un utilisateur appuie ou relâche un des boutons de la souris, chaque action (appui ou relâchement d'un bouton) engendre un appel à la fonction de rappel de la souris.

Le paramètre **button** peut prendre les valeurs : **GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON**, ou **GLUT_RIGHT_BUTTON**.

Le paramètre **state** indique si la fonction de rappel a été rappelée suite à l'appui ou au relâchement d'un bouton de la souris et les valeurs permises sont : **GLUT_UP** et **GLUT_DOWN**.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement de l'événement.

Si un menu est attaché à un bouton de la souris, aucun rappel de la fonction de la souris n'est effectué pour ce bouton.

Pendant le traitement d'un événement de la souris, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (**Shift** ou **Ctrl** ou **Alt**).

La valeur **NULL** pour la fonction **glutMouseFunc** désactive la génération de fonction de rappel pour la souris.

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

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

Les fonctions **glutMotionFunc** et **glutPassiveMotionFunc** établissent les fonctions de rappel pour la fenêtre courante pour un déplacement de la souris. La fonction de rappel spécifiée par **glutMotionFunc** est appelée lors du déplacement de la souris avec un ou

plusieurs boutons appuyés. La fonction de rappel spécifiée par **glutPassiveMotionFunc** est appelée lors du déplacement de la souris dans la fenêtre avec aucun bouton appuyé.

Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels.

La valeur **NULL** pour les fonctions **glutMotionFunc** ou **glutPassiveMotionFunc** désactive la génération de fonction de rappel lors du déplacement de la souris.

void glutVisibilityFunc (void (*func) (int state));

La fonction **glutVisibilityFunc** établit la fonction de rappel de visibilité pour la fenêtre courante . Cette fonction de rappel est appelée lorsque la visibilité de la fenêtre change. Le paramètre **state** peut prendre les valeurs **GLUT_VISIBLE** ou **GLUT_NOT_VISIBLE** selon la visibilité de la fenêtre. L'état **GLUT_VISIBLE** est valable pour une fenêtre partiellement ou totalement visible, i.e. à moins que la visibilité ne change, aucun rafraîchissement de la fenêtre n'est effectué. **GLUT_NOT_VISIBLE** signifie donc qu'aucun pixel de la fenêtre n'est visible.

La valeur **NULL** pour les fonctions glutVisibilityFunc désactive la fonction de rappel de visibilité. Si la fonction de rappel de visibilité est désactivée, l'état de la fenêtre devient indéfinie. Tout changement à la visibilité de la fenêtre est rapporté. Donc la réactivation de la fonction de rappel de visibilité garantit qu'un changement de visibilité est rapporté.

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

La fonction **glutSpecialFunc** établit la fonction de rappel du clavier pour les caractères non-ASCII pour la fenêtre courante. Des caractères non-ASCII sont générés du clavier lorsqu'une des touches de fonction (F1 à F12) ou une des touches de direction est utilisée. Le paramètre **key** est une constante correspondant à une touche spéciale (**GLUT_KEY_***). Les paramètres **x** et **y** indiquent les coordonnées relatives de la souris par rapport à la fenêtre en pixels lors du déclenchement d'un événement clavier. Pendant le traitement d'un événement du clavier, on peut faire appel à la fonction **glutGetModifiers** pour connaître l'état des touches modificatrices (**Shift ou Ctrl ou Alt**).

La valeur **NULL** pour la fonction **glutSpecialFunc** désactive la génération de fonction de rappel pour le clavier (touches spéciales).

Les valeurs correspondant aux touches spéciales sont les suivantes:

- **GLUT_KEY_F1** Touche F1.
- **GLUT_KEY_F2** Touche F2.
- **GLUT_KEY_F3** Touche F3.
- **GLUT_KEY_F4** Touche F4.
- **...**
- **GLUT_KEY_LEFT** Touche fléchée vers la gauche.
- **GLUT_KEY_UP** Touche fléchée vers le haut.
- **GLUT_KEY_RIGHT** Touche fléchée vers la droite.

- **GLUT_KEY_DOWN** Touche fléchée vers le bas.
- **GLUT_KEY_PAGE_UP** Touche page précédente (Page up).
- **GLUT_KEY_PAGE_DOWN** Touche page suivante (Page down).
- **GLUT_KEY_HOME** Touche Home.
- **GLUT_KEY_END** Touche End.
- **GLUT_KEY_INSERT** Touche d'insertion (ins)

Il est à noter que les touches d'échappement [Escape], de recul [Backspace] et d'élimination [delete] génèrent des caractères ASCII. Voici quelques valeurs importantes de caractères ASCII:

Backspace	8
Tabulation	9
Return	13
Escape	27
Delete	127

void glutMenuStatusFunc (void (*func) (int status, int x, int y));

La fonction **glutMenuStatusFunc** établit une fonction de rappel pour l'état du menu de sorte qu'une application utilisant *GLUT* puisse déterminer si le menu est en utilisation ou non. Quand une fonction de rappel d'état du menu est inscrite, un appel est effectué avec la valeur **GLUT_MENU_IN_USE** pour le paramètre **status** quand les menus déroulants sont utilisés; la valeur **GLUT_MENU_NOT_IN_USE** pour le paramètre **status** est utilisée lorsque les menus ne sont pas en utilisation. Les paramètres **x** et **y** indique la position, en coordonnées de fenêtre, lorsque le menu a été déclenché par un bouton de la souris. Le paramètre **func** représente la fonction de rappel.

Les autres fonctions de rappel (excepté les fonctions de rappel pour le déplacement de la souris) continuent à être actives pendant l'utilisation des menus, de sorte que la fonction de rappel pour l'état du menu peut suspendre une animation ou d'autres tâches lorsque le menu est en cours d'utilisation. Une cascade de sous-menus pour un menu initial déroulant ne génère pas d'appel à la fonction de rappel pour l'état du menu. Il y a une seule fonction de rappel pour l'état du menu dans *GLUT*.

La valeur **NULL** pour la fonction **glutMenuStatusFunc** désactive la génération de fonction de rappel pour l'état du menu.

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

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

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

void glutTimerFunc (unsigned int msecs, void (*func) (int value), value);

La fonction **glutTimerFunc** établit une fonction de rappel de minuterie qui est appelée dans un nombre déterminé de millisecondes. La valeur du paramètre **value** de la fonction de rappel est la valeur du paramètre de la fonction **glutTimerFunc**. Plusieurs appels de la fonction de rappel à la même heure ou à des heures différentes peuvent être inscrits simultanément.

Le nombre de millisecondes constitue une borne inférieure avant qu'un appel à la fonction de rappel soit effectué. *GLUT* essaie d'effectuer l'appel à la fonction de rappel aussitôt que possible après l'expiration du délai. Il n'y a aucun moyen pour annuler une inscription d'une fonction de rappel de minuterie. Il faut plutôt ignorer l'appel en se basant sur la valeur du paramètre value.

16. Quelques variables d'état de GLUT

La bibliothèque *GLUT* contient un grand nombre de variables d'état dont un certain nombre (pas tous) peut être interrogé directement.

int glutGet (GLenum state);

Les principaux états de *GLUT* sont (il y en a un bon nombre) :

- **GLUT_WINDOW_X** Position en x en pixels relative à l'origine de la fenêtre courante.
- **GLUT_WINDOW_Y** Position en y en pixels relative à l'origine de la fenêtre courante.
- **GLUT_WINDOW_WIDTH** Largeur en pixels de la fenêtre courante.
- **GLUT_WINDOW_HEIGHT** Hauteur en pixels de la fenêtre courante.
- **GLUT_WINDOW_DEPTH_SIZE** Nombre total de bits du tampon de profondeur de la fenêtre courante.
- **GLUT_WINDOW_CURSOR** Le curseur courante de la fenêtre courante.
- **GLUT_SCREEN_WIDTH** Indique la largeur de l'écran en pixels; 0 indique que la largeur est inconnue ou non disponible.
- **GLUT_SCREEN_HEIGHT** Indique la hauteur de l'écran en pixels; 0 indique que la hauteur est inconnue ou non disponible.
- **GLUT_INIT_WINDOW_X** Position initiale en x en pixels relative à l'origine de la fenêtre courante.
- **GLUT_INIT_WINDOW_Y** Position initiale en y en pixels relative à l'origine de la fenêtre courante.

- **GLUT_INIT_WINDOW_WIDTH** Largeur initiale en pixels de la fenêtre courante.
- **GLUT_INIT_WINDOW_HEIGHT** Hauteur initiale en pixels de la fenêtre courante.
- **GLUT_ELPASED_TIME** Nombre de millisecondes depuis l'appel à glutIinit ou depuis le premier appel à glutGet(GLUT_ELAPSED_TIME).

La fonction **glutGet** interroge les variables d'état représenté par des entiers de la bibliothèque *GLUT*. Le paramètre **state** détermine quel état doit être retourné. Les variables d'état dont le nom commence par **GLUT_WINDOW** retournent des valeurs correspondant à la fenêtre courante. Les variables d'état dont le nom commence par **GLUT_MENU** retourne des valeurs concernant le menu courant. Les autres variables correspondent à des états globaux. Si une requête est incorrecte, la valeur -1 est retournée.

int glutGetModifiers (void);

Les valeurs retournées par cette fonction sont:

- **GLUT_ACTIVE_SHIFT** Une des touches modificatrices Shift ou CapsLock.
- **GLUT_ACTIVE_CTRL** La touche modificatrice Ctrl.
- **GLUT_ACTIVE_ALT** La touche modificatrice Alt.

La fonction **glutGetModifiers** retourne la valeur d'une des touches modificatrices lorsqu'un événement d'entrée est généré à partir du clavier, d'une touche spéciale ou de la souris. On ne doit faire appel à cette fonction que lors du traitement d'une fonction de rappel du clavier, des touches spéciales ou de la souris. Le système de fenêtrage peut intercepter certaines touches modificatrices; dans ce cas, aucun appel à des fonctions de rappel n'est effectué.

17. Rendu des polices de caractères

La bibliothèque *GLUT* supporte deux types de polices de caractères: les polices haute **qualité [stroke fonts**] pour lesquelles chaque caractère est construit à l'aide de segments de lignes et les polices de basse qualité [**bitmap fonts**] qui sont formées d'un ensemble de pixels et affichées avec la fonction **glbitmap**. Les polices haute qualité ont l'avantage de pouvoir être mises à l'échelle. Les polices basse qualité sont moins flexibles mais habituellement plus rapide à afficher.

void glutBitmapCharacter (void *font, int character);

Sans aucune liste d'affichage, la fonction **glutBitmapCharacter** affiche le caractère **character** selon la police de caractères **font**. Les polices de caractères disponibles sont:

GLUT_BITMAP_8_BY_13, GLUT_BITMAP_9_BY_15, GLUT_BITMAP_TIMES_ROMAN_10, GLUT_BITMAP_TIMES_ROMAN_24, GLUT_BITMAP_HELVETICA_10, GLUT_BITMAP_HELVETICA_12, GLUT_BITMAP_HELVETICA_18

Pour une chaîne de caractères, on utilise la fonction **glutBitmapCharacter** dans une boucle pour la longueur de la chaîne. Pour se positionner pour le premier caractère de la chaîne, on utilise la fonction **glRasterPos2f**.

int glutBitmapWidth (GLUTbitmapFont font, int character);

La fonction **glutBitmapWidth** retourne en pixels, la largeur d'un caractère dans une police de caractères supportée. Pendant que la largeur d'une police de caractères peut varier (la largeur d'une police fixe ne varie pas), la taille maximum d'une police est toujours fixe.

void glutStrokeCharacter (void * font, int character);

En n'utilisant aucune liste d'affichage, le caractère **character** est affiché selon la police de caractères **font**. Les polices de caractères sont: **GLUT_STROKE_ROMAN** et **GLUT_STROKE_MONO_ ROMAN** (pour les caractères ASCII de 32 à 127).

La fonction **glTranslatef** est utilisée pour positionner le premier caractère d'une chaîne de texte.

void glutStrokeWidth (GLUTstrokeFont font, int character);

La fonction **glutStrokeWidth** retourne en pixels, la largeur d'un caractère dans une police de caractères supportée. Pendant que la largeur d'une police de caractères peut varier (la largeur d'une police fixe ne varie pas), la taille maximum d'une police est toujours fixe.

18. Rendu d'objets géométriques

Bien que cela ne soit pas le rôle principal de GLUT, il existe quelques fonctions permettant de construire des objets géométriques 3D de base.

void glutSolidSphere (GLdouble radius, GLint slices, GLint stacks);

void glutWireSphere (GLdouble radius, GLint slices, GLint stacks);

Affichent une sphère centrée à l'origine de rayon **radius**. La sphère est subdivisée en **tranches** et en **pile** autour et le long de l'axe des z.

void glutSolidCube (GLdouble size);

void glutWireCube (GLdouble size);

Affichent un cube plein ou en fil de fer centré à l'origine. La largeur du côté est donnée par **Size**.

void glutSolidCone (GLdouble base, GLdouble height, GLint slices, GLint stacks);

void glutWireCone (GLdouble base, GLdouble height, GLint slices, GLint stacks);

Affichent un cône plein ou en fil de fer. La **base** est à **z=0** et le **sommet** du cône est à **z=height**. Le cône est subdivisé en **tranches** autour de l'axe des z et en **pile** le long de l'axe des z.

void glutSolidTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void glutWireTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

Affichent un tore plein ou en fil de fer. Le rayon intérieur **innerRadius** est utilisé pour calculer une section de cercle qui tourne autour du rayon extérieur **outerRadius**. Le tore est composé de **rings** anneaux subdivisées en **nsides** côtés.

void glutSolidDodecahedron (void);

void glutWireDodecahedron (void);

Affichent un dodécaèdre (12 côtés réguliers) plein ou en fil de fer centré à l'origine de rayon 3 en coordonnées de modélisation.

void glutSolidOctahedron (void);

void glutWireOctahedron (void);

Affichent un octaèdre plein ou en fil de fer centré à l'origine de rayon 1 en coordonnées de modélisation.

void glutSolidTetrahedron (void);

void glutWireTetrahedron (void);

Affichent un tétraèdre plein ou en file de fer centré à l'origine de rayon 3 en coordonnées de modélisation.

void glutSolidIcosahedron (void);

void glutWireIcosahedron (void);

Affichent un icosaèdre plein ou en file de fer centré à l'origine de rayon 1.0 en coordonnées de modélisation.

void glutSolidTeapot (void);

void glutWireTeapot (void);

Les fonctions **glutSolidTeapot** et **glutWireTeapot** affichent une théière pleine ou en fil de fer.

III. Les primitives graphiques

Toute primitive surfacique 3D est décomposée en triangles par OpenGL.
Le triangle, la ligne et le point sont donc les seules primitives géométriques traitées par le hardware, ce qui permet de ramener toutes les interpolations au cas linéaire (facile à traiter de façon incrémentale au niveau hardware).

Bien que cela ne rentre pas dans la catégorie des primitives, nous allons tout d'abord présenter comment définir la couleur (voir annexe) qui sera employée pour le tracé (on choisit en quelque sorte son crayon avant de dessiner) . Cette couleur est mémorisée au moyen d'une variable d'état que l'on peut modifier à tout moment. Les primitives qui seront tracées par la suite recevront cette couleur jusqu'à la prochaine modification de cette variable.

6. La couleur

Une couleur est généralement caractérisée par 3 valeurs réelles (dans l'ordre : le rouge, le vert et le bleu) où chaque composante doit varier dans `[0.0, 1.0]`. Cette représentation flottante est privilégiée au niveau hardware, notamment pour les calculs de rendu.

Une quatrième valeur, dite *composante alpha*, peut être spécifiée. Il s'agit d'un coefficient *d'opacité* qui vaut 1 par défaut. Une valeur plus faible permettra de définir une certaine transparence pour une face et de « voir » les objets qui se trouvent derrière. La gestion de ce coefficient pose un certain nombre de problèmes que nous préférons ne pas aborder pour une initiation à OpenGL. La composante alpha sera forcée à 1 lorsqu'une fonction OpenGL la réclame.

7. La couleur du fond

Un dessin commence sur une feuille dont il faut définir la couleur (le fond). Cette opération pourrait consister à tracer un rectangle de cette couleur, mais :

- ce fond n'est pas simple à définir dans le cas d'une scène 3D projetée dans la fenêtre,
- il est plus efficace d'utiliser une commande spéciale (cablée),

Il est à noter que colorier le fond consiste aussi à effacer ce qu'il y avait dans la fenêtre de visualisation en passant une nouvelle couche de « peinture ».

```
glClearColor(0.0, 0.0, 0.0, 1.0) ; /* définit la couleur d'effacement, ici noire */
```

La couleur du fond est affectée à une variable d'état et sera utilisée pour chaque appel de

```
glClear(GL_COLOR_BUFFER_BIT) ; /* effacement du contenu de la fenêtre */
```

`GL_COLOR_BUFFER_BIT` est une constante GL qui désigne les pixels de la fenêtre d'affichage.

Remarque : on spécifie la composante alpha qui vaut généralement `1.0` (fond opaque !).

8. La couleur des primitives

`glColor3f(`*`rouge, vert, bleu`*`)` permet de spécifier la couleur pour toutes les primitives graphiques qui vont suivre. Il est possible de rajouter un quatrième paramètre pour caractériser l'opacité (paramètre *alpha*) mais sa gestion n'est pas simple sous OpenGL et nous préfèrerons l'ignorer ici.
Quelques exemples de couleurs :

```
glColor3f(0., 0., 0.) ; /* noir */
glColor3f(1., 1., 1.) ; /* blanc */
glColor3f(0.5, 0.5, 0.5) ; /* gris moyen */
glColor3f(1., 0., 0.) ; /* rouge */
```

On peut aussi utiliser la notation vectorielle :

```
GLfloat rouge[3] = {1., 0., 0.};
GLfloat jaune[3] = {1., 1., 0.};
glColor3fv(rouge);
```

On conseille de séparer, quand on le peut, l'affectation d'une couleur de la construction géométrique d'un objet. On favorise ainsi la conception modulaire. Cela permet par exemple de construire plusieurs « clônes » de couleurs différentes :

```
glColor3fv(rouge);
dessineBicyclette(position, direction);
glColor3fv(jaune);
dessineBicyclette(autrePosition, autreDirection);
```

Une couleur peut être employée « brutalement » sans faire référence à aucun modèle de lissage ou d'éclairage. On spécifie simplement une fois avant le tracé :

```
glShadeModel(GL_FLAT)
```

OpenGL propose aussi des rendus plus « sophistiqués » avec l'option `glShadeModel(GL_SMOOTH)` qui permet d'obtenir des dégradés de couleur ou de prendre en compte l'orientation des faces par rapport aux éclairages (cette partie sera développée dans la partie *VI. Amélioration du rendu* de ce cours).

9. Primitives graphiques

Les fonctions **`glBegin(…)`** et **`glEnd()`** délimitent la suite de sommets associés au tracé.
Cette suite de sommets pourra aussi bien définir des lignes brisées (contours) que des polygones (éléments surfaciques). Un polygone doit être obligatoirement plan.

Un sommet est défini par la fonction **`glVertex*(…)`**

Par exemple, on dessine un triangle « plein » de la façon suivante:

```
glColor3f(1., 0., 0.) ; /* crayon rouge */
glBegin(GL_TRIANGLES);
/* un triangle */
glVertex3f(x1,y1,z1);
glVertex3f(x2,y2,z2);
glVertex3f(x3,y3,z3);
/* un autre triangle */
glVertex3f(x4,y4,z4);
glVertex3f(x5,y5,z5);
glVertex3f(x6,y6,z6);
/* etc. */
glEnd();
```

L'argument de `glBegin spécifie le type de primitive (OpenGL en propose 10 ), notamment` des plus complexes (qui seront décomposées en triangles...) :

- des quadrilatères (convexes et **plans**) avec GL_QUADS,
- des polygones (convexes et **plans**) avec GL_POLYGON.

On peut également ne tracer que les sommets :

```
glBegin(GL_POINTS);
glVertex2i(x1,y1);
glVertex2i(x2,y2);
glVertex2i(x3,y3);
glVertex2i(x4,y4);
glEnd();
```

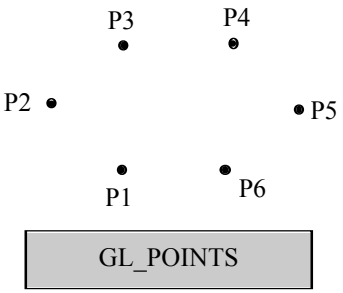
On notera au passage que `glVertex` est polymorphe :

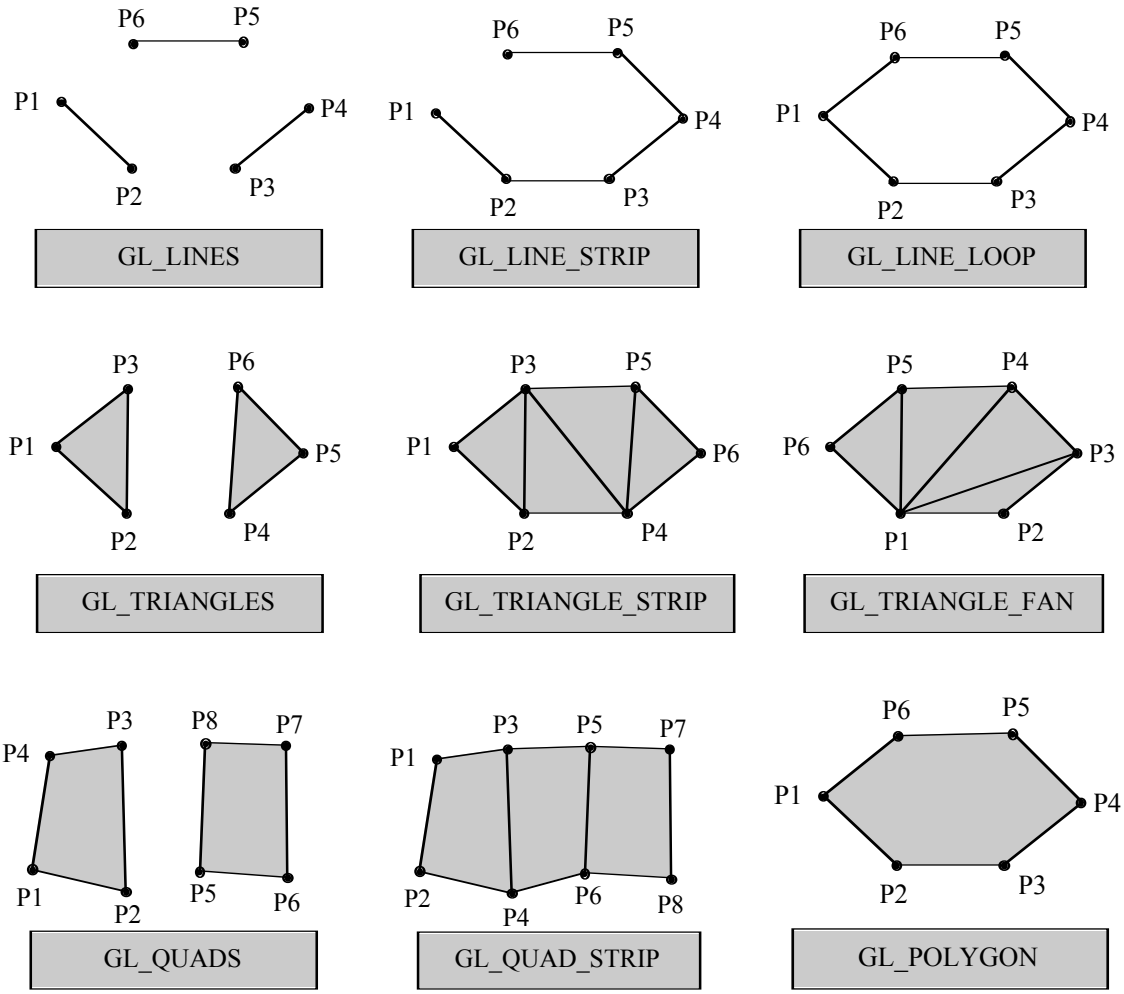
- on peut fournir 2 à 4 composantes (on peut se passer de la 3ième coordonnée si l'on fait des tracés 2D, le 4ième paramètre correspond à la coordonnée homogène...);
- on peut utiliser des `GLfloat`, des `GLdouble`, des `GLshort`, des `GLint` pour les coordonnées (suffixe `f,d,s` ou `i`);
- on peut citer explicitement les coordonnées, ou passer par un vecteur (sur-suffixe `v`).

```
GLfloat P1[3] = {0, 0, 0} ;
GLfloat P2[3] = {4, 0, 0} ;
GLfloat P3[3] = {0, 2, 0} ;
glBegin(GL_LINE_LOOP);
glVertex3fv(P1);
glVertex3fv(P2);
glVertex3fv(P3);
glEnd();
```

Valeurs du paramètre de `glBegin` :

```
glBegin(GL_primitive);
  glVertex3fv(P1);
  glVertex3fv(P2);
  glVertex3fv(P3);
  glVertex3fv(P4);
  glVertex3fv(P5);
  glVertex3fv(P6);
glEnd();
```





Remarques sur l'optimisation du code GL :

- GL_TRIANGLES est plus rapide que GL_POLYGON
- Il est plus efficace de regrouper le maximum de primitives entre `glBegin()` et `glEnd()`.
- La notation vectorielle (suffixe `v`) est généralement plus rapide.

10. Faces avant et arrière

L'orientation d'un polygone est définie par l'ordre dans lequel on parcourt ses sommets lorsqu'on le dessine : traditionnellement, on définit sa **face avant** vers nous lorsque l'on parcourt ses sommets dans le sens trigonométrique. C'est cette règle qui est aussi appliquée par OpenGL pour définir l'avant et l'arrière d'une face.

OpenGL permet de traiter différemment les faces avant et arrière d'un polygone, soit par un rendu différent pour mieux les distinguer, soit pour optimiser les calculs en « oubliant » de dessiner un coté d'une face (par exemple la face intérieure du coté d'un cube). Cette distinction s'opère à l'aide de :

```
GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.
```

On peut obtenir des polygones pleins, ou seulement leurs contours :

```
glPolygonMode (GL_FRONT, GL_FILL); /* faces */
glPolygonMode (GL_BACK, GL_LINE);  /* contours */
```

Ou encore, supprimer une des deux faces (gain de temps !) :

```
glCullFace (GL_BACK);
glEnable (GL_CULL_FACE);
```

Dans le cas d'une scène éclairée, il faudra préciser:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE) ;
```

si l'on veut éclairer les deux côtés d'une face.

IV Construction d'une scène 3D

Il s'agit de définir ici les différents objets qui composent une scène, et de les positionner dans l'espace à l'aide de transformations géométriques 3D.

Sous GLUT, la scène est construite dans la fonction déclarée par `glutDisplayFunc` avant de lancer la boucle d'événement (`glutMainLoop`). On rappelle que cette fonction d'affichage est déclenchée par le système :
- soit parce que la fenêtre a été modifiée (redimensionnement),
- soit parce que le programmeur l'a demandé par l'intermédiaire de la fonction `glutPostRedisplay()`

OpenGL appliquera automatiquement la matrice de projection aux objets que l'on a construit pour obtenir une image 2D qui sera affichée dans la fenêtre.

6. Transformations géométriques de bases

La construction d'un objet et son positionnement dans la scène vont se faire à partir de trois transformations géométriques de bases :
- la **translation**,
- la **rotation** (en degré) autour d'un axe porté par un vecteur,
- l'**homothétie** suivant les trois axes X, Y et Z.

Ces transformations sont représentées par des matrices de dimension 4 dans l'**espace projectif**. Appliquer une de ces transformations consiste à opérer sa matrice sur les coordonnées des différents sommets de l'objet considéré.

7. Préliminaire : les Espaces Projectifs

a. Rappel sur les transformations 2D

a) Translation 2D

Soit un vecteur T(dx, dy) et un point P(x, y), alors le translaté P'(x', y') de P par T est donné par :

$$P' = P + T$$

b) Homothétie 2D

elle se fait par rapport à l'origine.

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} h & 0 \\ 0 & k \end{vmatrix} \cdot \begin{vmatrix} x \\ y \end{vmatrix}$$

l'homothétie est dite **uniforme** si h.x = k.y et **différentielle** sinon

c) Rotation 2D

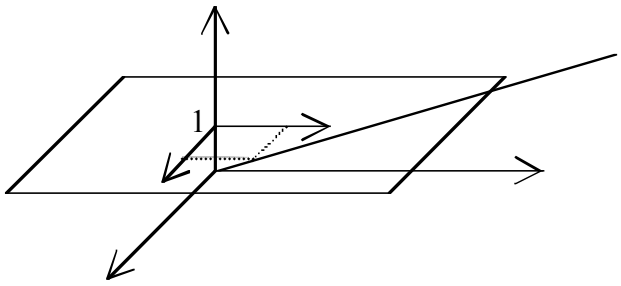
Une rotation d'angle θ autour de l'origine est définie par :

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{vmatrix} . \begin{vmatrix} x \\ y \end{vmatrix}$$

b. Coordonnées homogènes en 2D

Si l'enchaînement d'homothétie et de rotation s'exprime sous la forme d'un produit matriciel, la translation est une opération de nature différente. Celle-ci peut toutefois être aussi définie comme un produit matriciel si les objets sont exprimés en **coordonnées homogènes**.

Les coordonnées homogènes d'un point sont obtenues en ajoutant une coordonnée supplémentaire égale à 1. On considère que tout point (x, y, w) dans **l'espace projectif** est un représentant du point (x/w, y/w) dans l'image initiale.



On remarquera que :
- un point image est associé à une droite dans l'espace projectif,
- les points pour lesquels w=0 sont des points à l'infini,
- l'enchaînement des trois transformations précédentes s'expriment sous la forme d'un produit matriciel.

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \qquad \begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} hx & 0 & 0 \\ 0 & hy & 0 \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \qquad \begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} . \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

c. Composition de transformations 2D

Rappel : le produit matriciel est associatif mais non commutatif.

Etudions la rotation autour d'un point C (cx, cy). Cette opération se décompose en trois transformations élémentaires :
- une translation de C vers O
- la rotation de θ autour de O
- une translation de O vers C

et qui se traduit par une formule de la forme : P' = T(cx,cy) . R(θ) . T(-cx,-cy) . P

Cette opération étant effectuée sur tout les points P de l'image, on peut précalculer la composition des transformations :

$$\begin{vmatrix} 1 & 0 & cx \\ 0 & 1 & cy \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & -cx \\ 0 & 1 & -cy \\ 0 & 0 & 1 \end{vmatrix}$$

$$\begin{vmatrix} 1 & 0 & cx \\ 0 & 1 & cy \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} \cos\theta & -\sin\theta & -\cos\theta.cx+\sin\theta.cy \\ \sin\theta & \cos\theta & -\sin\theta.cx-\cos\theta.cy \\ 0 & 0 & 1 \end{vmatrix}$$

$$\begin{vmatrix} \cos\theta & -\sin\theta & (1-\cos\theta).cx+\sin\theta.cy \\ \sin\theta & \cos\theta & -\sin\theta.cx+(1-\cos\theta).cy \\ 0 & 0 & 1 \end{vmatrix}$$

On démontre que la combinaison de ces trois transformations géométriques de base donne toujours une matrice de la forme :

$$\begin{vmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{vmatrix}$$

d. Extension au 3D

Nous nous plaçons dans un repère orthonormé direct où les rotations positives s'effectuent dans le « sens inverse des aiguilles d'une montre », à savoir :

Axes de rotation	Direction d'une rotation positive
x	y à z
y	z à x
z	x à y

L'utilisation de coordonnées homogènes est naturellement applicable au 3D et les transformations géométriques de bases sont représentées par des matrices 4x4.

Un point image (x, y, z) est représenté par (x.w, y.w, z.w, w), (x, y, z, 1) étant les coordonnées homogènes.

Les matrices associées aux translations et aux homothéties sont respectivement de la forme :

$$\begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 1\,0 & & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} hx & 0 & 0 & 0 \\ 0 & hy & 0 & 0 \\ 0\,0 & & hz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Les rotations se décomposent facilement suivant les axes du repère.
Les rotations autour de Oz, Ox, et Oy sont respectivement de la forme :

$$\begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad \begin{vmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

8. Transformations géométriques sous OpenGL

OpenGL dispose d'une matrice de transformation courante pour la modélisation, matrice qui est rendue active par la fonction :

```
glMatrixMode(GL_MODELVIEW)
```

Cette **matrice de modélisation** est appliquée automatiquement à tous les objets qui vont être tracés.

Le positionnement d'un objet dans une scène est décomposé comme une succession de transformations de base qu'OpenGL traduit par un produit matriciel (cf les *espaces projectifs*). La matrice de transformation courante est construite :

- à partir de la matrice identité,
- et par produits successifs avec des matrices d'opérations de base.

La matrice de modélisation est initialisée avec l'identité en appelant la fonction :

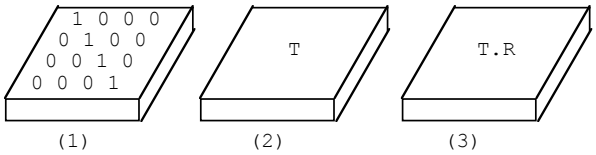
```
glLoadIdentity()
```

Il est possible de gérer soit même les opérations de base mais OpenGL propose des fonctions simples où le produit matriciel avec la matrice de transformation courante est implicite :

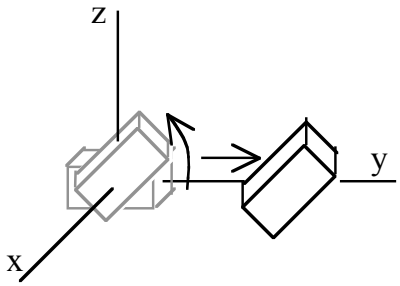
```
glTranslatef(GLfloat x, GLfloat y, GLfloat z) ;
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z) ;
glScalef(GLfloat x, GLfloat y, GLfloat z) ;
```

Les appels successifs de ces fonctions composent donc une seule transformation.
Remarque : il faut lire les opérations effectuées sur l'objet dans l'ordre inverse de leur apparition dans le code.

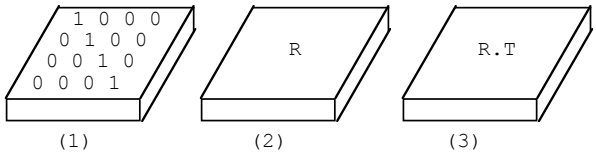
```
/* matrice de transformation A */
glLoadIdentity() ;      /* 1 */
glTranslatef(0, 5, 0);  /* 2 */
glRotatef(45, 1, 0, 0); /* 3 */
/* objet qui subira la transformation*/
dessineBoite();
```



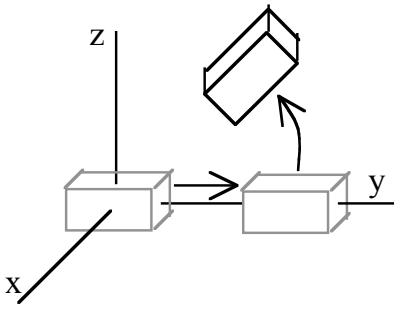
Evolution de la matrice de modélisation (A)




```
/* matrice de transformation B */
glLoadIdentity() ;      /* 1 */
glRotatef(45, 1, 0, 0); /* 2 */
glTranslatef(0, 5, 0);  /* 3 */
/* objet qui subira la transformation*/
dessineBoite();
```



Evolution de la matrice de modélisation (B)



9. Gestion des transformations

OpenGL dispose en fait d'une **pile de matrices de modélisation** qui va faciliter la description hiérarchique d'un objet complexe. La matrice courante (la seule active) est celle se trouvant au sommet de pile, mais les jeux d'empilage et dépilage permettront d'appliquer la même transformation à plusieurs assemblages ayant eux mêmes nécessité des transformations spécifiques pour leur construction.

`glLoadIdentity()` : mettre l'identité au sommet de pile
`glPushMatrix()` : empiler
`glPopMatrix()` : dépiler

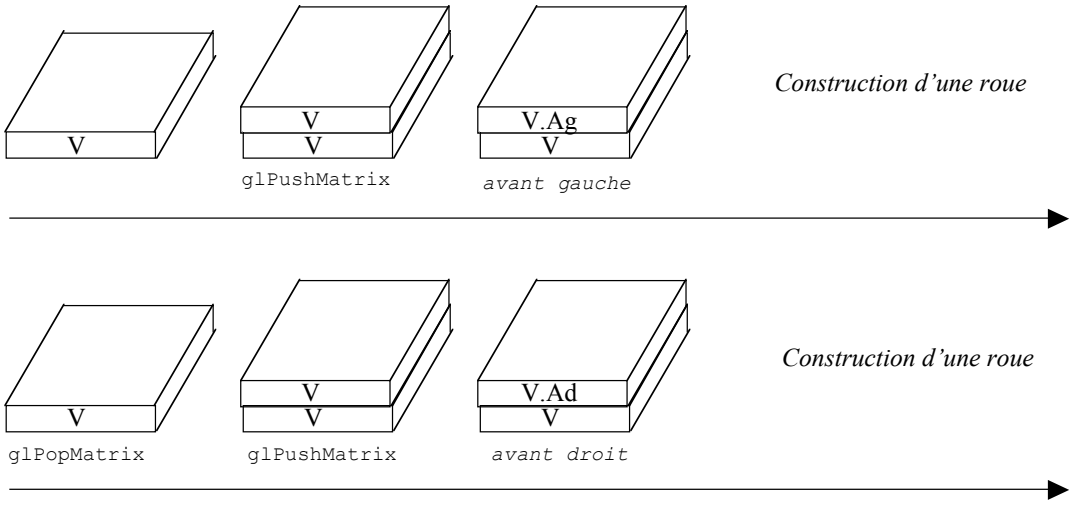
Si l'opération de dépilage ne présente pas de difficulté particulière (on retrouve la transformation précédente), l'opération d'empilage réclame quelques précisions : il s'agit en fait d'une duplication de la matrice se trouvant au sommet de pile. Toutes les opérations qui seront effectuées par la suite seront combinées à la transformation initiale (dupliquée dans le sommet de pile) de sorte que le sous-objet que l'on est en train de construire « hérite » de la transformation appliquée globalement à l'objet.
Bien sûr, il est toujours possible de faire suivre un `glPushMatrix()` par `glLoadIdentity() pour oublier temporairement une matrice de modélisation`.

Par exemple, pour dessiner une voiture, on définira ce qu'est la construction d'une roue dans le repère absolu, et on se positionnera successivement aux quatre coins de la voiture (repère voiture) avant d'appeler cette procédure. Si la voiture a été positionnée à un endroit spécifique de la scène, ses roues subiront aussi cette transformation « globale ».

```
void dessineRoueEtBoulons()
{ int i ;
  dessineRoue() ;
  for (i=0; i<3; i++)
  { glPushMatrix() ;
      glRotatef(120*i, 0, 0, 1) ;
      glTranslatef(2, 0, 0) ;
      dessineBoulon() ;
    glPopMatrix() ;
  }
}

void dessineVoiture()
{ int i, posx[4]={20, 20, -20, -20}, posz[4]={8, -8, 8, -8};
  dessineCarrosserie() ;
  for (i=0; i<4; i++)
  { glPushMatrix() ;
      glTranslatef(posx[i], 5, posz[i]) ;
      dessineRoueEtBoulons() ;
    glPopMatrix() ;
  }
}
```

V : matrice positionnant la voiture dans la scène (repère « scène »)
Ag : matrice définissant la roue avant gauche de la voiture dans le référentiel « voiture »
Ad : matrice définissant la roue avant droite de la voiture dans le référentiel « voiture »



Evolution de la pile de modélisation dans la construction hiérarchique d'une voiture

Remarques :

a. La pile est initialisée par OpenGL avec la matrice identité.
b. Il est intéressant de conserver la matrice identité en bas de pile pour éviter de rappeler systématiquement `glLoadIdentity()` (=> une construction débute toujours par un *push*) .

10. Listes d'affichage

Il est possible de stocker une suite de routines OpenGL (à l'exception de quelques fonctions...) dans une liste qui pourra être réutilisée plusieurs fois. Il y a alors une précompilation des instructions GL et cette opération sera particulièrement rentable lorsqu'un objet « définitif » est dessiné plusieurs fois, soit parce qu'il constitue une primitive employée à plusieurs reprises (roue d'une voiture), soit parce qu'il se déplace dans la scène (animation).

c. une liste est identifiée par un numéro (`GLint`) strictement positif,
d. la création et la suppression des listes est gérée par OpenGL : **`glGenLists(nbIndex)`** attribue *`nbIndex`* numéros de listes consécutifs, le premier numéro est retourné par cette fonction (0 si échec d'allocation),
e. **`glDeleteLists(numListe, nbre)`** restitue au système le *nbre* de listes indiqué à partir de *`numListe`*.
f. **`glNewList(numListe, mode)`** et **`glEndList()`** permettent de créer une liste,
g. **`glCallList(numListe)`** exécute une liste d'affichage,

Par exemple, pour dessiner un tricycle, on pourra stocker la construction d'une roue dans une liste et appeler 3 fois cette liste après avoir définie les spécificités de chaque roue (position, taille).

```
int listeRoue ;
...
listeRoue  = glGenLists(1);
...
glNewList(listeRoue, GL_COMPILE) ; // ou encore GL_COMPILE_AND_EXECUTE
  suite d'instructions pour dessiner une roue de tricycle
glEndList() ;


void dessinerTricycle()
{ ...
  transformations pour positionner la roue arrière gauche
  glCallList(listeRoue);
  transformations pour positionner la roue arrière droite
  glCallList(listeRoue);
  transformations pour positionner la roue avant
  glScalef(1.2, 1.2, 1.) ; // roue avant 20% plus grande mais même largeur
  glCallList(listeRoue);
  ...
}
```

Remarque : Une liste peut-être construite avant de déclencher la boucle d'événements dans le `main()` (`glutMainLoop()`). Il faut toutefois que le processus graphique soit déjà initialisé (i.e. après `glutInit()` et `glutInitDisplayMode()`) et que l'on choisisse l'option `GL_COMPILE` .

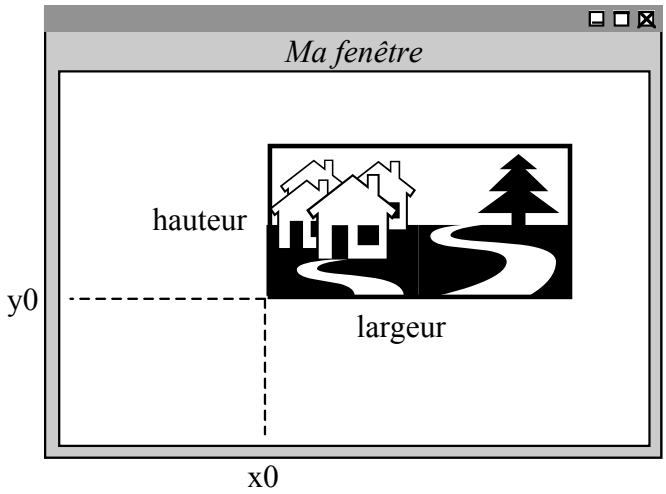
V. Visualisation d'une scène

7. Cadrage

Il ne faut pas confondre la fenêtre d'affichage (définie par le système de fenêtrage qui est indépendant d'OpenGL) et le cadre (partie de fenêtre) dans lequel on veut visualiser la scène. De même que l'on colle une photo sur un poster, on va positionner l'image de la scène dans la fenêtre en la déformant éventuellement pour la faire rentrer dans un cadre (il faut conserver le ratio largeur/hauteur pour obtenir une image non « déformée »). C'est le rôle de la fonction `glViewport` .

`glViewport(GLint x0, GLint y0, GLint largeur, GLint hauteur)`

`x0` et `y0` précisent le coin inférieur gauche du cadre
`largeur` et `hauteur` précisent ses dimensions en pixels



GLUT envoie un événement pour dire que la fenêtre courante a été créée, déplacée ou redimensionnée. Cet événement est intercepté par la fonction désignée dans le main par *glutReshapeFunc* (monCadrage dans notre exemple) et l'affichage de la scène est automatiquement relancée. Le programmeur pourra, à cette occasion, décider d'adapter ou non la projection de la scène aux dimensions de la fenêtre. Exemple :

```
void monCadrage(int large, int haut)
/* les arguments sont fournis par GLUT : nouvelle taille de la fenêtre */
{ /* taille du cadre d'affichage dans la fenetre */
  glViewport(0, 0, large, haut) ;
  /* On peut a cette occasion redéfinir la projection de la scene */
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  gluPerspective(90., (float)large/(float)haut, 5, 20) ;
  glMatrixMode(GL_MODELVIEW) ;
  /* la fonction afficheMaScene() va etre automatiquement
     declenchee par le systeme */
}
```

8. Le mode *projection*

Lorsqu'une scène est construite, sa visualisation nécessite deux types de transformation :
- des transformations dans l'espace 3D qui permettent de positionner le point de vue et les éventuels éclairages si l'on veut obtenir un rendu réaliste,
- la transformation qui consiste à projeter cette scène 3D sur une fenêtre 2D et qui caractérise les propriétés de la prise de vue.

Pour le premier type de transformation, on utilise encore la **matrice de modélisation** (`GL_MODELVIEW`) : les lumières et le point de vue sont positionnés comme les autres acteurs de la scène.

Pour la projection 2D, openGL dispose d'une autre matrice spécifique : la **matrice de projection** .

Une seule de ces deux matrices est active à un moment donné et l'on bascule de l'une à l'autre avec :

```
glMatrixMode(GL_PROJECTION)
```

ou bien

```
glMatrixMode(GL_MODELVIEW)
```

Dans la pratique, on pourra définir la projection soit :
- dans la fonction de cadrage (cf la fonction `cadrage` ci-dessus),
- dans la fonction d'affichage déclarée par `glutDisplayFunc(afficheMaScene)`.

Dans ce deuxième cas, la fonction `afficheMaScene` a la structure suivante :

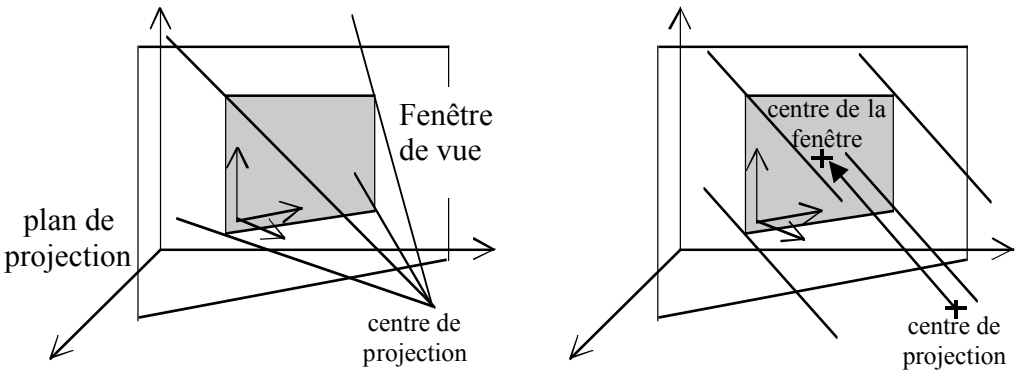
```
void afficheMaScene(void)
{ glMatrixMode(GL_PROJECTION) ;
  definir la projection
  glMatrixMode(GL_MODELVIEW) ;
  glClear(GL_COLOR_BUFFER_BIT); /* définit le fond de la scène */
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}


int main (int argc, char **argv)
{
      …
      glutReshapeFunc (monCadrage);
      glutDisplayFunc (afficheMaScene);
      …
      glutMainLoop ();
      return 0;
}
```

Remarque : dans les exemples de ce document, le cadrage et la projection sont definis dans la même fonction `monCadrage(int l, int h)`

9. Caractéristiques de l'appareil photo

OpenGL propose deux types de projection : la projection en **perspective** et la projection **parallèle**.



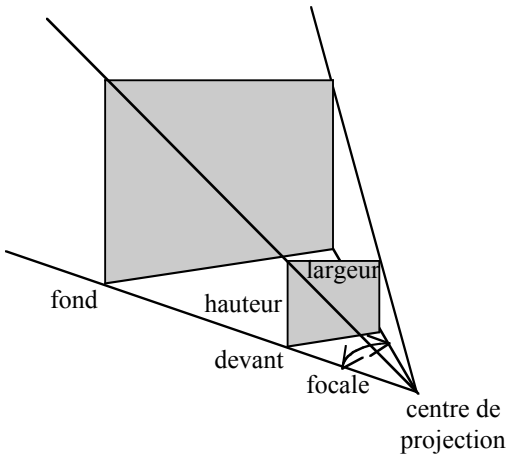
Pour un volume visionné en perspective conique :

`gluPerspective(GLdouble focale, GLdouble aspect, GLdouble devant, GLdouble fond)`

`focale` : angle du champ de vision (dans [0°, 180°])

`aspect` : rapport largeur/hauteur du plan de devant

`devant`, `fond` : distances (valeurs positives) du point de vue aux plans de clipping.



Pour un volume visionné en perspective cavalière (projection parallèle) :

`glOrtho(GLdouble gauche, GLdouble droite, GLdouble bas, GLdouble haut,`
`GLdouble devant, GLdouble fond)`

définit la « boite » de visualisation où (`gauche, bas, devant`) sont les coordonnées du point avant-inférieur-gauche et `(droite, haut, fond)` sont les coordonnées du point arrière-supérieur-droit.

Remarques :

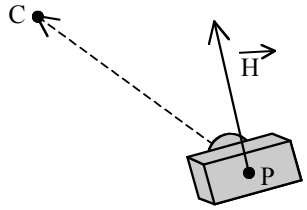
- Bien que cela soit possible, on ne compose généralement pas les projections.
- Il est important de minimiser au mieux la distance entre le plan de clipping avant et le plan de clipping arrière. En effet, OpenGL dispose d'une précision limitée pour représenter l'intervalle des profondeurs et une mauvaise gestion peut positionner des sommets artificiellement dans le même plan par effet d'arrondi, au risque de créer des artefacts de rendu. Ce phénomène peut se manifester occasionnellement lorsque l'on fait tourner un objet.

10. Positionnement de l'appareil photo

Le point de vue se situe par défaut à l'origine en regardant vers l'axe des z négatifs. Pour visualiser une scène, on peut
- soit la reculer pour la mettre dans le champ de vision,
- soit déplacer le point de vue avec `gluLookAt` .

Bien que le positionnement du point de vue soit déterminant pour effectuer la projection, on comprend pourquoi celui-ci doit être défini en mode **`GL_MODELVIEW`** : c'est un déplacement relatif de la scène.

```
gluLookAt( GLdouble Px, GLdouble Py, GLdouble Pz,      // position de l'appareil
           GLdouble Cx, GLdouble Cy, GLdouble Cz,      // point visé dans la scène
           GLdouble Hx, GLdouble Hy, GLdouble Hz)      // haut de l'appareil
```



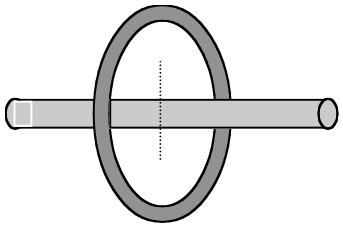
On peut maintenant donner un schéma un peu plus précis de la fonction d'affichage :

```
void afficheMaScene(void)
{ /* on recule de 5 dans la scène */
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity() ;
  gluLookAt(0,0,5, 0,0,0, 0,1,0) ;
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}
```

11. Z-buffer

Lorsque deux objets sont positionnés dans une scène, il est possible que l'un des deux soit partiellement ou totalement caché par l'autre en fonction de la position de l'observateur. Or, en pratique, le dernier dessiné écrase une partie du premier, et ceci indépendamment du point de vue qui peut changer.

L'algorithme du peintre est une solution qui n'est plus guère utilisée. Il consiste à trier les objets suivant l'ordre décroissant de leur distance au point de vue et à les dessiner dans cet ordre. Les objets en premier plan seront dessinés en dernier et « écraseront » les parties cachées des objets en arrière plan.



Découpage d'objets pour disposer d'une relation d'ordre totale.

Bien qu'un peu plus coûteux en espace mémoire, on préfère maintenant utiliser un tampon de profondeur (distance au point de vue) qui permet de s'affranchir de l'ordre de construction des objets : le **Z-buffer**. Le Z-buffer a la taille de la fenêtre de projection et est initialisé avec la plus grande profondeur possible (généralement le plan de clipping arrière). Lorsqu'un objet est dessiné, les pixels qui lui correspondent dans le plan de projection ont une valeur de profondeur qui est comparée à celle stockée dans le Z-buffer. Si un pixel est plus éloigné, il est abandonné. Sinon, le Z-buffer reçoit sa profondeur et la fenêtre reçoit ses valeurs chromatiques.

Mise en œuvre :

On déclare l'utilisation du Z-buffer à l'initialisation avant de rentrer dans la boucle d'événements avec la constante **`GLUT_DEPTH`** :

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

On active ou désactive le mode Z-buffer avec :

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
```

Ces deux opérations peuvent se faire à tout moment et permettent par exemple de rajouter des tracés « par dessus » la représentation d'une scène.

De même que l'on (re)définit la couleur du fond avec `glClearColor(r, v, b, a)`, on (re)définit au moins une fois la distance maximum de représentation d'un pixel :

```
glClearDepth(10.0) ; /* distance max visible au point de vue */
```

On définit ainsi la position d'un plan arrière de clipping lié au point de vue (ce qui est derrière sera occulté par ce plan).

Enfin, le dessin d'une scène sera toujours débuté par une réinitialisation du Z-buffer à la profondeur maximum :

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Cette opération est généralement associée à l'effacement de la fenêtre avec la couleur du fond :

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

Notre fonction d'affichage aura donc la forme suivante :

```
void afficheMaScene(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  /* on s'écarte de 5 dans la scène */
  glLoadIdentity() ;
  gluLookAt(0,0,5, 0,0,0, 0,1,0) ;
  construire la scene
  glutSwapBuffers(); /* ou glFlush() */
}
```

12. La face cachée d'OpenGL

12.1 Ordre des opérations de construction d'une scène

Nous avons décrit l'ensemble des opérations qui permettent d'afficher une scène 3D en suivant un enchaînement « naturel » de la construction : construction des objets, positionnement dans la scène 3D, calcul des éclairages et des parties visibles, projection à l'écran.

Cette présentation pourrait laisser penser que la scène 3D est mémorisée quelque part et que l'on pourrait intervenir localement dessus avant d'effectuer une nouvelle projection. Il n'en est rien ! et il aurait fallu pour cela une capacité mémoire phénoménale.

On remarquera que le contexte de visualisation d'une primitive doit toujours être défini au préalable. Dans la pratique, chaque fois qu'une instruction de tracé d'un point, segment ou triangle est exécutée, son rendu et sa projection sont immédiatement déclenchés pour mettre à jour la mémoire écran (seule information conservée !) .

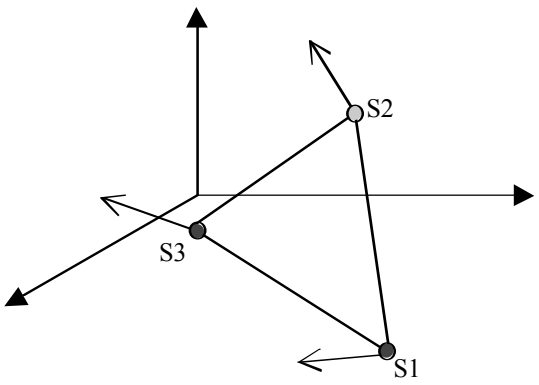
Une petite modification dans une scène implique donc un nouveau tracé complet de cette dernière. On pourra toutefois optimiser les calculs en découpant cette scène en plusieurs plans de profondeur que l'on mémorise. Par exemple, dans le cas d'une animation de personnage, on calcule et mémorise l'arrière-plan supposé ne pas évoluer ; la visualisation de la scène consiste alors à afficher l'arrière-plan (image 2D) et à superposer le personnage dans sa nouvelle position.

12.2 Détail du tracé d'une facette

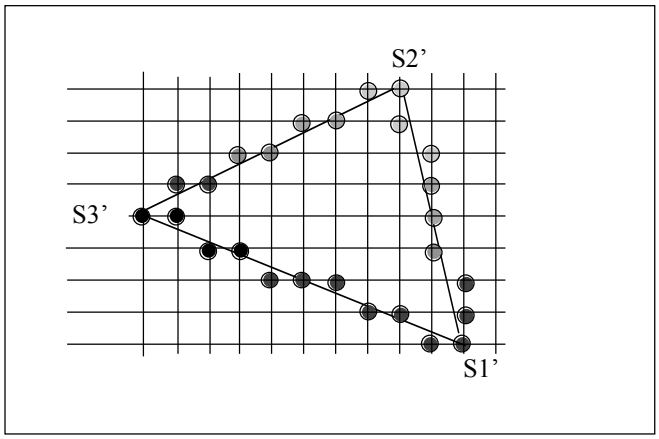
On peut s'intéresser un peu plus au fonctionnement de la « boîte noire » et étudier l'enchaînement des algorithmes implicitement mis en oeuvre par OpenGL lors du tracé d'une simple facette triangulaire. On pourra ainsi mieux « apprécier » les performances des matériels actuels lorsque l'on visualise des surfaces composées de plusieurs milliers de triangles…

Ainsi donc, la désignation de 3 sommets (`glVertex`) entre les deux instructions `glBegin(GL_TRIANGLES)` et `glEnd()` a pour effet de déclencher la séquence d'opérations suivante :

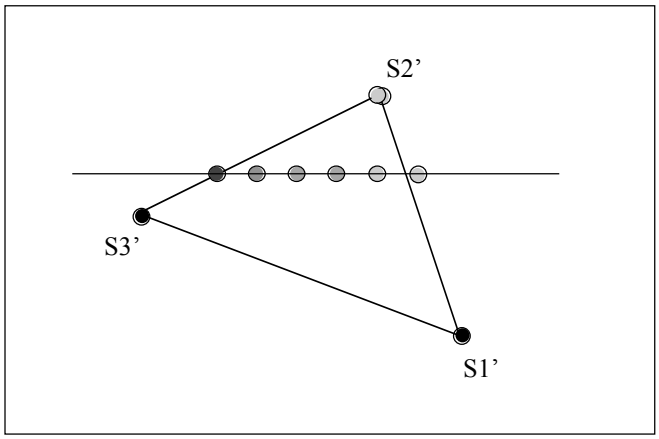
- produit des sommets par la matrice de modélisation (`GL_MODELVIEW`)
- évaluation de la couleur de chaque sommet en fonction du type de rendu (couleur brute ou simulation d'un éclairage avec gestion des normales, des lumières et du point de vue)



- évaluation de la distance des sommets au point de vue (profondeur)
- projection des sommets sur le plan image
- calcul des **pixels** constituant les 3 arêtes (algorithme de Bresenham) du triangle projeté
- extrapolation de la couleur et de la profondeur de chacun de ces pixels à partir des 3 sommets



- pour chaque ligne horizontale reliant deux arêtes
    - pour chaque pixel d’une ligne
        - extrapoler sa couleur
        - extrapoler sa profondeur
        - si sa profondeur est plus faible que son équivalent dans le z-buffer, alors tracer le point et affecter sa profondeur dans le z-buffer



VI. Amélioration du rendu

On parle de rendu réaliste lorsqu'une image contient la plupart des effets de lumière en interaction avec des objets physiques réels. Les travaux de recherche en ce domaine sont très nombreux et les solutions proposées sont parfois fort coûteuses suivant les effets recherchés.

Il ne faut toutefois pas perdre de vue que si le but principal est de communiquer une information, alors une image simplifiée peut être plus réussie qu'une image approchant la perfection d'une photographie : l'information n'est pas noyée dans un contexte peu pertinent pour l'observateur.

La réalité peut même parfois être intentionnellement altérée, voire même faussée, dans le but de faire encore mieux émerger le message que l'on veut transmettre : les films de science-fiction en sont un exemple flagrant lorsque les explosions dans l'espace sont accompagnées d'un effet sonore...

3. Le brouillard (fog)

La représentation 2D d'une scène 3D génère une perte d'information que l'observateur doit pouvoir reconstruire mentalement. Cette opération peut être rendue quasiment inconsciente si on utilise quelques « astuces » de rendu. La projection en perspective génère déjà la sensation de profondeur ; on peut renforcer cet effet en simulant un brouillard qui estompe les objets en fonction de leurs distances respectives au point de vue.

Le brouillard est activé (respectivement désactivé) avec :

```
glEnable(GL_FOG)
glDesable(GL_FOG)
```

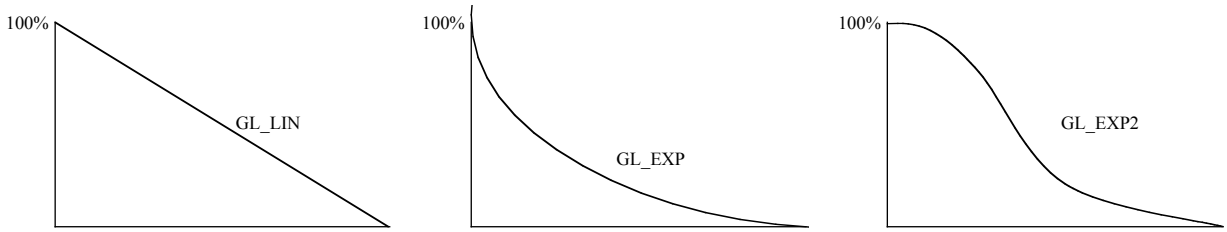
On lui associe une couleur vers laquelle tend un objet si on éloigne ce dernier du point de vue.

```
GLfloat fogColor[4] = {0.5, 0.5, 0.3, 1.} ; /* brouillard type Sirocco */
```

Cette couleur sera généralement utilisée pour le fond de la scène :

```
glClearColor(0.5, 0.5, 0.3, 1. ) ;
```

OpenGL propose trois types de courbe de mélange entre la couleur de l'objet et le brouillard : GL_LINEAR, GL_EXP et GL_EXP2



Pourcentage de la couleur d'origine d'après la distance au point de vue.

Les caractéristiques du brouillard sont définies à l'aide de la fonction `glFog`*`type`*`()` :

```
/* profil de la fonction brouillard */
glFogi(GL_FOG_MODE, GL_EXP2) ;
/* extrémités de la fonction brouillard */
glFogf(GL_FOG_START, 1.) ;
glFogf(GL_FOG_END, 5.) ;
/* coefficient de « cintrage » pour les profils EXP(2)*/
glFogf(GL_FOG_DENSITY, 0.35) ;
```

```
/* couleur du brouillard */
glFogfv(GL_FOG_COLOR, fogColor) ;
```

4. L'éclairage

OpenGL propose aussi un rendu plus réaliste avec un modèle d'illumination qui permet de prendre en compte l'orientation des surfaces par rapport aux lumières (voir annexe sur la couleur).

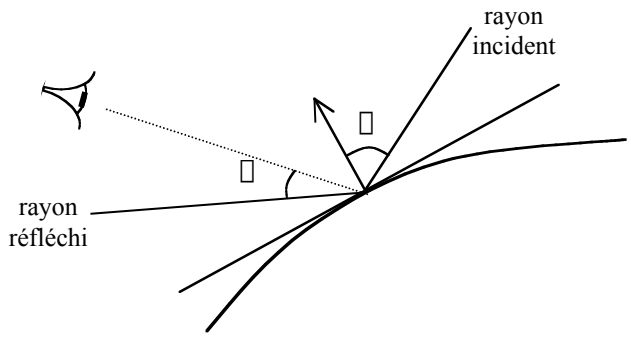
Ce rendu dépendra de :
- la position et des propriétés des éclairages,
- des propriétés optiques des matériaux employés pour la construction des objets,
- de l'orientation des surfaces vis-à-vis des éclairages et de l'observateur.

g) Un modèle physique simplifié

La lumière est représentée par la composition de trois valeurs : le rouge, le vert et le bleu. Les proportions entre ces trois valeurs vont définir une couleur que l'oeil est apte à percevoir. Nous sommes habitués à vivre avec une lumière blanche (fournie par le soleil), mais les sources peuvent être multiples (ex : éclairage d'un terrain de football générant 4 ombres sur chaque joueur) et de couleurs variées (batterie de projecteurs pour un spectacle).

Lorsqu'un rayon lumineux frappe une surface, une partie est absorbée (filtrage), une autre est réfléchie suivant les lois de la normale à la surface, le reste est restitué dans toutes les directions. OpenGL simule ces propriétés à l'aide de quatre composants (simulation d'après le modèle Lambertien, 19ème siècle) :

- l'**intensité ambiante** L_a simule une lumière qui a été dispersée par l'environnement : elle n'a pas de direction et sera réfléchie par une surface dans toutes les directions. Ainsi, une surface dans une zone d'ombre n'apparaîtra pas noire car éclairée par cette lumière ambiante.
- la **réflexion diffuse** L_d caractérise les surfaces mats. Aussi, lorsqu'un rayon lumineux provenant d'une direction particulière frappe cette surface, il sera filtré et dispersé dans toutes les directions avec une même intensité ($L_d = k_d \cos \theta$). L'effet sera donc lié à la position de la source lumineuse.
- la **réflexion spéculaire** L_s caractérise, quant à elle, le cône de réflexion de la lumière pour les surfaces brillantes. C'est elle qui va définir la brillance d'une surface lorsque l'oeil est dans l'axe symétrique de celui de la source par rapport à la normale. L'effet sera donc lié à la fois à la position de la source lumineuse et à la position de l'observateur.
- Les objets peuvent avoir une lumière **émissive** qui ajoute de l'intensité à l'objet. Par simplification, cette lumière n'ajoute pas d'éclairage supplémentaire à la scène.



$$L = L_a + k_d \cos \theta + k_s \cos^n \theta$$

Le terme n dans l'expression de la réflexion spéculaire est appelé "coefficient de surbrillance". C'est lui qui va déterminer l'étendue du reflet (saturation sur une portion de surface) que l'on peut observer sur une surface brillante lorsque certains rayons réfléchis se rapprochent de l'axe d'observation (proche de 0). Cette valeur caractérise les propriétés physiques de la surface éclairée. Un miroir parfait sera caractérisé par une valeur de n égal à l'infini : l'observateur n'est ébloui que si le rayon réfléchi coïncide avec l'axe d'observation.

h) Eclairage sous OpenGL

On passe du modèle couleur « simple » (`glColor3f(r, v, b )` au modèle d'éclairage avec :

```
glEnable (GL_LIGHTING);
glDisable (GL_LIGHTING);
```

Pour tracer une facette, il faudra définir :

- les propriétés des « lumières » (au plus 8)
- les positions des lumières
- l'interrupteur des lumières (allumer/éteindre)

- les propriétés de réflexion des matériaux (brillant, mat, …)

- le choix des faces « visibles » (avant, arrière, avant&arrière)

- le choix d'un rendu lisse où à facettes
- la normale pour chaque sommet

i) Les lumières

On peut mettre en place jusqu'à 8 lumières (`GL_LIGHT0, ..., GL_LIGHT7)` dont on peut spécifier de nombreux attributs par :

`glLightfv(GL_LIGHT0, attribut, vecteur de float)`

la position : `GL_POSITION . Sous OpenGL, une lumière est assimilée à un objet de la scène et subit les transformations géométriques définit pour GL_MODELVIEW. On peut donc la rendre fixe, la lier éventuellement à un objet, à la scène ou au point de vue : tout dépend de l'instant où on positionne cette lumière.`

```
GLfloat Lposition1 [4] = {-5.0, 0.0, 3.0, 0.0}; /* lumière a l'infini */
GLfloat Lposition2 [4] = {-5.0, 0.0, 3.0, 1.0}; /* position réelle */
glLightfv (GL_LIGHT0, GL_POSITION, Lposition1);
```

On notera dans cet exemple qu'une lumière peut être positionnée à l'infini (rayons parallèles) en mettant sa 4ème coordonnée à 0 (cf. les espaces projectifs). Les trois premières coordonnées définissent alors une direction et non plus une position.

la couleur : `GL_AMBIENT , GL_DIFFUSE` et `GL_SPECULAR` . On donne séparément les composantes ambiante, diffuse et spéculaire, ce qui permet de faire des choses peu physiques, comme des sources qui ne génèrent pas de reflets, ou que des reflets, ou qui ne contrôlent que la lumière ambiante. Chaque composante est un tableau de `float` définissant les 4 coefficients de base RVBA.

```
GLfloat Lambiant [4] = {0.4, 0.4, 0.4, 1.0};
GLfloat Lblanche [4] = {1.0, 1.0, 1.0, 1.0};

glLightfv (GL_LIGHT0, GL_AMBIENT, Lambiant);
glLightfv (GL_LIGHT0, GL_DIFFUSE, Lblanche);
```

```
glLightfv (GL_LIGHT0, GL_ SPECULAR, Lblanche);
```

Une lumière est activée et désactivée (« interrupteur ») par :

```
glEnable(GL_LIGHT0)
glDisable(GL_LIGHT0)
```

j) Matériau d'un objet

Dans un modèle d'éclairage, le rendu d'un objet n'est plus défini par une couleur brute (`glColor3f(r, v, b )`, mais par un matériau avec des propriétés de réflexion de la lumière qui sont spécifiques (cuivre, argent, peinture brillante, …). Sous OpenGL, les polygones qui seront construits recevront des propriétés optiques définies pour les 4 composantes RVBA :

- l'émission (cas d'un objet lumineux),
- la diffusion,
- la réflexion spéculaire.

Pour cette dernière, on dispose d'un coefficient qui précise la taille du reflet et son intensité (étroit et intense, ou faible et étalé).

```
GLfloat Lnoire [4] = {0.0, 0.0, 0.0, 1.0};
GLfloat mat_diffuse [4] = {0.057, 0.441, 0.361, 1.0};
GLfloat mat_specular [4] = {0.1, 0.1, 0.5, 1.0};
GLfloat mat_shininess [1] = {50.0};

glMaterialfv (GL_FRONT_AND_BACK, GL_EMISSION, Lnoire);
glMaterialfv (GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
glMaterialfv (GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
```

En pratique, on définit des fonctions qui définissent un matériau donné en regroupant ces propriétés :

```
void bronze() ;
void argent() ;
void peintureMetallisée(int couleur[3]) ;
```

Ces fonctions sont appelées juste avant le dessin des surfaces (à la place de `glColor`) pour définir la « couleur » des facettes.

k) La normale aux sommets

Le dernier élément déterminant pour la perception visuelle d'une surface est son orientation vis à vis de la source lumineuse et du point d'observation. Sous OpenGL, cette orientation est évaluée à partir du vecteur normal qui doit être de longueur 1 : cette dernière contrainte peut être gérée directement par le programmeur ou par OpenGL (généralement plus coûteux) en activant :

```
glEnable (GL_NORMALIZE);
```

La normale doit être spécifiée avant le tracé d'un polygone au moyen de :

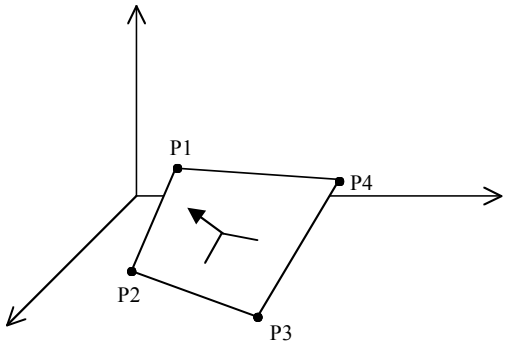
```
glNormal3f (x, y, z);
glNormal3fv(tab);
```

exemple :

```
glBegin(GL_TRIANGLES);
  glNormal3f(0., 0., 1.) ;
  glVertex3f(0., 0., 0.);
  glVertex3f(5., 0., 0.);
  glVertex3f(2.5, 5., 0.);
glEnd();
```

Si la normale n'est pas explicitement connue au moment de la programmation, il faut la calculer. Si on a pris soin de construire la liste des sommets dans l'ordre trigonométrique, les trois premiers sommets non alignés donnent deux vecteurs *u* et *v* dont le produit vectoriel donne la direction de la normale (face avant).

Rappel : $u \wedge v = [(y_u z_v - z_u y_v), -(x_u z_v - z_u x_v), (x_u y_v - y_u x_v)]$



On peut vouloir (ou non) éclairer les deux faces d'un polygones, il faut alors activer :

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE)`

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE)` *(valeur par défaut)*

Remarque : On se souvient que l'ordre dans lequel on dessine les sommets permet de définir la face avant et la face arrière d'un polygone (cf. chapitre sur les primitives). On pourrait penser que cette notion est redondante vis-à-vis de la normale. En fait, cela permet d'attribuer des propriétés spécifiques à chaque face (face pleine, face vide, couleur, matériau) alors que la normale ne servira qu'aux calculs d'éclairage comme le précise le paragraphe suivant.

l) Surfaces lisses ou « à facettes »

Bien que les surfaces « complexes » soient approximées par des données polygonales, OpenGL permet de donner à celles-ci un aspect « lisse » (modèle de Gouraud), même avec une discrétisation grossière. Une variable d'état permet de préciser si l'on souhaite un modèle de surface à facettes ou avec dégradé de couleur (plus lourd en calcul !) :

`glShadeModel(GL_FLAT)` *(plus rapide)*

`glShadeModel(GL_SMOOTH)` *(plus joli)*

En fait , les normales sont affectées aux sommets et non pas au polygone. Dans l'exemple ci-dessous, les trois sommets du triangles reçoivent la même normale `(0., 0., 1.)` : le triangle aura une couleur uniforme.

```
glBegin(GL_TRIANGLES);
/* définitions de la normale pour tous les sommets qui suivent : */
  glNormal3f(0., 0., 1.) ;
  glVertex3f(0., 0., 0.);
  glVertex3f(5., 0., 0.);
  glVertex3f(2.5, 5., 0.);
glEnd();
```

Pour obtenir un aspect non facetté, on affecte à chaque sommet la moyenne des normales des facettes voisines. OpenGL effectuera une extrapolation de la lumière réfléchie pour chaque pixel à partir des valeurs calculées aux sommets.

Eclairage d'un « cratère » avec et sans lissage

Schéma type du tracé d'une surface lissée à partir d'un maillage de points (`glFloat Point3D[3]`) :

```
glBegin(GL_TRIANGLES);
for (i=0 ; i<Longueur-1 ; i++)
 for (j=0 ; j<Largeur-1 ; j++)
 { /* 1 carreau = 2 triangles */
   glNormal3fv(Nsurface[i][j]);
   glVertex3fv(Surface[i][j]);
   glNormal3fv(Nsurface[i][j+1]);
   glVertex3fv(Surface[i][j+1]);
   glNormal3fv(Nsurface[i+1][j+1]);
   glVertex3fv(Surface[i+1][j+1]);

   glNormal3fv(Nsurface[i][j]);
   glVertex3fv(Surface[i][j]);
   glNormal3fv(Nsurface[i+1][j]);
   glVertex3fv(Surface[i+1][j]);
   glNormal3fv(Nsurface[i+1][j+1]);
   glVertex3fv(Surface[i+1][j+1]);
 }
}
glEnd();
```

Remarques :

- Lorsqu'une surface est représentée par un treillis de points, les sommets et les normales sont généralement rangés dans un tableau : cela facilite grandement l'évaluation des normales moyennes avant de dessiner la surface.
- Pour affiner le rendu, on pourra pondérer les normales de chaque facette par les secteurs angulaires respectifs.

VII. Images 2D

19. L'initialisation du 2D

Une image 2D est un tableau rectangulaire de pixels ayant chacun une « valeur » spécifique. Son affichage ne nécessitera pas de tampon de profondeur. Une initialisation courante se traduira dans la fonction *main* par un des deux appels ci-dessous :

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB) ;
```

ou

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

Bien que l'affichage se fasse en mode RGB (tampon image), on pourra disposer de plusieurs types de codage d'image en mémoire centrale. Nous nous limiterons ici aux :

- images RGB où un pixel est codé sur 3 octets consécutifs,
- images en niveaux de gris où un pixel est codé sur un octet.

On notera au passage que les données initiale d'une image peuvent avoir une dynamique bien supérieure au capacité d'affichage d'une carte graphique. Par exemple, une image IRM en imagerie médicale aura souvent des valeurs dans [-2048, 2047]. L'affichage d'une telle image demande un rééchelonnage préalable des amplitudes entre [0, 255] .

20. Matrice de projection

Il peut paraître étrange de définir une matrice de projection lorsque l'on manipule uniquement du 2D. Cette opération est pourtant nécessaire pour que la position d'un pixel image coïncide avec une coordonnée écran (entière). On utilise une matrice de projection spécifique au 2D :

```
gluOrtho2D(xmin, xmax, ymin, ymax)
```

Aussi, le cadrage d'une image aura la forme suivante :

```
void monCadrage(int large, int haut)
{ glViewport(0, 0, large, haut) ;
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  gluOrtho2D(0, large, 0, haut) ;
  glMatrixMode(GL_MODELVIEW) ;
}
```

21. Affichage d'une image 2D

Le prochain dessin d'une image (matrice rectangulaire de pixel) sera positionné dans la fenêtre en définissant l'angle inférieur gauche *(x0, y0)* :

```
glRasterPos2i(x0, y0) ;
```

OpenGL propose trois commandes de base pour manipuler des images :

- `glDrawPixels` qui recopie un tableau de pixels dans le tampon image (cf. `glRasterPos2i`),
- `glReadPixels` qui recopie une partie du tampon image dans un tableau,
- `glCopyPixels` qui recopie une zone à l'intérieur du tampon image (sans passer par la `CPU !`).

glDrawPixels(*largeur, hauteur, format, type, tableau*)
glReadPixels(*largeur, hauteur, format, type, tableau*)

`largeur, hauteur` : définissent la taille de l'image en terme de pixels,
`format` : format d'un pixel en mémoire centrale. On se limitera à **GL_RGB** ou **GL_LUMINANCE**,
`type` : on se limitera à **GL_UNSIGNED_BYTE** (un octet non signé),
`tableau` : adresse du tableau respectant le type précédemment définit.

Exemple :

```
#define hauteur 200
#define largeur 256
GLubyte imageRGB[hauteur][largeur][3] ;
GLubyte imageGRIS[hauteur][largeur] ;
```

affichage de `imageRGB`

```
glRasterPos2i(0, 0) ;
glDrawPixels(largeur, hauteur, GL_RGB, GL_UNSIGNED_BYTE, imageRGB) ;
glutSwapBuffers() ;
```

affichage de `imageGRIS`

```
glRasterPos2i(0, 0) ;
glDrawPixels(largeur, hauteur, GL_LUMINANCE, GL_UNSIGNED_BYTE, imageGRIS) ;
glutSwapBuffers() ;
```

glCopyPixels(*x0, y0, largeur, hauteur,* GL_COLOR)

La recopie s'effectue à partir de la trame active (cf. `glRasterPos2i`).
`x0 et y0` : Précise le coin inférieur gauche du rectangle (`largeur, hauteur`) de pixels à recopier.
Le dernier argument précise le buffer sur lequel l'opération est effectuée. Nous nous limiterons ici à **GL_COLOR**.

22. Image et processeur…

a. Alignement des octets

Les processeurs sont performants pour manipuler des mots machines : Pentium et G4 sont des processeurs 32 bits « grand public », mais nul doute sur l'arrivée prochaine des 64 bits…

Les fonctions `glDrawPixels` et `glReadPixels` tiennent compte de l'architecture du processeur et effectuent, par défaut, les transferts par paquets d'octets correspondants au mot machine. En particulier, elles démarrent chaque ligne image à la première adresse multiple d'un mot machine qui suit la fin de la précédente ligne. En conséquence, la largeur d'une image doit occuper un nombre d'octets multiple d'un mot machine, au risque d'obtenir un affichage avec une déformation latérale de l'image parce que le début de chaque ligne aura été artificiellement décalé.

Le contrôle des transferts se fait avec :

`glPixelStorei(GL_UNPACK_ALIGNMENT, taille)` pour `glDrawPixels`
`glPixelStorei(GL_PACK_ALIGNMENT, taille)` pour `glReadPixels`

où `taille` définit le nombre d'octets dans un paquet (1, 2, 4 ou 8).

Une solution sûre mais peu efficace est de forcer la taille des paquets à 1 pour qu'une ligne image contienne nécessairement un nombre entier de paquets d'octets.

Une autre solution consiste à définir une matrice image en mémoire dont la taille des lignes est un multiple de 4 (taille d'un mot machine pour les processeurs actuels), quitte à ne pas remplir les derniers octets de chaque ligne.

b. Ordre des octets

Là encore, la diversité existe et on constate qu'un mot machine peut ordonner ses octets dans un sens ou dans l'autre. Par exemple, l'octet de poids fort pour un G4 se trouve à la place de l'octet de poids faible pour un Pentium et réciproquement. L'exploitation sur une machine d'une image construite sur une autre machine peut demander au préalable quelques opérations de permutation.

23. Primitives graphiques 2D

Dans certaines applications graphiques, on peut souhaiter effectuer un tracé par dessus une image que l'on vient d'afficher. Par exemple, on peut tracer en rouge une route sur une image satellitaire pour la mettre en évidence.

Dans ce cas, on affichera l'image en premier et on utilisera ensuite les primitives de tracé 2D avec le suffixe 2i, principalement **`glVertex2i(x, y)`** .

Il sera aussi parfois plus commode de définir les couleurs de tracé en entier :
`glColor3ub(rouge, vert, bleu)`