OpenGL et GLUT
Une introduction

Edmond.Boyer@imag.fr

Résumé

Dans ce document, nous introduisons la librairie graphique OpenGL ainsi que la librairie GLUT qui lui est associée. Les concepts ainsi que quelques fonctionnalités de base sont présentés au travers d'exemples. L'objectif est de pouvoir programmer rapidement de petites applications graphiques à l'aide de ces outils.

Table des matières

1 Introduction

OpenGL

OpenGL [Ope] est une librairie graphique, c'est à dire une interface logiciel permettant d'effectuer des opérations d'affichage sur un écran graphique. Cette interface, développée par Silicon Graphics, est constituée d'environ 150 fonctions graphiques, de l'affichage de points et segments jusqu'au plaquage de textures sur des objets tridimensionnels.

OpenGL présente l'intérêt majeur d'être indépendante du matériel utilisé, donc de l'architecture sur laquelle l'application graphique est développée. De fait, OpenGL n'a pas de fonctions de contrôle du matériel, en sortie ou en entrée. Le développeur doit, soit utiliser directement les fonctions du système sur lequel il développe son application (Xwindow par exemple pour Unix), soit utiliser une interface logiciel située en amont d'OpenGL et qui propose des fonctions pour cela (Glut par exemple). De manière équivalente, OpenGL ne propose pas de fonctions graphiques de haut niveau utilisant des primitives géométriques complexes (sphères, splines, etc.). La réalisation des ces fonctions est laissée au développeur ou à l'interface logiciel en amont d'OpenGL. Cette spécification très ciblée de la librairie lui assure une portabilité importante et a fait d'OpenGL le standard graphique actuel.

En résumé OpenGL est une librairie graphique de bas niveau utilisant les primitives géométriques de base : points, segments et polygones. L'interfaçage avec le matériel (carte graphique, mémoire vidéo, etc.) ou avec des commandes de haut niveau (modèles 3D) doit être ajouté pour construire une application graphique.

GLUT

GLUT [Glu] est une librairie d'outils (Utility Toolkit) permettant l'interfaçage de la librairie OpenGL avec plusieurs architectures matérielles : stations de travail (SGI, SUN, ...), PCs, etc. GLUT propose un ensemble de fonctions pour la gestion de fenêtres, de la souris et du clavier. Ces fonctions sont simples à utiliser et permettent de réaliser facilement de petites applications graphiques. Par ailleurs, GLUT propose certaines primitives de haut niveau (sphère, cylindre par exemple). GLUT reste par contre limitée à la réalisation d'applications graphiques simples, permettant de tester et/ou de valider des algorithmes graphiques. Pour la réalisation d'applications graphiques plus complexes, le développeur aura intérêt à utiliser des librairies plus complètes (OpenInventor par exemple).

2 Construire des images avec OpenGL

2.1 La syntaxe d'OpenGL

La syntaxe d'OpenGL caractérise les constantes, types et fonctions de la manière suivante :

- les constantes : **GL_CONSTANTE** (GL_COLOR_BUFFER_BIT par exemple);
- les types : **GLtype** (GLbyte, GLint par exemple);
- les fonctions : **glLaFonction** (glDraw, glMatrixMode par exemple).

À cela s'ajoute, dans la syntaxe des fonctions, la caractérisation du nombre et du type des arguments par un suffixe. Par exemple :

glVertex3f(1.0, 2.0, 3.0);

définit les coordonnées dans l'espace d'un sommet (vertex) en simple précision réelle. Pour définir les coordonnées du plan d'un sommet par des valeurs entières, on utilise :

glVertex2i(1, 2);

Le tableau suivant donne les différents suffixes et les types correspondants :

suffixe	précision	type C	type OpenGL
b	entier 8 bits	char	GLbyte
s	entier 16 bits	short	GLshort
i	entier 32 bits	int/long	GLint, GLsizei
f	réel 32 bits	float	GLfloat
d	réel 64 bits	double	GLdouble
ub	entier non signé 8 bits	unsigned char	GLubyte
us	entier non signé 16 bits	unsigned short	GLushort
ui	entier non signé 32 bits	unsigned int	GLuint

Enfin les commandes d'OpenGL qui prennent en argument un tableau sont caractérisées par un suffixe se terminant par la lettre *v* (pour vector). Par exemple :

GLfloat coordonnees[3];
GLVertexfv(coordonnees);

permet de définir les coordonnées d'un sommet sous la forme d'un tableau de réels.

2.2 Le principe

OpenGL dessine dans une image tampon. Cette image tampon peut être soit directement la mémoire vidéo de la fenêtre graphique, soit une image tampon intermédiaire permettant de faire du "double buffering"[1].

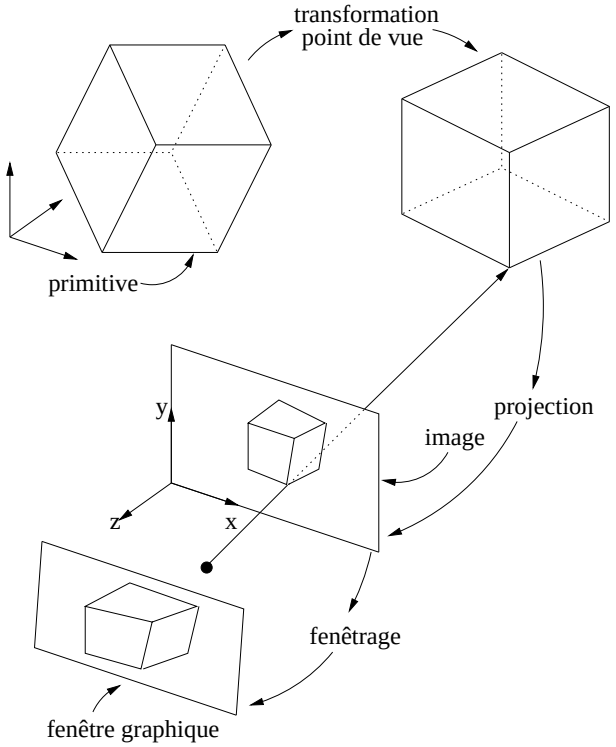


Figure 1: La construction d'une image.

Les différentes étapes de la génération d'une image sont (voir figure 1) :

1. spécification des primitives à dessiner : les primitives sont définies dans un certain repère,

2. transformation *point de vue* : une transformation est appliquée aux primitives à dessiner. Cette transformation sert à fixer le point de vue des primitives ou, en d'autres termes, la position du plan image.

[1] le double buffering consiste à dessiner, tout d'abord, dans une image tampon puis a remplacer l'ensemble de l'écran par cette image. Effectuer ces deux étapes successivement au lieu de dessiner directement à l'écran rend les animations plus fluides.

3. Projection : les primitives sont projetées sur le plan image suivant la projection spécifiée (orthographique, perspective).

4. Fenêtrage et numérisation : l'image obtenue est redimensionnée suivant les tailles de la fenêtre graphique et les primitives projetées sont numérisées sous la forme de pixels dans la mémoire vidéo.

Ces quatre étapes sont réalisées directement à l'appel d'une fonction de dessin d'OpenGL et ne nécessitent pas quatre appels spécifiques. En effet, les spécifications du point de vue, de la projection et du fenêtrage se font de manière indépendante de la spécification des primitives. Cela veut dire qu'à chaque appel d'une fonction de dessin, la transformation *point de vue* courante et la projection courante sont appliquées aux primitives. La figure 2 montre, par exemple, ce qui est effectué lors de l'appel d'une fonction de dessin d'une ligne polygonale.

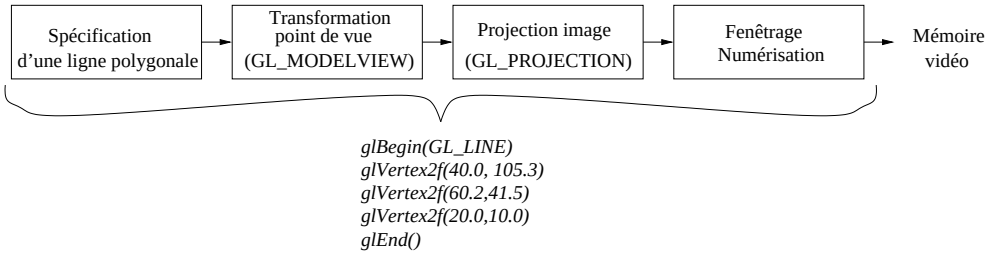


Figure 2: Le dessin d'une ligne polygonale avec OpenGL.

Les paragraphes suivant explicitent les spécifications de ces différentes étapes.

2.3 Spécifier la transformation *point de vue*

Les transformations sont représentées dans OpenGL sous la forme de matrices. Spécifier la transformation courante appliquée aux primitives avant la projection consiste donc à définir la matrice de cette transformation. OpenGL utilise les coordonnées homogènes pour effectuer des transformations. La transformation *point de vue* est donc représentée par une matrice 4x4. De plus OpenGL stocke les matrices dans des piles. Ainsi, la transformation *point de vue* correspond, dans OpenGL, non pas à une matrice mais à une pile de matrice. De cette manière, OpenGL appliquera l'ensemble des transformations empilées aux primitives à dessiner avant de les projeter.

Pour modifier la transformation *point de vue*, il faut tout d'abord sélectionner la pile des transformations *point de vue*, puis ensuite modifier la transformation en

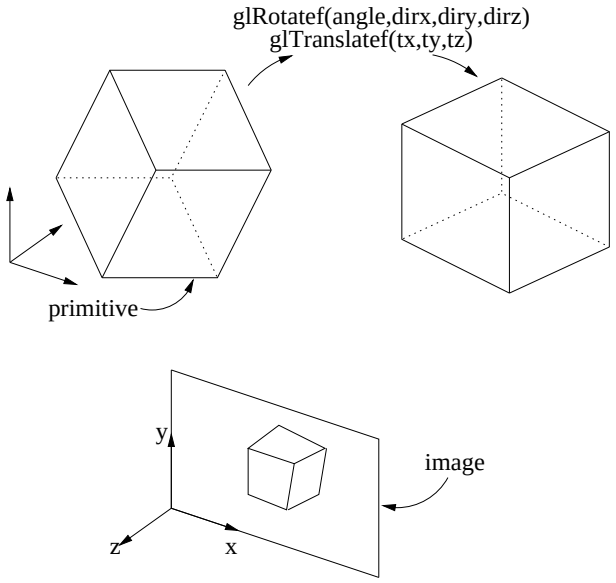


Figure 3: La transformation point de vue.

conséquence.

```
glMatrixMode(GL_MODELVIEW);     /* la pile des transformations point */
                                /* de vue est selectionnee */
glLoadIdentity();               /* initialisation de la transformation */
glRotatef(angle,                /* rotation de angle degres autour de */
          dirx, diry, dirz);    /* l'axe de direction (dirx, diry, dirz)*/
glTranslatef(tx, ty, tz);       /* translation de vecteur (tx,ty,tz) */
```

Algorithme de modification de la transformation *point de vue* (voir figure 3).

Quelques remarques :

- les rotations s'expriment en degrés.
- Sans appel de la fonction **glLoadIdentity()**, la rotation et la translation définies ensuite s'ajoutent à la transformation courante.
- Les modifications sont appliquées à la matrice en haut de pile alors que la transformation *point de vue* comprend l'ensemble des matrices empilées.

2.3.1 Ajouter une transformation pour le calcul du point de vue

Pour ajouter une matrice à la pile courante (GL_MODELVIEW, GL_PROJECTION, ...) ou en enlever une, on utilise les fonctions **glPushMatrix()**et **glPopMatrix()**. Cela peut être utilisé pour appliquer une transformation supplémentaire à une primitive donnée (voir figure 4). Par exemple, l'algorithme suivant trace la projection d'un cube auquel est appliqué une rotation puis les transformations contenues dans la pile *point de vue*. Ensuite le même cube est projeté après avoir subi les transformations *point de vue* sans la rotation précédente :

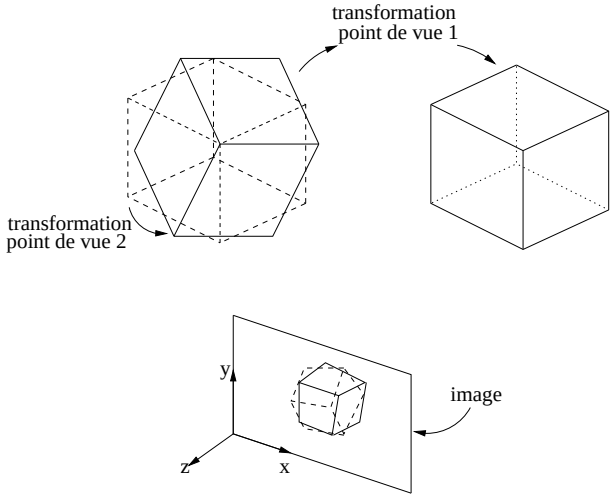


Figure 4: Empilement des matrices de transformation point de vue pour modifier position et orientation des primitives.

```
glMatrixMode(GL_MODELVIEW);     /* la pile des transformations point */
                                /* de vue est selectionnee */
glPushMatrix();                 /* selectionne une nouvelle matrice */
                                /* courante */
glRotatef(angle, dirx,
          diry, dirz);          /* transformation point de vue 2 */
glutWireCube(1.0);              /* dessin du cube */
glPopMatrix();                  /* effacement de la matrice courante */
                                /* et retour a la matrice precedente */
                                /* dans la pile */
glutWireCube(1.0);              /* 2eme dessin du cube */
```

Algorithme de transformation d'une primitive.

2.4 Modifier une transformation

Les fonctions de modifications d'une transformation s'appliquent à la matrice courante (haut de pile) de la transformation courante (pile courante). Ces fonctions prennent en paramètres des réels simple ou double précision (glRotatef(), glRotated() respectivement). Soit M la matrice 4x4 courante, alors :

`glRotate{fd}(angle,dirx, diry,dirz);` `/* rotation de angle degre autour de l'axe de direction*/`

$$M = M \cdot \begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`glTranslate{fd}(tx,ty,tz);`

$$M = M \cdot \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`glScale{fd}(fx,fx,fz);`

$$M = M \cdot \begin{pmatrix} fx & 0 & 0 & 0 \\ 0 & fy & 0 & 0 \\ 0 & 0 & fz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`glLoadMatrix{fd}(T);`

$$M = \begin{pmatrix} T[0] & T[4] & T[8] & T[12] \\ T[1] & T[5] & T[9] & T[13] \\ T[2] & T[6] & T[10] & T[14] \\ T[3] & T[7] & T[11] & T[15] \end{pmatrix}$$

`glMultMatrix{fd}(T);`

$$M = M \cdot \begin{pmatrix} T[0] & T[4] & T[8] & T[12] \\ T[1] & T[5] & T[9] & T[13] \\ T[2] & T[6] & T[10] & T[14] \\ T[3] & T[7] & T[11] & T[15] \end{pmatrix}$$

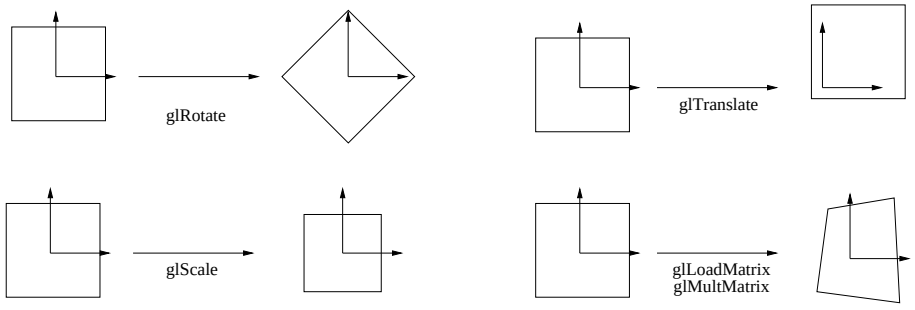


Figure 5: Modifications d'une transformation.

2.5 Spécifier la projection

De la même manière que la transformation *point de vue*, la projection appliquée ensuite aux primitives est définie, dans OpenGL, à l'aide d'une matrice 4x4. L'usage des coordonnées homogènes permet en effet de représenter les projections orthographiques et perspectives sous la forme de matrices. Ces deux types de projections sont accessibles dans OpenGL. Il est à noter que les matrices de projection sont, comme dans le cas des transformations *point de vue*, empilées. Toute modification apportée à la projection concerne la matrice de projection courante, située donc en haut de pile.

La spécification d'une projection commence classiquement par l'appel des deux fonctions suivantes :

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

La première opération sélectionne la pile des matrices de projection, la deuxième ré-initialise la matrice de projection et assure ainsi qu'une seule projection soit effectuée. Les opérations sur les matrices (transformations, empilement, dépilement) effectuées ensuite affecteront la matrice de projection courante.

2.5.1 Projection orthographique

Pour définir une projection image de type orthographique, soit une projection suivant la direction orthogonale au plan image, il faut définir la direction de projection. Dans OpenGL, la direction de projection est celle de l'axe des z du repère correspondant au point de vue (soit après la transformation *point de vue*). Pour spécifier une projection orthographique, OpenGL fournit la fonction :

glOrtho(gauche, droit, bas, haut, proche, loin)

Tous les paramètres étant ici de type **GLdouble**. Cette fonction prend en argument la position et les tailles du volume d'observation (voir figure 6). La matrice de projection correspondante est (où d représente la valeur droit, g la valeur gauche, etc.) :

$$\begin{pmatrix} \frac{2}{d-g} & 0 & 0 & \frac{d+g}{d-g} \\ 0 & \frac{2}{h-b} & 0 & -\frac{h+b}{h-b} \\ 0 & 0 & -\frac{2}{l-p} & \frac{l+p}{l-p} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Bien que non caractéristique d'une projection, le volume d'observation permet de définir une région de visibilité dans l'espace. Une primitive sera donc présente dans l'image si, et seulement si, elle appartient au volume d'observation. Dans le cas d'une projection orthographique, les primitives sont projetées suivant la direction orthogonale au plan image. Définir un volume d'observation consiste donc à spécifier les coordonnées d'un parallélépipède rectangle.

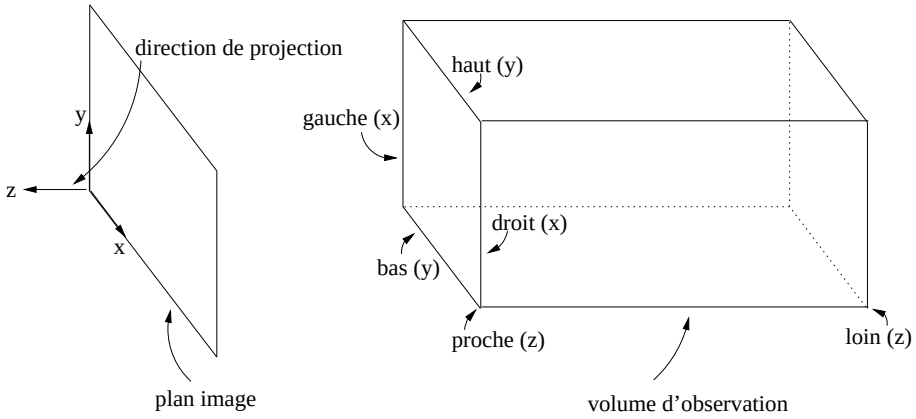


Figure 6: Projection orthographique : gauche et droit sont les coordonnées suivant l'axe des x du volume d'observation, haut et bas suivant l'axe des y et proche et loin suivant l'axe des z orienté négativement.

En résumé, pour la spécification d'une projection orthographique :

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(gauche, droit, bas, haut, proche, loin);
```

2.5.2 Projection perspective

Pour définir une projection de type perspectif, il faut définir un centre de projection, une distance focale (distance du centre de projection au plan de projection) et une direction de projection. Dans OpenGL, la direction de projection est toujours suivant l'axe des z. Le centre de projection est à l'origine du repère courant

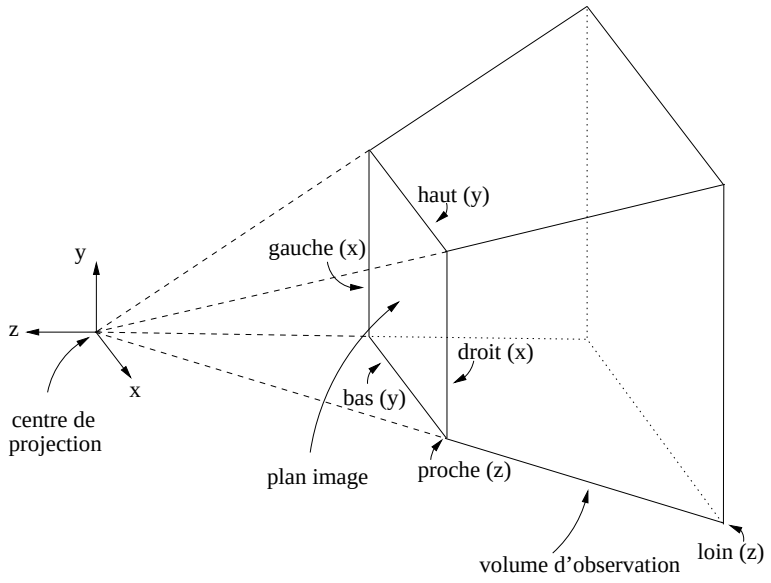


Figure 7: Projection perspective : gauche et droit sont les coordonnées suivant l'axe des x du volume d'observation, haut et bas suivant l'axe des y et proche et loin suivant l'axe des z orienté négativement.

et la distance focale est paramétrable. Pour spécifier une projection perspective, OpenGL fournit la fonction :

glFrustum(gauche, droit, bas, haut, proche, loin)

Tous les paramètres étant ici de type **GLdouble**. La matrice de projection correspondante est (où d représente la valeur droit, g la valeur gauche, etc.) :

$$\begin{pmatrix} \frac{2p}{d-g} & 0 & \frac{d+g}{d-g} & 0 \\ 0 & \frac{2p}{h-b} & \frac{h+b}{h-b} & 0 \\ 0 & 0 & -\frac{p+l}{l-p} & -\frac{2lp}{l-p} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

La fonction glFrustum() prend en argument, comme dans le cas orthographique, les tailles et position du volume d'observation. Encore une fois, ce volume n'est pas caractéristique de la projection mais permet de définir une région de visibilité. Celui-ci est, dans le cas perspectif, une pyramide tronquée. Le plan image de la projection correspondant à une des faces de la pyramide tronquée (voir la figure 7). La distance focale de la projection est donc ici égale à la valeur de la variable

proche.

Une autre façon de spécifier une projection perspective consiste à utiliser la fonction de la librairie GLU :

gluPerspective(fov, rapport, proche, loin)

Cette fonction sert à spécifier une matrice de projection perspective de la même manière que précédemment, c'est à dire par un volume d'observation. *fov* est l'angle (en degré) du champs d'observation (voir figure 8), *rapport* est le rapport d'échelle entre la largeur et la hauteur du volume d'observation, enfin *proche* et *loin* sont les distances entre le centre de projection et les plans proche et lointain du volume d'observation.

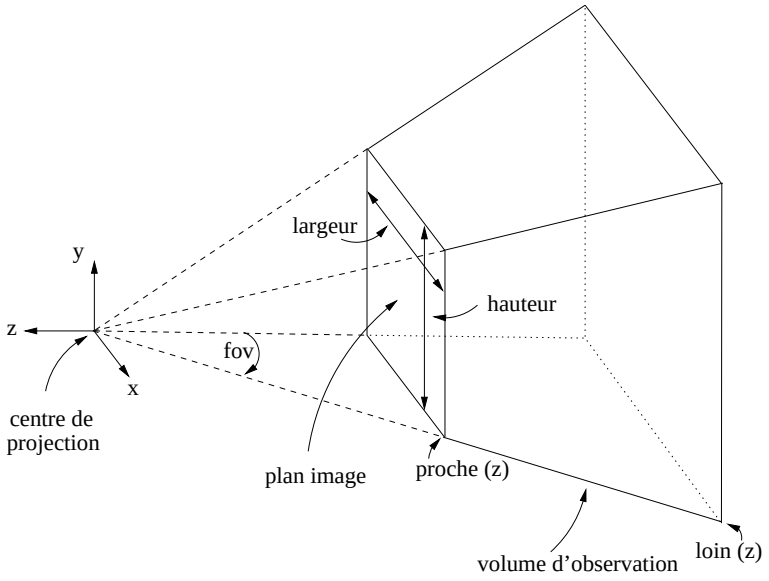


Figure 8: Projection perspective : le volume d'observation est défini avec la fonction gluPerspective(). fov est l'angle du champs d'observation dans le plan $x - z$, rapport est le rapport d'échelle largeur/hauteur, proche et loin sont les distances (positives) au plans du volume d'observation.

En résumé, pour la spécification d'une projection perspective :

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(gauche, droit, bas, haut, proche, loin);
```

```
//  gluPerspective(angle,rapport, proche, loin); /* angle [0 .. 180] */
                                                 /* rapport = largeur/haut
```

2.6 Spécifier le fenêtrage

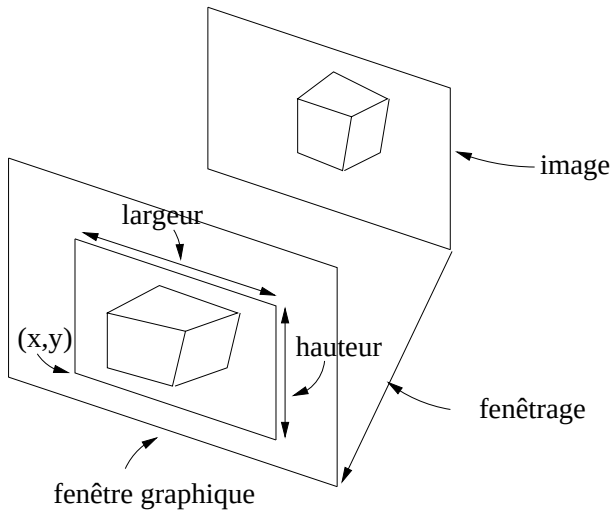


Figure 9: Fenêtrage : permet de définir la fenêtre d'affichage à l'intérieur de la fenêtre graphique.

Le fenêtrage s'applique à la suite de la transformation point de vue et de la projection. Il consiste à redimensionner l'image obtenue aux dimensions que l'on souhaite, celles de la fenêtre d'affichage. Par défaut OpenGL redimensionne l'image aux dimensions de la fenêtre graphique au moment de sa création. Comme les dimensions de cette fenêtre graphique peuvent être modifiées à tout moment par l'utilisateur à l'aide du gestionnaire de fenêtres (window manager), il est utile de disposer d'une fonction permettant de modifier les dimensions de la fenêtre d'affichage. OpenGL fournit la fonction :

glViewport(x, y, largeur, hauteur)

les paramètres étant de type entier. Cette fonction définit une fenêtre d'affichage à l'intérieur de la fenêtre graphique. Cette fenêtre d'affichage est positionnée en (x, y) et a pour largeur $largeur$ et hauteur $hauteur$ en pixels (voir la figure 9).

L'intérêt majeur de cette fonction est de pouvoir suivre les évolutions de la fenêtre graphique lorsque celle-ci est redimensionnée par l'utilisateur. Des exemples d'utilisation sont donnés dans la partie de ce document concernant la librairie

GLUT. On notera par contre que pour éviter des distorsions à l'affichage, il faut conserver le rapport d'échelle du volume d'observation. Ce dernier est défini par le rapport entre la largeur du volume du volume d'observation et sa hauteur ; le rapport entre la largeur de la fenêtre d'affichage et sa hauteur doit donc être équivalent.

2.7 Spécifier des couleurs

Le mode de couleur courant dans OpenGL est le mode RGBA : RGB pour les composantes rouge, vert et bleu d'une couleur, et A pour la composante alpha qui peut servir à fixer la transparence par exemple. De la même manière qu'OpenGL conserve des transformations courantes : projections, ..., OpenGL conserve une couleur courante qui sera appliquée à toutes les primitives à dessiner. Pour spécifier ou modifier cette couleur, il faut utiliser la fonction **glColorxx(...)** avec en paramètres les valeurs des différentes composantes de la couleur souhaitée.

```
glColor3f(0.5,0.5,0.5);       /* couleur (0.5,0.5,0.5,1.0) */
glColor4f(1.0,1.0,1.0,1.0);
```

Les composantes sont fixées à 1.0 par défaut. Si la couleur est spécifiée par trois valeurs, celles-ci sont affectées aux composantes RGB de la couleur et la composante A est fixée à 1.0. Lorsque la couleur est spécifiée par des réels, ceux-ci doivent être de valeurs comprises entre 0 et 1, les valeurs à l'extérieur de cet intervalle sont ramenées aux limites de cet intervalle. Pour les autres types : entier i, entier non signées ui, etc, les valeurs sont linéairement interpolées sur l'intervalle $[-1, 1]$. Par exemple, les *unsigned byte ub* sont convertis suivant : $[0, 255] \rightarrow [0, 1]$ et les *byte b* sont convertis suivant : $[-128, 127] \rightarrow [-1, 1]$. Comme pour les réels, les valeurs négatives pour les autres types correspondent à la valeur 0.

2.8 Spécifier des primitives géométriques

OpenGL fournit plusieurs fonctions pour définir des primitives. Nous ne présentons ici que quelques fonctions parmi les plus significatives : points, lignes et polygones. Les primitives sont toujours tridimensionnelles dans OpenGL et l'ensemble des transformations courantes : point de vue, projection, etc., leurs sont appliquées. La spécification d'une primitive consiste à définir tout d'abord le type de la primitive (GL_POINTS ou GL_LINE par exemple) puis les sommets (vertices) qui la constituent. Un sommet peut être spécifié par deux, trois ou quatre coordonnées correspondant, respectivement, à des coordonnées dans le plan $z = 0$, des coordonnées dans l'espace et des coordonnées homogènes. La structure générale de la spécification de ces primitives est la suivante :

```
glBegin(GL_POINTS - GL_LINES - GL_TRIANGLES - GL_POLYGON);
glVertex3f(x1,y1,z1);     /* le sommet 1 (x1,y1,z1) dans l'espace */
glVertex2i(x2,y2);        /* le sommet 2 (x2,y2,0) */
glVertex4f(x3,y3,z3,w3);  /* le sommet 3 (x3/w3, y3/w3, z3/w3) */
...
glEnd();
```

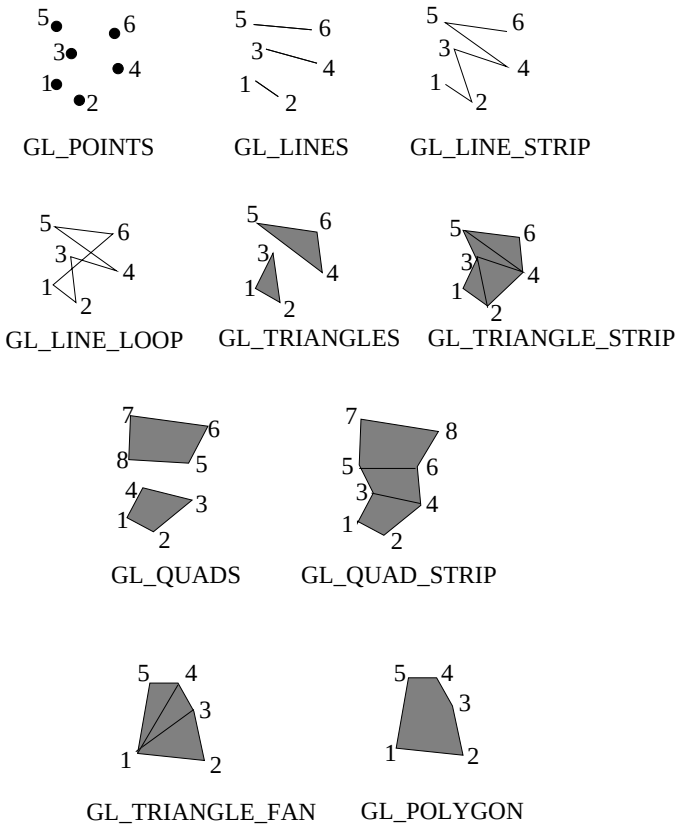


Figure 10: Les primitives géométriques d'OpenGL.

OpenGL trace, par défaut, des points et lignes d'épaisseurs égales à un pixel ainsi que des polygones remplis. Ces paramètres par défaut sont modifiables, les paragraphes suivants explicitent ces différents points.

2.8.1 Dessiner des points

L'exemple ci-dessous montre comment dessiner des points. La spécification de la couleur courante (r, g, b) se fait par appel de la fonction **glColorxx()**. Cette spécification reste valide jusqu'à l'appel suivant de **glColorxx()**. La taille des points est spécifiée par un réel à l'aide de la fonction **glPointSize()**. L'appel de cette dernière doit être à l'extérieur de **glBegin()-glEnd()**.

```
glPointSize(s);
glBegin(GL_POINTS);
glColor3f(r,g,b);
glVertex4f(x,y,z,w);        /* le point (x/w,y/w,z/w) dans l'espace */
glVertex3f(x,y,z);          /* le point (x,y,z) dans l'espace */
glVertex2i(x,y);            /* le point (x,y,0) dans l'espace */
glEnd();
```

2.8.2 Dessiner des lignes polygonales

De manière similaire, les lignes polygonales se dessinent à l'aide de **glBegin()-glEnd()**. Pour dessiner des segments de droites joignant les paires de points successives :

```
glLineWidth(s);
glColor3f(r,g,b);
glBegin(GL_LINES)
glVertex3f(x1,y1,z1)
glVertex3f(x2,y2,z2)   /* Segments (1,2) et (3,4) */
glVertex3f(x3,y3,z3)
glVertex3f(x4,y4,z4)
glEnd()
```

Pour dessiner une ligne polygonale entre les points 1,2 et 3 :

```
glBegin(GL_LINES_STRIP)
glVertex3f(x1,y1,z1)
glVertex3f(x2,y2,z2)   /* Segments (1,2) et (2,3) */
glVertex3f(x3,y3,z3)
glEnd()
```

Pour dessiner une ligne polygonale fermée entre les points 1,2 et 3 :

```
glBegin(GL_LINES_LOOP)
glVertex3f(x1,y1,z1)
```

```
glVertex3f(x2,y2,z2)   /* Segments (1,2), (2,3) et (3,1) */
glVertex3f(x3,y3,z3)
glEnd()
```

2.8.3 Dessiner un polygone

Le dessin d'un polygone fait intervenir des paramètres supplémentaires. En particulier un polygone a deux faces et peut être dessiné sous la forme de points, segments, ou faces remplies. Les faces d'un polygone sont définies par l'orientation de ce dernier et donc par l'ordre dans lequel sont spécifiés les sommets. Un polygone ne doit pas s'intersecter lui-même et doit être convexe pour être rempli. Par défaut, un polygone est dessiné les deux faces remplies. Ces paramètres peuvent être modifiés par appel de la fonction **glPolygonMode()**. La procédure standard de dessin d'un polygone est la suivante :

```
glPolygonMode(GL_FRONT - GL_BACK - GL_FRONT_AND_BACK,
              GL_POINT - GL_LINE - GL_FILL);
glColor3f(r,g,b);
glBegin(GL_POLYGON);
glVertex3f(x,y,z);
...
glEnd();
```

Les deux faces d'un polygone peuvent être traitées de manières différentes en effectuant deux appels successifs à la fonction **glPolygonMode()** :

```
glPolygonMode(GL_FRONT, GL_LINE);  /* face frontale segments */
glPolygonMode(GL_BACK, GL_FILL);   /* face arriere remplie */
```

Par ailleurs, le remplissage effectué sur un polygone est fonction du modèle d'ombrage courant : plat ou lisse. Dans le cas d'un ombrage plat, la couleur est constante à l'intérieur du polygone. Dans le cas d'un ombrage lisse (ombrage de Gouraud ici), la couleur à l'intérieur du polygone est interpolée entre les couleurs des sommets du polygone. Pour modifier le modèle d'ombrage courant, il faut utiliser la fonction **glShadeModel(GL_FLAT - GL_SMOOTH)** (voir le paragraphe 2.11.2 sur les ombrages).

2.9 Afficher une image

La fonction d'affichage d'une image (ou, de manière générale, d'une matrice de pixels) est la fonction **glDrawPixels()**. Cette fonction prend en paramètres :
glDrawPixels(ImgWidth, ImgHeight, ImgFormat, DataType, Image)

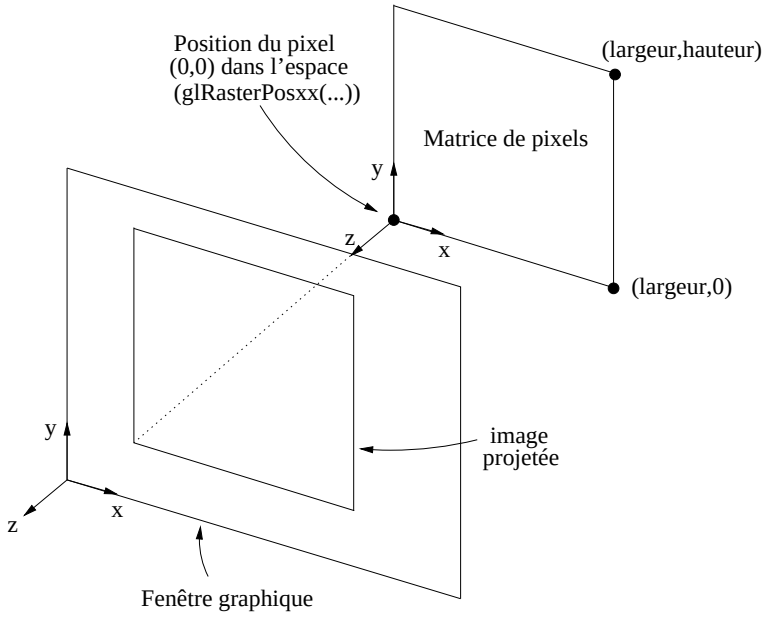


Figure 11: L'affichage d'une image à l'aide de la fonction glDrawPixels() dans le cas d'une projection orthographique.

- ImgWidth et ImgHeight : la largeur et la hauteur de l'image qui sont de type GLint.
- ImgFormat : le format de l'image qui est principalement[2] une des constantes suivantes : GL_RGB (couleur), GL_RGBA (couleur + alpha), GL_RED (composante rouge seulement), GL_GREEN (composante verte), GL_BLUE (composante bleu), GL_LUMINANCE (composante blanche).
- DataType : le format des données qui est principalement[2] une des constantes suivantes : GL_UNSIGNED_BYTE (entiers non signés sur 8 bit), GL_BYTE, GL_BITMAP (bits codés dans des unsigned bytes), GL_SHORT (entiers signés sur 16 bits), GL_UNSIGNED_SHORT (entiers non signés sur 16 bits).
- Image : la matrice de pixels de format DataType.

Comme cela a été dit précédemment, les primitives dans OpenGL sont toujours de nature tri-dimensionnelle et sont donc transformées (point de vue et projection)

[2] D'autres valeurs moins courantes existent.

en fonction des transformations courantes. Une image n'échappe pas à la règle et possède donc une position dans l'espace ; son orientation dans l'espace étant définie par les axes x et y du repère de la scène (voir figure 11). La position dans l'espace de l'image est, par défaut, $(0, 0, 0)$. Cette position courante est spécifiée/modifiée à l'aide la fonction **glRasterPosxx(...)**, ceci de la même manière qu'un sommet par deux, trois ou quatre coordonnées : $\{(x, y), (z = 0, w = 1)\}$), $\{(x, y, z)$,$w = 1\}$ ou $\{(x, y, z, w)\}$ de type réels (glRasterPos234f()), entiers (glRaster234i()) ou autre.

À noter : la librairie MESA [Mes] propose une fonction :

```
glWindowPosMESAxx()
```

qui permet de positionner le premier pixel de l'image dans la fenêtre d'affichage sans se préoccuper de la projection ou du point de vue appliquée.

Enfin, OpenGL impose, par défaut, que la matrice de pixels soit constituée de lignes contenant un nombre de bytes multiple de quatre. Pour modifier cela, il faut utiliser la fonction :

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1-2-4-8);
```

avec pour paramètre l'alignement souhaité (1,2,4 ou 8). Dans le cas général, les bytes sont stockés les uns à la suite des autres et l'alignement sera donc 1. À noter ici que cette fonction sert aussi à afficher une partie de l'image uniquement ou à définir des modes de stockage particulier des pixels (voir [WND97] pour plus d'informations).

La procédure classique d'affichage d'une image sera donc :

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glRasterPos2i(0,0);
/* glWindowPosMESA2i(0,0); */  /* positionnement independant des */
                               /* trans. proj. et point de vue */
glDrawPixels( 512, 512, GL_RGB, GL_UNSIGNED_BYTE, MatricePixels);
```

2.10 Éliminer les parties cachées

Pour éliminer les parties cachées d'un objet ou d'une scène, OpenGL propose deux méthodes : le back-face culling pour les polygones et le z-buffer ou depth-buffer en général.

2.10.1 Backface-culling

Le backface culling, ou élimination des faces arrières, consiste à éliminer de l'affichage les polygones pour lesquels la normale forme un angle supérieur à $\pi/2$ avec les lignes de vue de chaque sommet du polygone (voir figure 12).

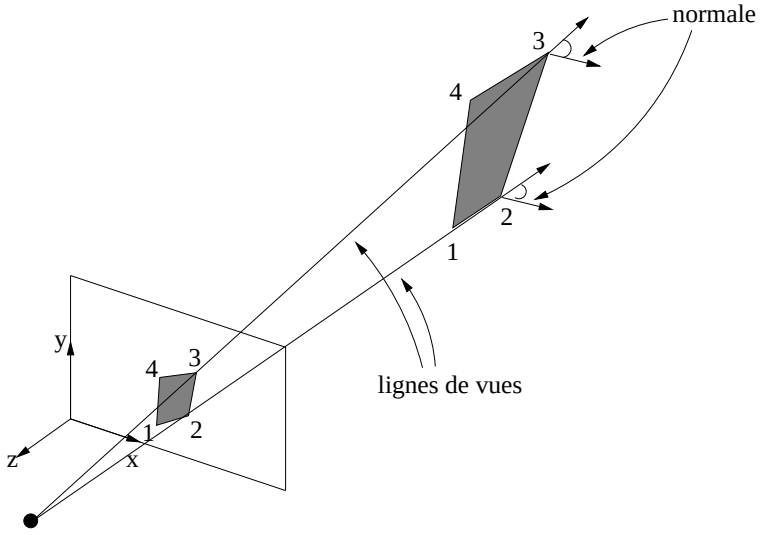


Figure 12: Un polygone est visible si l'angle que forme la normale au polygone avec la ligne en chacun de ses sommets est supérieur à $\pi/2$.

La normale à un polygone correspond à la normale au plan contenant le polygone. L'orientation de ce plan peut être choisie à partir de l'ordre des sommets du polygone. Par défaut, une orientation positive correspond à des sommets ordonnés dans le sens inverse des aiguilles d'une montre (voir figure 12). Cela peut être changé à l'aide de :

```
glFrontFace(GL_CCW - GL_CW)
```

où GL_CCW correspond au sens inverse des aiguilles d'une montre et GL_CW au sens des aiguilles d'une montre.

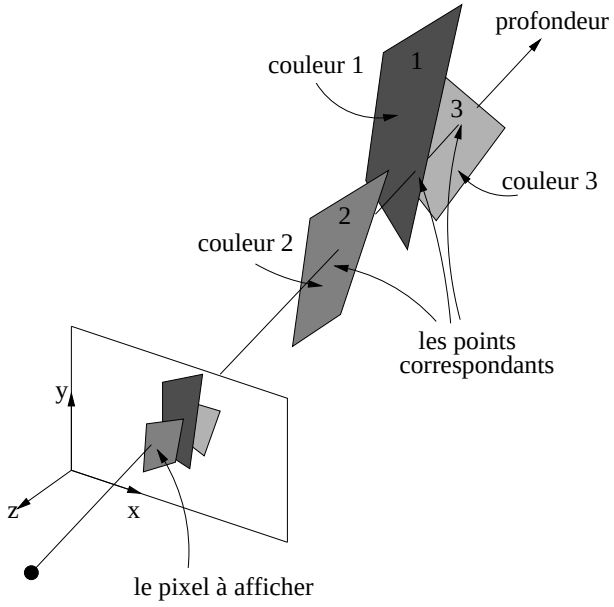
Ensuite pour valider l'élimination des faces arrières, il suffit de suivre la procédure suivante :

```
glEnable(GL_CULL_FACE);    /* inversement glDisable(GL_CULL_FACE) */
glCullFace(GL_FRONT - GL_BACK - GL_FRONT_AND_BACK);
```

qui valide donc l'élimination des polygones respectivement de face par rapport au point de vue et suivant l'orientation choisie (les sommets sont ordonnés dans l'image suivant l'ordre choisi), de dos (GL_BACK) ou les deux (GL_FRONT_AND_BACK).

2.10.2 z-buffer

L'idée directrice de la méthode du z-buffer est de maintenir à jour une mémoire tampon contenant la profondeur (le "z") du point observé en chaque pixel de l'image; cela de la même manière que la mémoire vidéo contient l'information RGBA du point observé en chaque pixel. Lors du dessin d'une primitive, la profondeur d'un pixel à afficher est remise à jour si et seulement le point correspondant à ce pixel sur la primitive présente une profondeur inférieure à celle déjà présente dans la mémoire tampon. Cette remise à jour dans le tampon de profondeur entraîne alors une remise à jour de l'information RGBA dans la mémoire vidéo (voir figure 13).



Affichage	valeur mémoire vidéo	valeur tampon de profondeur
état initial	fond d'écran	∞
point 1	couleur 1	prof. 1
point 2	couleur 2	prof. 2
point 3	couleur 2	prof 2

Figure 13: z-buffer, les polygones sont affichés dans l'ordre 1,2 et 3.

Pour OpenGL, la profondeur correspond à l'opposé de la coordonnée en z et le tampon mémoire s'appelle le *depth buffer*. Pour valider les tests de profondeur,

il faut suivre la procédure suivante :

```
glEnable(GL_DEPTH_TEST);    /* inv. glDisable(GL_DEPTH_TEST) */
glDepthFunc(GL_LESS -       /* <  valeur par defaut */
            GL_GREATER -    /* >  */
            GL_LEQUAL -     /* <= */
            GL_GEQUAL -     /* >= */
            GL_NOTEQUAL -   /* != */
            GL_EQUAL);      /* = */
```

La fonction glDepthFunc() sert ici à définir la comparaison effectuée pour la remise à jour d'un pixel dans le tampon de profondeur et la mémoire vidéo. La valeur par défaut est GL_LESS. Pour cette valeur, une remise à jour s'effectue si la profondeur d'un point à afficher est strictement inférieure à celle stockée dans le tampon de profondeur aux coordonnées image du point en question.

À noter que l'utilisation du z-buffer nécessite la création d'un tampon de profondeur et son accès par OpenGL. Cela n'est pas effectué par OpenGL et doit donc être géré par la librairie utilisant OpenGL. L'exemple de programme 3 montre comment utiliser un z-buffer avec GLUT pour afficher une surface sous la forme fil de fer en procédant à une élimination des lignes cachées.

2.11 Effectuer un rendu

Pour effectuer un rendu d'une scène, il faut placer des sources lumineuses dans la scène et préciser les caractéristiques de ces sources ainsi que celles des surfaces présentes. Un exemple de programme est donné au paragraphe 3.

OpenGL propose un certain nombre de fonctions permettant de spécifier les caractéristiques de sources lumineuses dans la scène : position, couleur, etc. Pour utiliser des sources lumineuses dans OpenGL, il faut tout d'abord valider l'illumination de manière globale :

```
glEnable(GL_LIGHTING);   /* inver. glDisable(..)*/
```

et ensuite valider individuellement chaque source lumineuse utilisée (jusqu'à 8 ou plus en fonction de l'implementation d'OpenGL) :

```
glEnable(GL_LIGHT0);     /* 1ere source lumineuse */
glEnable(GL_LIGHT1);     /* 2eme source lumineuse */
...
glEnable(GL_LIGHT7);     /* 8eme source lumineuse */
```

2.11.1 Le modèle d'illumination

L'illumination en un point de la scène se calcule comme la somme de différentes contributions des sources lumineuses en fonction du matériel. Classiquement, ces contributions sont au nombre de trois : ambiante, diffuse et spéculaire (voir figure 15). L'intensité lumineuse en point P de la surface S et pour une composante R, G ou B est donc :

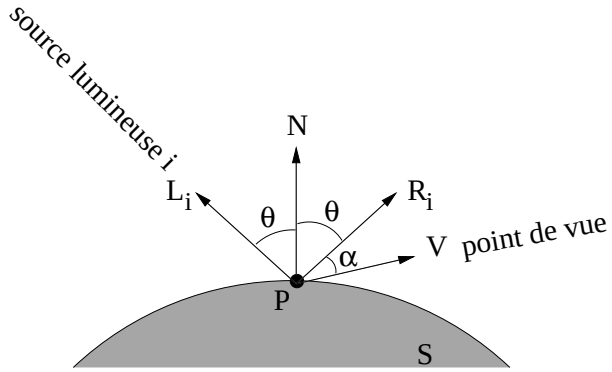


Figure 14: Les différentes directions au point P.

$$
\begin{aligned}
I_P = & I_{ambiante} \times S_{ambiante} + \\
& \sum_{i \in [0..7]} I^i_{diffuse} \times S_{diffuse} \times (N \cdot L_i) + \\
& \sum_{i \in [0..7]} I^i_{speculaire} \times S_{speculaire} \times (V \cdot R_i)^n
\end{aligned}
$$

avec :

- N : normale à la surface S en P;
- L_i : direction de P vers la source lumineuse i;
- V : direction de P vers le centre de projection;
- R_i : direction symétrique de L_i par rapport à N;
- n : l'exposant spéculaire ou brillance de la surface;

- $I_{ambiante}$, $I^i_{diffuse}$, $I^i_{speculaire}$: intensités des sources lumineuses pour une composante R,G,B donnée. Les valeurs sont comprises entre 0 et 1 pour OpenGL;

- $S_{ambiante}$, $S_{diffuse}$, $S_{speculaire}$: propriétés matérielles de la surface au point P pour une composante R,G,B donnée (valeurs également comprises entre 0 et 1 pour OpenGL).

La spécification des caractéristiques d'une source lumineuse se fait à l'aide de vecteurs. Pour la lumière ambiante (contribution $I_{ambiante}$ constante en tous points de l'espace) :

```
GLfloat lum_ambiente[]={0.2,0.2,0.2,1.0}; /* composantes R,G */
                                          /* B,A par defaut  */
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lum_ambiente);
```

Pour les sources lumineuses ponctuelles (contributions diffuse $I^i_{diffuse}$ et spéculaire $I^i_{speculaire}$ qui sont fonction des positions respectives de la source et du point traité), par exemple GL_LIGHT0 (voir figure 15) :

```
GLfloat lum0_diffuse[]={1.0,1.0,1.0,1.0};   /* composantes R,G,*/
GLfloat lum0_speculaire[]={1.0,1.0,1.0,1.0};/* B,A par defaut  */
GLfloat lum0_position[]={0.0,0.0,1.0,0.0};  /* pos. par defaut */
                                            /* (a l'infini) */
glLightfv(GL_LIGHT0,GL_DIFFUSE,lum0_diffuse);
glLightfv(GL_LIGHT0,GL_SPECULAR,lum0_speculaire);
glLightfv(GL_LIGHT0, GL_POSITION, lum0_position);
```

À noter :

1. À l'aide des coordonnées homogènes, une source lumineuse peut être placée à l'infini (4ème coordonnées nulle). Les trois premières coordonnées déterminent alors la direction de la source lumineuse dite *directionnelle*.

2. La position d'une source lumineuse est calculée au moment de l'appel de la fonction glLight(...,GL_POSITION,...) à partir de la position donnée en paramètre; cette position étant soumise à l'ensemble des transformations point de vue courantes (mais non les projections).

De la même manière, pour spécifier les caractéristiques matérielles ($S_{ambiante}$, $S_{diffuse}$, $S_{speculaire}$ et n) des surfaces observées :

```
GLfloat surf_ambient[]={0.2,0.2,0.2,1.0}; /* valeurs par defaut */
GLfloat surf_diffuse[]={0.8,0.8,0.8,1.0}; /*  R,G,B,A          */
GLfloat surf_speculaire[]={0.0,0.0,0.0,1.0}; /*  R,G,B,A       */
GLfloat surf_brillance[]={0.0};          /* exposant speculaire*/

glMaterialfv(GL_FRONT,GL_SPECULAR,surf_ambient);
glMaterialfv(GL_FRONT,GL_DIFFUSE,surf_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,surf_speculaire);
glMaterialfv(GL_FRONT,GL_SHININESS,surf_brillance);
```

Enfin OpenGL considère, par défaut, que le point de vue est situé à l'infini pour le calcul de la réflexion spéculaire. Cela diminue en effet le nombre d'opérations à réaliser mais entraîne des résultats moins "réalistes". Pour modifier cela :

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, /* valeur GL_FALSE */
              GL_TRUE);                     /* par defaut */
```

2.11.2 Le modèle d'ombrage

Dans le cas de surface définies sous la formes de facettes polygonales, il existe trois méthodes principales pour déterminer les valeurs d'intensité sur la facette :

1. **Ombrage plat :** le calcul de l'illumination est effectué en un point de la facette puis cette valeur est dupliquée pour tous les points de la facette.
2. **Ombrage de Gouraud :** l'illumination en chaque point de la facette est déterminée par interpolation linéaire des valeurs aux sommets de la facette .
3. **Ombrage de Phong :** l'illumination en chaque point de la facette est déterminée par interpolation linéaire des normales aux sommets de la facette puis calcul de l'intensité suivant la valeur de cette normale.

OpenGL propose les deux premières méthodes (voir figure 15). Le choix de l'une ou de l'autre se faisant par :

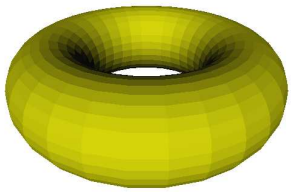
```
glShadeModel(GL_FLAT - GL_SMOOTH); /* ombrage plat - de Gouraud */
```

la valeur par défaut correspondant à un ombrage lisse : **GL_SMOOTH**.

Ajoutons pour finir qu'il existe des fonctions OpenGL permettant d'affiner encore le réglage des éclairages. Ces fonctions sortent du cadre de ce document, le lecteur intéressé pourra cependant consulter [WND97, Ope] pour plus de détails.

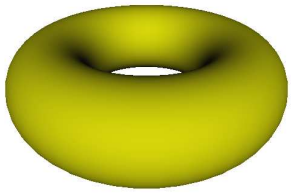
```
GLfloat lum_ambiante[]={1.0,1.0,0.0,1.0};
glShadeModel(GL_FLAT);
glEnable(GL_LIGHTING);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
                    lum_ambiante);
```

(a) Lumière ambiante.



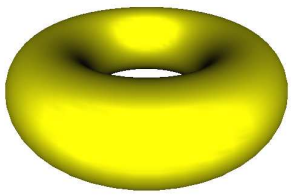
```
GLfloat lum_diffuse[]={0.8,0.8,0.0,1.0};
glShadeModel(GL_FLAT);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,lum_diffuse);
```

(b) Réflexion diffuse (surface lambertienne) et ombrage plat.



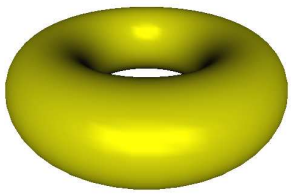
```
GLfloat surf_diffuse[]={0.8,0.8,0.0,1.0};
glShadeModel(GL_SMOOTH);   /* valeur par defaut */
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,surf_diffuse);
```

(c) Réflexion diffuse (surface lambertienne) et ombrage de Gouraud.



```
GLfloat surf_diffuse[]={0.8,0.8,0.0,1.0};
GLfloat surf_speculaire[]={1.0,1.0,0.0,1.0};
GLfloat surf_shininess[]={10.0};
glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,surf_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,surf_speculaire);
glMaterialfv(GL_FRONT,GL_SHININESS,surf_shininess);
```

(d) Réflexion diffuse et spéculaire (exposant spéculaire 10).



```
GLfloat surf_diffuse[]={0.8,0.8,0.0,1.0};
GLfloat surf_speculaire[]={1.0,1.0,0.0,1.0};
GLfloat surf_shininess[]={100.0};
glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,surf_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,surf_speculaire);
glMaterialfv(GL_FRONT,GL_SHININESS,surf_shininess);
```

(d) Réflexion diffuse et spéculaire (exposant spéculaire 100).

Figure 15: Différents ombrages et illuminations d'un tore. La source lumineuse directionelle GL_LIGHT0 a la position par défaut $(0, 0, 1, 0)$ et les intensités R,G,B,A par defaut $(1, 1, 1, 1)$.

2.12 Plaquer des textures

Les textures confèrent aux images de synthèse un réalisme indéniable. Le principe est d'utiliser, pour remplir un élément de surface de la scène observée, une texture (en général une image). Le plaquage de texture consiste donc à définir, pour les points texturés de l'image à synthétiser, quelles sont les points de références correspondants (*texels*) dans une image (2D ou 1D) de texture (ou de quelle manière déterminer ces points de références), et comment utiliser ces points de références.

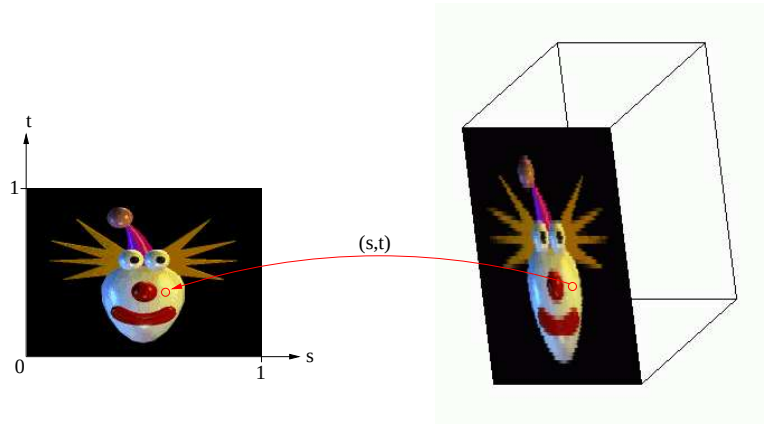


Figure 16: Le plaquage de texture consiste à utiliser des *textures* (ici une image 2D) pour remplir les éléments de surface de la scène.

L'utilisation des textures dans OpenGL s'effectue suivant deux étapes principales :

1. Création de la texture : spécifier l'image source pour la texture et la méthode de plaquage à utiliser.
2. Plaquage de la texture : spécifier les points de la scène délimitant la *zone* de plaquage.

L'utilisation des textures requière au préalable la validation des calculs les concernant dans OpenGL :

```
glEnable(GL_TEXTURE_1D - GL_TEXTURE_2D); /* inver. glDisable(..)*/
```

Il est à noter qu'OpenGL permet l'utilisation de textures 1D (tableaux de pixels à 1 dimension) ou 2D (tableaux de pixels à 2 dimensions). Nous ne traiterons ici que les textures 2D ; l'usage de textures 1D étant sensiblement similaire à celui de textures 2D.

2.12.1 Définir une texture

La création d'une texture nécessite dans un premier temps une image source. Cette image source peut soit être crée à l'intérieur du programme, soit être lue dans un fichier. La définition d'une texture à partir de cette image source s'effectue ensuite en plusieurs étapes[3] dont certaines découlent du besoin d'utiliser plusieurs textures de façon simultanée :

1. Identifier,
2. spécifier la texture courante,
3. spécifier le plaquage,
4. spécifier l'image source,
5. spécifier le mode de rendu.

Identifier Pour pouvoir manipuler plusieurs textures, il est nécessaire de les identifier. OpenGL utilise pour cela des entiers. Le choix des entiers peut être fait soit directement, soit à l'aide de la fonction *glGenTextures*. Cette dernière gère les identifiants de textures et fournit des entiers qui ne sont pas encore utilisés.

```
int textures[n+1];    /* un tableau d'identifiants vide */
glGenTextures(n, textures);  /* genere n entiers        */
                             /* disponibles             */
```

Spécifier la texture courante Il s'agit de spécifier la texture (1D ou 2D) sur laquelle s'appliqueront les opérations ultérieures. Cela se fait par l'identifiant de la texture.

```
glBindTexture(GL_TEXTURE_2D, textures[2]); /* textures[2]   */
                                           /* devient active*/
```

Spécifier le plaquage Il s'agit de spécifier la méthode à utiliser pour plaquer la texture courante. Cela se fait à l'aide la fonction *glTexParameter*. Cette dernière permet de plaquer les textures de différentes manières suivant la fonction de transfert choisie entre l'image destination d'une texture et son image source.

[3] Attention à la chronologie entre les différentes étapes !

- Répéter ou clamper la texture : les coordonnées textures (s,t) associées à un point sont comprises entre 0 et 1 pour parcourir l'ensemble de l'image de texture. Lorsque ces coordonnées sont en dehors de l'intervalle $[0\ 1]$, OpenGL peut soit répéter soit clamper la texture (voir figure 17) :

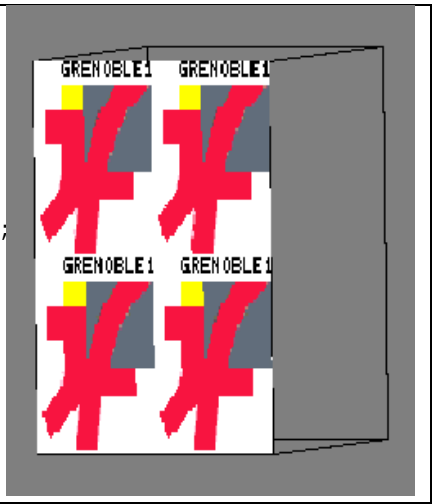
```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
                              GL_REPEAT - GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
                              GL_REPEAT - GL_CLAMP);
```

- Déterminer les valeurs des pixels à remplir : les coordonnées texture (s,t) associées à un pixel ne sont pas nécessairement des valeurs entières et ne correspondent donc pas nécessairement à un texel de l'image de texture. Les solutions consistent alors à prendre soit la valeur du texel le plus proche de (s,t), soit l'interpolation des valeurs des 4 pixels entourant (s,t). OpenGL offre ces deux possibilités dans le cas où la surface à remplir est plus grande que l'image de texture (pixel de taille inférieur au texel) et plus dans le cas où la surface est plus petite que l'image de texture (pixel de taille supérieur au texel).

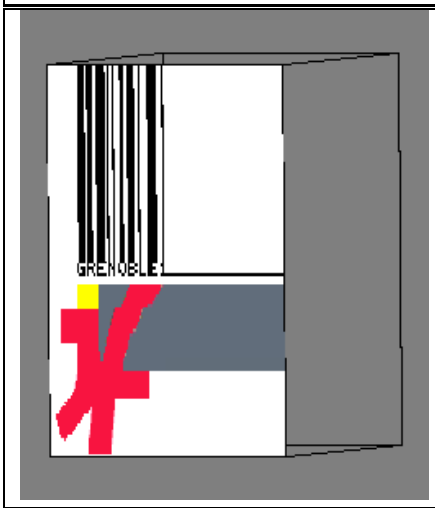
```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                              GL_NEAREST - GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                               GL_NEAREST - GL_LINEAR -
                               GL_NEAREST_MIPMAP_NEAREST -
                               GL_NEAREST_MIPMAP_LINEAR -
                               GL_LINEAR_MIPMAP_NEAREST -
                               GL_LINEAR_MIPMAP_LINEAR);
```

Lorsque la surface à remplir est plus grande que l'image de texture (GL_TEXTURE_MAG_FILTER), deux méthodes sont possibles : GL_NEAREST ou GL_LINEAR. La première consiste à choisir le texel le plus proche des coordonnées (s,t) spécifiées (ou calculées), la deuxième consiste à prendre la valeur interpolée des quatre texels entourant la position (s,t) (voir figure 18).
Lorsque la surface à remplir est plus petite (GL_TEXTURE_MIN_FILTER), plusieurs méthodes sont possibles, GL_NEAREST et GL_LINEAR comme précédemment ainsi que des méthodes faisant intervenir des *mipmaps*. Le principe des mipmaps est de stocker l'image de texture à différents niveaux de résolution : l'image même, l'image de tailles divisées par 4, l'image de

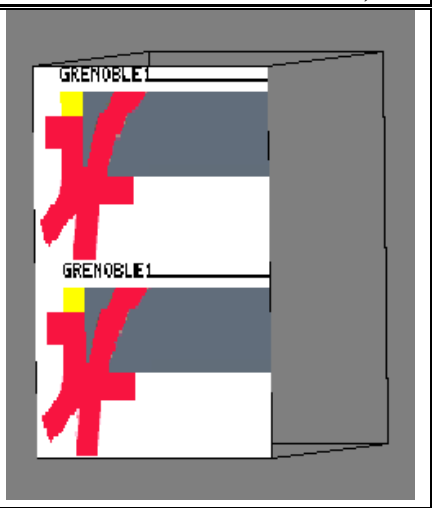
```
void Display()
{
  glutWireCube(200);

  glBegin(GL_QUADS) ;
   glTexCoord2f(0.0, 0.0);
   glVertex3f(-100.0, -100.0, 100);
   glTexCoord2f(2.0, 0.0);
   glVertex3f(100.0, -100.0, 100);
   glTexCoord2f(2.0, 2.0);
   glVertex3f(100.0, 100.0,100);
   glTexCoord2f(0.0, 2.0);
   glVertex3f(-100.0, 100.0,100);
  glEnd();
}
```



glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)



glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)



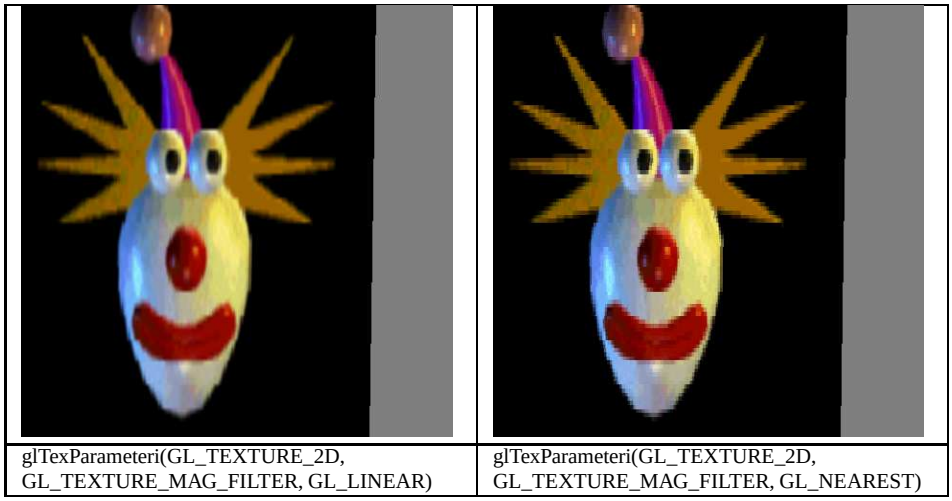
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)

Figure 17: Plaquage de texture : lorsque les coordonnées textures sont supérieures aux dimensions de l'image (donc supérieures à 1), OpenGL peut soit répéter la texture (GL_REPEAT) soit ramener les coordonnées à 1 (GL_CLAMP) en fonction de ce qui a été spécifié avec la fonction *glTexParameter*.

tailles divisées par 16, et ainsi de suite jusqu'à ce que l'image soit de la dimension d'un pixel. Ensuite, pour déterminer les valeurs d'intensités pour un pixel, on peut (voir figure 19) :

1. choisir l'image dans la mipmap dont les dimensions en pixels sont les plus proches de celles de la surface à remplir(_MIPMAP_NEAREST);
2. interpoler entre les deux images dans la mipmap dont les dimensions en pixels sont les plus proches de celles de la surface à remplir (_MIPMAP_LINEAR);
3. et enfin choisir le texel le plus proche (GL_NEAREST_MIPMAP_), ou interpoler entre les 4 plus proches (GL_LINEAR_MIPMAP_), dans les images choisies de la mipmap .

Il est à noter que l'option GL_LINEAR_MIPMAP_LINEAR faisant intervenir deux interpolations produira les résultats les plus lisses mais que cela à un coût en temps de calcul.



glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)

Figure 18: *Magnification* : la texture, dans l'image finale, a des dimensions supérieures à celles de l'image dont elle provient.

Spécifier l'image de texture Pour spécifier l'image de texture, il est possible de définir directement l'image concernée (soit un tableau de pixels) ou bien de définir ou de calculer une mipmap (l'image à différents niveaux de résolution) à partir de cette image.

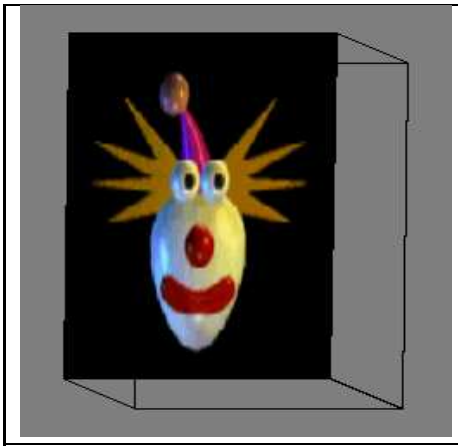
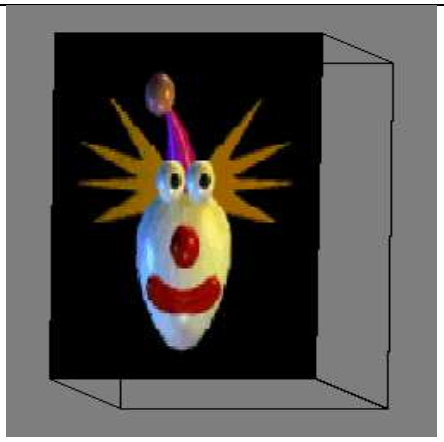
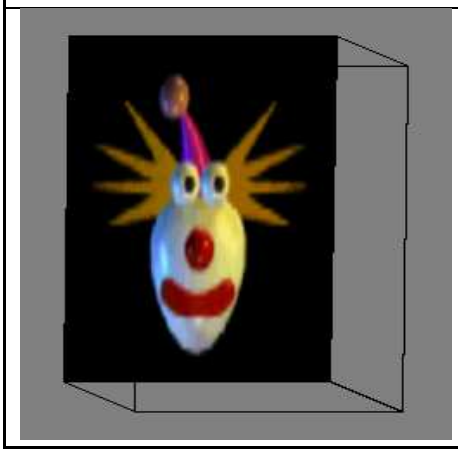
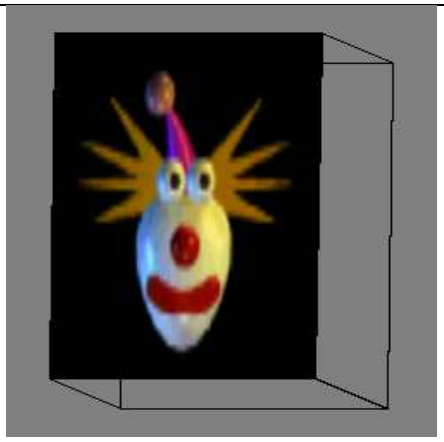
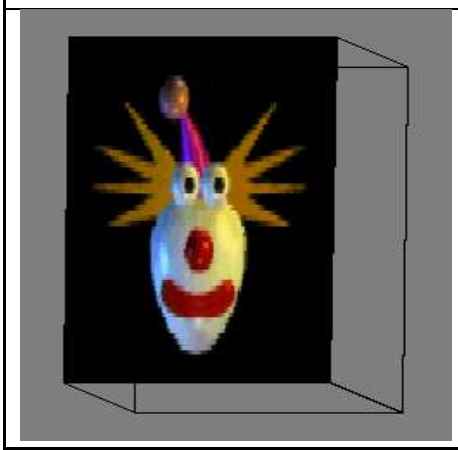
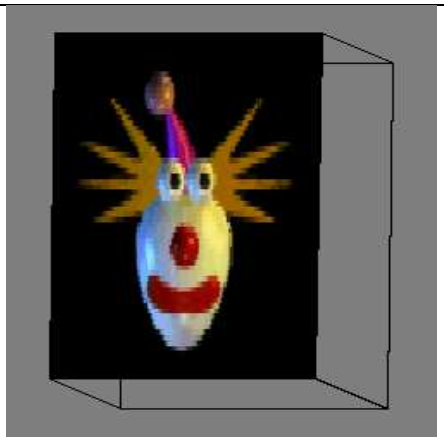
	
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
	
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR)	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST)
	
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR)	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST)

Figure 19: *Minification* : la texture, dans l'image finale, a des dimensions inférieures à celles de l'image dont elle provient.

La fonction suivante permet de définir une image de texture :

```
glTexImage2D(GL_TEXTURE_2D,
             Level,              /* niveau de resolution */
             Formatint,          /* format interne       */
             Largeur, Hauteur,   /* dimensions puissances de 2*/
             Bord,              /* largeur du bord (0 ou 1 )*/
             Formatext,          /* format de l'image    */
             Type,               /* type des donnees     */
             Image);             /* tableau de donnees   */
```

où :

- Level : caractérise le niveau de résolution de l'image, soit 0 pour une seule image. Ce paramètre sert à construire une mipmap lorsque l'on dispose de l'image à plusieurs niveaux de résolutions.
- Formatint : le format interne ou, en d'autres termes, ce que l'on souhaite utiliser dans l'image de texture (quelle composante ...). Les valeurs de ce paramètre sont principalement : GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA [4].
- Largeur, Hauteur : les dimensions de l'image de texture qui doivent être des puissances de 2 de valeur minimum 64. Si l'image que l'on souhaite utilisée ne vérifie pas ce critère, il est toujours possible d'en modifier les dimensions à l'aide de la fonction :

  ```
  gluScaleImage(Format, Largeur1, Hauteur1, Type1, Image1,
                        Largeur2, Hauteur2, Type2, Image2)},
  ```

  où Image1 est l'image initiale et Image2 est l'image redimensionnée.
- Bord : la largeur du bord, soit la valeur 0 ou 1.
- Formatext : le format externe de l'image de texture soit principalement GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA.
- Type : le type des données soit GL_UNSIGNED_BYTE (entiers non signés sur 8 bit), GL_BYTE, GL_BITMAP (bits codés dans des unsigned bytes), GL_SHORT (entiers signés sur 16 bits), GL_UNSIGNED_SHORT.

[4] D'autres valeurs existent, en particulier du fait que le nombre de bits utilisés par composantes peut être modifiable, GL_RGB8 ou GL_RGB10 par exemple.

- Image : le tableau de valeurs pour l'image de texture.

La fonction glTexImage2D peut aussi servir à construire une mipmap en utilisant différentes valeurs du paramètre Level. Par contre, il existe une fonction de la libraire glu qui construit une mipmap directement à partir de l'image initiale et qui, de plus, redimensionne automatiquement l'image si celle-ci n'a pas des dimensions puissances de 2 :

```
gluBuild2DMipmaps(GL_TEXTURE_2D,
                  Formatint,          /* format interne        */
                  Largeur, Hauteur,   /* dimensions de l'image */
                  Formatext,          /* format de l'image     */
                  Type,               /* type des donnees      */
                  Image);             /* tableau de donnees    */
```

Spécifier le mode de rendu Enfin, la dernière fonction concerne le mode de rendu à utiliser : soit la texture uniquement, soit la combinaison de la texture et d'un rendu.

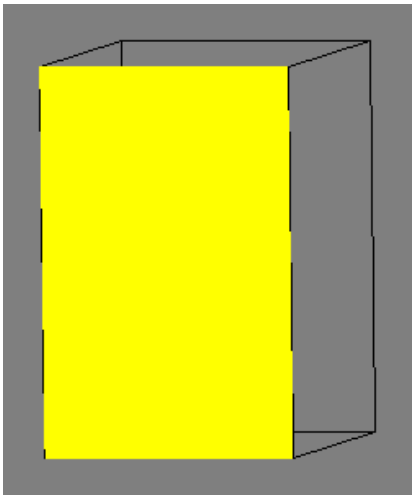
```
glTexEnvi(GL_TEXTURE_ENV,
          GL_TEXTURE_ENV_MODE,
          GL_DECAL - GL_REPLACE - GL_MODULATE - GL_BLEND)
```

la valeur par défaut est GL_MODULATE qui combine le rendu, si celui ci est défini, et la texture. Pour une texture de format RGB, le rendu s'effectue pour chaque composante suivant (voir figure 20):

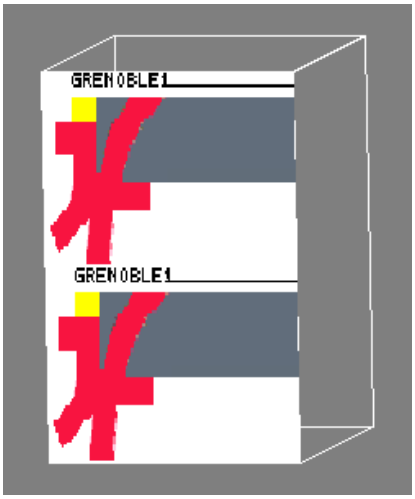
DECAL	REPLACE	MODULATE	BLEND
texture	texture	texture $\times$ rendu	rendu $\times$ (1 - texture)

En résumé Le code suivant illustre l'ensemble des appels de fonctions utiles à l'usage d'une texture :

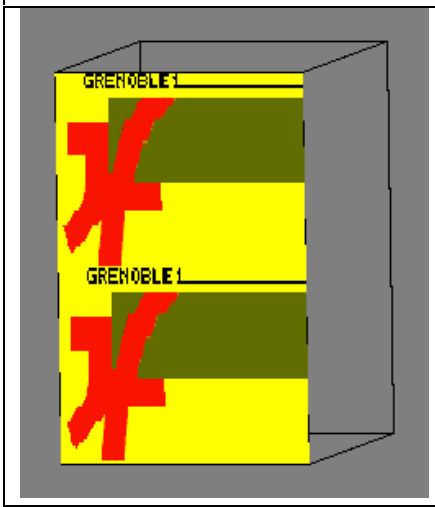
```
void init_texture()
{
  glGenTextures(1,&num_texture);
  glBindTexture(GL_TEXTURE_2D, num_texture);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```



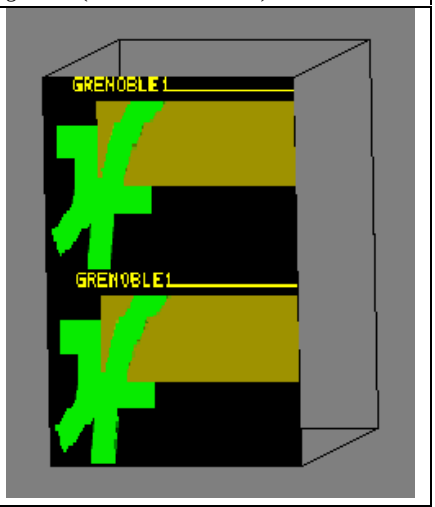
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL); glDisable(GL_TEXTURE_2D)



glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL - GL_REPLACE); glEnable(GL_TEXTURE_2D)



glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE); glEnable(GL_TEXTURE_2D)



glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND); glEnable(GL_TEXTURE_2D)

Figure 20: Spécifier le rendu : en fonction du mode rendu choisi, il est possible d'afficher le rendu uniquement, la texture uniquement (GL_DECAL - GL_REPLACE) ou une combinaison des deux (GL_MODULATE - GL_BLEND).

```
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  image = LoadPNMImage("image.ppm",&ImgWidth,&ImgHeight,&MaxVal);
  //glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, ImgWidth, ImgHeight,
                0, GL_RGB, GL_UNSIGNED_BYTE, Image);
  gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, ImgWidth, ImgHeight,
                    GL_RGB, GL_UNSIGNED_BYTE, image);
  glEnable(GL_TEXTURE_2D);
}
```

2.12.2 Utiliser une texture

3 Afficher et animer des images OpenGL avec GLUT

Les fonctions qui ont été présentées permettent de construire une image en effectuant des dessins dans une mémoire : vidéo ou tampon. Ces fonctions ne réalisent par contre pas la liaison entre la fenêtre graphique et la mémoire de dessin. Cela nécessite un environnement gérant les interconnexions avec le matériel. La libraire GLUT permet cela, et gère en particulier le clavier, la souris et les fenêtres graphiques. Nous présentons ici l'architecture générale d'un programme GLUT et quelques exemples démonstratifs.

3.1 Architecture générale d'un programme GLUT

Un programme GLUT est généralement constitué des parties suivantes (la liste est non-exhaustive) :

1. Une fonction d'initialisation : cette fonction permet d'initialiser différentes valeurs : couleurs, taille d'un point, etc.
2. Une fonction d'affichage : c'est la fonction appelée par GLUT pour rafraîchir la fenêtre graphique.
3. Une fonction de fenêtrage : c'est la fonction appelée lorsque la fenêtre graphique est modifiée.
4. Une fonction de gestion du clavier : c'est la fonction appelée lorsque l'utilisateur appuie sur une touche du clavier.
5. Des fonction de gestion des événements de la souris : c'est les fonctions appelées lorsque l'état de la souris est modifié.
6. Une partie principale : dans cette partie diverses initialisations sont réalisées, l'association des fonctions précédentes aux événements correspondants est effectuée et enfin, la boucle principale est lancée.

3.1.1 Squelette d'un programme GLUT

```
#include <GL/gl.h>
#include <GL/glu.h>     /* fichiers d'entetes OpenGL, GLU et GLUT */
#include <GL/glut.h>

static void initialiser(void)
{
  glClearColor(0,0,0,0);    /* definition de la couleur utilisee */
                            /* pour effacer */
  glColor3f(1.0,1.0,1.0);   /* couleur courante */
}

static void afficher(void)
{
  glClear(GL_COLOR_BUFFER_BIT);  /* effacement du tampon ou  */
                                 /* s'effectuent les dessins */
  glBegin(GL_POLYGON);
  glVertex3f(12.0,3.0,5.0);
  ...                            /* dessin */
  glEnd();
}

static void refenetrer(int w, int h)
{
 glViewport(0, 0, (GLsizei) w, (GLsizei) h); /* modification des tailles */
                                             /* du tampon d'affichage    */
 glMatrixMode(GL_PROJECTION);                /* pile courante = projection */
 glLoadIdentity();                           /* specification de la projection */
 glOrtho(-50.0,50.0,-50.0,50.0,-1.0,1.0);
 glMatrixMode(GL_MODELVIEW);                 /* pile courante = point de vue */
 glLoadIdentity();
}

static void clavier(unsigned char touche, int x, int y)
{
   switch (touche) {
      case 27:              /* sortie si touche ESC */
         exit(0);
   }
}

static void gerer_souris(int bouton, int etat, int x, int y)
{
  switch(bouton){
```

```
    case GLUT_LEFT_BUTTON :
      if(etat == GLUT_DOWN)
        ....
      break;
    case GLUT_MIDDLE_BUTTON :
      ...
      break;
    default:
      break;
  }
}

static void gerer_souris_mouvement(int x, int y)
{
    /* position courante (x,y) de la souris */
}

void main(int argvc, int **argv)
{
  glutInit(&argvc, argv);
  glutInitDisplayMode(GLUT_RGB);           /* affichage couleur */
  glutInitWindowSize(500, 500);            /* taille initiale fenetre graphique */
  glutInitWindowPosition(200,200);         /* position initiale */
  glutCreateWindow(argv[0]);               /* creation de la fenetre graphique */

  init();
  glutDisplayFunc(afficher);               /* fonction d'affichage */
  glutReshapeFunc(refenetrer);             /* fonction de refenetrage */
  glutKeyboardFunc(clavier);               /* gestion du clavier */
  glutMouseFunc(gerer_souris);                 /* fonction souris */
  glutMotionFunc(gerer_souris_mouvement);  /* deplacement de la souris */
  glutMainLoop();                          /* lancement de la boucle principale */
  return(0);
}
```

3.1.2 Makefile générique pour OpenGL

```
########################################
# Makefile OpenGL                      #
#                                      #
########################################

# Noms des programmes
PROG = sphere tore cube

# Librairies GL
INCDIR = -I/home/pythia/eboyer/Src/Mesa/include
LIBDIR = -L/home/pythia/eboyer/Src/Mesa/lib
MESAGL_LIBS = $(LIBDIR) -lglut -lMesaGLU -lMesaGL
GL_LIBS = $(LIBDIR) -lglut -L/usr/openwin/lib -lGLU -lGL

#Librairies X
XLIBS = -L/usr/openwin/lib -lX11 -lXext -lXmu -lXi

# Variables pour la compilation des fichiers
CC        =  gcc
CFLAGS    =  $(INCDIR) -g -Wall
CPPFLAGS  =  -DDEBUG

# Cibles
all : $(PROG)

clean:
        -rm $(PROG)
        -rm *.o *~

# Regles de compilation
.SUFFIXES: .o .c
.c.o:
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
.o:
        $(CC) $(CPPFLAGS) $(CFLAGS) $< $(GL_LIBS) $(XLIBS) -lm -o $@
```

References

[Glu] Glut. Graphics Library Utility Toolkit Specification. http://reality.sgi.com/mjk/glut3/glut3.html.

[Mes] Mesa. The Mesa 3D Graphics Library, Brian Paul. http://www.mesa3d.org/.

[Ope] OpenGL 1.2 Specification. http://www.sgi.com/software/opengl/manual.html.

[WND97] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide, second edition*. Addison Wesley, 1997.