Programmation 3D avec OpenGL

Rémy Malgouyres
LIMOS UMR 6158, IUT département info Université Clermont 1, B.P. 86 63172 AUBI

Une version PDF de ce document est téléchargeable sur mon site web, ansi que la version html.

Table des matières

Chapitre 1

Interface GLUT

OpenGL est une bibliothèque graphique qui permet de faire de la visualisation $2D$ et $3D$ avec une accélération *hardware* (utilisation de la carte graphique). La librairie *OpenGL* peut être utilisée en combinaison avec la *glut*, qui permet de créer une fenêtre graphique et de gérer les évennements tels que le click de souris ou le redimensionnement de la fenêtre par l'utilisateur.

1.1 Compiler avec la *glut*

Pour pouvoir compiler un programme C ou $C++$ avec la *glut*, il faut inclure la bibliothèque `glut.h` :

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

Pour compiler, il faut utiliser la librairie `lglut`. Sous linux ou unix avec le compilateur gcc, la ligne de commande est :

```
$ gcc programme.c -lglut -lGL -lGLU -o programme.exe
```

(on ferait de même pour compiler un programme $C++$ avec le compilateur $g++$).

1.2 Initialiser une fenêtre graphique

Voici un programme `glut` minimal qui crée une fenêtre graphique pour des animations $3D$ (et qui ne fait rien d'autre). Le lecteur trouvera des commentaires sur les différentes fonctions dans l'annexe A

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

/* variables globales */

/* position de la fenêtre graphique dans l'écran : */
GLushort pos_x_fenetre=100, pos_y_fenetre=100;

/* dimensions de la fenêtre graphique : */
```

```
GLushort largeur_fenetre==400, hauteur_fenetre=400;

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
  glutCreateWindow("OpenGL et glut : TP1");
  glEnable(GL_DEPTH_TEST);

  glutMainLoop();
  return 0;
}
```

1.3 Évennements de la *glut*

Les événements dans un programme informatique sont les interventions de l'utilsateur par le biais de la souris, du clavier, d'un joystick, ect...

1.3.1 L'affichage

L'événement d'affichage a lieu à chaque fois que la vue doit être rafraîchie. La fonction d'affichage (ou *display function*) doit alors redessiner les objets. L'affichage a lieu notament dans les circonstances suivantes :

- Création de la fenêtre graphique ;
- Passage de la fenêtre au premier plan ;
- Appel explicite du programmeur (voir la fonction `glutPostRedisplay`) lorsqu'il le juge nécessaire.

Avec la `glut`, on doit déclarer la fonction d'affichage en utilisant la fonction `glutDisplayFunc`, **qui prend en paramètre un pointeur de fonctions**.

La fonction `glutDisplayFunc` a pour prototype :

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

c'est à dire qu'elle prend en paramètre une fonction d'affichage qui, elle, ne prend aucun paramètre.

Exemple. Avec la fonction d'affichage suivant, le programme ne fait qu'afficher une fenêtre qui est un rectangle rouge (couleur du fond).

```
...

void Affichage(void)
{
  glClearColor(1,0,0,0);  /* coefficients RGB + A de la couleur de fond */
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  /* effaçage */
  glutSwapBuffers();  /* Envoyer le buffer à l'écran */
```

```
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
  glutCreateWindow("OpenGL et glut, TP 1");
  glEnable(GL_DEPTH_TEST);

  glutDisplayFunc(Affichage);  /* appel de glutDisplayFunc */

  glutMainLoop();
  return 0;
}
```

1.3.2 Les frappes de touches clavier

Les événements liés aux touches du clavier peuvent être déclarer grace aux fonctions `glutKeyboardFunc` (pour les caractères usuelles) et `glutSpecialFunc` (pour les touches spéciales telles que $F1$, $F2$, les flèches, etc...).

Exemple. Dans le programme suivant, la couleur du fond est grise de plus en plus claire lorsqu'on appuie sur la flèche droite au clavier, et de plus en plus foncée lorsqu'on appuie sur la flèche gauche. Le programme se termine lorsqu'on appuie sur la touche `'q'`.

```
...

GLfloat niveau_de_gris=0.5;

void Affichage(void)
{
   /* coefficients RGB + A de la couleur de fond : */
  glClearColor(niveau_de_gris,niveau_de_gris,niveau_de_gris,0);
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  /* effaçage */
  glutSwapBuffers();  /* Envoyer le buffer à l'écran */
}

void Special(int touche, int x, int y)
{
  switch(touche)
    {
    case GLUT_KEY_LEFT:
         niveau_de_gris -= 0.05;
 if(niveau_de_gris < 0)
    niveau_de_gris = 0;
         break;
```

```
    case GLUT_KEY_RIGHT:
         niveau_de_gris += 0.05;
 if(niveau_de_gris > 1)
    niveau_de_gris = 1;
         break;
    default:
       fprintf(stderr,"Touche non gérée\n");
    }
  glutPostRedisplay();  /* rafraîchissement de l'affichage */
}

void Clavier(unsigned char key, int x, int y)
{
  switch (key)
    {
    case 'q':
        exit(0);
    default:
        fprintf(stderr,"Touche non gérée\n");break;
    }
  glutPostRedisplay();  /* rafraîchissement de l'affichage */
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
  glutCreateWindow("OpenGL et glut, TP 1");
  glEnable(GL_DEPTH_TEST);

  glutDisplayFunc(Affichage);
  glutKeyboardFunc(Clavier);  /* appel de glutKeyboardFunc */
  glutSpecialFunc(Special);   /* appel de glutSpecialFunc */

  glutMainLoop();
  return 0;
}
```

1.3.3 Gestion de la souris

La fonction gérant l'événement de pression sur un bouton de la souris est déclarée par la fonction `glutMouseFunc`. La fonction gérant le mouvement de la souris avec un bouton pressé est déclarée par la fonction `glutMotionFunc`.

Exemple. Dans le programme suivant, lorsqu'on déplace la souris vers la droite, la couleur du fond devient plus bleue. Lorsqu'on déplace la souris vers la gauche la couleur du fond devient

moins bleue. Lorsqu'on déplace la souris vers le haut, la couleur du fond devient plus verte. Lorsqu'on déplace la souris vers le bas, la couleur du fond devient moins verte.

```
...

GLint mousex, mousey;  /* permet de mémoriser la dernière position */
                       /* de la souris */
GLfloat cof_g=0.5, coef_b=0.5;

int leftButtonDown=0;

void Affichage(void)
{
   /* coefficients RGB + A de la couleur de fond :*/
  glClearColor(0, coef_g, coef_b, 0);
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  /* effaçage */
  glutSwapBuffers();  /* Envoyer le buffer à l'écran */
}

void PresseBouton(int button, int state, int x, int y)
{
  if (button==GLUT_LEFT_BUTTON)
  {
    if (state==GLUT_DOWN)
    {
      leftButtonDown = 1;
      mousex = x;  /* mémorisation des coordonnées de la souris */
      mousey = y;  /* lorsqu'on enfonce le bouton gauche */
    }
    if (state==GLUT_UP)
    {
      leftButtonDown = 0;
    }
  }
}

void BougeSouris(int x, int y)
{
   if (leftButtonDown==1)
   {
     coef_b += 0.01*(x-mousex);
     if(coef_b > 1)
       coef_b = 1;
     if(coef_b < 0)
       coef_b = 0;
     coef_g += 0.01*(y-mousey);
     if(coef_g > 1)
```

```
      coef_g = 1;
    if(coef_g < 0)
      coef_g = 0;
    mousex = x;   /* enregistrement des nouvelles */
    mousey = y;   /* coordonnées de la souris */
  }
  glutPostRedisplay();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
  glutCreateWindow("OpenGL et glut, TP 1");
  glEnable(GL_DEPTH_TEST);

  glutDisplayFunc(Affichage);
  glutKeyboardFunc(Clavier);
  glutSpecialFunc(Special);
  glutMouseFunc(PresseBouton);   /* appel de glutMouseFunc */
  glutMotionFunc(BougeSouris);   /* appel de glutMotionFunc */

  glutMainLoop();
  \retu 0;
}
```

1.3.4 Évennement Idle : animation

L'évennement *Idle* est un évennement qui se produit régulièrement lorsqu'aucun autre évennement ne survient. L'évennement *Idle* permet de faire des animations en modifiant l'état de la vue pour créer du mouvement. Dans le programme suivant, la couleur du fond de l'image varie en fonction du temps.

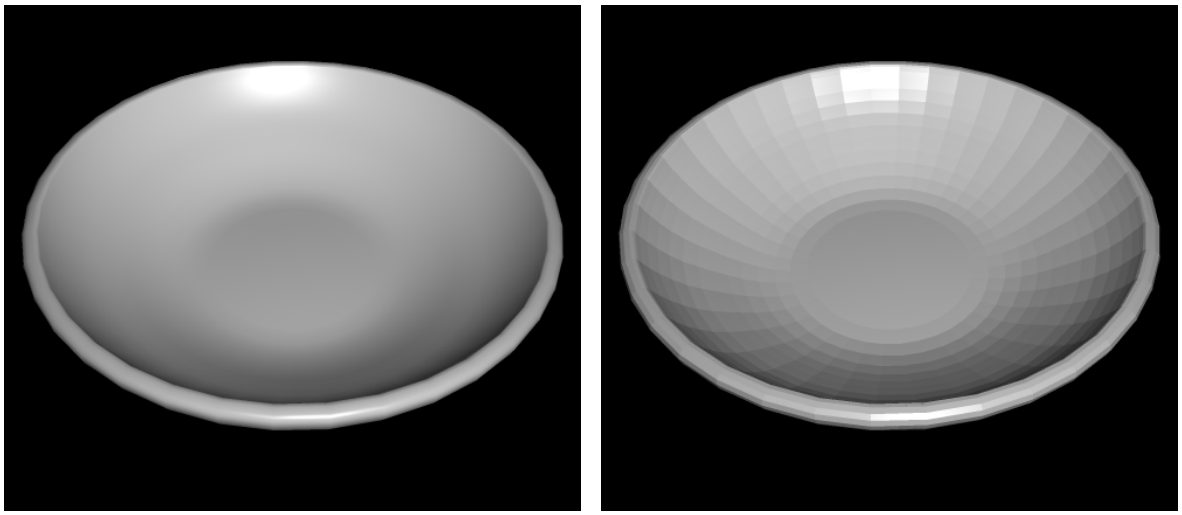
```
#include <math.h>
...

GLfloat cof_g=0.5, coef_b=0.5;

GLint parametre=0;
GLfloat vitesse=0.1;

void Affichage(void)
{
   /* coefficients RGB + A de la couleur de fond : */
  glClearColor(0, coef_g, coef_b, 0);
```

```
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  /* effaçage */
  glutSwapBuffers();  /* Envoyer le buffer à l'écran */
}

void IdleFunction(void)
{
  coef_b = sin(vitesse*parametre);
  parametre++;
  glutPostRedisplay();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
  glutCreateWindow("OpenGL et glut, TP 1");
  glEnable(GL_DEPTH_TEST);

  glutDisplayFunc(Affichage);
  glutKeyboardFunc(Clavier);
  glutSpecialFunc(Special);
  glutIdleFunc(IdleFunction);   /* appel de glutIdleFunc */

  glutMainLoop();
  return 0;
}
```

Chapitre 2

Dessiner des formes de base

La principale opération d'affichage est le dessin d'un polygone (triangle, quadrilatère,...) à l'écran. En effet, lorsqu'on souhaite représenter une surface autre qu'un polygone, on doit approcher cette surface par un maillage composé de polygones (voir la figure 2.1 et la figure 2.2).



(a) Exemple de surface (b) Aproximation par un maillage

FIGURE 2.1: L'approximation d'une surface par des polygones (maillage de surface).

2.1 Les primitives géométriques

Lorsqu'on décrit un polygone, on doit mettre les sommets du polygone entre un appel à la fonction `glBegin()` et un appel à `glEnd()`. Plusieurs primitives géométriques peuvent être dessinées, suivant l'argument passé à `glBegin()` (voir la figure 2.3).

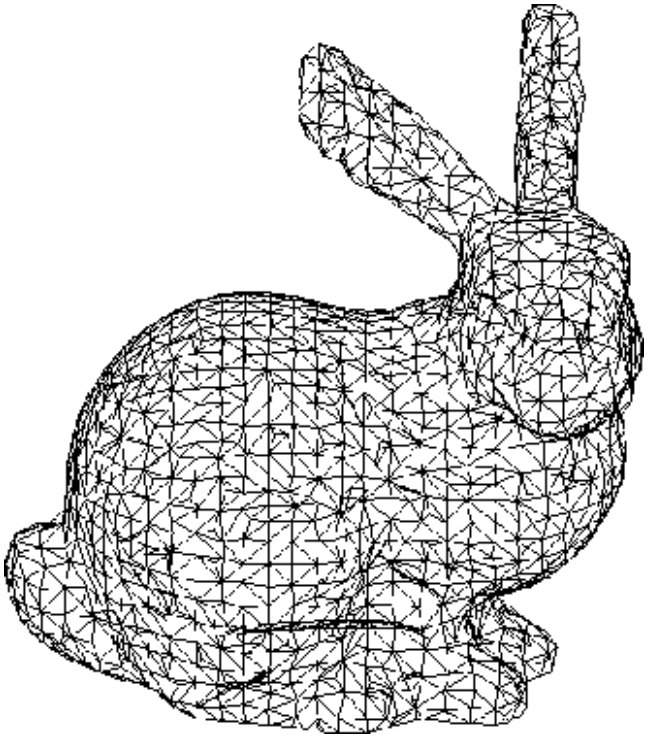


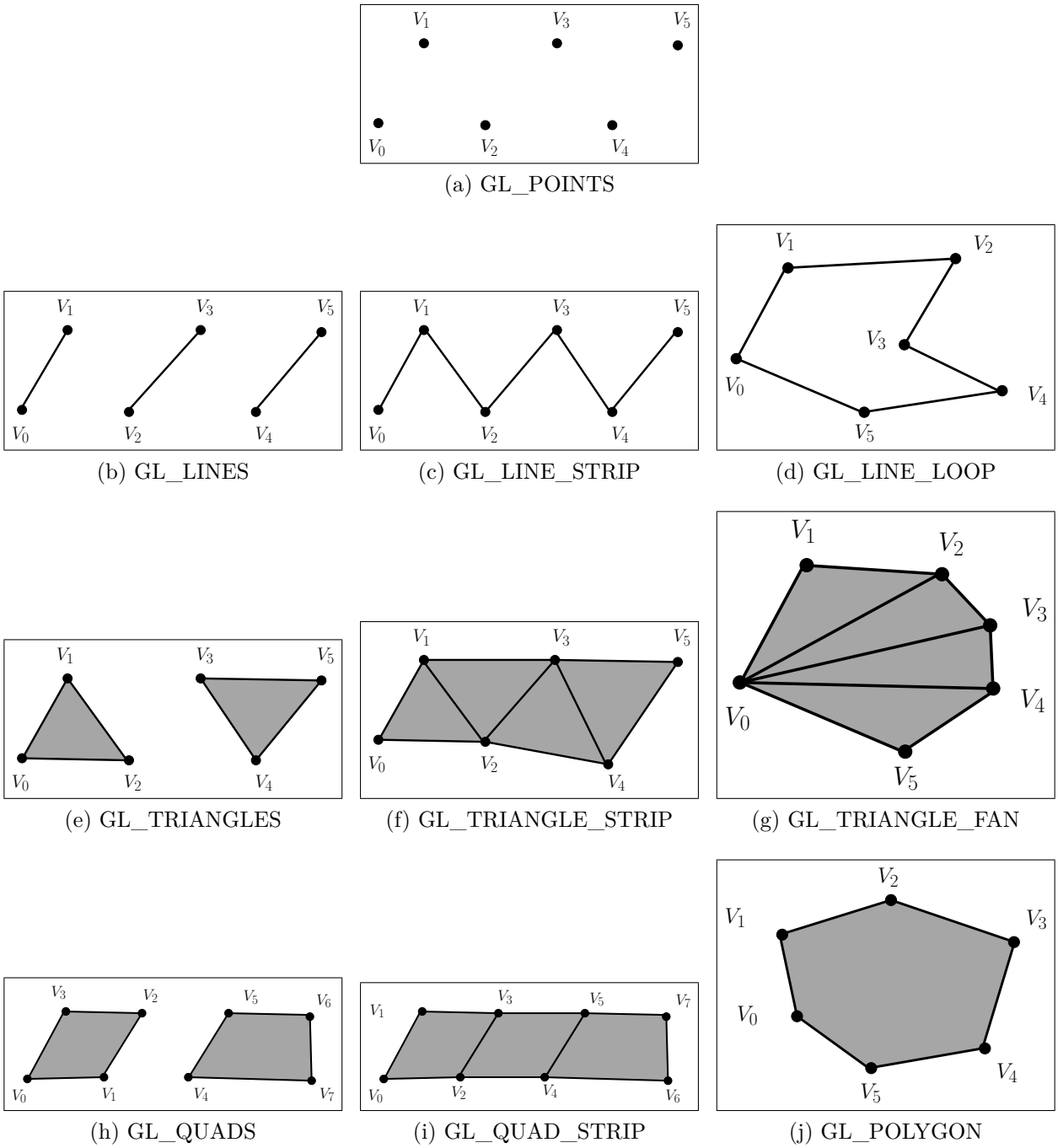
FIGURE 2.2: Exemple de maillage : reconstruction du “Stanford Bunny”.

Exemple 1. Pour dessiner un triangle rempli, on peut faire :

```
glBegin(GL_TRIANGLES);
  glVertex2f(0.0, 0.0);
  glVertex2f(1.0, 0.0);
  glVertex2f(0.0, 1.0);
glEnd();
```

Exemple 2. Pour dessiner un hexagone régulier rempli, on peut faire (en incluant la bibliothèque `math.h`) :

```
GLint i;

glBegin(GL_POLYGON);
  for (i=0 ; i<6 ; i++)
  glVertex2f(cos(2*i*M_PI/6), sin(2*i*M_PI/6));
glEnd();
```



(a) GL_POINTS

(b) GL_LINES

(c) GL_LINE_STRIP

(d) GL_LINE_LOOP

(e) GL_TRIANGLES

(f) GL_TRIANGLE_STRIP

(g) GL_TRIANGLE_FAN

(h) GL_QUADS

(i) GL_QUAD_STRIP

(j) GL_POLYGON

FIGURE 2.3: Types de primitives géométrique. Le polygone doit être convexe.

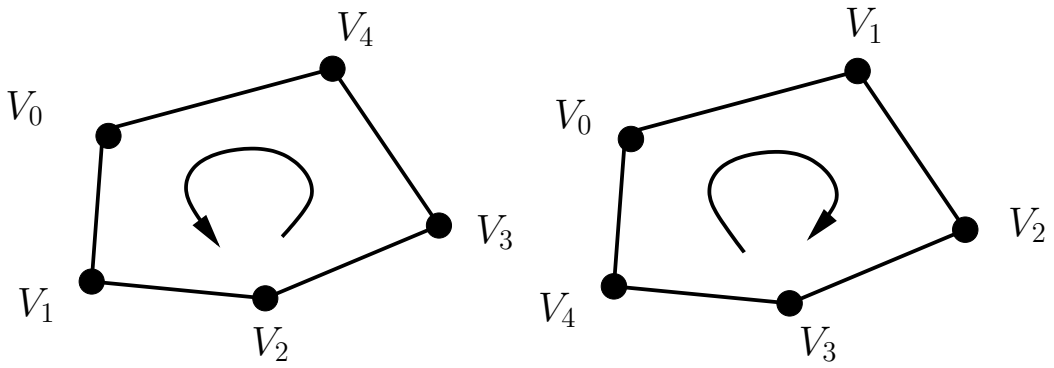
Les arguments possibles pour la fonction `glBegin()`, qui correspondent à différentes primitives géométriques, sont :

- `GL_POINTS` : seuls des points sont dessinés. (L'épaisseur des points peut être réglée avec la fonction `glPointSize()`).
- `GL_LINES` : Des segments de droites sont dessinées entre V_0 et V_1, entre V_2 et V_3, etc., les V_i étant les sommets successifs définis par des appels à `glVertex*()`. (L'épaisseur des droites peut être réglée avec la fonction `glLineWidth`).
- `GL_LINE_STRIP` : Une ligne polygonale est tracée.
- `GL_LINE_LOOP` : Une ligne polygonale fermée est tracée.
- `GL_TRIANGLES` : des triangles sont dessinés entre les trois premiers sommets, entre les trois suivants, etc... Si le nombre de sommets n'est pas multiple de 3, les 1 ou 2 derniers sommets sont ignorés.
- `GL_TRIANGLE_STRIP` : Des triangles sont dessinés entre V_0, V_1, V_2, puis entre V_1, V_2, V_3, entre V_2, V_3, V_4, etc.
- `GL_TRIANGLE_FAN` : Un éventail est dessiné formé des triangles V_0, V_1, V_2, puis V_0, V_2, V_3, etc.
- `GL_QUADS` : Un quadrilatère est un polygone à 4 sommets. Des quadrilatères sont dessinés entre les quatre premiers sommets, puis entre les 4 suivants, etc.
- `GL_QUAD_STRIP` : Une série de quadrilatères sont dessinés entre V_0, V_1, V_3, V_2, puis, V_2, V_3, V_5, V_4, puis V_4, V_5, V_7, V_6, etc.
- `GL_POLYGON` : Un polygone fermé rempli est déssiné. **Le polygone doit être convexe** faute de quoi le comportement est indéfini (dépend de l'implémentation).

2.2 Variantes de `glVertex*()` et représentations des sommets

Lorsqu'on spécifie un sommet, celui-ci est toujours dans l'espace à trois dimensions $\mathbb{R}^3$. On peut spécifier, 2, 3, ou même 4 coordonnées pour un point. Lorsqu'on ne spécifie que 2 des coordonnées, la troisième coordonnée est considérée comme valant 0 (point situé dans le plan xOy). Lorsqu'on spécifie 4 coordonnées, il s'agit de la représentation du point en coordonnées homogènes (voir la partie 3.2). De plus, on peut spécifier les coordonnées soit par une liste de paramètres, soit par un tableau (vecteur) de coordonnées. Enfin, les coordonnées peuvent être de différents types (`GLint`, `GLfloat`, `GLdouble`, etc.) À chacune de ces options correspond une fonction `glVertex*()` :

- `glVertex2f` : deux paramètres flottants en simple précision de type `GLfloat` (la troisième coordonnée vaut 0) ;
- `glVertex3f` : trois paramètres flottants en simple précision (la troisième coordonnée vaut 0)
- `glVertex2fv` : un paramètre de type tableau de 2 `GLfloat`s.
- `glVertex3fv` : un paramètre de type tableau de d `GLfloat`s.
- `glVertex2d, glVertex3d, glVertex2dv, glVertex3dv` : similaires aux précédents mais avec des floats en double précision de type `GLdouble`.
- `glVertex2i, glVertex3i, glVertex2di, glVertex3di` : similaires aux précédents mais avec des entiers 32 bits de type `GLint`.



(a) Sens `GL_CCW` (counterclockwise) (b) Sens `GL_CW` (clockwise)

FIGURE 2.4: Orientation d'un polygone. Par défaut, la vue de face correspond au cas `GL_CCW`

2.3 Modes d'affichage de polygones

Les polygones en 3D ont deux cotés : le devant et le derière. Par défaut, un polygone est vu de face (voir la figure 2.4) si sa projection dans la fenêtre graphique voit la liste de se sommets dans l'odre positif trigonométrique (on peut changer ce comportement avec la fonction `glFrontFace`).

On peut spécifier un mode d'affichage différents pour les polygones vus de face et pour les polygones vus de dos par la fonction `glPolygonMode`.

```
\void glPolygonMode(GLenum face, GLenum mode);
```

- `face` peut être `GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK` ;
- `mode` peut être `GL_POINT` (dessin uniquement des sommets du polygone), `GL_LINE` (dessin des contours du polygone) ou `GL_FILL` (dessin du polygone avec remplissage).

Enfin, on peut éliminer les faces (par exemple) qui sont vues de dos pour accélérer l'affichage, (si l'objet est fermé et vu de l'extérieur, on sait qu'aucune face vue de dos ne sera visible car elles seront cachées par d'autres faces. Pour cela, il faut activer le culling par

```
glEnable(GL_CULL_FACE);
```

et indiquer le type de polygones à éliminer par l'usage de la fonction

```
void glCullFace(GLenum mode).
```

- `face` peut être `GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK`.

2.4 Représentation d'un maillage

2.4.1 Définition d'un maillage

Un *maillage* $\mathcal{P}$ dans l'espace tridimensionnel $\mathbb{R}^3$ est la donnée de :

1. Une suite de points $P_0, P_1, \ldots, P_{n-1}$ de $\mathbb{R}^3$ appelés *sommets* du maillage ;
2. Un ensemble de *faces*, chaque face étant une suite de numéros de sommets dans $\{0, \ldots, n-1\}$.

2.4.2 Exemple

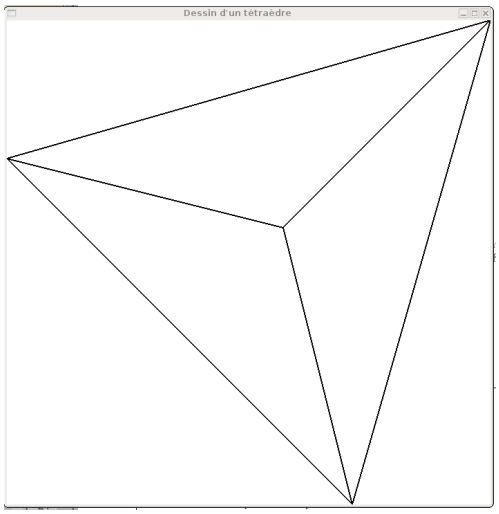


FIGURE 2.5: Exemple de maillage : un tétraèdre.

Par exemple, considérons le tétraèdre construit sur les quatre points $A = (400, 400, -10)$, $B = (700, 700, -10)$, $C = (0, 500, -10)$ et $D = (500, 0, -500)$ (voir figure 2.5). Le maillage correspondant est la donnée de :

1. Les sommets $P_0 = A$, $P_1 = B$, $P_2 = C$, et $P_3 = D$;
2. Les quatre faces qui sont :
   - La face numéro 0 représentant le triangle OAB : $(0, 1, 2)$;
   - La face numéro 1 représentant le triangle OAC : $(0, 1, 3)$;
   - La face numéro 2 représentant le triangle OBC : $(0, 2, 3)$;
   - La face numéro 3 représentant le triangle ABC : $(1, 2, 3)$.

Voici un programme affichant le tétraèdre comme représenté sur la figure 2.5 :

polygon//progc/exMaillage.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  GLushort largeur_fenetre=700;
6  GLushort hauteur_fenetre=700;
7
8  GLint nvertices=4, nfaces=4;
9
10 GLfloat vertices[][3] = {
11   {400.0, 400.0, -10.0},
12   {700.0, 700.0, -10.0},
13   {0.0, 500.0, -10.0},
14   {500.0, 0.0, -500.0}
15 };
16
17 GLuint faces[][3] = {
18   {0, 1, 2},
19   {0, 1, 3},
```

```
20    {0, 2, 3},
21    {1, 2, 3}
22  };
23
24  void Affichage(void)
25  {
26    int i, j;
27    glClearColor(1.0,1.0,1.0,1.0);
28    glClear(GL_COLOR_BUFFER_BIT);
29    glColor3f(0.0, 0.0, 0.0);
30    glLineWidth(2);
31    glBegin(GL_TRIANGLES);
32    for (i=0 ; i<nfaces ; i++)
33      for (j=0 ; j<3 ; j++)
34    glVertex3fv(vertices[faces[i][j]]);
35    glEnd();
36
37    glutSwapBuffers();  /* Envoyer le buffer  l 'cran */
38  }
39
40  int main(int argc, char**argv)
41  {
42    glutInit(&argc, argv);
43    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
44    glutInitWindowPosition(100,100);
45    glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
46    glutCreateWindow("Dessin d'un  ttradre");
47    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
48    glutDisplayFunc(Affichage);
49
50    /* Dfinition du volume visible */
51    /* Ncessaire pour afficher des objets  gomtriques */
52    /* voir Chapitre suivant pour personaliser */
53    glMatrixMode(GL_PROJECTION);
54    glLoadIdentity();
55    glOrtho(0,largeur_fenetre , 0, hauteur_fenetre, 1, 600);
56    glMatrixMode(GL_MODELVIEW);
57    glLoadIdentity();
58
59    glutMainLoop();
60    return 0;
61  }
```

Chapitre 3

Positioner caméras et objets

3.1 Gérer la caméra et la perspective

3.2 Coordonnées homogènes

3.2.1 Définition des coordonnées homogènes

On introduit un système de coordonnées différent des coordonnées cartésiennes pour repérer les points dans l'espace 3D. Pour celà, on ajoute une composante W non nulle aux trois composantes (x, y, z). Ainsi, un point M de l'espace 3D sera repéré en coordonnées homogènes par un quadruplet :

$$M(x, y, z, W)$$

Les coordonnées de M ne sont pas uniques, mais deux quadruplets (x, y, z, W) et (x', y', z', W') représentent le même point si ces quadruplets sont multiples l'un de l'autre (liés dans $\mathbb{R}^3$).

Connaissant les coordonnées cartésiennes (x, y, z) d'un point M, on peut obtenir un représentant de M en coordonnées homogènes par $(x, y, z, 1)$ (c'est à dire que $W = 1$).

Connaissant un représentant (x, y, z, W) de M en coordonnées homogènes, on peut obtenir les coordonnées cartésiennes de M par $M = (\frac{x}{W}, \frac{y}{W}, \frac{z}{W})$.

L'intérêt de représenter les points par des coordonnées homogènes est de pouvoir représenter toute application affine par une seule matrice, comprenant la partie linéaire et la translation.

3.2.2 Coordonnées homogènes et applications affines

Pour effectuer une translation T de vecteur $\overrightarrow{v} = (x_v, y_v, z_v)$ en coordonnées homogènes, il suffit de faire une multiplication matricielle :

$$T \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_v \\ 0 & 1 & 0 & y_v \\ 0 & 0 & 1 & z_v \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_v \\ y + y_v \\ z + z_v \\ 1 \end{bmatrix}$$

Pour appliquer une application linéaire f dont la matrice en coordonnées cartésienne est :

$$M_f = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

il suffit de faire en coordonnées homogènes la multiplication matricielle par la matrice

$$\mathcal{M}_f = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

N'importe quelle application affine, composée d'une application linéaire et d'une translation, peut être obtenue en coordonnées homogènes par la multiplication par une matrice :

$$\mathcal{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & x_v \\ a_{2,1} & a_{2,2} & a_{2,3} & y_v \\ a_{3,1} & a_{3,2} & a_{3,3} & z_v \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La matrice en coordonnées homogènes $\mathcal{M}_{f\circ g}$ de la composée de deux application affines f et g est le produit $\mathcal{M}_f.\mathcal{M}_g$ des matrices associées aux applications affines en coordonnées homogènes. De même, la matrice $\mathcal{M}_{f^{-1}}$ en coordonnées homogènes de l'inverse d'une application affine bijective est l'inverse $\mathcal{M}_f^{-1}$ de la matrice de cette application affine.

3.3 Changements de repère et coordonnées homogènes

Si l'on se donne deux repères affines $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ et $(O', \overrightarrow{\imath_1}, \overrightarrow{\jmath_1}, \overrightarrow{k_1})$ tels que

$$\left\{ \begin{array}{rcl} \overrightarrow{\imath_1} & = & a_{1,1}\overrightarrow{\imath} + a_{2,1}\overrightarrow{\jmath} + a_{3,1}\overrightarrow{k} \\ \overrightarrow{\jmath_1} & = & a_{1,2}\overrightarrow{\imath} + a_{2,2}\overrightarrow{\jmath} + a_{3,2}\overrightarrow{k} \\ \overrightarrow{k_1} & = & a_{1,3}\overrightarrow{\imath} + a_{2,3}\overrightarrow{\jmath} + a_{3,3}\overrightarrow{k} \end{array} \right. ,$$

On peut calculer les coordonnées X, Y et Z du point P dans le deuxième repère à partir des coordonnées x, y et z dans le premier repère.

En notant

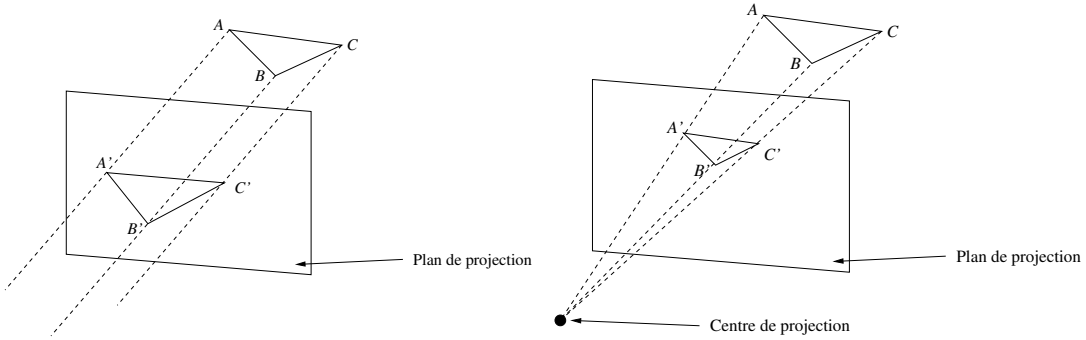
$$M = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}.$$

la *matrice de passage* du changement de base, la matrice de passage M est automatiquement inversible. En notant (O'_x, O'_y, O'_z) les coordonnées du point O' dans le repère $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$, on a :

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M^{-1}.\left(\begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix}\right)$$

En inversant les formules, on peut aussi calculer les coordonnées x, y et z dans le premier repère en fonction des coordonnées X, Y et Z dans le second repère :

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = M.\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix}.$$



(a) La projection parallèle (b) La projection en perspective

FIGURE 3.1: Les deux principaux types de projection.

Ces formules sont des formules de transformations affines qui s'écrivent en coordonnées homogènes

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & O'_x \\ a_{2,1} & a_{2,2} & a_{2,3} & O'_y \\ a_{3,1} & a_{3,2} & a_{3,3} & O'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Un changement de repère affine consiste donc en une multiplication matricielle en coordonnées homogènes.

3.4 Caméras et projections

Nous voyons ici comment projeter des données 3D sur un plan dans le but de visualiser ces données. Nous considérons ici des projections très particulières, le plan sur lequel on projette étant perpendiculaire au troisième axe de coordonnées Oz. Ceci correspondra au point de vue très particulier d'un observateur placé sur l'origine O et regardant dans la direction de l'axe des z. Par des changements de repères appropriés, on remène le cas d'un point de vue quelconque de l'observateur au cas particulier traité ici.

3.4.1 Projection parallèle

La *projection parallèle* est la projection sur un plan parallèlement à une direction (voir figure 3.1a).

Cette projection correspond à un observateur placé à l'infini. Elle est simple, mais n'est pas très réaliste. Plaçons nous dans un repère $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ tel que le plan de projection soit le plan $z = 0$ et la direction de projection soit parallèle au vecteur $\overrightarrow{k}$. Pour obtenir le projeté d'un point (x, y, z), il suffit d'oublier la coordonnée z : Le projeté a pour coordonnées : $x_p = x$, $y_p = y$, $z_p = 0$. Les formules de cette projection sont donc particulièrement simples.

3.4.2 Projection en perspective

Dans le modèle de caméra Pinhole, il y a un *centre de projection* (voir figure 3.1b), qui correspond à la position de l'observateur. Ce modèle de caméra est plus réaliste que le modèle

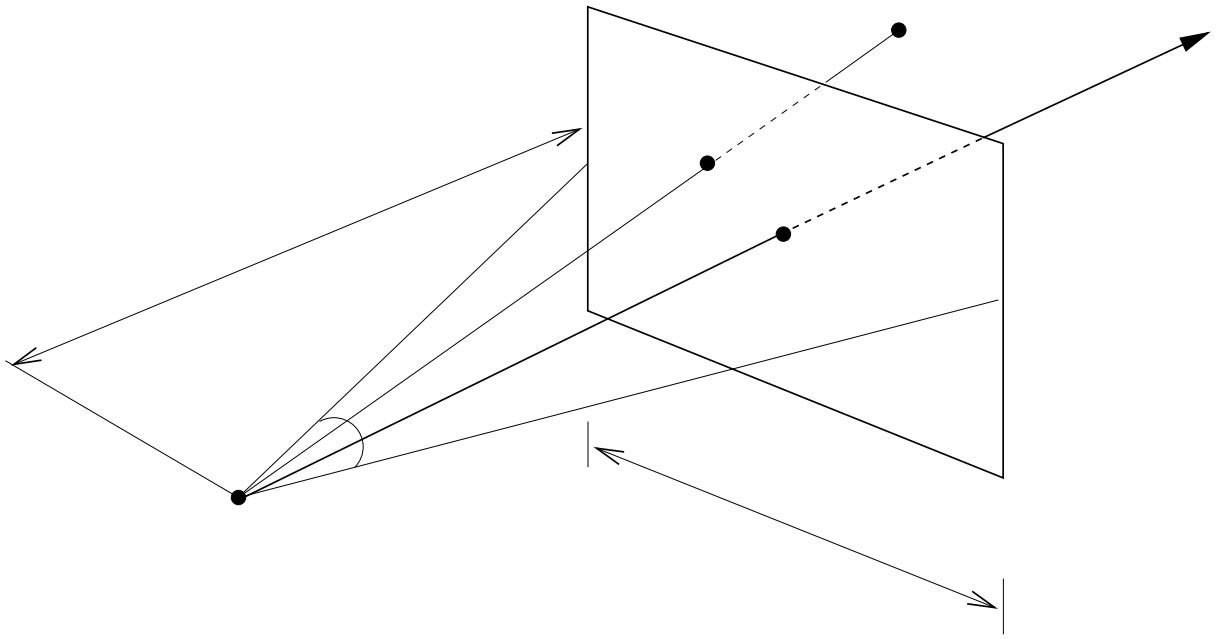


FIGURE 3.2: Le modèle de caméra *pinhole*

avec projection parallèle. Par contre, les formules donnant les coordonnées du projeté (x_p, y_p, z_p) d'un point en fonction des coordonnées (x, y, z) du point sont un peu plus compliquées.

3.4.3 Calcul des coordonnées du projeté d'un point

Plaçons nous dans un repère $(O, \overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})$ lié à la caméra, tel que le plan de projection soit le plan d'équation $z = d$ et tel que le centre de projection ait pour coordonnées $(0, 0, 0)$ (voir figure 3.2).

La troisième coordonnée z_p du projeté sera évidemment égale à d de part la définition du plan de projection.
D'après le théorème de Thalès (voir figure 3.3), on a :

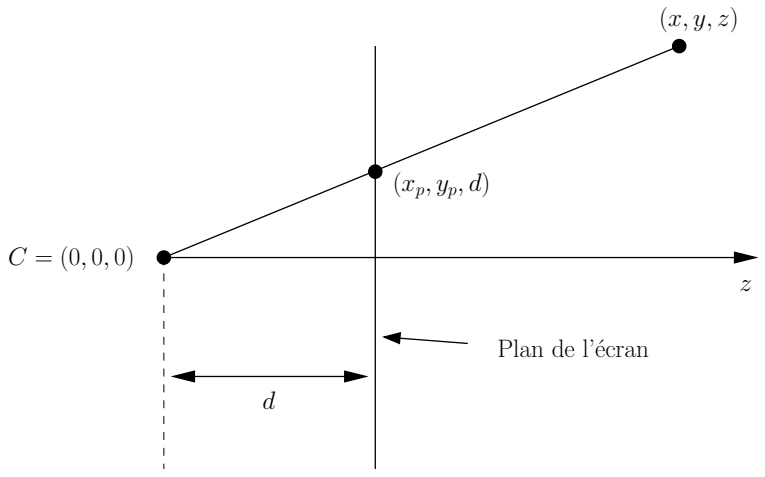


FIGURE 3.3: Calcul des coordonnées du projeté.

$$\frac{x_p}{d} = \frac{x}{z} \quad \text{et} \quad \frac{y_p}{d} = \frac{y}{z}$$

En multipliant ces égalités par d, on obtient :

$$x_p = x * \frac{d}{z} \quad \text{et} \quad y_p = y * \frac{d}{z}$$

En coordonnées homogènes, ces formules s'écrivent :

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = \begin{pmatrix} x\frac{d}{z} \\ y\frac{d}{z} \\ d \\ 1 \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ z \\ \frac{z}{d} \end{pmatrix}$$

D'où,

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = \mathcal{M}. \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

avec

$$\mathcal{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{pmatrix}$$

La projection en perspective s'exprime donc par la multiplication par une matrice en coordonnées homogènes.

3.4.4 Caméra Pinhole

Une caméra "Pinhole" est une spécification d'un point de vue de l'observateur avec projection en perspective. L'idée est celle d'un observateur regardant une scène à travers un trou d'épingle (*pin hole* en anglais), qui est le centre de projection. Cela correspond à l'un des tout premiers modèles historiques d'appareil photo. On se place ici dans le cas où la caméra est placé sur le point $O = (0, 0, 0)$, la direction de visée étant l'axe des z. Nous avons vu dans la partie 3.4.3 comment calculer les coordonnées du projeté d'un point en fonction du paramètre d donnant l'équation $z = d$ du plan de projection.

Cependant, dans une application graphique 3D, le paramètre d n'est pas donné directement, mais doit être déterminé à partir de l'angle d'ouverture θ_c de la caméra et la largeur dim_x de l'image à calculer. L'angle d'ouverture θ_c est l'angle sous lequel l'observateur placé en $O = (0, 0, 0)$ doit voir le rectangle de largeur dim_x constitué par l'image à calculer, ce rectangle étant placé dans le plan $z = d$ centré au point $A = (0, 0, d)$ (voir la figure 3.4). On peut calculer comme suit la valeur de d à partir de θ_c et dim_x.

Soit $B = (\frac{dim_x}{2}, 0, d)$ le point de l'image à calculer situé le plus à droite par rapport à l'observateur. Le triangle OAB est rectangle en A, et l'angle en O de ce triangle est égal à $\frac{\theta_c}{2}$. On a donc

$$d = OA = \frac{AB}{\tan(\frac{\theta_c}{2})} = \frac{dim_x}{2\tan(\frac{\theta_c}{2})}$$

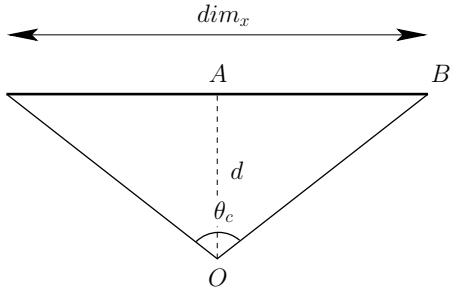


FIGURE 3.4: Calcul de la profondeur d du plan de l'écran

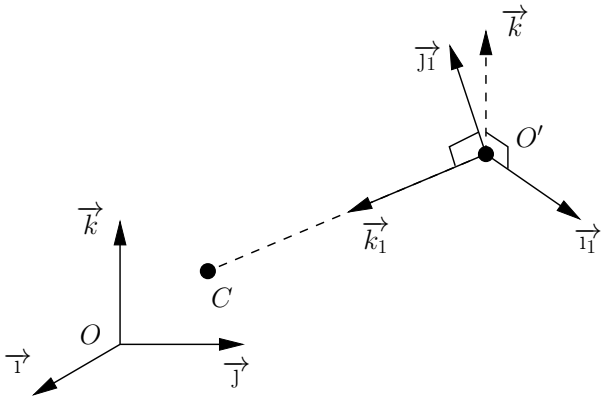


FIGURE 3.5: Le repère lié à la caméra

3.5 Positionnement quelconque d'une caméra

3.5.1 Repère lié à une caméra

En synthèse d'images, on considère des *caméras*, qui définissent le point de vue à partir duquel l'observateur voit la scène. Chaque modèle de caméra correspond à un type de projection dans l'espace. Nous avons vu dans la partie 3.4 qu'il existe principalement deux types de projection, la projection parallèle et la projection en perspective.

Les algorithmes d'affichage 3D supposent que la caméra (ou l'observateur) est placé au point $O = (0, 0, 0)$, et regarde dans la direction de l'axe Oz. Cette hypothèse peut être reformulée comme suit : les coordonnées qui définissent les objets (coordonnées des sommets de polyèdres), et les coordonnées des vecteurs normaux aux sommets de polyèdres, sont calculées *dans un repère lié à la caméra* (voir les figures 3.5 et 3.10).

Une scène 3D possède un repère fixe $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ dans lequel l'utilisateur définit les objets, les sources lumineuses, et la position de la caméra. Ce repère s'appelle le *repère de la scène*, ou le *repère du monde*.

La caméra possède un repère $(O', \overrightarrow{\imath_1}, \overrightarrow{\jmath_1}, \overrightarrow{k_1})$ qui lui est propre, appelé *repère de la caméra*, dont l'origine O' coïncide avec la position de la caméra, et tel que la direction $\overrightarrow{k_1}$ du troisième axe de coordonnée coïncide avec la direction de visée de la caméra.

On associe donc à la caméra une matrice $M = (a_{i,j})_{i,j=1,2,3}$ telle que :

$$\left\{\begin{array}{rcl} \overrightarrow{\imath_1} & = & a_{1,1}\overrightarrow{\imath} + a_{2,1}\overrightarrow{\jmath} + a_{3,1}\overrightarrow{k} \\ \overrightarrow{\jmath_1} & = & a_{1,2}\overrightarrow{\imath} + a_{2,2}\overrightarrow{\jmath} + a_{3,2}\overrightarrow{k} \\ \overrightarrow{k_1} & = & a_{1,3}\overrightarrow{\imath} + a_{2,3}\overrightarrow{\jmath} + a_{3,3}\overrightarrow{k} \end{array}\right.$$

Rappelons que M est la matrice de passage du repère de la scène au repère de la caméra.

3.5.2 Initialisation d'une caméra

Le but de cette partie est d'expliquer comment on peut calculer la matrice $M = (a_{i,j})_{i,j=1,2,3}$ d'une caméra à partir de la position et du point de focalisation (le *centre*) de cette caméra.

Soit O' la position d'une caméra et C un point, appelé le *centre* de la caméra, vers lequel est dirigé la caméra. Calculer les coefficients $(a_{i,j})$ de la matrice M associée à la caméra revient à calculer les coordonnées des vecteurs $\overrightarrow{i_1}$, $\overrightarrow{j_1}$ et $\overrightarrow{k_1}$ dans le repère $(O, \overrightarrow{\imath}, \overrightarrow{\jmath}, \overrightarrow{k})$ de la scène.

Le vecteur $\overrightarrow{k_1}$ du repère de la caméra, qui est le vecteur unitaire dans la direction de visée, est égal à

$$\overrightarrow{k_1} = \frac{\overrightarrow{O'C}}{||\overrightarrow{O'C}||}$$

Nous avons un choix pour définir les deuxième et troisième vecteur du repère de la caméra. En effet, pour le moment, nous n'avons pas fixé exactement la position de la caméra, qui peut tourner autour de l'axe $O'C$. Nous allons fixer le vecteur $\overrightarrow{i_1}$ de sorte que l'axe des X du repère de la caméra soit perpendiculaire à l'axe des z du repère de la scène. De cette manière, l'axe des z de la scène apparaîtra vertical lors de l'affichage, ce qui est assez naturel.

Le vecteur $\overrightarrow{i_1}$ doit donc être orthogonal au troisième vecteur $\overrightarrow{k}$ du repère de la scène. De plus, le vecteur $\overrightarrow{i_1}$ doit être orthogonal au vecteur $\overrightarrow{k_1}$ de la direction de visée, puisque nous souhaitons obtenir un repère $(O', \overrightarrow{i_1}, \overrightarrow{j_1}, \overrightarrow{k_1})$ orthonormé. Nous n'avons pas beaucoup de choix, et nous définirons donc

$$\overrightarrow{i_1} = \frac{\overrightarrow{k_1} \wedge \overrightarrow{k}}{||\overrightarrow{k_1} \wedge \overrightarrow{k}||}$$

Enfin, le vecteur $\overrightarrow{j_1}$ doit être à la fois orthogonal à $\overrightarrow{i_1}$ et $\overrightarrow{k_1}$, de manière à ce que le repère $(O', \overrightarrow{i_1}, \overrightarrow{j_1}, \overrightarrow{k_1})$ soit orthonormé direct. On a donc :

$$\overrightarrow{j_1} = \frac{\overrightarrow{k_1} \wedge \overrightarrow{i_1}}{||\overrightarrow{k_1} \wedge \overrightarrow{i_1}||}$$

3.5.3 Transformation des objets dans le repère de la caméra

Une fois la matrice M associée à la caméra initialisée, dans le but d'utiliser les algorithmes d'affichage, nous devons exprimer les coordonnées des sommets, ainsi que les vecteurs normaux aux sommets dans le cas d'une utilisation du modèle de Phong, dans le repère de la caméra.

Pour cela, on utilise simplement les formules de changement de base pour calculer les coordonnées (X, Y, Z) des sommets dans le repère de la caméra à partir de ses coordonnées (x, y, z) dans le repère de la scène. Les coordonnées de la position des sources lumineuses doivent subir la même transformation.

3.6 Transformation de visualisation dans OpenGL

3.6.1 Principe des transformations géométriques dans *OpenGL*

Avant d'être affiché à l'écran, un objet subit plusieurs transformations (voir la figure 3.6).

Dans l'ordre chronologique, *OpenGL* effectue les transformations suivantes :

1. La transformation du modèle : les objets graphiques sont définis dans un repère ou ils sont simples à concevoir, puis on transforme ces objets par des rotations, translations et changements d'échelle (affinités) pour les mettre dans le repère du monde. On fait subir ces transformations aux objets au moyen des fonctions `glRotatef`, `glTranslatef` et `glScalef` dans le mode `GL_MODELVIEW`.
2. La transformation de la caméra : une fois dans le repère du monde, tous les objets subissent un changement de repère vers le repère de la caméra. On fait subir cette transformation aux objets au moyen des fonctions `gluLookAt`, `glRotatef`, `glTranslatef` et `glScalef` dans le mode `GL_MODELVIEW`.
3. Projection : une fois dans le repère de la caméra, les objets subissent une projection vers une image $2D$ (dans un buffer `OpenGL`). Pour régler les paramètres de cette projection, on utilise la fonction `gluPerspective` dans le mode `GL_PROJECTION`.
4. Transformation $2D$: une fois générée l'image $2D$ dans un buffer, elle subit des changement d'échelle sur les axes (affinités orthogonales) pour être ajustée aux dimensions de la fenêtre graphique à l'écran. Cette dernière transformation est paramétrée par la fonction `glViewport`.

Le principe de ces transformations est très général, nous donons dans la suite une utilisation de base des fonctionalités d'*OpenGL* en la matière.

3.6.2 Paramètres de la caméra

Dans la fonction de redimensionnement (déclarée dans *glut* par un appel de `glutReshapeFunc`, On peut réglér les paramètres de la projection et de la transformation $2D$.

3.6.2.a La fonction `glviewport`

En général il n'est pas commode de créer toutes les scènes de façon à ce qu'elles se projettent directement sur la fenêtre graphique. On préfère utiliser des coordonnées normalisées dans le plan (par exemple avec des coordonnées allant de -1 à $+1$). La fonction `glViewport` spécifie une transformation affine pour passer des coordonnées $2D$ de l'image calculée aux coordonnées (en nombre de pixels) de la fenêtre (voir la figure 3.7). Ceci permet de choisir la portion du plan $2D$ qui sera visible. La fonction `glViewport` a pour prototype :

```
void glViewport(GLint x0, GLint y0, GLsizei dimx, GLsizei dimy);
```

où
- (x_0, y_0) sont les coordonnées du milieu de la portion de plan visible souhaitée ;
- L et H sont respectivement la largeur et la hauteur de la portion de plan visible souhaitée.

La transformation est

$$X_{fenetre} = (x_{normalise} + 1)\frac{dim_x}{2} + x_0$$

$$Y_{fenetre} = (y_{normalise} + 1)\frac{dim_y}{2} + y_0$$

Pour que l'aspect des objets soit préservé par la transformation c'est à dire pour que le changement d'échelle soit le même en x et en y, il faut que les rapports suivants soient égaux :

$$\frac{dim_x}{dim_y} = \frac{\texttt{largeur\_fenetre}}{\texttt{hauteur\_fenetre}}$$

On note

$$aspect = \frac{dim_x}{dim_y}.$$

3.6.2.b La fonction `gluPerspective`

La fonction `gluPerspective` permet de régler les caractéristiques de la céméra (voir la figure 3.8) telles que l'angle d'ouverture θ_y (en y).

Lorsqu'on affiche une scène avec *OpenGL*, seule une partie de la scène est visible. La partie qui

1. Se projette dans la portion de plan visible (de taille $dim_x \times dim_y$) ;
2. Est comprise entre deux plans $z = z_{proche}$ et $z = z_{eloigne}$.

Le prototype de la fonction `gluPerspective` est le suivant :

```
void gluPerspective(GLdouble theta_y,
                    GLdouble aspect,
                    GLdouble z_proche,
                    GLdouble z_eloigne)
```

Avant d'utiliser `gluPerspective`, il faut passer en mode `GL_PROJECTION`, et il faut réinitialiser la transformation de projection à l'identité.

En général, on appelle les fonctions `glViewport` et `gluPerspective` dans la fonction de redimentionnement de la fenêtre (avec la *glut*, cette fonction est déclarée par un appel à `glutReshapeFunc` dans le `main`. La fonction de redimensionnement est alors automatiquement appelée lorsque l'utilisateur redimentionne la fenêtre graphique.

```
...
GLdouble theta_y=45;
...
void Redimensionnement(int l, int h)
{
/* l et h sont les nouvelles dimensions de la fenêtre */

  largeur_fenetre = l;  /* mise à jour des variables */
  hauteur fenetre = h;
  glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
  glMatrixMode(GL_PROJECTION);  /* passage en mode GL_PROJECTION */
  glLoadIdentity(); /* réinitialisation de la matrice de transformation*/
  gluPerspective(theta_y,l/(GLdouble)h, 0.01, 10000);
  glMatrixMode(GL_MODELVIEW);  /* repassage en mode GL_MODELVIEW */
}

int main(void)
{
  glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
  glutInitWindowPosition(pos_x_fenetre,pos_y_fenetre);
  glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
```

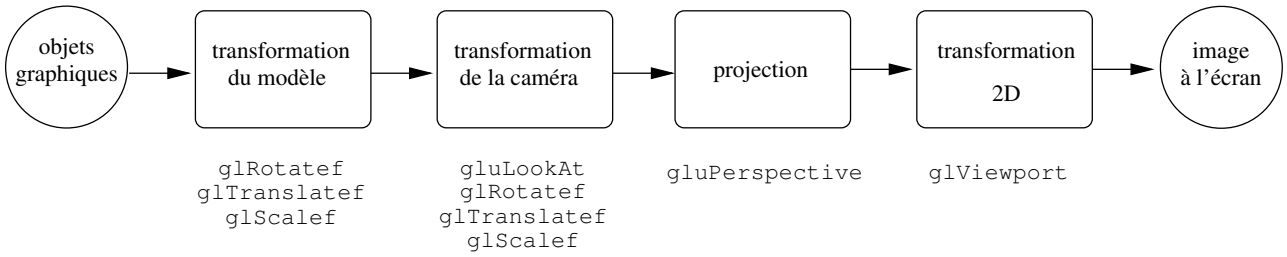


FIGURE 3.6: Les transformations géométriques subies par les objets avant affichage

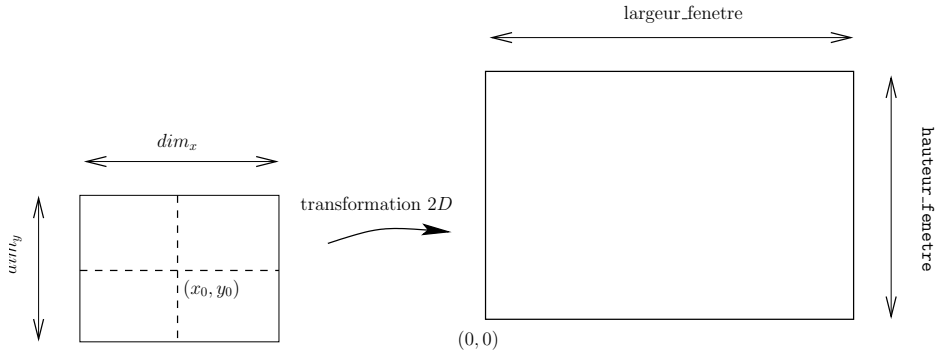


FIGURE 3.7: Les paramètres de `glViewport`

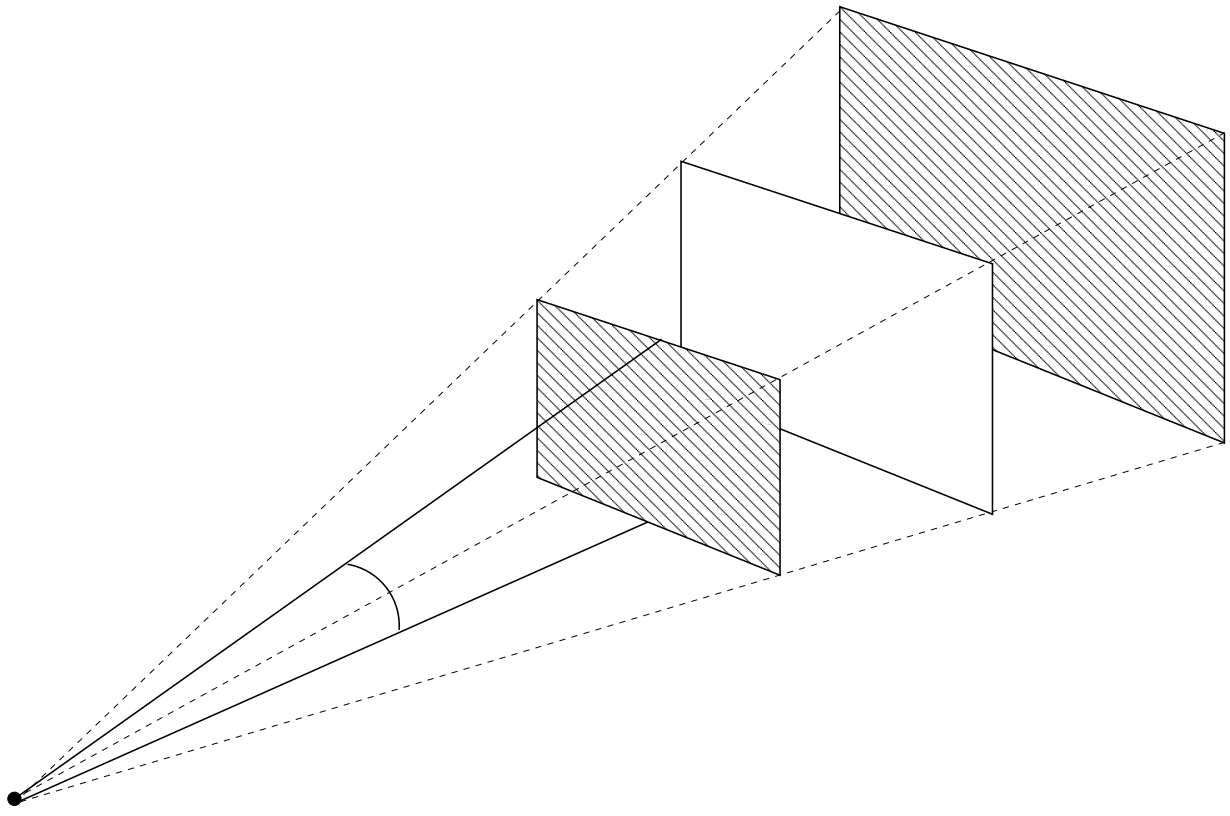


FIGURE 3.8: Les paramètres de `gluPerspective`

```
  glutCreateWindow("OpenGL et glut, TP 1");
  glEnable(GL_DEPTH_TEST);

  glutDisplayFunc(Affichage);
  glutKeyboardFunc(Clavier);
  glutSpecialFunc(Special);
  glutMotionFunc(BougeSouris);
  glutMouseFunc(PresseBouton);
  glutReshapeFunc(Redimensionnement);

  glutMainLoop();
  return 0;
}
```

3.6.3 Positionnement de la caméra

La fonction `gluLookAt` permet de modifier du `GL_MODELVIEW` pour régler la transformation de la caméra (voir la figure 3.9).

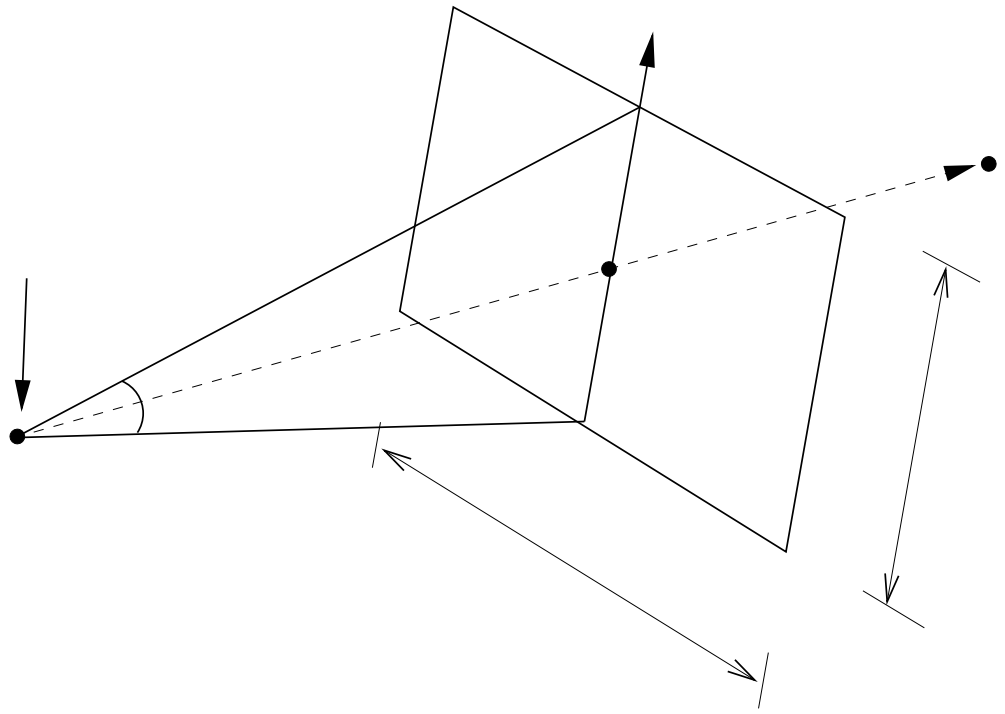


FIGURE 3.9: Les paramètres de `gluLookAt`

Les paramètres de `gluLookAt` sont :

1. les coordonnées de la position (pos_x, pos_y, pos_z) de la caméra ;
2. Les coordonnées (C_x, C_y, C_z) d'un point dans la direction de visée (ce point est aussi appelé le *centre*) ;
3. Les coordonnées d'un vecteur $\overrightarrow{V}$ qui doit apparaître vertical dans l'image obtenue par projection.

Le prototype de gluLookAt est :

```
void gluLookAt(GLdouble posx, GLdouble posy, GLdouble posz,
               GLdouble Cx, GLdouble Cy, GLdouble Cy,
               GLdouble Vx, GLdouble Vy, GLdouble Vz);
```

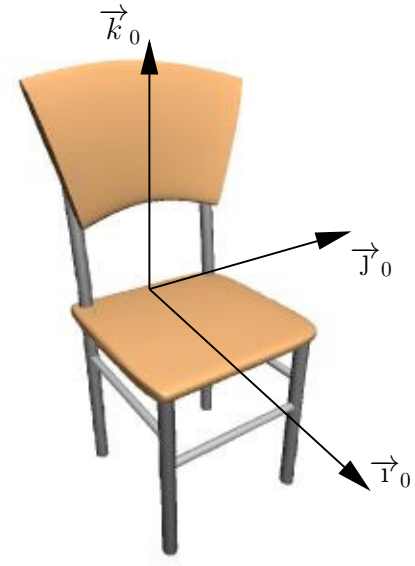
La fonction gluLookAt doit être utilisée dans le mode GL_MODELVIEW.

```
void Affichage(void)
{
  glClearColor(1.0,1.0,1.0,1.0);
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();     /* réinitialiation de la matrice de GL_MODELVIEW */
  gluLookAt(10, 10, 10,   /* réglage de la caméra */
            0,0,0,
            0,1,0);
  glColor3f(0.0,0.0,0.0); /* couleur des traits dessin en noir */
  glutWireTeapot(5);  /* dessin d'une théière */
  glutSwapBuffers();  /* Envoyer le buffer à l'écran */
}
```

3.7 Positionner un objet dans une scène

3.7.1 Positioner un objet

Chaque objet 3D est construit et modélisé dans un repère qui lui est propre, appelé repère de l'objet. Pour positioner l'objet dans le repère du monde, nous effectuons un changement de repère (voir la figure 3.10). Dans ce cas, il importe de respecter un ordre pour les changements de repères successifs. Nous utiliserons dans ce chapitre l'expression matricielle en coodonnées homogènes des changements de repère (voir la partie 3.2).



(a) repère lié à une chaise

(b) repère lié à la caméra

(c) repères du monde, de la chaise, et de la caméra

FIGURE 3.10: Les changements de repère dans la construction et le rendu d'une scène 3D

Pour tranformer l'objet de son repère vers le repère du monde, nous multiplierons les co-

ordonnées des sommets ou des points de contrôle par une matrice $\mathcal{T}$. Cette matrice $\mathcal{T}$ transformera les vecteurs $(1,0,0)$, $(0,1,0)$ et $(0,0,1)$ dans le repère de l'objet en des vecteurs $\overrightarrow{i}_0$, $\overrightarrow{j}_0$ et $\overrightarrow{k}_0$ (voir la figure 3.10). La matrice $\mathcal{T}$ n'est autre que la matrice de passage du repère du monde au repère de l'objet.

Pour passer du repère du monde au repère de la caméra, nous multiplierons par la matrice $\mathcal{M}^{-1}$, où $\mathcal{M}$ est la matrice en coordonnées homogènes associée à la caméra.

La transformation totale à appliquer à un sommet ou un point de contrôle P de l'objet sera (en coordonnées homogènes) :

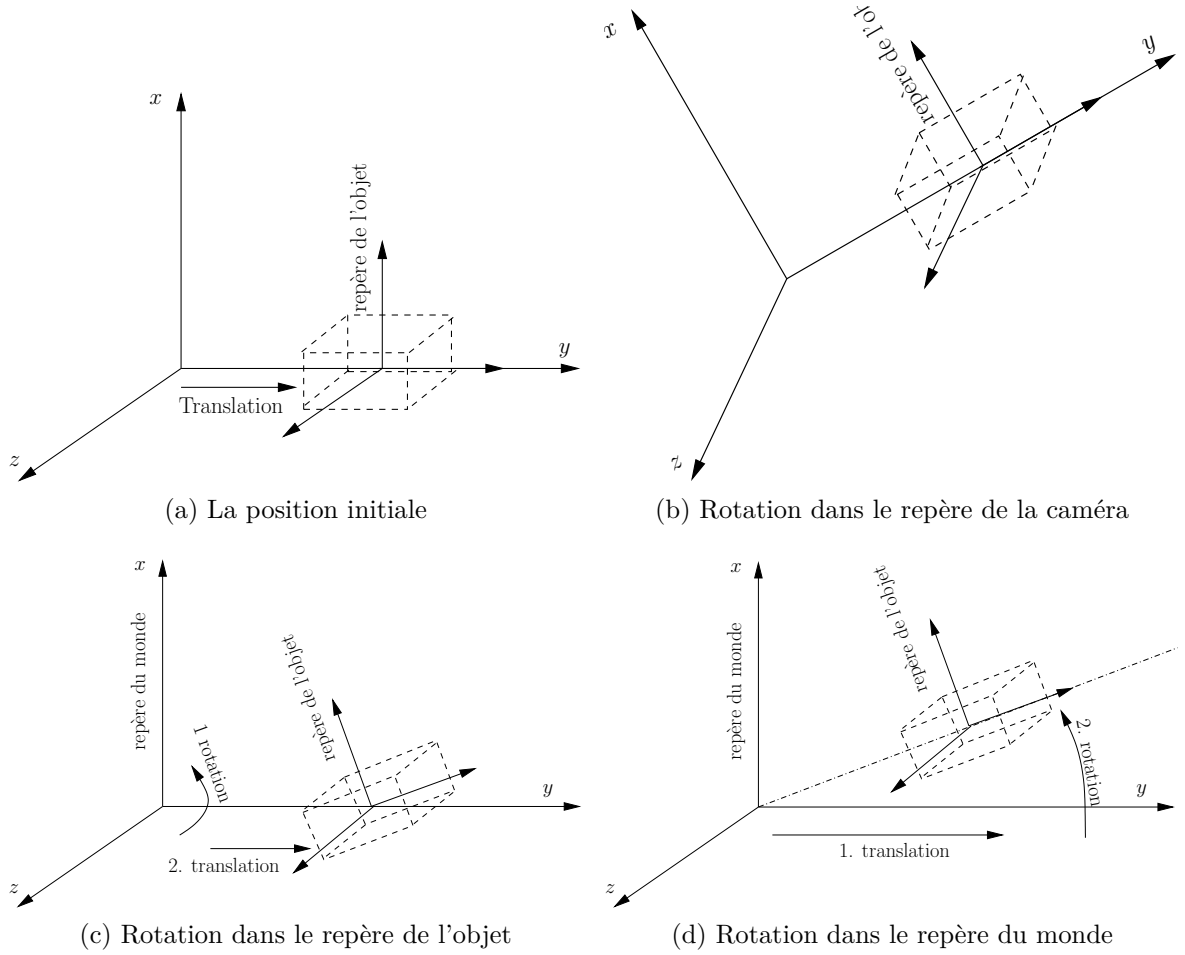
$$P' = \mathcal{M}^{-1}.\mathcal{T}.P$$

3.7.2 Transformer un objet

Pour appliquer une transformation d'un objet sur lui-même, c'est à dire dans son propre repère (voir la figure 3.11b pour le cas d'une rotation de 30 degrés), il faut multiplier la matrice de passage du repère du monde au repère de l'objet, **à droite**, par la matrice $\mathcal{R}$ de la transformation. En coordonnées homogènes, on obtient :

$$P' = \mathcal{M}^{-1}.\mathcal{T}.\mathcal{R}.P$$

Considérons l'exemple d'une parallélépipède translaté le long de l'axe des z. Le repère de l'objet est translaté par rapport au repère du monde. Pour faire une rotation dans le repère de l'objet, le parallélépipède doit subir d'abord la rotation, puis la translation (voir la figure 3.11).



(a) La position initiale

(b) Rotation dans le repère de la caméra

(c) Rotation dans le repère de l'objet

(d) Rotation dans le repère du monde

FIGURE 3.11: Rotations d'un objet initial autour de l'axe Oz dans différents repères.

Pour effectuer une transformation de l'objet dans le repère du monde (voir la figure 3.11d), il faut muliplier **à gauche** la matrice de passage du repère du monde au repère de l'objet par la matrice da la transformation :

$$P' = \mathcal{M}^{-1}.\mathcal{R}.\mathcal{T}.P$$

Si l'on reprend l'exemple du parallélépipède de la figure 3.11 qui est translaté le long de l'axe des y, pour lui faire subir une rotation dans le repère du monde, l'objet doit au total subir d'abord la translation, puis la rotation.

Pour effectuer une transformation $\mathcal{R}$ du point de vue dans le repère de la caméra (voir la figure 3.11c), il faut multiplier à gauche l'inverse $\mathcal{M}^{-1}$ de la matrice de passage du repère du monde au repère de la caméra par la matrice $\mathcal{R}$. (celà revient à multiplier la matrice $\mathcal{M}$ à droite par $\mathcal{R}^{-1}$). En coordonnées homogènes, on obtient :

$$P' = \mathcal{R}.\mathcal{M}^{-1}.\mathcal{T}.P$$

Pour reprendre l'exemple de la figure 3.11, la rotation a lieu après la projection sur le plan 2D.

3.8 Rotation, translation et changement d'échelle

3.8.1 Matrices GL_MODELVIEW et GL_PROJECTION

La succession de transformations que subit un objet avant d'être affiché correspond à une succession de produits de matrices. Les transformations $3D$ sont la composition de deux matrice 4×4 (voir la figure 3.12).

1. La matrice GL_MODELVIEW correspond à la transformation du modèle suivie de la transformation de la caméra ;
2. La matrice GL_PROJECTION correspond à la projection (transformation des coordonnées 3D en coordonées 2D.

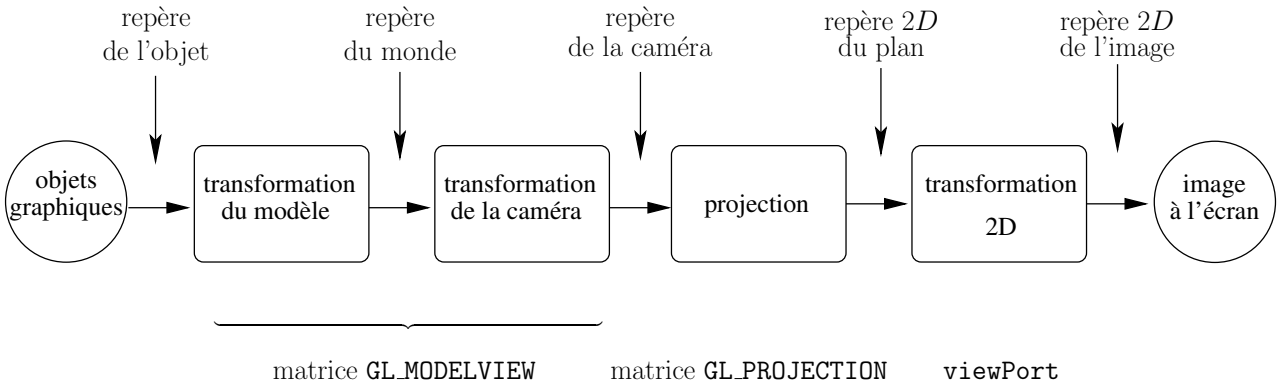


FIGURE 3.12: Les matrices GL_MODELVIEW et GL_PROJECTION

Toutes les transfomations $3D$ effectuées par *OpenGL* (rotations, translations, changements d'échelle, modification de la caméra, de l'angle d'ouverture, etc...) correspondent à la multiplication à droite de l'une des deux matrices GL_MODELVIEW et GL_PROJECTION par une matrice.

Pour modifier l'une des matrices GL_MODELVIEW ou GL_PROJECTION, il faut sélectionner le mode correspondant par un appel à glMatrixMode.

3.8.2 Les fonctions glRotatef, glTranslatef et glScalef

3.8.2.a La fonction glRotatef

La fonction glRotatef permet de faire des rotations, c'est à dire de multiplier l'une des matrices GL_MODELVIEW ou GL_PROJECTION par une matrice de rotation. Lorsqu'on veut faire tourner un objet dans le monde, on fait appel à glRotatef dans le mode GL_MODELVIEW. On conçoit que faire tourner le monde sur lui-même ou faire tourner la caméra autour du monde revient au même du point de vue de l'image obtenue. Ces deux choses se réalisent aussi dans le mode GL_MODELVIEW.

L'appel de glRotatef provoque une multiplication **à droite** par une matrice de rotation. En d'autres termes, la rotation en question sera effectuée **avant** toutes les autres transformations déjà accumulées dans le mode (GL_MODELVIEW ou GL_PROJECTION selon le cas).

Le prototype de la fonction glRotatef est le suivant :

```
\void glRotatef(GLfloat angle, GLfloat vx, GLfloat vy,  GLfloat vz);
```

Les coordonnées `vx`, `vy` et `vz` sont celles d'un vecteur directeur de l'axe de rotation. L'axe de rotation passe toujour par l'origine (rotation vectorielle). Pour faire une rotation dautour d'un axe ne passant pas par l'origine, il faut composer la rotation avec une translation.

3.8.2.b La fonction `glTranslatef`

La fonction `glTranslatef` permet de faire des translation, c'est à dire de multiplier l'une des matrices `GL_MODELVIEW` ou `GL_PROJECTION` par une matrice de translation en coordonnées homogènes. Lorsqu'on veut translater un objet dans le monde, ou encore le monde par rapport à la caméra, on fait appel à `glTranslatef` dans le mode `GL_MODELVIEW`.

Comme pour les rotations, l'appel de `glTranslatef` provoque une multiplication **à droite** par une matrice de translation. En d'autres termes, la translation en question sera effectuée **avant** toutes les autres transformations déjà accumulées dans le mode (`GL_MODELVIEW` ou `GL_PROJECTION` selon le cas).

Le prototype de la fonction `glTranslatef` est le suivant :

```
\void glTranslatef(GLfloat vx, GLfloat vy,  GLfloat vz);
```

Les coordonnées `vx`, `vy` et `vz` sont celles du vecteur de translation.

3.8.2.c La fonction `glScalef`

La fonction `glScalef` permet de faire des changlements d"échelle sur les axes (affinités orthogonales).

Le prototype de la fonction `glScalef` est le suivant :

```
\void glScalef(GLfloat facteurx, GLfloat facteury,  GLfloat facteurz);
```

Les coordonnées x des objets dessinés après l'appel à `glScalef` sont multipliées par `facteurx`, les coordonnées `y` par `facteury`, et les coordonnées `z` par `facteurz`.

la matrice de changement d'échelle est donc ma suivantes :

$$\begin{bmatrix} \texttt{facteurx} & 0 & 0 & 0 \\ 0 & \texttt{facteury} & 0 & 0 \\ 0 & 0 & \texttt{facteurz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Comme pour les rotations et translations, l'appel de `glScalef` provoque une multiplication **à droite** par une matrice de changement d'échelle. En d'autres termes, le changement d'échelle en question sera effectuée **avant** toutes les autres transformations déjà accumulées dans le mode (`GL_MODELVIEW` ou `GL_PROJECTION` selon le cas).

3.9 Pile de matrices

Avec ce que nous avons vu jusqu'à présent chaque appel à `glRotatef` affectera tous les objets dessinés par la suite. Les fonction `glPushMatrix` et `glPopMatrix` permettent d'appliquer une transformation à une partie des objets.

Les matrices `GL_MODELVIEW` et `GL_PROJECTION` sont en fait des piles de matrices. La matrice courante est la matrice qui se trouve au sommet de la pile.

Lors de l'appel à `glPushMatrix`, la matrice de transformation courante (`GL_MODELVIEW` ou `GL_PROJECTION` selon le mode sélectionné) est dupliquée, la copie est ajoutée au sommet de la pile de matrices, et devient la matrice courante. Lors de l'appel à `glPopMatrix`, elle est dépilée, et la matrice courante est restaurée à l'état d'avant l'appel à `glPopMatrix`. Toutes les transformations effectuées entre les appels de `glPushMatrix` et `glPopMatrix` n'auront aucun effet sur les objets dessinés après le `glPopMatrix`.

La structure de pile est particulièrement adaptée pour structurer les affichage en décomposant les objets en différentes parties. Par exemple, si l'on souhaite dessiner une voiture (voir la figure 3.13). avec deux essieux et deux roues par essieu, on va dessiner le corps de la voiture,

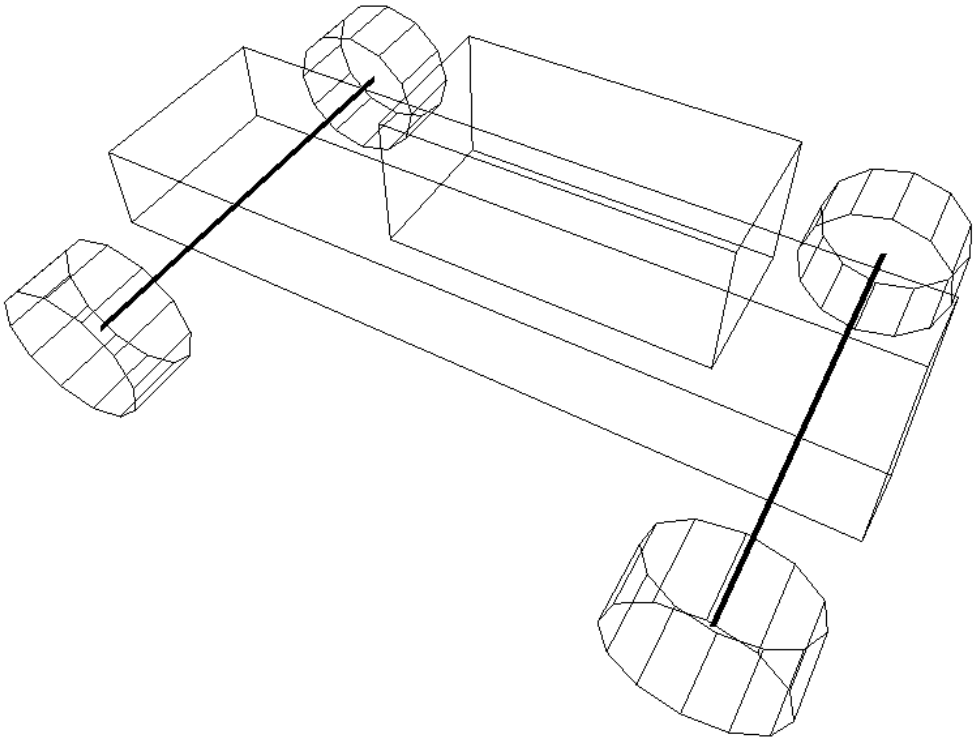


FIGURE 3.13: Dessin d'une voiture

on va translater pour dessiner l'essieu et les roues avant, puis on va revenir à la posistion de départ, puis translater pour dessiner l'essieu et les roues arrière, puis revenir à la position de départ. De même , pour dessiner l'essieu (par exemple avant), on va dessiner l'essieu, translater pour dessiner la roue de gauche, revenir à la position antérieure, translater pour dessiner la roue de droite, et revenir à la position antérieure.

Avec la structure de pile, on dessine le cors de la voiture, on duplique et on empile la matrice `GL_MODELVIEW`, on dessine l'essieu et les roues, puis on dépile la matrice `GL_MODELVIEW`, ce qui restore la matrice à l'état où elle était lors du dessin du corps de la voiture.

Chapitre 4

Éclairage

4.1 Élimination des parties cachées

Lorsqu'on affiche des objets en 3D, certaines parties de la scène sont cachées par d'autres qui se trouvent plus proches de l'observateur. Ainsi, tel un peintre, l'algorithme d'affichage doit dessiner les objets plus proches en dernier pour que seuls ceux-ci soient visibles. On appelle ce procédé l'*élimination des parties cachées*.

L'algorithme du $z-$buffer utilisé par *OpenGL* permet d'afficher des scènes dont les objets sont des maillages. Si une scène comprend d'autres types d'objets (sphères, quadriques, splines...), il faut facettiser ces surfaces (c'est à dire les approximer par des maillages pour pouvoir les afficher.

Dans l'algorithme du $z-$buffer, on calcule une image discrète mémorisée dans un "*frame-buffer*".

L'idée de l'algorithme est de calculer, pour chaque pixel, le point de profondeur minimale pour tous les polygones qui se projettent sur ce pixel, ou plus exactement la couleur du polygone en ce point. Pour celà, pour chaque facette du maillage, on parcourt tous les pixels de la projection de la facette (voir la figure 4.1), et on fait un test sur la profondeur z (dans le repère de la caméra) pour savoir si un objet plus proche a déjà été affiché sur ce pixel.

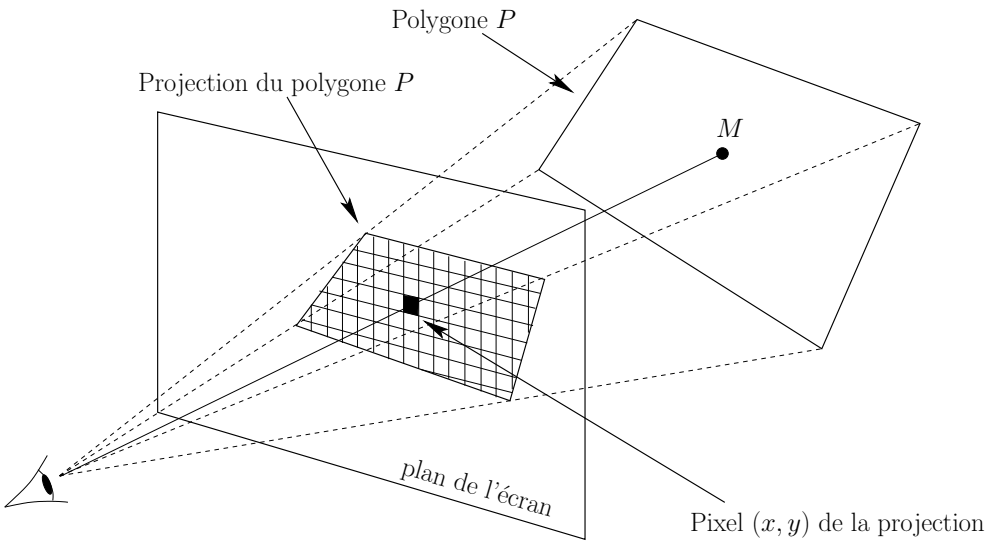


FIGURE 4.1: Principe de l'algorithme du $z-$buffer

Pour activer l'élimination des parties cachées dans une application *OpenGL* sous *Glut*, il faut activer l'option `GLUT_DEPTH` lors de l'appel de `glutInitDisplayMode` :

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

et activer le test de profondeur :

```
glEnable(GL_DEPTH_TEST);
```

4.2 Sources de lumière et matériaux

Seulement si aucun objet plus proche n'a été affiché, on stocke la couleur de la facette pour ce pixel dans le frame buffer. Pour réaliser les tests sur la profondeur, on doit stocker les profondeurs de tous les pixels dans un $z-$buffer.

L'affichage (plus ou moins) réaliste sous *OpenGL* repose sur une simulation simplifiée des éclairages. Pour l'utiliser, il faut d'abord activer l'éclairage par

```
glEnable(GL_LIGHTING);
```

4.2.1 Lumière ambiante

La lumière ambiante vient avec une même intensité de toutes les directions, et elle est réfléchie par les objets dans toutes les directions avec une même intensité. Un modèle d'illumination est traduit par une équation : *l'équation d'éclairement*. L'équation d'éclairement pour la lumière ambiante sur un objet est de la forme :

$$I = I_a k_a$$

où I_a est l'intensité de la lumière ambiante, caractéristique de la scène, et k_a est un coefficient entre 0 et 1, caractéristique de l'objet considéré.

Pour régler la lumière ambiante dans la scène, on peut utiliser la fonction `glLightModelfv` :

```
GLfloat ambient_scene[] = {\bf\{}0.2, 0.2, 0.2, 1.0{\bf\}};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_scene);
```

Les trois premiers coefficients du vecteur sont les intensités en RGB de la lumière ambiante. Le quatrième coefficient, appelé alpha, est pour le moment ignoré. On s'en sert pour faire du apha-blending, par exemple pour faire des effets de transparence.

Pour régler les coefficients de réflexion de la lumière ambiante du matériau courant, on utilise `glMaterialfv` :

```
GLfloat ambient[] = {\bf\{}0.2, 0.2, 0.2, 1.0{\bf\}};
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIANT, ambient);
```

Le premier paramètre de `glMaterialfv` peut être soit `GL_FRONT`, soit `GL_BACK`, soit `GL_FRONT_AND_BACK`. Cela permet de mettre un comportement différent pour les faces selon leur orientation. Voir aussi le culling dans la section 2.3. Les trois premiers coefficients du vecteurs sont les coefficients de réflexion de la lumière ambiante dans le rouge, vert et bleu.

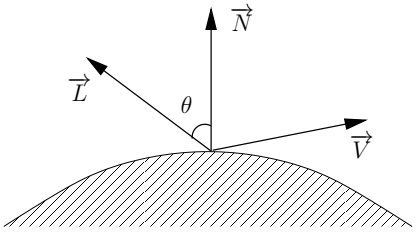


FIGURE 4.2: Le vecteur normal et le vecteur dirigé vers la source.

4.2.2 Réflexion diffuse

Dans le cas d'une source lumineuse ponctuelle non directionnelle, la réflexion diffuse tient compte, en un point d'un objet, de la direction du vecteur $\overrightarrow{N}$ normal sortant à l'objet en ce point, et de la direction du vecteur $\overrightarrow{L}$ dirigé vers la source lumineuse (voir figure 4.2).

$$I = I_S k_{rd} \cos\theta \text{ si } \cos\theta > 0$$

où I_S est l'intensité de la source lumineuse. Le *coefficient de réflexion diffuse* k_{rd} est une caractéristique de l'objet et varie entre 0 et 1.

Pour définir l'intensité d'une source lumineuse ponctuelle (pour le diffus), on utilise `glLightfv` :

```
GLfloat diffuse_light0[] = {\bf\{}1.0,1.0,1.0,1.0{\bf\}}
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_light0);
```

Les trois premières coordonnées du vecteur sont les intensités dans le rouge, vert et bleu de la source.

Il existe au moint 8 sources lumineuses (plus selon les implémentations d'*OpenGL*) nommées `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2`, etc...

Il faut activer chaque source individuellement par

```
glEnable(GL_LIGHT0);
```

Ne pas oublier d'activer l'éclairage par `glEnable(GL_LIGHTING)`.

Pour régler les coefficients de réflexion diffuse du matériau courant, on utilise `glMaterialfv` :

```
GLfloat diffuse[] = {\bf\{}0.2, 0.2, 0.2, 1.0{\bf\}};
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
```

Les trois premiers coefficients du vecteurs sont les coefficients de réflexion diffuse dans le rouge, vert et bleu.

4.2.3 Réflexion spéculaire

La réflexion spéculaire est un phénomène apparaissant sur les surfaces brillantes (voir la figure 4.4c). Considérons le vecteur $\overrightarrow{R}$, symétrique de $\overrightarrow{V}$ par rapport à $\overrightarrow{N}$ (voir figure 4.3). La direction $\overrightarrow{R}$ est la direction privilégiée de réflexion de la lumière vers l'observateur selon les lois de Descartes. Si la direction $\overrightarrow{L}$ de la source lumineuse est proche de la direction $\overrightarrow{R}$, une lumière vive apparaît à l'observateur en ce point de l'objet.

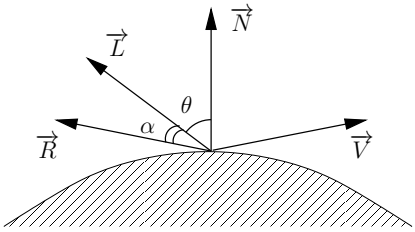
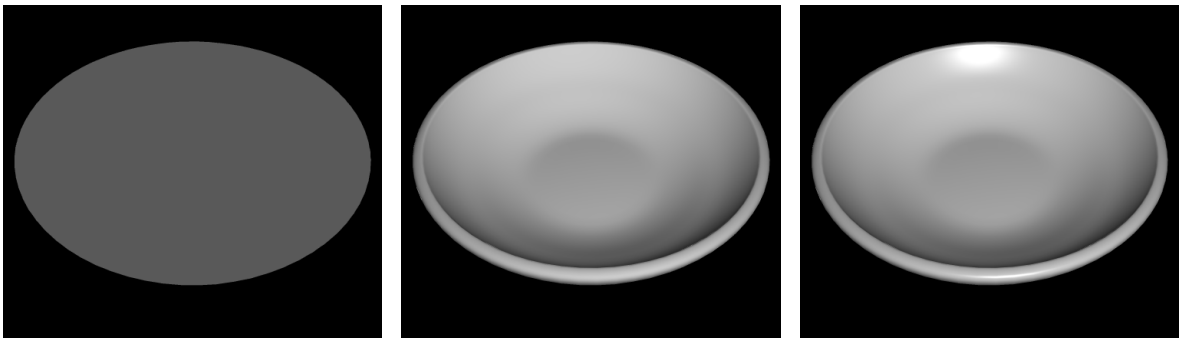


FIGURE 4.3: Le vecteur de la direction principale de réflexion et de la direction de l'observateur.



(a) Lumière ambiante (b) Réflexion diffuse (c) Réflexion spéculaire

FIGURE 4.4: Les différentes sortes de réflexion.

Le terme de l'équation d'éclairement correspondant à la réflexion spéculaire est couramment choisi proportionnel à $\cos^{n_s}\alpha$ (si $\alpha < \frac{\pi}{2}$), où α est l'angle entre $\overrightarrow{L}$ et $\overrightarrow{R}$ et $n_s \in \mathbb{N}^*$. On peut choisir une équation de la forme :

$$I_{s\lambda} = k_s \cos^{n_s}\alpha$$

où $\lambda \in \{R, G, B\}$ et k_s est le *coefficient de réflexion spéculaire*, caractéristique de l'objet, et variant entre 0 et 1. L'exposant n_s s'appelle l'*exposant spéculaire*, aussi appelé *brillance*. Plus l'exposant n_s est élevé, moins large est la tache claire liée à la réflexion spéculaire (voir la figure 4.5).

Pour définir l'intensité d'une source lumineuse ponctuelle pour le spéculaire, on utilise `glLightfv` :

```
GLfloat specular_light0[] = {\bf\{}1.0,1.0,1.0,1.0{\bf\}}
glLightfv(GL_LIGHT0, GL_SPECULAR, specular_light0);
```

Les trois premières coordonnées du vecteur sont les intensités dans le rouge, vert et bleu de la source.

Pour régler les coefficients de réflexion spéculaire du matériau courant, on utilise `glMaterialfv` :

```
GLfloat specular[] = {\bf\{}0.2, 0.2, 0.2, 1.0{\bf\}};
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);
glMaterialf((GL_FRONT_AND_BACK, GL_SHININESS, 110.0);
```

Les trois premiers coefficients du vecteurs sont les coefficients de réflexion specular dans le rouge, vert et bleu. La brillance (*shininess*) est l'exposant spéculaire. Dnas *OpenGL*, il est compris entre 0.0 et 128.0

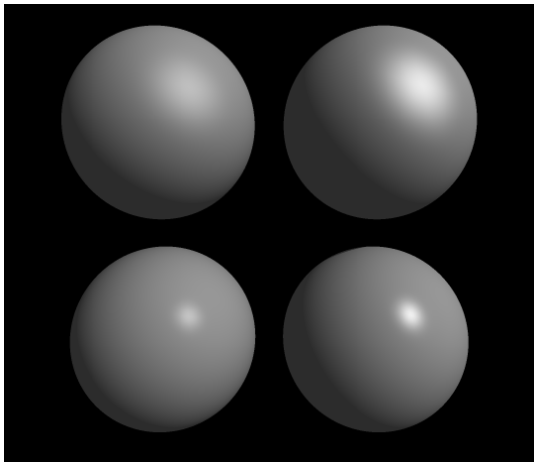


FIGURE 4.5: L'effet des coefficients spéculaires et brillance. Les coefficients sont : en haut à gauche $k_s = 0.2$, $n_s = 5$; en haut à droite $k_s = 0.4$, $n_s = 5$; en bas à gauche $k_s = 0.2$, $n_s = 30$; en bas à droite $k_s = 0.4$, $n_s = 30$.

4.2.4 Atténuation de la lumière avec la distance

L'effet de la lumière s'atténue avec la distance de l'objet considéré à la source lumineuse, et on modifie fréquemment le terme de l'équation d'éclairement lié aux réflexions diffuses et spéculaires en multipliant par une fonction f_{att}. On peut par exemple prendre une fonction f_{att} de la forme :

$$f_{att} = \min(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1)$$

où d_L est la distance du point de l'objet considéré à la source lumineuse, et c_1, c_2 et c_3 sont choisis par l'utilisateur, appelés respectivement atténuation constante, atténuation linéaire, et atténuation quadratique.

Pour définir les coefficients d'atténuation de la lumière avec la distance à la source lumineuse, on utilise la fonction `glLightf` :

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Les valeurs doivent être adaptées suivant les dimensions de la scène à éclairer.

4.2.5 Spots

Jusqu'à présent, nous n'avons étudié que des sources non directionnelles, c'est à dire que la lumière issue d'une telle source part de la même manière dans toutes les directions. En réalité, les sources de lumière, tels des spots, émettent souvent de la lumière dans une direction privilégiée. Cela correspond au modèle de source lumineuse de type spot.

4.3 Vecteurs normaux

Pour calculer les termes de réflexion diffuse et spéculaire, `OpenGL` a besoin de connaître les vecteurs normaux (vecteurs orthogonaux à la surface) aux sommets du maillage. Ces vecteurs

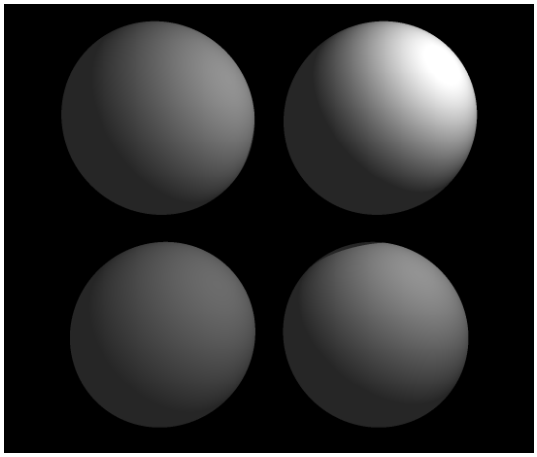


FIGURE 4.6: Effet de l'atténuation de la lumière avec la distance sur des sphères de même coefficient de réflexion diffuse. Les sphères sont à une distance à la source lumineuse de 302 (en haut à gauche et en bas à droite), 177 (en haut à droite), et 389 (en bas à gauche). Les coefficients d'atténuation sont de $c_1 = 0.1$, $c_2 = 5.10^{-5}$, $c_3 = 7.10^{-5}$.

normaux peuvent être calculés et mémorisés lors de la construction du maillage à partir d'un modèle analytique de surface, ou bien il peut être estimé sur le maillage. Pour estimer la normale sur un maillage, on fait pour chaque sommet une moyenne des normales incidentes au sommet. Une fois les normales connues, on peut les communiquer à `OpenGL` en utilisant la fonction `glNormal3f` avant un appel à `glVertex3f` lors du dessin d'une primitive géomérique. Par exemple, on peut indiquer les normales aux sommet d'un triangle comme suit :

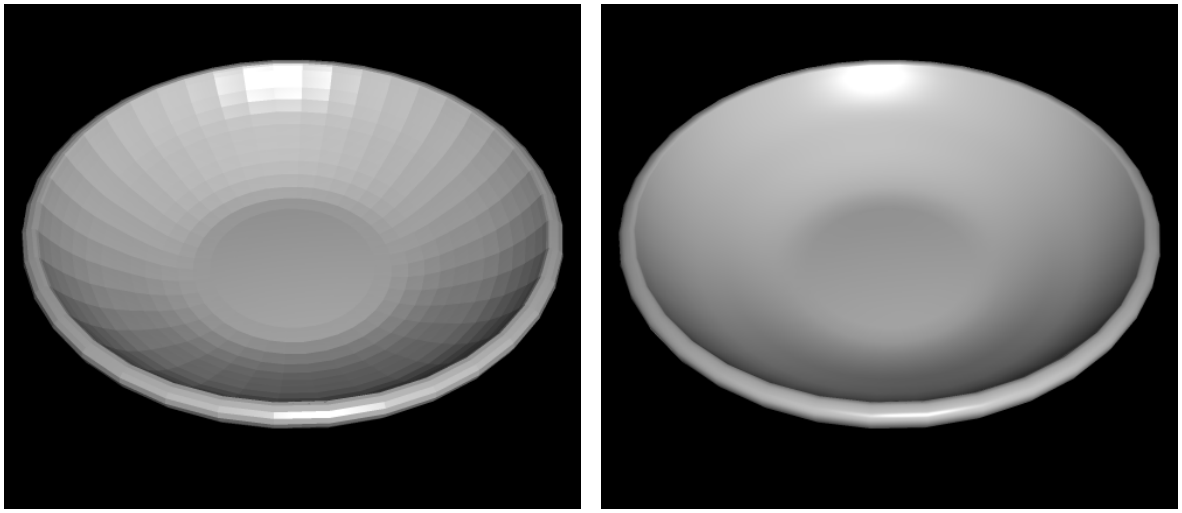
On doit tout d'abord appeler `glEnable(GL_NORMALIZE)` qui assure que les vecteurs normaux sont normalisés (de longuer 1), en calculant la norme du vecteur normal et en divisant ses coordonnées par la norme. C'est d'autant plus utile si l'on effectue des transformation de type changement d'échelle (`glScalef`). Ensuite, on appelle `glNormal3f` avant chaque définition de sommet :

```
glBegin(GL_TRIANGLES);
   glNormal3f(1.0,2.0,3.0); \com{Ce vecteur doit être normalisé}
   glVertex3f(0.0, 1.0, 0.0);
   glNormal3f(2.0,1.0, 3.0);
   glVertex3f(1.0, 0.0, 0.0);
   glNormal3f(1.0,1.0, 2.0);
   glVertex3f(0.0, 0.0, 0.5);
glEnd();
```

4.4 Éclairement plat et lissage de Gouraud

L'algorithme d'élimination des partie cachées utilisé par `OpenGL`, appelé $z-$buffer, utilise une approximation des tous les objets par des maillages. Si l'on visualise directement un tel maillage, des facettes sont visibles (voir la figure 4.7). Prendre des facettes diminue les performances et n'est pas vraiment envisageable. Aussi préfère-t-on en général faire des interpolations pour "lisser l'intensité lumineuse sur la surface". Pour celà, `OpenGL` propose un affichage par le modèle de Gouraud, dans lequel ine interpolation bilinéaire est effectuée à partir des intensités

des sommet .



(a) Éclairement plat `GL_FLAT` (b) Lissage de Gourand `GL_SMOOTH`

FIGURE 4.7: La fonction `glShadeModel` permet de sélectioner l'éclairement plat ou le lissage de Gouraud

Pour sélectionner l'éclairement plat :

```
glShadeModel(GL_FLAT);
```

Pour sélectionner l'éclairement lissé de Gouraud :

```
glShadeModel(GL_SMOOTH);
```

4.5 Gérer la position et direction des sources lumineuses

Pour définir la position d'une source lumineurs `GL_LIGHT0`, `GL_LIGHT1`, etc. on utilise la fonction `glLightfv`. Par exemple, pour placer la source `GL_LIGHT0` en position $(5, 10, 7)$

```
/* Exemple de source placée en (5,10,7) */
GLfloat lightPosition0[] = {\bf\{}5.0,10.0,7.0,1.0{\bf\}};
...
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
```

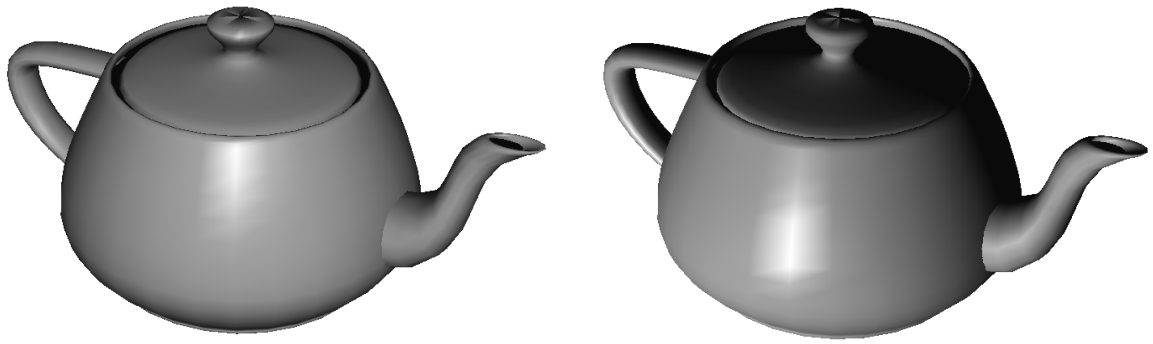
Les coordonnées de la position de la source sont des coordonnées homogènes. On peut définir une source placée à l'infini en définissant la quatriemme coordonnée w égale à 0.

```
/* Exemple de source placée à l'infini le long de l'axe des x */
GLfloat lightPosition0[] = {\bf\{}1.0,0.0,0.0,0.0{\bf\}};
...
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
```

Les sources placées à l'infini (comme le soleil) projettent des rayons lumineux parallèles dans la scène.

La position de la source est multipliée par la matrice `GL_MODELVIEW` courante. (la matrice `GL_PROJECTION` n'a pas d'effet sur la position d'une source lumineuse). On peut donc, suivant le point dans le code où l'on définit la postion de la source, positioner la source ou bien dans le repère du monde ou bien dans le repère de la caméra.

Exemple 1. Source fixe dans le repère de la caméra.
Pour faire une source lumineuse fixe dans le repère de la caméra, il faut définir la position de la source dans le code juste après l'initialisation de la matrice `GL_MODELVIEW`. Dans l'exemple suivant, la source est positionnée précisément à l'infini le long de l'axe de visée, derière l'observateur (voir la figure 4.8).



(a) Exemple 1 (b) Exemple 2

FIGURE 4.8: Dans l'exemple 1, la source est placée sur l'axe de visée du repère de la caméra. Dans l'exemple 2, la source subit la transformation `gluLookAt` et se trouve dans le repère de la théière sur l'axe des z

shading//progc/exshading1.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  GLushort largeur_fenetre=500;
6  GLushort hauteur_fenetre=500;
7
8  void Redimensionnement(int l,int h)
9  {
10   GLfloat lightPosition0[] = {0.0,0.0,10.0,0.0};
11   GLfloat diffuseLight0[] = {0.6,0.6,0.6,1.0};
12   GLfloat specularLight0[] = {0.6,0.6,0.6,1.0};
13   largeur_fenetre = l;
14   hauteur_fenetre = h;
15   glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
16   glMatrixMode(GL_PROJECTION);
17   glLoadIdentity();
18   gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
19   glMatrixMode(GL_MODELVIEW);
20   glLoadIdentity();
```

```
21    glLightfv(GL_LIGHT0,GL_DIFFUSE, diffuseLight0);
22    glLightfv(GL_LIGHT0,GL_SPECULAR, specularLight0);
23    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
24  }
25
26  void Affichage(void)
27  {
28    GLfloat diffuse[] = {1.0,1.0,1.0,1.0};
29    GLfloat specular[] = {1.0,1.0,1.0,1.0};
30    GLfloat ambient[] = {0.2,0.2,0.2, 1.0};
31    GLfloat shininess = 110.0;
32    glClearColor(1.0,1.0,1.0,1.0);
33    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
34    glMatrixMode(GL_MODELVIEW);
35    glPushMatrix();
36    gluLookAt(8, 10, 15,   /* rglage de la camra */
37              0,0,0,
38              0,1,0);
39    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
40    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,specular);
41    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
42    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
43    glutSolidTeapot(2);  /* dessin d'une  thire */
44    glPopMatrix();
45    glutSwapBuffers();  /* Envoyer le buffer  l'cran */
46  }
47
48  int main(int argc, char**argv)
49  {
50    glutInit(&argc, argv);
51    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
52    glutInitWindowPosition(100,100);
53    glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
54    glutCreateWindow("Source place  l'infini derire l'observateur");
55    glEnable(GL_DEPTH_TEST);
56    glEnable(GL_LIGHTING);
57    glEnable(GL_LIGHT0);
58    glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
59    glShadeModel(GL_SMOOTH);
60    glutDisplayFunc(Affichage);
61    glutReshapeFunc(Redimensionnement);
62    glutMainLoop();
63    return 0;
64  }
```

Exemple 2. Source placée dans le repère du monde

shading//progc/exshading2.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  GLushort largeur_fenetre=500;
6  GLushort hauteur_fenetre=500;
7
8  void Redimensionnement(int l,int h)
```

```
9  {
10   largeur_fenetre = l;
11   hauteur_fenetre = h;
12   glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
13   glMatrixMode(GL_PROJECTION);
14   glLoadIdentity();
15   gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
16   glMatrixMode(GL_MODELVIEW);
17   glLoadIdentity();
18 }
19
20 void Affichage(void)
21 {
22   GLfloat lightPosition0[] = {0.0,0.0,10.0,0.0};
23   GLfloat diffuseLight0[] = {0.6,0.6,0.6,1.0};
24   GLfloat specularLight0[] = {0.6,0.6,0.6,1.0};
25   GLfloat diffuse[] = {1.0,1.0,1.0,1.0};
26   GLfloat specular[] = {1.0,1.0,1.0,1.0};
27   GLfloat ambient[] = {0.2,0.2,0.2, 1.0};
28   GLfloat shininess = 110.0;
29   glClearColor(1.0,1.0,1.0,1.0);
30   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
31   glMatrixMode(GL_MODELVIEW);
32   glPushMatrix();
33   gluLookAt(8, 10, 15,   /* rglage de la camra */
34             0,0,0,
35             0,1,0);
36   glLightfv(GL_LIGHT0,GL_DIFFUSE, diffuseLight0);
37   glLightfv(GL_LIGHT0,GL_SPECULAR, specularLight0);
38   glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
39   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
40   glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,specular);
41   glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
42   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
43   glutSolidTeapot(2);  /* dessin d'une  thire */
44   glPopMatrix();
45   glutSwapBuffers();  /* Envoyer le buffer  l'cran */
46 }
47
48 int main(int argc, char**argv)
49 {
50   glutInit(&argc, argv);
51   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
52   glutInitWindowPosition(100,100);
53   glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
54   glutCreateWindow("Source place l'infini derire l'observateur");
55   glEnable(GL_DEPTH_TEST);
56   glEnable(GL_LIGHTING);
57   glEnable(GL_LIGHT0);
58   glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
59   glShadeModel(GL_SMOOTH);
60   glutDisplayFunc(Affichage);
61   glutReshapeFunc(Redimensionnement);
62   glutMainLoop();
63   return 0;
64 }
```

Chapitre 5

Aspects avancés du rendu

5.1 Optimisation de l'affichage : *Vertex Arrays*

5.1.1 Qu'est-ce qu'un *Vertex Array* ?

Lorsqu'on dessine un maillage, il peut y avoir un très grand nombre de fonctions, avec pour chaque sommet un appel de `glVertex3f`, un appel de *glNormal3f*, éventuellement *glColor3f*, etc. (voir l'exemple 1 version 1 d'affichage de tétraèdre ci-dessous). En outre, chaque sommet peut être traités plusiers fois, indépendament pour chaque face incidente.

Les *Vertex Arrays* permettent de définir tous les sommets et leur données telles que les normales, couleur, etc. en utilisant quelques tableaux. L'affichage ne nécessite alors plus de lister les sommets. Les tableaux concernant les sommets sont envoyés une fois en mémoire graphique et utilisés directement par *OpenGL* pour l'affichage. Les facettes sont décrites par des indices de sommets dans les tableaux.

5.1.2 Exemple : affichage d'un tétraèdre

Exemple 1. Version 1 Affichage non optimisé d'un tétraèdre. Le code suivant affiche un tétraèdre dont les sommets ont différentes couleurs (voir la figure 5.1). (Les couleurs sont interpolées à l'intérieur des facettes du tétraèdre).

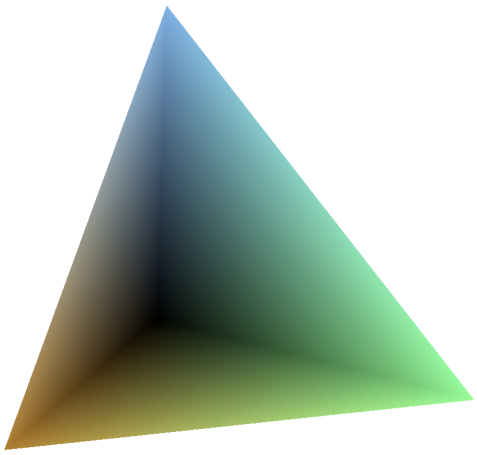


FIGURE 5.1: Affichage d'un tétraèdre avec des sommets de différentes couleurs.

vertexArray//progc/exVertexArray1.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  GLushort largeur_fenetre=500;
6  GLushort hauteur_fenetre=500;
7
8  GLint nvertices=4, nfaces=4;
9
10 GLfloat vertices[] = {
11   0.0, 0.0, 0.0,
12   1.0, 0.0, 0.0,
13   0.0, 1.0, 0.0,
14   0.0, 0.0, 1.0
15 };
16
17 GLfloat colors[] = {
18   0.0, 0.0, 0.0,
19   0.7, 0.5, 0.2,
20   0.5, 0.7, 0.9,
21   0.6, 1.0, 0.6
22 };
23
24 GLuint faces[] = {
25   0, 1, 2,
26   0, 1, 3,
27   0, 2, 3,
28   1, 2, 3
29 };
30
31 void Redimensionnement(int l,int h)
32 {
33   largeur_fenetre = l;
34   hauteur_fenetre = h;
35   glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
36   glMatrixMode(GL_PROJECTION);
37   glLoadIdentity();
38   gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
39   glMatrixMode(GL_MODELVIEW);
40   glLoadIdentity();
41 }
42
43 void DessinTetraedre()
44 {
45   int i, j;
46   glBegin(GL_TRIANGLES);
47   for (i=0 ; i<3*nfaces ; i++)
48     {
49       glColor3fv(&colors[3*faces[i]]);
50       glVertex3fv(&vertices[3*faces[i]]);
51     }
52   glEnd();
53 }
54
55 void Affichage(void)
```

```
56 {
57   glClearColor(1.0,1.0,1.0,1.0);
58   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
59   glMatrixMode(GL_MODELVIEW);
60   glPushMatrix();
61   glTranslatef(-0.2,-0.2,-4);
62   glRotatef(-30, 1.0, 0.0, 0.0);
63   glRotatef(120, 0.0, 1.0, 0.0);
64   DessinTetraedre();
65   glPopMatrix();
66   glutSwapBuffers();  /* Envoyer le buffer  l'cran */
67 }
68
69 int main(int argc, char**argv)
70 {
71   glutInit(&argc, argv);
72   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
73   glutInitWindowPosition(100,100);
74   glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
75   glutCreateWindow("Dessin d'un ttradre");
76   glEnable(GL_DEPTH_TEST);
77   glutDisplayFunc(Affichage);
78   glutReshapeFunc(Redimensionnement);
79   glutMainLoop();
80   return 0;
81 }
```

Exemple 1. Version 2 Affichage d'un tétraèdre avec *Vertex Arrays*. L'image obtenue par cette verion est la même que pour la version 1. Le code de l'affichage du maillage ne comprend qu'un seul appel de fonction.

vertexArray//progc/exVertexArray2.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  GLushort largeur_fenetre=500;
6  GLushort hauteur_fenetre=500;
7
8  GLint nvertices=4, nfaces=4;
9
10 GLfloat vertices[] = {
11   0.0, 0.0, 0.0,
12   1.0, 0.0, 0.0,
13   0.0, 1.0, 0.0,
14   0.0, 0.0, 1.0
15 };
16
17 GLfloat colors[] = {
18   0.0, 0.0, 0.0,
19   0.7, 0.5, 0.2,
20   0.5, 0.7, 0.9,
21   0.6, 1.0, 0.6
22 };
23
24 GLuint faces[] = {   /* {\bf mettre des UNSIGNED INT} */
```

```
25    0, 1, 2,
26    0, 1, 3,
27    0, 2, 3,
28    1, 2, 3
29  };
30
31  void Redimensionnement(int l,int h)
32  {
33    largeur_fenetre = l;
34    hauteur_fenetre = h;
35    glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
36    glMatrixMode(GL_PROJECTION);
37    glLoadIdentity();
38    gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
39    glMatrixMode(GL_MODELVIEW);
40    glLoadIdentity();
41  }
42
43  void DessinTetraedre()
44  {
45    glDrawElements(GL_TRIANGLES, 3*nfaces, GL_UNSIGNED_INT, faces);
46  }
47
48  void Affichage(void)
49  {
50    glClearColor(1.0,1.0,1.0,1.0);
51    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
52    glMatrixMode(GL_MODELVIEW);
53    glPushMatrix();
54    glTranslatef(-0.2,-0.2,-4);
55    glRotatef(-30, 1.0, 0.0, 0.0);
56    glRotatef(120, 0.0, 1.0, 0.0);
57    DessinTetraedre();
58    glPopMatrix();
59    glutSwapBuffers();  /* Envoyer le buffer  l 'cran */
60  }
61
62  int main(int argc, char**argv)
63  {
64    glutInit(&argc, argv);
65    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
66    glutInitWindowPosition(100,100);
67    glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
68    glutCreateWindow("Dessin d'un  ttradre : version rapide");
69    glEnable(GL_DEPTH_TEST);
70    glutDisplayFunc(Affichage);
71    glutReshapeFunc(Redimensionnement);
72
73    /* Activation des Vertex Array et Color Array */
74    glEnableClientState(GL_COLOR_ARRAY);
75    glEnableClientState(GL_VERTEX_ARRAY);
76
77    /* Chargement du Color Array */
78    glColorPointer(3, GL_FLOAT, 0, colors);
79    /* Chargement du Vertex Array */
80    glVertexPointer(3, GL_FLOAT, 0, vertices);
```

```
81
82    glutMainLoop();
83    return 0;
84  }
```

5.1.3 Principe général

On peut aussi mettre dans un *Normal Array* les normales aux sommets, les coordonnées de texture, etc.

Pour activer les *Arrays*, on utilise la fonction

```
void glEnableClientState(GLenum array);
```

qui peut prendre comme paramètre`GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_TEXTURE_COORD_ARRAY`...

On désigne l'adresse où se trouvent les sommets, normales, couleurs, etc. en utilisant les fonctions

```
\void glVertexPointer(GLint size, GLenum type, GLsizei stride,
                      const GLvoid *pointer);
\void glColorPointer(GLint size, GLenum type, GLsizei stride,
                     {\bf const} GLvoid *pointer);
\void glNormalPointer(GLenum type, GLsizei stride,
                      {\bf const} GLvoid *pointer);
\void glTexCoordPointer(GLint size, GLenum type, GLsizei stride,
                        {\bf const} GLvoid *pointer);
```

Les paramètres sont :

- `size`, qui vaut 2, 3 ou 4 et donne le combre de composantes pour chaque sommet. Une normale a toujours 3 composantes donc ce paramètre est absent de `glNormalPointer`.
- `type` peut valoir suivant le type de données `GL_FLOAT`, `GL_DOUBLE`, `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`.
- `stride` donne le nombre d'octets entre deux données consécutives (nombre de composantes multiplié par le nombre d'octets par composante), sachant qu'il peut y avoir des l'espace occupé par d'autre données entre deux sommets. Ceci permet par exemple de mettre les sommets et les normales dans un même tableau. Si le paramètre `stride` vaut 0, les données sont supposées contigües (sans espace entre deux données).
- `pointer` donne l'adresse d'un tableau contenant les données.

Pour l'affichage, on présente ici la fonction `glDrawElements` qui peut afficher des maillages dont les sommets sont toutes de même type (triangles, ou quads,...). D'autres fonctions existent, telles que `glDrawElement` qui permet de gérer à la main quels sommets sont affichés, et `glMultiDrawElements` qui permet d'afficher des maillages dont les faces peuvent avoir des nombres de sommets différents les une des autres.

La fonction `glDrawElements` a pour prototype :

```
\void glDrawElements(GLenum mode, GLsizei count, GLenum type,
                     \void *indices);
```

Les paramètres ont la signification suivante :

- `mode` est l'un des arguments valides de `glBegin` et décrit le type de primitive géométrique à affiche. Il peut être `GL_TRIANGLES`, `GL_QUADS`, `GL_POLYGON`, `GL_LINES`, `GL_POINTS`, etc.
- `count` est le nombre total d'éléments tracés (somme des nombre de sommets des faces).
- `type` décrit le type entier dans lequel sont donnés les indices des sommets dans les faces. Il peut être `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, ou `GL_UNSIGNED_INT`. On peut compresser le tableau des indices en utilisant `GL_UNSIGNED_BYTE` mais il faut alors s'assurer que les indices des sommets sont inférieurs à 256.
- `indices` est le tableau comprenant les indices. On met les indices de toutes les faces à la suite. Ceci est cohérent avec le comportement de l'affichage de primitives géométriques, qui permet par exemple de dessiner plusieurs triangle en passant tous les sommets entre le `glBegin(GL_TRIANGLES)` et le `glEnd()`.

Pour dessiner des maillages dont toutes les faces ne comportent pas le même nombre de sommet, on utilise :

```
void glMultiDrawElements(GLenum mode,
  const GLsizei * count,
  GLenum type,
  const GLvoid ** indices,
  GLsizei primcount);
```

Le paramètre `count` donne un tableau avec pour chaque face le nombre de sommets de la face, `primcount` donne le nombre de faces, et `indices` donne un tableau de tableaux d'indices de sommets.

5.2 Plaquage de textures

5.2.1 Principe du plaquage de texture

Le but du plaquage de texture est de représenter des objets qui n'ont pas une couleur uniforme (voir figures 5.2). De tels objets sont très courants dans le réel, leur représentation est donc essentielle en synthèse d'images.

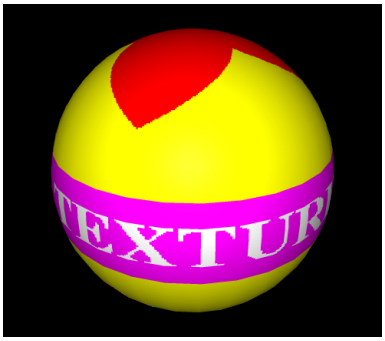
Le principe du plaquage de textures consiste à plaquer une image plane, appelée *image de texture*, sur la surface de l'objet. Par exemple, l'objet de la figure 5.2b a été obtenu en plaquant l'image de la figure 5.2a sur une sphère. Pour cela, étant donné un point M de la surface, il faut faire correspondre au point M un point $T(M)$ de l'image de texture. La couleur de l'objet au point M à prendre en compte pour l'affichage est alors la couleur de l'image de texture au point $T(M)$. Les coordonnées du point $T(M)$ dans le plan s'appellent les *coordonnées de texture* du point M.

En général, on associe au point M des coordonnées $s(M)$ et $t(M)$ dans $[0, 1]$.

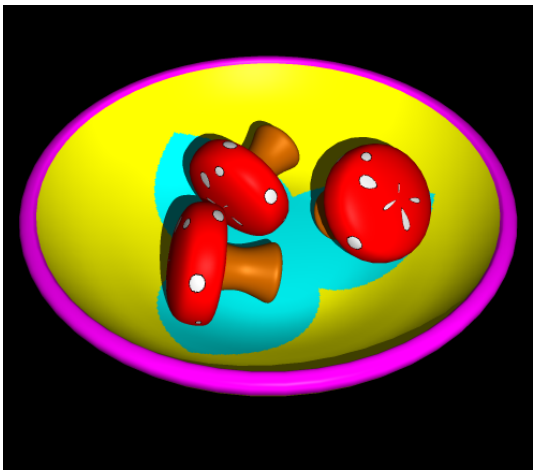
Par exemple, dans le cas d'une sphère, on fait correspondre à un point M un point d'une image suivant la latitude et longiture. (voir figure 5.3). Notons que les changements d'échelle sur les axes permettent de ramener l'intervalle $[0, 2\pi]$ où varie l'angle θ sur l'intervalle $[0, largeur - 1]$. Il en va de même pour l'intervalle où varie l'angle φ avec la hauteur.

Plus généralement, soit $\sigma : [0, 1] \times [0, 1] \longrightarrow \mathbb{R}^3$ une surface paramétrée, et soit une image de texture de dimensions $largeur \times hauteur$. On peut faire correspondre au point $M = \sigma(s, t)$ de la surface les coordonnées $s(M) = s$ et $t(M) = t$, on obtient donc le point $T(M) = (s.largeur, t.hauteur)$ dans l'image de texture.

(a) Image de texture

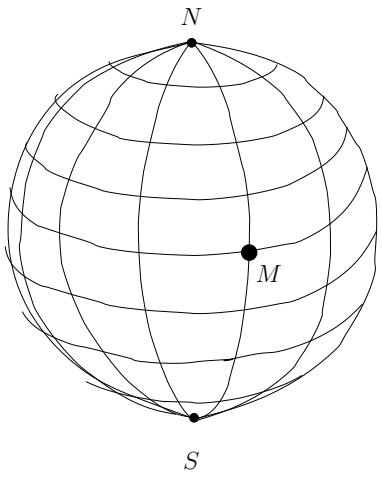


(b) Rendu de sphère texturée

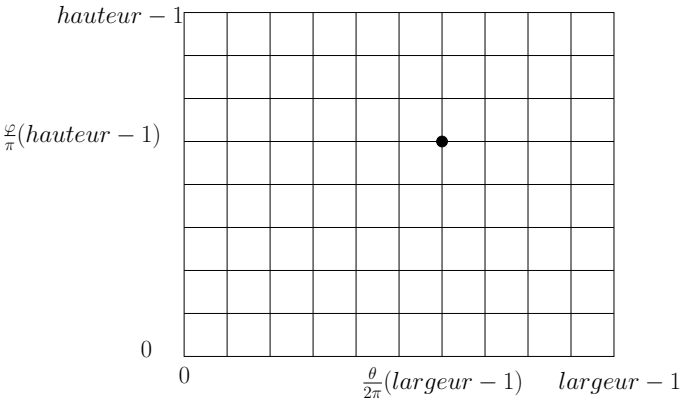


(c) Objets texturés

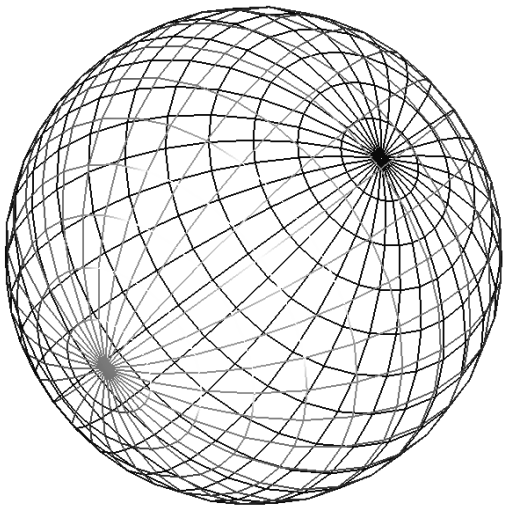
FIGURE 5.2: Exemple de plaquage de texture



(a) Le point M sur la sphère



(b) Le point correspondant dans l'image de texture

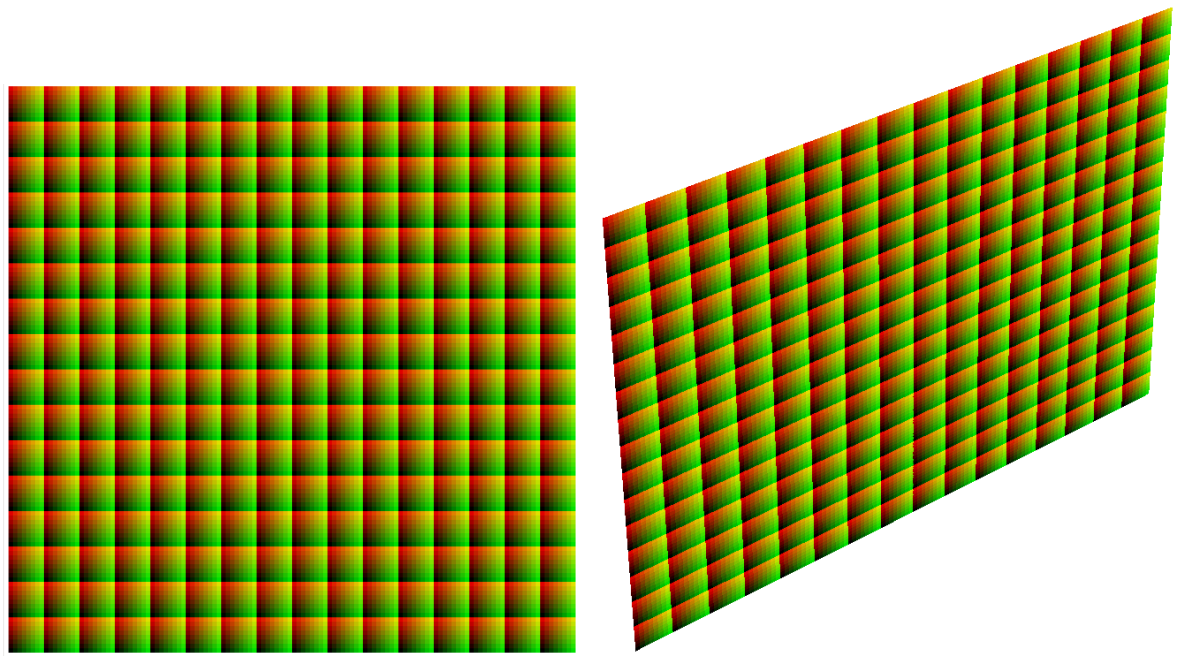


(c) Latitude et longitude sur la sphère

FIGURE 5.3: Correspondance Surface - Image de texture.

5.3 Plaquage de textures avec *OpenGL*

Exemple 1. Génération de texture par un algorithme puis plaquage de la texture. La fonction `CreeBitmap` génère les couleurs des *texels* (les texels sont l'équivalent des pixels dans une image de texture). Les données de textels sont stockés dans un grand tableau qui contient les lignes les unes à la suite des autres. La couleur de chaque texel est sockée sur 4 octets par ses composantes *RGBA*.



(a) Image de texture générée (b) Quadrilatère texturé vu en perspective

textures//progc/premierExemple.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define largeur_tex 128
6  #define hauteur_tex 128
7
8  GLushort largeur_fenetre=500;
9  GLushort hauteur_fenetre=500;
10
11 GLint angle=0;
12
13 GLuint id = 0;  /* texture ID */
14
15 /* bitmap contenant les couleurs RGBA de la texture */
16 GLubyte image_tex[largeur_tex*hauteur_tex*4];
17
18 void CreeBitmap()
19 {
20   int i, j;
21   GLubyte r, g, b;
22   for (i=0 ; i<hauteur_tex ; i++)
```

```
23      {
24        for (j=0 ; j<largeur_tex ; j++)
25    {
26      r = (i*32)%256;
27      g = (j*32)%256;
28      b = ((i*16)*(j*16))%256;
29      image_tex[(i*largeur_tex+j)*4]   = r;
30      image_tex[(i*largeur_tex+j)*4+1] = g;
31      image_tex[(i*largeur_tex+j)*4+2] = b;
32      image_tex[(i*largeur_tex+j)*4+3] = 255;
33    }
34      }
35  }
36
37  void CreeTexture2D(void)
38  {
39    struct gl_texture_t *png_tex = NULL;
40
41    CreeBitmap();
42    /* Generation texture */
43    glGenTextures(1, &id);
44    glBindTexture(GL_TEXTURE_2D, id);
45
46    /* Paramtres de filtres linaires */
47    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
48    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
49
50    /* Cration de la texture */
51    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, largeur_tex, hauteur_tex,
52            0, GL_RGBA, GL_UNSIGNED_BYTE, image_tex);
53  }
54
55  void Affichage(void)
56  {
57    glClearColor(1.0,1.0,1.0,1.0);
58    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
59
60    glMatrixMode(GL_MODELVIEW);
61    glPushMatrix();
62    gluLookAt(8, 10, 15,    /* rglage de la camra */
63              0,0,0,
64              0,1,0);
65    glRotatef(0.05*angle, 0.0, 1.0, 0.0);
66
67    glBindTexture (GL_TEXTURE_2D, id); /* slection de la texture */
68    glEnable (GL_TEXTURE_2D); /* activation des textures */
69
70    glBegin(GL_QUADS);
71    glTexCoord2f(0.0,0.0); glVertex3f(-3.0,-2.0,0.0);
72    glTexCoord2f(1.0,0.0); glVertex3f(3.0,-2.0,0.0);
73    glTexCoord2f(1.0,1.0); glVertex3f(3.0,2.0,0.0);
74    glTexCoord2f(0.0,1.0); glVertex3f(-3.0,2.0,0.0);
75    glEnd();
76
77    glDisable (GL_TEXTURE_2D); /* dsactivation des textures */
78
```

```
79    glPopMatrix();
80    glutSwapBuffers();  /* Envoyer le buffer  l'cran */
81  }
82
83  void Redimensionnement(int l,int h)
84  {
85    largeur_fenetre = l;
86    hauteur_fenetre = h;
87    glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
88    glMatrixMode(GL_PROJECTION);
89    glLoadIdentity();
90    gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
91    glMatrixMode(GL_MODELVIEW);
92    glLoadIdentity();
93  }
94
95  void IdleFonction(void)
96  {
97    angle++;
98    glutPostRedisplay();
99  }
100
101 int main(int argc, char**argv)
102 {
103   glutInit(&argc, argv);
104   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
105   glutInitWindowPosition(100,100);
106   glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
107   glutCreateWindow("Source place  l'infini derire l'observateur");
108   glEnable(GL_DEPTH_TEST);
109   glutDisplayFunc(Affichage);
110   glutReshapeFunc(Redimensionnement);
111   glutIdleFunc(IdleFonction);
112
113   CreeTexture2D();
114
115   glutMainLoop();
116   return 0;
117 }
```

Les principales fonctions utilisées ici sont :

```
void glGenTextures(GLsizei n, GLuint *textureNames);
```

Génère n identifiants de textures non utilisés et le met dans `textureNames`.

```
void glBindTexture(GLenum target, GLuint textureName);
```

Permet trois choses :

- D'activer (par un identifiant obtenu par `glGenTextures`) une texture pour l'utiliser, par exmple dans un affichage ;
- De créer une nouvelle texture si la texture n'a pas encore été utilisée ;
- De désactiver l'utilisation des textures si l'on passe 0 comme identifiant de texture.

```
void glDeleteTextures(GLsizei n, const GLuint *textureNames);
```

Libère la mémoire et les identifiants coorespondant aux textures dont les noms sont passés dans le tableau `textureNames`.

```
void glTexImage2D(GLenum target, GLint level, GLint internalFormat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type,
                  const GLvoid *pixels);
```

Définit une texture $2D$. On spécifie la largeur, hauteur, et les couleurs des pixels, et la texture précédement sélectionnée par `glBindTexture` est initialisée avec ces pixels. Le format interne indique lesquelles des composantes R,G,B,A, luminance ou intensités seront utilisées par *OpenGL* pour décrire les texels de l'image. Il y a trente 38 constantes possibles pour le format, qui peut aussi spécifier le nombre d'octets utilisés pour chaque composante. Les valeurs les plus simples sont `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE` ou encore `GL_ALPHA` pour le *alpha-blending*. Ces valeurs peuvent aussi être utilisées pour la variable `format` qui décrit le format du tableau `pixels`.

```
void glTexCoord2f(GLfloat s, GLfloat t);
```

Attribue les coordonnées de texture (s, t) au(x) prochain(s) sommet(s) créé avec `glVertex*()`.

```
void glTexParameteri(GLenum target, GLenum pname, TYPE param);
```

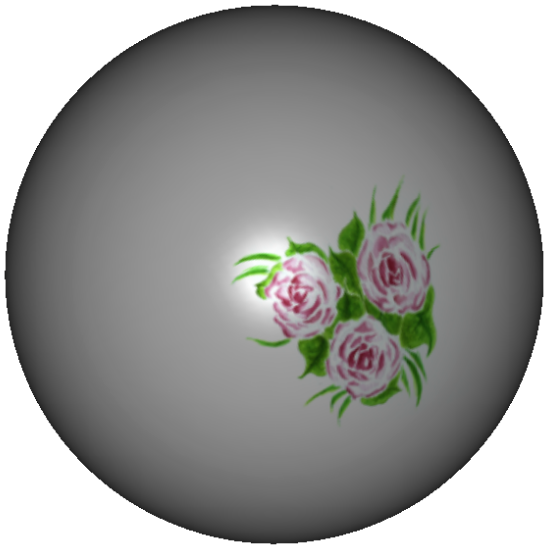
Permet de sélectionner des modes d'utilisation de texture. On peut par exemple créer des textures cycliques avec `pname` égal à `GL_TEXTURE_WRAP_S` ou `GL_TEXTURE_WRAP_T` et `param` égal à `GL_REPEAT`. Avec le paramètre `pname` égal à `GL_TEXTURE_MAG_FILTER` ou `GL_TEXTURE_MIN_FILTER`, on définit comment les couleurs de textures sont interpolées lorsqu'un pixel de l'écran ne correspond pas exactement à un pixel de l'image de texture (ce qui est généralement le cas).

Exemple 2. Plaquage de texture sur une sphère. La texture est chargée à partir d'un fichier PNG. On ne décrit pas ici le chargement de fichier PNG mais on suggère d'utiliser par exemple la librairie Open Source `libpng`. La sphère est affichée en tant que quadrique, ce qui permet de générer automatiquement des coordonnées de textures (contrairement à `glutSolidSphere`). La technique des quadriques permet aussi d'afficher des cônes, cylindres, paraboloïdes, etc.

textures//progc/exempleCode.c

```
1  #include <GL/glut.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <my_png_lib.h> /* utiliser une librairie comme libpng */
5
6  GLushort largeur_fenetre=500;
7  GLushort hauteur_fenetre=500;
8
9  GLint angle=0;
10
11 GLuint id = 0;  /* texture ID */
12
```

(a) Fichier PNG (extrait)



(b) Sphère texturée

```
13 void ChargeTexturePNG(const char *fichier)
14 {
15   /* Lecture du fichier PNG et initialisation d'un bitmap */
16   LirePNG(fichier, &largeur, &hauteur, &texels);
17   /* (voir librairies Open Source libpng, liblpg,...) */
18
19   /* Generation texture */
20   glGenTextures(1, &id);
21   glBindTexture(GL_TEXTURE_2D, id);
22
23   /* Paramtres de filtres linaires */
24   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
25   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
26
27   /* Cration de la texture */
28   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, largeur, hauteur, 0, GL_RGB,
29         GL_UNSIGNED_BYTE, png_tex->texels);
30
31   free(texels);
32 }
33
34 void Affichage(void)
35 {
36   GLUquadricObj *quadrique;
37   GLfloat diffuse[] = {1.0,1.0,1.0,1.0};
38   GLfloat specular[] = {1.0,1.0,1.0,1.0};
39   GLfloat ambient[] = {0.2,0.2,0.2, 1.0};
40   GLfloat shininess = 110.0;
41   glClearColor(1.0,1.0,1.0,1.0);
42   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
43
44   glMatrixMode(GL_MODELVIEW);
```

```
45    glPushMatrix();
46    gluLookAt(8, 10, 15,    /* rglage de la camra */
47              0,0,0,
48              0,1,0);
49    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
50    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,specular);
51    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
52    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
53    glRotatef(0.05*angle, 0.0, 1.0, 0.0);
54
55    glBindTexture (GL_TEXTURE_2D, id); /* slection de la texture */
56    glEnable (GL_TEXTURE_2D); /* activation des textures */
57
58    quadrique = gluNewQuadric(); /* cration d'une quadrique */
59    /* initialisation des coordonnes de texture : */
60    gluQuadricTexture(quadrique, GL_TRUE);
61    gluQuadricNormals(quadrique, GLU_SMOOTH); /* init normales */
62    gluSphere(quadrique, 2, 128 , 64); /* dessin d'une sphre */
63
64    glDisable (GL_TEXTURE_2D); /* dsactivation des textures */
65
66    glPopMatrix();
67    glutSwapBuffers();  /* Envoyer le buffer  l'cran */
68 }
69
70 void Redimensionnement(int l,int h)
71 {
72    GLfloat lightPosition0[] = {0.0,0.0,10.0,0.0};
73    GLfloat diffuseLight0[] = {0.6,0.6,0.6,1.0};
74    GLfloat specularLight0[] = {0.6,0.6,0.6,1.0};
75    largeur_fenetre = l;
76    hauteur_fenetre = h;
77    glViewport(0,0,(GLsizei)largeur_fenetre,(GLsizei)hauteur_fenetre);
78    glMatrixMode(GL_PROJECTION);
79    glLoadIdentity();
80    gluPerspective(20,l/(GLdouble)h, 0.01, 10000);
81    glMatrixMode(GL_MODELVIEW);
82    glLoadIdentity();
83    glLightfv(GL_LIGHT0,GL_DIFFUSE, diffuseLight0);
84    glLightfv(GL_LIGHT0,GL_SPECULAR, specularLight0);
85    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition0);
86 }
87
88 void IdleFonction(void)
89 {
90    angle++;
91    glutPostRedisplay();
92 }
93
94 int main(int argc, char**argv)
95 {
96    glutInit(&argc, argv);
97    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
98    glutInitWindowPosition(100,100);
99    glutInitWindowSize(largeur_fenetre,hauteur_fenetre);
100   glutCreateWindow("Source place  l'infini derire l'observateur");
```

```
101     glEnable(GL_DEPTH_TEST);
102     glEnable(GL_LIGHTING);
103     glEnable(GL_LIGHT0);
104     glShadeModel (GL_SMOOTH);
105     glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
106     glShadeModel(GL_SMOOTH);
107     glutDisplayFunc(Affichage);
108     glutReshapeFunc(Redimensionnement);
109     glutIdleFunc(IdleFonction);
110
111     /* Chargement de la texture  partir du fichier PNG */
112     /* pass en argument */
113     if (argc != 2)
114       {
115         fprintf(stderr, "Usage : %s fichier.png\n", argv[0]);
116         exit(0);
117       }
118     ChargeTexturePNG(argv[1]);
119
120     glutMainLoop();
121     return 0;
122 }
```

Annexe A

Aide mémoire d'*openGL*

A.1 Types

```
GLushort /* unsigned short : entier non sign  sur 2 octets*/
GLint /* entier sign  su 4 octets*/
GLfloat /* nombre flottant simple pr cision sur 4 octets*/
GLdouble /* nombre flottant double pr cision sur 8 octets*/
GLubyte /* octet non sign  (0,...,255)*/
```

A.2 Évennements, GLUT

A.2.1 Fen tre

```
/* initialisation et cr ation d'une fen tre*/

/* r glage de la taille initiale de la fen tre*/
glutInitWindowSize (GLushort sx, GLushort sy);
/* r glage de la position initiale de la fen tre*/
glutInitWindowPosition (GLushort px, GLushort py);
/* cr ation de la fen tre avec un titre*/
glutCreateWindow (char *titre);

/* Evennement de redimensionnement de la fen tre*/
void glutReshapeFunc(void (*func)(GLushort, GLushor));
```

A.2.2 Initialisation du mode d'affichage

```
glutInitDisplayMode(GLint masque);
GLUT_RGB   /* mode RGB*/
GLUT_RGBA; /* mode RGBA avec transparence*/
GLUT_DEPTH /* avec buffer de profondeur*/
GLUT_SINGLE  /* pour application avec image fixe*/
GLUT_DOUBLE  /* avec double buffer pour animations*/
glEnable(GL_DEPTH_TEST);  /* permet l' limination des parties cach es*/
```

```
glEnableGL_CULL_FACE*/; /* active le culling ( limination de faces)*/
glutMainLoop();  /* boucle d'attente des  vennements*/
```

A.2.3 Évennements

```
/* d claration d'une fonction d'affichage :*/
void glutDisplayFunc(void (*func)(void));

/* pour provoquer un affichage*/
void glutPostRedisplay(void);

/* d claration d'une fonction de redimensionnement de la fen tre :*/
void glutReshapeFunc(void (*func)(int, int));
/* (les deux int sont les nouvelles dimensions de la fen tre)*/

/* d claration d'une fonction timer pour les aniamtions*/
void glutIdleFunc(void (*func)(void));

/* d claration d'une fonction de tra tement des saisies clavier :*/
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
/* l'unsigned char est la touche pr ss e*/
/* x et y donnent les coordonn es de la position de la souris*/

/* d claration d'une fonction de saisie des touches sp ciales :*/
void glutSpecialFunc(void (*func)(int key, int x, int y));
/* le param tre key est la touche pr ss e*/
/* x et y donnent les coordonn es de la position de la souris*/

/* touches sp ciales pour les fl ches au clavier*/
GLUT_KEY_LEFT, GLUT_KEY_RIGHT, GLUT_KEY_UP, GLUT_KEY_DOWN

/* d claration d'une fonction de gestion des clicks de souris :*/
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
/* valeurs possible du param tre button :*/
GLUT_LEFT_BUTTON, GLUT_RIGHT_BUTTON, GLUT_MIDDLE_BUTTON
/* valeurs possibles du param tre state :*/
GLUT_DOWN, GLUT_UP
/* x et y donnent les coordonn es de la position de la souris*/

/* d claration d'une fonction de gestion du mouvement de la souris*/
/* avec au moins un bouton de la souris enfonc */
void glutMotionFunc(void (*func)(int x, int y));
/* x et y donnent les coordonn es de la position de la souris*/

/* d claration d'une fonction de gestion du mouvement de la souris*/
/* sans aucun bouton de la souris enfonc */
void glutPassiveMotionFunc(void (*func)(int x, int y));
```

```
/* x et y donnent les coordonn es de la position de la souris*/
```

A.2.4 Menus

```
/* associer un menu  une fonction :*/
/* la fonction prend en param tre*/
/* le num ro de la commande de menu*/
/* actionn e par l'utilisateur*/
/* (faire un switch)*/
int glutCreateMenu(void (*func)(int value));
/* ajouter une commande  un menu*/
/* avec un num ro de commande :*/
void glutAddMenuEntry(char *name, int value);
/* Attacher le dernier menu d fini  la pression*/
/* d'un bouton de la souris*/
void glutAttachMenu(int button);

/* rappel :*/
GLUT_LEFT_BUTTON
GLUT_MIDDLE_BUTTON
GLUT_RIGHT_BUTTON

/* ajouter un sous-menu  un menu*/
void glutAddSubMenu(char *entryName, int menuIndex);
```

A.3 Couleur

```
glClearColor(GLfloat, GLfloat, GLfloat); /* s lectioner la couleur du fond*/
glClear(int masque); /*  fa age de certains buffers ( cran, z-buffer,...)*/
GL_COLOR_BUFFER_BIT;  /* masque pour le buffer de couleurs (image)*/
GL_DEPTH_BUFFER_BIT   /* masque pour le buffer de profondeur (z-buffer)*/
glColor3f(GLfloat, GLfloat, GLfloat);   /* s lectioner une couleur RGB*/
glColor4f(GLfloat, GLfloat, GLfloat, GLfloat); /* s lectioner une couleur RGBA*/
```

A.4 Param tres de la cam ra

```
/* Positionnement du rep re de la cam ra sur la position pos, dirig e
vers centre, avec le vecteur up vertical sur la vue*/
void gluLookAt(GLdouble posX, GLdouble posY, GLdouble posZ,
               GLdouble centreX, GLdouble centreY, GLdouble centreZ,
               GLdouble upX, GLdouble upY, GLdouble upZ);
/* R glage des param tres de la projection 3D->2D, angle
d'ouverture, ratio L/H, cliping plane proche et  loign */
void gluPerspective(GLdouble angleOuvertureY, GLdouble aspect,
            GLdouble zNear, GLdouble zFar);
/* R glage de la position et dimention de la fen tre graphique
```

```
en 2D (coordonn es 2D virtuelles)*/
void glViewport(GLint x, GLint y, GLsizei largeur, GLsizei hauteur);
```

A.5 Position, transformations, rotations

```
glMatrixMode(int); /* passage dans un mode donn }
GL_PROJECTION      /* mode projection*/
GL_MODELVIEW       /* mode modelview*/
glLoadIdentity(); /* r initialisation de la matrice dans un mode fix */

/* translation d'un vecteur v :*/
void glTranslatef(GLfloat vx, GLfloat vy, GLfloat vz);

/* changements d' chelle sur les trois axes :*/
void glScalef(GLfloat facteurx, GLfloat facteury, GLfloat facteurz);

\mttrotation vectorielle autour d'une droite de vecteur directeur v*/
void glRotatef(GLfloat angle, GLfloat vx, GLfloat vy,  GLfloat vz);

glPushMatrix();  /* empiler une matrice pour la restorer plu tard*/
glPopMatrix();   /* d piler une matrice pour restorer celle du dessous*/
```

A.6 Sommets

```
glVertex3f(GLfloat, GLfloat, GLfloat);  /* dessiner un sommet*/
glVertex3d(GLdouble, GLdouble, GLdouble); /* dessiner un sommet*/
glVertex3dv(GLdouble[]); /* dessiner unsommet*/
```

A.7 Points, segments, polygones

```
glBegin(XXX);
glEnd();
GL_POINTS /* dessiner des points*/
GL_LINES  /* dessiner des segments entre paires de sommets*/
GL_LINE_STRIP  /* dessiner une ligne bris e*/
GL_LINE_LOOP /* dessiner une ligne bris e ferm e*/
GL_POLYGON  /* dessiner un polygone convexe*/
GL_QUADS
GL_QUAD_STRIP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
glRectf(GLfloat, GLfloat, GLfloat, GLfloat); /* dessiner un rectangle*/
```

```
/* r gler l' paisseur du trait*/
void glPointSize(GLfloat);  /*  paisseur d'un point*/
void glLineWidth(GLfloat);  /*  paisseur des lignes*/

/* antialiassage*/
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

/* pointill s sur des droites*/
glEnable(GL_LINE_STIPPLE);
glLineStipple (GLint, GLushort);

/* Stries sur un polygone*/
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(GLubyte *)
```

A.8 Dessin, formes

```
/* Dessiner un cube :*/
/* faire des changements d' chelle pour obtenir un parall l pip de*/
glutWireCube(GLfloat taille);
glutSolidCube(GLfloat taille);

/* dessiner une sph re :*/
/* faire des changements d' chelle pour obtenir un ellipso de*/
glutWireSphere(GLfloat, int nbre_paralleles, int nbre_meridiens);
glutSolidSphere(GLfloat, int nbre_paralleles, int nbre_meridiens);

/* dessiner une th i re :*/
glutSolidTeapot(GLdouble taille);
glutWireTeapot(GLdouble taille );

/* dessiner une quadrique :*/
/* obtenir un pointeur de quadrique*/
GLUquadricObj* gluNewQuadric(void);
/* dessiner un disque :*/
void gluDisk(GLUquadricObj*, GLfloat, GLfloat,GLint,GLint);
/* lib rer le pointeur de quadrique apr s utilistion :*/
void gluDeleteQuadric(GLUquadricObj*);

/* Dessiner un caract re  l' cran*/
void glutStrokeCharacter(void *font, int character);
GLUT_STROKE_ROMAN
```

A.9 Affichage 3D et clairage

```
/* Activer l'affichage z-buffer avec clairage*/
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);

/* Propri t s de r flexion des mat riaux*/
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIANT, vector);
GL_FRONT_AND_BACK, GL_FRONT, GL_BACK
GL_AMBIANT, GL_DIFFUSE, GL_SPECULAR
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, value);

/* Sources lumineuses*/
GL_LIGHT0, GL_LIGHT1, GL_LIGHT2...
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, vector);
GL_POSITION, GL_AMBIANT, GL_DIFFUSE, GL_SPECULAR.
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);

/* Vecteurs normaux*/
glNormal3f, glNormal3fv

/*Mod le d' clairement et lumi re ambiante*/
glShadeModel(GL_FLAT);
glShadeModel(GL_SMOOTH);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, vector);
```

A.10 Vertex Arrays

```
void glEnableClientState(GLenum array);
GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_NORMAL_ARRAY,
GL_TEXTURE_COORD_ARRAY
void glDisableClientState(GLenum array);

void glVertexPointer(GLint size, GLenum type, GLsizei stride,
                     const GLvoid *pointer);
void glColorPointer(GLint size, GLenum type, GLsizei stride,
                    const GLvoid *pointer);
void glNormalPointer(GLenum type, GLsizei stride,
                      const GLvoid *pointer);
void glTexCoordPointer(GLint size, GLenum type, GLsizei stride,
                       const GLvoid *pointer);

void glDrawElements(GLenum mode, GLsizei count, GLenum type,
```

```
                    void *indices);
GL_POINTS, GL_LINES, GL_LINE_STRIP , GL_LINE_LOOP,
GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES,
GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN.

void glArrayElement(GLint numVertex);
```

A.11 Temps

```
glutGet(int);
GLUT_ELAPSED_TIME
```