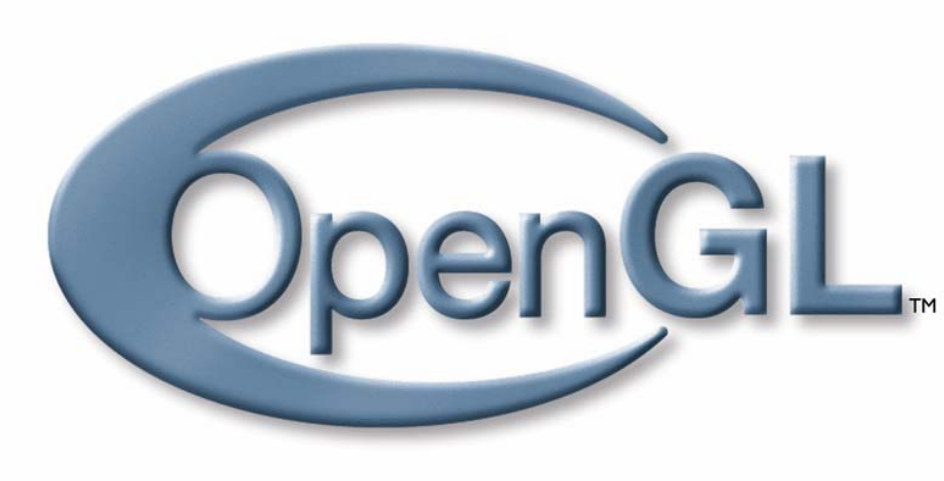
haute école arc ingénierie
neuchâtel berne jura saint-imier le locle porrentruy

COURS D'INFOGRAPHIE



21 novembre 2005, Pierre Chatelain

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

Table des matières

Introduction

Qu'est ce que OpenGL

OpenGL est une interface de programmation vers le matériel graphique. Cette interface compte environ 150 commandes qui sont utilisées dans les programmes interactifs en 3D. Etant donné que cette API est à l'origine professionnelle, il y a beaucoup de fonctions qui ne sont toujours pas câblées sur les cartes accélératrices 3D pour le grand public, et qui seront donc émulées en software.

La première particularité d'OpenGL est que la programmation est indépendante du matériel graphique utilisé, et que son domaine d'utilisation n'est pas confiné aux PC/Windows ou Mac. A l'origine développé par Silicon Graphics, OpenGL est aujourd'hui disponible sur les plates-formes x86 et 68000 entre autres, que ce soit sous Windows, Linux ou Mac. La conséquence immédiate est qu'aucune commande de fenêtrage, ou d'une manière générale toute commande dépendant de l'OS, n'est incluse et que seuls des commandes graphiques de bas niveaux existent (dessiner un point, un segment mais pas de sphère par exemple). Tout objet complexe est composé à partir d'un nombre limité de primitives graphiques.

Deuxièmement, OpenGL se base toujours sur d'autres librairies graphiques qui sont en général spécifiques au système d'exploitation ou qui apportent des fonctions complexes par dessus la librairie OpenGL. GLUT (l'OpenGL Utility Toolkit) et AUX en sont des exemples.

Une autre particularité est que certaines implémentations d'OpenGL sont conçues pour que la station sur laquelle on affiche ne soit pas celle sur laquelle tourne le programme. Cette fonctionnalité n'est utilisé que sur les systèmes basés sur l'X, et ne sera donc pas détaillée ici (en gros, cela est possible car OpenGL utilise une architecture client (un programme) – serveur (la carte vidéo et son driver OpenGL)).

Silicon Graphics n'a pas implémenté de version pour Linux de OpenGL. Il convient donc d'utiliser Mesa, qui est une librairie compatible OpenGL (attention, la compatibilité n'est ni garantie ni totale).

Les bonnes adresses

Voici quelques bonnes adresses à visiter pour avoir accès à des informations complémentaires et aux manuels de référence :

- OpenGL :
  http://www.opengl.org : site officiel, news, documentation, liens…
  http://trant.sgi.com/opengl : des exemples, des astuces
  http://reality.sgi.com/opengl : idem
  http://www.opengl.org/discussion_boards/cgi_directory/Ultimate.cgi : forum

- GLU :
  http://www.opengl.org/Documentation/GLX.html

- GLUT :
  http://www.opengl.org/resources/libraries/glut/glut_downloads.html

- Mesa :
  http://www.mesa3d.org

- AGL (Extension OpenGL pour Apple) :
  http://www.sd.tgs.com/Products/opengl.htm
  http://www.apple.com/opengl

- PGL (Extension OpenGL pour IBM OS/2 Warp
  http://www.austin.ibm.com/software/OpenGL/

Tout ce dont vous aurez besoin :

- Les documents ci-dessous peuvent se trouver, par exemple, à l'adresse suivante : http://rvirtual.free.fr/programmation/OpenGl/Utils
- Les manuels de référence : OpenGL_1_2_1.pdf et Glu1_3.pdf
- Les drivers OpenGL de base pour Windows (utilisez de préférence ceux de votre carte 3D, si ils fonctionnent. Il y a tout dans cet exécutable (fichiers .h, .lib, DLL ...).
- Les librairies de GLUT ou AUX pour tous ceux que la programmation Win32 n'intéresse pas, ou ceux qui ne connaissent pas le Visual Basic ou les MFC.
- Une API référence en HTML (une page par fonction !!!).
- Le document WORD dont ce tutorial s'inspire : OpenGL_cours.zip. (Le guide du routard dans OpenGL 1.2 de F. Gasser et J-F. Herouard, puis revu par Olivier Lanneluc)

GLUT et AUX : Pourquoi se compliquer la vie ?

GLUT, l'OpenGL Utility Toolkit

OpenGL ne s'occupant pas des opérations relatives au système ou au fenêtrage, il laisse la place à d'autres API, dont GLUT fait partie. Il existe des implémentations de GLUT pour Windows NT, X-Window et OS/2, ce qui permet de compiler le même code source sur toutes les plates-formes.

GLUT est une interface entre le programme et l'humanoïde qui se trouve derrière l'écran, la souris, ou le joystick. Sa programmation est de type événementielle et chaque agitation de l'humain est considérée comme un événement qui sera géré par des fonctions personnalisées (callback).

Un exemple

Voici un premier exemple de programme utilisant OpenGL :

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

int main(int argc, char** argv)
{
   GLint Xscreen = 800;
   GLint Yscreen = 600;

   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (Xscreen, Yscreen);
   glutCreateWindow ("Sur la barre de titre");

   glutDisplayFunc (Display);
   glutReshapeFunc (Reshape);
   glutIgnoreKeyRepeat(1);
   glutKeyboardFunc(Keyboard);
   glutKeyboardUpFunc(Keyboardup);
   glutSpecialFunc(Kpecial);
   glutSpecialUpFunc(Kpecialup) ;
   glutMouseFunc(Mouse);
   glutMotionFunc(Motion);
   glutPassiveMotionFunc(Motion);
   glutIdleFunc (Animation);
   glutMainLoop ();
  return 0;
}
```

L’initialisation

On remarque que l’on fait plusieurs appels à des fonctions GLUT pour créer une fenêtre. Dans l’ordre, on a :

- **glutInit(int *argc, char **argv)** : Doit être appelé avant toute autre fonction GLUT.
- **glutInitDisplayMode(unsigned int mode)** : Différencie les mode RGBA et couleur indexé, ainsi que le double buffering et les buffers à utiliser (depth, stencil, accumulation), que nous étudierons par la suite.
- **glutInitWindowPostion(int x, int y)** : no comment.
- **glutInitWindowSize(int width, int size)** : no comment.
- **int glutCreateWindow(char *string)** : Crée la fenêtre (et un contexte OpenGL) avec tous les paramètres spécifiés avant. Elle renvoie un identifiant unique pour la nouvelle fenêtre. Attention : la fenêtre ne sera pas affichée avant l’appel à **glutMainLoop().**

Les callbacks

Viennent ensuite les « callback ». Il s’agit de spécifier quelles sont les fonctions à appeler suivant les événements qui se produisent. On peut changer les fonctions appelées à n’importe quel moment. Si on spécifie la valeur NULL comme adresse de fonction à utiliser, cela a pour effet de désactiver le callback correspondant. Elles sont par défaut à NULL. Les plus utilisés des callback sont :

- **glutDisplayFunc(void(**func*)(void))** : Indique la fonction à utiliser lors de l'affichage graphique.
- **glutReshapeFunc(void(**func*)(int *w*, int *h*))** : Si la fenêtre est redimensionnée
- **glutKeyboardFunc(void(**func*)(unsigned char *key*, int *x*, int *y*))** et **glutMouseFunc(void(**func*)(int *button*, int *state*, int *x*, int *y*))** : Evénement clavier et souris
- **glutMotionFunc(void (**func*)(int *x*, int *y*))** : Déplacement souris avec bouton appuyé.

Les Animations

La fonction **glutIdleFunc(void (*func)(void))** est lancée lorsque le système n'a plus rien à faire (affichage terminé, pas d'évènements clavier …). NULL passé en paramètre désactive cette fonction. Elle sert à faire des animations.

Les fonctions de dessin 3D évoluées

GLUT peut se charger de dessiner des objets 3D centrés sur l'origine en mode fil de fer (wire) ou solide (solid). Les objets sont :

cône	icosahedron	teapot
cube	octahedron	tetrahedron
dodecahedron	sphère	torus

Pour dessiner un cube ou une sphère, on utilise par exemple les fonctions :

void glutWireCube(GLdouble *size*) ;
void glutSolidCube(GLdouble *size*) ;
void glutWireSphere(GLdouble *radius*, GLint *slices*, GLint *stacks*) ;
void glutSolidSphere(GLdouble *radius*, GLint *slices*, GLint *stacks*) ;

Exécuter le programme

Pour exécuter le programme et voir réellement quelque chose s'afficher après l'initialisation, il faut appeler la fonction :

- **glutMainLoop(void)** Ici, on rentre dans la boucle infinie d'exécution. Ça doit être à peut de choses prêt la dernière ligne du programme. On ne pourra sortir de cette boucle que par un exit().

Pour connaître la liste des fonctions utilisables de GLUT, reportez-vous à la page :
http://reality.sgi.com/mjk/spec3/spec3.html

AUX (AUXiliary Librairie), un GLUT pour Windows uniquement

Tout comme GLUT, AUX s'occupe pour vous des opérations relatives au système. Il est lui aussi de type évènementiel et adopte une syntaxe fort proche de celle de GLUT, à tel point que l'on

se demande qui a copié sur qui… Il suffit de remplacer GLUT par AUX dans le nom d'une fonction !!! On ne peut pas faire plus simple. Seuls changent les fonctions de gestions des évènements clavier et souris. GLUT n'a que 2 fonctions pour le clavier tandis que AUX a besoin d'une fonction par touche. Pour la souris, GLUT propose plus de fonctions que AUX.

Il suffit, pour utiliser AUX, d'inclure « glaux.h » et surtout « windows.h » dans votre programme et de compiler avec la librairie « glaux.lib » incluse dans votre projet. Là encore, tous ces fichiers sont fournis en annexe.

Exemple d'initialisation de la session OpenGL sous AUX :

```
#include <windows.h>       // obligatoire !!!
#include <gl\gl.h>         // fonctions OpenGL
#include <gl\glaux.h>      // fonctions AUX Library

int main(void)
{
     auxInitDisplayMode(AUX_DOUBLE | AUX_RGBA | AUX_DEPTH | AUX_STENCIL);
     auxInitPosition(100,100,LARGEUR,HAUTEUR);
     auxInitWindow("Moteur 3D");
     auxReshapeFunc(Reshape);
     auxKeyFunc(AUX_UP,Key_up);      //hé oui, une fonction par touche !
     auxKeyFunc(AUX_DOWN,Key_down);
     auxKeyFunc(AUX_LEFT,Key_left);
     auxKeyFunc(AUX_RIGHT,Key_right);
     auxKeyFunc(AUX_a,Key_a);
     auxKeyFunc(AUX_z,Key_z);
     auxKeyFunc(AUX_q,Key_q);
     auxKeyFunc(AUX_s,Key_s);
     auxKeyFunc(AUX_g,Key_g);
     auxKeyFunc(AUX_h,Key_h);
     auxKeyFunc(AUX_f,Key_f);
     auxKeyFunc(AUX_j,Key_j);
     auxKeyFunc(AUX_k,Key_k);
     auxKeyFunc(AUX_2,Key_2);
     auxKeyFunc(AUX_8,Key_8);
     auxKeyFunc(AUX_4,Key_4);
     auxKeyFunc(AUX_6,Key_6);
     auxKeyFunc(AUX_9,Key_9);
     auxKeyFunc(AUX_3,Key_3);
     auxMouseFunc(AUX_LEFTBUTTON,AUX_MOUSEDOWN,Mouse_Move); //pour la souris
     auxIdleFunc(Animation); //est lancé dès que le système n'a rien à faire.
     auxMainLoop(Display);   //Boucle principale
     return 0;
}
```

GLUT vs AUX

La librairie AUX a été développée par SGI au début de la vie d'OpenGL pour faciliter la création de petits programmes OpenGL de démonstration. Actuellement, l'utilisation de AUX n'est plus recommandée. Il vaut mieux utiliser GLUT. Cette dernière est plus complète que AUX et est utilisable sur un grand nombre de plates-formes.

Initialisation de la vue

Exemple d'initialisation 2D

OpenGL est plus connu pour son utilisation 3D, mais il est aussi prévu pour fonctionner en 2D. Comme c'est un peu plus facile à appréhender, on commence par là.

Une fois que le « contexte » OpenGL est crée, il reste à créer la vue. Cela se fait ainsi :

```
#define largeur 250
#define hauteur 250
void Reshape(int w, int h)   // En cas de changement de taille de la fenêtre.
{                            // et lors du lancement du programme
  GLfloat clipWidth, clipHeight;
  GLfloat Near = -100.0f;
  GLfloat Far = 100.0f;     // limites visuelles devant/derrière

  if (h == 0) h = 1;
  glViewport(0,0,w,h); //la partie qui va être dessinée va de (0,0) à (w,h)

  glMatrixMode(GL_PROJECTION); // mode projection
  glLoadIdentity();    // chargement de la matrice identité pour ce mode
  if (w<=h)
  {
    clipWidth  = (GLfloat)largeur;      // ces calculs ont pour but de garder
                                        // la proportionnalité entre les # axes
    clipHeight = (GLfloat)hauteur * h/w;// (un carré doit rester carré !!!)
  }
  else
  {
    clipWidth  = (GLfloat)largeur * w/h;
    clipHeight = (GLfloat)hauteur;
  }
   //          left          right        bottom         top
   glOrtho( -clipWidth/2, clipWidth/2, -clipHeight/2, clipHeight/2, Near, Far);

  glMatrixMode(GL_MODELVIEW); // mode modelview
  glLoadIdentity();           // chargement de la matrice identité pour ce mode
}
```

Il s'agit ici d'une initialisation d'une vue 2D.

Présentation des matrices d'OpenGL

En OpenGL, toutes les transformations géométriques sont entrées sous forme de matrice, la vue est définie par la position de la caméra (matrice de changement de repère) et par la matrice de projection.

La matrice de projection s'appelle **GL_PROJECTION**. Il n'est possible d'accéder qu'à une matrice à la fois, donc pour modifier une matrice, il faut d'abord l'activer avec **glMatrixMode(mode)**. La matrice choisie devient la « **matrice courante** ». Il reste à définir la

matrice de projection, mais inutile de se revoir le cours de prépa, car heureusement des fonctions de **gl** et de **glu** permettent de spécifier les matrices les plus courantes. La fonction :

void **gluOrtho2D**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

permet dans l'exemple de mettre une matrice de projection 2D de la dimension voulue.

```
glLoadIdentity();
```

initialise la matrice à l'identité.

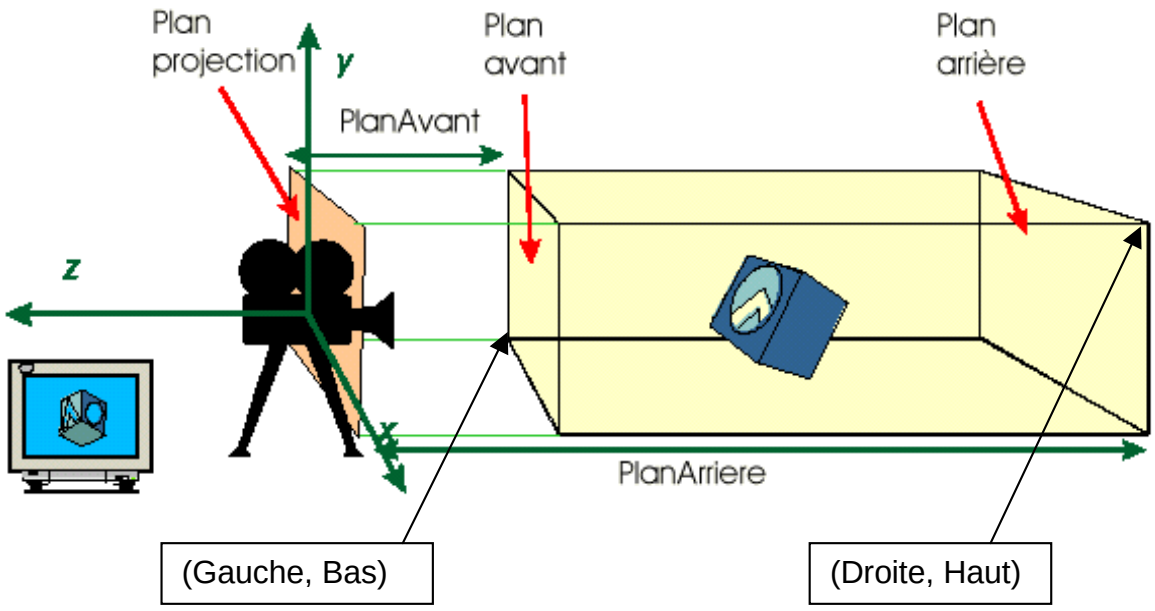
La matrice **GL_MODELVIEW** est en quelque sorte la matrice de positionnement de la vue. C'est le plus souvent une matrice de changement de repère. Il faut la considérer comme une matrice qui positionne les objets dans le repère du monde, et éviter de la considérer comme la position de la caméra, même si c'est aussi son rôle.

Il existe une troisième matrice, **GL_TEXTURE**, mais celle-ci n'est jamais utilisée directement.

Toutes sont des matrices 4x4 de float.

Projection parallèle

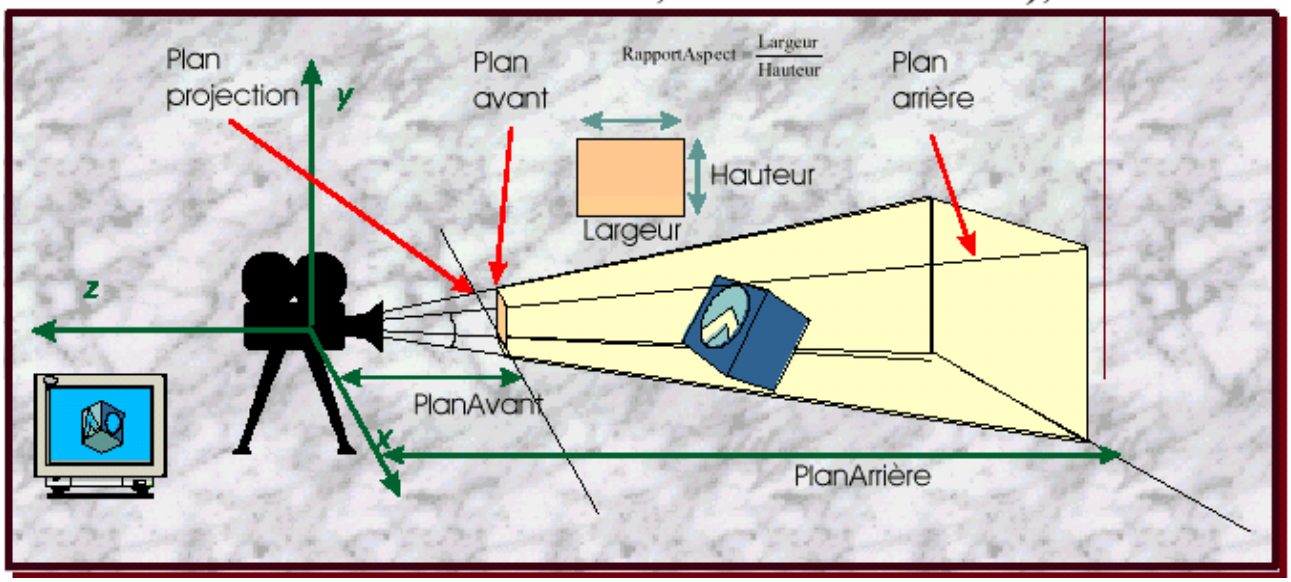
glOrtho (GLdouble Gauche, GLdouble Droite,
GLdouble Bas, GLdouble Haut,
GLdouble PlanAvant, GLdouble PlanArriere);



Un appel à **gluOrtho2D** est équivalent à celui de **glOrtho** avec :
PlanAvant = -1 et PlanArriere = +1

Projection perspective, 1ère méthode

gluPerspective (GLdouble AngleOuverture, GLdouble RapportAspect, GLdouble PlanAvant, GLdouble PlanArriere);



Projection perspective, 2ème méthode

glFrustrum (GLdouble Gauche, GLdouble Droite, GLdouble Bas, GLdouble Haut, GLdouble PlanAvant, GLdouble PlanArriere);

Dessiner

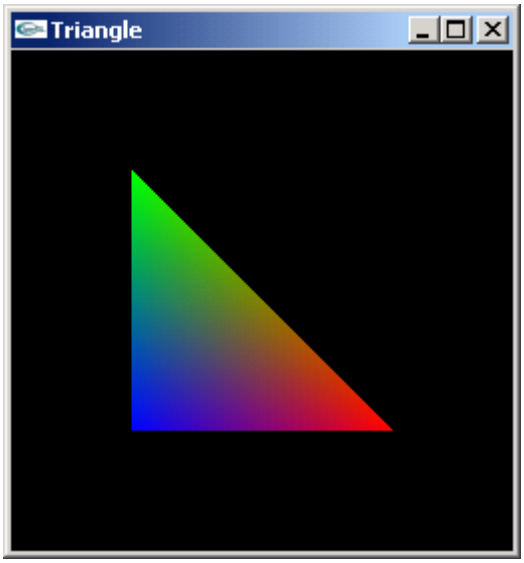
Un classique : le triangle

Les fonctions suivantes permettent de dessiner un triangle à l'écran :

```
void Display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);          //efface l'écran
  glBegin(GL_TRIANGLES);                 //commence le dessin d'un triangle
    glColor3f(0.0f,1.0f,0.0f);           //couleur du premier sommet
    glVertex3f(-65.0f,65.0f,0.0f);       //coordonnées du premier sommet

    glColor3f(1.0f,0.0f,0.0f);           //etc ...
    glVertex3f(65.0f,-65.0f,0.0f);
```

```
    glColor3f(0.0f,0.0f,1.0f);
    glVertex3f(-65.0f,-65.0f,0.0f);
  glEnd();                                  //fin du tracé du triangle
  glFlush();
}
```

Ce qui donne (imprimé en noir & blanc c'est moche, mais tant pis) :



Pour obtenir le dégradé des couleurs, il a simplement fallu déclarer les 3 sommets avec des couleurs différentes et l'OpenGL l'a fait, tout seul, comme un grand. On peut bien sûr désactiver cette fonctionnalité avec **glShadeModel(GL_FLAT),** qui demande à l'OpenGL de remplir le polygone avec la couleur du dernier point spécifié. Pour la réactiver, il faut taper : **glShadeModel(GL_SMOOTH)**.

Exemple d'initialisation de l'écran pour un rendu 3D

L'exemple qui va suivre utilise la fonction **glutReshapeFunc(Reshape)**. **Reshape** a pour but de redimensionner correctement la scène lorsqu'on agrandit la fenêtre d'exécution. Elle est exécutée au moins une fois, au début du programme.

```
//******************************************************
// Fonction gérant l'agrandissement de la fenêtre
// (elle est lancé une fois au début du programme par l'OpenGL)
void  Reshape(int w, int h) //changement de taille de la fenêtre ...
{
  #define LARGEUR 250              // largeur et hauteur de la fenêtre
  #define HAUTEUR 250
  #define PERSPECTIVE_ON 1

  GLdouble clipHeight;
  GLdouble clipWidth;
  GLdouble angle_of_view = 45.0;
  GLdouble aspect;

  if (h == 0) h = 1;
  if (w == 0) w = 1;
  glViewport(0,0,w,h);
```

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

  switch (PERSPECTIVE_ON) //si ce n'est pas une vue en perspective que
                          // l'on veut, alors c'est une vue orthogonale
  {
    case 0 :  if (w<=h)
              {
                clipWidth  = (GLdouble)LARGEUR;
                clipHeight = (GLdouble)HAUTEUR * h/w;
              }
              else
              {
                clipWidth = (GLdouble)LARGEUR * w/h;
                clipHeight = (GLdouble)HAUTEUR;
              }
              glOrtho(  -clipWidth/2.0, clipWidth/2.0,
                        -clipHeight/2.0, clipHeight/2.0,
                        -1.0, 1.0);
              break;
    case 1 :  aspect = (GLdouble) w / (GLdouble) h;
              gluPerspective(angle_of_view, aspect, 1.0, 100.0);
              break;
  }

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

Les Buffers

Il existe dans OpenGL des « buffers », qui d'une manière générale ne sont que des espaces mémoires contenant des données propres à l'OpenGL.

Le principal est le « **COLOR_BUFFER** », qui correspond à ce que l'on voit sur l'écran.

La fonction **glClear** permet de remplir tout un buffer avec la même valeur. La valeur par défaut est 0. Un appel à glClear sur le COLOR_BUFFER rempli l'écran de noir. **glClearColor** permet de changer la couleur de remplissage.

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f );   // Ce sont les valeurs par défaut.

...

glClear(GL_COLOR_BUFFER_BIT);            //efface l'écran

...
```

Un autre buffer, le back buffer, fonctionne différemment : pour ne pas voir se dessiner la scène à l'écran, on utilise la technique du « double buffering ». Le COLOR_BUFFER est en fait dédoublé : l'un est affiché (front buffer), l'autre est celui sur lequel on travaille (back buffer). A chaque fois qu'une image est calculée, on inverse (swap) les buffers avec la commande **glutSwapBuffers()** (à la place de **glFlush()**). Pour le programmeur, tout marche comme s'il dessinait toujours sur le même buffer. L'avantage est que l'utilisateur ne vois pas se dessiner l'image.

Les autres buffers standards sont le Depth buffer, le Stencil buffer, l'Accumulation buffer. Chaque buffer joue un rôle précis, mais ils ne seront présentés que lorsque nous en aurons besoin. D'autres sont des extensions propriétaires (T-Buffer, W-Buffer, …).

On initialise un ou plusieurs buffers au début du programme avec la commande :

```
glutInitDisplayMode(buffer1 | buffer2 | buffer3 | ...);
```

Les noms des buffers sont les suivants :

- Buffer simple **GLUT_SINGLE**
- Double buffering : **GLUT_DOUBLE**
- Color Buffer (image de l'écran) : **GLUT_RGBA**
- Depth Buffer : **GLUT_DEPTH**
- Stencil buffer : **GLUT_STENCIL**

De même, pour effacer ces buffers, on utilise la commande :

```
glClear( buffer1 | buffer2 | buffer 3 | ...);
```

avec, cette fois-ci, comme nom pour les buffers :

- Color Buffer (image de l'écran) : **GL_COLOR_BUFFER_BIT**
- Depth Buffer : **GL_DEPTH_BUFFER_BIT**
- Stencil buffer : **GL_STENCIL_BUFFER_BIT**

Les primitives graphiques

Le début de l'entrée d'une primitive se fait avec **glBegin**.
Par exemple : **glBegin(GL_TRIANGLES)**.
Les possibilités sont (tirées de la man page de glBegin) :

- **GL_POINTS**
  Treats each vertex as a single point. Vertex n defines point n. N points are drawn.

- **GL_LINES**
  Treates each pair of vertices as an independent line segment. Vertices 2n-1 and 2n define line n. N/2 lines are drawn.

- **GL_LINE_STRIP**
  Draws a connected group of line segments from the first vertex to the last. Vertices n and n+1 define line n. N-1 lines are drawn.

- **GL_LINE_LOOP**
  Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n+1 define line n. The last line, however, is defined by vertices N and 1. N lines are drawn.

- **GL_TRIANGLES**
  Treates each triplet of vertices as an independent triangle. Vertices 3n-2, 3n-1, and 3n define triangle n. N/3 triangles are drawn.

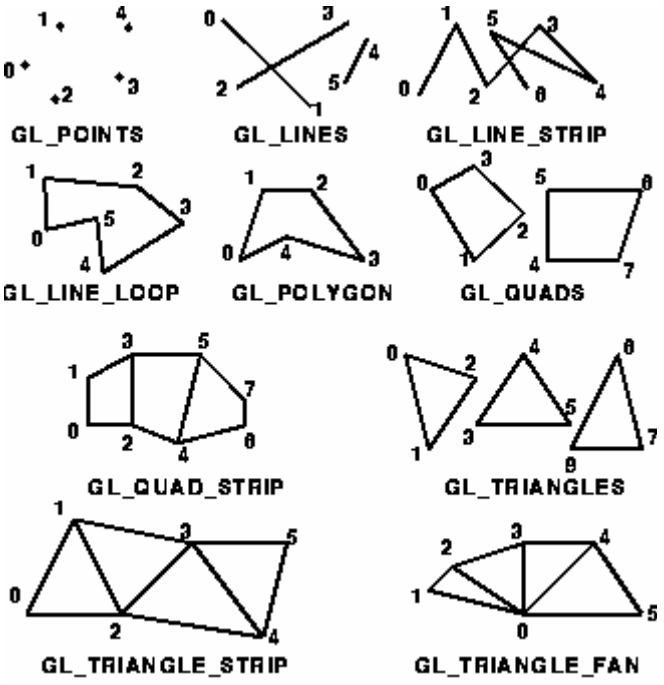
- **GL_TRIANGLE_STRIP**
  Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n, vertices n, n+1, and n+2 define triangle n. For even n, vertices n+1, n, and n+2 define triangle n. N-2 triangles are drawn.

- **GL_TRIANGLE_FAN**
  Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1, n+1, and n+2 define triangle n. N-2 triangles are drawn.

- **GL_QUADS**
  Treats each group of four vertices as an independent quadrilateral. Vertices 4n-3, 4n-2, 4n-1, and 4n define quadrilateral n. N/4 quadrilaterals are drawn.

- **GL_QUAD_STRIP**
  Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2n-1, 2n, 2n+2, and 2n+1 define quadrilateral n. N/2-1 quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

- **GL_POLYGON**
  Draws a single, convex polygon. Vertices 1 through N define this polygon.



glVertex permet d'entrer un point.

Le format général des fonctions d'OpenGL est :

```
{nom}{1234}{b s i d ub us ui}{v}
```

respectivement le nom de la fonction, le nombre de paramètres, leur type, passé directement ou par pointeur (vecteur). Toutes les caractéristiques de la fonction sont optionnelles sauf le nom.

glVertex existe par exemple sous les formes :

```
glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d,
glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f,
glVertex4i, glVertex4s, glVertex2dv, glVertex2fv, glVertex2iv,
glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv…
```

glColor fixe la couleur courante. De même elle existe sous de nombreuses formes.

glEnd indique la fin de l'entrée de coordonnées.

Cependant jusqu'ici rien n'est affiché : OpenGL a mis tout ce qu'il y a à faire dans un buffer, mais ne l'affiche que lorsqu'on lui demande, avec **glFlush()**, qui rend la main immédiatement, ou avec **glFinish()**, qui attend que tout soit affiché (peu d'intérêt). Si on utilise le double buffering, il faut appeler **glutSwapBuffers()** à la place de **glFlush()**.

Bouger le triangle, utilisation de la matrice de vue

Rotation et translation

Pour bouger la « caméra » ou un objet (c'est la même chose de bouger tous les objets ou la caméra), il faut utiliser la matrice de vue (**GL_MODELVIEW**).

Comme calculer directement une matrice de changement de repère n'est pas vraiment évidente, il existe des fonctions prédéfinies : **glRotate** et **glTranslate**.

```
void glRotated( GLdouble angle, GLdouble x, GLdouble y, GLdouble z );
void glRotatef( GLfloat angle, GLfloat x, GLfloat y, GLfloat z );

void glTranslated( GLdouble x, GLdouble y, GLdouble z );
void glTranslatef( GLfloat x, GLfloat y, GLfloat z );
```

Ces fonctions multiplient à **droite** la matrice de vue. Il faut donc faire attention à l'ordre dans lequel les transformations sont entrées : **c'est la dernière transformation entrée qui sera appliquée en premier.**

Rappels :

- soient f de matrice F, g de matrice G et h de matrice H. h°g°f a pour matrice HGF donc il faut entrer les matrices dans l'ordre H,G,F.
- les rotations et les translations ne sont pas commutatives

Par exemple, pour faire tourner un objet sur lui-même, il faut entrer en dernier la matrice de rotation. Pour simplifier, l'origine du repère local d'un objet est par convention son centre de rotation (d'inertie) pour permettre de le faire tourner sur lui-même sans calcul complexe.

Tout cela peut paraître compliqué, mais un ordinateur ne sait comprendre que les matrices, et un énorme avantage est le temps de calcul : même si de nombreuses transformations sont entrées, il ne reste qu'une seule matrice à appliquer à tous les points entrés.

Ordre des transformations

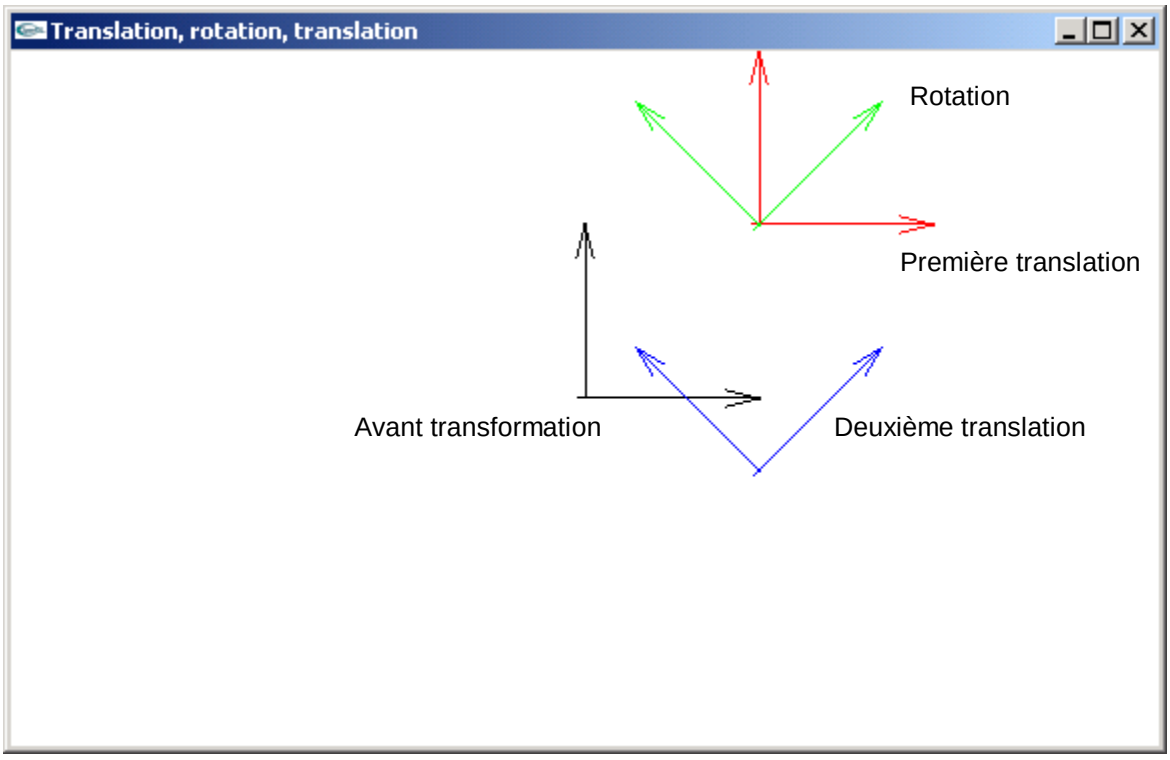
Soit l'extrait de code ci-dessous :

```
glMatrixMode(GL_MODELVIEW);          // Spécifie quelle pile sera utilisée
                                     // et quelle matrice sera multipliée.
glLoadIdentity();                    // Charger la matrice identité
glTranslatef(5.0, 5.0, 0.0);         // Une translation
glRotatef(45.0, 0.0, 0.0, 1.0);      // Une rotation
glTranslatef(-5.0, -5.0, 0.0);       // Une autre translation
```

La matrice courante de transformation (CTM) devient alors :

```
CTM = T(5.0, 5.0, 0.0) . R(45.0, 0.0, 0.0, 1.0) . T(-5.0, -5.0, 0.0)
```

On obtient le résultat suivant :



Pile de matrices

Mais comment inverser les transformations entrées ? Calculer les inverses est une perte de temps. Une pile est en effet associée à chaque matrice. Les fonctions sont :

void **glPushMatrix**(void);
void **glPopMatrix**(void);

qui respectivement empile et dépile la matrice courante. Même s'il n'y a que deux fonctions, il existe bien trois piles différentes (une par matrice). Comme pour les transformations, seule la matrice active peut-être empilée / dépilée (ce qui est logique, puisqu'une matrice est empilée pour la sauvegarder avant une transformation).

Souvent, un objet est caractérisé par ses angles de rotation (ax, ay, az) et sa position (x, y, z). Son affichage est alors du type :

```
glPushMatrix() ;
  glTranslatef(x,y,z) ;
  glRotatef(ax, 1.0, 0.0, 0.0) ;
  glRotatef(ay, 0.0, 1.0, 0.0) ;
  glRotatef(az, 0.0, 0.0, 1.0) ;
  glBegin(…)
    ...
  glEnd() ;
glPopMatrix() ;
```

Exemple d'utilisation : un soleil avec ses planètes, et les lunes des planètes. Le principe est le même :

- Dessin du soleil
- **PushMatrix**
  - Translation dans le repère local d'une planète
  - Dessin de la planète
  - **PushMatrix**
    - Translation dans le repère local d'une lune
    - Dessin de la lune
  - **PopMatrix**
  - **PushMatrix**
    - Translation dans le repère local d'une autre lune
    - Dessin de la lune
  - **PopMatrix**
- **PopMatrix**
- **PushMatrix**
  - Translation dans le repère local d'une autre planète
  - Dessin de la planète
  - **PushMatrix**
    - Translation dans le repère local d'une lune
    - Dessin de la lune
  - **PopMatrix**
- **PopMatrix**
- etc.

Dessiner en 3D

Initialisation

Quelques changements dans l'initialisation :

```
void  Reshape(int w, int h) //changement de taille de la fenêtre ...
{
  GLdouble clipHeight;
  GLdouble clipWidth;
  GLdouble angle_of_view = 45.0;
  GLdouble RapportAspect;

  if (h == 0) h = 1;
  if (w == 0) w = 1;
  glViewport(0,0,w,h);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();          // Charger la matrice de projection identité
  RapportAspect = (GLdouble)w / (GLdouble)h;
  gluPerspective(angle_of_view, RapportAspect, 1.0, 10.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();          // Charger la matrice de visualisation identité
  // Déplacer le point de vue (la caméra) :
  // La position de la caméra est en (0, 0, 3)
  // La direction de visée de la caméra est le point (0, 0, 0)
  // L'axe dirigé vers le haut est le vecteur de composantes (0, 1, 0)
  gluLookAt(0.0, 0.0, 5.0,               // Position de la caméra
            0.0, 0.0, 0.0,               // Direction de visée
            0.0, 1.0, 0.0);              // Axe dirigé vers le haut

  glEnable(GL_DEPTH_TEST);               // Activer le depth buffer
}
```

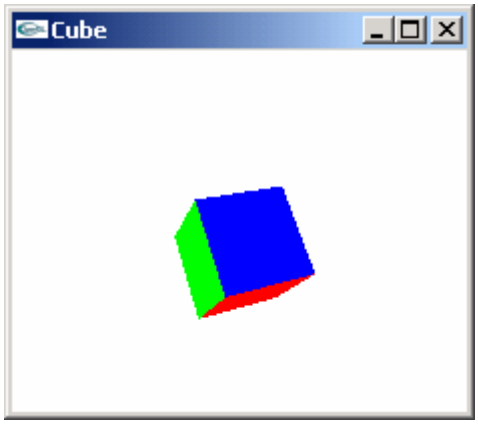
gluPerspective construit une matrice de projection de champ de vue de 45°, de rapport largeur/hauteur aspect, de profondeur de champ (1.0, 10.0). Seuls les objets situés à une distance comprise entre 1.0 et 10. de la caméra sont affichés.

gluLookAt construit une matrice de changement de repère à partir de trois vecteurs : la position de la caméra, la direction où elle regarde, un vecteur indiquant le haut du volume visionné.

void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,
GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

Exemple : un Cube

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
total %= 360*500;       // temps écoulé en ms
glPushMatrix();
  glTranslatef(0.0f, 5.0f+(GLfloat)sin(GLfloat(total)/500.0f)*3.0f, 0.0f);
  glRotatef(total/50.0f, 0.3f, 0.5f, 0.8f);
  glBegin(GL_QUADS);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f, -0.5f);
    glVertex3f( 0.5f, -0.5f, -0.5f);
    glVertex3f( 0.5f,  0.5f, -0.5f);
    glVertex3f(-0.5f,  0.5f, -0.5f);

    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f,  0.5f);
    glVertex3f( 0.5f, -0.5f,  0.5f);
    glVertex3f( 0.5f,  0.5f,  0.5f);
    glVertex3f(-0.5f,  0.5f,  0.5f);

    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f( 0.5f, -0.5f,  0.5f);
    glVertex3f( 0.5f, -0.5f, -0.5f);
    glVertex3f( 0.5f,  0.5f, -0.5f);
    glVertex3f( 0.5f,  0.5f,  0.5f);

    glColor3f(0.0f, 1.0f, 1.0f);
    glVertex3f(-0.5f, -0.5f,  0.5f);
    glVertex3f(-0.5f, -0.5f, -0.5f);
    glVertex3f(-0.5f,  0.5f, -0.5f);
    glVertex3f(-0.5f,  0.5f,  0.5f);

    glColor3f(1.0f, 0.0f, 1.0f);
    glVertex3f(-0.5f,  0.5f,  0.5f);
    glVertex3f(-0.5f,  0.5f, -0.5f);
    glVertex3f( 0.5f,  0.5f, -0.5f);
    glVertex3f( 0.5f,  0.5f,  0.5f);

    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-0.5f, -0.5f,  0.5f);
    glVertex3f(-0.5f, -0.5f, -0.5f);
    glVertex3f( 0.5f, -0.5f, -0.5f);
    glVertex3f( 0.5f, -0.5f,  0.5f);
  glEnd();
glPopMatrix();
```



Depth Buffer

En 3D, tout n'est pas affiché : il y a des faces cachées. Comment déterminer ce qui est affiché ? En affichant ce qui est le plus proche de la caméra.

C'est le rôle du depth buffer : il sauvegarde la distance du point actuellement affiché. Si un point plus proche arrive ensuite, il remplace le point et la valeur dans le depth buffer est changée.

Le depth buffer a les mêmes dimensions que le color buffer. Il s'active grâce à

glEnable(GL_DEPTH_TEST);

et se désactive par

glDisable(GL_DEPTH_TEST);

Il ne faut pas, bien sûr, oublier de l'initialiser au début du programme avec la commande :

glutInitDisplayMode(… | GLUT_DEPTH | …);

Il doit être effacé avec

glClear(… | GL_DEPTH_BUFFER_BIT | …);

à chaque fois que la scène est redessinée.

Paramètres du rendering

Anti-aliasing

L'OpenGL est une « state machine » : les paramètres de dessin consistent en une série de variables (très nombreuses, voir les annexes de la spécification OpenGL pour une liste complète).

La plupart sont modifiables par :

void **glEnable**(GLenum cap);
void **glDisable**(GLenum cap);

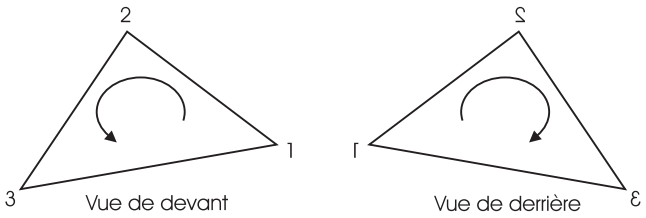
Pour l'anti-aliasing, les paramètres sont : **GL_POINT_SMOOTH**, **GL_LINE_SMOOTH**, et **GL_POLYGON_SMOOTH**. Le but de l'anti-aliasing est de rendre apparemment plus lisse les lignes en faisant un dégradé sur les contours des points.

Culling

Le culling consiste à ne dessiner que les triangles dont les points sont dans le sens des aiguilles d'une montre (« **clockwise** ») ou dans l'autre sens (« **counter-clockwise** »). L'intérêt est l'optimisation : l'objectif est de toujours dessiner les faces extérieures d'un objet dans le sens contraire des aiguilles d'une montre, et de filtrer les faces cachées avec le culling.

Prenons l'exemple du cube : les faces sont toujours affichées dans le même ordre, peut importe si elles sont visibles ou pas. C'est le depth buffer qui permet d'afficher les bonnes faces. Cependant le test du depth buffer est long : il faut calculer la distance de chaque point de chaque face par rapport à la caméra, puis tester le depth pour chaque point (et mettre à jour le depth si le point est plus près). C'est un test beaucoup trop important pour les faces dont on est sûr qu'elles ne seront pas affichées. Il est plus rapide d'éliminer une face entière d'un coup avec le culling.

L'astuce utilisée par le culling pour savoir de quel coté est vue une face consiste à regarder dans quel sens les points des triangles sont dessinés (en OpenGL tout est décomposé en triangles) : dans le sens des aiguilles d'une montre ou dans l'autre sens.



Cette technique est très rapide : en gros il suffit de regarder le signe du produit vectoriel de deux cotés d'un triangle. La seule contrainte pour le programmeur est de dessiner dans le bon sens les points.

Le standard est le **CCW** (counter-clockwise), que ce soit pour les primitives de glu ou glut, ou encore pour 3DS ou Lightwave.

En OpenGL, les fonctions correspondantes sont :

void **glCullFace**(GLenum mode);

avec mode à **GL_FRONT, GL_BACK** ou **GL_FRONT_AND_BACK**, pour indiquer quel coté doit être éliminé au culling.

void **glFrontFace**(GLenum mode);

avec mode à **GL_CW** ou **GL_CCW** pour indiquer quelles sont les faces de devant.

Un appel à **glEnable** / **glDisable** avec en paramètre **GL_CULL_FACE** active / désactive le culling.

Exemple complet

Il s'agit d'un simple triangle coupé par une ligne. Les sommets sont dessinés en tournant dans le sens contraire des aiguilles d'une montre (CCW). Un point est également dessiné.

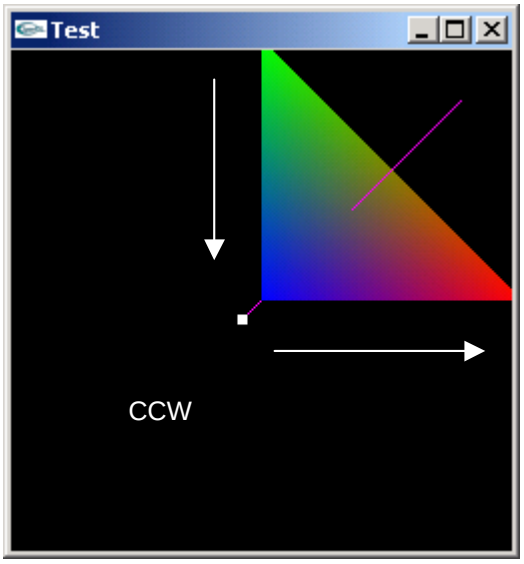
```
// Fichier Depth_buffer.cpp

#include <gl\glut.h>                // Librairie GLUT
#include "define.h"
#include "evenements.h"

void SetupRC()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f ); // Ce sont les valeurs par défaut
   // Anti-aliasing
   glShadeModel(GL_SMOOTH);
// glShadeModel(GL_FLAT);
   glEnable(GL_POINT_SMOOTH);
// glDisable(GL_POINT_SMOOTH);
   // Culling
   glFrontFace(GL_CCW);
// glFrontFace(GL_CW);
   glCullFace(GL_BACK);  // GL_BACK, GL_FRONT ou GL_FRONT_AND_BACK
   glEnable(GL_CULL_FACE);
// glDisable(GL_CULL_FACE);
   // Depth buffer
   glEnable(GL_DEPTH_TEST);
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  // Ne pas oublier d'initialiser le Depth buffer
  // Pas de Double buffering ici
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(LARGEUR, HAUTEUR);
  glutCreateWindow("Test");

  SetupRC(); // Initialisation
  glutReshapeFunc(Reshape);//fonction gérant le redimensionnement de la fenêtre
  glutDisplayFunc(Display);
  glutMainLoop();
  return 0;         //boucle principale
}
```

```
// Fichier Evenenements.h
#ifndef EVENEMENTS_H
#define EVENEMENTS_H

void Display();
void Reshape(int, int);

#endif
```

```
// Fichier Define.h
#ifndef DEFINE_H
#define DEFINE_H

#define LARGEUR 250          // largeur et hauteur de la fenêtre
#define HAUTEUR 250

#endif
```

```
// Fichier Global.h
#ifndef GLOBAL_H
#define GLOBAL_H

extern GLfloat clipHeight;
extern GLfloat clipWidth;

#endif
```

```
// Fichier Evenements.cpp

#include <gl\glut.h>                // Librairie GLUT
#include "define.h"
#include "global.h"
#include "evenements.h"

void Display(void)
{
  // Ne pas oublier d'effacer le Depth buffer
  // à chaque rafraîchissement de l'écran
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glColor3f(1.0f,1.0f,1.0f);
  glPointSize(5.0f);
  glBegin(GL_POINTS);
    glVertex2f(-10.0f,-10.0f);
  glEnd();

  glColor3f(1.0f,0.0f,1.0f);
  glBegin(GL_LINES);
    glVertex3f(-10.0f,-10.0f,-50.0f);
    glVertex3f(100.0f,100.0f,50.0f);
  glEnd();

  glBegin(GL_TRIANGLES);
    glColor3f(0.0f,1.0f,0.0f);
    glVertex3f(0.0f,130.0f,0.0f);

    glColor3f(0.0f,0.0f,1.0f);
    glVertex3f(0.0f,0.0f,0.0f);

    glColor3f(1.0f,0.0f,0.0f);
    glVertex3f(130.0f,0.0f,0.0f);
  glEnd();
  glFlush();
}
```



la ligne coupe le triangle et est partiellement cachée

```
void Reshape(int w, int h) //En cas de changement de taille de la fenêtre.
{                          //et lors du lancement du programme
  GLfloat Near = -100.0f;
  GLfloat Far = 100.0f;            //limites visuelles devant/derrière

  if (h == 0) h = 1;
  if (w == 0) w = 1;
  glViewport(0,0,w,h);   //la partie qui va être dessinée va de (0,0) à (w,h)

  glMatrixMode(GL_PROJECTION);     //mode projection
  glLoadIdentity();          //chargement de la matrice identité pour ce mode
  if (w<=h)
  {
    clipWidth  = (GLfloat)LARGEUR;     // ces calculs ont pour but de garder
                                       // la proportionnalité entre les # axes
    clipHeight = (GLfloat)HAUTEUR * h/w;  // (un carré doit rester carré !!!)
  }
  else
  {
    clipWidth  = (GLfloat)LARGEUR * w/h;
    clipHeight = (GLfloat)HAUTEUR;
  }
   //          left          right        bottom         top
   glOrtho( -clipWidth/2, clipWidth/2, -clipHeight/2, clipHeight/2, Near, Far);

  glMatrixMode(GL_MODELVIEW);      // Mode modelview
  glLoadIdentity();          // Chargement de la matrice identité pour ce mode
}
```

```
// Fichier Global.cpp
#include <gl\glut.h>                // Librairie GLUT

GLfloat clipHeight;
GLfloat clipWidth;
```

Les CallListes

Commandes OpenGL

Certains objets sont toujours les mêmes. Comme OpenGL traduit toutes les séquences entrées, il est intéressant de créer des listes pour regrouper les instructions. Cela accélère légèrement l'affichage et réduit le code. Une liste est alors codée par un entier (un seul entier est alors envoyé pour un objet quelconque).

En pratique, il faut :

- D'abord réserver des numéros de listes avec :

```
GLuint glGenLists( GLsizei range );
```

où "range" est le nombre de numéros de listes à réserver. Les numéros réservés vont de la valeur renvoyée à cette valeur + range.

- Ensuite appeler :

```
void glNewList( GLuint list, GLenum mode );
```

avec pour list le numéro de la liste à créer, et mode = **GL_COMPILE** ou **GL_COMPILE_AND_EXECUTE**. Les deux modes créent la liste, mais le second en plus l'exécute (l'affiche si on a un glFlush() à la fin).

- Mettre toutes les routines souhaitées (**glBegin, glColor, glVertex**…), elles sont enregistrées dans la liste. Certaines commandes ne peuvent pas être enregistrées dans une liste, mais il y en a peu et c'est plutôt logique (par exemple **glGenList, glFlush, glNewlist**…).

- Terminer par :

```
void glEndList( void );
```

- Pour afficher une liste, il faut appeler :

```
void glCallList( GLuint list );
```

ou

```
void glCallLists( GLsizei n, GLenum type, const GLvoid *lists);
```

pour afficher plusieurs listes d'un coup (type indique le type du tableau passé en paramètre : **GL_BYTE, GL_INT**…).

- Reste à effacer une liste lorsqu'elle n'est plus utile :

```
void glDeleteLists( GLuint list, GLsizei range );
```

dont les paramètres sont les même que **glGenList**.

Exemple

Il s'agit du même exemple qu'à la page précédente, mais cette fois un peu plus ordonnée avec des CallListes :

```
GLuint point, triangle, droite; // Pour les CallListes (variables globales)

void Make_CallListes()  // Cette fonction est appelée, par exemple, dans
                        // la fonction main
{
  point = glGenLists(3);
  triangle = point + 1;
  droite = triangle + 1;

  glNewList(point, GL_COMPILE);
    glColor3f(1.0f,1.0f,1.0f);
    glPointSize(5.0f);
    glBegin(GL_POINTS);
      glVertex2f(-10.0f,-10.0f);
    glEnd();
  glEndList();

  glNewList(triangle, GL_COMPILE);
    glBegin(GL_TRIANGLES);
      glColor3f(0.0f,1.0f,0.0f);
      glVertex3f(0.0f,130.0f,0.0f);
      glColor3f(0.0f,0.0f,1.0f);
      glVertex3f(0.0f,0.0f,0.0f);
      glColor3f(1.0f,0.0f,0.0f);
      glVertex3f(130.0f,0.0f,0.0f);
    glEnd();
  glEndList();

  glNewList(droite, GL_COMPILE);
    glColor3f(1.0f,0.0f,1.0f);
    glBegin(GL_LINES);
      glVertex3f(-10.0f,-10.0f,-50.0f);
      glVertex3f(100.0f,100.0f,50.0f);
    glEnd();
  glEndList();
}
```

```
void Display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glCallList(point);
  glCallList(triangle);
  glCallList(droite);

  glFlush();
}
```

L'éclairage

Les lumières

Modélisation de l'éclairage au moyen de quatre lumières :

- de la lumière **ambiante (GL_AMBIENT)** provenant de réflexions multiples et donc de toutes les directions (réfléchie dans toutes les directions)
- de la lumière **diffuse (GL_DIFFUSE)** provenant d'une direction particulière (réfléchie dans toutes les directions)
- de la lumière **spéculaire (GL_SPECULAR et GL_SHININESS)** provenant d'une direction particulière (réfléchie dans une direction préférentielle)
- de la lumière **émise** directement par des objets (**GL_EMISSION**, associée à une matière)

La couleur de la lumière est définie par quatre composantes **rouge, verte, bleue et alpha** (RGBA).

Remarques :

Dans la réalité, les sources de lumières sont des volumes (par exemple un filament), et la lumière se réfléchit à l'infini sur tous les objets.

Il est impossible de simuler un tel comportement, surtout quand il faut calculer une image en quelques millisecondes. Des simplifications ont été apportées :

- Les sources de lumière sont des points.
- Les ombres ne sont pas calculées automatiquement.
- Les objets illuminés ne rayonne pas de lumière sur les autres objets (la lumière ne se réfléchit pas sur les objets).

Pour plus de réalisme, les sources disposent de plusieurs paramètres qui n'ont pas forcément d'équivalent réel (voir le paragraphe suivant).

Une chose par contre ne change pas : la couleur visible d'un objet dépend à la fois de sa couleur propre et celle de la lampe, et l'intensité de la lumière qu'il reçoit dépend de l'angle de la surface avec la direction vers la source lumineuse.

Les sources lumineuses

a) La fonction **glEnable**() permet :

- de passer dans l'état d'éclairage d'une scène. ex. : glEnable(GL_LIGHTING);
- d'allumer une lumière précédemment définie. ex. : glEnable(GL_LIGHT0);

b) La fonction **glDisable**() permet:

- de quitter l'état d'éclairage d'une scène ex. : glDisable(GL_LIGHTING);
- d'éteindre une lumière précédemment définie ex. : glDisable(GL_LIGHT0);

c) Le nombre **maximal** de lumière est obtenu en appelant :

```
glGetIntegerv(GL_MAX_LIGHTS, &nb_lights);
```

d) La fonction **glLight*()** permet de préciser les caractéristiques d'une source lumineuse. Cette fonction accepte 3 paramètres :

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

- un nom (numéro) de lumière (**nb**). On peut créer au minimum 8 lumières : GL_LIGHT0 → GL_LIGHT7 (GL_LIGHTi = GL_LIGHT0 + 1)
- un nom de paramètre (**pname**) : le nom du paramètre que l'on veut modifier, pour la lumière dont le nom a été précisé en 1er paramètre
- une valeur (**v**) : la valeur du paramètre que l'on désire changer.

Voici un tableau récapitulatif des paramètres modifiables pour une lumière :

Nom du paramètre (pname)	***Valeur par défaut***	***Définition***
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	Intensité de la composante ambiante de la lumière exprimée en mode RGBA.
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) pour LIGHT0, (0.0, 0.0, 0.0, 1.0) pour les autres	Intensité de la composante diffuse de la lumière exprimée en mode RGBA.
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) pour LIGHT0, (0.0, 0.0, 0.0, 1.0) pour les autres	Intensité de la composante spéculaire de la lumière exprimée en mode RGBA.
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Position de la lumière, (x, y, z, w). Voir "Remarques iii)".
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	Direction de la lumière du spot.
GL_SPOT_EXPONENT	0.0	Facteur de concentration de la lumière du spot (à l'intérieur du cône de lumière autour de l'axe central du cône).
GL_SPOT_CUTOFF	180.0	Angle du cône de lumière mesuré entre l'axe du cône et ses bords. On prendra par exemple 180.0 pour un réverbère, 90.0 pour un plafonnier et 45.0 pour une lampe torche.

GL_CONSTANT_ATTENUATION	1.0	Atténuation constante de la lumière. Voir "Remarques i)".
GL_LINEAR_ATTENUATION	0.0	Atténuation linéaire de la lumière. Voir "Remarques i)".
GL_QUADRATIC_ATTENUATION	0.0	Atténuation quadratique de la lumière. Voir "Remarques i)".

Remarques :

i) atténuation de la lumière :

Elle est calculée selon la formule: $A=\frac{1}{k_c+k_l \cdot d+k_q \cdot d^2}$

Avec: d : distance à la source lumineuse
k_c : atténuation constante
k_l : atténuation linéaire
k_q : atténuation quadratique

ii) l'atténuation de la lumière affecte les composantes ambiante, diffuse et spéculaire de la lumière. Par contre l'atténuation n'agit pas sur les phénomènes d'émission.

iii) la position de la lumière est en coordonnées homogènes :

- si **w = 0.0**, la source lumineuse est à l'infini et le triplet de coordonnées (x, y, z) est interprétable comme le vecteur direction de la lumière (direction d'où vient la lumière).
- si **w = 1.0**, la source lumineuse est positionnée en (x, y, z) (par défaut, une lumière de position rayonne dans toutes les directions).
- Attention à la position de la lampe. Comme toutes les coordonnées entrées, elles sont immédiatement multipliées par la matrice ModelView. Si celle-ci change, il faut repositionner les lampes.
  En particulier :
  - Si glLight*() est appelée **avant** gluLookAt(), le déplacement de l'œil déplace aussi la position de la lampe.
  - Si glLight*() est appelée **après** gluLookAt(), le déplacement de l'œil ne modifie pas la position de la lumière.

Exemple

```
GLfloat luxPosition[]= {5.0, 5.0, 10.0, 1.0};
GLfloat ambiante[]= {0.0, 0.0, 0.5, 1.0};
GLfloat diffuse[]= {1.0, 0.5, 0.5, 1.0};
GLfloat speculaire[]= {0.5, 1.0, 1.0, 1.0};
.........................
glEnable(GL_LIGHTING);              // Lumière!
glLightfv(GL_LIGHT0, GL_POSITION, luxPosition);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambiante);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, speculaire);
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

```
glEnable(GL_LIGHT0);

dessineMoiUnMouton();

glDisable(GL_LIGHT0);
glDisable(GL_LIGHTING);
```

Les matériaux des objets de la scène

Il est possible, lors de l'illumination d'une scène 3D, de définir les propriétés des matériaux, la manière dont ils vont renvoyer la lumière. La fonction **glMaterial*()** permet cela.

Voici un tableau récapitulatif des paramètres modifiables pour un matériau :

void **glMaterial**{i f} [v](GLenum f, GLenum pn, TYPE v);

f indique quelle face (ou quelles faces) doit être remise à jour et peut prendre les valeurs : **GL_FRONT, GL_BACK, GL_FRONT_AND_BACK**

pn est le nom du paramètre et **v** sa valeur.

pn	***Valeur par défaut***	***Définition***
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Couleur ambiante.
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Couleur diffuse.
GL_AMBIENT_AND_DIFFUSE		Couleur ambiante et diffuse.
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Couleur spéculaire.
GL_SHININESS	0.0	Focalisation spéculaire (0 → 128)
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	Couleur émise.
GL_COLOR_INDEXES	(0, 1, 1)	Indexes des couleurs ambiantes, diffuse et spéculaire.

Exemple 1 :

```
GLfloat mat_lumineux0[]={0.0, 0.0, 0.0, 0.0};  // Noir
GLfloat mat_lumineux1[]={1.0, 0.0, 0.0, 0.0};  // Rouge
GLfloat mat_lumineux2[]={0.0, 1.0, 0.0, 0.0};  // Vert
...
glMaterialfv(GL_FRONT, GL_EMISSION, mat_lumineux1);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_lumineux0);
// les polygones dessinés dorénavant renverront de la lumière par leurs faces
// avants (aspect brillant)

DessineMoiUnMouton();     // premier mouton, rouge brillant

glMaterialfv(GL_FRONT, GL_EMISSION, mat_lumineux0);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_lumineux2);
// les polygones dessinés dorénavant renverront de la lumière par leurs faces
// avants (aspect mat)

DessineMoiUnMouton();     // deuxième mouton, vert mat
```

Exemple 2 :

On peut utiliser **glColorMaterial(…)** au lieu de **glMaterial(…).** A ce moment-là, c'est **glColor** qui détermine la couleur du matériel (Color Tracking).

```
glEnable(GL_COLOR_MATERIAL);

glColorMaterial(GL_FRONT, GL_EMISSION);
// Maintenant, glColor change la couleur d'émission.
glColor4f(1.0, 0.0, 0.0, 0.0);
DessineMoiUnMouton();              // premier mouton, brillant rouge
glColor4f(0.0, 0.0, 0.0, 0.0);     // Remettre la couleur d'émission à zéro

glColorMaterial(GL_FRONT, GL_SPECULAR);
// Maintenant glColor ne change plus la couleur d'émission, mais
// la couleur spéculaire.
glColor4f(0.0, 1.0, 0.0, 0.0);
DessineMoiUnAutreMouton();         // deuxième mouton, spéculaire vert
glColor4f(0.0, 0.0, 0.0, 0.0);     // Remettre la couleur spéculaire à zéro

glDisable(GL_COLOR_MATERIAL);
```

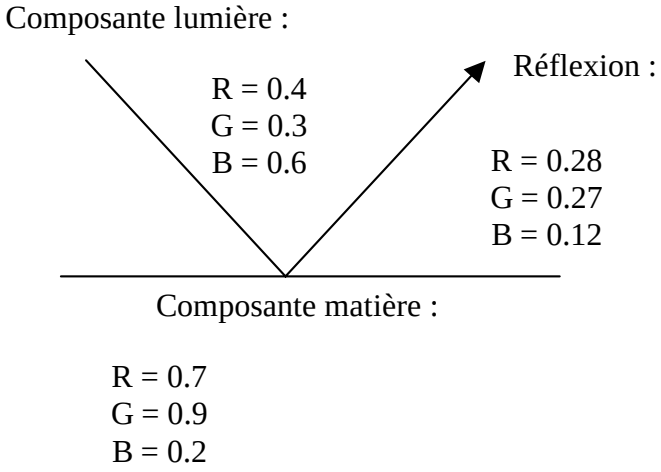
Remarque :

glMaterial*() et **glColorMaterial()** permettent les mêmes opérations. Cela dit, il est conseillé d'utiliser plutôt glColorMaterial() lorsque vous avez besoin de changer un seul paramètre du matériau de tous les objets qui vont être dessinés (comme ici le paramètre GL_EMISSION et GL_SPECULAR des moutons, dans l'exemple 2 uniquement).

void **glColorMaterial**(GLenum face, GLenum mode);

face : GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
mode : GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE (valeur par défaut), GL_SPECULAR, GL_EMISSION

L'effet obtenu est le produit de chaque composante de la lumière avec la composante correspondante de la propriété de la matière.



Choix d'un modèle d'illumination

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

- l'intensité de lumière ambiante globale permettant de fixer l'éclairage minimum au sein de la scène (par défaut : (0.2, 0.2, 0.2, 1.0))
- la position du point de vue (local à la scène, ou à l'infini) utilisé pour l'évaluation des réflexions spéculaires
- si les calculs d'éclairage doivent être effectués différemment pour les deux faces des objets ou non

void **glLightModel** {i f} [v] (GLenum md, TYPE v);

Configuration du modèle d'illumination :

md	***v***
GL_LIGHT_MODEL_AMBIENT (fv ou iv seulement)	Intensité ambiante de la scène, par défaut (0.2, 0.2, 0.2, 1.0)
GL_LIGHT_MODEL_LOCAL_VIEWER	0 : les reflets des spots sont calculés sur l'axe –z ≠ 0 : par rapport à la position de l'œil
GL_LIGHT_MODEL_TWO_SIDE	0 : seule la face « front » est calculée ≠ 0 : les deux faces d'une surface sont calculées

Les normales

Une fois que la source lumineuse est définie, ainsi que les caractéristiques du matériau, il reste encore une chose : l'angle d'incidence de la lumière. Cela ne change pas la couleur d'une surface, mais l'intensité de l'éclairage qu'elle reçoit.

L'angle d'incidence est l'angle entre la normale à une surface et la droite partant de la source lumineuse et passant par le point d'incidence. Il faut impérativement préciser pour chaque point la « normale » de ce point (plus rigoureusement, la normale à la surface dont ce point fait partie).

Le principe est le même que pour glColor, la normale est une des variables d'état d'OpenGL, qui est modifiée par l'une des fonctions suivantes :

```
void glNormal3b( GLbyte nx, GLbyte ny, GLbyte nz );
void glNormal3d( GLdouble nx, GLdouble ny, GLdouble nz );
void glNormal3f( GLfloat nx, GLfloat ny, GLfloat nz );
void glNormal3i( GLint nx, GLint ny, GLint nz );
void glNormal3s( GLshort nx, GLshort ny, GLshort nz );
void glNormal3bv( const GLbyte *v );
[...]
```

Les paramètres correspondent au coordonnées du vecteur normal. ATTENTION : il doit être UNITAIRE (de norme 1).

Il est possible de rendre unitaire automatiquement les vecteurs entrés, mais cela entraîne un **calcul supplémentaire** :

```
glEnable(GL_NORMALIZE);
```

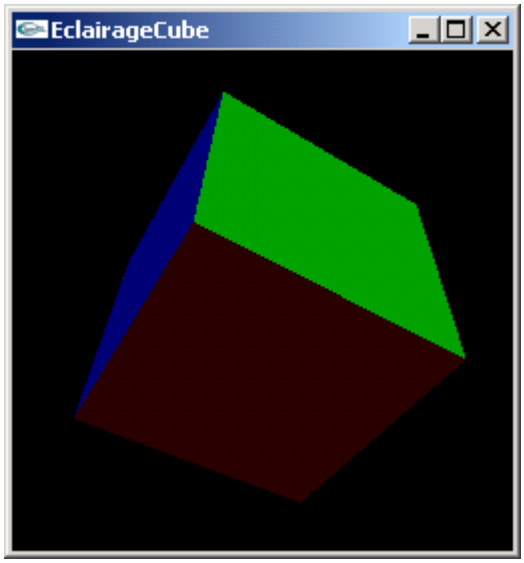
Pour éviter cette perte de temps, qui, pour les scènes assez complexes, peut devenir importante, on peut calculer soi-même la normale avec les deux fonctions ci-dessous qui permettent de calculer un vecteur normal, puis de le réduire à un vecteur unité :

```
// Fonction réduisant un vecteur normal donné par ces trois composantes
// en un vecteur normal unitaire.
void ReduceToUnit(float vector[3])
{
   float length;

   // Calculate the length of the vector
   length = (float)sqrt((vector[0]*vector[0]) +
               (vector[1]*vector[1]) +
               (vector[2]*vector[2]));

   // Keep the program from blowing up by providing an acceptable
   // value for vectors that may calculated too close to zero.
   if(length == 0.0f)
      length = 1.0f;

   // Dividing each element by the length will result in a
   // unit normal vector.
   vector[0] /= length;
   vector[1] /= length;
   vector[2] /= length;
}

// Fonction calculant un vecteur normal à partir de trois sommets d'un polygone
// quelconque.
// Points p1, p2, & p3 specified in counter clock-wise order
void calcNormal(float v[3][3], float out[3])
{
   float v1[3],v2[3];
   static const int x = 0;
   static const int y = 1;
   static const int z = 2;

   // Calculate two vectors from the three points
   v1[x] = v[0][x] - v[1][x];
   v1[y] = v[0][y] - v[1][y];
   v1[z] = v[0][z] - v[1][z];

   v2[x] = v[1][x] - v[2][x];
   v2[y] = v[1][y] - v[2][y];
   v2[z] = v[1][z] - v[2][z];

   // Take the cross product of the two vectors to get
   // the normal vector which will be stored in out
   out[x] = v1[y]*v2[z] - v1[z]*v2[y];
   out[y] = v1[z]*v2[x] - v1[x]*v2[z];
   out[z] = v1[x]*v2[y] - v1[y]*v2[x];

   // Normalize the vector (shorten length to one)
   ReduceToUnit(out);
}
```

Exemple

```
// Fichier EclairageCube.cpp
#include <gl\glut.h>
#include <windows.h>    // Pour la fonction Sleep(50) ; 50 ms

GLfloat total = 0.0;

// Lumière ambiante (blanche de faible intensité)
GLfloat ambientLight[] = {0.2f, 0.2f, 0.2f, 1.0f};

// Lumière diffuse (blanche de plus forte intensité)
GLfloat diffuseLight[] = {0.9f, 0.9f, 0.9f, 1.0f};

// La lumière est à l'infini et vient d'en haut
//                          x     y     z    0->infini
GLfloat positionLight[] = {0.0f, 5.0f, 0.0f, 0.0f};

void SetupRC()
{
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_POLYGON_SMOOTH);
   glClearColor (0.0f, 0.0f, 0.0f, 0.0f);
   // Lumière ambiante générale
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
   // Lumière diffuse
   glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseLight);
   // Lumière LIGHT 0
   glEnable(GL_LIGHT0);
   // Enclencher les lumières
   glEnable(GL_LIGHTING);
}

void Display (void)
{
   GLfloat Rouge[] = {0.8f, 0.0f, 0.0f, 1.0f};
   GLfloat Vert[] = {0.0f, 0.8f, 0.0f, 1.0f};
   GLfloat Bleu[] = {0.0f, 0.0f, 0.8f, 1.0f};
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // Positionner la lumière zéro
   // Cet appel est fait après gluLookAt !!!
   // Donc la lampe n'est pas accrochée à la caméra.
   glLightfv( GL_LIGHT0, GL_POSITION, positionLight);

   glPushMatrix();
      glRotatef((total/30.0f), 0.5f, 0.3f, 0.8f);
      glRotatef((total/50.0f), 0.5f, -0.3f, 0.8f);
      glBegin(GL_QUADS);
         // Les deux faces ci-dessous réfléchissent le rouge
         glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Rouge);
         glNormal3d(0.0,0.0,1.0);
         glVertex3d(-1, 1, 1);
         glVertex3d(-1, -1, 1);
         glVertex3d( 1, -1, 1);
         glVertex3d( 1, 1, 1);
         glNormal3d(0.0,0.0,-1.0);
         glVertex3d( 1, 1, -1);
         glVertex3d( 1, -1, -1);
         glVertex3d( -1, -1, -1);
         glVertex3d( -1, 1, -1);
```

```
        // Les deux faces ci-dessous réfléchissent le vert
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Vert);
        glNormal3d(1.0,0.0,0.0);
        glVertex3d( 1, 1, 1);
        glVertex3d( 1, -1, 1);
        glVertex3d( 1, -1, -1);
        glVertex3d( 1, 1, -1);
        glNormal3d(-1.0,0.0,0.0);
        glVertex3d( -1, 1, 1);
        glVertex3d( -1, 1, -1);
        glVertex3d( -1, -1, -1);
        glVertex3d( -1, -1, 1);
        // Les deux faces ci-dessous réfléchissent le bleu
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Bleu);
        glNormal3d(0.0,-1.0,0.0);
        glVertex3d( -1, -1, 1);
        glVertex3d( -1, -1, -1);
        glVertex3d( 1, -1, -1);
        glVertex3d( 1, -1, 1);
        glNormal3d(0.0,1.0,0.0);
        glVertex3d( -1, 1, 1);
        glVertex3d( 1, 1, 1);
        glVertex3d( 1, 1, -1);
        glVertex3d( -1, 1, -1);
     glEnd();
   glPopMatrix();

   glutSwapBuffers();
}

void Reshape (int width, int height)
{
  if(width == 0) width = 1;
  if(height == 0) height = 1;
   glViewport (0, 0, width, height);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45.0, ((GLdouble) width) / ((GLdouble) height),1.0,10.0);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (0.0f, 0.0f, 5.0f,   // La caméra est placée à 5 unités
              0.0f, 0.0f, 0.0f,   // le long de l'axe Oz
              0.0f, 1.0f, 0.0f);
}

void Animate()
{
   total += 50.0;
   Sleep(50);
   glutPostRedisplay(); // Uniquement pour GLUT
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(250, 250);
  glutCreateWindow("EclairageCube");
```



```
  SetupRC();
  glutReshapeFunc(Reshape);
  glutDisplayFunc(Display);
  glutIdleFunc( Animate );
  glutMainLoop();
  return 0;
}
```

Pour voir l'effet des différents paramètres, voir les très bons exemples de :
www.cs.utah.edu/~narobins/tutors/tutor
et le lightlab de :
www.student.nada.kth.se/~nv91-gta/OpenGL/projects/lightlab/

Notez qu'en ce qui concerne la réflexion sur les matériaux, vous pouvez changer ces propriétés avant de dessiner chaque objet, qui aura alors des caractéristiques lumineuses propres. (le bois ne reflète pas la lumière comme le métal...)

Fog (brouillard)

C'est ce qui simule les effets atmosphériques (je ne vous apprends rien là) comme la fumée, la pollution etc. On s'en sert aussi pour rendre les effets de vision limitée (en même temps, cela permet de virer de la scène une partie des objets 3D trop loin et d'accélérer le rendu).

Le fonctionnement du brouillard

Lorsque le brouillard est activé, les objets qui sont de plus en plus loin commencent à se fondre dans la couleur du brouillard. On peut contrôler aussi la densité du brouillard pour déterminer le rythme avec lequel les objets sont estompés lorsque la distance augmente.

Le brouillard est appliqué à la scène après les transformations matricielles, la lumière, et les textures, donc il affecte les objets transformés.

L'utilisation du brouillard

C'est très simple. On l'active en passant **GL_FOG** comme attribut de **glEnable()** et on détermine la couleur et l'équation de contrôle de la densité du brouillard grâce à :

void **glFogf**(GLenum pname, GLfloat param);
void **glFogi**(GLenum pname, GLint param);
void **glFogfv**(GLenum pname, const GLfloat *params);
void **glFogiv**(GLenum pname, const GLint *params);

avec :

- pname = **GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END,** ou **GL_FOG_COLOR**
- param = la nouvelle valeur.

Exemple

```
glEnable(GL_FOG) ;

GLfloat fogcolor[4] = {0.5, 0.5, 0.5, 1} ;
GLint fogmode = GL_EXP ;
glFogi (GL_FOG_MODE, fogmode) ;
glFogfv(GL_FOG_COLOR, fogcolor) ;
glFogf(GL_FOG_DENSITY, 0.35) ;
glFogf(GL_FOG_START, 1.0) ;         // Si GL_LINEAR
glFogf(GL_FOG_END, 5.0) ;           // Si GL_LINEAR
```

Les équations

Le brouillard se sert d'une variable f variant de 0 (brouillard complet) à 1 (transparent comme de l'eau de roche). Les 3 équations sont (le mode **GL_EXP** est le mode par défaut) :

- GL_LINEAR : $f = (end - z) / (end - start)$ Par défaut : start = 0 ; end = 1
- GL_EXP : $f = e^{-density * z}$ Par défaut : density = 1
- GL_EXP2 : $f = e^{-(density * z)^2}$

Remarque :
Penser à mettre le background à la couleur du brouillard : ainsi, l'effet sera plus frappant. La commande correspondante est :

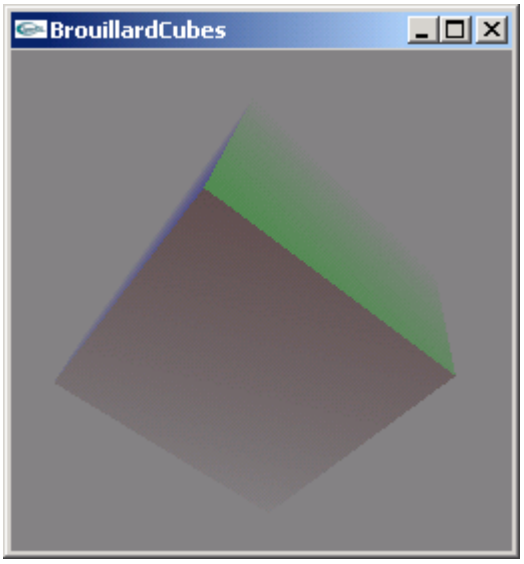
```
glClearColor ( couleur_brouillard [0],
               couleur_brouillard [1],
               couleur_brouillard [2],
               couleur_brouillard [3] );
```

Exemple

```
// Fichier BrouillardCube.cpp
#include <gl\glos.h>
#include <gl\glut.h>
#include <stdio.h>

GLfloat total = 0.0;

// Lumière ambiante
GLfloat ambientLight[] = {0.2f, 0.2f, 0.2f, 1.0f};

// Lumière diffuse
GLfloat diffuseLight[] = {0.9f, 0.9f, 0.9f, 1.0f};

// La lumière est à l'infini et vient d'en haut
//                          x     y     z    0->infini
GLfloat positionLight[] = {0.0f, 5.0f, 0.0f, 0.0f};
```

```
// Valeurs pour le brouillard
#define FOG_COLOR 0.5f, 0.5f, 0.5f, 1.0f
GLfloat fogColor[] = {FOG_COLOR};
GLint fogMode = /*GL_EXP GL_EXP2*/ GL_LINEAR ;

void SetupRC()
{
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_POLYGON_SMOOTH);
//glClearColor (0.0f, 0.0f, 0.0f, 0.0f);
  // Lumière ambiante générale
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
  // Lumière diffuse
   glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseLight);
  // Lumière LIGHT 0
   glEnable(GL_LIGHT0);
  // Enclencher les lumières
   glEnable(GL_LIGHTING);

  // Brouillard
  glEnable(GL_FOG);
  glFogi(GL_FOG_MODE, fogMode);
  printf("Le mode fog est GL_LINEAR\n");
  glFogfv(GL_FOG_COLOR, fogColor);
  glFogf(GL_FOG_DENSITY, 0.3);
  glFogf(GL_FOG_START, 1.0); // Utile si fogMode = GL_LINEAR
  glFogf(GL_FOG_END, 5.0);   // Utile si fogMode = GL_LINEAR
   glClearColor (FOG_COLOR);  // Couleur du fog
}

void Display (void)
{
   GLfloat Rouge[] = {0.8f, 0.0f, 0.0f, 1.0f};
   GLfloat Vert[] = {0.0f, 0.8f, 0.0f, 1.0f};
   GLfloat Bleu[] = {0.0f, 0.0f, 0.8f, 1.0f};
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // Positionner la lumière zéro
  // Cet appel est fait après gluLookAt !!!
  // Donc la lampe n'est pas accrochée à la caméra.
   glLightfv( GL_LIGHT0, GL_POSITION, positionLight);

   glPushMatrix();
     glRotatef((total/30.0f), 0.5f, 0.3f, 0.8f);
     glRotatef((total/50.0f), 0.5f, -0.3f, 0.8f);
     glBegin(GL_QUADS);
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Rouge);
       glNormal3d(0.0,0.0,1.0);
       glVertex3d(-1, 1, 1);
       glVertex3d(-1, -1, 1);
       glVertex3d( 1, -1, 1);
       glVertex3d( 1, 1, 1);
       glNormal3d(0.0,0.0,-1.0);
       glVertex3d( 1, 1, -1);
       glVertex3d( 1, -1, -1);
       glVertex3d( -1, -1, -1);
       glVertex3d( -1, 1, -1);
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Vert);
       glNormal3d(1.0,0.0,0.0);
       glVertex3d( 1, 1, 1);
```

```
        glVertex3d( 1, -1, 1);
        glVertex3d( 1, -1, -1);
        glVertex3d( 1, 1, -1);
        glNormal3d(-1.0,0.0,0.0);
        glVertex3d( -1, 1, 1);
        glVertex3d( -1, 1, -1);
        glVertex3d( -1, -1, -1);
        glVertex3d( -1, -1, 1);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Bleu);
        glNormal3d(0.0,-1.0,0.0);
        glVertex3d( -1, -1, 1);
        glVertex3d( -1, -1, -1);
        glVertex3d( 1, -1, -1);
        glVertex3d( 1, -1, 1);
        glNormal3d(0.0,1.0,0.0);
        glVertex3d( -1, 1, 1);
        glVertex3d( 1, 1, 1);
        glVertex3d( 1, 1, -1);
        glVertex3d( -1, 1, -1);
     glEnd();
   glPopMatrix();

   glutSwapBuffers();
}

void Reshape (int width, int height)
{
  if(width == 0) width = 1;
  if(height == 0) height = 1;
   glViewport (0, 0, width, height);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45.0, ((GLdouble) width) / ((GLdouble) height),1.0,10.0);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (0.0f, 0.0f, 5.0f,   // La caméra est placée à 5 unités
              0.0f, 0.0f, 0.0f,   // le long de l'axe Oz
              0.0f, 1.0f, 0.0f);
}

void Animate()
{
   total += 50.0;
   Sleep(50);
   glutPostRedisplay(); // Uniquement pour GLUT
}

void Keyboard(unsigned char key, int /*x*/, int /*y*/)
{
  switch(key)
  {
    case 'f' :
    case 'F' : if(fogMode == GL_EXP)
               {
                 fogMode = GL_EXP2;
                 printf("Le mode fog est GL_EXP2\n");
               }
```



```
                else if(fogMode == GL_EXP2)
                {
                  fogMode = GL_LINEAR;
                  printf("Le mode fog est GL_LINEAR\n");
                }
                else if(fogMode == GL_LINEAR)
                {
                  fogMode = GL_EXP;
                  printf("Le mode fog est GL_EXP\n");
                }
                glFogi(GL_FOG_MODE, fogMode);
                glutPostRedisplay();
                break;
    case 27 :   exit(0);
                break;
  }
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(250, 250);
  glutCreateWindow("BrouillardCubes");

  SetupRC();
  glutReshapeFunc(Reshape);
  glutDisplayFunc(Display);
  glutKeyboardFunc(Keyboard);
  glutIdleFunc( Animate );
  glutMainLoop();
  return 0;
}
```

Transparence et alpha-blending

Plusieurs fois déjà des couleurs ont été présentées sous forme RGBA : rouge, vert, bleu et alpha. Si les trois premières coordonnées sont évidentes, la quatrième quant à elle sert à indiquer la « transparence ».

Activation

Si pour l'instant rien n'a été dit sur les valeurs de l'alpha, c'est simplement parce que par défaut OpenGL s'en fiche.

Pour activer le canal alpha, il faut appeler comme d'habitude **glEnable** :

```
glEnable(GL_BLEND) ;
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) ;  // expliqué après
```

Ensuite il suffit de changer la valeur de A suivant la transparence (RGB représente alors la couleur du fragment, A représente son opacité : 0 → transparent, 1 → opaque).

Exemple

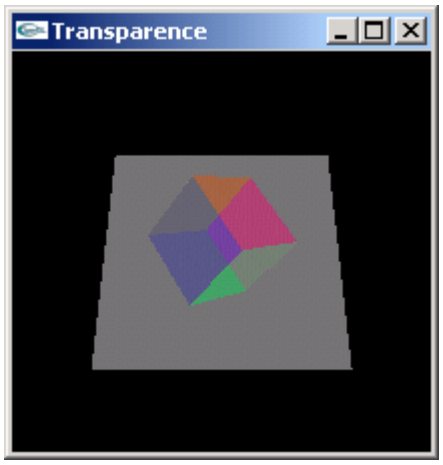
Il s'agit, toujours du même cube, mais dont les côtés sont maintenant transparents.

```
void Display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   drawFloor();
   glPushMatrix();
     //initialisation de la transparence
     glEnable(GL_BLEND);
     glDepthMask(GL_FALSE);   // Tampon de profondeur en lecture seule
     //la couleur de l'objet va être
     // (1-alpha_de_l_objet) * couleur du fond et (le_reste * couleur originale)
     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
     glTranslatef(0.0f, -2.0f, 5.0f);
     glRotatef((80.0f), -0.5f, 0.3f, -0.8f);
     glRotatef((10.0f), 0.5f, -0.3f, 0.8f);
     glCallList(cube);
     glDepthMask(GL_TRUE);    // Tampon de profondeur en lecture / écriture
     //on désactive la transparence
     glDisable(GL_BLEND);
   glPopMatrix();
   glutSwapBuffers();
}
```

```
void drawFloor()
{
  GLfloat alpha = 0.75f;                  // Ne change rien dans cet exemple
  glColor4f(0.5f, 0.5f, 0.5f, alpha);
  glBegin(GL_QUADS);
    glVertex3f(-4.0,  4.0, 0.0);
    glVertex3f( 4.0,  4.0, 0.0);
    glVertex3f( 4.0, -4.0, 0.0);
    glVertex3f(-4.0, -4.0, 0.0);
  glEnd();
}
```

```
void SetupRC()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f ); // Ce sont les valeurs par défaut.
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_POLYGON_SMOOTH);
   SetLight();
}

void Make_CallListes()
{
  //C'est la partie la plus importante
  //Les couleurs sont définies avec un alpha de 0.25
  //qui va servir dans le calcul de la transparence
  //Rappel : 0 = transparent, 1 = opaque
  GLfloat alpha = 0.25f;                        // Si 1, pas de transparence
  GLfloat Rouge[] = {0.8f, 0.0f, 0.0f, alpha };
  GLfloat Vert[]  = {0.0f, 0.8f, 0.0f, alpha };
  GLfloat Bleu[]  = {0.0f, 0.0f, 0.8f, alpha };
  GLfloat Jaune[]  = {0.8f, 0.8f, 0.0f, alpha };
  GLfloat Rose[]  = {0.8f, 0.0f, 0.8f, alpha };
  GLfloatCyan[]  = {0.0f, 0.8f, 0.8f, alpha };
```

```
   //on se contente de dessiner le cube en utilisant cependant la fonction
   //glColor4f() au lieu de glColor3f() car la composante alpha compte cette
  //fois-ci
   cube = glGenLists(1);
   glNewList(cube, GL_COMPILE);
     glBegin(GL_QUADS);
        glColor4fv(Jaune);
        glNormal3d(-1.0,0.0,0.0);
        glVertex3d( -1,  1,  1);
        glVertex3d( -1,  1, -1);
        glVertex3d( -1, -1, -1);
        glVertex3d( -1, -1,  1);
        glColor4fv(Rose);
        glNormal3d(0.0,1.0,0.0);
        glVertex3d( -1, 1,  1);
        glVertex3d(  1, 1,  1);
        glVertex3d(  1, 1, -1);
        glVertex3d( -1, 1, -1);
        glColor4fv(Bleu);
        glNormal3d(0.0,-1.0,0.0);
        glVertex3d( -1, -1,  1);
        glVertex3d( -1, -1, -1);
        glVertex3d(  1, -1, -1);
        glVertex3d(  1, -1,  1);
        glColor4fv(Cyan);
        glNormal3d(0.0,0.0,-1.0);
        glVertex3d(  1,  1, -1);
        glVertex3d(  1, -1, -1);
        glVertex3d( -1, -1, -1);
        glVertex3d( -1,  1, -1);
        glColor4fv(Vert);
        glNormal3d(1.0,0.0,0.0);
        glVertex3d( 1,  1,  1);
        glVertex3d( 1, -1,  1);
        glVertex3d( 1, -1, -1);
        glVertex3d( 1,  1, -1);
        glColor4fv(Rouge);
        glNormal3d(0.0,0.0,1.0);
        glVertex3d(-1,  1, 1);
        glVertex3d(-1, -1, 1);
        glVertex3d( 1, -1, 1);
        glVertex3d( 1,  1, 1);
     glEnd();
   glEndList();
}
int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(largeur, hauteur);
  glutCreateWindow("Transparence");
  SetupRC();
  Make_CallListes();
  glutReshapeFunc(Reshape);//fonction gérant le redimensionnement de la fenêtre
  glutDisplayFunc(Display);
  glutMainLoop();          //boucle principale
  return 0;
}
```



Paramètres

Le paramétrage principal est effectué par la fonction :

```
void glBlendFunc( GLenum sfactor, GLenum dfactor );
```

Si la fonction s'appelle « blend » (« mélanger » en anglais, ici en parlant des couleurs) et pas « transparency », c'est que la transparence n'est qu'une application de l'alpha-blending. C'est de loin la principale. Néanmoins ce que fait réellement l'alpha-blending, c'est mélanger la couleur déjà affichée avec une nouvelle.

Le mélange est précisé par un appel à **glBlendFunc**, avec :

- Le premier paramètre indiquant par quelle valeur multiplier (canal par canal) la nouvelle couleur (source).
- Le second paramètre indiquant par quelle valeur multiplier (canal par canal) la couleur déjà affichée (destination).
- La nouvelle couleur affichée est la somme de ces deux multiplications.

L'indice « s » indique que la nouvelle couleur est concernée (source), « d » qu'il s'agit de la couleur déjà affichée (destination).

La couleur affichée est (où C doit être remplacé par R, G, B, A) :

$$\mathbf{C = C_s S + C_d D}$$

S et D étant paramétrés respectivement par sfactor et dfactor.

Principales valeurs de S en fonction de sfactor :

Valeur de sfactor	Valeur de S
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_DST_COLOR	(R_d,G_d,B_d,A_d)
GL_ONE_MINUS_DST_COLOR	(1,1,1,1) - (R_d,G_d,B_d,A_d)
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	(1,1,1,1) - (A_s,A_s,A_s,A_s)
GL_DST_ALPHA	(A_d,A_d,A_d,A_d)
GL_ONE_MINUS_DST_ALPHA	(1,1,1,1) - (A_d,A_d,A_d,A_d)
GL_SRC_ALPHA_SATURATE	(f,f,f,1) avec f = min $(A_s, 1 - A_d)$

Principales valeurs de D en fonction de dfactor :

Valeur de dfactor	Valeur de D
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	(R_s,G_s,B_s,A_s)
GL_ONE_MINUS_SRC_COLOR	(1,1,1,1) - (R_s,G_s,B_s,A_s)

GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1,1) - (A_s,A_s,A_s,A_s)$
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	$(1,1,1,1) - (A_d,A_d,A_d,A_d)$
GL_SRC_ALPHA_SATURATE	(f,f,f,1) avec f = min (A_s, 1 - A_d)

Les paramètres les plus courants sont (**GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**).

Cela représente bien la transparence codée dans le canal Alpha : la couleur affichée est multipliée par la valeur de transparence (si 0, complètement transparent, si 1 opaque), et la couleur en dessous est multipliée par le complément.

Ordre des primitives et depth buffer

Il faut se souvenir qu'en utilisant l'alpha-blended pour la transparence, le test du depth buffer peut interférer avec l'effet que l'on désire obtenir. Pour être sûr que les lignes et les polygones transparents seront dessinés correctement, il faut toujours les dessiner de l'arrière vers l'avant et les dessiner après tous les objets solides (non transparents).

Pour plus de détail, on peut utiliser la fonction **glDepthMask(…)** avec GL_FALSE ou GL_TRUE pour activer le tampon de profondeur en lecture seule (GL_FALSE).

Textures

Jusqu'ici, les surfaces étaient simplement coloriées, ce qui ne permet pas d'avoir des rendus très réalistes. Les textures améliorent considérablement le rendu. Le principe est de remplir une surface avec une image bitmap correspondant à une photo de la surface. Les possibilités qu'offrent OpenGL dans le domaine des textures sont tellement vastes que les applications de cet outils sont quasi infinies. Donc ici, on ne va voir que les bases (ce qui permettra quand même de faire des trucs bien sympa), c'est-à-dire comment déclarer une texture à OpenGL, l'appliquer sur un polygone grâce aux TexCoord, et aussi quelques outils qui vont avec tel que le filtering, la répétition.

Création de la texture

Commençons par le commencement, c'est-à-dire par activer la gestion des textures. Vous aurez deviné qu'il suffit de faire un appel à **glEnable()** avec le paramètre approprié. Rajoutez donc dans notre petite **SetupRC()** la ligne suivante :

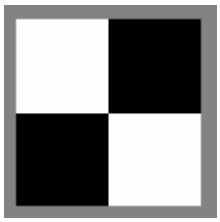
```
glEnable(GL_TEXTURE_2D);
```

Cela active la gestion des textures 2D. Nous devons maintenant créer un nouvel objet texture. Pour nous, ce sera juste un numéro, mais c'est ce numéro qui nous permettra de spécifier à OpenGL tous les paramètre de la texture, la texture elle-même, et aussi de lui dire quelle texture on veut mapper (appliquer). Pour cela il faut utiliser **glGenTextures(),** qui nous renvoie autant de numéros que de textures demandées. Ici on ne veut qu'une seule texture, donc on va récupérer un seul numéro, à bien retenir.

Ensuite, nous décidons quelle est la texture courante grâce à **glBindTexture().** La texture courante, c'est celle sur laquelle OpenGL va effectuer les modifications lorsqu'on lui demandera, et c'est aussi la texture qui sera appliquée sur les objets. Donc si vous avez des objets avec des textures différentes, il suffira d'appeler **glBindTexture()** avec le bon numéro de texture juste avant de dessiner l'objet sur lequel devra être appliquée la texture.

Et enfin, nous allons transmettre à OpenGL toutes les caractéristiques de la texture : largeur, hauteur, format, etc... et bien sûr l'image elle-même, grâce à **glTexImage2D().** Pour l'instant, pour simplifier les choses, on va appliquer à notre objet une texture de 2 pixels par 2 pixels, en forme d'échiquier. Voilà donc tout ce que nous devons écrire :

```
GLubyte Texture[16] =     //Image (2x2), 4 octets par pixel : RGBA
{                         //  s, t
  0,0,0,0,                // (0, 0) noir
  0xFF,0xFF,0xFF,0xFF,    // (1, 0) blanc
  0xFF,0xFF,0xFF,0xFF,    // (0, 1) blanc
  0,0,0,0                 // (1, 1) noir
};

GLuint Nom;

void SetupRC()
{
  glClearColor(.5,.5,.5,0);              //Change la couleur du fond
  glEnable(GL_DEPTH_TEST);               //Active le depth test

  glEnable(GL_TEXTURE_2D);               //Active le texturing
  glGenTextures(1,&Nom);                 //Génère un n° de texture
  glBindTexture(GL_TEXTURE_2D,Nom);      //Sélectionne ce n°
  glTexImage2D (
                GL_TEXTURE_2D,           //Type : texture 2D
                0,                       //Mipmap : aucun
                4,                       //Couleurs : 4
                2,                       //Largeur : 2 + 2b
                2,                       //Hauteur : 2 + 2b
                0,                       //Largeur du bord : 0 ou 1 (b)
                GL_RGBA,                 //Format : RGBA
                GL_UNSIGNED_BYTE,        //Type du tableau
                Texture                  //Adresse de l'image
                );

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
}
```



On appelle **glGenTextures()** avec le nombre de textures voulues et le tableau dans lequel il doit les renvoyer (ici il n'y en a qu'une donc autant mettre &Nom). J'appelle ensuite **glBindTexture()** pour lui dire avec quelle texture je vais travailler. Et enfin, **glTexImage2D()**, pour lui transmettre toutes les informations dont il a besoin. Les commentaires vous disent déjà à quoi correspondent les paramètres, mais nous allons tout de même insister sur 2 ou 3 choses :

- Pour ce qui est du mipmap, on verra cela plus bas. On met donc ce paramètre à 0.
- Le 3e paramètre c'est le nombre de composantes de couleurs par pixel. Ici il y en a 4 : R,G, B et A (pour Rouge, Vert, Bleu et Alpha). On peux en avoir de 1 à 4.
- Ensuite viennent la largeur et la hauteur. Elles ne sont pas forcément égales, mais doivent être **impérativement de la forme 2^n (2^n + 2*b)**. Si une de ces valeurs n'est pas une puissance de 2, ça ne marchera pas. Si vous êtes obligé de bosser sur des textures bâtardes,

utilisez **gluScaleImage()** pour les mettre à l'échelle (voyez avec votre compilateur ou le Reference Manual pour en savoir plus sur cette fonction).

- Le paramètre suivant est la largeur du bord (valeur b = 0 ou 1) de la texture. En effet, on peut rajouter une bordure d'un pixel autour de l'image, et d'une couleur unique. Cela peut servir pour éviter d'avoir des résultat bizarre sur les bords avec le filtering, mais on ne l'utilise pas souvent (utile pour des mosaïques avec plusieurs images différentes).
- Vient ensuite le format de stockage. Ici, j'ai rangé les octets de ma texture dans le format classique RGBA, donc je dis à OpenGL de les lire suivant le format RGBA. On peut mettre GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, et GL_LUMINANCE_ALPHA. On peut s'amuser à changer ce paramètre pour voir ce que ça donne.
- Le 8ème paramètre définit sous quel type sont enregistrées les composantes. Ici, ma texture est un tableau de GLubyte (c'est-à-dire unsigned char), donc je dit à OpenGL de les lire en tant que GLubyte. Ce paramètre peut être GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, ou GL_FLOAT.
- Et enfin, on donne à OpenGL l'endroit où est stockée l'image, histoire qu'il puisse la mapper.
- Les deux dernières lignes seront discutées dans le paragraphe **Filtering** ci-dessous.

Remarque :

Il est parfois utile d'utiliser un appel à :

```
glPixelStorei(GL_UNPACK_ALIGNMENT,1);
```

pour définir le mode de stockage des pixels.

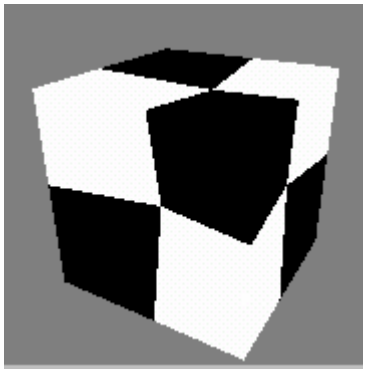
Coordonnées de textures (TexCoord)

Notre texture est maintenant chargée et prête à être appliquée : on n'a plus qu'à dessiner un objet sur lequel l'appliquer. Prenons l'exemple d'un cube :

```
void Draw()
{
  glBegin(GL_QUADS);
  //Et c'est parti pour le cube !
    glTexCoord2i(0,0);glVertex3i(-1,-1,-1);
    glTexCoord2i(1,0);glVertex3i(+1,-1,-1);
    glTexCoord2i(1,1);glVertex3i(+1,+1,-1);
    glTexCoord2i(0,1);glVertex3i(-1,+1,-1);
    //1 face

    glTexCoord2i(0,0);glVertex3i(-1,-1,+1);
    glTexCoord2i(1,0);glVertex3i(+1,-1,+1);
    glTexCoord2i(1,1);glVertex3i(+1,+1,+1);
    glTexCoord2i(0,1);glVertex3i(-1,+1,+1);
    //2 faces

    glTexCoord2i(0,0);glVertex3i(+1,-1,-1);
    glTexCoord2i(1,0);glVertex3i(+1,-1,+1);
    glTexCoord2i(1,1);glVertex3i(+1,+1,+1);
    glTexCoord2i(0,1);glVertex3i(+1,+1,-1);
    //3 faces
```

```
    glTexCoord2i(0,0);glVertex3i(-1,-1,-1);
    glTexCoord2i(1,0);glVertex3i(-1,-1,+1);
    glTexCoord2i(1,1);glVertex3i(-1,+1,+1);
    glTexCoord2i(0,1);glVertex3i(-1,+1,-1);
    //4 faces

    glTexCoord2i(1,0);glVertex3i(-1,+1,-1);
    glTexCoord2i(1,1);glVertex3i(+1,+1,-1);
    glTexCoord2i(0,1);glVertex3i(+1,+1,+1);
    glTexCoord2i(0,0);glVertex3i(-1,+1,+1);
    //5 faces

    glTexCoord2i(1,0);glVertex3i(-1,-1,+1);
    glTexCoord2i(1,1);glVertex3i(+1,-1,+1);
    glTexCoord2i(0,1);glVertex3i(+1,-1,-1);
    glTexCoord2i(0,0);glVertex3i(-1,-1,-1);
    //6 faces

  glEnd();

  glutSwapBuffers();
}
```

Si vous lancez le programme maintenant, ça va donner une magnifique image de cube mappé avec une texture d'échiquier, comme ça (la forme exacte va dépendre de la carte graphique) :

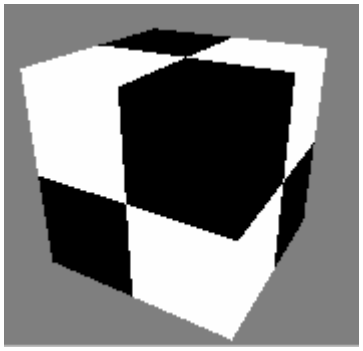


En fait, on se rend compte que pour que l'effet de mapping soit réaliste, il faut tenir compte de la profondeur.

On peut dire à OpenGL d'effectuer une correction de perspective sur les textures (et par la même occasion sur les couleurs) en ajoutant dans **SetupRC()** la ligne suivante :

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
```

Et maintenant, on obtient l'image ci-contre (avec n'importe quelle carte graphique) :

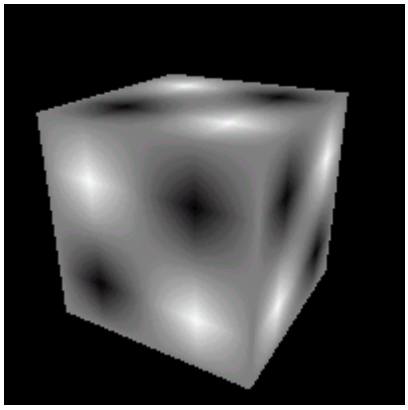


Voyons maintenant le contenu de notre fonction **Draw()**. Ici on découvre une nouvelle fonction :

void **glTexCoord**{1234}{sifd}[v]();
par exemple :
void **glTexCoord2i**(GLint s, GLint t);
ou bien :
void **glTexCoord2f**(GLfloat s, GLfloat t);

C'est elle qui nous permet de dire à OpenGL quel morceau de la texture est attaché à quel sommet. Ici on a une texture 2 dimensions, donc on utilise **glTexCoord2*().** On lui passe en paramètre 2 valeurs, qui sont les coordonnées du point de la texture à attacher au sommet qui va suivre. Le point **(0,0)** correspond au coin **inférieur gauche** de la texture, et **(1,1)** au coin **supérieur droit**. Mais on peut aussi choisir n'importe quel point de la texture (ou texel) entre 0 et 1). Par exemple si on avait voulu ne mettre que du noir, il aurait suffit de mapper la partie inférieure gauche de la texture, donc de remplacer les 1 par des 0.5 (et aussi d'utiliser **glTexCoord2f(),** parce que sinon les TexCoord auraient été transformées en int).

Filtering



Imaginez que vous êtes dans un simulateur de vol dernier cri : si vous regardez le sol, et même si vous êtes très près, normalement vous ne distinguez pas les pixels qui composent la texture. Alors que si vous jouez à un vieux jeu du genre de Duke Nukem ou Quake, si vous vous collez à un mur vous verrez distinctement les pixels de la texture, comme dans notre exemple. Mais dans Quake 3, ou dans un simulateur de vol, vous ne verrez jamais la séparation nette entre les pixels : il y aura toujours un dégradé entre le pixel et son voisin. Cette petite révolution du monde de la 3D s'appelle le **bilinear filtering**, et est parfaitement gérée par OpenGL. Vous vous rappelez les deux lignes qui n'ont pas été expliquées tout à l'heure ?

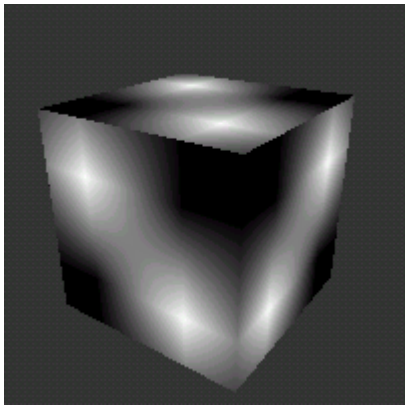
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Et bien elles disaient à OpenGL de ne **pas** faire de bilinear filtering, mais de prendre le texel le plus proche (d'où le terme **GL_NEAREST**) lorsqu'il dessine le pixel d'un polygone. Changez donc ce paramètre en **GL_LINEAR** dans les deux lignes, on obtient le résultat ci-dessus :

On voit bien qu'il y a un dégradé entre chaque texel, d'ailleurs on le voit un peu trop bien, puisque là on a une texture de 4 pixels (2x2), ça fait un peu juste. Voyons plutôt cette fonction **glTexParameter{if}[v]()** : avec elle on règle le filtering, mais en 2 fois. Pourquoi ? Et bien en fait il y a 2 filtering : un lorsqu'il y a magnification (**GL_TEXTURE_MAG_FILTER**), c'est-à-dire lorsqu'un texel s'étend sur plusieurs pixels, comme ici, et un lorsqu'il y a minification (**GL_TEXTURE_MIN_FILTER**), c'est-à-dire lorsqu'un pixel contient plusieurs texels. Dans le cas de la magnification, on l'a vu, le pixel est le résultat de la moyenne des texels environnants. Dans le cas de la minification, s'il y a filtering, le pixel est le résultat de la moyenne des texels contenus.

Répétition

Mais il y a quand même un truc bizarre : sur les bords des faces, la texture subi un dégradé vers le gris, alors que normalement elle devrait subir un dégradé vers la couleur du pixel de la texture (blanc ou noir). Pourquoi ? Et bien parce qu'OpenGL est configuré pour répéter la texture. C'est-à-dire que si on avait mis une valeur n>1 pour les texcoords, il aurait répété n fois la texture (essayez pour voir). Et lorsqu'il est au bord, il fait un filtering avec le bord opposé de la texture, ce qui conduit à rendre tous les bords gris !



En mode **GL_CLAMP** ("bridé"), les coordonnées sont ramenées entre 0 et 1 et sur les côtés de la texture, le filtering est fait entre la texture et le bord (celui que l'on spécifie dans **glTexImage2D()**). Pour choisir le mode de texturing , il faut appeler **glTexParameter()** avec comme premier paramètre **GL_TEXTURE_2D**, comme second paramètre **GL_TEXTURE_WRAP_S** (pour affecter les coordonnées horizontales) ou **GL_TEXTURE_WRAP_T** (pour les coordonnées verticales), et comme troisième paramètre **GL_CLAMP** (pour bridé) ou **GL_REPEAT** (pour répété). Et si on est en mode répété, on obtient bien le résultat que voici. Ici, comme la couleur du bord est (0,0,0,0) par défaut, il y a un dégradé vers le noir sur les bords des faces.

Vous vous demander peut-être comment ne **pas** faire de filtering sur les bord ? Et bien on ne peut pas. le meilleur moyen est encore de se mettre en **GL_REPEAT** de faire des textures dont les bords opposés sont semblables, ou d'utiliser le bord en mode **GL_CLAMP** si les couleurs ne varient pas trop. Ah oui j'oubliais : pour changer la couleur du bord, on appelle **glTexParameter{if}v()** avec **GL_TEXTURE_BORDER_COLOR** comme 2ème paramètre et un tableau de 4 valeurs (RGBA) comme 3ème paramètre.

Résumé

Dans ce cours, les routines de chargement d'images en mémoire ne seront pas expliquées. Ce dont a besoin OpenGL, c'est d'une image non compressée chargée sous la forme d'un tableau en mémoire. Il existe de nombreuses routines toutes faites pour charger les principaux formats d'images (BMP, PCX, GIF, PNG…).

La première chose à faire, c'est d'activer les textures :

```
glEnable(GL_TEXTURE_2D) ;
```

Spécifier l'image

Pour spécifier la texture à utiliser, appeler :

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

Avec :

Target	GL_TEXTURE_2D
Level	Indique le niveau de mipmapping, expliqué plus loin dans ce cours
Components	Nombre de composantes dans la texture (1, 2, 3 ou 4). Généralement 3 (pour RGB) ou 4 (pour RGBA).
Width	Largeur de l'image. Doit être $2^n + 2b$ (dépend de bordure).
Height	Hauteur de l'image. Doit être $2^m + 2b$ (dépend de bordure).
Bordure	Indique la largeur de la bordure : 0 ou 1 (b)
Format	Codage des couleurs : GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR, GL_BGRA, GL_LUMINANCE, ou GL_LUMINANCE_ALPHA
Type	Type du tableau : GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, ou GL_FLOAT
Pixel	Pointeur vers l'image.

Paramétrage

Application de la texture

Pour changer un paramètre, il faut appeler :

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

avec *target* = **GL_TEXTURE_2D**, *pname* l'un des noms de paramètres, et **param** la nouvelle valeur.

Les paramètres d'application d'une texture sont :

pname	***Type***	***param***	***Signification***
GL_TEXTURE_WRAP_S	Entier	GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT	Indique ce qui se passe si une coordonnée s sort de l'intervalle [0..1]. Soit la texture est répétée (mosaïque), soit seulement le bord est répété.
GL_TEXTURE_WRAP_T	Entier	Idem	Idem pour t
GL_TEXTURE_MIN_FILTER	Entier	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR	Indique la qualité de la réduction d'une texture. LINEAR est meilleur que NEAREST, mais plus lent.
GL_TEXTURE_MAG_FILTER	Entier	GL_NEAREST, GL_LINEAR	Idem en agrandissement.
GL_TEXTURE_BORDER_COLOR	v	Adresse d'un tableau contenant les valeurs (R, G, B, A).	Spécifie une couleur de bord pour une texture sans bord.

Il existe d'autres paramètres, mais ceux-ci sont de loin les plus importants.

glColor

La couleur courante (spécifiée par glColor), joue un rôle différent suivant les paramètres passés à :

```
void glTexEnvi( GLenum target, GLenum pname, GLint param );
```

avec

- target = GL_TEXTURE_ENV
- pname = GL_TEXTURE_ENV_MODE ou GL_TEXTURE_ENV_COLOR
- param = 1) GL_REPLACE, GL_MODULATE, GL_DECAL, ou GL_BLEND
  2) L'adresse d'un tableau contenant les valeurs (R, G, B, A), seulement avec GL_BLEND.

Les modes les plus courants sont :

- **GL_REPLACE**, la couleur courante n'intervient pas.
- **GL_MODULATE**, c'est la valeur par défaut, la couleur courante multiplie la texture. Il est alors possible de changer très facilement l'intensité d'une texture, en dessinant d'abord la surface en blanc, par exemple, lequel va être atténué selon la position de la lumière, puis en plaquant la texture. L'intensité de cette dernière va varier selon l'éclairage.

Les « épingles »

Pour poser une épingle, avant un vertex il faut appeler :

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

ou l'une de ses variantes.

Les coordonnées (s, t) sont les coordonnées du point de la texture à attacher au sommet qui va suivre. En OpenGL, le repère d'une texture fait toujours [0..1] sur [0..1], mais s et t peuvent sortir des ces intervalles, l'effet dépend alors du paramétrage choisi. Le point (0,0) correspond au coin inférieur gauche de la texture, et (1,1) au coin supérieur droit. Mais on peut aussi choisir n'importe quel point de la texture (ou texel) entre 0 et 1.

- Lorsque s et t sont compris entre 0 et 1, OpenGL va positionner les épingles directement sur la texture. Vous pourrez ainsi utiliser seulement une petite partie de celle-ci. Si la surface sur laquelle vous posez la texture est très grande, il va étirer la texture pour la placer sur votre surface.
- A l'inverse, si s et t (ou seulement un seul) sont supérieurs à 1, la texture va être répétée (avec GL_REPEAT).

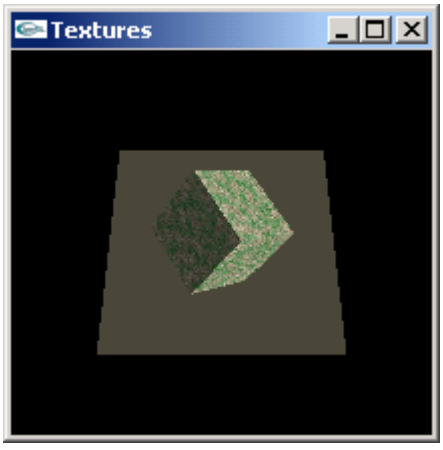
Exemple

Voici un cube texturé. Pour simplifier (et garantir la portabilité), le format d'image est ici le « Raw », c'est-à-dire l'image seule. Paint Shop Pro gère très bien ce format. Le problème est que la

taille de l'image n'est pas incluse dans ce fichier. L'image a une taille de 128*128 et est au format RGB (24 bits = 3 octets).

```
void SetLight()
{
   GLfloat ambientProperties[] = {0.2, 0.2f, 0.2f, 1.0f};
   GLfloat diffuseProperties[] = {1.0f, 1.0f, 1.0f, 1.0f};
   GLfloat positionProperties[] = {5.0f, 0.0f, 0.0f, 0.0f};
   glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
   glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
   glLightfv( GL_LIGHT0, GL_POSITION, positionProperties);
   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHTING);
}
```

```
void SetupRC()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f ); // Ce sont les valeurs par défaut.
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_POLYGON_SMOOTH);
   SetLight();
}

void Make_CallListes()
{
   GLfloat  Blanc[]  = {1.0f, 1.0f, 1.0f, 1.0f};

   cube = glGenLists(1);
   glNewList(cube, GL_COMPILE);
      glBegin(GL_QUADS);
         glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Blanc);
         glNormal3d(0.0,0.0,1.0);
         // On spécifie la position des épingles
         // (on tourne : une seule coordonnée change à la fois)
         glTexCoord2d(0,1);glVertex3d(-1,  1, 1);
         glTexCoord2d(0,0);glVertex3d(-1, -1, 1);
         glTexCoord2d(1,0);glVertex3d( 1, -1, 1);
         glTexCoord2d(1,1);glVertex3d( 1,  1, 1);
         glNormal3d(-1.0,0.0,0.0);
            //idem
         glTexCoord2d(1,1);glVertex3d( -1,  1,  1);
         glTexCoord2d(1,0);glVertex3d( -1,  1, -1);
         glTexCoord2d(0,0);glVertex3d( -1, -1, -1);
         glTexCoord2d(0,1);glVertex3d( -1, -1,  1);
            //idem
         glNormal3d(1.0,0.0,0.0);
         glTexCoord2d(1,1);glVertex3d( 1,  1,  1);
         glTexCoord2d(0,1);glVertex3d( 1, -1,  1);
         glTexCoord2d(0,0);glVertex3d( 1, -1, -1);
         glTexCoord2d(1,0);glVertex3d( 1,  1, -1);
         glNormal3d(0.0,1.0,0.0);
            //idem
         glTexCoord2d(0,1);glVertex3d( -1, 1,  1);
         glTexCoord2d(1,1);glVertex3d(  1, 1,  1);
         glTexCoord2d(1,0);glVertex3d(  1, 1, -1);
         glTexCoord2d(0,0);glVertex3d( -1, 1, -1);
         glNormal3d(0.0,-1.0,0.0);
            //idem
         glTexCoord2d(0,1);glVertex3d( -1, -1,  1);
         glTexCoord2d(0,0);glVertex3d( -1, -1, -1);
```



```
         glTexCoord2d(1,0);glVertex3d(  1, -1, -1);
         glTexCoord2d(1,1);glVertex3d(  1, -1,  1);
         glNormal3d(0.0,0.0,-1.0);
            //idem
         glTexCoord2d(1,1);glVertex3d(  1,  1, -1);
         glTexCoord2d(1,0);glVertex3d(  1, -1, -1);
         glTexCoord2d(0,0);glVertex3d( -1, -1, -1);
         glTexCoord2d(0,1);glVertex3d( -1,  1, -1);
      glEnd();
   glEndList();
}

void ChargerTextures()
{
   //Taille = 128 * 128. RGB 3 octets
   char buffer[128*128*3];
   FILE *f = fopen("herbe.raw", "rb");
   if (f)
   {
      fread(buffer, 128*128*3, 1, f);
      fclose(f);
      printf("texture chargée");

      //On spécifie quelle texture, sa taille, son type ...
      glTexImage2D(GL_TEXTURE_2D, 0, 3, 128, 128, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, buffer);

      // Cela définit la façon dont la texture est appliquée.
      // LINEAR est le meilleur, mais le + lent.
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

      //On répète la texture si s et t sortent des bornes [0,1]
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

      //ouf, ça y est !!!
      glEnable(GL_TEXTURE_2D);
   }
   else
      printf("Il y a un problème de fichier!!!!\n");
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(100, 100);
  glutInitWindowSize(largeur, hauteur);
  glutCreateWindow("Textures");

  SetupRC();
  ChargerTextures();
  Make_CallListes();
  glutReshapeFunc(Reshape);//fonction gérant le redimensionnement de la fenêtre
  glutDisplayFunc(Display);
  glutMainLoop();          //boucle principale
  return 0;
}
```

Autres Formats

Il existe 2 fichiers ZIP contenant les sources de petites librairies qui permettent de charger des fichiers TGA et BMP (**TGA.ZIP** et **BMP.ZIP**).

MipMapping

Le MipMapping consiste à avoir la même texture à plusieurs échelles, de façon à limiter les transferts mémoires lors du plaquage de texture (inutile de transférer une texture 256x256 si l'objet dessiné est très loin dans le décor et ne fait plus que du 5x5).

En théorie, le programmeur devrait avoir 3 ou 4 fois la même texture, mais avec une résolution différente. Il faut alors entrer toutes les textures à chaque fois, en changeant le paramètre *level* de **glTextImage2D** (sachant que 0 est l'image la plus grande, et les suivantes sont ses réductions). Lorsque la texture est appliquée, OpenGL choisi soit l'image de taille la plus proche de la surface à afficher (NEAREST_MIPMAP), soit fait une sorte de moyenne entre l'image de taille juste au dessus et celle juste en dessous (LINEAR_MIPMAP).

En pratique, il y a une fonction glu :

```
int gluBuild2DMipmaps( GLenum target,
                       GLint components,
                       GLint width,
                       GLint height,
                       GLenum format,
                       GLenum type,
                       const void *data );
```

Les paramètres sont les mêmes que pour **glTexImage2D**, sauf qu'il est cette fois possible d'utiliser des images de dimensions qui ne sont pas des puissances de 2. L'image passée est celle de meilleure qualité (level = 0).

Chargement dans la carte vidéo

L'intérêt du MipMapping est très limité si les textures ne sont pas stockées dans la carte vidéo. Le principe est similaire à celui des listes :

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

permet de réserver n textures, les numéros sont alors placés dans le tableau pointé par textures. Ensuite le numéro de la texture courante est changé par :

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

avec *target* = **GL_TEXTURE_2D** et *texture* le numéro de la texture à activer. Ensuite toutes les fonctions faisant intervenir une texture utilisent la texture active.

Les textures sont libérées par (suppression de *n* textures du tableau *textures*) :

void **glDeleteTextures**(GLsizei n, const GLuint * textures);

Pour résumé, une texture est le plus souvent chargée par :

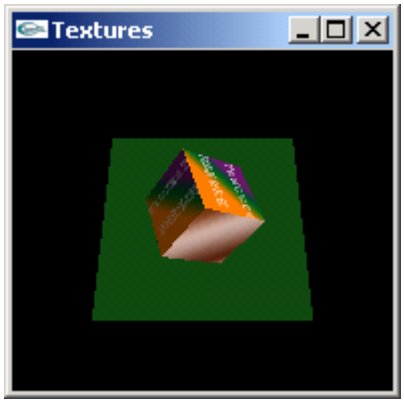
```
glGenTextures(1, &Texture_num);
glBindTexture(GL_TEXTURE_2D, Texture_num);
gluBuild2DMipmaps( GL_TEXTURE_2D, 3, width, height, GL_RGB,
                   GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Ensuite pour activer une texture, avant un glBegin :

```
glBindTexture(GL_TEXTURE_2D, Texture_num);
```

Exemple

```
GLuint Texture2, Texture1;

void SetupRC()
{
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f ); // Ce sont les valeurs par défaut.
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_POLYGON_SMOOTH);
}

void Make_CallListes()
{
  GLfloat Blanc[]  = {1.0f, 1.0f, 1.0f, 1.0f};
   cube = glGenLists(1);
   glNewList(cube, GL_COMPILE);
   glBindTexture(GL_TEXTURE_2D, Texture2);            // Deuxième texture
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Blanc);
   glBegin(GL_QUADS);
      glNormal3d(0.0,0.0,1.0);
      glTexCoord2f(0.0, 1.0); glVertex3d(-1, 1, 1);
      glTexCoord2f(1.0, 1.0); glVertex3d(-1, -1, 1);
      glTexCoord2f(1.0, 0.0); glVertex3d( 1, -1, 1);
      glTexCoord2f(0.0, 0.0); glVertex3d( 1, 1, 1);
      glNormal3d(0.0,0.0,-1.0);
      glTexCoord2f(1.0, 0.0); glVertex3d( 1, 1, -1);
      glTexCoord2f(1.0, 1.0); glVertex3d( 1, -1, -1);
      glTexCoord2f(0.0, 1.0); glVertex3d( -1, -1, -1);
      glTexCoord2f(0.0, 0.0); glVertex3d( -1, 1, -1);
      glNormal3d(1.0,0.0,0.0);
      glTexCoord2f(1.0, 1.0); glVertex3d( 1, 1, 1);
      glTexCoord2f(0.0, 1.0); glVertex3d( 1, -1, 1);
      glTexCoord2f(0.0, 0.0); glVertex3d( 1, -1, -1);
      glTexCoord2f(1.0, 0.0); glVertex3d( 1, 1, -1);
      glNormal3d(-1.0,0.0,0.0);
      glTexCoord2f(0.0, 0.0); glVertex3d( -1, 1, 1);
      glTexCoord2f(1.0, 0.0); glVertex3d( -1, 1, -1);
      glTexCoord2f(1.0, 1.0); glVertex3d( -1, -1, -1);
      glTexCoord2f(0.0, 1.0); glVertex3d( -1, -1, 1);
```



```
   glEnd();
   glBindTexture(GL_TEXTURE_2D, Texture1);            // Première texture
   glBegin(GL_QUADS);
     glNormal3d(0.0,-1.0,0.0);
     glTexCoord2f(0.0, 0.0); glVertex3d( -1, -1, 1);
     glTexCoord2f(1.0, 0.0); glVertex3d( -1, -1, -1);
     glTexCoord2f(1.0, 1.0); glVertex3d( 1, -1, -1);
     glTexCoord2f(0.0, 1.0); glVertex3d( 1, -1, 1);
     glNormal3d(0.0,1.0,0.0);
     glTexCoord2f(0.0, 0.0); glVertex3d( -1, 1, 1);
     glTexCoord2f(1.0, 0.0); glVertex3d( 1, 1, 1);
     glTexCoord2f(1.0, 1.0); glVertex3d( 1, 1, -1);
     glTexCoord2f(0.0, 1.0); glVertex3d( -1, 1, -1);
   glEnd();
   glEndList();
}

void ChargerTextures()
{
   char buffer[32*32*3];
   FILE *f = fopen("Marbre.raw", "rb");
   if (f)
   {
     fread(buffer, 3072, 1, f);     // Première texture
     fclose(f);
     glGenTextures(1, &Texture1);
     glBindTexture(GL_TEXTURE_2D, Texture1);
     glTexImage2D( GL_TEXTURE_2D,0,3, 32, 32, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, buffer);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
     glEnable(GL_TEXTURE_2D);
   }
   else
     printf("Problème au chargement de la texture\n");
   f = fopen("Marbre2.raw", "rb");
   if (f)
   {
     fread(buffer, 3072, 1, f);     // Deuxième texture
     fclose(f);
     glGenTextures(1, &Texture2);
     glBindTexture(GL_TEXTURE_2D, Texture2);
     glTexImage2D( GL_TEXTURE_2D,0,3, 32, 32, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, buffer);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   }
   else
     printf("Problème au chargement de la texture\n");
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
   glutInitWindowPosition(100, 100);
   glutInitWindowSize(largeur, hauteur);
   glutCreateWindow("Textures");

   SetupRC();
   ChargerTextures();
```

```
  Make_CallListes();
  glutReshapeFunc(Reshape);//fonction gérant le redimensionnement de la fenêtre
  glutDisplayFunc(Display);
  glutMainLoop();         //boucle principale
  return 0;
}
```

Priorités des textures

Comme il se peut que toutes les textures ne rentrent pas dans la carte vidéo, il est possible de donner des priorités au chargement des textures en mémoire vidéo.

void **glPrioritizeTextures**(GLsizei n, GLuint * textures, GLclampf * priorities);

avec *priorities* un tableau de *n* flottants appartenant à [0..1], 0 étant la priorité la plus faible et 1 la plus forte.

Les opérations sur les pixels

Introduction

Tout ce qu'on va voir ici concerne l'affichage de **bitmaps**, la copie et l'affichage de blocs de pixels et la façon dont les pixels sont stockés en mémoire.

L'utilisation la plus courante en OpenGL du transfert de pixels est l'ajout d'images qui recouvrent la scène (photo d'un cockpit, nom du joueur, score…). Il est plus facile de les afficher en utilisant les coordonnées écrans (par exemple 800x600), il faut alors repasser en mode 2D (ce qui ne fait que changer la matrice de projection, tout ce qui est 3D ne bouge pas) :

```
glMatrixMode(GL_PROJECTION) ;
glLoadIdentity() ;
gluOrtho2D(0.0, (GLfloat) width, 0.0, (GLfloat) height) ;
glMatrixMode(GL_MODEL_VIEW) ;
glLoadIdentity() ;
```

La Raster Position

La position actuelle de rastérisation est l'origine du prochain bitmap affiché. Le bitmap est affiché au dessus et à droite de cette position.

Exemple :

```
glRasterPos2i(20, 20) ;
```

La façon de trouver la position est la même que celle de **glVertex** dans le cas où l'environnement est en 3D. Si la position est hors du clipping (zone d'affichage), la position est dite invalide. Pour obtenir la Raster Position actuelle et sa validité , on utilise :

```
GLfloat vector[4] ; GLboolean Ok ;
glGetFloatv(GL_CURRENT_RASTER_POSITION, vector) ;
glGetBooleanv(GL_CURRENT_RASTER_POSITION_VALID, &Ok) ;
```

Dessiner le bitmap

Après avoir choisi la Raster Position, on peut dessiner le bitmap grâce à l'instruction **glBitmap** dont le prototype est :

```
void glBitmap( GLsizei width, GLsizei height,
               GLfloat Xbo, GLfloat Ybo, GLfloat Xbi, GLfloat Ybi,
               const GLubyte *bitmap) ;
```

Si la Raster Position est invalide, rien n'est affiché. On utilise **Xbo** et **Ybo** (**offset**) pour définir l'origine du bitmap par rapport à la Raster Position. **Xbi** et **Ybi** sont les **incréments** à la Raster Position après avoir affiché le bitmap. Pour choisir la couleur d'affichage du bitmap, on utilise glColor juste avant.

Lorsque le bit vaut 1, le pixel est dessiné avec la couleur choisie (ou éventuellement avec une combinaison). Lorsque le bit vaut 0, le contenu du pixel n'est pas affecté. Les fonctions des paragraphes suivants sont un peu plus lourdes à manipuler, mais plus générales.

Lire, écrire et copier des images

Pour lire un bloc de pixels depuis le frame buffer, la fonction **glReadPixels** doit être utilisée :

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

x et *y* représentent le coin en bas à gauche de l'image à lire, *width* et *height* sont ses dimensions. Le format et le type servent à définir le type de structure qui sera utilisé pour le stockage. Les différents types de format sont :

Format	
GL_COLOR_INDEX	
GL_RGB	Dans l'ordre : Rouge, Vert, Bleu
GL_RGBA	RGB et après Alpha
GL_BGR_EXT, GL_BGR	Dans l'ordre : Bleu, Vert, Rouge
GL_BGRA_EXT, GL_BGRA	BGR et après Alpha
GL_RED	
GL_GREEN	
GL_BLUE	
GL_ALPHA	
GL_LUMINANCE	

GL_LUMINANCE_ALPHA	luminance suivi de alpha
GL_STENCIL_INDEX	
GL_DEPTH_COMPONENT	

Type	
GL_UNSIGNED_BYTE	8bits
GL_BYTE	8 bits
GL_BITMAP	bits uniques dans des entiers 8 bits non signés utilisant le même format que glBitmap()
GL_UNSIGNED_SHORT	16 bits
GL_SHORT	16 bits
GL_UNSIGNED_INT	32 bits
GL_INT	32 bits
GL_FLOAT	flottant en simple précision

Il faut bien faire attention de faire correspondre le type de *pixel avec celui indiqué par le format et le type.

Pour écrire une image, on utilise la fonction **glDrawPixels** :

```
void glDrawPixels(  GLsizei width, GLsizei height,
                    GLenum format, GLenum type,
                    const GLvoid *pixels) ;
```

Cela dessine un rectangle contenant l'image pointée par **pixels* à la Raster Position et de la taille *width * height*. Les valeurs de format et de type sont les mêmes que celles de **glReadPixels**.

La fonction **glCopyPixels** permet de copier un rectangle d'un buffer spécifié par le cinquième paramètre (**GL_COLOR**, **GL_STENCIL** ou **GL_DEPTH**) vers la raster position.

```
void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum buffer) ;
```

Agrandissement, réduction et réflexion

Normalement, chaque pixel d'une image est écrit dans un pixel sur l'écran. La fonction :

```
void glPixelZoom(GLfloat zoomx, GLfloat zoomy) ;
```

permet de modifier la taille que chaque pixel prend sur l'écran. Si on donne 2 pour chaque argument, chaque pixel est représenté à l'écran par 4 pixels. Des valeurs non entières et négatives sont permises. Les facteurs négatifs de zoom font une réflexion par rapport à la raster position.

Tests de validation et stencil buffer

Chaîne de tests

Il est tant de voir la liste complète des tests de validation d'un point :

→ Scissor Test → Alpha Test → Stencil Test → Depth Test → […] → framebuffer

Scissor test

Le scissor test (test de scission) est paramétré par :

```
void glScissor( GLint x, GLint y, GLsizei width, GLsizei height );
```

Cela permet de limiter l'affichage à un rectangle de largeur-hauteur (width, height) dont le coin inférieur gauche a pour coordonnées (x, y) dans la fenêtre. Par défaut (x, y) = (0, 0) et (width, height) ont la même taille que la fenêtre.

Le test est activé par : **glEnable(GL_SCISSOR_TEST)**

Alpha test

Un point peut être éliminé par sa valeur alpha avec :

```
void glAlphaFunc( GLenum func, GLclampf ref );
```

avec :

- func = **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL** ou **GL_ALWAYS**
- ref = Valeur de référence (entre 0 et 1) qui est comparée avec la valeur alpha du pixel entrant.

Le test passe si (alpha entrant func ref) est vrai.

Ne pas oublier, comme toujours : **glEnable(GL_ALPHA_TEST)**

Stencil test

Ce test est plus complexe, mais plus utile que les deux précédents.

Tout comme le depth buffer, le stencil buffer est un espace mémoire de la taille du frame buffer. Mais cette fois son utilisation est libre.

Sous GLUT, ne pas oublier de le rajouter à l'initialisation :

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

Effacement

D'abord, le stencil étant comme le depth un buffer, il faut (le plus souvent) l'effacer à chaque image avec :

```
glClear(… | GL_STENCIL_BUFFER_BIT) ;
```

La valeur de remplissage est paramétrable avec :

```
void glClearStencil( GLint s );
```

Initialisation

Comme pour tous les tests, il faut appeler :

```
glEnable(GL_STENCIL_TEST) ;
```

Le test est paramétré par deux fonctions :

```
void glStencilFunc( GLenum func, GLint ref, GLuint mask);
```

qui sert à indiquer le test à effectuer. Les paramètres sont :

- **func** = **GL_NEVER**, **GL_LESS**, **GL_LEQUAL**, **GL_GREATER**, **GL_GEQUAL**, **GL_EQUAL**, **GL_NOTEQUAL**, ou **GL_ALWAYS**
- **ref** = **valeur** de référence du test
- **mask** = **masque** binaire à appliquer à la valeur du stencil et à ref pour le test

Le test passe si ((ref & mask) func (stencil & mask))

```
void glStencilOp( GLenum fail, GLenum zfail, GLenum zpass );
```

qui sert à indiquer ce qui se passe suivant le résultat du test du stencil et du depth, avec :

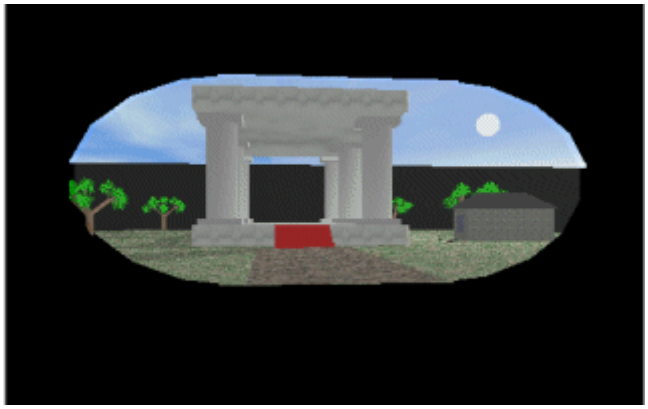
- **fail** indique l'opération à effectuer sur la valeur dans le stencil lorsque le stencil test a échoué.
  Peut être égal à **GL_KEEP**, **GL_ZERO**, **GL_REPLACE**, **GL_INCR**, **GL_DECR**, ou **GL_INVERT**
- **zfail** indique de même l'opération à effectuer si le stencil est passé mais pas le depth
- **zpass** indique l'opération à effectuer si le depth passe (ainsi que le stencil)

Utilité

Pour le moment, l'utilité du "Stencil buffer" ne doit pas vous sauter aux yeux. Et pourtant, il est extrêmement pratique. Nous verrons dans le chapitre suivant 2 grandes utilisation de ce "buffer" :

les ombres simples et le reflet dans un miroir. Mais pour l'instant, nous allons voir en gros, à quoi il sert.

Le stencil buffer permet de dessiner une scène quelconque, non pas dans la fenêtre tout entière, mais dans une forme quelconque. Imaginez que vous vouliez dessiner un cube à l'écran, mais au travers d'un hublot, rond et non pas la fenêtre tout entière, carrée. Vous allez activer le "Stencil-Test", dessiner votre hublot dans le Stencil buffer (et non pas à l'écran, nous verrons comment au prochain chapitre), puis dessiner votre cube, comme à l'accoutumé, à l'écran cette fois-ci. Seule la portion de votre cube se trouvant sur le hublot sera dessinée.



La scène n'est affichée que dans la pseudo-ellipse

Autre exemple : le "stencil buffer" sert aussi lorsque l'on veut afficher un tableau de score, un décor statique comme dans les jeux tels que Tétris (une partie de l'écran est fixe, ou presque)

Ombres et reflets

La principale application du stencil buffer, ce sont les ombres et les reflets.

Reflets

Principe

Les reflets sur un plan sont très simples : ce n'est qu'un double d'un objet recouvert d'une surface légèrement transparente pour donner l'illusion qu'il s'agit d'un reflet.

En plus, ce qui simplifie bien les choses, il existe la fonction :

```
void glScalef( GLfloat x, GLfloat y, GLfloat z );
```

qui multiplie la matrice courante par une matrice multipliant toutes les coordonnées entrées par les valeurs spécifiées, c'est-à-dire une matrice diagonale (x, y, z, 1).

Dans notre cas, pour effectuer une symétrie par rapport au plan y = 0, il suffit d'entrer **glScalef**(1.0f, -1.0f, 1.0f).

Seul problème : le reflet peut sortir de la surface du miroir ! C'est là que le stencil buffer intervient. La surface réfléchissante est d'abord « virtuellement » dessinée, en fait elle est dessinée normalement sauf que pour empêcher l'affichage on utilise :

void **glColorMask**(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);

avec tous les paramètres à **GL_FALSE**. Cela permet de changer la valeur du stencil là où la surface va être dessinée.

Les étapes du reflets sont finalement :

- Limitation du reflet à la surface réfléchissante :

```
glDisable(GL_DEPTH_TEST);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
Dessin de la surface réfléchissante
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glEnable(GL_DEPTH_TEST);
```

- Dessin du reflet :

```
glStencilFunc(GL_EQUAL, 1, 0xffffffff); /* draw if ==1 */
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glPushMatrix();
   glScalef(1.0, -1.0, 1.0);
   glEnable(GL_NORMALIZE);
   glCullFace(GL_FRONT);     // ou GL_BACK
   repositionnement des lumières
   dessin de l'objet reflété
   glCullFace(GL_BACK);              // ou GL_FRONT
   glDisable(GL_NORMALIZE);
glPopMatrix();
repositionnement des lumières
glDisable(GL_STENCIL_TEST);
```

- Dessin de la surface réfléchissante :

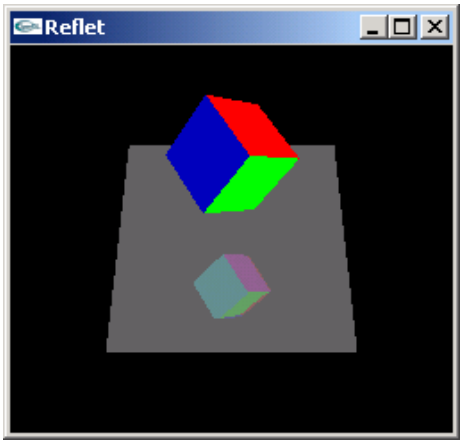
```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
Dessin de la surface réfléchissante
glDisable(GL_BLEND);
```

- Dessin de l'objet

Exemple

Voici le reflet d'un cube sur le sol :

```
void Display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
   glDisable(GL_DEPTH_TEST);

   //on ne va pas dessiner dans le buffer-écran, mais dans les autres buffers
   glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
   glEnable(GL_STENCIL_TEST);
   glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
   glStencilFunc(GL_ALWAYS, 1, 0xffffffff);

   //Dessin de la surface réfléchissante dans le stencil buffer
   drawFloor();

   //on réactive le dessin à l'écran
   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
   glEnable(GL_DEPTH_TEST);

   //cela va dessiner uniquement dans les endroits du stencil buffer
   // qui sont à 1, c.a.d. dans le miroir
   glStencilFunc(GL_EQUAL, 1, 0xffffffff);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   glPushMatrix();
         //on fait la symétrie par rapport à l'axe des Z
         glScalef(1.0, 1.0, -1.0);
         glEnable(GL_NORMALIZE);
         glCullFace(GL_FRONT);

         //les lumières aussi subissent la symétrie
         SetLight();

         //on dessine la scène qui va être reflétée
         glTranslatef(0.0f, 0.0f, 5.0f);
         glRotatef((80.0f), -0.5f, 0.3f, -0.8f);
         glRotatef((10.0f), 0.5f, -0.3f, 0.8f);
         glCallList(cube);
         glCullFace(GL_BACK);
         glDisable(GL_NORMALIZE);
   glPopMatrix();

   //fin de la symétrie
   //on remet les lumières et tout le reste
   SetLight();
   glDisable(GL_STENCIL_TEST);

   //on dessine le sol transparent
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glDisable(GL_LIGHTING);
   drawFloor();
   glEnable(GL_LIGHTING);
   glDisable(GL_BLEND);

   //puis le reste de la scène
   glPushMatrix();
         glTranslatef(0.0f, 0.0f, 5.0f);
```



```
         glRotatef((80.0f), -0.5f, 0.3f, -0.8f);
         glRotatef((10.0f), 0.5f, -0.3f, 0.8f);
         glCallList(cube);
   glPopMatrix();
   glutSwapBuffers();
}
```

```
void Make_CallListes()
{
     GLfloat Rouge[] = {0.8f, 0.0f, 0.0f, 1.0f};
     GLfloat Vert[]  = {0.0f, 0.8f, 0.0f, 1.0f};
     GLfloat Bleu[]  = {0.0f, 0.0f, 0.8f, 1.0f};
     GLfloat Jaune[]  = {0.8f, 0.8f, 0.0f, 1.0f};
     GLfloat Rose[]  = {0.8f, 0.0f, 0.8f, 1.0f};
     GLfloatCyan[]  = {0.0f, 0.8f, 0.8f, 1.0f};

     cube = glGenLists(1);
     glNewList(cube, GL_COMPILE);
       glBegin(GL_QUADS);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Rouge);
          glNormal3d(0.0,0.0,1.0);
          glVertex3d(-1,  1, 1);
          glVertex3d(-1, -1, 1);
          glVertex3d( 1, -1, 1);
          glVertex3d( 1,  1, 1);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Jaune);
          glNormal3d(-1.0,0.0,0.0);
          glVertex3d( -1,  1,  1);
          glVertex3d( -1,  1, -1);
          glVertex3d( -1, -1, -1);
          glVertex3d( -1, -1,  1);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Vert);
          glNormal3d(1.0,0.0,0.0);
          glVertex3d( 1,  1,  1);
          glVertex3d( 1, -1,  1);
          glVertex3d( 1, -1, -1);
          glVertex3d( 1,  1, -1);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Rose);
          glNormal3d(0.0,1.0,0.0);
          glVertex3d( -1, 1,  1);
          glVertex3d(  1, 1,  1);
          glVertex3d(  1, 1, -1);
          glVertex3d( -1, 1, -1);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Bleu);
          glNormal3d(0.0,-1.0,0.0);
          glVertex3d( -1, -1,  1);
          glVertex3d( -1, -1, -1);
          glVertex3d(  1, -1, -1);
          glVertex3d(  1, -1,  1);
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, Cyan);
          glNormal3d(0.0,0.0,-1.0);
          glVertex3d(  1,  1, -1);
          glVertex3d(  1, -1, -1);
          glVertex3d( -1, -1, -1);
          glVertex3d( -1,  1, -1);
       glEnd();
     glEndList();
}
```

Matrice « miroir »

L'utilisation de **glScale** est beaucoup moins pratique dès que la surface n'est pas un plan parallèle à un axe. On trouvera, à l'adresse :
http://www.opengl.org/developers/code/mjktips/Reflect.html (M. J. Kilgard)
un programme et des explications concernant les réflexions.

Ombres

Principe

Le stencil buffer joue toujours son rôle de délimitation, pour l'ombre cette fois. Reste à dessiner l'ombre elle-même, or qu'est-ce qu'une ombre ? C'est la projection de tout un objet sur un plan suivant la direction de la lumière. Cela fait (devrait faire ?) immédiatement penser à « matrice de projection ». Effectivement il suffit de multiplier la matrice ModelView par une matrice de projection pour avoir le résultat voulu.

Quelques détails tout de même :

- l'ombre n'est pas le dessin de l'objet lui-même mais une surface noire (à moitié transparente). Bien que dans la réalité, ce n'est pas exactement cela : c'est tout le reste de la scène qui est plus éclairée. Cependant il est bien plus rapide d'assombrir là où est l'ombre que d'éclaircir tout le reste. S'il n'y a aucune lumière ambiante, l'ombre n'est pas transparente.
- avec cette méthode, il faut faire attention à ne pas assombrir deux fois le même point : ce n'est pas parce que deux surfaces cachent la lumière que l'ombre est deux fois plus sombre ! L'astuce consiste ici à incrémenter le stencil lorsque le stencil test et le depth test passent (dès qu'un point est assombri, il ne fait plus « partie » de la surface).
- Cependant, il existe un cas où il faut volontairement oublier ce fait : lorsque l'objet projeté est en partie transparent. Il est alors possible d'avoir un assombrissement proportionnel au nombre de surfaces qui cachent la lumière. Sauf qu'il ne faut pas trop en abuser, le temps de calcul s'allongeant encore.
- le problème de la matrice de projection : elle n'est pas dans glu ! Voici un moyen de la calculer (M. J. Kilgard) :

```
/* Create a matrix that will project the desired shadow. */

enum { X = 0, Y, Z, W };

void shadowMatrix( GLfloat shadowMat[4][4],
                   GLfloat groundplane[4],
                   GLfloat lightpos[4])
{
  GLfloat dot;

  dot =  groundplane[X] * lightpos[X] +
         groundplane[Y] * lightpos[Y] +
         groundplane[Z] * lightpos[Z] +
         groundplane[W] * lightpos[W];
```

```
   shadowMat[0][0] = dot - lightpos[X] * groundplane[X];
   shadowMat[1][0] = 0.f - lightpos[X] * groundplane[Y];
   shadowMat[2][0] = 0.f - lightpos[X] * groundplane[Z];
   shadowMat[3][0] = 0.f - lightpos[X] * groundplane[W];

   shadowMat[X][1] = 0.f - lightpos[Y] * groundplane[X];
   shadowMat[1][1] = dot - lightpos[Y] * groundplane[Y];
   shadowMat[2][1] = 0.f - lightpos[Y] * groundplane[Z];
   shadowMat[3][1] = 0.f - lightpos[Y] * groundplane[W];

   shadowMat[X][2] = 0.f - lightpos[Z] * groundplane[X];
   shadowMat[1][2] = 0.f - lightpos[Z] * groundplane[Y];
   shadowMat[2][2] = dot - lightpos[Z] * groundplane[Z];
   shadowMat[3][2] = 0.f - lightpos[Z] * groundplane[W];

   shadowMat[X][3] = 0.f - lightpos[W] * groundplane[X];
   shadowMat[1][3] = 0.f - lightpos[W] * groundplane[Y];
   shadowMat[2][3] = 0.f - lightpos[W] * groundplane[Z];
   shadowMat[3][3] = dot - lightpos[W] * groundplane[W];
}
```

avec **shadowMat** la matrice à construire, **groundplane** l'équation du plan où projeter l'ombre (de type ax + by + cz + d = 0), et **lightpos** comme son nom l'indique est la position de la lumière (lightpos[3] de préférence à 0 → la source est à l'infini).

Et pour les coefficients du plan :

```
/* Find the plane equation given 3 points. */

enum { A = 0, B, C, D };

void findPlane(GLfloat plane[4], GLfloat v0[3], GLfloat v1[3], GLfloat v2[3])
{
   GLfloat vec0[3], vec1[3];

   /* Need 2 vectors to find cross product. */
   vec0[X] = v1[X] - v0[X];
   vec0[Y] = v1[Y] - v0[Y];
   vec0[Z] = v1[Z] - v0[Z];

   vec1[X] = v2[X] - v0[X];
   vec1[Y] = v2[Y] - v0[Y];
   vec1[Z] = v2[Z] - v0[Z];

   /* find cross product to get A, B, and C of plane equation */
   plane[A] = vec0[Y] * vec1[Z] - vec0[Z] * vec1[Y];
   plane[B] = -(vec0[X] * vec1[Z] - vec0[Z] * vec1[X]);
   plane[C] = vec0[X] * vec1[Y] - vec0[Y] * vec1[X];

   plane[D] = -(plane[A] * v0[X] + plane[B] * v0[Y] + plane[C] * v0[Z]);
}
```

Les étapes de l'ombre sont alors :

- Limitation de l'ombre à la surface :

```
glDisable(GL_DEPTH_TEST);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
Dessin de la surface                    (Elle n'est dessinée que dans le Stencil Buffer)
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glEnable(GL_DEPTH_TEST);
glDisable(GL_STENCIL_TEST);
```

- Dessiner la surface

- Dessin de l'objet

```
glEnable(GL_STENCIL_TEST);              // Pour éviter que l'ombre soit visible
glStencilFunc(GL_ALWAYS, 2, 0xffffffff); // à travers l'objet.
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glPushMatrix();
   dessin de l'objet
glPopMatrix();
```

- Calcul de la matrice de projection (avec vfloor trois points de la surface) :

```
GLfloat floorPlane[4];                  // Coefficients du plan (a, b, c, d)
GLfloat floorShadow[4][4];              // Matrice de projection
findPlane(floorPlane, vfloor[1], vfloor[2], vfloor[3]);
shadowMatrix(floorShadow, floorPlane, lightZeroPosition);
```

- Changement des paramètres pour dessiner une surface noire transparente :

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_EQUAL, 1, 0xffffffff);         // draw if ==1
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);         // Pour éviter d'assombrir 2 fois
glEnable(GL_BLEND);                             // le même point
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_LIGHTING);
glDisable(GL_DEPTH_TEST);
glColor4f(0.0, 0.0, 0.0, 0.5);                  // Valeur Alpha pour la transparence
```

- Dessin de l'ombre :

```
glPushMatrix();
   glMultMatrixf((GLfloat *) floorShadow);
   dessin de l'objet
glPopMatrix();
```

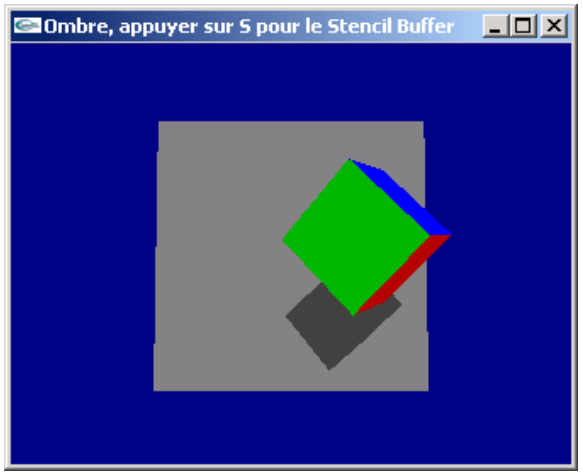
- Pour terminer

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);
glDisable(GL_BLEND);
glDisable(GL_STENCIL_TEST);
```

Exemple

```
GLfloat vfloor[4][3] = {{-4, 0, 4}, {4, 0, 4}, {4,0, -4},{-4,0, -4}};
GLfloat positionProperties[] = {0.0f, 15.0f, -5.0f, 0.0f}; // Lumière à l'infini

void drawFloor()
{
    glColor4f(0.5f, 0.5f, 0.5f, 1.0f);
    glBegin(GL_QUADS);
        glVertex3fv(vfloor[0]);
        glVertex3fv(vfloor[1]);
        glVertex3fv(vfloor[2]);
        glVertex3fv(vfloor[3]);
    glEnd();
}

void SetLight()
{
  GLfloat ambientProperties[] = {0.7f, 0.7f, 0.7f, 1.0f};
  GLfloat diffuseProperties[] = {1.0f, 1.0f, 1.0f, 1.0f};

  glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
  glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
  glLightfv( GL_LIGHT0, GL_POSITION, positionProperties);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  glEnable(GL_COLOR_MATERIAL);
}

void Display (void)
{
  static timeb tt ,t;
  static int total;
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

  // Pour l'animation
  ftime(&t);
  total += (t.time - tt.time)*1000 + (t.millitm - tt.millitm);
  total %= 360*300;
  tt = t;

  // Limitation de l'ombre à la surface
  glDisable(GL_DEPTH_TEST);
  glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
  glEnable(GL_STENCIL_TEST);
  glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
  glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
  drawFloor();
  glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
  glEnable(GL_DEPTH_TEST);
  glDisable(GL_STENCIL_TEST);

  // Dessiner la surface
  glDisable(GL_LIGHTING);
  drawFloor();
  glEnable(GL_LIGHTING);
```

```
  // Dessin de l'objet
  glEnable(GL_STENCIL_TEST);       // Pour éviter que l'ombre soit visible
  glStencilFunc(GL_ALWAYS, 2, 0xffffffff); // à travers l'objet.
  glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
  glPushMatrix();
    glTranslatef(sin(total/30.0f/180.0f*3.1415f)*5.0f, 0.0f, 3.0f);
    glRotatef((total/30.0f), 0.5f, 0.3f, 0.8f);
    glRotatef((total/50.0f), 0.5f, -0.3f, 0.8f);
    glCallList(cube);
  glPopMatrix();

  // Calcul de la matrice de projection
  // (avec vfloor trois points de la surface)
  static GLfloat floorPlane[4];
  static GLfloat floorShadow[4][4];
  findPlane(floorPlane, vfloor[1], vfloor[2], vfloor[3]);
  shadowMatrix(floorShadow, floorPlane, positionProperties);

  //Changement des paramètres pour dessiner une surface noire transparente
  glStencilFunc(GL_EQUAL, 1, 0xffffffff); // draw if ==1
  glStencilOp(GL_KEEP, GL_KEEP, GL_INCR); // Pour éviter d'assombrir 2 fois
  glEnable(GL_BLEND);                     // le même point.
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glDisable(GL_LIGHTING);
  glDisable(GL_DEPTH_TEST);
  glColor4f(0.0, 0.0, 0.0, 0.5);

  // Dessin de l'ombre
  glPushMatrix();
    glMultMatrixf((GLfloat *) floorShadow);
    glTranslatef(sin(total/30.0f/180.0f*3.1415f)*5.0f, 0.0f, 3.0f);
    glRotatef((total/30.0f), 0.5f, 0.3f, 0.8f);
    glRotatef((total/50.0f), 0.5f, -0.3f, 0.8f);
    glCallList(cube);
  glPopMatrix();

  // Pour terminer
  glEnable(GL_DEPTH_TEST);
  glDisable(GL_BLEND);
  glEnable(GL_LIGHTING);
  glDisable(GL_STENCIL_TEST);

  glutSwapBuffers();
}
```



Courbes et Surfaces calculées

Présentation

Pour certaines surfaces, comme par exemple les sphères, il se pose un problème : elles ne peuvent être qu'approchées par des triangles. Il faut faire un compromis : soit avoir une sphère très belle, ce qui peut prendre plusieurs milliers de triangles, ou bien avoir peu de triangles mais une sphère pas très sphérique. De plus, pourquoi avoir des surfaces précises si elles sont éloignées ? De toute façon elles ne font que quelques pixels.

D'où l'intérêt d'avoir des surfaces calculées : la précision de la surface est ajustée suivant les besoins (distance par rapport à l'œil), et même peut être ajustée dynamiquement suivant la puissance de la machine (si le programme tourne à 60 fps, une amélioration visuelle est raisonnable).

Quadrics

Présentation

Glu intègre déjà un certain nombre de formes : sphère, cylindre et disque. Le principe est toujours le même :

- Déclaration d'un « quadric » :

```
GLUquadricObj * qobj = gluNewQuadric();
```

- Appel(s) à l'une des fonctions suivantes (dessin immédiat) :

```
void gluCylinder(       GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius,
                        GLdouble height, GLint slices, GLint stacks );

void gluDisk(   GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint
                slices, GLint loops );

void gluPartialDisk(    GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius,
                        GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);

void gluSphere( GLUquadricObj *qobj, GLdouble radius, GLint slices, GLint stacks );
```

- Destruction avec :

```
void gluDeleteQuadric( GLUquadricObj * qobj );
```

Suivant les paramètres du rendu, il est possible de générer :

- Les normales (pour l'éclairage) :

void **gluQuadricNormals**(GLUquadricObj *qobj, GLenum normals); avec normals = GLU_NONE, GLU_FLAT, ou GLU_SMOOTH (défaut)

- Les coordonnées de texture :

void **gluQuadricTexture**(GLUquadricObj *qobj, GLboolean textureCoords); avec textureCoords = GL_FALSE (défaut) ou GL_TRUE

- Le mode de dessin :

void **gluQuadricDrawStyle**(GLUquadricObj *qobj, GLenum drawStyle): avec drawStyle = GLU_FILL (défaut), GLU_LINE, GLU_SILHOUETTE, ou GLU_POINT

- La direction dans laquelle les normales pointent :

void **gluQuadricOrientation**(GLUquadricObj *qobj, GLenum orientation); avec orientation = GLU_OUTSIDE (défaut) ou GLU_INSIDE

Exemple

```
GLfloat positionProperties[] = {0.0f, 6.0f, 0.0f, 0.0f};

void SetLight()
{
   GLfloat ambientProperties[] = {0.2f, 0.2f, 0.2f, 1.0f};
   GLfloat diffuseProperties[] = {1.0f, 1.0f, 1.0f, 1.0f};

   glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
   glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
   glLightfv( GL_LIGHT0, GL_POSITION, positionProperties);

   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHTING); // Active l'éclairage
   glEnable(GL_COLOR_MATERIAL);
}

void Display (void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   GLUquadricObj *obj = gluNewQuadric();
```

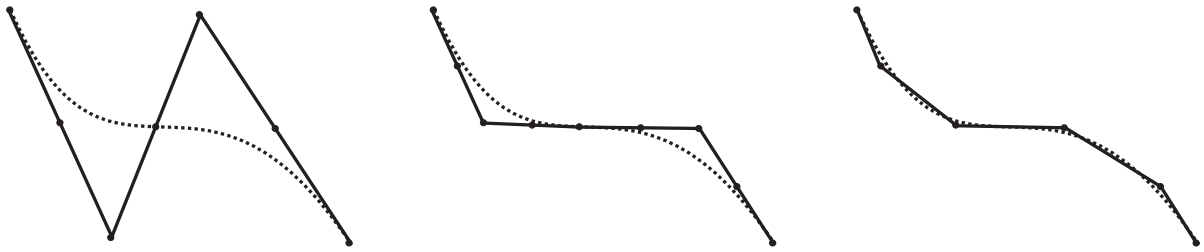
```
   gluQuadricNormals(obj, GL_SMOOTH);
   gluQuadricDrawStyle(obj, GLU_FILL);
   glColor3f(0.0f, 1.0f, 0.0f);           // Sphère verte
   gluSphere(obj, 1.0, 20, 20);
   gluDeleteQuadric(obj);

   glutSwapBuffers();
}
```

Bézier

OpenGL intègre (ce n'est pas glu cette fois) les courbes et les surfaces de Bézier. Cependant les cartes grand public n'accélèrent pas leur calcul. Pour l'instant…

Le principe d'une courbe de Bézier est le suivant :

- Soit n points de contrôle, la courbe part du premier et arrive au dernier.
- La courbe est calculée par récurrence, en ajoutant entre deux points de contrôle leur milieu jusqu'à la résolution de l'affichage. Seul les milieux du dernier niveau de récurrence sont affichés (en fait dès qu'un point, hormis les extrémités de la courbe, a servi pour calculer un milieu, il est éliminé). Un petit dessin pour essayer de mieux comprendre :



- Le principe est le même pour les surfaces.

Courbe

Fonctions OpenGL

La première chose à faire est d'entrer les points de contrôle :

```
void glMap1f(   GLenum target,                    // glMap "un" f
                GLfloat u1, GLfloat u2,
                GLint stride,
                GLint order,
                const GLfloat *points );
```

Target	Indique quel type de coordonnées il faut générer. Les choix sont : GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP1_INDEX, GL_MAP1_COLOR_4, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, ou GL_MAP1_TEXTURE_COORD_4 Pour une courbe de Bézier, il faut choisir **GL_MAP1_VERTEX_3**. Mais il est aussi possible d'utiliser GL_MAP1_COLOR_4 pour créer un dégradé de couleurs : quelle différence entre générer des coordonnées (x, y, z) et (R, G, B) ?
u1, u2	Une courbe est ensuite repérée par un « axe courbe ». Par la suite, la courbe débutera en (u1) et se terminera en (u2)
Stride	Indique le nombre de flottants qui sépare un point de contrôle dans points du suivant. Le seul but de cette valeur est de pouvoir utiliser des tableaux de structure avec d'autres données que les points.
Order	Nombre de points dans le tableau
Points	Pointeur vers les points de contrôle

Il faut appeler **glEnable** avec le même target pour activer la courbe.

Ensuite la fonction :

void **glEvalCoord1f**(GLfloat u);

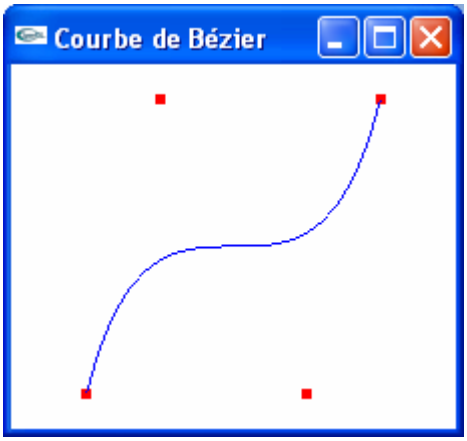
fait l'équivalent d'un **glVertex**(*coordonnées du point repéré par u*). Par exemple **glEvalCoord1f**(u1) a le même effet que **glVertex**(*premier point de contrôle*), et de même **glEvalCoord1f**(u2) a le même effet que **glVertex**(*dernier point de contrôle*). La valeur de u n'est pas limitée à [u1..u2]. Si elle sort de cette intervalle, cela prolonge simplement la courbe.

La fonction doit être appelée à l'intérieur d'un glBegin / glEnd.

Exemple

```
void Display (void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        static GLfloat ctrlpoints[4][3] = {
            {-4.0, -4.0, -15.0}, {-2.0, 4.0, -15.0},
            { 2.0, -4.0, -15.0}, { 4.0, 4.0, -15.0}};
        glColor3f(1.0f,1.0f,1.0f);
        // glMap1 peut être placé dans myInit() !
        glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
        glEnable(GL_MAP1_VERTEX_3);
        glBegin(GL_LINE_STRIP);
        for (GLfloat i = 0.0; i <= 1.0; i+=0.05f)
            glEvalCoord1f(i);
        glEnd();
        glDisable(GL_MAP1_VERTEX_3);
```

```
        glPointSize(5.0);
        glColor3f(1.0, 1.0, 0.0);
        glBegin(GL_POINTS);
        for (int j = 0; j < 4; j++)
            glVertex3fv(&ctrlpoints[j][0]);
        glEnd();
    glPopMatrix();

    glutSwapBuffers();
}

void Reshape (int width, int height)
{
  glViewport (0, 0, width, height);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective (45.0, ((GLfloat) width) / ((GLfloat) height),1.0,100.0);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  gluLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);


  glEnable(GL_DEPTH_TEST);
  glEnable(GL_POLYGON_SMOOTH);
}
```



Remarque

On peut remplacer les instructions :

```
glBegin(GL_LINE_STRIP);
 for (GLfloat i = 0.0; i <= 1.0; i+=0.05f)
   glEvalCoord1f(i);
glEnd();
```

Par :

```
// Définir une grille allant de u1 à u2 en Nb étapes régulièrement espacées.
// 20 -> Nb de subdivisions, 0.0 -> u1, 1.0 -> u2
glMapGrid1f(20, 0.0, 1.0);
// Appliquer la grille
// 0 -> 1ère  subdiv., 20 -> dernière subdiv.
// mode = GL_LINE ou GL_POINT
glEvalMesh1(mode, 0, 20);
```

Surface

Fonctions OpenGL

Les fonctions sont les mêmes avec des « 2 » au lieu de « 1 », et le tableau des points de contrôle est à 2 dimensions :

```
void glMap2f(  GLenum target,
               GLfloat u1, GLfloat u2,
               GLint ustride, GLint uorder,
               GLfloat v1, GLfloat v2,
               GLint vstride, GLint vorder,
               const GLfloat *points );

void glEvalCoord2f( GLfloat u, GLfloat v); // En plus de glEnable(GL_MAP2_VERTEX3) ;
```

En 2D, les commandes **glMapGrid2*()** et **glEvalMesh2()** sont semblables à leurs homologues 1D, mais il faut y inclure les informations u et v.

```
void glMapGrid2{fd}( GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1, TYPE v2 ) ;
void glEvalMesh2( GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2 );
```

mode peut prendre les valeurs : **GL_POINT, GL_LINE, GL_FILL.**

Si mode = GL_POINT (attention : GL_POINT sans « s »), glEvalMesh2 est équivalent à :

```
glBegin(GL_POINTS);                          // attention : GL_POINTS avec « s »
  for (i = i1; i <= i2; i++)
    for (j = j1; j <= j2; j++)
      glEvalCoord2f(u1 + i·du, v1 + j·dv);
glEnd();
```

avec :

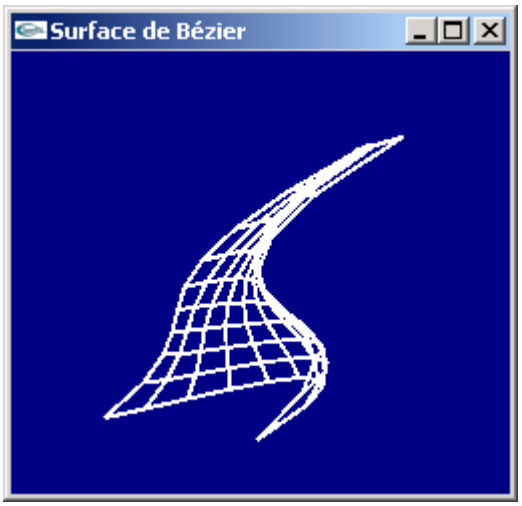
du = (u2 – u1) / nu

et

dv = (v2 – v1) / nv

Exemple

```
void Display (void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      static GLfloat ctrlpoints[4][4][3] = {
           {{-1.5, -1.5,  4.0}, {-0.5, -1.5,  2.0},
           { 0.5, -1.5, -1.0}, { 1.5, -1.5,  2.0}},
           {{-1.5, -0.5,  1.0}, {-0.5, -0.5,  3.0},
           { 0.5, -0.5,  0.0}, { 1.5, -0.5, -1.0}},
           {{-1.5,  0.5,  4.0}, {-0.5,  0.5,  0.0},
           { 0.5,  0.5,  3.0}, { 1.5,  0.5,  4.0}},
           {{-1.5,  1.5, -2.0}, {-0.5,  1.5, -2.0},
           { 0.5,  1.5,  0.0}, { 1.5,  1.5, -1.0}}
           };
      glEnable(GL_LINE_SMOOTH);
      glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4,&ctrlpoints[0][0][0]);
      glEnable(GL_MAP2_VERTEX_3);
      static GLfloat tt= 0;
      if ((tt += 0.05f) > 360) tt -= 360.0f;
      glTranslatef(0.0f, 0.0f, -10.0f);
      glRotatef(tt, 0.7f, 0.0f, 1.0f);
      glColor3f(1.0, 1.0, 1.0);
      glMapGrid2f(9, 0.0, 1.0, 9, 0.0, 1.0);
      glEvalMesh2(GL_LINE, 0, 9, 0, 9);
      glDisable(GL_MAP2_VERTEX_3);
   glPopMatrix();
   glutSwapBuffers();
}

void Reshape (int width, int height)
{
  glViewport (0, 0, width, height);

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective (45.0, ((GLfloat) width) / ((GLfloat) height),1.0,100.0);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  gluLookAt(0, 0, 0,
            0, 0, -0.5f,
            0, 1, 0);

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_POLYGON_SMOOTH);
}
```



Ce qui n’a pas de place ailleurs

Dessiner deux fois au même endroit

Il se peut qu’il faille dessiner deux fois au même endroit, pour mettre un logo sur un rectangle par exemple. Le depth buffer pose alors un problème : il n’est pas précis, ou en tout cas pas suffisamment précis pour être sûr qu’aucun point du logo ne va être éliminé.

La solution la plus simple : désactiver le depth-test. Mais ce n’est à utiliser que lorsque l’on est sûr qu’il n’y a rien de plus proche de la caméra.

Une autre solution, parfaitement sûre mais plus compliquée, c’est d’utiliser le stencil buffer : dessiner d’abord le logo et changer la valeur du stencil là où le logo est dessiné. Dessiner ensuite le rectangle sur lequel est placé le logo, mais seulement là où le stencil n’a pas bougé.

Dernière solution, la fonction :

void **glPolygonOffset**(GLfloat factor, GLfloat units);

avec factor qui permet de « lisser » la profondeur (limite la dérivée du depth, sans intérêt ici), et units le nombre « d’unités » de décalage à ajouter au point. L’unité est dépendante de l’implémentation GL, et est spécifiée comme étant « le plus petit décalage tel que deux points ayant la même profondeur soit quand même distincts ». Dans le cas qui nous intéresse, cela donne :

```
glPolygonOffset(0, 1);
```

Il faut appeler glEnable avec GL_POLYGON_OFFSET_FILL pour activer le décalage des points.

Cette méthode fonctionne assez bien, mais la précision n’atteint pas celle du stencil : si le nombre de calculs (nombre de matrices dans modelview) est trop différent pour le logo et la surface, il se peut très bien (pour cause de calcul en flottant et d’imprécision du depth) que cette technique ne marche pas. Elle est par contre très fiable si les matrices utilisées sont les mêmes pour le logo et la surface.

Clipping

Il est possible de rajouter des plans de clipping, pour exclure certaines parties du monde dessiné. Il en existe toujours au moins deux : ceux qui limitent l’intervalle de z (automatiquement générés lors de l’appel de gluPerspective).

Un plan quelconque peut être ajouté avec :

void **glClipPlane**(GLenum plane, const GLdouble *equation);

plane étant GL_CLIP_PLANEi (=GL_CLIP_PLANE0 + i), la valeur max étant GL_MAX_CLIP_PLANES.

Equation est un tableau de quatre flottants donnant l'équation du plan (ax + by + cz + d = 0). Le nombre minimal de plans possible est 6 (pour définir un cube).

Pour l'activer : glEnable(GL_CLIP_PLANEi);

La règle pour savoir si un vertex est coupé ou pas est : « If the dot product of the eye coordinates of a vertex with the stored plane equation components is positive or zero, the vertex is in with respect to that clipping plane. Otherwise, it is out. »

Multitexturing

Le but de cette partie est de montrer les possibilités d'OpenGL en combinant plusieurs des fonctions vues jusqu'à présent.

Le « multitexturing » consiste à faire plusieurs rendus d'une même surface en changeant certaines options, pour obtenir des effets lumineux. L'exemple donné ici consiste à dessiner deux fois le même objet, une fois avec une texture non éclairée, puis une deuxième fois pour ajouter un rendu sans texture mais éclairé.

En OpenGL, cela donne à peu près :

```
glEnable(GL_BLEND);
glDepthFunc(GL_LEQUAL);
glDepthMask(GL_TRUE);
glDisable(GL_LIGHTING);
glDisable(GL_DITHER);
glShadeModel(GL_FLAT);
glEnable(GL_TEXTURE_2D);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
Draw();
glEnable(GL_LIGHTING);
glDisable(GL_TEXTURE_2D);
glDepthFunc(GL_EQUAL);
glDepthMask(GL_FALSE);
glEnable(GL_DITHER);
glBlendFunc(GL_ONE, GL_ONE);
Draw();
```

Index

L

M