

# Introduction à OpenGL et GLUT

Nicolas Roussel

In Situ project-team  
LRI (Univ. Paris-Sud – CNRS) & INRIA  
<http://insitu.lri.fr/>

## OpenGL

## Quelques références

Quelques livres (voir aussi <http://www.opengl.org/>)

- “le rouge” : OpenGL Programming Guide
- “le “bleu” : OpenGL Reference Manual
- “le blanc” : OpenGL Extensions Guide
- “l’orange” : OpenGL Shading Language
- E. Angel. *OpenGL: a primer*. Addison Wesley, 2002



Quelques sites Web

- le site Web de Mesa : <http://mesa3d.org/>
- les tutoriaux de Nate Robins : <http://www.xmission.com/~nate/tutors.html>
- D. Shreiner, E. Angel & V. Shreiner. An interactive introduction to OpenGL programming. Cours donné à la conférence ACM SIGGRAPH 2001. <http://www.opengl.org/developers/code/s2001/>

## Qu'est ce qu'OpenGL ?

Une interface de programmation (API) graphique

- indépendante du langage de programmation
- indépendante du système d'exploitation
- indépendante du système de fenêtrage

Caractéristiques

- simplicité
- performance et qualité du rendu

Implémentation

- logicielle (ex : Mesa)
- matérielle (ex : cartes NVIDIA ou ATI)
- hybride...

## Comment fonctionne OpenGL ?

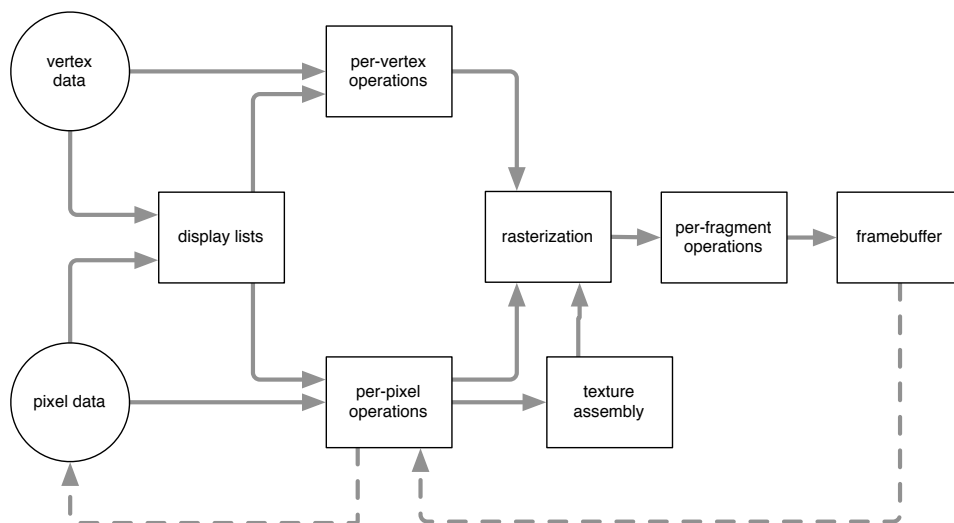
OpenGL sait dessiner des points, des lignes, des polygones convexes et des images

Ces primitives sont définies par des ensembles de vertex et rendues dans un *framebuffer*

Le rendu des primitives dépend de nombreuses variables d'état (matrice de transformation, couleur, matériau, texture, éclairage, etc.)

OpenGL ne sait pas ce que c'est qu'une souris, un clavier, une fenêtre ou même un écran...

## Le pipeline OpenGL



## Quelques autres API utiles

Pour quelques fonctions de plus : GLU (OpenGL Utility Library)

- NURBS, tessellations, formes quadriques (cylindres, etc.)
- GLU fait partie d'OpenGL

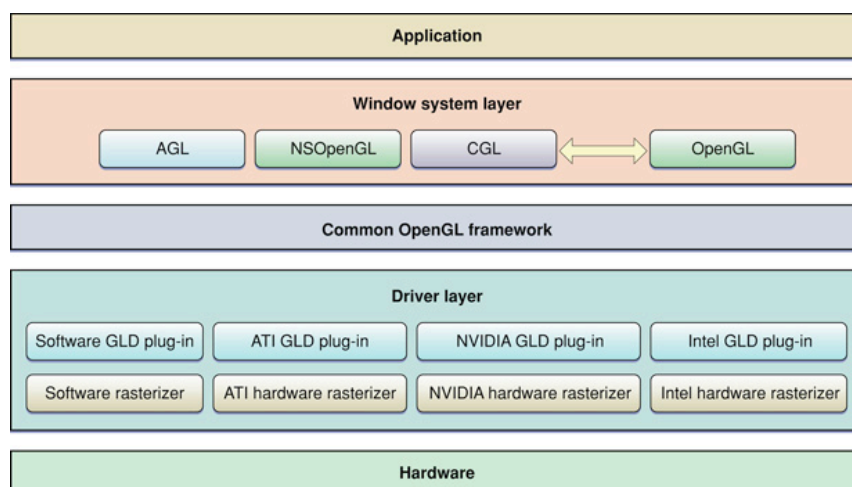
Pour accéder au système de fenêtrage

- CGL, AGL, NSOpenGL (Mac OS)
- GLX (X Window)
- WGL (Windows)

Pour se simplifier la vie, GLUT (OpenGL Utility Toolkit) : accès au système de fenêtrage, au clavier, à la souris indépendamment de la plateforme

Autres solutions multi-plateformes : SDL, GTK, QT, wxWidgets, etc.

## Exemple : OpenGL sur OS X



## GLUT

### Quelques remarques générales sur l'API OpenGL

Un certain nombre de types de données sont définis : GLfloat, GLint, GLenum par exemple  
Souvent, le nom de la fonction indique le type des paramètres attendus

Exemple : glVertex3fv

- ▶ gl : c'est une fonction OpenGL !
- ▶ 3 : 3 coordonnées (2, 3 ou 4)
- ▶ f : données flottantes (byte, unsigned byte, short, unsigned short, int, unsigned int, float ou double)
- ▶ v : données passées sous forme de tableau

Toutes les fonctions sont correctement documentées. Voir <http://www.opengl.org/sdk/docs/man/>

## Pour commencer

Inclure au minimum dans votre programme

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

et éventuellement

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

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

Structure générale d'une application GLUT

- création d'au moins une fenêtre
- initialisation de quelques variables d'états
- enregistrement de quelques *callbacks* pour l'animation et/ou l'interaction
- entrée dans la boucle de gestion des événements

## Exemple : mini.c

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

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glRectf(-0.5,-0.5,0.5,0.5);
    glutSwapBuffers();
}

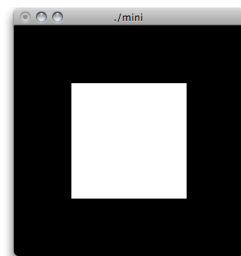
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);

    glutCreateWindow(argv[0]);

    glutDisplayFunc(display);

    glutMainLoop();

    return 0;
}
```



## Compiler mini.c sous Linux

Dans un terminal : `cc mini.c -o mini -lglut -lglu -lgl`

La commande *glxinfo* peut vous dire quelle implémentation d'OpenGL vous utilisez

Pour les entêtes (\*.h) et les librairies (libGL\* et autres)

- si ça ne marche pas, ajouter quelque chose du type `-I/usr/X11R6/include` et `-L/usr/X11R6/lib`
- sous OS X, c'est à peine plus compliqué : utiliser `GLUT/glut.h`, `OpenGL/glu.h` et `OpenGL/glex.h` au lieu de `GL/glut.h`, `GL/glu.h` et `GL/glex.h`
- sous Windows... je ne sais pas !

Sous Linux et OS X, un Makefile n'est pas inutile...

## Exemple de Makefile pour Linux et OS X

```
OS = $(shell uname -s)

CXX    = g++
CPPFLAGS =
CXXFLAGS = -O2
ifeq ($(OS), Darwin)
    LDFLAGS =
    LDLIBS = -framework GLUT -framework OpenGL -framework Cocoa
else
    LDFLAGS = -L/usr/X11R6/lib
    LDLIBS = -lglut -lGLU -lGL
endif

PROGRAM = mini

all: $(PROGRAM)

.PHONY: clean distclean

clean:
    @rm -rf *.o *~

distclean: clean
    @rm -f $(PROGRAM)
```

## Principales fonctions de rappel (*callbacks*) de GLUT

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

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

```
void glutEntryFunc(void (*func)(int state));
```

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

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

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

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

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

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

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

```
void glutTimerFunc(unsigned int millis, void (*func)(int value), int value);
```

et de nombreuses autres (menus, joystick, tablet, overlay, button box, dials, spaceball, etc.)

## Exemple d'utilisation de glutKeyboardFunc

(mini-k.c, basé sur mini.c)

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

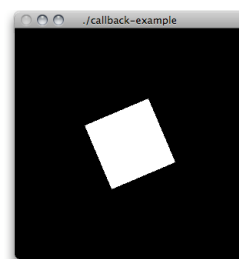
```
#include <stdlib.h>
```

```
GLfloat angle = 0.0 ;
```

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT) ;  
    glLoadIdentity() ;  
    glRotatef(angle,0,0,1) ;  
    glRectf(-0.5,-0.5,0.5,0.5) ;  
    glutSwapBuffers() ;  
}
```

```
void keyboard(unsigned char key, int x, int y) {  
    switch (key) {  
        case ' ':  
            angle ++ ;  
            glutPostRedisplay() ;  
            break ;  
        case 27 /* Esc */ :  
            exit(1) ;  
        }  
    }
```

```
int main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;  
  
    glutCreateWindow(argv[0]) ;  
  
    glutDisplayFunc(display) ;  
    glutKeyboardFunc(keyboard) ;  
  
    glutMainLoop() ;  
  
    return 0 ;  
}
```





## Exemple d'utilisation de glutMotionFunc

(mini-m.c, basé sur mini.c)

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

GLfloat angle = 0.0 ;
int prev_x = -1 ;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT) ;
    glLoadIdentity() ;
    glRotatef(angle,0,0,1) ;
    glRectf(-0.5,-0.5,0.5,0.5) ;
    glutSwapBuffers() ;
}

void motion(int x, int y) {
    if (prev_x!=-1) {
        angle += x-prev_x ;
        glutPostRedisplay() ;
    }
    prev_x = x ;
}

int main(int argc, char **argv) {
    glutInit(&argc, argv) ;
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;

    glutCreateWindow(argv[0]) ;

    glutDisplayFunc(display) ;
    glutMotionFunc(motion) ;

    glutMainLoop() ;

    return 0 ;
}
```

## Exemple d'utilisation de glutTimerFunc

(mini-t.c, basé sur mini.c)

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

GLfloat angle = 0.0 ;
unsigned int delay = 100 ; /* milliseconds */

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT) ;
    glLoadIdentity() ;
    glRotatef(angle,0,0,1) ;
    glRectf(-0.5,-0.5,0.5,0.5) ;
    glutSwapBuffers() ;
}

void timer(int theTimer) {
    angle ++ ;
    glutPostRedisplay() ;
    glutTimerFunc(delay, timer, theTimer) ;
}

int main(int argc, char **argv) {
    glutInit(&argc, argv) ;
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;

    glutCreateWindow(argv[0]) ;

    glutDisplayFunc(display) ;
    glutTimerFunc(delay, timer, 0) ;

    glutMainLoop() ;

    return 0 ;
}
```

## Exemple d'utilisation de glutIdleFunc

(mini-i.c, basé sur mini.c)

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

```
GLfloat angle = 0.0 ;
```

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT) ;  
    glLoadIdentity() ;  
    glRotatef(angle,0,0,1) ;  
    glRectf(-0.5,-0.5,0.5,0.5) ;  
    glutSwapBuffers() ;  
}
```

```
void idle(void) {  
    angle ++ ;  
    glutPostRedisplay() ;  
}
```

```
int main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;
```

```
    glutCreateWindow(argv[0]) ;
```

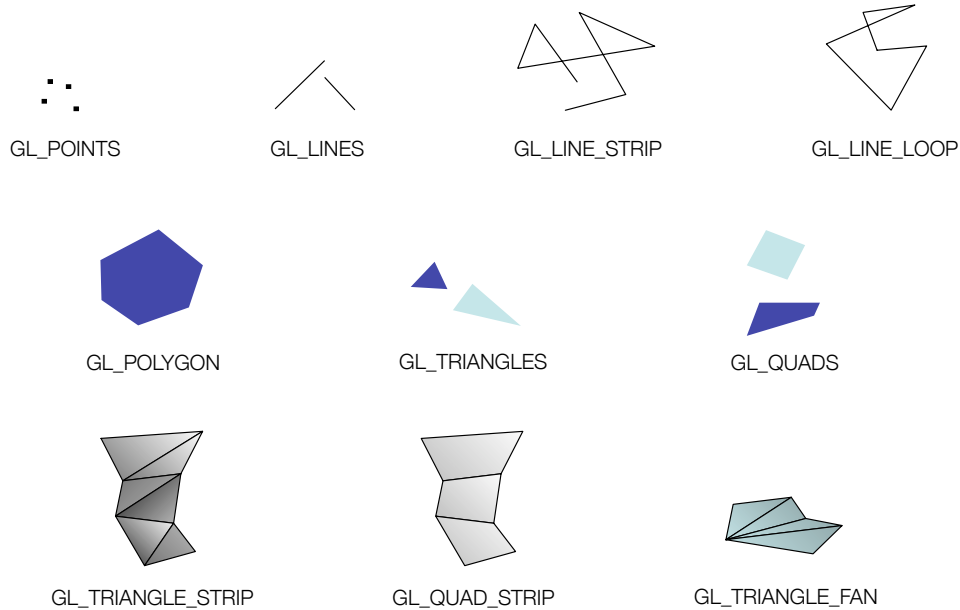
```
    glutDisplayFunc(display) ;  
    glutIdleFunc(idle) ;
```

```
    glutMainLoop() ;
```

```
    return 0 ;  
}
```

## Primitives géométriques

## Primitives géométriques



## Dessin d'une primitive

(primitives.c)

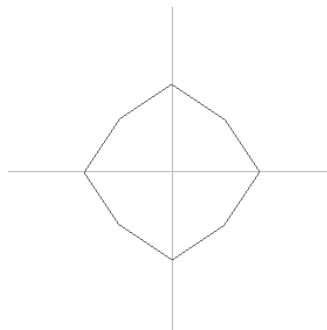
Les primitives sont décrites par un bloc d'instructions délimité par `glBegin` et `glEnd` qui contient une liste de vertex

Exemple

```
GLfloat points[8][2] = {
    {-0.5,0.0}, {-0.3,0.3}, {0.0,0.5}, {0.3,0.3},
    {0.5,0.0}, {0.3,-0.3}, {0.0,-0.5}, {-0.3,-0.3}
};

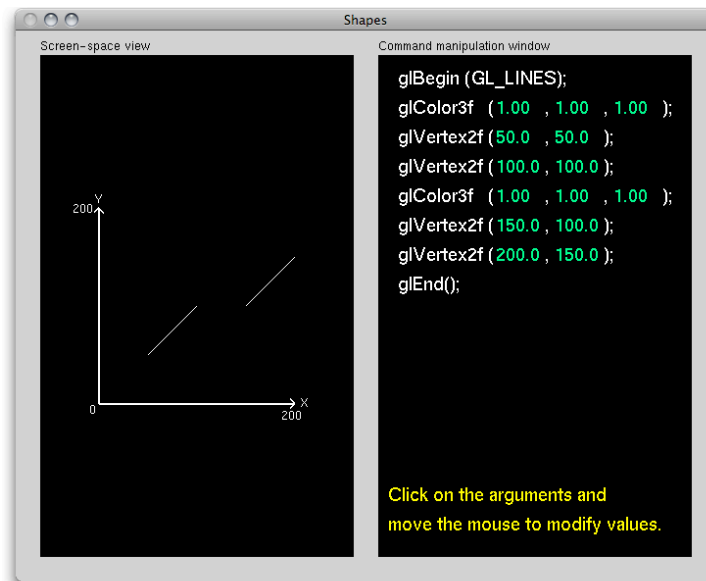
int nbpoints = 8 ;

glBegin(GL_LINE_LOOP) ;
for (i=0; i<nbpoints; ++i) glVertex2fv(points+2*i) ;
glEnd() ;
```



Différents attributs peuvent être spécifiés pour chaque vertex : couleur, normale, coordonnée de texture, etc.

## Tutorial *Shapes* de Nate Robins



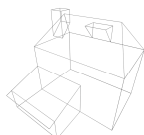
## OpenGL repose sur une machine à états

De nombreuses variables d'états contrôlent l'éclairage, le placage de texture, le style de rendu des primitives, etc.

Ces variables peuvent être manipulées

- ▶ glColor\*, glNormal\*, glTexCoord\*
- ▶ glPointSize, glPolygonMode, glShadeModel
- ▶ glEnable, glDisable
- ▶ etc.

En modifiant les variables d'états, on peut changer le style de rendu



## Systèmes de coordonnées et transformations géométriques

### Modèle de la caméra synthétique

Analogie : l'observateur regarde le monde à travers une caméra posée sur un trépied

Il peut changer l'objectif ou ajuster le zoom de la caméra

- définition de la projection (*projection transformations*)

Il peut déplacer et orienter le trépied et la caméra

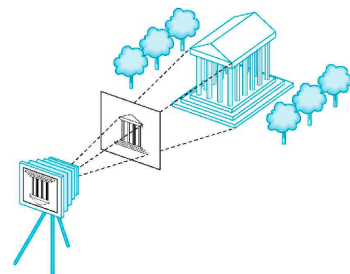
- position et orientation de la vue (*viewing transformations*)

Il peut déplacer les objets observés ou les faire déplacer

- transformation du modèle (*modeling transformations*)

Il peut enfin agir sur les images 2D produites

- agrandissement ou réduction des images (*viewport transformations*)



## Systèmes de coordonnées et transformations

Différentes étapes menant à la formation d'une image

- spécification des objets (*world coordinates*) et de la caméra (*camera coordinates*)
- projection (*window coordinates*)
- affichage dans un viewport (*screen coordinates*)

Chaque étape applique de nouvelles transformations et correspond à un changement de système de coordonnées

## Transformations affines

Les transformations affines préservent la géométrie

- une ligne reste une ligne
- un polygone reste un polygone
- une quadrique (ellipsoïde, paraboloid, hyperboloid, cônes et cylindres) reste une quadrique

Exemples de transformations affines

- translation
- changement d'échelle (*scale*)
- rotation
- projection
- glissement (*shear*)
- toute composition de transformations affines

## Coordonnées homogènes

Un point en 3 dimensions est spécifié par 4 coordonnées :  $V = [x, y, z, w]$

- $w=1$  habituellement (sinon, diviser par  $w$ )
- seule la projection change  $w$
- $[x, y, z, 0.0]$  définit une direction et non plus un point

Intérêt des coordonnées homogènes

- les coordonnées et les transformations s'expriment par des matrices  $4 \times 4$
- les compositions de transformations correspondent à des produits de matrices, mais attention à l'ordre des transformations (le produit de matrices n'est pas commutatif...)

## Programmation des transformations en OpenGL

OpenGL gère deux piles de matrices de transformation : l'une pour la projection, l'autre pour la vue et le modèle

- on passe d'une pile à une autre par l'instruction `glMatrixMode` en spécifiant `GL_PROJECTION` ou `GL_MODELVIEW`
- la pile courante peut être manipulée par les instructions `glPushMatrix` et `glPopMatrix`

La matrice courante (i.e., le haut de la pile courante) peut être modifiée par `glLoadIdentity`, `glLoadMatrix*`, `glMultMatrix*`

Des fonctions existent pour simplifier les translations, changements d'échelle et rotations ainsi que les projections

- `glTranslate*`, `glScale*`, `glRotate*`
- `glOrtho` et `gluOrtho2D`, `glFrustum` et `gluPerspective`

## Transformations de base

glTranslate(tx, ty, tz)

1	0	0	<b>tx</b>
0	1	0	<b>ty</b>
0	0	1	<b>tz</b>
0	0	0	1

glScale(sx, sy, sz)

<b>sx</b>	0	0	0
0	<b>sy</b>	0	0
0	0	<b>sz</b>	0
0	0	0	1

glRotate(a,x,y,z)

<b><math>x^2(1-c)+c</math></b>	<b><math>xy(1-c)-zs</math></b>	<b><math>xz(1-c)+ys</math></b>	0
<b><math>xy(1-c)+zs</math></b>	<b><math>y^2(1-c)+c</math></b>	<b><math>yz(1-c)-xs</math></b>	0
<b><math>xz(1-c)-ys</math></b>	<b><math>yz(1-c)+xs</math></b>	<b><math>z^2(1-c)+c</math></b>	0
0	0	0	1

$c = \cos(\text{angle})$

$s = \sin(\text{angle})$

$\|(x,y,z)\| = 1$

glRotate(a,1,0,0)

1	0	0	0
0	<b><math>\cos(a)</math></b>	<b><math>\sin(a)</math></b>	0
0	<b><math>-\sin(a)</math></b>	<b><math>\cos(a)</math></b>	0
0	0	0	1

glRotate(a,0,1,0)

<b><math>\cos(a)</math></b>	0	<b><math>-\sin(a)</math></b>	0
0	1	0	0
<b><math>\sin(a)</math></b>	0	<b><math>\cos(a)</math></b>	0
0	0	0	1

glRotate(a,0,0,1)

<b><math>\cos(a)</math></b>	<b><math>\sin(a)</math></b>	0	0
<b><math>-\sin(a)</math></b>	<b><math>\cos(a)</math></b>	0	0
0	0	1	0
0	0	0	1

## Transformations inverses et autres

Transformations inverses

$$T^{-1}(tx,ty,tz) = T(-tx,-ty,-tz)$$

$$S^{-1}(sx,sy,sz) = S(1/sx,1/sy,1/sz)$$

$$R^{-1}(\theta,x,y,z) = R(-\theta,x,y,z)$$

Les symétries selon un axe correspondent à des changements d'échelle avec un facteur -1 pour cet axe et 1 pour les autres



## Composition de transformations

(transformations.c)

glTranslate, glScale et glRotate multiplient la matrice de transformation courante par celle de la transformation correspondante (post-multiplication, "à droite")

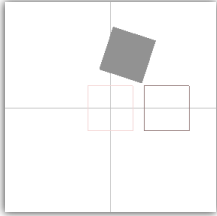
glRotatef(angle,0,0,1) ;  
glTranslatef(0.5,0,0) ;  
glRectf(-0.2,-0.2,0.2,0.2) ;

cos(a)	sin(a)	0	0
-sin(a)	cos(a)	0	0
0	0	1	0
0	0	0	1

glTranslatef(0.5,0,0,0)

1	0	0	tx
0	1	0	ty
0	0	1	tz
0	0	0	1

x
y
z
1



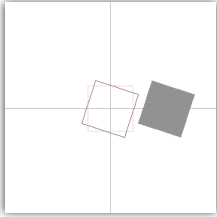
glTranslatef(0.5,0,0,0) ;  
glRotatef(angle,0,0,1) ;  
glRectf(-0.2,-0.2,0.2,0.2) ;

1	0	0	tx
0	1	0	ty
0	0	1	tz
0	0	0	1

glRotatef(angle,0,0,1)

cos(a)	sin(a)	0	0
-sin(a)	cos(a)	0	0
0	0	1	0
0	0	0	1

x
y
z
1



## Composition de transformations (suite)

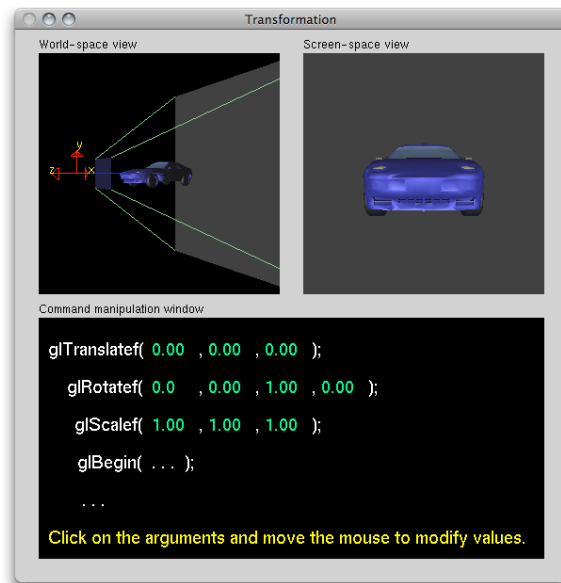
Exemple particulièrement adapté : un modèle hiérarchique

- la position et l'orientation de chaque élément dépend d'un autre élément auquel il se rattache
- chaque transformation d'un élément influe sur les éléments qui s'y rattachent
- post-multiplication des matrices : OpenGL fait ça tout seul !



Revers de la médaille : les transformations s'additionnant, il est difficile d'appliquer une transformation indépendamment des précédentes

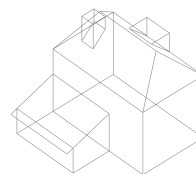
## Tutorial *Transformation* de Nate Robins



## Définition de la projection et du *viewport* OpenGL

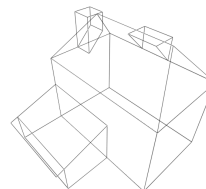
### Projection orthographique

- ▶ `glOrtho(left, right, bottom, top, near, far)`
- ▶ `gluOrtho2D(left, right, bottom, top)`



### Projection en perspective

- ▶ `glFrustum(left, right, bottom, top, near, far)`
- ▶ `gluPerspective(fovy, aspect, near, far)`



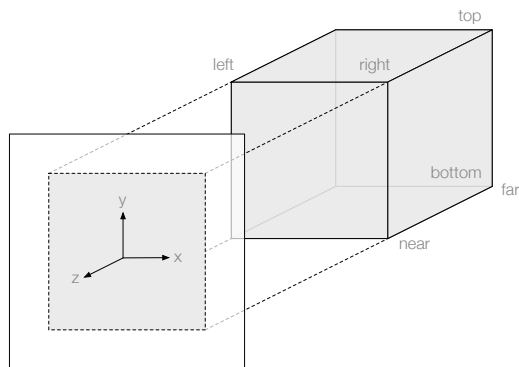
### Viewport

- ▶ défini par `glViewport(x,y,width,height)`
- ▶ sa taille correspond généralement à celle de la fenêtre
- ▶ attention : son aspect-ratio doit être identique à celui de la transformation de projection

## Projection orthographique

Dans la fonction main : glutReshapeFunc(reshape) ;

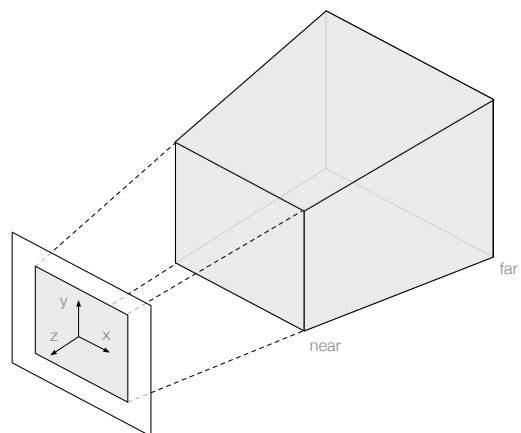
```
void reshape(int width, int height) {
    GLdouble aspect = (GLdouble) width / height ;
    GLdouble near = -1.0, far = 1.0 ;
    GLdouble left = -1.0, right = 1.0 ;
    GLdouble bottom = -1.0, top = 1.0 ;
    glViewport(0, 0, width, height) ;
    glMatrixMode(GL_PROJECTION) ;
    glLoadIdentity() ;
    if ( aspect < 1.0 ) {
        left /= aspect ;
        right /= aspect ;
    } else {
        bottom *= aspect ;
        top *= aspect ;
    }
    glOrtho(left, right, bottom, top, near, far) ;
    glMatrixMode(GL_MODELVIEW) ;
    glLoadIdentity() ;
}
```



## Projection en perspective

Dans la fonction main : glutReshapeFunc(reshape) ;

```
void reshape(int width, int height) {
    GLdouble aspect = (GLdouble) width / height ;
    GLdouble near = -1.0, far = 1.0 ;
    GLdouble fovy = 60.0 ;
    glViewport(0, 0, width, height) ;
    glMatrixMode(GL_PROJECTION) ;
    glLoadIdentity() ;
    gluPerspective(fovy, aspect, near, far) ;
    glMatrixMode(GL_MODELVIEW) ;
    glLoadIdentity() ;
    gluLookAt(0.0, 0.0, 5.0, /* œil */
              0.0, 0.0, 0.0, /* point observé */
              0.0, 1.0, 0.0) ; /* où est le ciel ? */
}
```



## Plans de clipping additionnels

Les paramètres de la projection (i.e. le volume de visualisation) définissent six plans de clipping

Au moins six autres plans arbitraires sont disponibles

Exemple

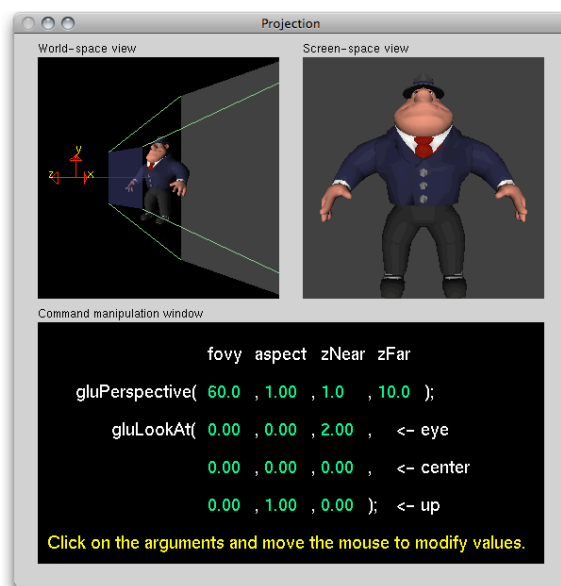
```
GLdouble planeEqn[] = {0.707, 0.707, 0.0, 0.0} ;  
glClipPlane(GL_CLIP_PLANE0, planeEqn) ;  
glEnable(GL_CLIP_PLANE0) ;
```

Les quatre doubles définissent l'équation du plan

$$ax + by + cz + d = 0$$

Nous en reparlerons plus tard, lorsque nous essaierons de construire un miroir...

## Tutorial *Projection* de Nate Robins



## Animation : le minimum à savoir

### Dessiner avec un seul buffer

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

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT) ;  
    glRectf(-0.3,-0.3,0.3,0.3) ;  
    glFlush() ; // ou glFinish() ;  
}
```

```
void main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE) ;  
  
    glutCreateWindow(argv[0]) ;  
  
    glClearColor(0.0,0.0,1.0,1.0) ;  
  
    glutDisplayFunc(display) ;  
  
    glutMainLoop() ;  
}
```

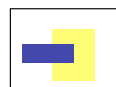


image n affichée



dessin de l'image n+1



dessin de l'image n+1 (suite)



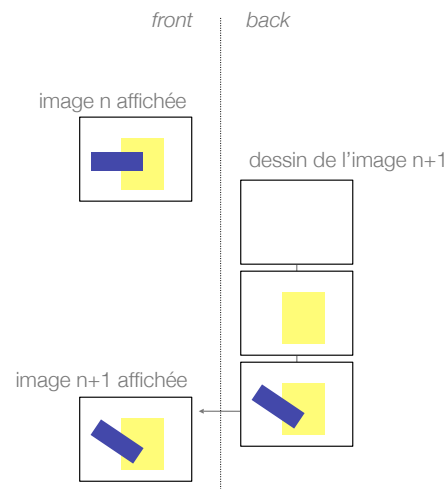
image n+1 affichée

## Dessiner avec deux buffers

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

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT) ;  
    glRectf(-0.3,-0.3,0.3,0.3) ;  
    glutSwapBuffers() ;  
}
```

```
void main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;  
  
    glutCreateWindow(argv[0]) ;  
  
    glClearColor(0.0,0.0,1.0,1.0) ;  
  
    glutDisplayFunc(display) ;  
  
    glutMainLoop() ;  
}
```



## Comment stopper une animation ?

### Rappel

- void glutIdleFunc(void (\*func)(void)) ;
- void glutTimerFunc(unsigned int msec, void (\*func)(int value), int value) ;

### Pour une animation basée sur glutIdleFunc

- glutIdleFunc(NULL)
- si la procédure associée traite plusieurs choses, utiliser des variables d'état booléennes

### Pour une animation basée sur glutTimerFunc

- impossible d'annuler un timer enclenché
- utiliser une variable d'état booléenne qui indique qu'au prochain déclenchement, il ne faut rien faire (ni exécuter l'action associée, ni le réenclencher)

## Exemple avec glutIdleFunc

```
void animate_idle(void) {  
    ... // change la vue ou les objets  
    glutPostRedisplay() ;  
}
```

```
void keyboard(unsigned char key, int x, int y) {  
    switch (key) {  
        case 'a':  
            glutIdleFunc(animate_idle) ;  
            break ;  
        case 'A':  
            glutIdleFunc(NULL) ;  
            break ;  
    }  
}
```

```
void display(void) {  
    ...  
}
```

```
void main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;  
    glutCreateWindow(argv[0]) ;  
  
    glutDisplayFunc(display) ;  
    glutKeyboardFunc(keyboard) ;  
  
    glutMainLoop() ;  
}
```

## Idle ou timer ?

### Avantage de glutTimerFunc

- prévu pour pouvoir associer plusieurs timers à une même callback (le dernier paramètre)
- on sait quand la callback sera exécutée

### Inconvénient de glutIdleFunc

- on ne sait pas quand la callback sera exécutée
- peut consommer tout le temps CPU...

En résumé : préférer des animations basées sur des timers

Dernier point : si la fenêtre n'est pas visible, les animations ne sont peut-être pas nécessaires (voir glutVisibilityFunc)

## Exemple avec glutTimerFunc

```
void animate_timer(int theTimer) {
    if (t_active) {
        ... // change la vue ou les objets
        glutPostRedisplay() ;
        glutTimerFunc(t_delay, animate_timer,
                      theTimer) ;
    }
}

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 'a':
            if (!t_active)
                glutTimerFunc(t_delay, animate_timer, 0) ;
            t_active = true ;
            break ;
        case 'A':
            t_active = false ;
            break ;
    }
}
```

```
unsigned int t_delay = 100 ;
bool t_active = false ;

void display(void) {
    ...
}

void main(int argc, char **argv) {
    glutInit(&argc, argv) ;
    glutInitDisplayMode(GLUT_RGB
                        | GLUT_DOUBLE) ;
    glutCreateWindow(argv[0]) ;

    glutDisplayFunc(display) ;
    glutKeyboardFunc(keyboard) ;

    glutMainLoop() ;
}
```

Couleur et opacité



## Deux modes de gestion des couleurs

### Mode indexé

- (très) rarement utilisé
- de nombreuses choses ne marchent pas ou pas complètement dans ce mode (l'éclairage par exemple)
- ex : `glutSetColor(0, 0, 0, 1)` puis `glIndexi(0)` pour du bleu

### Mode RGB ou RGBA

- le plus courant
- rouge, vert, bleu et alpha sont spécifiés entre  $[0, 1]$  ou  $[0, 255]$
- ex : `glColor4f(0, 0, 1, 1)` pour du bleu

## Alpha : opacité/transparence

(alpha.c)

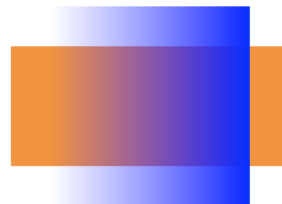
La composante alpha mesure l'opacité du fragment

- 0.0 = transparent
- 1.0 = opaque
- on a droit aux valeurs intermédiaires !

A quoi sert la composante alpha ?

- à simuler des objets transparents ou translucides
- à composer des images
- à faire de l'*antialiasing*

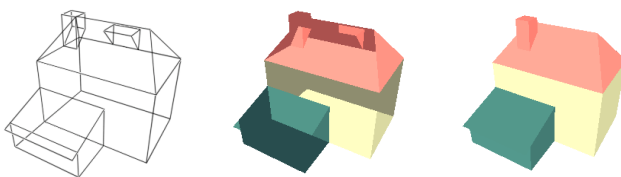
Remarque : l'alpha est ignorée si le *blending* est désactivé



## Suppression des parties cachées

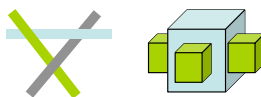
## Suppression des parties cachées

Objectif : éviter le résultat du milieu



Première solution : l'algorithme dit "du peintre"

- ▶ on dessine les objets du plus éloigné au plus proche
- ▶ solution très utilisé en 2D (la distance est facile à calculer)
- ▶ plus difficile en 3D
- ▶ quelques cas pathologiques



## Le test de profondeur

Deuxième solution : utilisation d'un *Z buffer* (ou *depth buffer*)

```
if (pixel.z < depthBuffer(x,y).z) {  
    depthBuffer(x,y).z = pixel.z ;  
    colorBuffer(x,y).color = pixel.color ;  
}
```

Les valeurs sont comprises entre  $[0,1]$  (1=infiniment loin)

Le depth buffer doit être créé lors de la création de la fenêtre et il doit être effacé de temps en temps...

## Utilisation du test de profondeur avec OpenGL

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) ;  
    ... // dessine ce qu'il faut  
    glutSwapBuffers() ;  
}  
  
void main(int argc, char **argv) {  
    glutInit(&argc, argv) ;  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH) ;  
    glutCreateWindow(argv[0]) ;  
  
    glClearColor(0.0,0.0,1.0,1.0) ;  
    glEnable(GL_DEPTH_TEST) ;  
  
    glutDisplayFunc(display) ;  
    glutMainLoop() ;  
}
```

## Eclairement et ombrage

### Préliminaires...

Les sources lumineuses présentes dans la scène introduisent un éclairage différent des faces en fonction de leur orientation

La lumière émise ou réfléchie donne aux objets un aspect brillant ou réfléchissant

On distingue deux composantes de la lumière

- la lumière ambiante (non directionnelle)
- les sources directionnelles à distance finie (un spot) ou infinie (le soleil)

et trois interactions objet/lumière

- absorption et conversion en chaleur (objet noir)
- réflexion vers d'autres objets (objet brillant ou réfléchissant)
- transmission de lumière à travers l'objet (objet transparent)

Un modèle physiquement correct est nécessairement global (chaque objet reflète plus ou moins de lumière vers les autres), mais OpenGL ne travaille que localement, polygone par polygone...

## Modèles d'éclairage et d'ombrage

Un modèle d'éclairage définit l'intensité lumineuse en un point

Un modèle d'ombrage définit les points de calcul et l'interpolation utilisés pour calculer l'éclairage d'une surface

Exemples de modèles d'éclairage : Gouraud, Phong

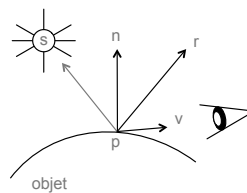
Exemples de modèles d'ombrage : plat, Gouraud, Phong

OpenGL utilise le modèle d'éclairage de Phong et propose le choix entre deux modèles d'ombrage, le modèle plat et celui de Gouraud

## Modèle d'éclairage d'OpenGL : le modèle de Phong

Dans ce modèle, l'intensité calculée en un point prend en compte quatre composantes

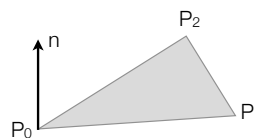
- ▶ la réflexion diffuse
- ▶ la réflexion spéculaire
- ▶ la réflexion ambiante
- ▶ la lumière émise



Calcul de la normale à une face

$$n = (P_2 - P_0) \times (P_1 - P_0)$$

$$\text{normalisation : } n = n / |n|$$



## 1<sup>ère</sup> composante : la réflexion diffuse

La forme la plus simple de réflexion

- aussi appelée réflexion Lambertienne
- modélise les surfaces mates (ex : craie)

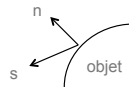
Réflexion diffuse idéale

- la lumière est diffusée dans toutes les directions
- l'apparence est identique quelle que soit la vue
- l'intensité ne dépend que de l'orientation

$$I_{\text{diff}} = I_s * k_d * \max(n \cdot s, 0)$$

$I_s$  : intensité de la source

$k_d$  : coefficient de réflexion diffuse [0, 1]

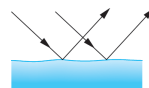
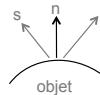


## 2<sup>ème</sup> composante : la réflexion spéculaire (1/2)

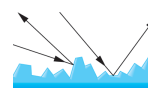
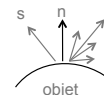
Beaucoup d'objets sont brillants...

- leur apparence change lorsqu'on tourne autour
- ils ont des "spécularités" parce qu'ils réfléchissent la lumière dans une direction particulière

Réflexion spéculaire parfaite : le miroir



La plupart des surfaces brillantes sont des miroirs imparfaits



Le modèle de réflexion spéculaire de Phong est très utilisé, mais purement empirique (pas de base physique)

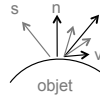
## 2<sup>ème</sup> composante : la réflexion spéculaire (2/2)

$$I_{\text{spec}} = I_s * k_s * \max(r.v, 0)^b$$

$I_s$  : intensité de la source

$k_s$  : coefficient de réflexion spéculaire [0,1]

$b$  : coefficient de brillance



$r$  correspond à la direction de réflexion spéculaire idéale

Concernant le coefficient de brillance ( $b$ )

- ▶ plus il est grand, plus l'angle de réflectance (angle entre  $r$  et  $v$ ) est petit
- ▶ les valeurs entre 100 et 200 donnent un look *métal*
- ▶ les valeurs entre 5 et 10 donnent un look *plastique*

## 3<sup>ème</sup> composante : la réflexion ambiante

Pour le moment, avec la réflexion diffuse et spéculaire, les objets qui ne sont pas directement éclairés sont noirs...

Dans le monde réel, il y a beaucoup de lumière ambiante

Phong propose donc d'ajouter une nouvelle source de lumière, purement fictive

$$I_{\text{amb}} = I_a * k_a$$

$I_a$  : intensité de la lumière ambiante

$k_a$  : coefficient de réflexion ambiante de la surface

## 4<sup>ème</sup> composante : la composante d'émission

Les sources de lumière du monde réel ont une aire finie (i.e. ne sont pas ponctuelles)

Comment représenter les sources de lumière ?

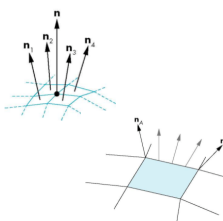
Phong propose d'ajouter une composante d'émission qui n'est pas affectée par la lumière reçue ou par la direction de la vue

## Modèles d'ombrage d'OpenGL

(shading.c)

Différences entre l'ombrage plat, de Gouraud et de Phong

- ombrage plat : une seule intensité par face, calculée sur le premier sommet
- l'ombrage de Gouraud calcule une intensité en chaque sommet (en utilisant la moyenne des normales des faces adjacentes) puis interpole entre ces intensités en chaque point
- l'ombrage de Phong interpole les vecteurs normaux et calcule l'intensité en chaque point



OpenGL implémente les modèles d'ombrage plat (GL\_FLAT) et Gouraud (GL\_SMOOTH)



Attention : pour un ombrage de Gouraud correct, il faut de nombreux points !



## Comment éclairer un polygone ? (1/2)

Première méthode : en demandant à OpenGL de le faire

```
glBegin(GL_TRIANGLES);  
for (int i=0; i<n; ++i) {  
    glNormal3fv(t[i].n);  
    glVertexfv(t[i].p1);  
    glVertexfv(t[i].p2);  
    glVertexfv(t[i].p3);  
}  
  
glBegin(GL_TRIANGLES);  
for (int i=0; i<n; ++i) {  
    glNormal3fv(t[i].n1);  
    glVertexfv(t[i].p1);  
    glNormal3fv(t[i].n2);  
    glVertexfv(t[i].p2);  
    glNormal3fv(t[i].n3);  
    glVertexfv(t[i].p3);  
}
```

Les normales doivent être calculées par le programme et doivent être unitaires

On peut éventuellement utiliser `glEnable(GL_NORMALIZE)` ou `glEnable(GL_RESCALE_NORMAL)`

## Comment éclairer un polygone ? (2/2)

Deuxième méthode : en spécifiant "à la main" les couleurs à utiliser

```
glBegin(GL_TRIANGLES);  
for (int i=0; i<n; ++i) {  
    glColor3fv(t[i].color);  
    glVertexfv(t[i].p1);  
    glVertexfv(t[i].p2);  
    glVertexfv(t[i].p3);  
}  
  
glBegin(GL_TRIANGLES);  
for (int i=0; i<n; ++i) {  
    glColor3fv(t[i].color1);  
    glVertexfv(t[i].p1);  
    glColor3fv(t[i].color2);  
    glVertexfv(t[i].p2);  
    glColor3fv(t[i].color3);  
    glVertexfv(t[i].p3);  
}
```

Les couleurs peuvent avoir été calculées avec des modèles d'éclairage et d'ombrage différents de ceux d'OpenGL

## Définition d'un matériau

```
Gfloat c_amb[] = {0.8, 0.7, 0.3, 1.0} ;  
Gfloat c_diff[] = {...} ;  
Gfloat c_spec[] = {...} ;  
Gfloat c_em[] = {...} ;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, c_diff) ;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, c_spec) ;  
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50.0) ;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, c_amb) ;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, c_em) ;
```

## Définition d'une source

### Sources ponctuelles

- la source est un point infiniment petit
- la lumière est émise dans toutes les directions (isotropique)

Il y a au moins 8 sources disponibles (utiliser glGetIntegerv avec GL\_MAX\_LIGHTS)

```
glEnable(GL_LIGHT) ; // active l'éclairage  
                        // maintenant, glColor ne sert plus à rien...  
glEnable(GL_LIGHT0) ; // allume la première source  
glLightfv(GL_LIGHT0, GL_POSITION, p) ;  
glLightfv(GL_LIGHT0, GL_DIFFUSE, c1) ;  
glLightfv(GL_LIGHT0, GL_SPECULAR, c2) ;  
glLightfv(GL_LIGHT0, GL_AMBIENT, c3) ;
```

Le positionnement des sources peut être de deux types

- local (un point)
- infini (une direction)

Le type est déterminé par la coordonnée w du tuple (x,y,z,w) spécifiée avec GL\_POSITION

- w=0 : direction (x,y,z)
- w=1 : point (x/w, y/w, z/w)

## Eclairage avancé

Il est possible de créer des lumières de type spot

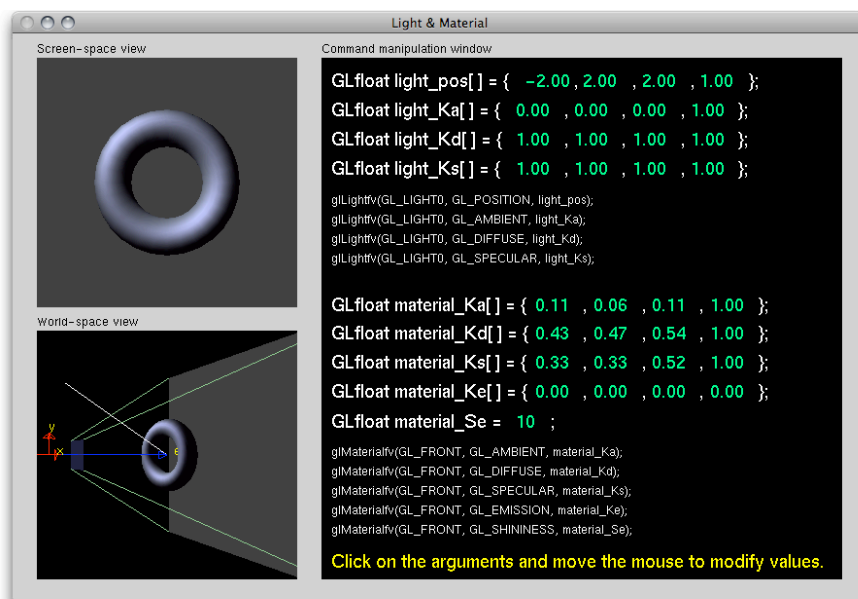
- lumière limitée à un cône
- cône défini par GL\_SPOT\_DIRECTION, GL\_SPOT\_CUTOFF et GL\_SPOT\_EXPONENT

Il est possible de spécifier une fonction d'atténuation

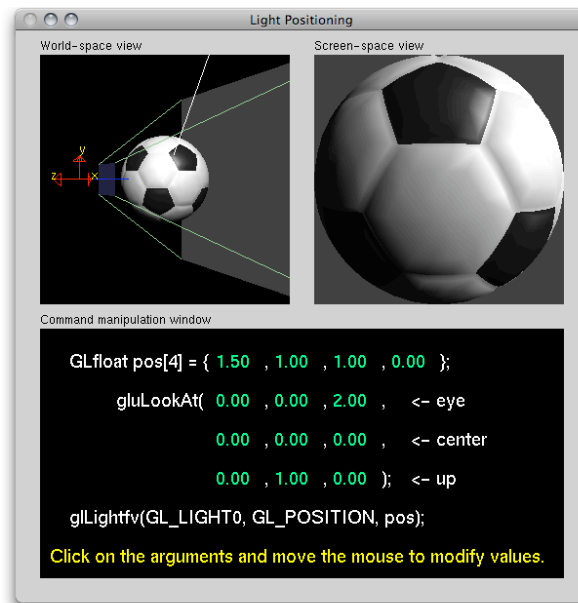
- fonction définie par GL\_CONSTANT\_ATTENUATION, GL\_LINEAR\_ATTENUATION et GL\_QUADRATIC\_ATTENUATION :  $f_i = 1 / (k_c + k_l d + k_q d^2)$
- (1,0,0) par défaut, (0,0,1) correspond au monde réel

Quelques éléments du modèle peuvent être ajustés globalement

## Tutorial *Light Material* de Nate Robins



## Tutorial *Light Position* de Nate Robins



## Ce qui n'est pas pris en compte...

Certains phénomènes locaux

- ombres
- objets transparents

Illumination globale

- la réflexion d'un objet dans un autre
- la diffusion indirecte (la lumière ambiante est un hack sordide)

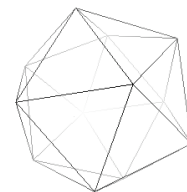
Les détails de surface (e.g. peau d'orange)

## Le brouillard

### Le brouillard

(depthcueing.c)

Pour quoi faire ? Du *depth cueing*, par exemple : modifier la couleur ou la transparence pour donner une idée de l'éloignement d'un point



Deux types de brouillard

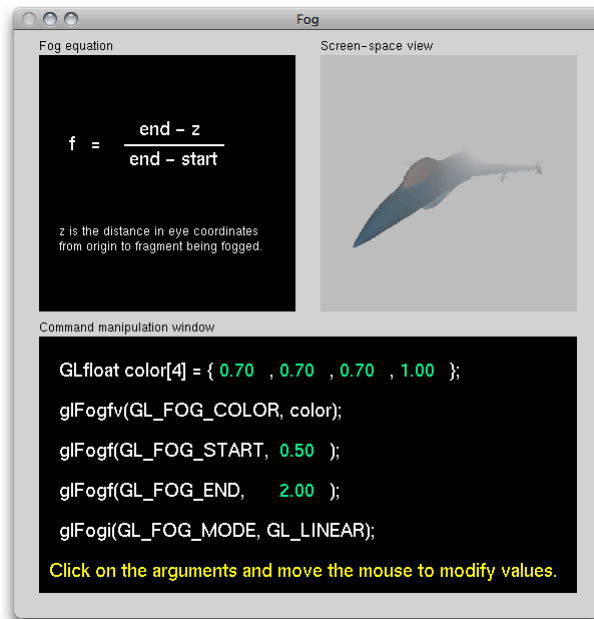
- GL\_LINEAR
- GL\_FOG\_EXP ou GL\_FOG\_EXP2

La couleur du brouillard est mélangée à la couleur du fragment

Exemple

```
glEnable(GL_FOG) ;  
glFog*(...) ; // couleur, début et fin de la zone  
glHint(GL_FOG_HINT, GL_NICEST) ; // autres modes : GL_FASTEST ou GL_DONT_CARE
```

## Tutorial *Fog* de Nate Robins



Affichage et capture d'images

## Bitmaps et images

### Bitmap

- un tableau de bits
- une seule couleur (on/off)
- généralement utilisé comme masque

### Image

- un tableau de pixels
- formats : couleur indexée, L, RGB, RGBA, etc.
- types : unsigned byte, unsigned short, float, etc.

OpenGL ne connaît rien à PNG, JPEG, TIFF, TGA, etc.

Bitmaps et images sont affichés en bout de pipeline

Un pixel n'est pas un fragment, encore moins un vertex...

## Affichage des bitmaps et images

```
glRasterPos*(x,y)
```

```
glLogicOp(Glenum op) // COPY, OR et XOR (facultatif)
```

```
glBitmap(GLsizei w, GLsizei h, GLfloat x, GLfloat y, GLfloat dx, GLfloat dy, const GLubyte *bitmap)
```

```
glDrawPixels(Glsizei w, Glsizei h, Glenum format, Glenum type, Glvoid *array)
```

Le coin inférieur gauche de l'image est dessiné en (x,y) spécifié par glRasterPos

Les images sont toujours affichées "droites"

## Problème classique avec glRasterPos

La matrice de transformation courante est utilisée pour transformer les coordonnées passées

Si le résultat de la transformation est en dehors du viewport, l'image (ou le bitmap) n'est pas affichée, même si un bout de l'image devrait être visible



Une ruse de sioux qui marche si 0,0 est dans le viewport

```
void glImagePosition(float x, float y) {  
    glRasterPos2f(0, 0) ;  
    glBitmap(0,0, 0,0, x,y, 0) ;  
}
```

## Quelques fonctions utiles

glPixelZoom(GLfloat xfactor, GLfloat yfactor)

glCopyPixels(GLint x, GLint y, GLsizei w, GLsizei h, GLenum type)

- copie une partie du frame buffer à la RasterPos courante
- résultat indéterminé si la source ou la destination sont partiellement clippés

glReadPixels(GLint x, GLint y, GLsizei w, GLsizei h, GLenum format, GLenum type, GLvoid \*pixels)

Attention : l'image obtenue est inversée verticalement



## Exemple d'utilisation de glReadPixels: faire une copie d'écran

```
void glScreenCapture(const char *filename) {
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    int x=viewport[0], y=viewport[1];
    int width=viewport[2], height=viewport[3];
    unsigned int lineSize = width*3;
    unsigned char *data = new unsigned char [height*lineSize];

    glReadBuffer(GL_FRONT);
    glReadPixels(x, y, width, height, GL_RGB, GL_UNSIGNED_BYTE, data);

    int fd = createFile (filename);
    char buffer[256];
    sprintf(buffer, "P6\n%d %d\n255\n", width, height);
    write(fd, buffer, strlen(buffer));
    for (int l=height-1; l>=0; --l) {
        unsigned char *ptr = data + l*lineSize;
        write(fd, ptr, lineSize);
    }
    close(fd);

    delete [] data;
}
```

## Exemple d'utilisation de glReadPixels : remonter le pipeline

```
void unproject(int x, int y, GLdouble *ox, GLdouble *oy, GLdouble *oz) {
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);

    ... // faire ici les transformations de projection
    GLdouble projmatrix[16];
    glGetDoublev(GL_PROJECTION_MATRIX, projmatrix);

    ... // faire ici les transformations de vue
    GLdouble mvmatrix[16];
    glGetDoublev(GL_MODELVIEW_MATRIX, mvmatrix);

    GLfloat z = -10.0;
    glReadBuffer(GL_FRONT);
    glReadPixels(x, viewport[3]-y, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, (GLvoid *)&z);

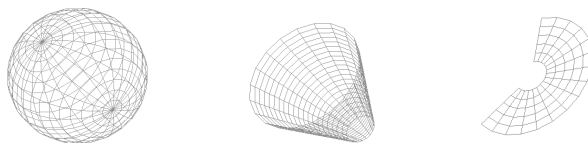
    gluUnProject(x, viewport[3]-y, z, mvmatrix, projmatrix, viewport, ox, oy, oz);
}
```

## Quelques primitives géométriques de plus

### Les quadriques de GLU

(more-primitives.c, glu-sphere.c)

Quadriques : sphere, cylindre et disques

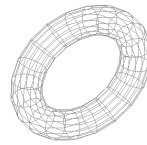
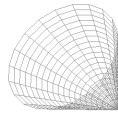
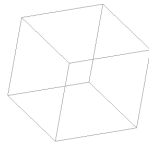


Exemple d'utilisation

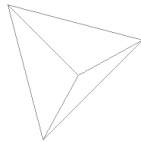
```
GLUquadricObj *obj = gluNewQuadric() ;  
...  
gluQuadricDrawStyle(obj, GLU_LINE) ;  
gluSphere(obj, 1.0 /*radius*/, 20 /*slices*/, 20 /*stacks*/) ;  
...  
gluDeleteQuadric(obj) ;
```

## Les objets de GLUT

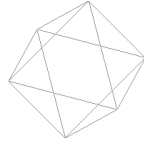
Cubes, cônes et tores



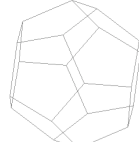
Solides platoniciens (polyèdres réguliers convexes, peuvent servir de dés...)



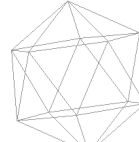
tétraèdre (4)



octaèdre (8)

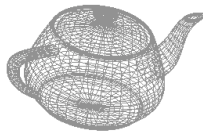


dodécaèdre (12)



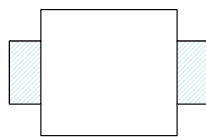
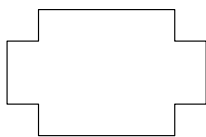
icosaèdre (20)

La théière de l'Université d'Utah



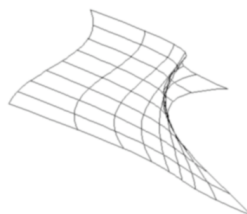
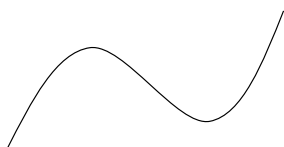
## Il y a aussi...

Pour les polygones concaves : la tessellation de GLU



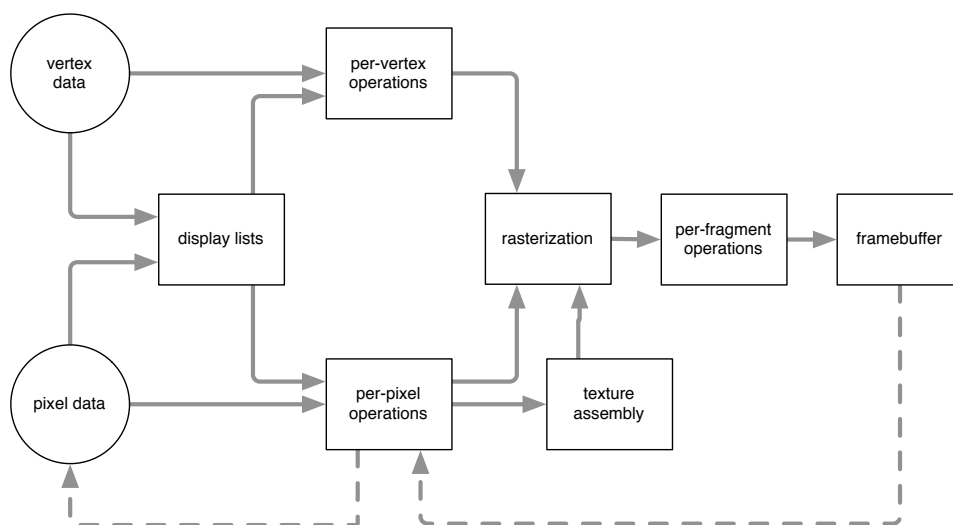
Courbes et surfaces de Bézier : évaluateurs d'OpenGL

NURBS (non uniform rational B-splines) : GLU



## Placage de textures

## Retour sur le *pipeline* OpenGL



## Des vertices au fragments

### Vertices

- normales, coordonnées de texture
- transformations, projection, clipping
- éclairage

rastérisation

### Fragments

- placage de texture
- brouillard, antialiasing
- tests (scissor, alpha, stencil, depth)
- blending, dithering, opérations logiques

## Placage de textures : pour quoi faire ?

### Modèle d'éclairage de Phong et ombrage de Gouraud

- assez limité en terme de rendu (canette de soda ?)
- produit des objets très ennuyeux

### Comment rendre les choses plus intéressantes ?

- en créant des objets plus complexes
- en utilisant des textures

### Intérêt du placage de texture

- permet de simuler des matériaux ou des phénomènes naturels (eau, feu, etc.) ou optiques
- permet d'alléger la géométrie des objets
- facile, pas cher
- le résultat est souvent aussi bon !

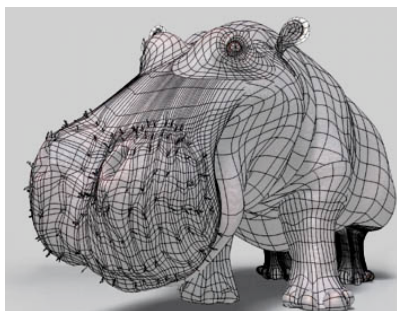
## Qu'est-ce qu'une texture

Ce qui définit l'apparence et le toucher (*feel*) d'une surface

Une image utilisée pour définir les caractéristiques d'une surface

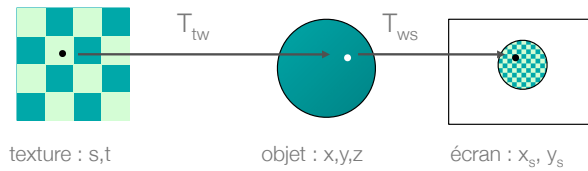
Une image multi-dimensionnelle qui est plaquée sur un espace multi-dimensionnel

Exemple : Honda Hippo (source : 3dRender.com)



## Principe de base du placage de texture 2D

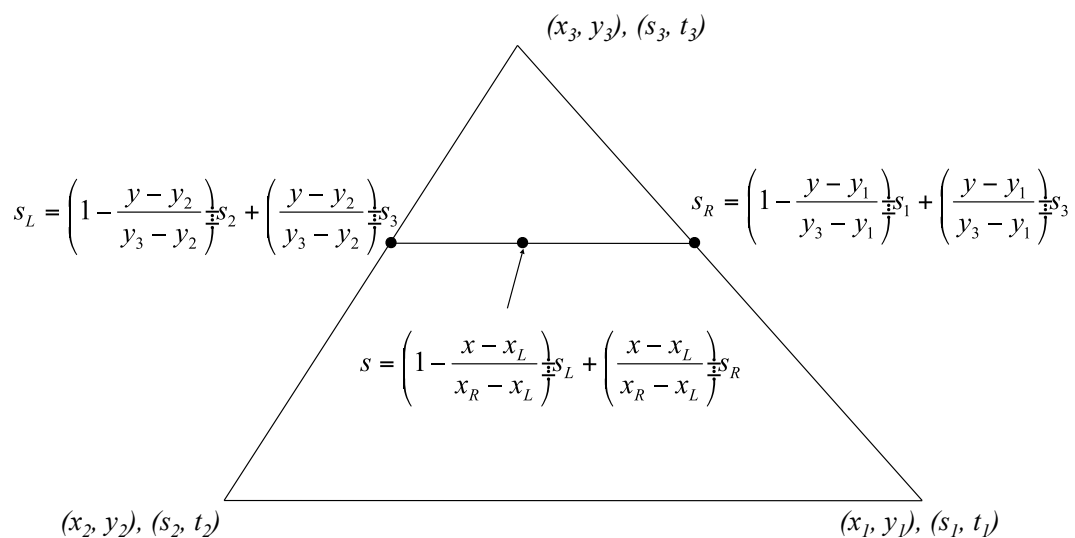
Un texel est un couple  $(s, t)$  de coordonnées  $\in [0, 1]$  identifiant un point de la texture



Problème : comment trouver le(s) texel(s) correspondant à un fragment  $x_s, y_s$  ?

Solution : couples vertex/texel et interpolation

## Interpolation de coordonnées

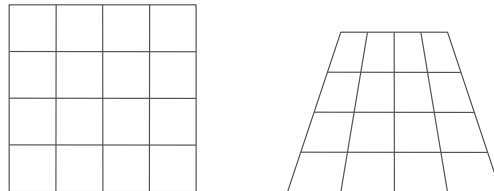


## Premier problème : prise en compte de la perspective

(texpersp.c)

Le placage de texture est effectué une fois que le polygone a été rastérisé

Si on utilise une projection en perspective, la taille des segments est réduite en fonction de la distance (sauf pour les lignes parallèles au plan image)

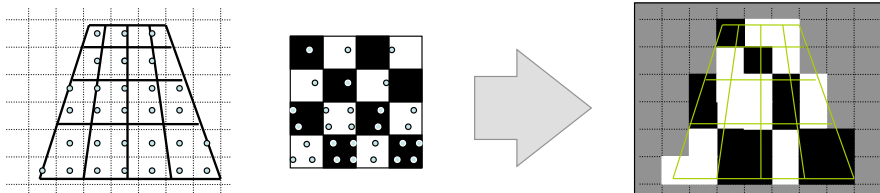


L'interpolation correcte est possible (GL\_PERSPECTIVE\_CORRECTION\_HINT), plus coûteuse, mais aujourd'hui implémentée efficacement par la plupart des cartes

Le problème ne se pose pas en projection orthographique

## Deuxième problème : *aliasing*

Problème : la correspondance entre le pixel et le texel n'est pas forcément bonne



Ce problème est similaire à celui du redimensionnement d'une image, excepté le fait que les rapports ne sont pas constants

Deux approches pour résoudre ce problème

- pré-filtrage : les textures sont filtrées avant le placage
- post-filtrage : plusieurs texels sont utilisés pour le calcul de chaque pixel



## Pré-filtrage

(mipmaps.c)

Technique du *mipmapping*

- construction d'un ensemble d'images de taille  $t/2$ ,  $t/4$ , ..., 1
- au moment du placage, l'image choisie est telle que un fragment corresponde au plus à 4 texels



Inconvénient : l'ensemble des mipmaps occupe deux fois plus de place que l'image originale

Intérêt : les images utilisées peuvent être différentes

## Post-filtrage

(mipmaps.c)

Principe : description de ce qui doit se passer quand l'image est agrandie ou rétrécie

En gros, deux possibilités

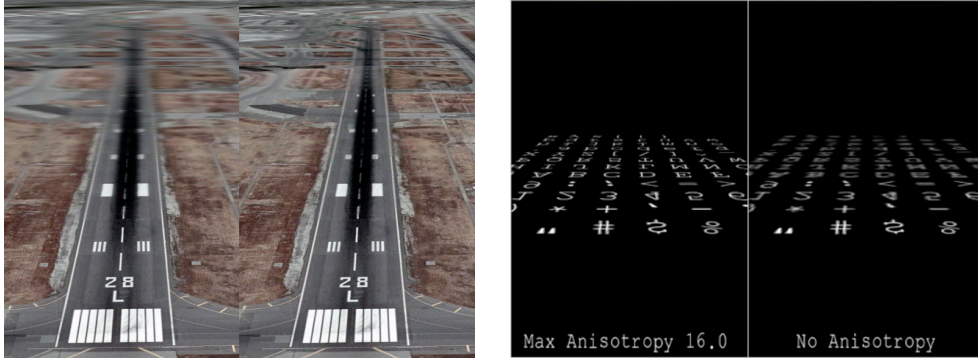
- choisir le texel le plus proche
- interpoler entre les  $n$  texels les plus proches (4 pour OpenGL)

Si on utilise le mipmapping, il faut aussi choisir le mipmap (ou interpoler entre deux)

Autre technique : le filtrage anisotropique

- jusqu'à 16 texels pris en compte
- une "sensibilité" différente en X et en Y spécialement conçue pour les cas tordus...

## Exemples de filtrage anisotropique



## Placage de texture 2D avec OpenGL

### 1. spécifier les données de la texture

- lire les données ou les générer
- associer les données à un *texture object*

### 2. spécifier les paramètres du placage

- correction de la perspective
- comportement au niveau des bords
- post-filtrage
- combinaison avec le fragment

### 3. dessiner la (les) primitive(s)

- activer le *texture object*
- associer des coordonnées de textures à chaque vertex (manuellement ou automatiquement)

## Spécification des données d'une texture 2D

A partir de données en mémoire

```
glTexImage2D(target, mipmaplevel, internalformat,  
             width, height, border,  
             format, type, texels)  
glTexSubImage2D(...)
```

A partir d'une partie du frame buffer

```
glCopyTexImage2D(...)  
glCopyTexSubImage2D(...)
```

Valeurs possibles pour *target* : *GL\_TEXTURE\_2D*, *GL\_PROXY\_TEXTURE\_2D* ou autre (extensions)

Initialement, les dimensions de l'image devaient être des puissances de 2 (restriction levée depuis septembre 2004 par OpenGL 2.0)

Les *texture objects* permettent de basculer rapidement entre plusieurs textures

## Exemple

```
// création d'un texture object  
GLuint texture ;  
glGenTextures(1, &texture) ;  
glBindTexture(GL_TEXTURE_2D, texture) ;  
// chargement des données (on suppose que les données retournées sont au format RGB)  
unsigned int width, height ;  
unsigned char *data = loadImage(filename, &width, &height) ;  
// copie des données dans la texture  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data) ;  
// les données ayant été copiées, on peut maintenant libérer la mémoire pointée par data  
...  
  
// activation du placage de texture  
glEnable(GL_TEXTURE_2D) ;  
// activation du texture object précédemment créé  
glBindTexture(GL_TEXTURE_2D, texture) ;  
... dessin de l'objet à texturer
```

## Charger une image JPEG avec libjpeg

```
#include <jpeglib.h>
#include <jerror.h>

unsigned char *loadImage(const char *filename,
                        unsigned int *width, unsigned int *height) {
    FILE *file=fopen(filename,"rb");
    if (file==NULL) {
        printf( "JPEG file (%s) not found!\n" , fichier);
        return 0 ;
    }

    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, file);
    jpeg_read_header(&cinfo, true);

    *width = cinfo.image_width ;
    *height = cinfo.image_height ;
    unsigned char *data =
    malloc(cinfo.image_width*cinfo.image_height*3) ;
    jpeg_start_decompress(&cinfo);

    jpeg_start_decompress(&cinfo);
    while (cinfo.output_scanline<cinfo.output_height) {
        unsigned char *ligne=data
        +3*cinfo.image_width*cinfo.output_scanline;
        jpeg_read_scanlines(&cinfo,&ligne,1);
    }
    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);

    return data ;
}
```

## Paramétrage : correction de la perspective

Deux modes d'interpolation

- linéaire
- utilisant les informations de profondeur/perspective (lent)

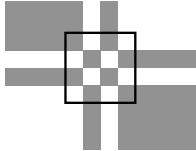
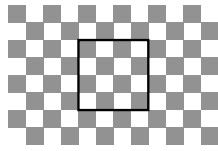
Choix du mode par

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, hint) ;
```

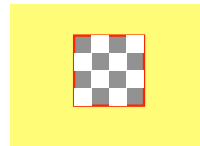
où hint est GL\_DONT\_CARE, GL\_NICEST ou GL\_FASTEST

## Paramétrage : bords de texture

Pour chaque dimension (s et t), il est possible de répéter la texture ou de la prolonger avec la dernière valeur



Une bordure peut également être spécifiée



```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, R,G,B,A)
```

## Paramétrage : post-filtrage

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, type)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, type)
```

Pour la réduction

- ▶ GL\_NEAREST : point le plus proche dans la texture
- ▶ GL\_LINEAR : interpole entre les 4 points les plus proches
- ▶ GL\_NEAREST\_MIPMAP\_LINEAR : interpolation des 4 points les plus proches du mipmap le plus proche
- ▶ GL\_LINEAR\_MIPMAP\_LINEAR : interpolation des 4 points les plus proches dans l'interpolation des deux mipmaps les plus proches

Pour l'agrandissement : GL\_NEAREST ou GL\_LINEAR

Pour créer automatiquement les mipmaps : gluBuild2DMipmaps

## Paramétrage : combinaison avec le fragment

glTexEnv permet de contrôler la façon dont la texture est appliquée sur le fragment

```
glTexEnv (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode) ;
```

Les modes les plus utilisés sont

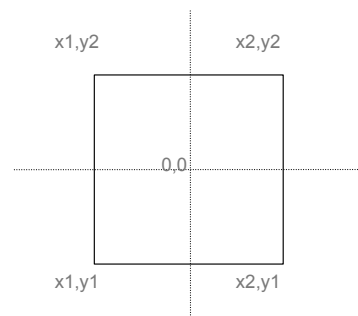
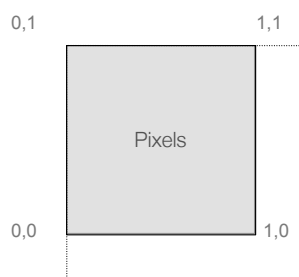
- ▶ GL\_MODULATE : multiplie la couleur du fragment par celle du texel
- ▶ GL\_BLEND : mélange la couleur du fragment et celle du texel
- ▶ GL\_REPLACE : remplace le fragment par le texel

Il existe aussi un mode GL\_DECAL similaire à GL\_REPLACE pour RGB

## Exemple de spécification des coordonnées de texture

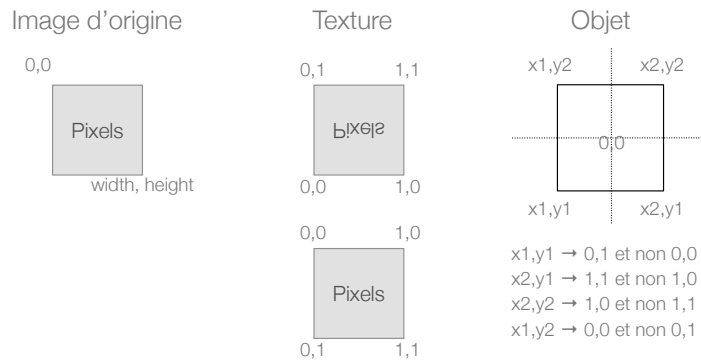
```
glEnable(GL_TEXTURE_2D) ;
```

```
glBegin(GL_QUADS) ;  
  glTexCoord2f(0,0) ;  
  glVertex2f(x1,y1) ;  
  glTexCoord2f(1,0) ;  
  glVertex2f(x2,y1) ;  
  glTexCoord2f(1,1) ;  
  glVertex2f(x2,y2) ;  
  glTexCoord2f(0,1) ;  
  glVertex2f(x1,y2) ;  
glEnd() ;
```



## Attention !

Pour OpenGL, le premier point d'une image est en bas à gauche... mais en général, les données sont organisées de telle sorte que le premier point est en haut à gauche



## Génération automatique des coordonnées

(texgen.c)

Génération à l'aide de la fonction `glTexGen*` selon 3 modes

- `GL_OBJECT_LINEAR`
- `GL_EYE_LINEAR`
- `GL_SPHERE_MAP`

Les deux premiers peuvent être utilisés pour dessiner des contours, le troisième pour l'*environment mapping*



## Quelques détails pratiques

Toutes les coordonnées de texture sont transformées par une matrice à part

Les textures peuvent être 1D, 2D ou 3D

Les données peuvent être stockées en L, A, RGB, RGBA, ...

Plusieurs fonctions permettent de spécifier l'organisation des données de la texture (e.g. `glPixelStorei(GL_UNPACK_ALIGNMENT,a)`)

Sur la plupart des architectures, la quantité de mémoire de texture est fixe, mais on peut souvent mettre plusieurs textures dans une image et on n'est pas obligé d'utiliser toute la place

## Quelques détails pratiques (suite)

Il est possible de demander si une texture particulière peut être chargée en mémoire

```
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGB,
             width, height, 0,
             GL_RGB, GL_UNSIGNED_BYTE, 0);
GLint allocwidth;
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
                        GL_TEXTURE_WIDTH, &allocwidth);
// si tout va bien, width==allocwidth
```

Un certain nombre de textures sont résidentes, ce qui signifie que l'accès à leur données est accéléré par le matériel

Pour savoir si une ou plusieurs textures sont résidentes, utiliser `glGetTexParameter_i` (avec `GL_TEXTURE_RESIDENT`) ou `glAreTexturesResident`

Il est possible d'affecter une priorité entre 0.0 et 1.0 à une ou plusieurs textures avec les commandes `glTexParameterf` (avec `GL_TEXTURE_PRIORITY`) ou `glPrioritizeTextures`



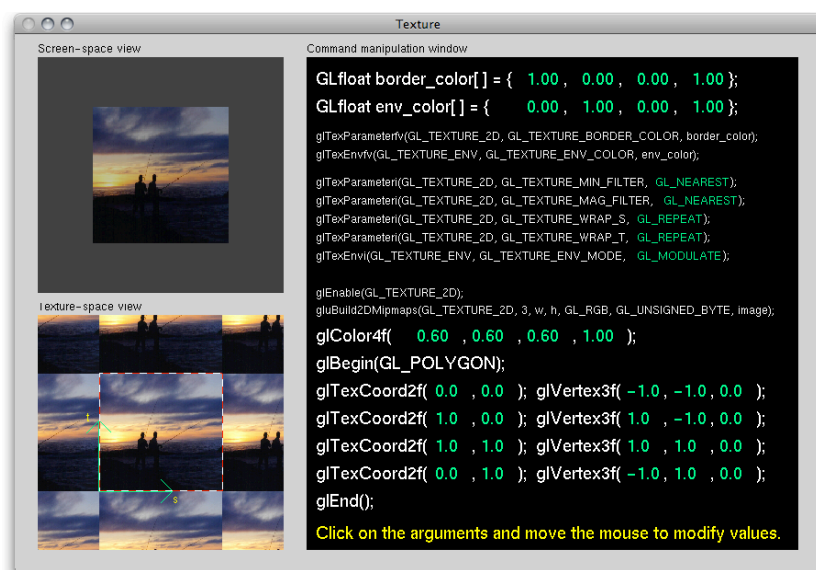
## Quelques détails pratiques (suite)

Plusieurs extensions permettent d'éviter une copie de données

Exemple sur MacOS : `glPixelStorei(GL_UNPACK_CLIENT_STORAGE_APPLE, v)` ;

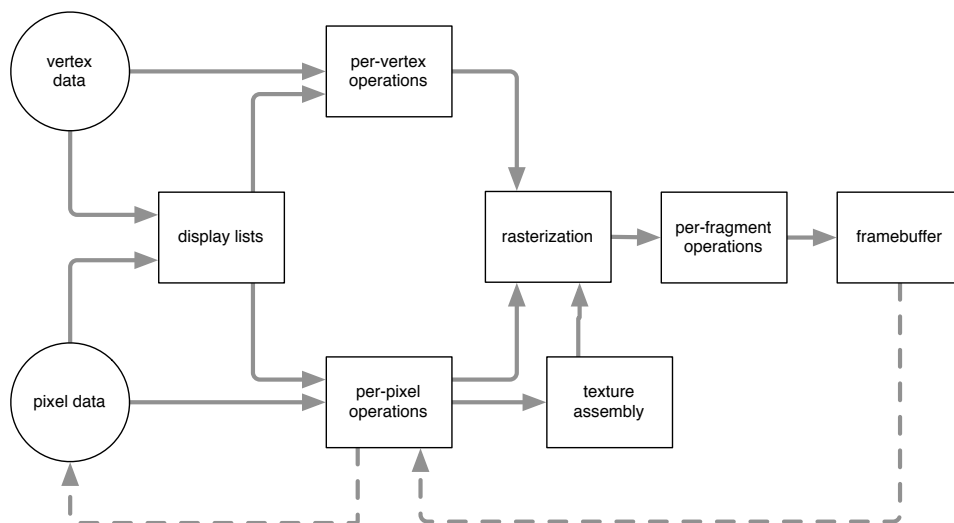
Attention cependant : dans ce cas, la zone mémoire contenant les données doit rester allouée tant que la texture peut être utilisée...

## Tutorial *Texture* de Nate Robins



Du fragment au pixel

Retour sur le *pipeline* OpenGL



## Du fragment au pixel

Fragment

scissor test

alpha test

stencil test

depth test

blending

dithering

logical operations

Pixel (framebuffer)



## *Scissor test*

Un test de clipping additionnel (en plus du viewport et des plans de clipping déjà mentionnés)

Utilisation

```
glScissor(x,y,w,h) ;
```

```
glEnable(GL_SCISSOR_TEST) ;
```

Les coordonnées et dimensions sont en pixels (comme pour glViewport)

Utilisé généralement pour effacer ou mettre à jour une partie de la fenêtre seulement

## *Alpha test*

Permet de rejeter certains fragments en fonction de la valeur de leur composante alpha

### Utilisation

```
glAlphaFunc(GL_GREATER, 0.5) ;  
glEnable(GL_ALPHA_TEST) ;
```

Très pratique pour réduire le nombre de fragments dessinés lorsqu'on a des objets transparents

## *Stencil test*

Permet de faire du clipping non rectangulaire en utilisant un buffer additionnel : le *stencil buffer*  
Généralement utilisé avec une approche multi-passes

### Utilisation

```
glStencilFunc définit le test à effectuer  
glStencilOp décrit comment le stencil buffer est mis à jour en fonction du résultat du test et  
éventuellement du test de profondeur suivant  
glStencilMask détermine si l'on peut écrire dans le stencil buffer  
ne pas oublier glEnable(GL_STENCIL_TEST)...
```

## *Stencil test : exemple*

```
glutInitDisplayMode(...|GLUT_STENCIL|...);  
...  
  
glEnable(GL_STENCIL_TEST);  
glClearStencil(0); // glClear initialisera à 0  
...  
glStencilFunc(GL_ALWAYS, 0x1, 0x1);  
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);  
... // dessine le masque  
glStencilFunc(GL_EQUAL, 0x1, 0x1);  
... // les objets suivants seront dessinés qu'à l'intérieur du  
    // masque
```

## *Depth test*

Test de profondeur utilisant le depth buffer

Utilisation

- glDepthFunc définit la fonction de comparaison
- glDepthMask détermine si l'on peut écrire dans le depth buffer
- glDepthRange pour un contrôle sioux avec les coordonnées écran...
- ne pas oublier glEnable(GL\_DEPTH\_TEST)...

## *Blending*

Permet de combiner les composantes du fragment avec les valeurs courantes du framebuffer

Utilisation

`glBlendFunc` définit la fonction de mélange

`glEnable(GL_BLEND)`

Exemple

`glBlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA) ;`

$d = a*s + (1-a)*d$

## *Dithering*

Permet de simuler un nombre de couleurs plus grand que celui que le matériel peut effectivement afficher

Utilisation

`glEnable(GL_DITHER)`

Remarque : le dithering est actif par défaut... et généralement inutile

## Opérations logiques

Permet de combiner les fragments avec des opérateurs logiques (and, or, xor, not, etc.)

Utilisation : `glLogicOp(mode) ;`

## *Imaging subset*

Extension ARB pour faire du traitement d'image sur les pixels (mais pas sur le résultat de la rasterisation...)

Exemples de traitements

- convolution
- histogrammes
- fonctions avancées de mélange

## Utilisation des menus de GLUT

## Utilisation des menus de GLUT

Pour créer et activer un menu

```
int glutCreateMenu(void (*f)(int value)) ;  
void glutSetMenu(int id) ;
```

Pour ajouter des choses au menu courant

```
void glutAddMenuEntry(char *name, int value) ;  
void glutAddSubMenu(char *name, int menu_id) ;
```

Pour attacher le menu courant à un bouton de la souris

```
void glutAttachMenu(int button) ;
```



## Exemple

(menu.c)

```
typedef enum {RECTANGLE, TEAPOT, SPHERE} obj_type ;

obj_type obj = TEAPOT ;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT) ;

    switch (obj) {
        case RECTANGLE: glRectf(-0.3,-0.3,0.3,0.3) ; break ;
        case SPHERE: glutSolidSphere(0.5, 64, 64) ; break ;
        case TEAPOT: glutSolidTeapot(0.5) ; break ;
    }

    glutSwapBuffers() ;
}

void objMenu(int choice) {
    obj = choice ;
    glutPostRedisplay() ;
}

int main(int argc, char **argv) {
    glutInit(&argc, argv) ;
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE) ;

    glutCreateWindow(argv[0]) ;

    int m = glutCreateMenu(objMenu) ;
    glutSetMenu(m) ;
    glutAddMenuEntry("rectangle",RECTANGLE) ;
    glutAddMenuEntry("sphere",SPHERE) ;
    glutAddMenuEntry("teapot",TEAPOT) ;
    glutAttachMenu(GLUT_RIGHT_BUTTON) ;

    glClearColor(1.0,1.0,1.0,0.0) ;
    glutDisplayFunc(display) ;

    glutMainLoop() ;
    return 0 ;
}
```

Affichage de texte avec GLUT

## glutBitmapCharacter

Affiche un caractère en mode bitmap

Utilisation

```
void glutBitmapCharacter(void *font, int character);  
  
int glutBitmapWidth(void *font, int character)  
  
int glutBitmapLength(void *font, const unsigned char *string)
```

Les fontes sont prédéfinies

- 9\_BY\_15, 8\_BY\_13
- TIMES\_ROMAN\_10, TIMES\_ROMAN\_24
- HELVETICA\_10, HELVETICA\_12, HELVETICA\_18

La position du caractère est déterminée par glRasterPos\*

La boîte englobante est difficile à calculer : la taille des caractères est exprimée en pixels

## glutStrokeCharacter

Affiche un caractère à l'aide d'un ensemble de segments (utilise GL\_LINE\_STRIP)

Utilisation

```
void glutStrokeCharacter(void *font, int character);  
  
int glutStrokeWidth(void *font, int character);  
  
int glutStrokeLength(void *font, const unsigned char *string);
```

Les fontes sont toujours prédéfinies

- ROMAN
- MONO\_ROMAN

L'affichage est plus lent qu'avec les fontes bitmap

La boîte englobante est plus facile à calculer : les dimensions des caractères sont exprimées en unités de la vue modèle (pour MONO\_ROMAN,  $-33,33 < h < 119,05$  et  $w=104,76$ )

## Exemples

(glut-fonts.c)

```
void bCompBox(void *font, const char *message,
              GLfloat *x1, GLfloat *y1,
              GLfloat *x2, GLfloat *y2) {
    *x1 = 0 ;
    *x2 = glutBitmapLength(font,
                           (const unsigned char *)message) ;
    *y1 = 0 ;
    if (font==GLUT_BITMAP_HELVETICA_18)
        *y2 = 18 ;
    else if (font==GLUT_BITMAP_TIMES_ROMAN_24)
        *y2 = 24 ;
    else *y2 = 0 ;
}

void bDispText(void *font, const char *message) {
    int i ;
    for (i=0; i<strlen(message); ++i)
        glutBitmapCharacter(font, message[i]) ;
}
```

```
void sCompBox(void *font, const char *message,
              GLfloat *x1, GLfloat *y1,
              GLfloat *x2, GLfloat *y2) {
    *x1 = 0 ;
    *x2 = glutStrokeLength(font,
                           (const unsigned char *)message) ;
    *y1 = -33.33 ;
    *y2 = 119.05 ;
}

void sDispText(void *font, const char *message) {
    int i ;
    for (i=0; i<strlen(message); ++i)
        glutStrokeCharacter(font, message[i]) ;
}
```

## Quelques remarques

(glut-fonts.c)

Ce qu'on attend de l'affichage du texte (au minimum)

- qu'il soit rapide
- que l'on puisse transformer le texte (translations, rotations, changements d'échelle)
- que l'on puisse connaître la boîte englobante

GLUT ne répond pas vraiment à ces critères

- peu de fontes (2\*1 pour *stroke*, 2+2\*1+3\*1 pour *bitmap*)
- difficile de transformer le texte *bitmap*
- choix nécessaire entre vitesse (*bitmap*) et souplesse (*stroke*)

On peut afficher du texte sans GLUT...

## Affichage de texte sans GLUT

## Affichage de texte avec OpenGL, sans GLUT

Différentes librairies sont disponibles (cf. <http://www.opengl.org/resources/features/fontsurvey>)

Deux façon de dessiner le texte

- dessiner les polygones correspondant à un caractère
- utiliser une texture

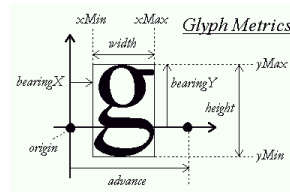
En plus des services de base (large choix de fontes, calcul de la boîte englobante, transformations), on trouve d'autres services

Exemple : extrusion de caractères

**ABC**

## Affichage de glyph

Glyph : dessin d'un caractère



Certaines parties d'un glyph sont plus importantes que d'autre

- les espaces entre les pattes d'un **ℓ** sont plus importants que les pieds des pattes
- la barre d'un **I** et d'un **T** doivent être de même épaisseur

Afin de maximiser la lisibilité, les glyphs ne doivent pas être les mêmes à toutes les échelles

Un moteur de rendu spécialisé est donc nécessaire...

## Exemple #1 : glft

Librairie C++ implémentée il y a quelques années par Stéphane Conversy (ENAC/CENA)

Principes de conception

- descriptions vectorielles des polices (TrueType)
- moteur de rendu spécialisé de haute qualité (FreeType)

Deux modes d'affichage

- textures
- polygones

## glft : affichage d'une ligne de texte

Affichage vectoriel pour les caractères de grande taille et à l'aide d'une texture pour ceux de petite taille

- affichage rapide pour les petites tailles
- économique en mémoire pour les grandes tailles

Affichage d'une ligne de texte

```
std::string message = "informatique graphique" ;  
fontManager->getBoundingBox(&message, &x,&y,&w,&h) ;  
glTranslated(-w/2.0, -h/2.0, 0.0) ;  
fontManager->render (&message) ;
```


glft ne gère pas le formatage (justification, alignement)

## glft : affichage à l'aide d'une texture

On suppose qu'il y a peu de changements de police dans le texte à afficher

Utilisation d'une seule texture pour plusieurs glyphs

- la texture est au format GL\_ALPHA8
- les caractères sont affichés avec des quads avec les coordonnées de texture appropriées

***abcdefghijklmnopqrstuvwxyz***  
***òóôõö×øùúûüýþßääåääæçèé*** 

Remarque : l'antialiasing est réalisé par FreeType au moment de la génération de la texture, pas par OpenGL

## glft : problèmes liés aux textures

Changements d'échelle : quelle taille de caractères utiliser ?

- font size=20, scale=0.5 : la taille apparente est 10
- glft propose un mécanisme d'autoscale

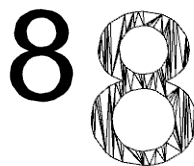
Les coordonnées des quads affichés sont alignées sur les pixels pour éviter des déplacements non voulus

glft cache les textures générées pour pouvoir les réutiliser rapidement par la suite

## glft : affichage vectoriel

Les contours des caractères sont décrits par des droites et des courbes de Bézier

Le tessellateur de GLU est utilisé pour les caractères "à trous" comme le 8 ou le A

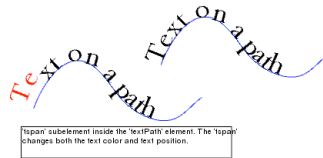


Le programmeur peut utiliser des display lists pour accélérer l'affichage (une par caractère, par paragraphe ou par page par exemple)

Le dessin des caractères peut être anti-aliasé avec la méthode GL\_MULTISAMPLE\_ARB

glft : exemple de rendu svg

Gradient on fill



## Exemple #2 : classes glFontManager, glFont et glString de Núcleo

Núcleo : une librairie C++ pour créer des applications OpenGL+vidéo+réseau

<http://insitu.lri.fr/~rousseau/projects/nucleo/>

### Caractéristiques

- ▶ texte affichable sous forme d'image ou de texture uniquement
- ▶ une texture par glyph

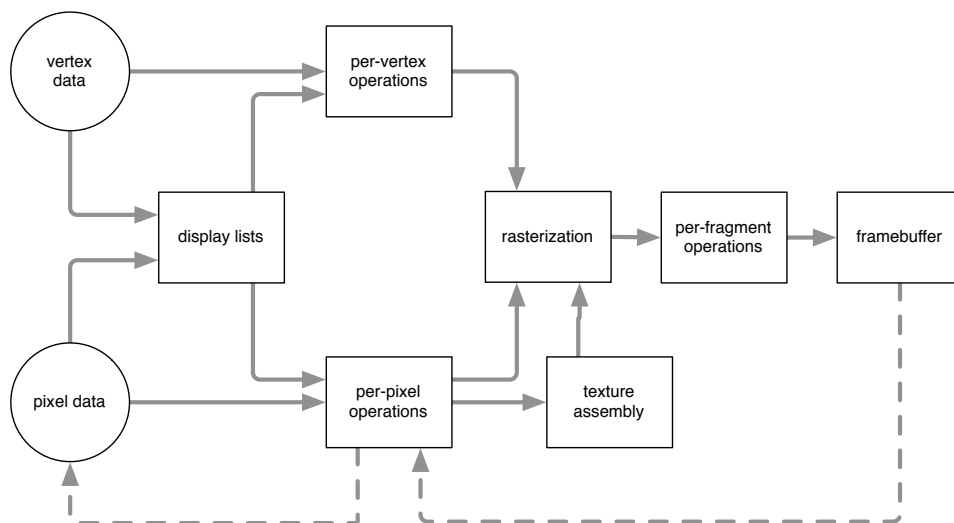
### Exemple

```
glFont *normal = glFontManager::getFont("file:Georgia.ttf?size=24") ;  
glFont *italic = glFontManager::getFont("file:Georgia.ttf?size=24&italic") ;  
glString text ;  
text << normal << "J'aime particulièrement le texte en " << italic << "italique" ;  
text.renderAsTexture() ;
```



*Feedback, selection, picking... et interaction*

Retour sur le *pipeline* OpenGL



## Le mode *feedback*

Le mode *feedback* sert à savoir quelles primitives seront affichées après les étapes de transformation et de *clipping*

Utilisation

```
glFeedbackBuffer(max_size, type, buffer) ;  
  
int size_used = glRenderMode(GL_FEEDBACK) ;
```

La nature des valeur retournées dans buffer dépend du type d'information demandé (coordonnées, couleur, texture)

Chaque ensemble de valeurs est délimité par un code indiquant le type de primitive :  
GL\_POINT\_TOKEN, GL\_LINE\_TOKEN, GL\_POLYGON\_TOKEN, GL\_BITMAP\_TOKEN,  
GL\_DRAW\_PIXELS\_TOKEN, GL\_PASS\_THROUGH\_TOKEN

## Le mode *selection*

Comme le mode *feedback*, le mode *selection* permet de savoir quelles primitives seront affichées après les étapes de transformation et de *clipping*

Mais au lieu de récupérer les données liées à la primitive (coordonnées, couleurs, etc.), seul un nom est renvoyé

Pour identifier une primitive, il suffit de la nommer

- le nom est un entier (un pointeur est un entier...)
- les noms sont placés dans une pile (hiérarchies !)

## Mode *selection* : gestion de la pile de noms

Pour vider la pile

```
gllnitNames() ;
```

Pour remplacer le nom au sommet de la pile

```
glLoadName(name) ;
```

Pour ajouter un nom au-dessus de la pile

```
glPushName(name) ;
```

Pour enlever de la pile le nom au sommet

```
glPopName() ;
```

En mode *selection*, l'affichage de chaque primitive nommée génère un *hit* contenant la liste des noms empilés au moment de l'affichage

## Le *picking*

Le picking est une utilisation spéciale du mode *selection* qui permet de savoir quels objets sont dessinés en un point de l'écran

Usage classique : savoir ce qu'il y a sous le curseur de la souris

Principe

- ▶ on restreint la zone d'affichage à quelques pixels autour du centre d'intérêt
- ▶ on affiche la scène en mode *selection*
- ▶ les primitives proches du centre d'intérêt génèrent des *hits*
- ▶ on analyse les *hits*

## Exemple de mise en œuvre

```
int pickAll(int x, int y, GLuint *buffer, GLuint size) {
    glSelectBuffer(size, buffer) ;

    GLint viewport[4] ;
    glGetIntegerv(GL_VIEWPORT, viewport) ;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPickMatrix( (GLdouble) x, (GLdouble) (viewport[3] - y),
                   2.0, 2.0, viewport ) ;
    ... // transformations de projection (e.g. glOrtho, gluPerspective)

    glMatrixMode(GL_MODELVIEW) ;
    glLoadIdentity() ;
    ... // transformations de vue
    glRenderMode(GL_SELECT) ;
    glInitNames() ;
    ... // dessine la scene en ajoutant les noms sur la pile

    return glRenderMode(GL_RENDER) ;
}
```

## Analyse des résultats

Pour chaque *hit*, le buffer contient

- le nombre de noms dans la pile au moment du *hit*
- *depth value* minimale des vertex touchés
- *depth value* maximale des vertex touchés
- le contenu de la pile de noms

Voir la documentation de `glSelectBuffer` pour les détails...

## Quelques remarques

Le rendu des primitives en mode *selection* peut être différent de celui effectué pour l’affichage

### Exemples

- suppression des textures
- simplification de la géométrie (e.g. boîte englobante)

D’autres techniques peuvent être utilisées à la place du *picking* (e.g. coder l’identité de l’objet dessiné dans sa couleur)

Le *picking* ouvre la voie vers l’interaction...

## Programmation par événements (sans GLUT)

Petite digression...

```
for (bool loop=true; loop; ) {  
    bool refresh = false ;  
    Event e = getNextEvent() ;  
    switch (e.type) {  
        ... // refresh peut être modifié  
    }  
    if (refresh) {  
        glClear(...) ;  
        ...  
        swapBuffers() ;  
    }  
}
```

## Quelques techniques d'interaction courantes

### Translations (*panning*)

- déplacement d'un ou plusieurs objets
- contrôle en X et Y à la souris (en Z avec un modificateur)

### Changements d'échelle (*zoom*)

- agrandissement/réduction d'un ou plusieurs objets
- contrôle à la souris (modificateur pour différencier du *panning*)

### Et les rotations ?

- elles sont difficiles à composer...
- parce qu'elles ne sont pas commutatives

## Angles d'Euler et quaternions

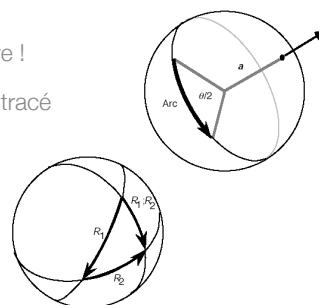
(ext/shoemake-arcball)

### Angles d'Euler : trois angles

- définissent des rotations autour des axes X, Y et Z
- faciles à décrire
- facile à programmer avec OpenGL

### Quaternions

- $w + ix + jy + kz$ ,  $i^2 = j^2 = k^2 = -1$ ,  $ij = k = -ji$
- pas très intuitif... mais idéal pour une manipulation interactive !
- exemple : ArcBall, où une rotation est interprétée comme le tracé d'un arc sur la surface d'une sphère



## En cas d'erreur ?

## En cas d'erreur...

En cas d'erreur... rien ne se passe !

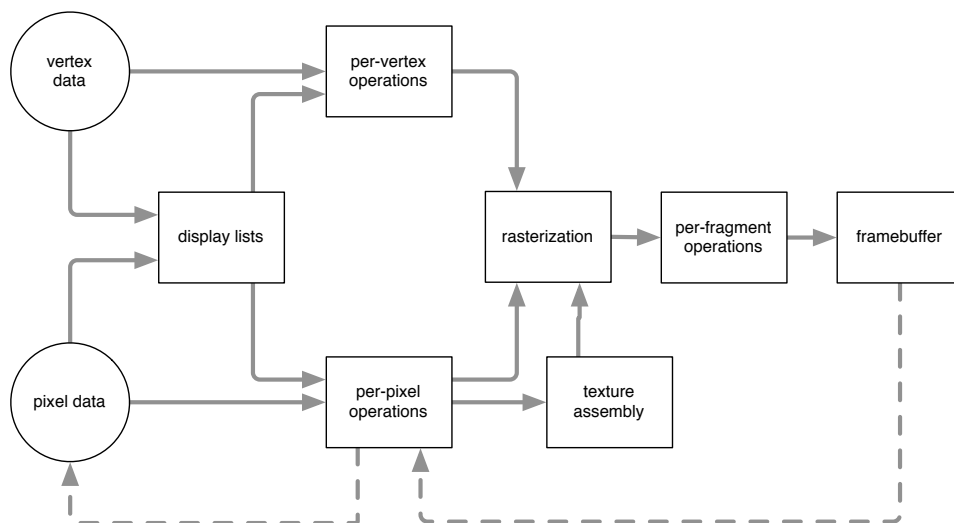
Heureusement, glGetError et gluErrorString sont là pour vous aider

```
void checkError(const char *filename, int line) {
    GLenum error = glGetError();
    if (error!=GL_NO_ERROR) {
        const GLubyte *errstr = gluErrorString(error);
        fprintf(stderr, "%s (%d): %s\n",
                filename, line, (const char *)errstr);
    }
}
```

```
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    checkError(__FILE__, __LINE__);
    glBegin(GL_LINE_LOOP);
    glRotatef(45,0,0,1);
    glVertex2f(-0.5,-0.5);
    glVertex2f(0.5,-0.5);
    glVertex2f(0.5,0.5);
    glVertex2f(-0.5,0.5);
    glEnd();
    checkError(__FILE__, __LINE__);
    glutSwapBuffers();
}
```

*Display lists et vertex arrays*

Retour sur le *pipeline* OpenGL





## Mode immédiat et *display lists*

### Mode immédiat

- les primitives sont envoyées dans le pipeline et affichées immédiatement (ou presque)
- le pipeline n'a aucune mémoire de ce qui l'a traversé

### *Display lists*

- les commandes sont placées dans des listes
- les listes sont conservées par le serveur graphique
- une liste peut être exécutée dans un environnement différent (certaines variables d'état peuvent avoir été changées)

## *Display lists*

Les display lists sont identifiées par des entiers

L'instruction

```
GLint dlist = glGenLists(n) ;
```

crée n display lists [dlist .. dlist+n-1]

Pour compiler (ou recompiler) une liste

```
glNewList(dlist, GL_COMPILE) ; // ou GL_COMPILE_AND_EXECUTE  
... // instructions OpenGL  
glEndList() ;
```

Pour exécuter une liste : `glCallList(dlist) ;`

Pour détruire une liste : `glDeleteLists(dlist, 1) ;`

## *Display lists* : quelques remarques

Certaines instructions ne sont pas valides lorsqu'on est en train de créer une *display list* (glGet et glEnable par exemple)

On ne peut pas créer une *display list* si on est en train d'en créer une... mais une *display list* peut en appeler une autre

Les changements d'état provoqués par l'exécution d'une *display list* persistent après elle (effets de bord)

Quand utiliser une display list ? Dès qu'une même séquence d'instructions va être exécutée plus de n fois

Il vaut mieux éviter

- des listes trop petites
- des hiérarchies trop profondes
- trop de listes

L'utilisation des *display lists* est fortement liée à la structure de données utilisée pour décrire la scène

## *Vertex arrays*

Les *vertex arrays* permettent de spécifier les vertex et leurs attributs par blocs

Exemple d'utilisation : n points décrits chacun par 3 coordonnées et une couleur RGBA

```
glVertexPointer(3, GL_FLOAT, 0, coords) ;  
glEnableClientState( GL_VERTEX_ARRAY) ;  
glColorPointer(4, GL_FLOAT, 0, colors) ;  
glEnableClientState(GL_COLOR_ARRAY) ;  
glDrawArrays(GL_TRIANGLE_STRIP, 0, n) ;
```

Variations sur le thème : glInterleavedArrays, glDrawElements, glArrayElement

Plus vite ! (d'après D. Shreiner, B. Grantham et B. Paul)

## Comment accélérer votre application ?

1. chercher et corriger les erreurs éventuelles (voir transparents précédents)
2. identifier le(s) goulot(s) d'étranglement (*bottleneck*)
  - ▶ au niveau de l'application : les données ne sont pas insérées assez rapidement dans le pipeline d'OpenGL
  - ▶ au niveau des transformations : OpenGL n'arrive pas à transformer vos vertex assez vite (transform-limited)
  - ▶ au niveau de la rasterisation : OpenGL n'arrive pas à dessiner les primitives assez vite (fill-limited)

## Problèmes au niveau de la rasterisation

Diagnostic simple à effectuer : réduire le nombre de pixels à afficher

Solutions simples à mettre en oeuvre

- réduire la taille du viewport
- accepter un frame-rate plus faible

Autres solutions

- n'afficher qu'un seul côté des polygones
- désactiver le *dithering*
- trier les primitives pour rejeter un maximum de fragments avec le *depth test* ou l'*alpha test* (économise une lecture liée à la phase de *blending*)
- utiliser les types et formats de données de texture les plus proches du matériel pour éviter les conversions inutiles, préférer les types non signés

## Problèmes au niveau de la rasterisation (suite)

Autres solutions (suite)

- utiliser les texture objects et `glTexSubImage2D` pour limiter le nombre de copies
- utiliser les priorités de texture
- choisir une taille de texture adaptée à son utilisation

Si rien de tout ça ne marche

- acheter du matériel plus récent...
- utiliser le temps "perdu" pour améliorer la qualité du rendu ou de la simulation

## Problèmes au niveau des transformations

### Diagnostic

- l'application ne va pas plus vite lorsqu'on réduit le *viewport*
- l'application va plus vite si on désactive l'éclairage ou la génération de coordonnées de texture

### Solutions

- utiliser des sources de lumière directionnelles
- éviter les lumières de type spot
- utiliser moins de sources de lumière (pas toutes accélérées ?)
- simuler l'éclairage avec des textures
- éviter `glLoadMatrix*` et `glMultMatrix*` (utiliser `glRotate` et al.)

## Problèmes au niveau de l'application

Problème de transfert entre l'application et OpenGL

Idée pour aider au diagnostic : modifier l'application pour que les données soient toujours transférées mais que rien ne soit dessiné

Comment : remplacer `glVertex*` et `glColor*` par `glNormal*`

### Solutions

- revoir les structures de données utilisées et les types et formats de stockage
- utiliser des debugger/profiler standards

## Problèmes de validation

Un changement d'état peut avoir des conséquences plus profondes que la modification d'une variable booléenne

- activation du moteur d'éclairage
- choix d'un autre mode de rendu (fill, line, etc.)

Le *pipeline* peut être reconfiguré et des caches internes peuvent être invalidés

Les phases de validation sont coûteuses et doivent être évitées

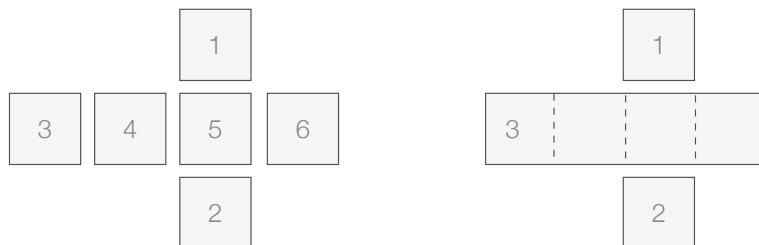
Comment ? En changeant l'état le moins souvent possible

- grouper les primitives par type (*state sorting*, une technique simple mais très efficace)
- éviter les `glPushAttrib/glPopAttrib` (trop de choses modifiées)

Estimation du coût : texture > éclairage > transformations

## Exemple : affichage d'un cube

Il y a quelques années, sur 100 000 cubes, le rapport de performance pouvait aller jusqu'à 2 entre les deux méthodes ci-dessous

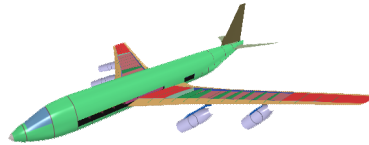


## Autre exemple

1.02M triangles, 507K vertex

*Vertex arrays* (couleur, normale et coords)

Color material



L'application tourne sur une machine capable d'afficher 13M polygones/seconde mais n'affiche qu'une image par seconde...

L'application n'utilise que 18 couleurs différentes mais sélectionne la couleur pour chaque triangle affiché

Une fois les triangles réordonnés par couleur : 6 images/sec

## Quelques bons conseils de Brian Paul

Ne pas croire qu'en développant l'application sur une implémentation logicielle, elle tournera forcément plus vite sur une implémentation matérielle

Envisager l'utilisation de plusieurs *threads*

Savoir trouver un compromis entre la qualité de l'image et la performance (être prêt à accepter de dégrader temporairement la qualité, par exemple)

Faire du *level of detail* : simplifier le modèle des objets éloignés et ne plus les dessiner à partir d'un certain point (facile si on dispose des boîtes englobantes)

Utiliser les versions \*v de glVertex, glColor, glNormal et glTexCoord

Ne pas spécifier des choses qui ne servent à rien, comme les normales si l'éclairage n'est pas activé (ou les coordonnées de texture si le placage de texture n'est pas activé)

Minimiser le code non-OpenGL entre glBegin et glEnd (e.g. sortir les tests hors du bloc)

## Utilisation de l'*accumulation buffer*

### *Accumulation buffer*

Le buffer de couleur a une précision limitée et stocke généralement les couleurs sur des entiers

Le buffer d'accumulation permet de créer des images en additionnant les primitives tout en évitant les problèmes d'*overflow* liés à la faible précision du buffer de couleur

Le buffer d'accumulation est souvent implémenté en logiciel

Exemple d'utilisation : `glAccum(GL_ACCUM, 0.5)` ;

Ici, chaque valeur écrite dans le buffer de couleur est divisée par 2 et ajoutée au buffer d'accumulation



## A quoi ça sert ?

(motionblur.c)

Le buffer d'accumulation peut être utilisé pour mettre en œuvre divers effets

- filtrage d'images
- *motion blur*
- profondeur de champ
- *antialiasing* de toute la scène



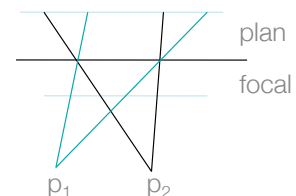
## Profondeur de champ

(depthfield.c)

Le modèle de projection d'OpenGL simule une caméra à tête d'épingle : toute la scène est parfaitement nette

Les lentilles réelles introduisent un effet de profondeur de champ : seuls les objets à une certaine distance sont nets, les autres, plus proches ou plus éloignés, sont flous

L'effet de profondeur de champ peut être créé en accumulant les images calculées à partir de positions proches du point de vue, sur une ligne parallèle au plan focal



## *Antialiasing* par accumulation

(accumaa.c)

Idée similaire à l'effet précédent : bouger légèrement le point de vue et accumuler les images produites



Pour que les effets d'*aliasing* disparaissent, il faut que les déplacements de point de vue soient visible dans l'image résultat

*Antialiasing*

## Antialiasing des points et des lignes

Pour les points

```
glEnable(GL_POINT_SMOOTH) ;  
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST) ;
```

Pour les lignes

```
glEnable(GL_LINE_SMOOTH) ;  
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST) ;
```

Dans les deux cas

```
glEnable(GL_BLEND) ;  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) ;
```

## Antialiasing des polygones

```
glEnable(GL_POLYGON_SMOOTH) ;  
glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST) ;
```

```
glEnable(GL_BLEND) ;  
glBlendFunc(GL_SRC_ALPHA_SATURATE, GL_ONE) ;
```

Important : les polygones doivent être triés *front-to-back* et le contexte doit être RGBA (spécifié lors de la création de la fenêtre)

## ARB Multisampling

Technique utilisée pour l'antialiasing de toute la scène

Mise en oeuvre

- ▶ passer GLUT\_MULTISAMPLE à glutInitDisplayMode
- ▶ glEnable(GL\_MULTISAMPLE\_ARB) ;
- ▶ glHint (GL\_MULTISAMPLE\_FILTER\_HINT\_NV, GL\_NICEST);

Principe : la couleur, la profondeur et les valeurs de stencil sont calculées avec une précision inférieure au pixel et ensuite combinées pour obtenir la valeur des pixels