# OpenGL

K. Bouatouch et R. Caubet*
IRISA, Rennes
IFSIC, Université de Rennes 1
IRIT, Toulouse

## Introduction

**OpenGL est une bibliothèque graphique 3D**

*** Indépendante de tout système de fenêtrage**

*** Comprend des fonctions pour :**

**Définir des objets graphiques simples,**
**Définir des couleurs,**
**Définir les points de vue,**
**Définir les sources de lumière, ....**

## Introduction (Suite)

*** Pas de fonctions de fenêtrage**
*** Pas de gestion des événements**

**Architecture de type "Machine Abstraite" avec des variables d'états accessibles par des fonctions OpenGL : glEnable(), etc.**

**Etat : couleur, caractéristique de la lumière, matrice de transformation, # modes, etc.**

## Introduction (Suite)

**Des bibliothèques construites au dessus d'OpenGL fournissent des fonctions de plus haut niveau :**

- **OpenGL Utility Library GLU glu...()**
- **OpenGL Utility Toolkit Library GLUT glut...()**
- **OpenGL Extension pour Xwindow GLX glx...()**
- **OpenGL for Windows WGL wgl...()**

# Introduction (Suite)

- OpenGL Utility Library GLU
  - routines bas niveau
  - mise en place de matrices de projection
  - maillage et rendu de surfaces
  - etc.

# Introduction (Suite)

- OpenGL Utility Library GLX
  - extension de X Window pour supporter le rendu avec OpenGL
  - les routines GLX utilisent le préfixe *glx*
- Pour Microsoft Windows, les routines WGL
  - fournissent les fenêtres à l'interface OpenGL
  - routines avec préfixe *wgl*

# Introduction (Suite)

- OpenGL Utility Library Toolkit GLUT
  - système de fenêtrage indépendant du système d'exploitation
  - création, affichage
  - gestion d'événements : clavier, souris
  - callbacks, boucle de prise en compte des événements

# Introduction (Suite)

- GLUT : événements
  - **glutReshapeFunction(void (*func)(int w, int h)) :**
    - action à exécuter quand la fenêtre est redimensionnée
  - **glutKeyboardFunc(void (* func)(unsigned char key, int x, int y))**
    - Action à exécuter quand une touche du clavier est enfoncée
  - **glutMouseFunc(void (* func)(int button, int state, int x, int y))**
    - idem mais pour un bouton de la souris (pression ou relâchement)
  - **glutMotionFunc(void (*func)(int x, int y))**
    - appel d'une routine pendant que la souris se déplace tout en appuyant sur un bouton

# Introduction (Suite)

- **GLUT : fenêtres**
  - **glutInit(int *arg, char **argv)**
    - initialise GLUT
  - **glutInitDisplayMode(unsigned int mode)**
    - choix : RGBA ou color index, single ou double buffer, avec ou sans stencil buffer, avec ou sans depth buffer (Z-buffer)
  - **glutInitWindowPosition (int x, int y)**
    - coordonnées écran du coin haut gauche de la fenêtre

# Introduction (Suite)

- **GLUT : fenêtres**
  - **glutInitWindowSize(int width, int size)**
    - taille en pixels de la fenêtre
  - **glutCreateWindow(char *title)**
    - crée une fenêtre avec un nom qui apparaît dans la barre de titre, la fenêtre n'est créée que lorsque glutMainLoop() est appelée

# Introduction (Suite)

- **GLUT : callback pour affichage et exécution**
  - **glutDisplayFunc(void *(func)(void))**
    - événement le plus important, callback d'affichage
  - **glutMainLoop(void)**
    - dernière fonction à appeler dans la fonction main()
    - toutes les fenêtres qui ont été créées sont maintenant affichées
    - et le rendu dans ces fenêtres effectué

# Syntaxe des commandes OpenGL

**Toutes les commandes OpenGL sont préfixées par gl et une lettre majuscule pour chaque mot définissant la commande :**

**glClearColor()**

**OpenGL définit des constantes :**

**GL_COLOR_BUFFER_BIT**

# Syntaxe des commandes OpenGL (suite)

**Certaines commandes peuvent utiliser différents types de paramètres.**
**Deux caractères complètent le nom de la commande :**

**glColor3f()**

**3 indique qu'il y a 3 paramètres,**
**f indique qu'ils sont de type float**

# Syntaxe des commandes OpenGL (suite)

| Suffixe | Type de données | Type C | Type OpenGL |
|---|---|---|---|
| b | 8 bits integer | signed char | GLbyte |
| s | 16 bit integer | short | GLshort |
| i | 32 bit integer | long | GLint, GLsizei |
| f | 32 bit float | float | GLfloat, GLclampf |
| d | 64 bit float | double | GLdouble, GLclampd |
| ub | 8 bit unsigned integer | unsigned char | GLubyte, GLboolean |
| us | 16 bit unsigned integer | unsigned short | GLushort |
| ui | 32 bit unsigned integer | unsigned long | GLuint, GLenum,GLbitfield |

# Syntaxe des commandes OpenGL : exemple

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

void display (void)
{
  glClear(  GL_COLOR_BUFFER_BIT
          | GL_DEPTH_BUFFER_BIT); /* effacer les pixels et le depth buffer*/
  glColor3f(1.0f, 1.0f, 1.0f) ; /* donner une couleur */
  glBegin(GL_QUADS) ; /* afficher un quadrilatère */
        glVertex3f (0.25f, 0.25f, 0.0f);
        glVertex3f (0.75f, 0.25f, 0.0f);
        glVertex3f (0.75f, 0.75f, 0.0f);
        glVertex3f (0.25f, 0.75f, 0.0f);
  glEnd() ;
  glutSwapBuffers() ;
}
```

# Syntaxe des commandes OpenGL : exemple

```
void init (void)
{

/* choix couleur de fond*/

  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

/* initialiser les paramètres de visualisation

  glMatrixMode(GL_PROJECTION); /* matrice de projection */
  glLoadIdentity();
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0); /* projection orthographique */
}
```

# Syntaxe des commandes OpenGL : exemple

```
int main (int argc, char** argv)
{
glutInit(&argc, argv) ;
glutInitDisplay(GLUT_DOUBLE | GLUT_DEPTH
                                    | GLUT_RGB) ;
glutInitWindowSize(250, 250) ;
glutInitWindowPosition(100, 100) ;
glutCreateWindow('coucou ') ;
init () ;
glutDisplayFunction(display) ;
glutMainLoop() ;
return 0 ;
}
```

# Objets graphiques

**Les commandes de tracé d'OpenGL sont limitées aux primitives géométriques simples :**

- ☞ **Points**
- ☞ **Lignes**
- ☞ **Polygones (triangles, quadrilatères, …)**

**La bibliothèque GLUT propose des fonctions pour construire des objets 3D complexes : sphère, tore, cylindre, cône, théière.....**

# Primitives géométriques

**Toutes les primitives géométriques sont décrites en termes de coordonnées de sommets (vertex).**

**Attention aux limitations informatiques**

**Calculs**

**en flottant**

**en entier (pixel)**

# Primitives géométriques (suite)

**Pour dire à OpenGL de créer un ensemble de points, lignes ou polygones, on doit placer les sommets entre :**

```
glBegin(mode);
    .
    .
glEnd();
```

**Avec mode = GL_POINTS, GL_LINES, GL_POLYGONS, GL_TRIANGLES, GL_QUADS, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP**

## Primitives géométriques (Point)

### Points

**Les coordonnées sont toujours données en 3D (en 2D, z=0).**

**OpenGL travaille en coordonnées homogènes**

**(x, y, z, w) si w est différent de zéro alors les coordonnées du point dans l'espace sont (x/w, y/w, z/w)**

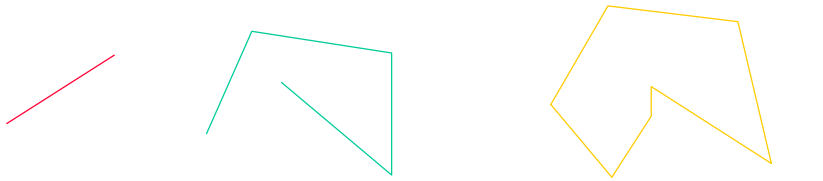
**Commande : glVertex{234}{sidf}[v]()**

**Attribut : glPointSize()**

## Primitives géométriques (Line)

### Lines

**Il s'agit de segments de droite qui peuvent être connectés.**
**Les coordonnées des extrémités sont des sommets.**



**GL_LINES** **GL_LINE_STRIP** **GL_LINE_LOOP**

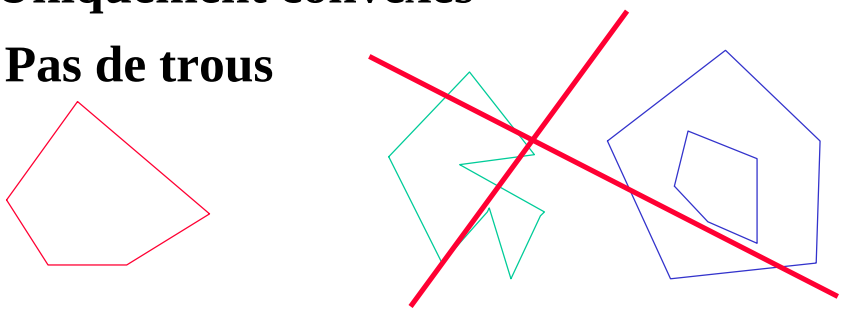
## Primitives géométriques (Polygones)

### POLYGONS

**Boucle fermée de segments de droite**

**Uniquement convexes**

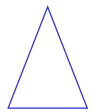
**Pas de trous**



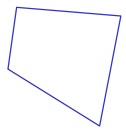
## Primitives géométriques (Polygones)

### TRIANGLES et QUADS

**Polygones convexes simples, utilisant 3 et 4 sommets.**



**Conseillés car les polygones quelconques ont besoin d'une triangulation coûteuse.**

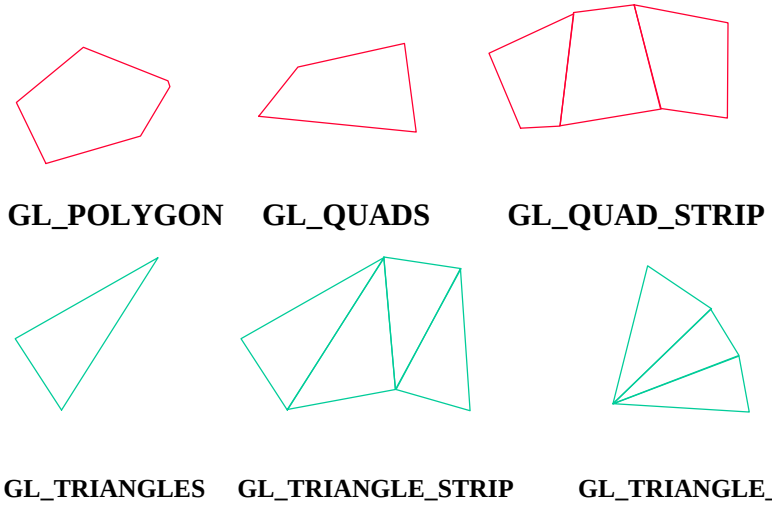


# Exemples

```
glBegin(GL_POINTS)
   glVertex2f(0.0f, 0.0f);
   glVertex2f(0.0f, 4.0f);
   glVertex2f(4.0f, 4.0f);
   glVertex2f(6.0f, 2.0f);
   glVertex2f(4.0f, 0.0f);
glEnd();
```

```
glBegin(GL_POLYGON)
   glVertex2f(0.0f, 0.0f);
   glVertex2f(0.0f, 4.0f);
   glVertex2f(4.0f, 4.0f);
   glVertex2f(6.0f, 2.0f);
   glVertex2f(4.0f, 0.0f);
glEnd();
```

# Polygones (suite)




# Polygones (suite)

**Un polygone a deux faces : avant et arrière**

**Très important pour les objets solides**

**Par défaut les faces avant et arrière sont tracées de la même manière**

**Pour changer : glPolygonMode(face, mode)**

face : **GL_FRONT, GL_BACK, GL_FRONT_AND_BACK**

mode : **GL_POINT, GL_LINE, GL_FILL**

# Polygones (suite)

**Par convention les polygones dont les sommets sont donnés dans le sens trigo sont appelés "front-facing"**

**On peut changer cette convention :**

**glFrontFace(mode)**

**mode : GL_CCW, GL_CW**

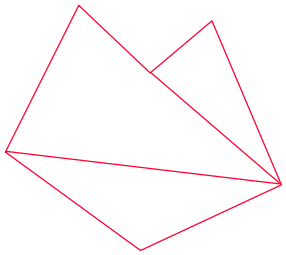
**glCullFace(mode) pour éliminer les faces**

mode : **GL_FRONT, GL_BACK**

**glEnable(GL_CULL_FACE)**
**glDisable(GL_CULL_FACE)**

## Polygones (suite)

**Pour afficher des polygones non convexes, il faut les subdiviser en polygones convexes, généralement des triangles.**
**On ne peut pas utiliser glPolygonMode() pour tracer les cotés.**



## Polygones (suite)

**glNormal3{bsidf}[v]()**

**associe une normale aux sommets et précède glVertex()**

**glColor{34}{bsidf}[v]()**

**associe une couleur aux sommets et précède glVertex()**

## Vertex Arrays

**glBegin() … glEnd()**
**correspond au *mode immédiat* : chaque sommet, normale, … est défini en appelant une fonction depuis l'application.**
**→Peu performant, trop d'appels de fonctions**
**→Les données doivent être regroupées en tableaux**
**→Utilisation de *vertex arrays***

## Vertex Arrays

1) **Spécification des pointeurs sur les données**
2) **Activation de l'utilisation des vertex arrays (il faut activer chaque type de donnée séparément (sommet, normale, …))**
3) **Rendu**
4) **Désactivation des vertex arrays**

# Vertex Arrays

**glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);**

***size* est le nombre de composantes du sommet (2 pour XY, 3 pour XYZ)**
***type* est le type des données (GL_FLOAT en général)**
***stride* est le décalage en nombre d'octets entre chaque sommet (0 en général)**
***pointer* est le pointeur sur les données**

**Autres fonctions disponibles :**

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

# Vertex Arrays

**glEnableClientState(GL_VERTEX_ARRAY);**

**Active l'utilisation des vertex arrays (tableaux de sommets). Chaque type de tableau doit être activé séparément:**

**glEnableClientState(GL_COLOR_ARRAY);**
**glEnableClientState(GL_NORMAL_ARRAY);**
**glEnableClientState(GL_TEXTURE_COORD_ARRAY);**
**glEnableClientState(GL_EDGE_FLAG_ARRAY);**

**Pour désactiver l'utilisation des vertex arrays:**

**glDisableClientState(GL_VERTEX_ARRAY);**
**…**

# Vertex Arrays

**glDrawArrays(GLenum mode, GLint first, GLsizei count);**

**Utilise le contenu des vertex arrays séquentiellement pour tracer des primitives**

***mode* est le type de primitive à rendre (GL_TRIANGLES, …)**
***first* est l'index du premier sommet à utiliser**
***count* est le nombre de sommets à utiliser (multiple de 3 par exemple pour les triangles)**

# Vertex Arrays

**glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);**

**Lit une liste d'indices pour spécifier quels sommets utiliser pour tracer une série de primitives (plus rapide que glDrawArrays() car les sommets peuvent être utilisés plusieurs fois → les calculs d'éclairage sont donc moins coûteux et une mise en cache est possible)**

***mode* est le type de primitive à rendre (GL_TRIANGLES, …)**
***count* est le nombre d'indices (multiple de 3 par exemple pour les triangles)**
***type* est le type des indices (GL_UNSIGNED_SHORT ou GL_UNSIGNED_INT)**
***indices* est un tableau d'indices. Exemple (pour 2 triangles) :**

**const float indices[2*3] = { 0, 1, 2,  2, 1, 3 };**

# Vertex Arrays

```
float vertices[3*2] = { 0.0f, 0.0f,     0.0f, 4.0f,     4.0f, 4.0f };
float colors[3*3] = { 1.0f, 0.0f, 0.0f,     0.0f, 1.0f, 0.0f,     0.0f, 0.0f, 1.0f };

glVertexPointer(2, GL_FLOAT, 0, vertices);        // donner les pointeurs
glColorPointer(3, GL_FLOAT, 0, colors);           // sur les données
glEnableClientState(GL_VERTEX_ARRAY);     // activer vertex arrays
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, 3);   // afficher des triangles
                                    // en utilisant (ici) 3 sommets
                                    // en commençant par 0

glDisableClientState(GL_VERTEX_ARRAY);     // désactiver vertex arrays
glDisableClientState(GL_COLOR_ARRAY);
```

# Vertex Buffer Objects (VBOs)

**Les vertex arrays sont stockés en mémoire vive**
**→le transfert vers la carte graphique de chaque primitive pour le rendu est coûteux, la bande passante du bus est limitée**
**→les tableaux de données devraient être stockés dans la carte graphique**
**→utilisation des VBOs**
**→légère modification par-dessus les vertex arrays**

# Vertex Buffer Objects (VBOs)

1) **Création d'un VBO**
2) **Bind du VBO (pour dire qu'il devient le VBO courant)**
3) **Allocation de mémoire dans la carte graphique**
4) **Copie des données des sommets dans la carte graphique**

**→ Le rendu s'effectue ensuite comme avec des vertex arrays sauf que les pointeurs vers les données sont remplacés par des décalages. On utilise la macro suivante à la place du pointeur dans *glVertexArray()* :**

```
#define BUFFER_OFFSET(i)        ((char *)NULL + (i))
```

**avec i le décalage en nombre d'octets des données à l'intérieur du VBO**

# Vertex Buffer Objects (VBOs)

```
GLuint VBOhandle;
glGenBuffers(1, &VBOhandle);                          // Création du VBO
glBindBuffer(GL_ARRAY_BUFFER, VBOhandle);             // Bind du VBO

glBufferData(GL_ARRAY_BUFFER,                         // Allocation de mémoire
             (3*2+3*3)*sizeof(float),                 // pour les données du VBO
             0, GL_STATIC_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, 0,                   // Copie des coordonnées
                (3*2)*sizeof(float), vertices);       // des sommets

glBufferSubData(GL_ARRAY_BUFFER, (3*2)*sizeof(float),
                (3*3)*sizeof(float), colors);         // Copie des couleurs
                                                      // des sommets
```

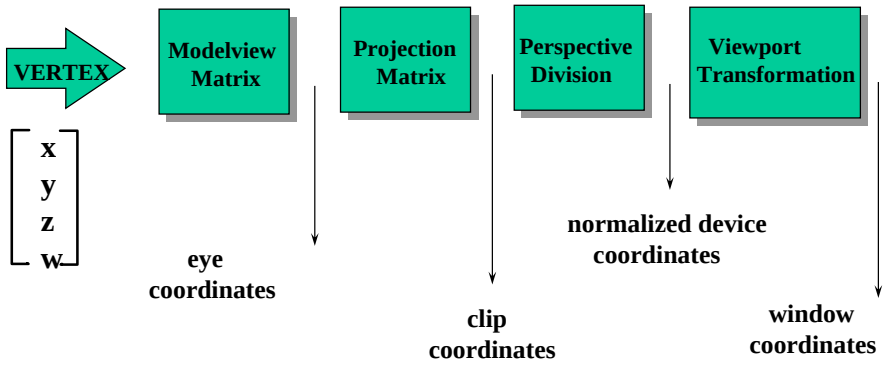
# Vertex Buffer Objects (VBOs)

```
glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(0));
glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET((3*2)*sizeof(float)));

glEnableClientState(GL_VERTEX_ARRAY);          // activer vertex arrays
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, 3);              // afficher des triangles
                                               // en utilisant (ici) 3 sommets
                                               // en commençant par 0

glDisableClientState(GL_VERTEX_ARRAY);         // désactiver vertex arrays
glDisableClientState(GL_COLOR_ARRAY);
```

# Visualisation

## Les repères utilisés sont directs

## Pipeline de visualisation




# Visualisation (pipe line)

**Transformation de Modélisation et de Visualisation**

Place les objets et la caméra (position et direction)

**Transformation de Projection**

Spécifie la forme et l'orientation du volume de vision et le type de projection

**Transformation écran (viewport)**

Transforme les coordonnées 3D en coordonnées écran

**Remarques :**

Utilisation des matrices de transformations :v' = Mv
Les matrices de visualisation et de modélisation sont appliquées aux sommets et aux normales

# Visualisation (Exemple)

```
# include ....
void display (void)
{
  glClear (  GL_COLOR_BUFFER_BIT
           | GL_DEPTH_BUFFER_BIT);
  glColor3f (1.0f, 1.0f, 1.0f);
  glLoadIdentity ();  /* clear the modelview matrix*/
  glTranslatef (0.0f, 0.0f, -5.0f) ; /*viewing transfo*/
  glScalef (1.0f, 2.0f, 1.0f); /*modeling transfo*/
  glutWireCube (1.0); /*draw the cube*/
  glutSwapBuffers ();
}
```

## Visualisation (Exemple)

```
void myinit (void)
{
  glshadeModel (GL_FLAT);
}

void myreshape (GLsizei w, Glsizei h)
{
  glViewPort (0, 0, w, h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
  glMatrixMode (GL_MODELVIEW);
}
```

## Visualisation (Exemple)

```
int main(int argc, char** argv)
{
 glutInit(&argc, argv);
 glutInitDisplayMode( GLUT_DOUBLE
                      | GLUT_RGBA | GLUT_DEPTH);
 glutInitPosition(0, 0, 500, 500);
 glutInitWindow(argv[0]);
 myinit();
 glutReshapeFunc(myReshape);
 glutDisplayFunc(myDisplay);
 glutMainLoop();
}
```

## Transformations (Généralités)

**Ceux sont des transformations de repère.**

### glMatrixMode(mode)

**Indique quel type de matrice on va utiliser**

**mode :**
**GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE**

### glLoadIdentity()

**Charge la matrice courante avec une matrice identité**

## Transformations (Généralités)

### glLoadMatrix{fd}(m)

**Charge la matrice m dans la matrice courante c**

### glMultMatrix{fd}(m)

**Multiplie la matrice m par la matrice courante c**

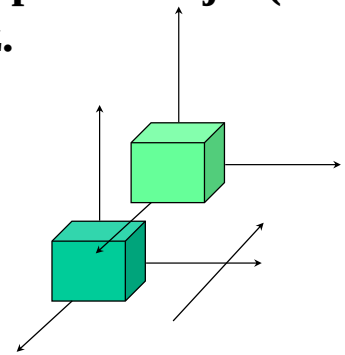
**Résultat : m = cm**

**Rque : En C m[4][4] (ligne, colonne)**
**En OpenGL c'est l'inverse**
**Solution : m[16]**

# Modélisation (Modeling Transformations)

## glTranslatef(x, y, z)

**Multiplie la matrice courante par une matrice qui déplace l'objet (ou son repère local) de x, y, z.**



# Modélisation (Modelling Transformations)

## glRotate{fd}(angle, x, y, z)

**Multiplie la matrice courante par une matrice qui applique une rotation à l'objet (ou son repère local) selon un axe donné (x, y, z) et d'un angle donné en degrés.**

## glScale{fd}($s_x$, $s_y$, $s_z$)

**Permet de faire une mise à l'échelle, ou des symétries.**

# Visualisation (Viewing Transformations)

**Position initiale de la caméra : (0.0f, 0.0f, 0.0f)**
**Axe de visée vers les z négatifs**

## glLoadIdentity()

**Charge la matrice identité dans la matrice courante**

## glTranslatef(position)

**Positionne la caméra en x, y, z**

## glRotatef(rotation, axe)

**Oriente la caméra**

**Rque : Les transformations de visualisation doivent être appelées avant les transformations de modélisation.**

# Visualisation (Viewing Transformations)

## gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)

**Fonction qui gère toutes les transformations de la caméra d'un seul coup.**
**up indique l'orientation du volume de vision (haut de l'image)**

# Projections (Perspective)

**Ne pas oublier d'appeler :**

**glMatrixMode(GL_PROJECTION);**

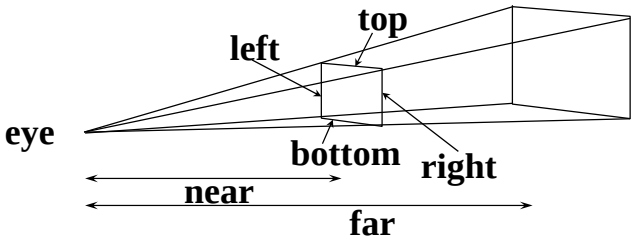
**glLoadIdentity();**

**(left,bottom,-near) : coin BG du plan clip proche**

**(right,top,-near) : coin HD du plan clip proche**

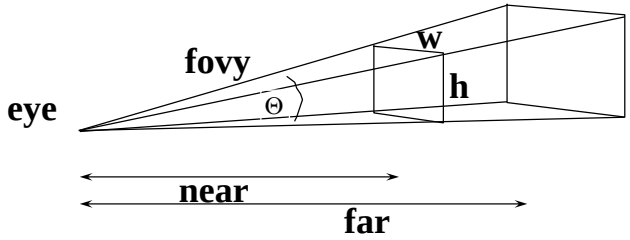
**Perspective**

**glFrustum(left, right, bottom, top, near, far)**




# Projections (Perspective)

**gluPerspective(fovy, aspect, zNear, zFar)**




**aspect=w/h**

**Comme pour glFrustum, on peut appliquer des translations et rotations pour changer l'orientation par défaut du volume de vision.**

# Projections (Orthographique)

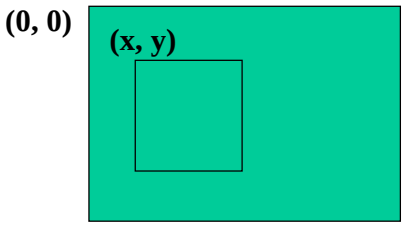
**glOrtho(left, right, bottom, top, near, far)**

**Crée une matrice pour un volume de vision parallélépipédique et la multiplie par c.**

**glOrtho2D(left, right, bottom, top)**

**Projection d'un objet 2D sur un écran**

# Transformations (Viewport)




**Window**

**glViewport(x, y, width, height)**

**Par défaut le viewport a la taille de la fenêtre**

**Le ratio du viewport doit être le même que celui du volume de vision, sinon déformations.**

# Matrices (stacks)

Les matrices de projection et de visualisation créées précédemment occupent le sommet de pile.

Les piles de matrices sont utiles pour construire des modèles hiérarchiques.

Les opérations matricielles composent avec la matrice courante ou le sommet de pile

# Matrices (stacks)

Il existe des piles de matrices différentes :

- Modelview Matrix Stack
  Cette pile contient 32 éléments ou plus. Initialement, le sommet de pile contient la matrice identité vue comme matrice c.

- Projection Matrix Stack
  Cette pile est utilisée surtout pour sauvegarder temporairement la matrice

# Matrices (stacks)

Opérations sur les piles :

glMatrixMode() détermine le type de matrice : ModelView ou Projection

## glPushMatrix()

Recopie le sommet de pile, effectue les multiplications de matrices et place le résultat à ce nouveau sommet.


# Matrices (stacks)

## glPopMatrix()


Supprime le sommet de pile. La matrice courante devient le nouveau sommet de pile.

# Matrices (Exemple)

```
draw_wheel_and_bolts()
{
        long i;
        draw_wheel();
        for (i=0;i<5;i++)
         {
          glPushMatrix();
               glRotatef(72.0f*i, 0.0f, 0.0f, 1.0f);
               glTranslatef(3.0f, 0.0f, 0.0f);
               draw_bolt();
          glPopMatrix();
         }
 }
```

# Matrices (Exemple) (suite)

```
draw_body_and_wheel_and_bolts()
{
        draw_car_body();
        glPushMatrix();
           glTranslatef(30.0f, 0.0f, 20.0f); /* position roue*/
           draw_wheel_and_bolts();
        glPopMatrix();

        glPushMatrix();
           glTranslatef(30.0f, 0.0f, -20.0f); /* position roue*/
           draw_wheel_and_bolts();
        glPopMatrix();
.....
}
```

# GLUT

## Quelques commandes utiles

### Fenêtres

```
glutInitWindow(*titre)
glutInitDisplayMode(mask)  /* OU logique */
   mask = GLUT_RGBA, GLUT_INDEX, GLUT_SINGLE,
          GLUT_DOUBLE, GLUT_DEPTH, GLUT_STENCIL,
          GLUT_ACCUM
glutInitPosition(x, y, width, height)
```

### Evénements

```
glutReshapeFunc ( (*function)(GLsizei,Glsizei) )
glutKeyboardFunc (void(*func)(unsigned int key, int x, int y))
glutMouseFunc (void (*func)(int button, int x, int y))
```

# GLUT (suite)

### Objets 3D

```
glutWireSphere(radius)
glutSolidSphere( adius)

glutWireCube(edge)
glutSolidCube(edge)

glutWireBox(width, height, depth)
glutSolidBox(width, height, depth)
```

# GLUT (suite)

**glutWireCylinder(radius, height)**
**glutSolidCylinder(radius, height)**

**glutWireTorus(innerRadius, outerRadius)**
**glutWireTorus(innerRadius, outerRadius)**

**glutWireIcosahedron(radius)**
**glutSolidIcosahedron(radius)**

**glutWireOctahedron(radius)**
**glutSolidOctahedron(radius)**

**glutWireTetrahedron(radius)**
**glutSolidTetrahedron(radius)**

# GLUT (suite)

**glutWireDodecahedron(radius)**
**glutSolidDodecahedron(radius)**

**glutWireCone(radius, height)**
**glutSolidCone(radius, height)**

**glutWireTeapot(size)**
**glutSolidTeapot(size)**

## Traitement de fond et exécution

**glutIdleFunc(*function)**
**glutMainLoop()**

# L'éclairage

# Eclairage (Lighting)

**OpenGL permet de manipuler l'éclairage et les objets de la scène pour produire différents effets.**
**La lumière est définie par 3 composantes :**

- **rouge**
- **vert**
- **bleu**

**Les matériaux composant les surfaces sont caractérisés par 3 réflectances (RVB) comprises entre 0 et 1.**

## Eclairage (Lighting)

**Quatre composantes dans les modèles d'éclairage :**

**- lumière émise par un objet et non affectée par une source de lumière**

**- lumière ambiante provenant d'une source dans toutes les directions**

**- lumière diffuse provenant d'une seule direction et réfléchie dans toutes les directions lorsqu'elle rencontre un objet**

**- lumière spéculaire venant d'une direction particulière et se réfléchissant selon une direction privilégiée**

## Couleur des matériaux (Material Colors)

**Les matériaux composant les surfaces sont caractérisés par le pourcentage des 3 composantes rouge, vert, bleu qu'ils réfléchissent.**

**Une balle rouge éclairée par une lumière blanche absorbe le vert et le bleu.**

**Comme pour les lumières, les matériaux ont différentes types de couleur (ambient, diffus, spéculaire) qui déterminent leurs réflectances.**

## Couleur des matériaux (Material Colors)

**Les composantes de la lumière (ambiante, diffuse et spéculaire) sont composées avec les réflectances des matériaux.**

**Les réflectances ambiante et diffuse déterminent la couleur du matériau.**

**La réflectance spéculaire est généralement de la couleur de l'intensité spéculaire de la source.**

## Valeurs RGB

**Composantes des sources de lumière :**
**LR, LV, LB**

**Réflectances des matériaux :**
**MR, MV, MB**

**Lumière arrivant à l'œil :**
**(f(LR,MR), f(LV,MV), f(LB,MB))**

# Sources de lumière (Création)

**Caractéristiques des sources :**

- Couleur
- Position
- Direction

**Spécifiées par :**

**glLight{if}[v](light, pname, param)**

**light peut être : GL_LIGHT0 .. GL_LIGHT7**

**pname définit les caractéristiques de la source**

**param donne les valeurs de ces caractéristiques**

# Sources de lumière (Création)

| pname | param par défaut | résultat |
|---|---|---|
| GL_AMBIENT | (0.0, 0.0, 0.0, 1.0) | intensité ambiante de la source |
| GL_DIFFUSE | (1.0, 1.0, 1.0, 1.0) | intensité diffuse de la source |
| GL_SPECULAR | (1.0, 1.0, 1.0, 1.0) | intensité spéculaire de la source |
| GL_POSITION | (0.0, 0.0, 0.0, 1.0) | position (x, y, z, w) de la source |
| GL_SPOT_DIRECTION | (0.0, 0.0, -1.0) | direction du spot |
| GL_SPOT_EXPONENT | 0.0 | exposant du spot |
| GL_SPOT_CUTOFF | 180.0 | angle d'ouverture |
| GL_CONSTANT_ATTENUATION | 1.0 | facteur d'atténuation constant |
| GL_LINEAR_ATTENUATION | 0.0 | facteur d'atténuation linéaire |
| GL_QUADRATIC_ATTENUATION | 0.0 | facteur d'atténuation quadratique |

**Les valeurs par défaut de GL_DIFFUSE, GL_SPECULAR s'appliquent à GL_LIGHT0 seulement**

# Sources de lumière (Couleur)

**OpenGL permet d'associer différentes couleurs à GL_AMBIENT, GL_DIFFUSE et GL_SPECULAR**

**Exemple :**

```
GLfloat light_ambient []={0.0, 0.0, 1.0,1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

# Sources de lumière (Position et atténuation)

## Position

```
GLfloat light_position []={0.0, 0.0, 1.0, 1.0}
glLightfv(GL_LIGHT0, GL_POSITION, light_position)
```

## Atténuation

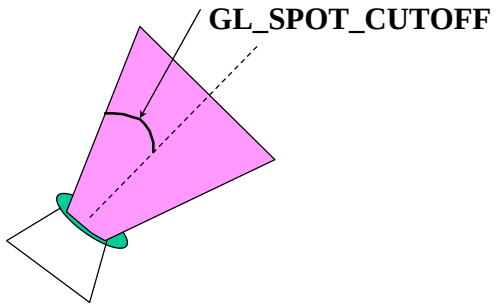
```
glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0)
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0)
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5)
```

**Facteur d'atténuation :** $\frac{1}{k_c + k_l d + k_q d^2}$

**d distance de la source aux sommets**

# Sources de lumière (Spot)

## Emission de la lumière selon un cône




```
glLightf (GL_LIGHT0, GL_SPOT_CUTOFF, 45.0)
GLfloat spot_direction []={0.0, 0.0, 1.0,1.0}
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction)
```

# Sources de lumière (Spot)

## Spotlight Effect

```
glLightf (GL_LIGHT0, GL_SPOT_CUTOFF, 45.0)
GLfloat spot_direction []={0.0, 0.0, 1.0,1.0}
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction)
```

- 1 if the light is not a spot light
- 0 if the light is spotlight but the vertex lies outside the cone
- (max {v. d, 0})GL_SPOT_EXPONENT
- v = unit vector that points from the spotlight (GL_POSITION) to the vertex
- d = direction of the spotlight (GL_DIRECTION)

# Sources de lumière (contrôles)

Comme pour les objets, les sources de lumière sont sujettes aux matrices de transformations. La position et la direction sont affectées par la matrice MODELVIEW courante et stockées en coordonnées caméra.

La position de la source peut être fixe, se déplacer autour d'un objet fixe ou suivre le point de vue.

Attention aux piles de matrices

# Modèle d ’illumination

Les modèles d'éclairage ont 3 composantes :

- intensité ambiante globale
- position du point de vue (local ou à l'infini)
- exécution des calculs selon le type de face (avant ou arrière)

```
GLfloat lmodel_ambient[]={0.1f, 0.1f, 0.1f, 1.0f}
glLightModel{fv}(GL_LIGHT_MODEL_AMBIENT,
                 lmodel_ambient)
```

Permet de spécifier une composante ambiante ne provenant pas d'une source particulière

# Modèle d'éclairage (suite)

**Position du point de vue**

**glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE) ; Sinon observateur à l'infini**

**Type de face**

**glLighti(LIGHT_MODEL_TWO_SIDE, GL_TRUE)**

**Activer, Désactiver l'éclairage**

**glEnable(GL_LIGHTING), glEnable(GL_LIGHT0)**

**glDisable(GL_LIGHTING)**
**glDisable(GL_LIGHT0)**

# Propriétés des matériaux

**La plupart des propriétés des matériaux sont semblables à celles des sources de lumière.**
**La commande utilisée est :**

**glMaterial{if}[v](face, pname, param)**

| pname | param par défaut | résultat |
|---|---|---|
| GL_AMBIENT | (0.2, 0.2, 0.2, 1.0) | couleur ambiante du matériau |
| GL_DIFFUSE | (0.8, 0.8, 0.8, 1.0) | couleur diffuse |
| GL_SPECULAR | (0.0, 0.0, 0.0, 1.0) | couleur spéculaire |
| GL_AMBIENT_AND_DIFFUSE | | |
| GL_SHININESS | 0.0 | exposant spéculaire |
| GL_EMISSION | (0.0, 0.0, 0.0, 1.0) | couleur émissive du matériau |
| GL_COLOR_INDEX | (0, 1, 1) | indice de la couleur ambiante, diffuse ou spéculaire |

```
Glfloat mat_amb []={0.1f, 0.5f, 0.8f, 1.0f};
glMaterial*(GL_FRONT, GL_AMBIENT, mat_ambient);
```

# Pile d'Attributs (stack attribute)

**Les variables d'états peuvent être sauvegardées et restaurées dans une pile avec les commandes**

**glPushAttrib(mask) et glPopAttrib()**

**Exemple :**
**glPushAttrib(GL_LIGHTING_BIT);**
**fait référence aux variables d'états concernant l'éclairage : couleur des matériaux, intensité émise, ambiante, diffuse et spéculaire de la lumière, une liste de sources actives, la direction des spots lumineux.**

# Pile d'Attributs (stack attribute)

| Mask | Attribute Group |
|---|---|
| GL_ACCUM_BUFFER_BIT | accum-buffer |
| GL_ALL_ATTRIB_BITS | |
| GL_COLOR_BUFFER_BIT | color-buffer |
| GL_CURRENT_BIT | current |
| GL_DEPTH_BUFFER_BIT | depth-buffer |
| GL_ENABLE_BIT | enable |
| GL_EVAL_BIT | eval |
| GL_FOG_BIT | fog |
| GL_HINT_BIT | hint |
| GL_LIGHTING_BIT | lighting |
| GL_LINE_BIT | line |
| GL_LIST_BIT | list |
| GL_PIXEL_MODE_BIT | pixel |
| GL_POINT_BIT | point |
| GL_POLYGON_BIT | polygon |
| GL_POLYGON_STIPPLE_BIT | polygon-stipple |

# Lissage

**Une ligne ou un polygone peut être tracé d'une seule couleur (flat shading) ou de différentes couleurs obtenues par lissage (smooth shading appelé aussi Gouraud shading)**

**glShadeModel(mode)**

**mode = GL_FLAT, GL_SMOOTH**

# Le modèle d 'illumination complet

**Toutes les équations mathématiques sont exécutées pour chaque composante R V B**

**La couleur résultante en un sommet est :**

**L'émission du matériau au sommet**

**+**

**la lumière ambiante globale et celle due au matériau**

**+**

**les composantes ambiante et spéculaire du matériau**

# Le modèle d 'illumination complet

**L = Light direction (from the point light source to the vertex**

**H = Normalized vector resulting from the summation of the light direction vector and the viewing direction vector (half-angle vector)**

# Le modèle d 'illumination complet: pour R,V,B

**Intensity of a vertex =** $\text{emission}_{mat}$ +

$\text{ambient}_{light\text{-}model} * \text{ambient}_{mat}$ +

$$\sum_{1=0}^{n-1} \left\{ \frac{1 \bullet}{k_c + k_l d + k_q d^2} \right\}$$ ***(spotlight effect)$_i$ ***

$[\text{ambient}_{light} * \text{ambient}_{mat}$ +

$(\max\{L \bullet n, 0\} * \text{diffuse}_{light} * \text{diffuse}_{mat}$ +

$(\max\{H \bullet n, 0\})^{shininess} * \text{specular}_{light} *$

$\text{specular}_{mat}]_i$

# Mélange et Anticrênelage

## Mélange (blending)

- RGBA : A est appelé Alpha
- Alpha spécifié avec : glColor() ou glClearColor()
- Si mélange : Alpha utilisé pour combiner la couleur du fragment traité (source) avec le pixel correspondant déjà stocké (destination) dans la mémoire d'image (framebuffer)

## Mélange

- Comment calculer les facteurs de mélange source et destination : quadruplets RGBA ?
- Facteurs source : $(S_r, S_g, S_b, S_a)$
- Facteurs destination : $(D_r, D_g, D_b, D_a)$
- valeurs source : $( R_s, G_s, B_s, A_s )$
- valeurs destination : $( R_d, G_d, B_d, A_d )$
- Valeurs finales après mélange :

$$( R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a )$$

## Mélange

- glEnable(GL_BLEND) ; permettre le mélange
- glDisable(GL_BLEND) : ne pas le considérer
- glBlendFunc(Glenum sfactor, Glenum dfactor) ;
  - sfactor : comment calculer les facteurs source
  - dfactor : comment calculer les facteurs destination
- facteurs de mélange : compris entre entre 0 et 1

# Mélange

| constante | Facteur | facteur de mélange calculé |
|---|---|---|
| GL_ZERO | source ou destination | (0, 0 , 0, 0) |
| GL_ONE | source ou destination | (1, 1, 1, 1) |
| GL_DST_COLOR | source | $(R_d, G_d, B_d, A_d)$ |
| GL_SRC_COLOR | destination | $(R_s, G_s, B_s, A_s)$ |
| GL_ONE_MINUS_DST_COLOR | source | $(1,1,1,1) - (R_d, G_d, B_d, A_d)$ |
| GL_ONE_MINUS_DST_COLOR | destination | $(1,1,1,1) - (R_s, G_s, B_s, A_s)$ |
| GL_SRC_ALPHA | source ou destination | $(A_s, A_s, A_s, A_s)$ |
| GL_ONE_MINUS_SRC_ALPHA | source ou destination | $(1,1,1,1) -(A_s, A_s, A_s, A_s)$ |
| GL_DST_ALPHA | source ou destination | $(A_d, A_d, A_d, A_d)$ |
| GL_ONE_MINUS_DST_ALPHA | source ou destination | $(1,1,1,1)-(A_d, A_d, A_d, A_d)$ |
| GL_SRC_ALPHA_SATURATE | source | $(f,f,f,1)$; $f=\min(A_s,1-A_d)$ |

# Mélange (avec Z-buffer)

- Mélange tout en considérant le Z-buffer
- Si rendu d'objets opaques et transparents
- Utiliser le Z-buffer pour éliminer les surfaces cachées pour tout objet derrière des objets opaques
- Un objet opaque A peut cacher soit un objet transparent T soit un autre objet opaque B
- Si Z(A) < Z(B) < Z(T) alors utiliser Z-Buffer
- Si Z(T) < Z(A) < Z(B) alors mélange entre intensités de T et de A

# Mélange (avec Z-buffer)

- Si Z(T) < Z(A) : Z-buffer inchangé, en lecture
- Si Z(A) < Z(B) : Z-buffer peut changer, écriture
- Contrôle du Z-buffer : glDepthMask(mode)
- mode :
  - GL_FALSE : Z-buffer en lecture seulement
  - GL_TRUE : Z-buffer en lecture-écriture

# Mélange (anticrênelage)

- Calcul de la couverture de chaque pixel
- Couverture par un segment ou un polygone
- OpenGL multiplie la valeur de Alpha par cette couverture
- Utiliser la nouvelle valeur de Alpha pour effectuer un mélange
- En fait ces nouvelles valeurs de Alpha correspondent aux coeffs d'un filtre passe-bas

# Mélange (anticrênelage)

- **Contrôle : glHint(Glenum *target*, Glenum *hint*)**
- ***hint* :**
  - **GL_FASTEST : option utilisée la + efficace**
  - **GL_NICEST : option de plus grande qualité**
  - **GL_DONT_CARE : aucune préférence**

# Mélange (anticrênelage)

- **Contrôle : glHint(Glenum *target*, Glenum *hint*)**
- ***target* :**

| Paramètre | Signification |
|---|---|
| GL_POINT_SMOOTH_HINT<br>GL_LINE_SMOOTH_HINT<br>GL_POLYGON_SMOOTH_HINT | Spécifie la qualité de l 'échantillonnage désirée des points, segments et polygones pendant les opérations d'anticrênelage |
| GL_FOG_HINT | Spécifie si le calcul de l'effet de brouillard est calculé par pixel (GL_NICEST) ou par sommet (GL_FASTEST) |
| GL_PERSPECTIVE_CORRECTION_HINT | Spécifie la qualité désirée de l'interpolation de la couleur ou texture |

# Mélange (anticrênelage)

- **Points et segments : glEnable(mode)**
- **mode :**
  - **GL_POINT_SMOOTH**
  - **GL_LINE_SMOOTH**
- **Pour plus de qualité utiliser glHint()**
- **Exemple de séquence**

```
glEnable(GL_LINE_SMOOTH) ;

glEnable(GL_BLEND);

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE) ;
```

# Les Textures

# Les textures ( Texture Mapping)

**Plaquage d'une image sur un objet.**
**Toutes les transformations sont appliquées à l'objet texturé.**

**Les textures peuvent être appliquées sur toutes les primitives (points, lines, polygons) de différentes manières :**
**directement sur les objets, moduler la couleur des objets, mélanger la texture à la couleur des objets, ..**

**Les textures sont des tableaux 2D de couleurs, de luminances ou de alpha couleurs.**
**Les valeurs individuelles d'une texture sont appelées *texels*.**

# Les textures ( Texture Mapping)

**Les différentes transformations (translations, rotations, mises à l'échelle, projections déforment les objets, mais aussi les textures appliquées sur ces objets.**

**Ces textures sont dilatées (duplication de texels) ou compressées (perte de texels).**

**Les textures sont faites d'éléments discrets.**

**Des opérations de filtrage doivent êtres faites.**

# Les textures ( Texture Mapping)

**Le placage de texture travaille en mode RGB**

**Etapes dans le texture mapping :**

- **Spécifier la texture**
- **Indiquer comment la texture doit être appliquée**
- **Activer le placage de texture**
- **Visualiser la scène**

# Spécifier une texture

**Spécifier une texture**

**Une texture est une image en 1D ou 2D dont chaque texel est représenté par 1, 2, 3 ou 4 coefficients de modulation de RGBA**

**En utilisant la technique de mipmapping on peut préciser différentes résolutions**

## Spécifier une texture

**glTexImage2D (target, level, components, width, height, border, format, type, *pixels)**

**target = GL_TEXTURE_2D**
**level = niveau de résolution (0 en général)**
**components = GL_RGB ou GL_RGBA**
**width, height = dimensions de la texture**
**border = largeur de la bordure (0 en général)**
**format, type = décrit le format et le type de données (même signification que pour glDrawPixels() )**
**pixels = contient les données texture**

## Résolution

**Les objets texturés sont visualisés comme les autres à des distances différentes du point de vue**

**Dans une scène animée, la texture doit être modifiée en fonction de la taille de l'objet projeté**

**OpenGL filtre la texture à une taille appropriée pour ensuite être plaquée sur l'objet**

**OpenGL détermine automatiquement la texture à utiliser**

**OpenGL ne fournit pas d'outils pour obtenir ces différents niveaux**

## Résolution

**Les textures à différents niveaux sont spécifiées par des glTexImage2D() avec le niveau débutant à 0 pour la résolution la plus grande**

**La GLU contient 3 routines pour manipuler les images devant être utilisées comme texture**

**Ayant défini le niveau 0, glBuild1DMipmaps() et glBuild2DMipmaps() construisent la pyramide de résolution 1D ou 2D. La taille de la texture initiale doit être une puissance de 2.**

## Résolution

**gluBuild1DMipmaps(target, components,width, format, type, void*data)**

**gluBuild2DMipmaps(target, components,width, height, format, type, void*data)**

**Les paramètres ont la même signification que dans glTexImage1D ou glTexImage2D**

**gluScaleImage(format, widthin, heightin, typein, void*datain, widthout, heightout, typeout, void*data)**

**L'image est mise à l'échelle par interpolation linéaire et une boite de filtrage en entrée (widthin, heightin) vers une boite de sortie**

# Filtrage

**Après transformations les texels correspondent rarement avec les pixels. A un pixel peut correspondre une portion de texel (grossissement) ou plus d'un texel (rétrécissement)**


**Magnification** **Minification**

# Filtrage

**Dans les deux cas, quelle valeur de texel doit on utiliser pour le pixel ?**

**Comment doivent-elles être interpolées ou moyennées?**

**OpenGL permet de préciser différents filtres pour effectuer ces calculs.**

**OpenGL fait un choix, dans la plupart des cas,entre *magnification* et *minification* pour obtenir le meilleur résultat.**

**Il est préférable de fixer les méthodes de filtrage (*magnification* et *minification* ) par :**

**glTexParameter*()**

# Filtrage

**glTexParameter{if}[v](target, pname, param)**

**target = GL_TEXTURE_2D, GL_TEXTURE_1D**

| Paramètres | Valeurs |
|---|---|
| GL_TEXTURE_WRAP_S | GL_CLAMP_TO_EDGE, GL_REPEAT |
| GL_TEXTURE_WRAP_T | GL_CLAMP_TO_EDGE, GL_REPEAT |
| GL_TEXTURE_MAG_FILTER | GL_NEAREST, GL_LINEAR |
| GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR<br>GL_NEAREST_MIPMAP_NEAREST<br>GL_NEAREST_MIPMAP_LINEAR<br>GL_LINEAR_MIPMAP_NEAREST<br>GL_LINEAR_MIPMAP_LINEAR |
| GL_TEXTURE_BORDER_COLOR | 4 valeurs entre 0 et 1 |

**Ces paramètres contrôlent comment une texture est traitée et appliquée**

# Filtrage

**GL_NEAREST choisit le texel le plus près du centre du pixel**

**GL_LINEAR fait une moyenne des 4 texels les plus près.**

**Lorsque les coordonnées texels sont près de la bordure de la texture, le résultat dépend de GL_REPEAT ou de GL_CLAMP_TO_EDGE**

**Avec *magnification* le niveau 0 de texture est toujours utilisé**
**Avec *minification* on peut choisir une méthode de filtrage en utilisant 1 ou 2 plans mipmap.**

# Filtrage

**Quatre choix de filtres sont possibles avec *minification*.**

**Pour un plan mipmap donné on peut choisir :**

**GL_NEAREST_MIPMAP_NEAREST**
**GL_NEAREST_MIPMAP_LINEAR**

**Pour choisir les 2 meilleurs plans utiliser :**

**GL_LINEAR_MIPMAP_NEAREST**
**GL_LINEAR_MIPMAP_LINEAR**

# Coordonnées textures

**Pour tracer une scène texturée on doit fournir pour chaque sommet (vertex) les coordonnées objets et les coordonnées texture (un texel, un sommet).**

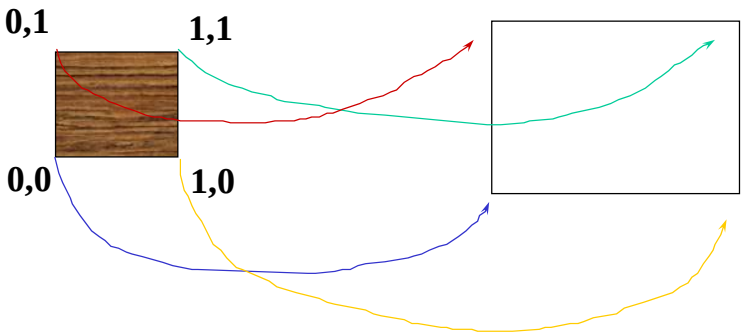
**Comme pour les couleurs, les coordonnées texels sont interpolées entre sommets.**

**La commande glTexCoord*() pour spécifier les coordonnées textures est similaire à : glVertex*()**

**glTexCoord{1234}{sifd}[v]( coords)**

# Coordonnées textures

**Comment déterminer les coordonnées texture ?**




**Attention aux distorsions si les ratios sont différents.**
**Pour éviter cela, prendre une portion de texture**

# Pile de matrices texture

**Comme pour les objets, on peut appliquer des matrices de transformations sur les textures.**

**Ne pas oublier de se mettre dans le mode correspondant**

**Exemple :**

```
glMatrixMode(GL_TEXTURE);
glTranslatef(.....);
.....

glMatrixMode(GL_MODELVIEW);
```

# Modulation et mélange

**Dans ce qui a été vu précédemment, les textures ont été utilisées directement comme couleur des surfaces.**

**On peut utiliser les textures pour moduler les couleurs des objets ou pour mélanger la texture avec la couleur des objets.**

**Quatre types de fonctions selon les arguments de**

**glTexEnv*()**

# Modulation et mélange

## glTexEnv{if}[v](target, pname, param)

**target = GL_TEXTURE_ENV**
**si pname = GL_TEXTURE_ENV_MODE**
**param = GL_REPLACE, GL_DECAL, GL_MODULATE, GL_BLEND**
**indique comment la texture est combinée avec la couleur de l'objet**

**si pname = GL_TEXTURE_ENV_COLOR**
**param est un vecteur RGBA. Ces valeurs sont utilisées si la fonction de texture GL_BLEND a été spécifiée**