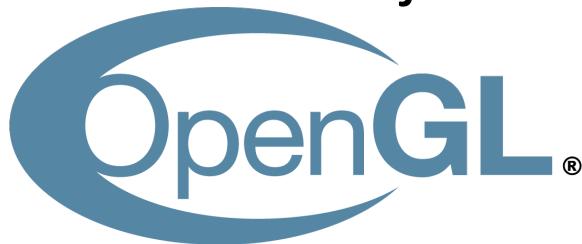


Introduction à la 3D temps réel avec OpenGL

Jean-Marie Normand

Bureau IM3-B

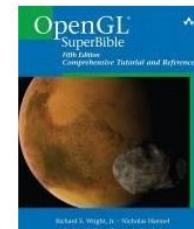
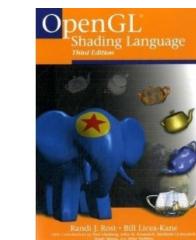
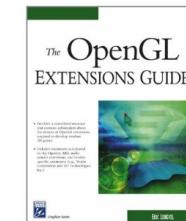
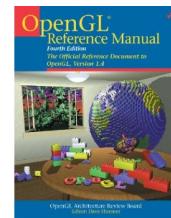
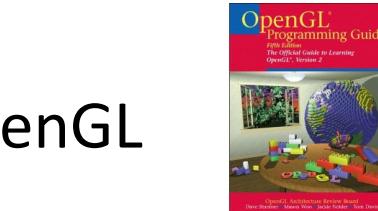
jean-marie.normand@ec-nantes.fr



Références Bibliographiques

- Livres :

- Le **Rouge** (« Red Book ») : OpenGL Programming Guide
- Le **Bleu** : OpenGL Reference Manual
- Le **Blanc** : OpenGL Extensions Guide
- L'**Orange** (depuis OpenGL 2.0) : OpenGL Shading Language
- La « Super Bible » : OpenGL Super Bible



Références Bibliographiques

- Académiques :
 - <http://maverick.inria.fr/~Nicolas.Holzschuch/>
 - [http://maverick.inria.fr/~Gilles.Debunne/
index.html](http://maverick.inria.fr/~Gilles.Debunne/index.html)
 - <http://interaction.lille.inria.fr/~roussel/>
 - <http://maverick.inria.fr/~Xavier.Decoret/>
 - <http://www.malgouyres.fr/enseignement>
 - [http://www-evasion.imag.fr/Membres/
Francois.Faure/enseignement/ressources/
opengl.html](http://www-evasion.imag.fr/Membres/Francois.Faure/enseignement/ressources/opengl.html)

Références Bibliographiques

- Internet :
 - <http://www.opengl.org>
 - <http://nehe.gamedev.net/>
 - <http://www.khronos.org/opengl>
 - <http://www.glprogramming.com/red/>
 - <http://www.opengl.org/documentation/>

Références Bibliographiques

- Internet :
 - <http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html>
 - <http://raphaello.univ-fcomte.fr/ig/opengl/opengl-1.htm>
 - <http://www.songho.ca/opengl/index.html>
 - <http://iel.ucdavis.edu/projects/chopengl/demos.html>
 - <http://www-evasion.imag.fr/Membres/Antoine.Bouthors/teaching/opengl/>

Références Bibliographiques

- Internet :
 - [http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#Transformation matrices](http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#Transformation_matrices)
 - <http://www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl>

OpenGL – Kézako ?

- OpenGL : Open Graphics Library développée par SGI et standardisée en 1992.
 - sous l'égide de l'ARB (Architecture Review Board) qui regroupe en son sein de nombreuses entreprises ayant un intérêt pour la création d'applications 3D jusqu'en 2006
 - depuis 2006 sous l'égide du Khronos Group (qui contrôle aussi la standardisation de nombreuses autres ressources)
- OpenGL = Interface de Programmation (API) graphique (principalement 3D)
 - Relativement indépendante du langage de programmation
 - Indépendante du système d'exploitation
 - Indépendante du système de fenétrage

OpenGL – Kézako ?

- Caractéristiques d'OpenGL :
 - Simplicité
 - Performance
 - Qualité de rendu
 - Evolutivité
- Implémentation d'OpenGL :
 - Logicielle (p. ex. librairie Mesa3D sous linux)
 - Matérielle (via les cartes avec GPU – Graphics Processing Unit – NVIDIA ou ATI/AMD)

OpenGL – Kézako ?

- Versions d'OpenGL :
 - OpenGL 1.0 – 1992 : Version de base
 - OpenGL 1.1 – 1997 : Introduction des textures
 - OpenGL 1.2 – 1998
 - OpenGL 1.3 – 2001
 - OpenGL 1.4 – 2002
 - OpenGL 1.5 – 2003 : Introduction des Vertex Buffer Objects (VBOs)
 - OpenGL 2.0 – 2004 : Introduction du langage GLSL (OpenGL Shading Language) pour la programmation des « shaders » → écriture et exécution de programmes utilisateurs sur la carte graphique
 - OpenGL 2.1 – 2006
 - OpenGL 3.0 – 2008 : Introduction d'un mécanisme de « dépréciation »
 - OpenGL 3.1 – 2009
 - OpenGL 3.2 – 2009
 - OpenGL 3.3 – 2010
 - OpenGL 4.0 – 2010 : Introduction de l'arithmétique 64bits, etc.
 - OpenGL 4.1 – 2010
 - OpenGL 4.2 – 2011
 - OpenGL 4.3 – 2012 (version actuelle)

OpenGL – Kézako ?

- « Évolutions » d'OpenGL:
 - OpenGL ES : OpenGL for Embedded Systems (version « mobile » d'OpenGL)
 - Différentes versions : OpenGL ES 1.0, 1.1, 2.0, 3.0
 - OpenGL SC : OpenGL Safety Critical (version pour systèmes critiques, p. ex. aviation, nucléaire, etc.)
 - WebGL : Web Graphics Library (API JavaScript basée sur OpenGL ES 2.0 pour les navigateurs Internet)
- Langages supportant OpenGL:
 - C (historique), Java, Perl, Python, Caml, C#, etc.

Quelques API utiles avec OpenGL

- **GLU** (OpenGL Utility Library) fait partie d'OpenGL
 - NURBS, tessellations, formes quadriques (cylindres, etc.)
- Pour accéder au système de fenêtrage :
 - Bibliothèques dédiées à une plateforme : GLX, WGL, etc.
 - Bibliothèques multiplateformes : SDL, GTK, QT, wxWidgets, etc.
- **GLUT** (OpenGL Utility Toolkit) : accès au système de fenêtrage, aux périphériques (clavier, souris, etc.) indépendamment de la plateforme de développement.

Concurrent d'OpenGL

- Le principal « concurrent » d'OpenGL est Direct3D, bibliothèque faisant partie de l'API Microsoft DirectX.
- Direct3D fonctionne uniquement sur les plateformes Microsoft (Windows, Xbox, Xbox 360, XBOX One).

Quelques utilisations d'OpenGL

- Applications :
 - Modeleurs 3D : Blender, Rhinoceros, Autodesk Maya, 3D Studio Max
 - Google Earth, Google Sketch-Up
 - Adobe : After Effects, Photoshop, Premiere
 - Applications scientifiques, etc.
- Jeux Vidéo :
 - Tous les jeux idSoftware (Quake, Doom)
 - Tous les jeux Blizzard sous Mac OS (Starcraft, Warcraft, etc.)

La bibliothèque OpenGL

- Fonctions de base pour l'affichage 2D/3D :
 - dessin de sommets, de segments de droite et de facettes,
 - matériaux et lumières,
 - transformations géométriques,
 - caméras,
 - textures,
 - gestion des paramètres de rendu,
 - etc.

La bibliothèque OpenGL

- Pas de fonctions pour la création et la gestion d'une interface utilisateur (fenêtre, entrées/sorties, etc.)
 - Pour cela **nous utiliserons GLUT** (par simplicité, dans un « vrai » projet il faut utiliser une autre bibliothèque pour des raisons de performances)
- Les fonctions OpenGL utilisent les règles de syntaxe du C, elles sont préfixées par ***gl*** et les définitions se trouvent dans le fichier ***gl.h***

La bibliothèque OpenGL

- Conventions d'écriture:
 - Fonctions débutent par *gl*
 - Constantes en majuscules et débutent par *GL_*
 - Certains noms de fonctions se terminent par un suffixe (p. ex. *glVertex2f*) qui indique le nombre et le type des arguments de la fonction (ici 2 flottants)
 - Ni types, ni structures définies en OpenGL → seuls types manipulés sont les **types primitifs d'OpenGL** (basés sur ceux du C et qui débutent par *GL*) ou des tableaux de ces types primitifs

La bibliothèque OpenGL

Suffixe	Type	Type C	Type OpenGL
b	entier 8 bits	signed char	GLbyte
s	entier 16 bits	short	GLshort
i	entier 32 bits	long, int	GLint, GLsizei
f	réel 32 bits	float	GLfloat, GLclampf
d	réel 64 bits	double	GLdouble, GLclampd
ub	entier non signé 8 bits	unsigned char	GLubyte, GLboolean
us	entier non signé 16 bits	unsigned short	GLushort
ui	entier non signé 32 bits	unsigned int	GLuint, GLenum, GLbitfield

Le suffixe **v** correspond à une donnée sous forme de vecteur

Exemple:

glVertex2i(1,3); → sommet 2D de coordonnées entières

glVertex2f(1.0,3.0); → sommet 2D de coordonnées réelles

glColor3i(1,0,1); → couleur au format RGB (ici du violet)

int c[3]={1,0,1}; glColor3iv(c); → même couleur que ci dessus

La bibliothèque OpenGL

- Comme toujours vous pouvez trouver toutes les informations nécessaires dans la documentation !
- On ne peut connaître toutes les fonctions et tous leurs paramètres par cœur ! Aller donc consulter la documentation :
 - *<http://www.opengl.org/sdk/docs/man/xhtml/>*

La bibliothèque GLU

- Regroupe des commandes bas-niveau écrites en GL:
 - Transformations géométriques spécifiques à certaines actions
 - Transformation des polygones quelconques en ensembles de polygones triangulaires
 - Rendu des surfaces paramétriques et quadriques (cylindres, sphères, etc.)
- Les fonctions sont préfixées par : *glu*
- Les définitions sont dans le fichier *glu.h*

La bibliothèque GLUT

- Fonctionnalités :
 - Gestion d'une fenêtre d'affichage,
 - Gestion des périphériques d'entrée,
 - Gestion des menus,
 - Gestion des environnements multifenêtres,
 - Gestion des polices de caractères (bitmap et vectorielles) pour l'écriture de texte
- Les fonctions sont préfixées par *glut* et sont définies dans *glut.h*

Comment fonctionne OpenGL ?

- OpenGL sait dessiner des **points**, des **lignes**, des **polygones convexes** et des **images**
- Ces primitives sont définies par un ensemble de sommets (**vertices**) et affichées (rendues) dans un **framebuffer**
- Le rendu des primitives dépend de nombreuses variables d'état (matrices de transformation, couleur, matériau, texture, éclairage, etc.)
- OpenGL ne sait pas ce qu'est une souris, un clavier, une fenêtre ou même un écran.

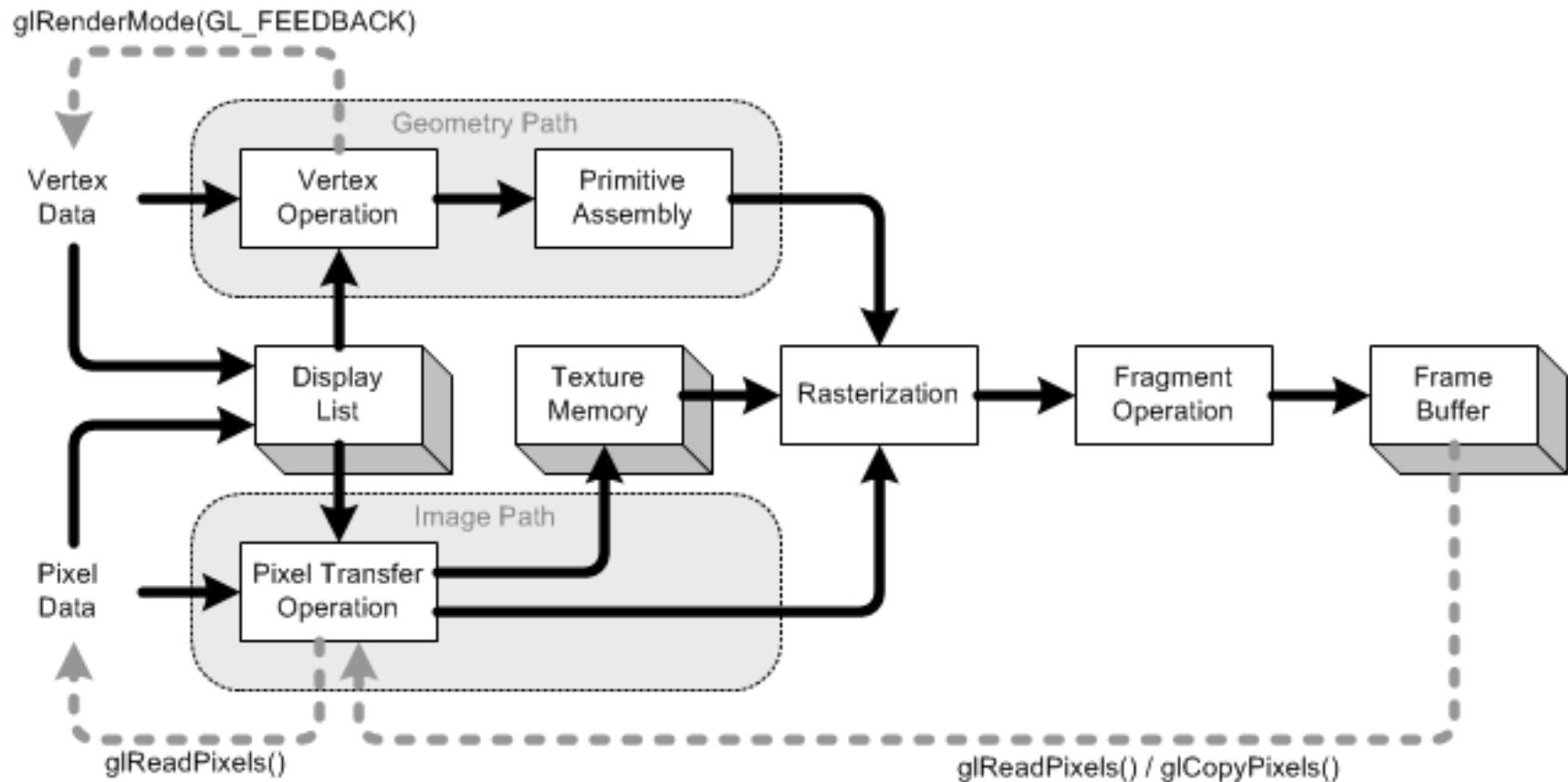
Le pipeline graphique OpenGL

- OpenGL fonctionne comme un « pipeline » avec des étapes successives réalisant les opérations nécessaires à la visualisation d'une scène.
- Pour simplifier le pipeline graphique :
 - En entrée :
 - des sommets 3D, leurs normales (vecteurs 3D) → données de type « *sommets* » ou données « *géométriques* »
 - des textures, images → données de type « *pixels* »
 - des paramètres d'affichage → *environnement* OpenGL
 - En sortie : une image → *framebuffer*

OpenGL « old school » vs. OpenGL avec des shaders

- Dans ce cours d'introduction nous allons nous intéresser principalement à la version « **ancienne** » de faire de l'OpenGL, c'est à dire sans shaders !
- Nous aborderons les **shaders** en fin de slides (et dans un prochain cours)
- Mais l'idée de base du fonctionnement d'OpenGL est le même et est **plus simple à comprendre** avec l'ancienne syntaxe
- Nous préciserons toutefois le fonctionnement « **moderne** » d'OpenGL pour que vous connaissiez la différence entre les deux modes de fonctionnement

Le pipeline graphique OpenGL



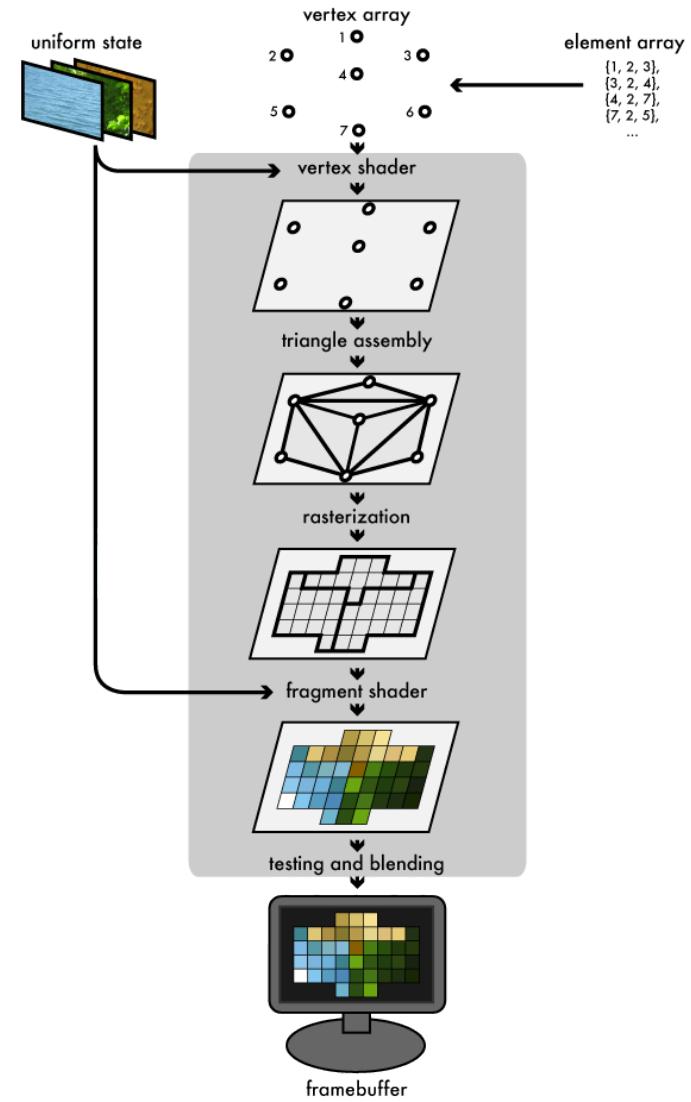
Source : http://www.songho.ca/opengl/gl_pipeline.html

Le pipeline OpenGL – Explication Simplifiée

- *Vertex Operation* : transformation matricielle des sommets et normales afin de changer de repère (du repère objet vers le repère de la caméra, cf. Transformations Ius loin) + gestion de l'éclairage au niveau des sommets.
- *Pixel Transfer Operation* : transformation des images de la mémoire utilisateur vers la mémoire graphique (*Texture Memory*).
- *Primitive Assembly* : projection des primitives (points, lignes, polygones) dans un repère 2D d'affichage.
- *Rasterization* : conversion des primitives géométriques en « *fragments* ». Un *fragment* correspond à un ensemble d'informations permettant l'affichage final des *pixels* dans le framebuffer.
- *Fragment Operation* : conversion des *fragments* en *pixels* en deux étapes :
 1. Génération et application d'un *texel* (texture element) depuis la Texture Memory à chaque *fragment*.
 2. Application d'un ensemble de traitements à chaque *fragment* afin de produire un *pixel* colorié qui sera stocké dans le *framebuffer*.

Le pipeline OpenGL

- Remplissage des buffers OpenGL de *vertex arrays* (tableaux de sommets et de normales)
- Projection des sommets dans l'espace écran (screen space) par les *vertex shaders*
- Assemblage des sommets en triangles
- « Rasterization » des triangles en fragments
- Transformations des fragments en pixels par les *fragment shaders*



OpenGL = une machine à états

- Les valeurs des paramètres d'affichage d'OpenGL sont stockées comme des variables d'états (ayant une valeur par défaut).
- OpenGL offre des fonctions permettant de modifier ces états → ceux-ci seront utilisés jusqu'à ce qu'ils soient changés!
 - p. ex. `glLineWidth(5.0);` → l'épaisseur des lignes sera de 5 pixels jusqu'à ce que `glLineWidth` soit rappelée
- L'ensemble de ces variables d'états définit :
 - la manière dont est dessiné une primitive (triangle, point, ligne)
 - c'est ce que l'on appelle l'environnement OpenGL

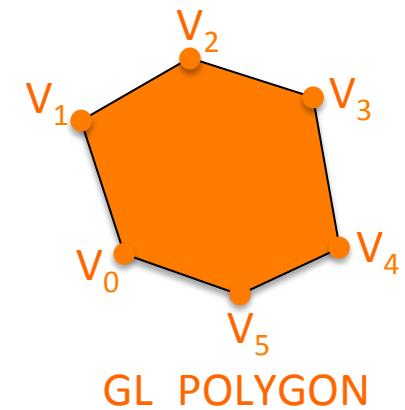
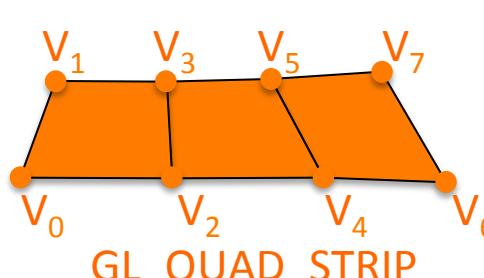
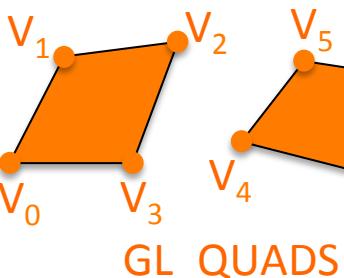
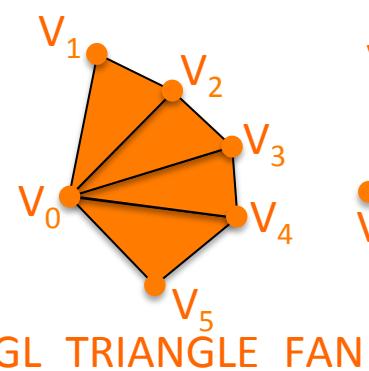
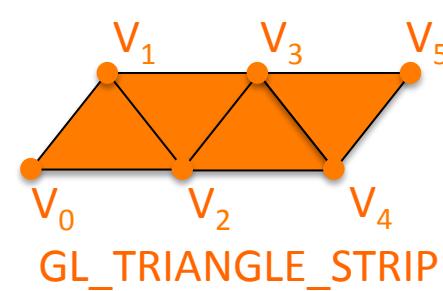
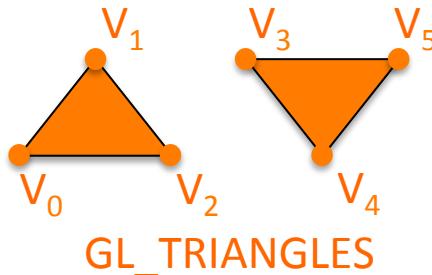
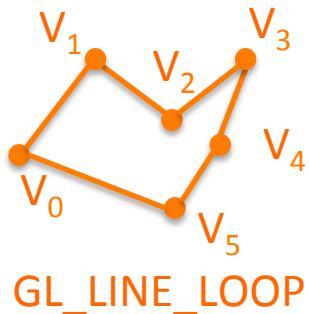
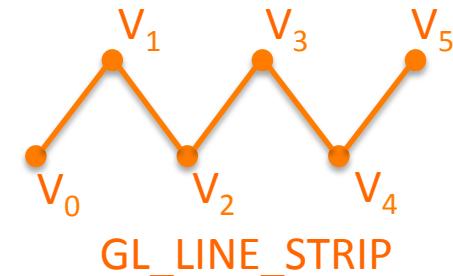
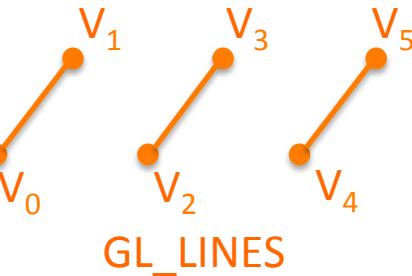
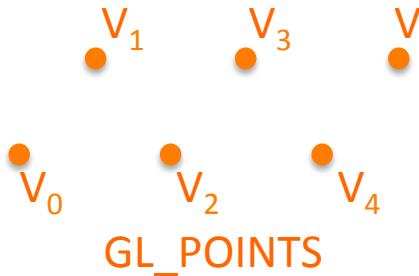
Vérifier/Interroger un état

- Interroger l'état courant avec les fonctions *glGet* et *glIsEnabled*
- `void glGet{Boolean,Integer,Float,Double}v(GlEnum pname, T* params)`
- `Glboolean glIsEnabled(GlEnum cap);`
- cf. <http://www.opengl.org/sdk/docs/man/xhtml/glGet.xml> et <http://www.opengl.org/sdk/docs/man/xhtml/glGet.xml>
- Exemple : `GLboolean cullFace = GL_TRUE;`
`glGetBooleanv(GL_CULL_FACE, &cullFace);`
- ou
`GLboolean;`
`cullFace = glIsEnabled(GL_CULL_FACE);`

Modifier un état

- Il existe de nombreuses fonctions de l'API pour modifier un état de l'environnement OpenGL :
 - *glEnable*/*glDisable*
 - *glLineWidth*, *glPointSize*, *glPolygonMode*,
glCullFace, *glColor{34}{ifd}(...)*, etc.
- Il faudra vous rapporter à la documentation

Primitives Géométriques



Dessiner une primitive géométrique

- Les primitives sont décrites par une liste de sommets délimitée par *glBegin* et *glEnd* et paramétrée par le type de primitive
- Exemples :

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

```
#include <math.h>
...
GLint i;
```

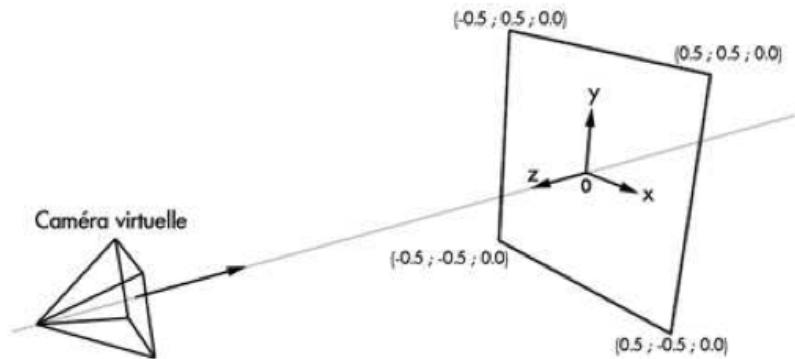
```
glBegin(GL_POLYGON);
    for(i=0; i<6; i++) {
        glVertex2f(cos(2*i*M_PI/6.0),
                   sin(2*i*M_PI/6.0));
    }
glEnd();
```

Dessiner une primitive géométrique

- Différents attributs peuvent être spécifiés pour chaque sommet (vertex) :
 - Couleur, fonction `glColor3f`
 - Normale, fonction `glNormal3f`
 - Coordonnée de texture, fonction `glTexCoord2f`
 - Etc.

Un premier exemple simple

- But : afficher à l'écran un carré coloré
- Bonus : utiliser **GLUT** pour la gestion des fenêtres et pour implémenter une gestion minimale des évènements utilisateur
- Scène « 3D » :
- Pour le moment on se contente de la **2D** → nous verrons plus loin les matrices, transformations et repères permettant de faire de la **3D**



Un premier exemple simple

- Première chose à faire:
inclure les fichier d'entête
pour **OpenGL** et **GLUT**
- Initialisation de **GLUT** !
- Choix du mode
d'affichage
- Taille et position initiales
de la fenêtre **GLUT**
- Création de la fenêtre

```
// En-têtes pour OpenGL et GLUT
#include <OpenGL/gl.h>
#include <GLUT/GLUT.h>
// Initialisation de GLUT
glutInit(&argc, argv);
// Choix du mode d'affichage (ici RVB)
glutInitDisplayMode(GLUT_RGB);
// Position initiale de la fenêtre GLUT
glutInitWindowPosition(200, 200);
// Taille initiale de la fenêtre GLUT
glutInitWindowSize(350, 350);
// Crédit de la fenêtre GLUT
glutCreateWindow("Premier Exemple");
```

Un premier exemple simple

- On doit définir la couleur d'effacement du frame buffer
- Définir des fonctions de rappel (callbacks) pour GLUT
- Lancer la « boucle infinie » de GLUT

```
// Définition de la couleur d'effacement  
// du framebuffer OpenGL  
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);  
  
// Définition des fonctions de callbacks  
glutDisplayFunc(affichage);  
glutKeyboardFunc(clavier);  
  
// Lancement de la boucle infinie GLUT  
glutMainLoop();
```

Un premier exemple simple

- Fonctions de rappel (**callbacks**) :
 - Fonctions associées aux différents événements envoyés par le gestionnaire de fenêtres
 - À chaque fois qu'un événement est émis par le gestionnaire de fenêtres la fonction de rappel associée est appelée
 - Il est **obligatoire** de définir au moins une fonction de rappel : la **fonction d'affichage** dans laquelle vous devrez créer votre scène 3D
 - Il est également possible de définir des callbacks associés aux événements liés :
 - Au **clavier**,
 - À la **souris**,
 - À des périphériques d'entrée spéciaux (tablettes, spaceballs, etc.)
 - À la modification de la forme de la fenêtre (**redimensionnement**)
 - Il existe 2 callbacks spéciaux permettant d'être appelés :
 - À intervalle de temps réguliers,
 - Lorsque l'application ne « fait rien » (**idle**) → utile pour l'animation

Un premier exemple simple

- La fonction d'affichage : **affichage**
- **glClear** permet d'effacer les buffers OpenGL, ici nous effacerons le frame buffer (**GL_COLOR_BUFFER_BIT**)
- **glColor** change la valeur de l'état OpenGL contrôlant la couleur d'affichage
- **glFlush** indique à OpenGL de forcer l'affichage du framebuffer

```
// Définition de la fonction d'affichage
GLvoid affichage(){
    // Effacement du frame buffer
    glClear(GL_COLOR_BUFFER_BIT);
    // Dessin d'un carre colore
    glBegin(GL_QUADS);
    // Premier sommet : Rouge
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex2f(-0.5f, -0.5f);
    // Second sommet : Vert
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(0.5f, -0.5f);
    // Troisième sommet : Bleu
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(0.5f, 0.5f);
    // Quatrième sommet : Blanc
    glColor3f(1.0f, 1.0f, 1.0f);
    glVertex2f(-0.5f, 0.5f);
    glEnd();

    glFlush();
}
```

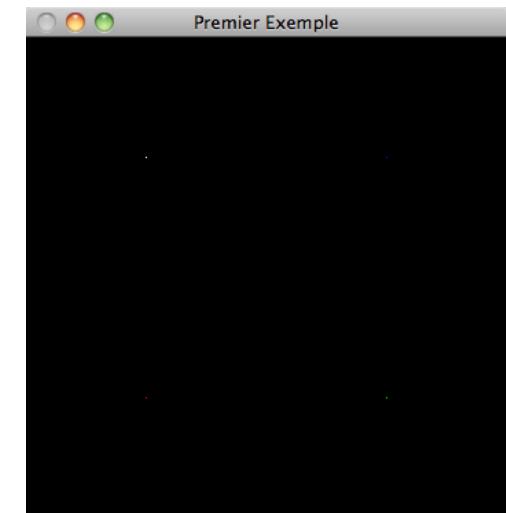
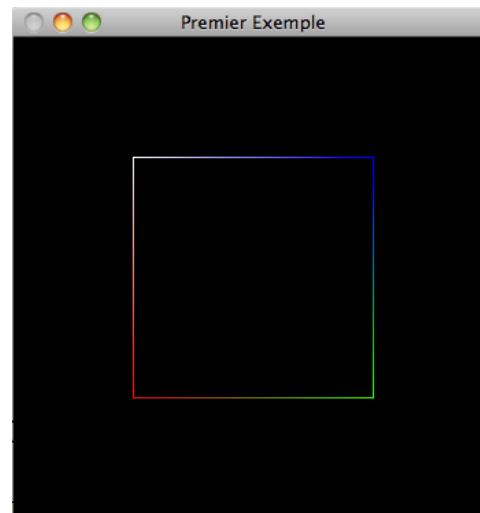
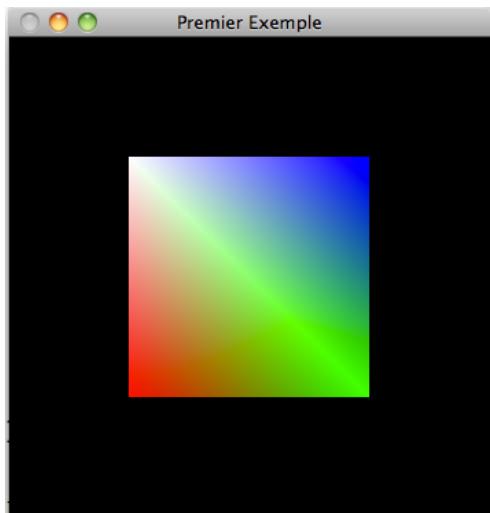
Un premier exemple simple

- Lorsque l'utilisateur appuiera sur une touche du clavier, GLUT appellera la fonction de gestion du clavier : *clavier*
- *touche* contient le code de la touche frappée
- *x* et *y* contiennent les coordonnées de la souris au moment de la frappe
- En fonction de la touche frappée, différents actions sont effectuées :
 - p : le carré est affiché « *plein* »
 - f : le carré est affiché en « *fil de fer* »
 - s : les sommets du carré sont affichés en tant que *points*
 - q ou ESCAPE : on quitte le programme (appel à la fonction C *exit*)

```
// Définition de la fonction gérant les interruptions clavier
GLvoid clavier(unsigned char touche, int x, int y) {
    switch(touche) {
        case 'p': // carre plein
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            break;
        case 'f': // fil de fer
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            break;
        case 's': // sommets du carre
            glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
            break;
        case 'q': // quitter
        case 27:
            exit(0);
            break;
    }
    // Demande a GLUT de reafficher la scene
    glutPostRedisplay();
}
```

Un premier exemple simple

- Résultat :



- À vous !

Création d'un projet Visual Studio

1. Lancer Visual Studio 2012
2. Choisir « **Nouveau Projet** »
3. Choisir « **Application Console Win32** »
4. Donner un nom à votre projet
5. Cliquer sur « **Suivant** »
6. **Décocher** « **Entêtes précompilés** » et « **Vérification SDL** »
7. Cliquer sur « **Terminer** »
8. Une fois votre solution ouverte : cliquer droit sur le nom de votre projet (icône )
9. Choisir « **Propriétés de configuration** »
 - Puis « **Général** » et vérifier que « Jeu de caractères » est bien à « **Utiliser le jeu de caractères Unicode** »
10. Optionnel : aller dans « **Editeur de liens** »
11. Puis « **Entrée** » et rajouter les librairies suivantes : **opengl32.lib;glu32.lib;glut32.lib**
12. Dans l'éditeur de code :
 1. Supprimer la ligne suivante : **#include "StdAfx.h"**
 2. Rajouter à la place **#include <Windows.h>**
 3. Rajouter les fichiers d'entête OpenGL :
 - **#include <GL/gl.h>**
 - **#include <GL/GLUT.h>**
13. C'est parti pour le code !

Utilisation de la VM Linux

- Solution 1 : Programmation « Old School » mais suffisante pour nous dans un premier temps 😊 :
 - Utilisation d'un éditeur de texte pour taper notre code C/OpenGL
 - Compilation et exécution depuis le terminal
- Créer un **dossier OpenGL** là où vous souhaitez stocker le code source
- Créer un **sous-dossier FirstEx** pour le premier exemple
- Récupérer le fichier **CMakeLists.txt** sur Hippocampus (cours **IMAGRIV** répertoire OpenGL) et le **copier dans le répertoire FirstEx**
- Créer un fichier **main.cpp** dans le dossier **FirstEx** : c'est le fichier qui contiendra votre code C/OpenGL
- **Ecrire** votre code source C/OpenGL dans **main.cpp**
- **Ouvrir un terminal** puis aller dans le dossier **FirstEx** en utilisant la commande '**cd / chemin/...**'
- Lancer la commande '**cmake .**'
- Normalement **CMake** vous dit que tout se passe bien et qu'il a généré un fichier **Makefile**
- Taper '**make**'
- En théorie un message apparaît sur le terminal affichant : '**[100%] Built target OGL_FirstEx**'
- Lancer votre exécutable ! Taper la commande '**./OGL_FirstEx**'
- Si le code écrit est correct, admirer le résultat attendu !

Utilisation de la VM Linux (2)

- Solution 2 : Code::Blocks
- Créer un dossier **OpenGL** et un sous-répertoire **OGL_FirstEx**
- Installer un paquet manquant : **xterm** (*pas sur qu'il soit manquant à voir*)
 - Lancer **Synaptic** (Menu → Administration → Gestionnaire de paquets Synaptic) et s'authentifier
 - Cliquer sur Rechercher et entrer : '**xterm**' (sans les apostrophes)
 - Descendre en bas de la liste et sélectionner le paquet '**xterm**' pour installation
 - Cliquer sur **Appliquer** puis quitter Synaptic
- Lancer Code::Blocks et choisir **New → Project → Empty Project**
- Nommer le projet **OGL_FirstEx** et vérifier que vous le stockez bien dans les répertoires créés plus haut
- Laisser le compilateur par défaut : **GCC** et les autres options par défaut de la page de sélection du compilateur
- Cliquer sur **Finish** : le projet est créé !

Utilisation de la VM Linux (3)

- Solution 2 : Code::Blocks (suite)
- Avant de rajouter du code, il nous faut configurer le projet pour OpenGL/GLUT :
 - Aller dans le menu **Project → Build Options**
 - Onglet **Linker Settings**
 - Link libraries : Ajouter ‘GL’, ‘GLU’ et ‘glut’ (sans apostrophes et attention aux majuscules)
 - Onglet **Search Directories** : dans **Compiler** : Add ‘/usr/include/GL/’
 - Cliquer sur Valider
- Créer un nouveau fichier nommé ‘**main.cpp**’ : c’est le fichier qui contiendra votre code C/OpenGL
 - Aller dans le menu **File → New → File → C/C++ Source**
 - Entrer ‘**main.cpp**’ comme nom de fichier puis cliquer sur le bouton ‘...’ pour aller chercher le chemin complet où sera stocké votre fichier (oui Code::Blocks est très pénible)
 - Cocher les cases **Debug** et **Release** puis cliquer sur ‘**Finish**’
- **Ecrire** votre code source C/OpenGL dans **main.cpp**
- Cliquer sur le bouton de **Compilation** (Roue dentée jaune)
- Cliquer sur le bouton de **Lancement** de votre exécutable (Flèche verte)
- Si le code écrit est correct, admirer le résultat attendu !

Et maintenant la 3D ?

- Pas encore !
- Avant de passer à la 3D il nous faut parler des **transformations** appliquées aux sommets et des différents **systèmes de coordonnées** impliqués dans une application 3D
- **Transformations** = multiplications matricielles
- **Repères** = caméra, image, fenêtre, etc.

Modèle de caméra virtuelle

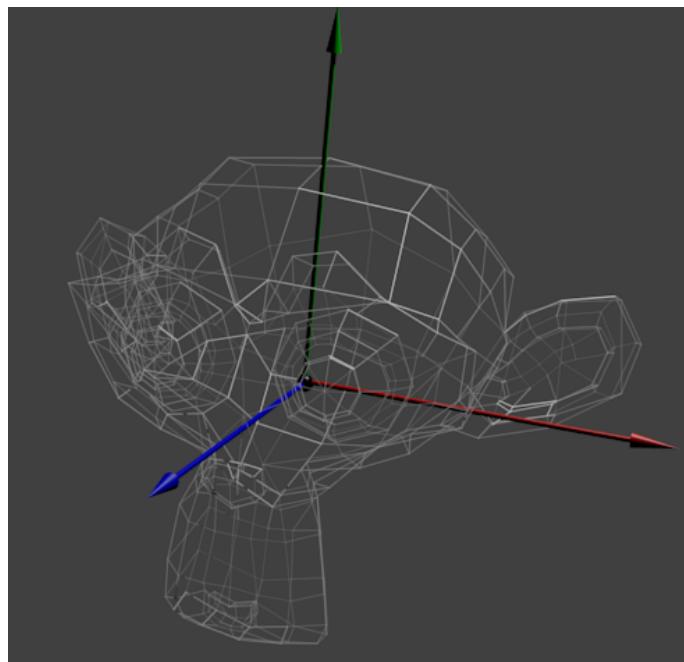
- Analogie : l'observateur observe la scène 3D au travers d'une caméra posée sur un trépied
- Il peut:
 - Changer l'**objectif** ou **ajuster le zoom** de la caméra → définition de la **projection**
 - **Déplacer** et **orienter** le trépied et la caméra → position et orientation de la **vue (viewing)**
 - **Déplacer les objets** observés ou les faire déplacer → transformation du **modèle**
 - Agir sur les **images 2D** produites → agrandissement ou réduction des images (**viewport** transformations)

Systèmes de coordonnées et transformations

- Systèmes de coordonnées utilisés en OpenGL :
 - **Model Space**: repère propre à chacun des objets 3D. Un vertex de coordonnée (0,0,0) sera au centre de l'objet.

Convention **RGB** pour les axes, cf. TP Unity :

- **Rouge = X**
- **Vert = Y**
- **Bleu = Z**

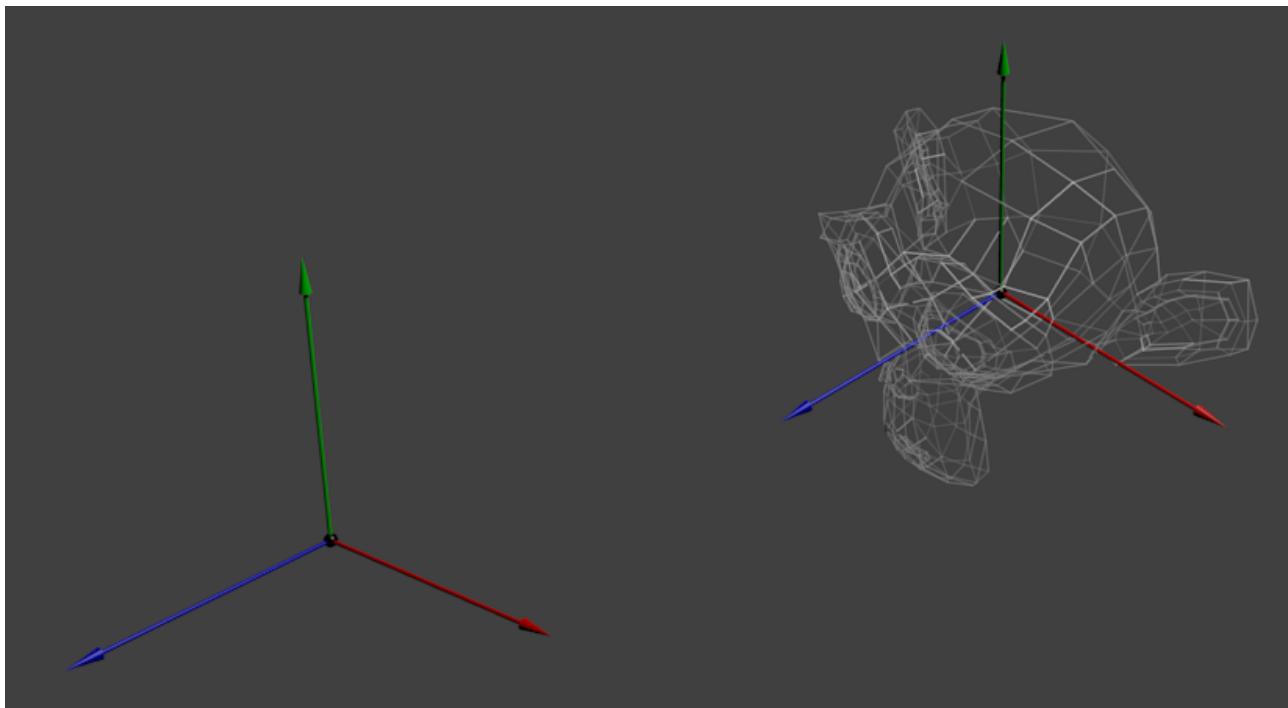


Source

[http://www.opengl-tutorial.org/
beginners-tutorials/tutorial-3-matrices/](http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/)

Systèmes de coordonnées et transformations

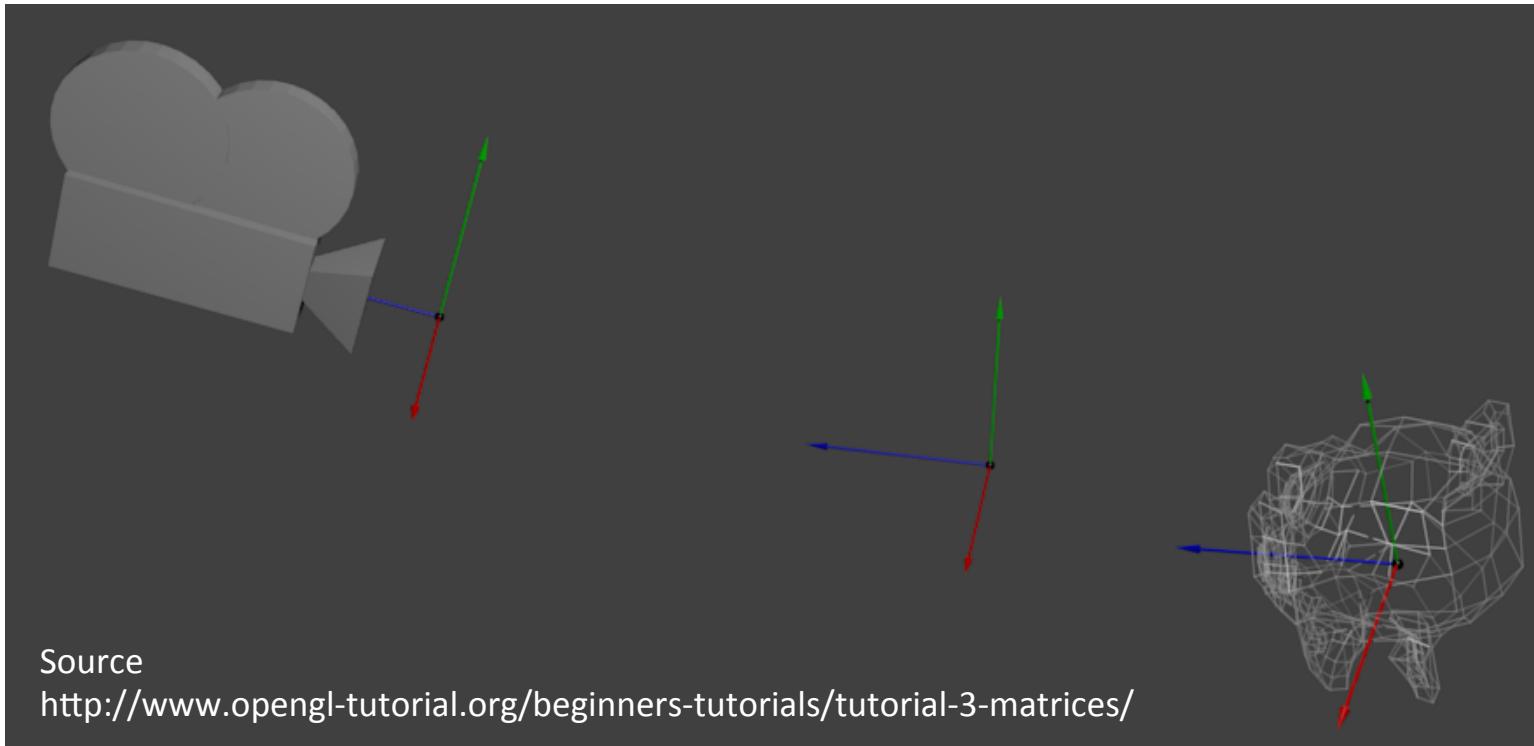
- Systèmes de coordonnées utilisés en OpenGL :
 - **World Space**: repère du monde. Un vertex de coordonnée $(0,0,0)$ sera au centre du monde.



Source
<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>

Systèmes de coordonnées et transformations

- Systèmes de coordonnées utilisés en OpenGL :
 - **Camera Space**: repère de la caméra. Un vertex de coordonnée (0,0) sera rendu au centre de l'écran.



Systèmes de coordonnées et transformations

- À partir d'un modèle 3D il nous faut donc combiner plusieurs transformations :
 - Du **model space** au **world space**
 - Du **world space** au **camera space**
 - Du **camera space** au **screen space** (opération de **projection** que l'on verra un peu plus loin)
- Les transformations sont effectuées par une succession de multiplications matricielles.

Transformations affines

- Les transformations affines préservent la géométrie. Après transformation affine :
 - Une ligne reste une ligne,
 - Un polygone reste un polygone,
 - Une quadrique (ellipsoïde, paraboloïde, hyperboloïdes, cônes, cylindres) reste une quadrique.
- Exemple de transformations affines :
 - Translation,
 - Changement d'échelle,
 - Rotation,
 - Projection,
 - Glissement,
 - Toute composition de transformations affines.

Coordonnées homogènes

- Une coordonnée homogène est un quadruplet (x,y,z,w)
- Si $w = 1 \rightarrow$ la coordonnée homogène représente un point de l'espace
- Si $w = 0 \rightarrow$ la coordonnée homogène représente une direction dans l'espace
- Si $0 < w < 1 \rightarrow$ il faut normaliser : diviser toutes les coordonnées par w
- Seule l'application d'une projection change la valeur de w

Coordonnées homogènes

- Intérêt des coordonnées homogènes :
 - Les coordonnées et les transformations s'expriment par des matrices 4×4
 - Les compositions de transformations correspondent à des produits de matrices
 - Attention toutefois à l'ordre des transformations !
→ le produit matriciel n'est pas commutatif

Transformations de base

glTranslate(tX,tY,tZ)

1	0	0	tX
0	1	0	tY
0	0	1	tZ
0	0	0	1

glScale(sX,sY,sZ)

sX	0	0	0
0	sY	0	0
0	0	sZ	0
0	0	0	1

glRotate(a,x,y,z)

$x^2(1-c)+c$	$xy(1-c)-zs$	$xz(1-c)+ys$	0
$xy(1-c)+zs$	$y^2(1-c)+c$	$yz(1-c)-xs$	0
$xz(1-c)-ys$	$yz(1-c)+xs$	$z^2(1-c)+c$	0
0	0	0	1

avec $c=\cos(a)$; $s=\sin(a)$; $\|(x,y,z)\| = 1$

glRotate(a,1,0,0)

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

NB : a est exprimé en degrés

glRotate(a,0,1,0)

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

Identité

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

glRotate(a,0,0,1)

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

Transformations inverses et symétries

- Transformations inverses :
 - $\text{Translate}(tx, ty, tz)^{-1} = \text{Translate}(-tx, -ty, -tz)$
 - $\text{Scale}(sx, sy, sz)^{-1} = \text{Scale}(1/sx, 1/sy, 1/sz)$
 - $\text{Rotate}(\theta, x, y, z)^{-1} = \text{Rotate}(-\theta, x, y, z)$
- 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

- `glTranslate`, `glScale` et `glRotate` multiplient la matrice de transformation courante par celle de la transformation correspondante (post-multiplication, « à droite »)
- Attention à l'ordre d'application des transformations !
- Cf. exemple et slides suivants

Composition de transformations

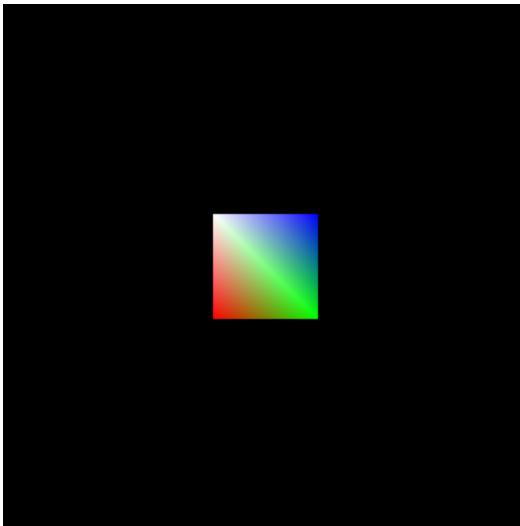
1. `glRect(-0.2f,-0.2f,0.2f,0.2f);`
2. `glRotate(45.0,0,0,1);`
3. `glTranslate(0.5,0,0);`

`glRotate(a,0,0,1)`

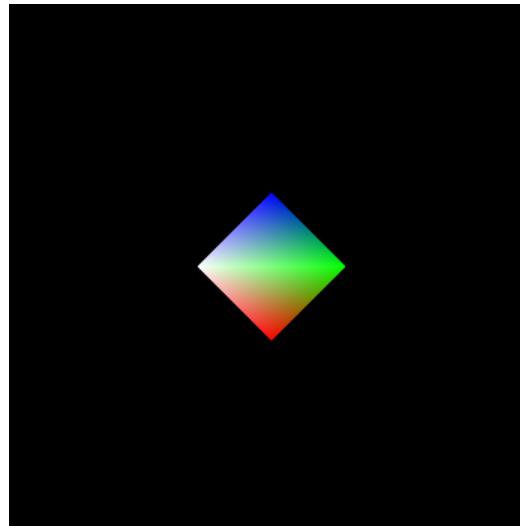
$\cos(a)$	$\sin(a)$	0	0
$-\sin(a)$	$\cos(a)$	0	0
0	0	1	0
0	0	0	1

`glTranslate(tX,tY,tZ)`

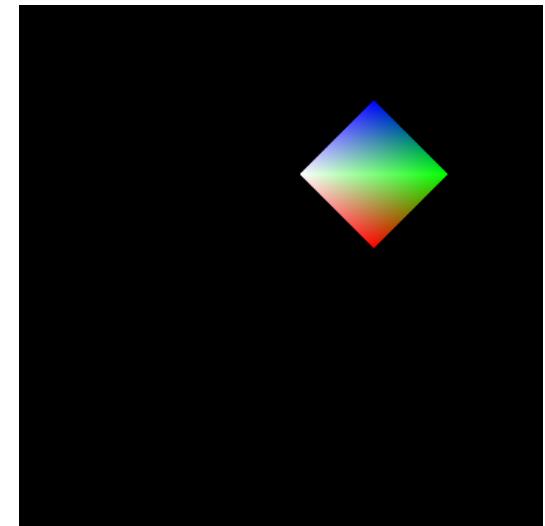
1	0	0	tX
0	1	0	tY
0	0	1	tZ
0	0	0	1



1.



2.



3.

Composition de transformations

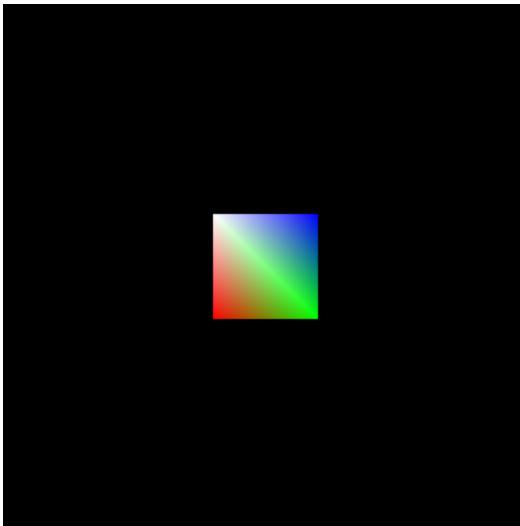
1. `glRect(-0.2f,-0.2f,0.2f,0.2f);`
2. `glTranslate(0.5,0,0);`
3. `glRotate(45.0,0,0,1);`

`glTranslate(tX,tY,tZ)`

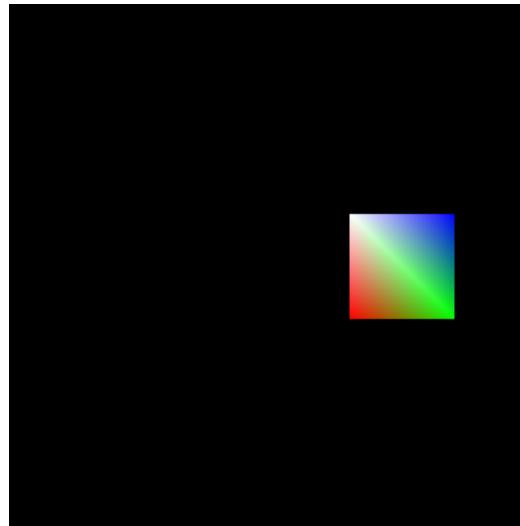
1	0	0	tX
0	1	0	tY
0	0	1	tZ
0	0	0	1

`glRotate(a,0,0,1)`

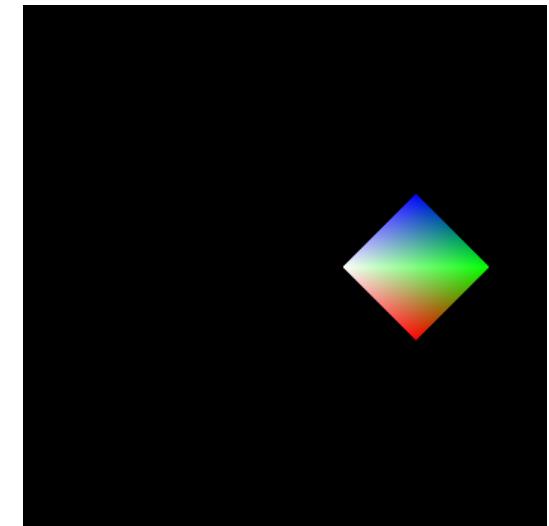
cos(a)	sin(a)	0	0	x
-sin(a)	cos(a)	0	0	y
0	0	1	0	z
0	0	0	1	1



1.



2.



3.

Composition de transformations

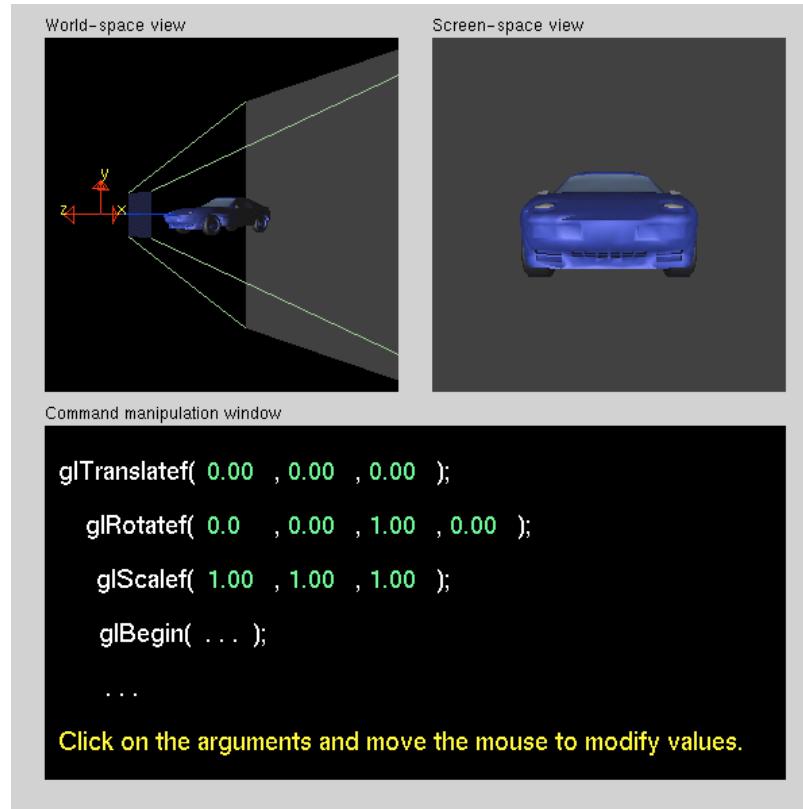
- L'exemple pour mieux comprendre : un **modèle hiérarchique** (p. ex. une grue)
 - La position et l'orientation de chaque élément dépend d'un autre élément auquel il se rattaché
 - Chaque transformation d'un élément influe sur les éléments qui s'y rattachent
 - Post-multiplication de matrices : OpenGL fait ça tout seul !
- Revers de la médaille : les transformations s'additionnant, il est difficile d'appliquer une transformation indépendamment des précédentes

Programmation des transformations en OpenGL

- OpenGL gère deux piles de matrices de transformation : l'une pour la projection et l'autre pour la vue et le modèle
 - On passe de l'une à l'autre grâce à 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 (en haut de la pile) peut être modifiée par les instructions `glLoadIdentity`, `glLoadMatrix*` et `glMultMatrix*`
- Des fonctions existent pour simplifier les translations, les changements d'échelle, les rotations et les projections :
 - `glTranslate*`, `glScale*`, `glRotate*`
 - `glOrtho`, `gluOrtho2D`, `glFrustum` et `gluPerspective`

Transformations en OpenGL

- Tutoriel *Transformation* de Nate Robins :
 - <http://iel.ucdavis.edu/projects/chopengl/demos.html>

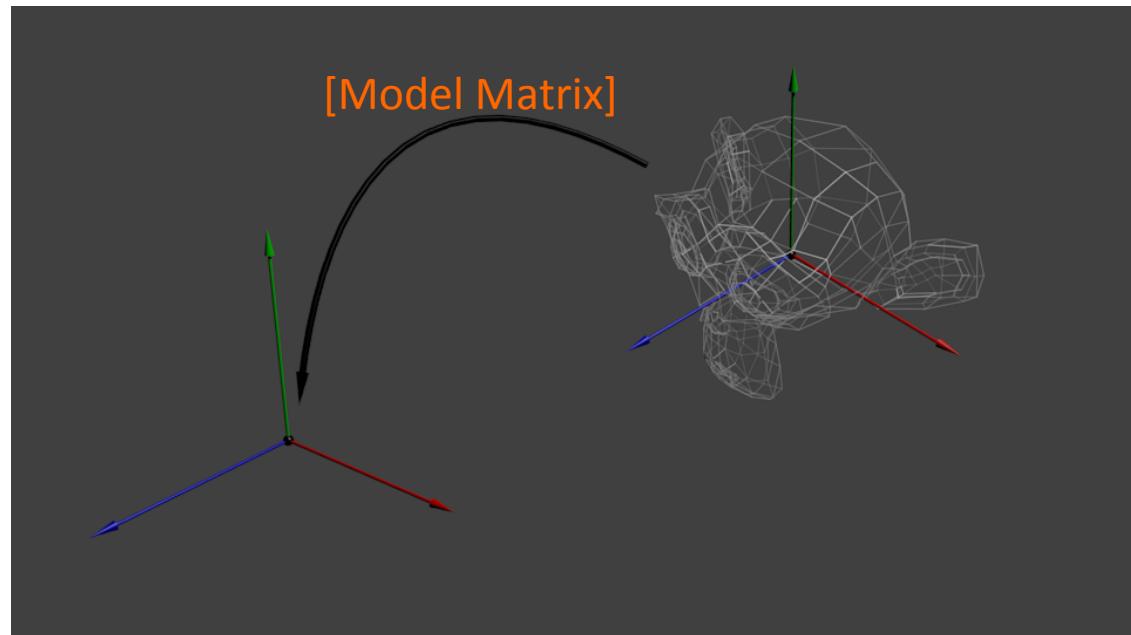
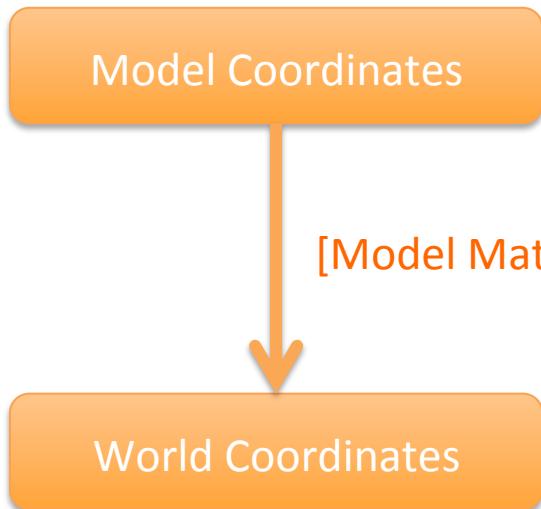


Systèmes de coordonnées et matrices de transformation

- Nous avons vu les différents *systèmes de coordonnées* (*Model Space*, *World Space*, *Camera Space*) et la représentation de transformations géométriques sous forme matricielle
- Il nous reste à lier les deux, i.e. quelles matrices permettent les *changements de systèmes de coordonnées*
- Et à tout combiner pour *passer du monde 3D à une image 2D*

Systèmes de coordonnées et matrices de transformation

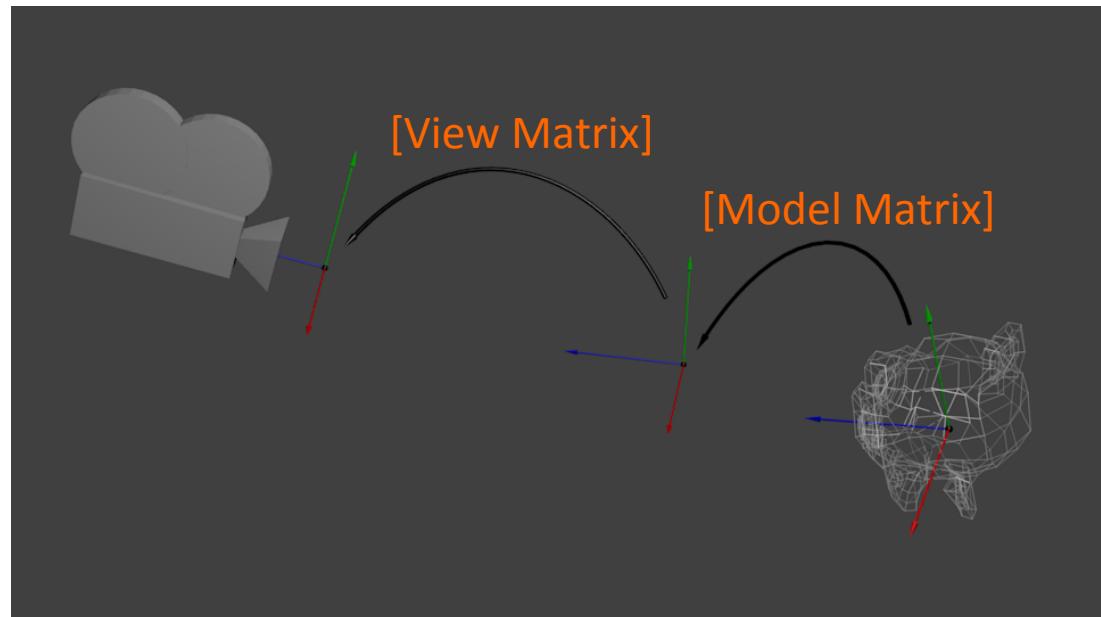
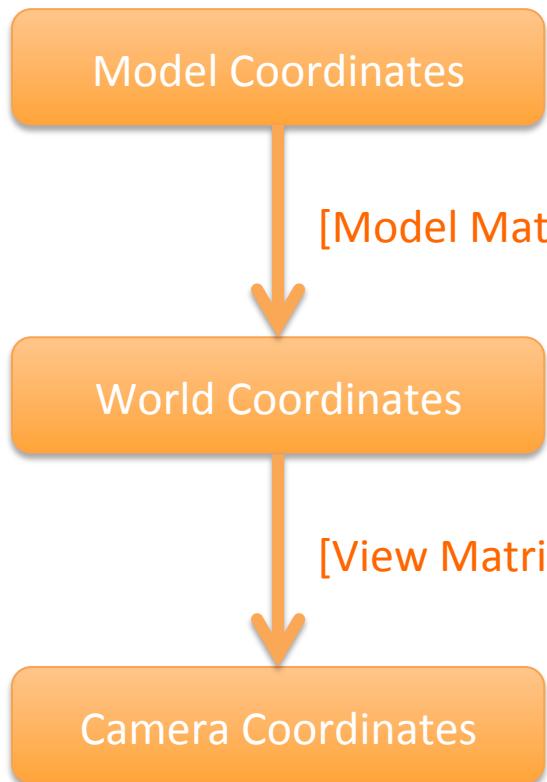
- **Model Matrix** : transforme les sommets du **Model Space** au **World Space**



Source
<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>

Systèmes de coordonnées et matrices de transformation

- **View Matrix** : transforme les sommets du **World Space** au **Camera Space**



Source

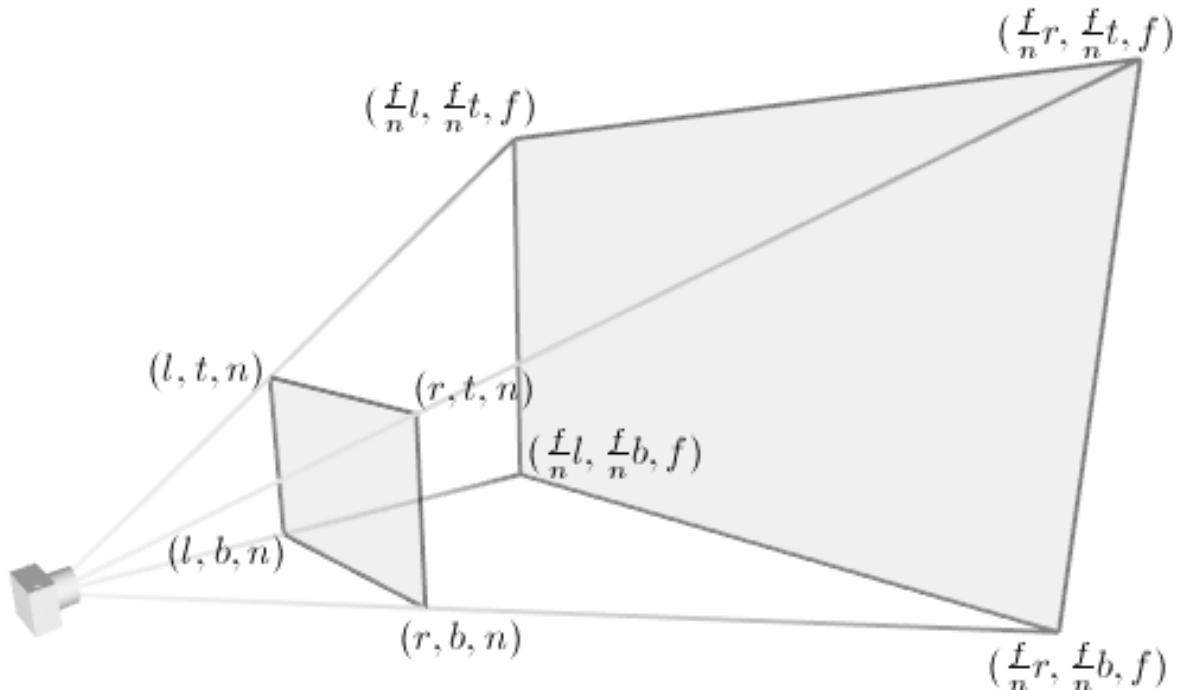
<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>

Et maintenant ?

- Nous sommes arrivés dans le **Camera Space** !
- Cela signifie qu'un vertex de coordonnées (0,0) sera affiché au centre de l'écran.
- Mais que faire de la coordonnées Z ? → Elle doit être prise en compte également !
- C'est ce le but de la **matrice de projection** : elle définit un **volume de visualisation (viewing volume)**
- Il existe deux sortes de projections :
 - Projection perspective
 - Projection orthographique ou parallèle
- Le **volume de visualisation** est différent en fonction de la projection utilisée

Projection Perspective

- La projection perspective définit la volume de visualisation comme une pyramide tronquée : le **frustum** de vue



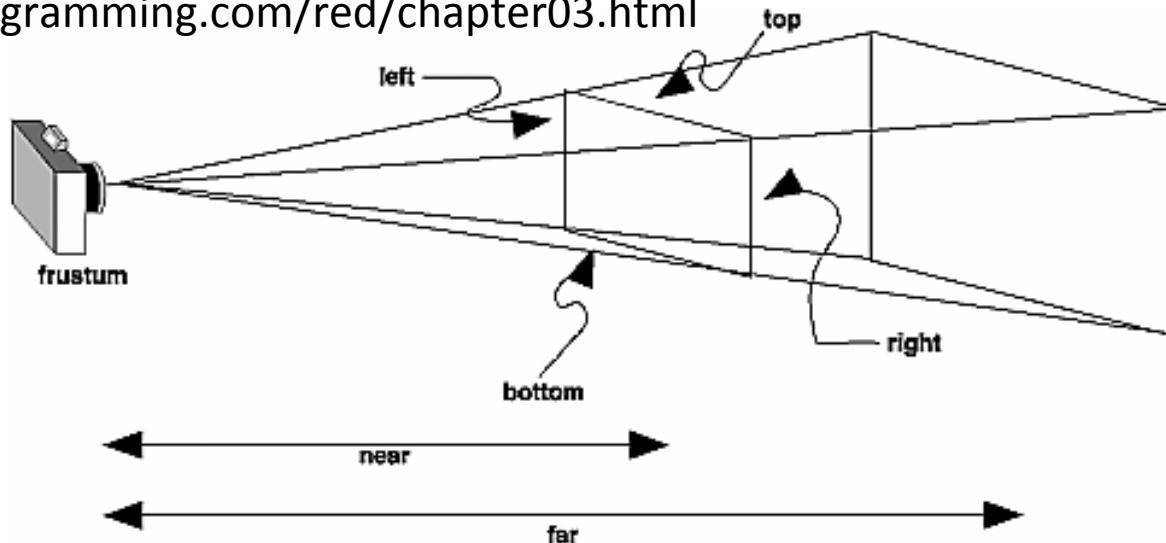
Source

http://www.songho.ca/opengl/gl_transform.html

Projection Perspective

- Il existe deux fonctions OpenGL permettant de définir ce **frustum** :
 - **glFrustum**
 - void **glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);**

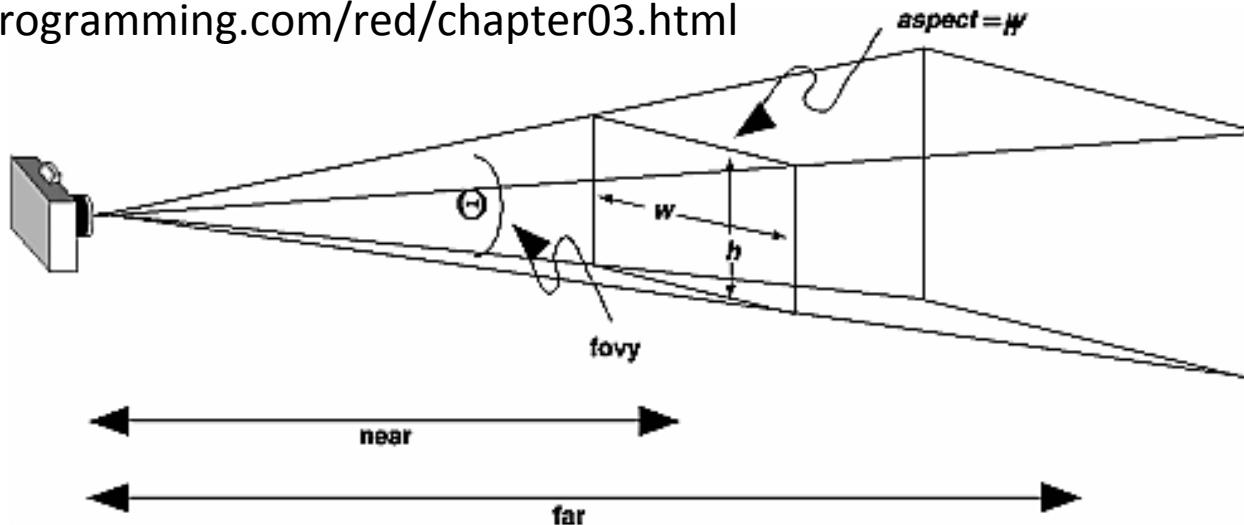
<http://www.glprogramming.com/red/chapter03.html>



Projection Perspective

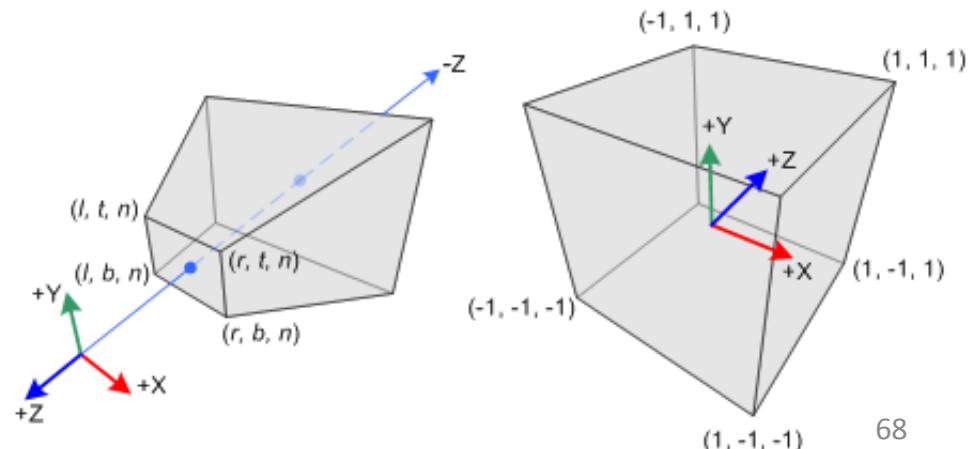
- Il existe deux fonctions OpenGL permettant de définir ce **frustum** :
 - **gluPerspective**
 - void **gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);**

<http://www.glprogramming.com/red/chapter03.html>



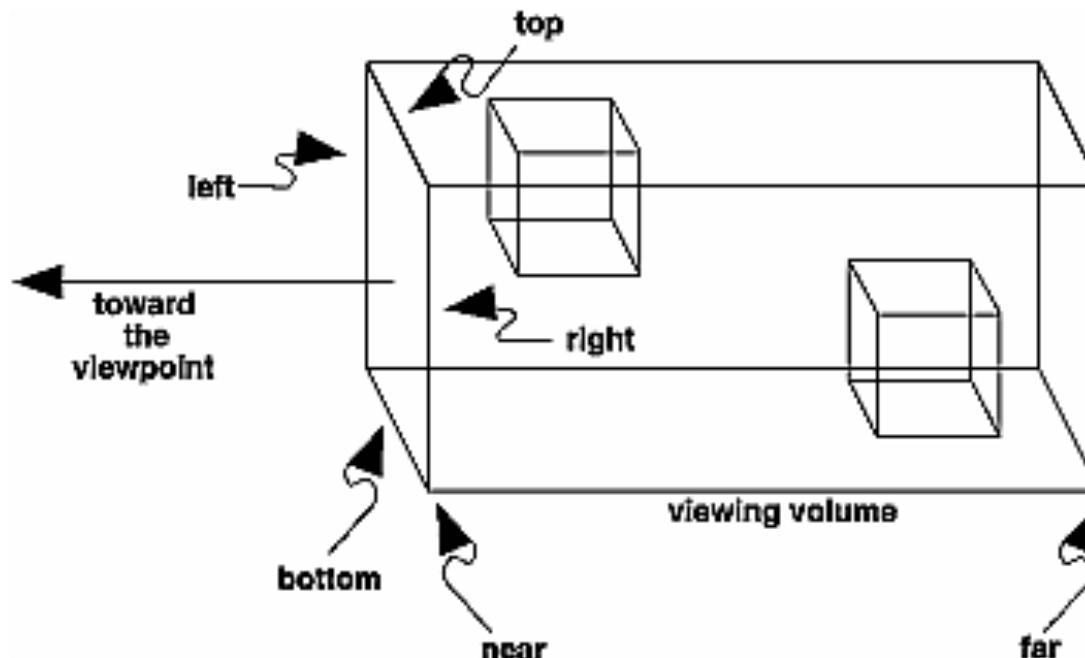
Projection Perspective

- Ce **frustum** de vue définit ce que l'on appelle l'**Eye Space** ou système de coordonnées de l'œil.
- Il nous reste une dernière étape de projection qui vise à transformer ce **frustum** en un **cube unitaire** ($[-1,1]$ pour chaque axe) appelé **Normalized Device Coordinates (NDC)** :
 - $[\text{left}, \text{right}] \rightarrow [-1,1]$
 - $[\text{bottom}, \text{top}] \rightarrow [-1,1]$
 - $[\text{near}, \text{far}] \rightarrow [-1,1]$



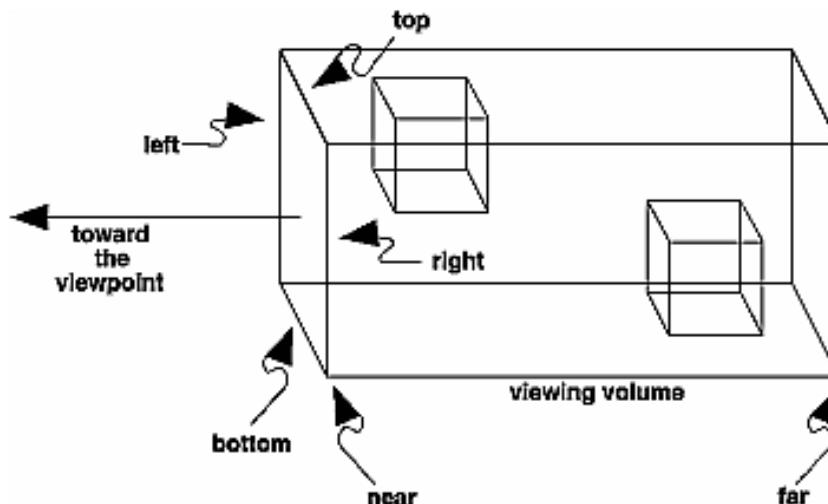
Projection Orthographique (Parallèle)

- La projection perspective définit la volume de visualisation comme un **parallélépipède** aussi appelé une « boîte »



Projection Orthographique (Parallèle)

- Contrairement à la projection perspective, le **volume de vue** a les mêmes dimensions tout au long de son axe principal → la **distance de la caméra** a un point 3D n'a pas d'influence sur la projection de ce point!

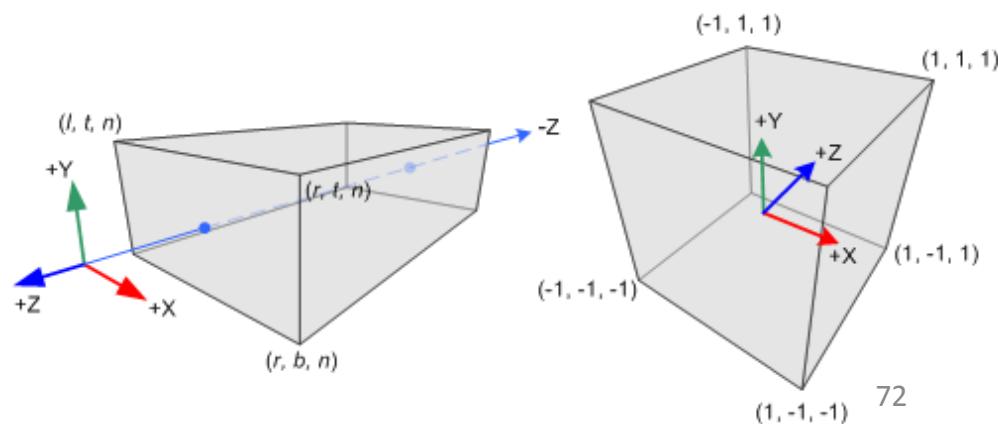


Projection Orthographique (Parallèle)

- Il existe deux fonctions en OpenGL pour réaliser une **projection orthographique (parallèle)** :
 - **glOrtho**
 - void **glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);**
 - **gluOrtho2D**
 - void **glOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);**
 - Ici **near** vaut **-1** et **far** vaut **+1**

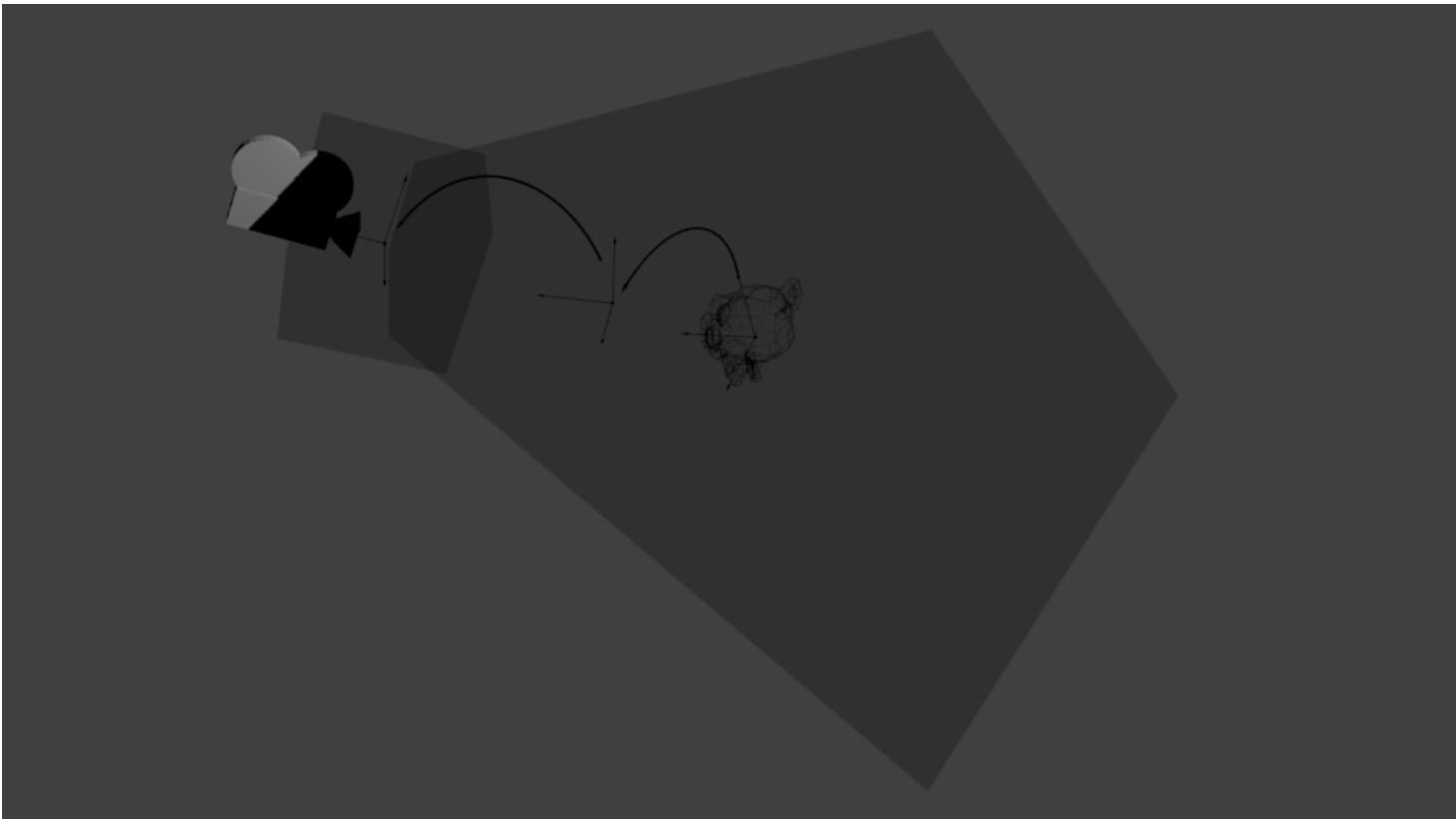
Projection Orthographique (Parallèle)

- Le **parallélépipède** définit ce que l'on appelle **l'Eye Space** ou système de coordonnées de l'œil.
- Il nous reste une dernière étape de projection qui vise à transformer ce **volume** en un **cube unitaire** ($[-1,1]$ pour chaque axe) appelé **Normalized Device Coordinates (NDC)** :
 - $[\text{left}, \text{right}] \rightarrow [-1,1]$
 - $[\text{bottom}, \text{top}] \rightarrow [-1,1]$
 - $[\text{near}, \text{far}] \rightarrow [-1,1]$



Matrice de Projection

- Comme nous venons de le voir nous devons transformer le **volume de vue** en un **cube unitaire**



Matrice de Projection

- C'est le rôle joué par la matrice de projection !

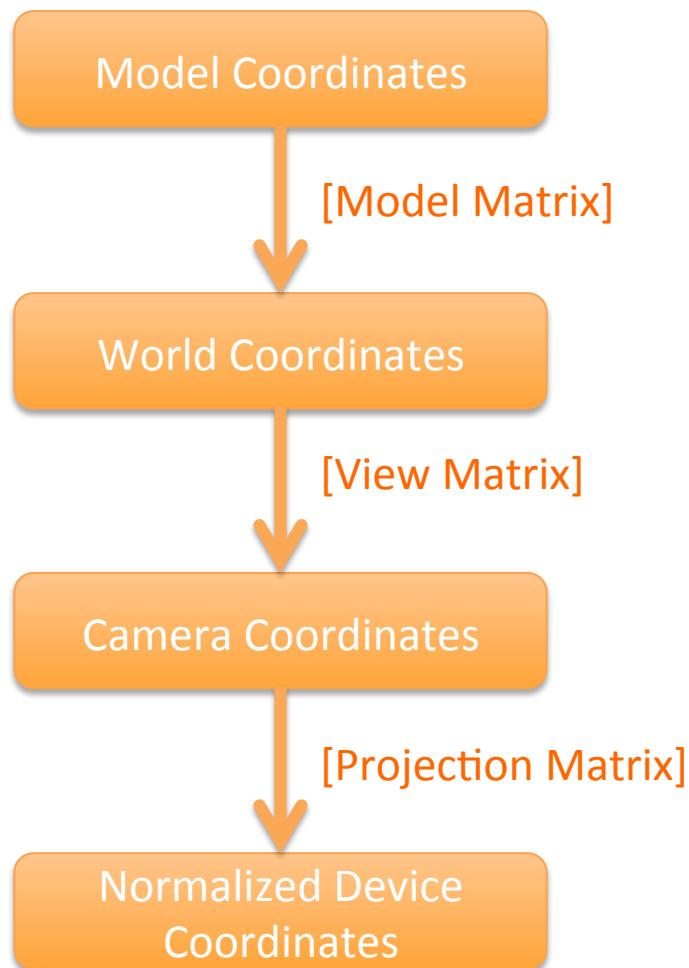


Illustration du rôle de la matrice de projection

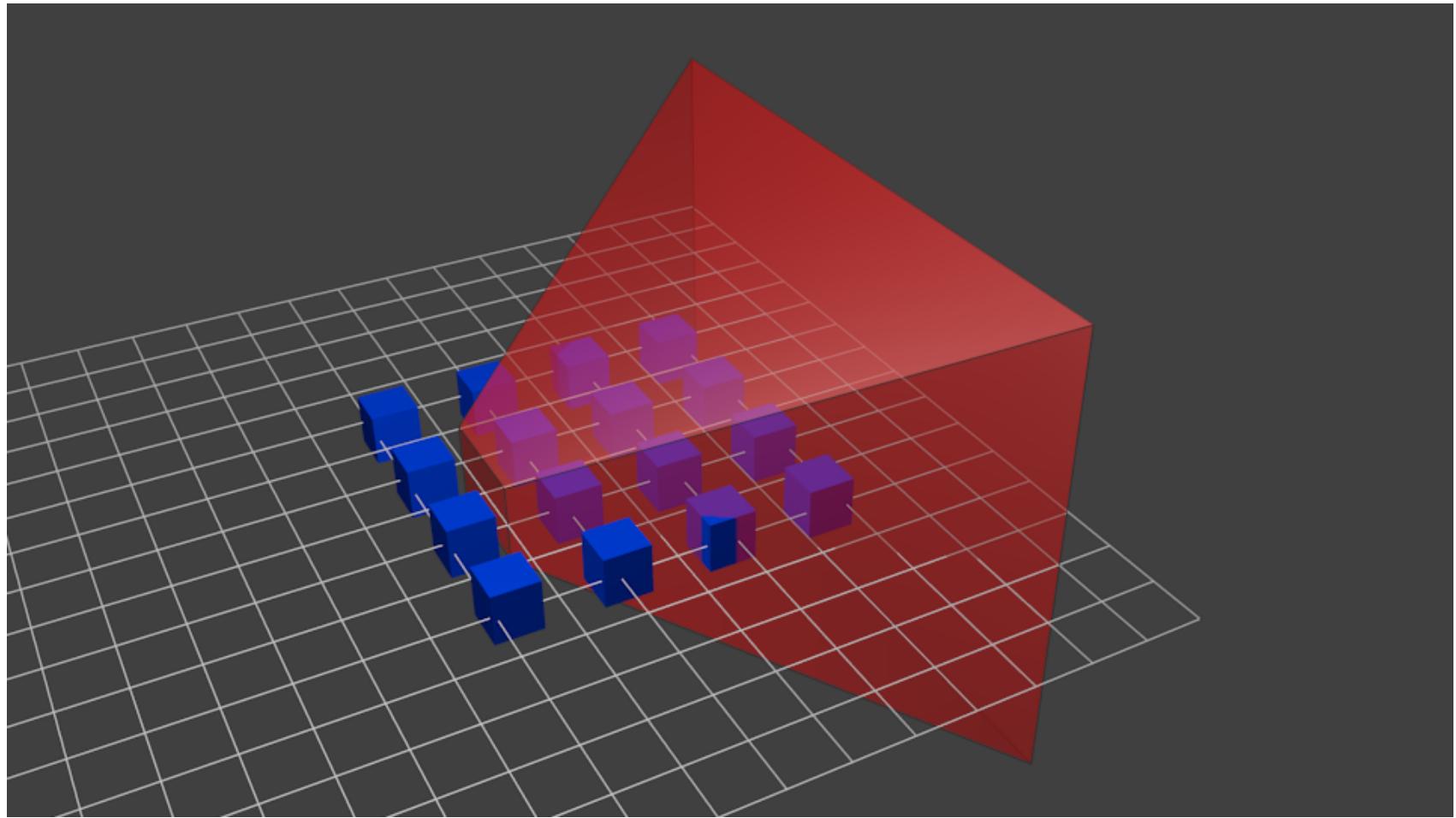


Illustration du rôle de la matrice de projection

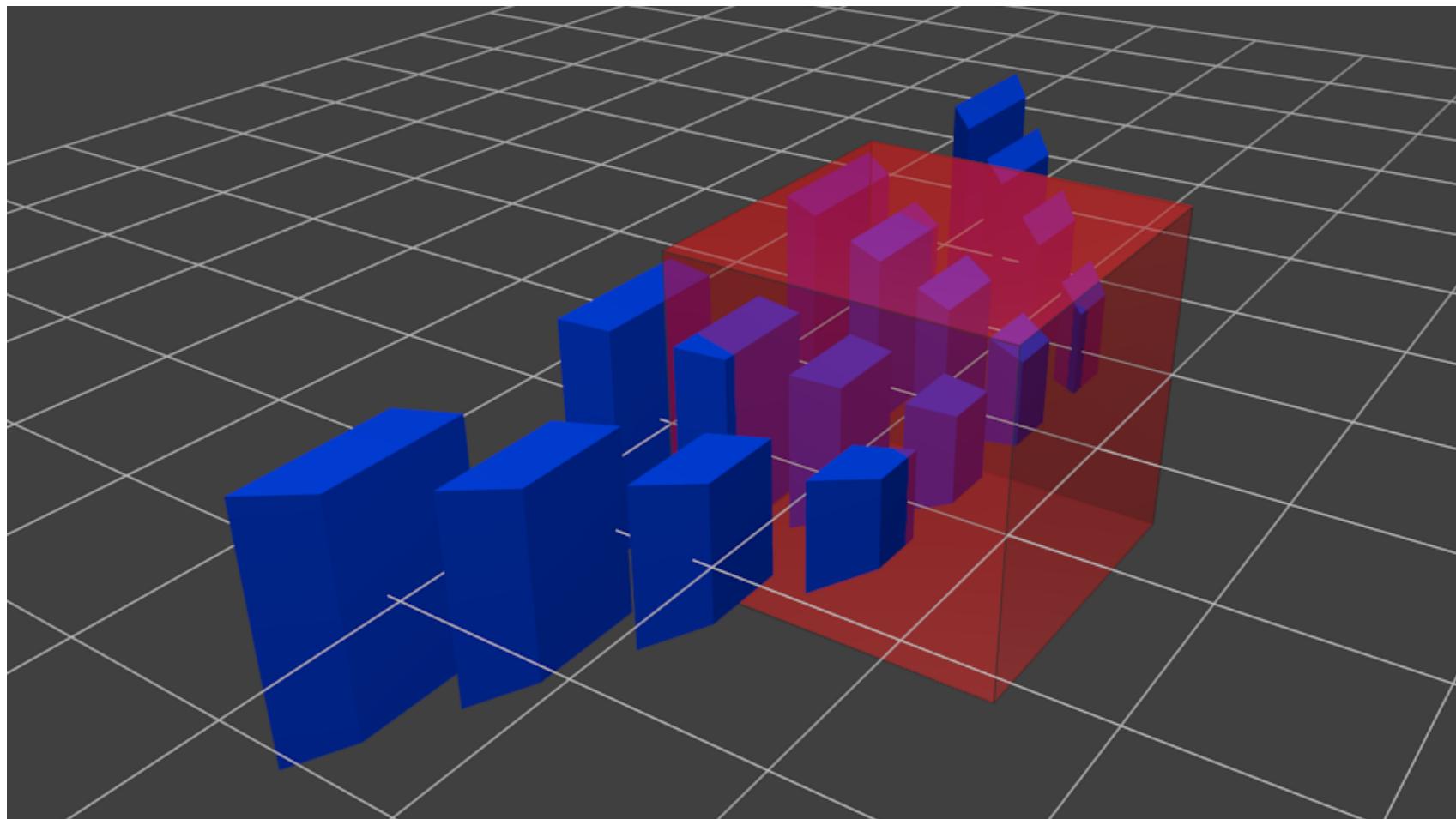
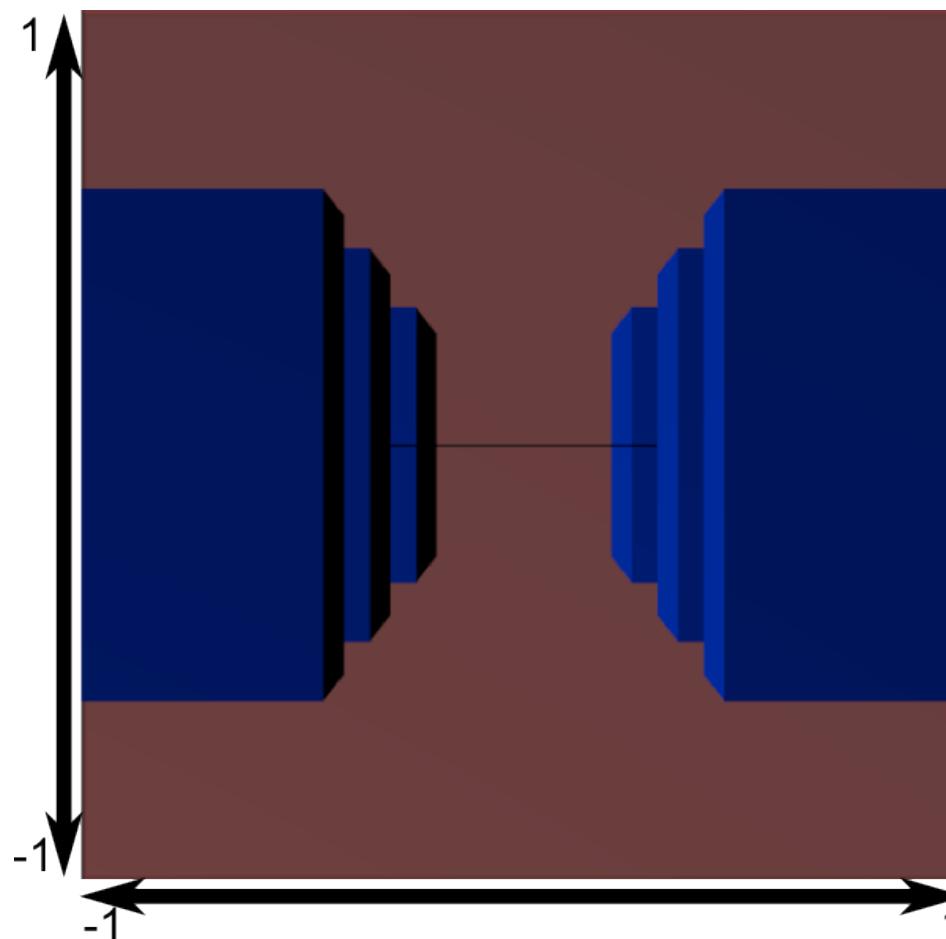


Illustration du rôle de la matrice de projection

- Voici le résultat vu depuis la caméra :



On a presque fini !

- Nous sommes dans le repère normalisé NDC ou tout est compris entre -1 et 1
- La dernière étape est de passer dans le repère de l'écran (**Screen Space**) ou plus précisément de la fenêtre (**viewport**) qui affichera notre vue OpenGL

Fenêtre OpenGL

- On crée une fenêtre GLUT grâce à la fonction `glutCreateWindow` :
 - `void glutCreateWindow(char *name);`
 - `name` : chaîne de caractères représentant le nom de la fenêtre
- Attention n'oubliez pas d'initialiser la position et la taille de la fenêtre :
 - `void glutInitWindowPosition(int x, int y);`
 - `void glutInitWindowSize(int width, int height);`
 - Valeur par défaut de `x` & `y` : -1
 - Valeur par défaut de `width` & `height` : 300

Fenêtre OpenGL

- La fonction `glViewport` permet de représenter la transformation affine entre les coordonnées NDC et celles de la fenêtre OpenGL :
 - void `glViewport(GLint x, GLint y, GLint width, GLint height);`
 - (`x,y`) : coin bas gauche de la fenêtre
 - `width, height` : largeur et hauteur de la fenêtre (en pixels)
 - Attention l'« `aspect-ratio` » (rapport largeur/hauteur) du `viewport` doit être le même que celui définit par la `projection` sinon la vue sera déformée

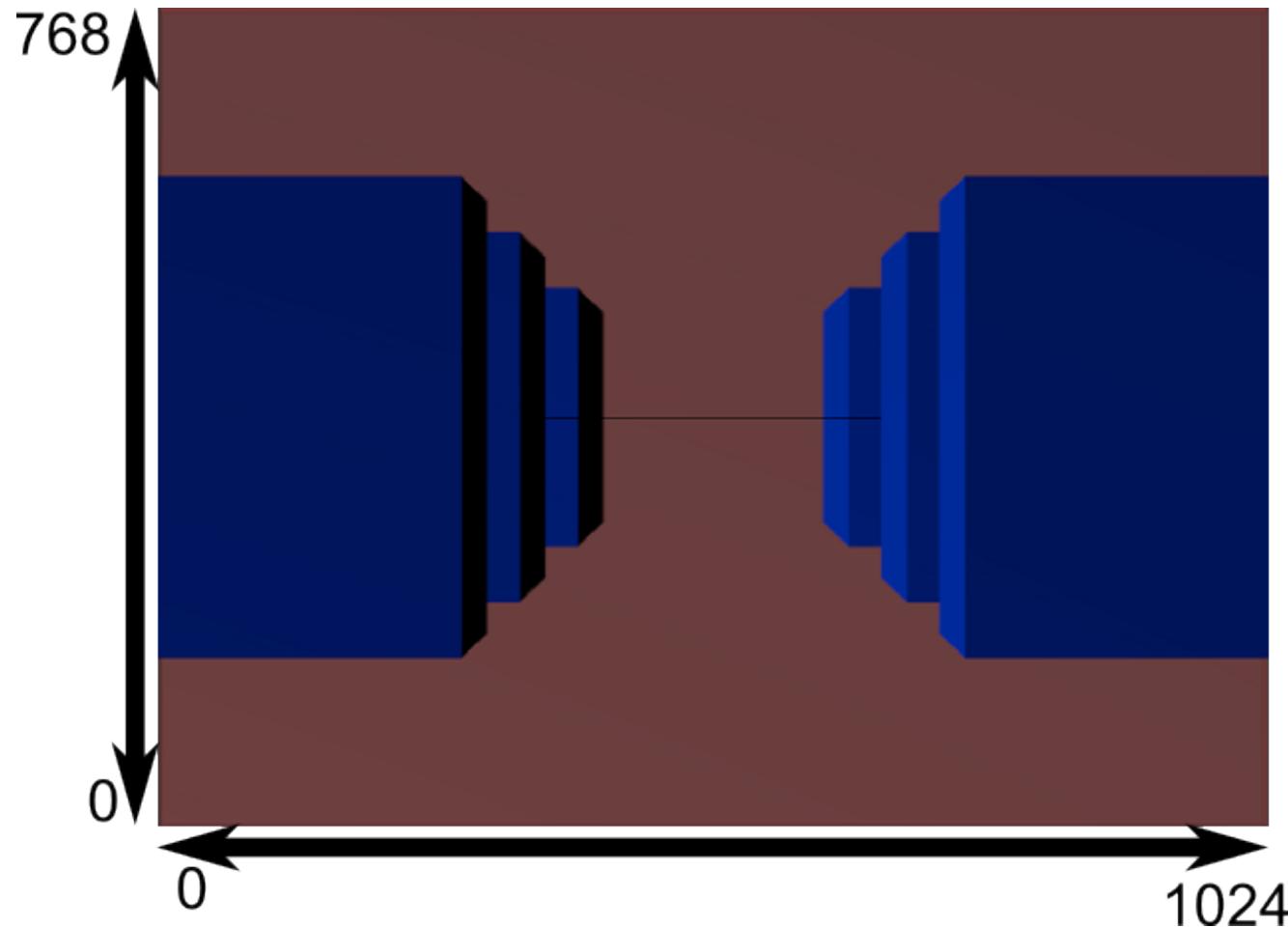
Fenêtre OpenGL

- Transformation entre les coordonnées NDC et la fenêtre OpenGL
- `void glViewport(GLint x, GLint y, GLint width, GLint height);`
- Soient (x_{ndc}, y_{ndc}) les coordonnées NDC, alors (x_w, y_w) les coordonnées de la fenêtre sont calculées de la manière suivante :

$$x_w = (x_{ndc} + 1) \left(\frac{width}{2} \right) + x$$

$$y_w = (y_{ndc} + 1) \left(\frac{height}{2} \right) + y$$

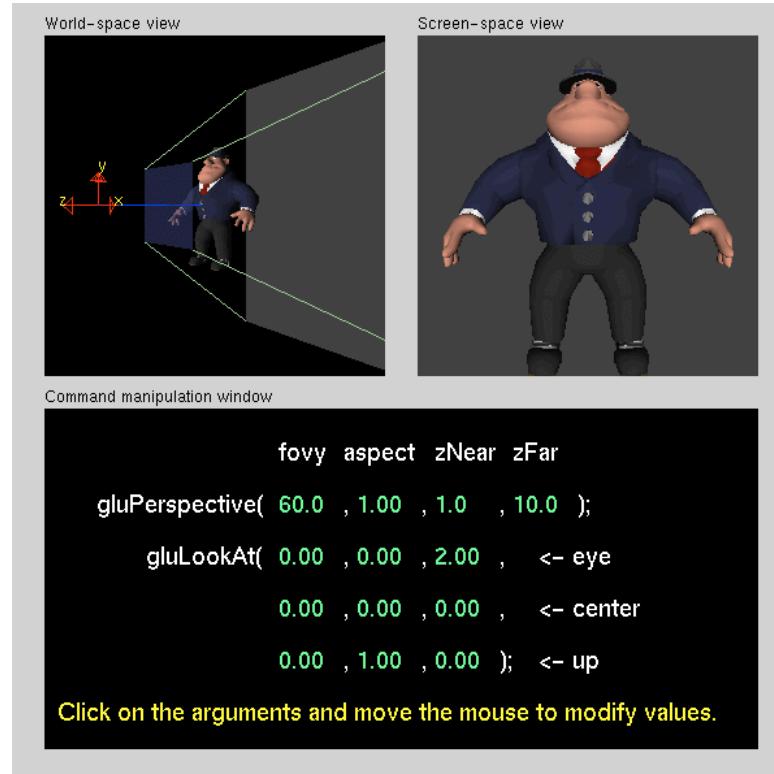
Screen Space



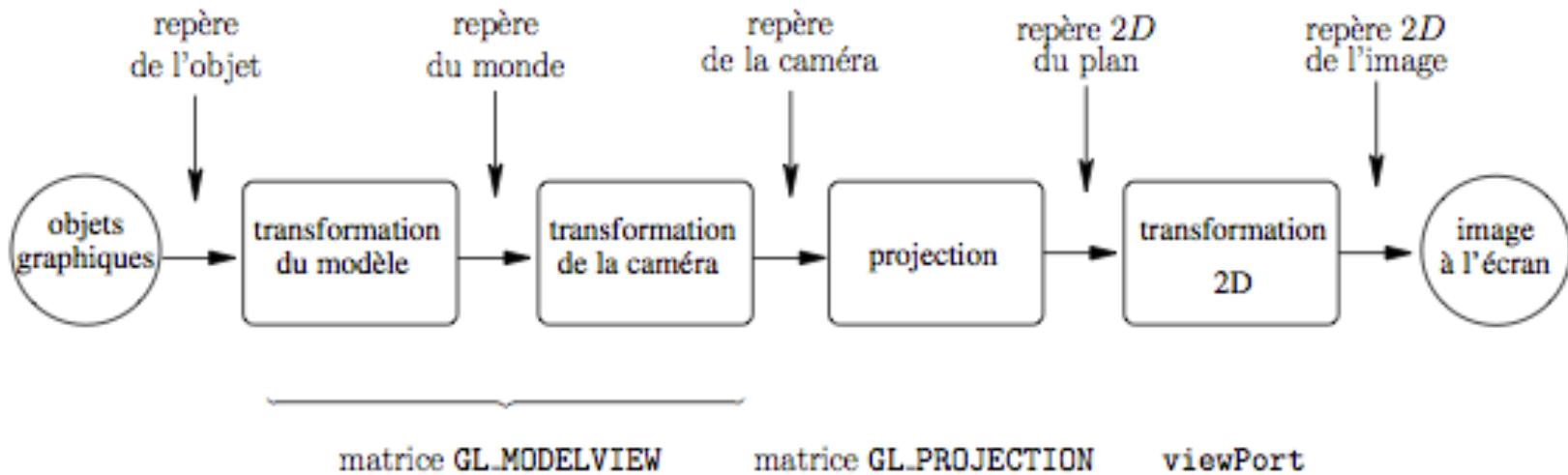
Avec une fenêtre de largeur (width) 1024 et hauteur (height) 768

Pour mieux comprendre

- Tutoriel *Projection* de Nate Robins :
 - <http://iel.ucdavis.edu/projects/chopengl/demos.html>



Bilan des transformations

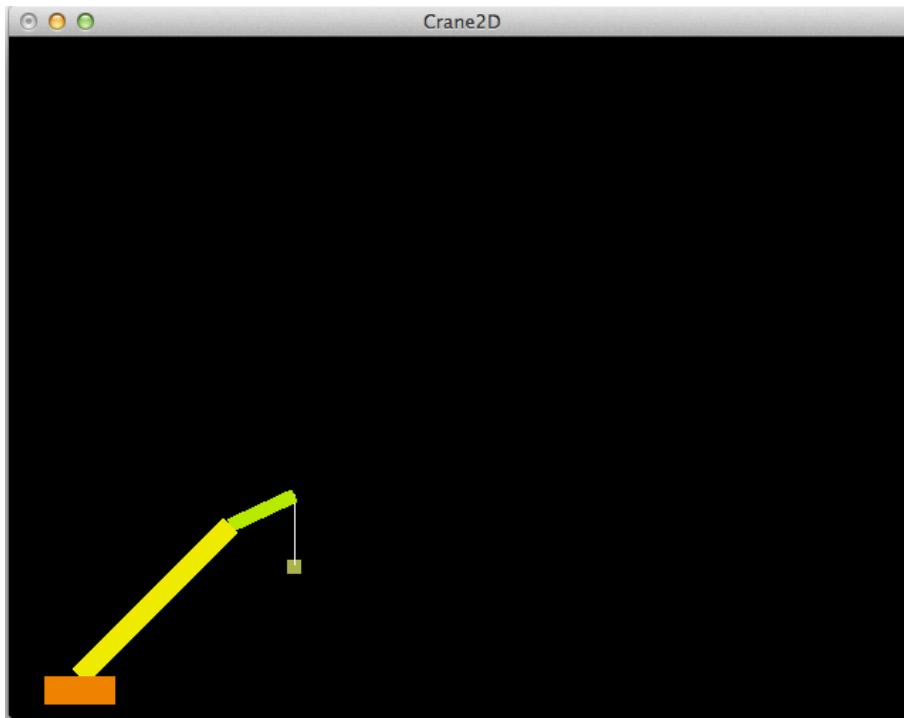


Exemple : Grue 2D

- Créez un projet OpenGL en utilisant GLUT pour le système de fenêtrage et l'interaction utilisateur (clavier) :
 - Visual Studio
 - Linux
- Vous devrez être capables d'afficher une **grue 2D** que l'on contrôlera avec les flèches du clavier
- La grue est constituée de 3 rectangles et une caisse est attachée au bout de la grue

Exemple : Grue 2D

- Utilisez éventuellement le squelette préparé pour vous (si vous le souhaitez)
- Résultat souhaité (ou approchant) :



Exemple : Grue 2D

- Pour avoir des bases communes, utilisez les valeurs suivantes (au moins au début) :
 - Largeur de la fenêtre : 640
 - Hauteur de la fenêtre : 480
 - Base de la grue : 50 x 20 unités (largeur x hauteur)
 - Bras 1 : 150 x 15 unités (largeur x hauteur)
 - Bras 2 : 50 x 10 unités (largeur x hauteur)
 - Caisse : 10 x 10 unités (largeur x hauteur)
 - Longueur initiale du filin : 50
 - Angle initial pour Bras 1 : 45° (`angle1`)
 - Angle initial pour Bras 2 : -20° (`angle2`)

Exemple : Grue 2D

- Interactions possibles :
 - Modifier `angle1` (augmenter ou diminuer)
 - Modifier `angle2` (augmenter ou diminuer)
 - Modifier la longueur du filin (augmenter ou diminuer)
 - Modifier le mode d'affichage des polygones (cf. premier exemple)
 - Quitter avec la touche ‘q’ ou ESCAPE

Exemple : Grue 2D

- Fonction OpenGL à connaître pour définir le repère 2D de la fenêtre **en dehors de l'intervalle $[-1,1] \times [-1,1]$** :
 - void **gluOrtho2D(int xMin, int xMax, int yMin, int yMax);**
- Définit le repère de la fenêtre comme étant : **[xMin,xMax] x [yMin,yMax]** (x dans $[-1,1]$ en Z)
- Cette fonction va modifier la projection de notre scène OpenGL → elle doit être appelée après un appel à **glMatrixMode(GL_PROJECTION)**

Exemple : Grue 2D

- Exemple d'utilisation de gluOrtho2D, dans une fonction dédiée à la mise à jour de la fenêtre OpenGL en cas de redimensionnement :

```
// Callback de redimensionnement de la fenêtre
GLvoid redimensionner(int w, int h) {
    // On stocke les valeurs dans nos variables
    // globales (moche mais simple)
    windowW = w;
    windowH = h;

    // eviter une division par 0
    if(windowH==0)
        windowH = 1;

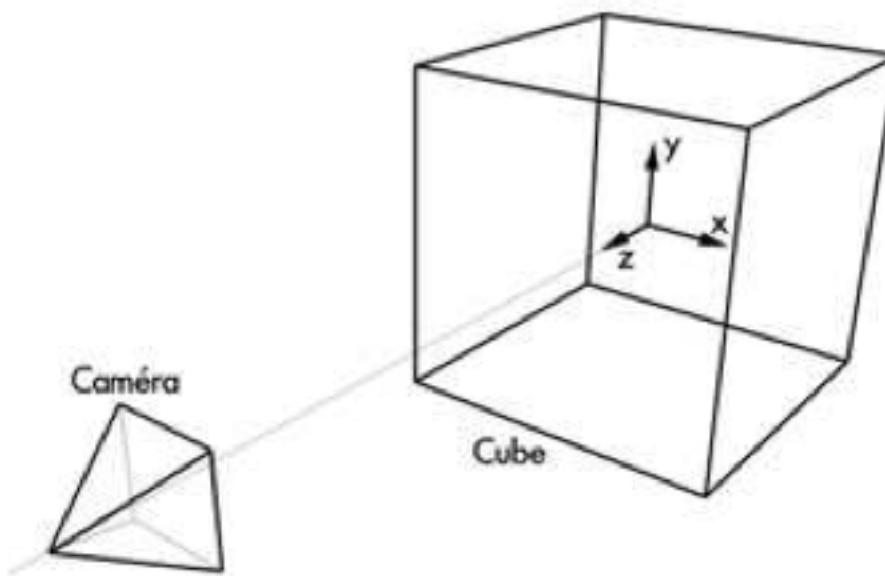
    float ratio = (float)windowW / (float)windowH;
    // on verifie le calcul du ratio
    std::cout << "Ratio : " << ratio << std::endl;
    // suite de la fonction
    // On va modifier la pile de Projection
    glMatrixMode(GL_PROJECTION);
    // On charge la matrice d'identite
    glLoadIdentity();
    // Appel a Ortho2D
    gluOrtho2D(0, windowW, 0, windowH);
    // On retourne a la pile modelview
    glMatrixMode(GL_MODELVIEW);
} // fin de la fonction redimensionner
```

Exemple : Grue 2D

- À vous de jouer !

Affichons de la 3D !

- Afin d'illustrer ce que nous venons de voir, nous allons réaliser un grand classique en affichant un **cube 3D en rotation** !



Cube3D

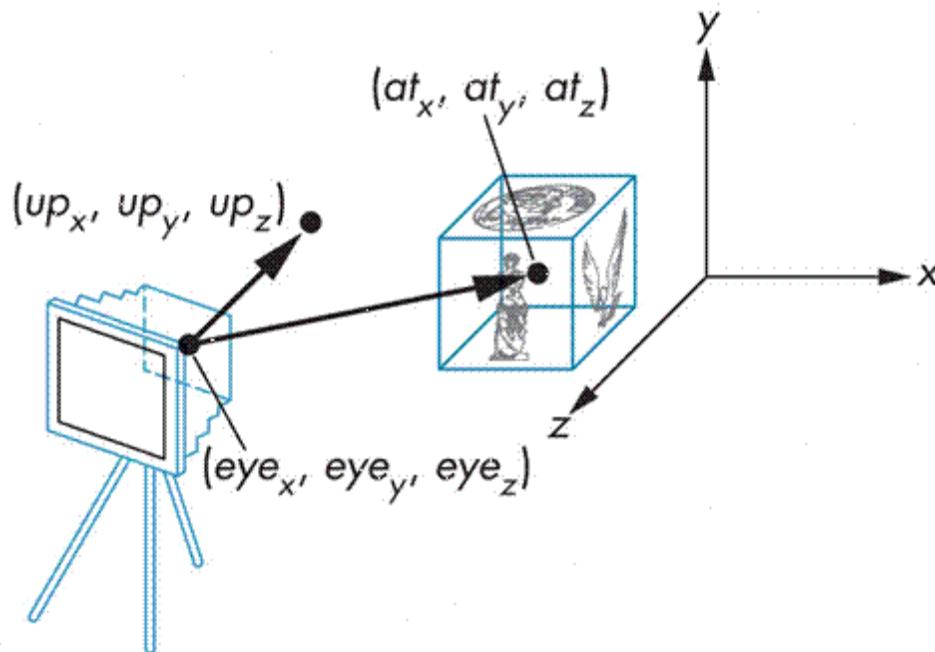
- Que devons-nous faire :
 - Créer une fenêtre et définir une **projection perspective**
 - Afficher un cube (soit 8 sommets et 6 faces)
 - Animer le cube (le faire tourner)
 - Gérer les fonctions de callbacks pour l'interface
- Nous profiterons de cet exemple pour présenter de **nouveaux problèmes** !

Cube3D

- Gestion de la perspective :
 - Nous allons utiliser deux fonctions OpenGL (ou plus précisément GLU)
 - `gluPerspective` qui nous permet de régler la perspective de la scène (déjà vue précédemment)
 - `gluLookAt` qui permet de bouger la position et l'orientation de la caméra OpenGL !
 - Ensuite il faudra gérer le redimensionnement de la fenêtre → `gluPerspective` utilise le ratio de la fenêtre
 - Donc si la fenêtre est redimensionnée son ratio (largeur/hauteur) peut être modifié et donc la projection perspective ne serait plus correcte !

Cube3D

- **gluLookAt** permet de positionner la caméra
 - void **gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
GLdouble centerx, GLdouble centery, GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);**



Cube3D

- Redimensionnement de la fenêtre → GLUT propose une fonction de rappel permettant de détecter les redimensionnements de la fenêtre : `glutReshapeFunc`
- Prototype de la fonction de rappel :

```
GLvoid redimensionner(int w, int h);
```
- Le paramètre `w` (resp. `h`) est la nouvelle **largeur** (resp. **hauteur**) de la fenêtre GLUT
- Ainsi nous pouvons recalculer la projection perspective (`gluPerspective`) et le `gluLookAt`
- NB : La fonction de rappel est appelée aussi lors de la création de la fenêtre GLUT → il suffit de placer les appels à `gluPerspective` et `gluLookAt` dans cette fonction pas besoin de les appeler à l'initialisation

Cube3D

- Animation du cube
- Nous allons faire tourner le cube grâce aux mouvements de la souris
- Il faut appliquer au cube des rotations autour des axes X et Y grâce à la fonction `glRotatef` // Animation du cube!
`glLoadIdentity();`
`glRotatef(-angleY,1.0f,0.0f,0.0f);`
`glRotatef(-angleX,0.0f,1.0f,0.0f);`
- La fonction `glLoadIdentity` réinitialise la matrice courante
- Nous appliquons une rotation de `-angleX` autour de `Y` et de `-angleY` autour de `X`
- Pourquoi ?
 - Un mouvement `horizontal` de la souris fera tourner le cube autour de l'axe `Y`
 - Un mouvement `vertical` de la souris fera tourner le cube autour de l'axe `X`
- Il nous faut déterminer les valeurs de `angleX` et `angleY`

Cube3D

- Les fonctions de rappel liées à la souris
- Pour la gestion des boutons, nous utiliserons la fonction de rappel `glutMouseFunc` qui est appelée à chaque fois qu'un bouton de la souris est enfoncé ou relâché
- Le prototype de la fonction associée est :
`GLvoid souris(int bouton, int etat, int x, int y);`
- GLUT définit les boutons suivants : `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` et `GLUT_RIGHT_BUTTON`
- GLUT définit les états suivants pour chaque bouton : `GLUT_DOWN` et `GLUT_UP` lorsque un bouton est respectivement enfoncé ou relâché
- Nous appliquerons des mouvements au cube uniquement lorsque le bouton gauche de la souris est enfoncé
- Il nous faudra nous rappeler des positions de la souris lorsque le bouton gauche est cliqué
 - `oldX` = position de la souris en X lorsque le bouton gauche est cliqué
 - `oldY` = position de la souris en Y lorsque le bouton gauche est cliqué

Cube3D

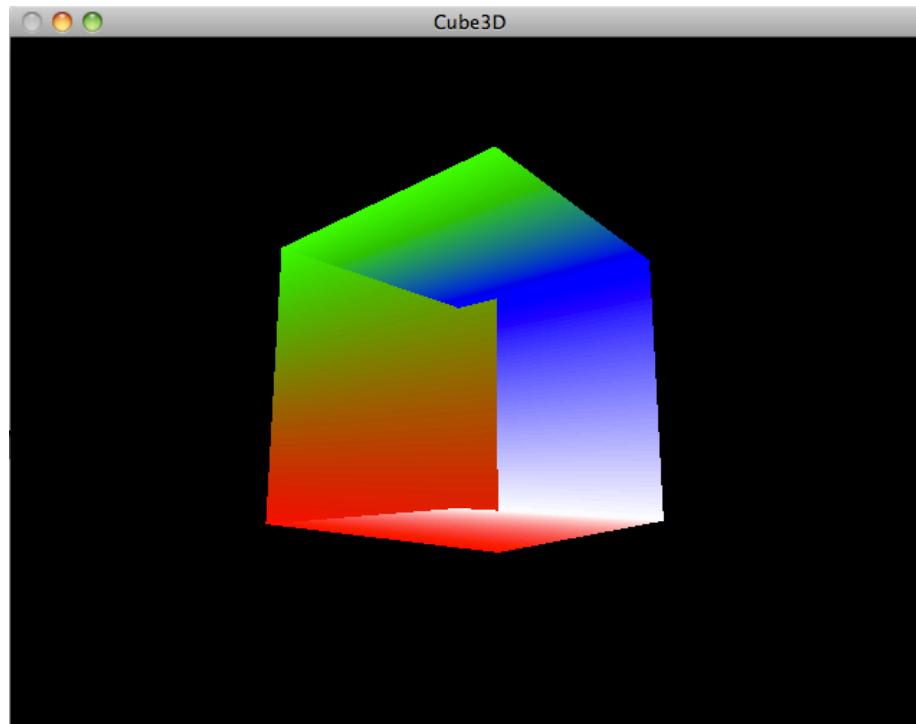
- Il faut également être capable de suivre les mouvements de la souris, la fonction de rappel GLUT associée est **glutMotionFunc**
- Le prototype de la fonction associée est :
GLvoid **deplacementSouris**(int **x**, int **y**);
- On peut calculer les valeurs de **angleX** et **angleY** en fonction des mouvements de la souris et des valeurs de **oldX** et **oldY**
- Dans notre programme ces valeurs sont mises à jour uniquement si le bouton gauche est enfoncé !

Cube3D

- À vous de jouer !

Cube3D

- Résultat : cf. démo



- Le cube est bizarre! → Quel est le souci ?

Cube3D

- Le problème est l'ordre d'affichage des faces du polygone !
- Pour une scène en 3D pour que le résultat soit correct, il faut afficher les faces (ou triangles) de la plus éloignée à la plus proche !
- Pour résoudre ce problème OpenGL utilise un tampon de profondeur aussi appelé Z-buffer

Le tampon de profondeur

- Nous avons vu qu'OpenGL utilise un **tampon colorimétrique** afin d'afficher la scène (ce que nous avons appelé avant le *framebuffer*)
- Le **tampon de profondeur (*Z-buffer*)** est un autre tampon de la même taille que le framebuffer qui va stocker pour chaque pixel une **profondeur** : la **distance entre la caméra et l'objet auquel correspond le pixel considéré**

Le tampon de profondeur



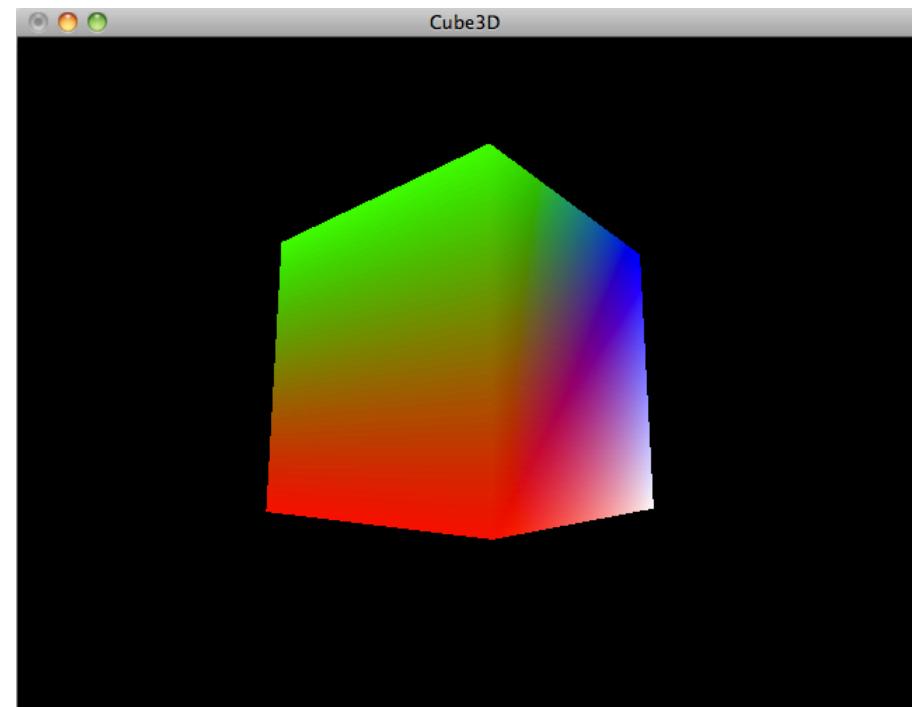
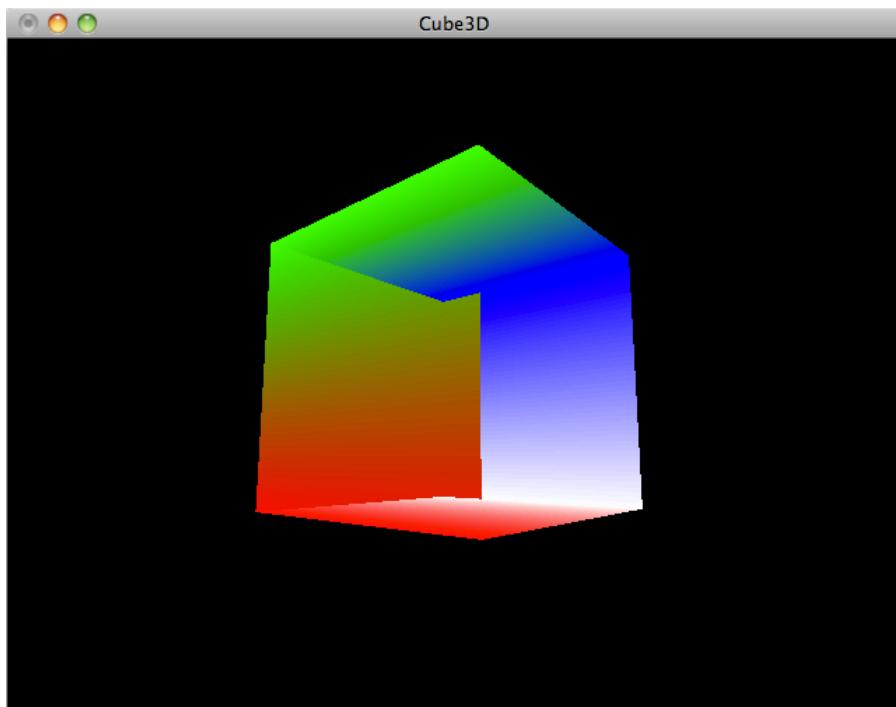
$$\begin{array}{c} \text{Initial Depth Buffer:} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline & \infty \\ \hline \end{array} \end{array} + \begin{array}{c} \text{Depth Map:} \\ \begin{array}{|c|c|c|c|c|c|c|} \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ \hline \end{array} \end{array} = \begin{array}{c} \text{Resulting Depth Buffer:} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & \infty \\ \hline & 5 & \infty \\ \hline & \infty \\ \hline \end{array} \end{array}$$
$$\begin{array}{c} \text{Initial Depth Buffer:} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & \infty & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & \infty & \infty & \infty \\ \hline & 5 & 5 & 5 & 5 & \infty & \infty & \infty & \infty \\ \hline & 5 & 5 & 5 & \infty & \infty & \infty & \infty & \infty \\ \hline & 5 & 5 & \infty & \infty & \infty & \infty & \infty & \infty \\ \hline & 5 & \infty \\ \hline & \infty \\ \hline \end{array} \end{array} + \begin{array}{c} \text{Depth Map:} \\ \begin{array}{|c|c|c|c|c|c|c|} \hline & 7 & & & & & & \\ \hline & 6 & 7 & & & & & \\ \hline & 5 & 6 & 7 & & & & \\ \hline & 4 & 5 & 6 & 7 & & & \\ \hline & 3 & 4 & 5 & 6 & 7 & & \\ \hline & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline & & & & & & & \\ \hline \end{array} \end{array} = \begin{array}{c} \text{Resulting Depth Buffer:} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 5 & 5 & 5 & 5 & 5 & 5 & 5 & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & 5 & \infty & \infty \\ \hline & 5 & 5 & 5 & 5 & 5 & \infty & \infty & \infty \\ \hline & 5 & 5 & 5 & 5 & \infty & \infty & \infty & \infty \\ \hline & 4 & 5 & 5 & 7 & \infty & \infty & \infty & \infty \\ \hline & 3 & 4 & 5 & 6 & 7 & \infty & \infty & \infty \\ \hline & 2 & 3 & 4 & 5 & 6 & 7 & \infty & \infty \\ \hline & \infty \\ \hline \end{array} \end{array}$$

Cube3D

- Mise en place d'un tampon de profondeur avec GLUT
- Il faut modifier l'appel à la fonction `glutInitDisplayMode` :
 - `glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);`
- OpenGL permet d'activer ou non le tampon de profondeur grâce à la variable d'état `GL_DEPTH_TEST`
 - On peut l'activer en appelant `glEnable(GL_DEPTH_TEST);`
 - Et le désactiver en appelant `glDisable(GL_DEPTH_TEST);`
- Tout comme pour le `framebuffer`, il convient d'effacer le `Z-buffer` à chaque fois que la scène est redessinée :
 - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

Cube3D

- Sans Z-buffer
- Avec Z-buffer



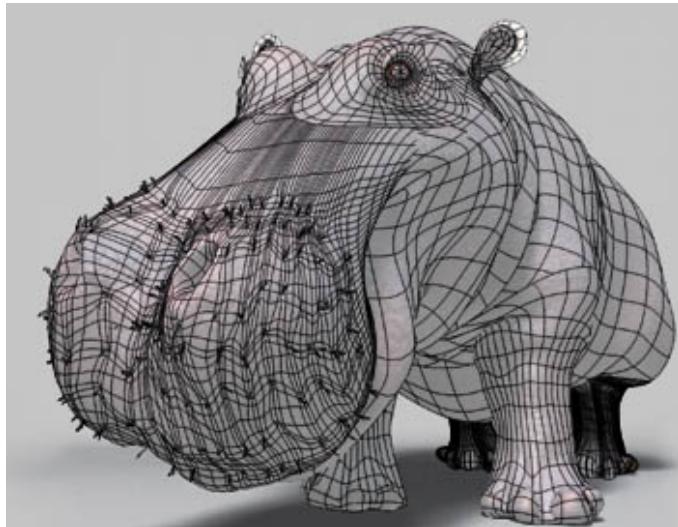
Les textures

- Nous savons afficher et animer un objet 3D ainsi que lui donner des couleurs
- C'est encore assez pauvre !
- Pour augmenter le réalisme des scènes 3D il est important d'avoir recours aux **textures**
- Ce qu'il faut savoir faire :
 - **Charger** une texture
 - « **Plaquer** » une texture sur un objet 3D

Les textures

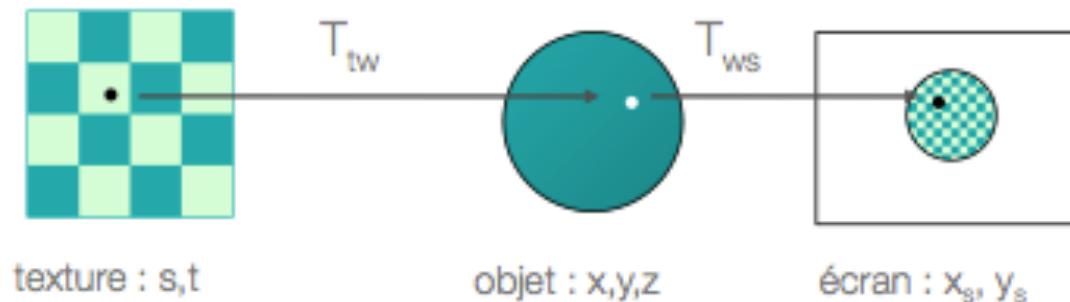
- Qu'est ce qu'une texture ?
 - Ce qui définit l'**apparence** (look) et le « **toucher** » (feel) d'une surface
 - Une **image** qui est utilisée pour définir les caractéristiques d'une surface
 - Une **image multidimensionnelle** qui est plaquée sur un **espace multidimensionnel**

Les textures : exemple



Principe du plaquage de texture

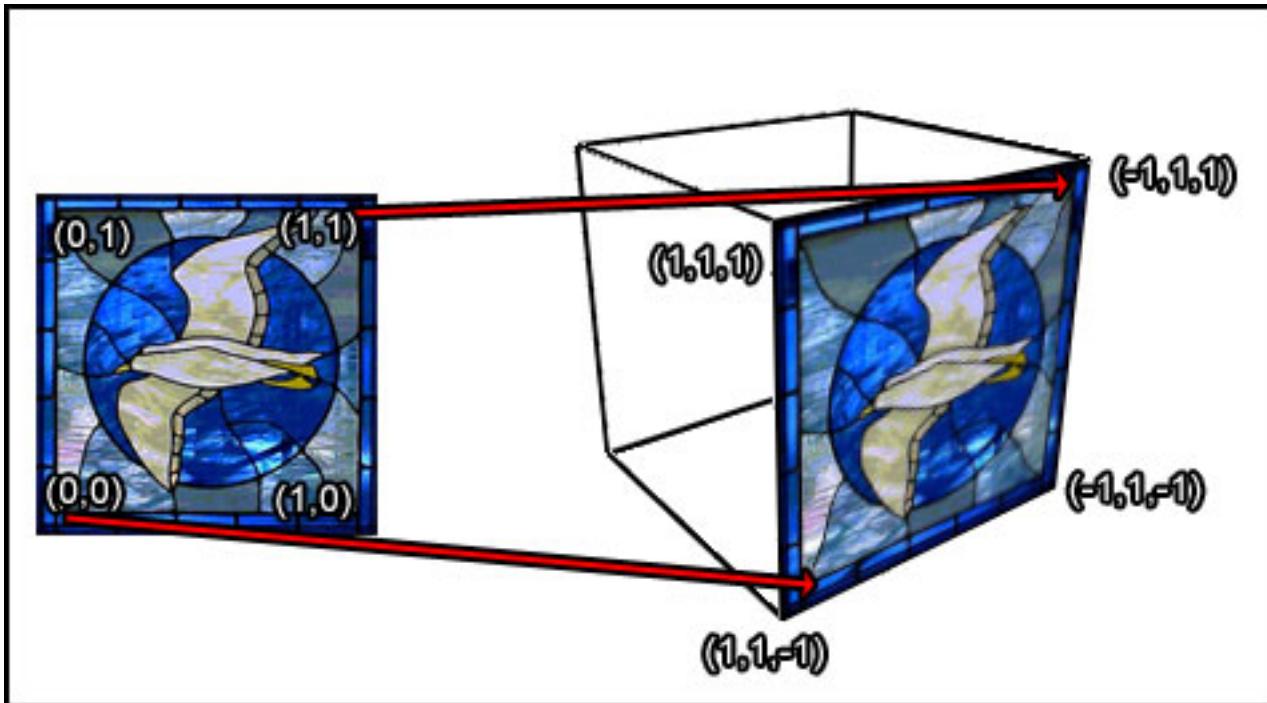
- Un **texel** est un couple (s, t) de coordonnées appartenant à $[0, 1]$ identifiant un point de la texture



- Problème : comment trouver **le(s) texel(s)** correspondant à un **fragment** (x_s, y_s) de l'écran ?
- Solution : **couples vertex/texels et interpolation**

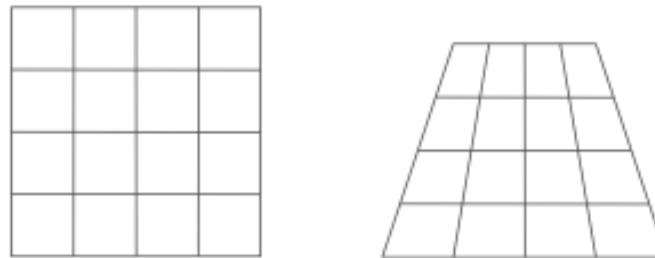
Principe du plaquage de texture

- Associer un **texel** à chaque sommet de notre modèle 3D



Problèmes liés au plaquage de texture

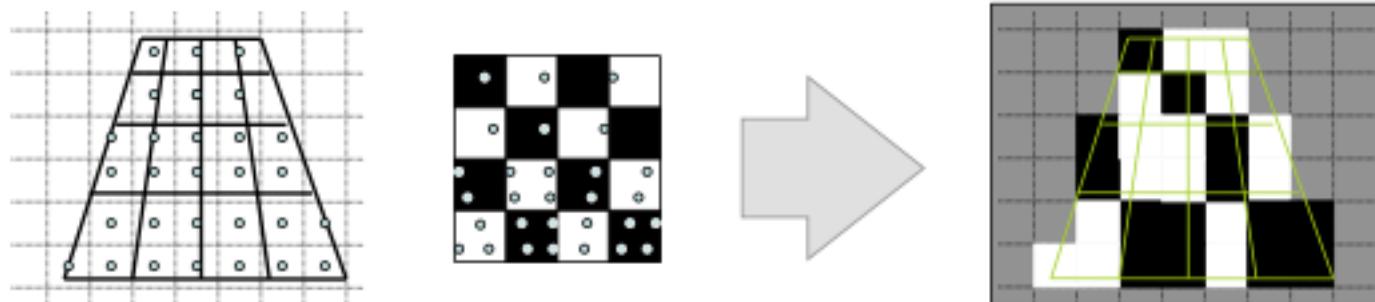
- 1) Prise en compte de la **perspective**
 - Le plaquage de texture est effectué une fois que le polygone a été rastérisé
 - Si on utilise une projection perspective, la taille des segments varie en fonction de la distance à la caméra



- L'interpolation correcte est possible par OpenGL directement (**GL_PERSPECTIVE_CORRECTION_HINT**)
- Le problème n'existe pas en projection parallèle

Problèmes liés au plaquage de texture

- 2) **Aliasing (crénelage en fr).** La correspondance **texel / pixel** n'est pas forcément la bonne



- Deux approches pour résoudre ce problème :
 - **Pré-filtrage** : les textures sont filtrées avant le plaquage (mip-mapping)
 - **Post-filtrage** : plusieurs texels sont utilisés pour le calcul de chaque pixel

Pré-filtrage: Mip-mapping

- Construction d'un ensemble de textures de tailles différentes
- Au moment du plaquage de texture, l'image choisie est telle qu'un fragment correspond à au plus 4 texels



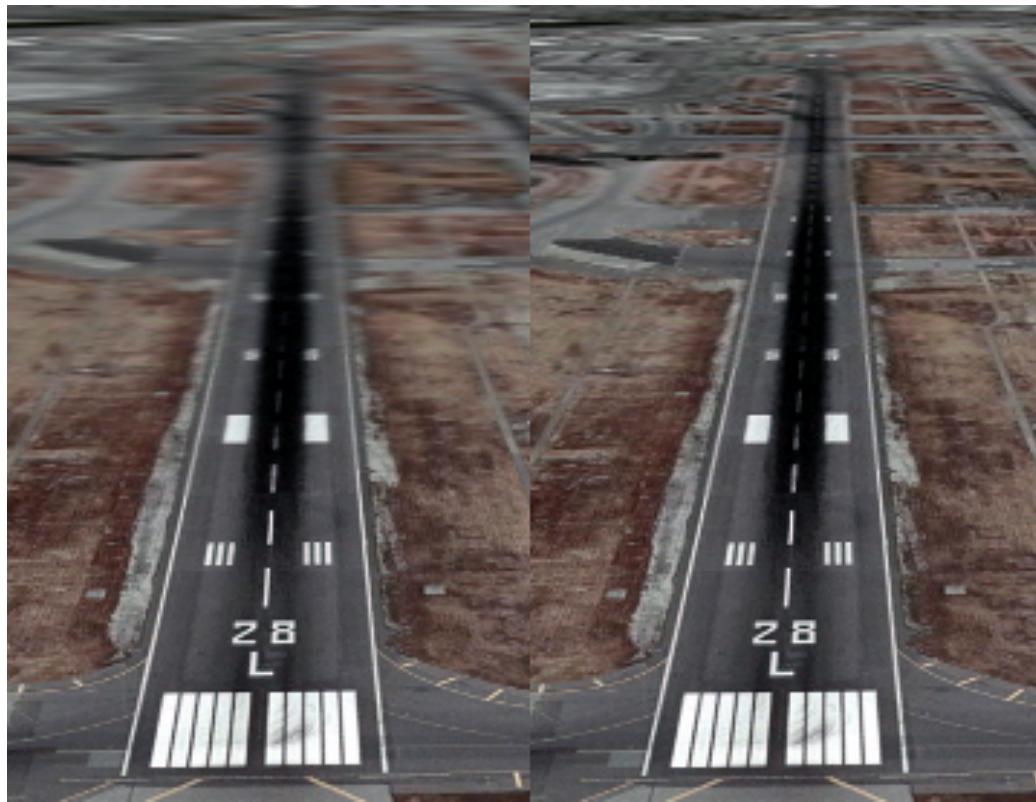
- Inconvénients : l'ensemble des mipmaps occupe 2 fois plus de place que l'image originale
- Intérêt : les images utilisées peuvent être différentes et le rendu est moins **aliasé**

Post-filtrage

- Principe : description de ce qui doit se passer quand l'image est agrandie ou rétrécie
- Deux grandes possibilités :
 - Choisir le **texel le plus proche**
 - Interpoler entre les **n texels** les plus proches ($n = 4 \rightarrow$ **filtrage bilinéaire** ; $n = 8 \rightarrow$ **filtrage trilinéaire**)
- Si on utilise le mip-mapping \rightarrow il faut choisir le « bon » mipmap (ou interpoler entre 2)
- Autre technique : le **filtrage anisotropique**
 - jusqu'à 16 texels pris en compte
 - une « sensibilité » différente entre les axes X et Y spécialement conçue pour prendre en compte les cas problématiques

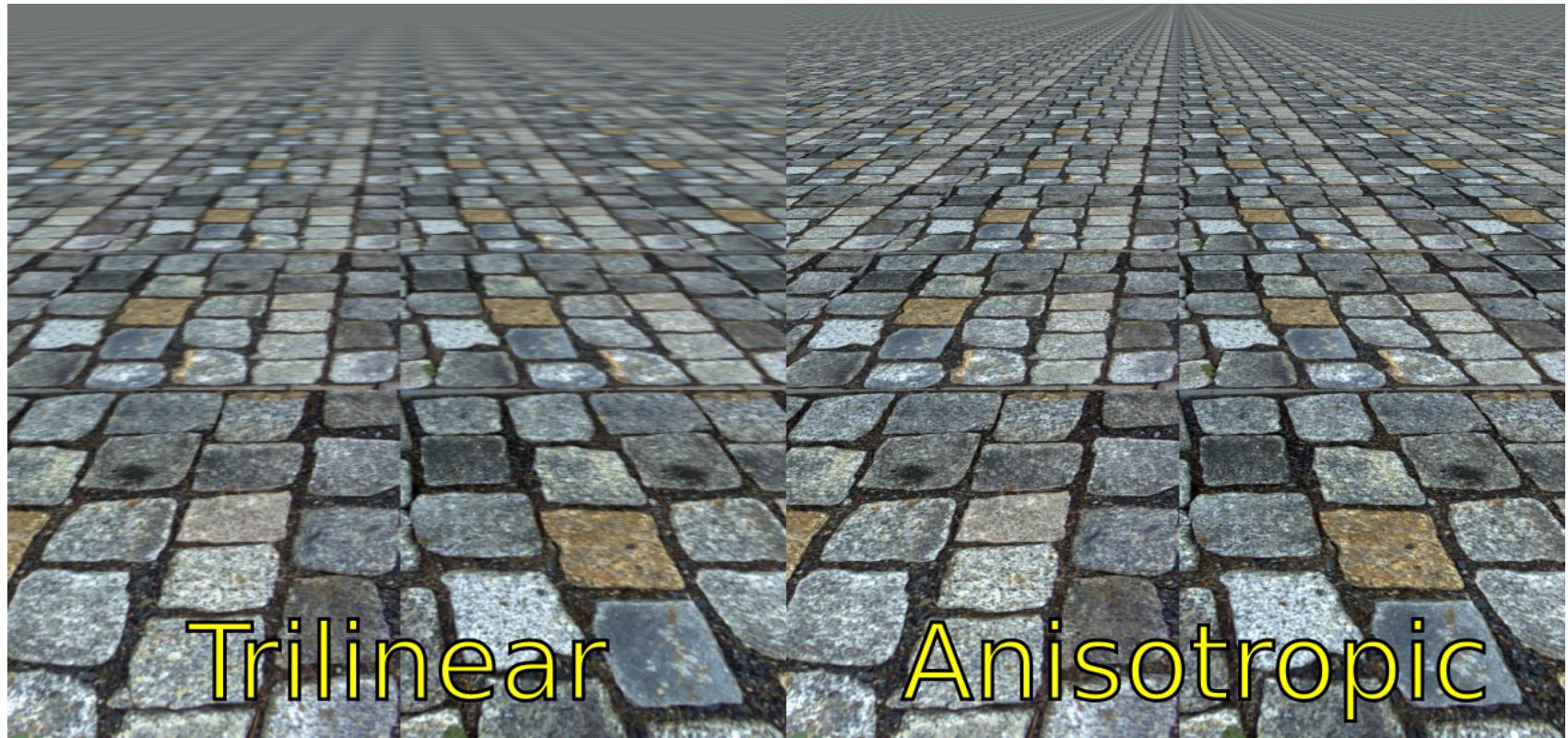
Filtrage Anisotropique

- Quelques résultats (1) :



Filtrage Anisotropique

- Quelques résultats (2) :

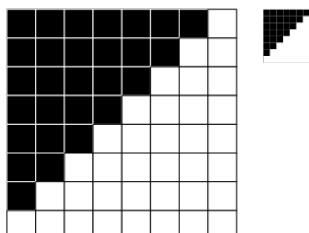
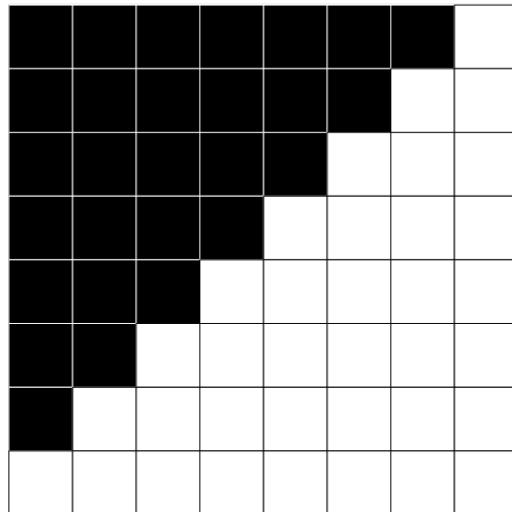


Filtrage Anisotropique

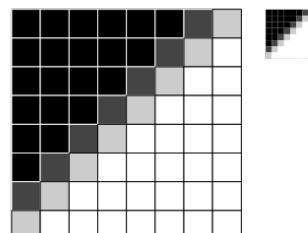
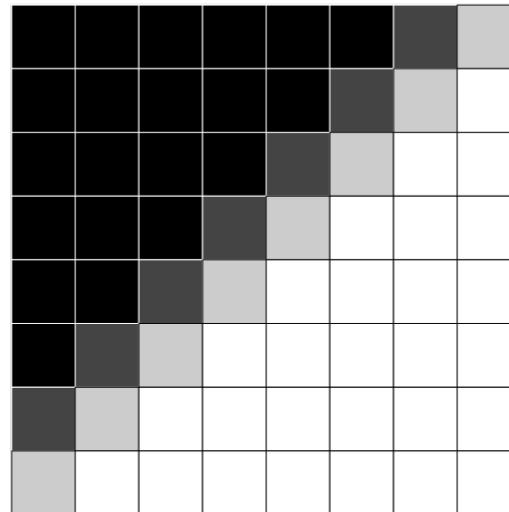


Antialiasing – Anti-crénelage

AA



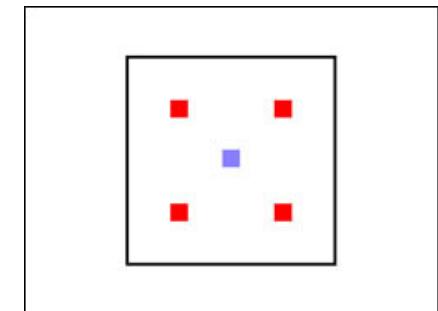
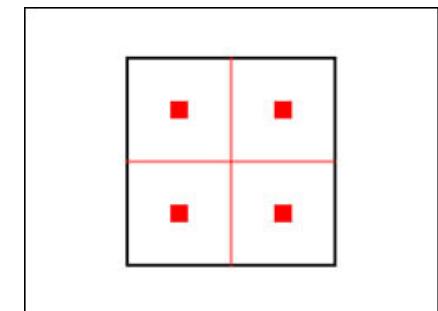
Without Antialiasing



With Antialiasing

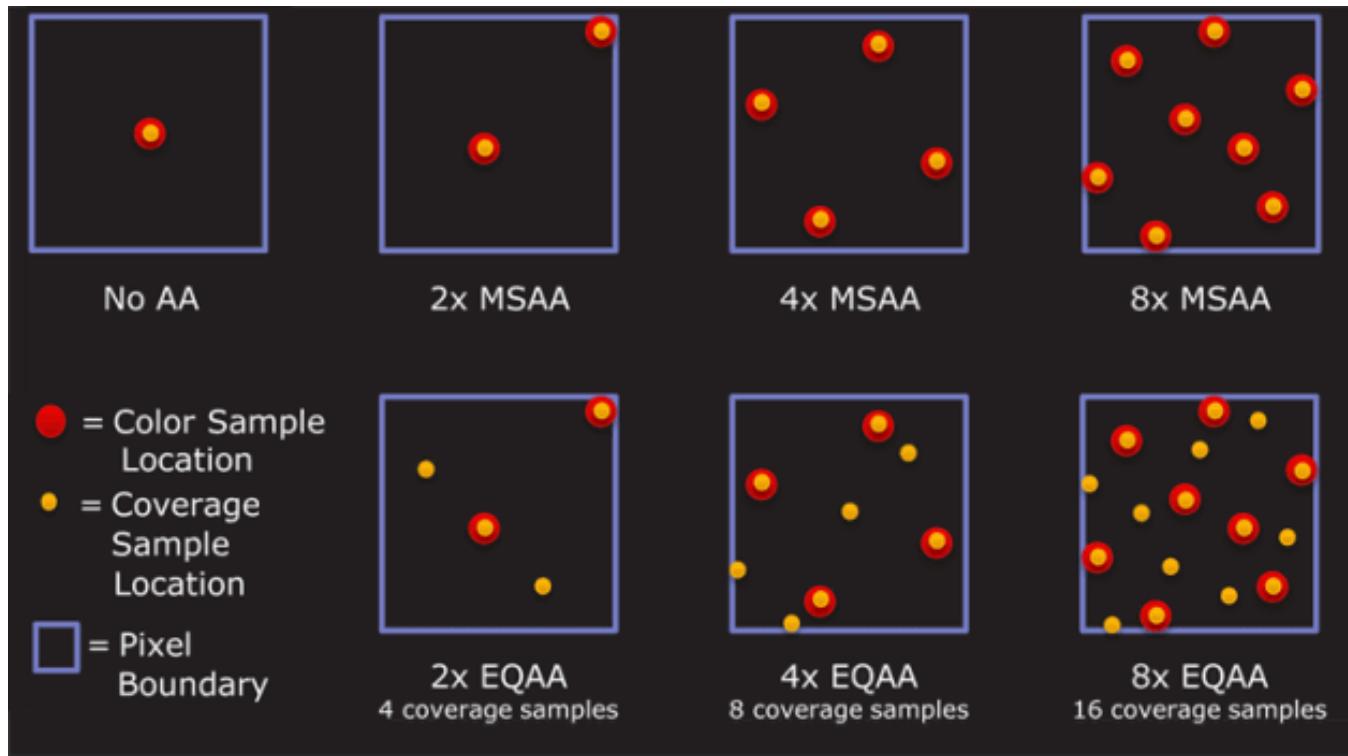
Antialiasing – Anti-crénelage (2)

- Principe : on « fait comme si » on dessinait dans plus de pixels que la résolution réelle de notre écran
 - Ici on va simuler 4 pixels (en rouge) alors qu'on en a qu'un seul (le noir)
 - Pour calculer la couleur finale du pixel : on moyenne la couleur des autres



Antialiasing – Anti-crénelage (3)

- Il existe de multiples stratégies ayant des résultats mais aussi des coûts diverses !



Antialiasing – Anti-crénelage (3)

- Exemple dans Half-Life 2

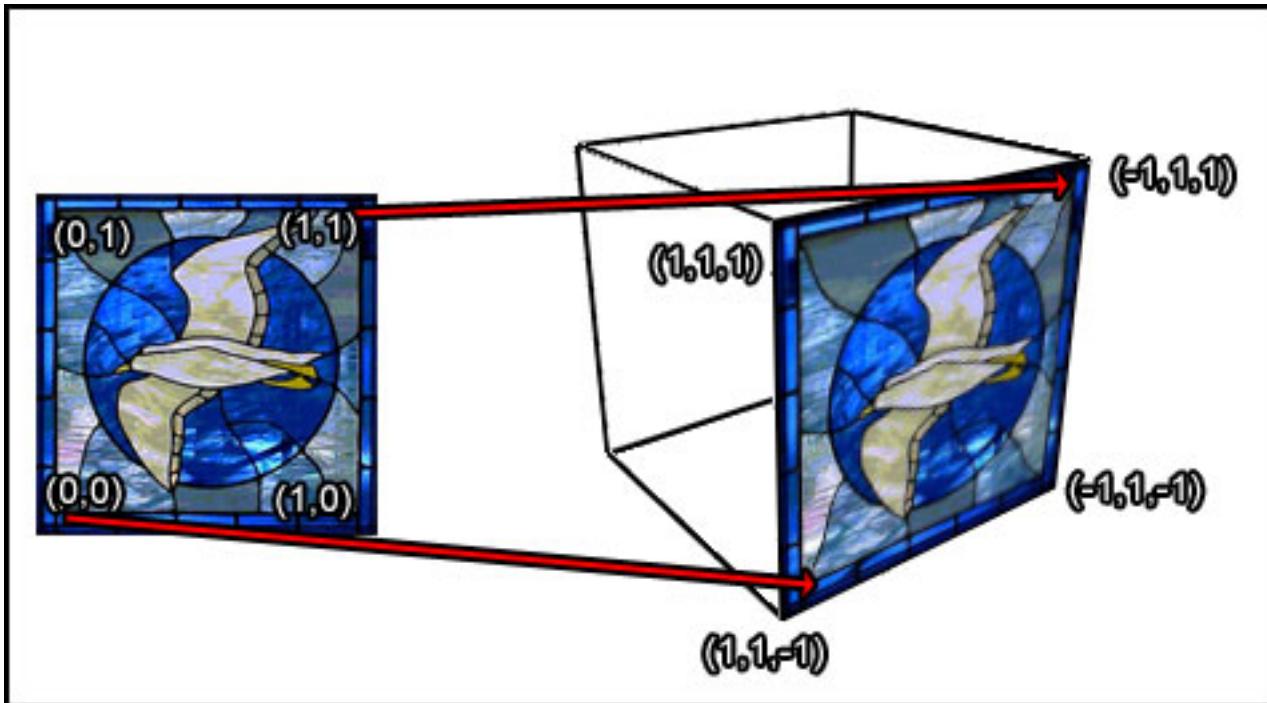


Plaquage de texture 2D avec OpenGL

- Étapes :
 1. Spécifier les données de la texture
 - Charger les données en mémoire ou les générer
 - Associer les données à un **texture object**
 2. Spécifier les paramètres du plaqage
 - 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 (texels) à chaque sommet** (manuellement ou automatiquement)

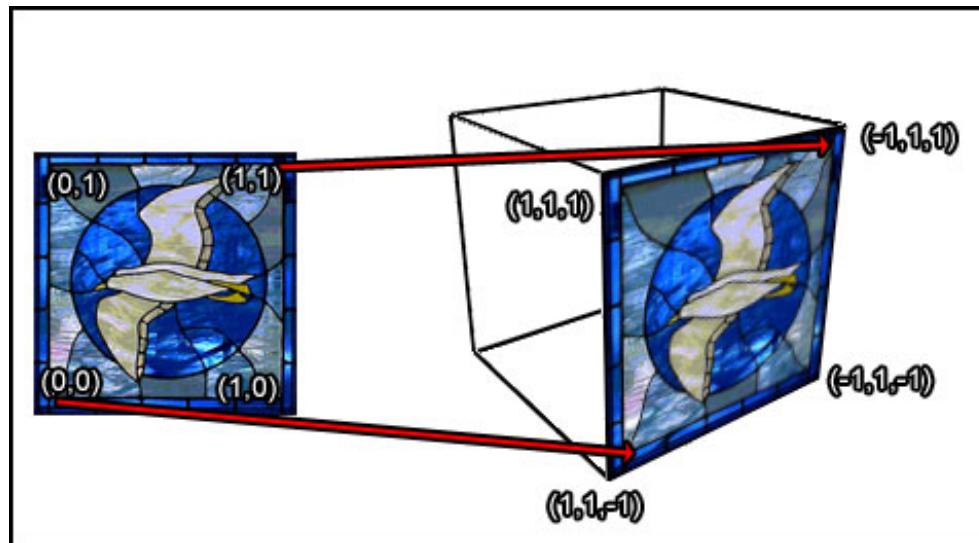
Exemple en OpenGL : cube texturé

- Modifions notre cube multicolore en lui ajoutant une texture :



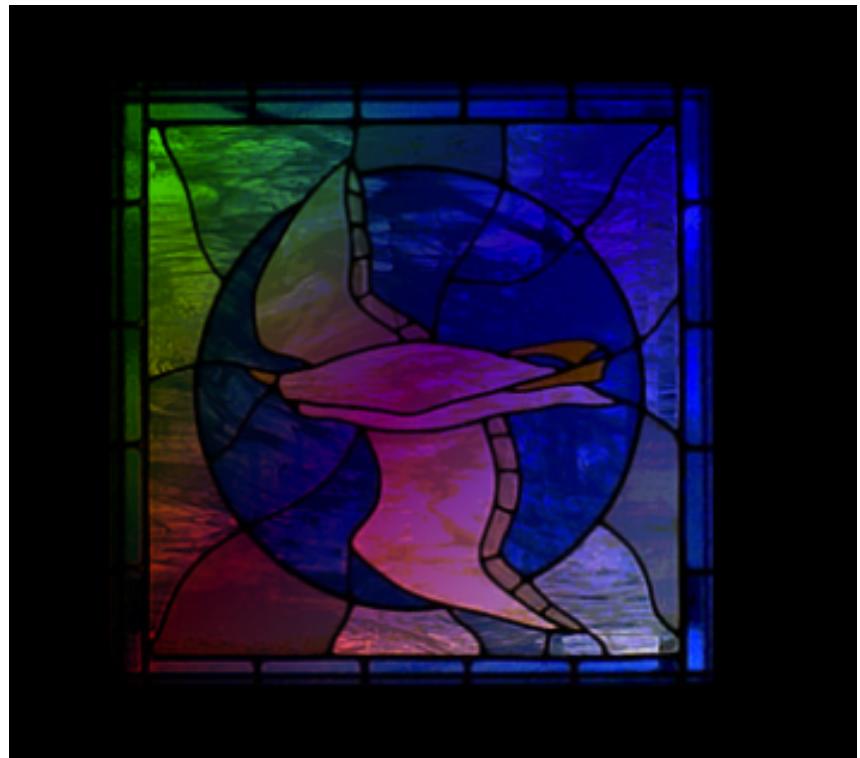
Exemple en OpenGL : cube texturé

- Il faut déterminer les « **bonnes** » coordonnées de texture pour **chaque sommet** de chacune des faces :
 - Un sommet a des coordonnées de textures différentes suivant la face que l'on dessine (comme pour les couleurs)



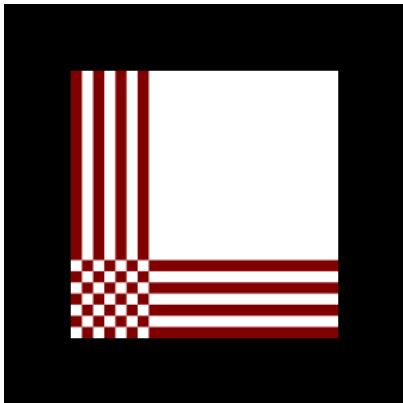
Résultat

- Que se passe t'il ?
- Les couleurs des sommets influent sur la texture !
- Pour garder la couleur « naturelle » de la texture
 - il faut spécifier une couleur neutre pour le sommet (blanc)
 - Ou choisir le mode de combinaison avec le fragment (cf. un peu plus loin)

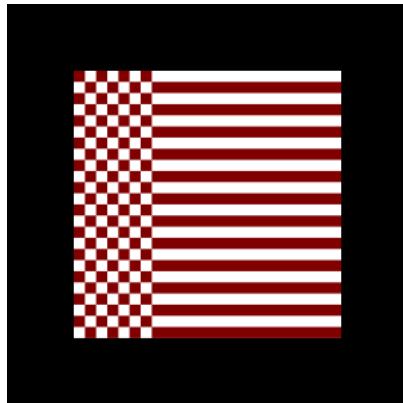


Paramétrage : bords de texture

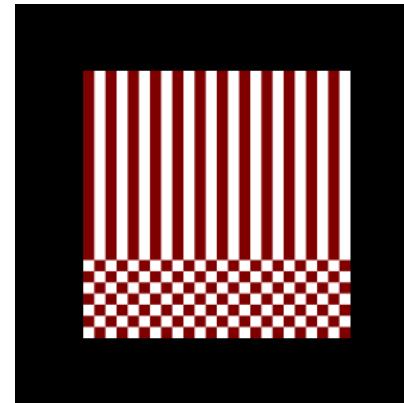
- Pour chaque dimension (s, t) , il est possible de répéter la texture ou de la prolonger avec la dernière valeur
 - `glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_X, YYY)`



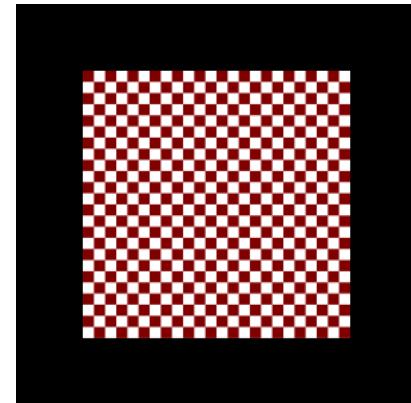
WRAP_S : GL_CLAMP
WRAP_T : GL_CLAMP



WRAP_S : GL_CLAMP
WRAP_T : GL_REPEAT



WRAP_S : GL_REPEAT
WRAP_T : GL_CLAMP



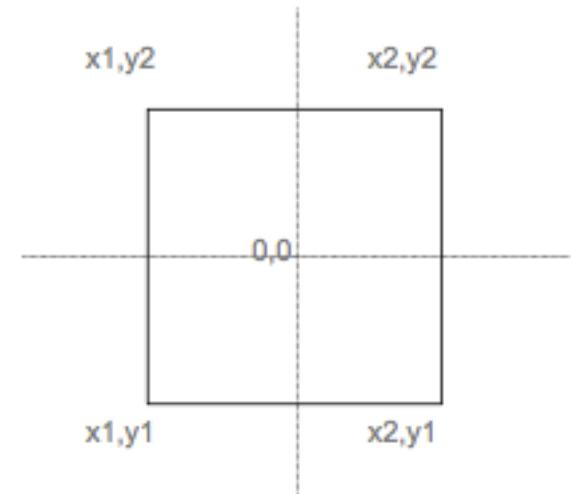
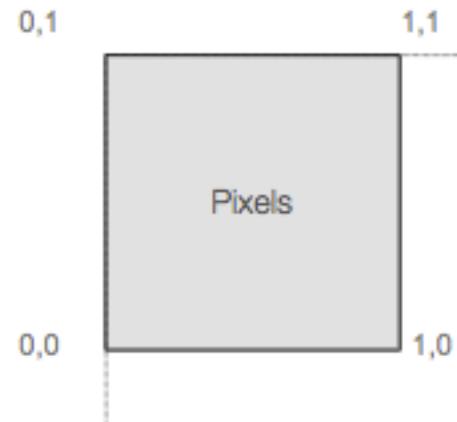
WRAP_S : GL_REPEAT
WRAP_T : GL_REPEAT

Paramétrage : combinaison avec le fragment

- Contrôle 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 la couleur du fragment par celle du texel

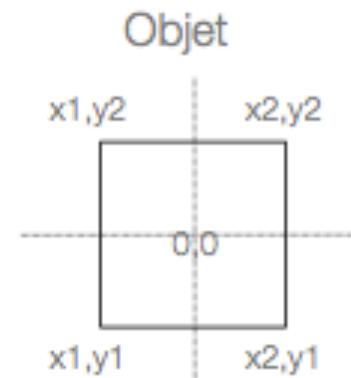
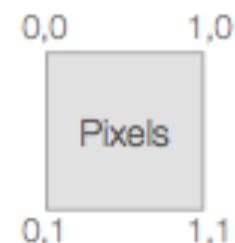
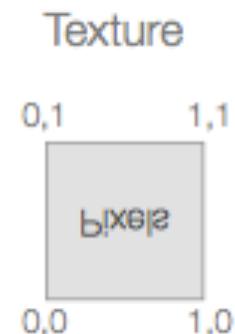
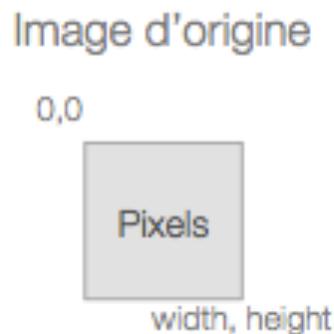
Exemple de spécification des coordonnées de texture

```
glEnable(GL_TEXTURE_2D);  
  
glBegin(GL_QUADS);  
    glTexCoord2f(0,0);  
    glVertex2f(x1,y1);  
    glTexCoord2f(1,0);  
    glVertex2f(x2,y1);  
    glTexCoord(1,1);  
    glVertex2f(x2,y2);  
    glTexCoord(0,1);  
    glVertex2f(x1,y2);  
glEnd();
```



Attention !

- Pour OpenGL le **premier pixel** d'une image est en **bas à gauche** ! Mais en général les données sont organisées de telle sorte que le **premier pixel** est en **haut à gauche**



$x_1, y_1 \rightarrow 0, 1$ et non $0, 0$
 $x_2, y_1 \rightarrow 1, 1$ et non $1, 0$
 $x_2, y_2 \rightarrow 1, 0$ et non $1, 1$
 $x_1, y_2 \rightarrow 0, 0$ et non $0, 1$

Génération automatique des coordonnées

- Utilisation de la fonction
 - `glTexGen*`
- Avec différents modes :
 - `GL_OBJECT_LINEAR` : l'objet sert de référence, les coordonnées sont générées indépendamment de sa position/rotation
 - `GL_EYE_LINEAR` : l'œil sert de référence, les coordonnées sont générées en fonction de la position de la caméra. Si la caméra bouge relativement par rapport à l'objet, les coordonnées de texture seront différentes
 - `GL_SPHERE_MAP` : sert pour l'environnement mapping (pas le temps d'en parler ici)
 - `GL_REFLECTION_MAP` : reflection mapping (idem)

Tutorial Texture de Nate Robins

Screen-space view



Command manipulation window

```
GLfloat border_color[] = { 1.00 , 0.00 , 0.00 , 1.00 };
GLfloat env_color[] = { 0.00 , 1.00 , 0.00 , 1.00 };

glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border_color);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, env_color);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glEnable(GL_TEXTURE_2D);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, w, h, GL_RGB, GL_UNSIGNED_BYTE, image);

	glColor4f( 0.60 , 0.60 , 0.60 , 1.00 );
 glBegin(GL_POLYGON);
 glTexCoord2f( 0.0 , 0.0 ); glVertex3f( -1.0 , -1.0 , 0.0 );
 glTexCoord2f( 1.0 , 0.0 ); glVertex3f( 1.0 , -1.0 , 0.0 );
 glTexCoord2f( 1.0 , 1.0 ); glVertex3f( 1.0 , 1.0 , 0.0 );
 glTexCoord2f( 0.0 , 1.0 ); glVertex3f( -1.0 , 1.0 , 0.0 );
 glEnd();

Click on the arguments and move the mouse to modify values.
```



Couleur

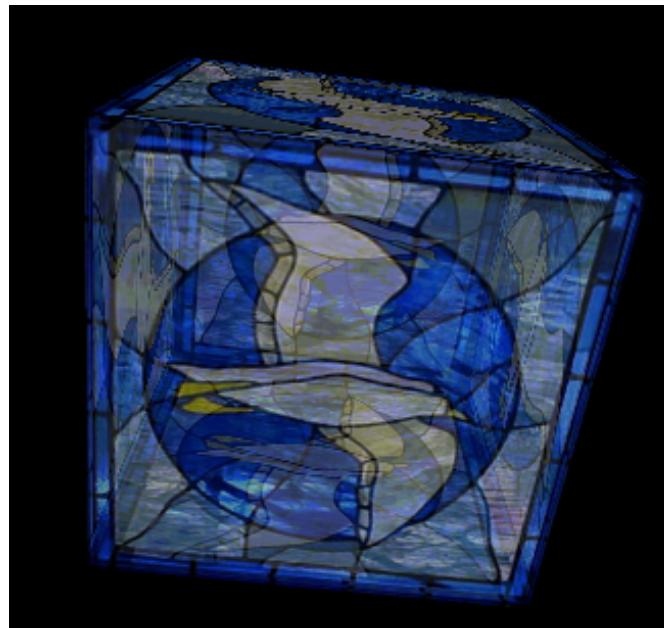
- Classiquement les couleurs sont définies comme un triplet RVB pour **Rouge Vert Bleu**
- Il est également possible d'utiliser une quatrième composante : **A** pour **Alpha**
- Cette composante permet de gérer la transparence des objets
- Si **Alpha = 0** → l'objet est **complètement transparent** et sera donc **invisible**
- Si **Alpha = 1** → l'objet est **complètement opaque** et rien ne pourra être aperçu à travers lui

Transparence ou *Blending*

- Permet de combiner les composantes du fragment avec les valeurs courantes du framebuffer :
 - Rendre un objet transparent
 - Mixer deux images
 - Créer un filtre
- Utilisation
 - `glBlendFunc` définit la fonction de mélange
 - `glEnable(GL_BLEND)` permet d'activer la gestion du *blending*
 - Exemple :
 - `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
 - Destination = $\text{alpha} * \text{source} + (1-\text{alpha}) * \text{destination}$

Blending

- Exemple du cube



- MAIS !! Attention à l'ordre d'affichage de vos objets !!!

Eclairage, Ombrage

- Maintenant que nous savons dessiner une scène 3D (et interagir avec elle)
- Il nous reste à approfondir les notions de :
 - Eclairage
 - Matériaux
 - Ombrage
- Et bien d'autres que nous verrons plus tard si nous avons le temps

Eclairage

- Pour le moment nous n'avons pas parlé de notion de **sources lumineuses** ou **d'illumination** de notre scène 3D
- Lorsque nous **dessinons un point rouge** il nous apparaît rouge quel que soit le point de vue depuis lequel on l'observe
- Dans la réalité un point rouge apparaitra différemment selon son **environnement**, suivant **l'éclairage** et **l'angle** selon lequel on l'observe

Eclairage

- Pour simuler de manière réaliste une scène 3D il faut prendre en compte **l'illumination des objets**
- Problème : **les lois physiques régissant l'éclairement sont complexes et difficiles** (voire impossible) à reproduire ou à simuler en temps réel
- Solution : on choisit un **modèle mathématique simpliste** qui permet de reproduire la **réalité aussi fidèlement que possible** (dans le temps imparti)
- Il convient de mesurer le compromis entre un modèle simple de représentation de la lumière et la rapidité de rendu d'une scène 3D
- Les deux sont généralement inversement proportionnels

Eclairage

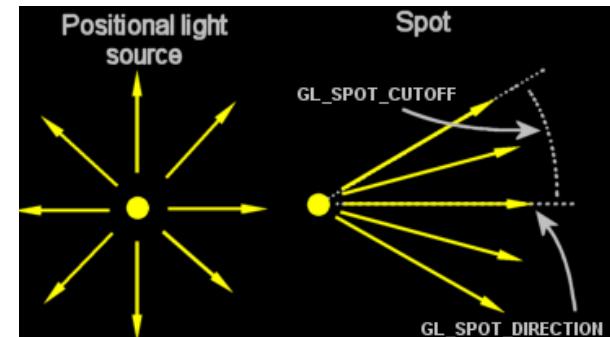
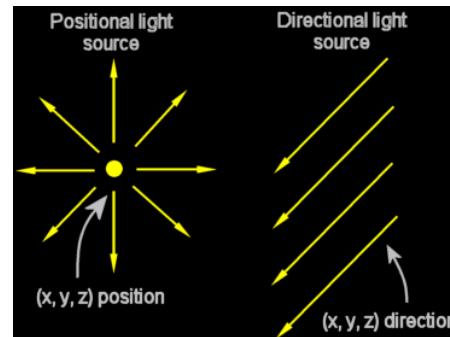
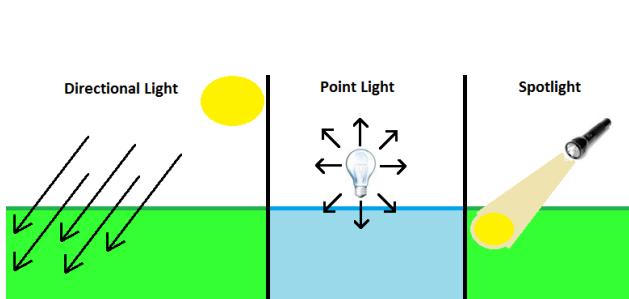
- En OpenGL les sources lumineuses présentes dans la scène introduisent un éclairement différent pour **chaque face** d'un objet 3D en fonction de sa **normale** (vecteur orthogonal à la facette) qui représente **l'orientation** de la facette
- La lumière émise ou réfléchie donne aux objets un aspect brillant ou réfléchissant

Modèle d'illumination d'OpenGL

- Pour étudier la manière dont OpenGL calcule l'illumination d'une scène 3D il faut aborder :
 - Les sources lumineuses : quels sont les paramètres permettant de définir une source de lumière ?
 - Les matériaux : Une brique ne réfléchit pas la lumière de la même manière qu'une plaque d'aluminium. Le matériau est un facteur primordial pour l'éclairage d'une scène
 - L'algorithme de remplissage (*shading model*) : calculer l'éclairage en chaque pixel de l'image est trop coûteux en temps. OpenGL utilise une astuce pour accélérer le rendu

Sources lumineuses

- OpenGL supporte jusqu'à 8 sources lumineuses dans une scène 3D (selon les implémentations d'OpenGL il est possible d'en avoir plus que 8 dans les faits)
- Il existe trois types de sources lumineuses :
 - « **Directionnelle** » (*directional light*) : la source est considérée comme étant à l'infini et la lumière nous parvient selon une direction privilégiée (comme le soleil)
 - « **Ponctuelle** » (*point light*) : la source est positionnée à un endroit de la scène et émet de la lumière dans toutes les directions
 - « **Spot** » (*spotlight*) : la source est positionnée dans la scène et émet de la lumière selon un cône (comme une lampe torche)



Paramètres des sources lumineuses

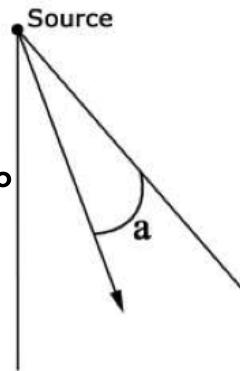
- Une source lumineuse OpenGL est caractérisée par 10 paramètres !
- Trois composantes :
 - Diffuse : réfléchie par un objet dans toutes les directions
 - Spéculaire : lumière réfléchie dans une direction privilégiée (à l'origine de l'effet de brillance)
 - Ambiante : lumière non directionnelle que l'on peut considérer comme issue des multiples réflexions de rayons lumineux
 - OpenGL permet d'affecter une lumière ambiante pour toute la scène
 - La composante ambiante d'une source lumineuse permet d'ajouter une contribution à cette lumière ambiante globale

Paramètres des sources lumineuses

- Possibilité de spécifier une couleur pour chacune des trois composantes
- Il est difficile au début d'appréhender à quoi correspondent les composantes des sources lumineuses
- Le mieux est de jouer avec par soi même !

Paramètres des sources lumineuses

- Position : permet de définir la position de la source lumineuse
- Paramètres de spot :
 - L'angle de coupure : permet de définir le type de source que l'on souhaite utiliser (compris entre 0° et $180^\circ \rightarrow$ omnidirectionnelle)
 - Direction du spot (si non omnidirectionnelle)
 - Exposant : fait varier la concentration de la lumière à l'intérieur du cône définissant le spot. Si exposant = 0 la lumière est répartie également dans toutes les directions. Plus exposant ↗ plus la lumière est concentrée autour de la direction définie ci-dessus.

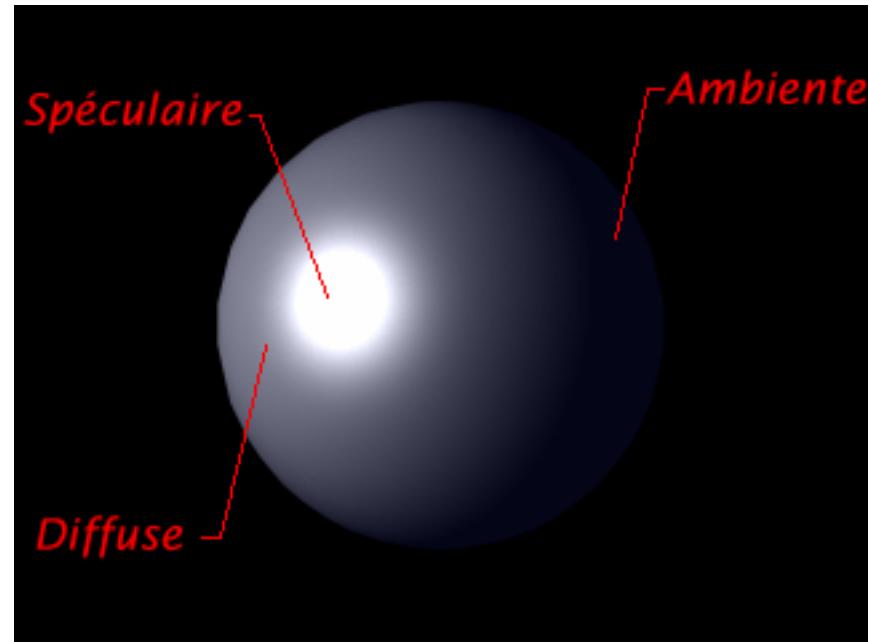


Paramètres des sources lumineuses

- Paramètres d'atténuation : plus un objet est éloigné de la source lumineuse, moins il est éclairé par cette dernière. Il y a trois paramètres d'atténuation :
 - Le facteur d'atténuation constante (A_c)
 - Le facteur d'atténuation linéaire (A_l)
 - La facteur d'atténuation quadratique (A_q)
- Pour un point P situé à une distance d , l'intensité lumineuse est divisée par :
 - $A_c + A_l * d + A_q * d^2$, par défaut $A_c = 1$, $A_l = A_q = 0 \rightarrow$ atténuation nulle (division par 1)

Matériaux

- La spécification d'un matériau pour un objet se fait via cinq paramètres :
 - La couleur diffuse
 - La couleur spéculaire
 - La couleur ambiante
 - La couleur émise
 - Le coefficient de brillance



Source de l'image :

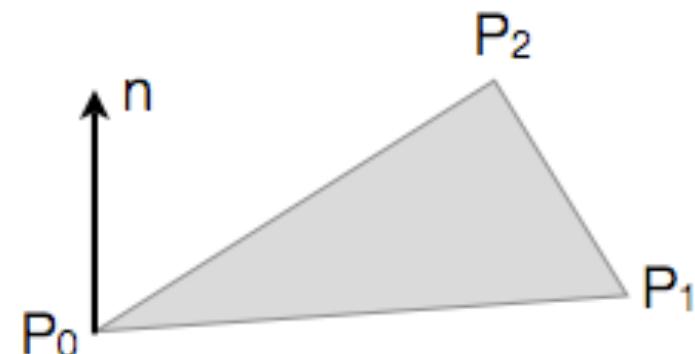
<http://www-evasion.imag.fr/Membres/Antoine.Bouthors/teaching/opengl/opengl4.html>

L'algorithme de remplissage

- Pour des raisons d'efficacité, OpenGL ne calcule pas la couleur de chaque pixels d'un polygone, soit :
 - Il remplit chaque polygone avec une couleur unie (mode de remplissage plat ou « *flat* »)
 - Il utilise un algorithme de Gouraud (mode « *smooth* ») : la couleur de chaque sommet d'un polygone est calculée, le polygone est rempli par interpolation entre ces différentes couleurs
- Pour calculer la réflexion des rayons lumineux en un point, OpenGL a besoin de connaître la perpendiculaire à la surface en ce point : la normale (ou vecteur normal)

Normales

- Les algorithmes d'éclairage ont besoin de connaître l'orientation des faces d'un objet
- Cela est représenté par la notion de normale : vecteur perpendiculaire à la face considérée
- $n = (P_2 - P_0) \times (P_1 - P_0)$
- Normalisation : $n = n / \|n\|$
- \times : produit vectoriel
- $\|n\|$: norme de n



Normales

- Les normales OpenGL peuvent être définies au niveau :
 - des faces
 - des sommets
- `glNormal3f(float nx, float ny, float nz);`
- Utilisé de la même manière que `glColor3f` par exemple
- Permet de modifier le mode de remplissage des polygones (cf. exemple plus loin)

Exemple d'éclairage en OpenGL

- Testons avec une théière (objet « mythique » en informatique graphique de l'université de l'Utah)
- Fonctions essentielles en OpenGL :
 - `glEnable(GL_LIGHTING)` → activation de la lumière
 - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, col)` → définition de la couleur `col` de la lumière ambiante globale
 - `glEnable(GL_LIGHTi)` avec $0 \leq i \leq 7$ → active la i^e lumière d'OpenGL

Exemple d'éclairage en OpenGL

- Modification des paramètres des lumières
- `glLightfv(GL_LIGHTi, param, value)`
 - `GL_LIGHTi` : i^e lumière d'OpenGL
 - Param :
 - `GL_POSITION` : position de la lumière
 - `GL_AMBIENT` : paramètre ambiant de la lumière
 - `GL_DIFFUSE` : paramètre diffus de la lumière
 - `GL_SPECULAR` : paramètre spéculaire de la lumière
 - `GL_SPOT_DIRECTION` : direction du spot
 - `GL_SPOT_EXPONENT` : exposant du spot
 - `GL_SPOT_CUTOFF` : coupure du spot
 - `GL_CONSTANT_ATTENUATION` : atténuation constante
 - `GL_LINEAR_ATTENUATION` : atténuation linéaire
 - `GL_QUADRATIC_ATTENUATION` : atténuation linéaire
 - Value : nouvelle valeur du paramètre

Exemple d'éclairage en OpenGL

- Changement des paramètres de matériau d'un objet
- `glMaterialfv(face, param, value)`
- Face : côté des faces auquel appliquer le matériau
 - `GL_FRONT` : face « avant »
 - `GL_BACK` : face « arrière »
 - `GL_FRONT_AND_BACK` : faces « avant » et « arrière »
- Param : paramètre à modifier
 - `GL_AMBIENT` : couleur ambiante du matériau
 - `GL_DIFFUSE` : couleur diffuse du matériau
 - `GL_AMBIENT_AND_DIFFUSE` : permet de spécifier en même temps les couleurs ambiante et diffuses
 - `GL_SPECULAR` : couleur spéculaire du matériau
 - `GL_EMISSION` : couleur émise par le matériau
 - `GL_SHININESS` : exposant spéculaire du matériau (valeur réelle entre 0 et 128)
- Value : nouvelle valeur du paramètre

Exemple d'éclairage en OpenGL

```
// Lumiere  
GLfloat lightpos[] = { 0.0f, 0.0f, 15.0f };  
GLfloat lightcolor[] = { 1.0f, 1.0f, 0.0f };  
GLfloat ambcolor[] = { 0.0f, 0.0f, 1.0f };  
  
glEnable(GL_LIGHTING);  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambcolor);  
glEnable(GL_LIGHT0);  
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);  
glLightfv(GL_LIGHT0,GL_AMBIENT,lightcolor);  
glLightfv(GL_LIGHT0,GL_DIFFUSE,lightcolor);
```



Exemple d'éclairage en OpenGL

```
void setMaterial (
GLfloat ambientR, GLfloat ambientG, GLfloat ambientB,
GLfloat diffuseR, GLfloat diffuseG, GLfloat diffuseB,
GLfloat specularR, GLfloat specularG, GLfloat specularB,
GLfloat shininess ) {

GLfloat ambient[] = { ambientR, ambientG, ambientB };
GLfloat diffuse[] = { diffuseR, diffuseG, diffuseB };
GLfloat specular[] = { specularR, specularG, specularB };

glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,ambient);
glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,diffuse);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,specular);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,shininess);
}
```



Exemple d'éclairage en OpenGL

```
// Affichage  
setMaterial(0.0f,0.5f,1.0f,0.0f,0.5f,1.0f,1.0f,1.0f,1.0f,1.0f);  
glutSolidTeapot(0.5f);
```



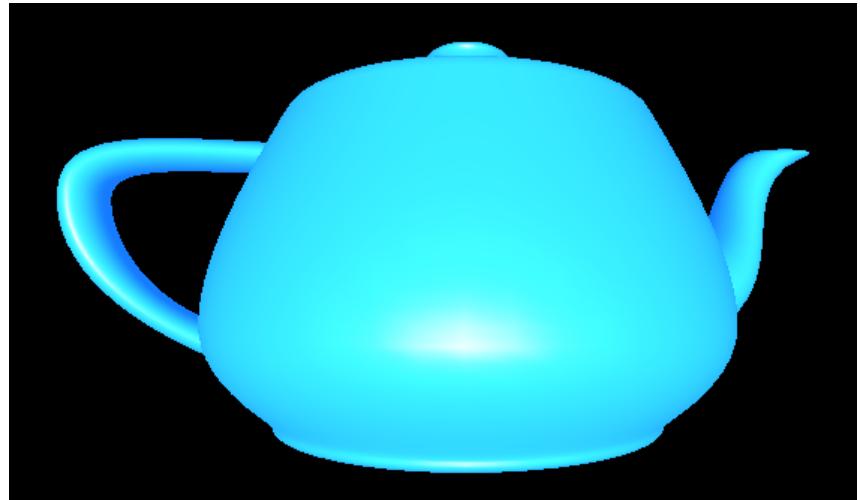
Exemple d'éclairage en OpenGL

- Changer le mode de remplissage OpenGL (**shading model**) : très simple !
- **glShadingModel(value);**
- Value :
 - **GL_FLAT** : une couleur par polygone
 - **GL_SMOOTH** : interpolation des couleurs des sommets des polygones



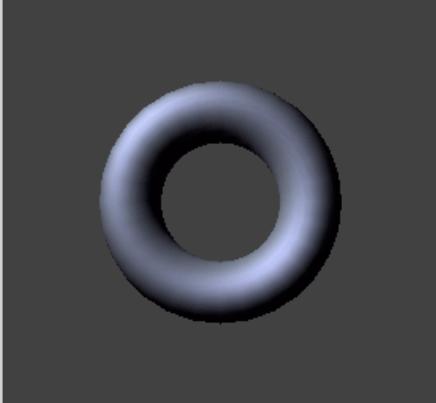
Exemple d'éclairage en OpenGL

- Changer le mode de remplissage OpenGL (**shading model**) : très simple !
- **glShadingModel(value);**
- Value :
 - **GL_FLAT** : une couleur par polygone
 - **GL_SMOOTH** : interpolation des couleurs des sommets des polygones



Tutoriel *Light Material* de Nate Robins

Screen-space view



Command manipulation window

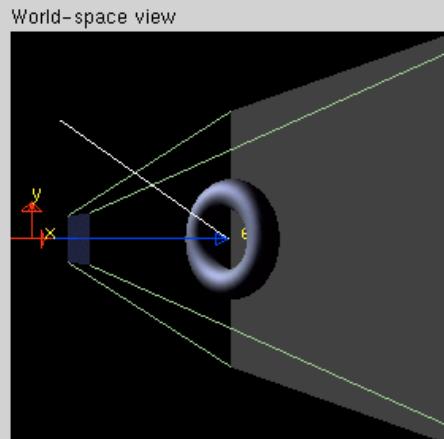
```
GLfloat light_pos[] = { -2.00, 2.00, 2.00, 1.00 };
GLfloat light_Ka[] = { 0.00, 0.00, 0.00, 1.00 };
GLfloat light_Kd[] = { 1.00, 1.00, 1.00, 1.00 };
GLfloat light_Ks[] = { 1.00, 1.00, 1.00, 1.00 };

glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_Ka);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_Kd);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_Ks);

GLfloat material_Ka[] = { 0.11, 0.06, 0.11, 1.00 };
GLfloat material_Kd[] = { 0.43, 0.47, 0.54, 1.00 };
GLfloat material_Ks[] = { 0.33, 0.33, 0.52, 1.00 };
GLfloat material_Ke[] = { 0.00, 0.00, 0.00, 0.00 };
GLfloat material_Se = 10;

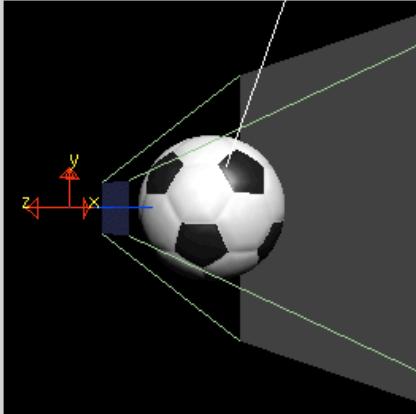
glMaterialfv(GL_FRONT, GL_AMBIENT, material_Ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, material_Kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, material_Ks);
glMaterialfv(GL_FRONT, GL_EMISSION, material_Ke);
glMaterialfv(GL_FRONT, GL_SHININESS, material_Se);
```

Click on the arguments and move the mouse to modify values.



Tutoriel *Light Position* de Nate Robins

World-space view



A 3D rendering of a soccer ball in a world-space coordinate system. A blue cube represents the light source, and green lines show the light rays hitting the ball. A red, green, and blue coordinate system is shown at the bottom left.

Screen-space view



A 2D rendering of the soccer ball on a screen, showing the lighting effects from the light source.

Command manipulation window

```
GLfloat pos[4] = { 1.50 , 1.00 , 1.00 , 0.00 };

gluLookAt( 0.00 , 0.00 , 2.00 ,   <- eye
            0.00 , 0.00 , 0.00 ,   <- center
            0.00 , 1.00 , 0.00 ); <- up

glLightfv(GL_LIGHT0, GL_POSITION, pos);
```

Click on the arguments and move the mouse to modify values.

Ce qui manque encore ...

- Ombres
- Plusieurs objets transparents et leurs interactions
- Illumination globale :
 - Réflexion d'un objet dans un autre
 - Diffusion indirecte
- Détails d'une surface (p. ex. peau d'orange)
- Miroirs
- Brouillard
- Objets avec des « trous »

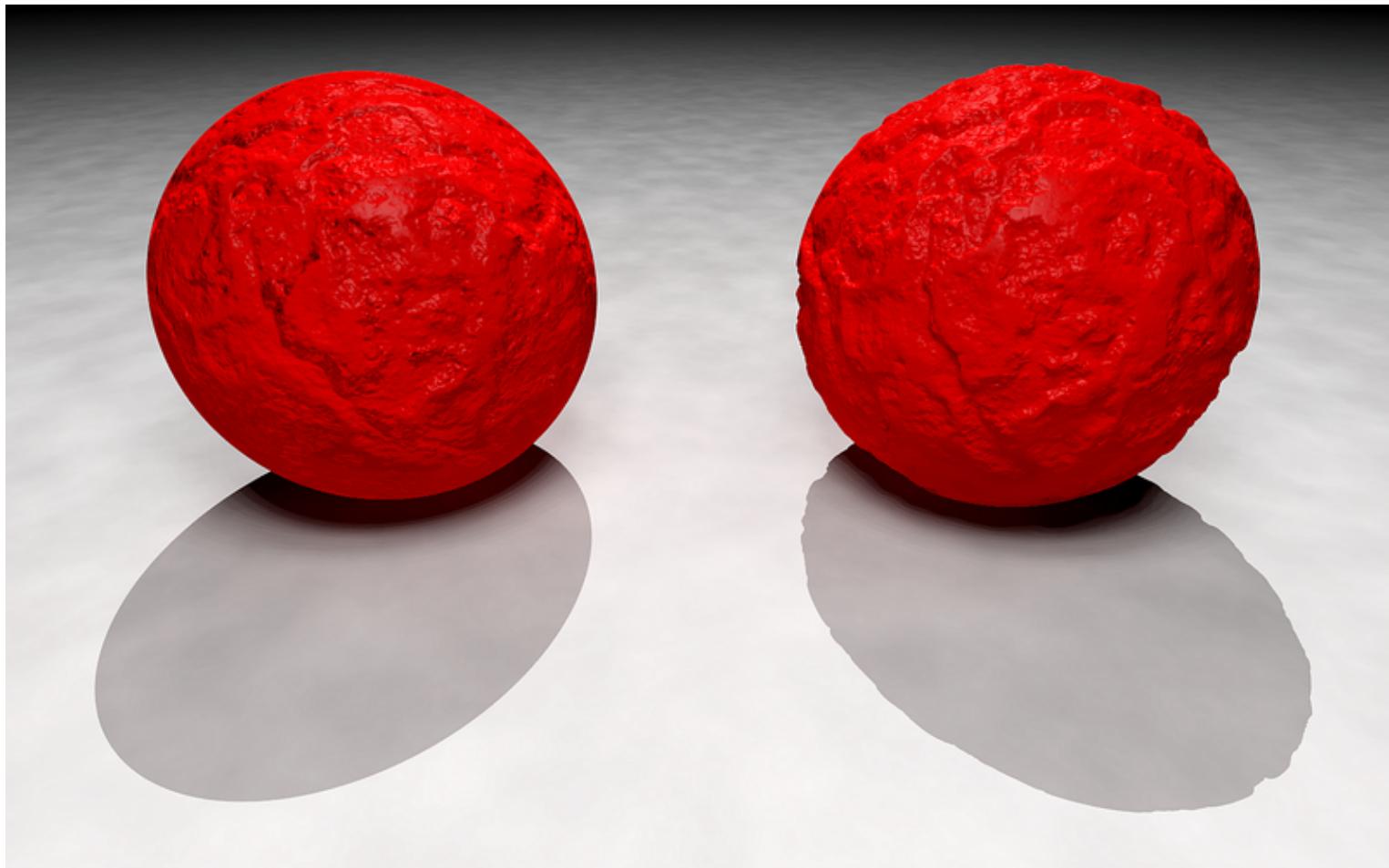
Mais ...

- Nous n'aurons pas le temps de tout voir !
- Je vais parler très rapidement de **bump mapping**
- Puis passer à quelque chose de plus récent : l'utilisation des nouvelles fonctionnalités d'OpenGL (version ≥ 3)
- **Les Shaders !**

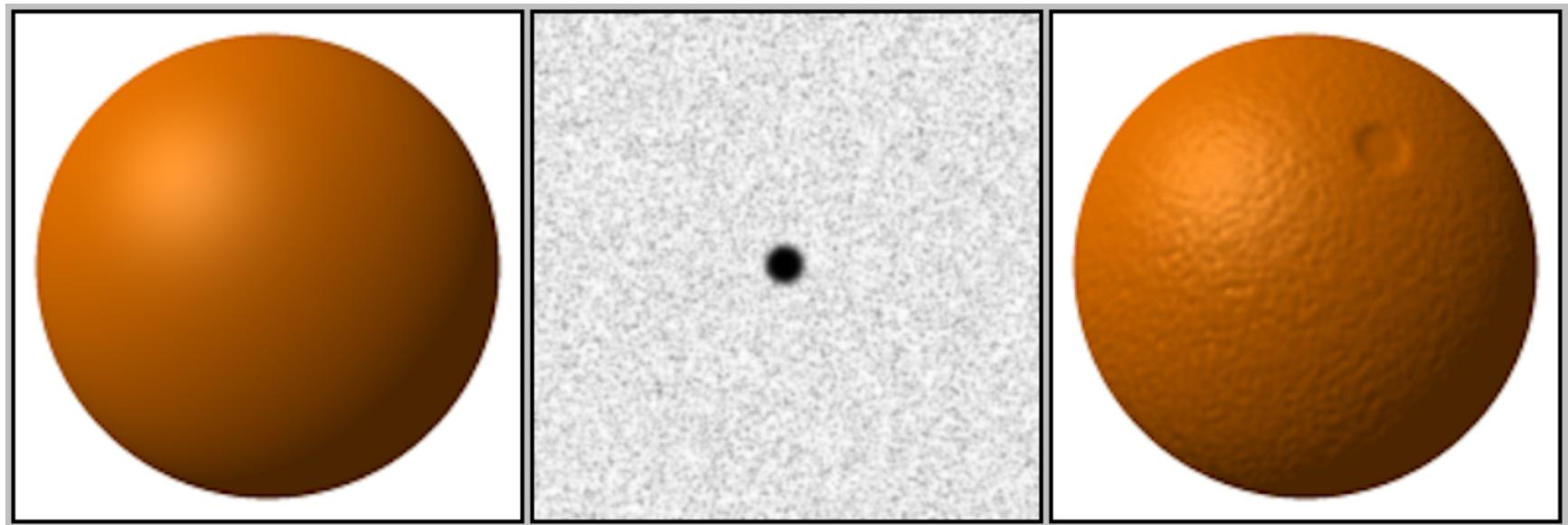
Bump Mapping

- Comment donner du relief à une surface ?
 - Physiquement : changer la **structure physique de l'objet** → ajouter des polygones pour créer des déformations
 - Virtuellement : perturber les vecteurs normaux des polygones avant le shading : le **bump mapping** !
- Inconvénients du bump mapping : la silhouette n'est pas modifiée !
 - Visible au niveau du contour de l'objet
 - Pas d'ombres liées au relief

Bump Mapping



Exemple Bump Mapping



<http://en.wikipedia.org/wiki/File:Bump-map-demo-full.png>

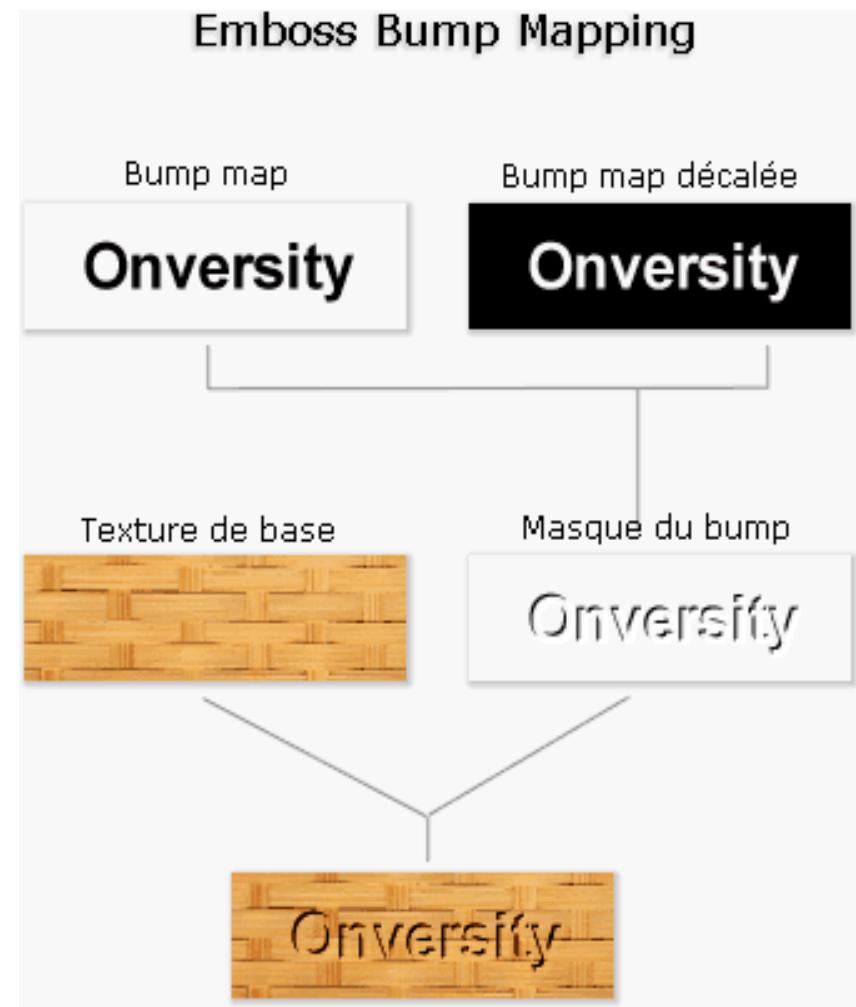
Bump Mapping : principe

- Plusieurs techniques :
 - True Bump Mapping [Blinn76]
 - Emboss Bump Mapping
 - DotProduct3
 - Environment Bump Mapping
- True Bump Mapping
 - Principe →



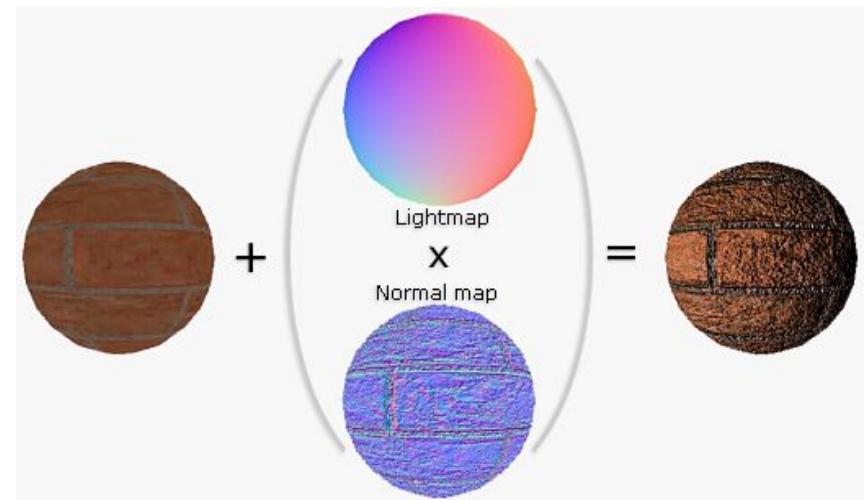
Emboss Bump Mapping (EBM)

- Nécessite 2 textures : la **texture de base** et la « **bump map** »
 1. À partir du bump map → **création d'un bump map « décalé »** qui contient l'effet d'ombre du bump map. **On décale le bump map selon la direction d'éclairage**
 2. On **superpose le bump map au « bump map décalé »** pour obtenir le « **masque de bump mapping** »
 3. On **superpose la texture de base au « masque du bump mapping »** pour obtenir le **résultat final**



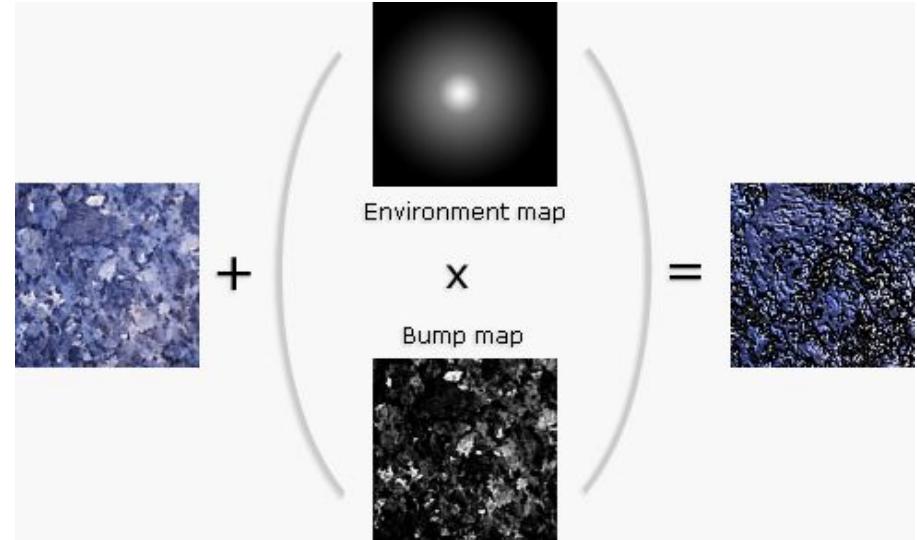
DotProduct3 Bump Mapping

- 3 textures :
 - **Light Map** : information d'éclairage pour chaque point
 - **Normal Map** : information de relief pour chaque point (normale)
 - **Texture de base** : information colorimétriques
- Principe : multiplication des vecteurs de la **light map** avec ceux de la **normal map** point par point et combinaison avec la texture de base



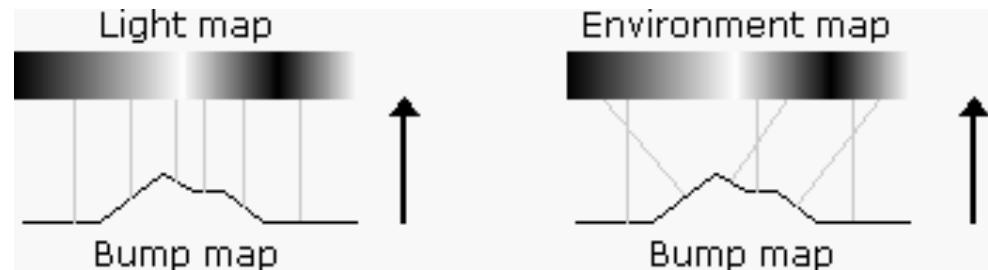
Environment Mapped Bump Mapping (EMBM)

- 3 textures :
 - Texture de base
 - Environment Map : représentation d'un éclairage de la scène
 - Bump map : représentation de la perturbation des normales
- Principe : combinaison des trois textures mais différemment du dot3!



Environment Mapped Bump Mapping (EMBM)

- EMBM :
 - Les points de la **Bump Map** et de l'**Environment Map** ne sont pas en correspondance directe !
1. On lit les informations de la **bump map**
 2. Formule mathématique qui va déterminer le point de **l'Environment Map** correspondant
 3. L'information de **l'Environment Map** détermine l'intensité lumineuse à la texture de base
 4. Go to 1 jusqu'à ce que la bump map soit traitée!



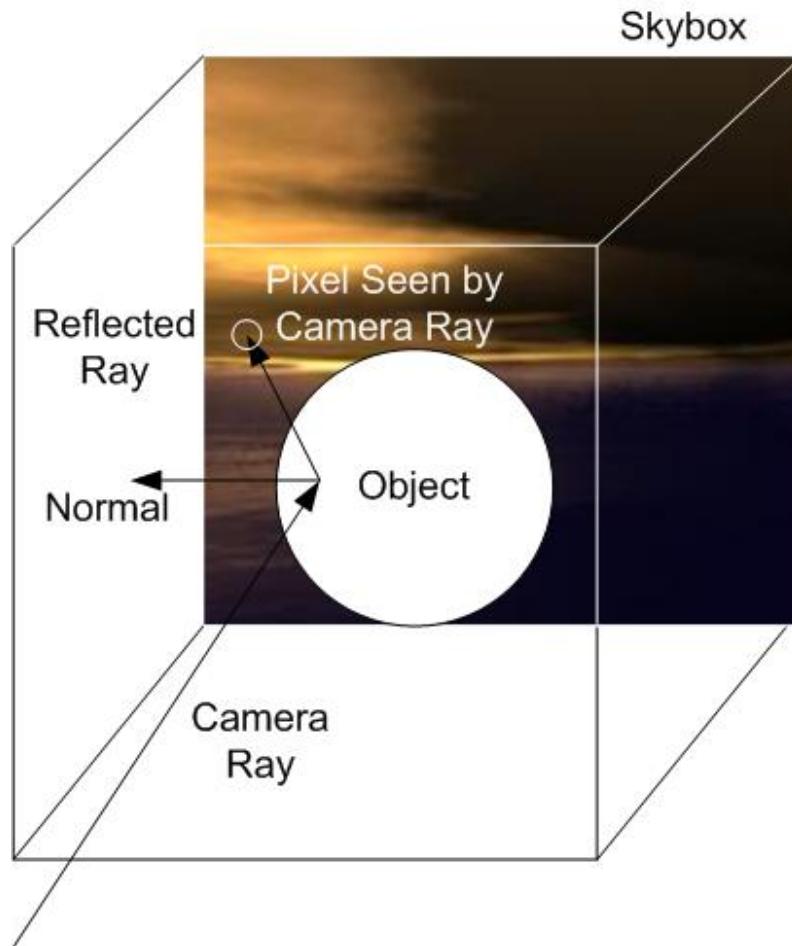
Environment Mapping

- Technique utilisée pour le rendu d'objets réfléchissants (p. ex. chromé)



- Idée : refléter sur l'objet l'environnement

Environment Mapping : principe



Source : http://en.wikipedia.org/wiki/File:Cube_mapped_reflection_example.jpg 171

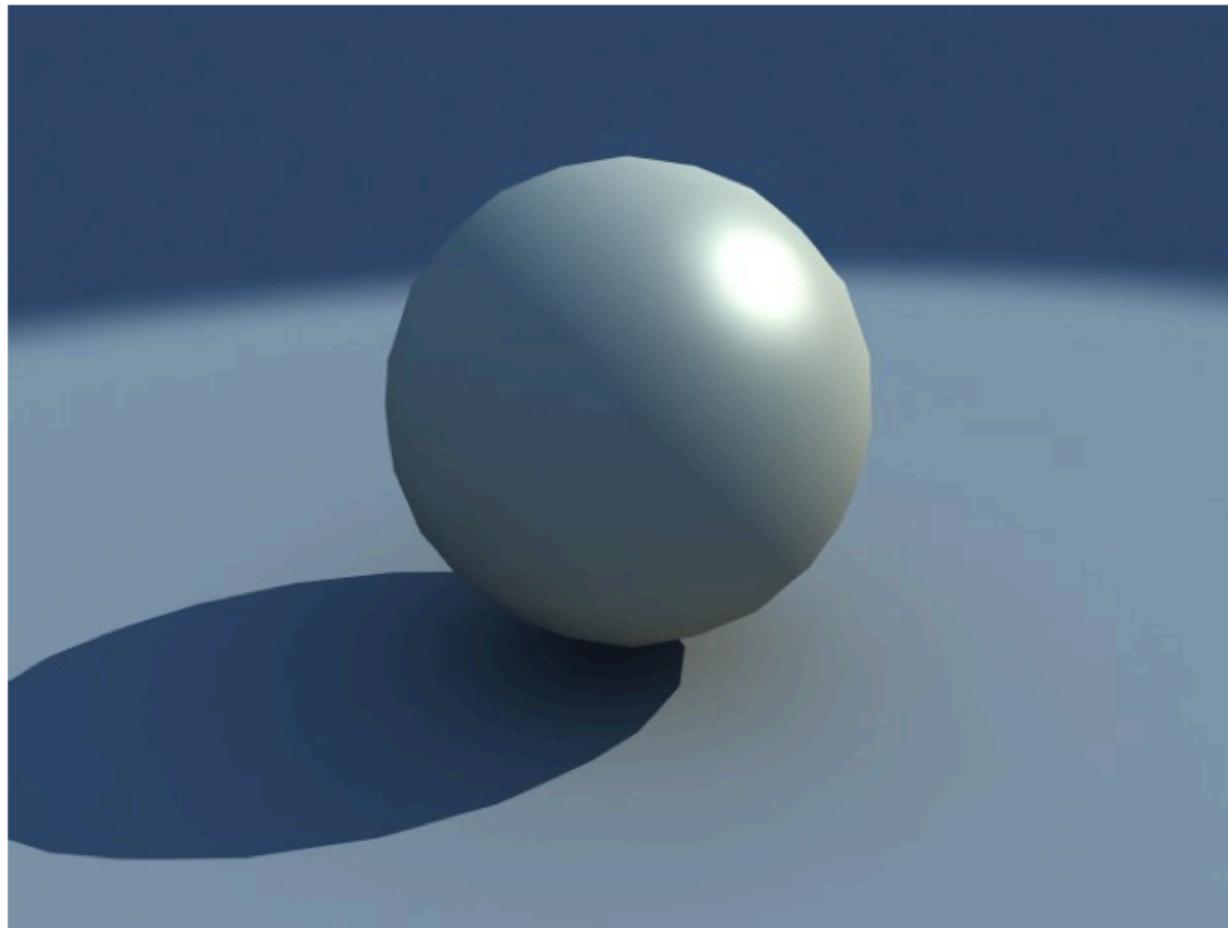
Résultats



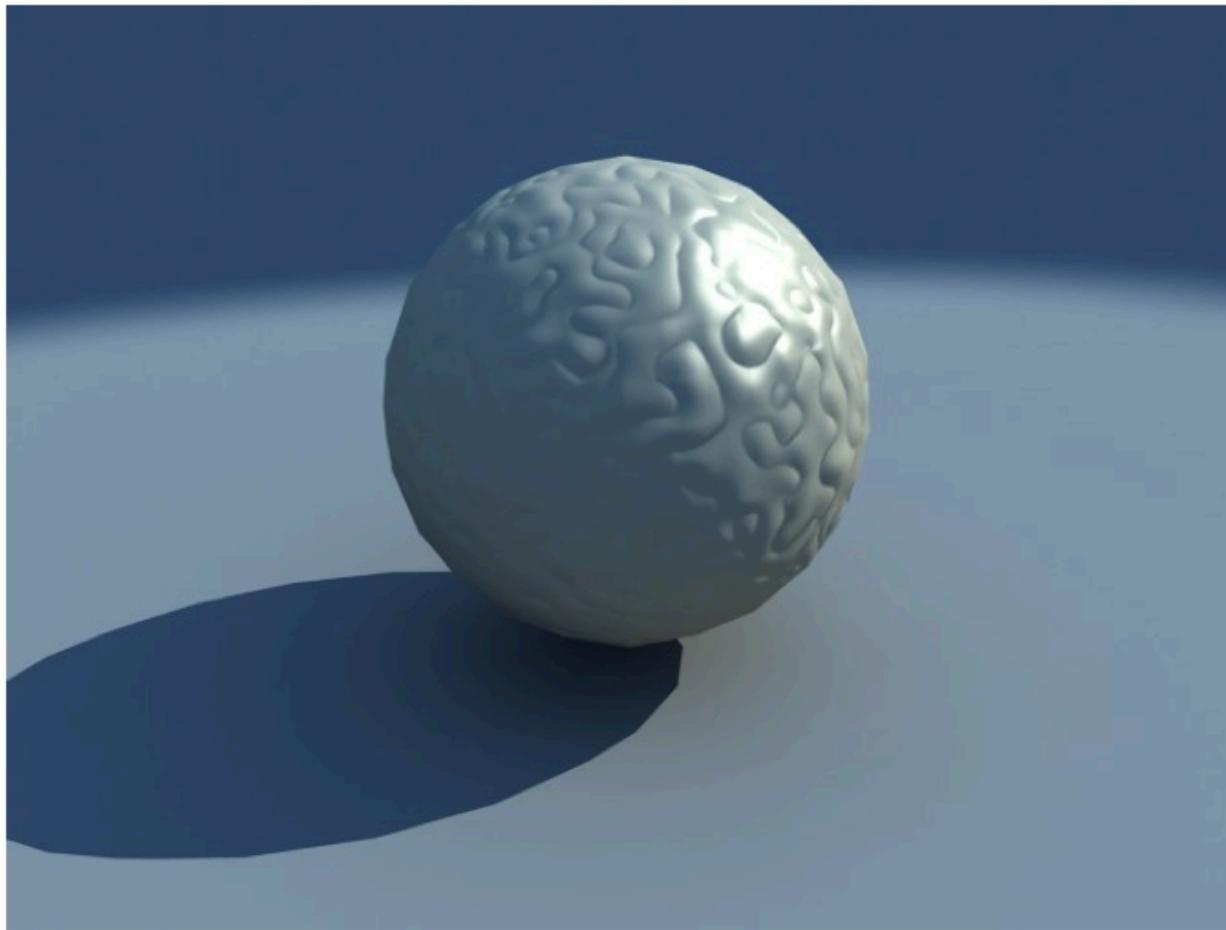
Displacement Mapping

- Autre technique pour donner du relief à des objets surfaciques
- N'est pas une technique de bump mapping
- Déplace les sommets dans une direction particulière :
 - Permet d'obtenir des silhouettes d'objets complexes
 - Nécessite beaucoup de sommets dans le modèle !

Displacement Mapping



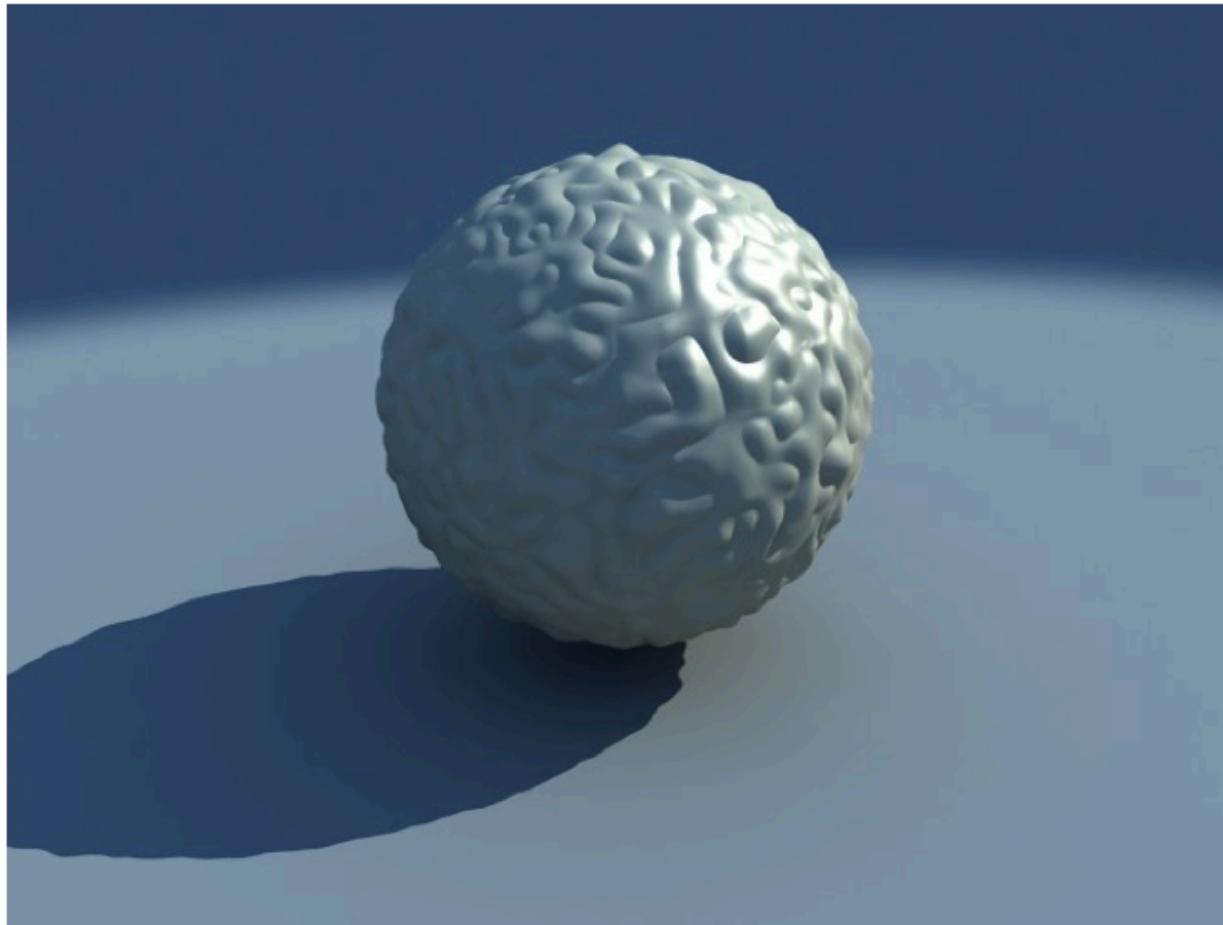
Displacement Mapping



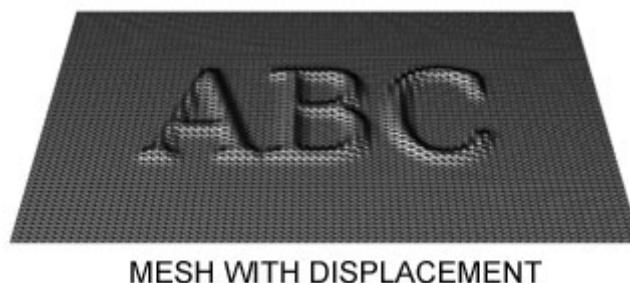
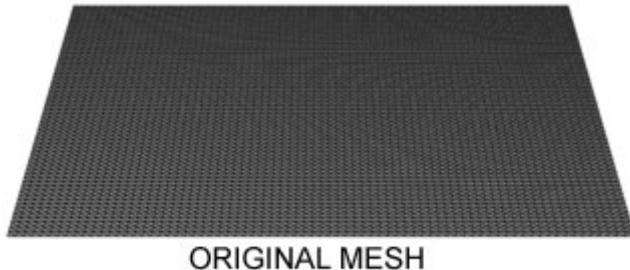
Bump-mapping

175

Displacement Mapping



Displacement Mapping : Principe



Les shaders !

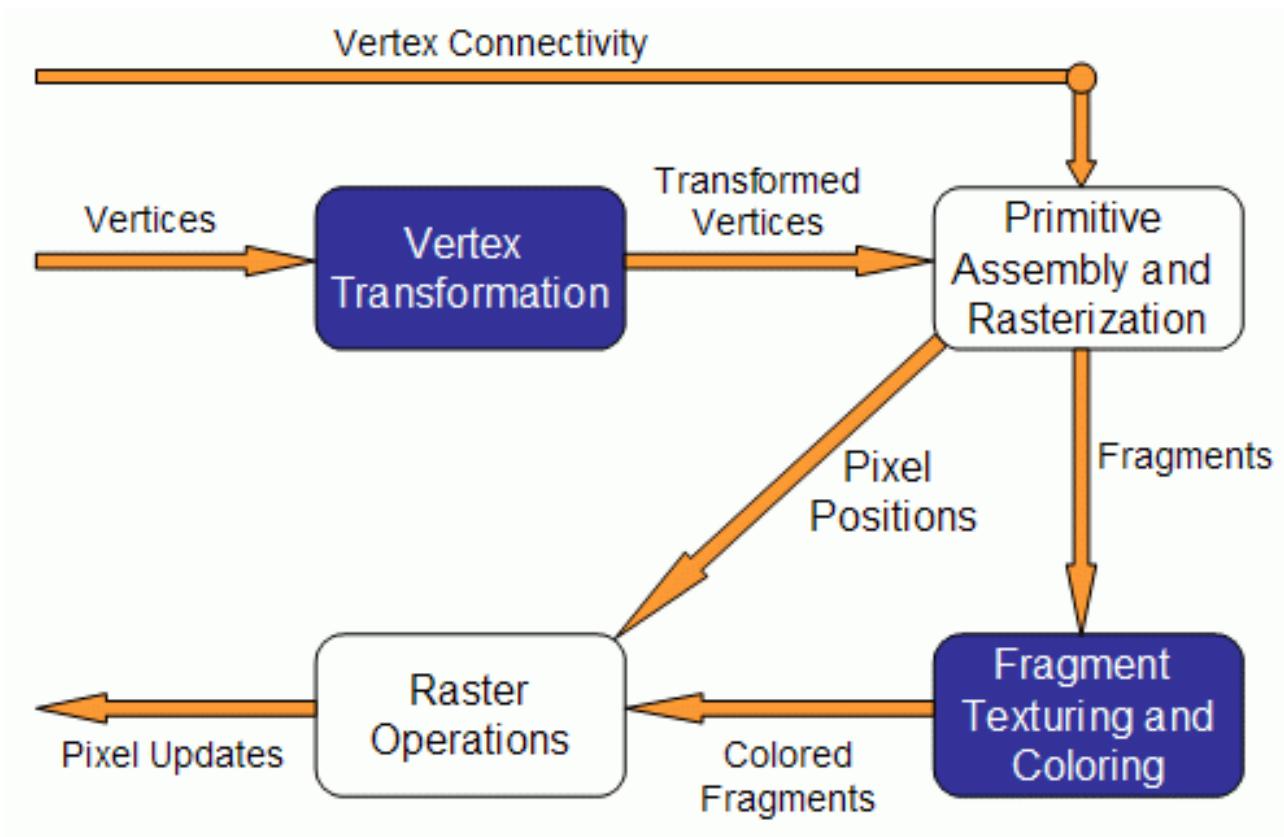
- Parlons maintenant des versions plus récentes d'OpenGL qui ont introduit la notion de **shaders**
- Mais un **shader** c'est quoi donc??
- Un **shader** est un programme (donc une suite d'instructions) qui permet de **paramétrer** une partie du pipeline de rendu OpenGL

Les shaders !

- Il existe (pour le moment) 3 types de shaders :
 - *Vertex*
 - *Fragment*
 - *Geometry*
- Depuis OpenGL 4.3, introduction des *Compute Shaders* qui permettent de faire des calculs sur le GPU (cf. CUDA, OpenCL)

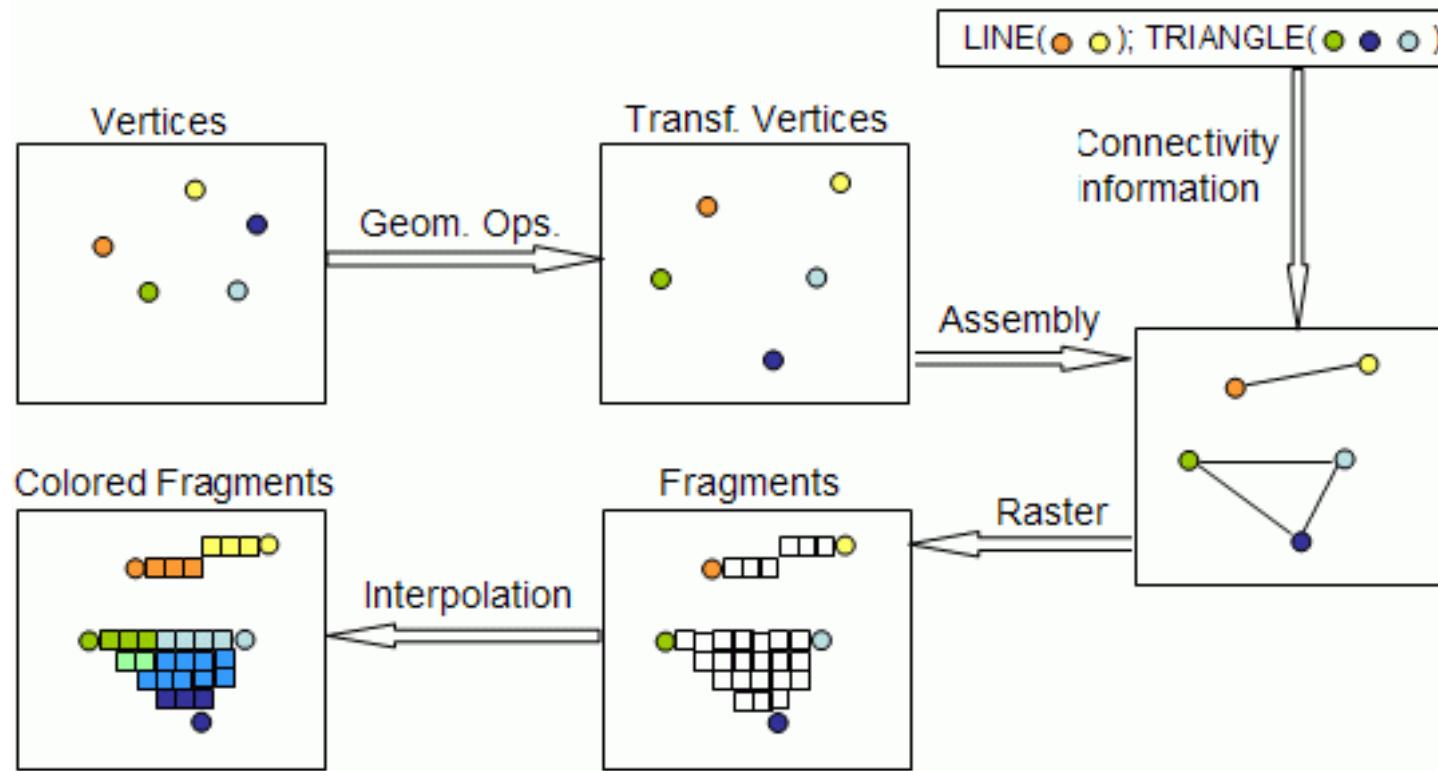
Les shaders comment ça marche ?

- Rappel (simplifié) du pipeline



Les shaders comment ça marche ?

- Rappel (simplifié) du pipeline



Les shaders comment ça marche ?

- Les shaders permettent de remplacer les fonctionnalités fixes du pipeline graphique :
 - *Vertex shaders* → opérations de transformations des sommets :
 - Transformation des coordonnées des sommets
 - Transformation des normales des sommets
 - Calculs lumineux au niveau des sommets
 - Calculs colorimétriques au niveau des sommets
 - Génération et transformation des coordonnées de texture des sommets

Les shaders comment ça marche ?

- Les shaders permettent de remplacer les fonctionnalités fixes du pipeline graphique :
 - *Geometry shaders* → opérations de transformations des primitives :
 - Création et assemblage des primitives

Les shaders comment ça marche ?

- Les shaders permettent de remplacer les fonctionnalités fixes du pipeline graphique :
 - *Fragment shaders* → opérations de transformations des « pixels »:
 - Calculs colorimétriques au niveau des pixels
 - Application de textures
 - Calculs lumineux au niveau des pixels
 - Calculs de brouillard
 - Transformation de la profondeur du pixel (*depth*)

Les shaders : résumé

- Les **shaders** sont **responsables** du remplacement des fonctionnalités du pipeline fixe d'OpenGL
- Ils **peuvent** donc implémenter les opérations qui étaient réalisées par le pipeline fixe, en particulier :
 - Les transformations de repère
 - Les calculs colorimétriques
 - Les calculs lumineux
- Ou bien nous pouvons leur faire faire des choses totalement différentes !

Ecrire des **shaders** : le langage GLSL

- OpenGL possède un langage dédié à l'écriture de shaders : le GLSL
- Syntaxe proche du C
- Types spécifiques pour les opérations graphiques :
 - vec3, vec4, mat3, mat4, etc.
- **Swizzling** : accès des composants d'un vecteur de manière très simple
 - vec4 someVec;
 - float test = someVec.x+someVec.y
 - vec2 someVec2;
 - vec4 otherVec = someVec2.xyxx;

Ecrire des **shaders** : le langage GLSL

- GLSL possède également es opérations prédéfinies dédiées aux opérations classiques
- Pour plus d'infos allez sur :
- <http://www.opengl.org/documentation/glsl/>

GLSL « Hello World »

- Programme minimal avec des shaders :
 - Transformation des sommets d'une scène
 - Coloration des pixels d'une couleur unie
- C'est le minimum que l'on puisse faire
- Nous utiliserons uniquement deux shaders :
 - un vertex
 - un fragment
- Tiré du site : www.lighthouse3d.com

GLSL « Hello World »

- Vertex shader
 - Doit remplacer la fonctionnalité du pipeline fixe :
 - Multiplication par les matrices de **modelview** et de projection
 - Ces matrices peuvent être accessibles dans le shader (GLSL 1.2) ou non (GLSL > 1.5)
 - GLSL utilise des variables aux noms spécifiques pour représenter les informations importantes des différentes étapes du pipeline graphique :
 - **gl_Position**, **gl_FragColor**, etc.

Vertex Shader « Hello World »

```
// Vertex shader GLSL 1.2
void main() {
    gl_Position = gl_ProjectionMatrix *
                  gl_ModelViewMatrix * gl_Vertex;
}
// équivalent à
void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
// équivalent à
void main() {
    gl_Position = ftransform();
}
```

GLSL « Hello World »

- Fragment shader
 - Doit remplacer la fonctionnalité du pipeline fixe :
 - Donner une couleur aux pixels
 - Comme le vertex shader, en attribuant une valeur à une variable spécifique de GLSL :
 - `gl_FragColor`

Fragment Shader « Hello World »

```
// Fragment shader GLSL 1.2
void main() {
    gl_FragColor = vec4(0.8, 0.4, 0.4, 1.0);
}
```



GLSL : Toon Shading

- Toon shading : une forme simple de rendu non-photoréaliste
- Voici le rendu attendu



- www.lighthouse3d.com

GLSL : Toon Shading

- Comment fait-on ça ?
 - Les couleurs sont calculées en fonction de l'angle entre la direction de la lumière et de la normale des sommets
 - Si la normale est proche de la direction de la lumière → couleur intense
 - Sinon → couleur sombre
- Plusieurs versions :
 - per-vertex
 - per-fragment

GLSL Toon Shading Per-Vertex

- Cet exemple va montrer comment on peut communiquer entre les vertex et les fragment shaders
- Utilisation de variables spéciales: les **varying**
- Cette variable va être **calculée dans le vertex shader** puis **passée au fragment shader** pour définir la **couleur** des fragments.

GLSL Toon Shading Per-Vertex

```
// Vertex Shader GLSL 1.2
varying float intensity;

void main()
{
    vec3 lightDir = normalize(vec3(gl_LightSource[0].position));
    intensity = dot(lightDir,gl_Normal);

    gl_Position = ftransform();
}
```

GLSL Toon Shading Per-Vertex

- Nous avons calculé l'intensité de la lumière dans le vertex shader
- Cette variable sera passée au fragment shader grâce au mécanisme des varying
- Il nous reste à choisir la couleur des fragments en fonction de cette intensité
- Le fragment shader est plutôt simple :
 - Nous allons créer 4 « paliers » de couleur

GLSL Toon Shading Per-Vertex

```
// Fragment Shader GLSL 1.2
```

```
varying float intensity;
```

```
void main()
{
    vec4 color;
    if(intensity > 0.95)
        color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5)
        color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25)
        color = vec4(0.4,0.2,0.2,1.0);
    else
        color = vec4(0.2,0.1,0.1,1.0);

    gl_FragColor = color;
}
```



GLSL Toon Shading Per-Fragment

- Le résultat précédent n'est pas génial car on **interpole** dans le fragment shader les résultats calculs dans le vertex shader
- Cela ne revient pas au même de calculer l'intensité avec une normale par fragment !
- Essayons de corriger ce problème

GLSL Toon Shading Per-Fragment

- On voudrait avoir une normale par fragment
- Solution :
 - Le vertex shader calcule la **normale**
 - La stocke dans une variable **varying**
 - **L'envoie** au fragment shader
 - Le fragment shader aura donc une **normale interpolée**
- Vertex shader simplifié
- Fragment shader légèrement plus compliqué

GLSL Toon Shading Per-Fragment

```
// Vertex Shader GLSL 1.2
varying vec3 normal;

void main()
{
    normal = gl_Normal;
    gl_Position = ftransform();
}
```

GLSL Toon Shading Per-Fragment

```
// Fragment Shader GLSL 1.2
```

```
uniform vec3 lightDir;
```

```
varying vec3 normal;
```

```
void main()
{
```

```
    float intensity;
```

```
    vec4 color;
```

```
    intensity = dot(lightDir,normal);
```

```
    if(intensity > 0.95)
```

```
        color = vec4(1.0,0.5,0.5,1.0);
```

```
    else if (intensity > 0.5)
```

```
        color = vec4(0.6,0.3,0.3,1.0);
```

```
    else if (intensity > 0.25)
```

```
        color = vec4(0.4,0.2,0.2,1.0);
```

```
    else
```

```
        color = vec4(0.2,0.1,0.1,1.0);
```

```
    gl_FragColor = color;
```

```
}
```



What ????

GLSL Toon Shading Per-Fragment

- :’o(:’o(:’o(
- Pourquoi les deux résultats sont-ils les mêmes ?
 - 1^{er} exemple : on calcule l’intensité (produit scalaire) au niveau des sommets puis interpolation dans le fragment shader
 - 2^e exemple : on interpole la normale dans le fragment shader puis on calcule le produit scalaire
 - Comme l’interpolation et le calcul du produit scalaire sont des opérations linéaires → équivalence entre calcul du produit scalaire suivi d’une interpolation et interpolation suivi du calcul du produit scalaire !

GLSL Toon Shading Per-Fragment

- Problèmes :
 - utilisation d'une normale interpolée dans le fragment shader
 - cela fonctionnerait si la normale était normée (i.e. si elle avait une longueur unitaire)
 - Dans le cas contraire, la direction du vecteur interpolé est correcte, mais sa norme est fausse
- Solution : normaliser la normale !

GLSL Toon Shading Per-Fragment

```
// Fragment Shader GLSL 1.2
uniform vec3 lightDir;
varying vec3 normal;

void main()
{
    float intensity;
    vec4 color;
    intensity = dot(lightDir,normalize(normal));

    if(intensity > 0.95)
        color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5)
        color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25)
        color = vec4(0.4,0.2,0.2,1.0);
    else
        color = vec4(0.2,0.1,0.1,1.0);

    gl_FragColor = color;
}
```



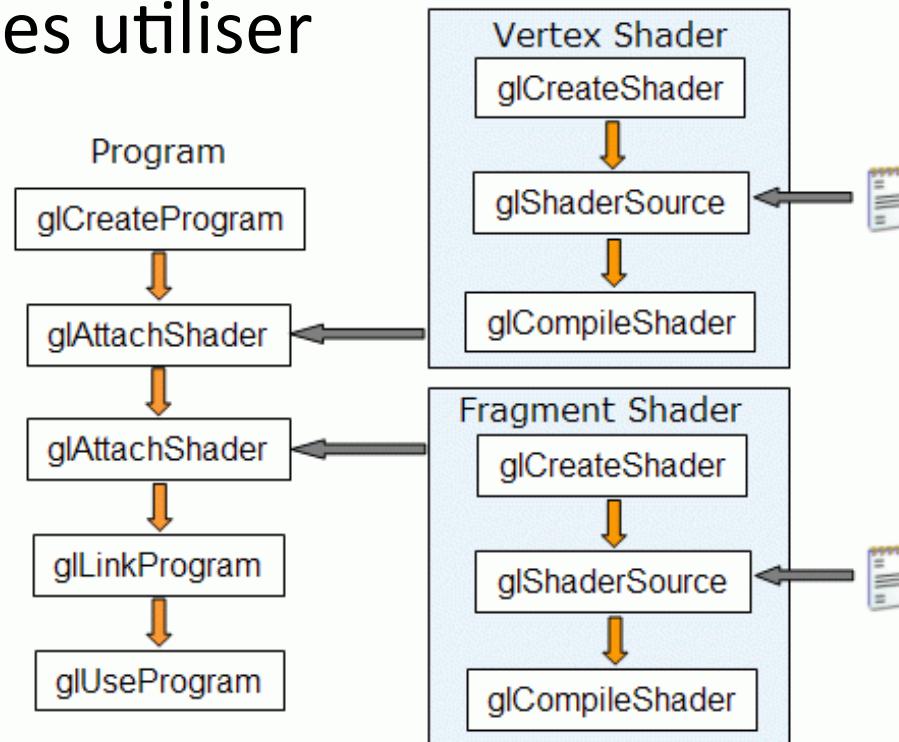
C'est mieux !

Shaders : résumé

- Les shaders ont pour but de remplacer les étapes du pipeline fixe de rendu
- Il existe différents types de shaders :
 - Vertex,
 - Fragment,
 - Geometry, etc.
- A se rappeler :
 - Un vertex shader est exécuté pour chaque sommet de la scène
 - Un fragment shader est exécuté pour chaque fragment (pixel) dessiné

Shaders : résumé

- Généralement on utilisera des vertex et des fragment shaders (plus rarement des geometry shaders)
- Il faut les linker avec votre programme OpenGL afin de les utiliser



Shaders : résumé

- Je ne présente pas ici les étapes de :
 - création, compilation d'un shader
 - création, attachement de shaders, link et utilisation d'un program
- Ces étapes sont assez faciles à réaliser, il faut juste utiliser la bonne syntaxe

Shaders : bilan

- Les shaders permettent d'apporter une très grande flexibilité à vos programmes en permettant d'implémenter à votre sauce le pipeline de rendu
- Vous pouvez donc créer des effets de rendu impossibles avant leur introduction
- On peut même combiner plusieurs shaders en différentes passes de rendu pour arriver à des résultats plus compliqués

TP : caméra à la « FPS »

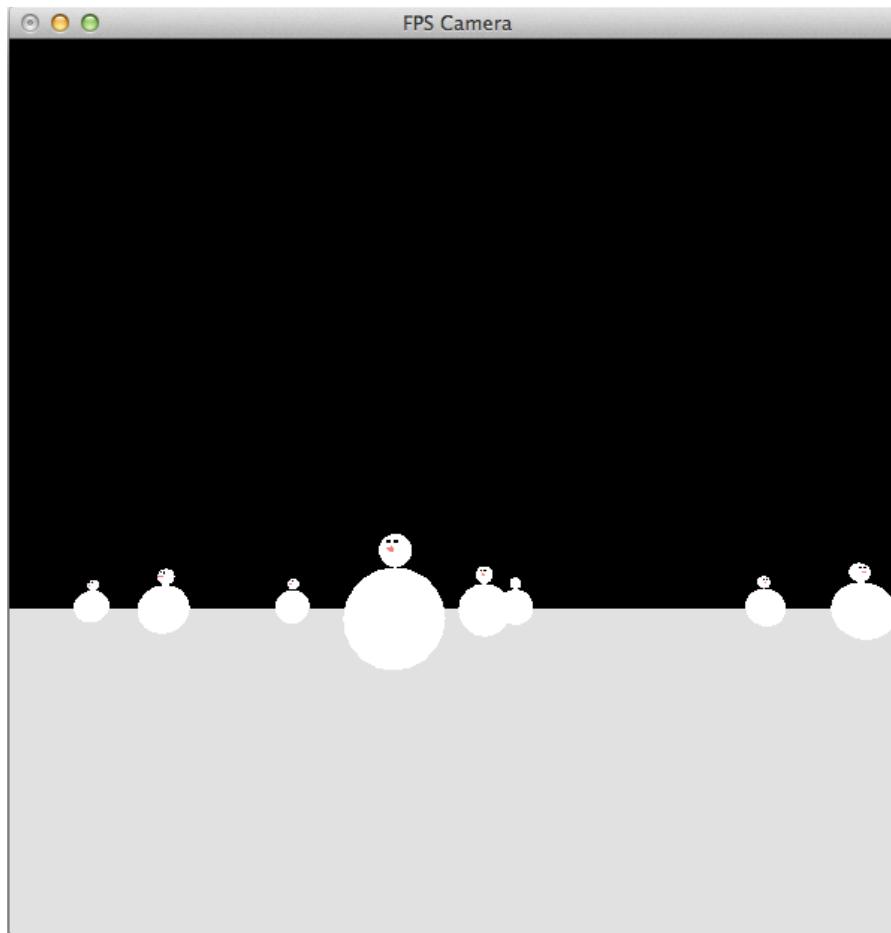
- Pour ce mini TP, le but sera de créer un système de gestion de caméra virtuelle en OpenGL permettant deux modes de déplacements dans une scène 3D :
 - Un mode « Fly » où vous pouvez vous déplacer partout dans le monde
 - Un mode « FPS » où vous pouvez regarder partout dans le monde, mais vous vous déplacez sur le « sol » de votre monde (un plan)
 - L'orientation de la caméra est contrôlée par la souris (avec un clic gauche)
 - Le déplacement de la caméra est contrôlé par les flèches du clavier ou les touches d'un FPS classique (Q,D,Z,S)

TP : caméra à la « FPS »

- Pour ce faire, vous utiliserez le squelette fourni sur le serveur pédagogique :
 - Fichiers `vector3d.h` et `vector3d.cpp` : une classe simpliste représentant un vecteur 3D et les méthodes basiques associées (normalisation, produit vectoriel, etc.)
 - Fichier `main.cpp` : squelette d'une application GLUT/OpenGL vous permettant de vous concentrer sur les choses intéressantes du TP

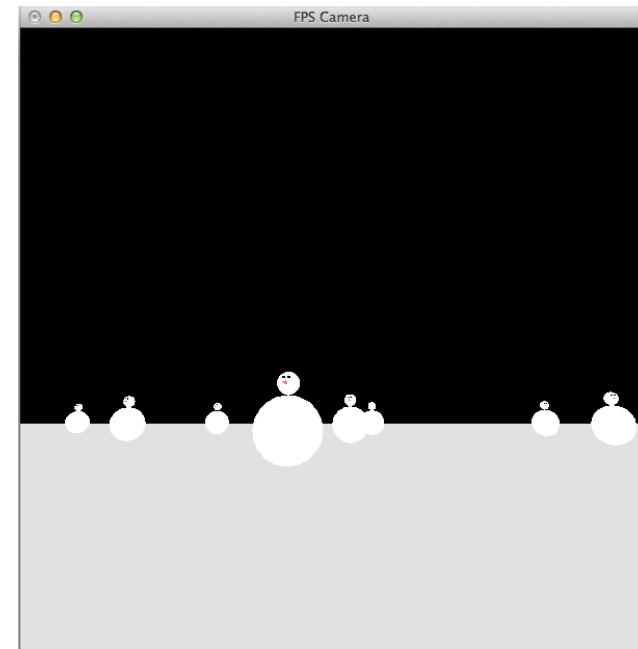
TP : caméra à la « FPS »

- Résultat attendu :



TP : caméra à la « FPS »

- Bonus :
 - Débrouillez vous pour que les bonhommes de neige soient positionnés en fonction de la position de la caméra dans le monde !
 - Ainsi vous serez toujours entourés de bonhommes de neige, peu importe vos déplacements !



TP : caméra à la « FPS »

- Aide :
 - Vous aurez besoin de deux vecteurs pour la caméra (vecteur « **forward** » et « **right** », on considérera que le vecteur « **up** » est toujours **(0,1,0)** pour nous simplifier la vie)
 - Pensez à l'utilisation des **coordonnées sphériques** pour la représentation des vecteurs de la caméra !
 - En particulier du vecteur « **forward** »
 - Le « **right** » peut être calculé en fonction des vecteurs « **forward** » et « **up** »
 - Utilisez la méthode **gluLookAt** pour gérer l'orientation et le vecteur vision (« **forward** ») de la caméra !

