

# Cours synthèse d'images en temps réel

# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



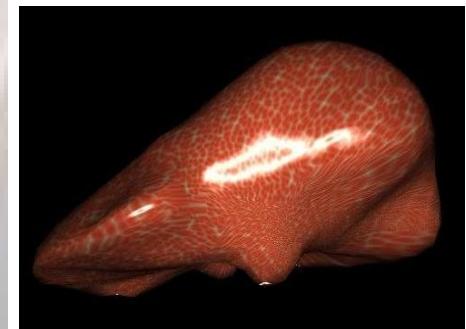
# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



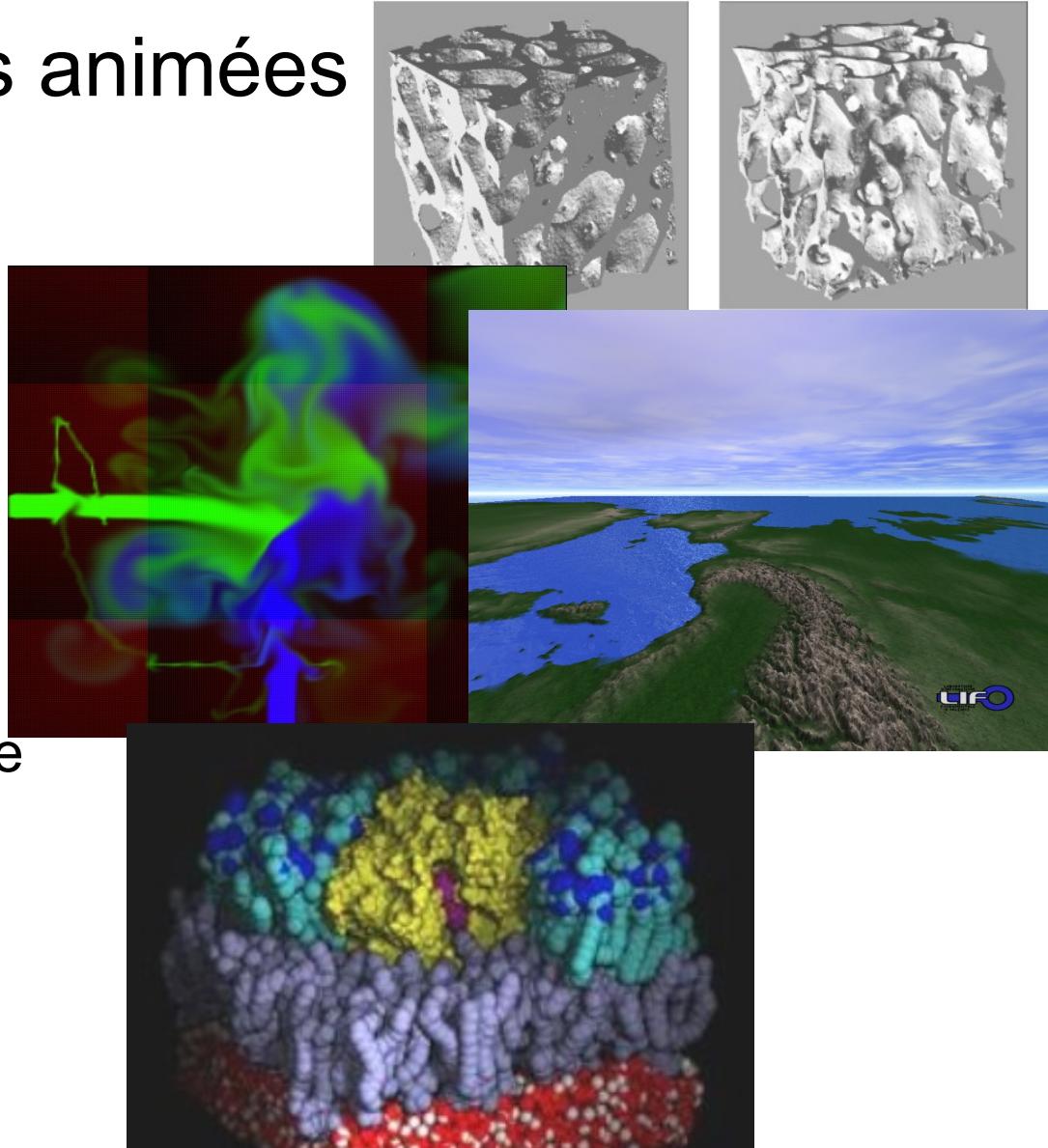
# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



# Synthèse d'images animées

- Audiovisuel
- Effets spéciaux
- Jeux vidéos
- Etudes d'impact
- Simulateurs
- Visualisation scientifique



# Synthèse d'images animées

- Audiovisuel
  - Effets spéciaux
  - Jeux vidéos
  - Études d'impact
  - Simulateurs
  - Visualisation scientifique
- Réalisme
- Temps réel

# Synthèse d'images animées

- Modélisation
  - des objets, d'une scène
- Rendu d'une image
  - à partir des objets, matières, éclairages, caméras...
- Animation
  - spécifier ou calculer
  - mouvements et déformations
- La bibliothèque OpenGL

# OpenGL

# Définition

- **GL et OpenGL** sont des librairies graphiques d'affichage trois dimensions.
- **GL**: Librairie graphique standard de Silicon Graphics (orientée visualisation). Utilisée sur les stations **SGI** ainsi que sur les stations d'autres constructeurs (sous licence).
- **OpenGL**: Sous-ensemble de GL ne nécessitant pas une licence SGI (orienté visualisation)
- Sous Linux: MESA
- API graphique multi plate-forme.
- Depuis 1992, OpenGL est géré par l'  
« OpenGL Architecture Review Board » (**ARB**).

# L'ARB

- Constitué des majors de l'industrie de l'informatique et de l'informatique graphique:
  - ATI, Compaq, Evans & Sutherland, HP, IBM, nVidia, Intergraph, Microsoft, SGI...
  - Microsoft, l'un des membres fondateurs, s'est retiré en mars 2003.
- L'ARB a pour rôle de maintenir et de faire évoluer les spécifications d'OpenGL. Il sélectionne et intègre les nouvelles extensions de l'API.

# L'ARB

- En 2006, l'ARB a transféré le contrôle de la spécification OpenGL au Khronos Group, qui s'occupait déjà de différentes spécifications OpenGL pour les systèmes embarqués et les consoles de jeux vidéo
- Les spécifications d'OpenGL sont relativement stables. Aujourd'hui nous sommes à la version 4.6. OpenGL 2.1, OpenGL 3.3 et OpenGL 4.6, Vulkan cohabitent.

- OpenGL est indépendante de la plate-forme matérielle, et du système d'exploitation.
- Ouverture
- Souplesse d'utilisation
- Disponibilité sur des plate-formes variées,
- OpenGL est utilisée dans de nombreuses applications scientifiques, industrielles ou artistiques 3D et certaines applications 2D vectorielles.
- Cette bibliothèque est également populaire dans l'industrie du jeu vidéo où elle est en rivalité avec Direct3D (sous Microsoft Windows).
- Une version nommée OpenGL ES a été conçue spécifiquement pour les applications embarquées (téléphones portables, agenda de poche...).

# Compléments d'OpenGL

- OpenGL ne gère plus certaines données Mathématiques nécessaire à OpenGL
- GLM (OpenGL Mathematics) est une bibliothèque libre utilitaire d'OpenGL apportant au programmeur C++ tout un ensemble de classes et de fonctions permettant de manipuler les données pour OpenGL
  - Opérations sur les matrices 3D
  - Création de matrices de rotation, translations, dilatation...
  - Créations de matrices de projection...
  - ...

```
#include <glm/glm.h>

view=glm::translate(glm::mat4(1.0f),glm::vec3(0.0f,0.0f,-5.0f));
```

# Compléments d'OpenGL

- OpenGL ne gère pas non plus le fenêtrage, et les entrées/sorties avec le clavier ou la souris.
- Pour celà, on utilise une bibliothèque complémentaire :
  - GLX pour X Windows (fonctions ayant pour préfixe **glx**)
  - WGL pour Microsoft Windows (fonctions ayant pour préfixe **wgl**)
  - GLUT: OpenGL Utility Toolkit, une boite à outils indépendante du système de fenêtrage, écrite par Mark Kilgard pour simplifier la tâche d'utiliser des systèmes différents (fonctions ayant pour préfixe **glut**).
  - Swing / awt pour Java
- D'autres toolkits complets existent, comme Qt

# La librairie GLUT

- OpenGL Utility Toolkit est une librairie développée pour la plupart des plates-formes.
- Elle gère de manière transparente le fenêtrage, les menus, la gestion du clavier et de la souris...
- Elle permet de développer un code multi plates-formes intégrant les gestions spécifiques des différents systèmes d'exploitation.

# Principe: boucle événementielle

- GLUT est basé sur une boucle infinie `glutMainLoop()` de lecture des événements qui se produisent sur les fenêtres qu'il gère.
- La réaction aux événements est programmée au travers de fonctions dit de « callback ».
- L'association entre un événement et son callback est effectuée par l'appel à la fonction `glutNomEv(&NomFonction)`.
- Un certain nombre d'initialisations sont nécessaires au paramétrage de la fenêtre.

# Exemple

```
void DrawGLScene()  
{ Affichage de la sc`ene}  
  
void keyPressed(unsigned char key, int x, int y)  
{ traitement des touches clavier }  
  
int main(int argc, char *argv[ ]) {  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);  
    glutInitWindowSize(640, 480);  
    glutInitWindowPosition(0, 0);  
    window = glutCreateWindow("Mon appli");  
    glutDisplayFunc(&DrawGLScene);  
    glutKeyboardFunc(&keyPressed);  
    glutMainLoop();  
  
    return 1;
```

# GLUT: les fonctions de gestion d'une fenêtre

- `glutInit(int * argc, char **argv)` initialise GLUT et traite les arguments de la ligne de commande.
- `glutInitDisplayMode(unsigned int mode)` permet de choisir le mode d'affichage ( couleurs RVB vs couleurs indexées , déclaration des buffers utilisés).
- `glutInitWindowPosition(int x, int y)` spécifie la localisation dans l'écran du coin haut gauche de la fenêtre.
- `glutInitWindowSize(ind width, int size)` spécifie la taille en pixels de la fenêtre.
- `int glutCreateWindow(char * string)` crée une fenêtre contenant un contexte OpenGL et renvoie l'identificateur de la fenêtre. La fenêtre ne sera affichée que lors du premier `glutMainLoop()`

# GLUT: la fonction d'affichage

- `glutDisplayFunc(void (*func)(void))` spécifie la fonction à appeler pour l'affichage
- `glutPostRedisplay(void)` permet de forcer le réaffichage de la fenêtre.

# GLUT: Lancement de la boucle d'événements

- 1 Attente d'un événement
- 2 Traitement de l'événement reçu
- 3 Retour en 1.

`glutMainLoop(void)` lance la boucle principale qui tourne pendant tout le programme, attend et appelle les fonctions spécifiques pour traiter les événements.

# GLUT: Traitement des événements

- `glutReshapeFunc(void (*func)(int w, int h))` spécifie la fonction à appeler lorsque la fenêtre est retaillée.
- `glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))` spécifie la fonction à appeler lorsqu'une touche est pressée
- `glutSpecialFunc(void (*func)(int key, int x, int y))` spécifie la fonction à appeler lorsqu'une touche spéciale (caractère non-ASCII) est pressée : les flèches, les touches de fonction, etc...
- `glutMouseFunc(void (*func)(int button, int state, int x, int y))` spécifie la fonction à appeler lorsqu'un bouton de la souris est pressé.
- `glutMotionFunc(void (*func)(int x, int y))` spécifie la fonction à appeler lorsque la souris est déplacée tout en gardant un bouton appuyé.

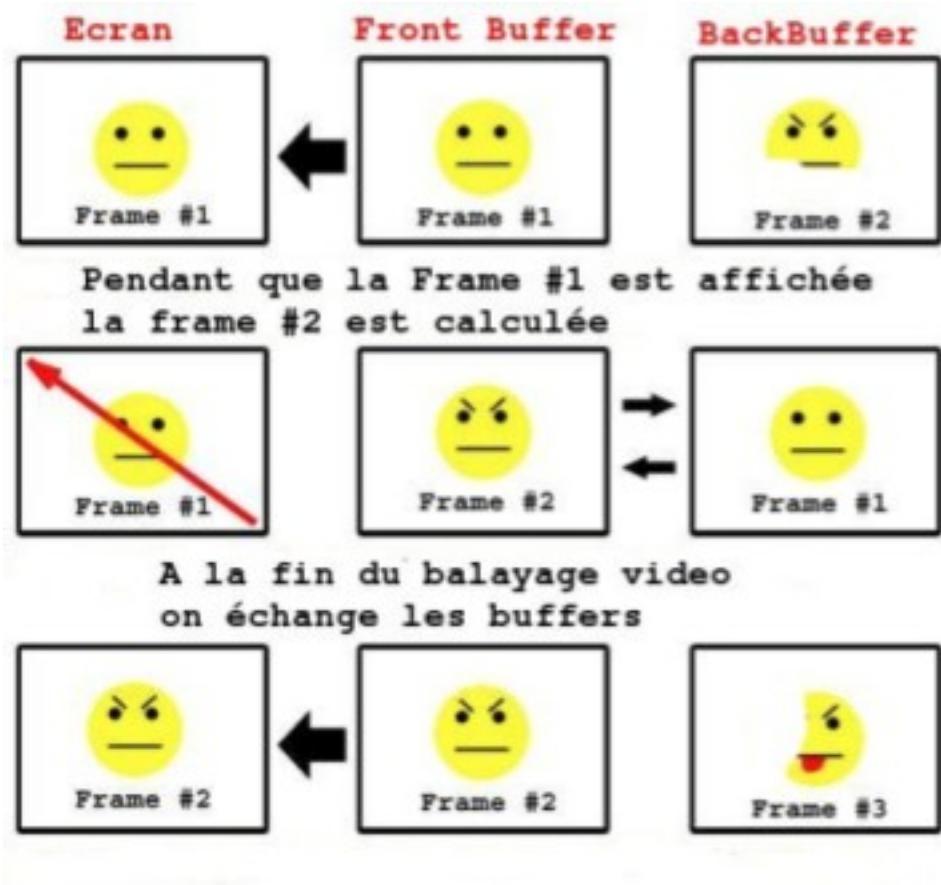
# GLUT: Exécution en tâche de fond

- `glutIdleFunc(void (*func)(void))` spécifie la fonction à appeler lorsqu'il n'y a pas d'autre événement.

Exemple : un `glutPostRedisplay()` dedans permet de forcer le ré-affichage de la fenêtre dès la fin des événements.

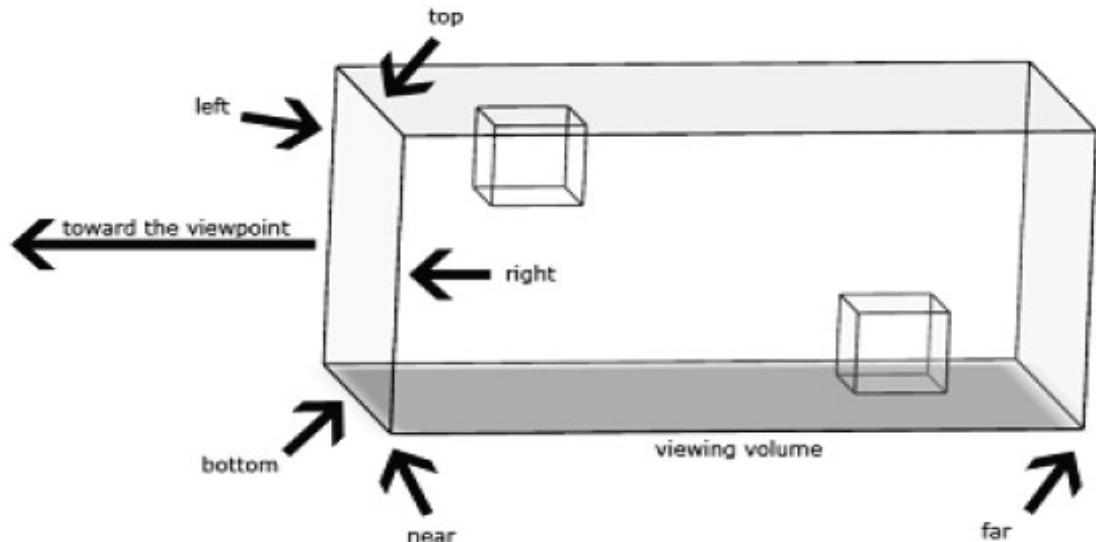
# GLUT: Animation

- Une animation peut se réaliser simplement en utilisant la technique du double buffer :
  - montrer à l'écran une image correspondant à une première zone mémoire (buffer)
  - dessiner les objets dans une deuxième zone mémoire qui n'est pas encore à l'écran,
  - elle sera affiché lorsque la scène entière y sera calculée, et à une fréquence régulière.
  - L'échange des buffers peut se faire par la commande `glutSwapBuffers(void)`



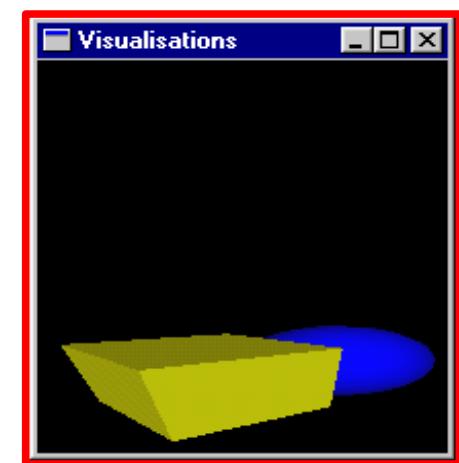
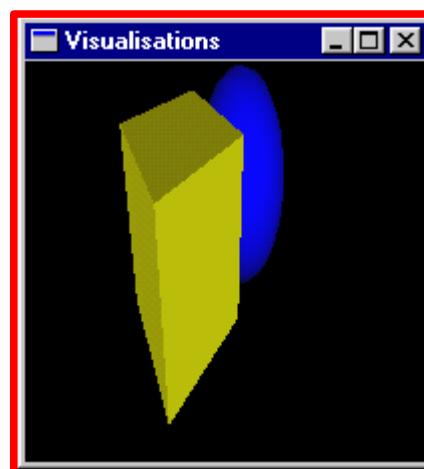
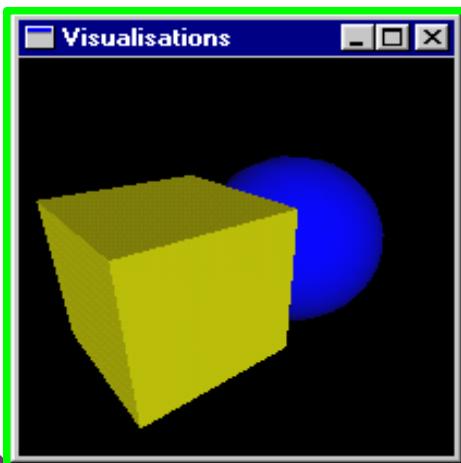
# Avant de commencer le dessin...

- Volume de visualisation
  - La scène est définie dans un repère absolu
  - Volume de visualisation: partie de la scène prise en considération
  - Dans un premier temps : `glm::ortho(GLdouble g, GLdouble d, GLdouble b, GLdouble h, GLdouble cmin, GLdouble cmax);`
  - Seuls les objets à l'intérieur du volume de visualisation sont dessinés
  - Par défaut:  
`glOrtho(-1.,1.-1.,1.,-1.,1.)`



# Projection sur la fenêtre

- Projection de la scène
  - La scène incluse dans le volume de visualisation est projetée orthogonalement au plan near
  - Le cadrage de cette projection par rapport à la fenêtre d'affichage est défini par `glViewport(x,y,width,height)`
    - `x,y` indiquent l'angle inférieur gauche du cadrage dans la fenêtre
    - `width,height` indiquent la largeur et la hauteur du cadre
  - Par défaut le cadre correspond exactement à la fenêtre d'affichage



- Effacement de la fenêtre de dessin
- Par effacement de la fenêtre de dessin, on entend effacement des zones de mémoire suivantes (définies au sein de l'environnement OpenGL):
  - le tampon couleur -> stockage de la couleur des pixels de l'image (le tampon couleur doit être vidé avant le calcul d'une image),
  - le tampon profondeur -> stockage, pour chaque pixel de l'image, d'une information de profondeur utilisée lors de l'élimination des parties cachées (si l'élimination des parties cachées est activée, le tampon profondeur doit être vidé avant le calcul d'une image),
  - le tampon accumulation -> accumulation d'images les unes sur les autres,
  - le tampon stencil-> restreindre le rendu à certaines portions de l'écran
- L'effacement est réalisé par:

```
void glClear(GLbitfield mask);
```

mask: choix du buffer à effacer

## *mask*

### **GL\_COLOR\_BUFFER\_BIT**

Effacement des pixels de la fenêtre

### **GL\_DEPTH\_BUFFER\_BIT**

Effacement de l'information de profondeur associée à chaque pixel de la fenêtre (information utilisée pour l'élimination des parties cachées)

### **GL\_ACCUM\_BUFFER\_BIT**

Effacement du tampon accumulation utilisé pour composer des images

### **GL\_STENCIL\_BUFFER\_BIT**

Non renseigné

Les valeurs de masque sont composables par "ou" pour effectuer plusieurs opérations d'effacement en une seule instruction `glClear` et rendre possible l'optimisation de ces opérations.

# Les couleurs sont définies en OpenGL de deux manières :

- Couleurs indexées : 256 couleurs sont choisies, et on se réfère au numéro de la couleur (son index). C'est un mode qui était intéressant lorsque les écrans d'ordinateurs ne savaient afficher que 256 couleurs simultanées.
- Couleurs RVBA : une couleur est définie par son intensité sur 3 composantes Rouge, Vert, Bleu. La quatrième composante est appelée canal Alpha, et code l'opacité.

- Choix de la couleur d'effacement du tampon couleur:
  - `void glClearColor(GLclampf r, GLclampf v, GLclampf b, GLclampf alpha) ;`

r, v, b, alpha: couleurs de remplissage du tampon couleur lors d'un effacement

- Choix de la profondeur d'initialisation du tampon profondeur:
  - `void glClearDepth(GLclampf depth) ;`

depth: valeur de remplissage du tampon profondeur lors d'un effacement

# OpenGL : Une machine à état

- OpenGL est basé sur le principe d'une machine à état.
  - `glEnable` / `glDisable`:
    - Permet d'activer ou non un état de la machine
    - Exemple: `glEnableVertexAttribArray( pos );` active une variable de shader.

# Sommets, lignes et polygones

- Sommet: Ensemble de trois coordonnées représentant une position dans l'espace
- Ligne: Segment rectiligne entre deux sommets
- Polygone: Surface délimitée par une boucle fermée de lignes
  - Par définition, un polygone OpenGL ne se recoupe pas, n'a pas de trou et est de bord convexe.
    - Si on veut représenter un polygone ne vérifiant pas ces conditions, on devra le subdiviser (tessellation) en un ensemble de polygones convenables.
    - GLU propose des fonctions pour gérer les polygones spéciaux.
  - Les polygones OpenGL ne doivent pas forcément être plans.
    - En revanche, le résultat à l'affichage n'est pas déterministe en cas de non planéité

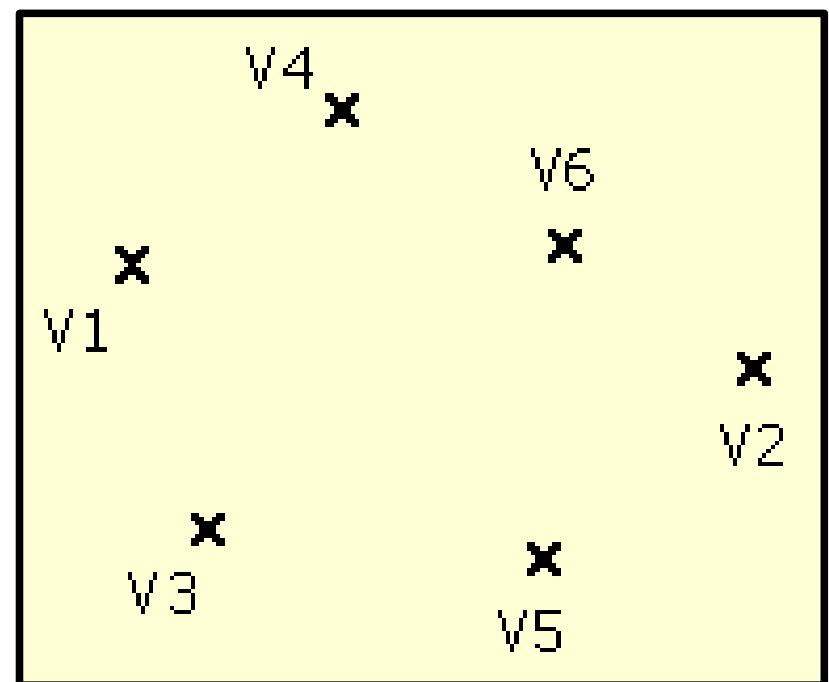
- Déclaration des positions :

```
std::vector< float > vertices = {  
    -0.5f,  0.0f,  0.0f,  
    0.0f,   0.5f,  0.0f,  
    0.5f,   0.0f,  0.0f  
};
```

- coords: coordonnées de la position
- Sert uniquement à la déclaration de sommets au sein d'une primitive graphique.
- Déclaration d'une primitive graphique
  - Une primitive graphique est constituée d'un ensemble de positions.
  - `glDrawElements( GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0 );`
- modes: GL\_POINTS, GL\_LINES, GL\_LINE\_STRIP, GL\_LINE\_LOOP, GL\_POLYGON, GL\_QUADS, GL\_QUAD\_STRIP, **GL\_TRIANGLES**, GL\_TRIANGLE\_STRIP ou GL\_TRIANGLE\_FAN

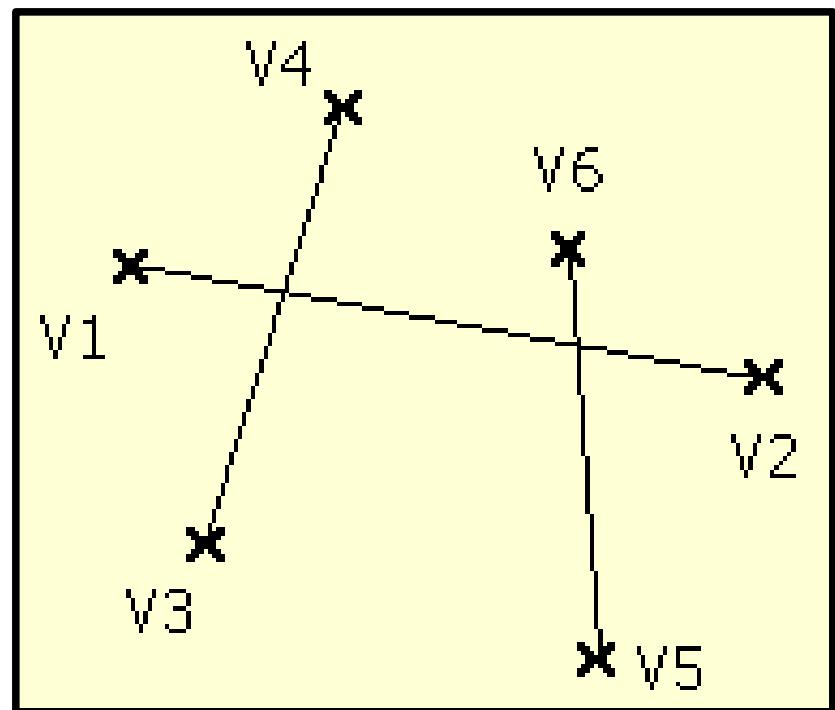
# Exemple

GL POINTS



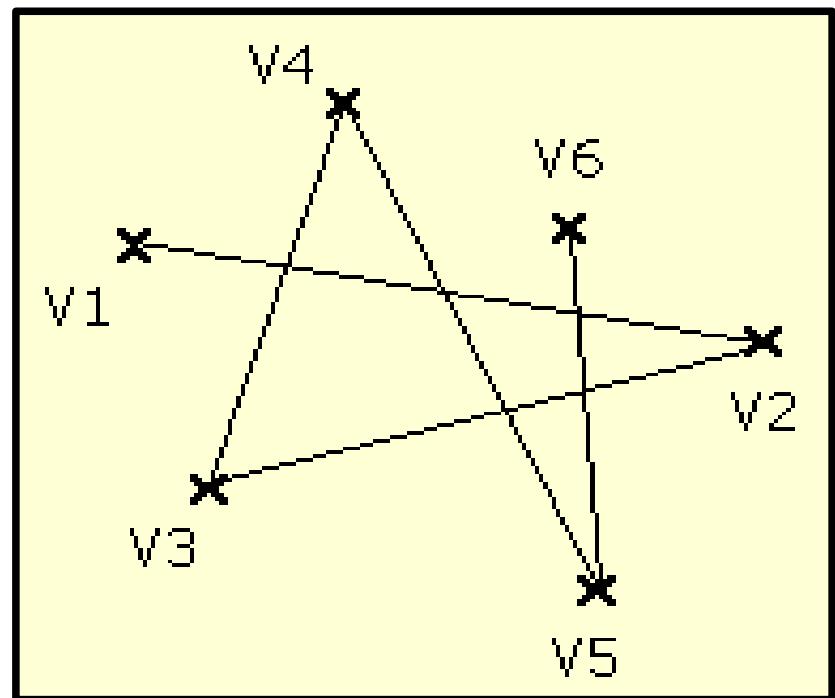
# Exemple

GL\_LINES



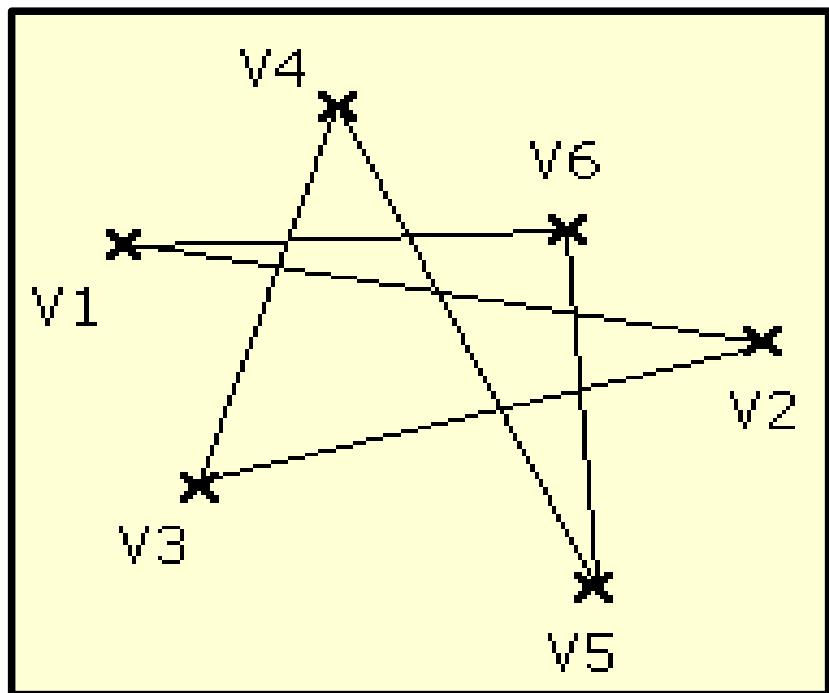
# Exemple

GL\_LINE\_STRIPS



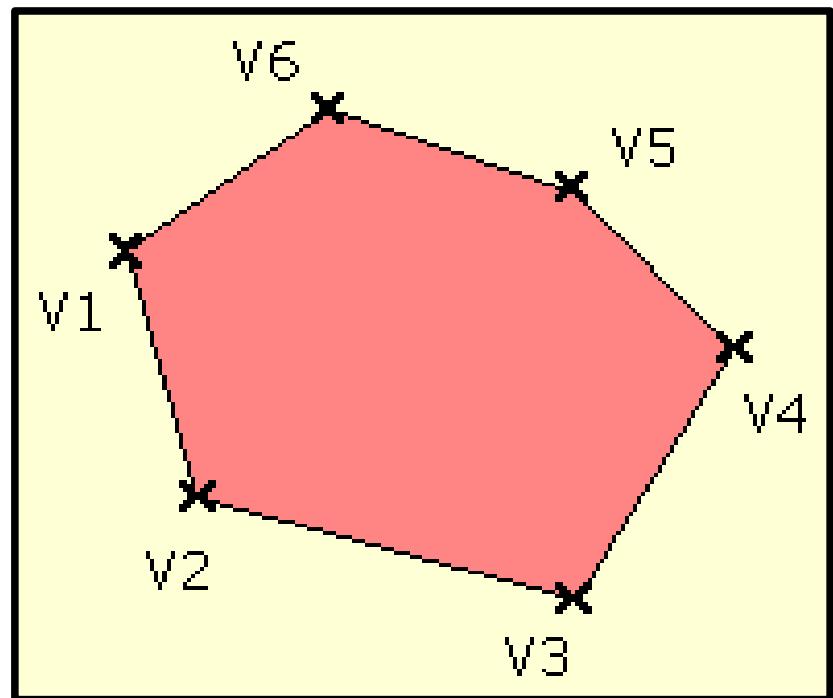
# Exemple

GL\_LINE\_LOOP



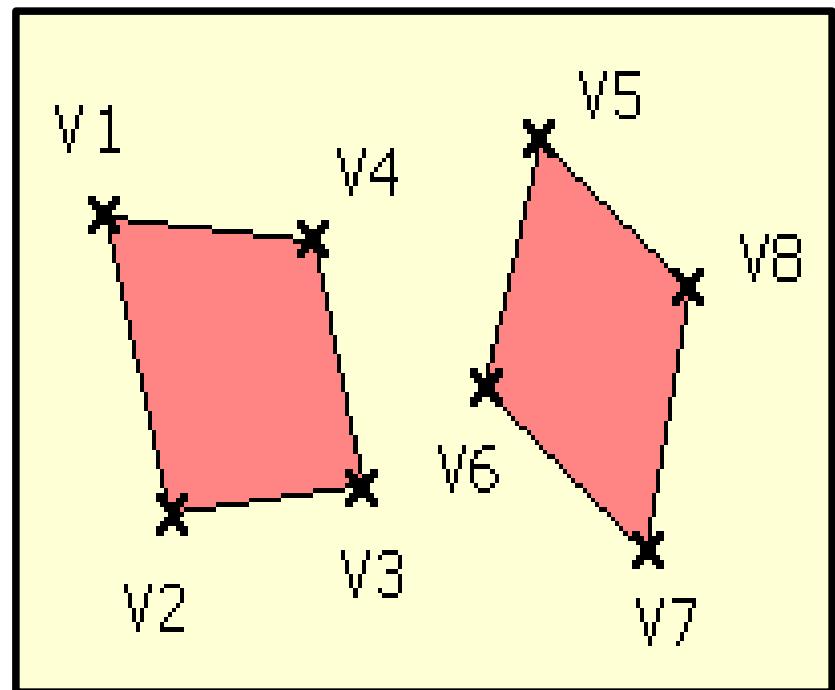
# Exemple

GL\_POLYGON



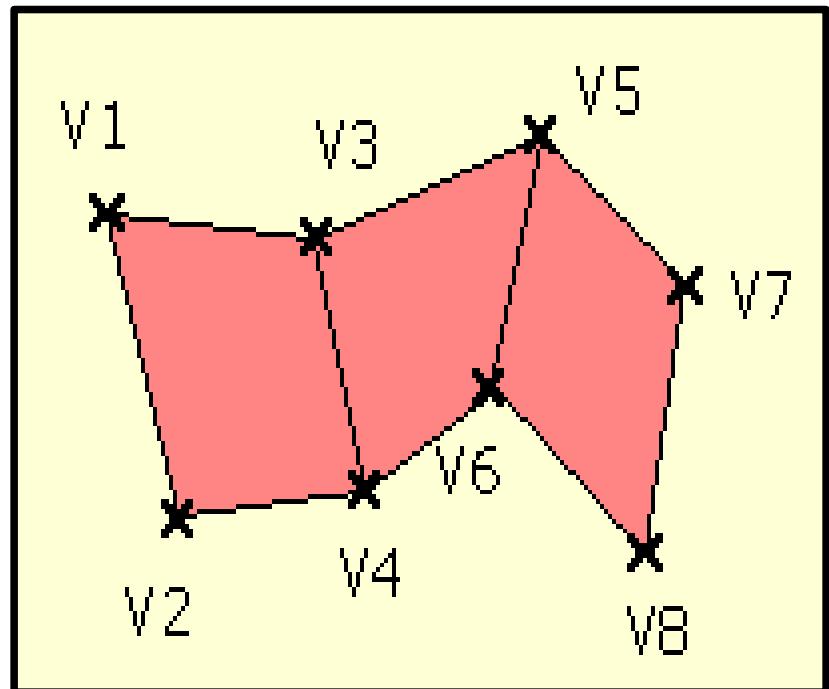
# Exemple

GL\_QUADS



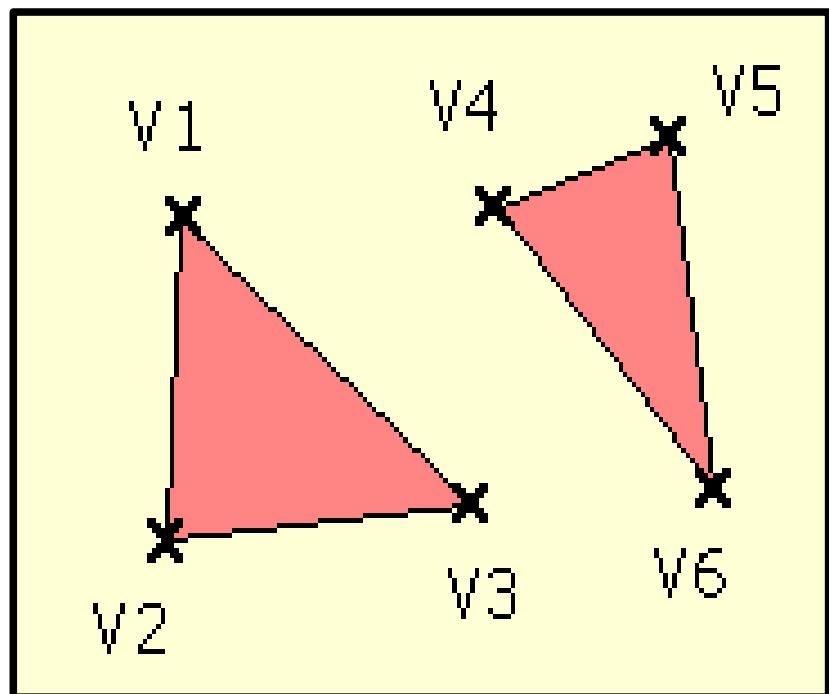
# Exemple

GL\_QUAD\_STRIP



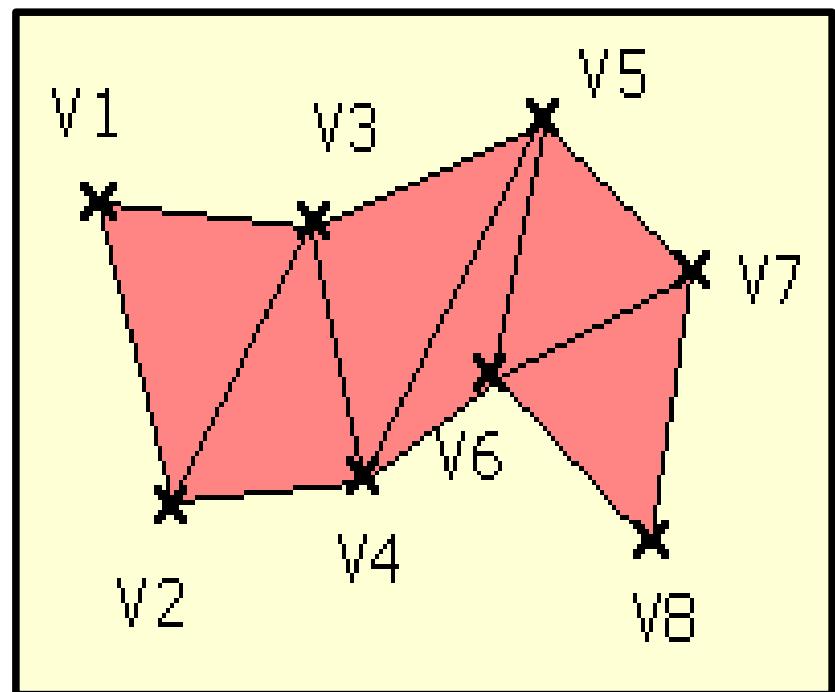
# Exemple

GL\_TRIANGLES



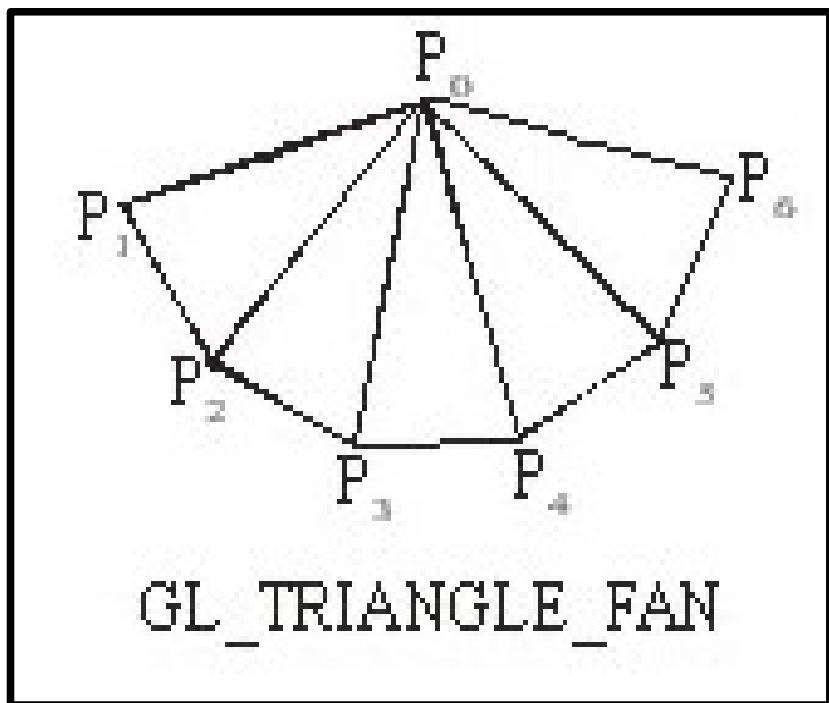
# Exemple

GL\_TRIANGLES\_STRIP



# Exemple

`GL_TRIANGLE_FAN`

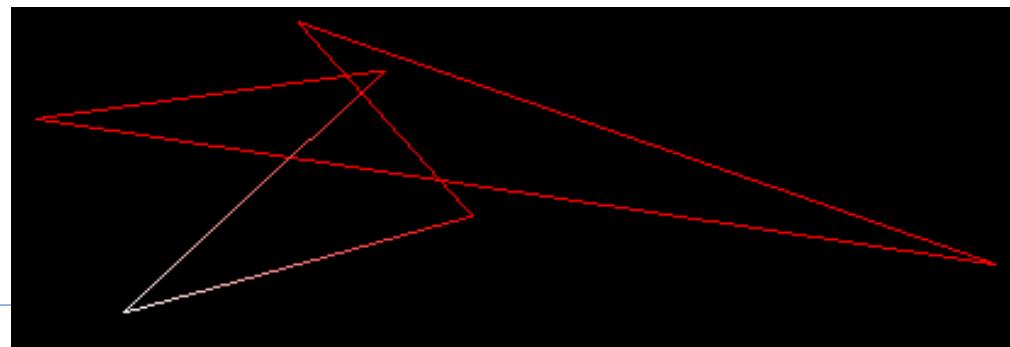


# Exemple: Création d'un polygone

```
glBegin(GL_POLYGON);

glColor3f(1.0F,1.0F,1.0F);      std::vector< float > colors = {
std::vector< float >
vertices =
{0.0F,0.0F,0.0F,
2.0F,1.0F,0.0F,
1.0F,3.0F,0.0F,
5.0F,0.5F,0.0F,
-.5F,2.0F,0.0F,
1.5F,2.5F,0.0F
};

glDrawElements( GL_POLYGON, 6,
GL_UNSIGNED_SHORT, 0 );
```



# Terminaison du tracé d'une image

Utilisé pour forcer l'exécution entière du tracé d'une image (vidage d'un pipeline d'affichage sur une station graphique, exécution des ordres de tracé sur un réseau).

- `void glFlush(void);`
- `void glFinish(void);`

`glFinish` se met en attente d'une notification d'exécution de la part du dispositif graphique tandis que `glFlush` n'est pas bloquant.

# Le Pipeline Graphique

Transformation du modèle

- Modèle géométrique: objets, surfaces et sources de lumière
- Modèle d'illumination: calcul de la lumière
- Caméra: point de vue, cône de vision
- Fenêtre: là où l'image est construite

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage

- pixels, couleurs et intensités portés par un signal analogique ou numérique adapté à l'écran/projecteur

# Le Pipeline Graphique

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rasterisation

Visibilité/Affichage

- Chaque primitive passe les étapes dans l'ordre
- Le pipeline peut être indifféremment implanté matériellement ou logiquement
- On peut ajouter à certaines étapes des moyen d'expression de traitement libres (vertex/pixel program)

# Le Pipeline Graphique

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

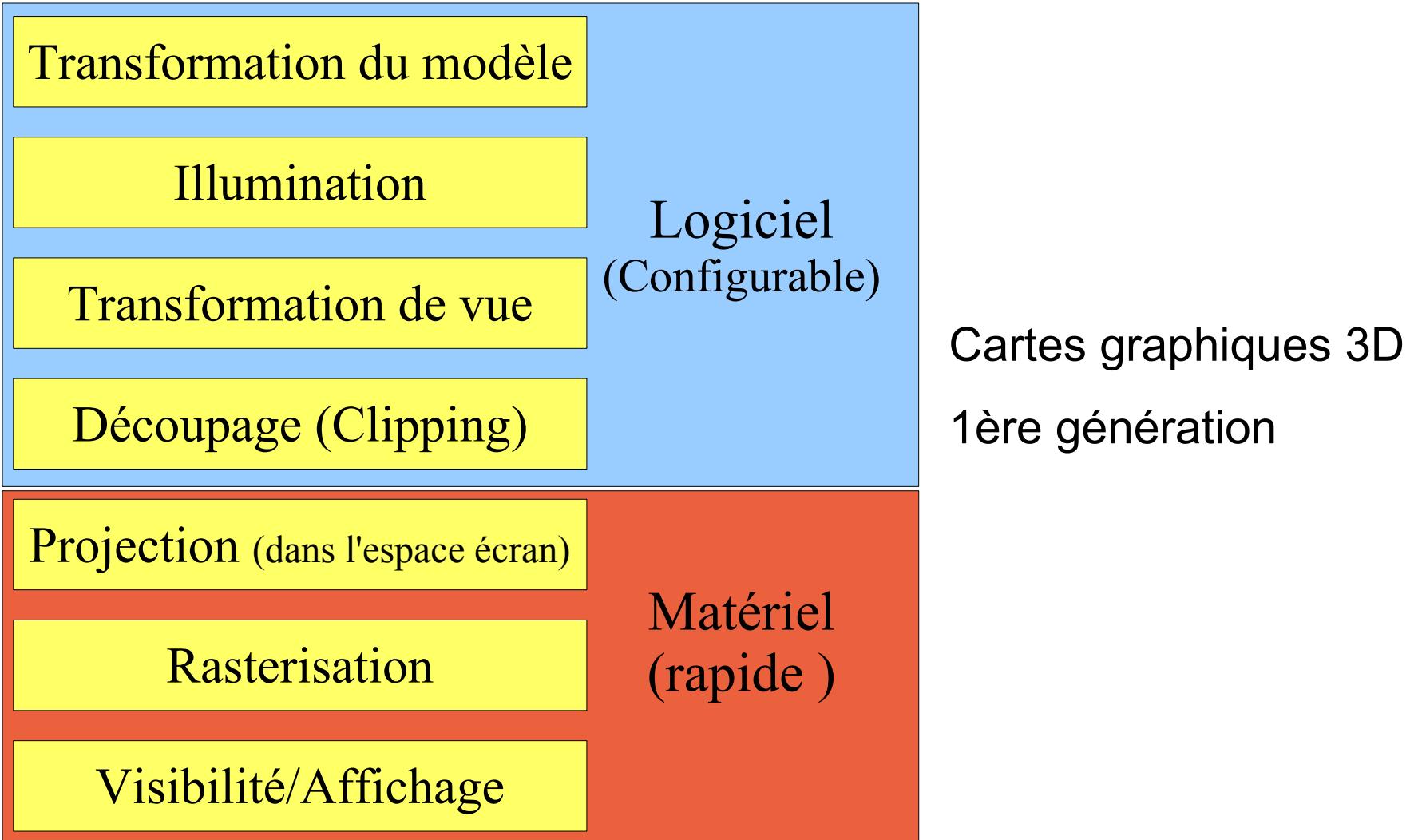
Rasterisation

Visibilité/Affichage

Logiciel  
(Configurable)

Sans carte graphique 3D

# Le Pipeline Graphique



# Le Pipeline Graphique

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rasterisation

Visibilité/Affichage

Matériel

Cartes graphiques 3D  
2ème génération

# Le Pipeline Graphique

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

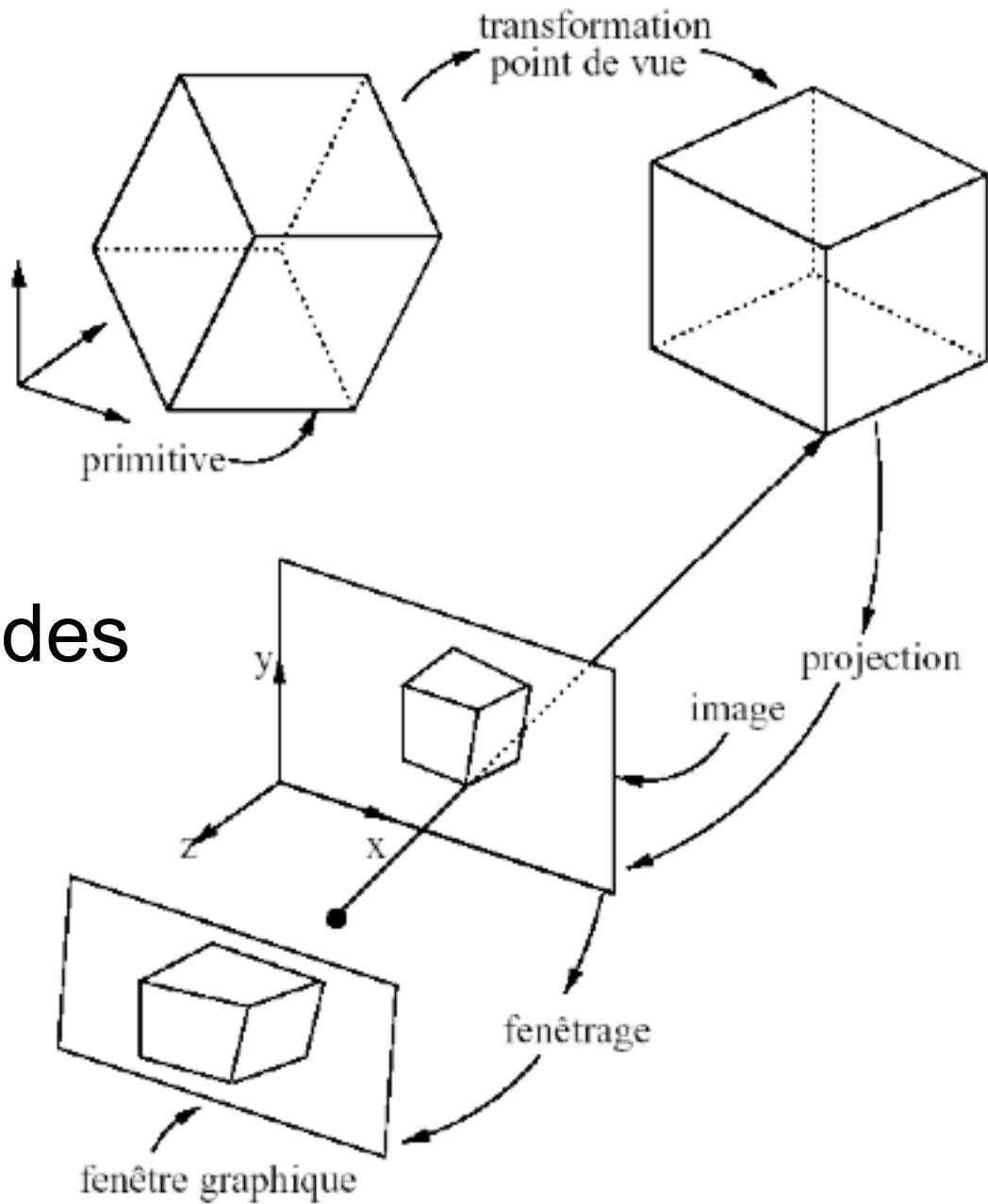
Rasterisation

Visibilité/Affichage

Matériel

Cartes graphiques 3D  
(programmable) 3ème génération

# Transformations des primitives 3D



# Transformation du modèle

Transformation du modèle

← Passage du système de coordonnées local vers des coordonnées dans un repère global

Illumination

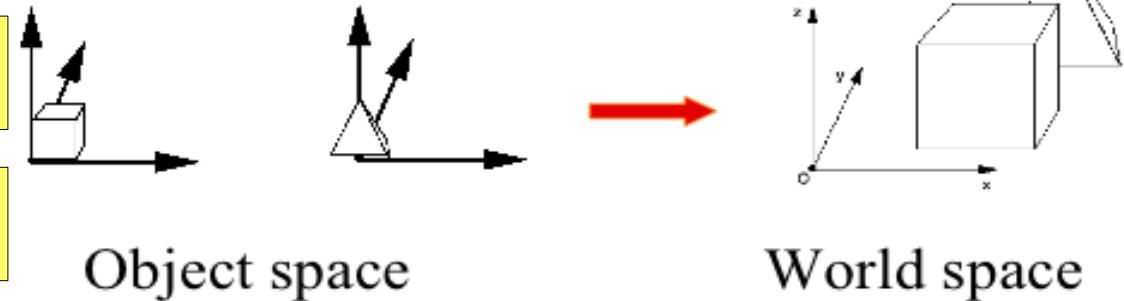
Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage



# Illumination (shading)

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

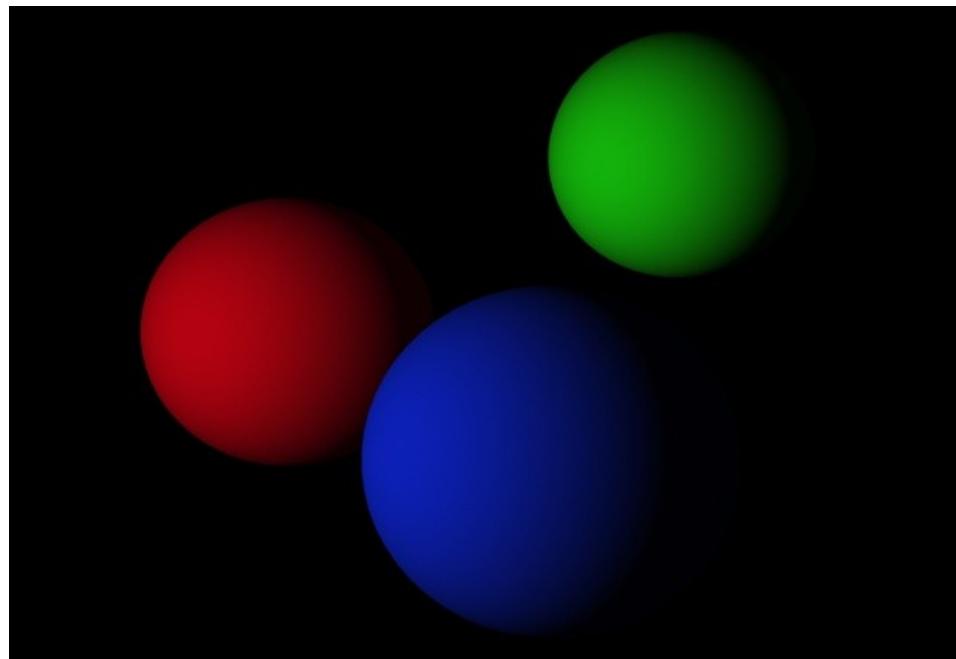
Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage

Les primitives sont éclairées selon leur matériau, leur surface, et les sources de lumières

Le calcul est local aux primitives: pas d'ombrage





Rendu facette

Gouraud

Phong

# Transformation de vue

Transformation du modèle

Illumination

Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

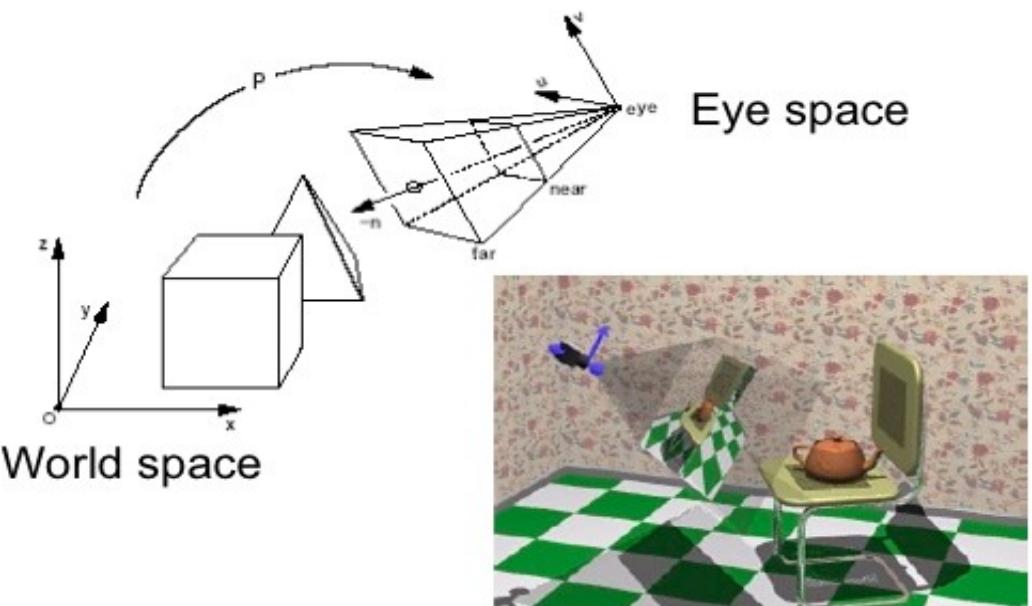
Rastérisation

Visibilité/Affichage

On passe des coordonnées du monde à celles du point de vue (espace caméra). Les sommets sont transformés par la matrice de modèle / vue (modelview chez OpenGL)



Le repère est aligné selon Z



# Transformation de vue

- Les coordonnées sont transformées par la matrice de projection afin de leur faire subir l'effet de perspective.

Transformation du modèle

Illumination

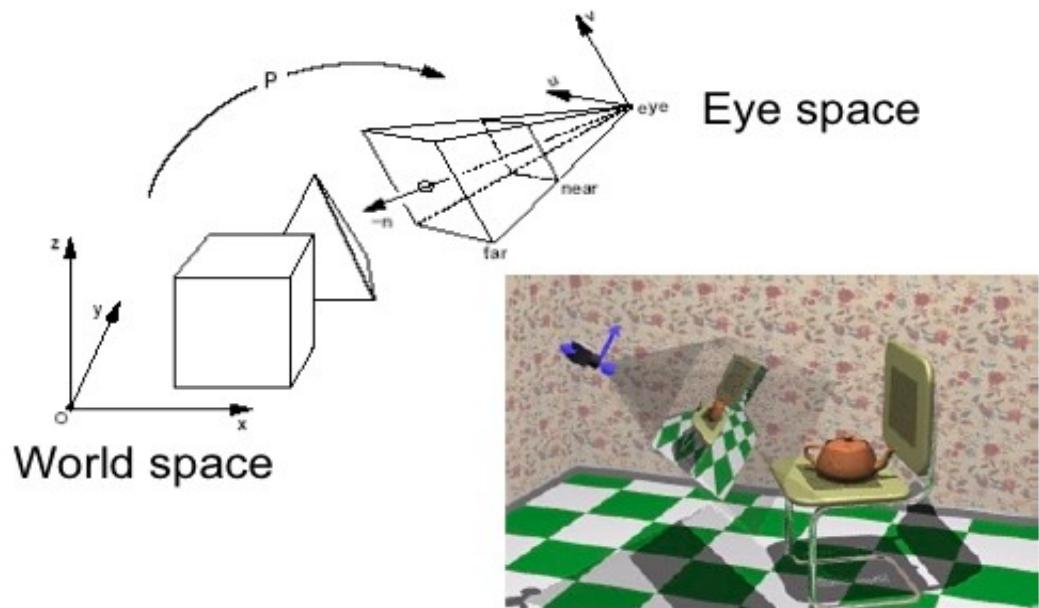
Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage



# Découpage

Transformation du modèle

Illumination

Transformation de vue

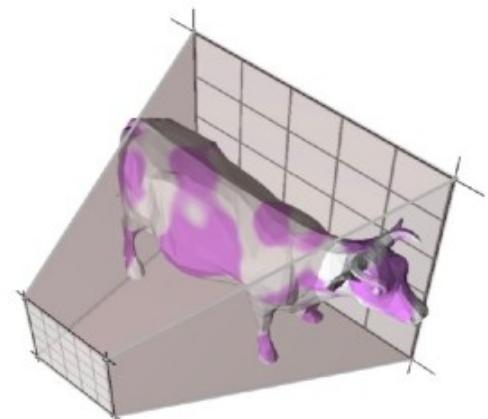
Découpage (Clipping) ←

Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage

Suppression de la géométrie en dehors du cone de vision (frustum culling)



# Découpage

- Les coordonnées ( $X$ ,  $Y$ ,  $Z$ ) obtenues précédemment sont divisées par la composante  $W$  afin d'être normalisées (NDC: normalized device coordinates). Elles se retrouvent donc entre -1 et 1.

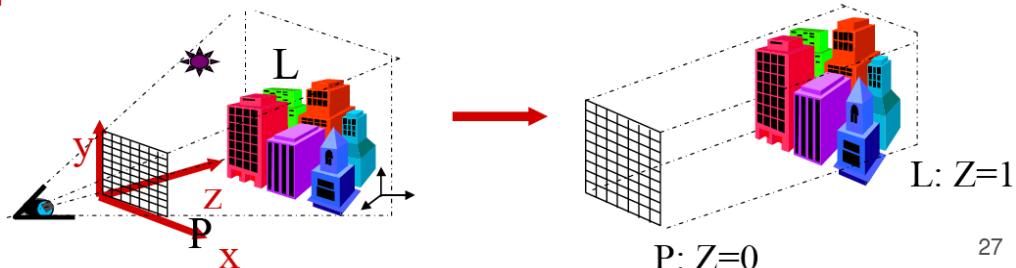
Transformation de vue

Découpage (Clipping)

Projection (dans l'espace écran)

Rastérisation

Visibilité/Affichage



27

# Projection

- Les coordonnées normalisées sont projetées sur l'image 2D. (-> window coordinates)

Transformation du modèle

Illumination

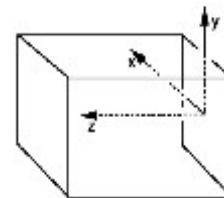
Transformation de vue

Découpage (Clipping)

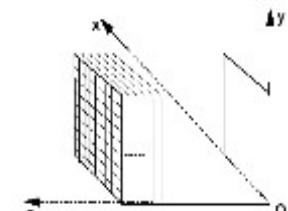
Projection (dans l'espace écran) 

Rastérisation

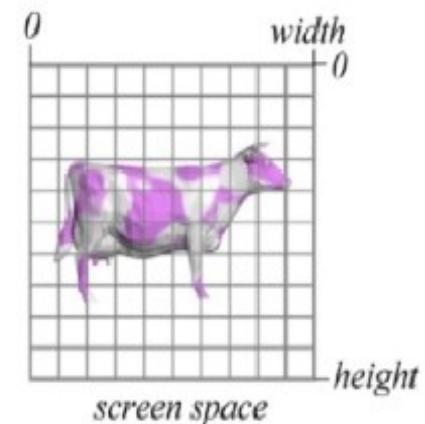
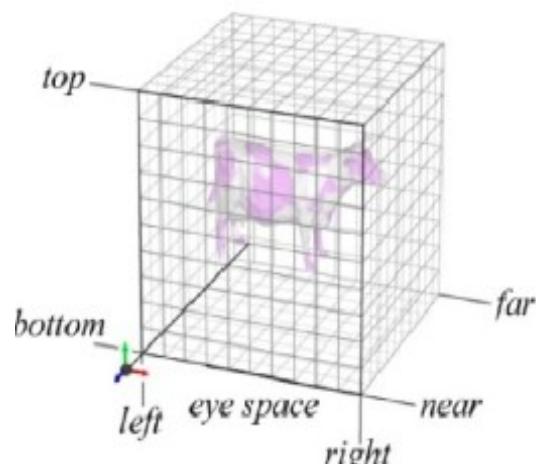
Visibilité/Affichage



NDC



Screen Space



# Processus de visualisation OpenGL

- Quatre transformations successives utilisées au cours du processus de création d'une image:
  - Transformation de modélisation (Model)
    - Permet de créer la scène à afficher par création, placement et orientation des objets qui la composent.
  - Transformation de visualisation (View)
    - Permet de fixer la position et l'orientation de la caméra de visualisation.
  - Transformation de projection (Projection)
    - Permet de fixer les caractéristiques optiques de la caméra de visualisation (type de projection, ouverture, ...).
  - Transformation d'affichage (Viewport)
    - Permet de fixer la taille et la position de l'image sur la fenêtre d'affichage.

# Processus de visualisation OpenGL

- Les transformations de visualisation et de modélisation n'en forment traditionnellement qu'une pour OpenGL (transformation **modelview**).
- L'autre transformation est celle de projection.
- Chacune de ces deux transformations peut être modifiée indépendamment de l'autre ce qui permet d'obtenir une indépendance des scènes modélisées vis à vis des caractéristiques de la "caméra" de visualisation.
- La transformation d'affichage dans la fenêtre est elle aussi paramétrable en OpenGL.

# Processus de visualisation OpenGL

- Exemple

```
void reshape( int w, int h )  
{  
    std::cout << "reshape\n";  
  
    // Modification de la zone d'affichage OpenGL.  
    glViewport(0, 0, w, h);  
  
    // Modification de la matrice de projection à chaque  
    redimensionnement de la fenêtre.  
    proj = glm::perspective( 45.0f, w/static_cast< float >( h ), 0.1f, 100.0f );  
}
```

# Transformations géométriques

```
proj = glm::perspective( 45.0f, w/static_cast< float >( h ), 0.1f, 100.0f );  
  
view = glm::translate( glm::mat4( 1.0f ) ,  
glm::vec3( 0.0f, 0.0f, -5.0f ) );  
  
mvp = proj * view;
```

- Transformation identité : `glm::mat4( 1.0f )`
- Transformation de vue : Translation de 5 en arrière.
- Multiplication de matrices pour obtenir une transformation totale des points.
- On pourrait ajouter une transformation de modélisation :

```
model=glm::rotate( view, glm::degrees( angle ) ,  
glm::vec3( 0.0f, 1.0f, 0.0f ) );  
  
mvp = proj * view * model; //ou mvp = mvp * model ;
```

- OpenGL dessine principalement des points, des lignes et des polygones:
  - L' élément de base de ces objets est le sommet ou vertex
  - Un sommet est un ensemble de 3 coordonnées ( + 1 pour avoir des coordonnées homogènes)
- OpenGL travaille en coordonnées homogènes afin de linéariser des opérations sur les matrices qui sont affines en 3d.
  - Principe:

*Une matrice de rotation en 3d est de taille 3\*3:  $A = \begin{bmatrix} A_{1\ 1} & A_{1\ 2} & A_{1\ 3} \\ A_{2\ 1} & A_{2\ 2} & A_{2\ 3} \\ A_{3\ 1} & A_{3\ 2} & A_{3\ 3} \end{bmatrix}$*

*alors qu ' une translation T s ' écrit sous la forme d ' un vecteur 3:  $T = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$*

*donc quand on veut calculer la nouvelle position d ' un point V (x , y , z) il faut faire:  $V_{nouveau} = AV + T$*

# En coordonnées homogènes...

Une matrice de rotation en 3d est de taille 4\*4:  $M_{rot} = \begin{bmatrix} A_{1\ 1} & A_{1\ 2} & A_{1\ 3} & 0 \\ A_{2\ 1} & A_{2\ 2} & A_{2\ 3} & 0 \\ A_{3\ 1} & A_{3\ 2} & A_{3\ 3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

alors une translation  $T$  s'écrit sous la forme aussi d'une matrice 4\*4:

$$M_{trans} = \begin{bmatrix} 1 & 0 & 0 & V_1 \\ 0 & 1 & 0 & V_2 \\ 0 & 0 & 1 & V_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

donc quand on veut calculer la nouvelle position

d'un point  $V(x, y, z, 1)$  il faut faire:  $V_{nouveau} = M_{rot} * M_{trans} * V$

- Par ailleurs cela permet de définir des vecteurs infinis:  $(x, y, z, 0)$
- Et de définir des classes d'équivalences constituées de vecteurs colinéaires: si  $c$  est un scalaire alors  $(cx, cy, cz, cw) = (x, y, z, w)$

# Transformations géométriques

- `model = glm::translate( model, glm::vec3(x,y,z));`
  - Compose la transformation courante par la translation de vecteur  $(x,y,z)$ . Très utilisé en modélisation.
- `Model = glm::rotate( model,a,glm::vec3(dx,dy,dz));`
  - Compose la transformation courante par la rotation d'angle **a** degrés autour de l'axe **(dx,dy,dz)** passant par l'origine. Très utilisé en modélisation.
- `model =glm::scale(model,rx,ry,rz);`
  - Compose la matrice courante par la transformation composition des affinités d'axe **x**, **y** et **z**, de rapports respectifs **rx**, **ry** et **rz** selon ces axes. Très utilisé en modélisation.

# Transformations géométriques

- Exemple:

```
glm::mat4 Mod = glm::mat4(1.0f); // Mod = I  
Mod=Mod*N; // Mod = N  
Mod=Mod*M; // Mod = NM  
Mod=Mod*L; // Mod = NML
```

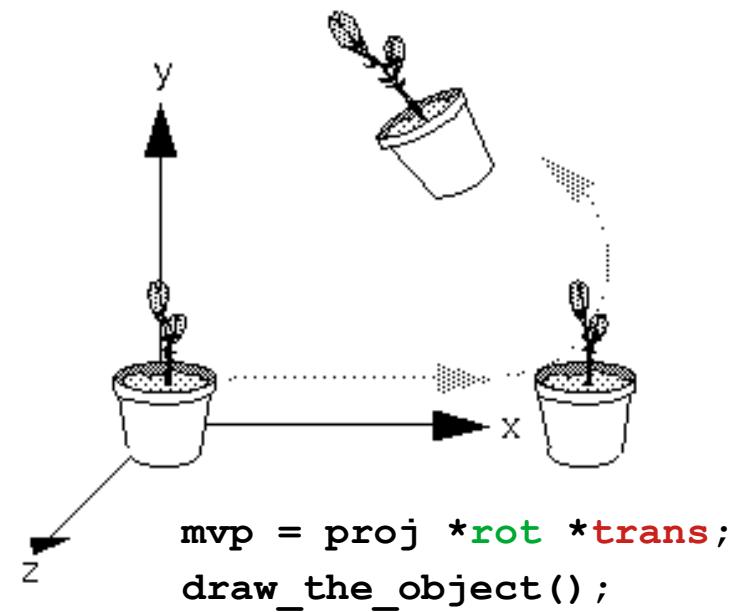
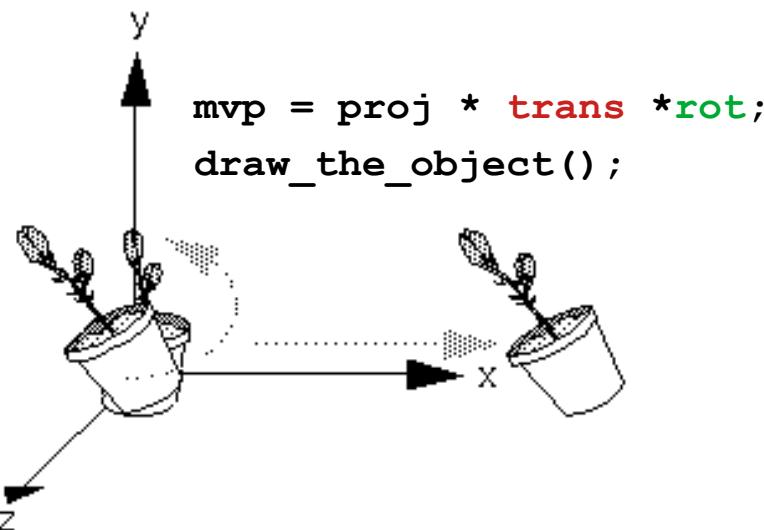
```
glUniformMatrix4fv( Mod_id , 1, GL_FALSE, &Mod[0][0]);  
glDrawElements( GL_POLYGON, 6, GL_UNSIGNED_SHORT, 0 );  
  
// chaque point v subit cette transformation : N(M(Lv))
```

- Ici L est appliqué en premier sur le point et N en dernier.

# Transformations géométriques

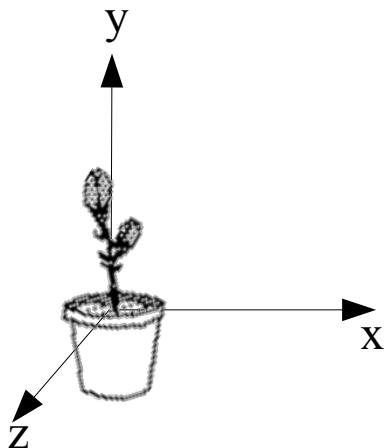
- L'ordre est important
- Exemple: une rotation de 45° et une translation

```
trans = glm::translate( glm::mat4( 1.0f ) ,  
glm::vec3(...)) ;  
  
rot= glm::rotate(glm::mat4( 1.0f ) ,  
glm::degrees( angle ), glm::vec3( 0.0f, 1.0f,  
0.0f ) );
```



# Transformations géométriques

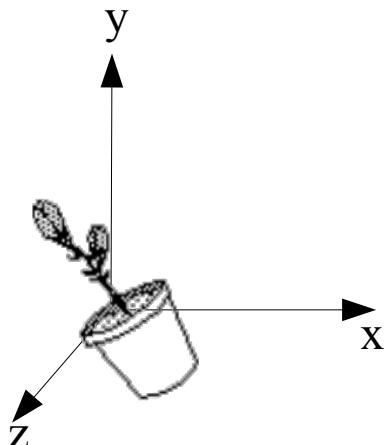
- Deux intuitions cohabitent
- Exemple: une rotation de 45° et une translation



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Transformations géométriques

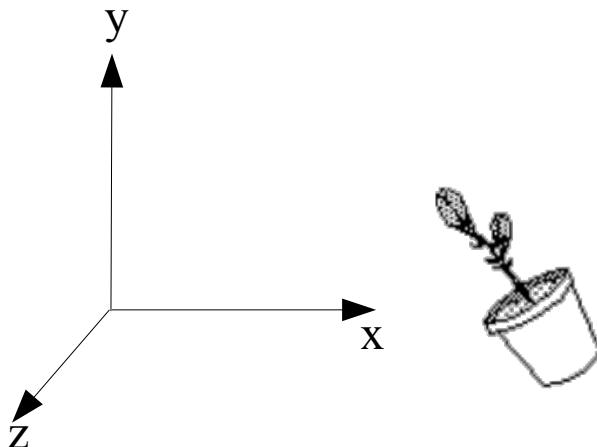
- Le repère est fixe
- Lecture du code à l'envers!



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Transformations géométriques

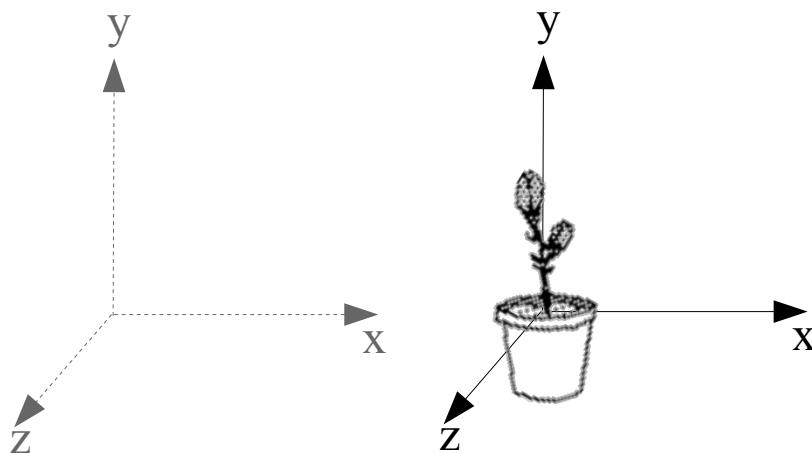
- Le repère est fixe
- Lecture du code à l'envers!



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Transformations géométriques

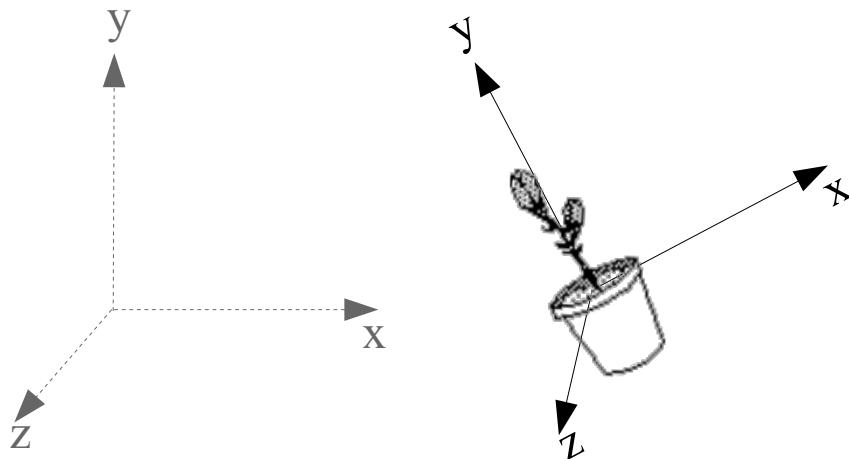
- Le repère bouge avec les transformations
- Lecture du code classique



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Transformations géométriques

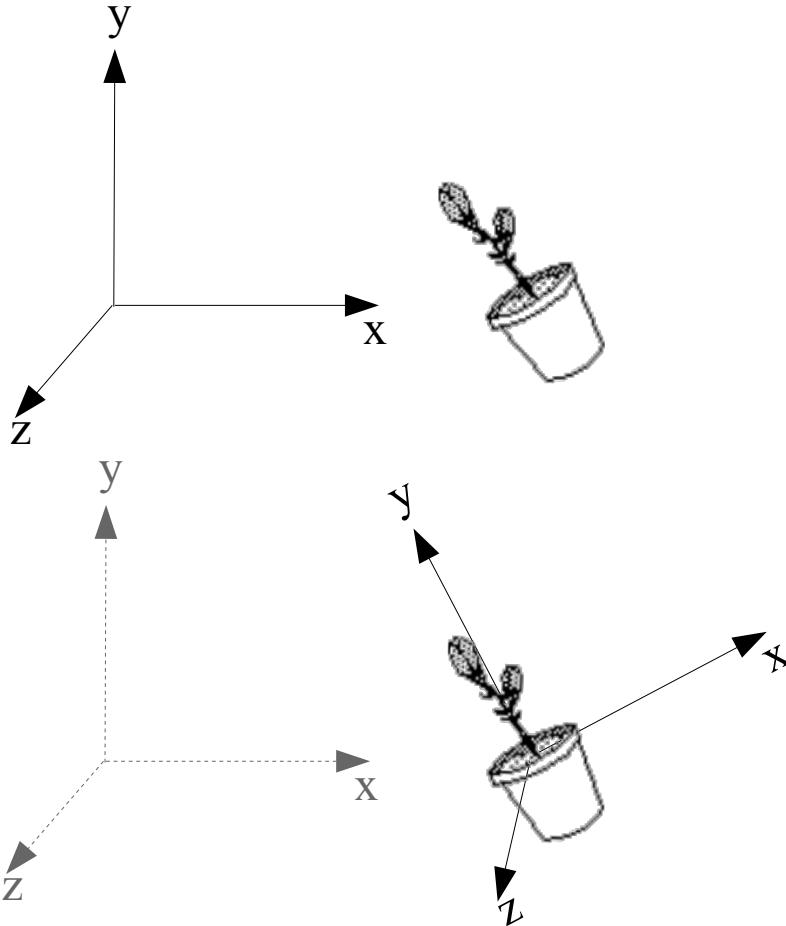
- Le repère bouge avec les transformations
- Lecture du code classique



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Transformations géométriques

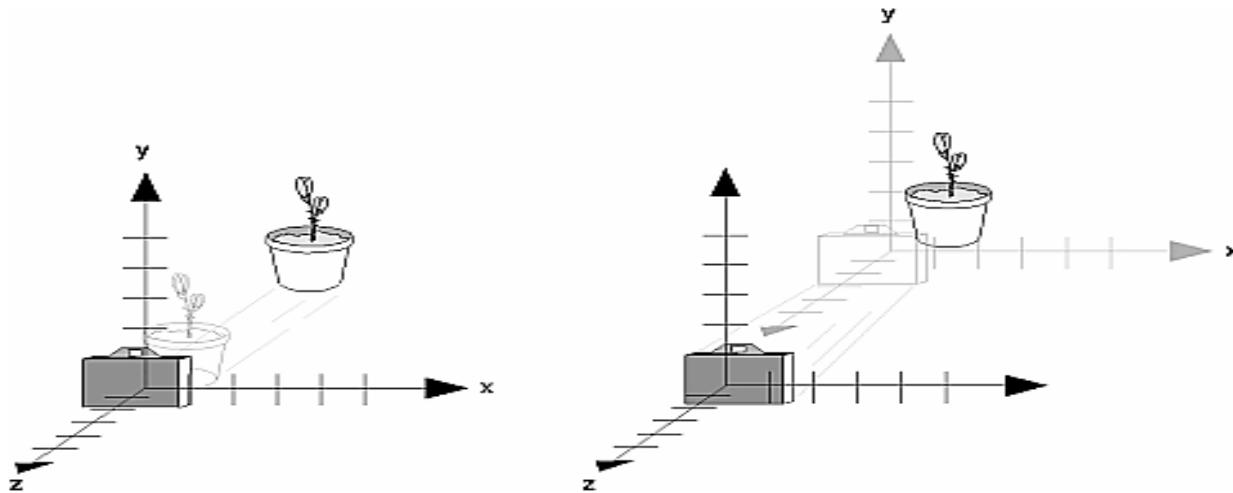
- Dans les deux cas le résultat est le même!



```
mvp = proj  
mvp= mvp * trans;  
mvp= mvp * rot;  
draw_the_object();
```

# Dualité de la modélisation et de la visualisation

- Les effets sont relatifs
  - Translater un objet de  $(0,0,-5)$  a le même effet que de translater la caméra de  $(0,0,5)$



- Transformations de modélisation+Transformations de visualisation = Transformations ModelView

# Transformation spécifique à la visualisation

```
view = glm::lookAt( glm::vec3( ex,ey,ez ) ,  
glm::vec3( cx,cy,cz ) , glm::vec3( upx,upy,upz ) );
```

- Compose la transformation courante (généralement MODELVIEW) par la transformation donnant un point de vue depuis (ex,ey,ez) avec une direction de visualisation passant par (cx,cy,cz). (upx,upy,upz) indique quelle est la direction du repère courant (fréquemment le repère global) qui devient la direction y (0.0, 1.0, 0.0) dans le repère écran.  
gluLookAt est une fonction de la bibliothèque GLU.

Exemple:

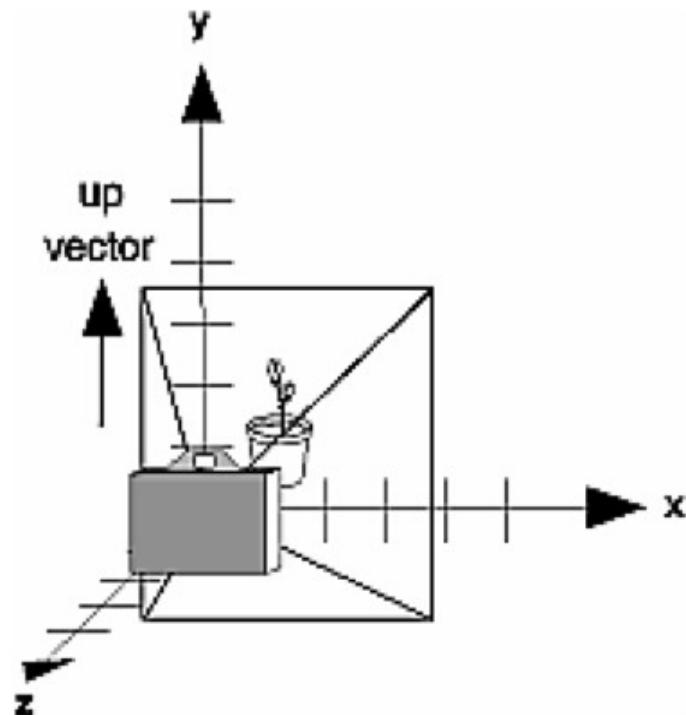
```
view =  
glm::lookAt(glm::vec3(10.0,15.0,10.0),glm::vec3(3.0,  
5.0,-2.0),glm::vec3(0.0,1.0,0.0))
```

- place la caméra en position (10.0,15.0,10.0), l'oriente pour qu'elle vise le point (3.0,5.0,-2.0) et visualisera la direction (0.0,1.0,0.0) de telle manière qu'elle apparaisse verticale dans la fenêtre de visualisation. Ces valeurs sont considérées dans le repère global.

# gluLookAt

```
view = glm::lookAt(glm::vec3(0.0, 0.0, 0.0),  
glm::vec3(0.0, 0.0, -100.0), glm::vec3(0.0, 1.0,  
0.0));
```

- La Valeur z du point de référence est -100.0
- Mais n'importe quel négatif eu suffit.
- C'est la position par défaut
- L'appel est superflu.

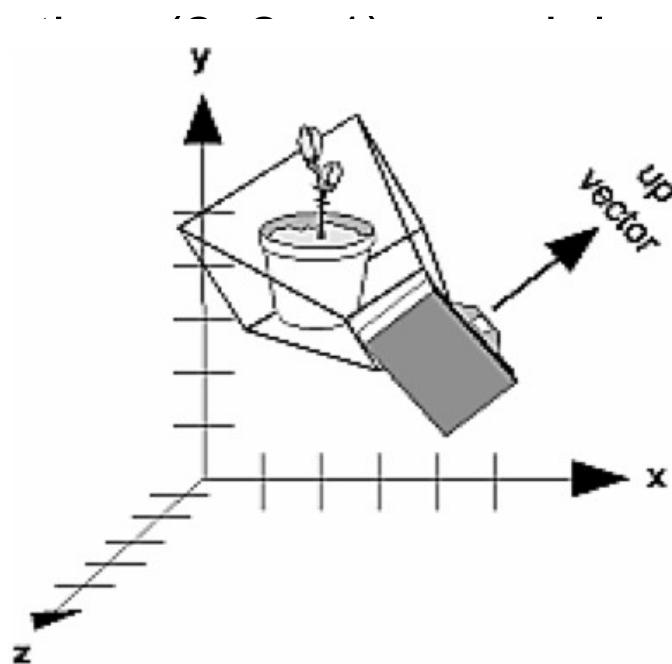


# gluLookAt

```
view = glm::lookAt(glm::vec3(4.0, 2.0, 1.0),  
glm::vec3(2.0, 4.0, -3.0),glm::vec3( 2.0, 2.0, -  
1.0));
```

- Position de la caméra :(4, 2, 1)
- Point de référence en (2, 4, -3)
- Vecteur d'orientation : une rotation du point de vue de 45°

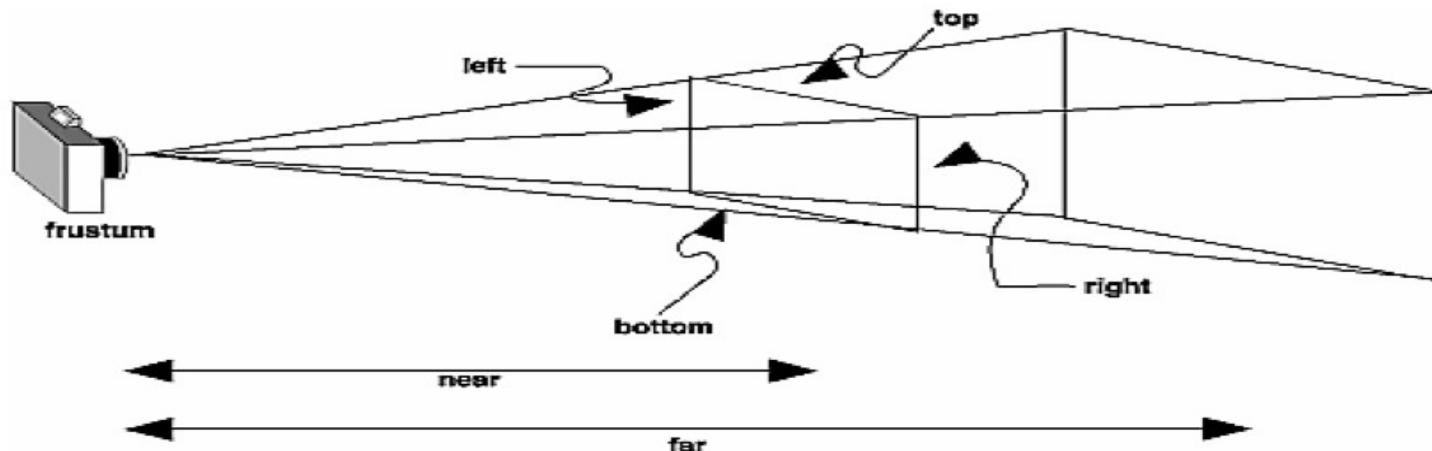
une rotation du point



# Transformations de projection

```
proj = glm::frustum(GLdouble g, GLdouble d, GLdouble b,  
b, GLdouble h, GLdouble cmin, GLdouble cmax);
```

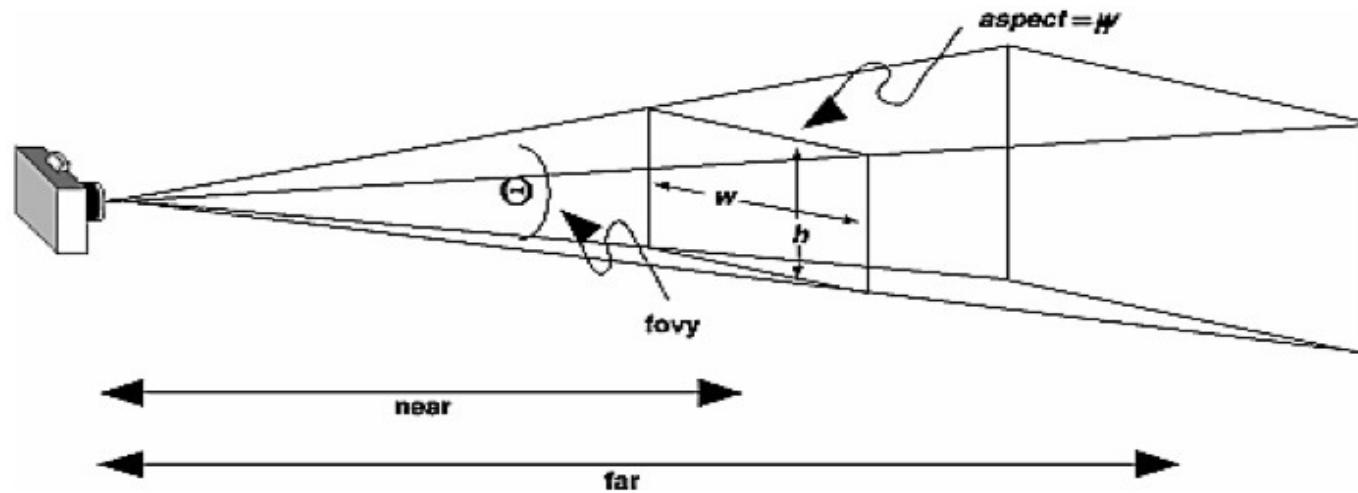
- Compose la transformation courante par la transformation de projection en perspective de volume de visualisation défini par la pyramide tronquée de sommet l'origine O, orientée selon l'axe -z et de plan supérieur défini par la diagonale  $(g,b,-cmin)$   $(d,h,-cmin)$ .
- $cmin$  et  $cmax$  sont les distances entre l'origine et les plans de clipping proche ( $-cmin$  en z) et éloignés ( $-cmax$  en z).  $cmin$  et  $cmax$  doivent avoir une valeur positive et respecter  $cmin < cmax$  car il s'agit de distances à l'origine selon l'axe -z.



# Transformations de projection

```
proj = glm::perspective(GLdouble foc, GLdouble  
ratio, GLdouble cmin, GLdouble cmax);
```

- Compose la transformation courante par la transformation de projection en perspective de volume de visualisation défini par la pyramide tronquée de sommet l'origine O, orientée selon l'axe -z, possédant l'angle foc comme angle d'ouverture verticale, l'aspect-ratio ratio (rapport largeur/hauteur) et les plans de clipping proche et éloignés -cmin et -cmax. cmin et cmax doivent avoir une valeur positive et respecter  $cmin < cmax$  car il s'agit de distances à l'origine selon l'axe -z.



# Transformations de projection

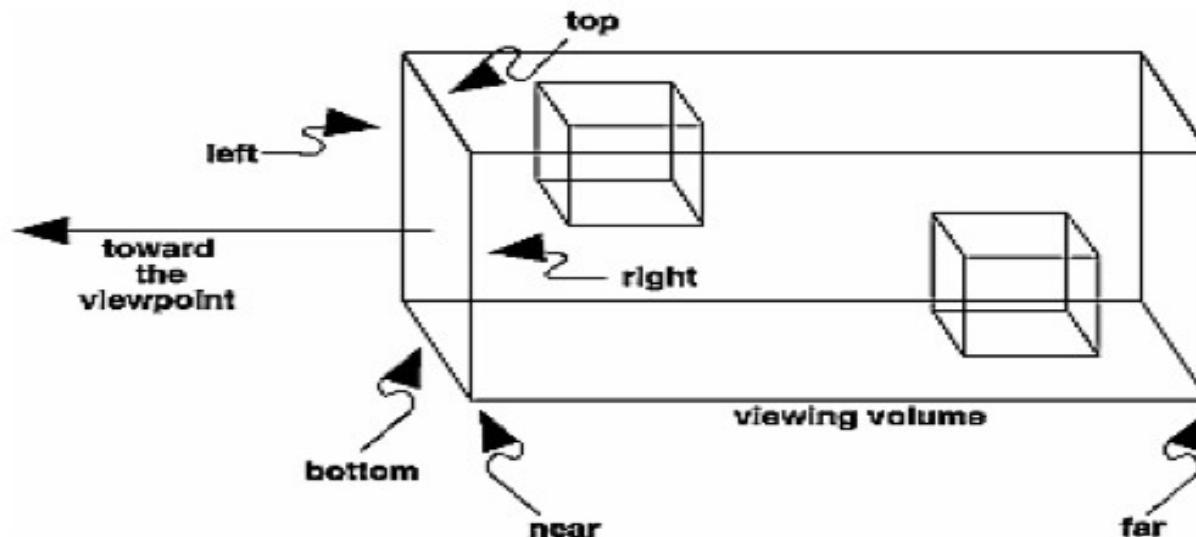
- Exemple:
- `proj = glm::perspective(40.0, 1.5, 50.0, 100.0)`
  - configure une caméra de visualisation en perspective placée à l'origine (c'est obligatoire) et orientée selon l'axe -z (c'est obligatoire) avec:
    - un angle d'ouverture verticale de 40.0°,
    - un angle d'ouverture horizontale de  $40.0 * 1.5 = 60.0$ °,
    - un plan de clipping qui élimine tous les objets ou morceaux d'objet situés en  $z > -50.0$
    - et un plan de clipping qui élimine tous les objets ou morceaux d'objets situés en  $z < -100.0$ .
    - Ces valeurs sont considérées dans le repère global.

# Transformations de projection

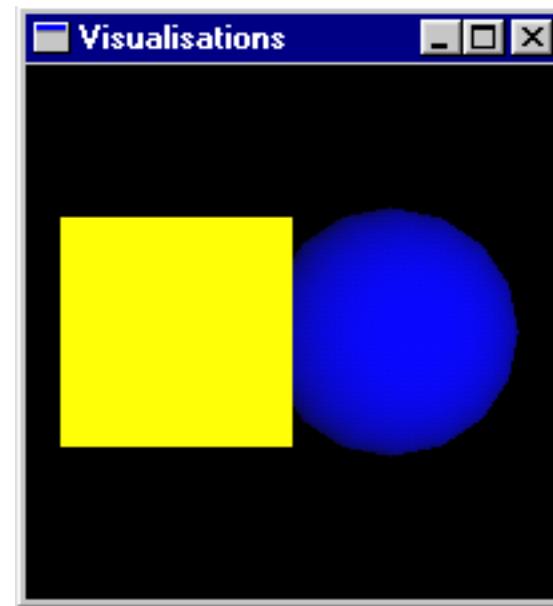
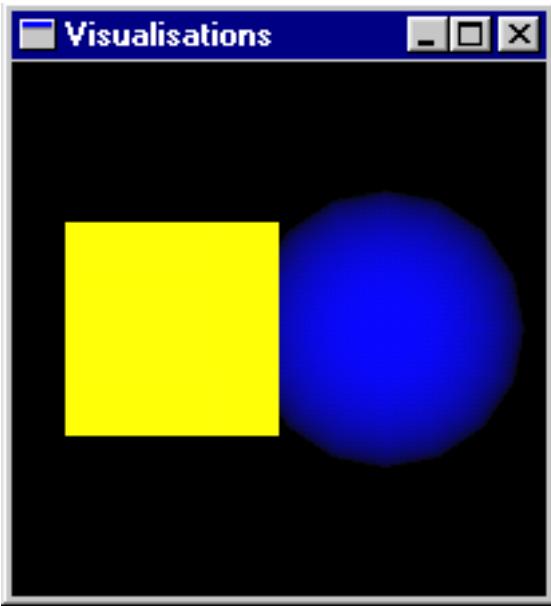
```
proj = glm::ortho(GLdouble g, GLdouble d, GLdouble b,  
GLdouble h, GLdouble cmin, GLdouble cmax);
```

Compose la transformation courante par la transformation de projection orthographique selon l'axe -z et définie par le volume de visualisation parallélépipédique ( $g, d, b, h, -c_{min}, -c_{max}$ ).

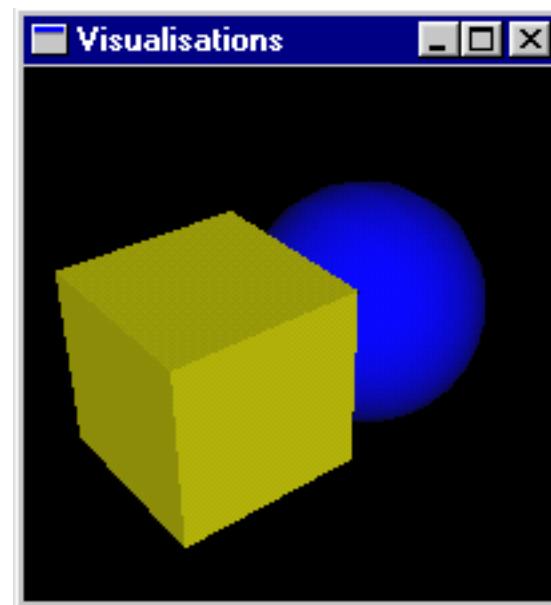
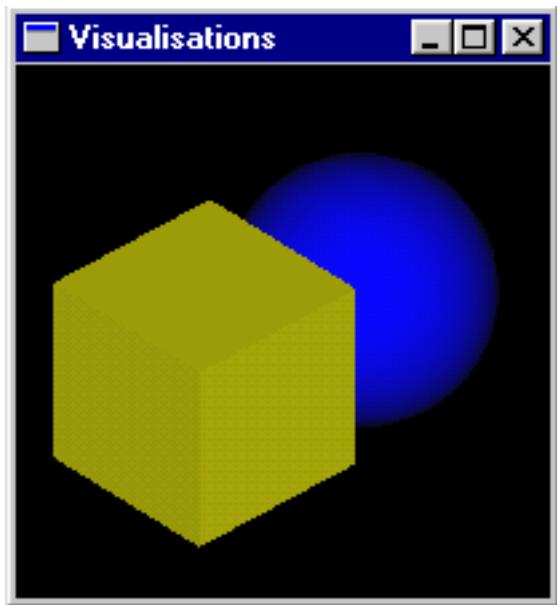
$c_{min}$  et  $c_{max}$  peuvent être positifs ou négatifs et mais doivent respecter  $c_{min} < c_{max}$ .



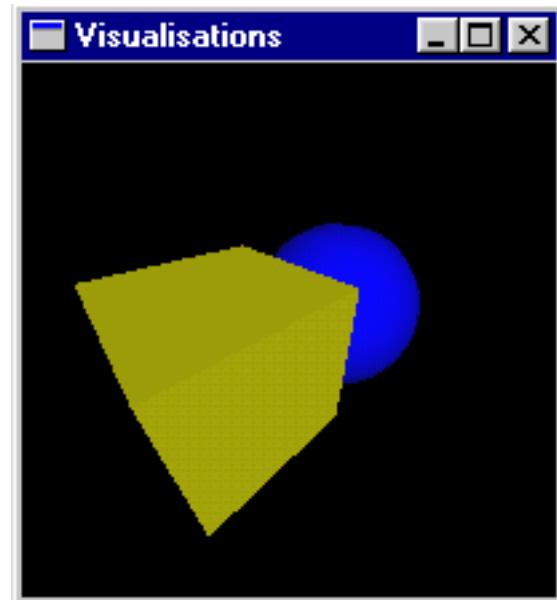
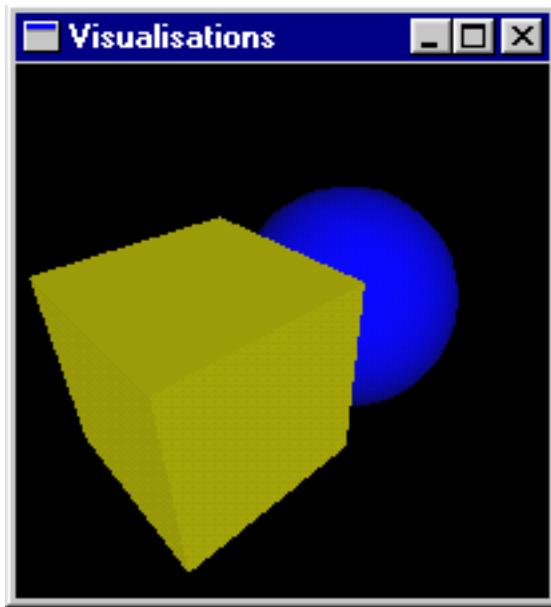
# Visualisation en projection parallèle et projection en perspective



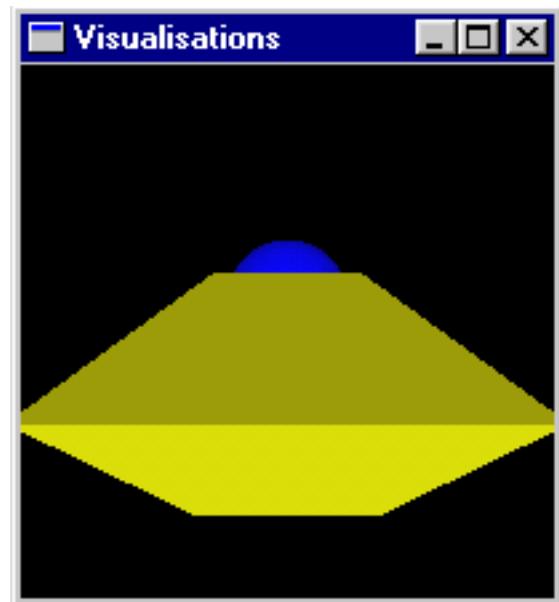
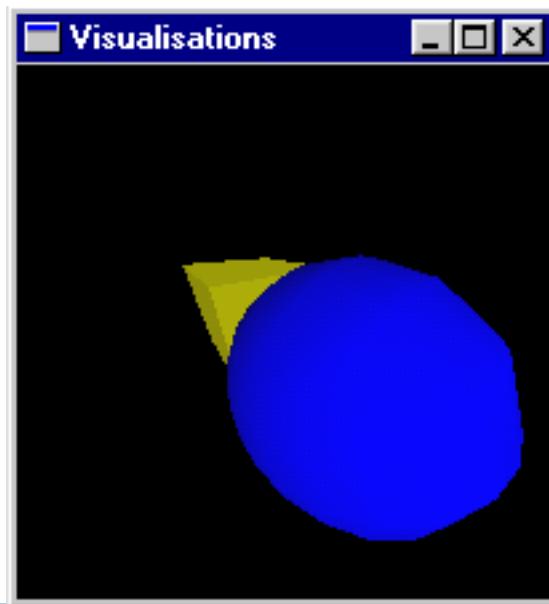
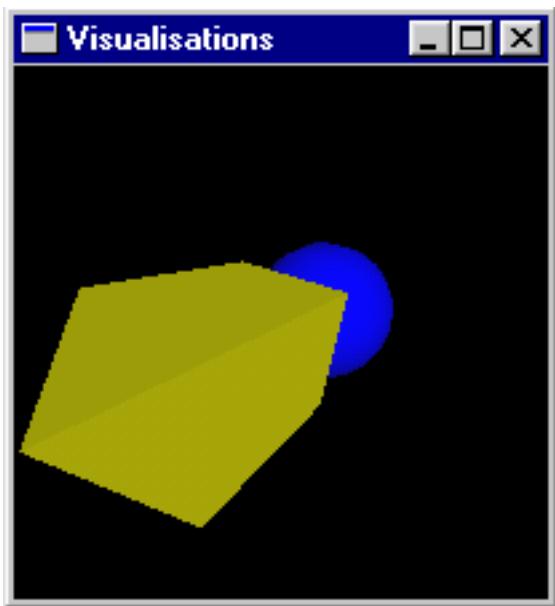
# Visualisation en projection parallèle et projection en perspective



# Visualisation en projection en perspective



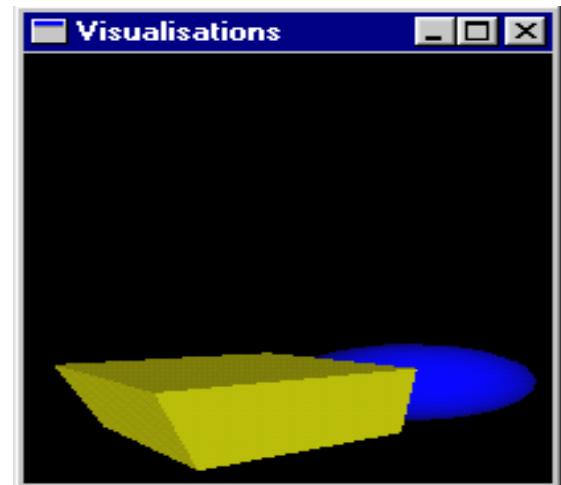
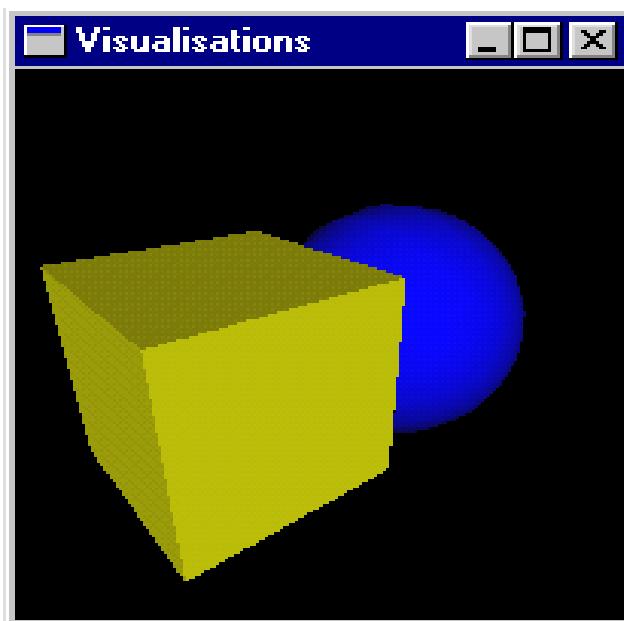
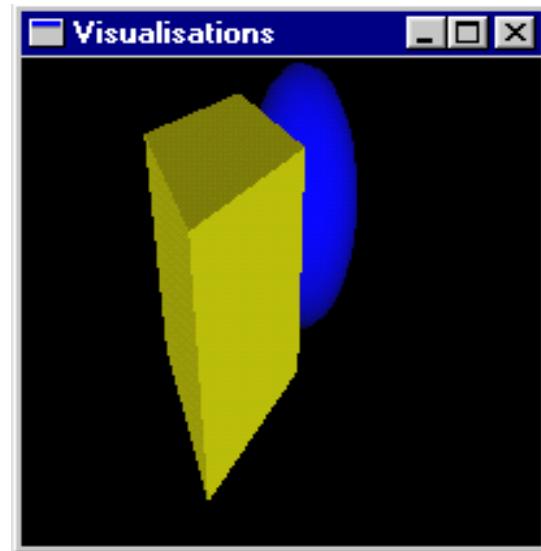
# Visualisation en projection en perspective Avec rapprochement de scène



# Transformation d'affichage

```
void glViewport(GLint x,GLint  
y,GLsizei l,GLsizei h);
```

Définit le rectangle de pixels dans la fenêtre d'affichage dans lequel l'image calculée sera affichée. (position et taille dans la fenêtre)



# Transformation d'affichage

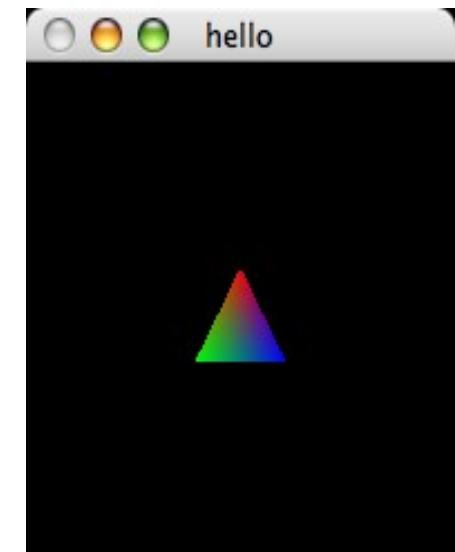
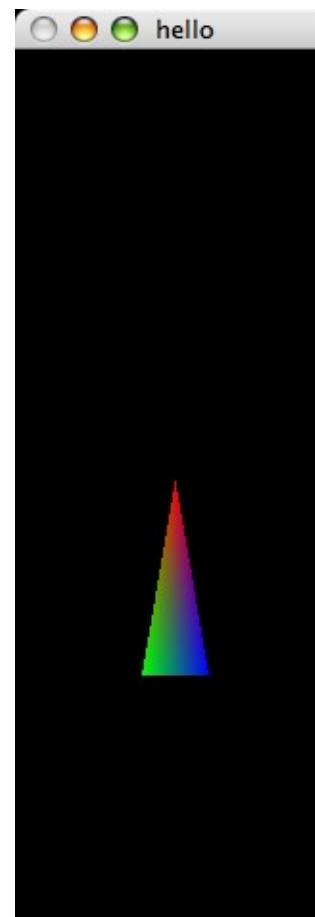
- Si le rapport du volume visualisé n'est pas égal à celui du volume de visualisation une distorsion apparaît ( les carrés deviennent des rectangles et les cercles des ellipses).

- Exemple distordu:

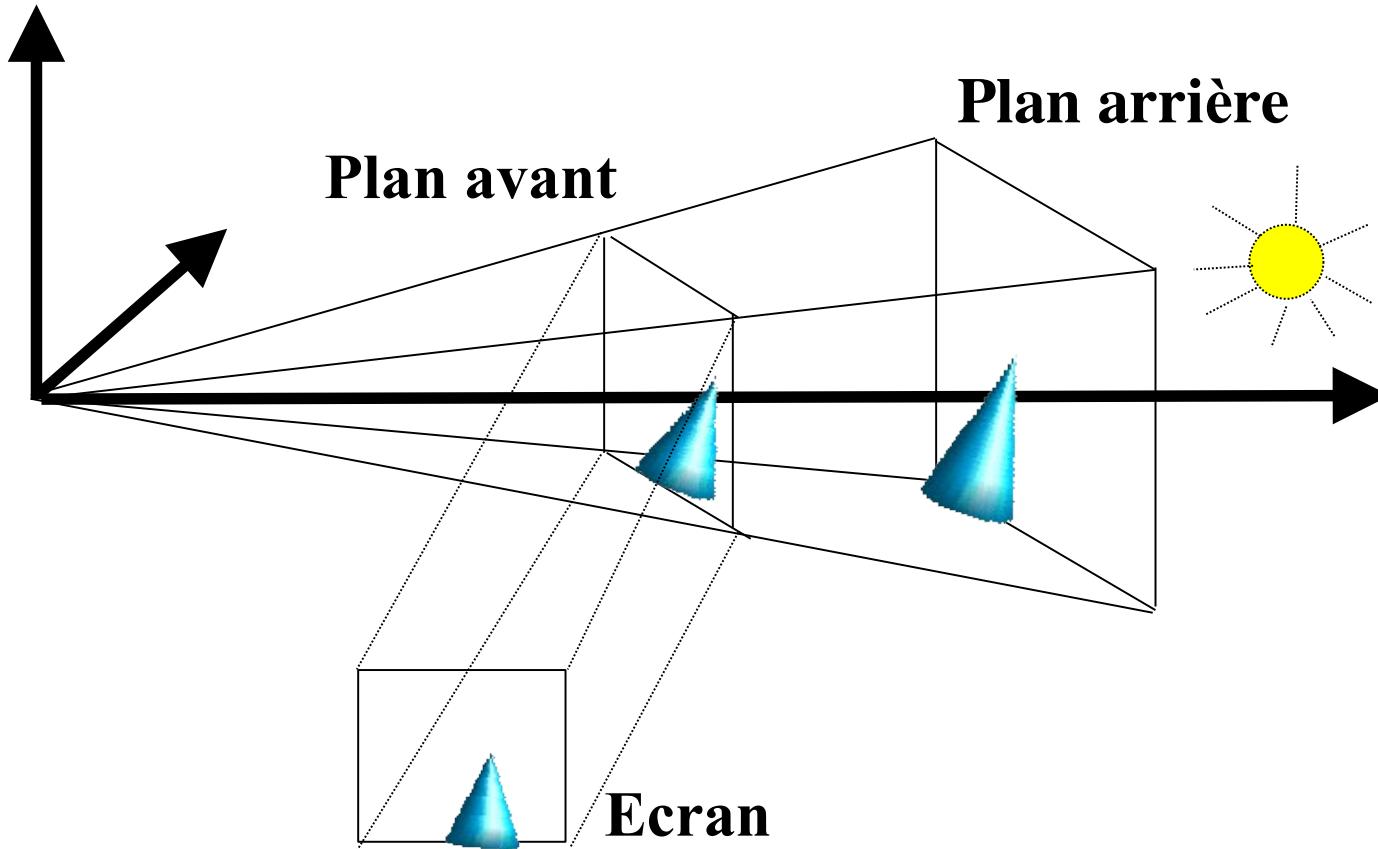
```
→ proj = glm::perspective(fovy,  
    1.0, near, far);  
→ glViewport(0,0,800,400) ;
```

- Exemple non distordu:

```
→ proj = glm::perspective(fovy,  
    1.0, near, far);  
→ glViewport(0,0,400,400) ;
```



# Résumé



# Gérer les matrices

```
mvpSAV=mvp;
```

- Sauve la matrice courante.

```
mvp=mvp*Rot...;
```

...

```
mvp=mvpSAV
```

- Restauration de la matrice modelview.

# Exemple

```
glClear (GL_COLOR_BUFFER_BIT);

SAV1model=model;

model = glm::translate( model, glm::vec3(-
1.0, 0.0, 0.0));

model
=glm::rotate( model,glm::degrees( shoulder)
, glm::vec3( 0.0f, 0.0f, 1.0f ));

model = glm::translate( model,
glm::vec3(1.0, 0.0, 0.0));

SAV2model=model;

model =glm::scale(model,2.0, 0.4, 1.0);

DrawWireCube (model,1.0);

model=SAV2model ;

model = glm::translate( model,
glm::vec3(1.0, 0.0, 0.0));

model
=glm::rotate( model,glm::degrees( elbow),
glm::vec3( 0.0f, 0.0f, 1.0f ));

model = glm::translate( model,
glm::vec3(1.0, 0.0, 0.0));

model =
glm::scale(model,2.0, 0.4,
1.0);

DrawWireCube (model,1.0);

model=SAV1model ;

glutSwapBuffers();
```

