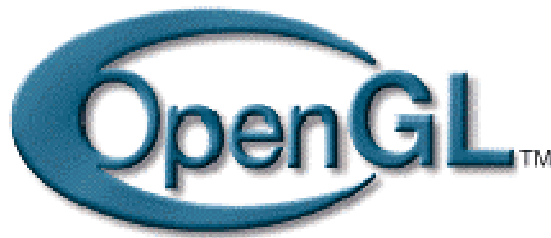


RSX205

## Programmation de la 3D



Alexandre Topol

Spécialité Informatique

Conservatoire National des Arts & Métiers

2009-2010

# Objectifs du cours

---

- Rappels et compléments images de synthèse
- Présenter quelques API existantes
- Expliquer leur principe d'utilisation
- Décrire les fonctionnalités les plus courantes
- Fournir du code de départ, des références, des exemples et le goût de faire de la 3D

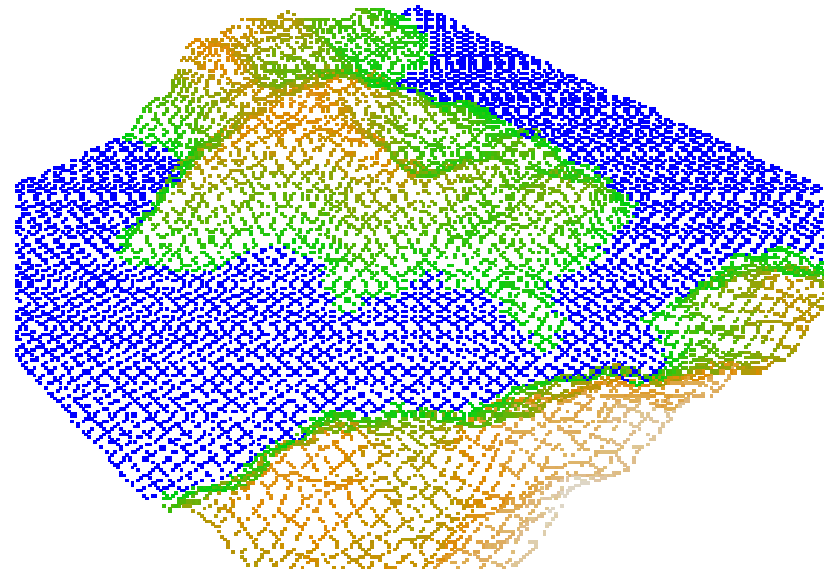
# Description des objets

---

- Elle est quasi identique quelque soit la méthode employée pour les visualiser
- La seule exception est l'utilisation des opérations booléennes (surtout dans le lancé de rayons)
- Les deux étapes :
  - Décrire les objets
    - Par liste de sommets et de faces, par profil, par fractales, ...
  - Composer la scène
    - Par positionnement des objets les uns par rapport aux autres

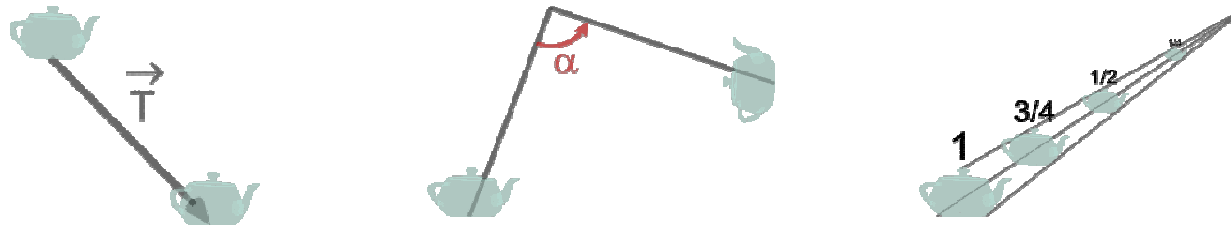
# Description des objets

- En numérisant des objets
  - En modelant l'objet
  - Par fractales
- 
- Donner l'ensemble des géométries constituant l'objet

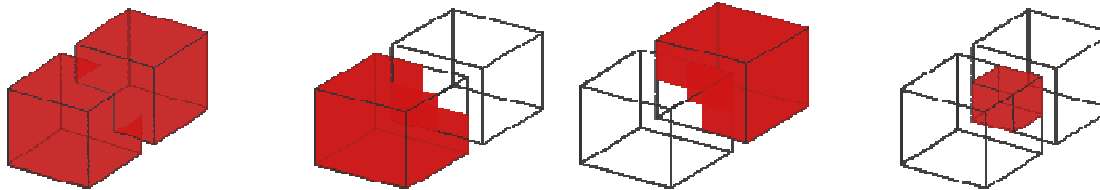


# Composition de la scène

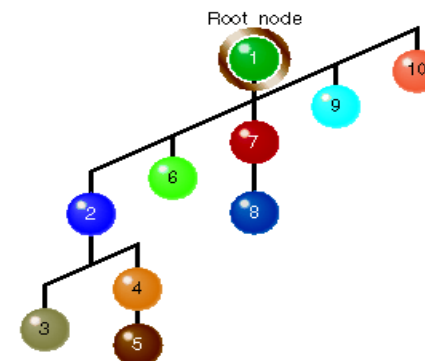
## ■ Par transformations géométriques



## ■ Par opérations booléennes



## ■ Dans un graphe de scène



# Visualisation des scènes : deux méthodes

---

## ■ Projection de facettes :

- Les objets doivent être "triangulés" (*tessalated*)
- Et tout triangle ou polygone subit les opérations du pipeline 3D
- Transformations, visibilité, *clipping*, projection, coloriage

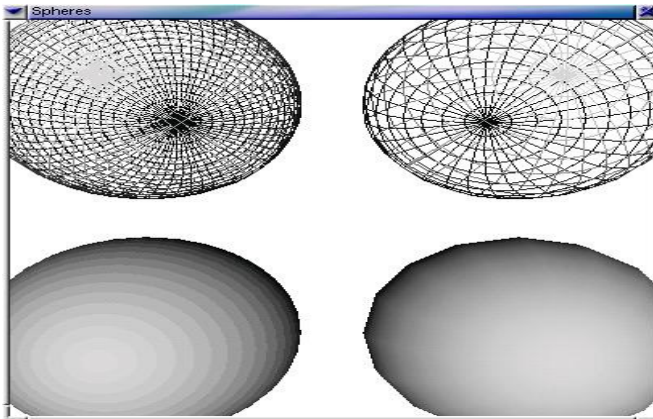
## ■ Lancé de rayons :

- On utilise ici les équations de surface des objets donc pas besoin de "trianguliser" les objets
- Calcul pour chaque pixel de l'écran de l'équation partant de l'observateur et passant par ce pixel
- Pour chaque rayon, calcul des intersections avec les objets pour déterminer la couleur du pixel

# Visualisation des scènes : deux méthodes

## Par facettage

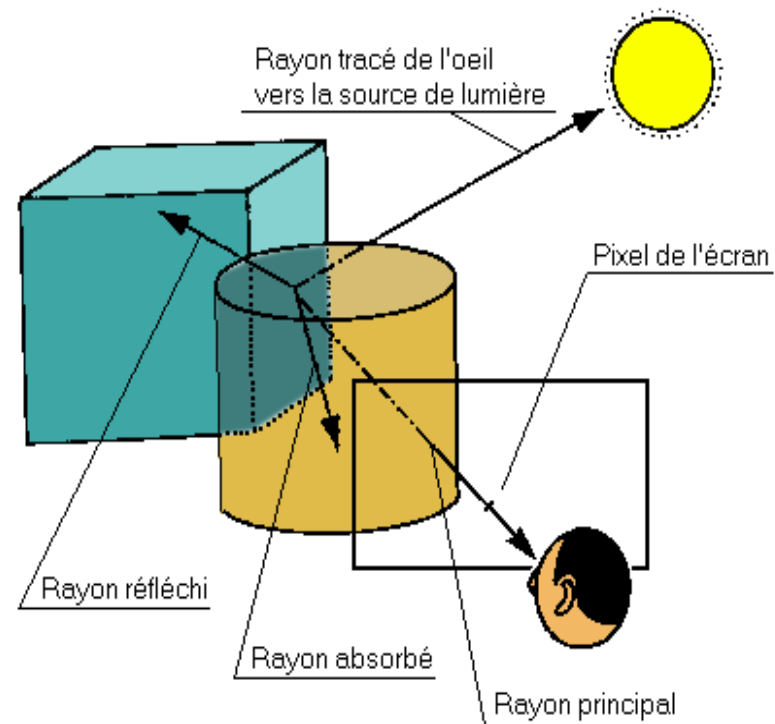
1. Transformation des formes complexes en ensemble de triangles



2. Transformation, illumination, projection et coloriage de chacun des triangles → pipeline 3D

« quantité plutôt que qualité »

## Raytracing



« qualité plutôt que quantité »

## Visualisation des scènes : deux méthodes

Par facettage



© Activision

« quantité plutôt que qualité »

Raytracing



© Dreamworks

« qualité plutôt que quantité »



# Visualisation des scènes

## Projection de facettes

---

### ■ L'algorithme :

Initialisation()

Chargement de la scène

Pour chaque objet Faire

    Décomposer en polygones (triangles) si nécessaire

    (si l'objet n'est pas déjà constitué de faces)

Fin Pour

Rendu()

    Initialiser Z-Buffer et FrameBuffer (l'image)

    Pour chaque triangle Faire

        1. Transformations - se placer dans le repère de l'observateur

        2. Test de visibilité - rejet des faces non visibles

        3. Clipping - rejet des parties du hors du volume de vue

        4. Calcul de l'intensité lumineuse

        5. Projection

        6. Z-Buffer et Coloriage

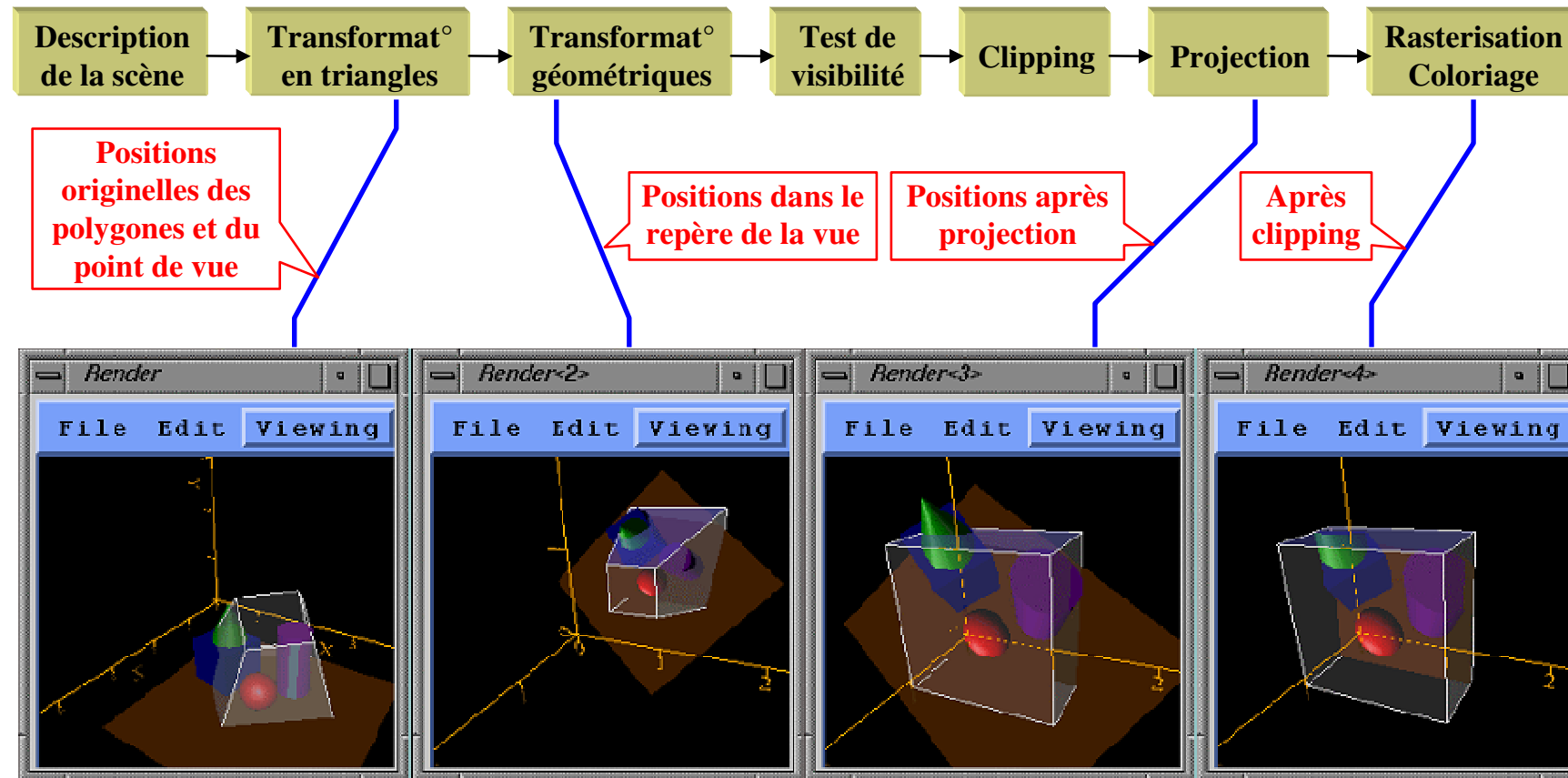
    Fin Pour

Afficher l'image

# Visualisation des scènes

## Projection de facettes

### ■ Pipeline 3D (FFP)



# Visualisation des scènes

## Lancé de rayons

---

### ■ L'algorithme (de base) :

```
Pour chaque pixel de l'écran Faire
  Calculer l'équation de la droite (point de vue, pixel)
  maxlocal := -oo
  Pour chaque objet de la scène Faire
    pz = intersection de valeur Z minimum entre le rayon et l'objet
    Si pz existe et pz >= maxlocal Alors
      maxlocal = pz
      pixel (i,j) = couleur de la face
    Fin Si
  Fin Pour
Fin Pour
```

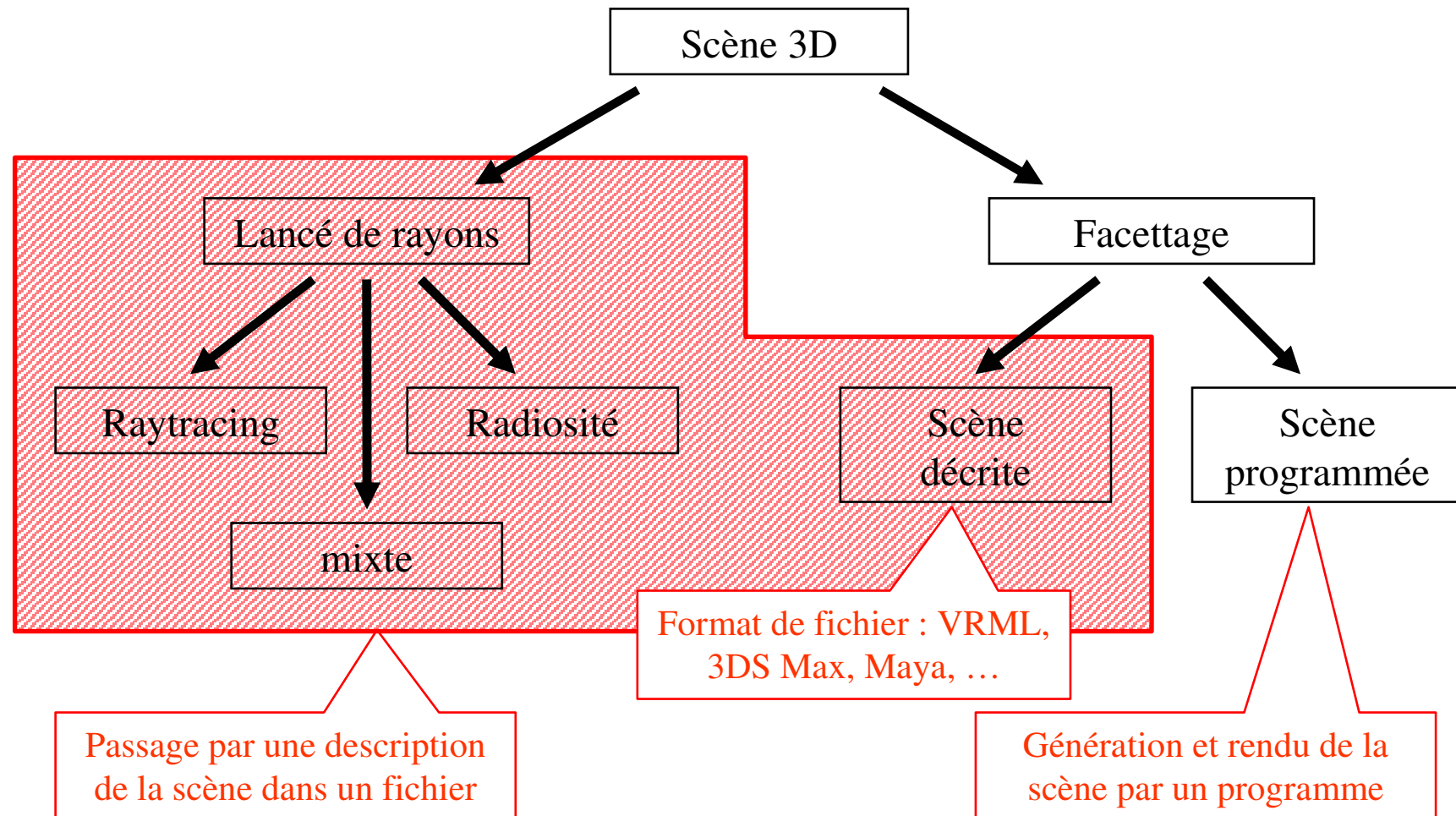
Exemple

# Accès à ces méthodes de rendu 3D décrite ou programmée ?

---

- 2 manières de mettre en œuvre ces 2 méthodes :
  - La 3D programmée : utilisation de fonctions dans un programme pour donner à la fois :
    - Les caractéristiques des objets
    - L'ordre de rendu
  - La 3D décrite :
    - Spécification de la scène dans un fichier
    - Interprétation et rendu de la scène par un programme spécialisé
    - Choix d'un format cible (VRML, POV, 3DS ...)

# La 3D décrite ou programmée ?



# La 3D décrite ou programmée ?

## La 3D programmée

---

- Utilisation d'une API 3D : OpenGL, Direct3D, Java3D, ...
- Avantages :
  - Accélération graphique
  - Contrôle sur le rendu - application "sur mesure"
- Inconvénients :
  - Nécessite la connaissance d'un langage de programmation<sup>o</sup>
  - Difficulté de maintenance
  - Pauvre "réutilisabilité"
  - Temps de développement

# La 3D décrite ou programmée ?

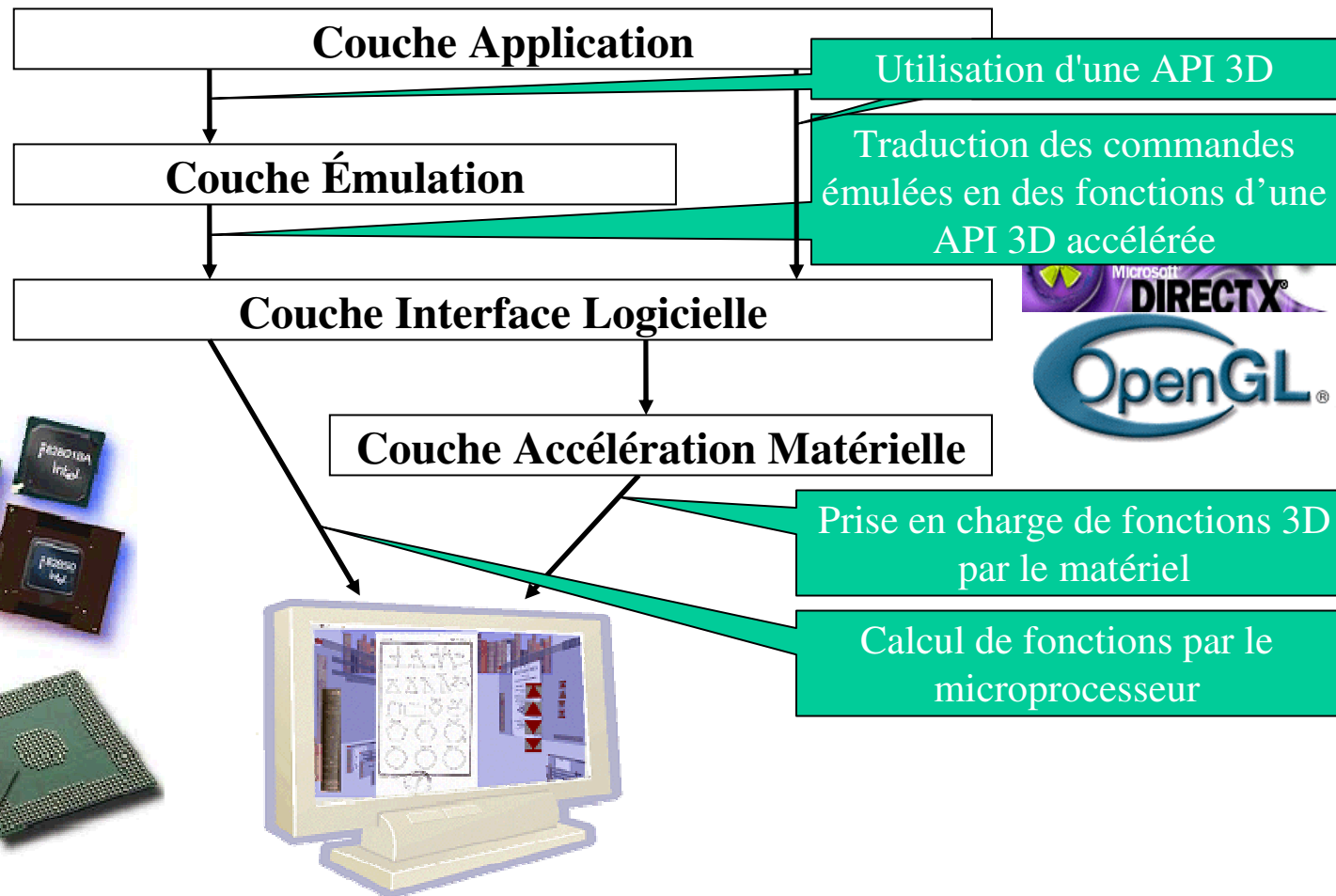
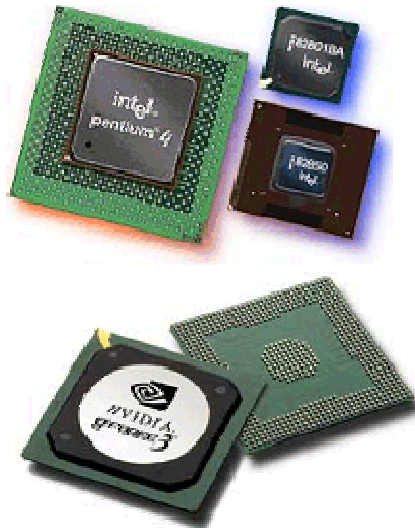
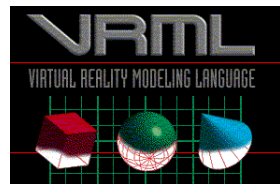
## La 3D décrite

---

- Utilisation d'un langage de description et d'un programme ou plugin associé : VRML, Collada, ...
- Avantages :
  - Bonne description de l'apparence des objets
  - Outils pour la création, l'édition et le rendu
  - Diffusion sur internet – génération par scripts CGI
  - Facilité pour la maintenance
  - "Réutilisabilité" - partage
- Inconvénients :
  - Pauvre description des interactions
  - N'inclut pas les dernières techniques réalistes

# La 3D décrite ou programmée ?

## Les couches d'une architecture 3D





# La 3D décrite

## Lancé de rayons

---

- Des modeleurs (3DS Max, Maya, Blender, ...)
  - Construction des scènes par clics de souris
  - Pré-visualisation en 3D par facettage
  - Calcul de l'image ou de l'animation finale par lancer de rayons
- POV (*Persistence Of Vision*)
  - Construction des scènes par langage
  - Pas de pré-visualisation
  - Mais ... gratuit et sources disponibles

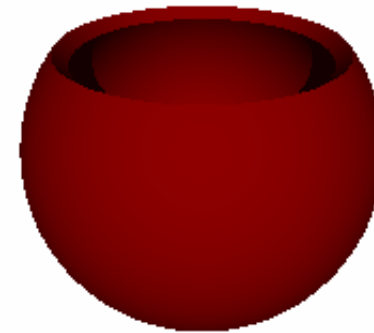
<http://www.povray.org>

# La 3D décrite

## Lancé de rayons – POV

- Tout y est et bien plus ...
  - Formes élémentaires : polygon, sphere, cone, torus ...
  - Formes complexes : splines, surfaces d'élévation, ...
  - Opérations booléennes : union, difference, intersection
  - Modèles de coloriage : flat, phong, bump mapping, ...

```
camera {  
    location <0.0, 0.0, -3.0>  
    look_at <0.0, 0.0, 0.0>  
}  
  
light_source { <0, 0, -100> rgb <1.0, 1.0, 1.0>  
  
background { rgb <1.0, 1.0, 1.0> }  
  
difference {  
    difference {  
        sphere { <0, 0, 0>, 1 }  
        sphere { <0, 0, 0>, 0.9 }  
    }  
    cone { <0, 1.1, 0>, 1.3, <0, 0, 0>, 0 }  
    pigment { rgb <0.8, 0.0, 0.0> }  
    rotate <-30, 0, 0>  
}
```



# La 3D décrite

## 3D par facettage – VRML, X3D, Collada

---

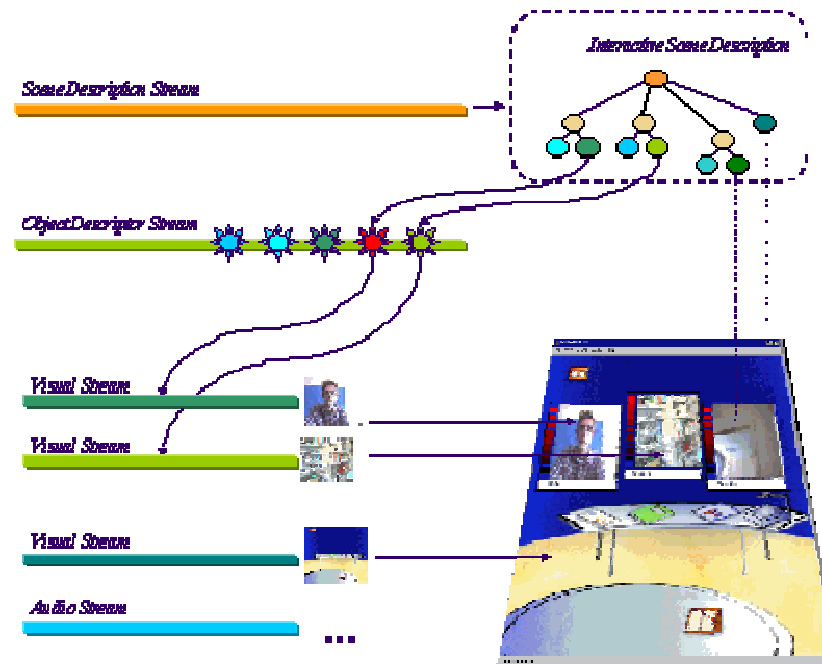
- VRML est un format de fichier et non pas un langage !
- Un *plugin* associé permet de les visualiser
- Organisation hiérarchique d'objets contenant des attributs  
Graphe de scène      Noeuds      Champs

Les nœuds fils et leur descendance subissent tous le comportement défini par les attributs de leur nœud père  
⇒ cascade de comportements

# La 3D décrite

## 3D par facettage – MPEG4

### MPEG-4 Systems Principles



# La 3D programmée

## Interface de programmation (API) 3D

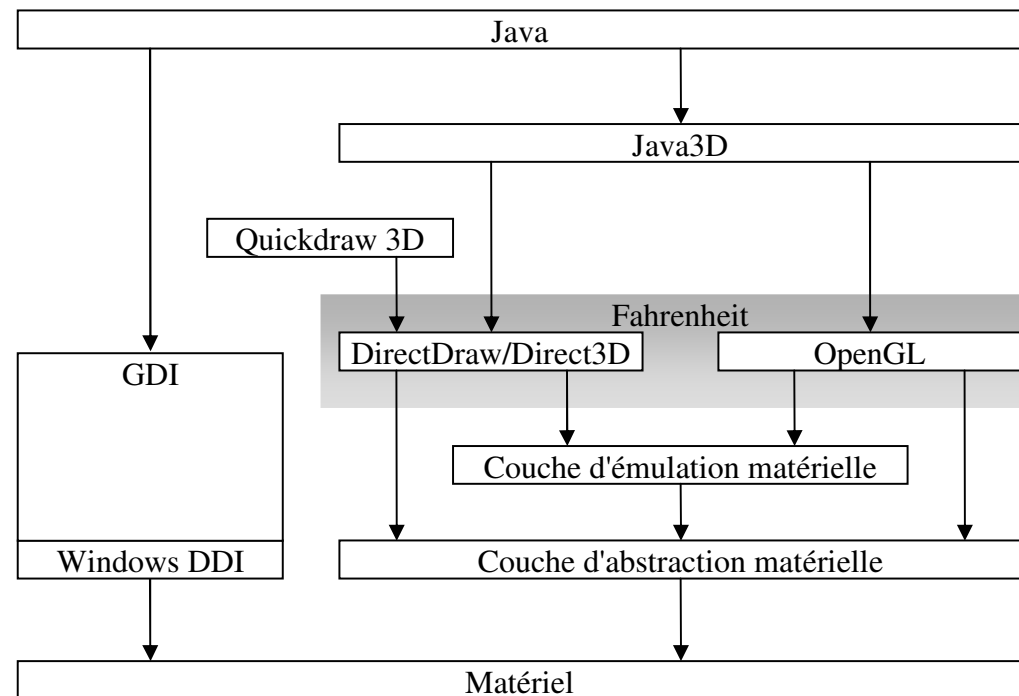
---

- Permet d'afficher l'image d'une scène 3D
- Interface avec un langage (C++, Java, Ada...)
- Différentes représentations de la scène
  - Primitives de plus ou moins haut niveau
  - "3 théières alignées" ... "un pixel bleu en (123,456)"
- Différentes architectures cibles
- Plus ou moins proche du hardware

# La 3D programmée

## Les APIs 3D

- Gestion du pipeline 3D par appels de fonctions
- Utilisation des drivers de la carte 3D pour accélérer le rendu



# Retour à l'algorithme de base

## ■ L'algorithme :

Initialisation()

Chargement de la scène

Pour chaque objet Faire

    Décomposer en polygones (**triangles**) si nécessaire

    (si l'objet n'est pas déjà constitué de faces)

Fin Pour

Rendu()

    Initialiser **Z-Buffer** et **FrameBuffer** (l'image)

    Pour chaque triangle Faire

        1. Transformations - se placer dans le repère de l'observateur

        2. **Test de visibilité** - rejet des faces non visibles

        3. **Cloturage** - rejet des parties du hors du volume de vue

        4. Calcul de l'intensité lumineuse

        5. **Projection**

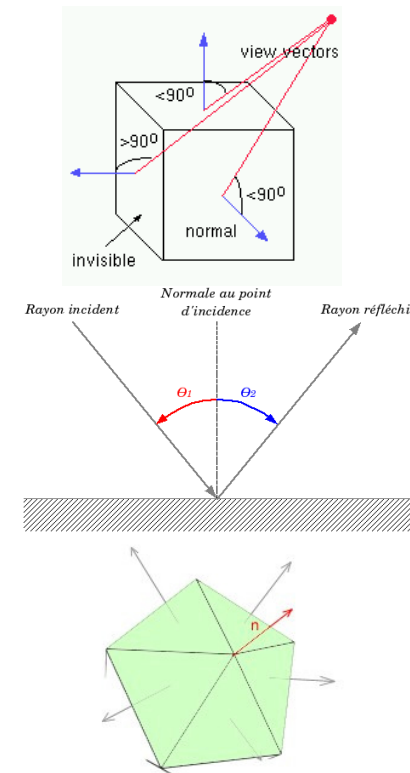
        6. **Z-Buffer** et **Coloriage**

    Fin Pour

Afficher l'image

# Pourquoi des triangles ?

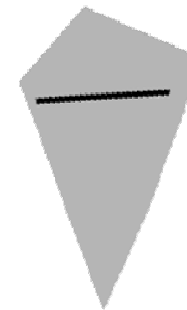
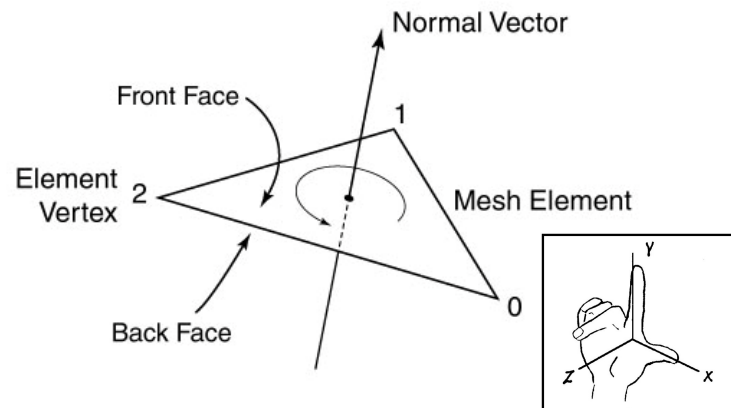
- On a besoin des normales aux faces pour pas mal de calculs :
  - détermination des faces vues de dernière (backface culling)
  - d'illumination des faces (basés sur la loi de Descartes)
  - les calculs des normales aux sommets pour le coloriage Gouraud ou Phong



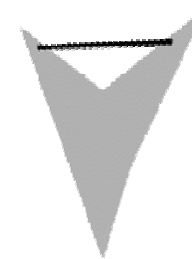


## Pourquoi des triangles ?

- Or, les sommets d'un triangle sont nécessairement coplanaires
  - donc une seule normale pour tout point sur la face
  - les autres polygones peuvent être vrillés
- De plus, un triangle est nécessairement convexe  
=> plus facile à colorier



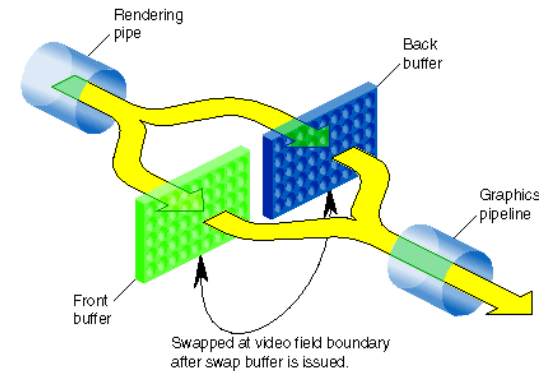
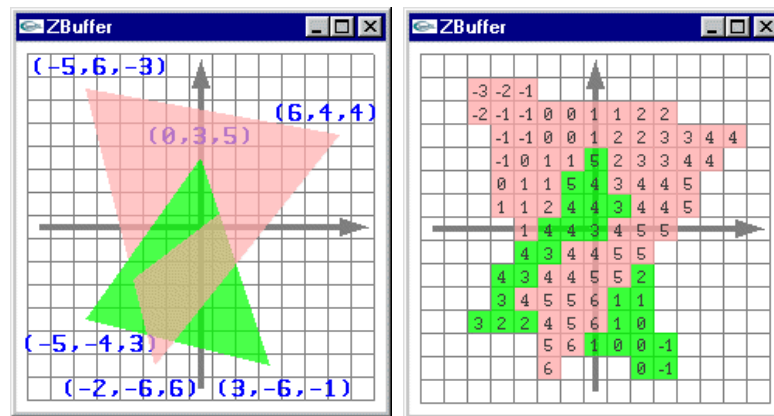
convexe



concave

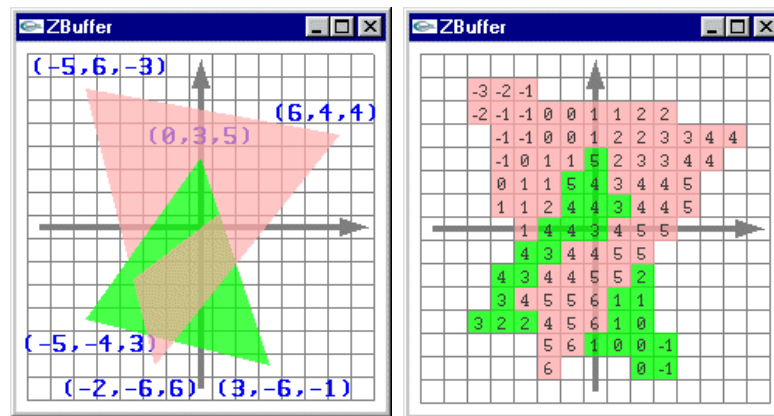
# FrameBuffer et ZBuffer

- Les deux tableaux de valeurs les plus importants pour le calcul d'une image
  - Framebuffer = valeur des pixels à afficher
    - En pratique 2 framebuffers (celui affiché et celui dans lequel on calcule l'image suivante)
  - ZBuffer = profondeur de ces pixels



# FrameBuffer et ZBuffer

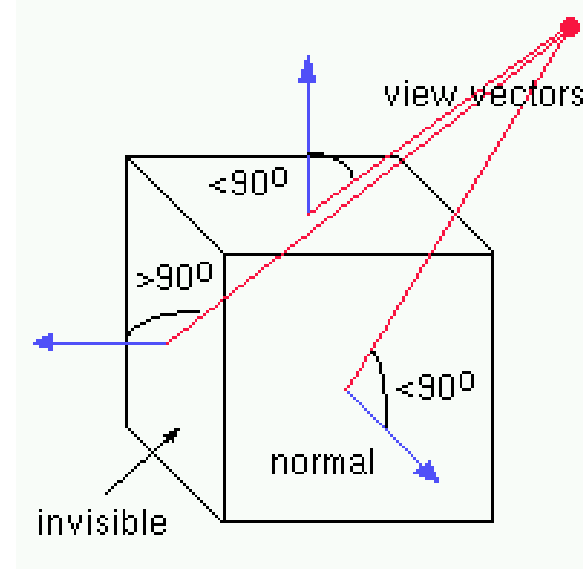
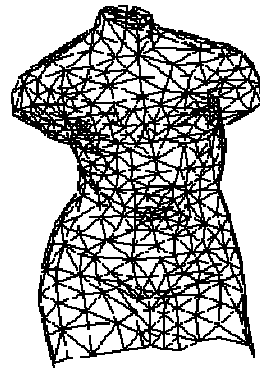
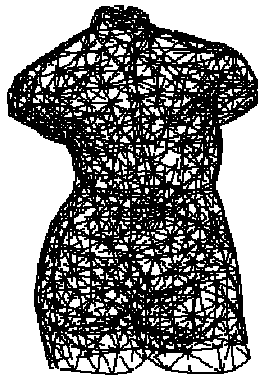
- Le ZBuffer permet l'élimination des faces cachées par d'autres (ce que ne fait pas le backface culling)
  - Après projection d'un nouveau triangle, on regarde dans le ZBuffer pour chacun de ces pixels (par balayage) si ils sont plus proches de l'observateur que les pixels précédemment calculés.
    - Si oui, on stocke les valeurs de pixel et de profondeur dans les buffers
    - Sinon, on ne fait rien.



Exemple

## Test de visibilité – *Backface Culling*

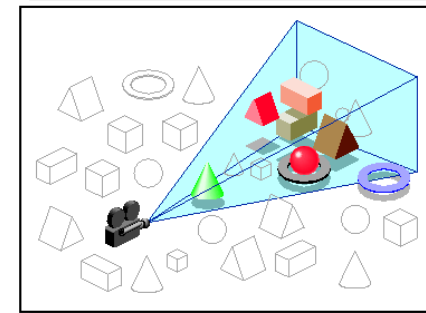
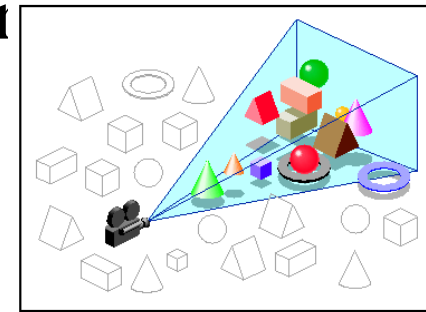
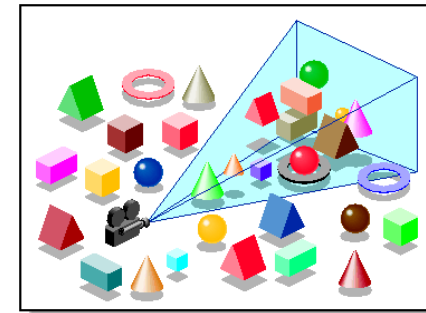
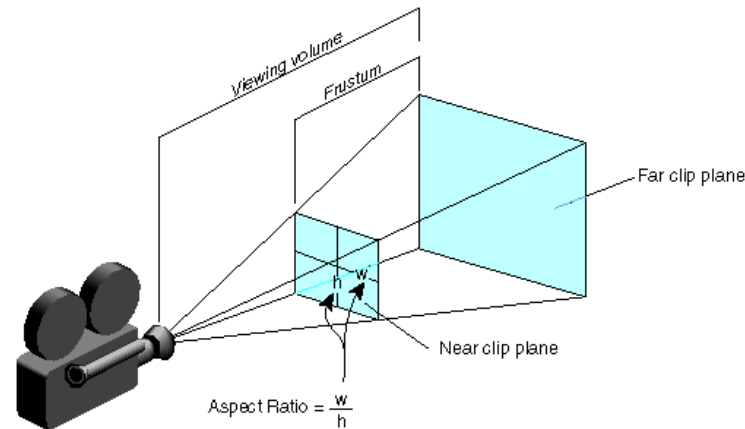
- Étape optionnelle
  - Élimination du pipeline 3D des faces vues de derrière :
    - Si la face est dirigée vers l'observateur (la caméra) on la garde
    - Sinon, on la rejette et cette face n'est pas projetée
- ➔ Utile pour les mondes clos,  
pour les objets transparents  
ou les objets fil de fer



# Cloturage – *Clipping*

## ■ Volume de vue et *View Frustum*

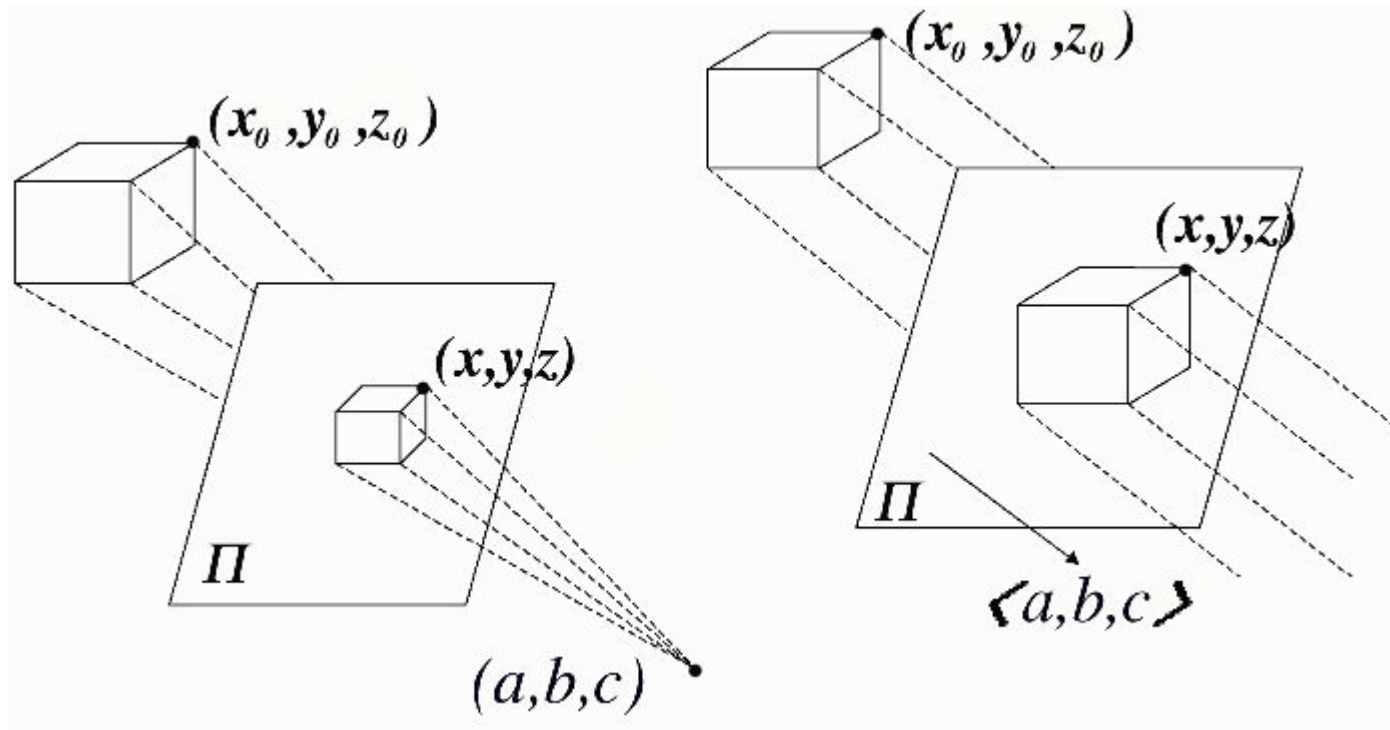
C'est le volume donnant la portion de l'espace dans laquelle les objets sont visibles (en fonction de la caméra)



Effets combinés du clipping et du ZBuffer

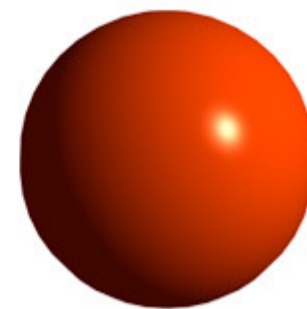
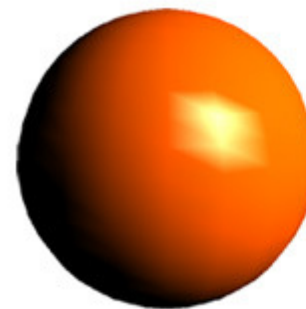
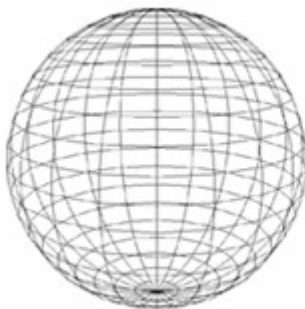
# Projection

- Calcul de la position d'un point 3D sur l'écran 2D simulant l'image résultante



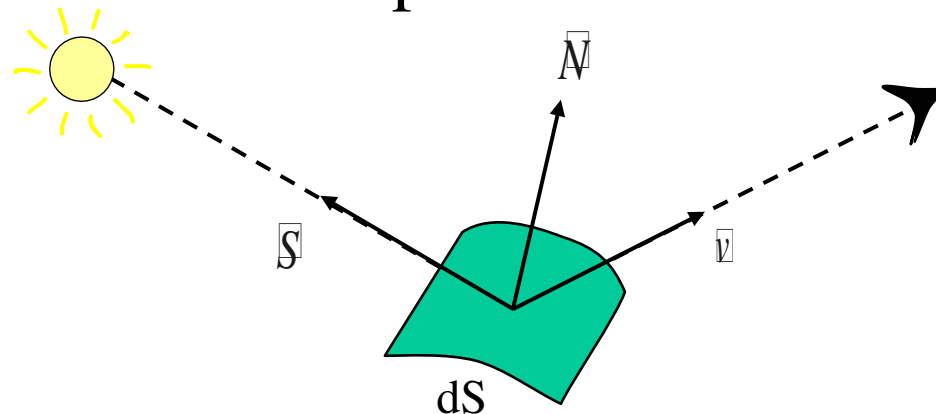
## Coloriage – *Rasterization*

- Balayage de la face projetée en vue de calculer une couleur pour chacun de leur pixel visible (qui réussit le test du ZBuffer)
- Beaucoup de techniques de coloriages
- Les plus simples : wireframe, flat shading, gouraud shading et phong shading



# Lumière et réflexion

- Interaction lumière / objet
  - une partie de la puissance lumineuse reçue est absorbée par l'objet et convertie en chaleur
  - une partie est réfléchie par la surface
  - le reste est transmis à l'intérieur de l'objet (réfraction)
- Première approximation en 3D : la lumière ambiante est le produit des multiples réflexions dans la scène

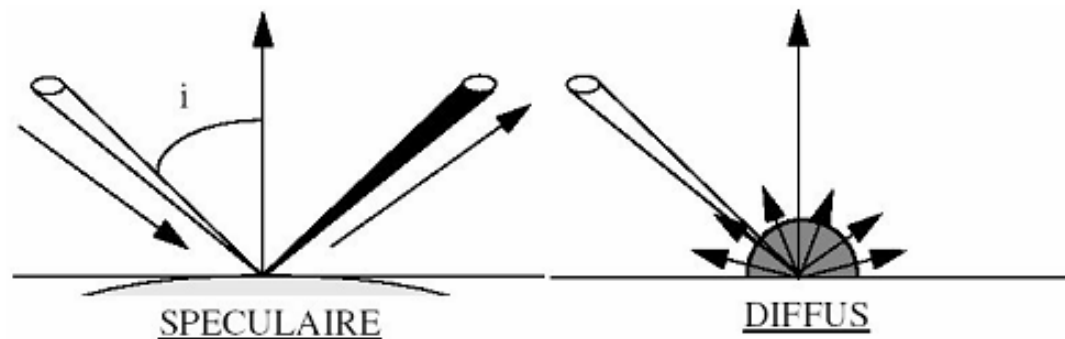




# Lumière et réflexion

## ■ Deux types de réflexion :

- Diffuse : La surface de l'objet "réagit". La lumière est ré-émise dans toutes les directions. Sa couleur est affectée.
- Spéculaire : La surface de l'objet ne "réagit" pas. La lumière est ré-émise selon l'angle d'incidence. Sa couleur n'est pas affectée.



Exemple

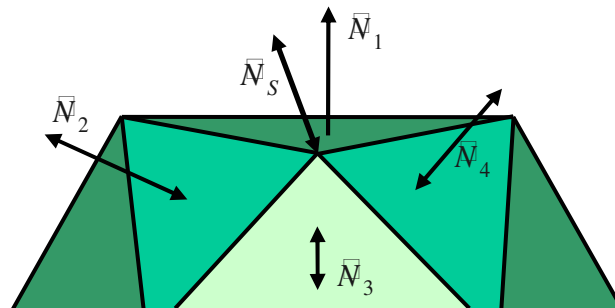
## Le coloriage à plat (*flat shading*)

- Bouknight 1970
- On utilise la loi de Lambert. L'intensité réfléchie est calculée pour chaque face et supposée constante pour toute la surface du polygone
- La face est plus ou moins éclairée en fonction de son orientation par rapport aux sources lumineuses
- L'approximation est valide si
  - la source est infiniment loin
  - l'observateur aussi
  - la surface n'est pas réfléchissante



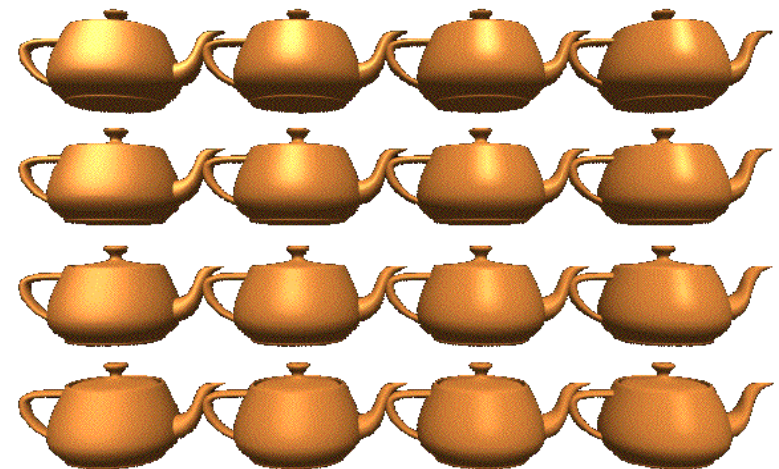
## Gouraud (*smooth shading*)

- Principe : interpolation linéaire de l'intensité pour éliminer les discontinuités visibles de face en face
- Suppose la connaissance de l'intensité lumineuse des sommets **donc** de leur orientation par rapport à la lumière **donc** de leur normale



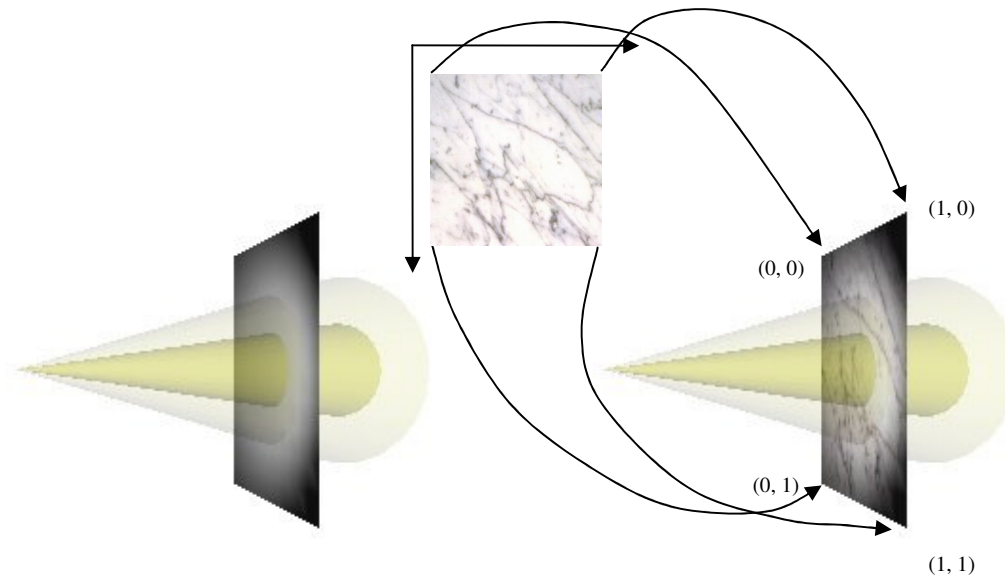
## Modèle de Phong Bui-Tuong

- *Phong Shading* (1975), le principe :
  - Calcul d'une réflexion spéculaire en chaque point de la face, variable selon le matériau
  - Donne un objet avec une réflexion spéculaire mais pas entièrement, d'où l'aspect de brillance locale



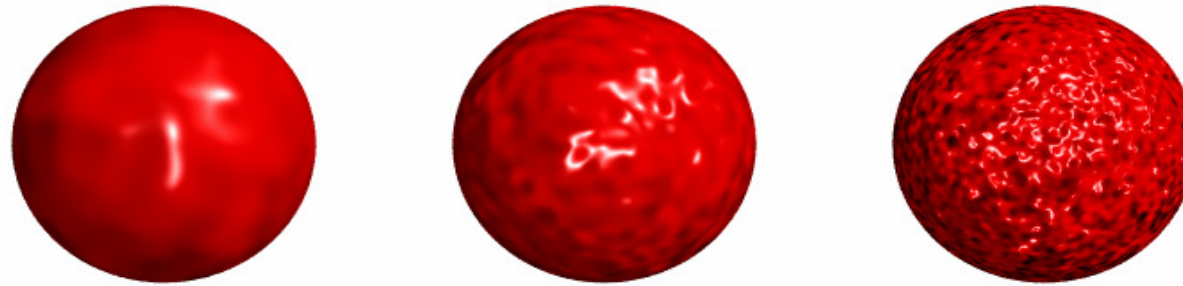
## Placage de textures

- Les algorithmes précédents ne retirent pas complètement le caractère "artificiel" du coloriage.
- Une solution souvent employée : le plaqué de textures sur les faces

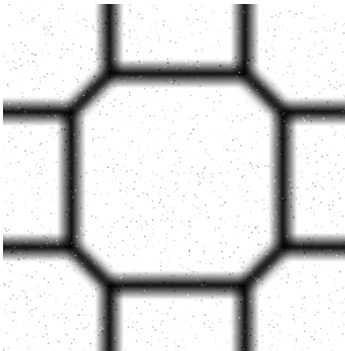


# Bump mapping

## ■ Perturbation aléatoire



## ■ Utilisation d'une texture de bump



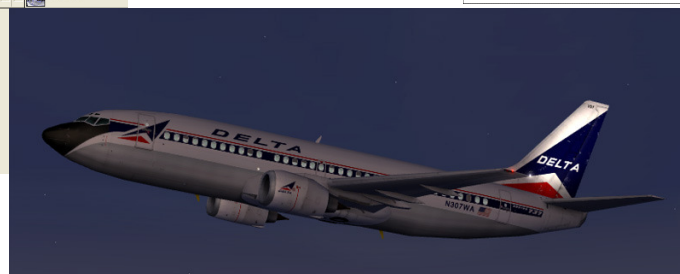
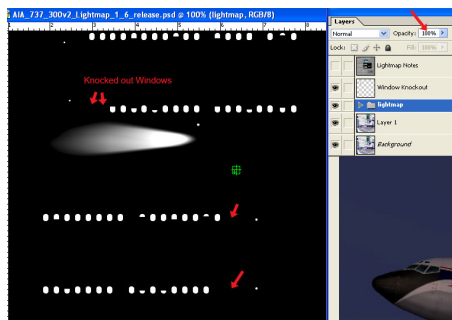
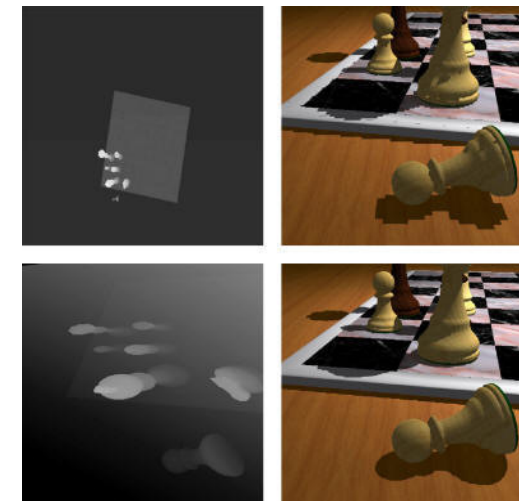
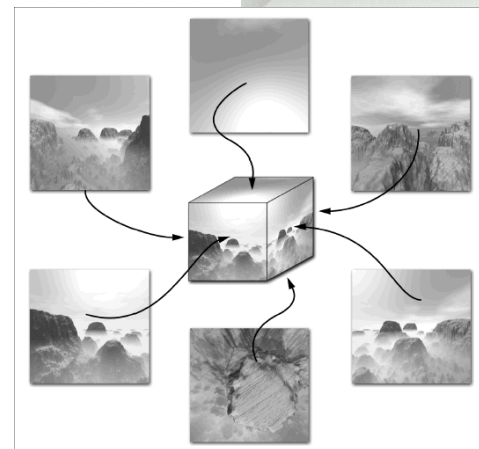
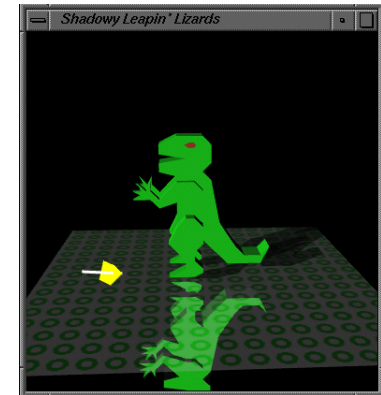
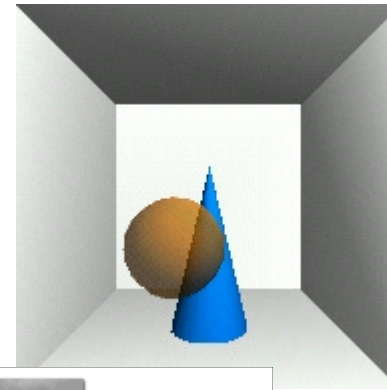
# Autres techniques de coloriage employant les textures

## ■ Par rendu multi passes :

- La transparence
- La réflexion sur un sol
- Les ombres dynamiques

## ■ Par multitexturage :

- Lumières fixes
- Ombres fixes
- Environment mapping





# Et pour aller plus loin ...

## Les shaders !!



*Doom III*



*Halo 2*



*Jet Set Radio Future*

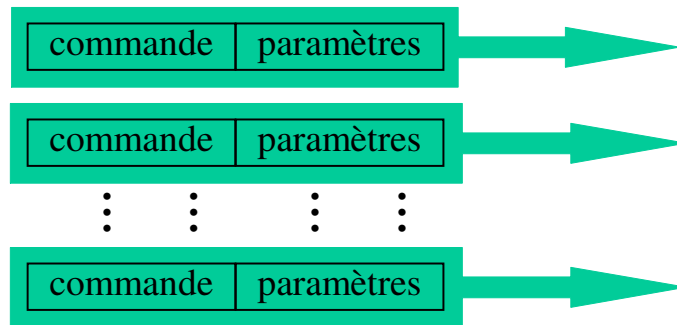


# API 3D

## ■ Deux modes de transmission des données à l'API

### Mode immédiat

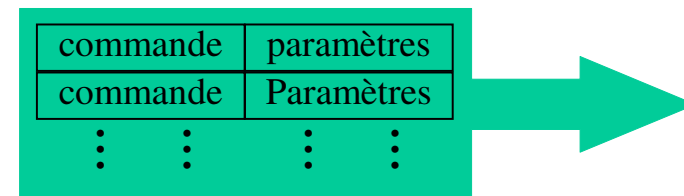
- Transmission **immédiate** des ordres à l'API par appels de fonctions



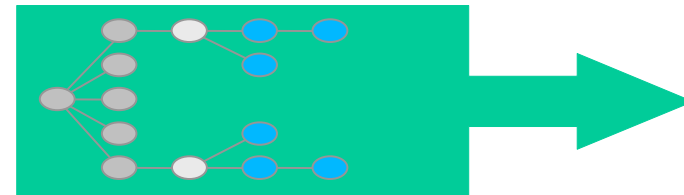
### Mode retenu

- Utilisation d'une structure **retenant** les ordres 3D **puis** transmission de cette structure à l'API
- Types de structures :

- buffer d'exécution, liste d'appels



- graphe de scène



# API 3D

## OpenGL/Mesa

---

- Librairie graphique 2D/3D multi-plateforme
- Bibliothèques d'outils pour deux aspects :
  - La gestion des événements et des fenêtres :
    - GLX permet de relier une application OpenGL avec Xwindow
    - GLUT gère (par procédures de type callback) les événements et le rendu dans une fenêtre
  - Des primitives graphiques 3D de plus haut niveau :
    - Dans GLUT : sphères, cubes, cylindres, cônes ...
    - Dans GLU : tassellation, splines, ...
    - Dans OpenInventor : structuration sous forme de graphe de scène

# La 3D programmée

## OpenGL/Mesa

### ■ Un squelette d'application en C (mode immédiat)

```
#include <GL/gl.h> /* Déclaration des fonctions d'OpenGL */
#include <GL/glut.h> /* Déclaration des fonctions de GLUT */

void init(void) {
    glClearColor(0.0, 0.0, 0.0, 0.0); /* La couleur d'effacement */
    glShadeModel(GL_FLAT); /* Le style de rendu */
}

void display(void) { /* Fonction dessinant chaque trame */
    /* effacement du frame-buffer et du Z-buffer */
    glClear(GL_COLOR_BUFFER_BIT, GL_CLEAR_DEPTH_BUFFER);
    glColor3f(1.0, 1.0, 1.0); /* La couleur pour dessiner */
    /* Initialisation de la matrice de transformation */
    glLoadIdentity();
    /* Transformation des objets par rapport au point de vue */
    glTranslatef(0.0, 0.0, 3);
    /* dessin des objets */
    glBegin(GL_POLYGON);
    glVertexf(0.25, 0.25, 0.0);
    glVertexf(0.75, 0.25, 0.0);
    glVertexf(0.75, 0.75, 0.0);
    glVertexf(0.25, 0.75, 0.0);
    glEnd();
    glutSolidSphere(1.0, 16, 16);
    glSwapBuffers(); /* Affichage de l'image calculée */
}
```

# La 3D programmée

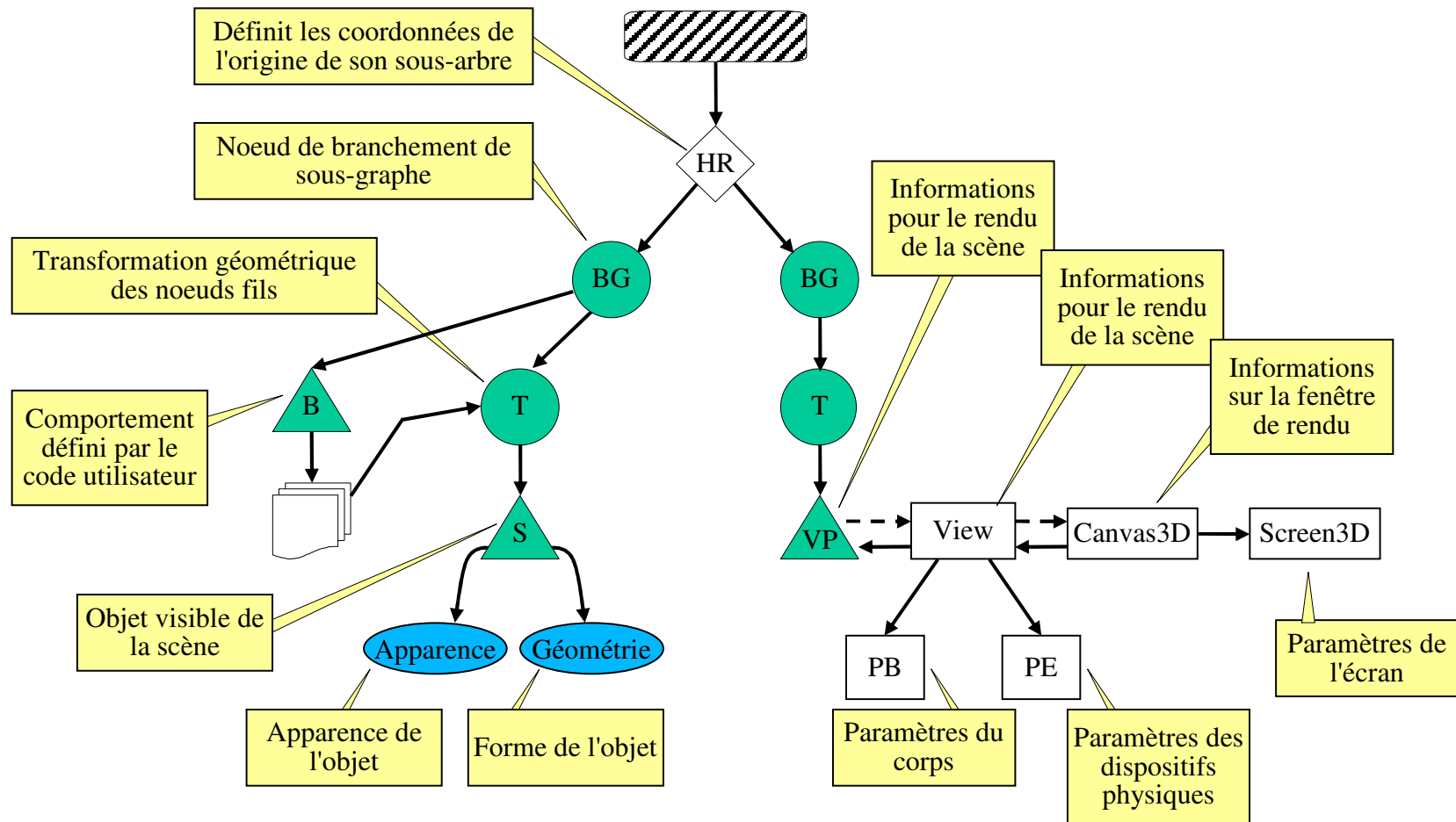
## OpenGL/Mesa

---

- Un squelette d'application en C (mode retenu)
- Utilisation d'une liste de commandes (*display list*)
  - Initialisation
    - Début de remplissage de la liste par *glNewList*  
`glNewList(monObjet, GL_COMPILE);`
    - Appel des commandes OpenGL à intégrer dans la liste
    - Fin de remplissage de la liste par appel à *glEndList()*
  - Utilisation lors du rendu (dans *display*)
    - Appel à *glCallList()* pour exécuter les commandes stockées  
`glCallList(monObjet);`

# La 3D programmée

## Java 3D



# La 3D programmée

## Direct 3D



- Très proche d'OpenGL dans les principes
  - C et C++
  - Modes retenu et immédiat
  - Accélérations graphiques matérielles
- Direct3D  $\in$  DirectX
- Très réactive et très utilisée pour les jeux
- Flexible vertex format
- Managed DirectX depuis d'autres langages de programmation

# La 3D programmée Performer



**IRIS PERFORMER**

- Bibliothèque moyen niveau professionnelle sur SGI/Linux
- Gère le parallélisme (plusieurs pipelines)
- Gestion de scènes complexes
- Charge de nombreux formats de fichiers 3D
- Graphe de scène
- Crée et gère ses fenêtres

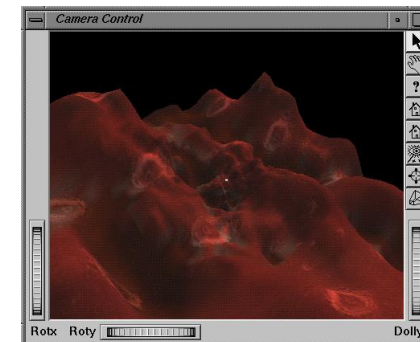
[www.sgi.com/software/performer/](http://www.sgi.com/software/performer/)



# La 3D programmée Open Inventor

- Surcouche moyen niveau de OpenGL
- SGI, portage Windows payant
- Graphe de scène
- Visualiseur avec sélection, extensible
- Lié au format wrl (VRML 1 & 2)
- Pas très efficace

`oss.sgi.com/projects/inventor/`





## Quelle API choisir ?

---

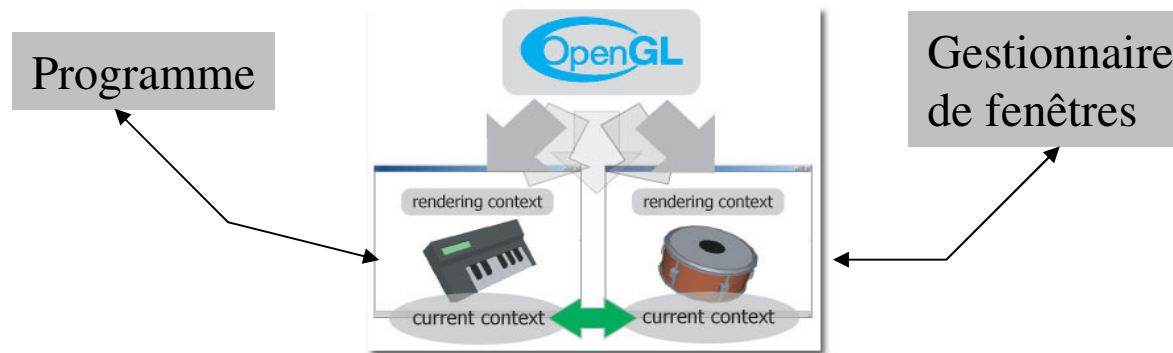
- Visualiser un objet 3D : difficile (cf formats)  
Maya, 3D StudioMax,  
perfly (Performer), ivview (Inventor), Java3D
- Petite application graphique non critique  
OpenGL  
Java3D (bibliothèque de classes réutilisables)
- Application graphique performante  
OpenGL, DirectX

# Structure d'un programme

---

- La structure d'un programme OpenGL
  - Initialisation
    - Demande de création d'un contexte GL  
Double buffer, stéréo, transparence, ...
  - Boucle principale
    - Gestion des interactions
    - Mise à jour des données – animations, interactions
    - Calcul de l'image (*render*)
    - Affichage de la nouvelle image (*swap*)
- Le code dépend du système utilisé (X/Windows, Win32, GLUT, VR Juggler)

# Le contexte GL



- Machine à états

Matrices de transformation, type d’affichage, couleurs, normale ...

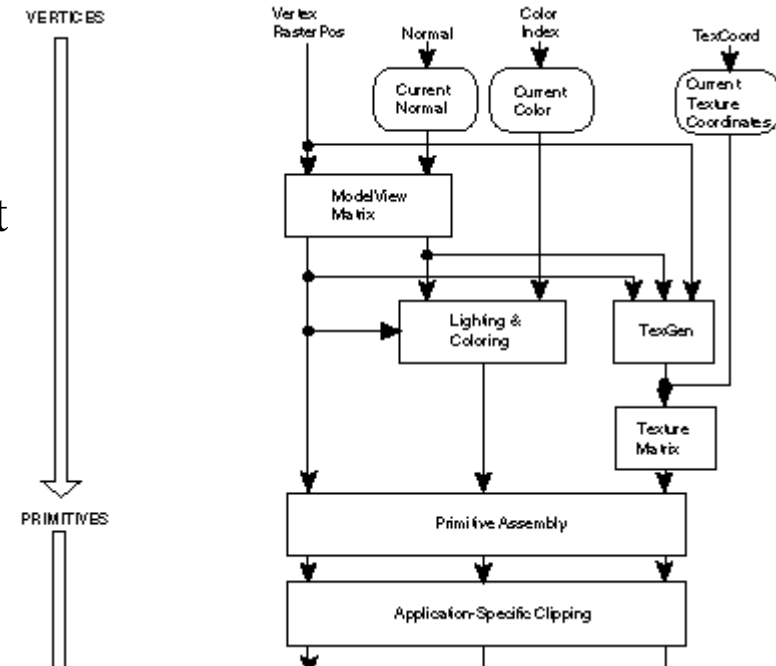
- Buffers (color, depth, stencil, ...)

Stockant, pour chaque pixel, plus ou moins de bits

- Avec les librairies utilitaires (GLUT, GLX, ...) il est géré de manière transparente et automatique

# Procédure d'affichage principale

- Effacer l'écran
- Description de la caméra
- Pour chaque objet de la scène
  - Placement de l'objet
  - Modification de la machine à état
  - Ordres d'affichage



# Effacer l'écran

---

- Couleur de fond – à l'initialisation

```
glClearColor(r, g, b, a);
```

- Effacement – à chaque trame

```
glClear(flags);
```

```
flag = GL_COLOR_BUFFER_BIT
```

```
et/ou GL_DEPTH_BUFFER_BIT
```

```
et/ou GL_ACCUM_BUFFER_BIT
```

```
et/ou GL_STENCIL_BUFFER_BIT
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

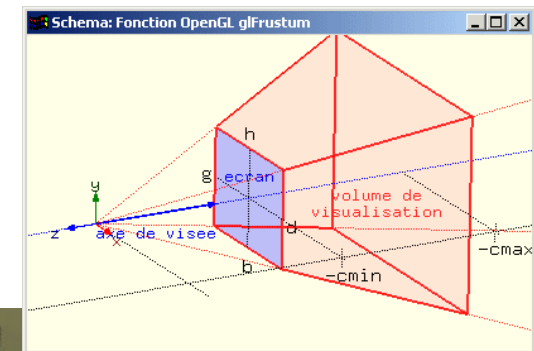
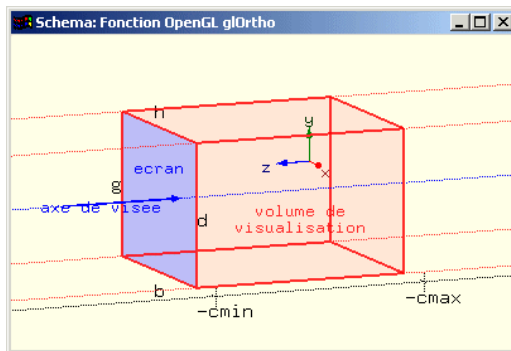
# Transformations

---

- Chaque vertex subit des transformations avant d'être affiché
  - Position et orientation de l'objet dans la scène
  - Inverse de la position et l'orientation de la caméra
  - Projection 3D  $\rightarrow$  2D
- Les 2 premières transformations sont stockées dans la matrice *MODELVIEW*
- La projection est stockée dans la matrice *PROJECTION*

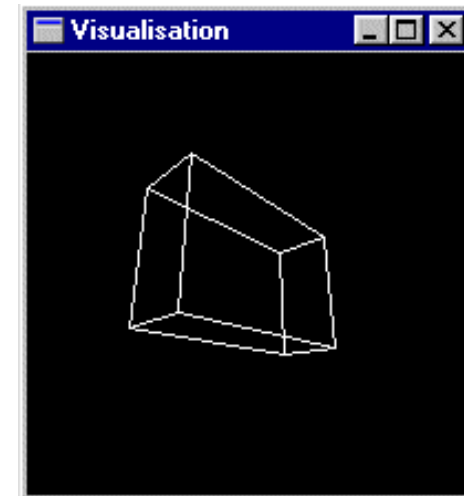
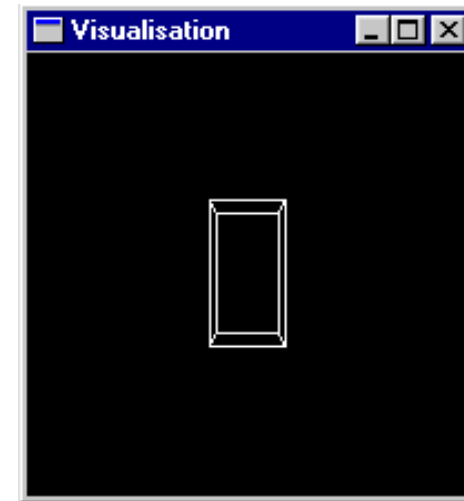
# Description de la caméra

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glOrtho(...); ou glFrustum(...);  
Ou gluPerspective(); et gluLookAt(...);
```



# Positionnement des objets

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(x, y, z);  
glRotatef(alpha, x, y, z);  
glScalef(sx, sy, sz);  
OU glMultMatrixf(m);  
  
glBegin(...);  
...  
glEnd();
```





# Translation, Rotation, Scaling

---

- `glTranslate[d/f](x, y, z)` déplace l'objet du vecteur spécifié
  - Ex: `glTranslatef(0.0f, 0.0f, -6.0f);`
- `glRotate[d, f](angle, x, y, z)` tourne l'objet autour de l'axe spécifié (angle en degré)
  - Ex: `glRotatef(90.0f, 0.0f, 1.0f, 0.0f);`
- `glScale[d/f](x, y, z)` étire l'objet selon les facteurs spécifié pour chacun des axes
  - Ex: `glScalef(2.0f, 2.0f, 2.0f);`

# Pile de matrices

## ■ Structure de repères hiérarchiques



$$P(x,y,z) = P * T1 * R1 * R2 * T2 * (x,y,z)$$

## ■ Pour chaque glMatrixMode, on a une pile

```
glPushMatrix();  
// Modification et utilisation de la matrice  
glPopMatrix();
```

## ■ Très utile pour le dessin de structures hiérarchiques

# Primitives graphiques : `glBegin(...)`



GL\_POINTS



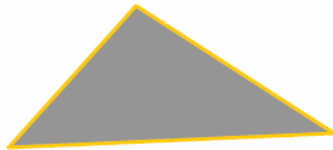
GL\_LINES



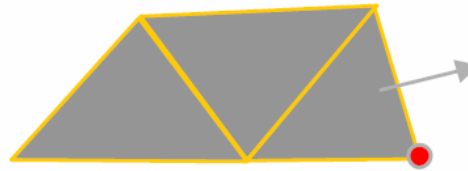
GL\_LINE\_STRIP



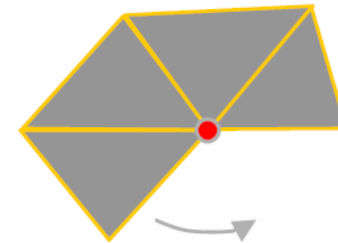
GL\_LINE\_LOOP



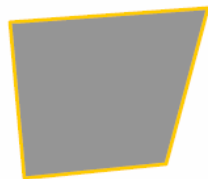
GL\_TRIANGLES



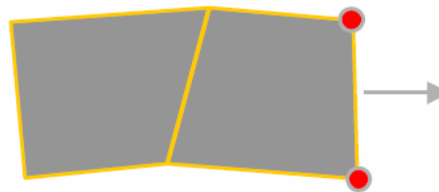
GL\_TRIANGLE\_STRIP



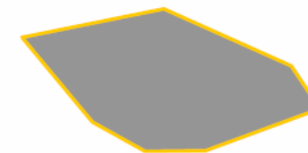
GL\_TRIANGLE\_FAN



GL\_QUADS



GL\_QUAD\_STRIP



GL\_POLYGON

# Affichage de primitives

```
glBegin(primitive);  
// primitive = GL_POINTS, GL_LINES, GL_TRIANGLES...  
glColor3f(r,g,b);  
glNormal3f(nx, ny, nz);  
glTexCoord(u,v);  
glVertex3f(x,y,z);  
...  
glEnd();
```

➤ Répéter n fois  
(n selon la primitive)

- Seul le glVertex est obligatoire,  
sinon la dernière valeur spécifiée est utilisée

# Exemple

## Afficher un triangle

---

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

void render() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        glColor3f(1.0f,0.0f,0.0f);    // couleur V1
        glVertex3f( 0.0f, 1.0f,-3.0f); // position V1
        glColor3f(0.0f,1.0f,0.0f);    // couleur V2
        glVertex3f(-1.0f,-1.0f,-3.0f); // position V2
        glColor3f(0.0f,0.0f,1.0f);    // couleur V3
        glVertex3f( 1.0f,-1.0f,-3.0f); // position V3
    glEnd(); // GL_TRIANGLES
}
```

# Exemple

## Programme principal avec GLUT (1)

---

```
#include <GL/glut.h>
void init();
void display();
void idle();
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutIdleFunc(idle);
    glutMainLoop();
}
```

# Exemple

## Programme principal avec GLUT (2)

---

```
void init() { // initialise la caméra
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60, 1.3333, 1.0, 5000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void display() { // calcule et affiche une image
    render();
    glutSwapBuffers();
}

void idle() { // reaffiche une fois terminé
    glutPostRedisplay();
}
```

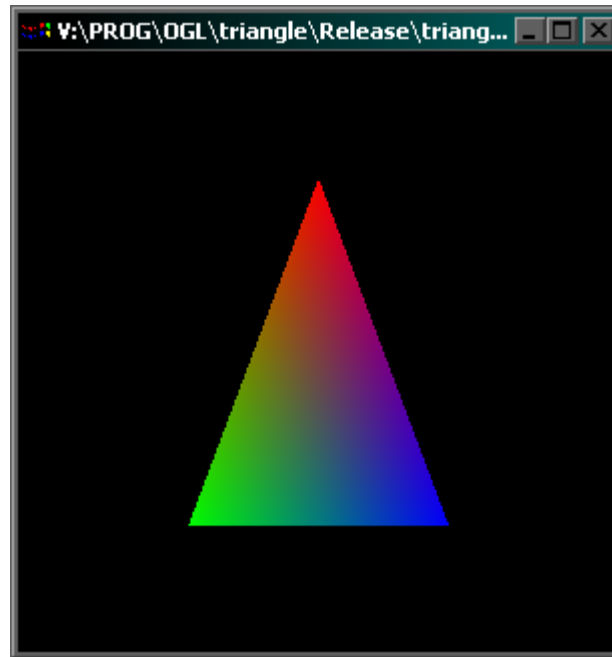
# Exemple

## Compilation avec GLUT

### ■ Compilation:

```
gcc -o triangle main_glut.c render_triangle.c -lglut -  
lGLU -lGL -lXmu -lXext -lX11 -lm
```

### ■ Résultat:

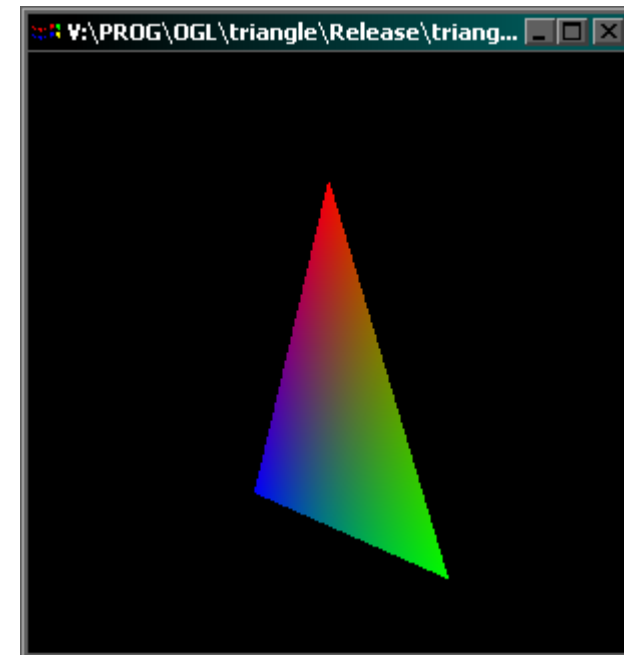




# Exemple

## Ajout de transformations

```
// render_rotate.c
#include <GL/gl.h>
int nbframe=0;
void render() {
    glClear(...); // comme Exemple 1
    glPushMatrix();
    glTranslatef(0,0,-3);
    glRotatef(nbframe,0,1,0);
    glBegin(GL_TRIANGLES);
    ... // comme Exemple 1
    glEnd(); // GL_TRIANGLES
    glPopMatrix();
    glBegin(...)
    ...
    glEnd();
    nbframe++;
}
```



# Options d'affichage

```
glEnable(GL_LIGHTING);
```

**Prise en compte de la normale**

```
glCullFace(face);
```

```
glEnable(GL_CULL_FACE);
```

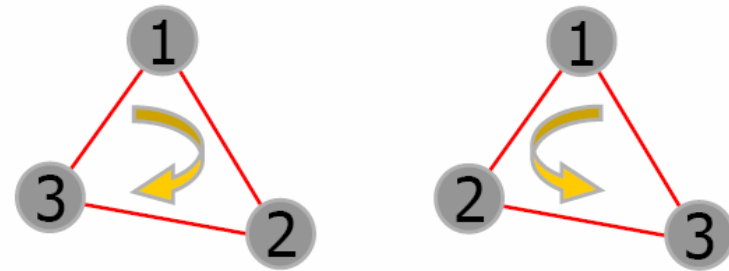
*face* = GL\_FRONT, GL\_BACK, GL\_FRONT\_AND\_BACK

```
glPolygonMode(face, mode);
```

*mode* = GL\_FILL, GL\_LINE

```
glLineWidth(3);
```

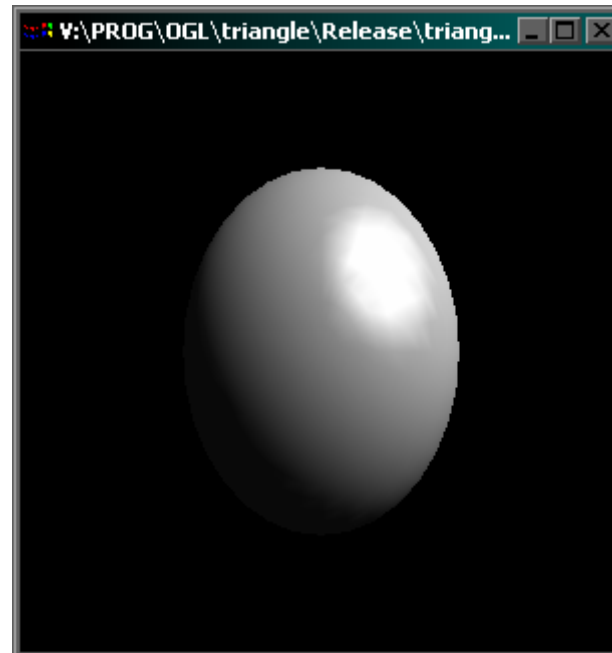
```
glPointSize(12);
```



# Matériaux et Lumières

## Exemple

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 50.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```



## Et plein d'autres choses dans ...

---

- OpenGL Programming Guide (The Red Book)  
[http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI\\_Developer/OpenGL\\_PG/](http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI_Developer/OpenGL_PG/)
- OpenGL Reference Manual (The Blue Book)  
[http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI\\_Developer/OpenGL\\_RM/](http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI_Developer/OpenGL_RM/)
- <http://www.opengl.org> : site officiel. Forums
- <http://nehe.gamedev.net> : le site de tutoriels OpenGL
- Une mine d'infos en français : le site de Nicolas Janey : <http://raphaello.univ-fcomte.fr/Ig/>

# Sélection

---

Savoir quel est l'objet sous la souris

Rendre la scène (pas d'affichage)

```
glRenderMode(mode) gluPickMatrix(...)
```

Limitation à une région de quelques pixels

Etiquetage des primitives :

```
glInitNames(); glPushName(); glPopName();
```

Interpréter les résultats

Pile d'étiquettes

Autres informations (clipping, couleur, z, ...)

# Les formats de fichiers

---

## Modèles 3D

Le possible futur : X3D

VRML (1 ou 2), WRL, 3DS, smf, pfb, obj, dxf,...

“3D Graphics File Formats : A Programmer's Reference”, K. Rule

[www.3dcafe.com](http://www.3dcafe.com)

## Formats d'images

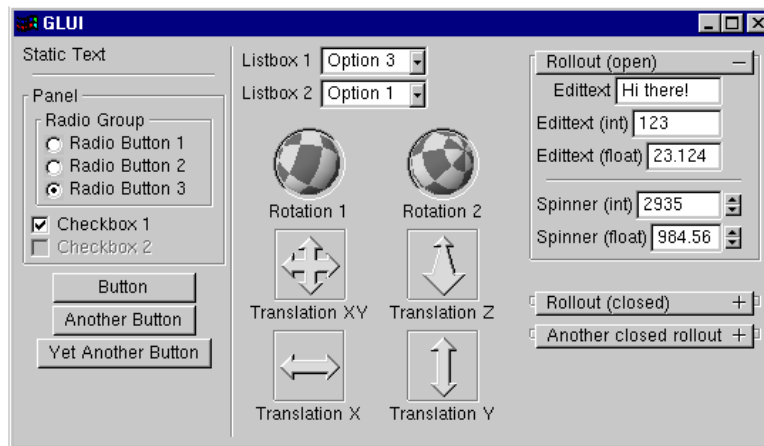
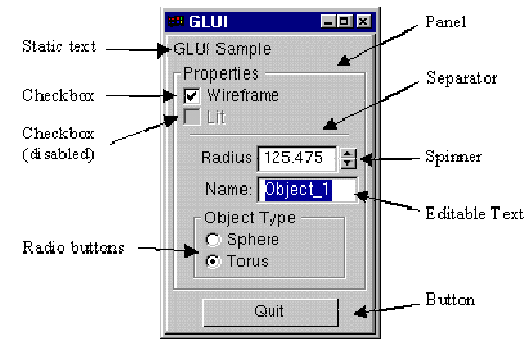
RGB, JPEG, TIF, ...

Puissances de 2 dans les tailles des textures

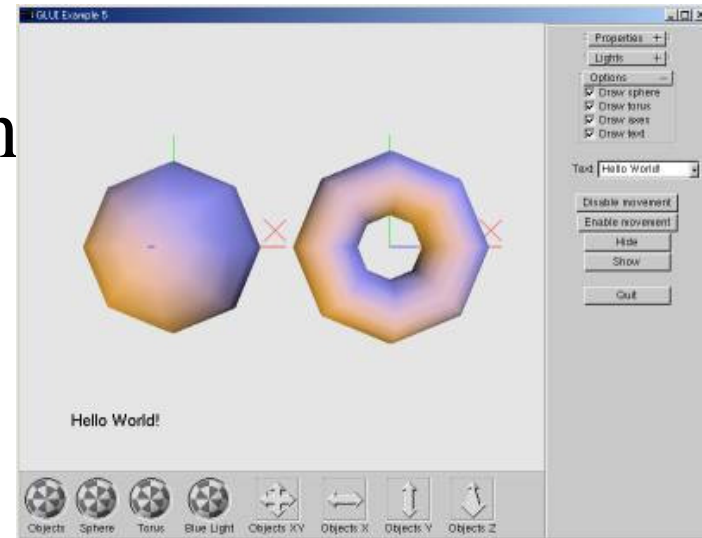
Avec OpenGL, il faut tout programmer soit même

# Interface Utilisateur (GUI)

- GLUI
  - Simple mais pas objet (void\*)
  - Variables globales

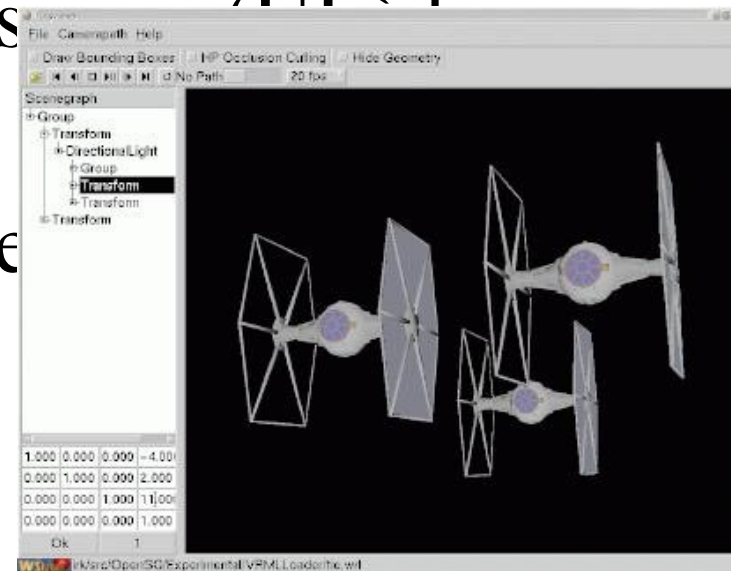


Lin



# Interface Utilisateur (GUI)

- Qt
  - Multi-plateformes (Unix, Windows, Mac, ...)
  - qmake (Makefile), des boutons et une souris)
  - Passage de messages entre les composants





# Interface Utilisateur (GUI)

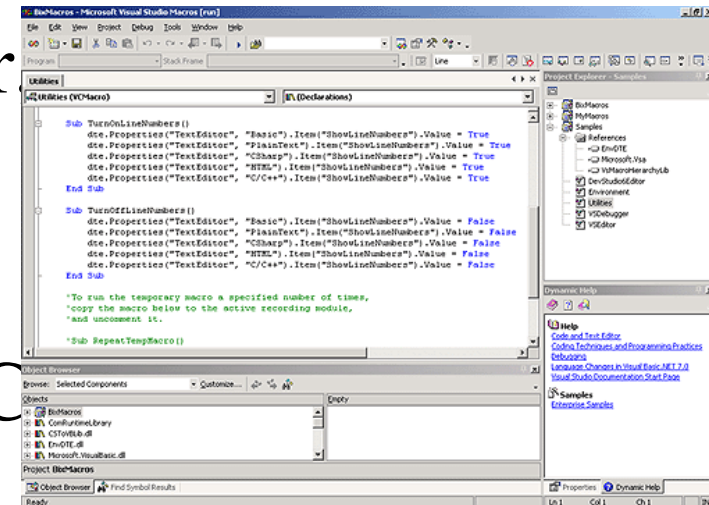
- Windows
- Interface intégrée Windows

*Component Object Modeling*  
Communication par messages

Menus, cases à cocher

- Deux systèmes possibles

Microsoft Foundation C  
Win32



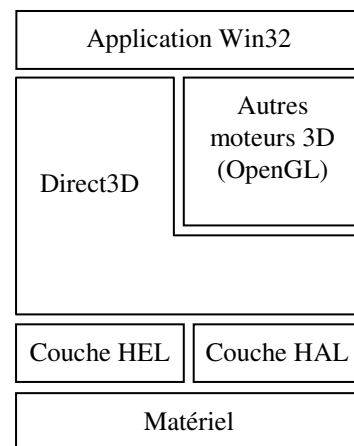
# Présentation rapide de Direct3D

## Architecture logicielle et pipeline graphique

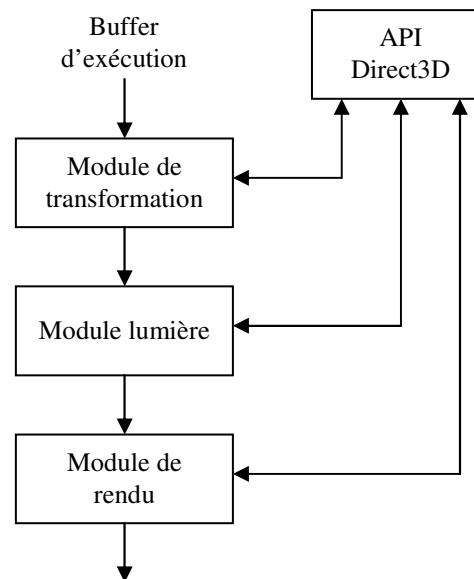
Architecture classique –accéder à l'accélération

Trois modules séparés dans le pipeline 3D

*Architecture Direct3D*



*Pipeline graphique*



# Présentation rapide de Direct3D

---

Les 3 modules du pipeline :

Ils peuvent être remplacés par des modules sur mesure

Le module de transformation : visibilité, projection, *clipping*

Le module de lumières : pour gérer l'éclairement des faces

Module RGB pour les lumières colorées

Module Ramp pour les lumières monochromes

Le module de rendu (*rasterization*) affiche la scène en fonction des faces projetées et de leur éclairement

Modes immédiat et retenu : 3 niveaux

Mode immédiat avec listes (*Execution Buffer*)

Mode retenu utilisant un graphe de scène

# Présentation rapide de Direct3D

## Un squelette d'application en C (mode immédiat)

```

/// Remplissage du buffer d'exécution avec un triangle. /
/// Pour cela, on utilise un pointeur sur le champ lpData de notre buffer
fface = (LPD3DVERTEX)d3dExeBufDesc.lpData;

/// Premier sommet - Coordonnées
fface->dvX = D3DVAL( 0.0); face->dvY = D3DVAL( 1.0); face->dvZ = D3DVAL( 0.0);
/// Premier sommet - Normale
fface->dvNX = D3DVAL( 0.0); face->dvNY = D3DVAL( 0.0); face->dvNZ = D3DVAL(-1.0);
/// Premier sommet - Coordonnées de texture
fface->dvTU = D3DVAL( 0.0); face->dvTV = D3DVAL( 1.0);

fface++;
/// Deuxième sommet
fface->dvX = D3DVAL( 1.0); face->dvY = D3DVAL(-1.0); face->dvZ = D3DVAL( 0.0);
fface->dvNX = D3DVAL( 0.0); face->dvNY = D3DVAL( 0.0); face->dvNZ = D3DVAL(-1.0);
fface->dvTU = D3DVAL( 1.0); face->dvTV = D3DVAL( 1.0);

fface++;
/// Troisième sommet
fface->dvX = D3DVAL(-1.0); face->dvY = D3DVAL(-1.0); face->dvZ = D3DVAL( 0.0);
fface->dvNX = D3DVAL( 0.0); face->dvNY = D3DVAL( 0.0); face->dvNZ = D3DVAL(-1.0);
fface->dvTU = D3DVAL( 1.0); face->dvTV = D3DVAL( 0.0);

```

# Présentation rapide de Direct3D

---

mode immédiat – Avantages :

- format plus flexible des sommets : possibilité de rendu de sommets pré-transformés et/ou pré-illuminés
- multi-texturage
- plus grand contrôle des animations
- création de modules "sur mesure"

Cependant, le programmeur devra connaître les algorithmes

3D de base pour manipuler :

- les matrices de transformation
- le calcul des normales aux sommets
- utilise uniquement des triangles
- gérer entièrement la boucle de rendu

# Présentation rapide de Direct3D

---

## L'initialisation

```
// 1. Création de l'interface Direct3D 8
g_pD3D = Direct3DCreate8( D3D_SDK_VERSION );

// 2. Récupération du mode vidéo par défaut
D3DDISPLAYMODE d3ddm;
g_pD3D->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,&d3ddm );

// 3. Création de la surface de rendu pour le mode par défaut,
//      un rendu hardware et un traitement des sommets logiciel
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;
g_pD3D->CreateDevice( D3DADAPTER_DEFAULT,
D3DDEVTYPE_HAL, hWnd,
D3DCREATE_SOFTWARE_VERTEXPROCESSING,
```

# Présentation rapide de Direct3D

---

## L'initialisation (suite)

```
// 4. Création de la matrice de projection
D3DXMATRIX matProj;
FLOAT fAspect = 1.0f;
D3DXMatrixPerspectiveFovLH(&matProj,
    D3DX_PI/2, fAspect, 10.0f, 1000.0f );
g_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj )

// 5. Création de la matrice du modele
D3DXMATRIX mat;
D3DXVECTOR3 from = D3DXVECTOR3(0.0f,-300.0f,200.0f);
D3DXVECTOR3 to =
D3DXVECTOR3(0.0f,0.0f,90.14f);
D3DXVECTOR3 up =
D3DXVECTOR3(0.0f,0.0f,1.0f);
D3DXMatrixLookAtLH(&mat,&from,&to,&up);
g_pd3dDevice->SetTransform(D3DTS_VIEW,&mat)
```

# Présentation rapide de Direct3D

---

## L'initialisation (suite et fin)

```
// 6. Création d'une source lumineuse
D3DXVECTOR3 vecDir;
D3DLIGHT8 light;
ZeroMemory(&light, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
vecDir = D3DXVECTOR3(1.0f,0.0f,-1.0f);
D3DXVec3Normalize((D3DXVECTOR3*)&light.Direction, &vecDir );
light.Range = 1000.0f;
g_pd3dDevice->SetLight(0, &light );
g_pd3dDevice->LightEnable(0, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_LIGHTING, TRUE );
```



# Présentation rapide de Direct3D

---

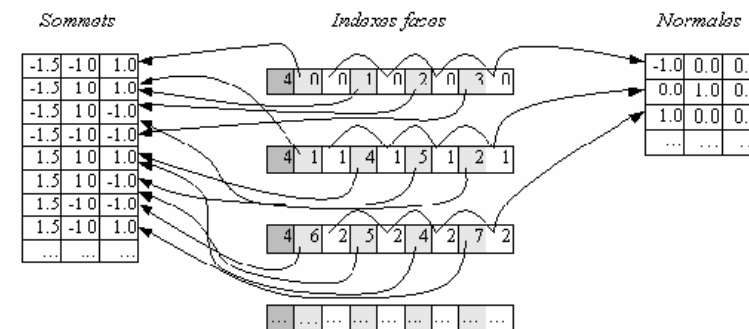
## La structure des sommets

```
// Une structure pour stocker nos sommets
struct CUSTOMVERTEX
{
    float x, y, z; // The position for the vertex
    float nx, ny, nz; // The normal for the vertex
    DWORD colour; // The vertex colour
};
// Notre structure sur mesure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE)
```

# Présentation rapide de Direct3D

- Remplissage de la structure des sommets

```
CUSTOMVERTEX g_Vertices[] = {
g_pd3dDevice->CreateVertexBuffer(
    24*sizeof(CUSTOMVERTEX),
    0,
    D3DFVF_CUSTOMVERTEX,
    D3DPOOL_DEFAULT,
    &g_pVB );
```



```
VOID* pVertices;
g_pVB->Lock( 0, sizeof(g_Vertices), (BYTE**)&pVertices,
    0);
memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
```

# Présentation rapide de Direct3D

---

## Remplissage de la structure des sommets

```
// Efface l'écran et préparation au rendu
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
    D3DCOLOR_XRGB(0xFF,0xFF,0xFF), 1.0f, 0 );
g_pd3dDevice->BeginScene();
// Configuration du rendu
g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
// Configuration de la matrice de transformation du modèle
D3DXMATRIX mat;
D3DXMatrixTranslation(&mat,-150.0f,0.0f,0.0f);
g_pd3dDevice->SetTransform(D3DTS_WORLD,&mat);
D3DXMatrixRotationZ(&mat,frot);
g_pd3dDevice->MultiplyTransform(D3DTS_WORLD,&mat);
// Dessiner la primitive géométrique
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 4 );
// Fin du rendu
g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```