Multimédia et Interaction Homme-Machine

Programmation de la 3D

Alexandre Topol

Département Informatique
Conservatoire National des Arts & Métiers

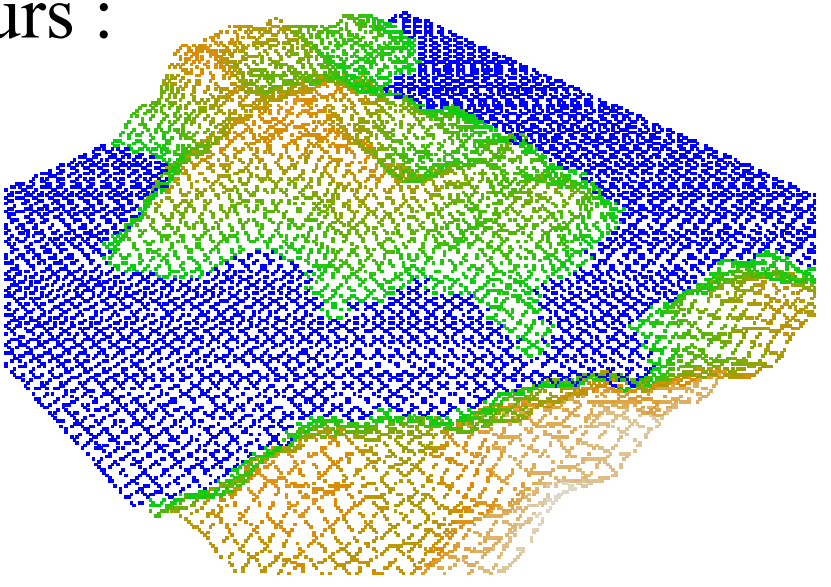
2007-2008

Objectifs du cours

- Rappels et compléments images de synthèse
- Présenter les différentes API existantes
- Expliquer le principe d'utilisation
- Décrire les fonctionnalités les plus courantes
- Interfaces utilisateur (GUI)
- Fournir du code de départ, des exemples

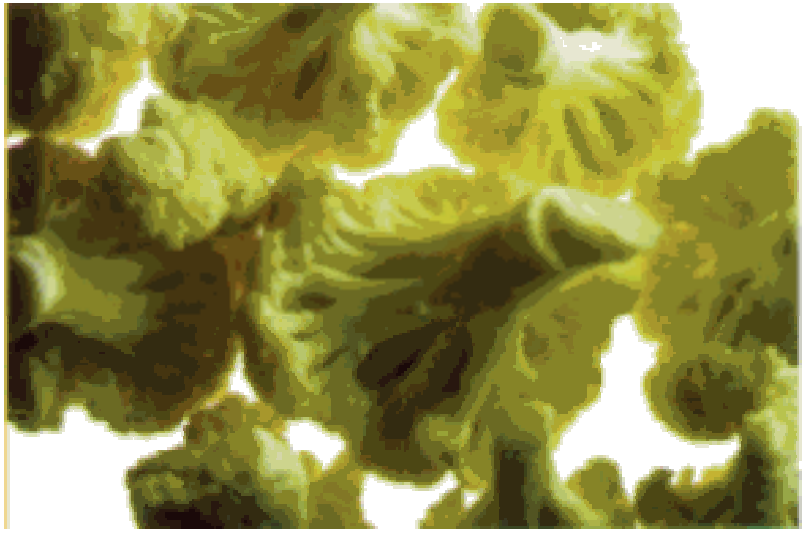
Fractales et modèles procéduraux

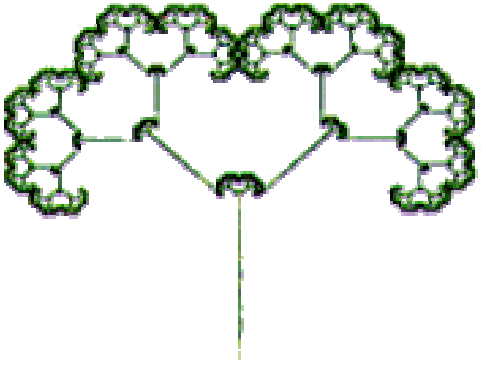
- Le besoin : appréhender la complexité des formes naturelles
- Approche : récurrence et/ou hasard simulé
- Exemples de tous les jours :
  - Le chou-fleur
  - La fougère
  - Cristaux de neige
  - Les montagnes
  - …



Fractales et modèles procéduraux
Motif et récurrence

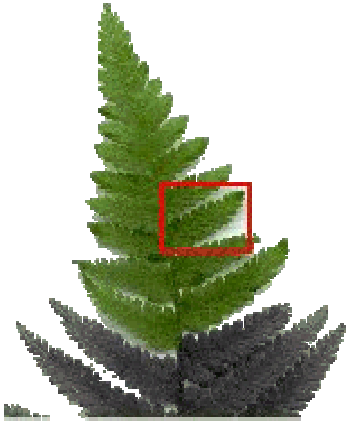
- Exemple du chou-fleur





Fractales et modèles procéduraux
Motif et récurrence

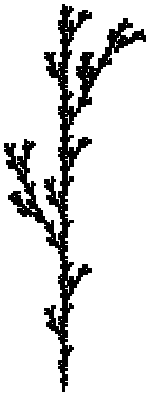
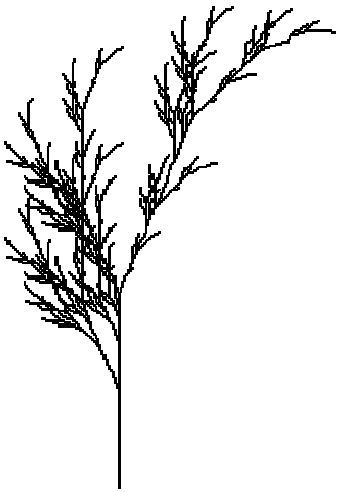
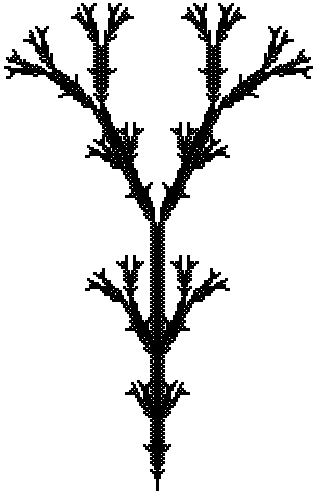
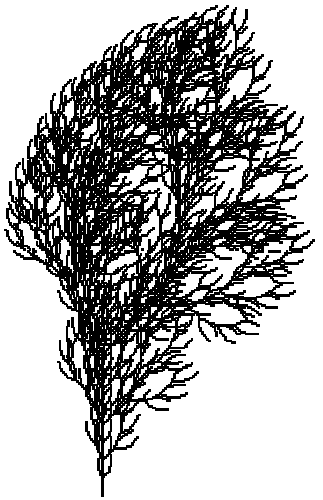
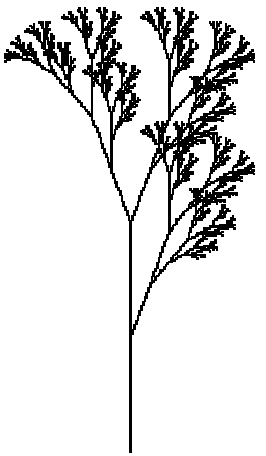
- Exemple de la fougère



Fractales et modèles procéduraux
Motif et récurrence

- Autres plantes

Initiateur : F
Générateur : F[+F]F[-F]F
Angle : 22.5

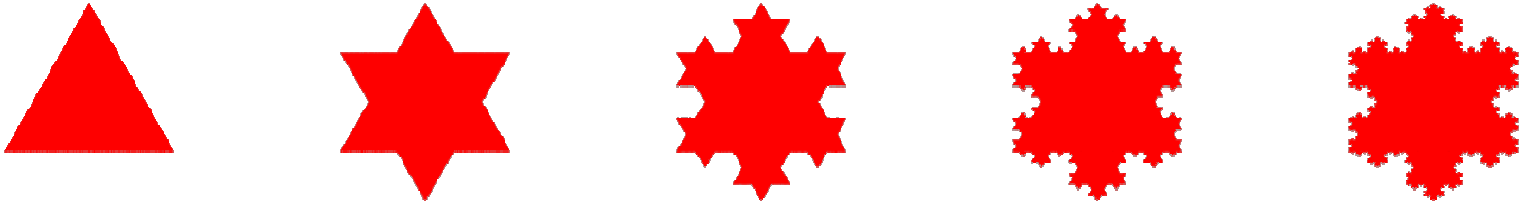







Le logiciel utilisé ici pour tracer cette plante est "L-systems" de P.Bourke

Fractales et modèles procéduraux
Motif et récurrence

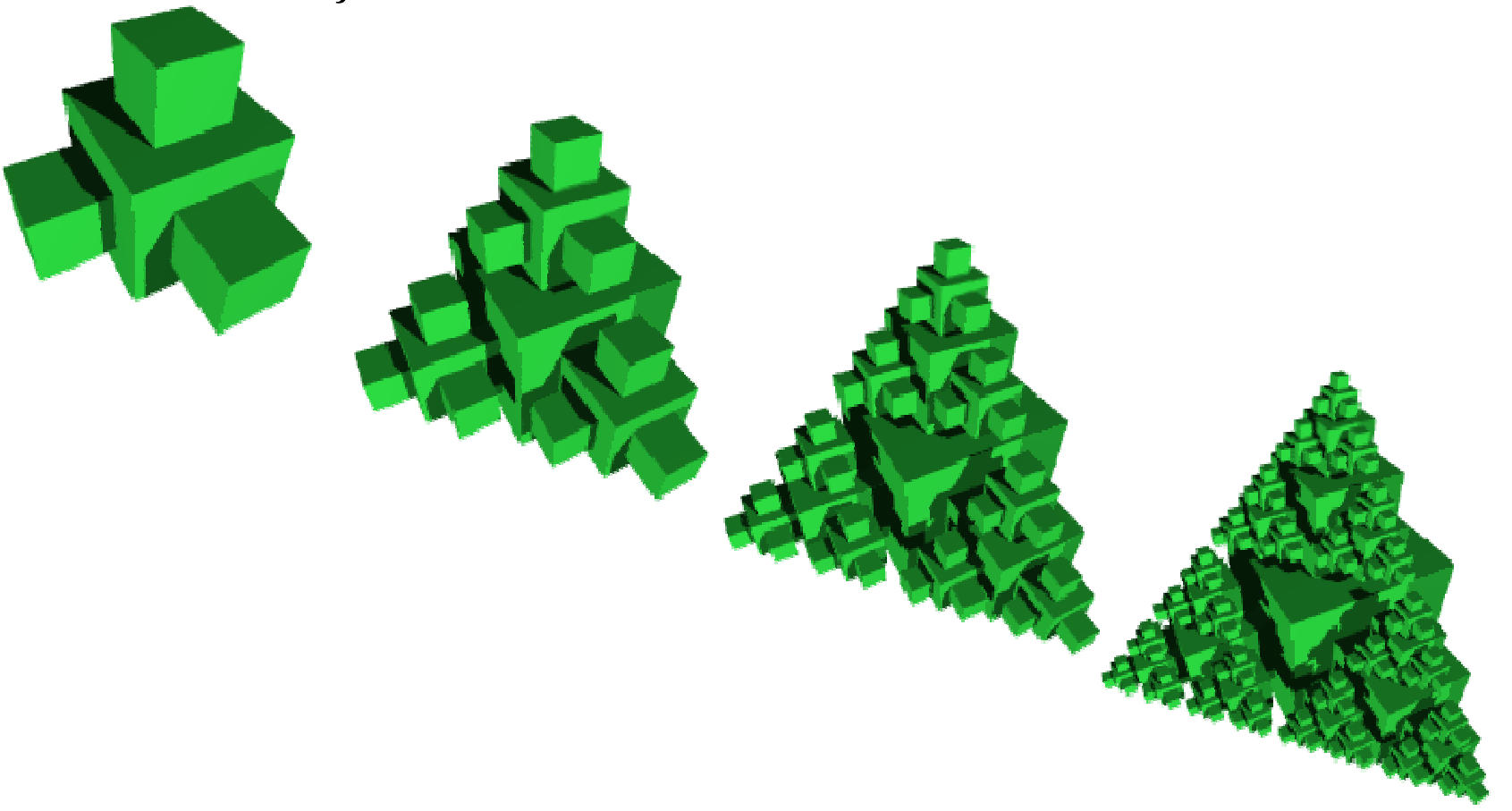
- Exemple fondateur : le flocon de von Koch (1919)

niveau 0	on trace le segment [0, 1].	
niveau 1	on dessine un triangle équilatéral dont le côté a une longueur égale au tiers de l'intervalle initial.	
niveau 2	on applique le même procédé à chacun des nouveaux segments.	
…	…	…



Fractales et modèles procéduraux
Motif et récurrence

- Et en 3D ça marche aussi …



Fractales et modèles procéduraux
Fractales stochastiques

- Utilisation d'un hasard simulé par générateurs aléatoires
  - Gauss
  - Poisson
- L'introduction du hasard évite de reproduire les mêmes motifs à des échelles différentes
- Permet de créer des formes plus naturelles donc réalistes

Fractales et modèles procéduraux
Fractales stochastiques

- Exemple de l'arbre

Fractales et modèles procéduraux
Fractales stochastiques

- Modèles de reliefs en utilisant un exemple de marche aléatoire : le gain au jeu de pile ou face

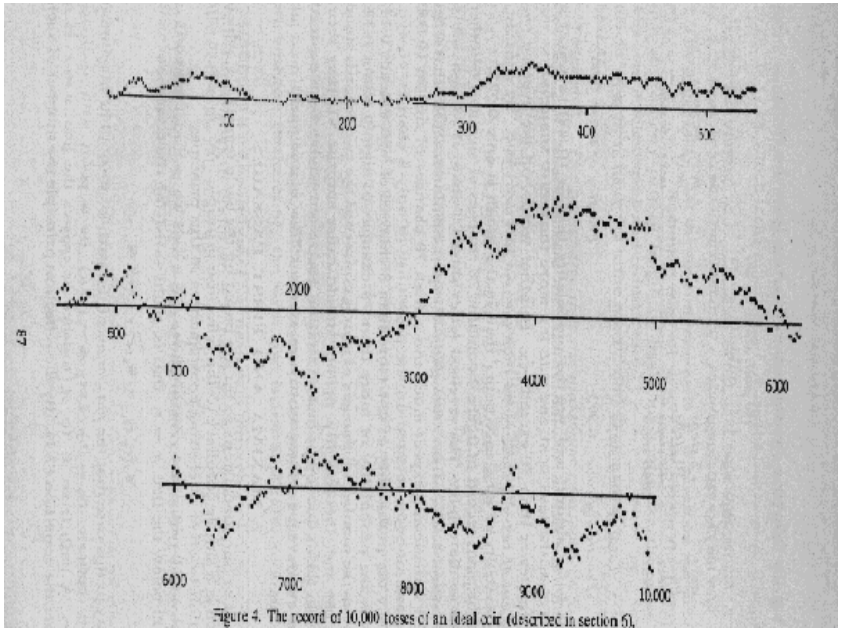
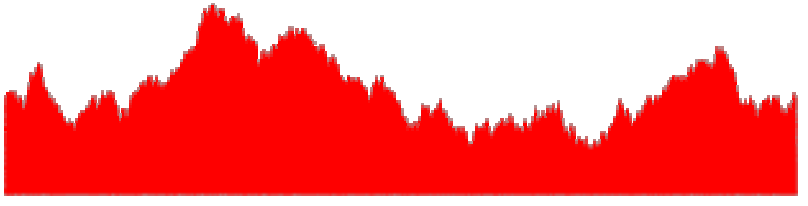


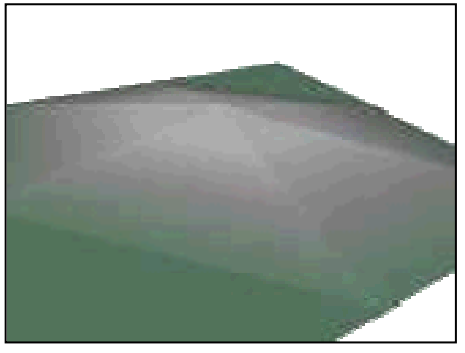
Figure 4. The record of 10,000 tosses of an ideal coin (described in section 6).

Feller - An introduction to probability theory… - vol 1. - Wiley, 1950. p. 87

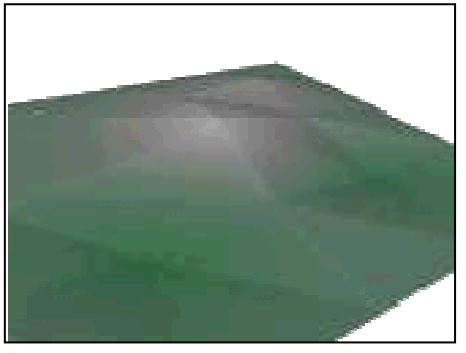


Fractales et modèles procéduraux
Fractales stochastiques

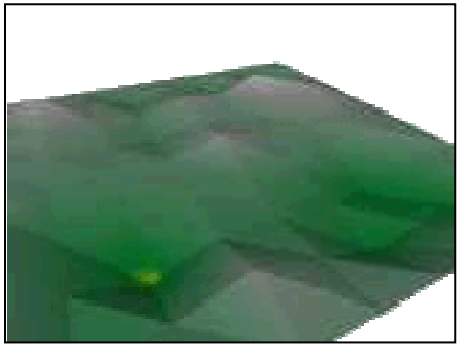
- Modèles de reliefs par subdivision :



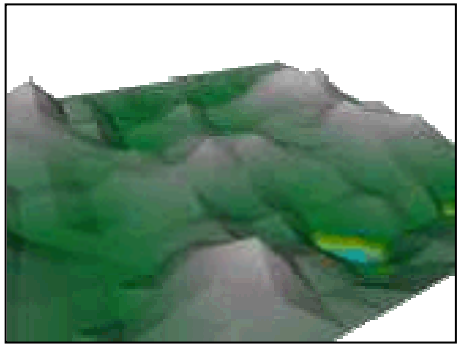
Première itération



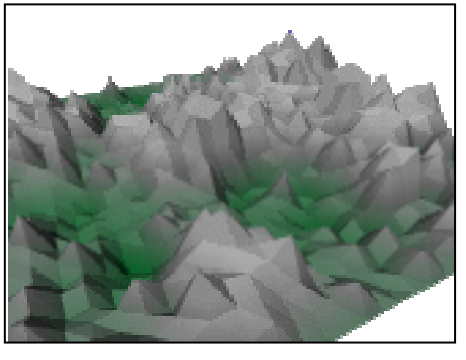
Deuxième itération



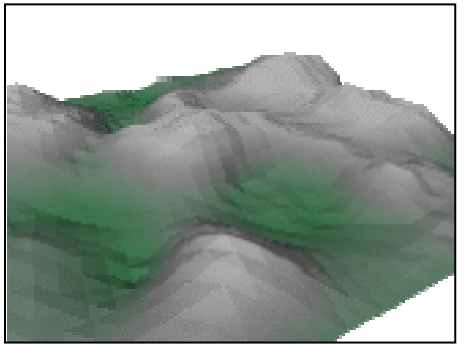
Troisième itération



Quatrième itération



Huitième itération



… et après érosion

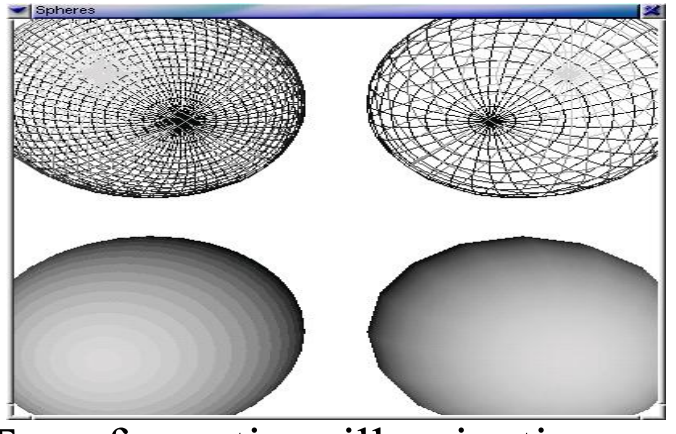
Rappels
Visualisation des scènes : deux méthodes

- Projection de facettes :
  - Les objets doivent être "triangulisés" (*tessaleted*)
  - 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

Rappels
Visualisation des scènes : deux méthodes

Par facettage

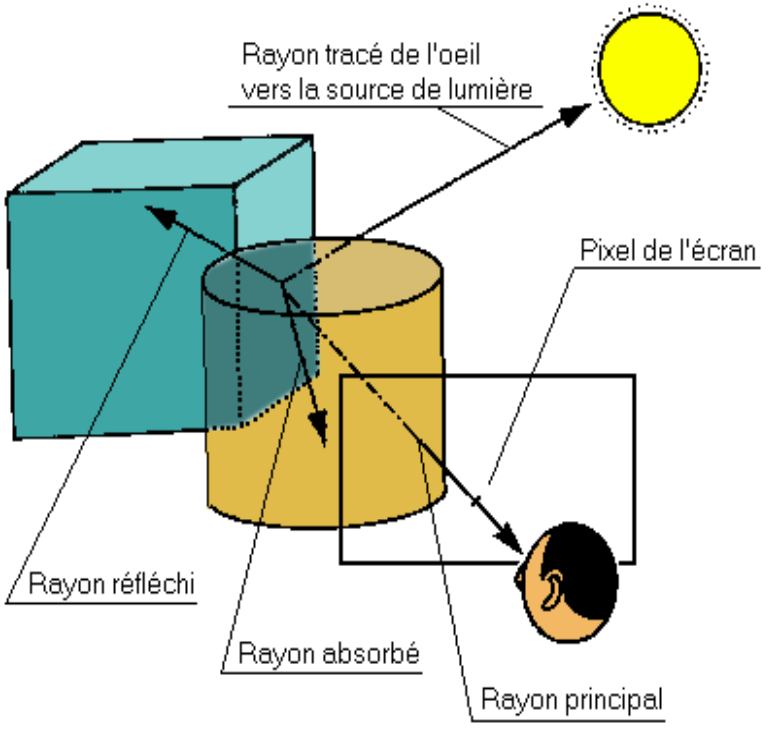
1. Transformation des formes complexes en ensemble de triangles



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

« quantité plutôt que qualité »

Raytracing



« qualité plutôt que quantité »

Rappels
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
```

Rappels
Utilité du lancé de rayons

- Temps d'obtention des images important
  - Utilisations de techniques coûteuses (calculs d'intersections, radiosité, …)
  - Non temps réel mais on y vient …
- Utilisé pour obtenir des images réalistes
  - Réflexions, ombres, brouillards, …
- Utilisé pour obtenir des séquences animées précalculées
  - Dans les films, les pubs, les transitions dans les jeux …

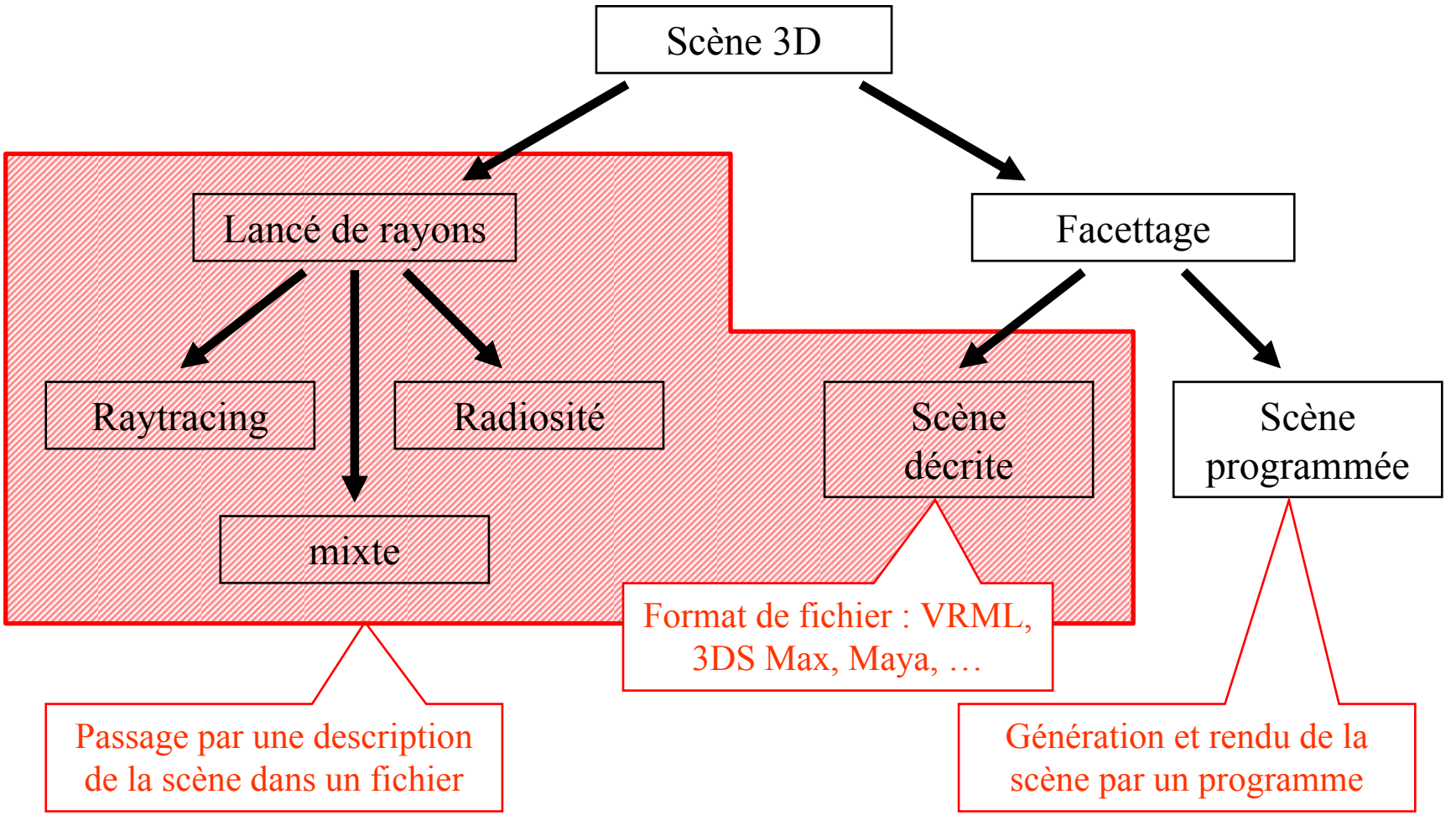
Rappels
Description des objets

- 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

La 3D décrite ou programmée ?

- Et oui … après le choix de la méthode de rendu, on a encore deux 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 programmat°
  - Difficulté de maintenance
  - Pauvre "réutilisabilité"
  - Temps de développement

La 3D décrite ou programmée ?
La 3D décrite

- Utilisation d'un langage de description
- 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

Couche Application

Utilisation d'une API 3D

Couche Émulation

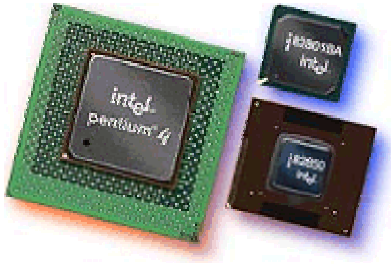
Traduction des commandes émulées en des fonctions d'une API 3D accélérée

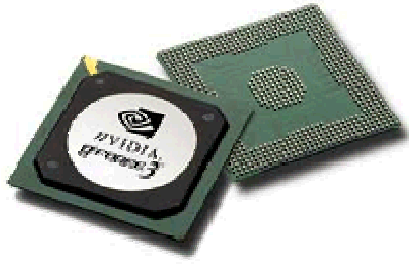
Couche Interface Logicielle

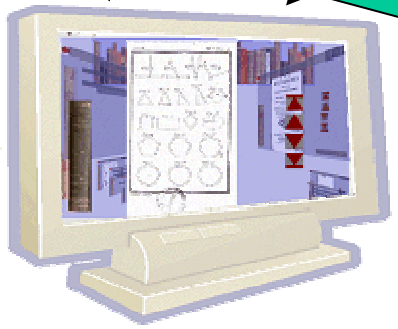
Couche Accélération Matérielle

Prise en charge de fonctions 3D par le matériel

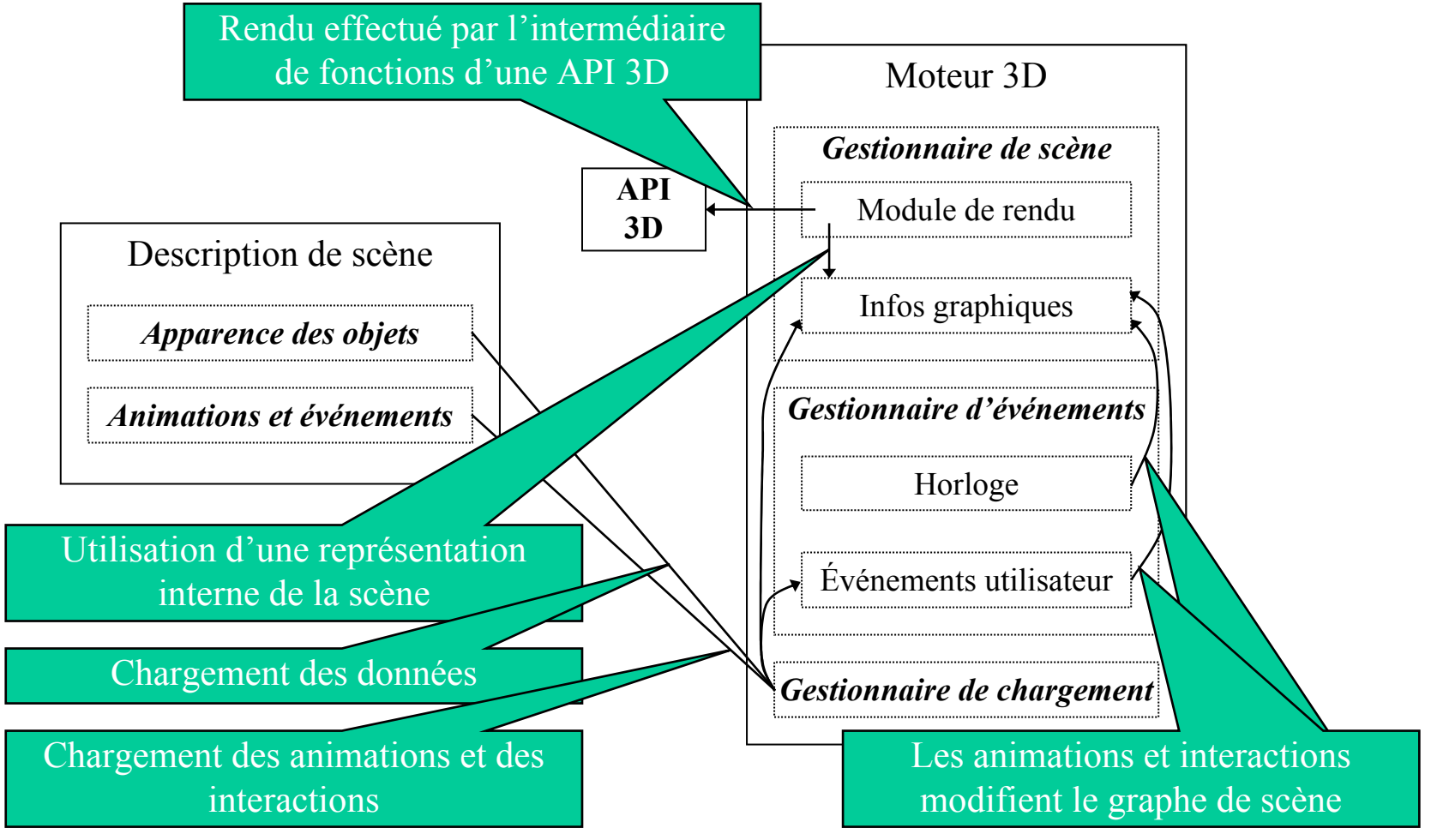
Calcul de fonctions par le microprocesseur







La 3D décrite ou programmée ?
Relations entre les types d'applications 3D



La 3D décrite
Lancé de rayons

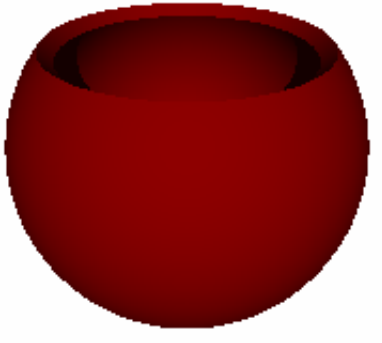
- Des modeleurs (3DS Max, Maya, Blender, …)
  - Construction des scènes par clics de souris
  - Pré-visualisation en 3D par facettage
  - Calcul de l'image ou de l'animation finale par lancer de rayons
- POV (*Persistance Of Vision*)
  - Construction des scènes par langage
  - Pas de pré-visualisation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

- Définition : *Virtual Reality Modeling Language*

  (format de fichier pour la description de scènes 3D interactives avec hyperliens)

- Historique :

  1994 : VRML 1.0 – *Labyrinth* de Pesce et Parisi

  1996 : VRML 2.0 – Moving Worlds de SGI

  1997 : VRML97, normalisation ISO/IEC 14772-1:1997

  - Un monde réaliste
  - Un monde animé
  - Un monde interactif
  - Un monde connecté

  Et puis … VRML200x/X3D

  – DTD XML

  – Ajout de nouveaux nœuds

La 3D décrite
3D par facettage – VRML

- 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

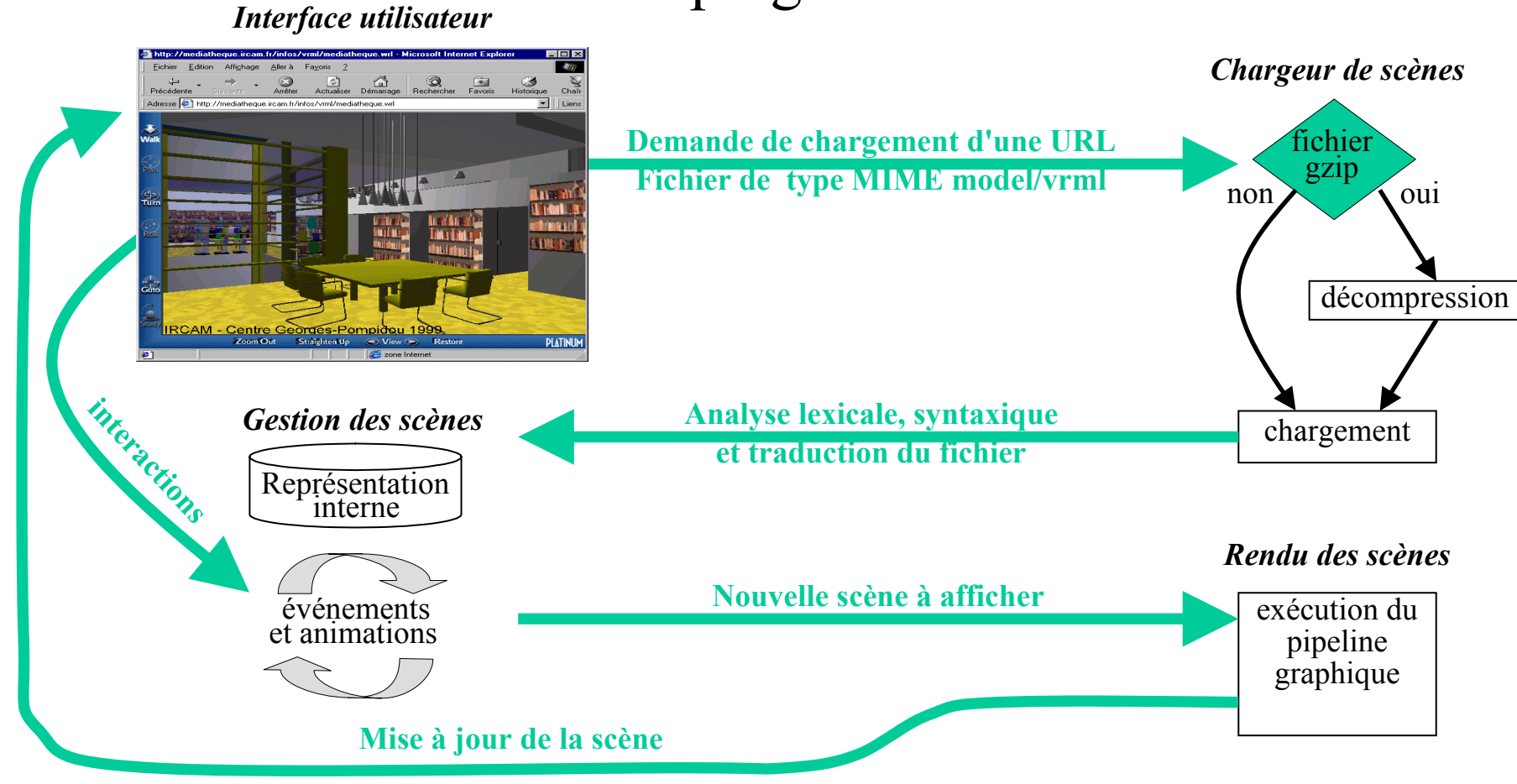
Graphe de scène — Noeuds — Champs

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

⇒cascade de comportements

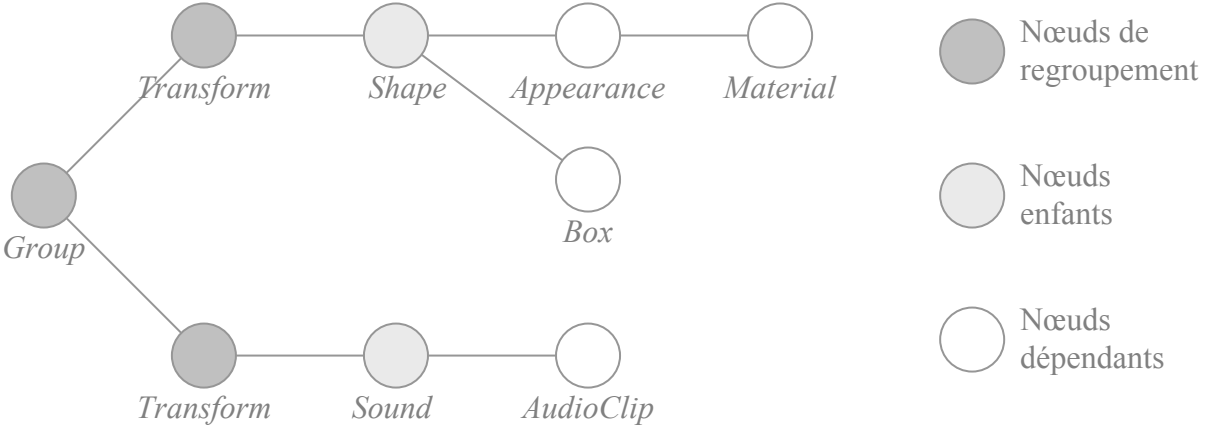
La 3D décrite
3D par facettage – VRML

Fonctionnement d'un plugin VRML



La 3D décrite
3D par facettage – VRML

- Représentation des informations
  - Sous forme d'un graphe de scène
    - Regroupement spatial utile pour le *clipping*
    - Proche des API 3D récentes
  - Utilisation de 3 types de nœuds différents :



La 3D décrite
3D par facettage – VRML

- Format des fichiers : fichiers textes (compressés ou non par Gzip) composés de 4 parties :
  - L'entête : `#VRML <version> <type d'encodage> [commentaires]`

```
#VRML V1.0 ascii    mon fichier VRML 1.0
#VRML V2.0 utf8     mon fichier VRML 2.0
```

  - La définition des prototypes
    - les prototypes externes sont référencés par une URL dont le contenu est un fichier VRML valide (respectant ce format)
  - Le graphe de scène
  - Les routages d'événements
- Précisions :
  - Pour écrire une ligne en commentaire commencer par #
  - Un nœud doit toujours être suivi d'accolades (même vides)
  - Par convention, le repère cartésien utilisé est "Main droite"

La 3D décrite
3D par facettage – MPEG4

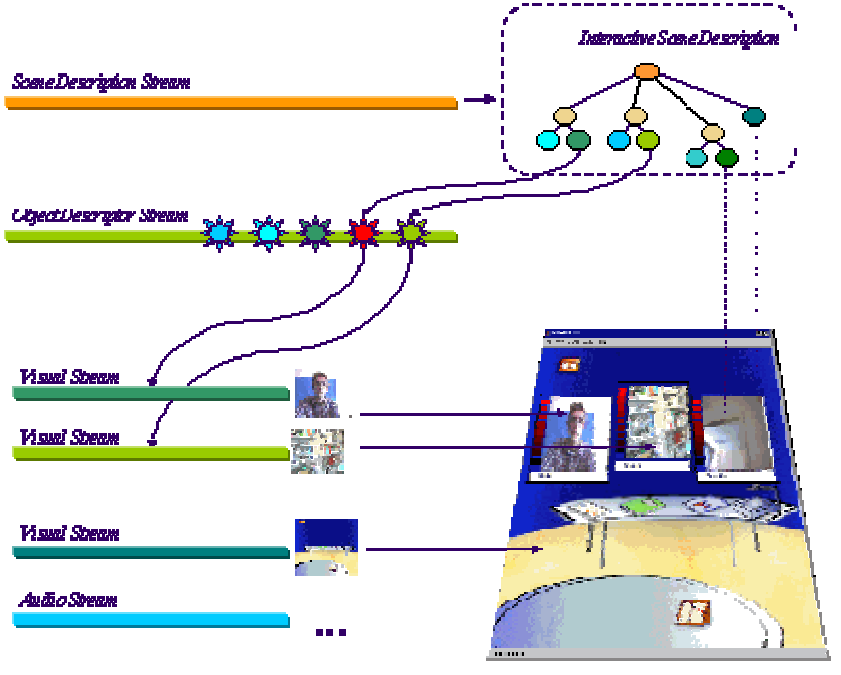
- Une architecture logicielle complète :
  - les méthodes de codage des AVOs (*Audio Visual Objects*)
  - le codage d'une scène complète composée d'AVOs agencés sous forme de graphe
  - le multiplexage et la synchronisation des différents flux de données contenant les AVOs
  - l'interface de communication entre les clients et le serveur MPEG4

La 3D décrite
3D par facettage – MPEG4

- Pour gérer ces différents aspects, 3 couches :
  - La couche distribution
    - La communication avec le serveur
    - Assure le transport des différents flux
    - Multiplexage et démultiplexage des flux
  - La couche système
    - Les Mécanismes d'identification, de description et d'association des différents flux de données
    - Synchronisation entre les objets composant une scène
  - La couche compression
    - Compression/décompression de la scène et des différents objets la composant avec les *codecs* appropriés

La 3D décrite
3D par facettage – MPEG4

MPEG-4 Systems Principles



La 3D décrite
3D par facettage – Metastream

- L'affichage des objets commence dés le début du chargement des premières faces *(Streaming* des scènes 3D)
- Les fichiers sont plus compressé

  ➔ transport plus rapide
- Le *plugin* ajuste automatiquement les attributs des objets afin d'obtenir un rapport qualité/vitesse optimal chez le client

La 3D décrite
3D par facettage – Metastream

La 3D décrite
3D par facettage – Les autres outils

- Des outils très complets, orientés Internet, mais qui ne sont pas des normes
- Construction graphique des interfaces 3D ou langage de programmation (Virtools, SCOL, Blaxxun, …)
- Ce sont des outils multimédia gérant les aspects suivants :
  - les graphiques 2D et 3D,
  - le son et la vidéo,
  - la liaison avec les navigateurs HTML,
  - les communications réseaux, les protocoles SMTP et HTML,
  - les bases de données.

La 3D programmée
Interface de programmation (API) 3D

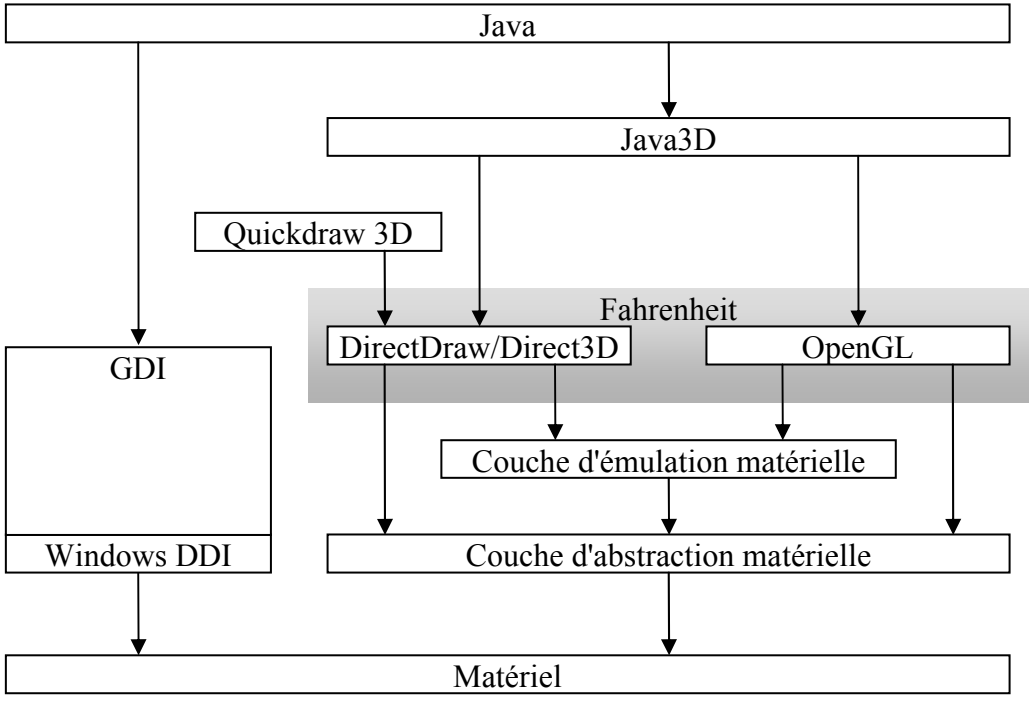
- Permet d'afficher l'image d'une scène 3D
- Interface avec un langage (C++, Java, Ada…)
- Différentes représentations de la scène

  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

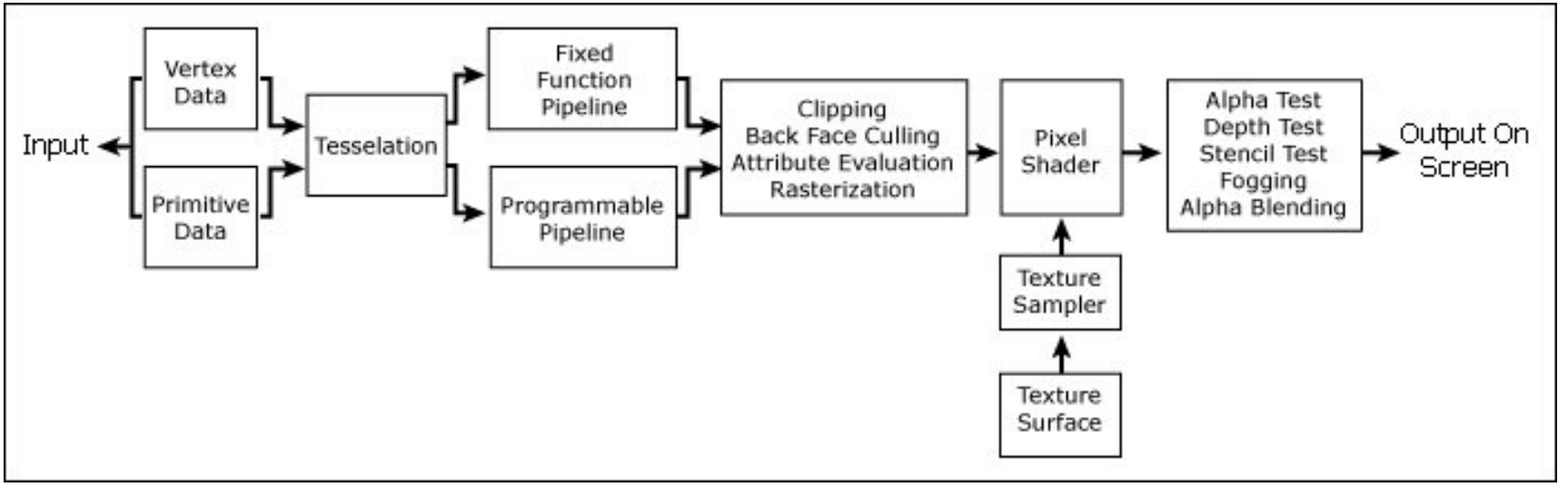
La 3D programmée
Les APIs 3D

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



La 3D programmée
Concepts (1)

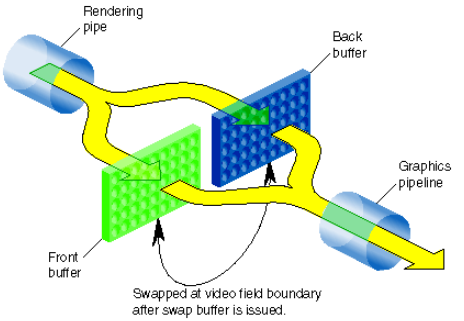
- Le pipeline de rendu



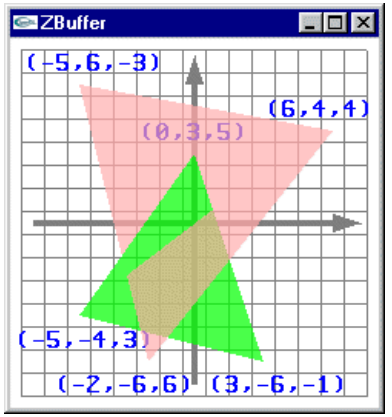
- Toutes les API 3D le prennent en charge

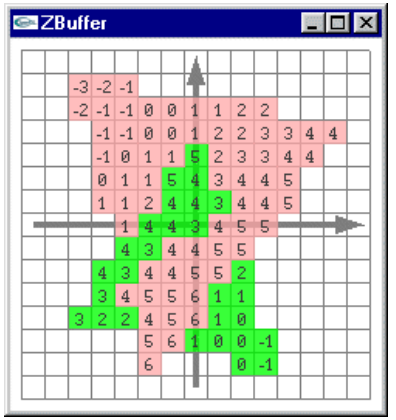
La 3D programmée
Concepts (2)

- *Framebuffer*: buffer contenant l'image affichée sur l'écran.
  Utilise le *Double-Buffering*:
  - *Front* buffer : image actuellement affichée
  - *Back* buffer : image en cours de calcul



- Plusieurs composantes:
  - *Color Buffer*: Couleurs de l'image (composante R G B)
  - *Z Buffer*: Distances (calcul des parties cachées)
  - *Stencil Buffer*
    Alpha Buffer
    Accumulation Buffer





La 3D programmée
Concepts (3)

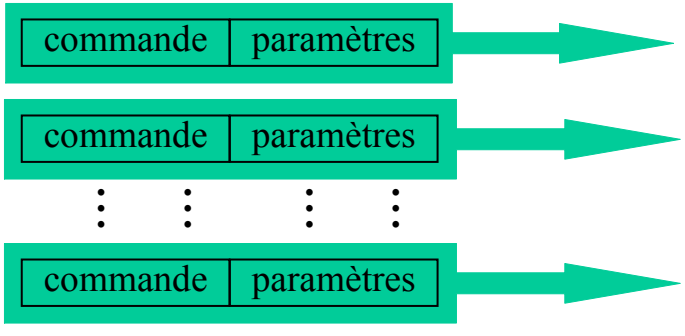
- *Pixel*: point d'une image
- *Vertex*: sommet d'un objet. Possède une position 3D et des attributs facultatifs:
  - *Normal*: vecteur normal au sommet
  - *Color*: couleur (et transparence)
  - *TexCoord*: coordonnée de mapping de la texture
- *Polygone*: primitive d 'affichage. Liste de *n* points (généralement des triangles).
- *Objet*: ensemble de polygones

La 3D programmée
Concepts (4)

Deux modes de transmission des données à l'API

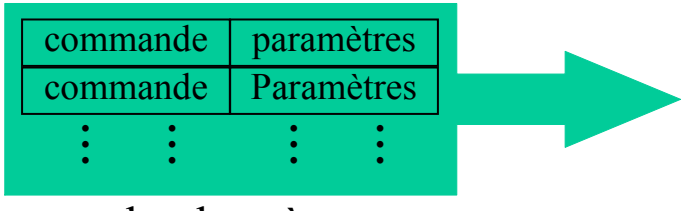
Mode immédiat

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

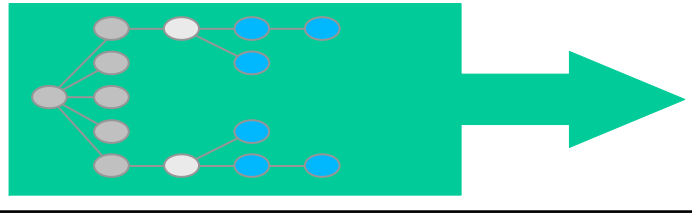


Mode retenu

- 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

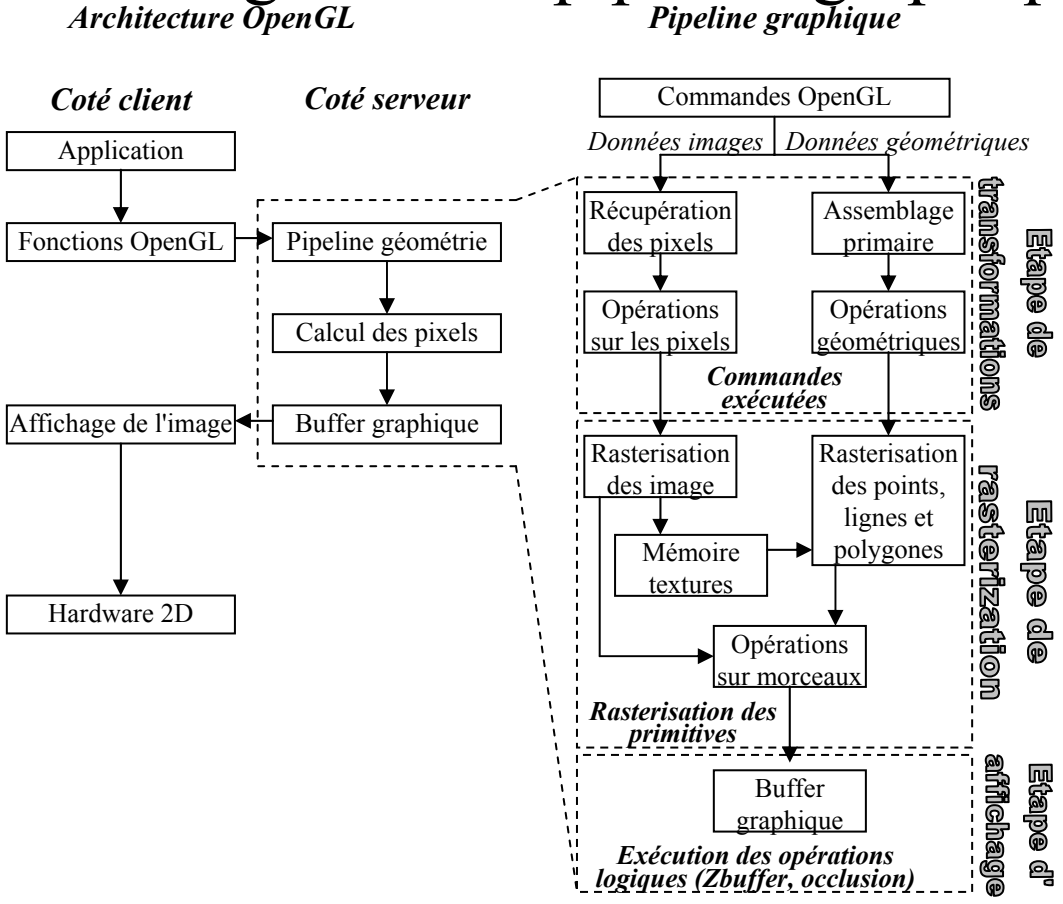


  - graphe de scène



La 3D programmée OpenGL/Mesa

Architecture logicielle et pipeline graphique



La 3D programmée OpenGL/Mesa

- Librairies d'outils pour deux aspects :
  - La gestion des événements et des fenêtres :
    - GLX permet de relier une application OpenGL avec Xwindow
    - AUX et GLW pareil pour Windows
    - 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 : tasselation, 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 */
 /* Initilisation 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_POLYGONE);
 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
    - Création de la (des) listes par *glGenLists*

      ```
      monObjet = glGenLists(3);
      ```

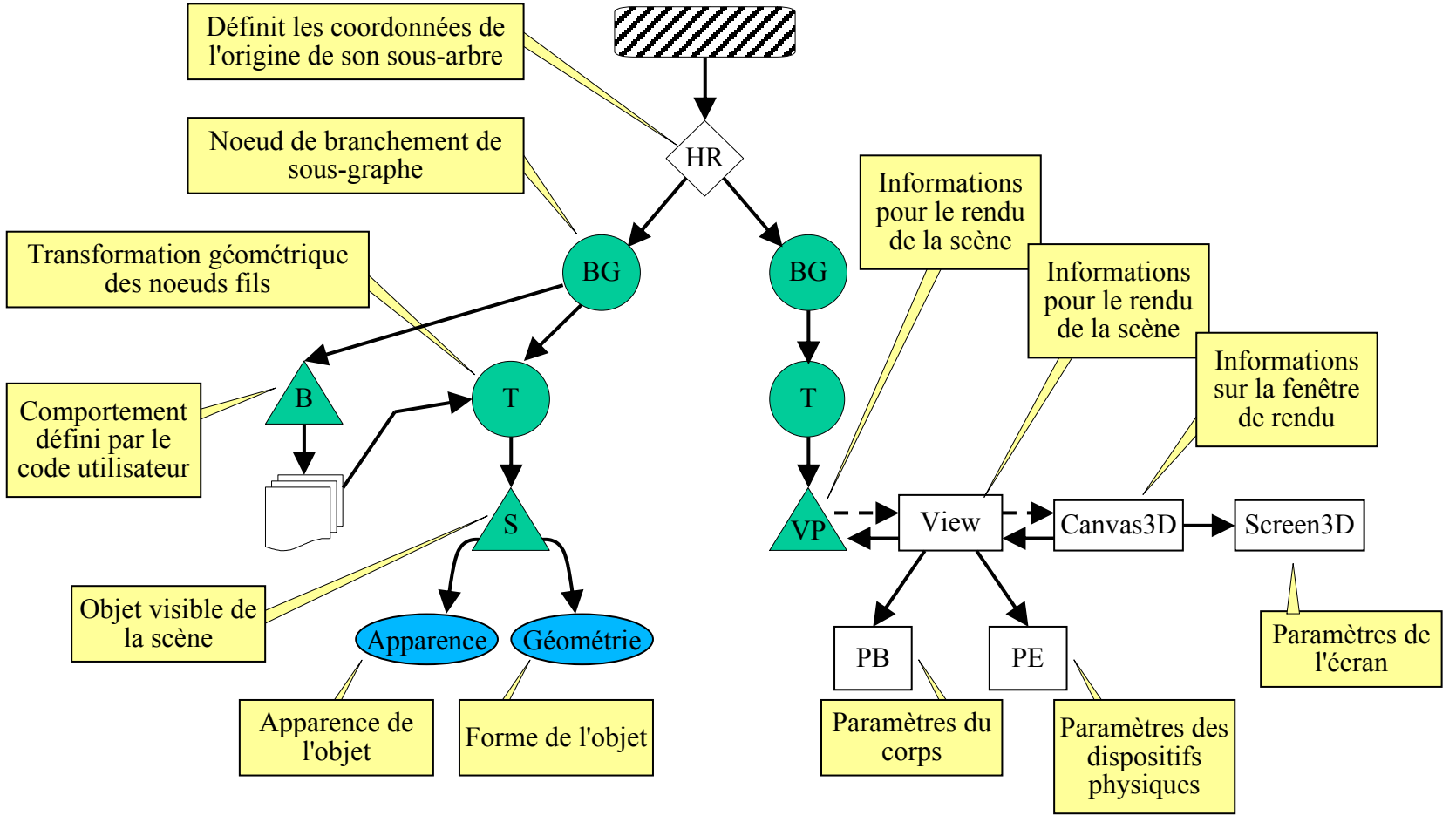
    - Début de remplissage de la liste par *glNewList*

      ```
      monObjet = glNewList(1, 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
Java 3D

1. Création d'un objet *Canvas3D* que l'on ajoute à un objet *Panel* d'une Applet ou d'une application autonome

```
Canvas3D c = new Canvas3D(null);
add("Center", c);
```

BG

2. Création d'un objet *BranchGroup* servant à regrouper les noeuds constituant la scène

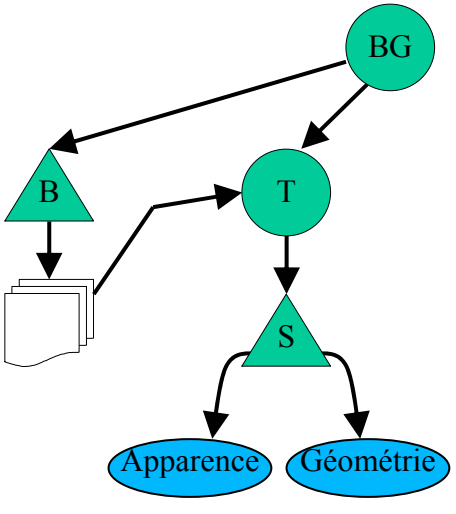
```
Branchgroup objRoot = new BrancGroup();
```

Canvas3D

La 3D programmée
Java 3D

3. Construction des objets de la scène avec des objets *Shape3D* regroupés en fonction des transformations qu'ils subissent dans des noeuds *TransformGroup*

```
TransformGroup objTrans = new TransformGroup();
objRoot.addChild(objTrans);
objTrans.addChild(new ColorCube());
Transform3D yAxis = new Transform3D();
```



4. Création des noeuds *Behaviour* et des codes utilisateurs modifiant les noeuds *TransformGroup*

```
Alpha ra = new Alpha(-1, Alpha.INCREASING_ENABLE, 0, 0,
4000, 0, 0, 0, 0, 0);
RotationInterpolator ri = new RotationInterpolator (ra,
objTrans, yAxis, 0.0, (float)Math.PI*2.0);
BoundingSphere bounds = BoundingSphere(
                    new Point3d(0.0, 0.0, 0.0),
100.0);ri.setSchedulingBounds(bounds);
objTrans.addChild(ri);
```

Canvas3D

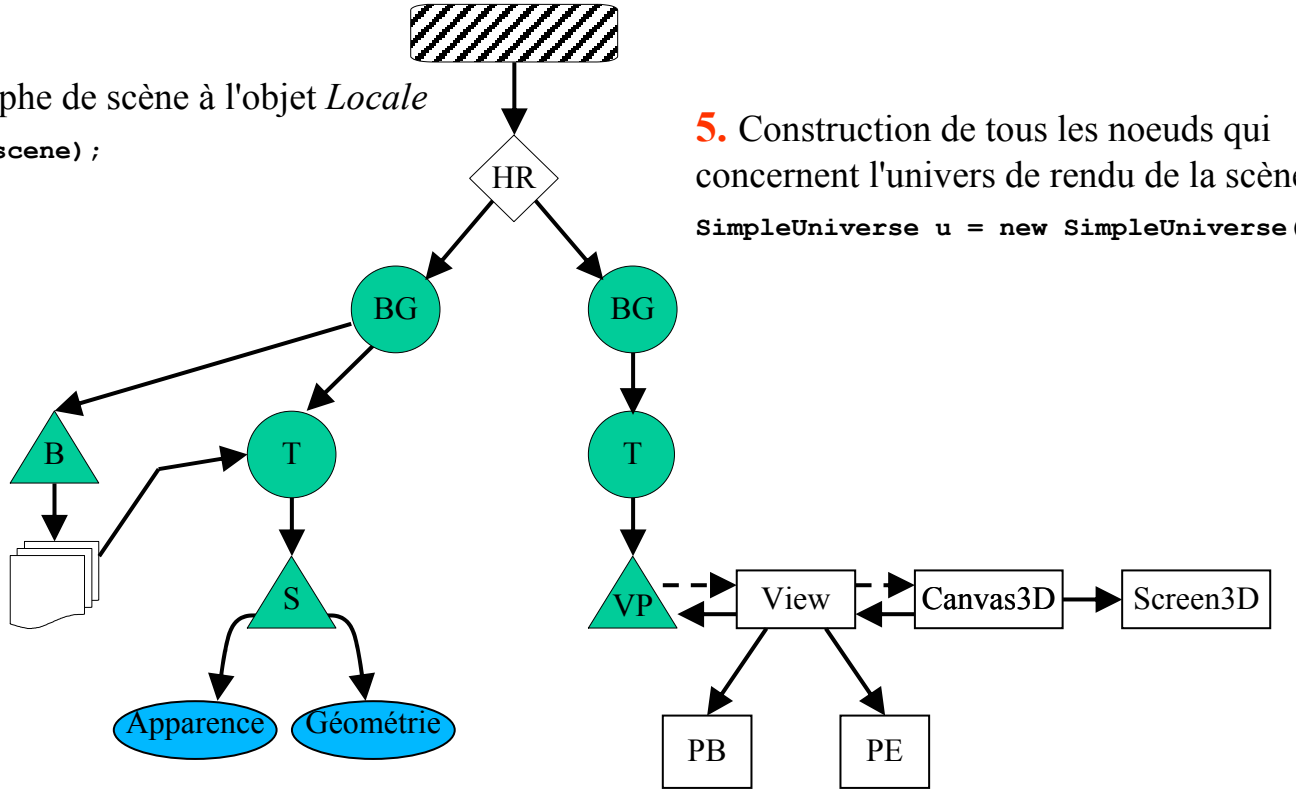
La 3D programmée
Java 3D

6. Insertion du graphe de scène à l'objet *Locale*

```
u.addBranchGraph(scene);
```

5. Construction de tous les noeuds qui concernent l'univers de rendu de la scène

```
SimpleUniverse u = new SimpleUniverse(c);
```



La 3D programmée
Java 3D

1. Création d'un objet *Canvas3D* que l'on ajoute à un objet *Panel* d'une Applet ou d'une application autonome
2. Création d'un objet *BranchGroup* servant à regrouper les noeuds constituant la scène
3. Construction des objets de la scène avec des objets *Shape3D* regroupés en fonction des transformations qu'ils subissent dans des noeuds *TransformGroup*
4. Création des noeuds *Behaviour* et des codes utilisateurs modifiant les noeuds *TransformGroup* (pour une rotation automatique de la scène, on pourra par exemple utiliser un noeud *RotationInterpolator* qui effectue des rotations d'un angle donné tous les intervalles de temps donnés)
5. Construction de tous les noeuds qui concernent l'univers de rendu de la scène :
    a. Création de l'objet *VirtualUniverse* et du ou des objets *Locale*
    b. Création des objets *PhysicalBody*, *PhysicalEnvironment*, *View* et *ViewPlatform*
    c. Création de l'objet *BranchGroup* auquel on attache un noeud *TransformGroup* qui contient lui même le graphe de vue créé précédemment
    d. Insertion de ce noeud *BranchGroup* à l'objet *HiResLocale*
6. Insertion du graphe de scène à l'objet *Locale*

Ce que fait la méthode SimpleUniverse

La 3D programmée
Direct 3D

- Très proche d'OpenGL dans les principes
  - C et C++
  - Modes retenu et immédiat
  - Accélérations graphiques matérielles
- Direct3D ∈ 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

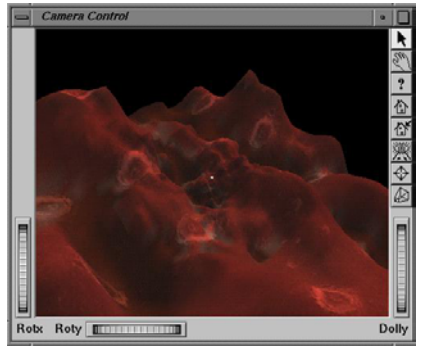
- Bibliothèque moyen niveau professionnelle sur SGI/Linux
- Gère le parallélisme (plusieurs pipelines)
- Gestion de scènes complexes
- Charge de nombreux formats de fichiers 3D
- Graphe de scène
- Crée et gère ses fenêtres

`www.sgi.com/software/performer/`

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)

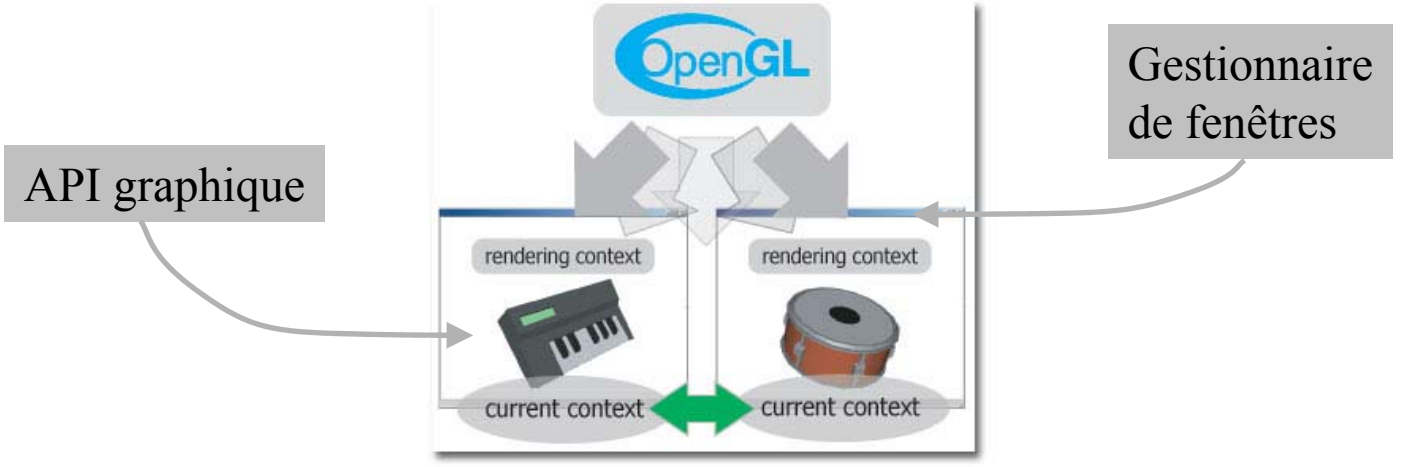
- Application graphique performante

  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 a 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

Création d'une fenêtre OpenGL

- Windows + GLAux

```
winMain … createWindow …
createContext …
```

- XWindow + GLX

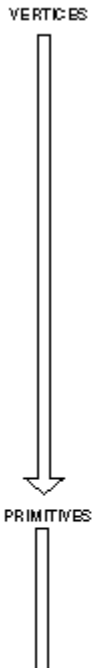
```
GLXCreateContext …
```

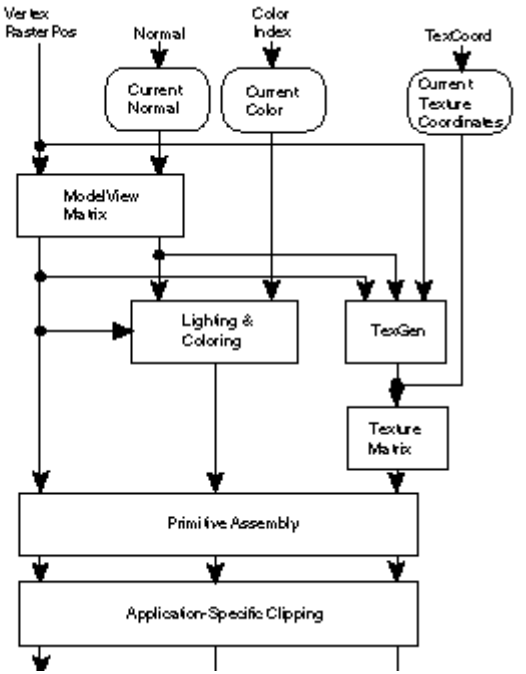
- GLUT (GL Utility Toolkit)

```
int glutCreateWindow(char* name);
glutDisplayFunc(display);
glutMainLoop();
```

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);
```

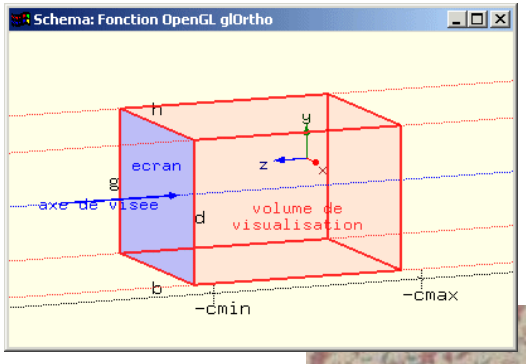
Transformations

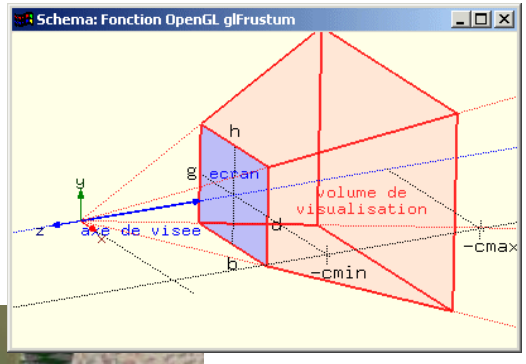
- Chaque vertex subit des transformations avant d'être affiché
  - Position et orientation de l'objet dans la scène
  - Inverse de la position et l'orientation de la caméra
  - Projection 3D → 2D
- Les 2 premières transformations sont stockées dans la matrice *MODELVIEW*
- La projection est stockée dans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

Description de la caméra

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(…); ou glFrustum(…);
```

Ou `gluPerspective(); et gluLookAt(…);`





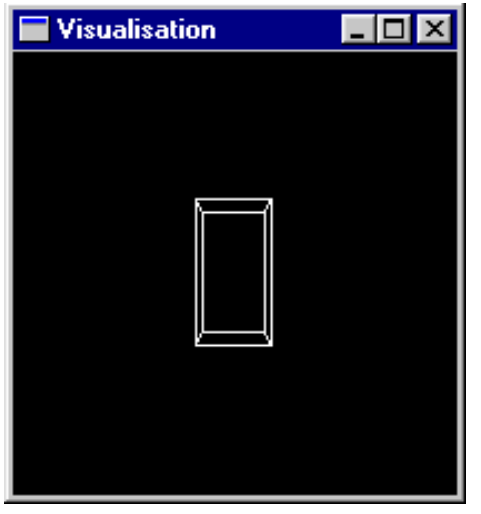
Positionnement des objets

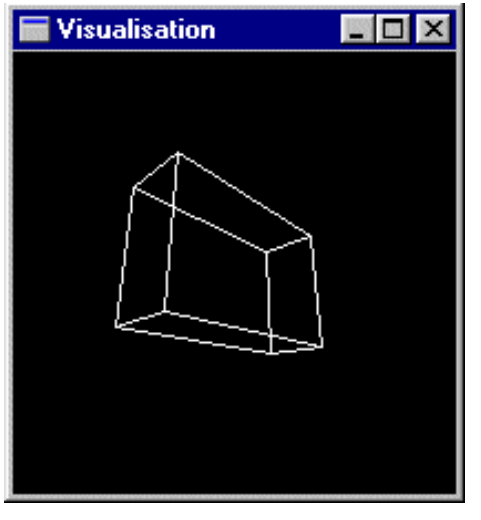
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(x,y,z);
glRotatef(alpha, x,y,z);
glScalef(sx, sy, sz);
```

OU `glMultMatrixf(m);`

```
glBegin(…);
…
glEnd();
```

Ordre inversé !





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

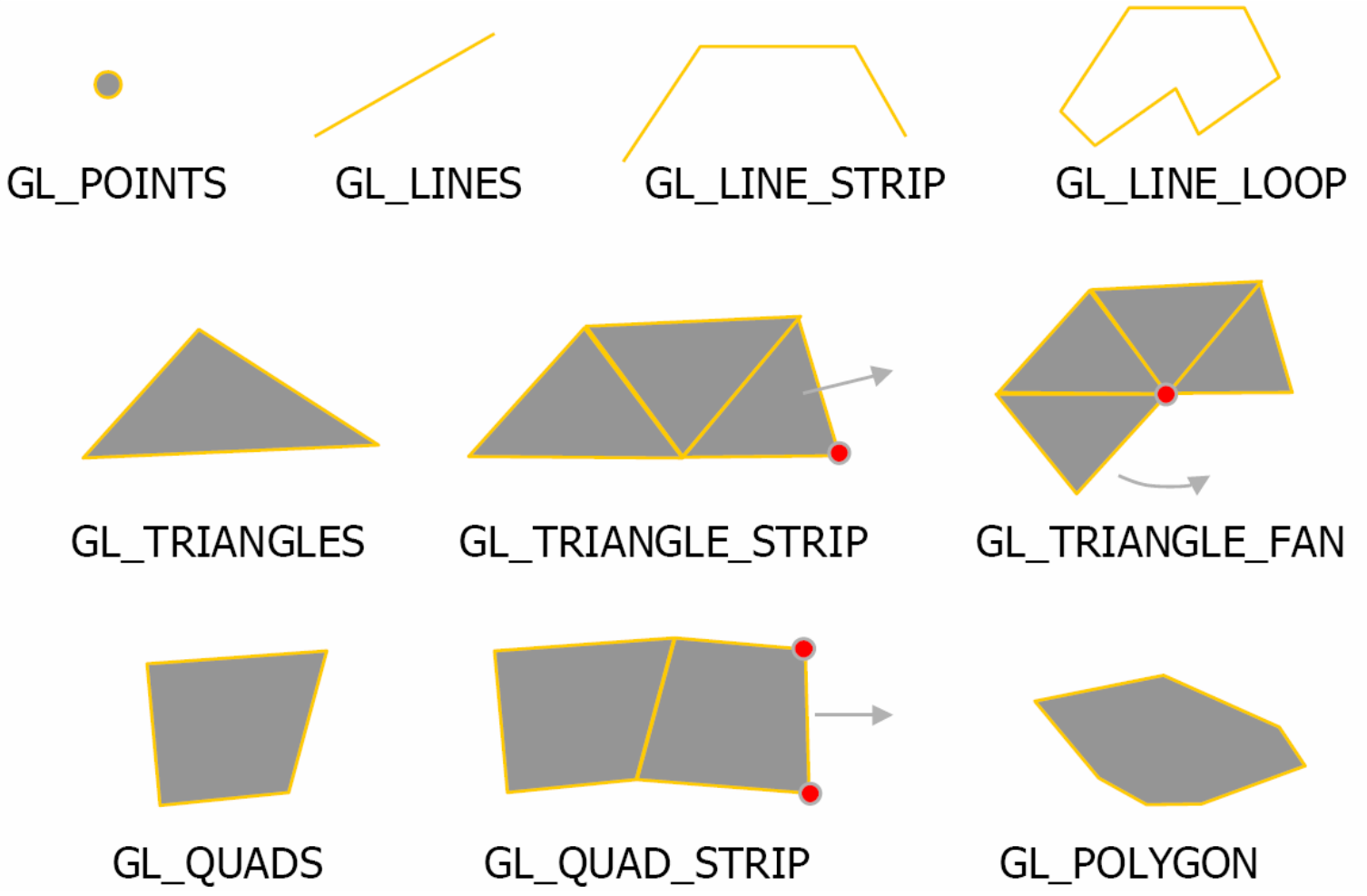
P(x,y,z) = P * T1 * R1 * R2 * T2 * (x,y,z)

- 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(…)`



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

```
// render_triangle.c
#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)

```
// main_glut.c
#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)

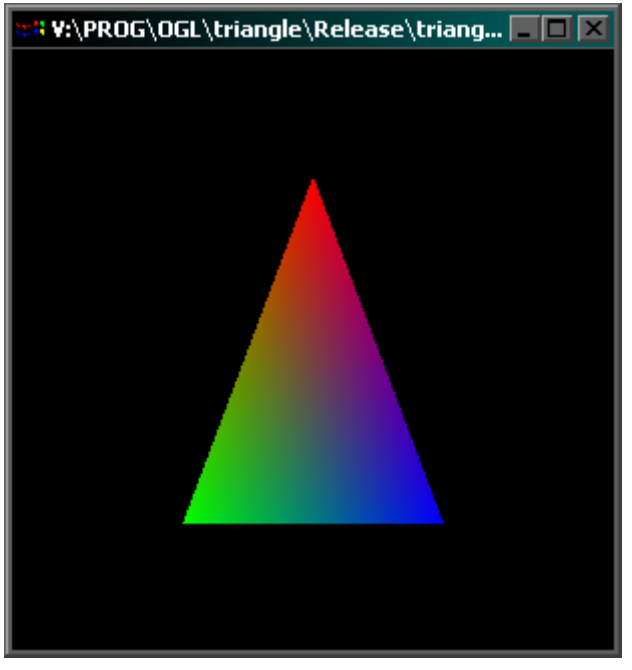
```
void init() { // initialise la caméra
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60, 1.3333, 1.0, 5000.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void display() { // calcule et affiche une image
  render();
  glutSwapBuffers();
}
void idle() { // reaffiche une fois terminé
  glutPostRedisplay();
}
```

Exemple
Compilation avec GLUT

- Compilation:

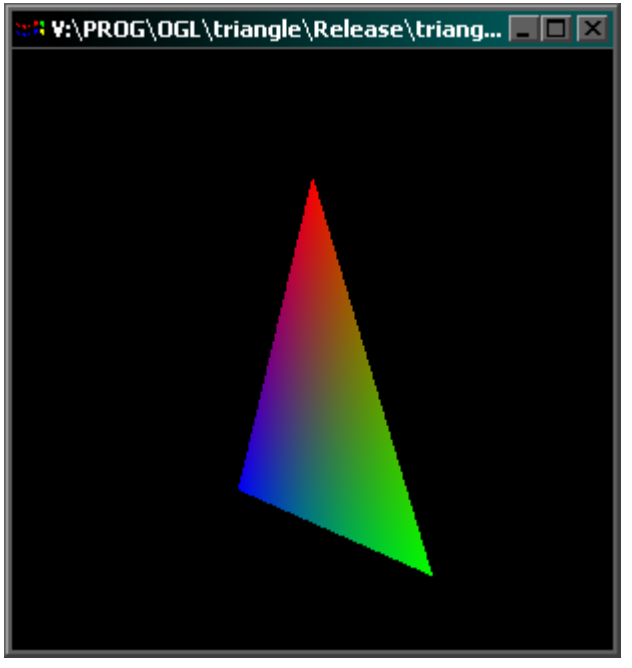
```
gcc -o triangle main_glut.c render_triangle.c -lglut -
   lGLU -lGL -lXmu -lXext -lX11 -lm
```

- 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();
```

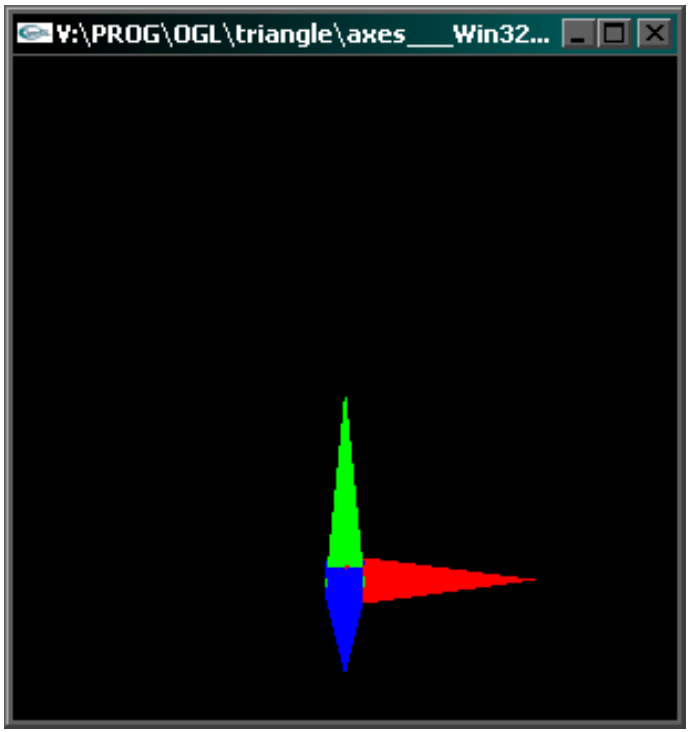


Objets
Exemple procédural

```
void renderAxeX() {
   glBegin(GL_QUADS); // base
        glVertex3f(0,-0.1f,-0.1f);
        glVertex3f(0, 0.1f,-0.1f);
        glVertex3f(0, 0.1f, 0.1f);
        glVertex3f(0,-0.1f, 0.1f);
   glEnd();
   glBegin(GL_TRIANGLE_FAN); // cotés
        glVertex3f(1,0,0);
        glVertex3f(0,-0.1f,-0.1f);
        glVertex3f(0, 0.1f,-0.1f);
        glVertex3f(0, 0.1f, 0.1f);
        glVertex3f(0,-0.1f, 0.1f);
        glVertex3f(0,-0.1f,-0.1f);
   glEnd();
}
```

Objets
Exemple procédural (2)

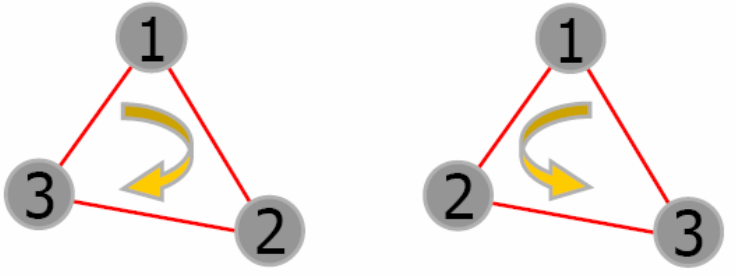
```
void render() {
   glEnable(GL_DEPTH_TEST);
   glPushMatrix();
   glTranslatef(0,-1,-3);
   glRotatef(nbframe,0,1,0);
   glClear(…);
   glColor3f(1,0,0);
   renderAxeX(); // axe X
   glRotatef(90,0,0,1);
   glColor3f(0,1,0);
   renderAxeX(); // axe Y
   glRotatef(90,0,-1,0);
   glColor3f(0,0,1);
   renderAxeX(); // axe Z
   glPopMatrix();
   ++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);
```

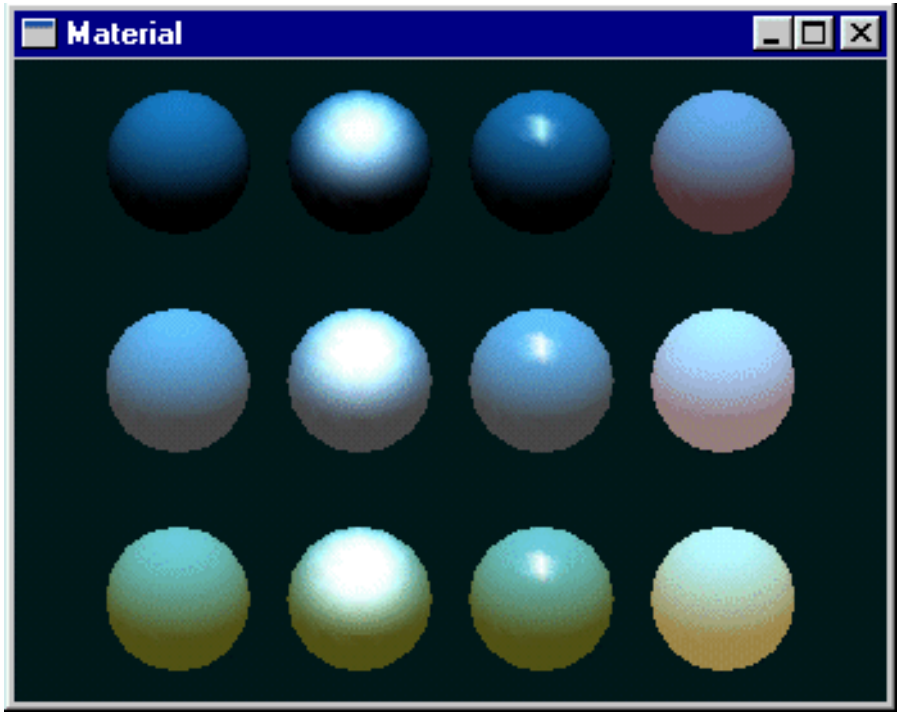
mode = GL_FILL, GL_LINE

```
glLineWidth(3);
glPointSize(12);
```

Matériaux des objets

- Couleur ambiente, diffuse, spéculaire, émission
- Pour les faces avant et /ou arrière

```
glMaterialfv(GL_FRONT, GL_SPECULAR, spec);
```



glColorMaterial

face = `GL_FRONT`, `GL_BACK`, **`GL_FRONT_AND_BACK`**

material = `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, **`GL_AMBIENT_AND_DIFFUSE`**

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(face, material);
glColor3f(r,g,b);
```

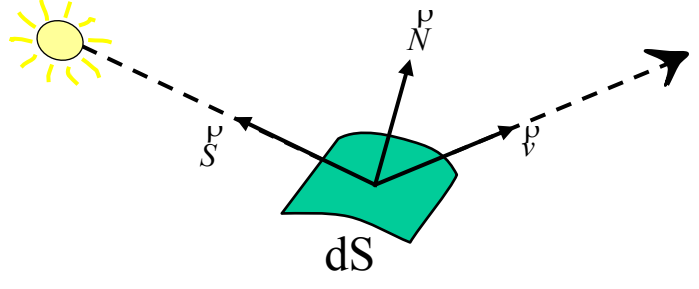
- `glColor` court-circuite `GL_AMBIENT` et `GL_DIFFUSE` lorsque `GL_LIGHTING` est activé
- Et est utilisé lorsque GL_LIGHTING est désactivé !

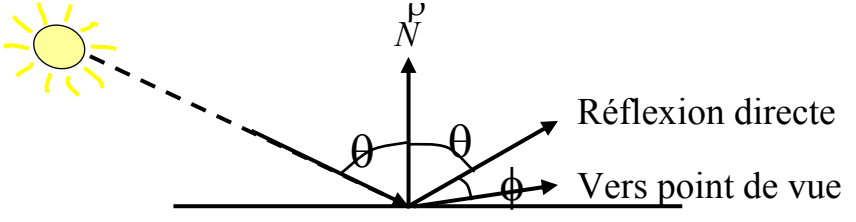
Lumières

- Sources à l'infini, directionelles, spots
- Normales du modèle nécessaires

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

- Objet comme les autres
- → subit la MODELVIEW

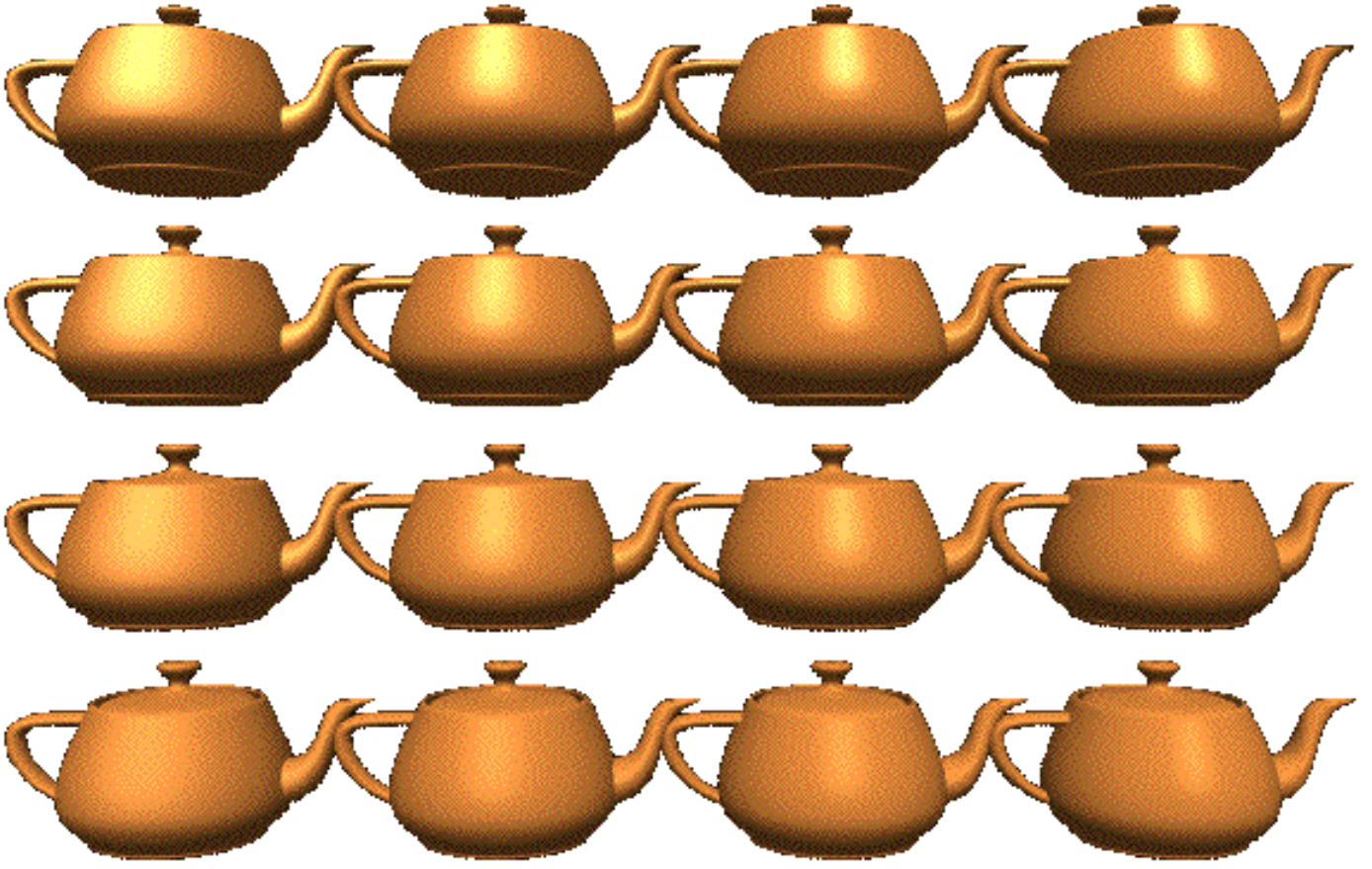




Matériaux et Lumières

- OpenGL supporte le model de lumière Phong
  $C = Ca + Cd.(N.L) + Cs.(N.K)^{shininess}$
  - C : couleur finale
  - Ca : couleur ambiante
  - Cd : couleur diffuse
  - Cs : couleur specular
  - N : vecteur normal de la surface
  - L : vecteur unitaire reliant le vertex a la lumiere
  - K : 1/2 vecteur (vecteur au milieu de L et du vecteur reliant la vertex a la caméra).
  - Shininess: puissance du spécular

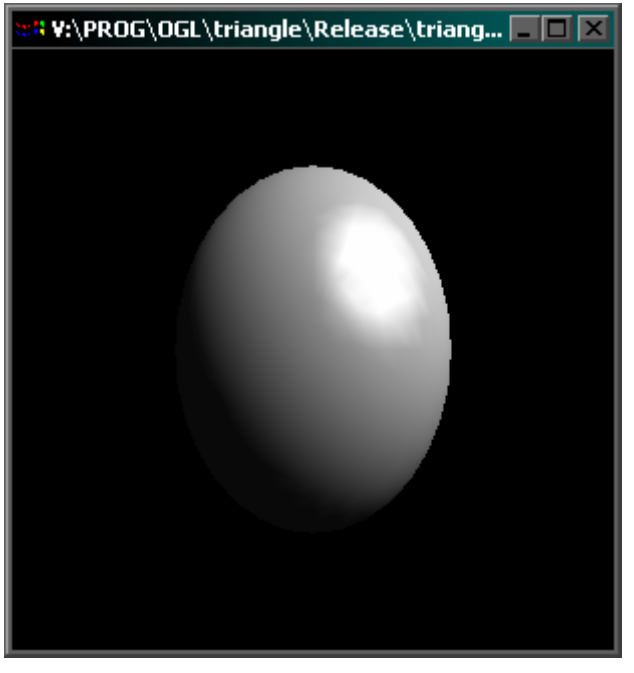
Matériaux et Lumières



Matériaux et Lumières
Exemple

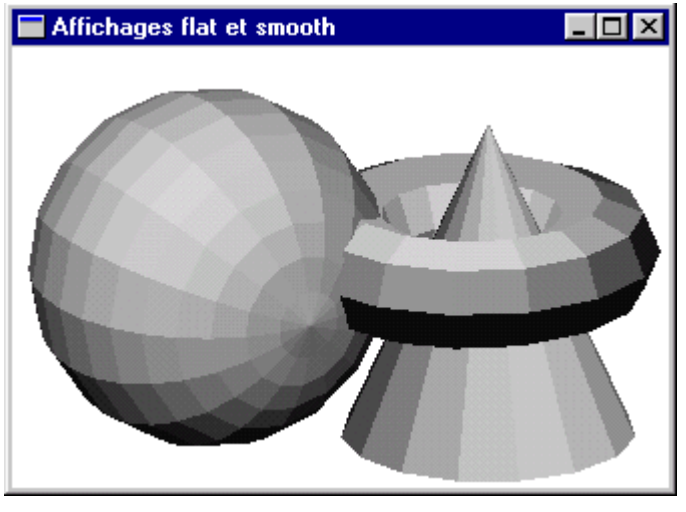
```
…
  glPushMatrix();
  glRotatef(nbframe,1,0,0);
  glRotatef(nbframe/2,0,0,1);
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glPopMatrix();

  GLUquadricObj* sph = gluNewQuadric ();
  gluQuadricOrientation(sph, GLU_OUTSIDE);
  gluSphere(sph, 1, 32,32);
  gluDeleteQuadric(sph);
…
```

Matériaux et Lumières
Exemple (2)



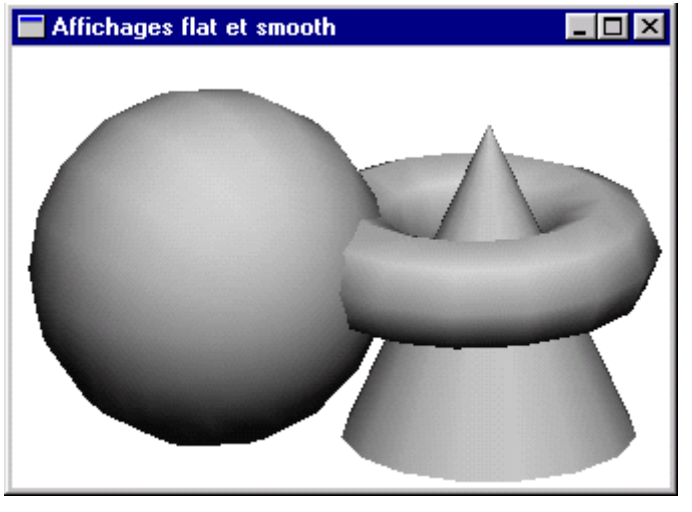
Interpolation des couleurs

Coloriage à plat
Une couleur par face




```
glShadeModel(GL_FLAT);
```

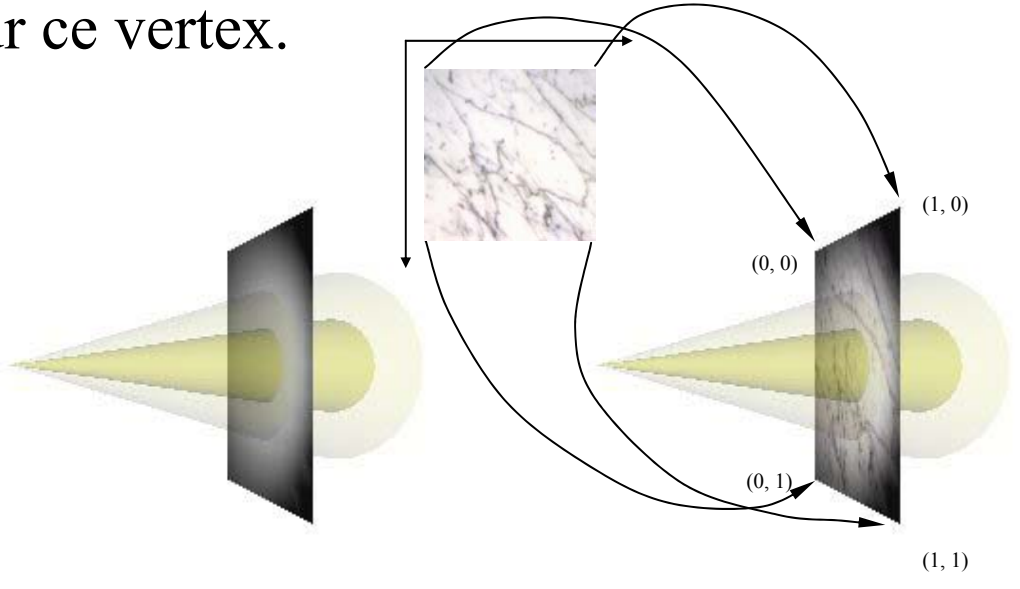
Coloriage gouraud
Interpolation des couleurs




```
glShadeModel(GL_SMOOTH);
```

Textures

- Une texture est une image pouvant être appliqué a un objet
- Elle est en général chargée à partir d'un fichier
- A chaque vertex il faut spécifier les coordonnées de mapping (2 composantes u et v) spécifiant le point de la texture appliqué sur ce vertex.



Textures
Exemple

```
#include "tga.h"
unsigned int textureID = 0;
void initgl() {
   glGenTextures(1, &textureID); // crée un ID
   loadTGA("texture.tga",textureID); // chage la texture
}
void render() {
… glBindTexture(GL_TEXTURE_2D, textureID);
  glEnable(GL_TEXTURE_2D);
  glBegin(GL_QUADS);
    glColor3f(1.0f,1.0f,1.0f);      glTexCoord2f(1,1);
    glVertex3f( 1.0f, 1.0f, 0.0f);
    glColor3f(1.0f,0.0f,0.0f);      glTexCoord2f(0,1);
    glVertex3f(-1.0f, 1.0f, 0.0f);
    glColor3f(0.0f,1.0f,0.0f);      glTexCoord2f(0,0);
    glVertex3f(-1.0f,-1.0f,0.0f);
    glColor3f(0.0f,0.0f,1.0f);      glTexCoord2f(1,0);
    glVertex3f( 1.0f,-1.0f,0.0f);
  glEnd();
  … }
```

Textures
Exemple (2)

Transparence α=opacité

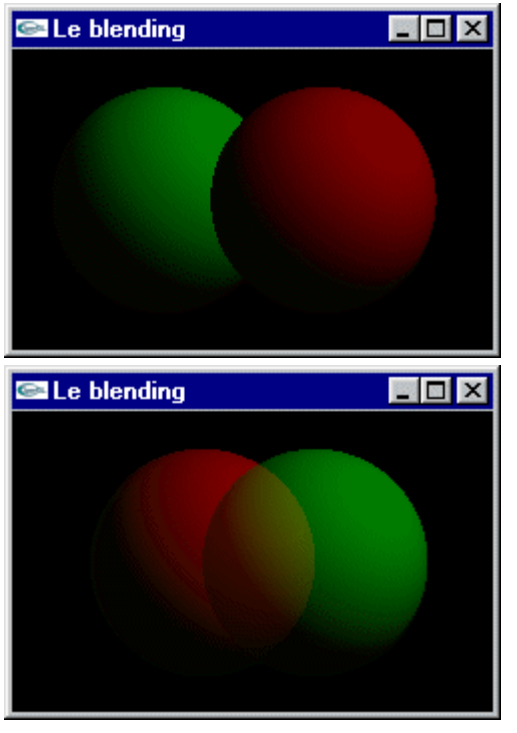
- Combinaison des couleurs

  Fragment dessiné c1=(r1,g1,b1,α1)

  Couleur actuelle du pixel c2=(r2,g2,b2,α2)

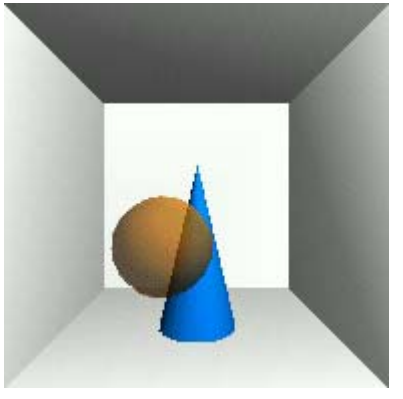
$$c = \alpha_1 c_1 + \alpha_2 c_2$$

- Différentes fonctions de mélange
- `glBlendFunc(`*source*, *dest*`);`
- `glEnable(GL_BLEND);`



Transparence : difficultés

- Limitations
  - Approximation de la physique
  - Rendu de l'arrière vers l'avant
    Sauf si une seule épaisseur est transparente
- Mauvaise fonction de mélange par défaut !
- Utiliser généralement :



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

Les buffers

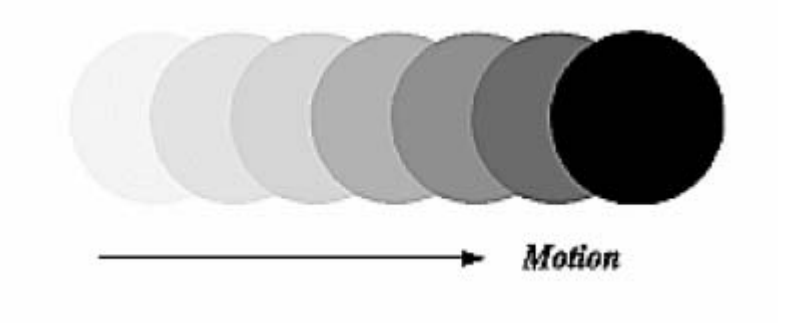
- Frame-buffer : couleur (r,g,b,[a])

  Simple ou double, stéréo

  ```
  glDrawBuffer(GL_FRONT ou GL_BACK, …);
  ```

- Profondeur (Z-buffer)
- Pochoir (Stencil buffer)
- Accumulation
- Sous-fenêtrage

  ```
  glScissor(…); glViewport(…);
  ```



Anti-aliassage

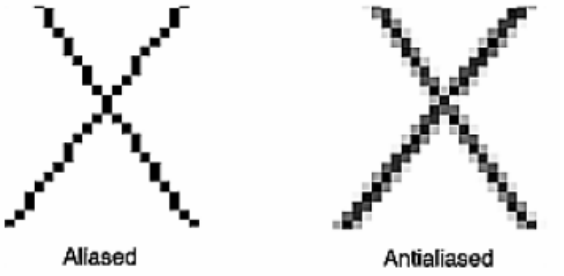
- Utilisation du canal alpha

  `glEnable(GL_{POINT, LINE, POLYGON}_SMOOTH);`

- Suréchantillonnage

  Rendu d'une fenêtre plus grande

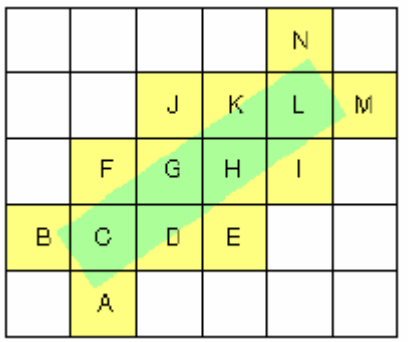
  Anti-aliassage en hardware



- Nvidia sur Linux

  `setenv __FSAA_MODE n (n ∈ 1..5)`

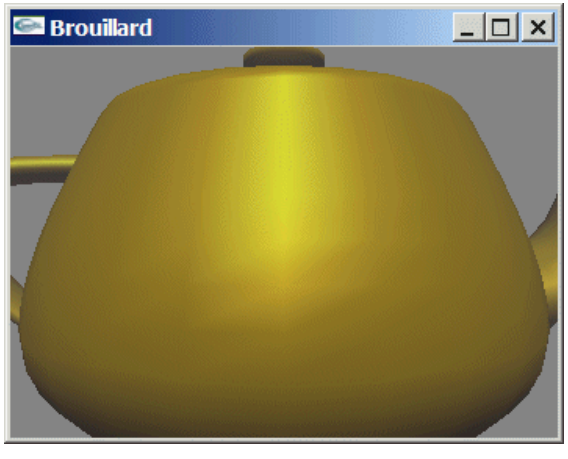
  `glEnable(GL_MULTISAMPLE_ARB);`

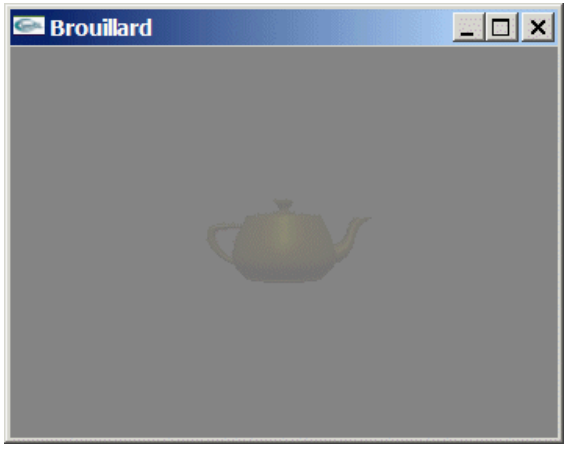


Brouillard

- Limite le nombre de primitives à afficher
  C'est un LOD (*Level Of Detail*) automatique
- Accentue l'effet de profondeur

```
fogMode = GL_EXP;
glFogi(GL_FOG_MODE,fogMode);
glFogfv(GL_FOG_COLOR,couleurGrisMoyen());
glFogf(GL_FOG_DENSITY,0.3F);
```





Les formats de fichiers

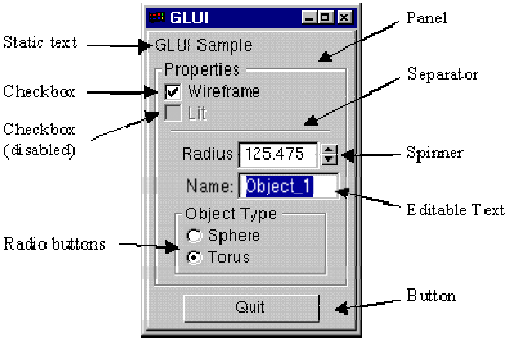
- Modèles 3D
  - Le possible futur : X3D
  - VRML (1 ou 2), WRL, 3DS, smf, pfb, obj, dxf,…
  - “3D Graphics File Formats : A Programmer's Reference”, K. Rule
  - `www.3dcafe.com`
- Formats d’images
  - RGB, JPEG, TIF, …
  - Puissances de 2 dans les tailles des textures
- Avec OpenGL, il faut tout programmer soit même

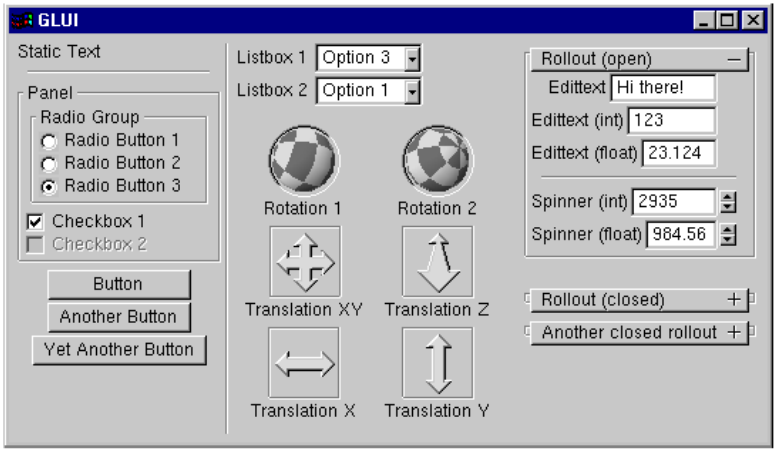
Interface Utilisateur (GUI)

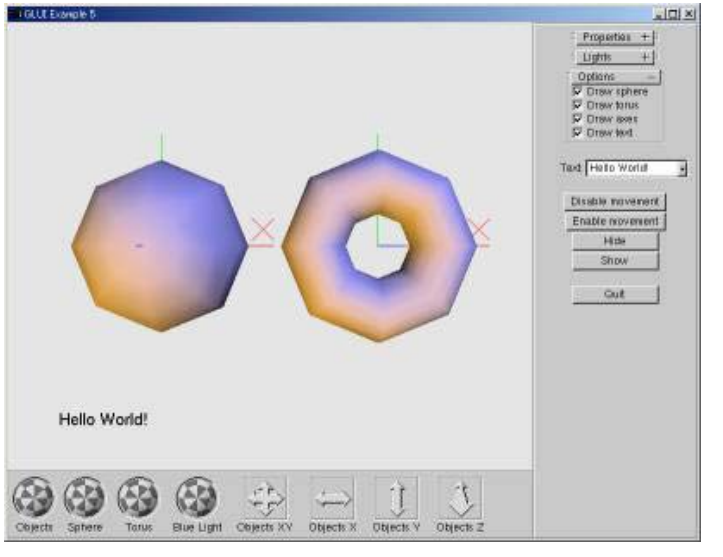
- GLUI
  - Simple mais pas objet

    (void*)

    Variables globales
  - Windows et Unix-Linux

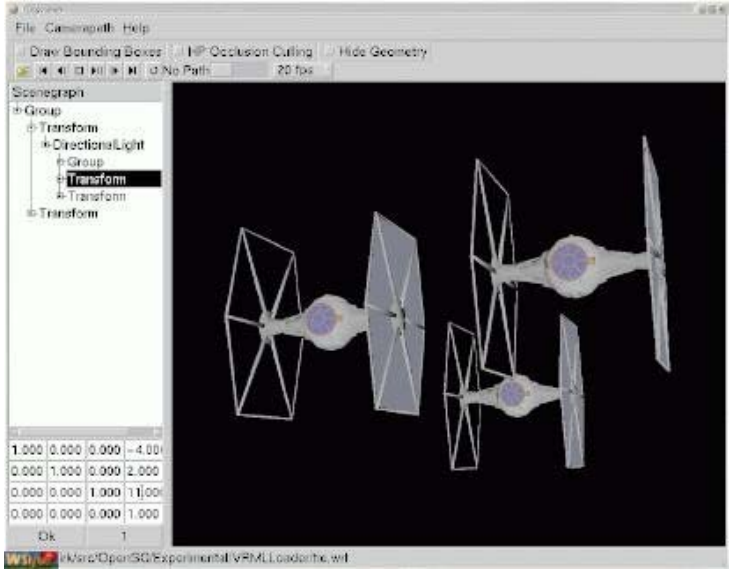






Interface Utilisateur (GUI)

- Qt
  - Multi-plateformes (Unix, Windows, Mac, …)
  - qmake (Makefile), designer (UI à la souris)
  - Passage de messages entre objets



Interface Utilisateur (GUI)

- Windows
- Interface integrée Windows

  Component Object Modeling

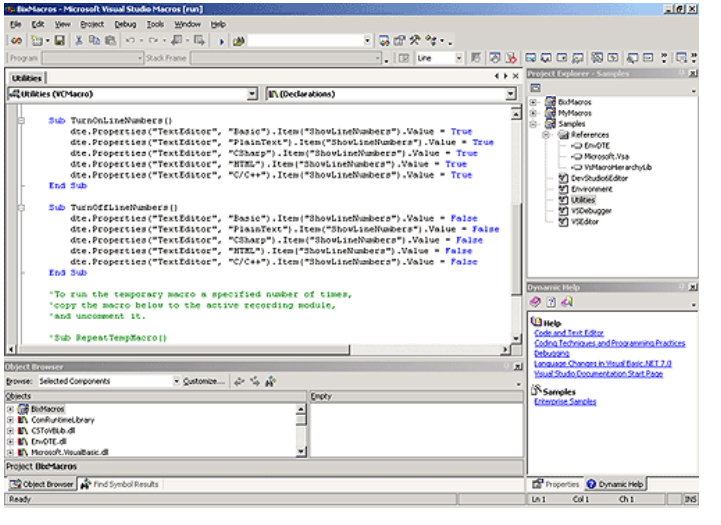
  Communication par messages

  Menus, cases à cocher, souris, clavier…

- Deux systèmes possibles

  Microsoft Fondation Classes

  Win32



Références

- OpenGL Programming Guide (The Red Book)

  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://www.opengl.org : site officiel. Forums
- http://nehe.gamedev.net : le site de tutoriels OpenGL