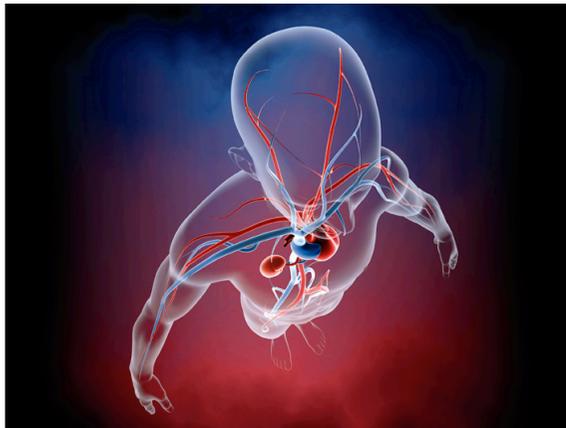
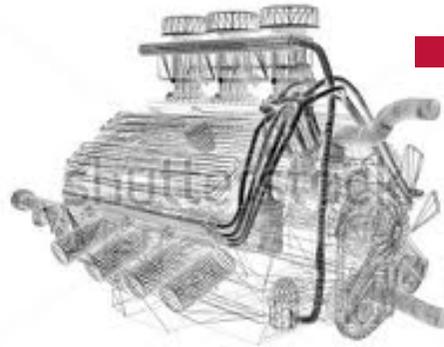




Synthèse Graphique

PACT – Mini-Cours

Pour qui ?



■ Divertissement

- Jeux: FPS, arcades, plateaux
- Simulation (arts plastiques, musique, ..)

■ Industriel

- CAO
- Tests et simulations
- Impression 3D

■ Tertiaire

- Médical
- Aérospatial
- Architecture
- ...





Modéliser un environnement 3D

■ **Objet: description mathématique 2D/3D**

- **Forme**
 - Personnages
 - Décors
 - Terrain
- **Aspect**
 - *Textures, couleurs, modèle de lumière*

■ **Scène: ensemble des objets**

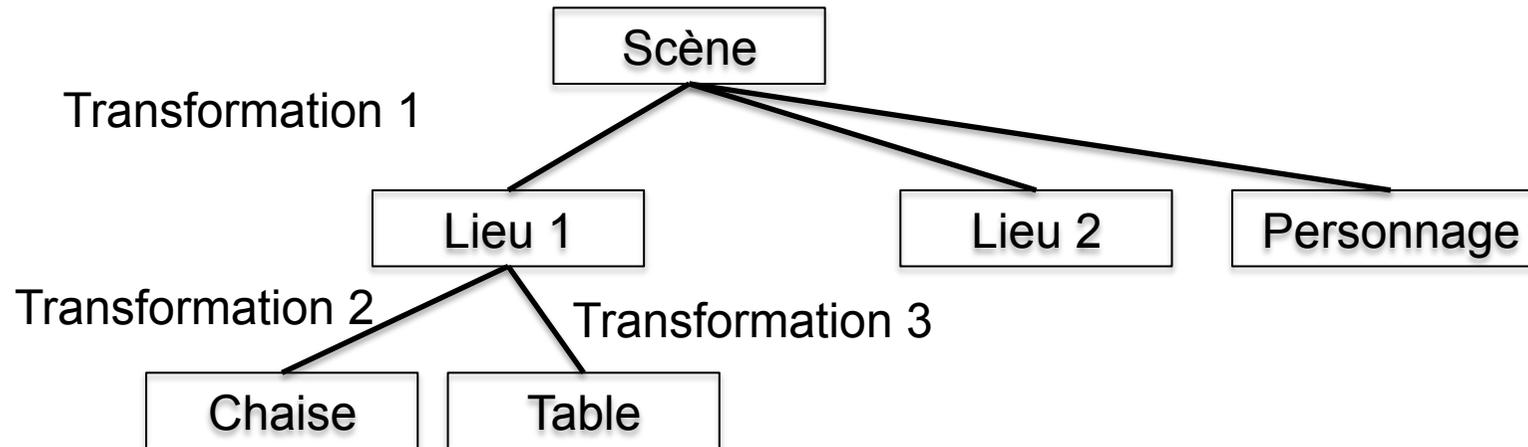
■ **Caméra: point de vue dans la scène**

- Unique pour tous les objets

■ ***Modèle pour simulation physique***

- *Poids, élasticité des matériaux, ...*

Arbre de scène et Liste d'affichage



Liste Graphique, dans l'ordre d'affichage:

- Transformation 1 + Transformation 2 + Chaise
- Transformation 1 + Transformation 3 + Table
- Transformation 1 + Transformation X + Personnage

Dans la pratique, on utilise souvent un mélange des deux:

Arbre de scène pour décrire l'ensemble

Liste Graphique pour les objets à afficher à T

Les objets

■ Primitive: description mathématique 2D/3D

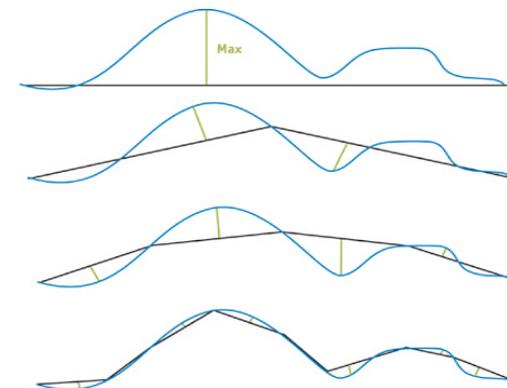
- Point: « Vertex »
- Primitives simples
 - Point, Ligne, Triangles, Carrés
- Primitives complexes:
 - courbes/surfaces de Bézier, d'ordre 2 ou 3

$$B(t) = (1 - t)^2 \mathbf{P}_0 + 2t(1 - t) \mathbf{P}_1 + t^2 \mathbf{P}_2, t \in [0, 1].$$

$$B(t) = \mathbf{P}_0(1 - t)^3 + 3\mathbf{P}_1t(1 - t)^2 + 3\mathbf{P}_2t^2(1 - t) + \mathbf{P}_3t^3, t \in [0, 1].$$

— Puis *tessellation*

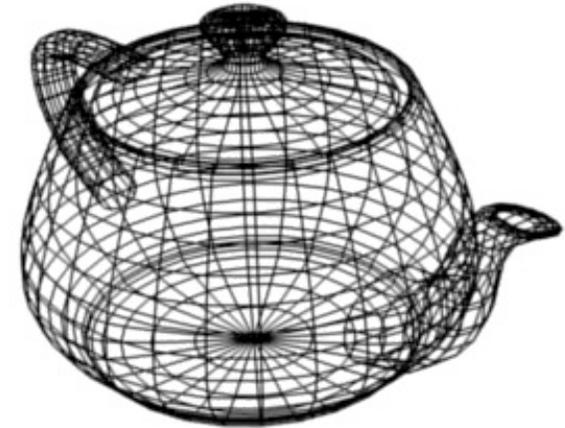
- Approximation linéaire par primitive simples
- Pour une « finesse » donnée
- Accélération matérielle très récente ...



Les objets

■ Objet

- Assemblage de ces primitives
 - Notion de « groupe »
 - Non déformable: même aspect à chaque trame
 - Ex: Chaise, table
 - Déformable
 - Animation de personnage
 - CAO Industrielle
- Via des transformation affines



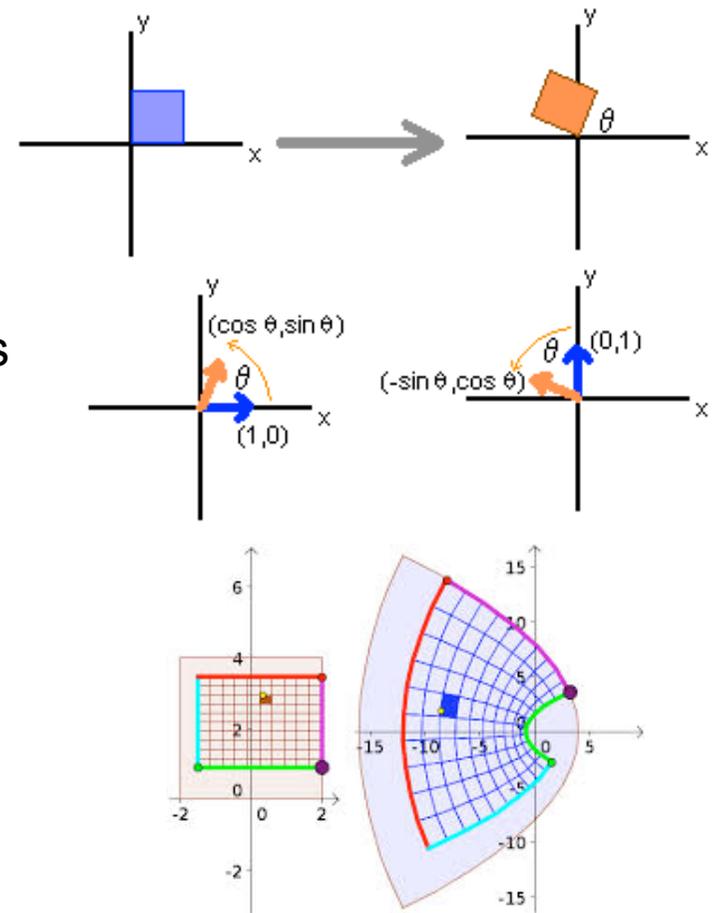
Les Transformations

■ Déplacer les objets

- Rotation, translation, mise à l'échelle
 - Calculs simples sur l'ensemble des points
- Transformations affines
 - 2D: $x' = ax + by + c$, $y' = dx + ey + f$
 - 3D: ...
- Transformations non linéaires
 - Modification point par point

■ Cumuler les transformations

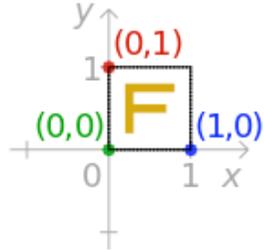
- $(x', y) = R(\alpha)(x, y) + T(a, b)(x, y) + S$
 $(2, 1)(x, y) + \dots$
- Représentation Matricielle:
 - $Tr = S * T * Ra$
 - $(x', y') = Tr * (x, y)$





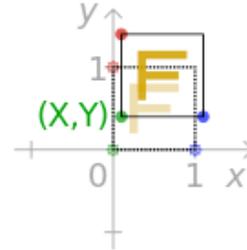
No change

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



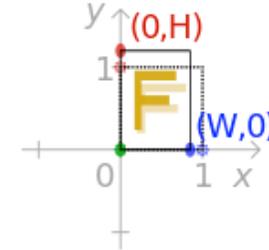
Translate

$$\begin{bmatrix} 1 & 0 & X \\ 0 & 1 & Y \\ 0 & 0 & 1 \end{bmatrix}$$



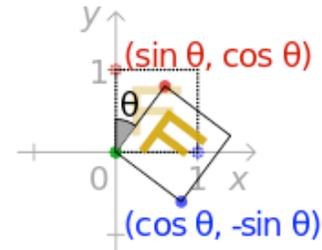
Scale about origin

$$\begin{bmatrix} W & 0 & 0 \\ 0 & H & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



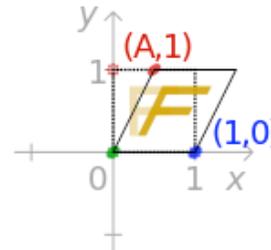
Rotate about origin

$$\begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



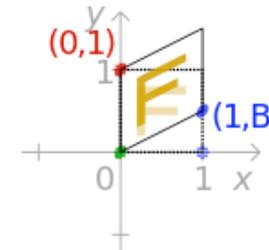
Shear in x direction

$$\begin{bmatrix} 1 & A & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



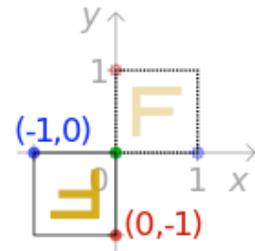
Shear in y direction

$$\begin{bmatrix} 1 & 0 & 0 \\ B & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



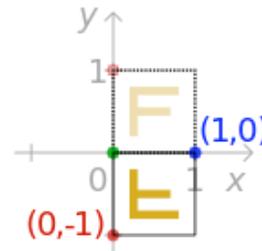
Reflect about origin

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



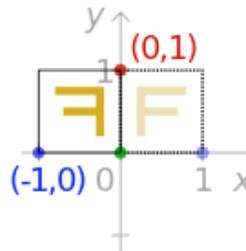
Reflect about x-axis

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Reflect about y-axis

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$





Systeme de projection

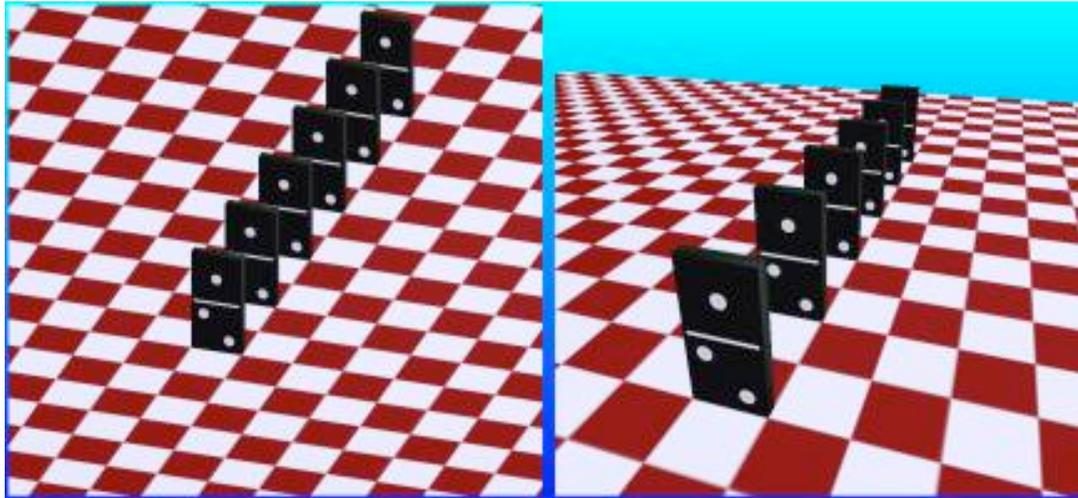
■ L'affichage est

- En 2D
- En pixels:
 - x, y dans \mathbb{N}^2
 - Origine haut/gauche ou bas/gauche

■ Comment passer de la description mathématique à une coordonnée en pixel ?

- Projection \mathbb{R}^2 ou $\mathbb{R}^3 \rightarrow \mathbb{R}^2$
 - Bien évidemment sous forme matricielle
 - Avec normalisation sur $[-1.0, 1.0]$ de l'espace visible
- Projection $[-1.0, 1.0]^2$ vers $\{0,0\}, \{\text{Largeur}, \text{Hauteur}\}$

Orthogonal vs Perspective



■ Projection Orthogonale

- Angles droits respectés
- 2D, CAO 3D

■ Projection Perspective

- Point de fuite au centre de l'image
- Objets lointains plus petits
- Monde virtuels, jeux, ...

Paramètres de perspective

■ Z-far

- « far clipping plane »: les objets derrière ce plan ne sont pas dessinés

■ Z-near

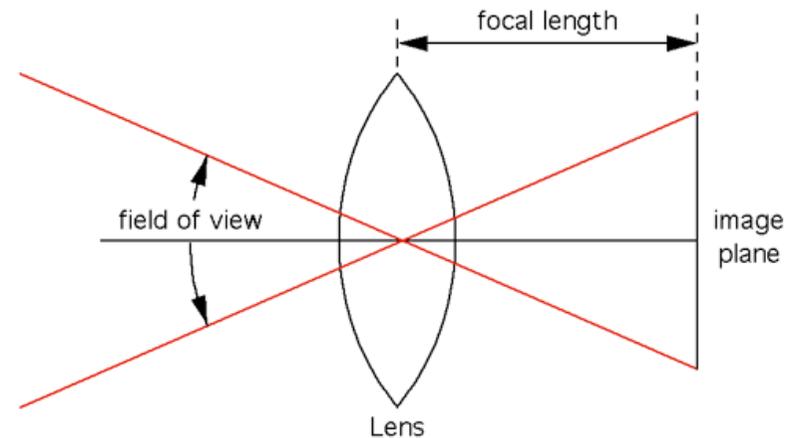
- « near clipping plane »: les objets devant ce plan ne sont pas dessinés

■ Aspect Ratio

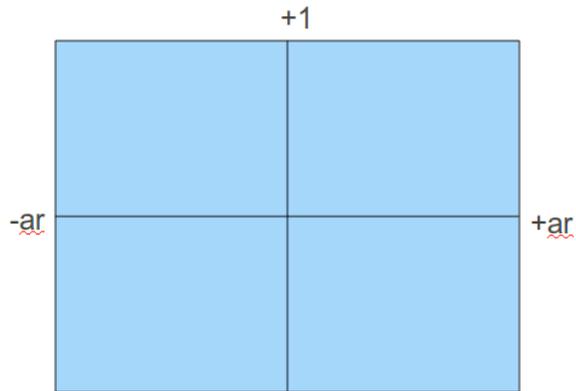
- Rapport largeur/hauteur

■ Notion d'ouverture

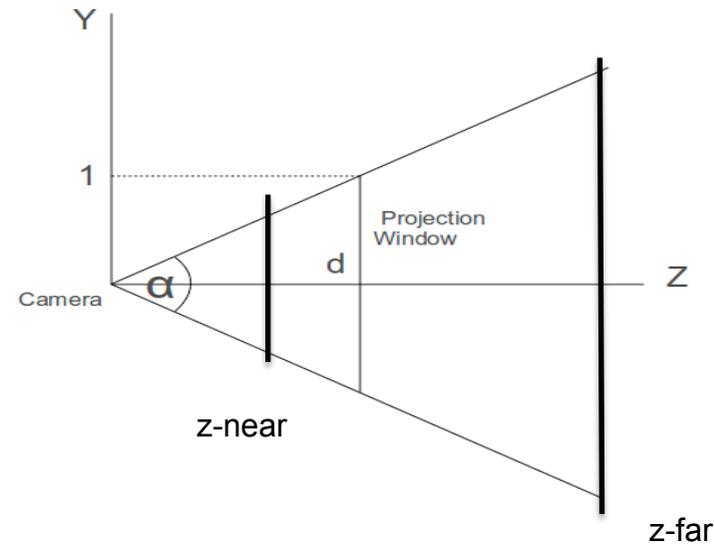
- En optique, notion de focale
- En 3D, « field of view »



La matrice de perspective



Fenêtre virtuelle⁻¹ de projection



Coordonnées normalisées sur $[-1,1]$

$$x_p = \frac{x}{ar \cdot z \cdot \tan\left(\frac{\alpha}{2}\right)}$$

Division par z pas pratique:

- z varie à chaque vertex
- ce n'est pas un produit matriciel !

$$y_p = \frac{y}{z \cdot \tan\left(\frac{\alpha}{2}\right)}$$

La matrice de perspective

■ Division par Z gérée par le matériel

- « perspective division »
- Pour un « z » normalisé entre « z-near » et « z-far »

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ ar \cdot \tan\left(\frac{\alpha}{2}\right) & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-NearZ - FarZ}{NearZ - FarZ} & \frac{2 \cdot FarZ \cdot NearZ}{NearZ - FarZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

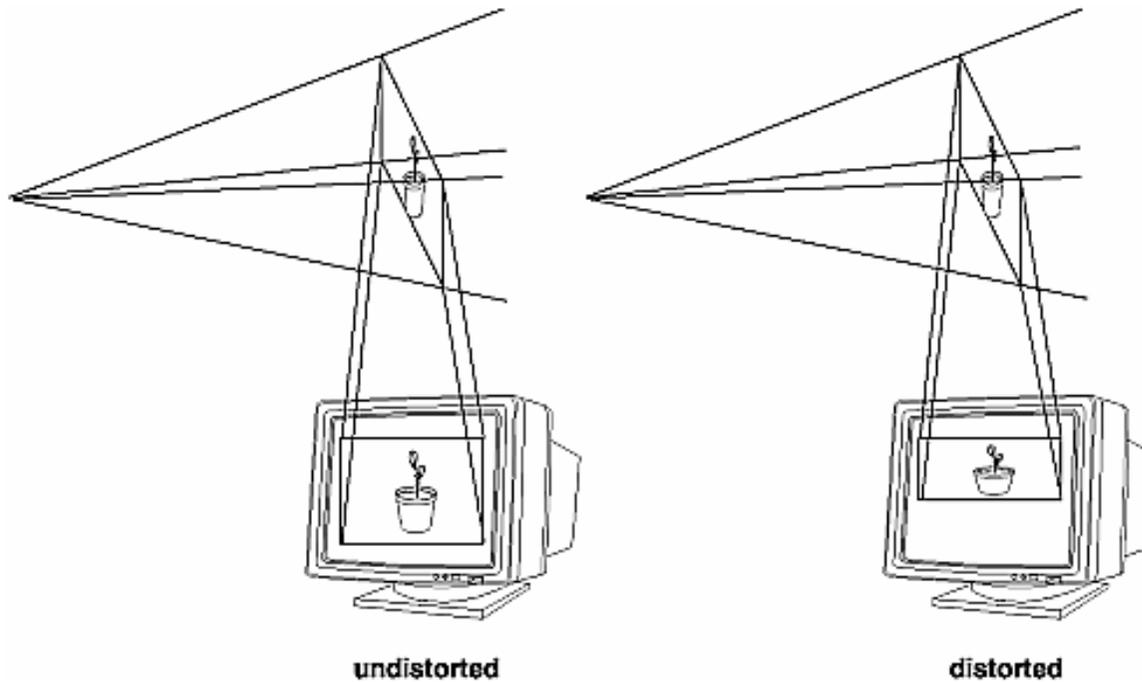
■ Dans la pratique

- gluPerspective: Utilisation de FoV, z-near, z-far
- glFrustum: Utilisation de haut/bas/gauche/droite (fenêtre de projection), z-near et z-far

Notion de viewport

■ Passer de $[-1.0,1.0]^2$ à l'écran:

- Viewport = rectangle $\{x,y,W,H\}$ d'affichage
 - en coordonnées pixels



Rendu des objets

■ Photo-Réalisme

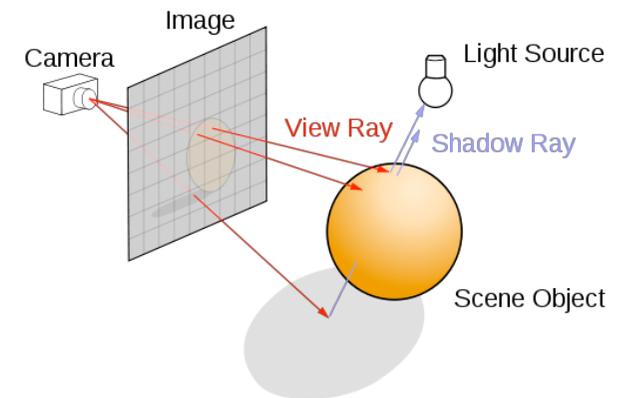
- Propagation de la lumière dans la scène virtuelle

■ Techniques

- Ray Tracing
 - Pour chaque pixel de l'écran, trouver toutes les contributions au rayon de lumière arrivant à la camera
- Ray Casting
 - Uniquement les contributions primaires

■ Inconvénients

- Très coûteux
- Pas adapté au temps réel
- Nécessite de connaître toute la scène au moment des calculs
 - Bande passante, stockage, etc ...



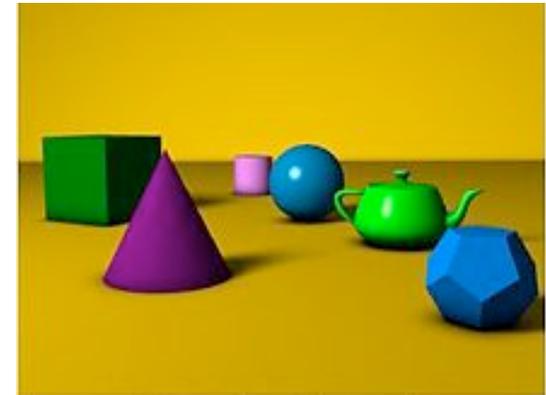
Rendu des objets

■ Via Carte Graphique (GPU):

- Dessin objet par objet
- Primitives après projection: Point, lignes, triangles ou quadrilatères
- Pour chaque point V dans la primitive correspondant à un pixel P ,
 - Calcul de la couleur
 - Calcul de la profondeur Z
- Si $Z(V) > Z(P)$, pas de dessin
 - Autre objet plus proche
- Si $Z(V) \leq Z(P)$, dessin

■ $Z(P)$ est le z-buffer ou tampon de profondeur

- Stocke les valeurs de Z des pixels à l'écran
 - Non linéaire $f(\text{near}, \text{far}, 1/z)$
- Précision finie: 8, 16, 24 bits
- Importance du choix de $z\text{-near}$ et $z\text{-far}$

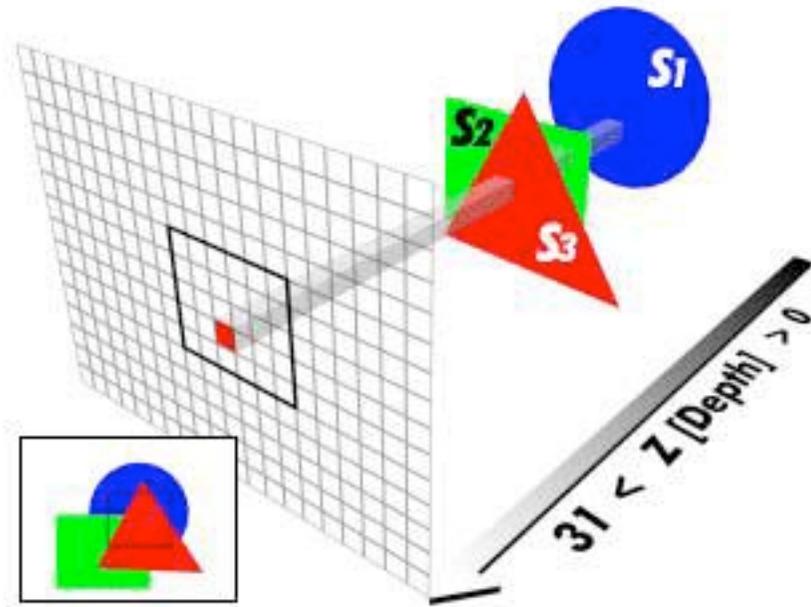


Une scène 3d simple



Le Tampon de profondeur

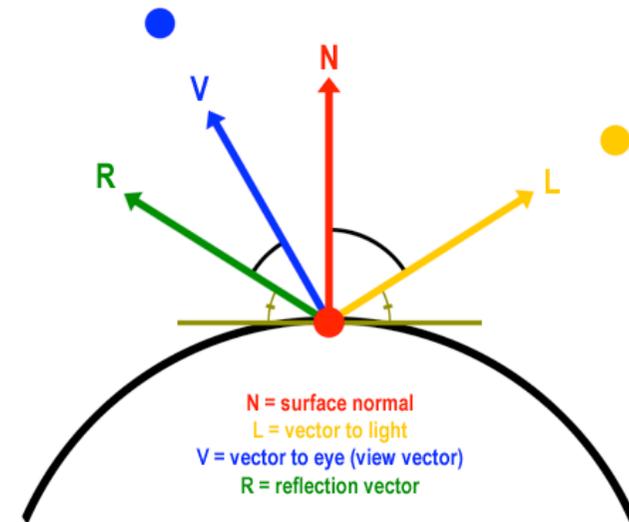
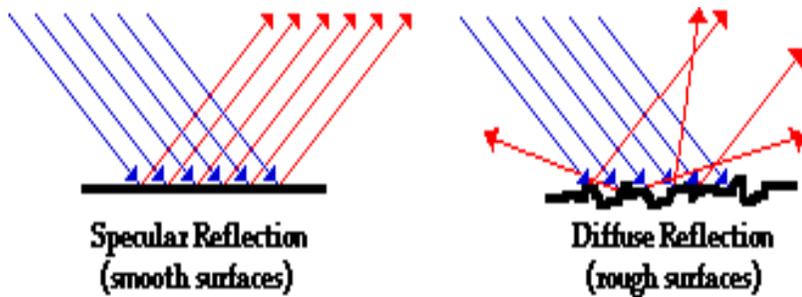
Principe du Z-buffer



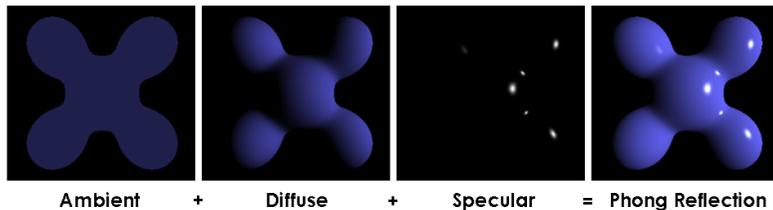
1	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
2	0	0	0	0	0	0
	0	0	0	0	0	0
	10	10	10	10	0	0
	10	10	10	10	0	0
	10	10	10	10	0	0
3	5	5	5	5	5	5
	5	5	5	5	5	5
	10	10	10	10	5	5
	10	10	10	10	5	5
	10	10	10	10	5	5
4	5	5	15	15	5	5
	5	5	15	15	15	5
	10	15	15	15	15	15
	10	15	15	15	15	15
	15	15	15	15	15	15

Modèles de lumières

■ Mélange entre « lumière diffuse » et « lumière spéculaire »



■ Modèle de Phong

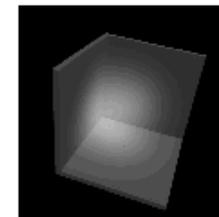
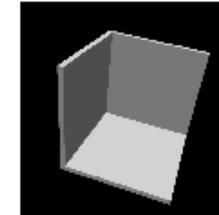
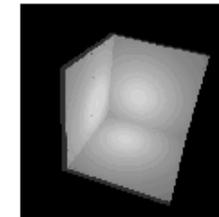
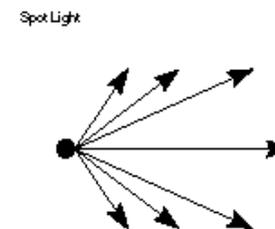
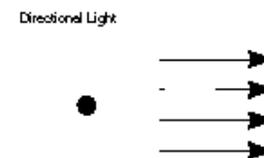
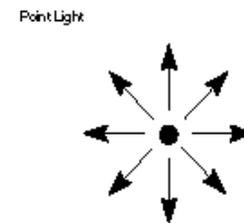
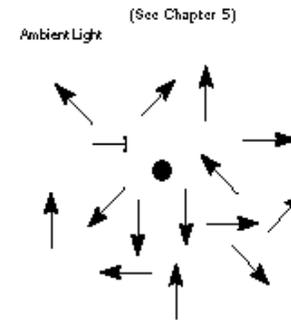


$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

- IMPORTANCE DES NORMALES

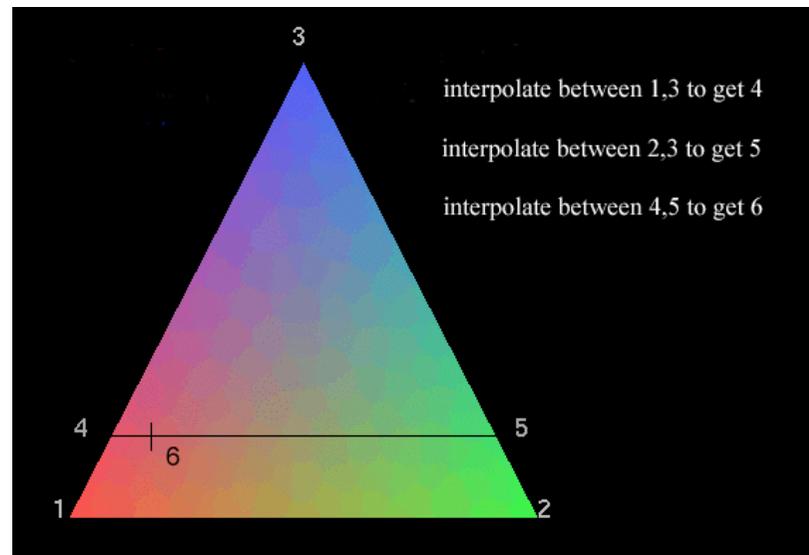
Modélisation des Lumières

- Ambiante: illumine toute la scène, sans « source » associée
- Point: illumine la scène localement dans toute les direction
- Directionnelle: illumine toute la scène dans une direction donnée
- Spot: illumine la scène localement selon un cône de lumière



Couleur des objets

- Définition d'une couleur pour chaque sommet de la primitive
- Pour les points de la face, interpolation bilinéaire



- Pas très réaliste pour beaucoup de matériaux

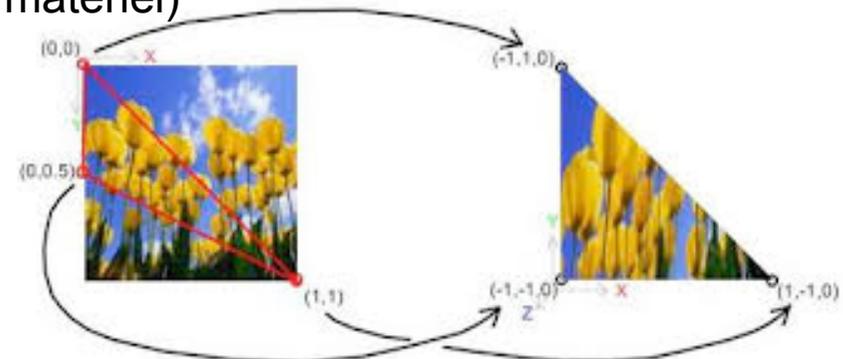
Image et Objets

■ Associer une image à un objet

- Création d'une « texture », contenant une image RGB, RGB+Alpha, ...

■ Associer une zone de l'image à une primitive

- Définition d'un espace de coordonnées pour les textures
 - U,V dans $[0.0, 1.0]^2$
- Définition de coordonnées de textures pour chaque point de la primitive
- Interpolation pour les autres points de la face
 - Prise en compte de la perspective (via matériel)



■ Déformations possibles via des matrices de texture

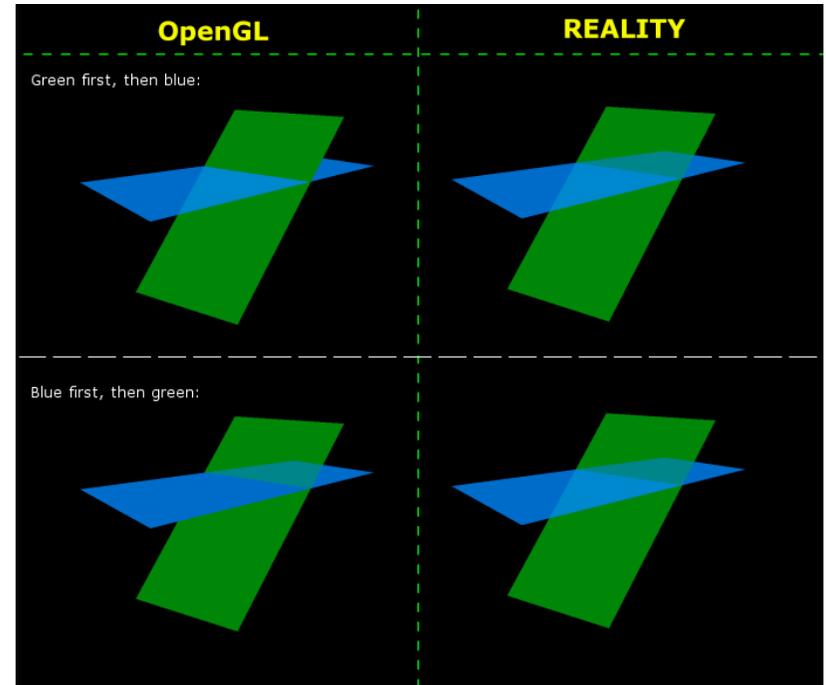
Objets et transparence

■ Problématique

- La carte graphique dessine les objets
 - Les uns après les autres
 - dans l'ordre demandé par l'utilisateur
 - Sans garder la couleur de chaque objet

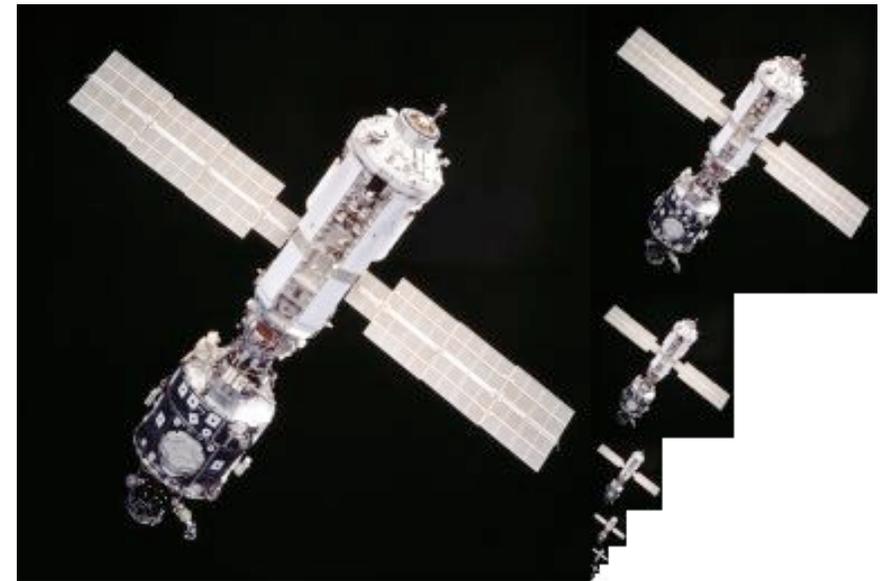
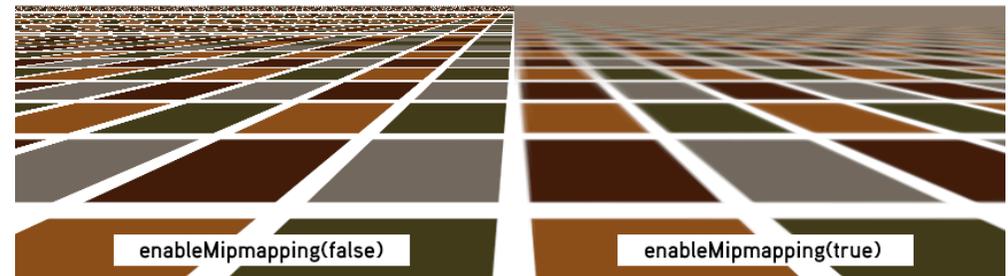
■ Solutions

- Trier les objets
- Si intersections:
 - Re-couper les objets
 - Lourd
 - Tri pas toujours possible
 - Dessiner en plusieurs passes
 - Pas simple ...



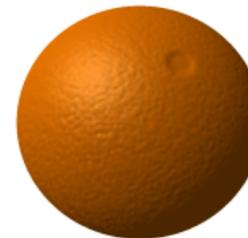
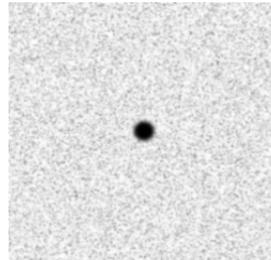
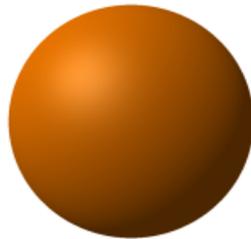
Textures et niveau de détail

- **Taille= $2^n \times 2^k$**
- **Ex: Texture 512x512**
 - Si la face fait 3x3 , 50x50 pixel après projection ?
 - Pixelisation/crénelage
- **Mip-Mapping**
 - Versions de la texture à différentes résolutions
 - Améliore le rendu des objets lointains



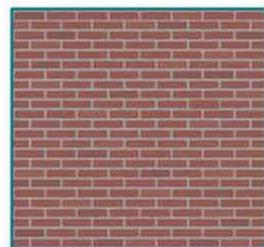
Textures avancées

- **Possibilité d'avoir plusieurs textures / objet**
 - Plusieurs coordonnées de textures par vertex
- **Bump Mapping & Displacement Mapping**
 - Altération des normales
 - Altération des vertex

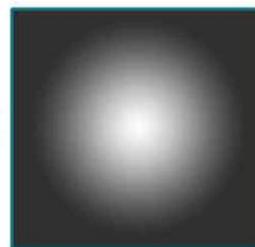


- **Light Maps**

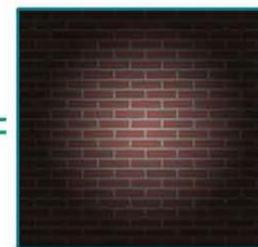
- Lumière pré-calculée et stockée en texture



X



=



DIFFUSE

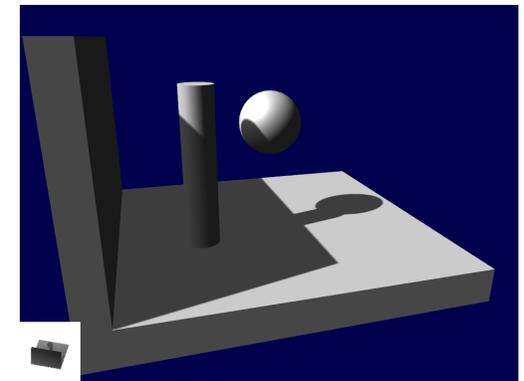
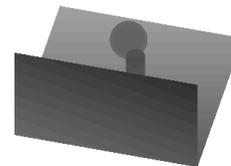
LIGHTMAP

DIFFUSE x LIGHTMAP

Synthèse 3D: Réalisme Avancé

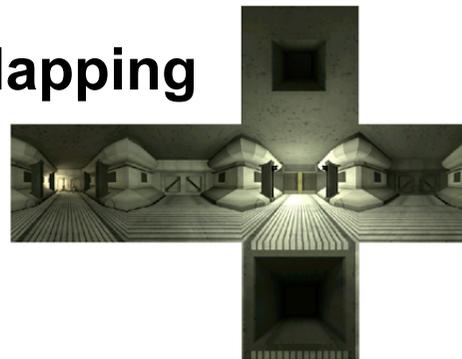
■ Ombres

- Shadow Mapping
 - Scène vue depuis la lumière
 - « distance » pixel \leftrightarrow source
 - Deuxième passe
 - Distance \rightarrow ombre
- Shadow Volume
 - Très complexe mais plus précise
- Et bien d'autre encore
 - Selon les besoins...



■ Reflexion / Environment Mapping

- Monde vu depuis l'objet
- Cube map





Déploiements 3D: Accélération Matérielle

■ GPU: Graphics Processing Unit

- Implémentation matérielle des opérations couteuses
 - Calcul vectoriel/matriciel
 - Gestion des textures
 - Interpolation des couleurs
 - Dessin des pixels, gestion de la profondeur (Z-buffer)
- Version logicielle (via pilote de carte) pour le reste
 - Pas toujours de garantie sur ce qui est accéléré ou non
- Traitement en parallèle des primitives

■ Qui ?

- Direct3D (Microsoft)
- OpenGL (Khronos)
- OpenGL-ES (Khronos)

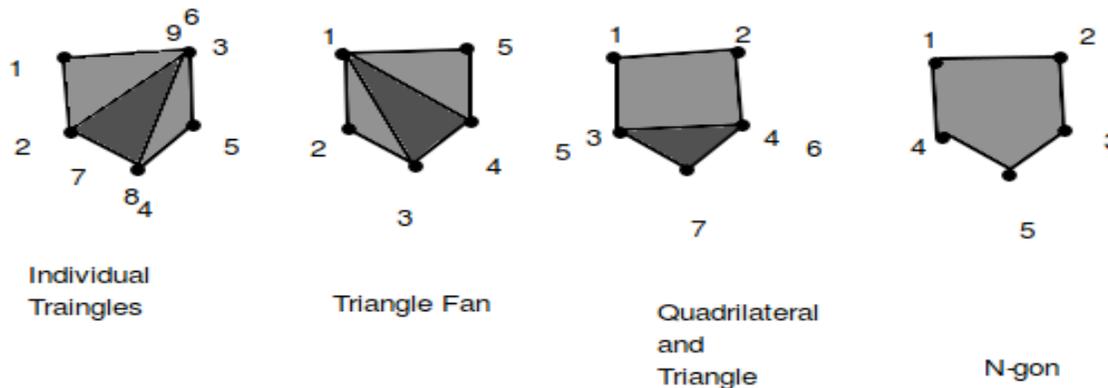
Principes de l'accélération matérielle 3D

■ Suite d'instructions programmant le GPU

- Sortie directe sur la mémoire vidéo
 - Extensions pour le rendu hors écran

■ Primitives

- Simples: triangle, carré (quad), polygone convexe simple
- Complexe: « soupe de triangle » + indexes





Principes de l'accélération matérielle 3D

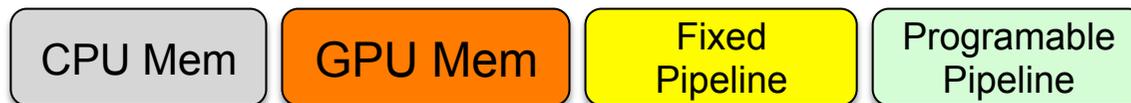
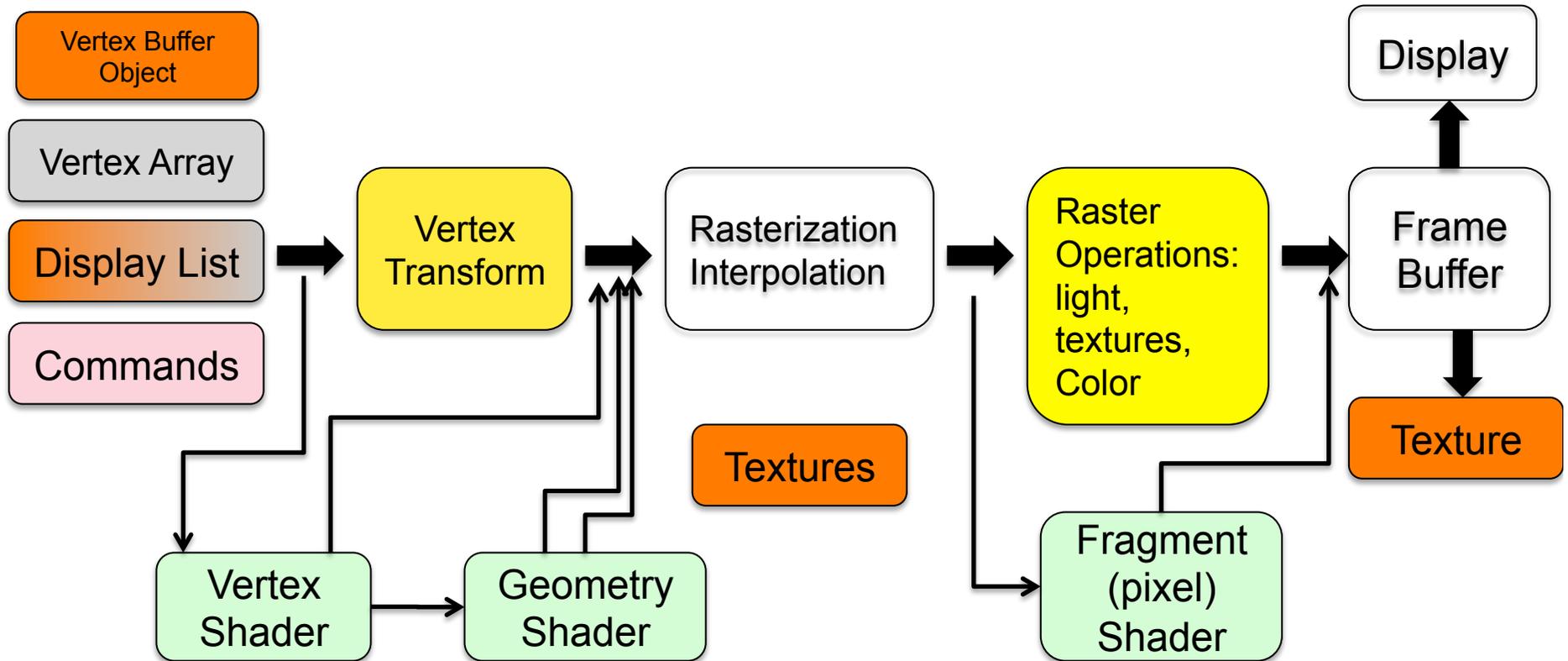
■ Version « fixed pipeline » (old school)

- Variables d'état
 - Lumière, couleurs, paramètres de filtrages
 - Options de ciseaux
 - Transparence, brouillard, ...
- Piles de variables
 - Matrices de transformations (texture, modèles et perspectives) et états actifs
 - Push/pop

■ Version Shader

- Notion de programme (langage: GLSL)
- Le programme décrit toutes les manipulations à faire:
 - Sur les vertex
 - Sur les pixels

GPU Pipeline



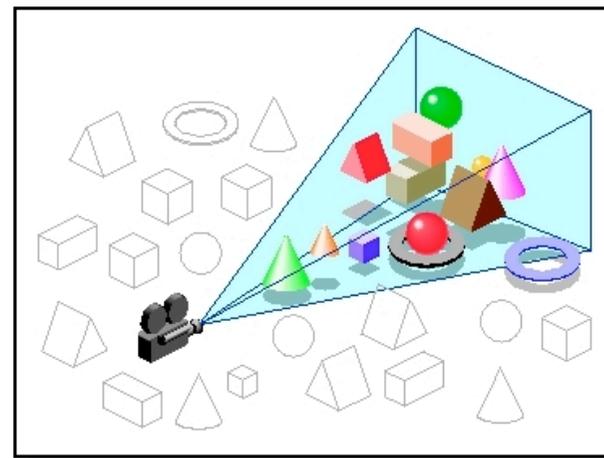
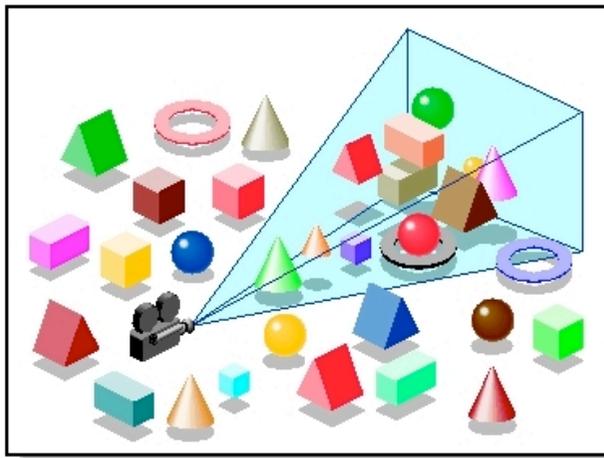


Gestion des données

- **Données lues par le programme (mémoire système)**
 - Vertex info: position, normale, coordonnées de textures
 - Textures
- **Mémoire GPU**
 - != mémoire système
 - Limite de taille et de bande passante CPU->GPU
- **Transmettre uniquement les objets utiles**
 - « liste graphique » pour un trame donnée
 - Libérer les ressources matérielles si non utilisées
 - Ex: changement de niveau, changement d'accessoire d'un personnage, etc ...

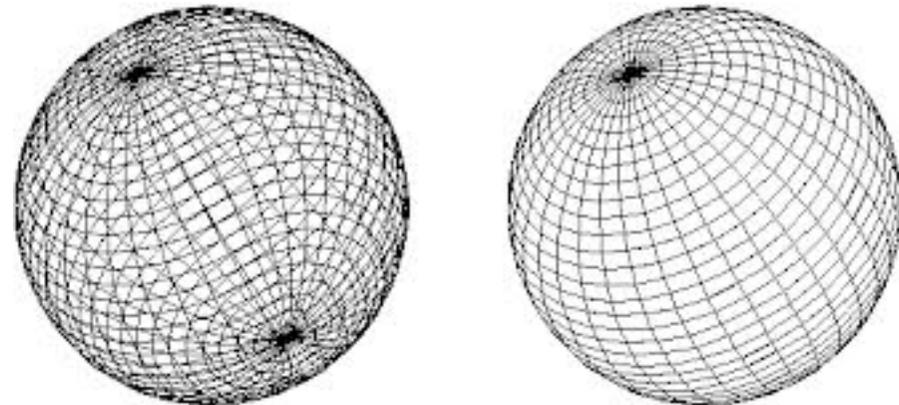
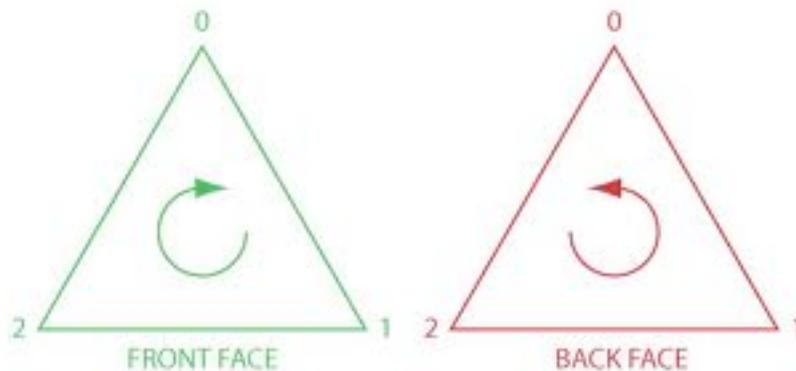
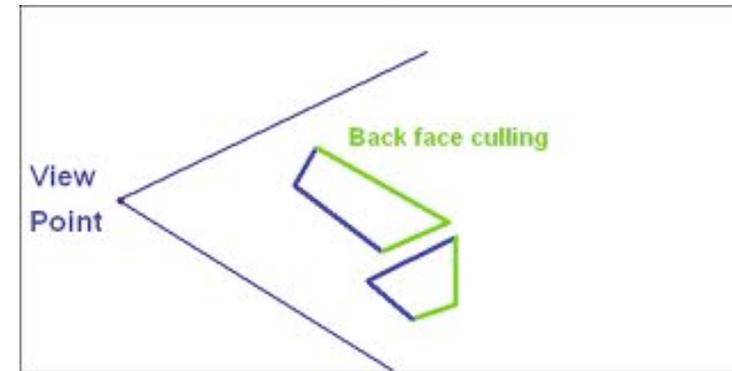
■ Elimination d'objets

- « Culling »
- Si objet hors de la zone de dessin, pas de calcul
 - Notion de Frustum (volume de visualisation)
- Pas effectué par le matériel



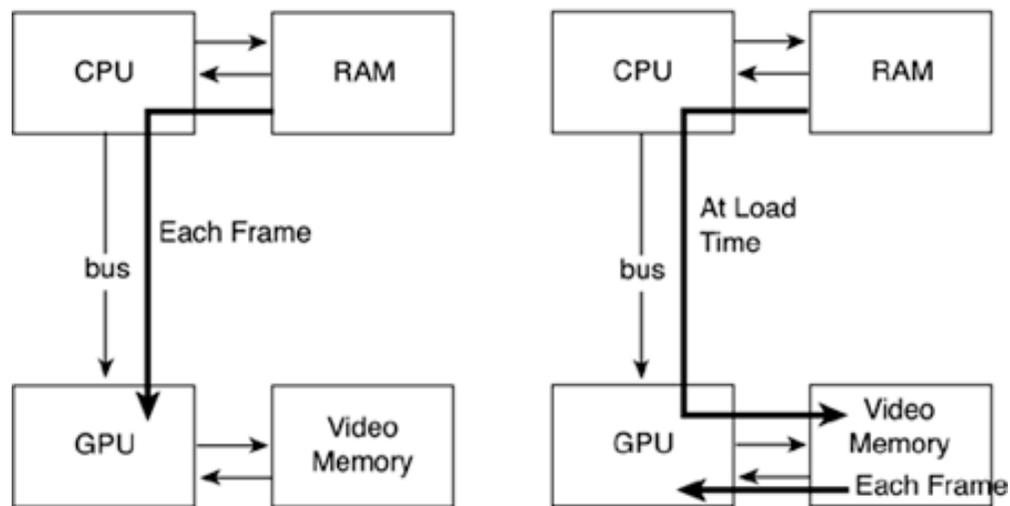
■ Éliminer les faces non visibles

- Faces arrières des objets
 - Masquées par une autre face
 - Objets fermés et opaques
- Peut être effectué par le matériel
 - « back-face culling »
- Test Vue.Normal
 - !! FACES ORIENTEES



Gestion des données: animations

- **Plusieurs trames de rendu pour un même objet**
 - Envoyer N fois les mêmes objets au GPU n'est pas efficace !
- **DisplayList**
 - Enregistre une série de commandes envoyées au GPU
 - « Relecture » de cette série
 - Plus efficace si la série est stockée en mémoire GPU
 - Non paramétrable
 - Pas possible de ré invoquer en ne changeant qu'un paramètre (ex couleur du vertex K)
- **VertexBufferObject**
 - Attributs de vertex stockés sur le GPU
 - Ré-utilisation du VBO ne nécessite pas de retransmettre les données
 - Utilisé pour position, normale, coordonnées de textures, couleur





Programme simple typique

```
main()
{
//1- mise en place du programme

//2- mise en place OpenGL

while (!fin) {
    //3- interactions du programmes, mise à jour des variables (dont le temps
    pour les animations), etc

    //4- identification des objets actifs pour la trame courante (liste
    graphique), chargement de nouveaux objets / textures

    //5- rendu openGL - PAS DE MODIFICATIONS DES VARIABLES D'ETAT NON-OPENGL
    SI PLUSIEURS PASSES NECESSAIRES (ex: stereo ou multivue)!

    //6- afficher la trame
    glSwapBuffer();
}

//nettoyage des ressources
}
```



Les Shaders

■ Vertex Shader

- Modifie les attributs d'un vertex
 - Position, couleurs, coordonnées de textures, etc
- Application des matrices (transformations + projection)
 - Typiquement, les normales sont ajustées pour l'éclairage

■ Fragment Shader

- Modifie les attributs d'un pixel
 - Couleur/texture, transparence
- Pixel écrit en mémoire vidéo si test profondeur OK

■ Autres

- Geometry Shader: permet de cloner des vertex à la volée (rendu multiple d'objet)
- Tessellation Evaluation Shader

■ Programme

- 1! vertex shader + 1! fragment shader
- En option, 1 geometry shader, 1 TES



Le Langage GLSL

■ But

- Interopérabilité de la programmation entre GPUs
- Simple et proche du C
- Permettre l'exécution parallèle des instructions de rendus
 - Ex: 32 triangles en //

■ Problèmes

- Différentes versions du langage
 - Entre génération de cartes
 - Entre desktop (OpenGL) et Mobile (OpenGL ES)

■ Principes

- Lors de la compilation, définition
 - Variables locales et code
 - Données d'entrée (du précédent shader)
 - Données de sortie (pour shader suivant ou résultat final)
- Avant de dessiner
 - Paramètres des shaders liés au programme ou « uniform »
 - Constant pour tout l'objet
 - Couleur, matrices, vecteurs, etc ...
 - Attributs de l'objet (« attributes »)
 - Données changeantes par vertex

Exemples triviaux de shader (GLSL<3)

■ Vertex Shader

```
#version 110

void main()
{
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

■ Fragment Shader

```
#version 110

void main()
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Exemples triviaux de shader (GLSL>3)

Vertex Shader

```
#version 330
```

```
in vec4 vert;
```

```
uniform mat4 projection;  
uniform mat4 view;  
uniform mat4 model;
```

```
void main()
```

```
{  
    gl_Position = projection * view * model * vert;  
}
```

via glBindAttributeLocation



via glUniformZZZ

Fragment Shader

```
#version 330
```

```
out vec4 fragColor;
```

```
void main()
```

```
{  
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```



Framebuffer

■ Par défaut

- Le GPU utilise
 - Mémoire de couleur et Mémoire de profondeur (Z)
 - Option: mémoire de stencil (masques de pixels)
- N'affiche le résultat qu'à la demande (swap buffer)

■ Pour Dessiner dans une texture

- Old school: copier l'écran dans la texture (`glCopyTexImage2d`)
 - Nécessite une copie -> couteux en temps
- OpenGL 3 / GLES 2: `FramebufferObject`
 - mémoires ad-hoc: texture X pour la couleur, Y pour la profondeur
 - Pas de copie nécessaire, juste un changement de FBO



Règles de bases

■ Gestion des objets

- Les données sont définies dans des fichiers à part, **PAS DANS LE PROGRAMME**
 - Possibilité de mettre à jour un modèle sans changer le programme
- Exemple de format: .obj, VRML, Collada, MD5Mesh

■ Shaders

- Factoriser le code: **un seul compilateur de shaders !**
- Code des shaders dans des fichiers séparés
- Obligatoires pour Android, recommandé pour le reste

■ Gestion des ressources

- Prévoir de libérer les ressources de la carte graphique si un objet n'est plus utilisé pour une longue période de temps



Pour démarrer

- **Sous windows/linux:** <http://lwjgl.org/>
- **Sous Android:**
<http://developer.android.com/training/graphics/opengl/index.html>
- **Petite liste d'exercices:**
 - <http://perso.telecom-paristech.fr/~lefeuvre/PACT/OpenGL/ModuleOpenGL.pdf>



Au Travail !

PACT – Mini-Cours

Animation de personnages par sprite

■ Texture

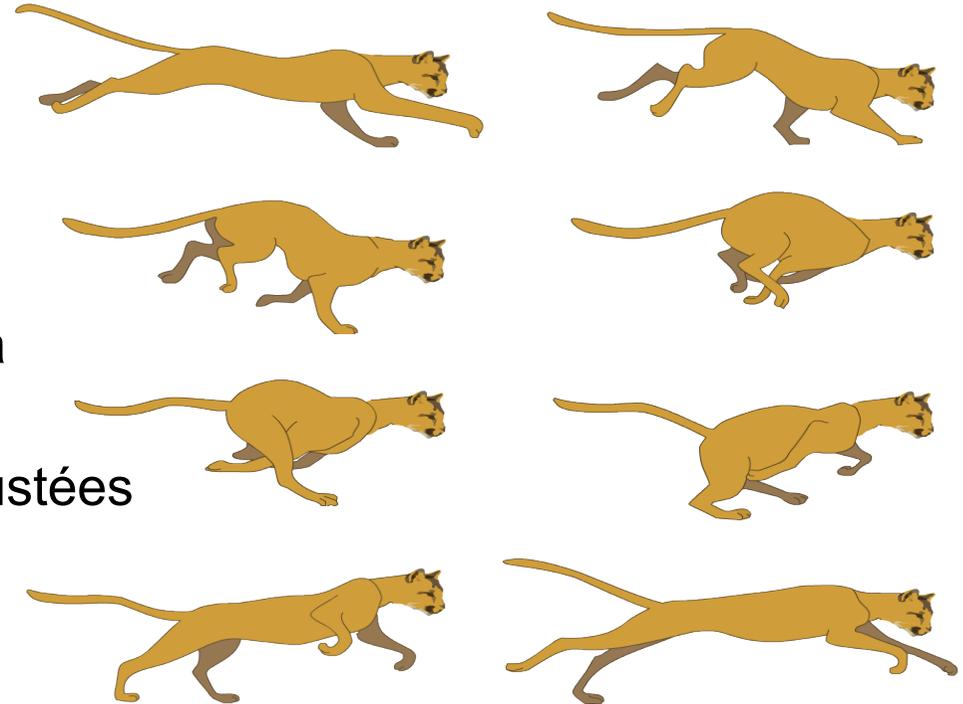
- Toutes les positions du personnage

■ Géométrie

- Carré toujours en face de la caméra (« billboard »)
- Coordonnées de texture ajustées en fonction du temps

■ Simple et rapide

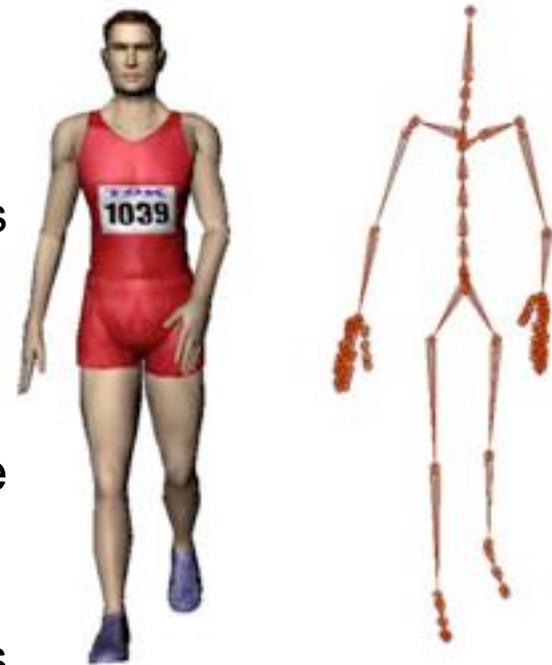
- Moins esthétique ...



Animation par déformation

■ Définir

- une armature
 - Liste de segments de droite (« bone »)
 - Définie pour une pose A donnée
- une enveloppe
 - Définie dans la pose A
 - Chaque vertex est associé à un ou plusieurs « bones » avec un poids P_i / bone
 - La somme des poids = 1



■ Animer

- Déformation de l'armatures via matrices de A vers la position souhaitée
- Transformation du vertex = somme des transformations/bone du vertex, pondérées par le poids