

Cours 3: Sous le capot

Xavier Décoret – INF 683 - École Polytechnique

Overview

- ▶ **Carte graphique**
 - ▶ Pipeline
 - ▶ Framebuffer et tests
 - ▶ Considérations techniques
 - ▶ Interpolation
 - ▶ Rendu offscreen
- ▶ **Shaders**
 - ▶ Histoire & motivations
 - ▶ Principes & mise en oeuvre

Pipeline

- ▶ **Le GPU est un *stream processor***

- ▶ Prend en entrée un flux de sommets et de faces
- ▶ Produit en sortie un flux de fragments
- ▶ Combine ces fragments en une image

Fait ça en
parallèle

- ▶ **Les étapes sont**

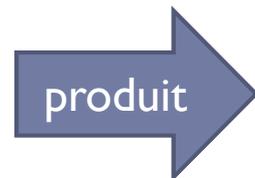
- ▶ Transform & lighting
- ▶ Assembly stage
- ▶ Scan conversion
- ▶ Blending & tests



Attention au problème de concurrence

Transform & lighting

- ▶ Les coordonnées homogènes du sommet sont multipliées par la *modelview matrix* puis la *projection matrix*
- ▶ Les normales sont transformées
 - ▶ Voir le transparent suivant...
- ▶ Pour chaque source (1 à 8)
 - ▶ Le vecteur sommet/source est calculé
 - ▶ On calcule une couleur avec le modèle de Phong



- Normalized Device Coordinates
- Couleur
- Coordonnées textures

Transformation des normales

- ▶ Un plan est caractérisé par

$$\underbrace{[a \ b \ c \ d]}_n \cdot [x \ y \ z \ w]^T = 0$$

- ▶ Quelle est la caractérisation de la transformée d'un plan par une transformation affine de matrice M ?

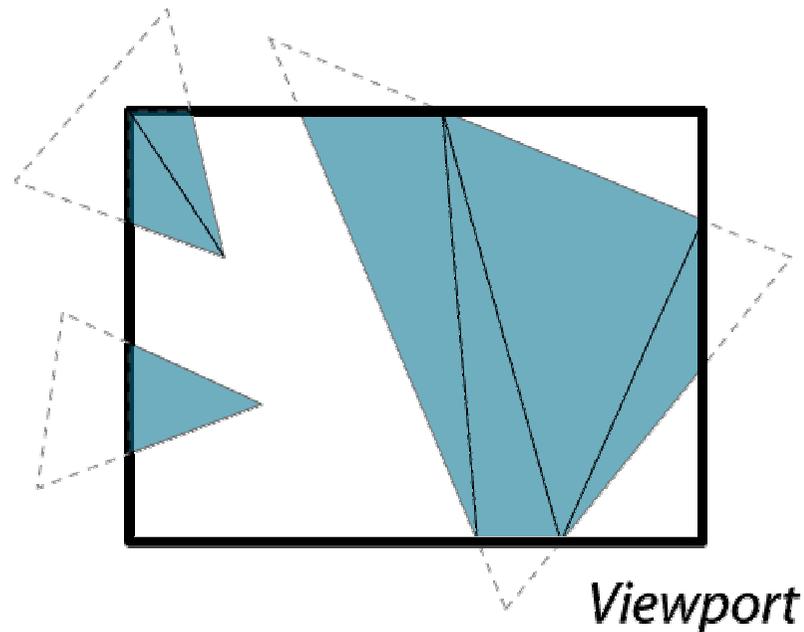
On cherche n' tel que $n' \cdot (M[x \ y \ z \ w]^T) = 0 \Leftrightarrow n' \cdot [x \ y \ z \ w]^T$

Une solution est $n' = (M^T)^{-1}n$

Les normales doivent être transformées par la **transposée inverse**!

Assembly stage (1 / 2)

- ▶ **Clippe les primitives contre le *viewport***
 - ▶ Évite de produire des fragments en dehors de l'image
 - ▶ Décompose en un ou plusieurs triangles
- ▶ **On peut spécifier des plans de *clipping* (`glClipPlane`)**

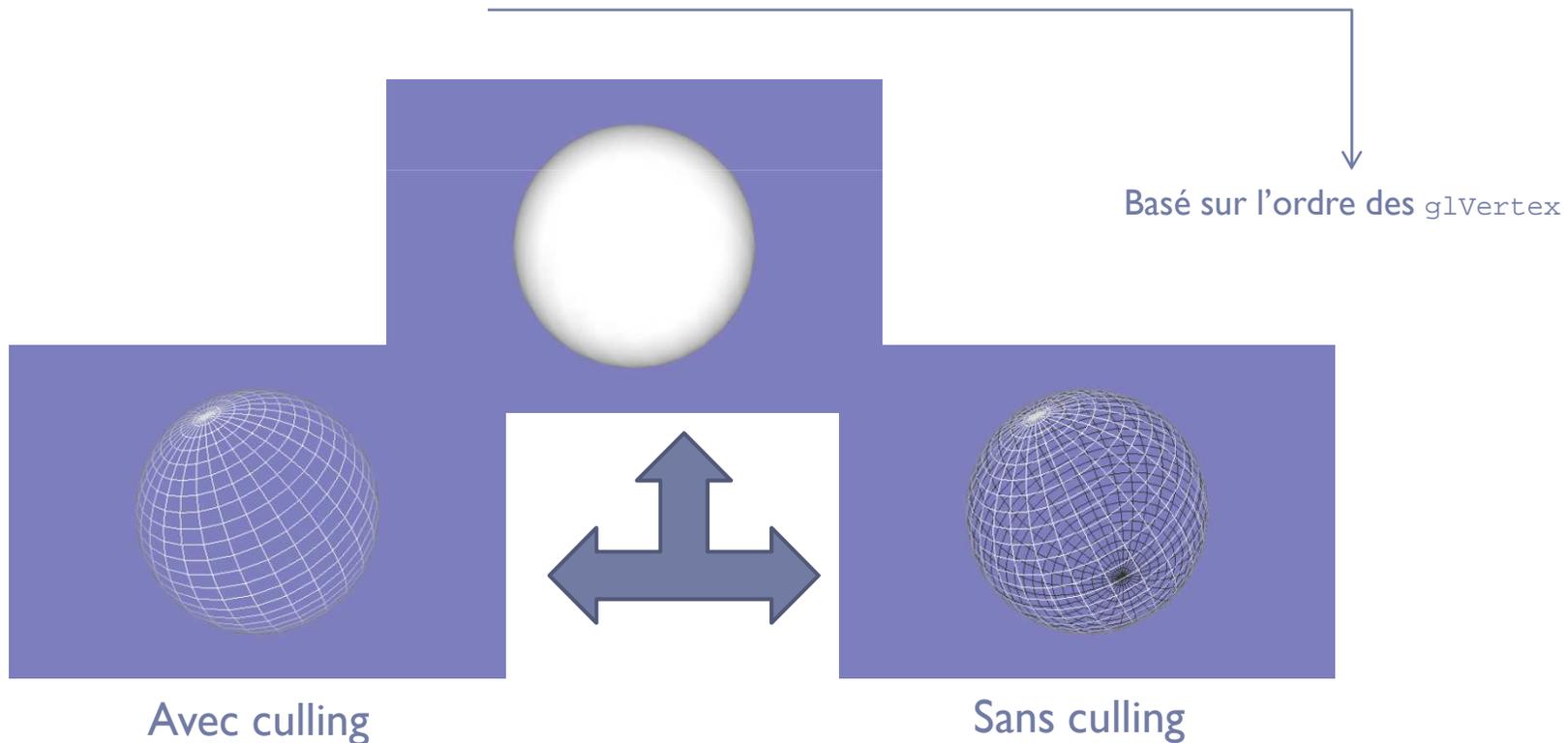


Assembly stage (2/2)

▶ Backface culling

→ On peut définir un intérieur et extérieur

- ▶ Si un objet est “étanche”, alors il n’y a pas besoin de dessiner les faces qui tournent le dos à l’observateur



Scan conversion *aka* Raster (1 / 2)

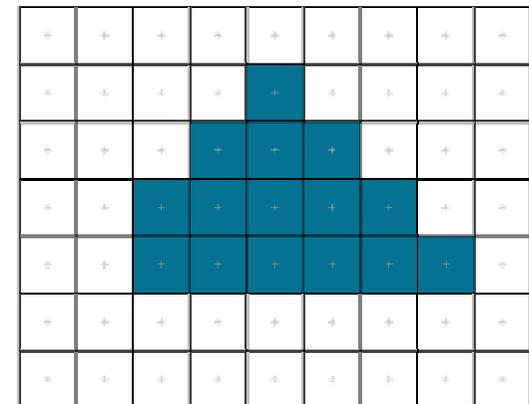
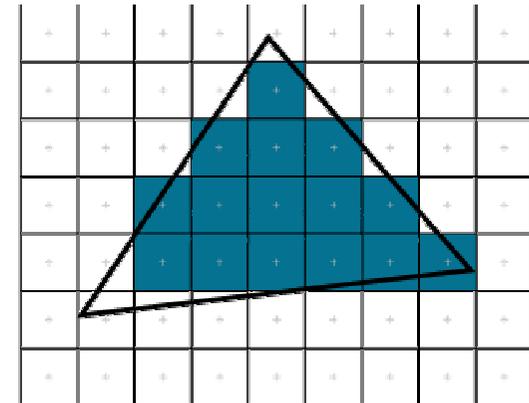
- ▶ On considère les pixels dont *le centre* est dans le triangle
- ▶ Cela crée de l'aliasing
- ▶ Il faut filtrer
 - ▶ On peut filtrer soit-même
 - ▶ On double la résolution du viewport
 - ▶ On filtre l'image obtenue
 - ▶ Les cartes gèrent du *subsampling*

```
glSampleCoverage()
```

```
glHint()
```

Attention à l'activer sur la carte e.g.

- par le panneau de configuration
- par une variable d'environnement



Scan conversion *aka* Raster (2/2)

- ▶ On considère les pixels dont le centre est dans le triangle
- ▶ Cela crée de l'aliasing
- ▶ Il faut filtrer



Framebuffer (1 / 2)

- ▶ Au final on obtient une image couleur
- ▶ Mais en réalité, les pixels stockent plus d'information
 - ▶ Couleur RVB (chaque canal sur 8 bits)
 - ▶ Profondeur z (sur 24 bits)
 - ▶ Opacité A (sur 8 bits, packés avec RVB)
 - ▶ Valeur de *Stencil** (sur 8 bits, packés avec z)
- ▶ L'ensemble de ces pixels/valeurs forment le *framebuffer*

Comment les fragments produits par le *scan convert* sont-ils combinés avec les pixels correspondants du *framebuffer* ?

**Stencil* = pochoir

Framebuffer (2/2)

- ▶ Les valeurs RVBA et z d'un fragment sont obtenues par interpolation linéaire des valeurs calculées aux sommets
- ▶ On effectue un certains nombres de tests
 - ▶ Depth test
 - ▶ Alpha test
 - ▶ Stencil test

- ▶ Selon le résultats de ces tests:

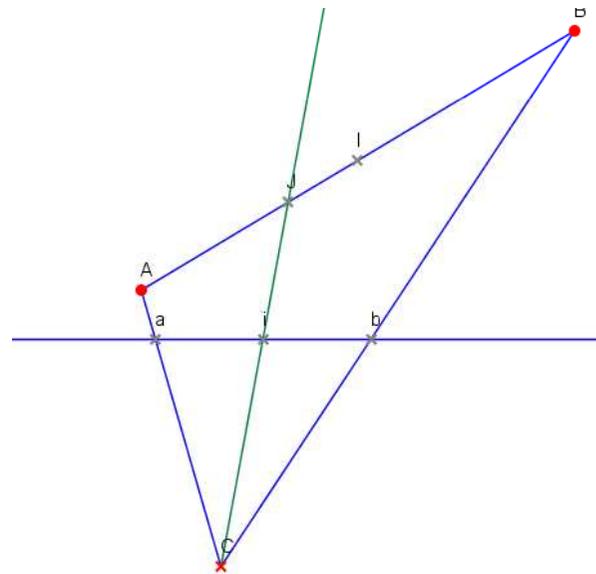
- ▶ La couleur du fragment est combinée avec celle du pixel
- ▶ Le z de ce pixel est remplacé par celui du fragment
- ▶ La valeur de stencil est mise à jour

3 comportements différents

Interpolation et perspective (1 / 4)

- ▶ Il faut faire attention en interpolant en espace image

Le milieu des projections des extrémités d'un segment ne coïncide pas avec la projection du milieu du segment



Interpolation et perspective (2/4)

Projeter le point c'est trouver l ...

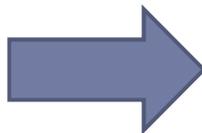
Par Thalès, on obtient $l = \frac{y}{w} \times \frac{-d}{-d + \frac{z}{w}}$

On vérifie que si on pose

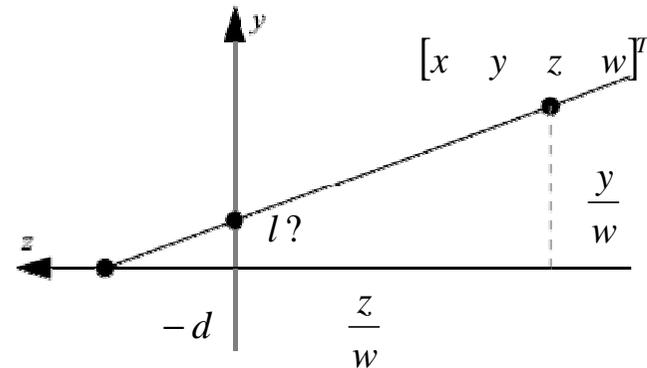
$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{-d} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

On a

$$\begin{aligned} \frac{Y}{W} &= \frac{y}{\frac{z}{-d} + w} \\ &= \frac{y}{w} \times \frac{d}{d - \frac{z}{w}} \\ &= l \end{aligned}$$



La matrice 4x4 représente la projection en coordonnées homogènes

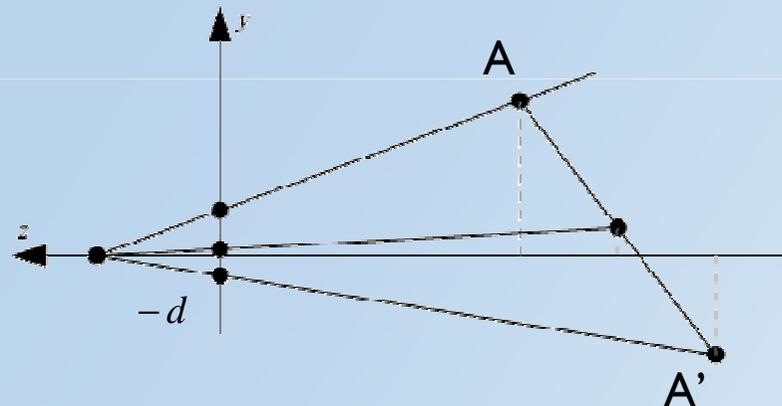


Interpolation perspective (3/4)

Si $a = [x \ y \ z \ w]$
 $a' = [x' \ y' \ z' \ w']$ sont des coordonnées homogènes de A et A'

alors le projeté du barycentre de $(\alpha, A), (\alpha', A')$ admet pour coordonnées homogènes

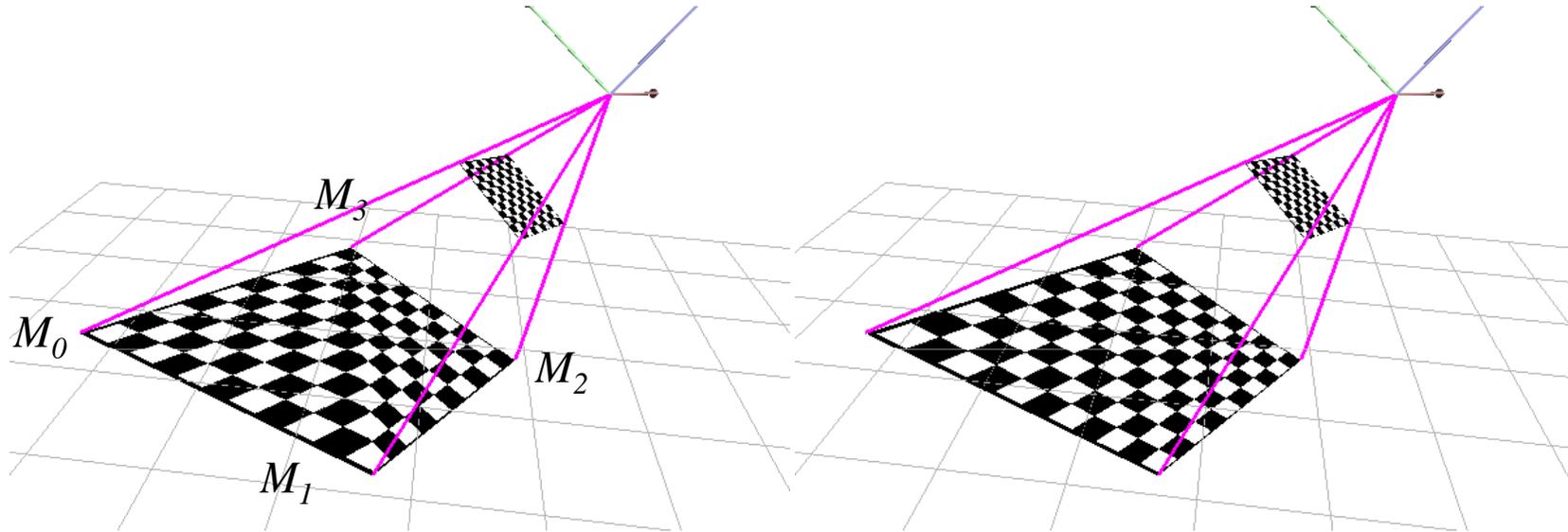
$$b = \alpha Ma + \alpha' Ma' \quad \text{avec} \quad M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{-d} & 1 \end{pmatrix}$$



On peut interpoler linéairement les coordonnées homogènes!

Interpolation perspective (4 / 4)

► Exemple du *Projective Texture Mapping*



On prend des coord. textures “unitaires”

$M_0 : (0,0)$ $M_2 : (1,1)$

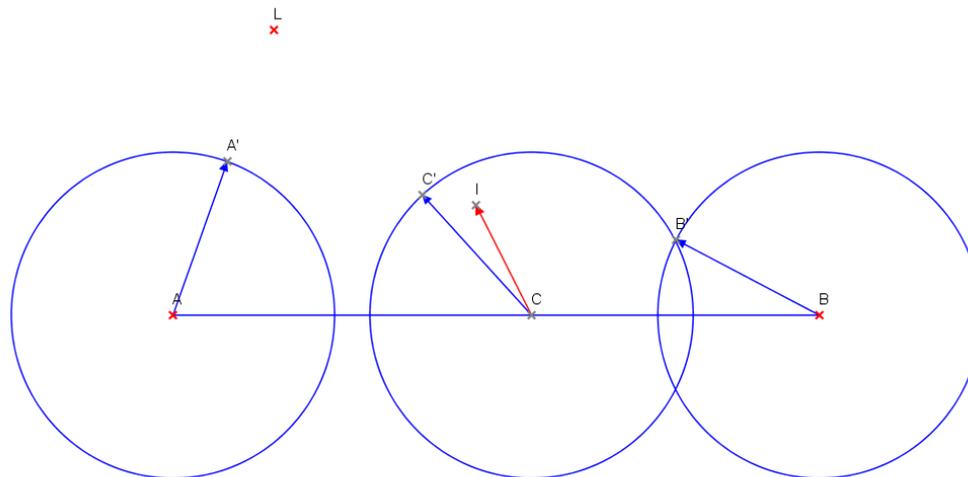
$M_1 : (1,0)$ $M_3 : (0,1)$

On calcule les coord. textures en utilisant une transformation 4x4 qui projete les M_i dans l'espace texture et peut être interpolé correctement.

Voir le code source de l'exemple sur le site du cours

Interpolation & shading

- ▶ Il faut renormaliser les normales
 - ▶ Une combinaison linéaire de vecteur unitaires n'est pas unitaire
 - ▶ On peut utiliser `glEnable(GL_NORMALIZE)`
- ▶ L'interpolation du *light vector* est incorrecte
 - ▶ Mais c'est bon si la lampe est "loin"



Framebuffer

- ▶ Les valeurs RVBA et z d'un fragment sont obtenues par interpolation linéaire des valeurs calculées aux sommets
- ▶ **On effectue un certains nombres de tests**
 - ▶ Depth test
 - ▶ Alpha test
 - ▶ Stencil test
- ▶ Selon le résultats de ces tests:
 - ▶ La couleur du fragment est combinée avec celle du pixel
 - ▶ Le z de ce pixel est remplacé par celui du fragment
 - ▶ La valeur de stencil est mise à jour

Depth-test (1 / 4)

Sortie du scan convert

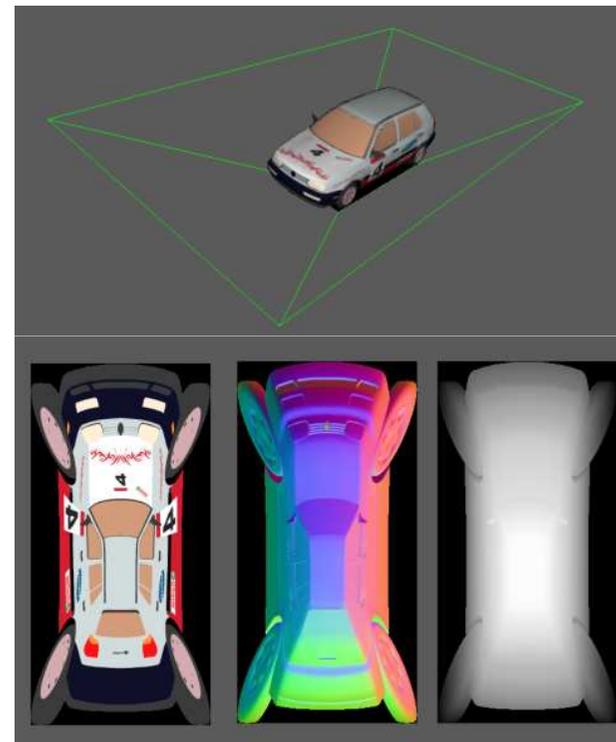
Stocké dans le z-buffer

- ▶ **Le z du fragment est comparé avec le z du pixel**
`glDepthFunc(GL_LEQUAL)`
// ou GL_GREATER, GL_LESS, GL_ALWAYS, GL_NONE, GL_EQUAL
- ▶ **En cas d'échec, le fragment est discardé**
 - ▶ Autres tests pas effectués
 - ▶ Mais mise à jour possible du framebuffer (cf. stencil test)
- ▶ **C'est inventé pour l'élimination des faces cachées**
- ▶ **Mais il y a pleins d'usages détournés...**

T.Theoharis, The Magic of the Z-Buffer: A Survey, 2001

Depth-test (2/4)

▶ Cubist images

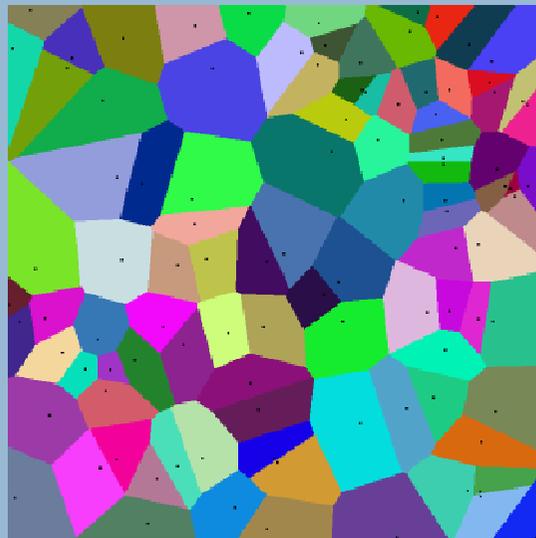


Hanson A. et al., Image-Based Rendering with Occlusions via Cubist Images, 1998

Baboud L. and Décoret X., Rendering Geometry with Relief Textures, 2006

Depth-test (2/4)

► Diagramme de Voronoï



Le diagramme de Voronoï associe à des points p_i des régions $V(p_i)$ telles que tous les points de $V(p_i)$ sont plus proches de p_i que de n'importe quel p_j

2D site	Shape of Distance Function	Figure
Point	Right circular cone	3a
Line segment	"Tent"	3b
Polygon	Cones and tents	5

Table 1: Shape of Distance Functions for 2D Sites

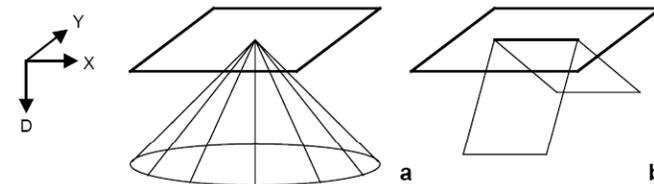
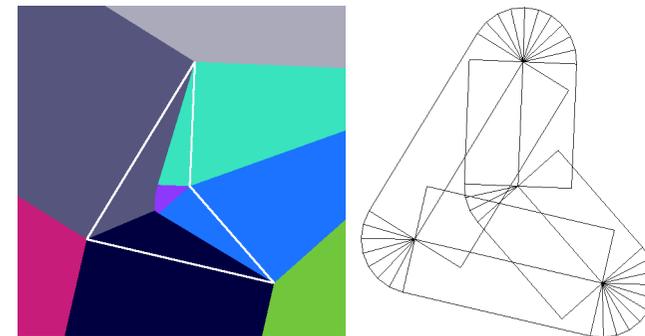
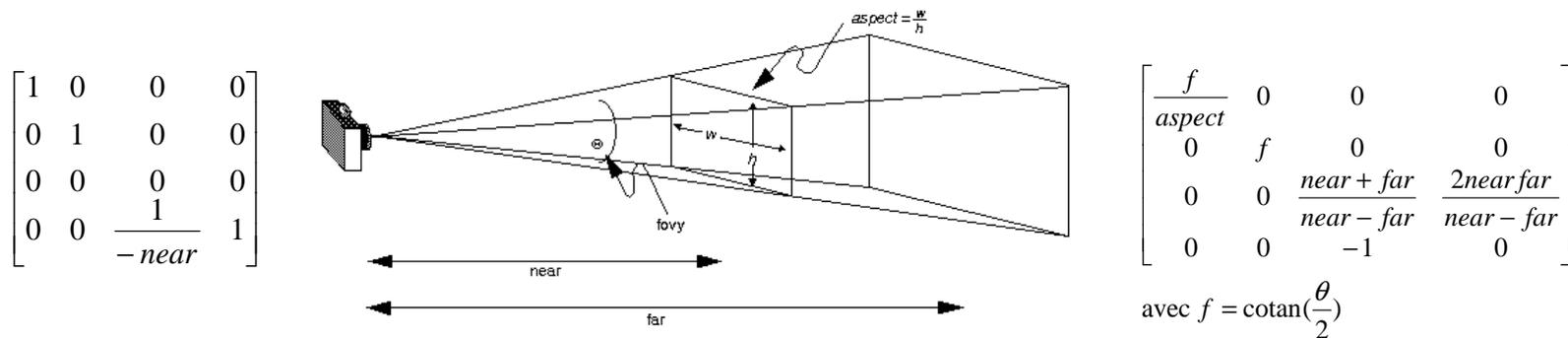


Figure 3: The distance meshes used for a point (left) and a line segment (right). The XY-plane containing the site is shown above each mesh.



Depth-test (3 / 4)

- ▶ Le z-buffer a une précision limité (24 bits)
- ▶ On ne peut pas représenter tous les z
 - ▶ Il faut limiter l'intervalle de profondeur!
 - ▶ C'est pour cela que la caméra *pinhole* a des *near/far planes*!
 - ▶ Les points en deçà/au delà sont clippés



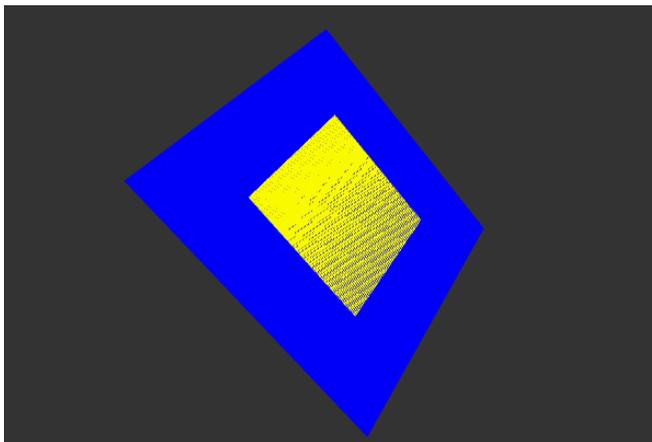
Matrice que l'on a trouvé tout à l'heure pour effectuer la division par z (perspective)



Matrice corrigé pour mapper le view frustum sur le cube unité (NDC)

Depth-test (4 / 4)

- ▶ **Le z-buffer n'a pas une dynamique linéaire!**
 - ▶ À cause de l'interpolation en coordonnées homogènes
 - ▶ À cause du *near* et *far plane* http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html
- ▶ **On a des problèmes de *z-fighting***
 - ▶ Ça vient aussi du fait que la rasterisation n'est pas "constante"
 - ▶ C'est accentué par la non-linéarité du *z*



Bien optimiser ses near/far!

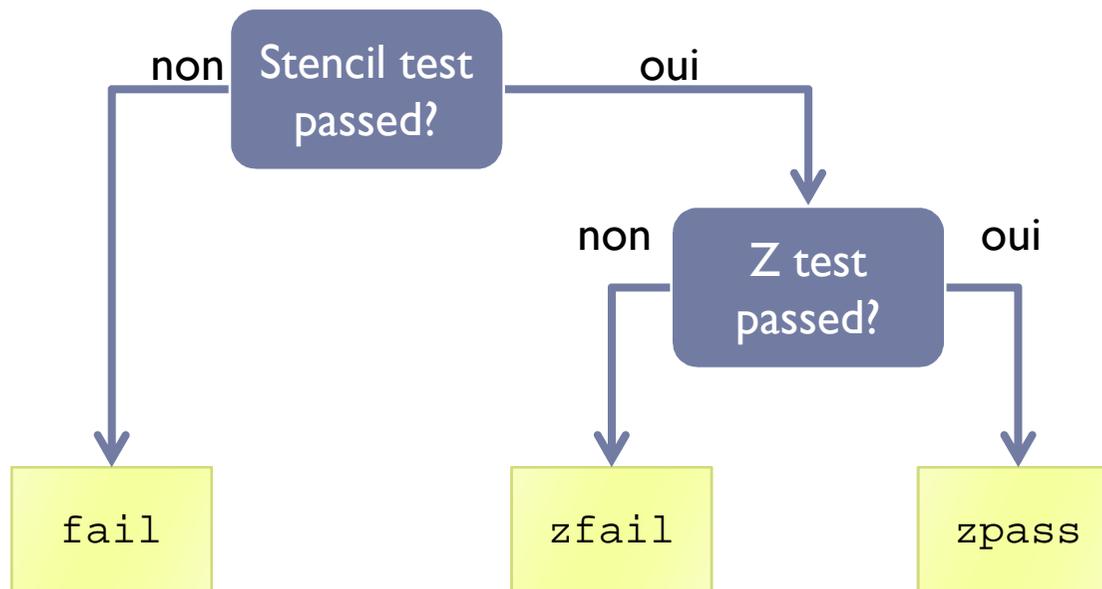
On reparlera de ce problème dans le cours sur les ombres

Stencil Test (1 / 2)

Attention, pas du fragment!

- ▶ Compare le stencil du pixel avec une constante
`glStencilFunc(GL_EQUAL, 1, ~0)`
- ▶ Met à jour par incrément/décément ou constante
`glStencilOp(GL_INCR, GL_KEEP, GL_REPLACE)`

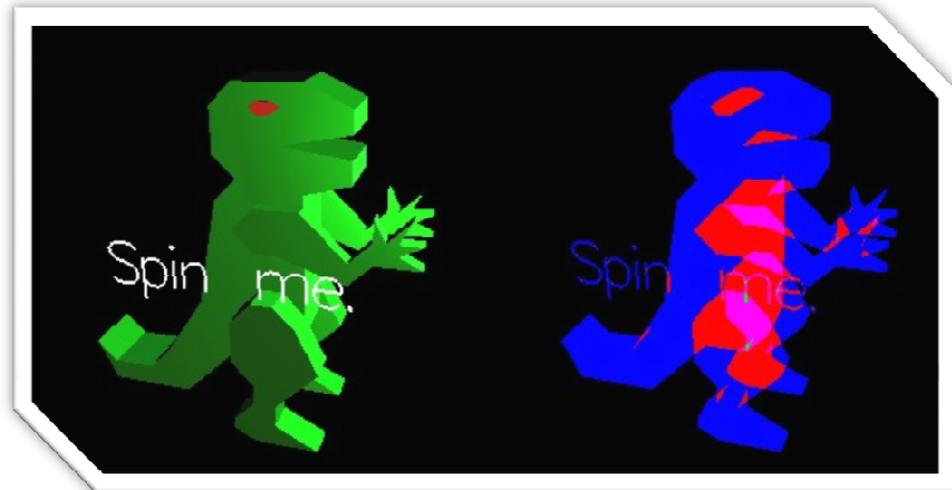
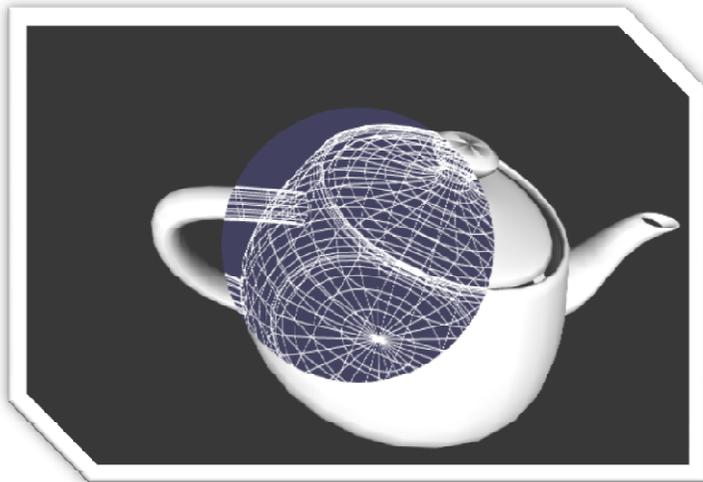
fail zfail zpass



Stencil Test (2/2)

- ▶ Applications nombreuses

- ▶ Effectuer un traitement spécial dans une zone du *viewport*
- ▶ Mesurer l'*overdraw*
- ▶ Calculer des ombres (cf. cours sur les ombres)
- ▶ ...



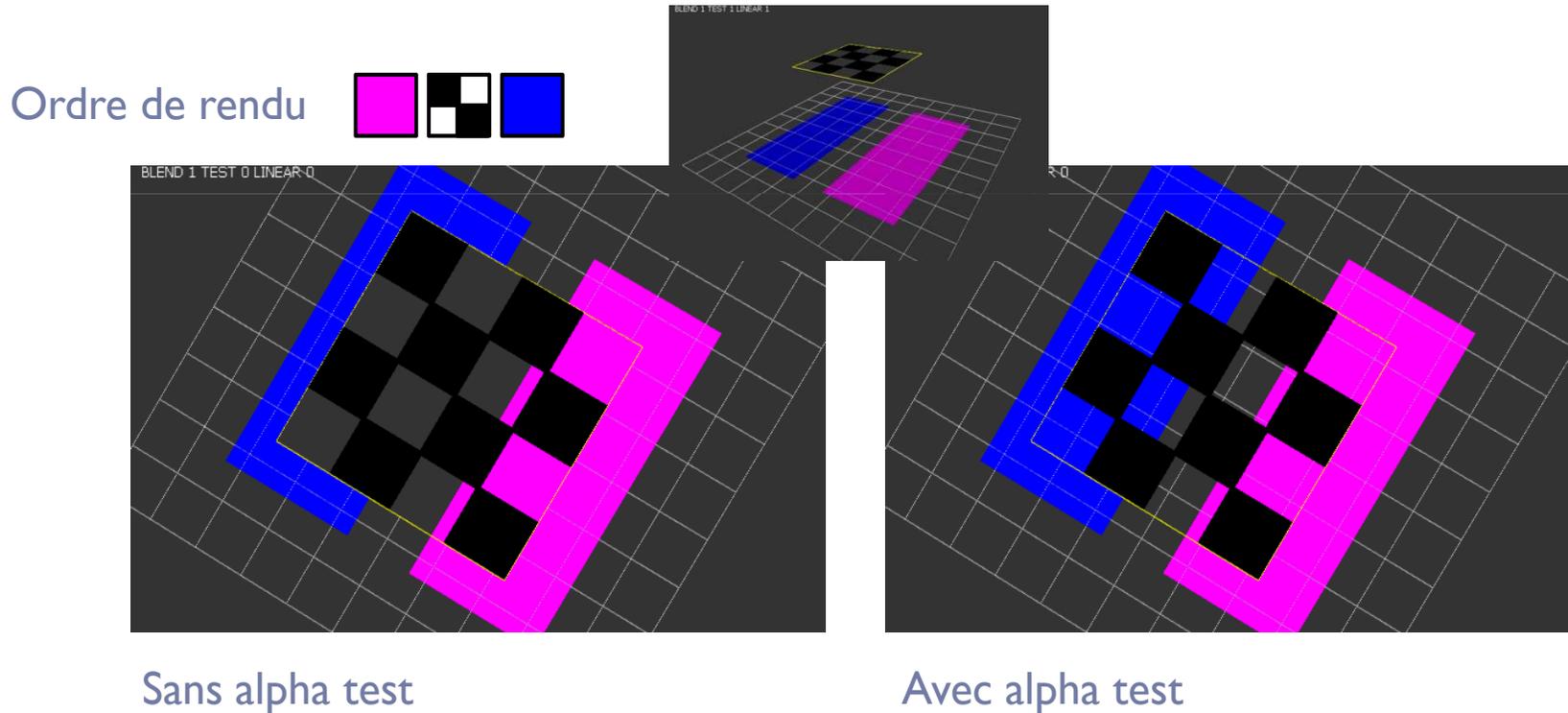
<http://excamera.com/articles/2/index.html>

Alpha test

- ▶ **Compare le A à une constante**
`glAlphaFunc(GL_GREATER, 0.0f)`
- ▶ **Discarde le fragment en cas d'échec**
 - ▶ Notamment, n'écrit rien dans le z-buffer
 - ▶ On va voir dans un instant à quoi ça sert...

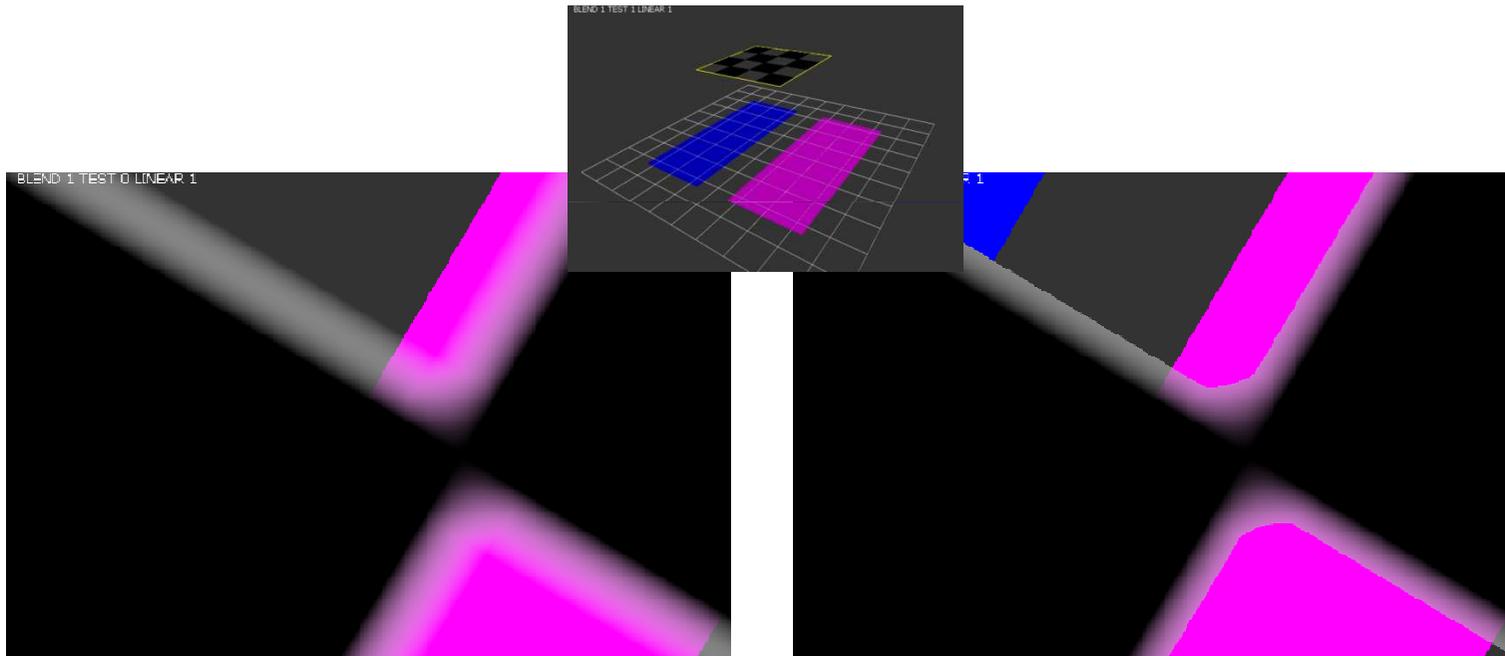
Alpha test & alpha blending (1 / 2)

- ▶ L'alpha test permet de régler le problème du tri pour du blending "binaire" ($A=0$ ou 1)



Alpha test & alpha blending (2/2)

- ▶ Mais ça crée des problèmes avec la magnification linéaire



Accéder au framebuffer

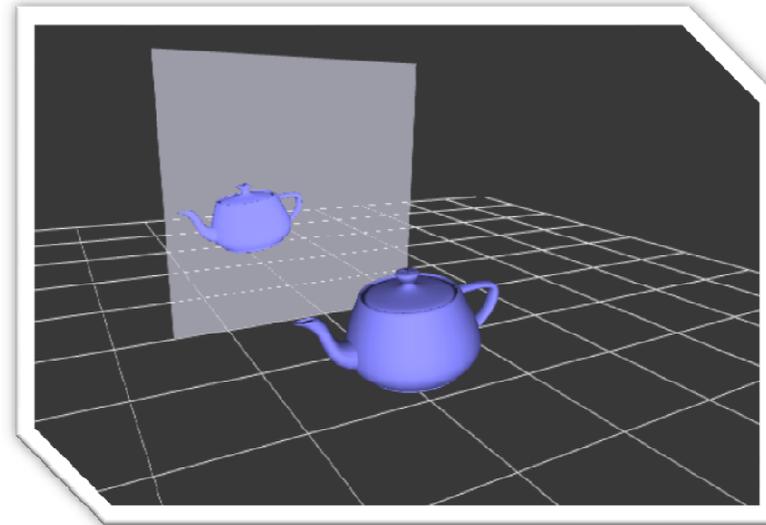
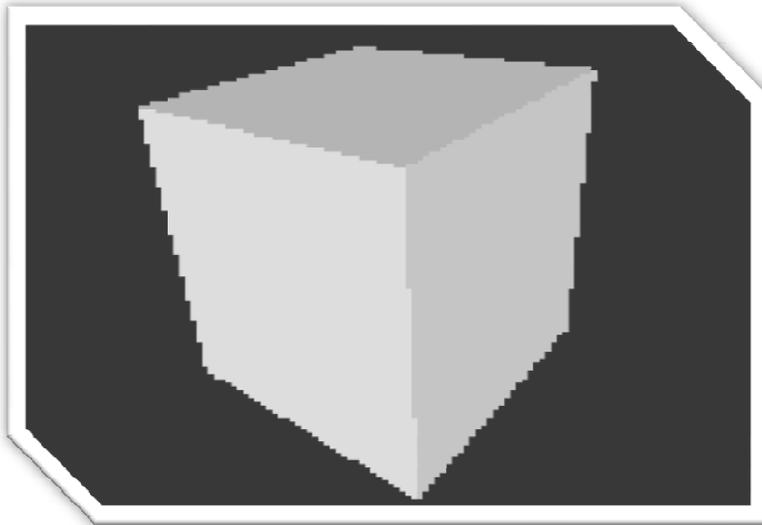
- ▶ On “voit” la couleur dans la fenêtre associée au contexte OpenGL, mais quid des autres composantes?
- ▶ On peut les récupérer avec un `glReadPixels()`
 - ▶ Transfert CPU/GPU potentiellement coûteux
 - ▶ Visualisation à définir (e.g. code couleur pour stencil)
- ▶ On peut utiliser un framebuffer temporaire
 - ▶ On voit ça tout de suite avec le rendu *offscreen*...
- ▶ On peut reprogrammer la couleur des fragments
 - ▶ On voit ça juste après le rendu *offscreen*!

Rendu *offscreen* (1 / 2)

- ▶ Rendu dans un framebuffer qui n'est pas associé à une fenêtre à l'écran (utilise sa propre zone mémoire)
 - ▶ Utilise son propre contexte OpenGL: *P-buffer*
 - ▶ Partage le contexte avec celui de la fenêtre: *framebuffer*
 - ▶ Fait la même chose qu'un `glReadPixels+glTexImage` mais sans transferts CPU/GPU
- ▶ Un peu technique à mettre en oeuvre
 - ▶ Et encore, aujourd'hui c'est standardisé!
 - ▶ Partir d'un exemple qui marche (cf. site web du cours)
- ▶ Quelques limitations
 - ▶ Le multisampling ne marche pas sur les *framebuffer*
 - ▶ Le z-buffer nécessite une recopie manuelle

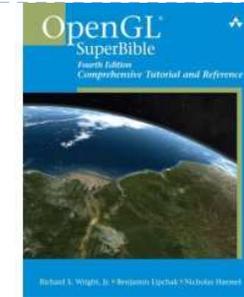
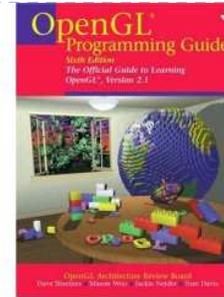
Rendu *offscreen* (2 / 2)

- ▶ Permet de faire des calculs temporaires
- ▶ Pleins d'applications en graphisme
 - ▶ Exemple du miroir
 - ▶ *Shadow map* (cf. le cours sur les ombres)
- ▶ Pleins d'applications aux calculs génériques



Bilan (soufflons un peu)

- ▶ On a vu beaucoup de choses
 - ▶ Mais ce n'est qu'un petit bout!
- ▶ Beaucoup de ces choses sont comme elles sont pour des raisons historiques
 - ▶ Les cartes cablaient "en dur" certaines fonctionnalités 3D
 - ▶ C'est ce qu'on appelle le *fixed pipeline*
- ▶ En 2003, il y a eu un gros bouleversement...



Besoins en programmabilité...



Virtua Fighter
(SEGA Corporation)

NV1
50K triangles/sec
1M pixel ops/sec
1M transistors

1995



Dead or Alive 3
(Tecmo Corporation)

Xbox (NV2A)
100M triangles/sec
1G pixel ops/sec
20M transistors

2001



Dawn
(NVIDIA Corporation)

GeForce FX (NV30)
200M triangles/sec
2G pixel ops/sec
120M transistors

2003

Besoins en programmabilité...

GEFORCE™ 7800 GTX	Graphics Bus Technology	PCI Express
	Memory	512MB
	Memory Interface	256-bit
	Memory Bandwidth (GB/sec.)	54.4
	Fill Rate (billion pixels/sec.)	13.2
	Vertices/sec. (million)	1100
	Pixels per clock (peak)	24
	RAMDACs (MHz)	400

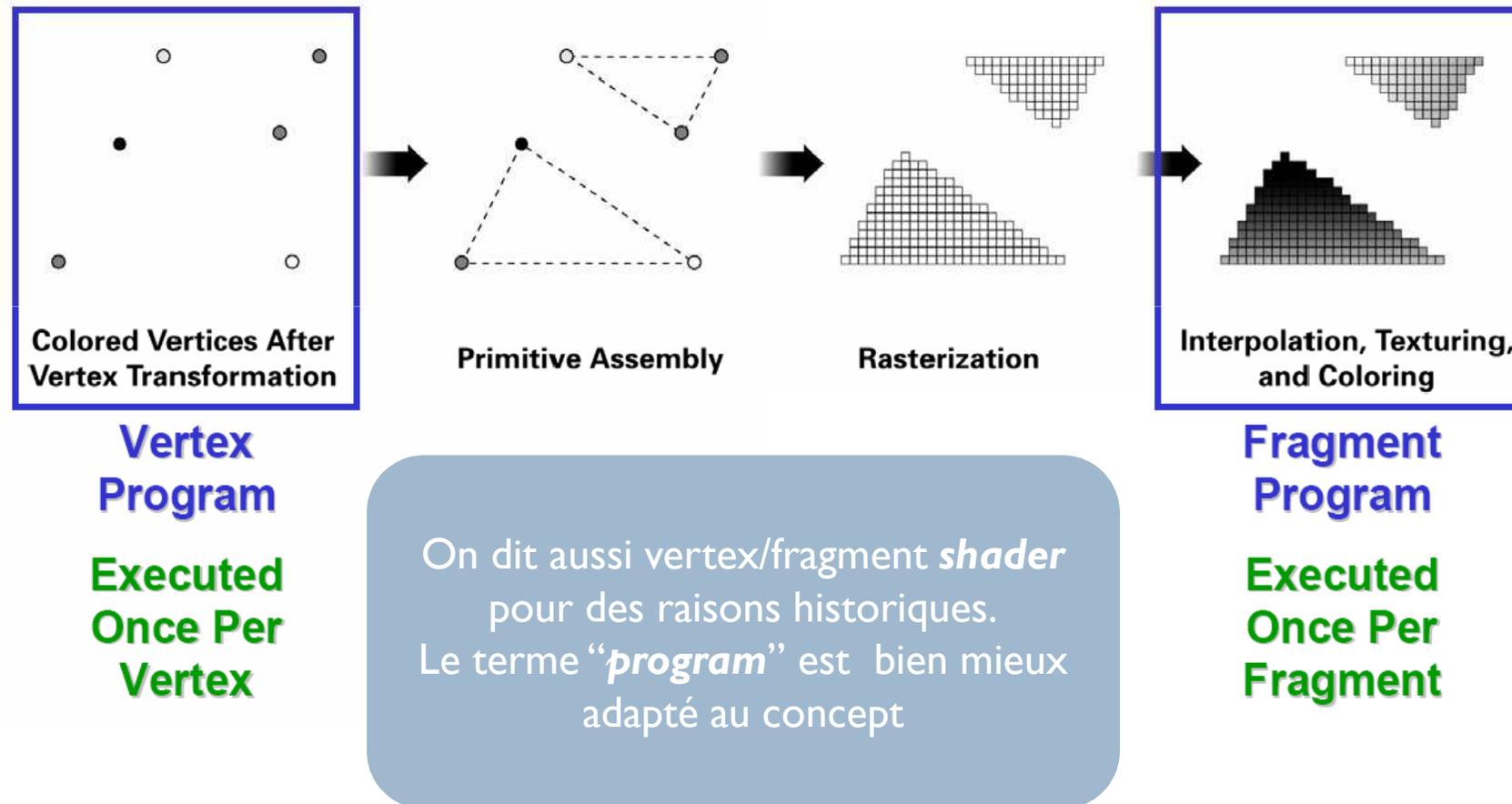


Photo courtesy of id Software, © 2004.

Besoins en programmabilité...

- ▶ **La puissance disponible augmente les attentes**
 - ▶ Plus de complexité/détails dans la forme et le *shading*
 - ▶ Un rendu temps-réel avec une qualité type cinéFX
- ▶ **Le hard est devenu programmable**
 - ▶ Dans une certaine mesure
 - ▶ Mais pas exposé par les drivers
- ▶ **Le *shading* réaliste est complexe**
 - ▶ Son design doit être *artist-friendly*
 - ▶ Son design doit être modulaire = réutilisable

Qu'est ce qui est programmable?



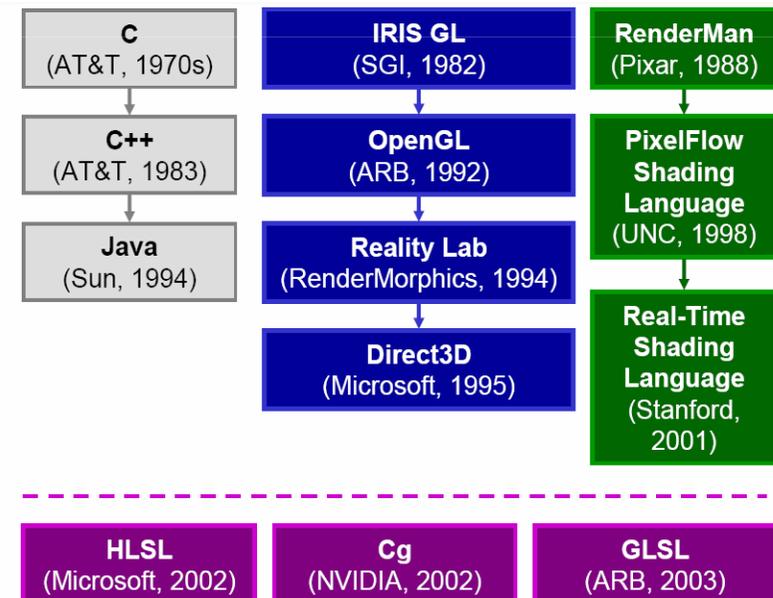
Avec quel langage?

▶ Bas niveau

- ▶ Comme l'assembleur mais un peu plus simple
- ▶ C'est l'approche historique
- ▶ Formalisée sous forme d'extension OpenGL

▶ Haut niveau

- ▶ Syntaxe proche du C/C++
- ▶ Déjà existant pour le cinéFX
- ▶ Plusieurs concurrents



Haut *vs.* bas niveau (1 / 2)

▶ Bas niveau

- ▶ Très proche du métal
 - ▶ Plus facile de comprendre les performances
 - ▶ Permet des optimisations sioux
- ▶ Dur à programmer
- ▶ Dépendent du hardware
 - ▶ Doit être ré-écrit/optimisé pour chaque plateforme
 - ▶ N'anticipe pas les évolutions du hard

Assembly

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

Haut *vs.* bas niveau (1 / 2)

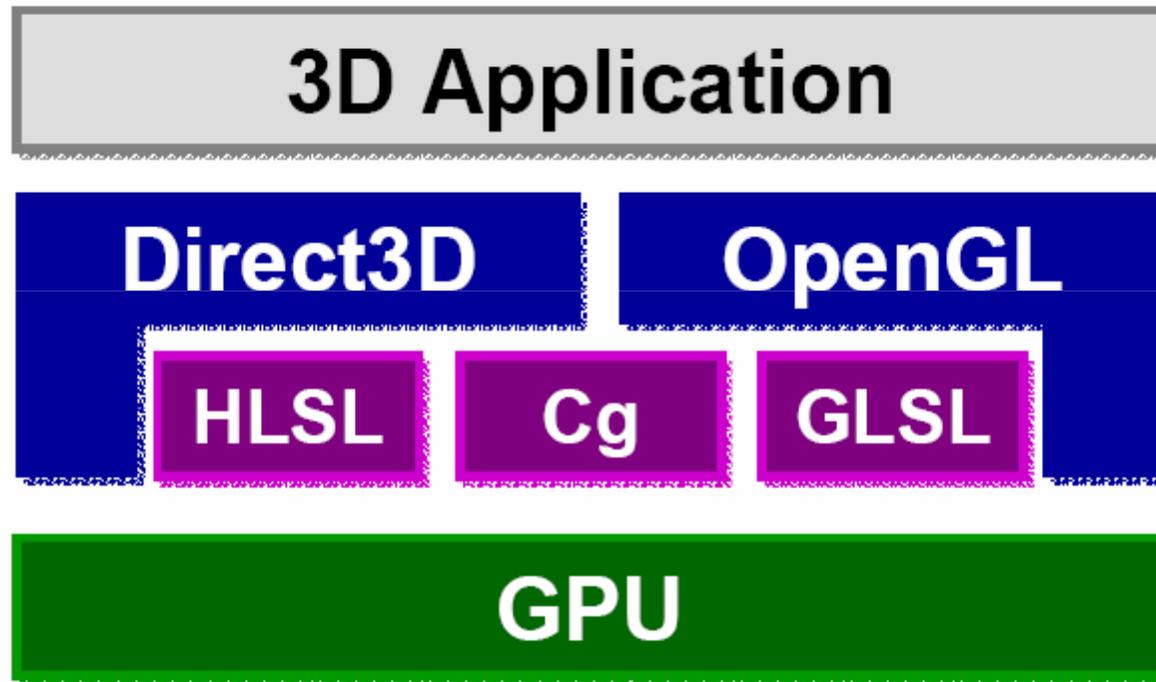
▶ Haut niveau

- ▶ Facile à programmer
- ▶ Facile à réutiliser
- ▶ Facile à réutiliser (vraiment!)
- ▶ Compilé
 - ▶ Indépendent du hardware
 - ▶ Difficile d'identifier les goulots

High-Level Language

```
...  
float3 cSpecular = pow(max(0, dot(Nf, H)),  
                      phongExp).xxx;  
float3 cPlastic = Cd * (cAmbient + cDiffuse) +  
                Cs * cSpecular;  
...
```

Les langages existants



Language features (1 / 3)

- ▶ **Control flow statements**
 - ▶ if, for, while, break, continue
 - ▶ pas de goto
- ▶ **Fonctions et struct définies par l'utilisateur**
 - ▶ bien pour la modularité et la réusabilité
- ▶ **Fonctions *builtins***
 - ▶ math :abs, pow, sqrt, step, smoothstep...
 - ▶ geometrie :dot, cross, normalize, reflect, refract...
 - ▶ texture lookups
 - ▶ accès à la machine à état OpenGL

Language features (2/3)

▶ Support pour les vecteurs et matrices

- Component-wise `+ - * /` for vectors
- Dot product
 - `dot(v1,v2); // returns a scalar`
- Matrix multiplications:
 - assuming a `float4x4 M` and a `float4 v`
 - matrix-vector: `mul(M, v); // returns a vector`
 - vector-matrix: `mul(v, M); // returns a vector`
 - matrix-matrix: `mul(M, N); // returns a matrix`

Language features (3/3)

► Nouveaux opérateurs

- Swizzle operator extracts elements from vector or matrix

```
a = b.xyy;
```

- Examples:

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);  
float2 vec2 = vec1.yx;      // vec2 = (-2.0, 4.0)  
float scalar = vec1.w;     // scalar = 3.0  
float3 vec3 = scalar.xxx;  // vec3 = (3.0, 3.0, 3.0)  
float4x4 myMatrix;  
  
// Set myFloatScalar to myMatrix[3][2]  
float myFloatScalar = myMatrix._m32;
```

- Vector constructor builds vector

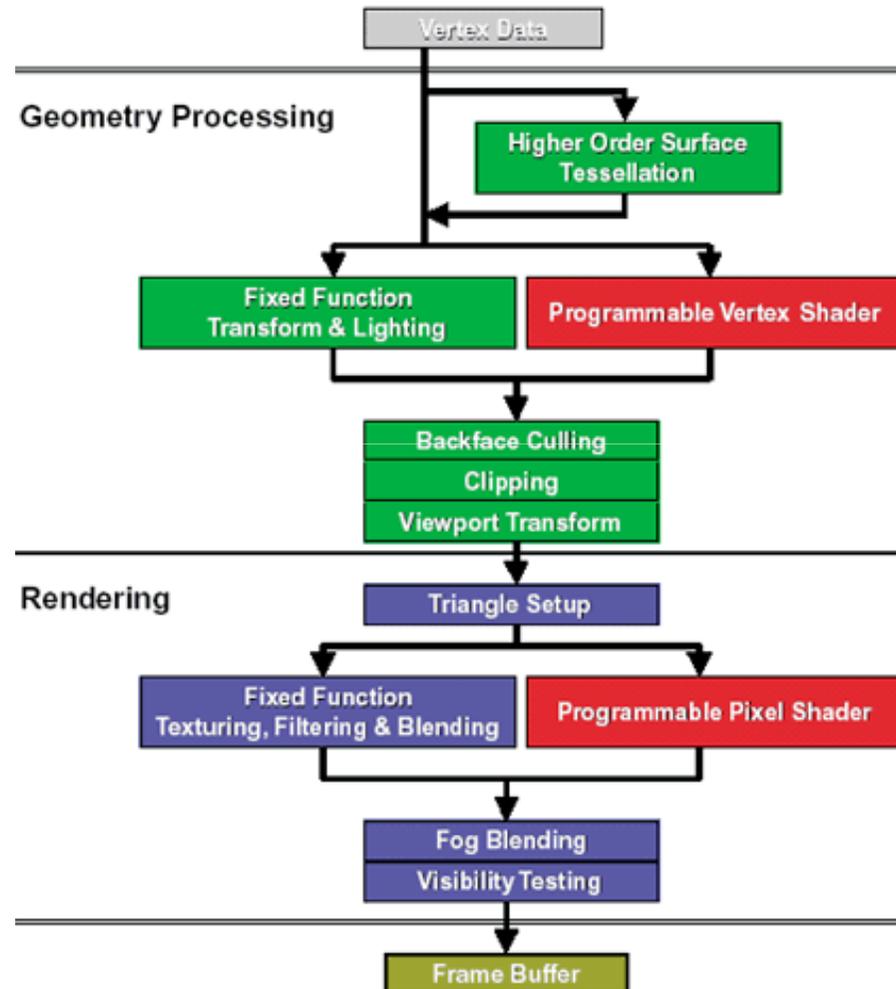
```
a = float4(1.0, 0.0, 0.0, 1.0);
```



Qu'est ce qui n'est pas programmable?

- ▶ **L'assembly est en partie programmable**
 - ▶ Ce sont les *geometry shaders/programs*
 - ▶ Seules les cartes très récentes le supportent (DX10)
 - ▶ Affecte fortement les perfs si mal utilisés
- ▶ **Ne sont pas modifiables/programmables**
 - ▶ Les tests (z,alpha,stencil)
 - ▶ L'interpolation pendant le scan convert
- ▶ **On ne peut pas accéder au framebuffer dans un *program***
 - ▶ Poserait des problèmes de concurrence (SIMD)
 - ▶ Peut parfois être contourné par du *multipass+render2texture*

Résumé (retour au cours 1)



En pratique

- ▶ Appels à l'API pour
 - ▶ Spécifier les vertex & fragment programs
 - ▶ Activer les vertex & fragment programs
 - ▶ Passer les paramètres “globaux” aux programs
- ▶ Géométrie rendue comme précédemment
 - ▶ Le *vertex program* sera exécuté pour chaque *vertex*
 - ▶ Le *fragment program* sera exécuté pour chaque *fragment*
 - ▶ Sauf en cas d'échec du z-test si le shader ne modifie pas le z
 - ▶ C'est ce qu'on appelle *early z rejection*

Exemple: *toon shading* (1 / 3)

```
varying vec3 normal;
```

Fichier toon.vert

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

```
varying vec3 normal;
```

Fichier toon.frag

```
uniform vec3 t;
```

```
void main() {  
    vec4 color;  
    vec3 n = normalize(normal);  
    float i = dot(vec3(gl_LightSource[0].position),n);  
    if (i>threshhold[0]) color = vec4(1.0,0.5,0.5,1.0);  
    else if (i>threshhold[1]) color = vec4(0.6,0.3,0.3,1.0);  
    else if (i>threshhold[2]) color = vec4(0.4,0.2,0.2,1.0);  
    else color = vec4(0.2,0.1,0.1,1.0);  
  
    gl_FragColor = color;  
}
```

Exemple: *toon shading* (2/3)

```
GLhandleARB v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLhandleARB f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
char* vs = vs = textFileRead("toon.vert");
char* fs = textFileRead("toon.frag");
const char* vv = vs;
const char* ff = fs;
glShaderSourceARB(v, 1, &vv, NULL);
glShaderSourceARB(f, 1, &ff, NULL);
free(vs);
free(fs);
glCompileShaderARB(v);
glCompileShaderARB(f);

GLhandleARB p = glCreateProgramObjectARB();
glAttachObjectARB(p, v);
glAttachObjectARB(p, f);
glLinkProgramARB(p);

glUseProgramObjectARB(p);
```

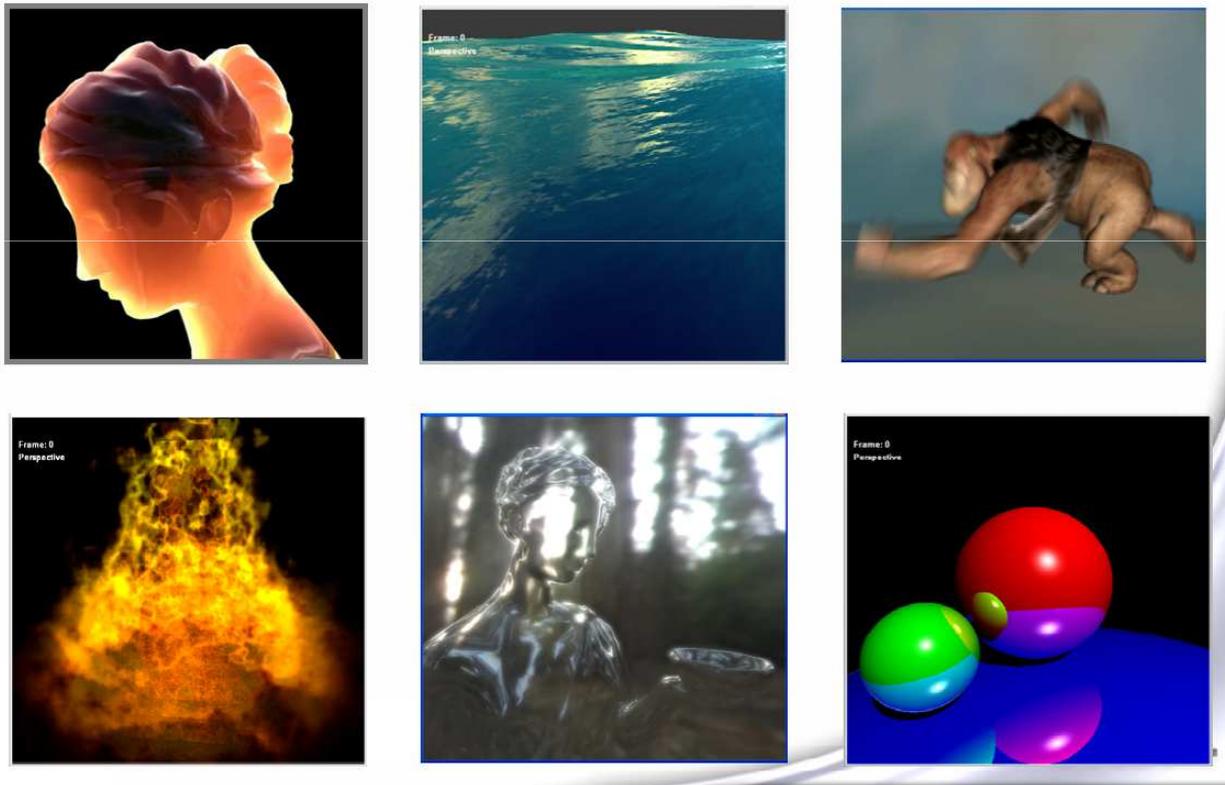
Exemple: *toon shading* (3 / 3)

```
// once for all (in QGLViewer::init())
GLint thresholdParam = glGetUniformLocationARB(p, "threshold");

// at every frame (in QGLViewer::draw())
glUseProgramObjectARB(p);
glUniform3fARB(thresholdParam, 0.95f, 0.5f, 0.25f);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
// draw teapot
glNormal3fv(...); glVertex3v(...);
...
glEnd();
glUseProgramObjectARB(0);
```

Applications des shaders/programs

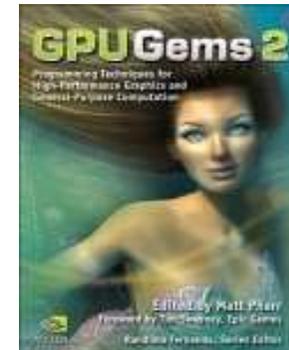
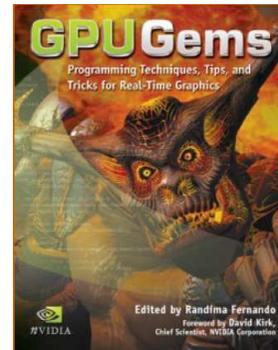
- ▶ Computer graphics
 - ▶ Pleins de beaux effets



Applications des shaders/programs

- ▶ **Computer graphics**

- ▶ Pleins de beaux effets
- ▶ Des structures d'accélération
- ▶ Voir *GPU Gems I & II*



http://developer.nvidia.com/object/gpu_gems_2_home.html

- ▶ **General Purpose GPU (GPGPU)**

- ▶ Algèbre linéaire
- ▶ Simulation
- ▶ ...

C'est fini pour aujourd'hui



C'est l'heure de la pause

(can you find the hidden tiger?)