Valeur C CAM & DEA MM
Option Synthèse d'images avancée

Les *shaders*

Doom III

Halo 2

Jet Set Radio Future

Alexandre Topol

Département Informatique
Conservatoire National des Arts & Métiers

2003-2004

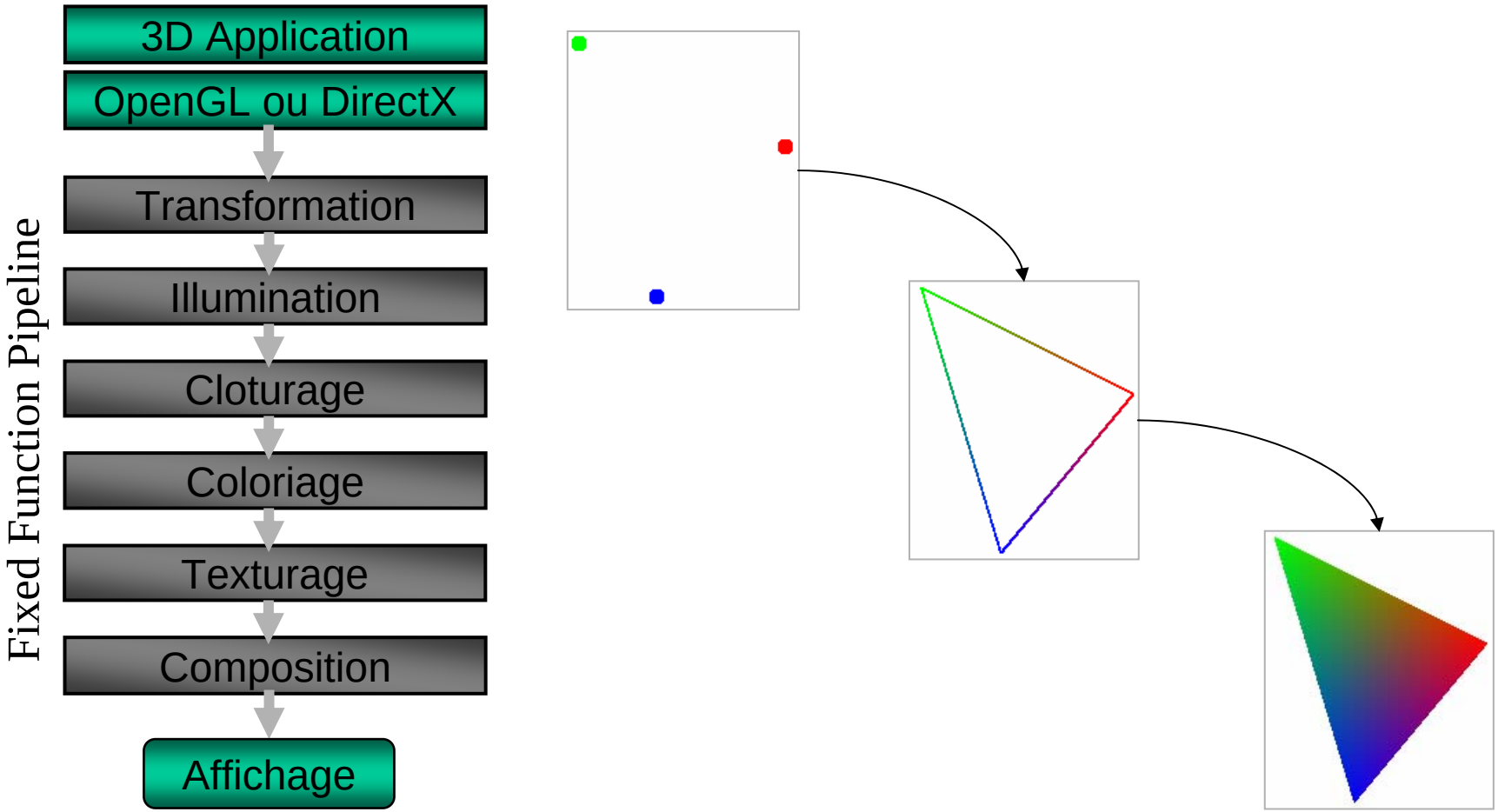
Plan du cours

- Le pipeline de rendu sur les cartes graphiques actuelle
- Les langages de *shading* bas niveau
  - *Vertex shader*
  - *Fragment/pixel shader*
- Les langages de *shading* haut niveau

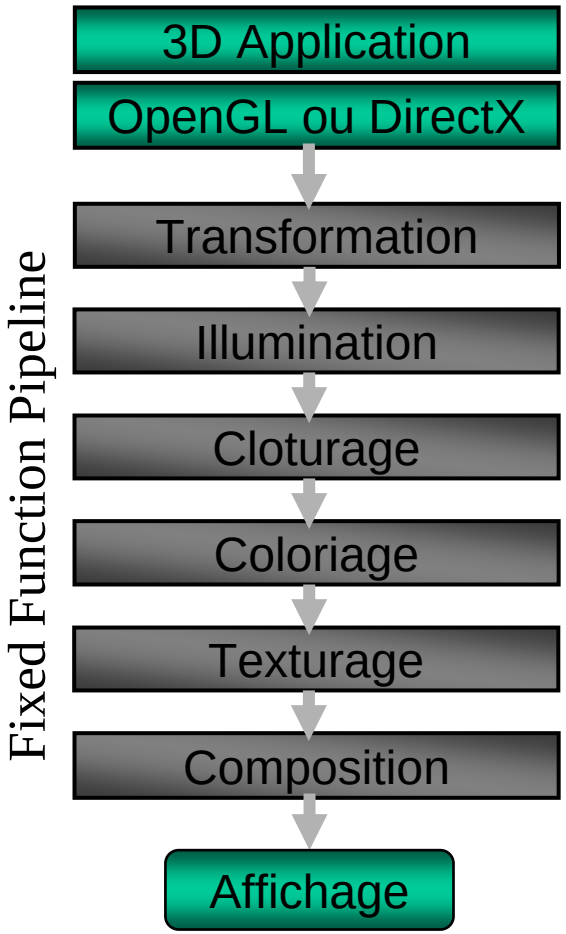
Présentation

- La plupart des étapes du pipeline 3D peuvent être programmées
- Les programmes de *shading* ont pour but de changer les comportement par défaut du matériel graphique
  - Transform and lighting : vertex shaders / vertex programs
  - Fragment processing : pixel shaders / fragment programs
- Historique: depuis le pipeline classique vers le pipeline programmable

Le pipeline 3D classique



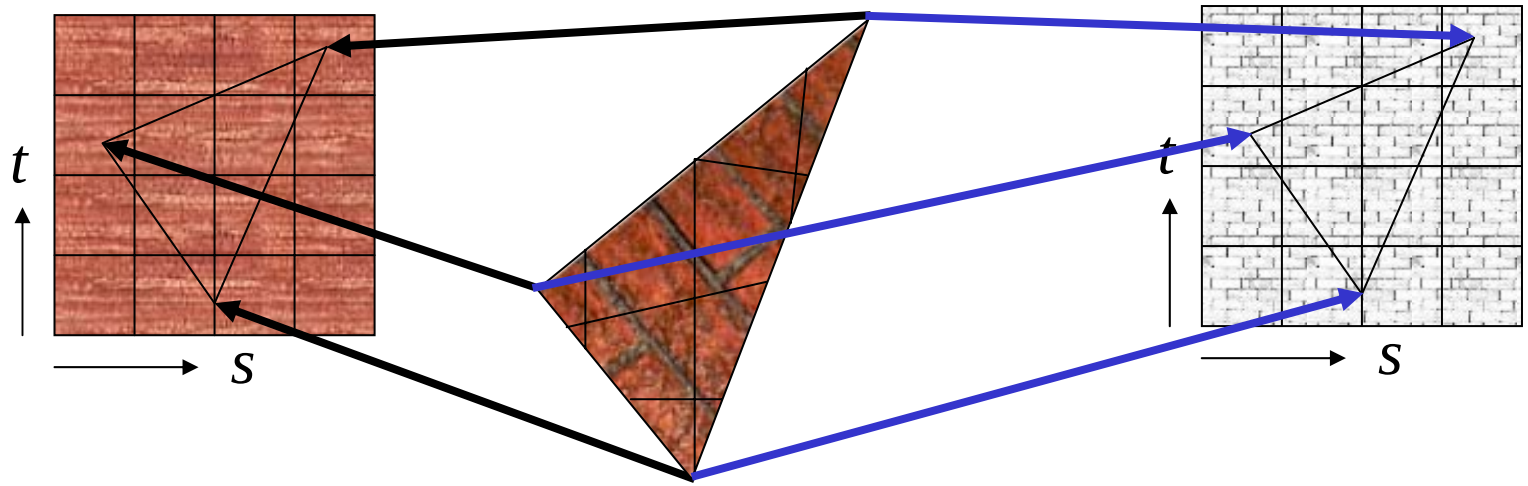
Le pipeline 3D classique



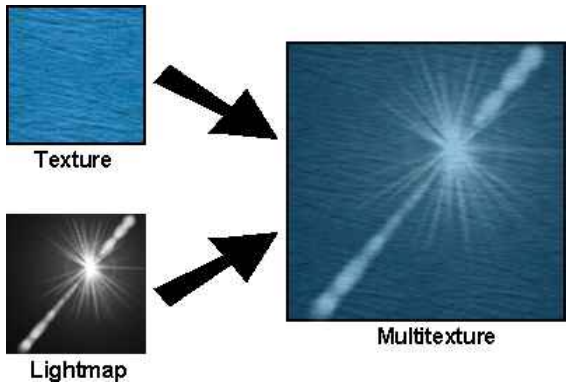
Multi-texturage

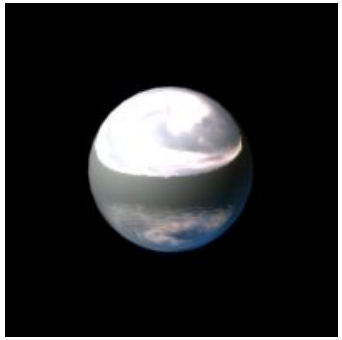
- Les GPU évoluent très vite pour répondre aux besoins de réalisme des jeux vidéos
- La précédente évolution (avant les shaders) :
  - *Transform & Lightning (TnL, T&L)*
  - *Multi-texture, Bump Mapping, Environnement Mapping*
- Permet de gérer plusieurs textures pour faire différentes choses

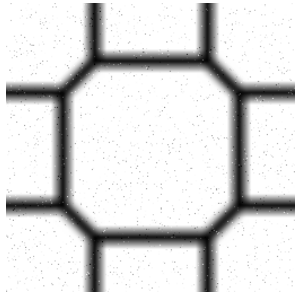
Multi-texturage



Multi-texturage



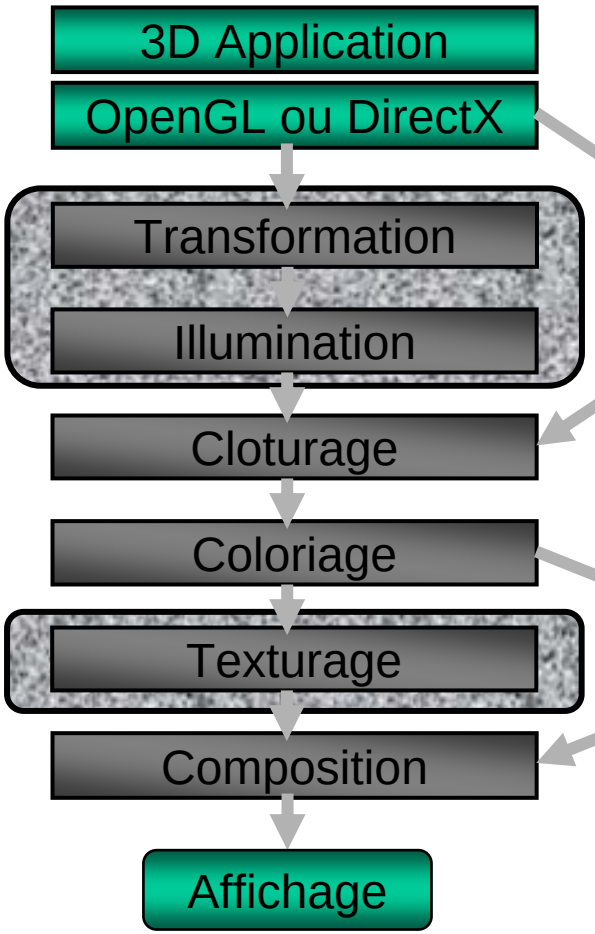




Multi-texturage

- Problème : chaque algorithme de rendu utilisant les textures (les *maps*) doit être implémenté traité par le pipeline
- Ex : rajouter un coloriage de type cartoon ?
  - C'est une nouvelle utilisation des textures
  - Mais pas implémenté sur les cartes graphiques
- Nécessité de pouvoir programmer les effets

Pipeline programmable



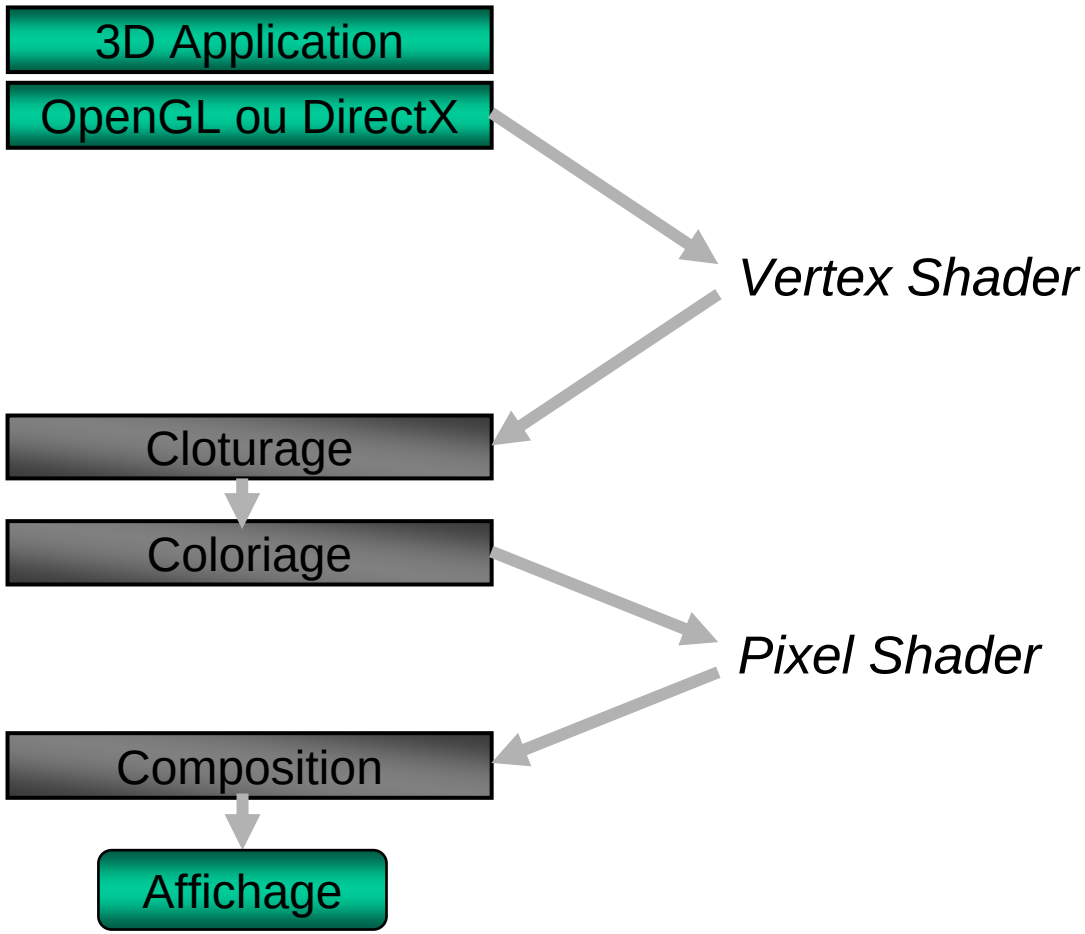
Vertex Shader

- Opère sur chaque sommet de la scène
- Effets
  - Blur
  - Réflexion Diffuse et spéculaire

Pixel Shader

- Opère sur chaque pixel
- Effets
  - Texturing
  - Fog blending

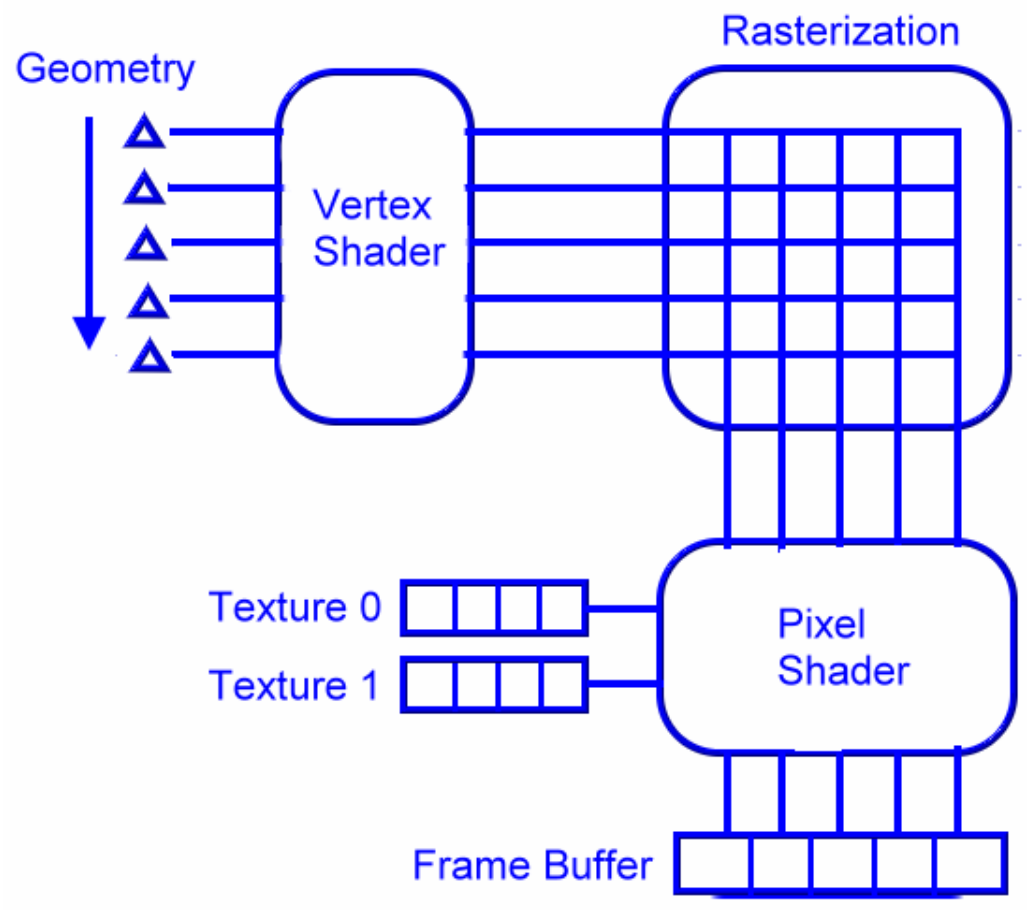
Pipeline programmable



Fonctionnement

- Comment spécifier les vertex et pixel shaders ?
  - Par des langages bas-niveau (comme de l'assembleur)
  - Par des langages haut-niveau (comme du C)
- Les données échangées par les composants
  - Per-vertex data (pour les vertex shaders)
  - Per-fragment data (pour les pixel shaders)
  - Informations uniformes (constantes) :
    modelview matrix, material parameters, …

Fonctionnement



Plan du cours

- Le pipeline de rendu sur les cartes graphiques actuelle
- **Les langages de *shading* bas niveau**
  - *Vertex shader*
  - *Fragment/pixel shader*
- Les langages de *shading* haut niveau

Les APIs bas-niveau

- Comme de l'assembleur
  - Proche des fonctionnalités du matériel
  - En entrée : les attributs de vertex/fragment
  - En sortie : de nouveaux attributs de vertex/fragment
  - Séquence d'instructions sur les registres
  - Control de flux très limité (s'il y en a)
  - Dépendant de la plateforme
    Mais : il y a une certaine convergence

Les APIs bas-niveau

- Les APIs bas-niveau courantes :
  - Les extensions OpenGL : GL_ARB_vertex_program, GL_ARB_fragment_program
  - DirectX 9 : Vertex Shader 2.0, Pixel Shader 2.0
- Les anciennes :
  - DirectX 8.x : Vertex Shader 1.x, Pixel Shader 1.x
  - Les extensions OpenGL : GL_ATI_fragment_shader, GL_NV_vertex_program, …

Pourquoi utiliser les APIs bas-niveau ?

- Elles offrent de meilleurs performance et fonctionnalités
- Elles aident à comprendre le fonctionnement des cartes graphiques (ATI’s r300, NVIDIA’s nv30)
- Elles aident à comprendre les APIs haut-niveau (Cg, HLSL, ...)
- Plus faciles à utiliser que de spécifier un pipeline graphique configurable (par registres)

Application des vertex shader

- Calcul « sur mesure » des attributs de chaque sommet
- Calcul de tout ce qui peut être interpolé linéairement entre sommets
- Limitations:
  - Les sommets ne peuvent ni être générés ni détruits
  - Aucun information sur l'ordre des sommets n'est accessible
  - Les sommets se traitent indépendamment les uns des autres
- Exemples en OpenGL (pareil en Direct3D)

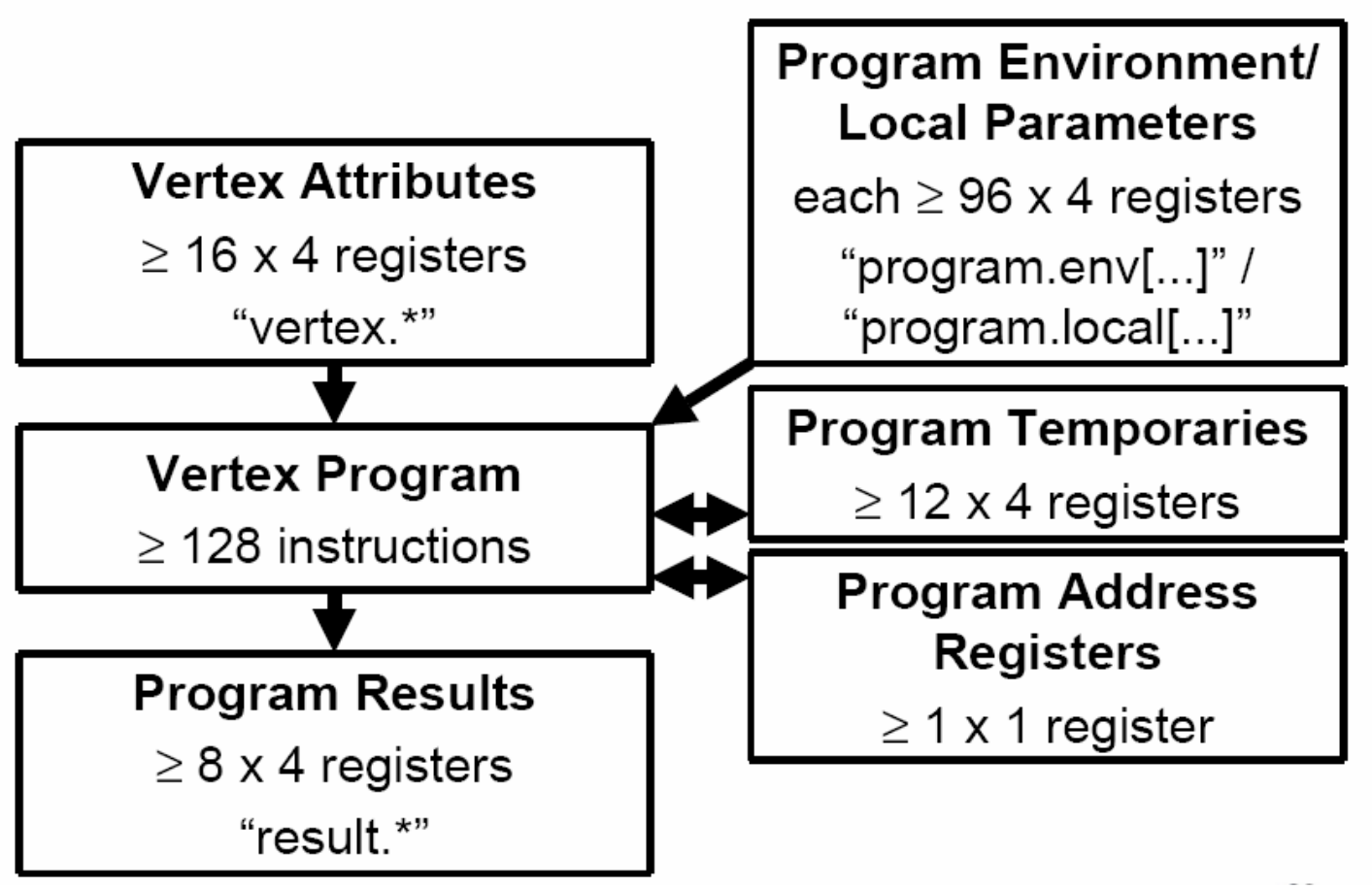
OpenGL GL_ARB_vertex_program

- Shunte le traitement des sommets dans le pipeline classique
- Qu’est-ce qui est fait dans un vertex shader ?
  - Transformations des sommets
  - Calcul des normales aux sommets
  - Gestion des matériaux par sommets
  - Illumination des sommets
  - Génération des coordonnées de textures
  - Matrices de transformation des textures

OpenGL GL_ARB_vertex_program

- Ce qui n’est pas remplacé ?
  - *Clipping* au volume de vue
  - La projection (perspective ou parallèle)
  - Calcul de la position finale à l’écran
  - Les transformations dues à la profondeur (*fog*)
  - Les opération de coloriage
    ➔ pixel shader pour du « sur mesure »

OpenGL GL_ARB_vertex_program
Modèle de la machine



OpenGL GL_ARB_vertex_program
Les variables

- Les attributs des sommets
  - vertex.position, vertex.color, …
  - Liason explicite :

```
ATTRIB name = vertex.*;
```

- Variables d'états
  - state.material.* (diffuse, ...), state.matrix.* (modelview[*n*], …), …
- Résultats du programme et variables en sortie
  - result.color.* (primary, secondary, …), result.position, ...

```
OUTPUT name = result.*;
```

OpenGL GL_ARB_vertex_program
Les instructions

- Ensemble d'instructions
  - 27 instructions
  - Opèrent sur des flottants ou sur des vecteurs de 4 flottants
  - La syntaxe de base :

```
OP destination [,src1 [,src2 [,src3]]]; # comm
```

  - Exemple :

```
MOV result.position, vertex.position;
```

  - Opération numériques : ADD, MUL, LOG, EXP, …
  - Modifieurs : negation, masquage, saturation

OpenGL GL_ARB_vertex_program Exemple

```
!!ARBvp1.0
# Les entrées
ATTRIB pos = vertex.position;
ATTRIB col = vertex.color;
# Les sorties
OUTPUT clippos = result.position;
OUTPUT newcol = result.color;
# Les variables d'états
PARAM modelviewproj[4] = { state.matrix.mvp
};
# Le programme
DP4 clippos.x, modelviewproj[0], pos;
DP4 clippos.y, modelviewproj[1], pos;
DP4 clippos.z, modelviewproj[2], pos;
DP4 clippos.w, modelviewproj[3], pos;
MOV newcol, col;
END
```

DirectX 9 Vertex Shader

- Vertex Shader 2.0 utilisable dans DirectX 9.0
- Même fonctionnalités et limitations que OpenGL GL_ARB_vertex_program
- Registres et syntaxe similaires
- Quelques différences
  - Soumission des données par constantes (et non par attributs de sommets)
  - Blocs conditionnels, boucles, procedures

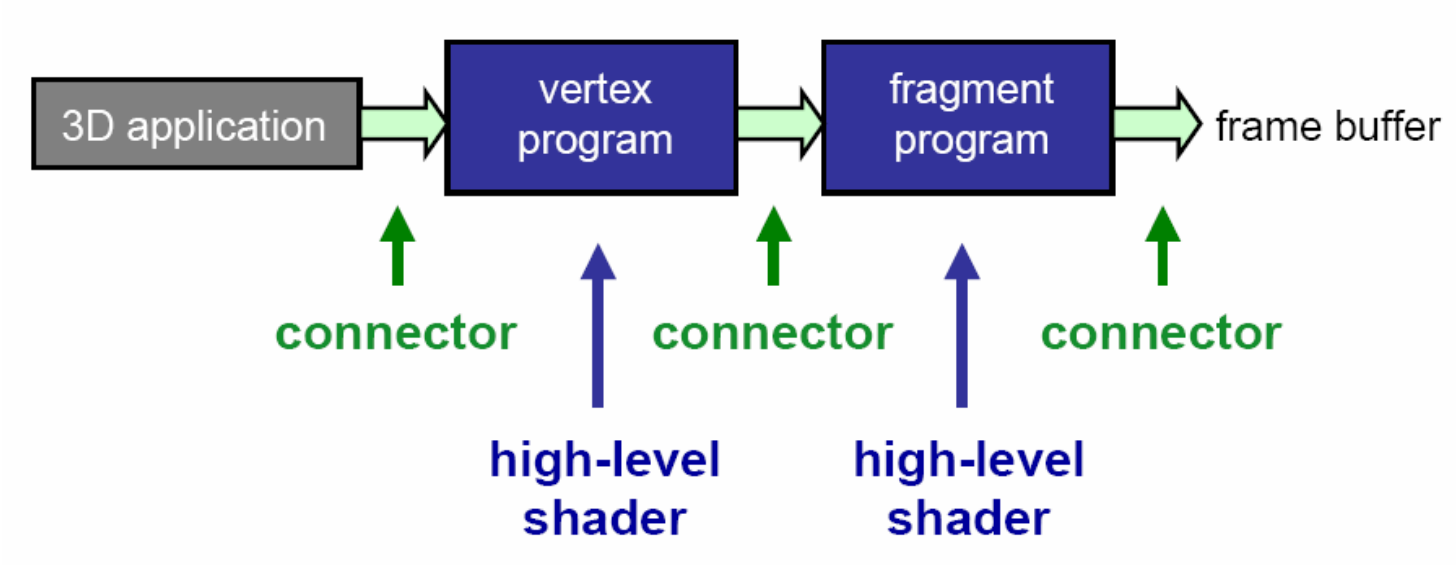
Exemple de vertex shaders

- Exemple sur les sommets : DirectX SDK
  DolphinVS.exe

- Exemple sur les lumières : DirectX SDK
  LightningVS.exe

Liens entre les deux shaders

- Au niveau de l'API 3D il faut spécifier quels shaders employer
- Cela se fait par connexions entre le FFP, et les deux shaders



Plan du cours

- Le pipeline de rendu sur les cartes graphiques actuelle
- Les langages de *shading* bas niveau
  - *Vertex shader*
  - ***Fragment/pixel shader***
- Les langages de *shading* haut niveau

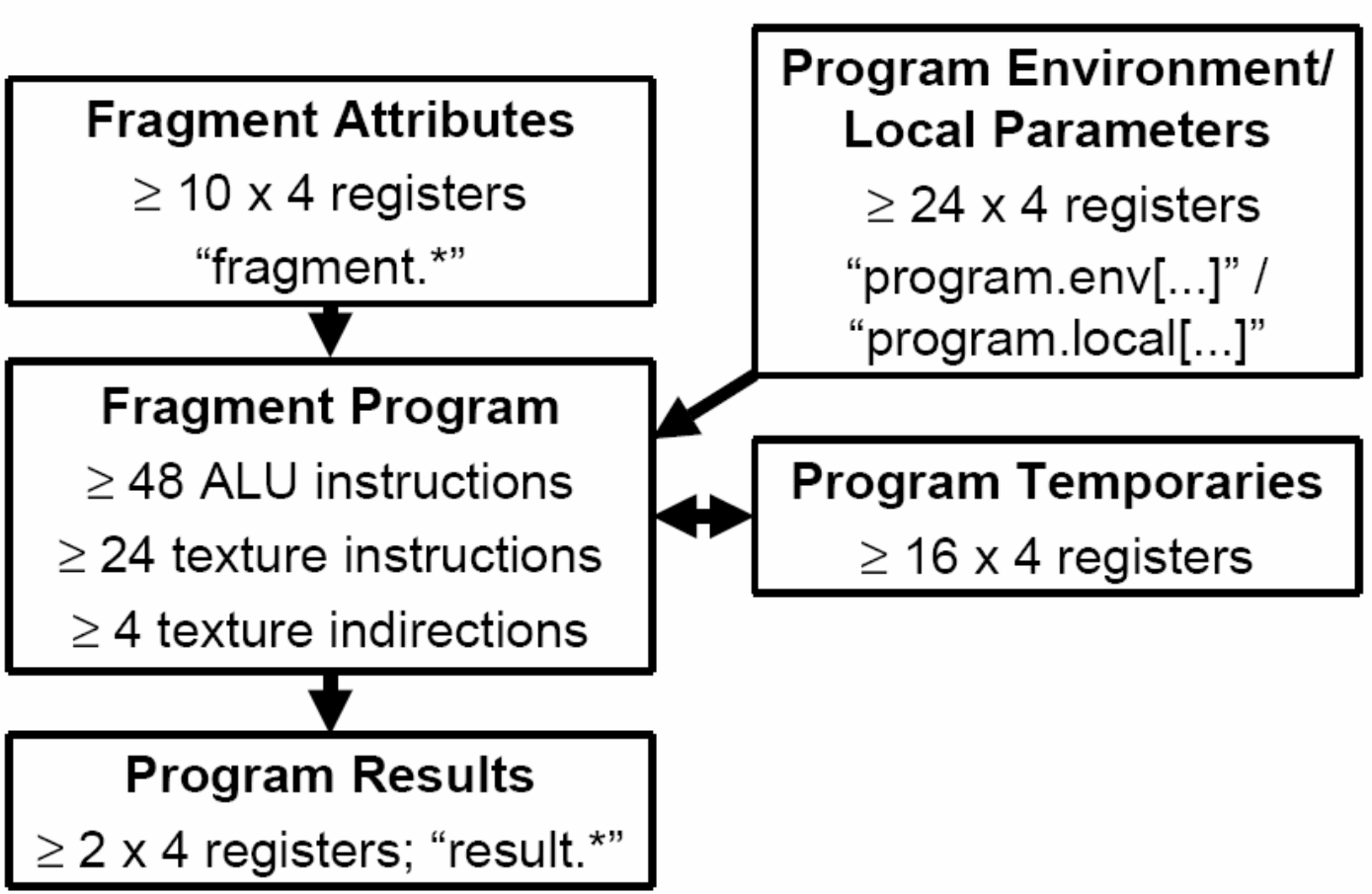
Application des pixel shader

- Calcul sur mesure des attributs d'un fragment
- S'y trouve tout ce qui doit être calculé par pixel
- Limitations:
  - Les fragments ne peuvent être générés
  - La positions des fragments ne peut être changée
  - Aucune information sur la géométrie de l'objet n'est fourni à ce niveau

OpenGL GL_ARB_fragment_program

- Shunte l'étape de coloriage du pipeline classique
- Qu'est-ce qui est fait dans un pixel shader ?
  - Texturage
  - Calcul de couleurs
  - Brouillard
  - Tout ce qui touche au coloriage des primitives géométriques (points, lignes, polygones, bitmaps)
- Ce que ne fait pas un pixel shader
  - Les tests sur les fragments (alpha, profondeur, …)
  - Mélanges (*blending*)

OpenGL GL_ARB_fragment_program
Modèle de la machine



OpenGL GL_ARB_fragment_program Instructions

- Ensemble d'instructions
  - Similaires à celles du vertex shader
  - Opèrent sur des flottants et des vecteurs de 4 flottants
- Echantillonnage de textures

```
OP destination, source, texture[index], type;
```

  - Texture types: 1D, 2D, 3D, CUBE, RECT

```
TEX result.color,fragment.texcoord[1],texture[0],2D;
```

échantillonne une texture 2D stockée en 0 avec l'ensemble 1 des coordonnées de texture et écrit le résultat dans result.color

OpenGL GL_ARB_fragment_program Exemple

```
!!ARBfp1.0
# Les entrées
ATTRIB tex = fragment.texcoord;
ATTRIB col = fragment.color.primary;
# La sortie
OUTPUT outColor = result.color;
# Le programme
TEMP tmp;
TXP tmp, tex, texture[0], 2D;
MUL outColor, tmp, col;
END
```

DirectX 9 Pixel Shader

- Pixel Shader 2.0 dans DirectX 9.0
- Fonctionnalités et limitations similaires à GL_ARB_fragment_program
- Registres et syntaxe similaires également

DirectX 9 Pixel Shader Instructions

- Déclaration des textures utilisées

```
dcl_type s*
```

- Déclaration des couleurs et coordonnées de texture

```
dcl v*[.mask]
dcl t*[.mask]
```

- Ensemble d'instructions
  - Operate on floating-point scalars or 4-vectors

```
op destination [, src1 [, src2 [, src3]]]
```

- Echantillonnage de texture

```
op destination, source, sn
```

DirectX 9 Pixel Shader
Exemple

```
ps_2_0

dcl_2d s0
dcl t0.xy

texld r1, t0, s0
mov oC0, r1
```

Déclaration de la version du langage de shading utilisé

Déclaration d'une texture et des coordonnées de texture

Le programme : calcul de la couleur en sortie en fonction de la texture à appliquer

Attachement des shaders au pipeline

- Ca dépend de l'API utilisée
- Fonctions du type :
  - CreatePixelShader
  - SetPixelShader
  - SetPixelShaderParameters
  - DeletePixelShader
  - …

Exemple de pixel shaders

- Exemple sur le bump mapping : DirectX SDK
  BumpEarth.exe
- Exemple sur l'environnement : DirectX SDK
  BumpLens.exe
- Exemple de distorsion : DirectX SDK
  FishEye.exe

Plan du cours

- Le pipeline de rendu sur les cartes graphiques actuelle
- Les langages de *shading* bas niveau
  - *Vertex shader*
  - *Fragment/pixel shader*
- **Les langages de *shading* haut niveau**

Les langages haut niveau

- Leur utilité ?
  - Pour éviter de programmer, de déboguer, de maintenir de longs programmes de shading en assembleur
  - Facile à lire
  - Facile à modifier
  - Optimisation automatique du code
  - Nombreuses plateformes compatibles

Assembleur vs. Langage haut-niveau

En assembleur

```
…
dp3 r0, r0, r1
max r1.x, c5.x, r0.x
pow r0.x, r1.x, c4.x
mul r0, c3.x, r0.x
mov r1, c2
add r1, c1, r1
mad r0, c0.x, r1, r0
…
```

Blinn-Phong shader écrit en utilisant les deux méthodes

En langage haut-niveau

```
…
float4 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float4 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
…
```

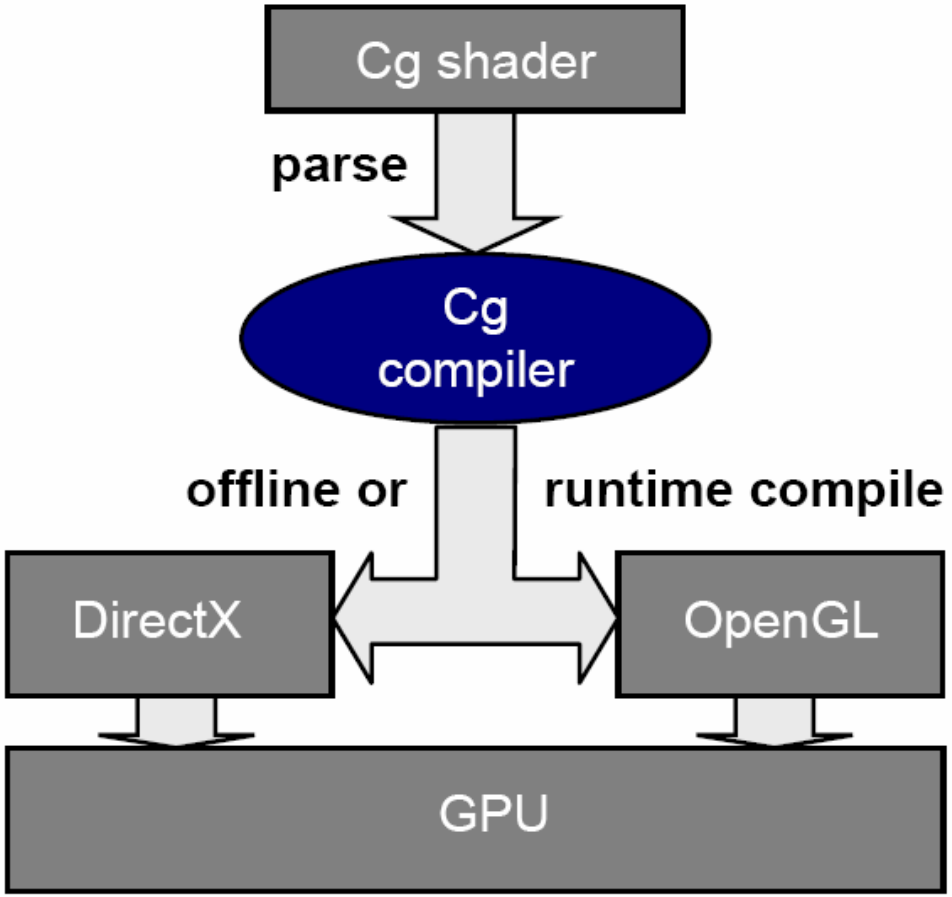
Les langages haut niveau

- Renderman – la référence issue du raytracing
- Cg
  - “C for Graphics”
  - De NVIDIA
- HLSL
  - “High-level shading language”
  - Fait partie DirectX 9 (Microsoft)
- OpenGL 2.0 Shading Language
  - Proposé par 3D Labs

Cg

- Exemple typique des langages haut-niveau
- Quasi identique à DirectX HLSL
- Syntaxe, opérateurs, fonctions C/C++
- Présence de conditionnelles
- Accède au fonctionnalités spécifiques des cartes 3D
  - Opérations matricielles et vectorielles
  - Les registres qui vont bien pour une performance max
  - Accès au fonctions du GPU: mul, sqrt, dot, …
  - Fonctions mathématiques pour le graphisme

La chaîne de production de Cg



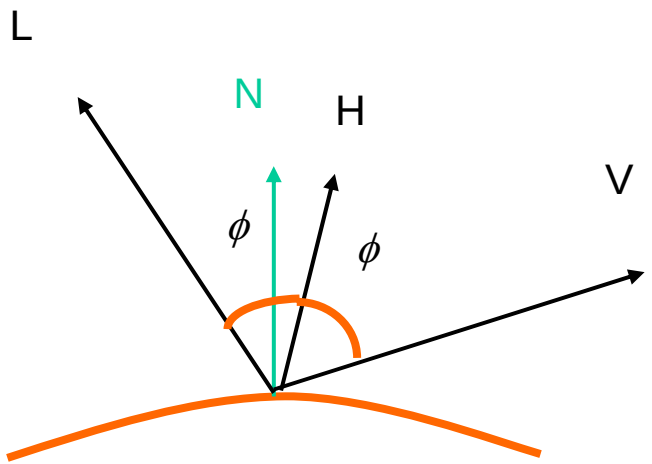
Compilation et optimisation

Assembleur bas-niveau

Langage machine

Un shader pour le Phong (méthode de Blinn)

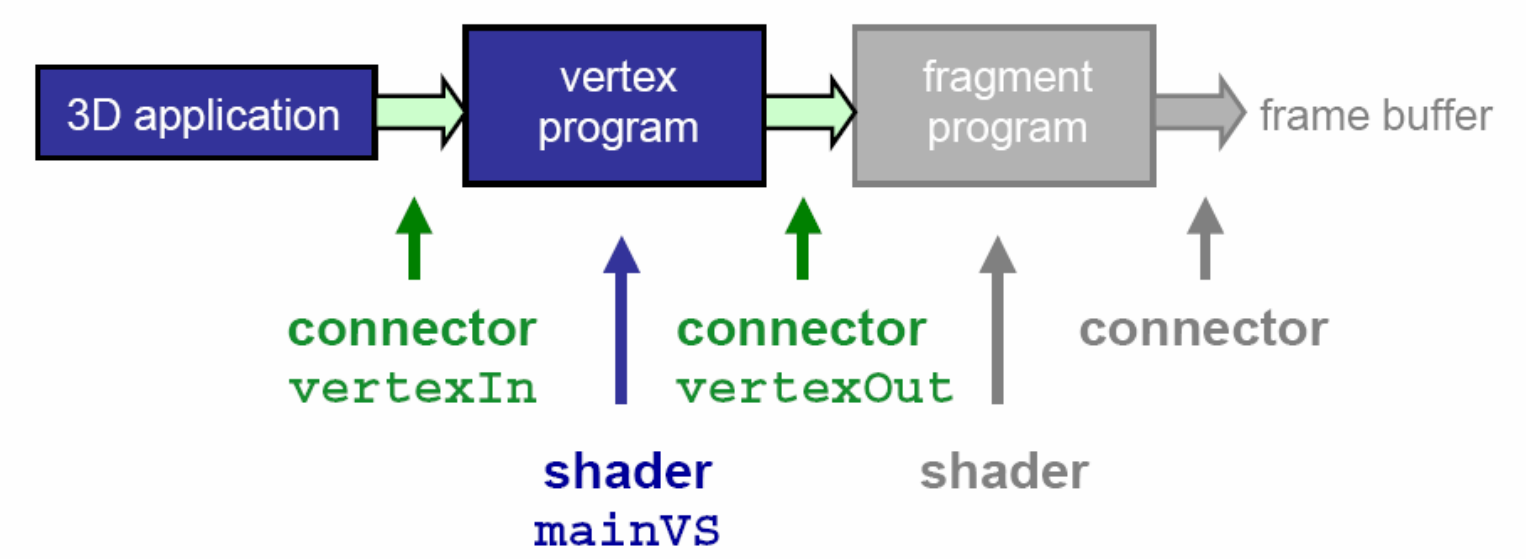
- Variation populaire du coloriage de type phong
- Utilise le vecteur *halfway H*.
- $I_s = k_s I_{incident} (N \cdot H)^n$.
  - H = L+V / | L+V |



- Rapide à calculer
- Dépend de la vue puisque H dépend de V

Phong Shading en Cg: Vertex Shader

- Première partie du pipeline
- Connecteurs : quelles données sont transmises et émises au vertex shader ?



Phong Shading en Cg: Connecteurs

- Décrivent les entrées/sorties
- Les données varies
- Spécifiées dans des structures
- Extensible
- Seuls les données importantes (utilisées dans le programme) sont transmises
- Registres prédéfinis: **POSITION, NORMAL,** ...

Phong Shading en Cg: Connecteurs

```
// données de l'application vers le vertex shader
struct vertexIn {
  float3 Position : POSITION; // registres prédéfinis
  float4 UV : TEXCOORD0;
  float4 Normal : NORMAL;
};
// du vertex shader vers le pixel shader
struct vertexOut {
  float4 HPosition : POSITION;
  float4 TexCoord : TEXCOORD0;
  float3 LightVec : TEXCOORD1;
  float3 WorldNormal : TEXCOORD2;
  float3 WorldPos : TEXCOORD3;
  float3 WorldView : TEXCOORD4;
};
```

Phong Shading en Cg: Vertex Shader

- Un vertex shader est une fonction nécessitant
  - des paramètes d'entrée fournis par l'application
  - une structure de sortie pour le pixel shader
- Un vertex shader pour le coloriage Phong calcule dans le repère global
  - la position,
  - le vecteur normal,
  - le vecteur de vue,
  - et le vecteur vers la source lumineuse

Phong Shading en Cg: Vertex Shader

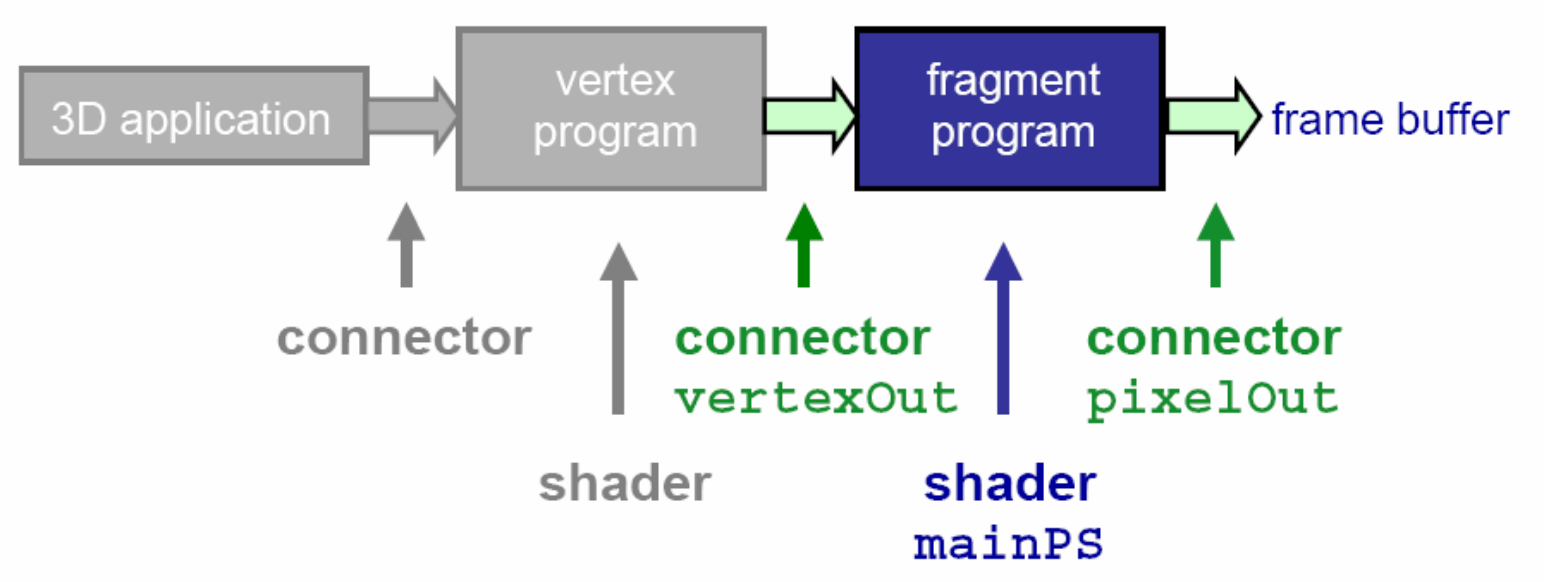
```
// vertex shader
vertexOut mainVS(vertexIn IN,  // le sommet et
uniform float4x4 WorldViewProj,// les parameteres fournis
uniform float4x4 WorldIT,      // par l'appli: +sieurs
uniform float4x4 World,        // matrices
uniform float4x4 ViewIT,       // de transformation et
uniform float3 LightPos        // une source lumineuse
)
{
vertexOut OUT;                 // sortie du vertex shader
```

Phong Shading en Cg: Vertex Shader

```
OUT.WorldNormal = mul(WorldIT, IN.Normal).xyz;
// position dans le repère de l'objet
float4 Po = float4(IN.Position.x,IN.Position.y,
IN.Position.z,1.0);
float3 Pw = mul(World, Po).xyz; // pos repère global
OUT.WorldPos = Pw; // pos in world coords
OUT.LightVec = LightPos - Pw;   // vecteur lumière
OUT.TexCoord = IN.UV; // original tex coords
// vecteur de vue repère global
OUT.WorldView = normalize(ViewIT[3].xyz - Pw);
// position dans le repère de l'observateur
OUT.HPosition = mul(WorldViewProj, Po);
return OUT; // sortie du vertex shader
}
```

Phong Shading in Cg: Pixel Shader

- Deuxième partie du traitement
- Connecteurs: du vertex au pixel shader, du pixel shader au frame buffer



Phong Shading in Cg: Pixel Shader

- Le pixel shader est une fonction qui nécessite
  - Des paramètres en entrée pour calculer la couleur du pixel
  - Une structure de sortie exploitable par le *frame buffer*
- Ce que va faire notre pixel shader pour le coloriage Phong
  - Des vecteur normalisés de lumière, de vue et de normale au point considéré normalisés
  - Calcule le vecteur *Halfway* H
  - Fait la somme des contributions des composantes spéculaire, diffuse, et ambiante

Phong Shading in Cg: Pixel Shader

```
// sortie = couleur pour le frame buffer
struct pixelOut {
float4 col : COLOR;
};
// pixel shader
pixelOut mainPS(vertexOut IN, // données du vertex shader
uniform float SpecExpon,      // paramètres fournis
uniform float4 AmbiColor,     // par l'application
uniform float4 SurfColor,
uniform float4 LightColor
) {
pixelOut OUT;                 // sortie du pixel shader
float3 Ln = normalize(IN.LightVec);
float3 Nn = normalize(IN.WorldNormal);
float3 Vn = normalize(IN.WorldView);
float3 Hn = normalize(Vn + Ln);
```

Phong Shading in Cg: Pixel Shader

```
// produit scalaire entre les vecteurs lumière et normale
float ldn = dot(Ln,Nn);
// produit scalaire entre les vecteurs halfway et normale
float hdn = dot(Hn,Nn);
// la fonction “lit” calcule les composantes
// diffuse and specular
float4 litV = lit(ldn,hdn,SpecExpon);
float4 diffContrib =
SurfColor * ( litV.y * LightColor + AmbiColor);
float4 specContrib = litV.y*litV.z * LightColor;
// somme des différentes contributions lumineuses
float4 result = diffContrib + specContrib;
OUT.col = result;
return OUT; // couleur du pixel en sortie
}
```

Attachement au programme principal

- Pour les librairies Cg :
  - `#include <Cg/cg.h>`
  - `#include <Cg/cgGL.h>` ou `#include <Cg/cgD3D9.h>`
- Via des fonctions du type :
  - `cgCreateContext`
  - `cgCreateProgram`
  - `cgGLLoadProgram` ou `cgD3D9LoadProgram`
  - `cgGLSetParameter4fv` ou `cgD3D9SetUniform`
  - `cgGLBindProgram` ou `cgD3D9BindProgram`
  - …

Et dans le futur …

- Encore et encore des shaders …
- Et des nouvelles techniques de rendu réalistes
- Raytracing temps-réel + renderman-like shading ?
- Encore besoin d'un CPU ?
- Utilisation pour des taches spécifiques
  - Visualisation
  - Mette en valeur les données volumétriques
  - Mais aussi pour le filtrage, mapping, et rendu