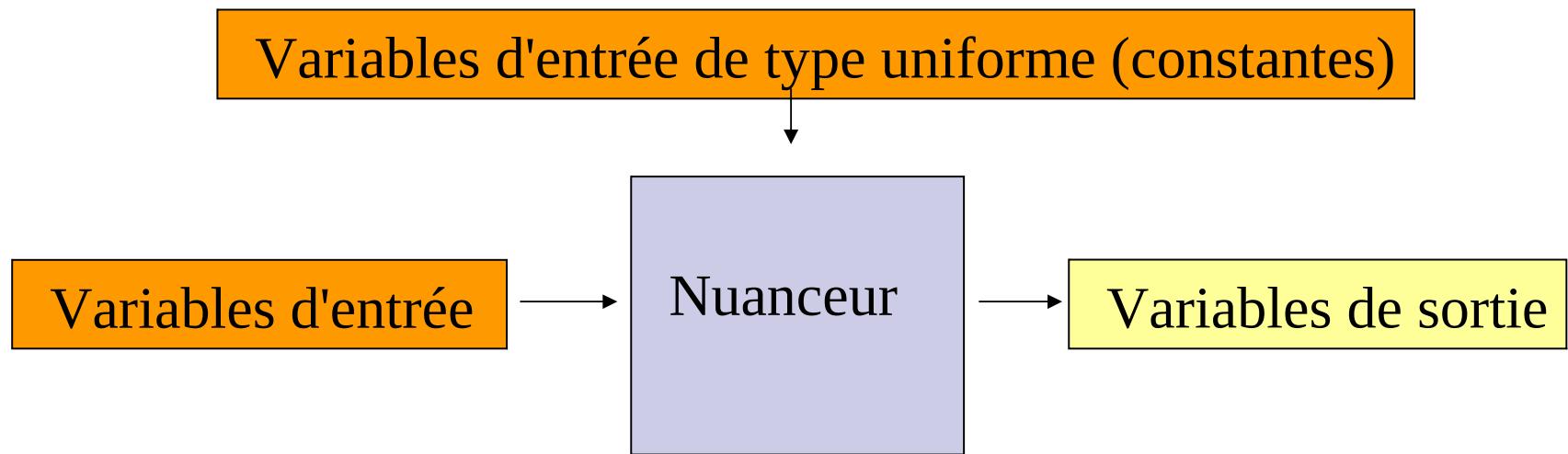


Utilisation de GLSL

L'utilisation des nuanceurs

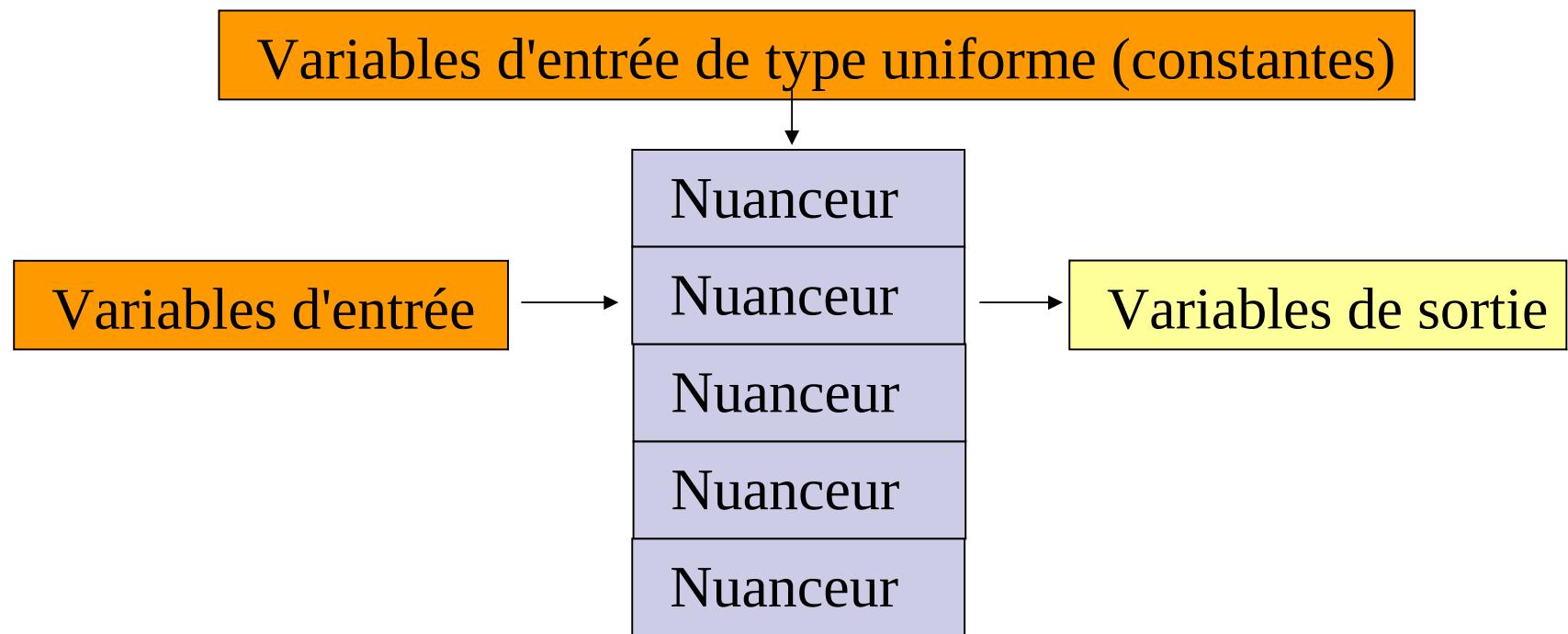
- Pour chaque nuanceur :



- Exemples de type variable : sommets, fragments, ...
- Exemples de type uniforme : matrices, textures, temps, ...

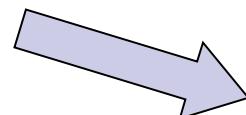
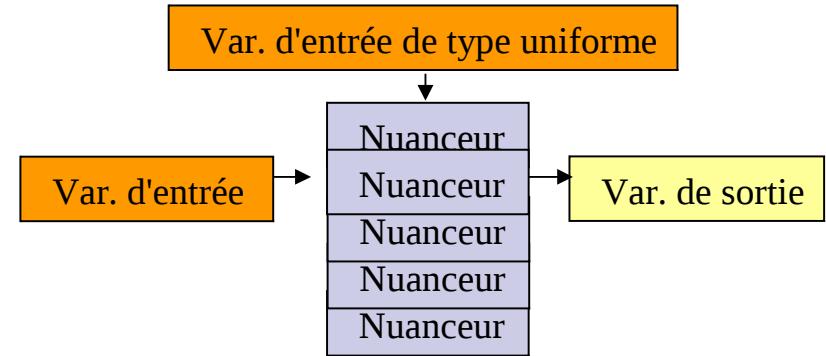
L'utilisation des nuanceurs

- Les nuanceurs fonctionnent en parallèle sur le GPU



L'utilisation des nuanceurs

- Chaque nuanceur
 - Partage les mêmes variables uniformes (en lecture)
 - A des variables différentes en entrée
 - Écrit sa propre sortie
 - N'a pas d'effet de bord
 - S'exécute indépendamment sans communiquer avec les autres nuanceurs ...



... en GLSL. Avec CUDA/OpenCL, les *kernels (shaders)* peuvent se synchroniser

L'utilisation des nuanceurs

- Les nuanceurs s'exécute en mode **SIMT**
 - Single Instruction Multiple Thread
- Chaque fil (*thread*) exécute les mêmes instructions, mais sur des données différentes
- Le parallélisme est implicite
 - Un appel à une primitive invoque un processeur parallèle : le GPU
 - Le pilote et la carte graphique s'occupent de l'ordonnancement et de la synchronisation
- Les programmeurs écrivent des applications parallèles sans le savoir!

Les nuanceurs dans le pipeline

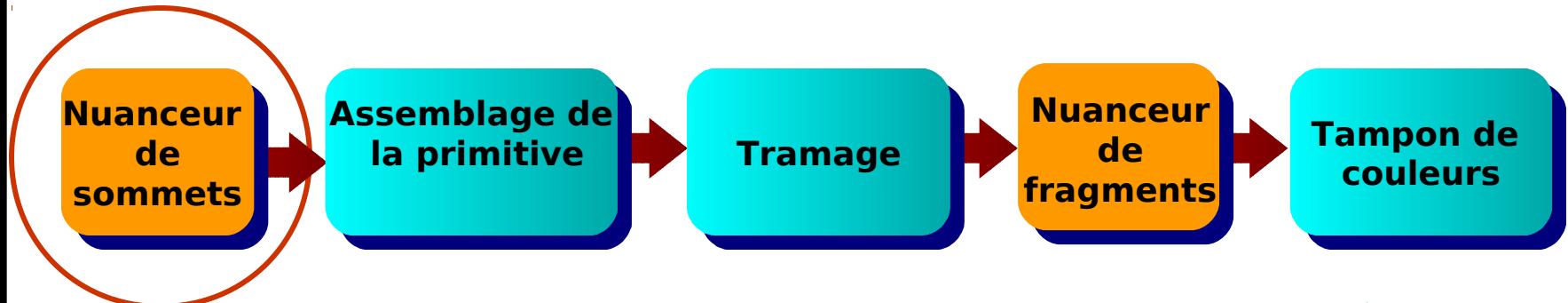
- Le pipeline standard est (en version simplifiée) :



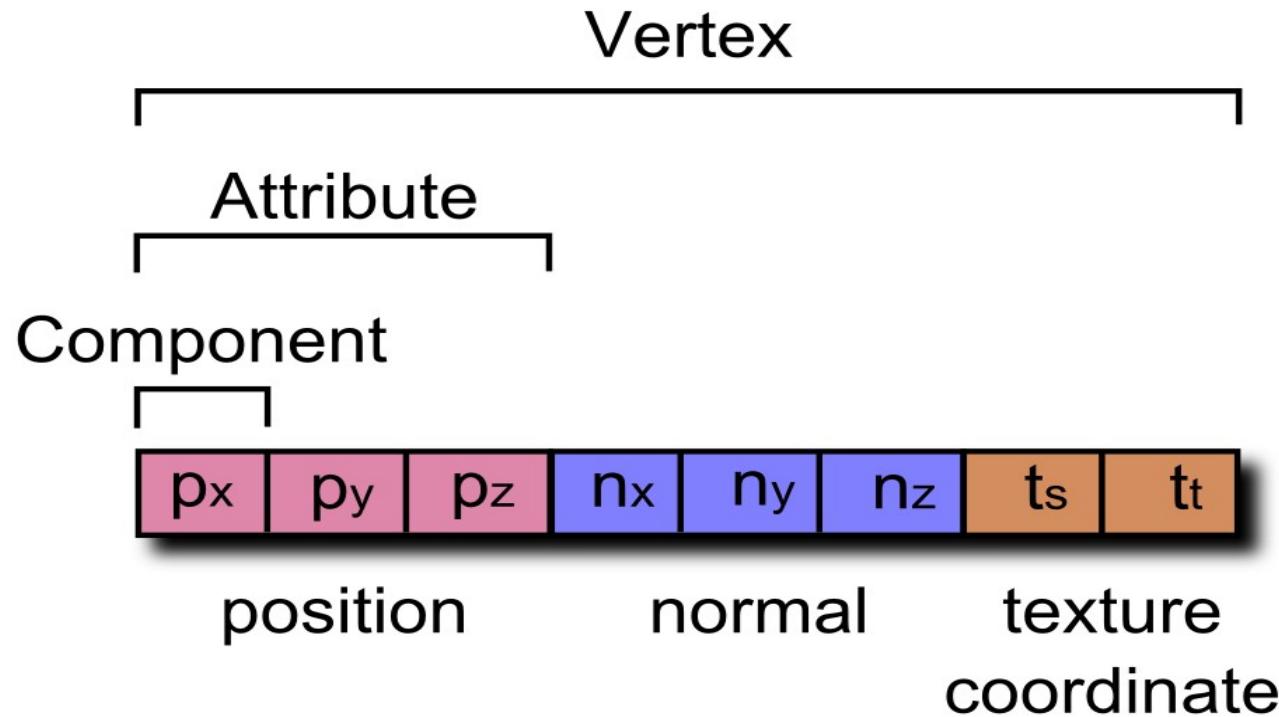
- On utilise généralement :
 - Un nuageur de sommets et
 - Un nuageur de fragments
- D'autres nuageurs existent aussi :
 - Un nuageur de géométrie
 - Un nuageur de *tessellation* (découpage)

Le nuanceur de sommets

- Entrée
 - Un sommet avec sa position en coordonnées de monde
 - Les variables uniformes
- Sortie
 - Un sommet avec sa position en coordonnées normalisées

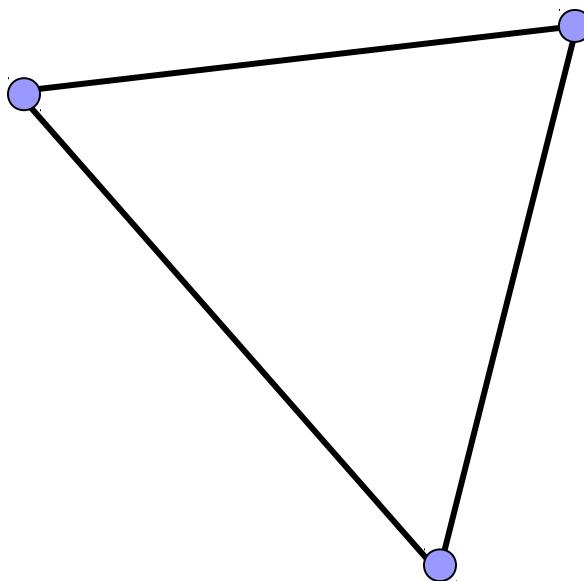


Le nuanceur de sommets



(A K Peters, Ltd. www.virtualglobebook.com)

Le nuanceur de sommets



- Ce triangle est composé de trois *sommets*.
- Chaque *sommet* possède le même nombre et le même type d'*attributs*.
- Chaque sommet possède habituellement des valeurs différentes pour ses attributs, p.e. sa position.

(Même si les termes sont quelquefois utilisés de façon interchangeable, une position n'est pas un sommet et un sommet n'est pas une position.)

Le nuanceur de sommets

- Un nuanceur de sommets simple (version 1.1) :

```
// #version 110

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

Le nuanceur de sommets

- Un nuanceur de sommets simple (version 3.3) :

```
#version 330

uniform mat4 u_ModelView;
in vec3 Position;

void main(void)
{
    gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

Le nuanceur de sommets

- Un nuanceur de sommets simple :

```
#version 330

uniform mat4 u_ModelView;
in vec3 Position;

void main(void)
{
    gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

gl_Position est la valeur de sortie standard du nuanceur de sommets.
Vous **devez** y affecter une valeur.

La même matrice de transformations est utilisée pour chaque sommet d'une même primitive

Chaque nuanceur de sommets s'exécute dans un fil différent avec une valeur différente de **Position**.

Une matrice 4x4 fois un vecteur 4x1 transforme les coordonnées du modèle en coordonnées normalisées

Le nuanceur de sommets

- Un nuanceur de sommets simple avec deux attributs :

```
#version 330

uniform mat4 u_ModelView;
in vec3 Position;
in vec3 Color;
out vec3 fs_Color;

void main(void)
{
    fs_Color = Color;
    gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

Le nuanceur de sommets

- Un nuanceur de sommets simple avec deux attributs :

```
#version 330

uniform mat4 u_ModelView;
in vec3 Position;
in vec3 Color;
out vec3 fs_Color;

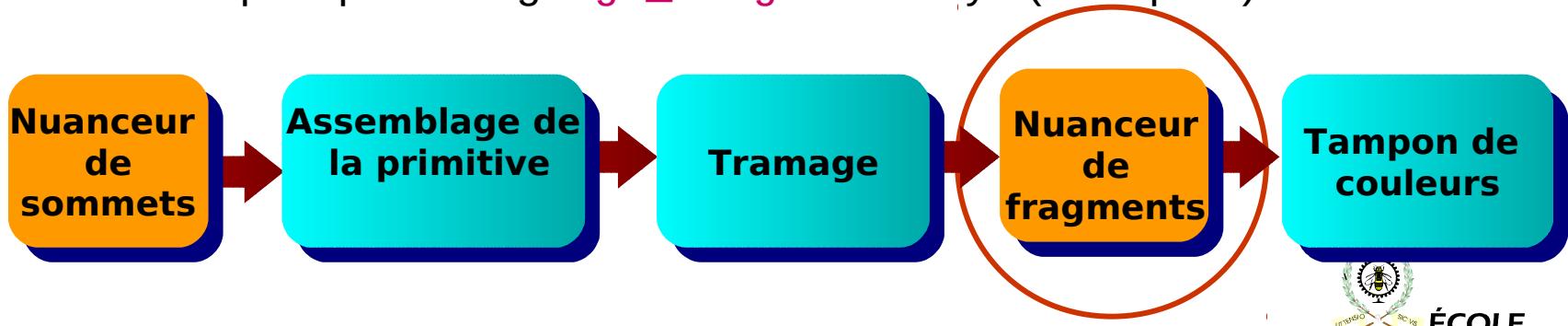
void main(void)
{
    fs_Color = Color;
    gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

Chaque nuanceur de sommets s'exécute dans un fil distinct avec des valeurs différentes de **Position** et **Color**.

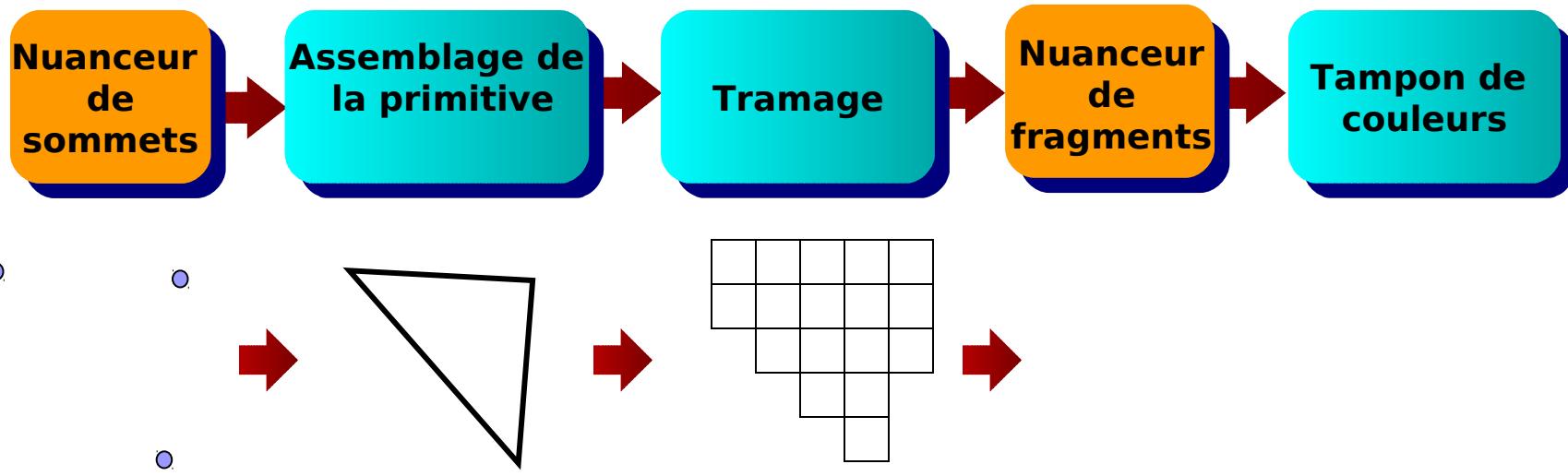
Ce nuanceur produit une couleur **vec3** en plus d'une **gl_Position**.

Le nuanceur de fragments

- Entrée
 - Position du fragment en coordonnées d'écran : `gl_FragCoord.xy`
 - La profondeur du fragment : `gl_FragCoord.z`
 - Les valeurs interpolées de la sortie du nuanceur de sommets
 - Les variables uniformes
- Sortie
 - La couleur du fragment
 - Optionnel : la profondeur du fragment : `gl_FragDepth`. (Pourquoi?)
 - Optionnel : des couleurs multiples provenant de textures
 - `discard` permet d'arrêter le traitement du fragment. (Pourquoi?)
 - On ne peut pas changer `gl_FragCoord.xy`. (Pourquoi?)



L'entrée du nuanceur de fragments



- Le tramage convertit les primitives en fragments qui constituent l'entrée du nuanceur de fragments
- Les variables en sortie du nuanceur de sommets sont *interpolées* pour chaque primitive

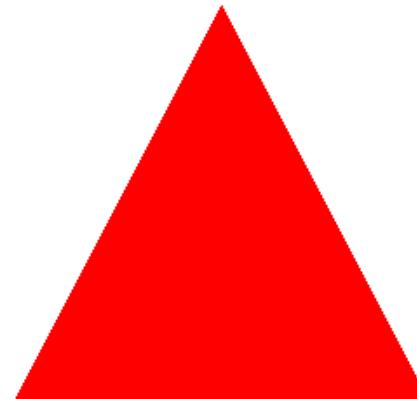
Le nuanceur de fragments

- Un nuanceur de fragments simple :

```
#version 330

out vec3 out_Color;

void main(void)
{
    out_Color = vec3(1.0, 0.0, 0.0);
}
```



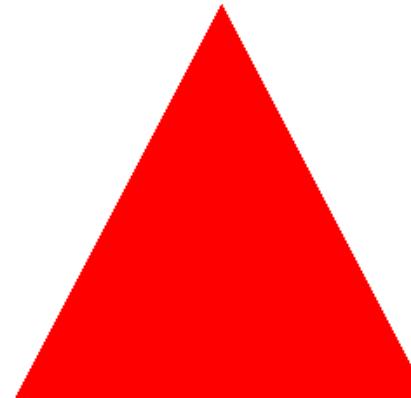
Le nuanceur de fragments

- Un nuanceur de fragments simple :

```
#version 330  
  
out vec3 out_Color;  
  
void main(void)  
{  
    out_Color = vec3(1.0, 0.0, 0.0);  
}
```

Chaque nuanceur de fragments s'exécute dans un fil distinct et produit une couleur pour un fragment différent.
(Pourquoi **vec3** et pas **vec4** ?)

Couleur rouge uniforme



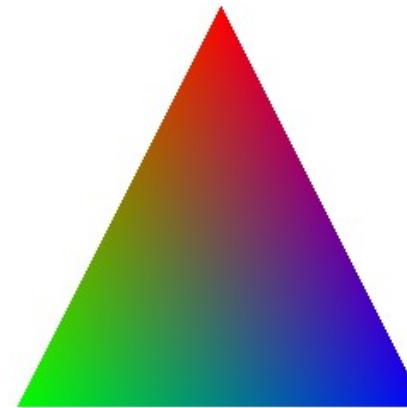
Le nuanceur de fragments

- Un nuanceur de fragments un peu moins simple :

```
#version 330

in vec3 fs_Color;
out vec3 out_Color;

void main(void)
{
    out_Color = fs_Color;
}
```



Le nuanceur de fragments

- Un nuanceur de fragments un peu moins simple :

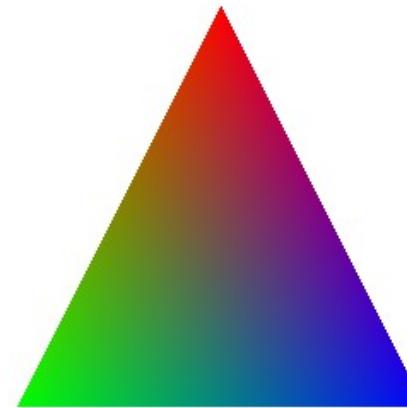
```
#version 330

in vec3 fs_Color;
out vec3 out_Color;

void main(void)
{
    out_Color = fs_Color;
}
```

Entrée du nuanceur de fragments provenant de la sortie du nuanceur de sommets

La couleur produite



Syntaxe GLSL

- GLSL est comme du C sans
 - pointeurs
 - récursion
 - Allocation dynamique de mémoire
- GLSL est comme du C avec
 - Des types standards : vector, matrix, sampler
 - Constructeurs
 - Une bonne librairie mathématique
 - Des qualificatifs 'input' et 'output'

Ces caractéristiques permettent d'écrire des nuanceurs courts et efficaces.

Syntaxe GLSL

- GLSL possède un préprocesseur

```
#version 330

#ifndef METHODE_VITE_APPROX
    ViteApproximation();
#else
    LenteExacte();
#endif

// ... beaucoup d'autres
```

- Tous les nuanceurs ont un `main()`

```
void main(void)
{
}
```

Syntaxe GLSL

- Types scalaires: `float, int, uint, bool`
- Vecteurs:
 - `vec2, vec3, vec4 (float)`
 - Also `ivec*, uvec*, bvec* (int, uint, bool)`
- L'accès aux composantes est varié:
 - `.x, .y, .z, .w` ← Position ou direction
 - `.r, .g, .b, .a` ← Couleur
 - `.s, .t, .p, .q` ← Coordonnées de texture

Syntaxe GLSL

- Les vecteurs ont des constructeurs

```
vec3 xyz = vec3(1.0, 2.0, 3.0);  
  
vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]  
  
vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
```

Syntaxe GLSL : swizzle

- **Swizzle**: sélection ou réarrangement des composantes

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;    // [0.5, 1.0, 0.8]
vec3 bgr = c.bgr;    // [0.8, 1.0, 0.5]
vec3 rrr = c.rrr;    // [0.5, 0.5, 0.5]

c.a = 0.5;           // [0.5, 1.0, 0.8, 0.5]
c.rg = 0.0;           // [0.0, 1.0, 0.0, 0.5]

float g = rg[1];    // 0.5, indice, pas un swizzling
```

- *L'essayer, c'est l'adopter!*

Syntaxe GLSL: matrices

- Les matrices sont standards:
 - carrées: `mat2`, `mat3`, `mat4`
 - rectangulaire: `mat m x n` . m colonnes, n rangées
- Valeurs stockées par colonnes.

Syntaxe GLSL

- Constructeurs de matrices

```
mat3 i = mat3(1.0); // 3x3 matrice identité  
  
mat2 m = mat2(1.0, 2.0,  // [1.0 3.0]  
              3.0, 4.0); // [2.0 4.0]
```

- Accès aux éléments

```
float f = m[column][row];  
  
float x = m[0].x; // composante x de la colonne 1  
  
vec2 yz = m[1].yz; // composantes yz de la seconde colonne
```

Syntaxe GLSL

- Les opérations matricielles et vectorielles sont rapides et faciles :

```
vec3 xyz = // ...
```

```
vec3 v0 = 2.0 * xyz; // mise à l'échelle  
vec3 v1 = v0 + xyz; // par composante  
vec3 v2 = v0 * xyz; // par composante
```

```
mat3 m = // ...  
mat3 v = // ...
```

```
mat3 mv = v * m; // matrice * matrice  
mat3 xyz2 = mv * xyz; // matrice * vecteur  
mat3 xyz3 = xyz * mv; // vecteur * matrice
```

Syntaxe GLSL : int / out / uniform

```
#version 330

uniform mat4 u_ModelView;
in vec3 Position;
in vec3 Color;
out vec3 fs_Color;

void main(void)
{
    fs_Color = Color;
    gl_Position = u_ModelView * vec4(Position, 1.0);
}
```

uniform: entrée constante

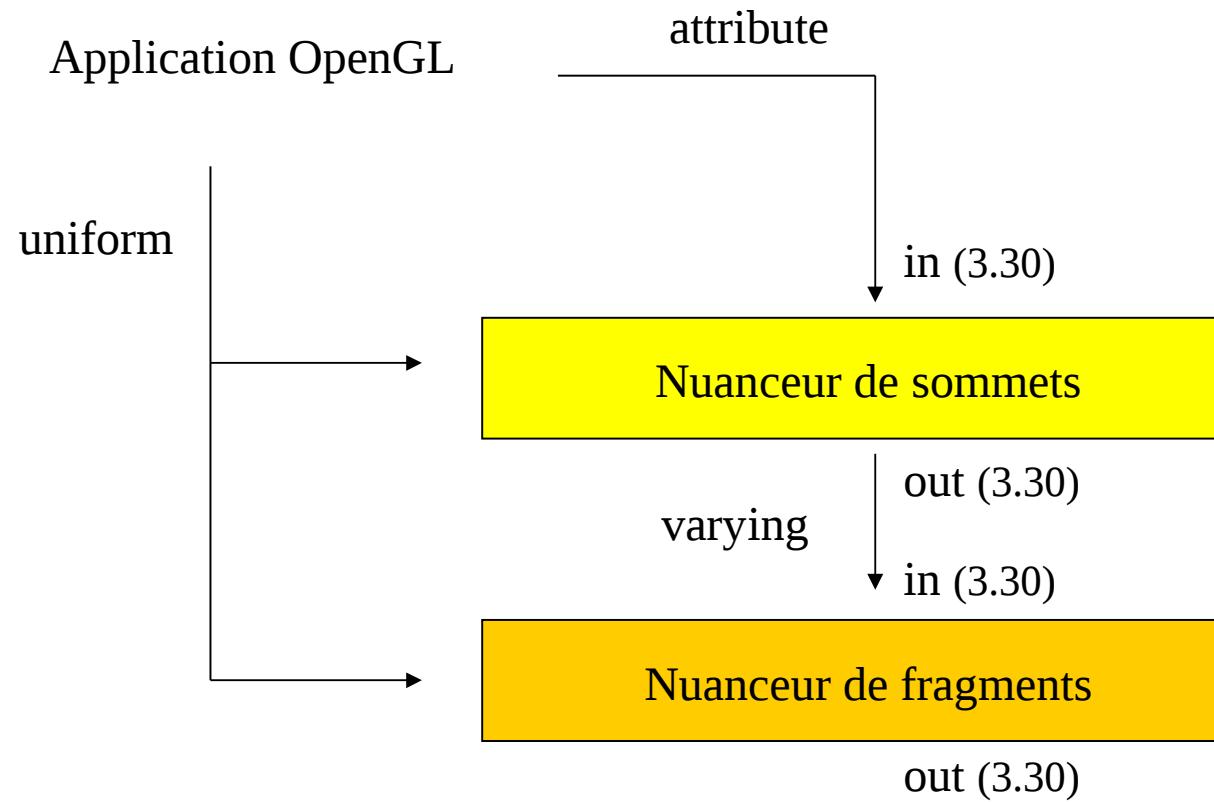
in: entrée variable pour chaque sommet

out: la sortie du nuanceur

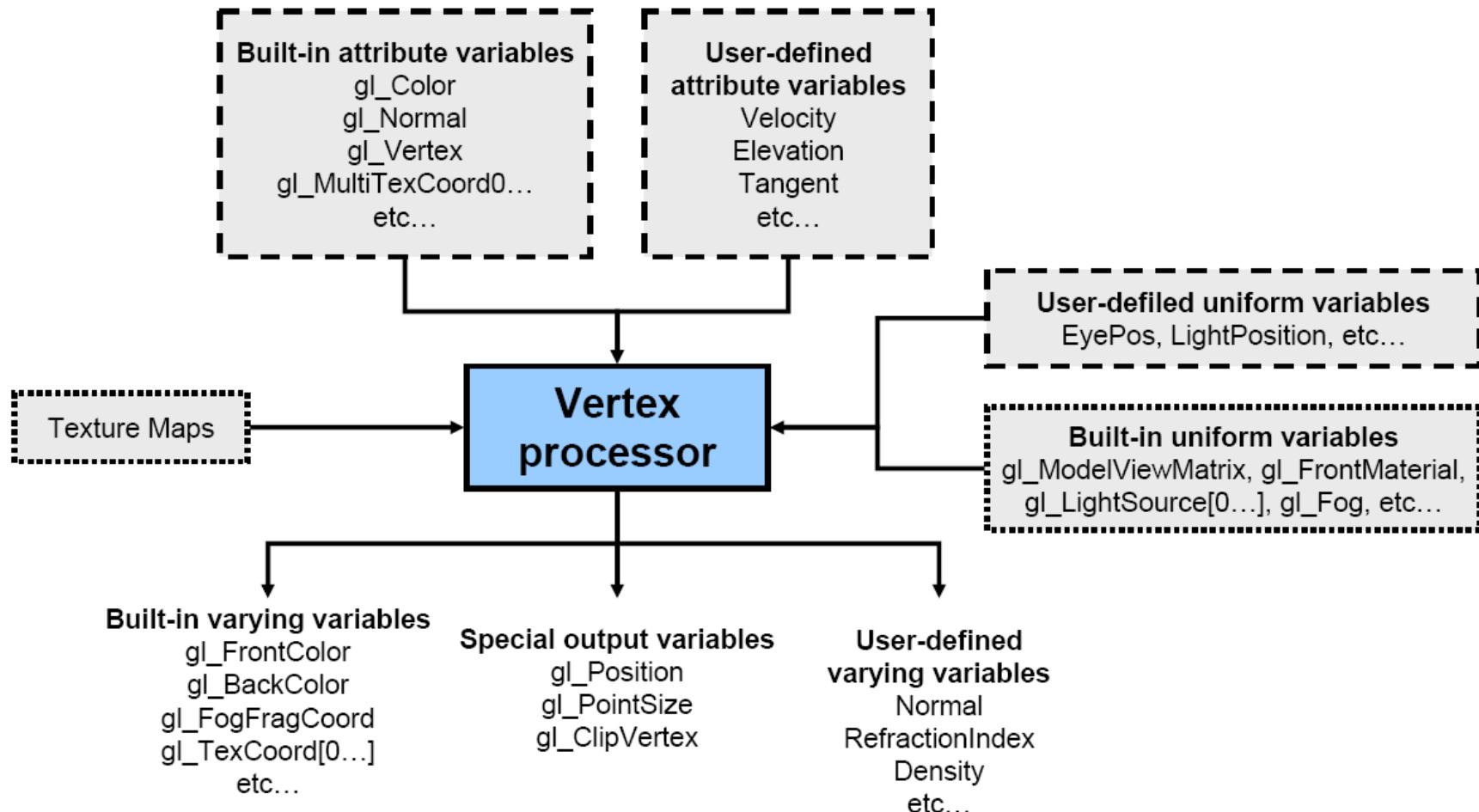
Syntaxe GLSL : types

- Le type sert à définir l'usage des variables globales:
 - *uniform*
 - Communique les variables de l'application vers n'importe quel nuanceur
 - *attribute*
 - Communique les variables de l'application vers le nuanceur de sommets
 - *varying*
 - Communique les variables interpolées du nuanceur de sommets vers le nuanceur de fragments
- Avec glsl 3.3, on utilise plutôt *in* et *out* pour l'entrée et la sortie de chaque nuanceur.

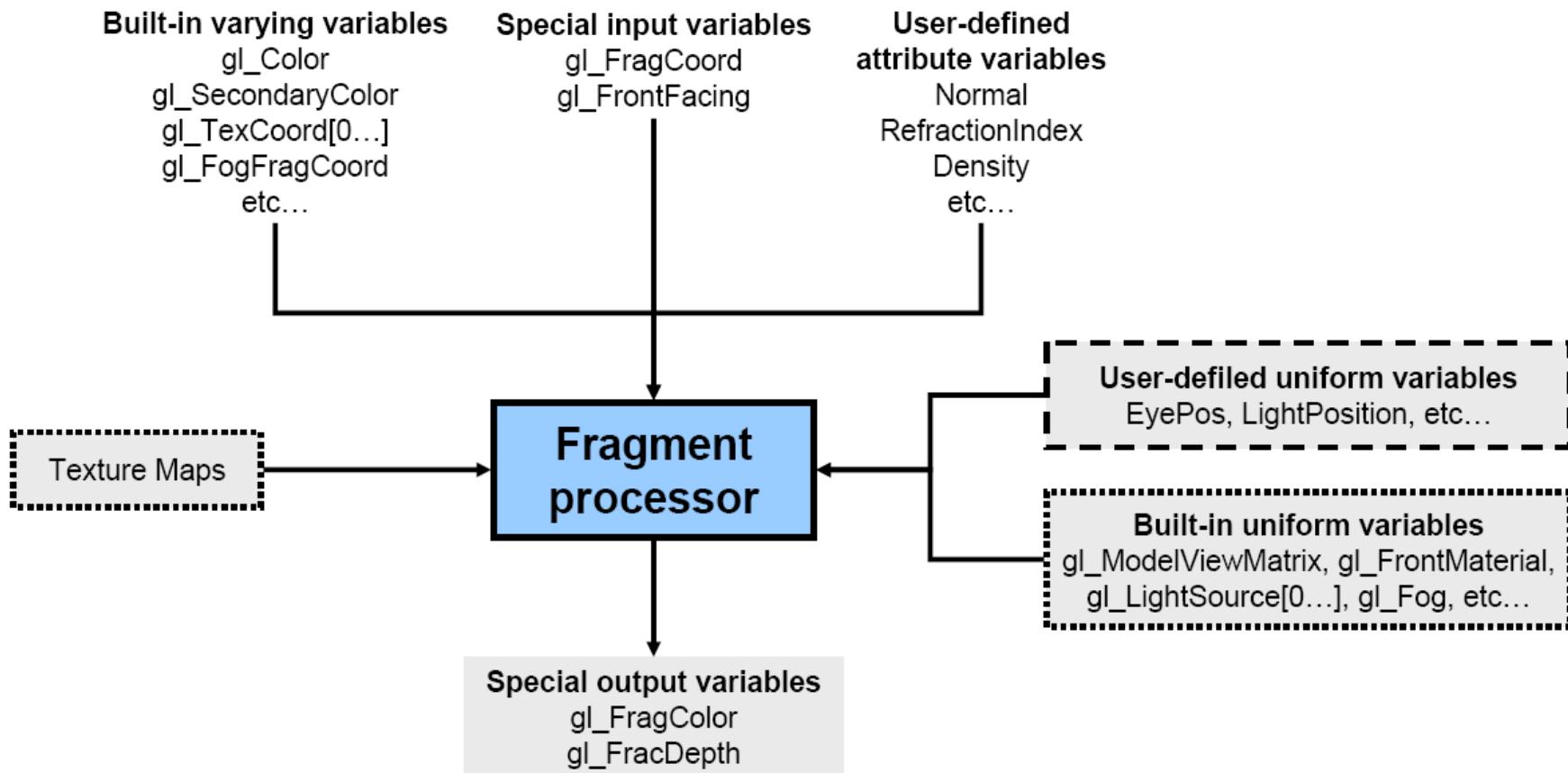
Syntaxe GLSL : types



Le nuanceur de sommets



Le nuanceur de fragments



Syntaxe GLSL : accès aux textures

- Il existe des types *opaques* pour accéder aux textures

```
uniform sampler2D colorMap; // texture 2D (doit être uniform!)  
  
vec4 couleur = texture(colorMap, gl_TexCoord[0].xy );  
  
vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;  
  
vec3 colorAbove = textureOffset(colorMap,  
                                 vec2(0.5, 0.5), ivec2(0, 1)).rgb;  
  
vec2 size = textureSize(colorMap, 0);  
  
// Plusieurs types de sampler: sampler1D,  
// sampler3D, sampler2DRect, samplerCube,  
// isampler*, usampler*, ...  
  
// Plusieurs fonctions avec les samplers: texelFetch, textureLod, ...
```

Syntaxe GLSL : accès aux textures

- Il existe des types *opaques* pour accéder aux textures

```
uniform sampler2D colorMap; // texture 2D (doit être uniform!)
```

```
vec4 couleur = texture(colorMap, gl_TexCoord[0].xy );
```

```
vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;
```

```
vec3 colorAbove = textureOffset(colorMap,  
vec2(0.5, 0.5), ivec2(0, 1)).rgb;
```

```
vec2 size = textureSize(colorMap, 0);
```

texture() retourne un vec4;
on extrait la composante dont on
a besoin

Les textures 2D utilisent des
coordonnées de texture 2D

Syntaxe GLSL : accès aux textures

- Il existe des types *opaques* pour accéder aux textures

```
uniform sampler2D colorMap; // texture 2D (doit être uniform!)
```

```
vec4 couleur = texture(colorMap, gl_TexCoord[0].xy );
```

```
vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;
```

```
vec3 colorAbove = textureOffset(colorMap,  
vec2(0.5, 0.5), ivec2(0, 1)).rgb;
```

```
vec2 size = textureSize(colorMap, 0);
```

Coordonnées
de texture 2D

Décalage entier 2D

L'accès aléatoire est
appelé *gather*. (Plus
à ce sujet plus tard.)

Syntaxe GLSL : accès aux textures

- Il existe des types *opaques* pour accéder aux textures

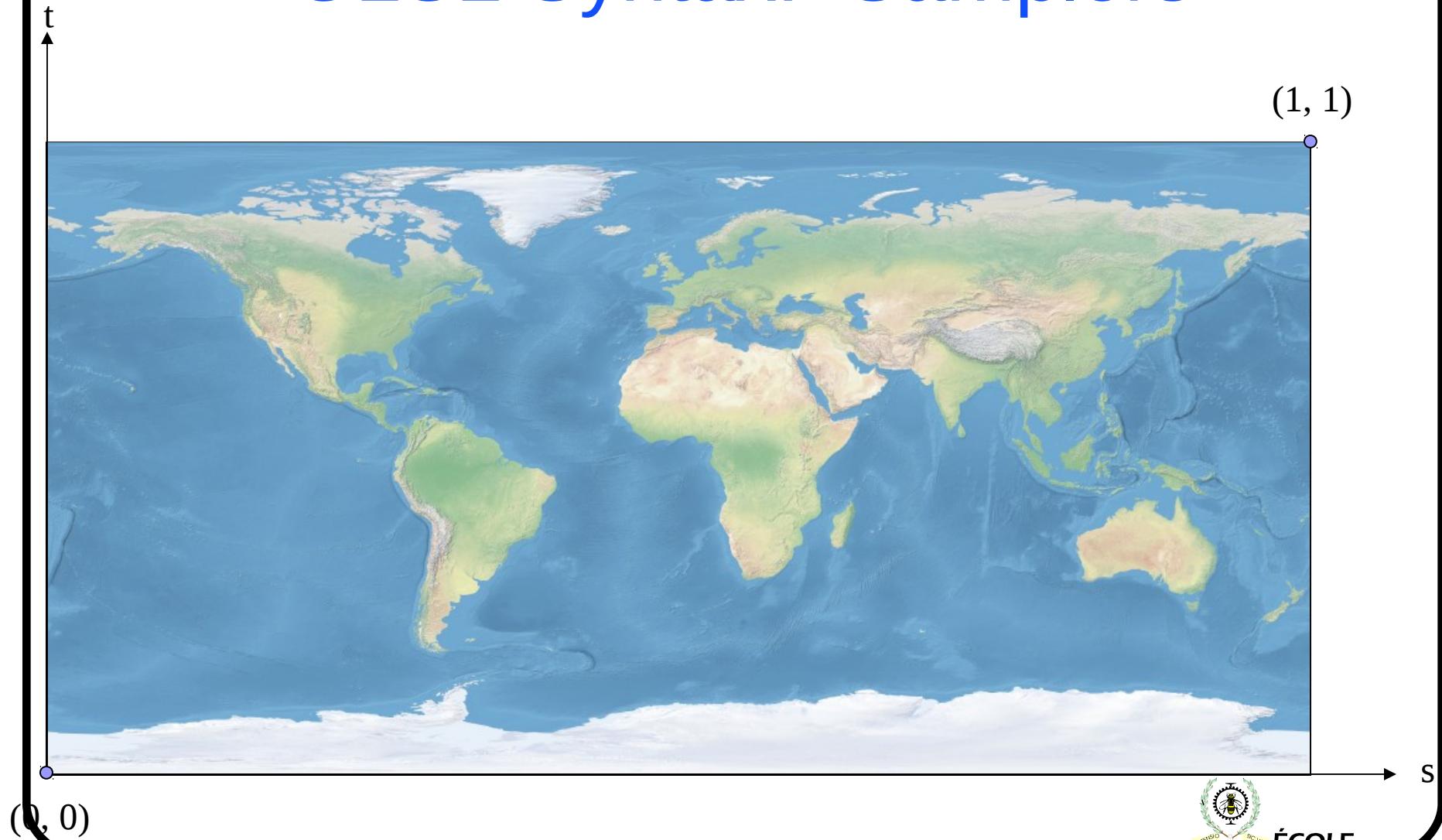
```
uniform sampler2D colorMap; // texture 2D (doit être uniform!)  
  
vec4 couleur = texture(colorMap, gl_TexCoord[0].xy );  
  
vec3 color = texture(colorMap, vec2(0.5, 0.5)).rgb;  
  
vec3 colorAbove = textureOffset(colorMap,  
                                 vec2(0.5, 0.5), ivec2(0, 1)).rgb;  
  
vec2 size = textureSize(colorMap, 0);
```

Taille du niveau 0

Syntaxe GLSL : accès aux textures

- Les textures ...
 - En 2D, sont comme un tableau à deux dimensions
 - Habituellement, mais pas toujours :
 - Les textures sont carrées, p.e. 256x256
 - Les dimensions sont des puissances de deux
 - Les coordonnées sont souvent normalisées, c'est-à-dire entre [0, 1]
 - Un *texel* (*texture element*) d'une texture est l'équivalent d'un *pixel* (*picture element*) d'une image
 - La fonction *texture()* applique un filtre en utilisant une fonctionnalité fixe du matériel graphique
- Les textures ont été la base du *GPGPU* [GPGPU.org]
(General Programming on GPU)

GLSL Syntax: Samplers



Fonctions standards GLSL

- Fonctions trigonométriques

```
float s = sin(theta);
float c = cos(theta);
float t = tan(theta);

float as = asin(theta);
// ...

vec3 angles = vec3(/* ... */);
vec3 vs = sin(angles);
```

Fonctions standards GLSL

- Fonctions exponentielles

```
float xToTheY = pow(x, y);
float eToTheX = exp(x);
float twoToTheX = exp2(x);

float l = log(x);    // ln
float l2 = log2(x); // log2

float s = sqrt(x);
float is = inversesqrt(x); // 1/sqrt()
```

Fonctions standards GLSL

- Fonctions diverses

```
float ax = abs(x); // valeur absolue
float sx = sign(x); // -1.0, 0.0, 1.0

float m0 = min(x, y); // valeur minimum
float m1 = max(x, y); // valeur maximum
float c = clamp(x, 0.0, 1.0);

// d'autres: floor(), ceil(),
// step(), smoothstep(), ...
```

Fonctions standards GLSL

- Un seul appel :

```
float minimum = // ...
float maximum = // ...
float x = // ...

float f = min(max(x, minimum), maximum);
```

Fonctions standards GLSL

- Exercice : ré-écrire en un seul appel :

```
float x = // ...
float f;

if (x > 0.0)
{
    f = 2.0;
}
else
{
    f = -2.0;
}
```

Indice : sign()

Fonctions standards GLSL

- Exercice : ré-écrire sans le `if` :

```
float root1 = // ...
float root2 = // ...

if (root1 < root2)
{
    return vec3(0.0, 0.0, root1);
}
else
{
    return vec3(0.0, 0.0, root2);
}
```

Indice : `min()`

Fonctions standards GLSL

- Exercice : ré-écrire sans le `if` :

```
bool b = // ...
vec3 color;

if (b)
{
    color = vec3(1.0, 0.0, 0.0);
}
else
{
    color = vec3(0.0, 1.0, 0.0);
}
```

Indice : aucun appel à des fonctions glsl

Fonctions standards GLSL

- Quelques fonctions géométriques :

```
vec3 l = // ...
vec3 n = // ...
vec3 p = // ...
vec3 q = // ...

float f = length(l);           // longeur du vecteur
float d = distance(p, q);    // distance entre deux points

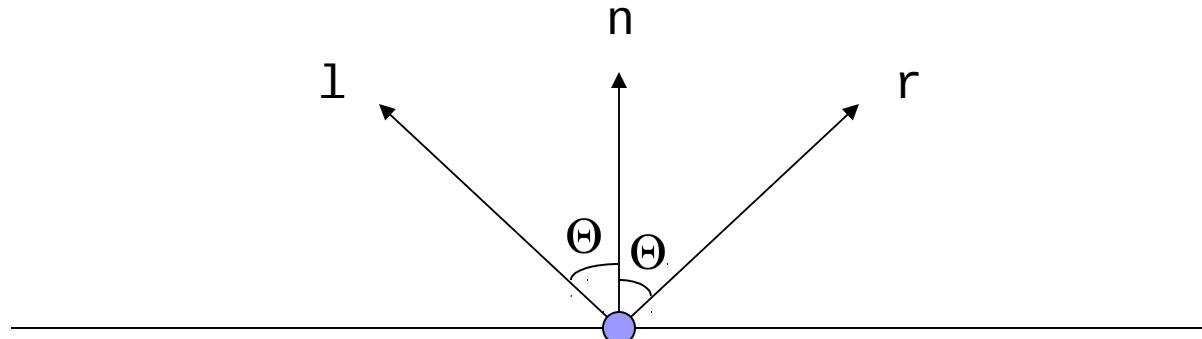
float d2 = dot(l, n);         // produit scalaire
vec3 v2 = cross(l, n);       // produit vectoriel
vec3 v3 = normalize(l);      // normalisation

vec3 v3 = reflect(l, n);     // réflexion

// aussi: faceforward() et refract()
```

Fonctions standards GLSL

- **reflect(-l, n)**
 - Avec l et n, trouve r qui est une réflexion selon n :



Fonctions standards GLSL

- Fonctions matricielles

```
mat4 m = // ...  
  
mat4 t = transpose(m);  
float d = determinant(m);  
mat4 d = inverse(m);
```

Fonctions standards GLSL

- Quelques fonctions relationnelles

```
vec3 p = vec3(1.0, 2.0, 3.0);
vec3 q = vec3(3.0, 2.0, 1.0);

bvec3 b = equal(p, q);           // (false, true, false)
bvec3 b2 = lessThan(p, q);      // (true, false, false)
bvec3 b3 = greaterThan(p, q);   // (false, false, true)

bvec3 b4 = any(b);              // true
bvec3 b5 = all(b);              // false
```

Fonctions standards GLSL

- Exercice : ré-écrire en une seule ligne

```
bool foo(vec3 p, vec3 q)
{
    if (p.x < q.x)
    {
        return true;
    }
    else if (p.y < q.y)
    {
        return true;
    }
    else if (p.z < q.z)
    {
        return true;
    }
    return false;
}
```

Syntaxe et fonctions GLSL

- Nous n'avons pas parlé de:
 - Tableaux
 - Structs
 - Appels à des fonctions
 - Du qualificatif **const**
 - Du énoncés **if** / **while** / **for**
 - **dFdX**, **dFdy**, **fwidth**
 - ...

Utilisation des nuanceurs

- Les étapes :

1. Créer les objets de type nuanceur
2. Charger le code source pour chaque nuanceur
3. Compiler le code source
4. Créer un objet de type programme
5. Y attacher les objets nuanceurs
6. Faire l'édition des liens
7. Utiliser le programme
8. Attacher les variables uniformes

(Voir les exemples du cours)

Utilisation des nuanceurs

Exemples

```
void initialiserNuanceurs()
{
    // déclaration des chaînes qui devront recevoir le code des nuanceurs
    // lire les fichiers de nuanceurs de sommets et de fragments
    const char *ns = textFileRead( "nuanceurSommets.glsl" );
    const char *nf = textFileRead( "nuanceurFragments.glsl" );

    // créer le programme
    prog = glCreateProgram();

    // créer les nuanceurs de sommets et de fragments
    GLuint nuanceurSommets = glCreateShader( GL_VERTEX_SHADER );
    GLuint nuanceurFragments = glCreateShader( GL_FRAGMENT_SHADER );
```

Utilisation des nuanceurs

Exemples

```
// associer le fichier de sommets au nuageur de sommets
glShaderSource( nuageurSommets, 1, &ns, NULL );
glCompileShader( nuageurSommets );
glAttachShader( prog, nuageurSommets );
afficherShaderInfoLog( nuageurSommets, "nuageurSommets" );

// associer le fichier de fragments au nuageur de fragments
glShaderSource( nuageurFragments, 1, &nf, NULL );
glCompileShader( nuageurFragments );
glAttachShader( prog, nuageurFragments );
afficherShaderInfoLog( nuageurFragments, "nuageurFragments" );

// linker le programme
glLinkProgram( prog );
afficherProgramInfoLog( prog, "prog" );
}
```

Utilisation des nuanceurs

Exemples

```
void afficherShaderInfoLog( GLuint obj, const char* message )
{
    // afficher le message d'en-tête
    cout << message << endl;
    // afficher le message d'erreur, le cas échéant
    int infologLength = 0;
    glGetShaderiv( obj, GL_INFO_LOG_LENGTH, &infologLength );
    if ( infologLength > 1 )
    {
        char* infoLog = new char[infologLength+1];
        int charsWritten = 0;
        glGetShaderInfoLog( obj, infologLength, &charsWritten, infoLog );
        cout << infoLog << endl;
        delete[] infoLog;
    }
    else
        cout << "Aucune erreur : -)" << endl;
}
```

Utilisation des nuanceurs

Exemples

```
void afficherProgramInfoLog( GLuint obj, const char* message )
{
    // afficher le message d'en-tête
    cout << message << endl;
    // afficher le message d'erreur, le cas échéant
    int infologLength = 0;
    glGetProgramiv( obj, GL_INFO_LOG_LENGTH, &infologLength );
    if ( infologLength > 1 )
    {
        char* infoLog = new char[infologLength+1];
        int charsWritten = 0;
        glGetProgramInfoLog( obj, infologLength, &charsWritten, infoLog );
        cout << infoLog << endl;
        delete[] infoLog;
    }
    else
        cout << "Aucune erreur : -)" << endl;
}
```

Utilisation des nuances

1. Créer les objets de type nuanceur

```
GLuint glCreateShaderObject( GLenum shaderType );
```

- Cette fonction crée un objet nuanceur vide
- shaderType peut être GL_VERTEX_SHADER_ARB, GL_FRAGMENT_SHADER_ARB (ou GL_GEOMETRY_SHADER_ARB) afin d'indiquer quel type de nuanceur sera créé
- La valeur renournée sera utilisée comme référence
- Exemples :

```
GLuint nuanceurSommets = glCreateShader( GL_VERTEX_SHADER );  
GLuint nuanceurFragments = glCreateShader( GL_FRAGMENT_SHADER );
```

Utilisation des nuanceurs

2. Charger le code source pour chaque nuanceur

```
void glShaderSource( GLuint shader,  
                      GLsizei nstrings,  
                      const GLchar **string,  
                      GLint *length );
```

- Pour charger le code source à partir d'un tableau de chaînes de caractères
- `nstring` est la taille du tableau
- `string` est le tableau de chaînes (possiblement avec des '\n')
- `length` est un tableau d'entiers donnant la longueur de chaque chaînes (si NULL, alors c'est une seule chaîne et `nstring=1`)
- Exemple :

```
glShaderSource( nuanceurSommets, 1, &ns, NULL );
```

Utilisation des nuanceurs

3. Compiler le code source

```
void glCompileShader( GLuint shader );
```

- Pour compiler les nuanceurs
- Les messages de la compilation peuvent être obtenus en utilisant la fonction `glGetShaderInfoLog(. . .)`
- Exemple :

```
glCompileShader( nuageurSommets );
```

Utilisation des nuances

4. Créer un objet de type programme

```
GLuint glCreateProgramObject( void );
```

- Pour créer un objet programme et obtenir sa référence
- Exemple :

```
prog = glCreateProgram();
```

Utilisation des nuanceurs

5. Y attacher les objets nuanceurs

```
void glAttachObject( GLuint program, GLuint shader );
```

- Pour attacher les nuanceurs au programme
- Les nuanceurs de sommets et de fragments sont attachés au même programme
- Un programme peut contenir plusieurs nuanceurs
- OpenGL n'exécute qu'un seul programme à la fois
- Exemples :

```
glAttachShader( prog, nuanceurSommets );  
glAttachShader( prog, nuanceurFragments );
```

Utilisation des nuanceurs

6. Faire l'édition des liens

```
void glLinkProgram( GLuint program );
```

- Pour faire l'édition des liens du programme
- Les messages de la compilation peuvent être obtenus en utilisant la fonction `glGetProgramInfoLog(. . .)`;
- Exemple :

```
glLinkProgram( prog );
```

Utilisation des nuanceurs

7. Utiliser le programme

```
void glUseProgramObject( GLuint program );
```

- Pour dire à OpenGL d'utiliser notre programme en remplacement du pipeline fixe
- Pour désactiver les nuanceurs et revenir au pipeline fixe, on utilise :
`glUseProgramObject(NULL);`
- Exemples :
`glUseProgram(prog);`
`glUseProgram(NULL);`

Utilisation des nuanceurs

8. Attacher les variables uniformes

```
GLint glGetUniformLocation( GLuint program,  
                           const GLchar *name );  
void glUniform{1,2,3,4}{f,i,ui}( GLint location,  
                               TYPE v0, ... );
```

- Pour spécifier les variables uniformes (constantes)

- Exemples :

```
glUniform1f( glGetUniformLocation( prog, "facteurZ" ), 50.0 );  
glUniform2i( glGetUniformLocation( prog, "indices" ), 3, 12 );
```