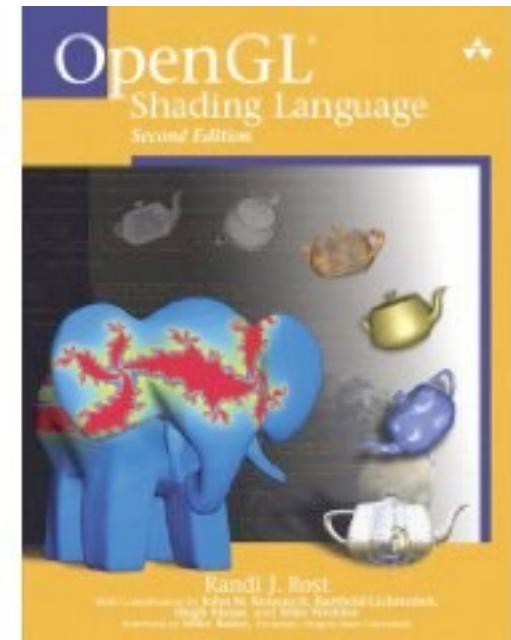


Les Shaders

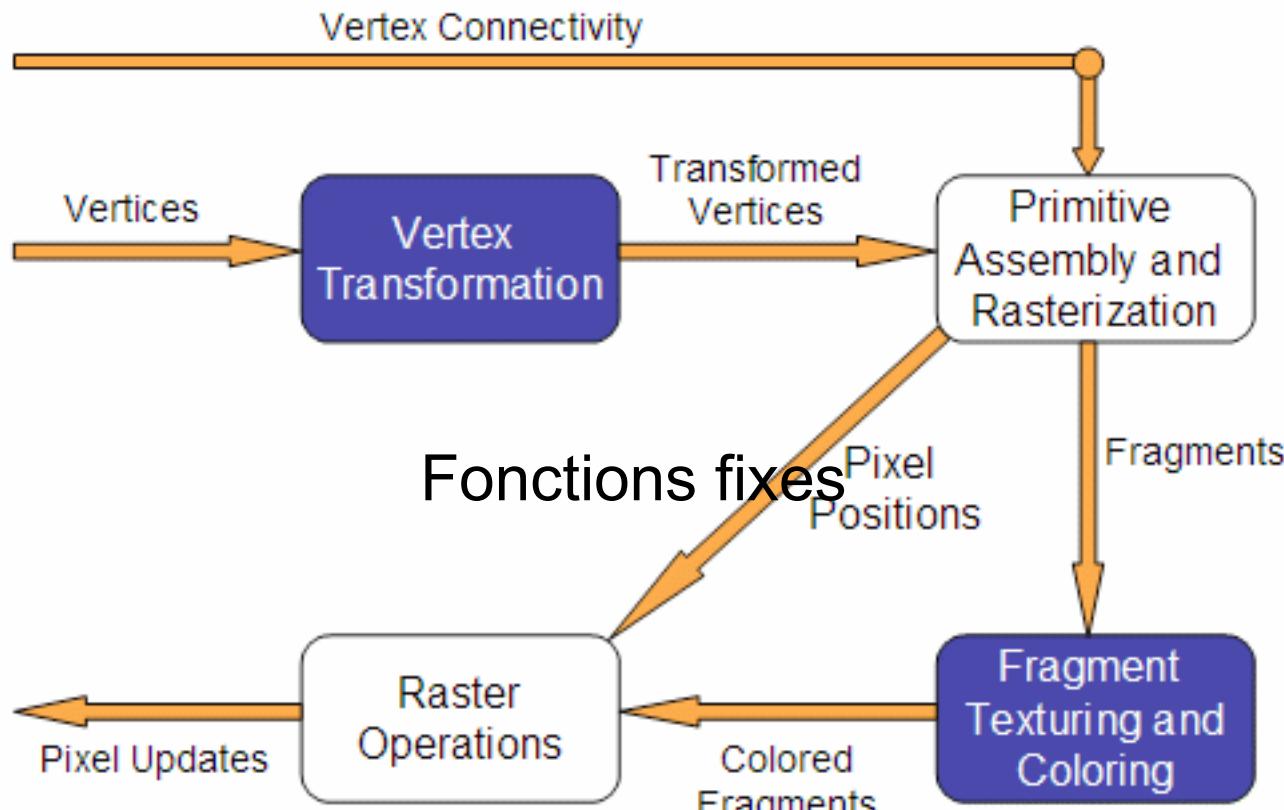
(anglais, du verbe **to shade** : ombrager ou estomper, nuancer)

Avant propos

- GLSL: GL Shading Language définit par l'ARB
- Cg: spécifiquement développé par Nvidia
- Les 2 produisent le même code compilé
- Le GLSL fait partie de OpenGL 2.0 et de requière pas de librairie spécifique.
- GLSL est plus proche de la syntaxe OpenGL.
- Ref: Orange Book
- <http://www.lighthouse3d.com>



Le pipeline Graphique

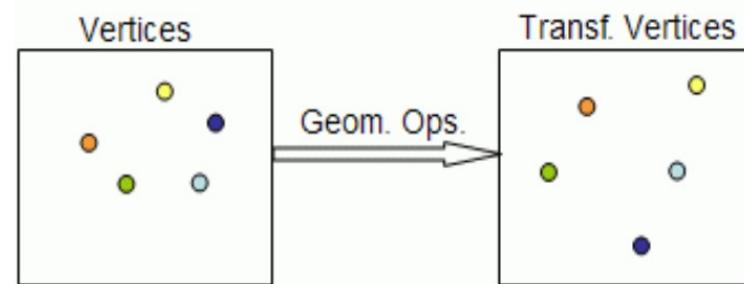
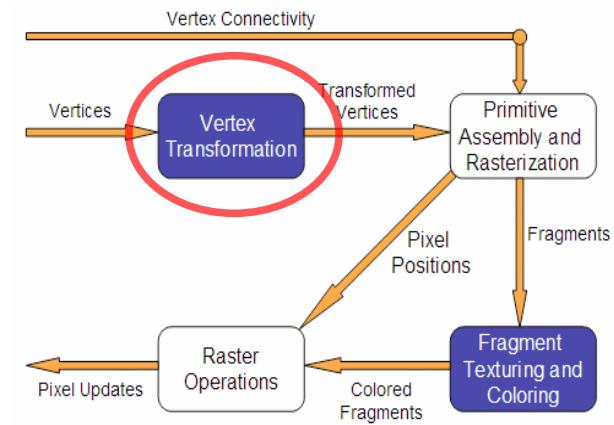


Fonctions fixes

Fonctions fixes ou programmables

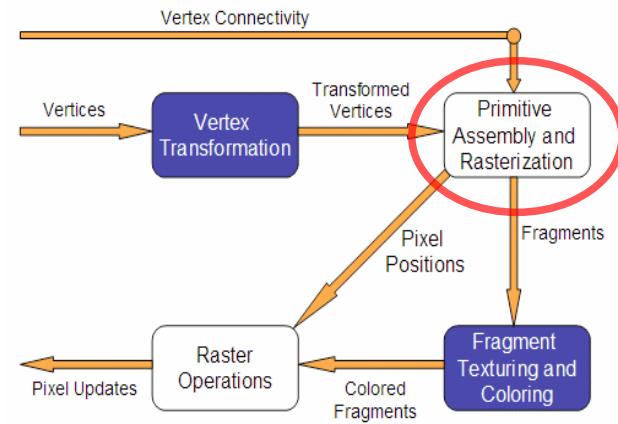
Les transformations de points

- Un point:
 - ensemble d'attributs:
 - localisation
 - couleur
 - normal
 - coordonnées de texture
 - ...
 - Ce sont les entrées de ce traitement
- Traitements fixes:
 - Transformations sur les positions des points
 - Calculs d'éclairages
 - Génération et transformations des coordonnées de textures

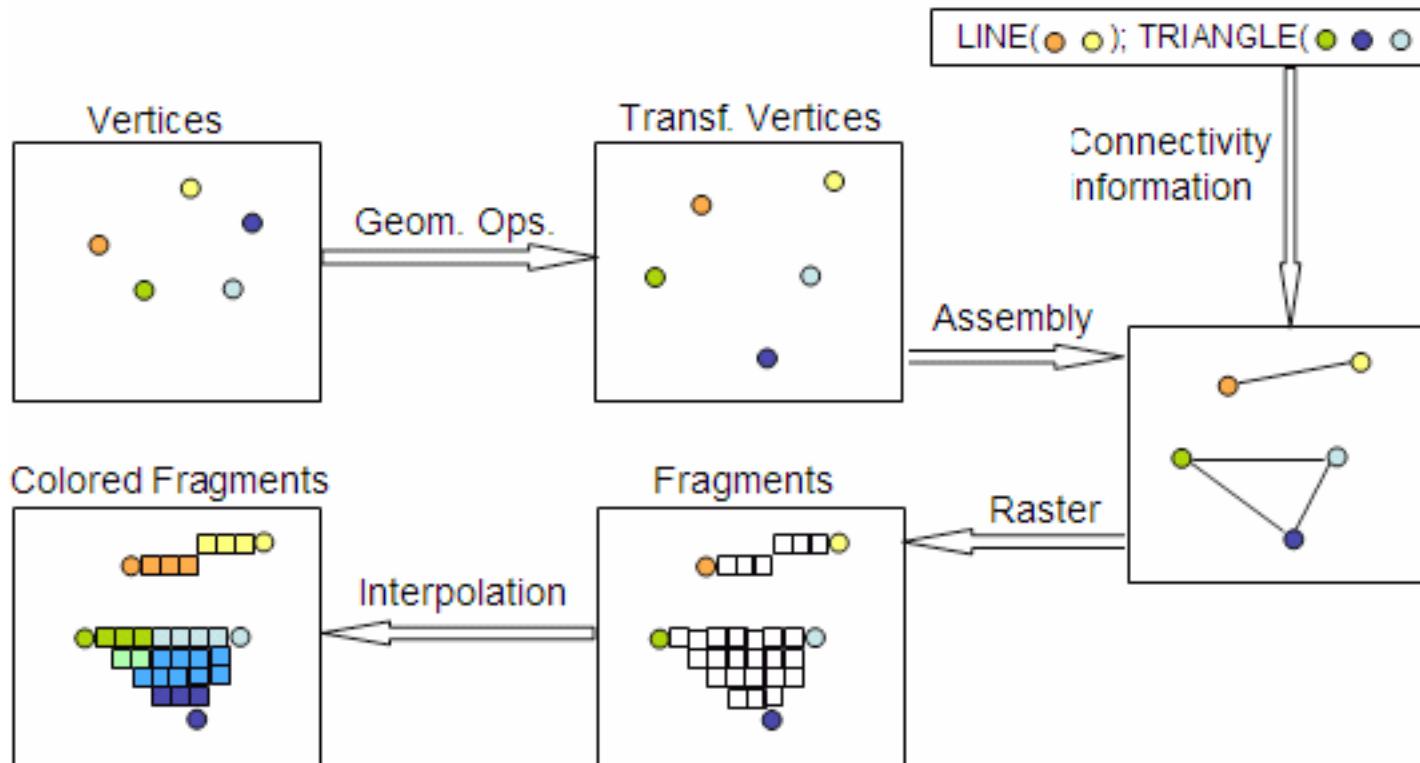
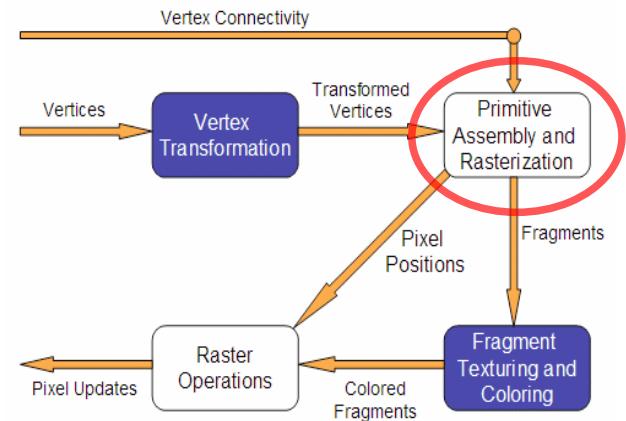


Assemblage des primitives et rasterisation

- Inputs:
 - Points transformés
 - Connectivité
- Traitements fixes:
 - Assemblage des primitives:
 - création des fragments et de leur positions
 - Fragment=Donnée qui permet de composer un pixel à une position spécifique.
 - Un fragment contient une couleur, des coordonnées de texture, ...
- Outputs:
 - Position du fragment dans le framebuffer
 - Les valeurs interpolées pour chaque fragment des attributs calculés à l'étape de transformation des points

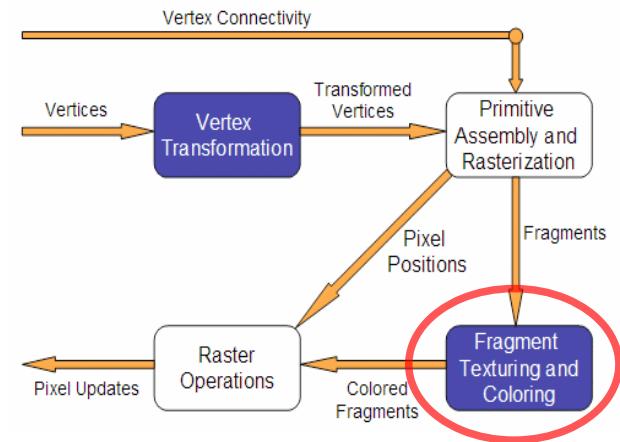


Assemblage des primitives et rasterisation



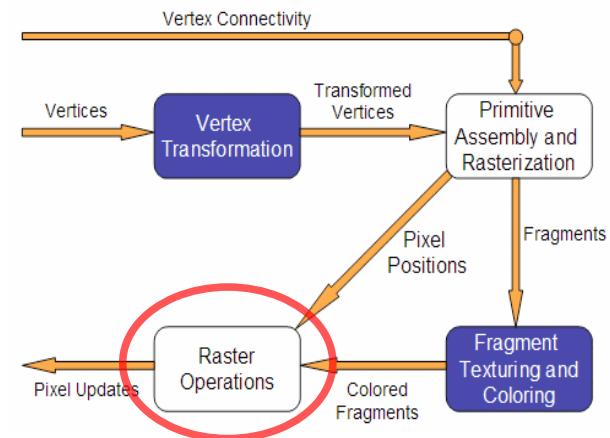
Texturation et coloration des fragments

- Inputs:
 - Informations interpolées des fragments
- Traitements fixes:
 - Combinaison:
 - couleurs
 - textures
 - Effet de fog
- Outputs:
 - Fragment colorées
 - Profondeur du fragment

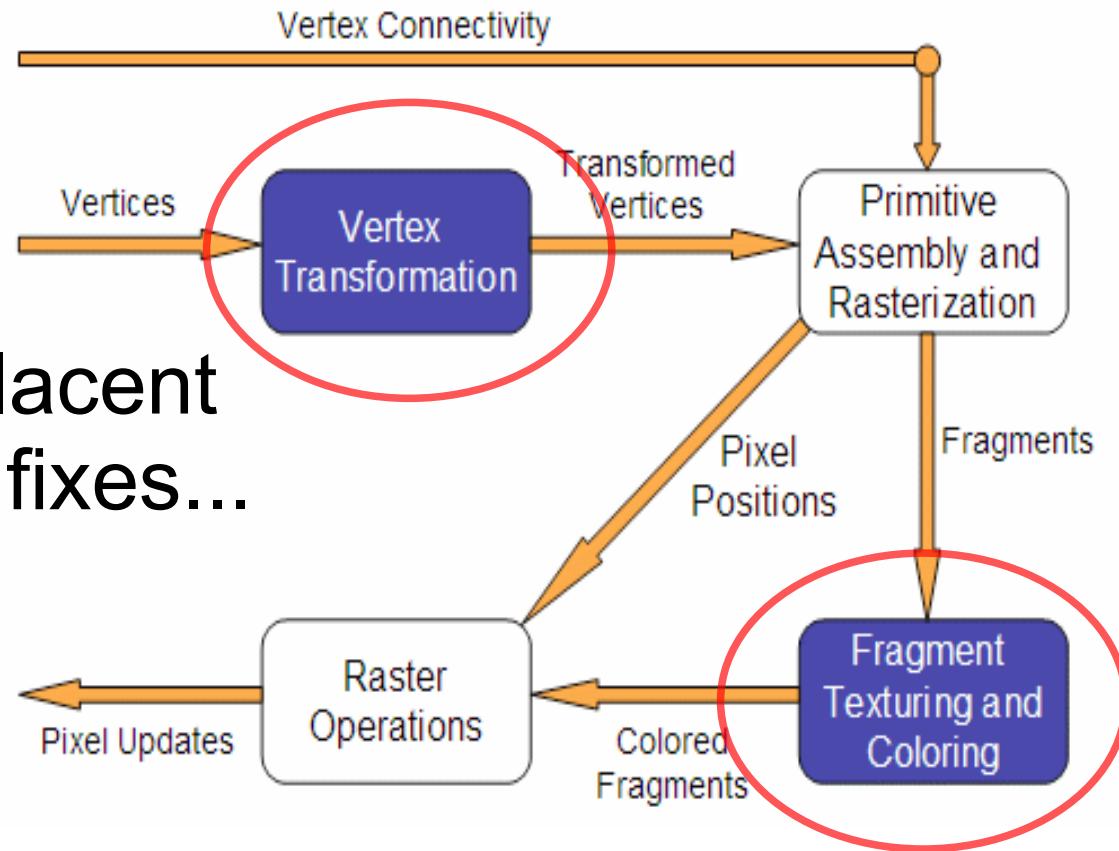


Rastering final

- Inputs:
 - Position des pixels
 - Les profondeurs et couleurs des fragments.
- Traitements fixes:
 - Scissor test
 - Alpha test
 - Stencil test
 - Depth test
- Outputs:
 - Si tests ok: alors mise à jour du framebuffer
 - Ici a lieu le blending puisque le framebuffer y est accessible seulement là.



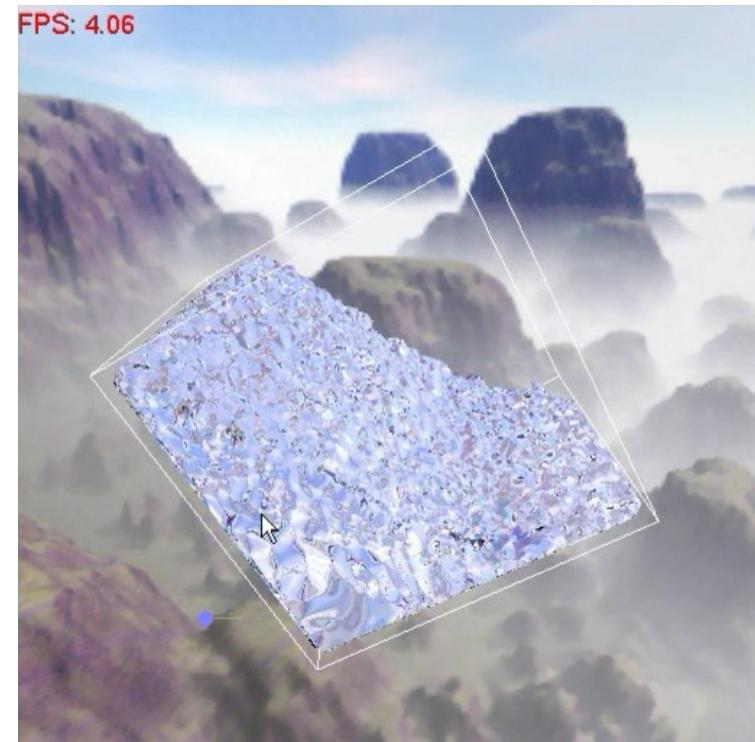
Les shaders remplacent certaines fonctions fixes...



- Pour les vertex shaders:
les transformations de points
- Pour les pixel shaders:
La texturation et coloration des fragments

Les geometry shaders!

- Nouveaux depuis les shader model 4.0, le geometry shader permet de modifier la géométrie de chaque polygone primitif.
- Il est exécuté entre le vertex shader et le fragment shader. Les geometry shader ont été implémentés avec la nouvelle version DirectX 10 et OpenGL 3.0
- Un geometry shader prend en entrée les données d'une primitive, et renvoie une ou plusieurs primitives (incluant souvent la même qu'en entrée éventuellement modifiée).



Le processeur de vertex

- Il exécute les vertex shaders
- Inputs:
 - position
 - normales
 - couleur
 - ...ça dépend de ce qu'envoie OpenGL...
- Exemple qui n'envoie que couleur et position.

```
glBegin(...);  
glColor3f(0.2,0.4,0.6);  
glVertex3f(-1.0,1.0,2.0);  
  
glColor3f(0.2,0.4,0.8);  
glVertex3f(1.0,-1.0,2.0);  
glEnd();
```

Le processeur de vertex

- Ce que l'on peut coder:
 - Transformation sur la position en utilisant les matrices modelview et projection.
 - Calcul de normale
 - Génération et transformation de coordonnées de texture
 - Calcul d'éclairage par point et valeurs utiles pour le calcul d'éclairage par pixel.
 - Calcul de couleur
- Pas obligé de tout faire!
- MAIS ce qui n'est pas fait ne l'est pas non plus par les fonctions fixes:
 - Pas de mélange des traitements
 - Il faut remplacer tout l'étage du pipeline graphique

Le processeur de vertex

- Le vertex processor n'a pas d'info concernant la connectivité:
 - Il ne peut pas faire des opérations qui requièrent ce genre d'info.
 - Exemple: pas de back face culling
 - Les points sont traité individuellement sans lien entre eux.
- Le vertex shader doit au minimum écrire la variable de position: `gl_Position`.
 - en utilisant les matrices modelview et de projection.
- Il a accès aux états OpenGL , donc il peut faire des opérations en conséquence comme celles sur la lumière qui utilise les paramétrages de matériaux.
- Il peut aussi accéder aux textures sur les architectures modernes.

Le processeur de vertex

- Exemple:

```
#version 150 core
```

```
in vec2 pos;
```

```
void main()
```

```
{
```

```
    gl_Position = vec4(pos, 0.0, 1.0);
```

```
}
```

Le processeur de fragment

- Il exécute le fragment shader
- Input:
 - Les valeurs de l'étage précédent:
 - Couleurs
 - Normales
 - Positions
- Ce qui est codé:
 - Les couleurs par pixel
 - Les coordonnées de texture par pixel
 - L'application des textures
 - Le brouillard
 - Les normales en cas d'éclairage par pixel

Le processeur de fragment

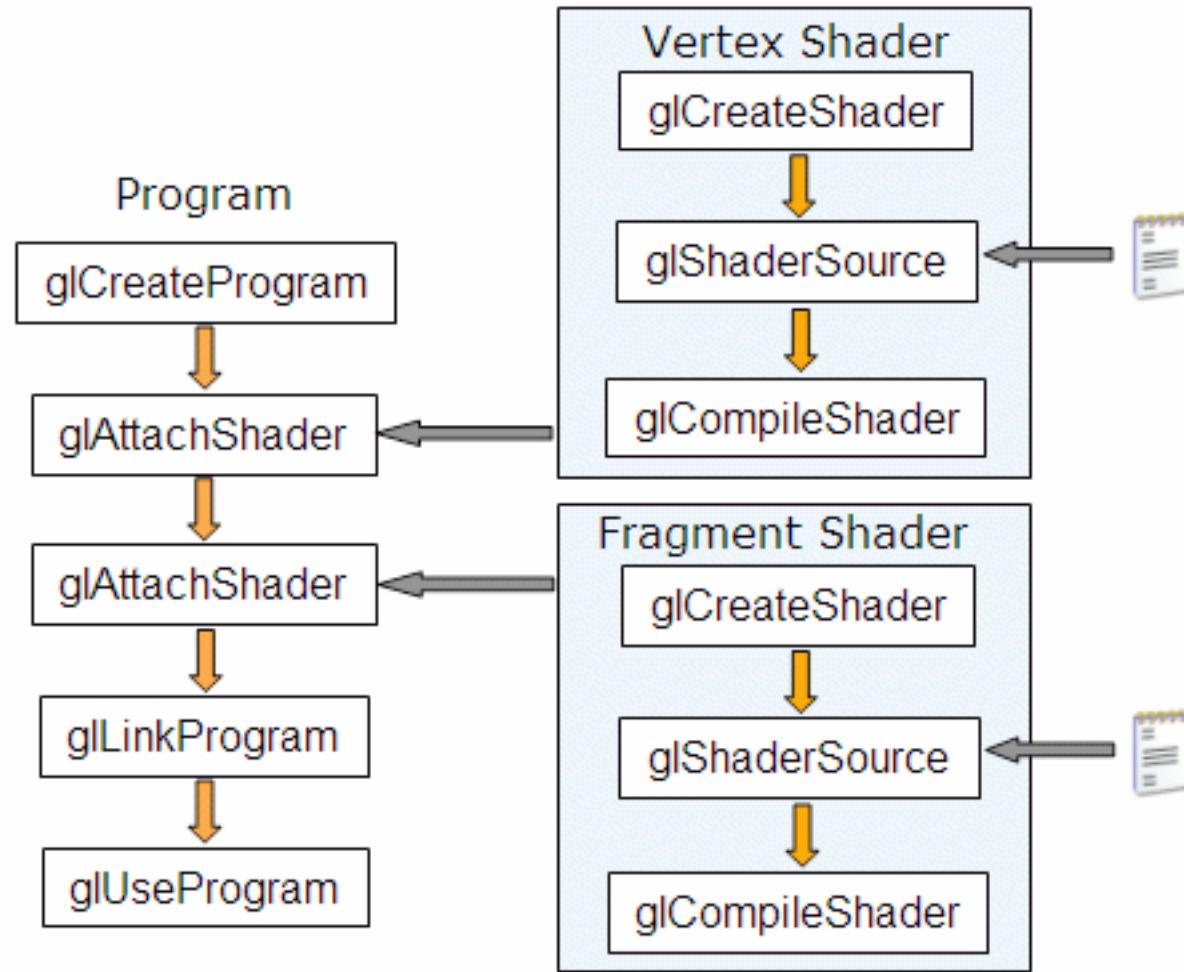
- Comme pour le précédent shader, il est impossible de mélanger fonctions fixes et shader.
- Il faut tout coder ou rien!
- Le shader n'opère que sur un fragment, on n'a pas accès aux voisins.
- On a accès à la position du pixel mais on ne peut pas la changer.
- Options de sortie:
 - désactiver le fragment (aucune sortie)
 - calcul de la couleur finale du fragment
- La profondeur peut être écrite aussi.

Le processeur de fragment

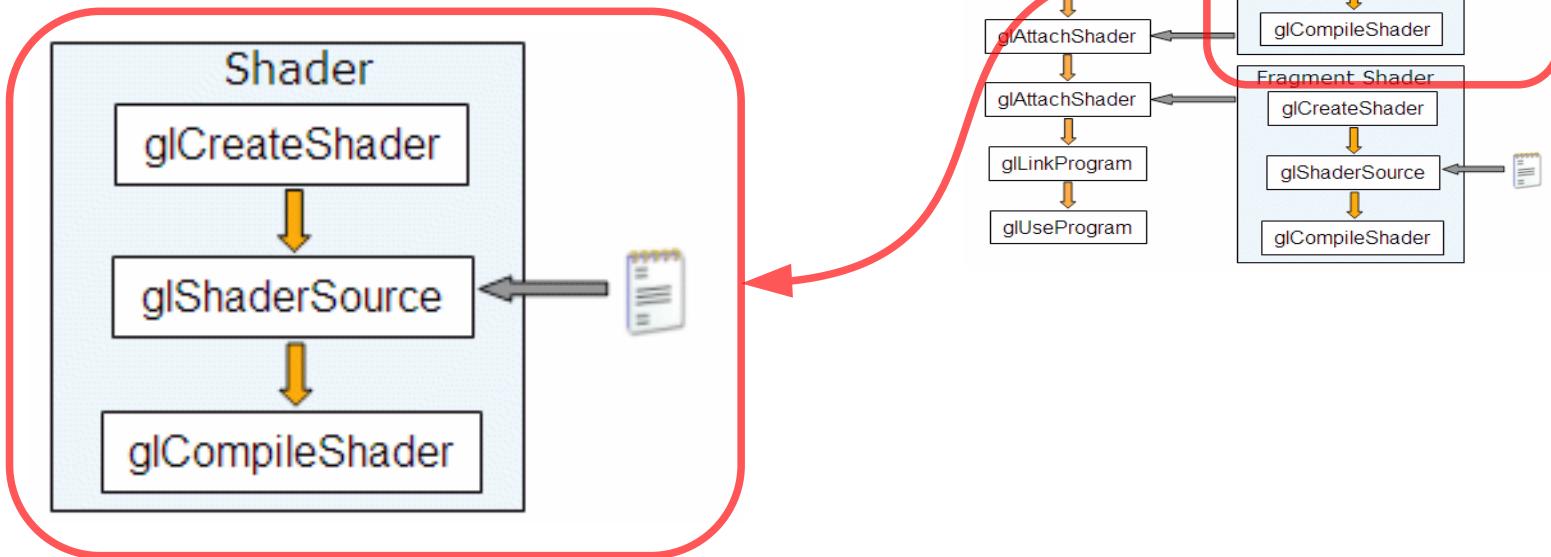
- Exemple:

```
#version 150 core  
  
out vec4 outColor;  
  
void main()  
{  
    outColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

Etapes pour créer les shaders



Etapes pour créer les shaders



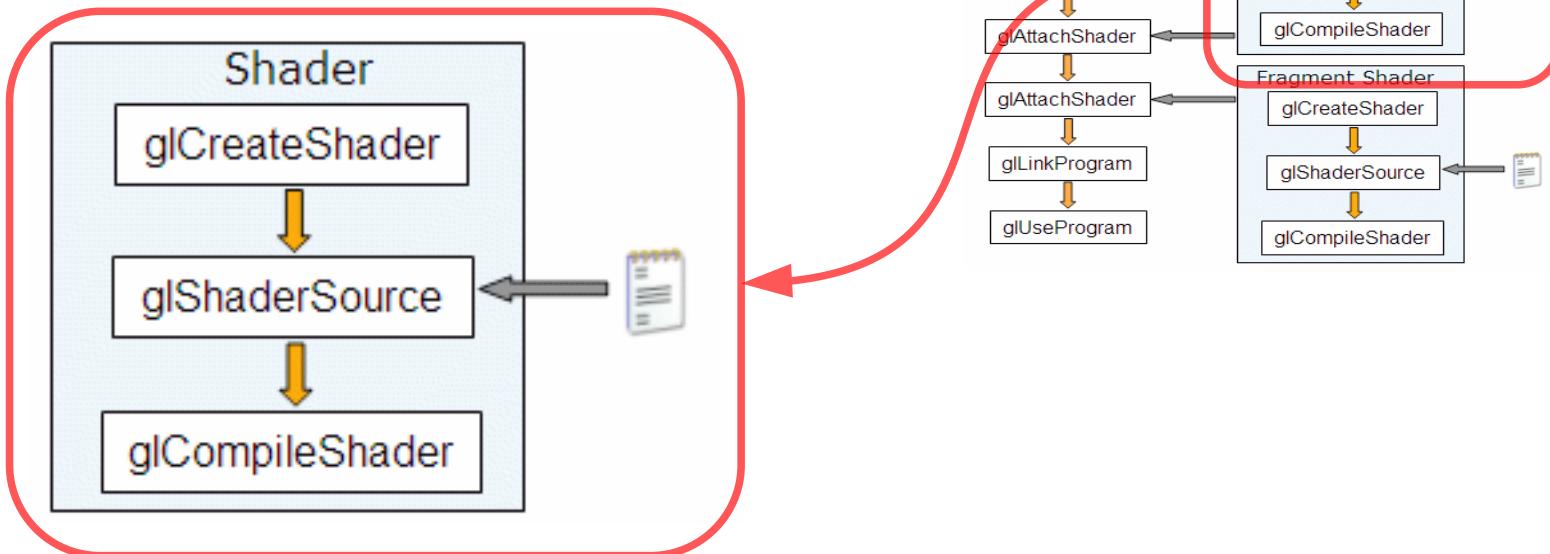
- Création d'un container pour le shader

```
GLuint glCreateShader(GLenum shaderType);
```

- Paramètre:

shaderType - **GL_VERTEX_SHADER** or
GL_FRAGMENT_SHADER.

Etapes pour créer les shaders



- On peut créer plusieurs shader mais il faut une seule fonction main() pour chaque type de shader.

Etapes pour créer les shaders

- Création du code du shader
- Il est passé sous la forme d'un tableau de chaînes de caractères:

```
void glShaderSource(GLuint shader, int  
numOfStrings, const char **strings, int  
*lenOfStrings);
```

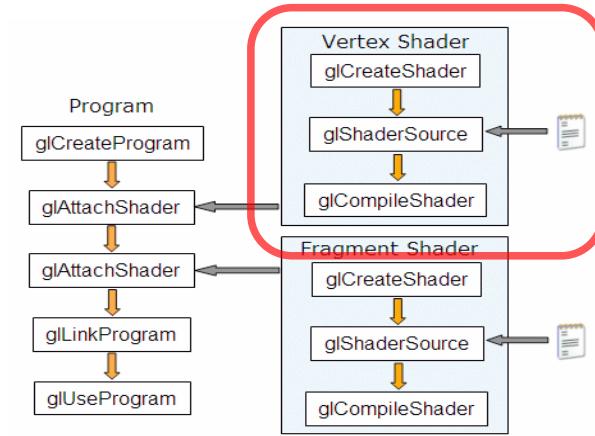
Paramètres:

shader – le container du shader.

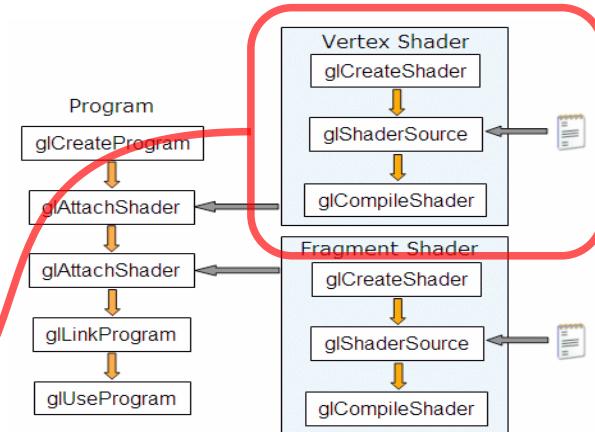
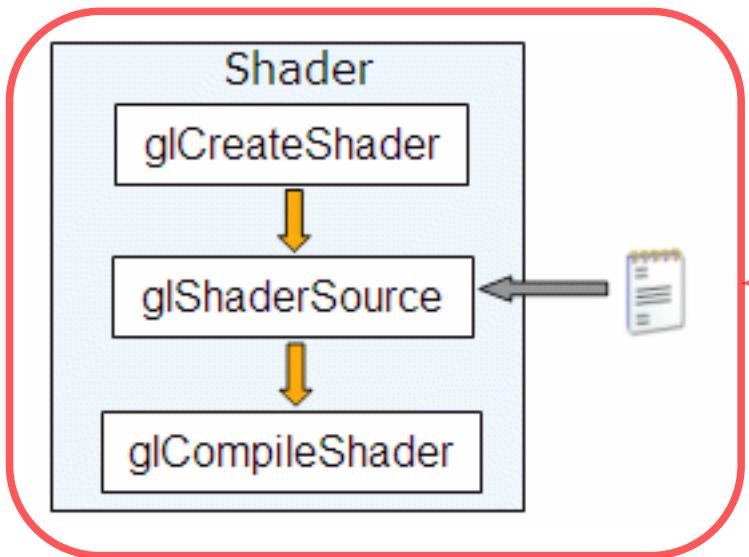
numOfStrings – le nombre de chaînes dans le tableau.

strings – Le tableau.

lenOfStrings – Tableau avec la longueur de chaque chaîne ou bien NULL si la fin de chaîne est marquée par NULL.



Etapes pour créer les shaders



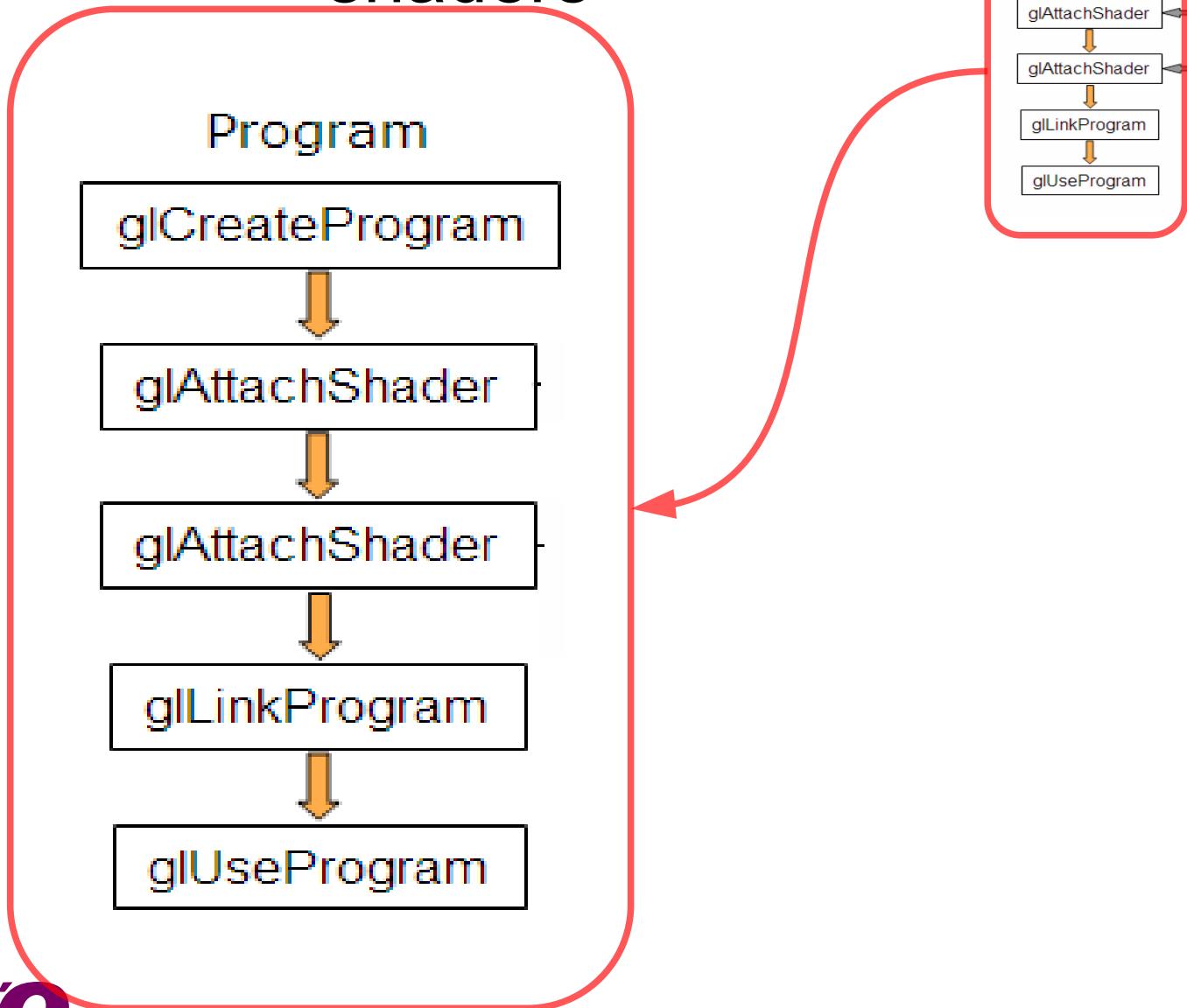
- Compilation du shader:

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

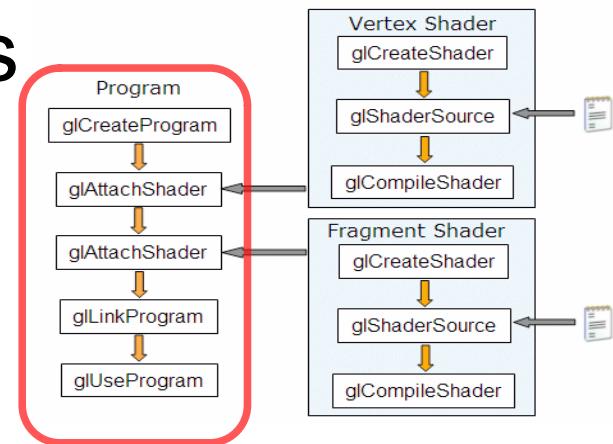
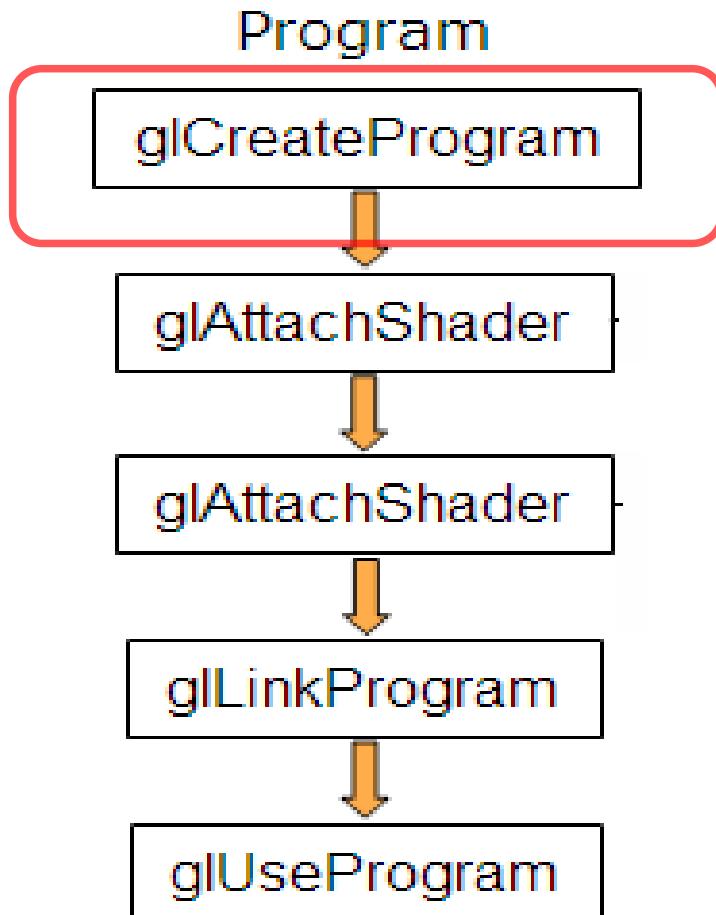
- Paramètres:

shader – le container du shader.

Etapes pour rendre effectifs les shaders

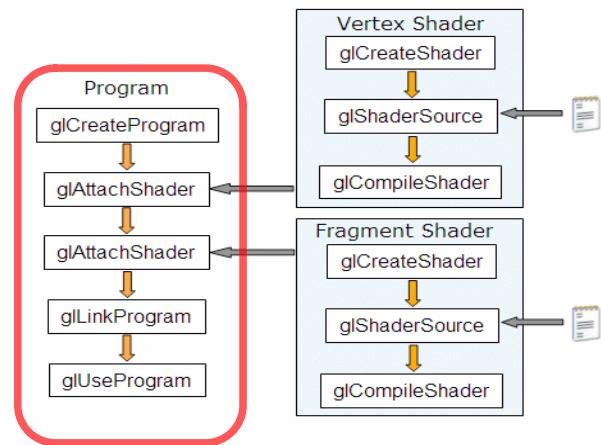
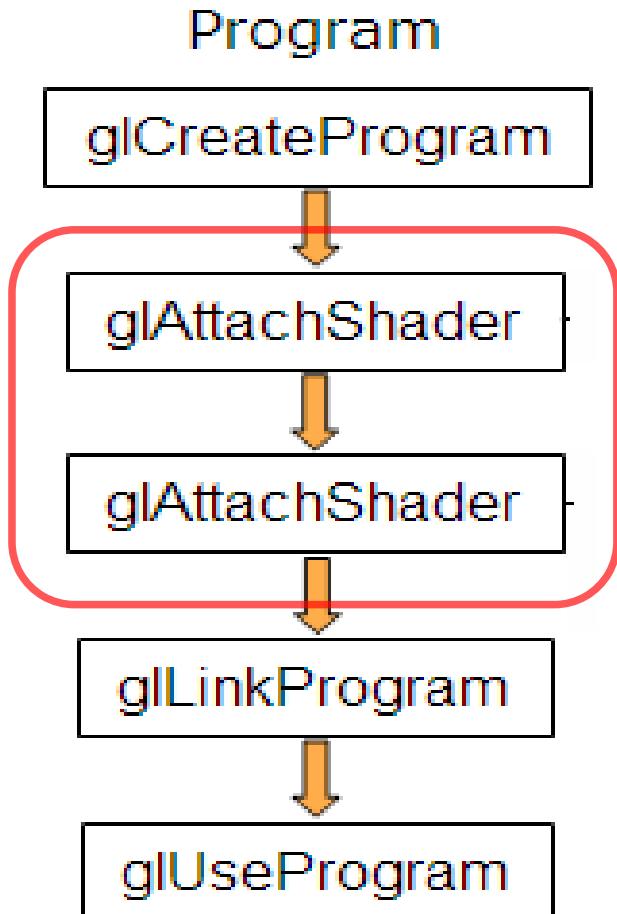


Etapes pour rendre effectifs les shaders



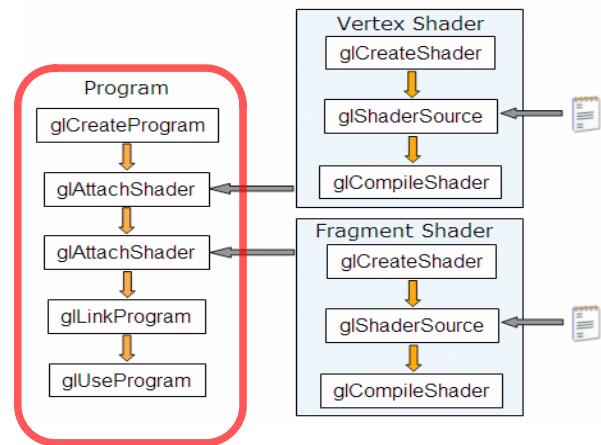
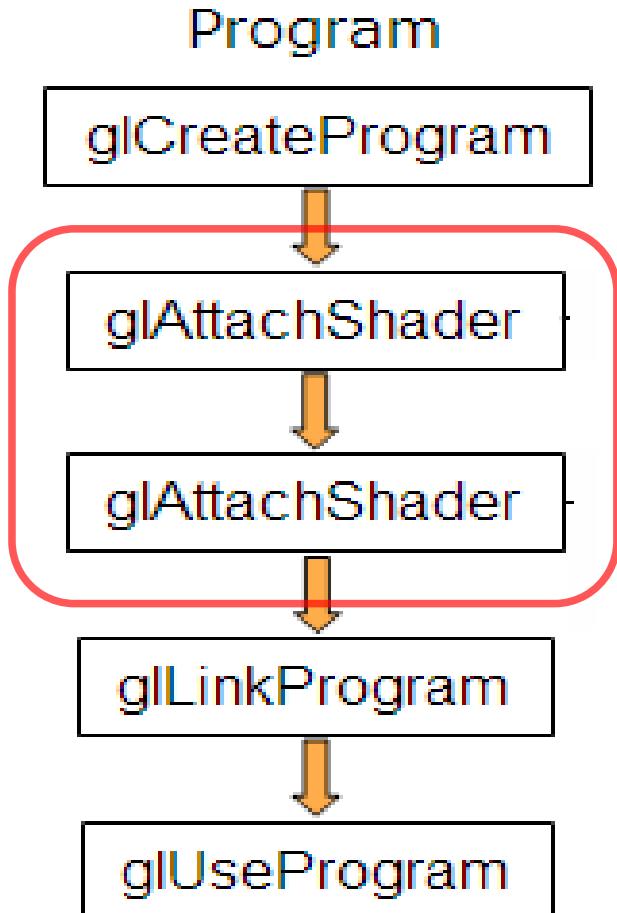
- Création d'un container:
GLuint
glCreateProgram(void);
- Il est possible de créer plusieurs programmes et ainsi d'en changer en cours de rendu pour varier les effets

Etapes pour rendre effectifs les shaders



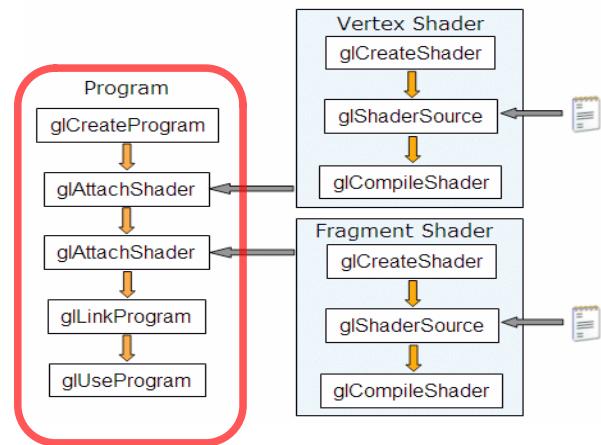
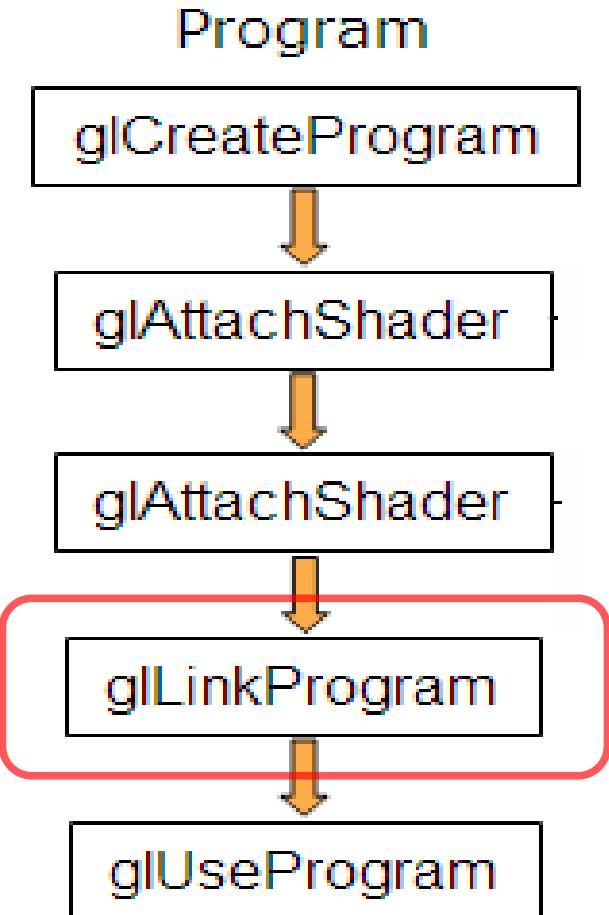
- On attache les shaders
`void glAttachShader(GLuint program, GLuint shader);`
- Parameters:
 - `program` – Le container du programme
 - `shader` - Le container du shader visé.

Etapes pour rendre effectifs les shaders



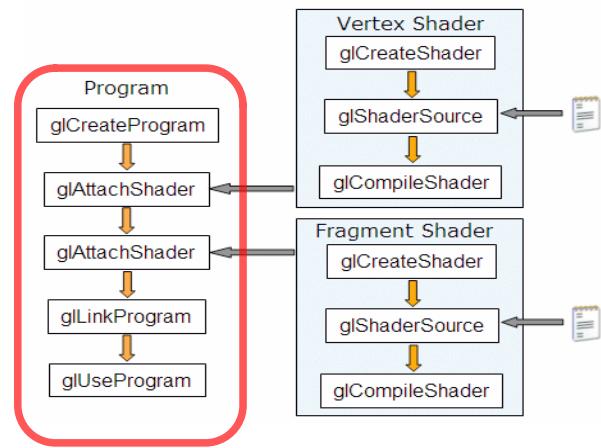
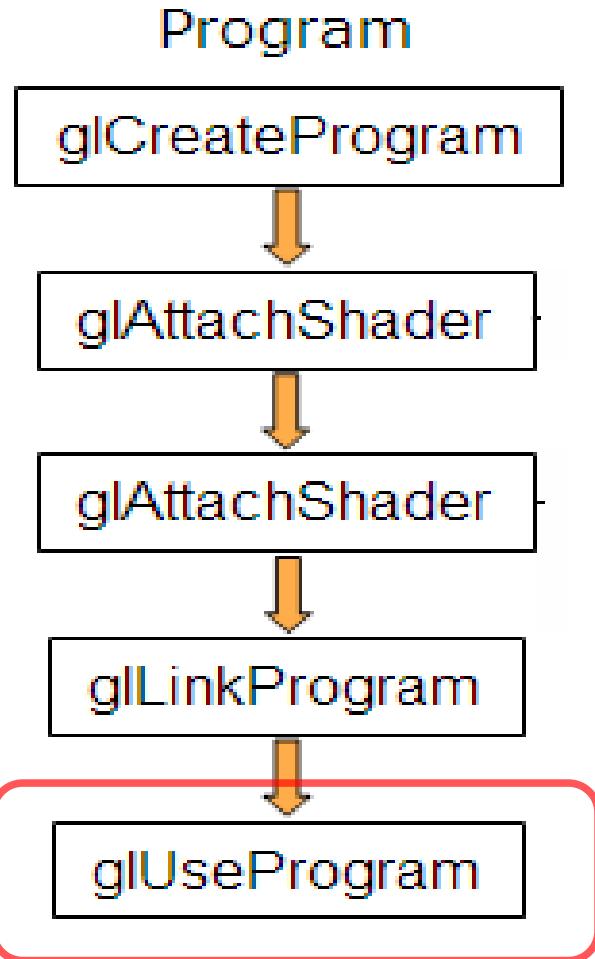
- Il faut attacher une paire de shaders :vertex/fragment
- Il est possible d'attacher un même shader à plusieurs programmes
- il est possible d'attacher plus d'un shader par type de shader si on a plusieurs fonctions.
(mais un seul main)

Etapes pour rendre effectifs les shaders



- Edition de lien:
glLinkProgram(GLuint program);
- Parameters:
program – le container

Etapes pour rendre effectifs les shaders



- Désignation du programme que l'on utilise:
`void glUseProgram(GLuint prog);`
- Paramètres:
- `prog` – le container du programme ou zero pour retourner aux fonctions fixes.

Se débarrasser des shader

- Les détacher:

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

- Les effacer:

```
void glDeleteShader(GLuint id);
```

- id – Le container du shader ou du programme

Communication entre OpenGL et les shader

- OpenGL \Rightarrow shader:
 - Les variables d'état OpenGL
 - Des variables non prédéfinies
 - exemple: temps passé pour une animation donnée...
 - Les textures : qui n'ont pas forcément une sémantique classique
- shader \Rightarrow OpenGL:
 - Les couleurs, framebuffers, etc

Communication entre OpenGL et les shader

- Les variables de GLSL:
 - Uniform
 - Attribute
- Read only du point de vue des shaders

Communication entre OpenGL et les shader

- Les variables Uniform:
 - Ne peut être changée que par primitive
 - ne peut pas être entre un glBegin/glEnd
 - Ne peut être une caractéristique d'un vertex
 - Ne peut être écrite par le shader
- Mise en oeuvre:
 - Allocation mémoire:
 - `GLint glGetUniformLocation(GLuint program, const char *name);`
 - `program` – container de program
 - `name` – nom de la variable sous la forme d'une chaîne.

Les variables Uniform

- Mise en oeuvre :
 - Affectation (ici f pour float mais idem avec i pour les integers):
 - `void glUniform1f(GLint location, GLfloat v0);`
 - `void glUniform2f(GLint location, GLfloat v0, GLfloat v1);`
 - `void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);`
 - `void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);`
 - `GLint glUniform{1,2,3,4}fv(GLint location, GLsizei count, GLfloat *v);`
 - Paramètres:
 - `location` – position mémoire.
 - `v0, v1, v2, v3` - float values.
 - `count` – nombre d'éléments dans le tableau
 - `v` – tableau de floats

Les variables Uniform

- Mise en oeuvre :
 - Affectation des matrices:
 - `GLint glUniformMatrix{2,3,4}fv(GLint location, GLsizei count, GLboolean transpose, GLfloat *v);`
 - Paramètres:
 - location – position mémoire
 - count – le nombre de matrices
 - transpose - 1 indique que la matrice est en mode row major et zero est réservé au mode column major.
 - v – tableau de floats

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Exemple

```
GLint loc1,loc2,loc3,loc4;  
float specIntensity = 0.98;  
float sc[4] = {0.8,0.8,0.8,1.0};  
float threshold[2] = {0.5,0.25};  
float colors[12] = {0.4,0.4,0.8,1.0,  
                    0.2,0.2,0.4,1.0,  
                    0.1,0.1,0.1,1.0};  
  
loc1 =  
glGetUniformLocation(p,"specIntensity");  
glUniform1f(loc1,specIntensity);  
  
loc2 = glGetUniformLocation(p,"specColor");  
glUniform4fv(loc2,1,sc);  
  
loc3 = glGetUniformLocation(p,"t");  
glUniform1fv(loc3,2,threshold);  
  
loc4 = glGetUniformLocation(p,"colors");  
glUniform4fv(loc4,3,colors);
```

```
uniform float  
specIntensity;  
uniform vec4 specColor;  
uniform float t[2];  
uniform vec4 colors[3];
```

Les variables Attribute

- Elles sont utilisées pour les paramètres qui changent avec les vertex.
 - Entre glBegin et glEnd.
 - Ne peuvent pas être lues par le fragment shader
- Mise en oeuvre:
 - Allocation mémoire:
 - `GLint glGetAttribLocation(GLuint program, const char *name);`
 - `program` – container de program
 - `name` – nom de la variable sous la forme d'une chaîne.

Les variables Attribute

- Mise en oeuvre :
 - Affectation (ici f pour float mais idem avec i pour les integers):
 - `void glAttrib1f(GLint location, GLfloat v0);`
 - `void glAttrib2f(GLint location, GLfloat v0, GLfloat v1);`
 - `void glAttrib3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);`
 - `void glAttrib4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);`
 - `GLint glAttrib{1,2,3,4}fv(GLint location, GLsizei count, GLfloat *v);`
 - Paramètres:
 - `location` – position mémoire.
 - `v0, v1, v2, v3` - float values.
 - `count` – nombre d'éléments dans le tableau
 - `v` – tableau de floats

Exemple

```
loc = glGetUniformLocation(p, "height");
glBegin(GL_TRIANGLE_STRIP);

    glVertexAttrib1f(loc,2.0);
    glVertex2f(-1,1);

    glVertexAttrib1f(loc,2.0);
    glVertex2f(1,1);

    glVertexAttrib1f(loc,-2.0);
    glVertex2f(-1,-1);

    glVertexAttrib1f(loc,-2.0);
    glVertex2f(1,-1);

glEnd();
```

```
in float height;
```

Les variables Attribute

- Avec les vertex array :
 - déclaration:
 - `void glEnableVertexAttribArray(GLint loc);`
 - `loc` - the location of the variable.
 - Le pointeur sur les données:
 - `glVertexAttribPointer(GLint loc, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);`
 - `loc` - position
 - `size` - the number of components per element, for instance: 1 for float; 2 for vec2; 3 for vec3, and so on.
 - `type` - The data type associated: `GL_FLOAT` is an example.
 - `normalized` - if set to 1 then the array values will be normalized, converted to a range from -1 to 1 for signed data, or 0 to 1 for unsigned data.
 - `stride` - the spacing between elements. Exactly the same as in OpenGL.
 - `pointer` - pointer to the array containing the data.

Exemple

```
in float height;
```

```
float vertices[8] = {-1,1, 1,1, -1,-1, 1,-1};  
float heights[4] = {2,2,-2,-2};
```

```
...
```

```
loc = glGetAttribLocation(p,"height");
```

```
glEnableClientState(GL_VERTEX_ARRAY);
```

```
glEnableVertexAttribArray(loc);
```

```
glVertexPointer(2,GL_FLOAT,0,vertices);
```

```
glVertexAttribPointer(loc,1,GL_FLOAT,0,0,heights);
```

Variables GLSL

- Simples:

`float`

`bool`

`int`

- Vecteurs:

`vec{2,3,4}` a vector of 2,3,or 4 floats

`bvec{2,3,4}` bool vector

`ivec{2,3,4}` vector of integers

- Matrices:

`mat2`

`mat3`

`mat4`

Variables GLSL

```
struct dirlight {  
    vec3 direction;  
    vec3 color;  
};  
  
float a,b;      // two vector (yes, the comments are like in C)  
int c = 2;       // c is initialized with 2  
bool d = true; // d is true  
  
float b = 2;      // incorrect, there is no automatic type casting  
float e = (float)2;// incorrect, requires constructors for type  
casting
```

Variables GLSL

```
int a = 2;  
  
float c = float(a); // correct. c is 2.0  
  
  
vec3 f;    // declaring f as a vec3  
  
vec3 g = vec3(1.0,2.0,3.0); // declaring and initializing g  
  
  
  
vec2 a = vec2(1.0,2.0);  
  
vec2 b = vec2(3.0,4.0);  
  
vec4 c = vec4(a,b) // c = vec4(1.0,2.0,3.0,4.0);  
  
  
  
vec2 g = vec2(1.0,2.0);  
  
float h = 3.0;  
  
vec3 j = vec3(g,h);
```

Variables GLSL

```
mat4 m = mat4(1.0) // initializing the diagonal of the matrix with  
1.0
```

```
vec2 a = vec2(1.0,2.0);  
vec2 b = vec2(3.0,4.0);
```

```
mat2 n = mat2(a,b); // matrices are assigned in column major order
```

```
mat2 k = mat2(1.0,0.0,1.0,0.0); // all elements are specified
```

Variables GLSL

```
struct dirlight {      // type definition
    vec3 direction;
    vec3 color;
};

dirlight d1;

dirlight d2 = dirlight(vec3(1.0,1.0,0.0),vec3(0.8,0.8,0.4));

d1.direction = vec3(1.0,1.0,1.0);
```

Variables GLSL

- Des facilités

```
vec4 a = vec4(1.0,2.0,3.0,4.0);
```

```
float posX = a.x;
```

```
float posY = a[1];
```

```
vec2 posXY = a.xy;
```

```
float depth = a.w
```

Structures de contrôle

```
if (bool expression)
```

```
    ...
```

```
else
```

```
    ...
```

```
for (initialization; bool expression; loop expression)
```

```
    ...
```

```
while (bool expression)
```

```
    ...
```

```
do
```

```
    ...
```

```
while (bool expression)
```

continue – saut à la prochaine
itération
break – fin de la boucle
discard -sortie du shader de
fragment et pas d'écriture en sortie

Fonctions

- Au moins une fonction:

```
void main()
```

- Type de paramètres:

- **in** - lecture
- **out** - sortie
- **inout** – lecture écriture

- Le Return permet aussi de réaliser une sortie

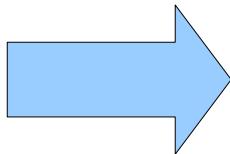
```
vec4 toonify(in float intensity) {  
    vec4 color;  
    if (intensity > 0.98)  
        color =  
    vec4(0.8,0.8,0.8,1.0);  
    else if (intensity > 0.5)  
        color =  
    vec4(0.4,0.4,0.8,1.0);  
    else if (intensity > 0.25)  
        color =  
    vec4(0.2,0.2,0.4,1.0);  
    else  
        color =  
    vec4(0.1,0.1,0.1,1.0);  
    return(color);  
}
```

Les variables Varying

- Seul façon de transmettre des données depuis le shader de vertex vers celui de fragments.
- Ces données seront interpolées
- Elles sont en lecture seules pour les fragment shader

```
out float intensity; //shader de vertex  
in float intensity; //shader de fragment
```

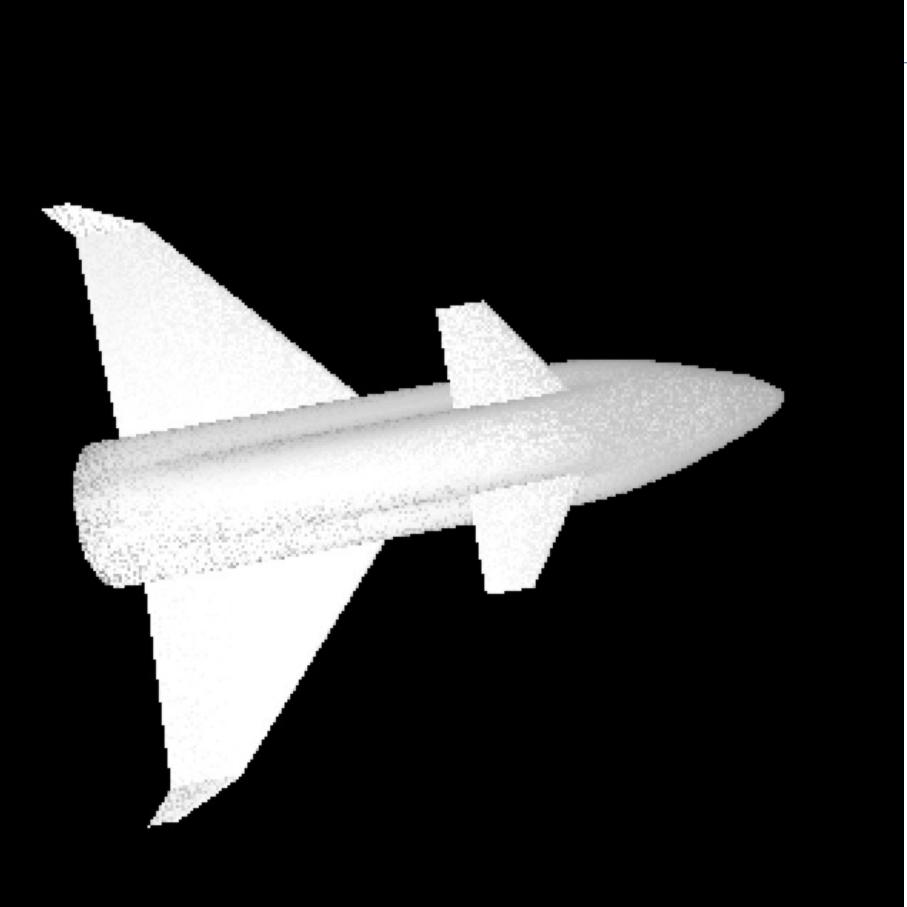
Exemples: position bruitée



```
void main(void)  
{  
    gl_Position = P*MV * (vec4(pos, 1.0)+.4*noise1(pos.x ));  
}
```

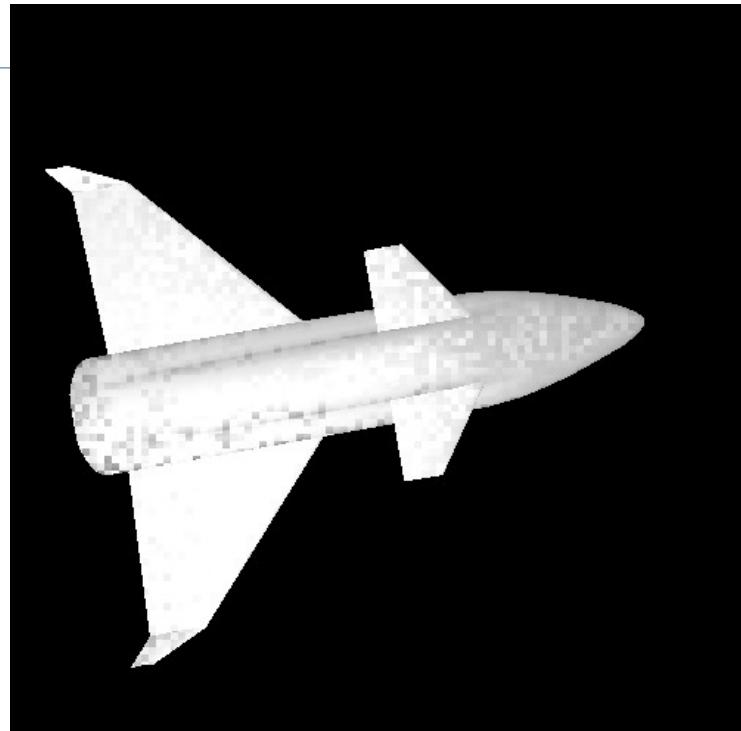
Exemples: normale bruitée

```
float random (vec2 st) {  
    return fract(sin(dot(st.xy,  
        vec2(12.9898,78.233)) *  
        43758.5453123);  
}  
  
void main(void)  
{  
    vec3 N = normalize(normalEyeSpace+1.0*random(normalEyeSpace.xy));  
}
```



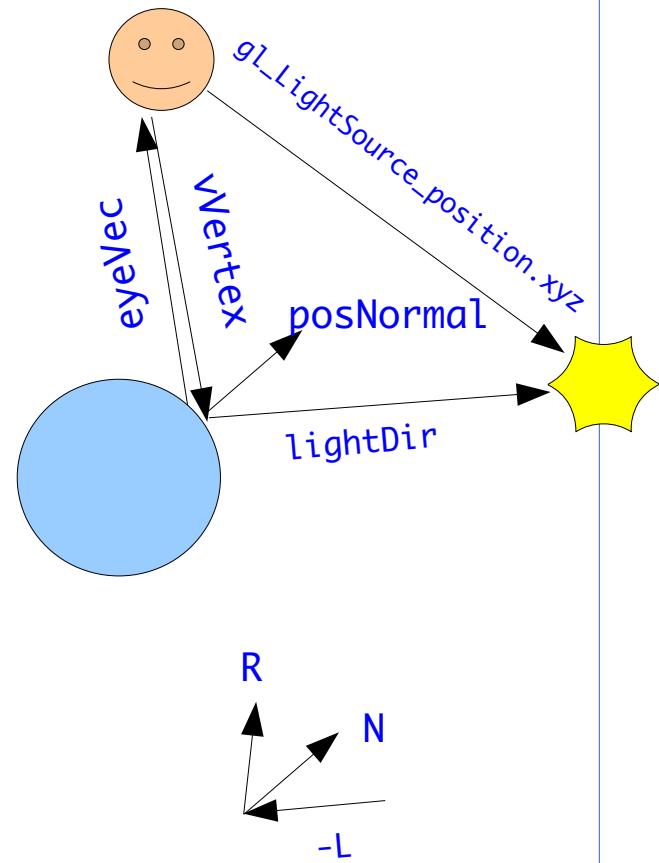
Exemples: normale bruitée

```
float random (vec2 st) {  
  
    return fract(sin(dot(st.xy,vec2(12.9898,78.233)))*  
43758.5453123);}  
  
  
  
  
  
  
void main(void)  
{ vec2 st = gl_FragCoord.xy/vec2(500,500);  
  
    st *= 100.0; // Scale the coordinate system by 100  
  
    vec2 ipos = floor(st); // get the integer coords  
  
    vec2 fpos = fract(st); // get the fractional coords  
  
    vec4 final_color =vec4(0.5,0.5,0.5, 1.0);  
  
    vec3 N = normalize(normalEyeSpace+vec3(random( ipos )));}
```

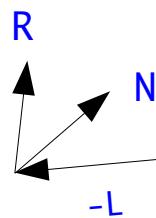
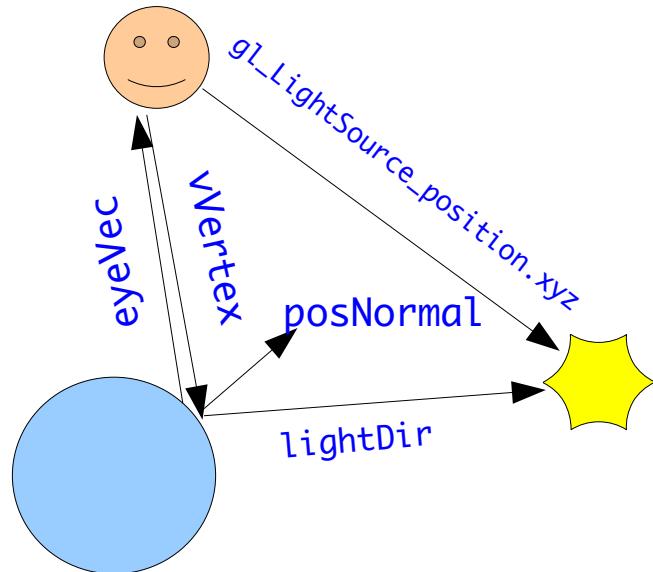


Exemple: phong

```
#version 150 core  
  
in vec3 pos;  
  
in vec3 posNormal;  
  
uniform mat4 MV;  
  
uniform mat4 P;  
  
uniform vec4 LightSource_position;  
  
out vec3 lightDir;  
  
out vec3 normalEyeSpace, eyeVec;  
  
void main()  
{  
    gl_Position = P*MV * vec4(pos, 1.0);  
  
    normalEyeSpace = vec3(MV * vec4(posNormal,0.0));  
  
    vec4 vVertex = MV * vec4(pos, 1.0);  
  
    eyeVec = -vVertex.xyz;  
  
    lightDir=vec3(LightSource_position.xyz - vVertex.xyz);  
}
```



Exemple: phong



```
#version 150 core

in vec3 lightDir;

in vec3 normalEyeSpace, eyeVec;

void main()

{ vec4 final_color =vec4(0.5,0.5,0.5, 1.0);

  vec3 N = normalize(normalEyeSpace);

  vec3 L = normalize(lightDir);

  float lambertTerm = dot(N,L);

  if (lambertTerm > 0.0)

  {

    final_color +=0.5*lambertTerm;

    vec3 E = normalize(eyeVec);

    vec3 R = reflect(-L, N);

    float specular = pow(max(dot(R, E), 0.0),0.5);

    final_color +=0.1*specular;

  }

  outColor = final_color;
```