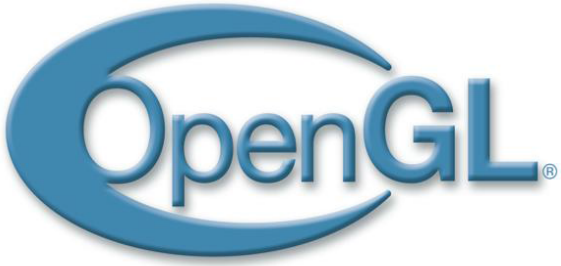


OpenGL 3.x
Shaders avec
GL Shading Language

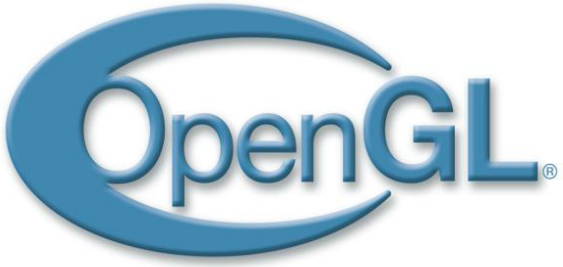
Marc Neveu

Avec des morceaux de cours d'Olivier Nucent

http://www.opengl.org/documentation/glsl/

http://www.opengl.org/documentation/current_version/

OpenGL 3.x



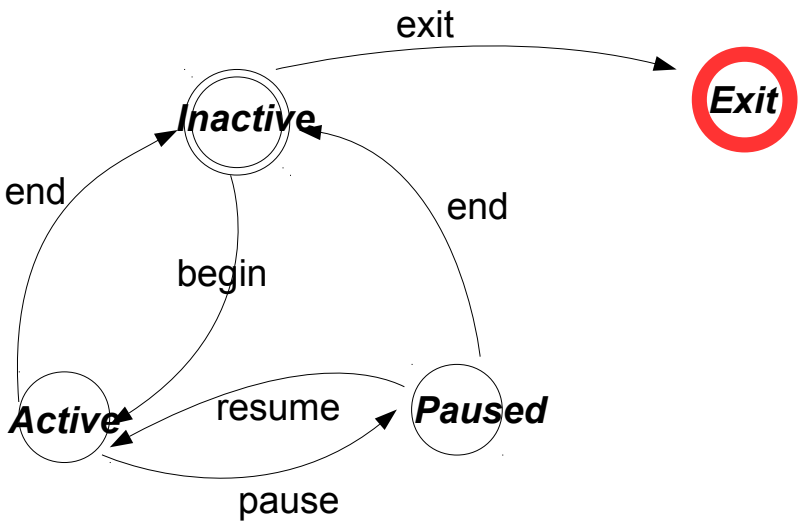
Il est important d'avoir du retard : on en est au moins à OpenGL 4.x et GLSL 4.x

Librairie graphique, API standard pour graphique, libre, multiplateformes, accélérée par hardware, implémentée par les constructeurs de cartes graphiques, utilisable avec C, C++, C#, Java, Fortran, Perl, Python, Delphi, etc.

Rappels de base OpenGL

OpenGL est une **machine à états :**

- Un état reste actif tant qu'il n'est pas changé par une transition.
- Une transition est réalisée par un appel à une fonction OpenGL.
- Les états sont déterminés par les objets-OpenGL courants



Rappels de base OpenGL

Inclure OpenGL + Link avec bibliothèque OpenGL (opengl.a, ...) et autres

(GLEW, GLFW, ...):

```
#include <GL/gl.h>                    //  OpenGL standard
```

Les fonctions d'OpenGL sont préfixées avec “gl”:

```
gl<Function>{<1234>}{<bsifd...>}{<v>}(T arg1, T  arg2,...);
Exemple: glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

Les constantes OpenGL sont préfixées avec “GL_”:

```
GL_<UNE_CONSTANTE>
Exemple: GL_TRIANGLES
```

Les types OpenGL sont préfixés avec “GL”:

```
GL<type>
Exemple: GLfloat
```

OpenGL : Evolution de 2 à 4

2.1 (Août 2006)	3.0 (Août 2008) puis 3.1, 3.2	4.0
FF	Deprecated Features and Non-FWD-CC	Compatibility- Profile

“fixed function” (FF)
GLSL-Shaders
Mixer FF et shaders possible, mais confus et lourd
Supporté par tous les drivers

Beaucoup de FF«deprecated »

Contextes puis profils permettant la compatibilité et des extensions

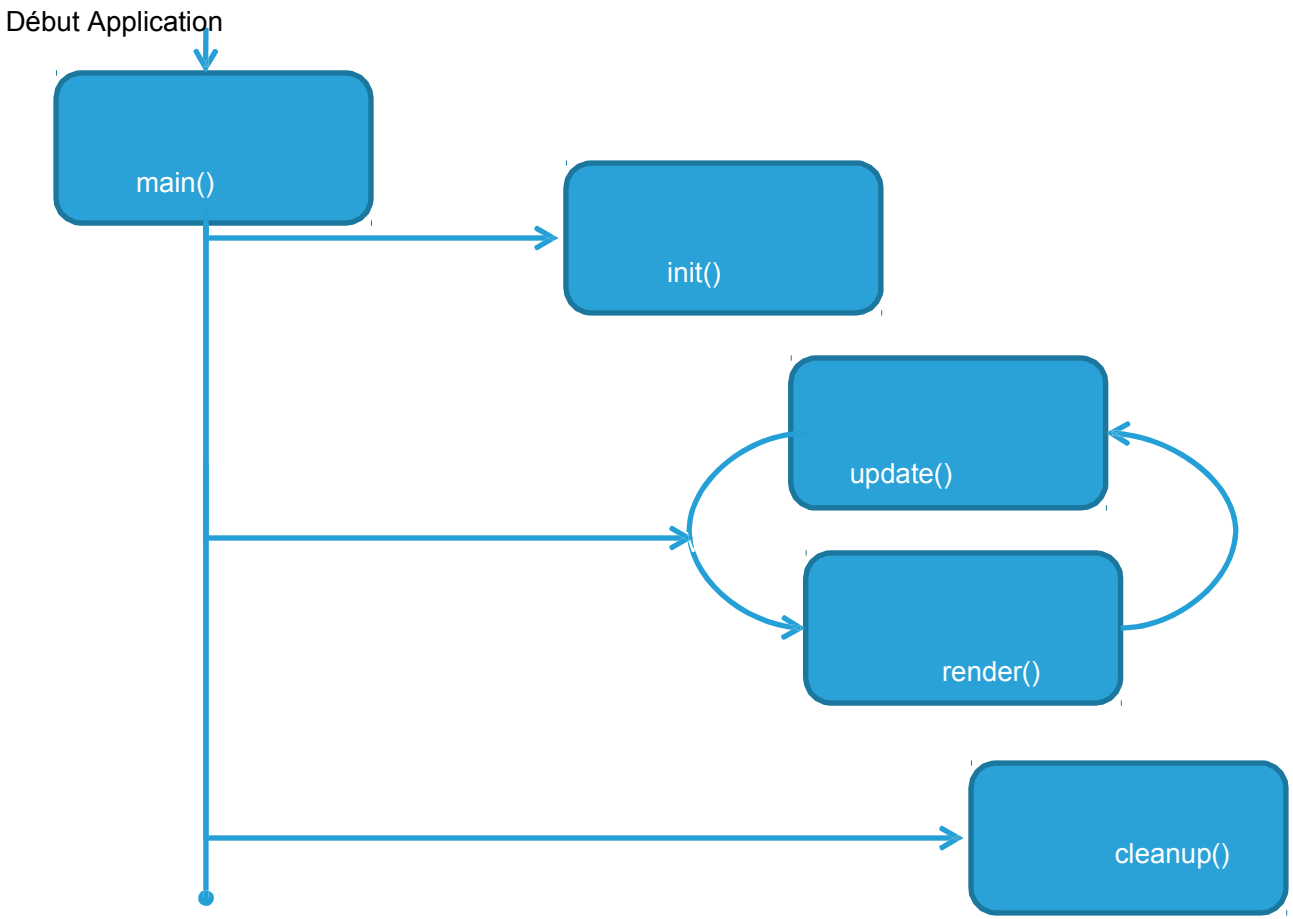
De GLSL 1.5 à GLSL 3.3
Supporté par presque tous les drivers.

Améliorations des contextes et profils permettant la compatibilité et des extensions

Note : depuis OpenGL 3.x il n'y a plus:
De piles de matrices et de transformations
De 'Lighting' et 'Materials', ...
A faire soi-même => shader.

Squelette de programme OpenGL

Application OpenGL classique :

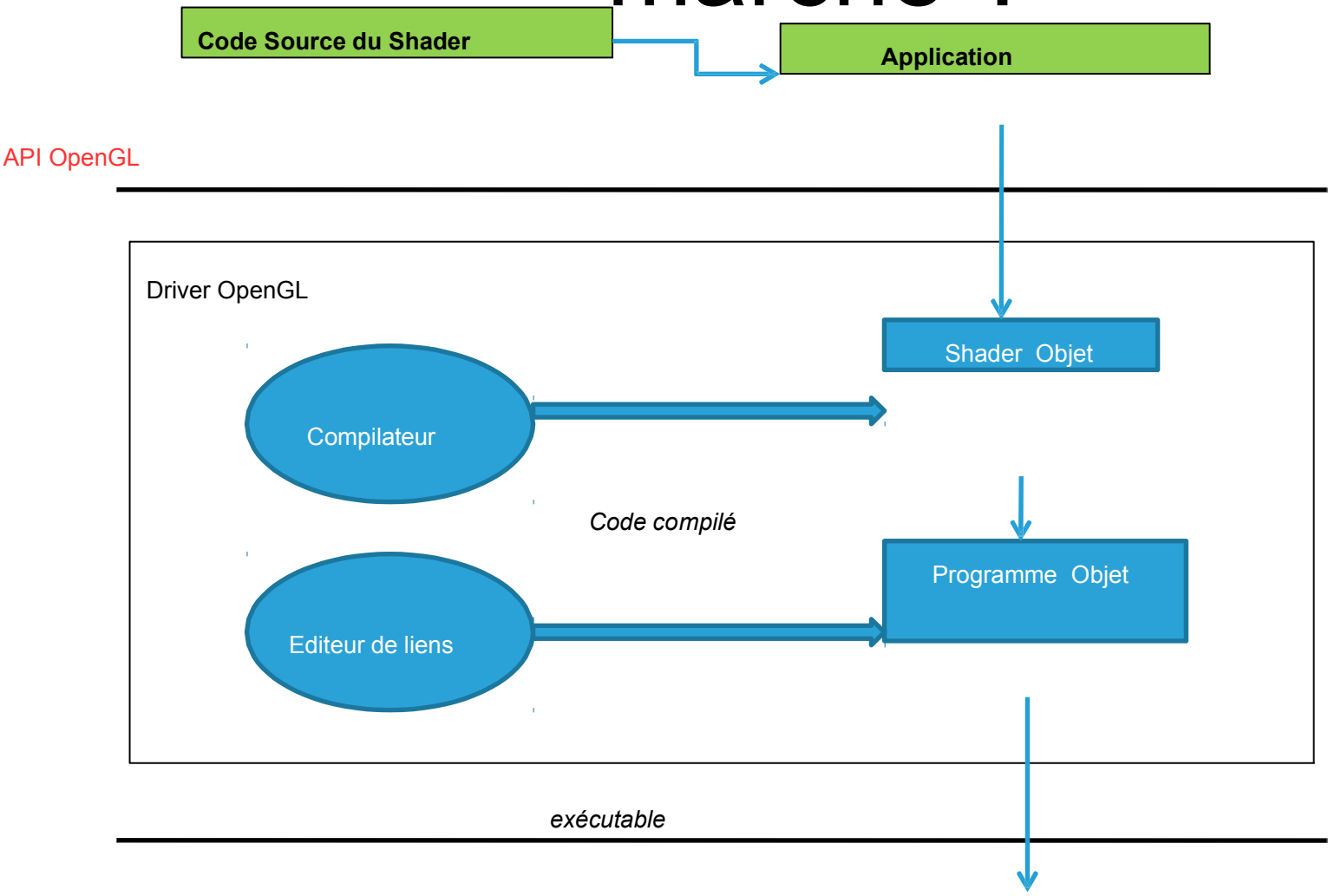


SHADERS : Qu'est ce que c'est ?

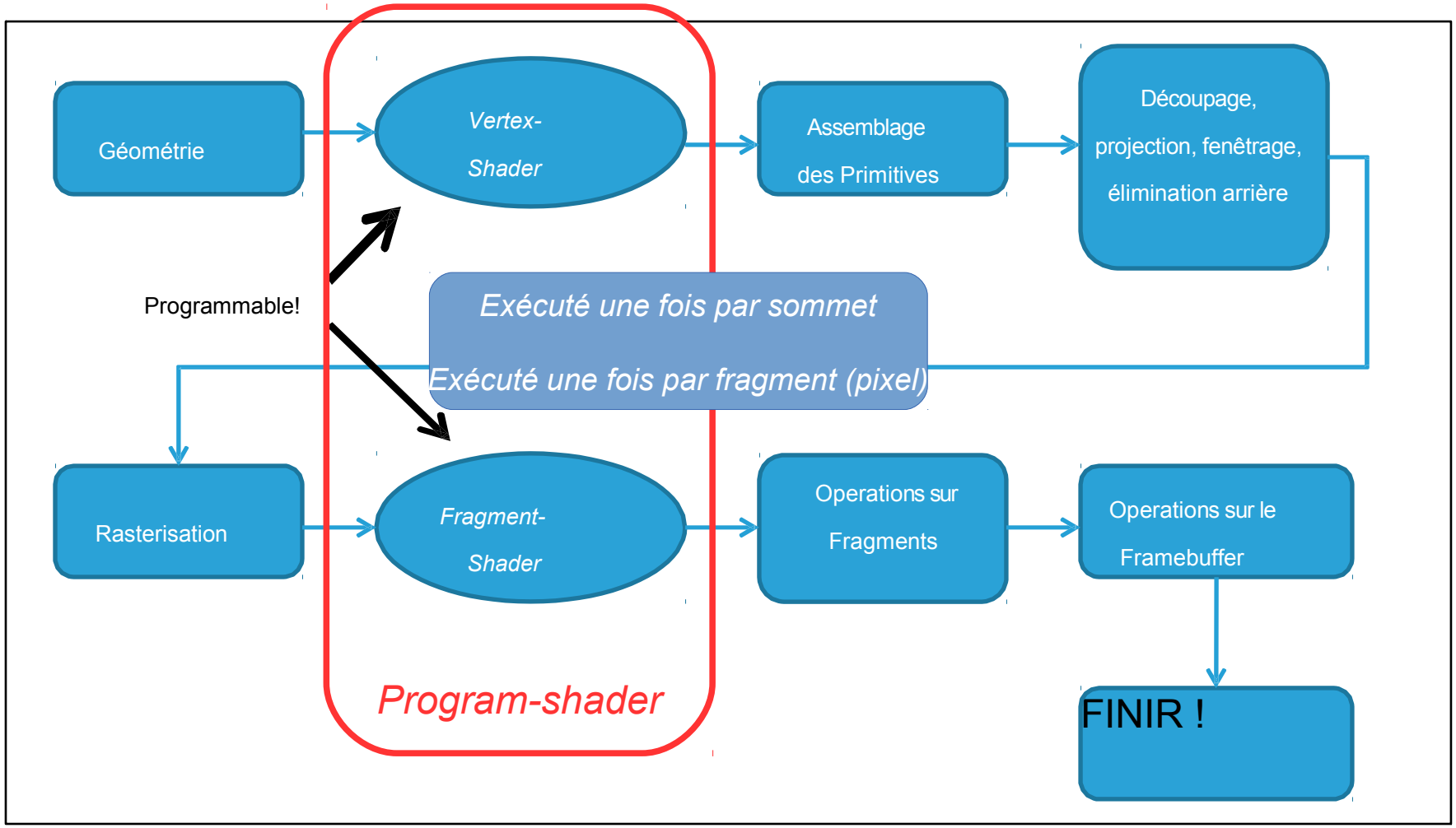
- Programme (du genre C, C+...)exécuté sur la carte graphique
- Remplace le pipeline de fonctions fixées par des shaders
- Types de Shader
  - Vertex Shader (VS): opérations sur sommets
  - Geometry Shader (GS): opérations sur primitives
  - Fragment shader (FS): opérations sur fragments (pixels)

Utilisations principales : transformations et rendu

Comment ça marche ?

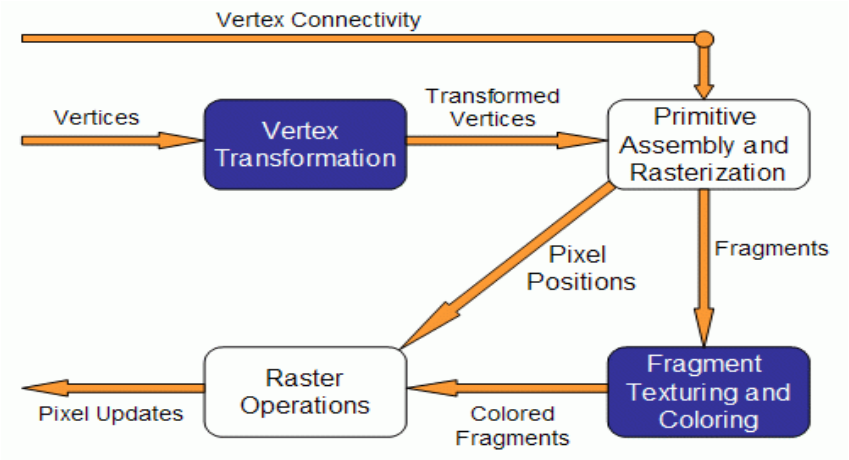


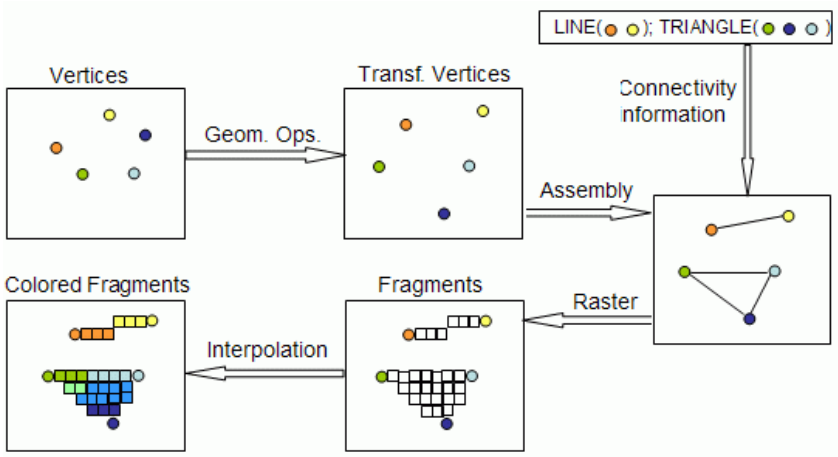
Pipeline de rendu



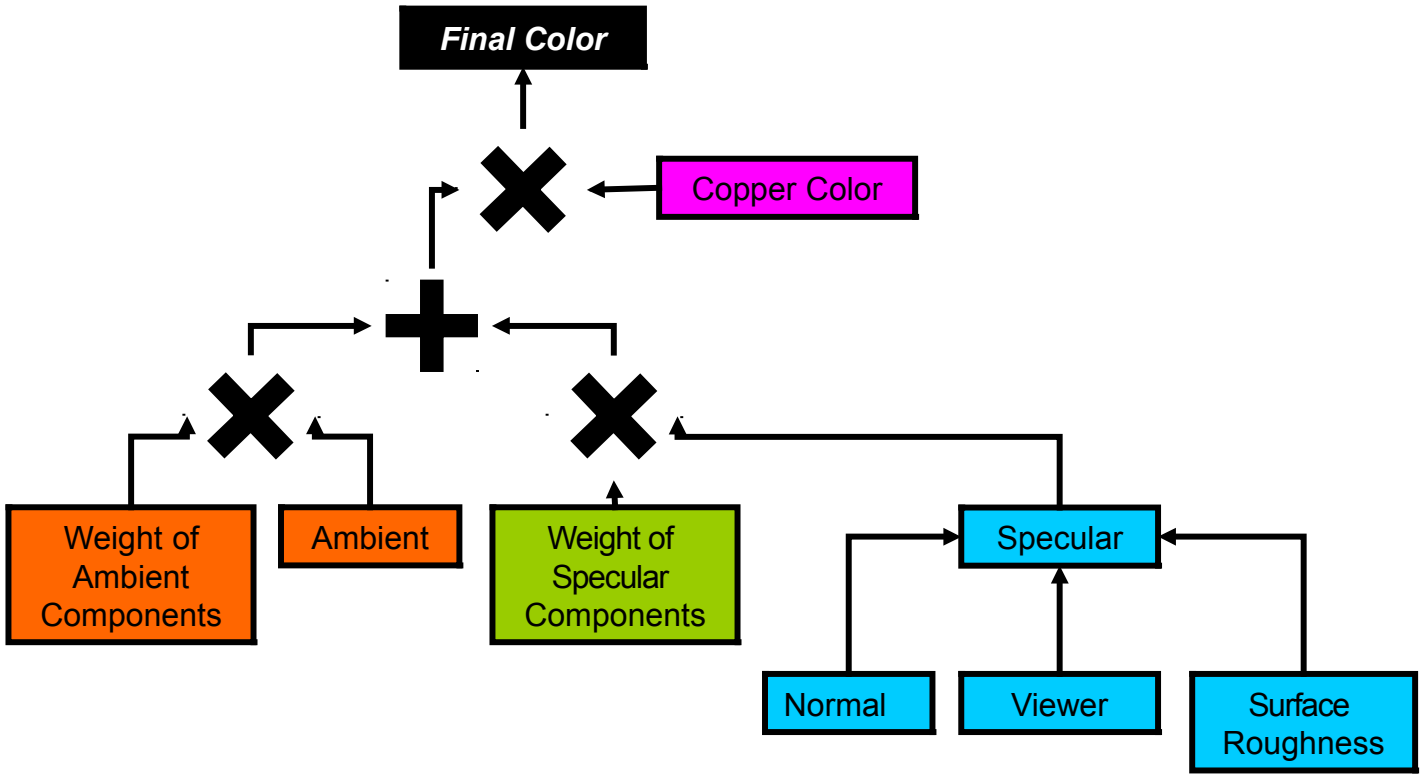
OpenGL 3.x

Comment on en est arrivé là ?





1984 : «Shade Trees» par Rob Cook (Lucasfilm Ltd.)



- **Chaque *feuille* de l'arbre représente une valeur.**
  (couleur, vecteur, coefficient, …)
- **Chaque *nœud* de l'arbre représente une opération.**
  (addition, multiplication, produit scalaire, …)

Premiers « Shading Languages »

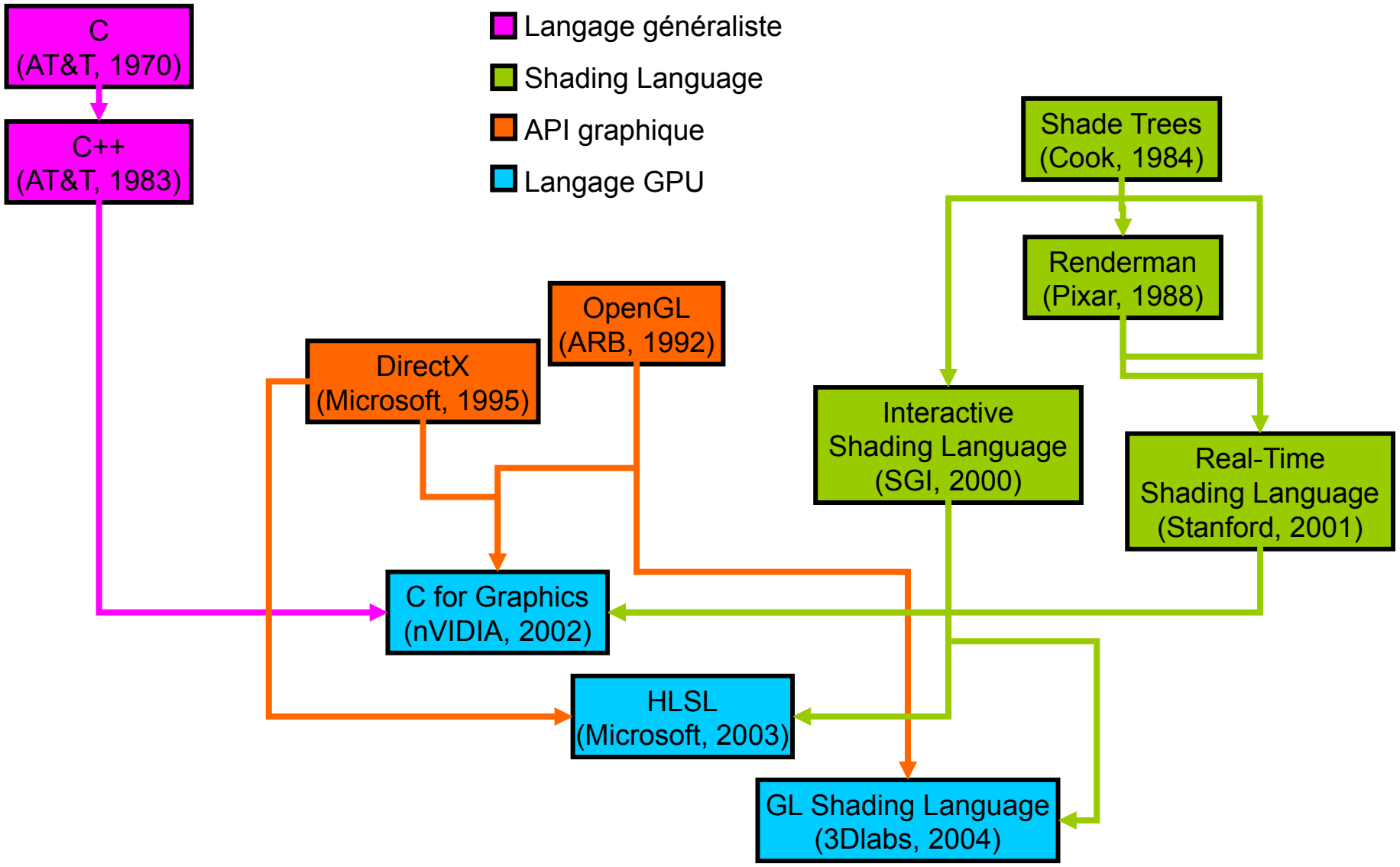
Plus de souplesse dans la description des interactions Lumière/Ma
Prise en compte des aspects «dynamiques» (textures procédurales)

1988 : «Renderman» par Pat Hanrahan (Pixar)

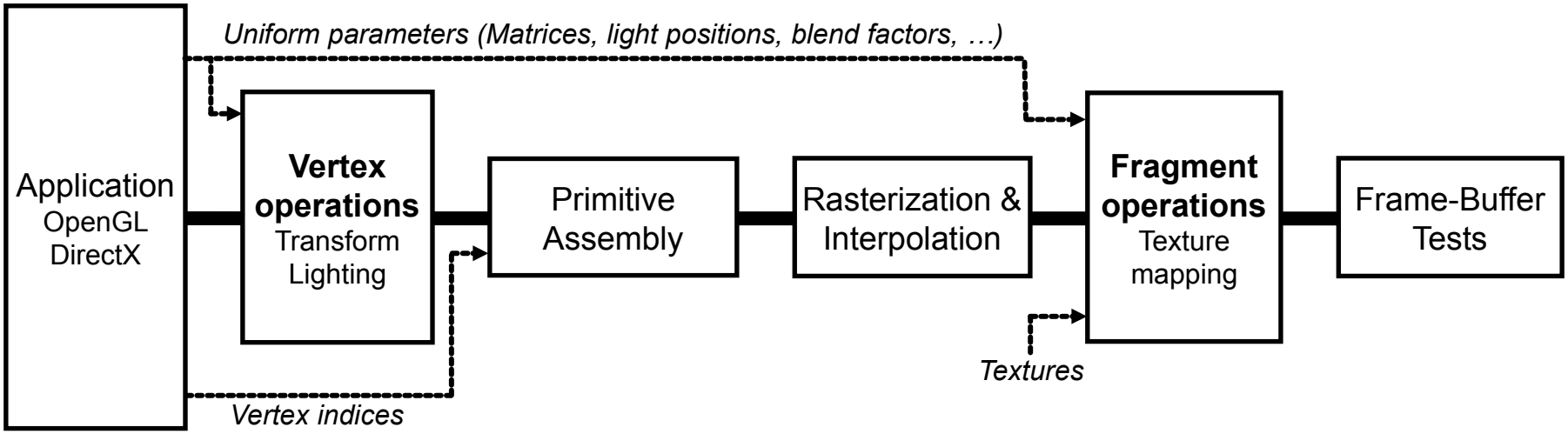
2000 : «Interactive Shading Language» (SGI)

2001 : «Stanford Real-Time Shading Language»

Arbre généalogique des « Shading Languages » (d'après Olivier Nucent)

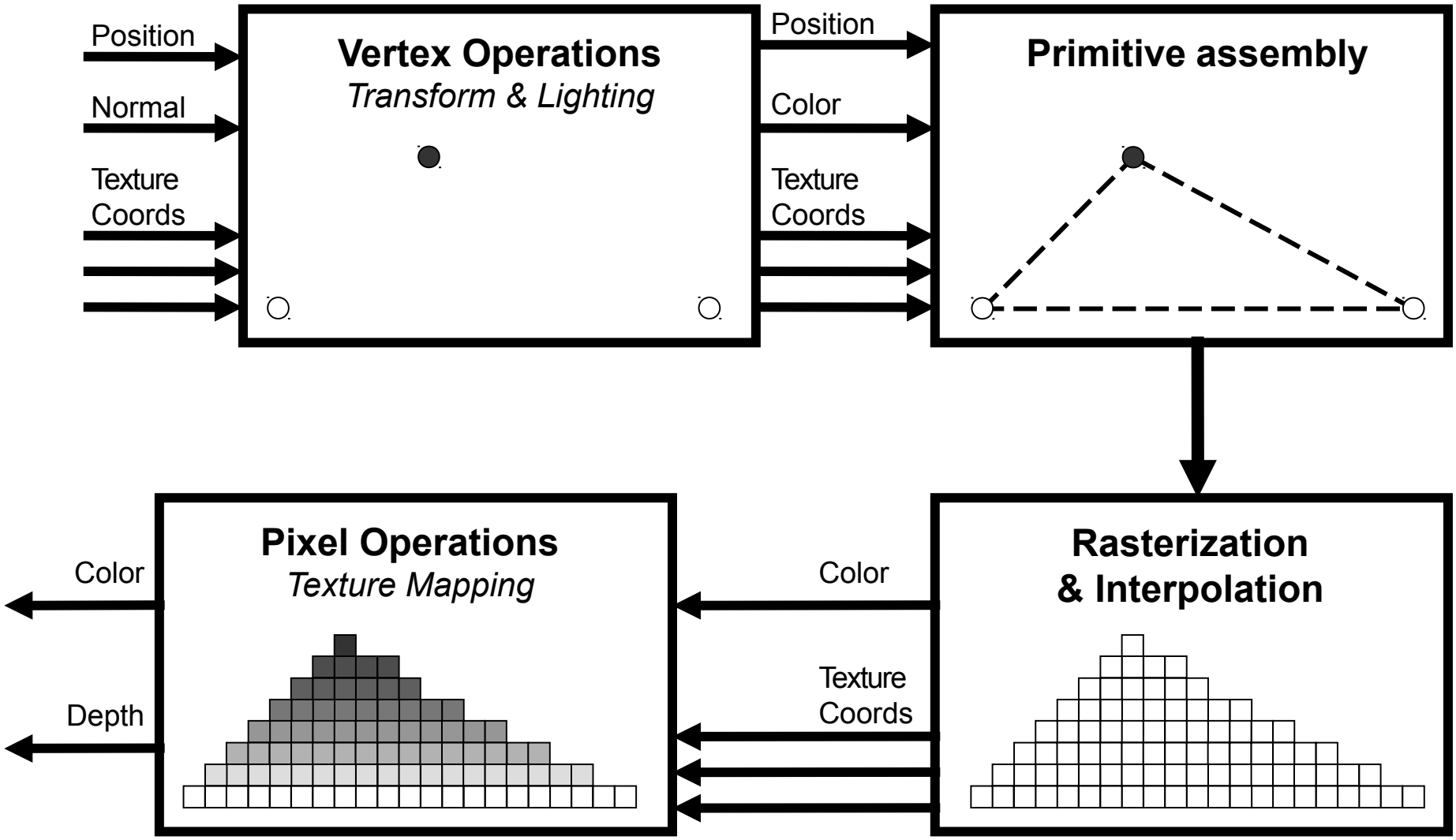


Graphics Pipeline

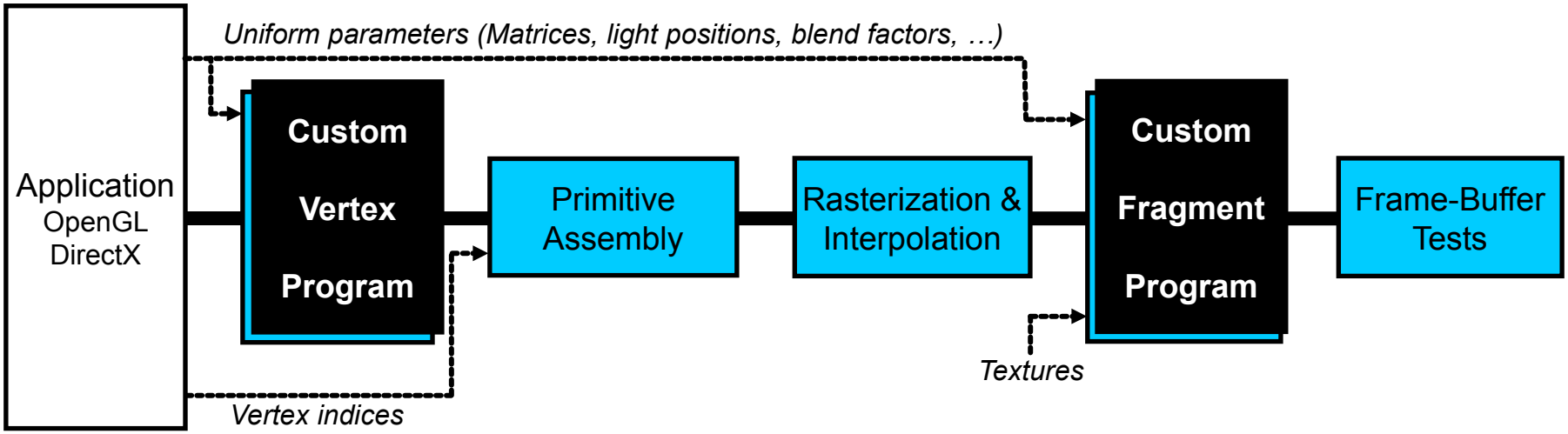


Graphics pipeline **enchaînement d'opérations appliquées aux donne**
géométriques afin de produire l'affichage d'une scène 3D.

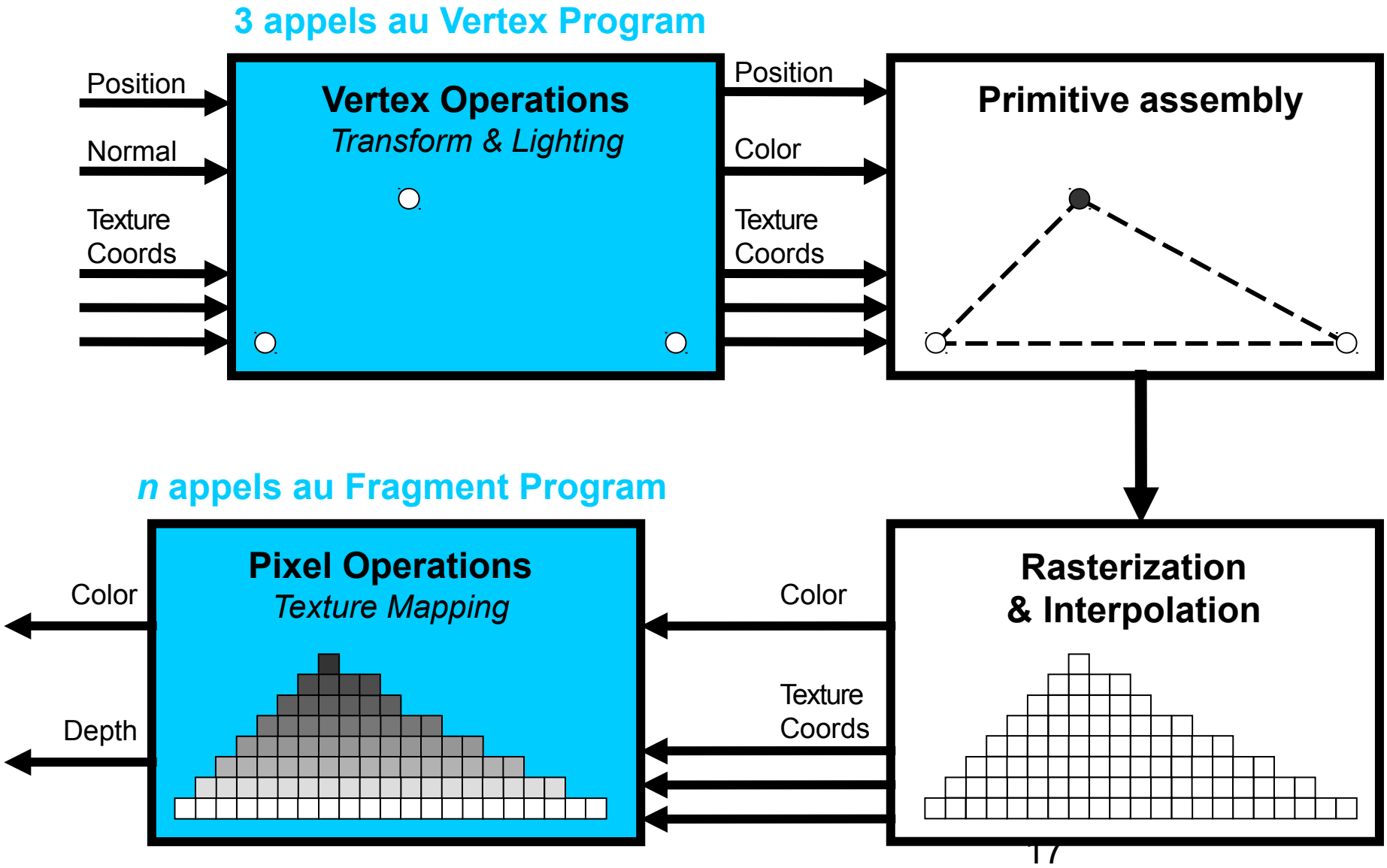
Graphics Pipeline



Gérer les sommets et fragments en direct



Graphic Pipeline



Quelques chiffres…

Génération	Année	Gamme	Transistors	Fill rate	Polygon rate
1ère	**Fin 1998**	**RIVA TNT**	**7 M**	**50 M/s**	**6 M/s**
1ère	**Début 1999**	**RIVA TNT2**	**9 M**	**75 M/s**	**9 M/s**
2ème	**Fin 1999**	**GeForce 256**	**23 M**	**120 M/s**	**15 M/s**
2ème	**Début 2000**	**GeForce 2**	**25 M**	**200 M/s**	**25 M/s**
3ème	**Début 2001**	**GeForce 3**	**57 M**	**800 M/s**	**30 M/s**
3ème	**Début 2002**	**GeForce 4 Ti**	**63 M**	**1200 M/s**	**60 M/s**
4ème	**Début 2003**	**GeForce FX**	**125 M**	**2000 M/s**	**200 M/s**

NB: le microprocesseur Pentium 4 d'Intel contient 55 M de transistors.

	Radeon R9 295X2	Radeon R9 290X	GeForce GTX Titan	GeForce GTX Titan X
Gravure	28 nm	28 nm	28 nm	28 nm
Nb. Transistors	2 x 6,2 milliards	6,2 milliards	7,1 milliards	8 milliards
Fréquence GPU	max. 1018 MHz	max. 1 GHz	837 MHz	1000 MHz
Nb. shaders	2 x 2816	2816	2688	3072
Puissance FP32	max. 11,5 TFLOPS	5,6 TFLOPS	4,5 TFLOPS	192
Unités de texture	2 x 176	176	224	192 GT/s
Débit texels	max. 358,3 GT/s	176 GT/s	188 GT/s	96
ROPs	2 x 64	64	48	96 GP/s
Débit pixels	max. 130,3 GP/s	64 GP/s	40 GP/s	384 bits
Bus mémoire	2 x 512 bits	512 bits	384 bits	12 Go de GDDR5
Quantité mém.	2 x 4 Go GDDR5	4 Go GDDR5	6 Go GDDR5	7 GT/s
Bande passante	2 x 320 Go/s	320 Go/s	288 Go/s	336,5 GB/s
TDP carte	500 W	250 W	250 W	250 W

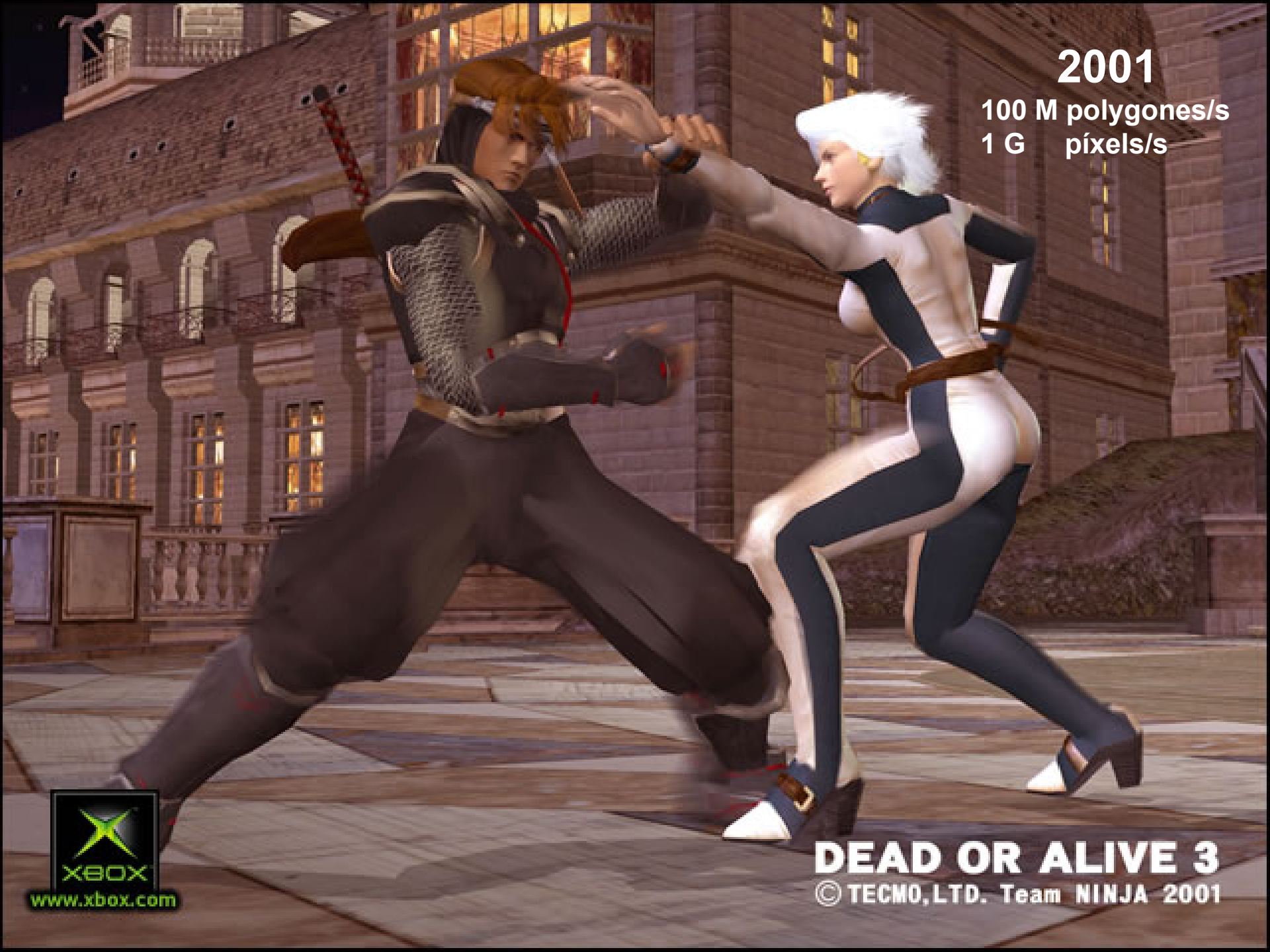
Nom du produit	ntel® Core™ i7-6700HQ ocessor (6M Cache, up to 3.50 GHz)
Nom du code	Skylake

1 à 2 Mds

Performances	
Nb. de cœurs	4
Nb. de threads	8
Fréquence de base	2.6 GHz
Fréquence Turbo maxi	3.5 GHz
PDT	45 W
PDT-down configurable	35 W
Spécifications de la mémoire	
Capacité mémoire maxi (dépend du type de mémoire)	64 GB
Types de mémoire	DR4-2133, LPDDR3-1866 DDR3L-1600
Nb. max. de canaux mémoire	2
Bande passante mémoire maxi	34,1 GB/s

Souvenirs,
souvenirs …


1995
SEGA Virtua Fighter
50.000 polygones/s
1 M píxels/s


2001
100 M polygones/s
1 G píxels/s
XBOX
www.xbox.com
DEAD OR ALIVE 3
© TECMO, LTD. Team NINJA 2001

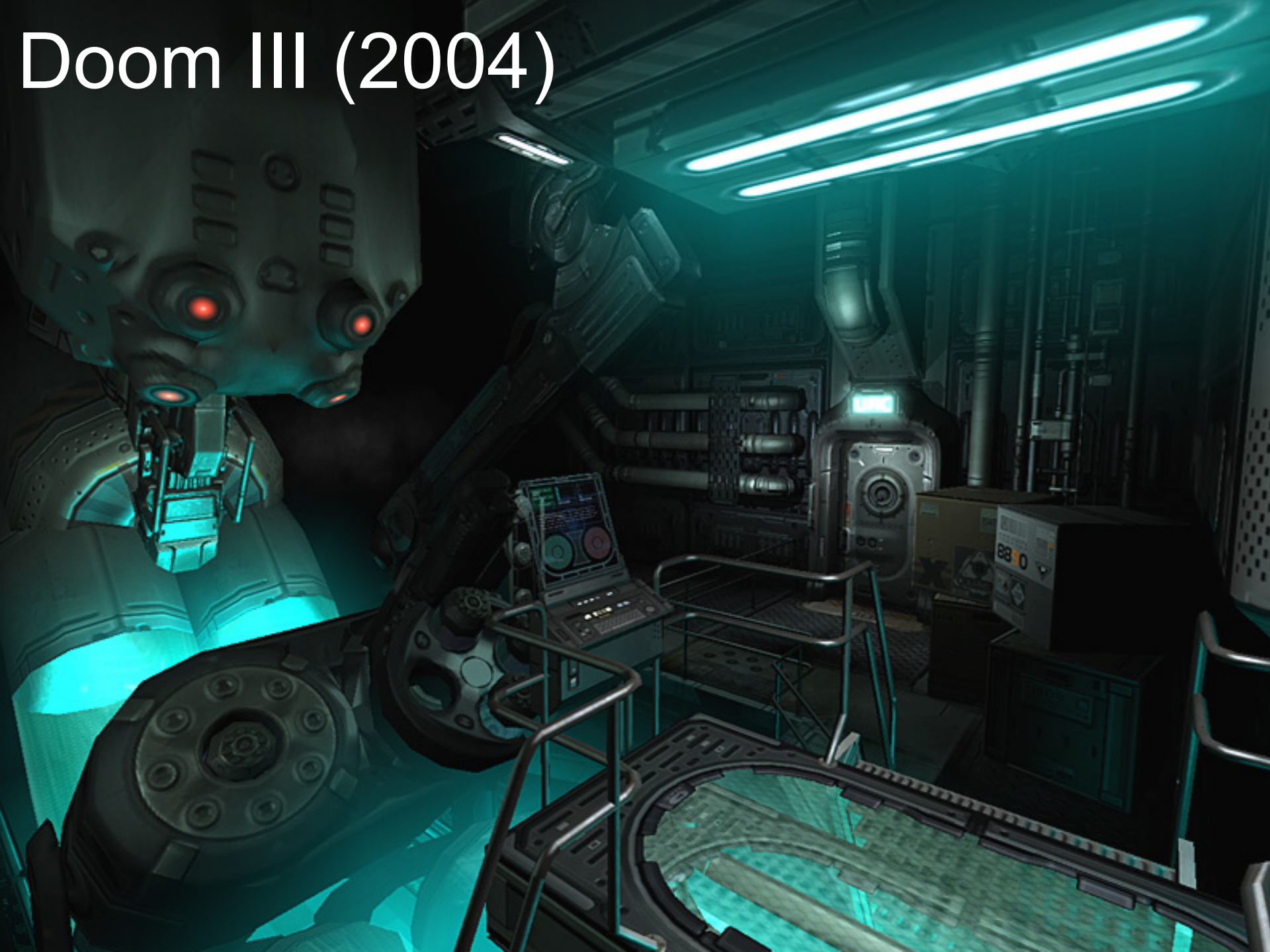
Doom (1993)

Doom II (1994)


10
AMMO
107%
HEALTH
2 3 4
5 6 7
ARMS
95%
ARMOR
BULL 160 / 200
SHEL 10 / 50
ROKT 0 / 50
CELL 0 / 300

Doom III (2004)

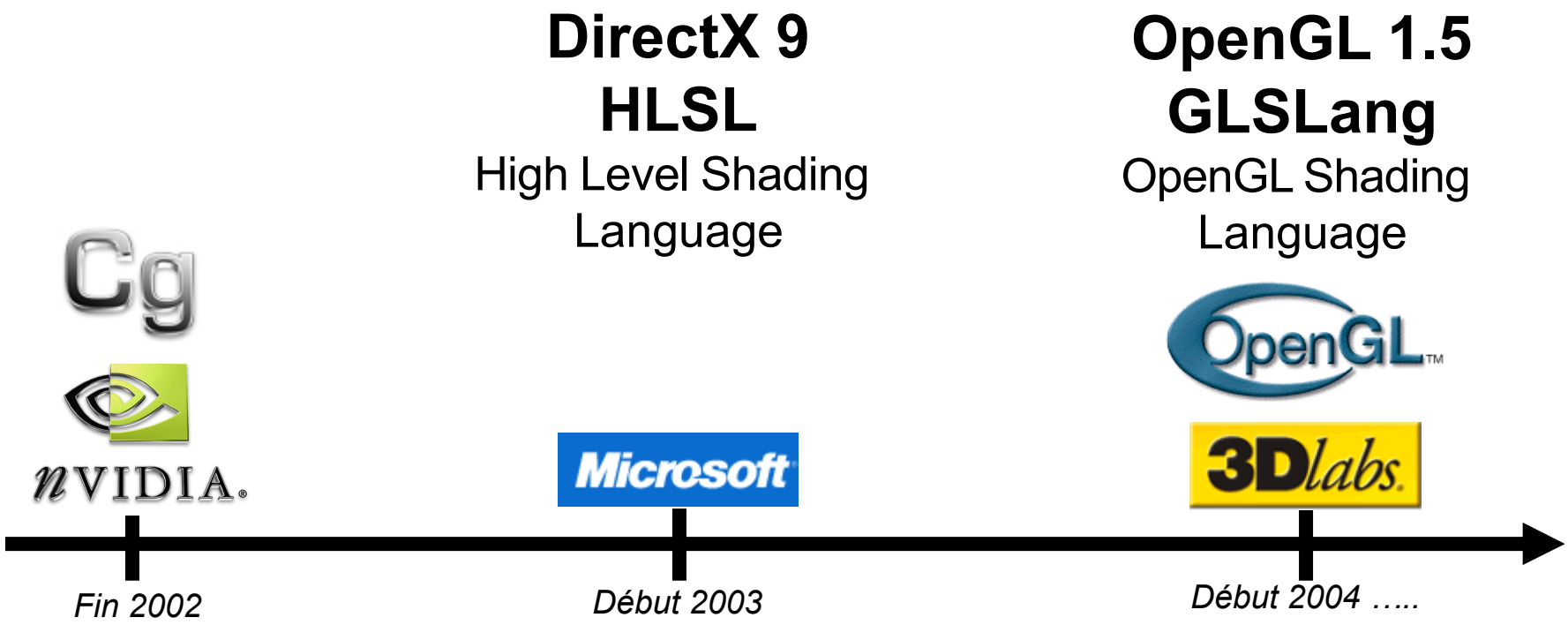
Doom III (2004)



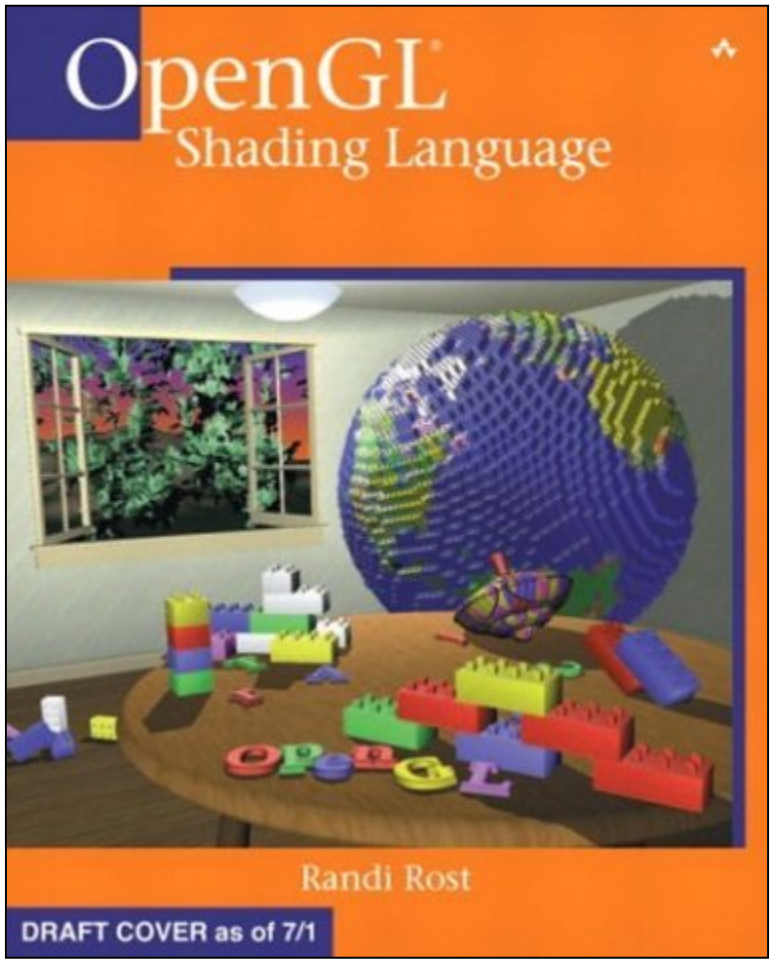
Des images de game plays

Vers Doom4 en ???

Cg et GLSL : Chronologie



Références bibliographiques



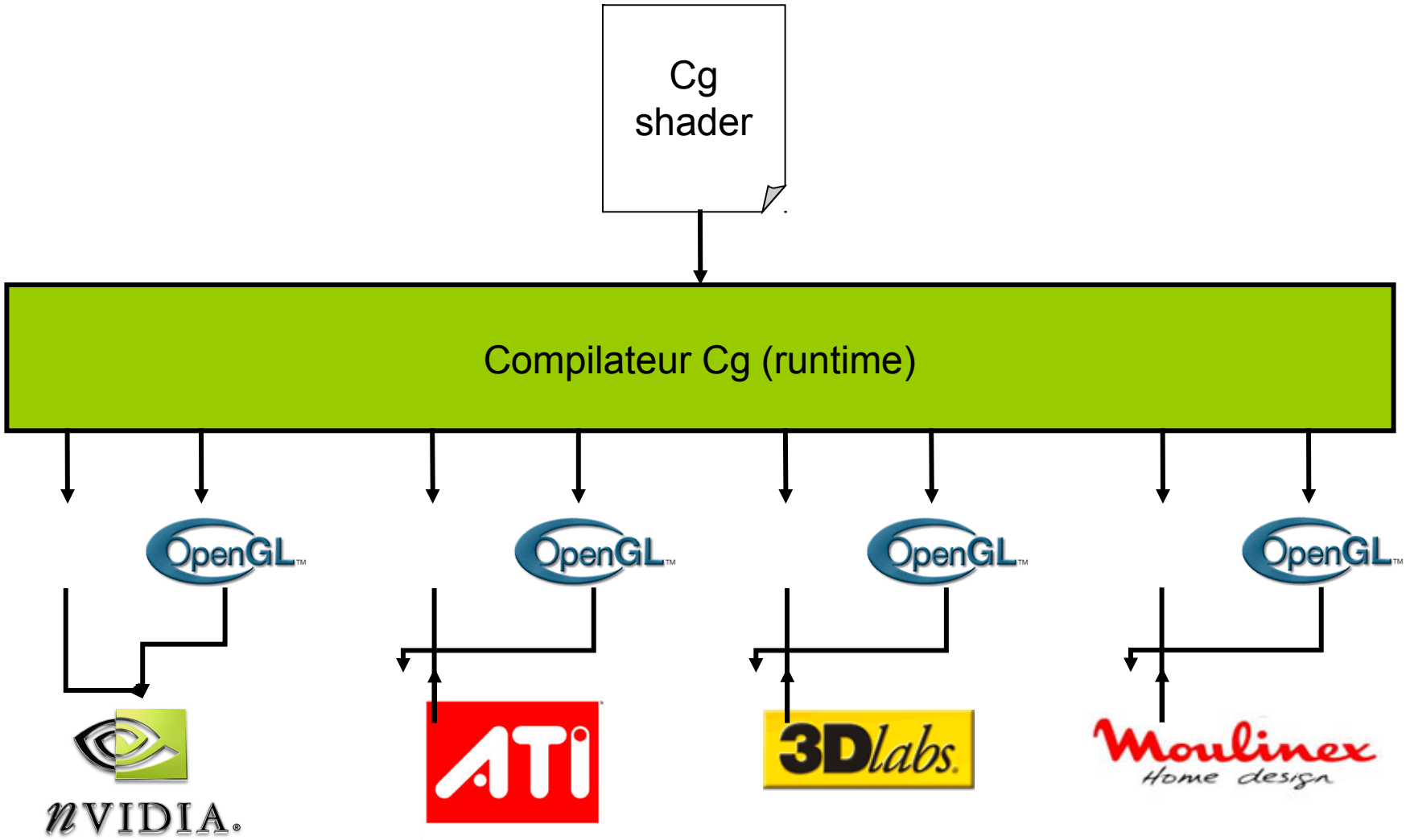
OpenGL(R) Shading Language

GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics

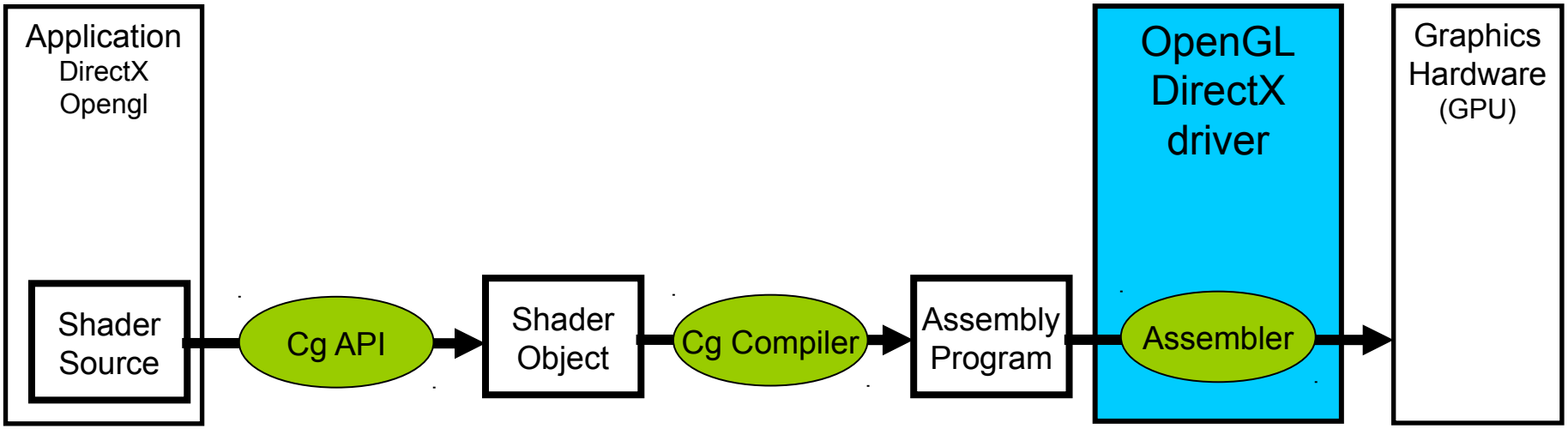
GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation

Et le ouaib

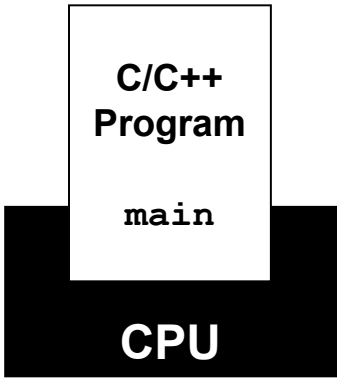
Cg : principe général

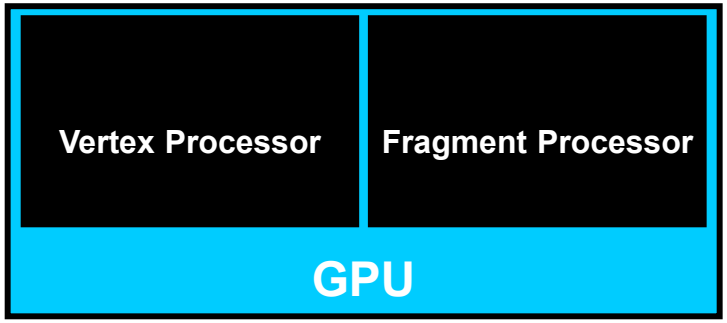


Cg : du code source au code exécutable

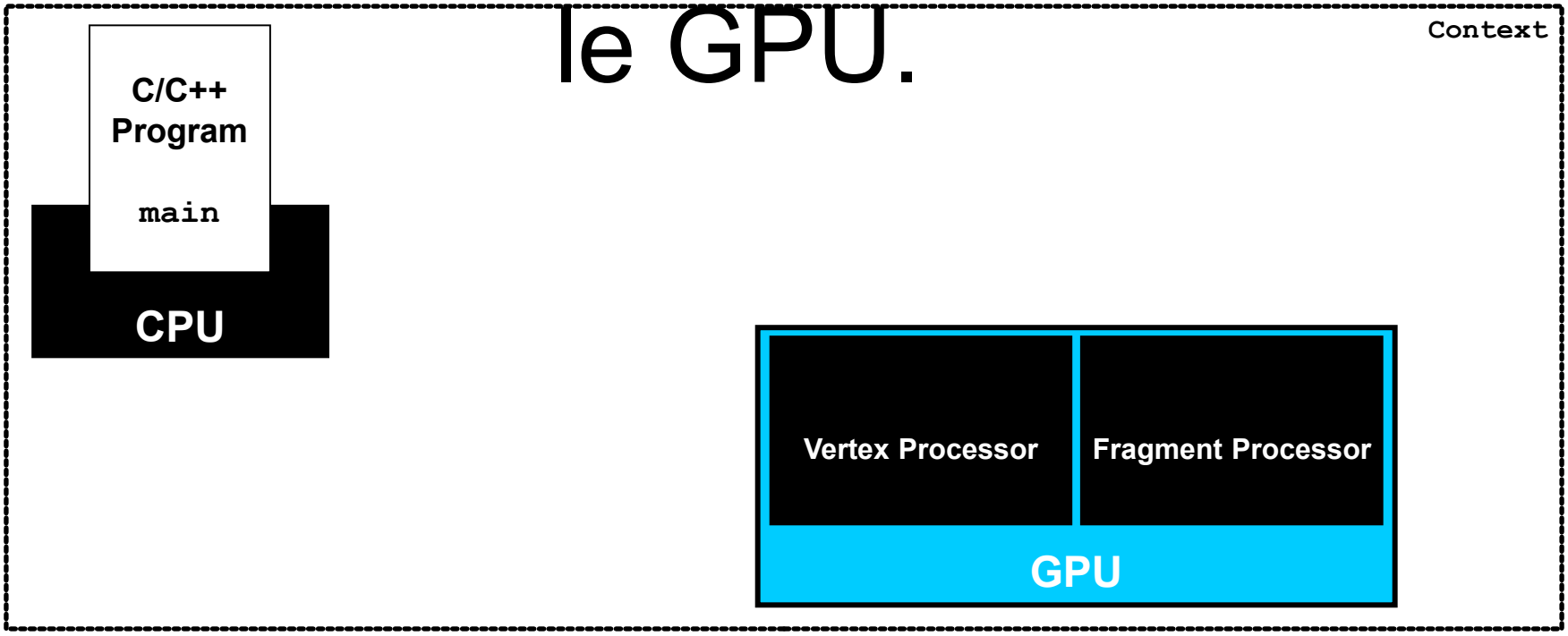


Acte I : présentation des acteurs.



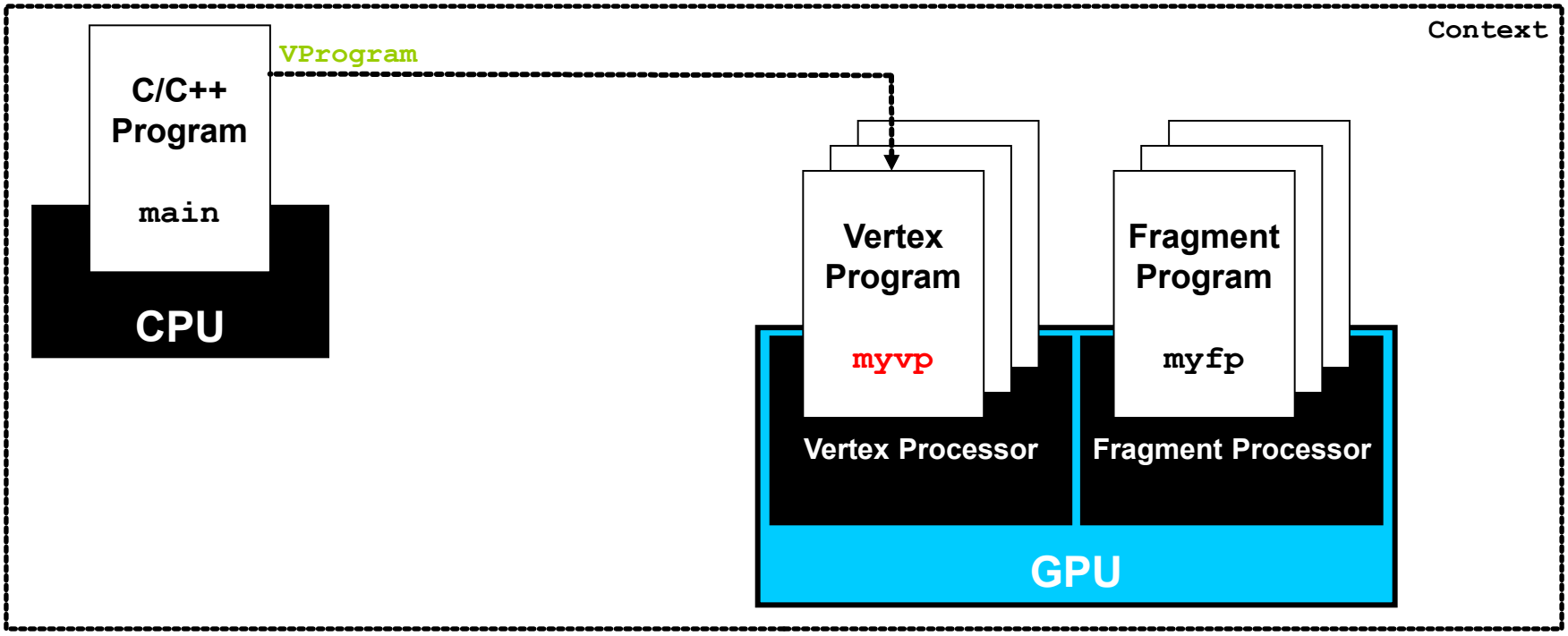


Acte II : contexte de communication entre le CPU et le GPU.



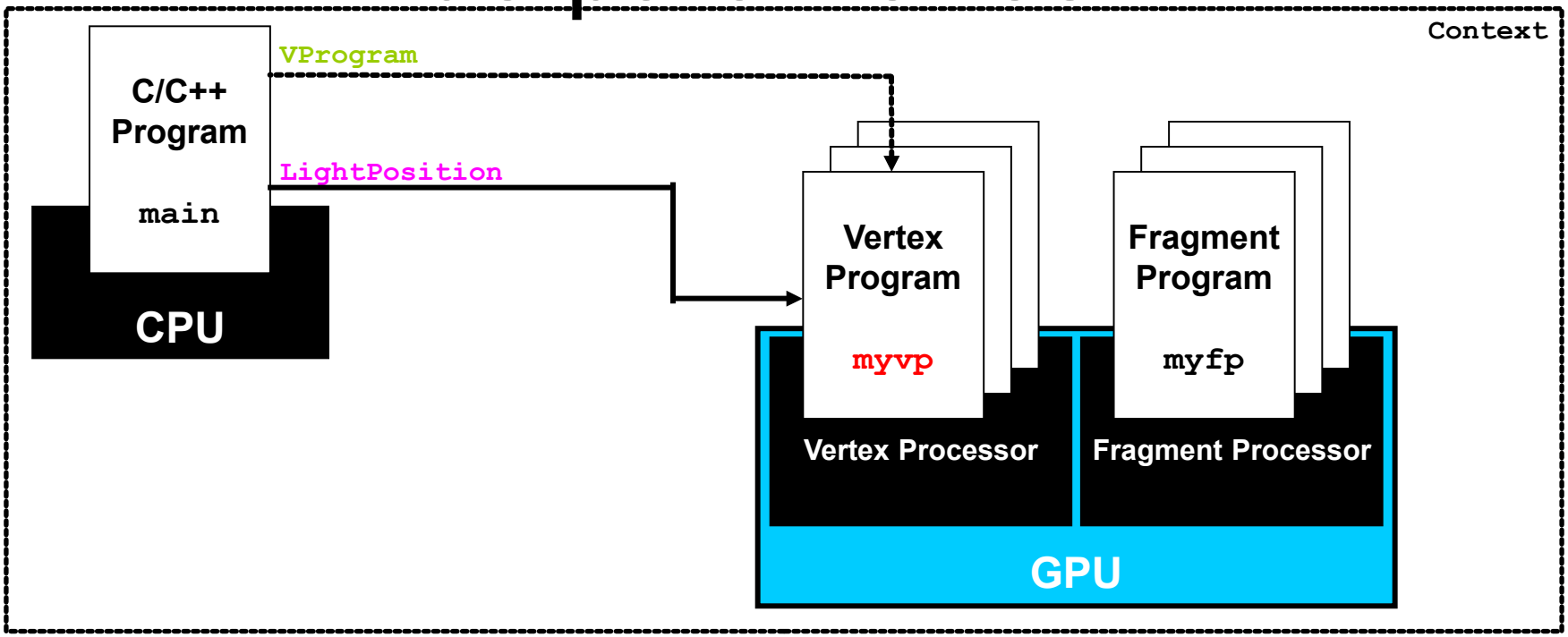

```
CGcontext* Context = cgCreateContext();
```

Acte III : compilation et chargement du « shader ».



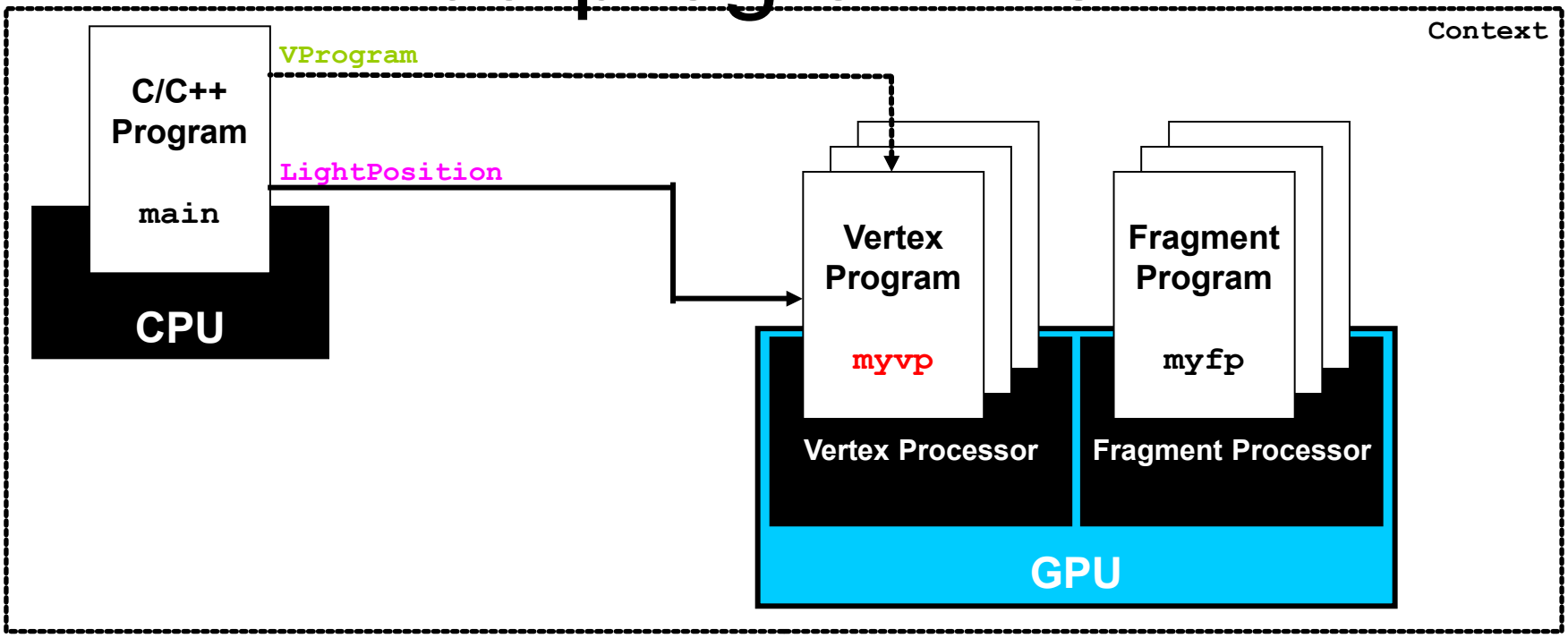

```
VProgram= cgCreateProgramFromFile(Context,CG_SOURCE,
          "pixel_true_phong.cg",CG_PROFILE_VP30,"myvp",NULL);

cgGLLoadProgram(VProgram);
```

Acte IV : association et passage de paramètres.



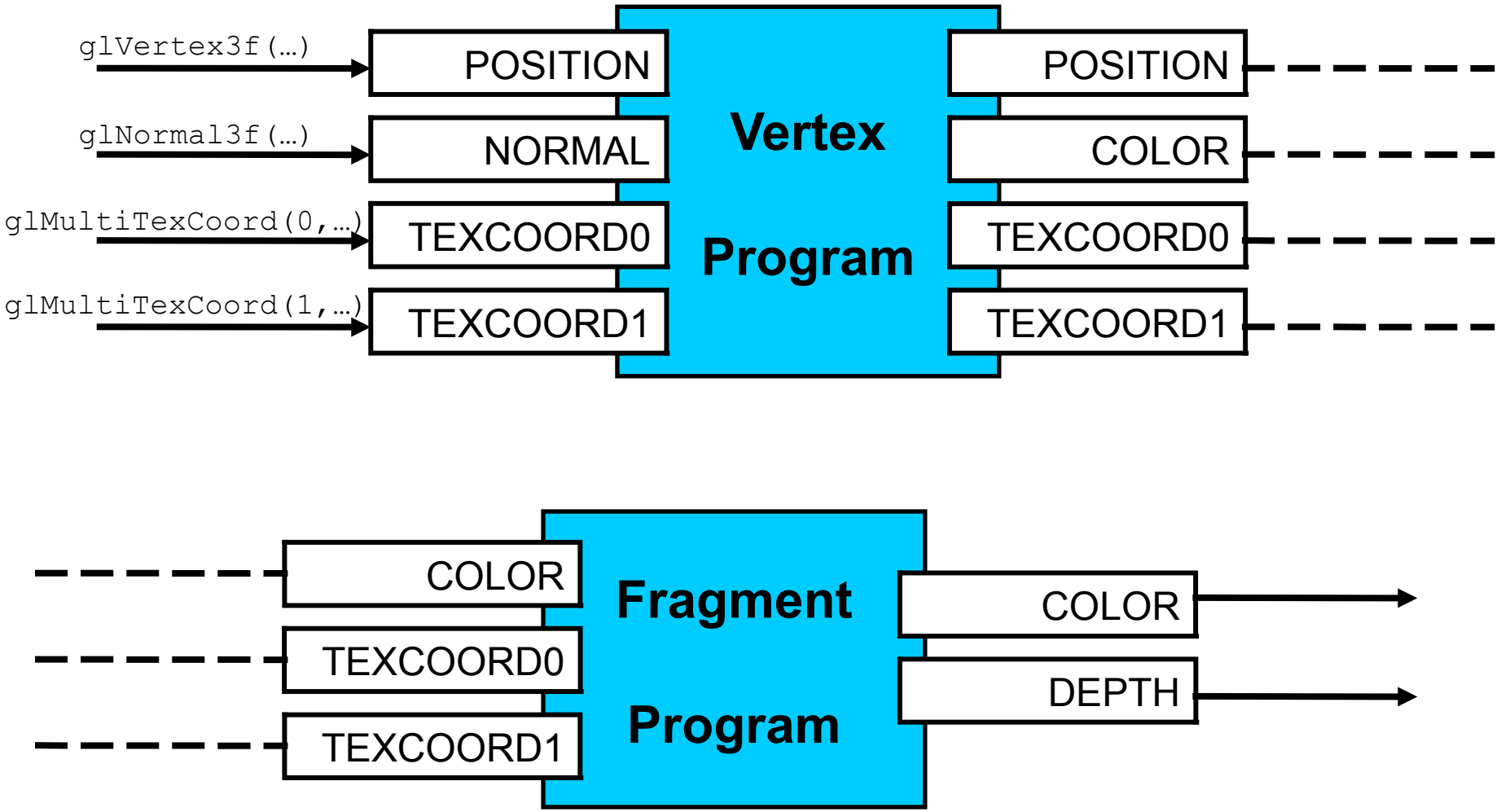

```
LightPositionBind = cgGetNamedParameter(VProgram, "LightPosition");

cgGLSetParameter4f(LightPositionBind, x, y, z, w);
```

Acte V : activation/désactivation du programme.

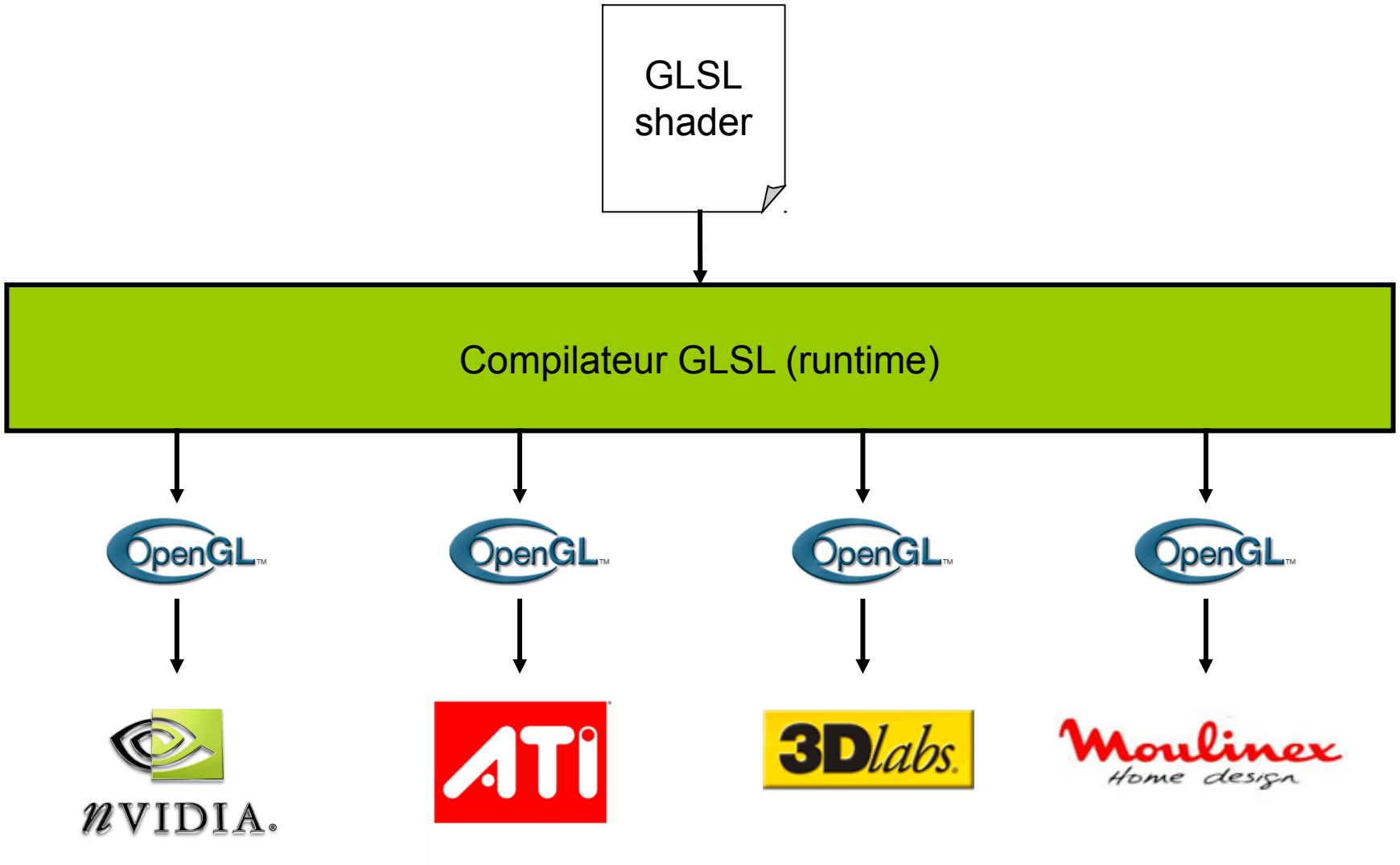



```
cgGLEnableProfile(CG_PROFILE_VP30);
cgGLBindProgram(VProgram);
...
cgGLDisableProfile(CG_PROFILE_VP30);
```

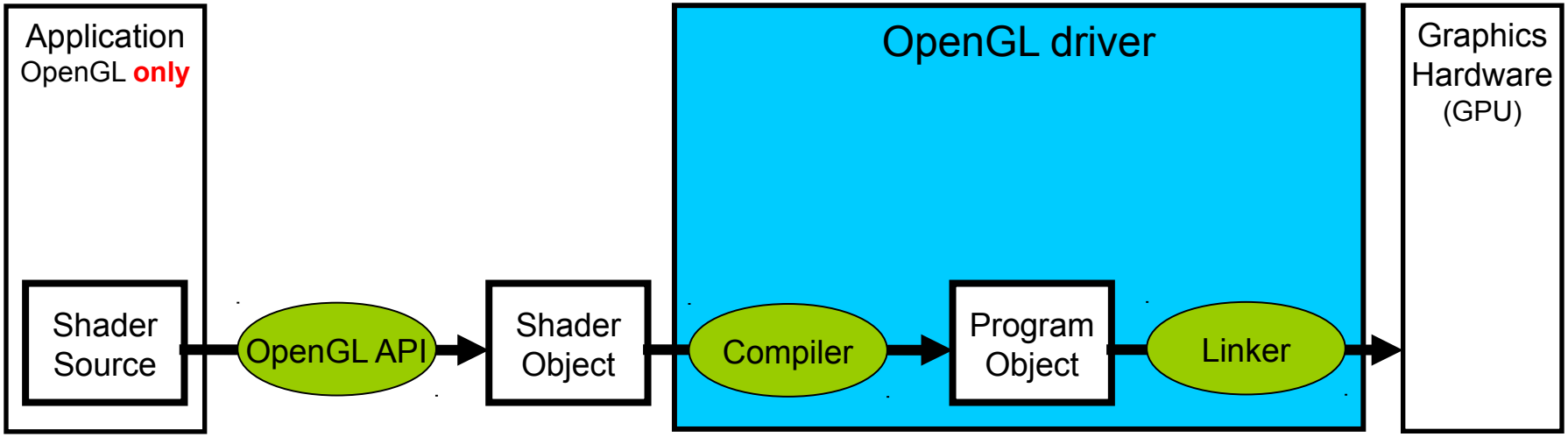
« Semantics »


glVertex3f(…)
glNormal3f(…)
glMultiTexCoord(0,…)
glMultiTexCoord(1,…)
POSITION
NORMAL
TEXCOORD0
TEXCOORD1
Vertex
Program
POSITION
COLOR
TEXCOORD0
TEXCOORD1
COLOR
TEXCOORD0
TEXCOORD1
Fragment
Program
COLOR
DEPTH

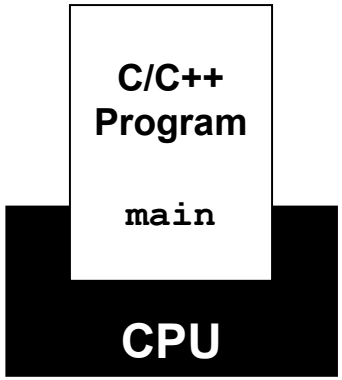
OpenGL Shading Language : principe général

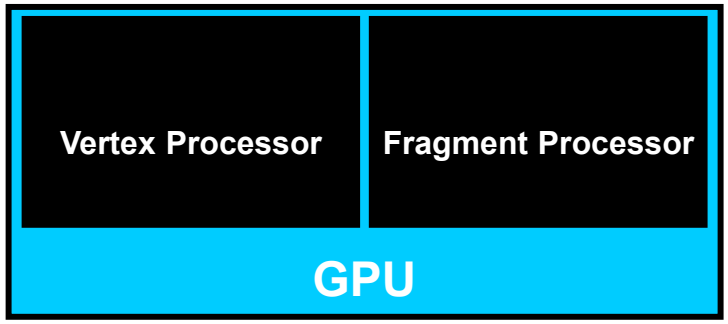


GLSL : du code source au code exécutable

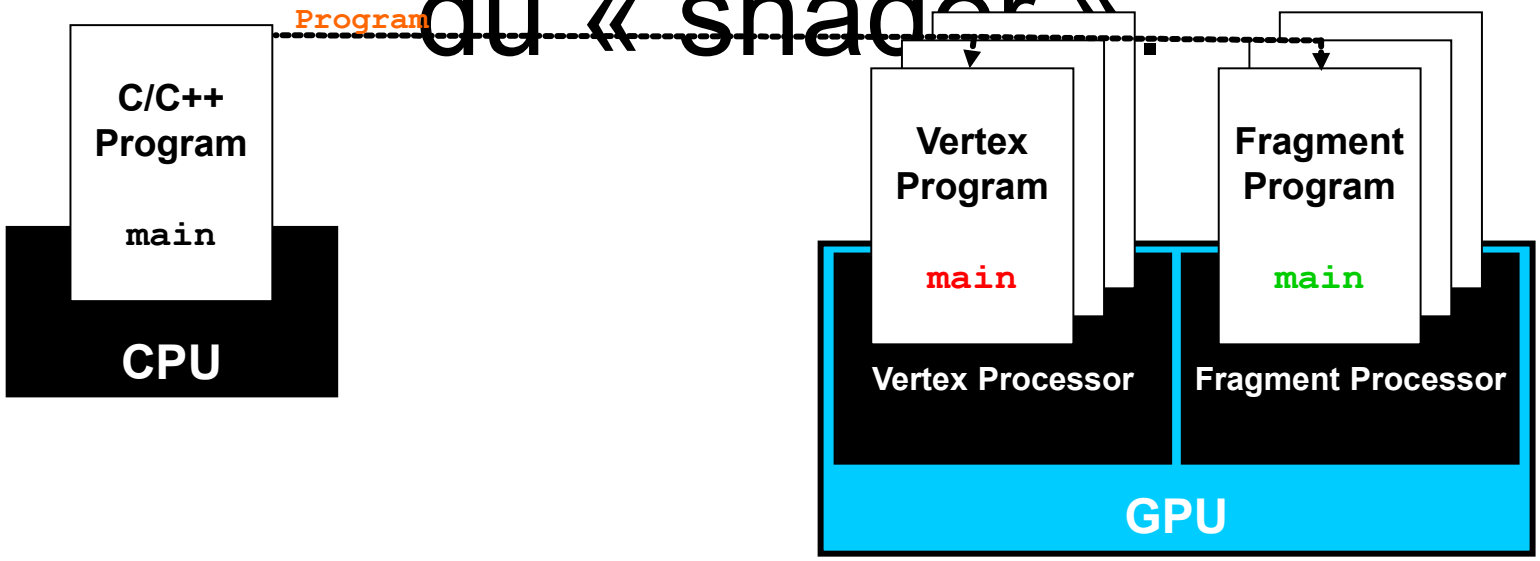


Acte I : présentation des acteurs.



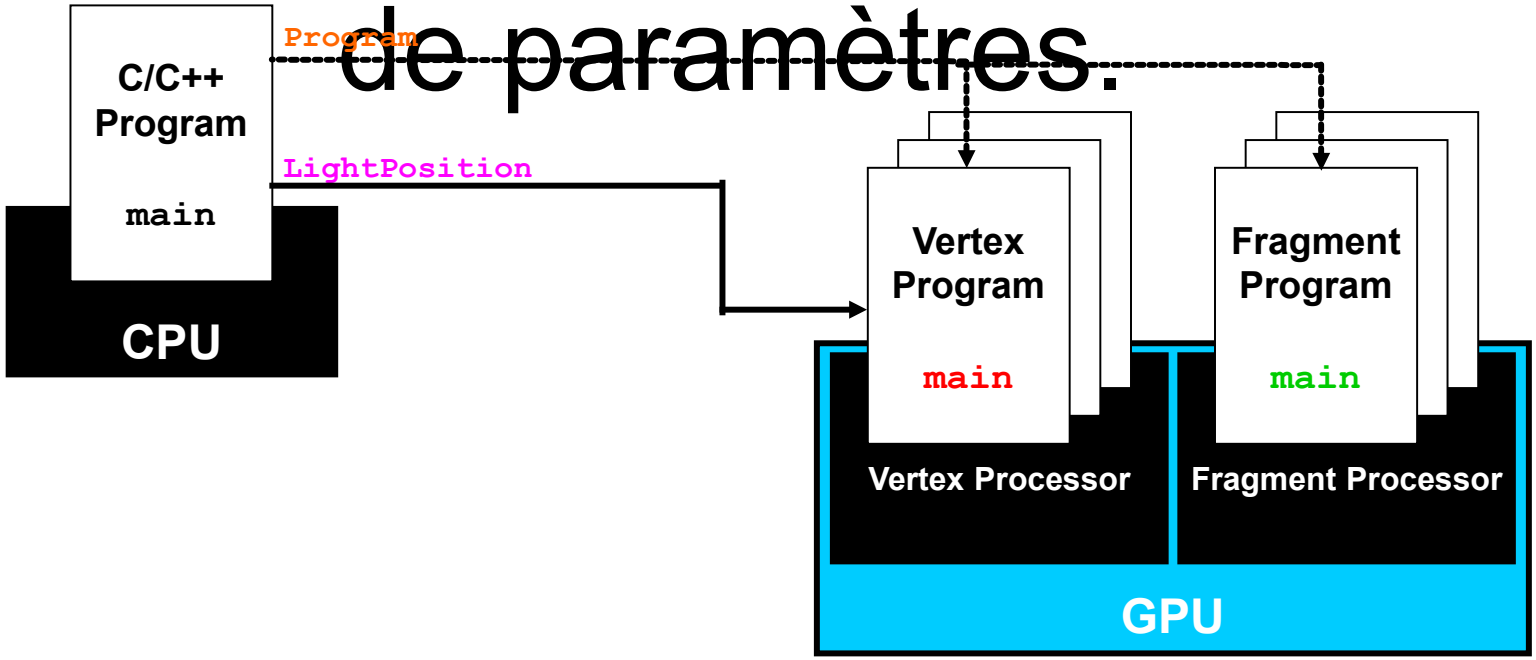


Acte II : compilation et chargement du « shader »



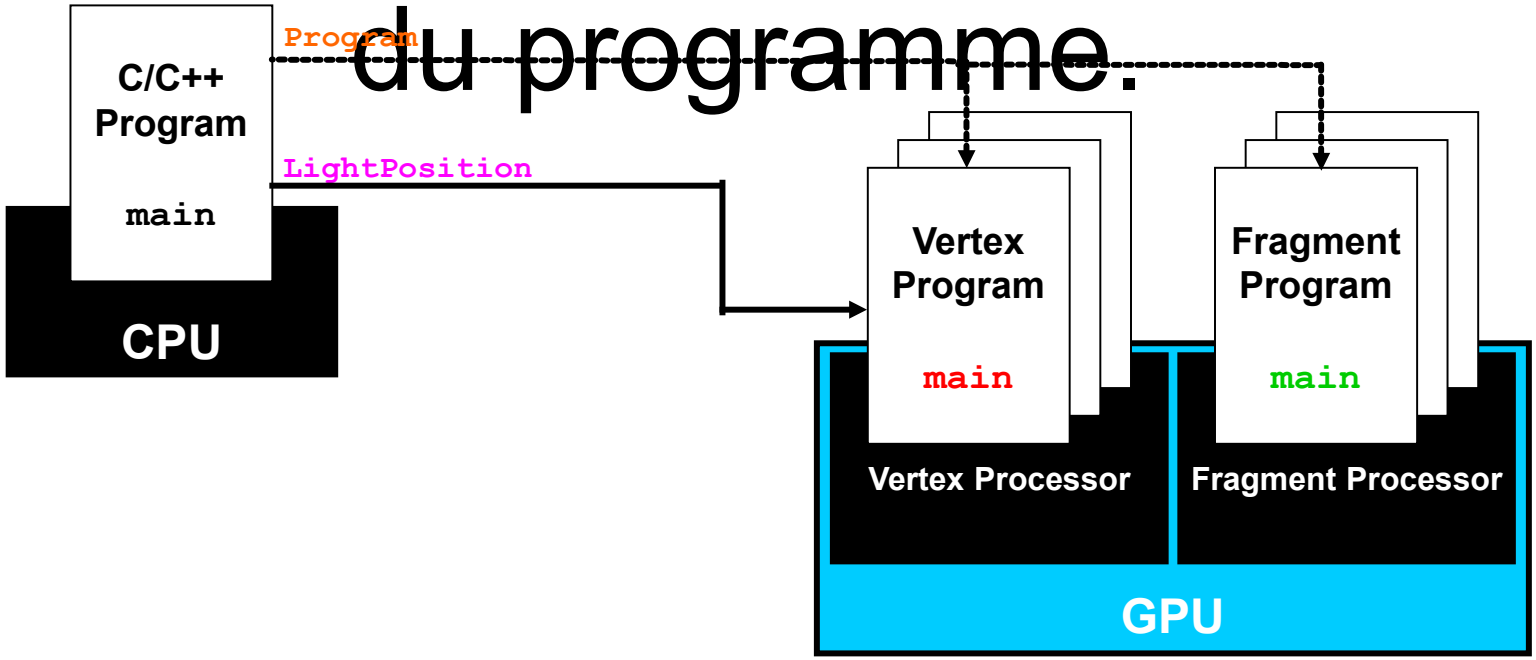

```
VShader = glCreateShaderObject(GL_VERTEX_SHADER);
glShaderSource(VShader, 1, (const GLchar**) &VSource, NULL);
glCompileShader(VShader);

FShader = glCreateShaderObject(GL_FRAGMENT_SHADER);
glShaderSource(FShader, 1, (const GLchar**) &FSource, NULL);
glCompileShader(FShader);

Program = glCreateProgramObject();
glAttachObject(Program, VShader);
glAttachObject(Program, FShader);
glLinkProgram(Program);
```

Acte III : association et passage de paramètres.



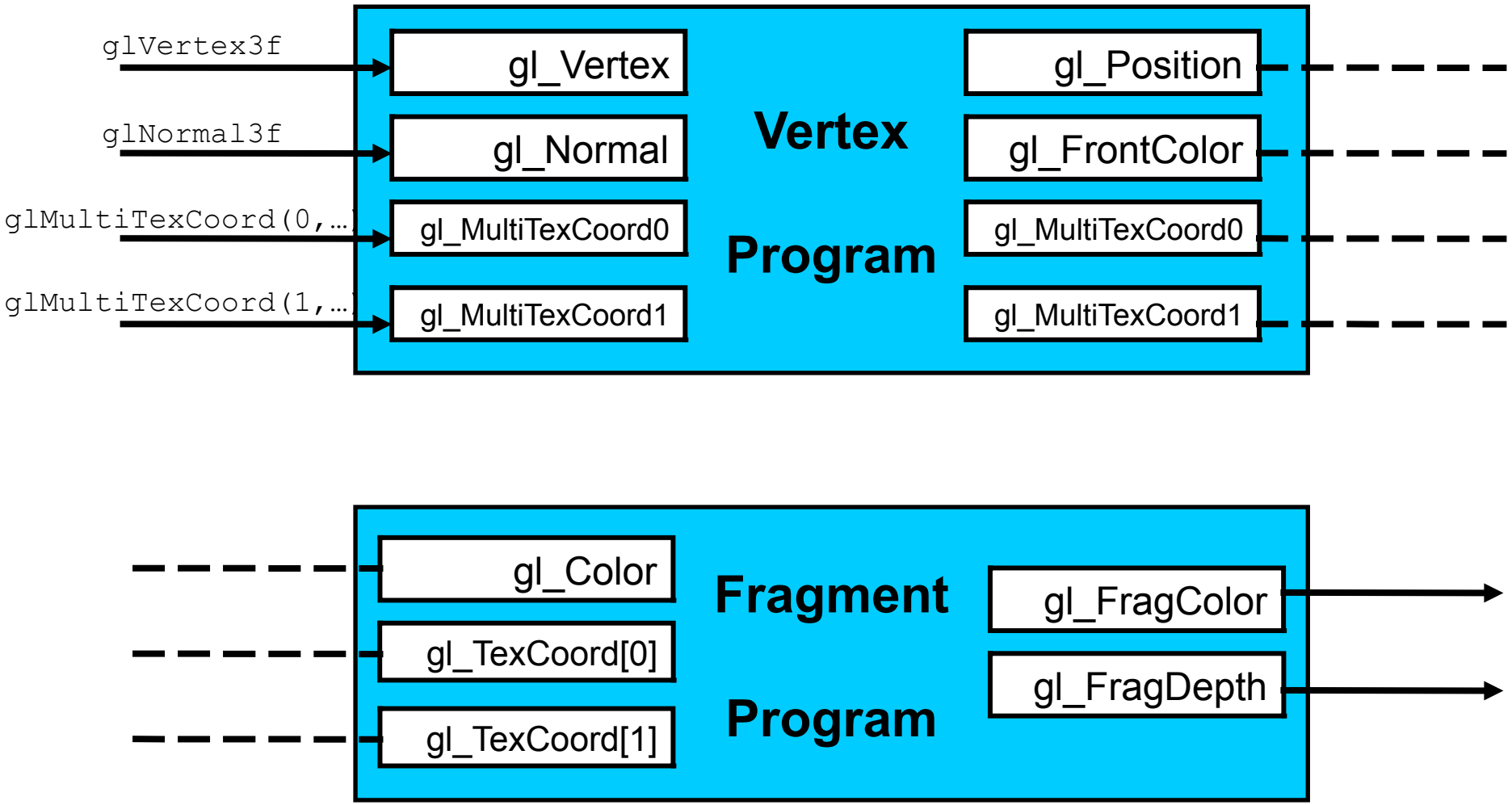

```
GLint LightPositionloc = glGetUniformLocation(Program,
                                              "LightPosition");
glUniform4f(LightPositionloc, x, y, z, w);
```

Acte IV : activation/désactivation du programme.




```
glUseProgramObject(Program);

...

glUseProgramObject(0);
```

« Built-in » variables



Un tout petit peu de syntaxe

Vertex Processor

- **Exécute les “vertex shaders”.**
- **En entrée : les données sommets (position, couleur, normales, etc,)**

vertex shader : envoie au moins une variable: *gl_Position*,

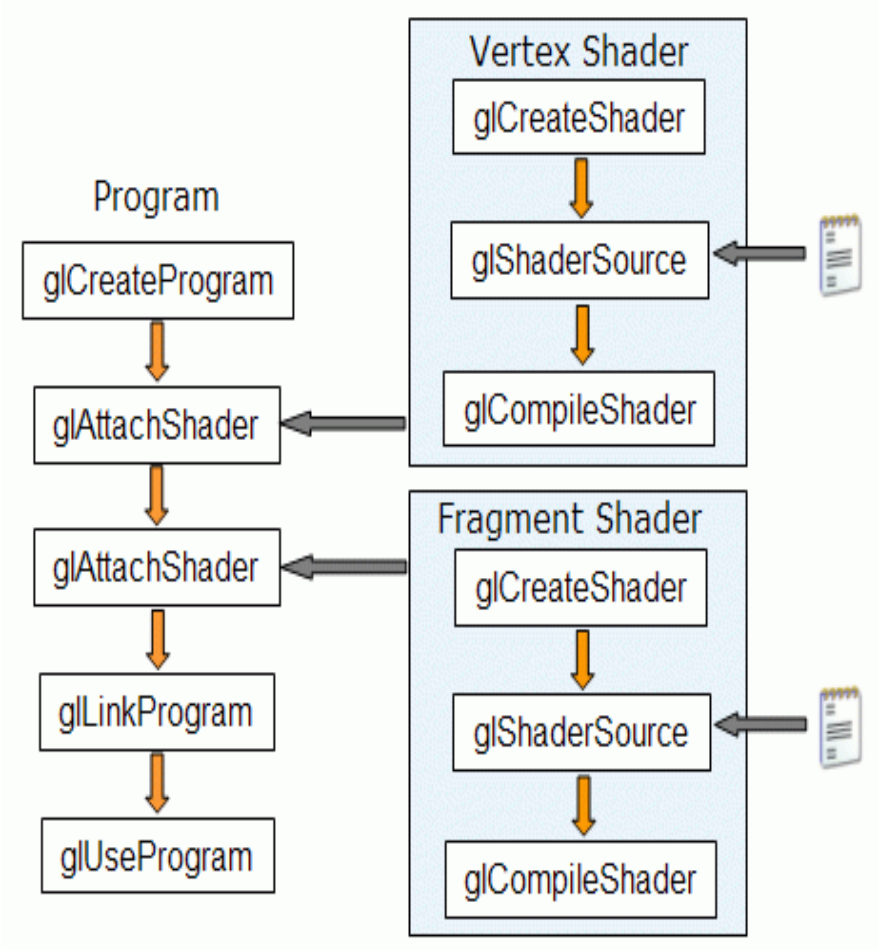
Transformation du sommet par les matrices modelview et projection.

Fragment Processor

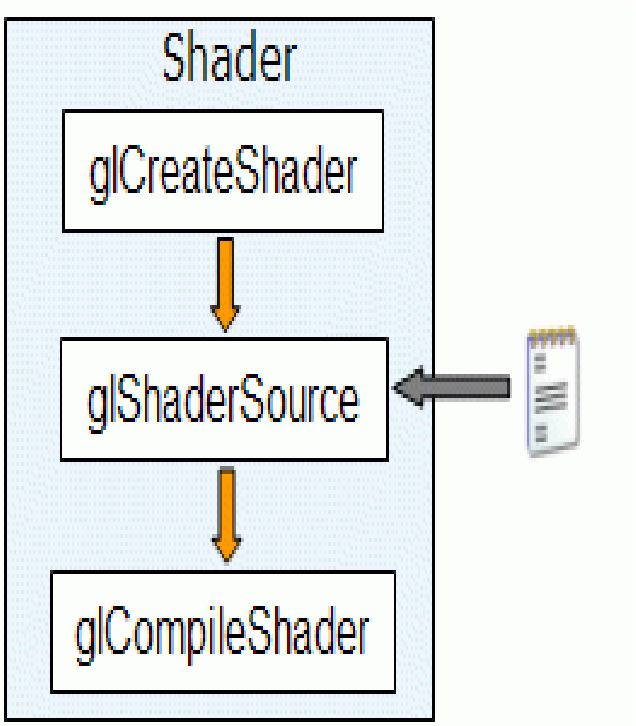
- **Exécute les “fragment shaders”**
  - Calcul des couleurs, et coordonnées texture par pixel
  - Application de texture
  - Brouillard
  - Calcul de normales (illumination par pixel)
- **Entrée : pour chaque sommet : positions, couleurs, normales, etc...**
- **Calculs par sommets → pour les fragments valeurs interpolées.**

Avec OpenGL

- **On a :**
- **un “ vertex shader”**
- **et un “ fragment shader”**
- **à utiliser dans une application OpenGL**



Création d'un shader



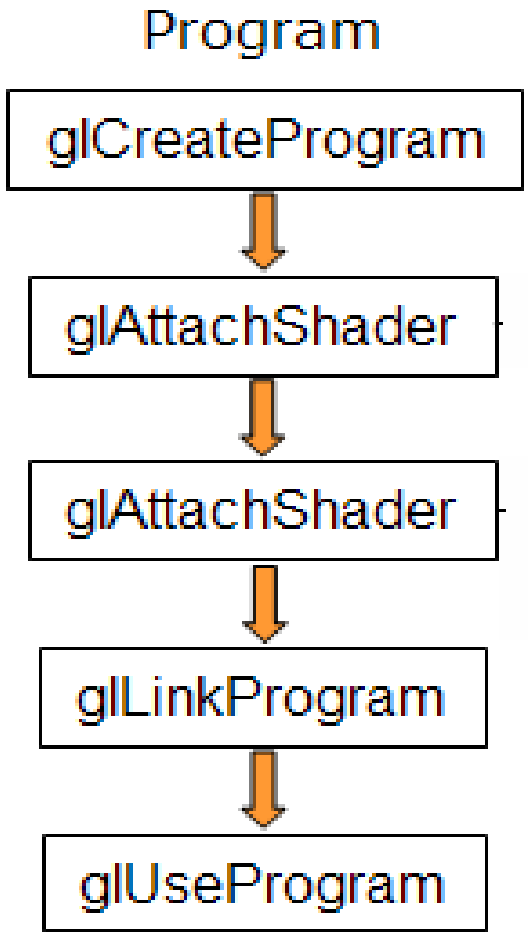
GLuint glCreateShader(GLenum shaderType);
Parameter:
shaderType - GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.

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

- shader - the handler to the shader.
- numOfStrings - the number of strings in the array.
- strings - the array of strings.
- lenOfStrings - an array with the length of each string, or NULL, meaning that the strings are NULL terminated.

void glCompileShader(GLuint shader);
Parameters:
shader - the handler to the shader.

Création d'un Programme



GLuint glCreateProgram(void);

void glAttachShader(GLuint program, GLuint shader);
Parameters:
program - the handler to the program.
shader - the handler to the shader you want to attach.

void glLinkProgram(GLuint program);
Parameters:
program - the handler to the program.

void glUseProgram(GLuint prog);
Parameters:
prog - the handler to the program you want to use,
or zero to return to fixed functionality

Définition des shaders et programmes

Compilation des shaders:

```
char* shaderSource;           // nom du shadersource
int shaderHandle = glCreateShader(GL_SHADER_TYPE);
    // shader-types: vertex || geometry || fragment
glShaderSource(shaderHandle, 1, shaderSource, NULL);
glCompileShader(shaderHandle);
```

Creer le programme et y attacher les shaders:

```
int programHandle = glCreateProgram();
glAttachShader(programHandle, shaderHandle); // pour vertex-
shader ET fragment-shader (ET geometry si nécessaire)!
```

Enfin édition des liens du programme :

```
glLinkProgram(programHandle);
```

Rendre les shaders actifs

Rendre un programme GLSL actif:

```
glUseProgram(programHandle); // shader activé
```

Le shader actif est utilisé jusqu'à ce qu'un autre `glUseProgram()` soit appelé avec un autre nom (program-handle).

Un appel à `glUseProgram(0)` entraîne qu'aucun programme n'est actif (état indéfini!).

Glunuse ?????

```
void setShaders()
 {
        char *vs,*fs;
        v = glCreateShader(GL_VERTEX_SHADER);
        f = glCreateShader(GL_FRAGMENT_SHADER);
        vs = textFileRead("toon.vert");
        fs = textFileRead("toon.frag");
        const char * vv = vs;
        const char * ff = fs;
        glShaderSource(v, 1, &vv,NULL);
        glShaderSource(f, 1, &ff,NULL);
        free(vs);free(fs);
        glCompileShader(v);
        glCompileShader(f);
        p = glCreateProgram();
        glAttachShader(p,v);
        glAttachShader(p,f);
        glLinkProgram(p);
        glUseProgram(p);
}
```

Desallouer (Cleaning Up)

void glDetachShader(GLuint program, GLuint shader)
Parameter:

- **program - The program to detach from.**
- **shader - The shader to detach.**

void glDeleteShader(GLuint id);
void glDeleteProgram(GLuint id);
Parameter:

- **id - The handler of the shader or program to delete.**

Communication OpenGL -> Shaders

- **Variable «Uniform » : « en lecture »**
  - sa valeur est modifiée par une primitive (ce ne peut être dans un *glBegin-glEnd,* ce ne peut être un attribut de sommet)

  Pour récupérer une variable uniforme :
  - GLint glGetUniformLocation(GLuint program, const char *name);
  - Parameters:
    - program - the handler to the program
    - name - the name of the variable.

  - GLint glUniform{1,2,3,4}fv(GLint location, GLsizei count, GLfloat *v);
  - Parameters:
    - location - the previously queried location.
    - v0,v1,v2,v3 - float values.
    - count - the number of elements in the array
    - v - an array of floats.

  - Idem avec le type *integer*, avec "i“ au lieu de "f"
  - Idem avec le type *matrix* :GLint glUniformMatrix{2,3,4}fv(GLint location, GLsizei count, GLboolean transpose, GLfloat *v);

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

Dans une application OpenGL:

```
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);
```

Communication OpenGL -> Shaders

- **Variable “Attribute” : “locale”**
  - pour définir une variable par sommet

  - GLint glGetAttribLocation(GLuint program,char *name);
  - Parameters:
    - program - the handle to the program.
    - Name - the name of the variable

  - GLint glVertexAttrib{1,2,3,4}fv(GLint location, GLfloat *v);
  - Parameters:
    - location - the previously queried location.
    - v0,v1,v2,v3 - float values.
    - v - an array of floats.

  - Idem avec le type *integer*, avec "i“ au lieu de "f"

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


Le rendu en OpenGL:
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();
```

Communication OpenGL -> Shaders

À la déclaration de la variable (en dehors des fonctions) :

uniform : la variable est une donnée venant du programme OpenGL pour le vertex shader et/ou le fragment shader. (LECTURE UNIQUEMENT).
attribute : indique une variable par vertex, venant du programme OpenGL. Ne peut être placée que dans le vertex shader. (LECTURE UNIQUEMENT). Obsolète. à partir de la version 150 du GLSL, remplacé par 'in'.
varying : donnée en sortie du vertex shader (LECTURE et ECRITURE), envoyée au fragment shader après avoir été interpolée (LECTURE UNIQUEMENT). Obsolète à partir de la version 150 du GLSL, remplacé par 'out'.
const : indique une variable qui est constante (LECTURE UNIQUEMENT).

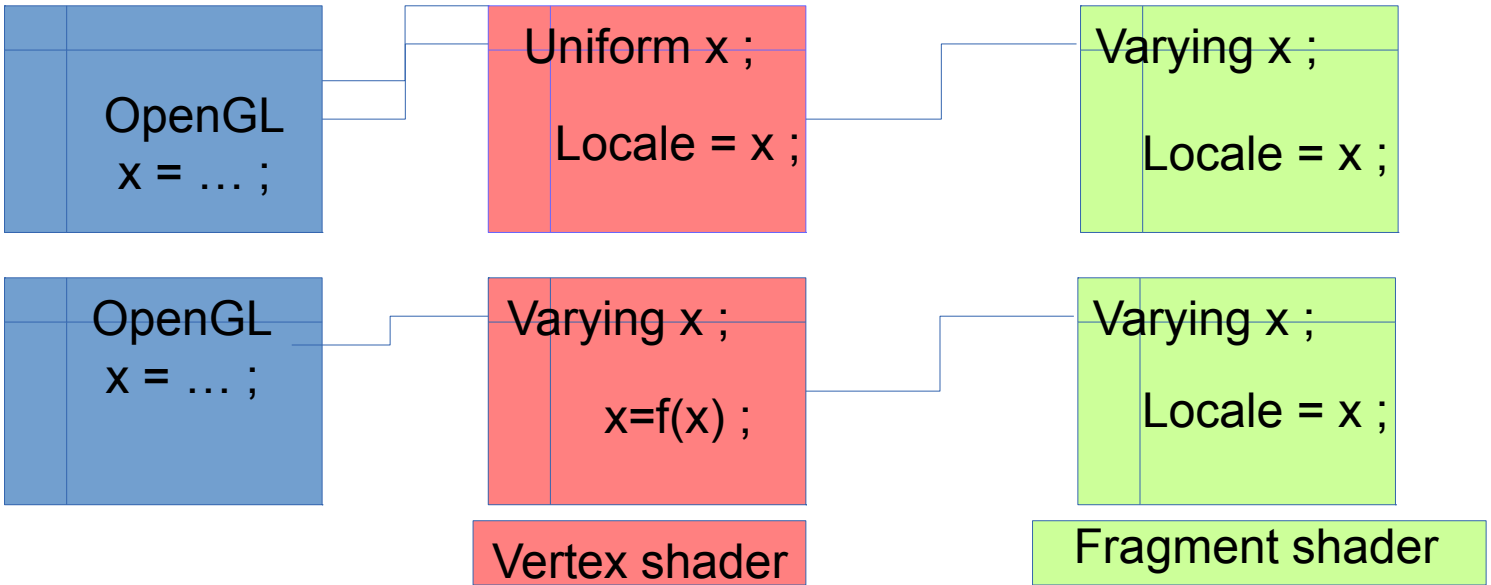
Dans la déclaration des fonctions :

in : variable initialisée en entrée, mais pas copiée en retour (qualificatifs par défaut)
out : variable copiée en retour, mais non initialisée en entrée.
inout : initialisée en entrée, copiée en retour.
const : variable d'entrée constante.

Communication OpenGL -> Shaders

- **Variable « Uniform » ou “in” : « en lecture »**
  - Valeur lue par le shader, fournie par le pgm OpenGL

- **Variable « Varying » ou “out” : « en lecture-écriture »**
  - sa valeur est modifiable dans le Vertex Shader pour passage au Fragment Shader



Communication OpenGL -> Shaders

```
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
GLuint TextureID  = glGetUniformLocation(programID, "myTextureSampler");
glUniform1i(TextureID, 0);
```

```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;

// Output data ; will be interpolated for each fragment.
out vec2 UV;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;

void main(){

// Output position of the vertex, in clip space : MVP * position
gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

// UV of the vertex. No special space for this one.
UV = vertexUV;
}
```

```
// Interpolated values from the vertex shaders
in vec2 UV;

// Ouput data
out vec3 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;

void main(){

// Output color = color of the texture at the specified UV
color = texture2D( myTextureSampler, UV ).rgb;
}
```

Types & Variables

- **float, bool, int**
- **vec{2,3,4}, bvec{2,3,4}, ivec{2,3,4} : 2,3,ou 4 float, bool, intege**
- **mat2, mat3 , mat4 : matrices 2x2, 3x3 et 4x4**
- **sampler1D, sampler2D, sampler3D : pour textures 1D, 2D, 3D**
- **samplerCube : pour les textures « cube map »**
- **sampler1DShadow, sampler2DShadow : pour shadow maps**
- **Structures :**

```
struct dirlight {
        vec3 direction;
        vec3 color;
        };
```

- **Variables ≈ comme en C**
- **Qualificatifs de Variables : const -attribute -uniform -varying**

Instructions et Fonctions

∀ ≈ C

- if (expression booléenne) ... else ...
- for (initialization; expression booléenne; expression de boucle)
  ...
- while (expression booléenne) ...
- do ... while (expression booléenne)

- Au moins une fonction main : “void main()”
- Elle peut retourner un type qcq (sauf array)
- Paramètres :
  - in
  - out
  - inout

Exemple « Concon »

- **Transforme les sommets et trace les primitives avec une seule couleur**

- **Vertex Shader**

  La transfo : vTrans = projection * modelview * incomingVertex

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
attribute vec4 gl_Vertex;
void main() {
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
    }
```

- **Fragment Shader**
  -
```
void main() {
gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
gl_Position = ftransform();
```

Exemple Couleur

- Accès à la couleur spécifiée par *glColor*.

1) OpenGL envoie la couleur : glColor
2) Le vertex shader reçoit la couleur (attribute gl_Color)
3) Le vertex shader calcule les couleurs face avant et face arrière (gl_FrontColor, &gl_BackColor)
4) Le fragment shader reçoit une couleur interpolée (varying gl_Color)
5) Le fragment shader assigne gl_FragColor à partir de gl_Color

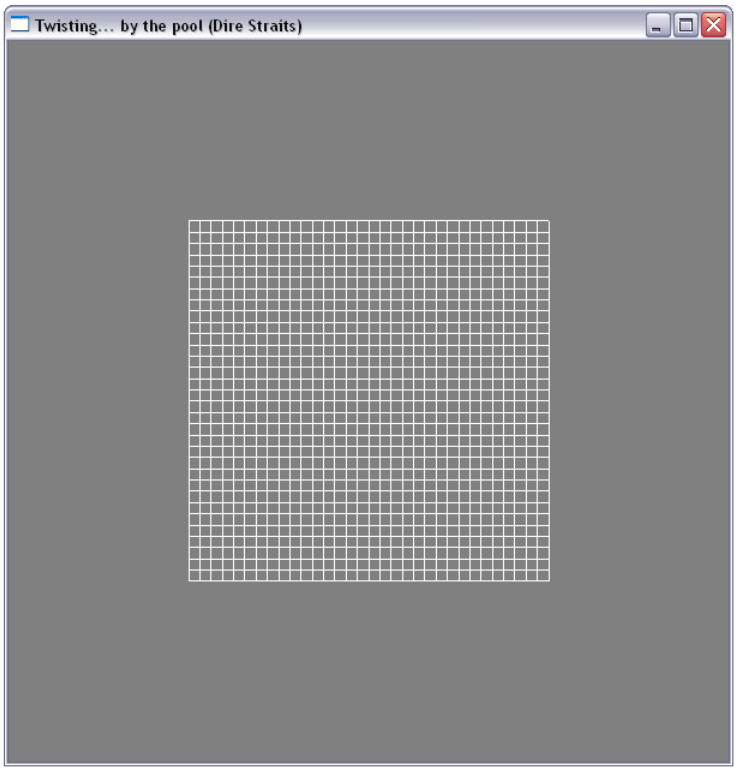
```
attribute vec4 gl_Color;

varying vec4 gl_FrontColor; //
    writable on the vertex shader
varying vec4 gl_BackColor; //
    writable on the vertex shader
varying vec4 gl_Color; // readable on
    the fragment shader

void main(){    // Vertex Shader
    gl_FrontColor = gl_Color;
    gl_Position = ftransform();
}

void main(){      // Fragment Shader
    gl_FragColor = gl_Color;
}
```

Des exemples

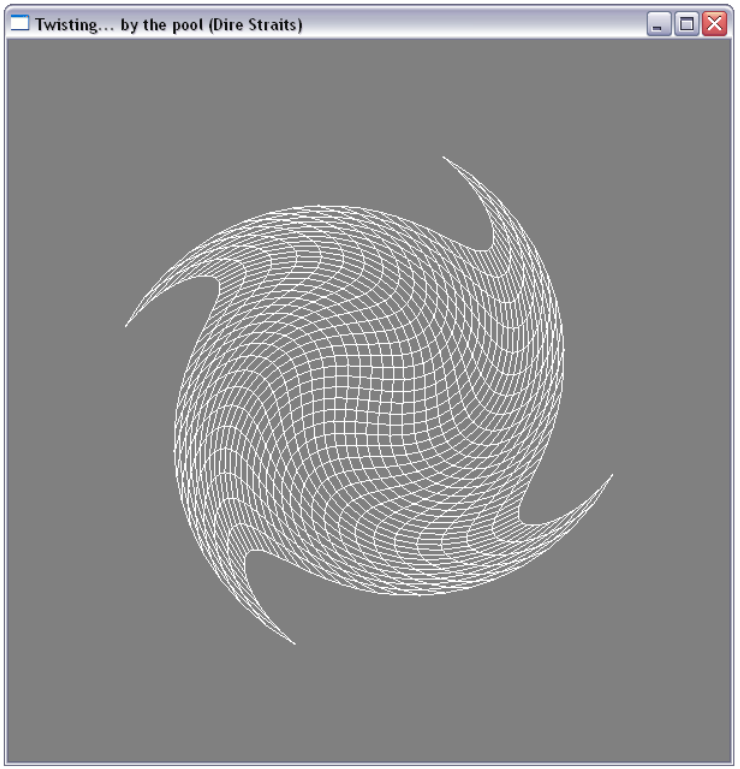
- http://www.lighthouse3d.com/opengl/glsl/examples/
- http://developer.3dlabs.com/downloads/index.htm
- **Etc……**

Twisting

$$P' = \text{RotationZ}(\text{twist} \times ||P||) \times P$$



twist = 0

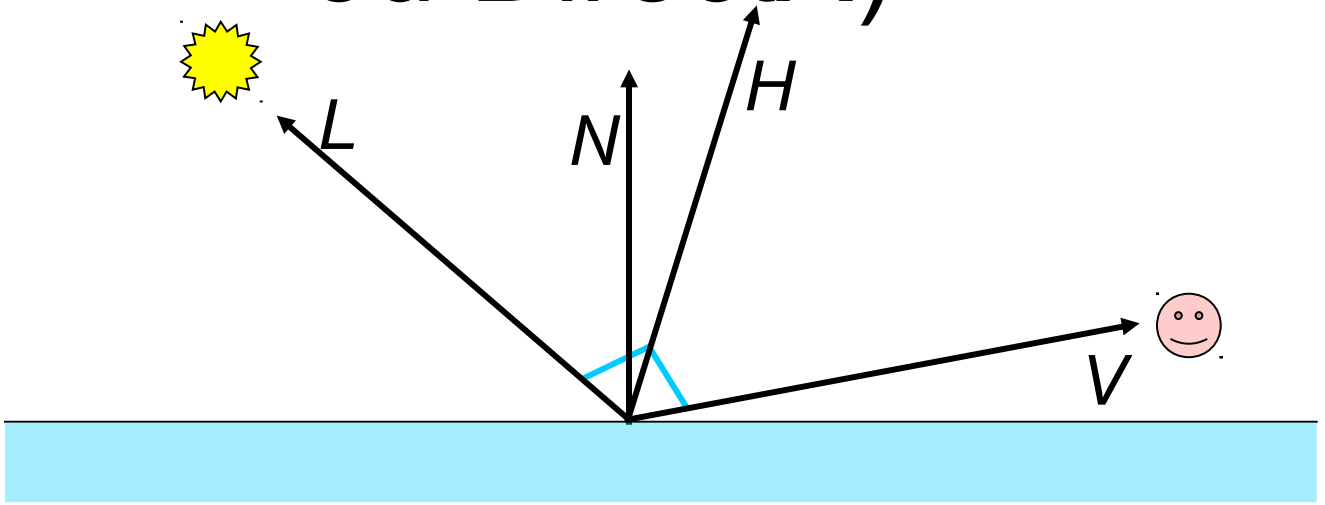


twist = 3

twisting.vert [GLSL]

```
uniform float twisting;

void main(void)
{
   float angle = twisting * length(gl_Vertex);
   float cosLength = cos(angle);
   float sinLength = sin(angle);

   gl_Position.x = cosLength*gl_Vertex.x - sinLength*gl_Vertex.y;
   gl_Position.y = sinLength*gl_Vertex.x + cosLength*gl_Vertex.y;
   gl_Position.z = 0.0;
   gl_Position.w = 1.0;

   gl_Position = gl_ModelViewProjectionMatrix * gl_Position;

   gl_FrontColor = gl_Color;
}
```

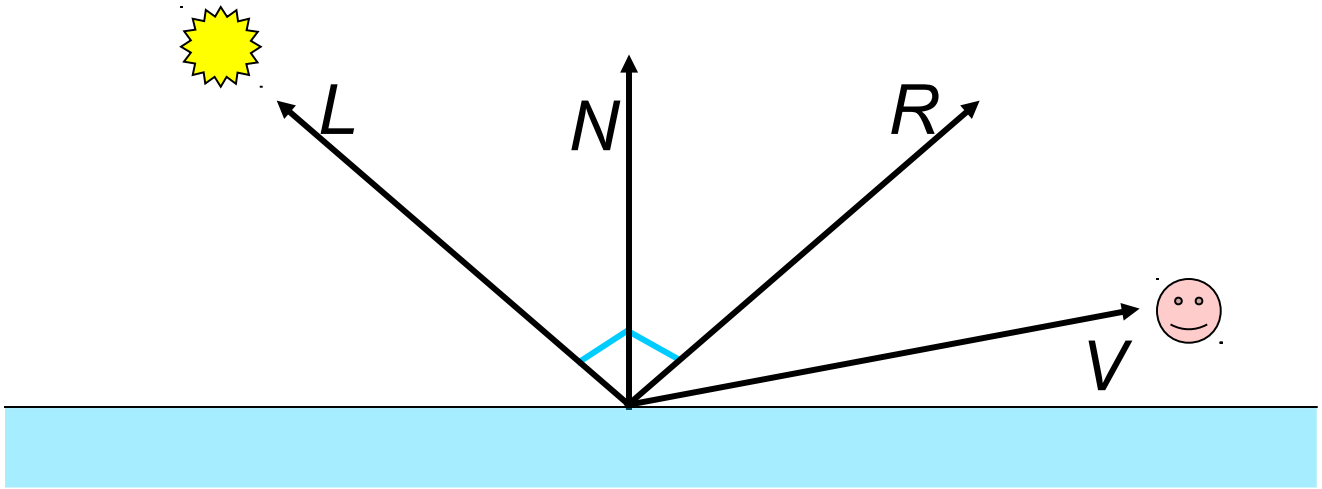
Simple Phong shading (OpenGL ou DirectX)



Color = AmbientColor +
DiffuseColor x (N . L) +
SpecularColor x (N . H)^n

H = (V + L)/2 (half vector)

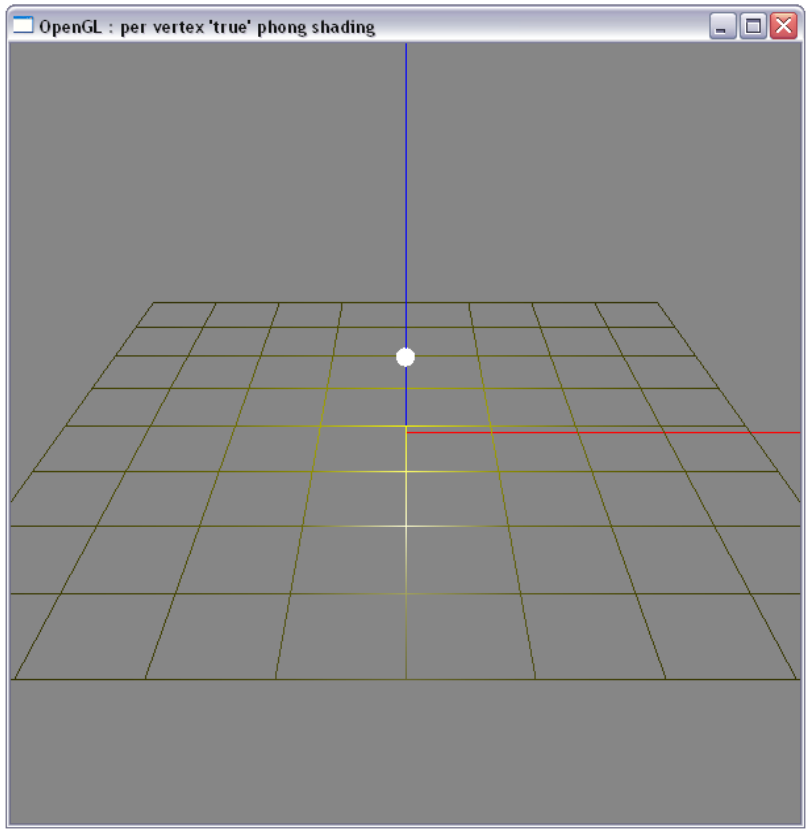
True Phong shading

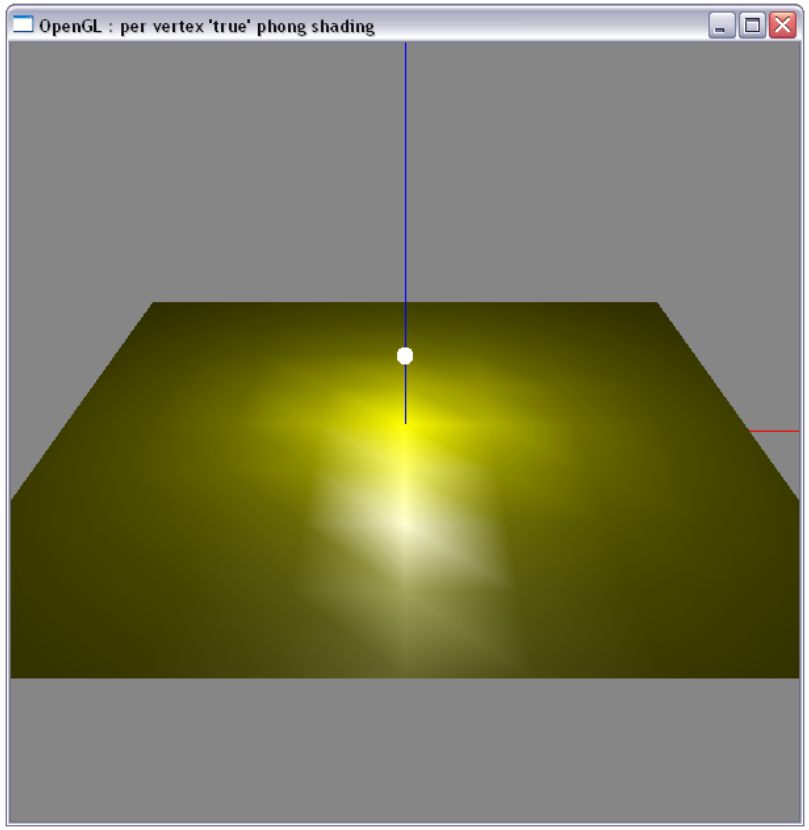


Color = AmbientColor +
DiffuseColor x (N . L) +
SpecularColor x (R . V)^n

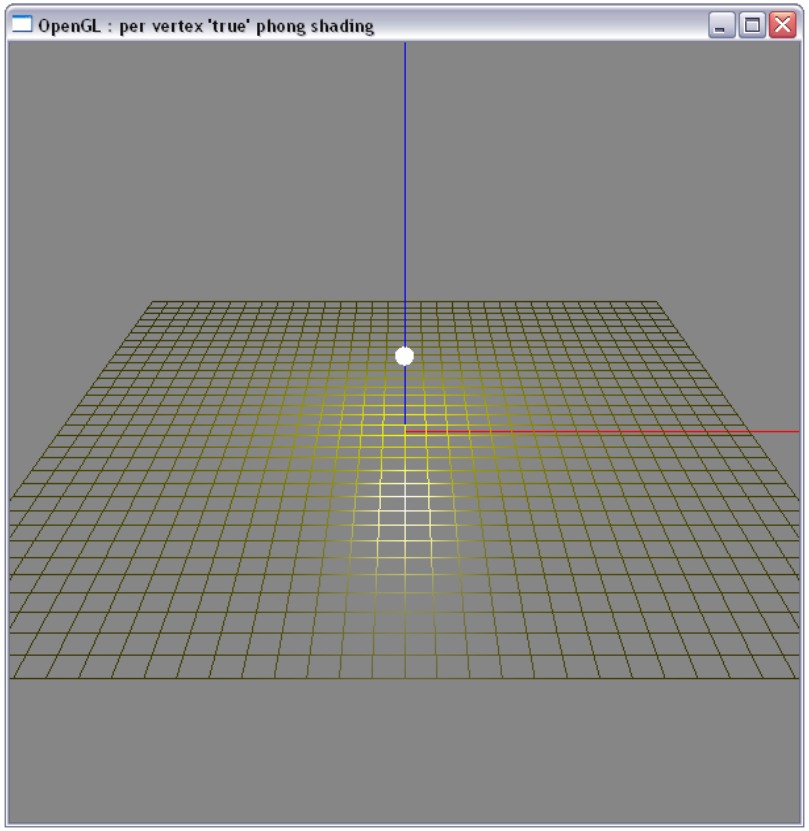
R : Reflected light vector

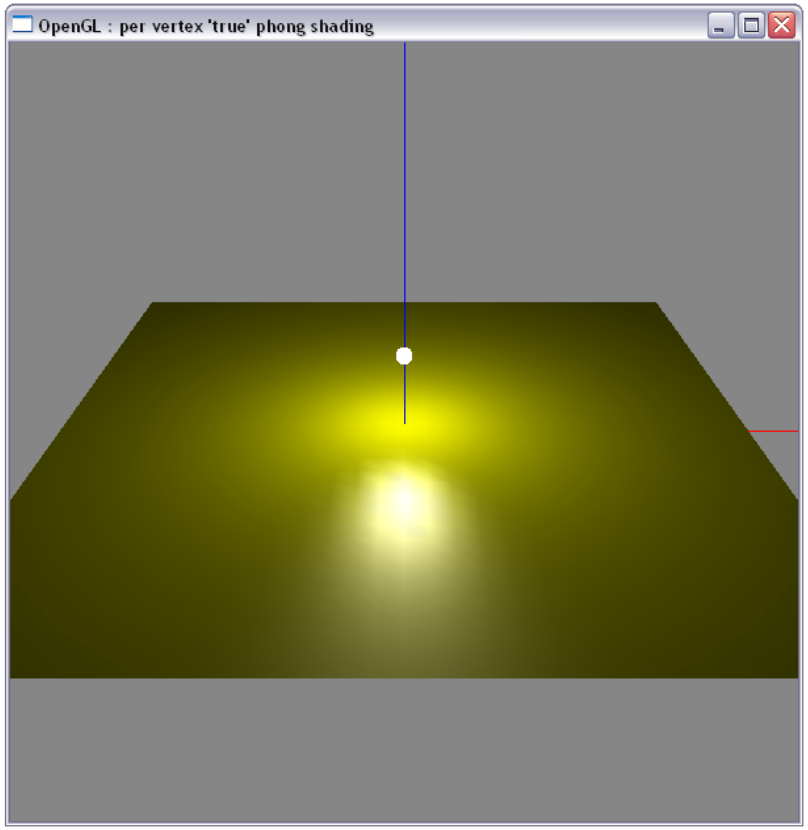
Per vertex true phong shading (1)



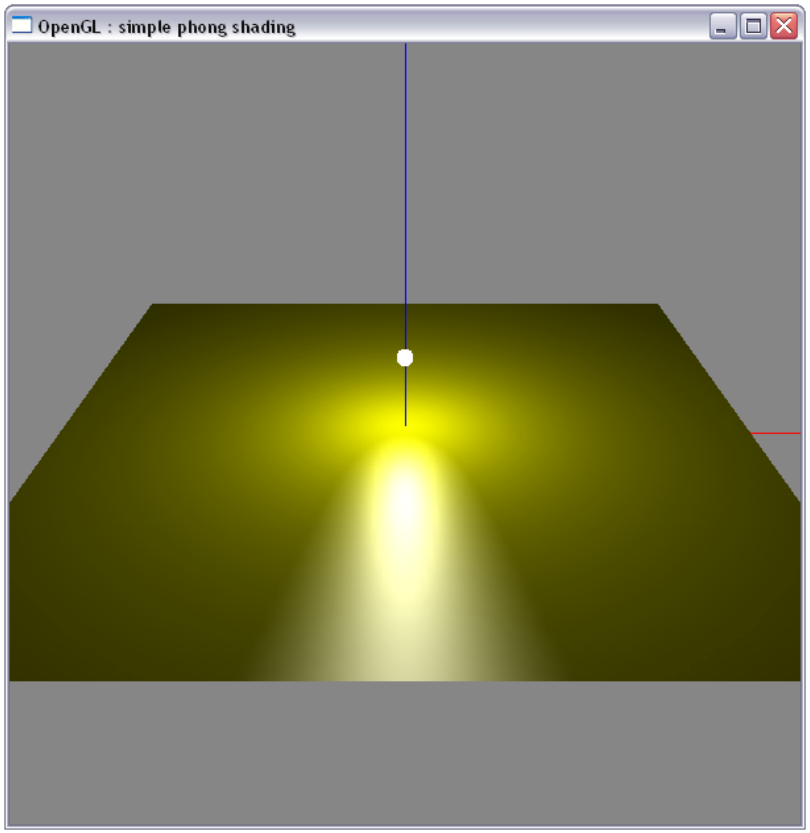


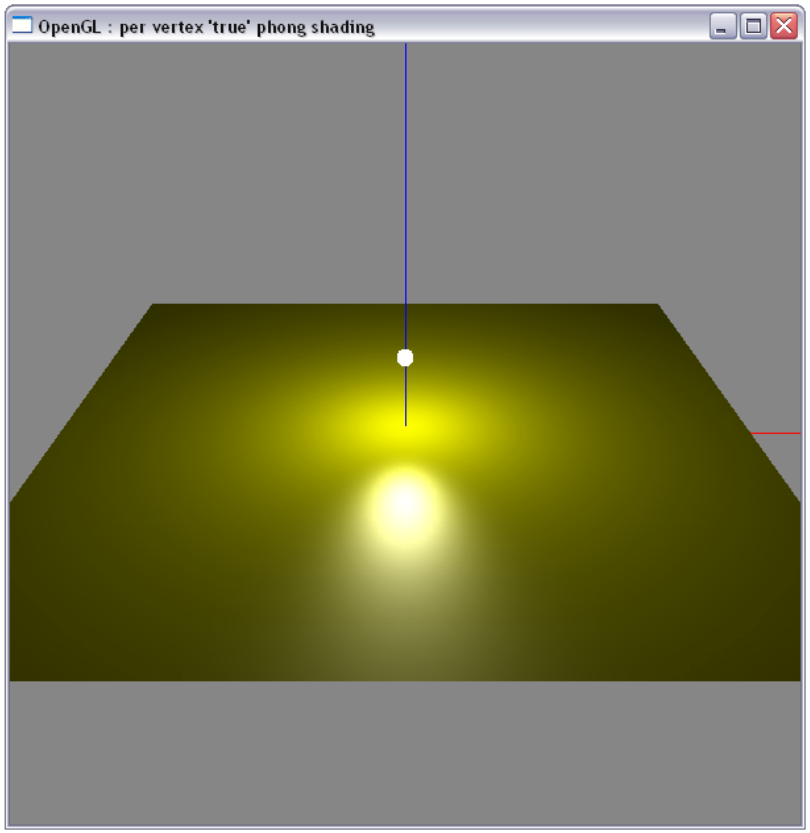
Per vertex true phong shading (2)





Comparaison des deux modèles.





per_vertex_true_phong.vert [GLSL]

```
uniform vec4 LightPosition;
uniform vec4 DiffuseMaterial;
uniform vec4 SpecularMaterial;

void main() {
    gl_Position = gl_ModelViewMatrix * gl_Vertex;           // Vertex position in View
    space.

    vec3 N = normalize(gl_NormalMatrix * gl_Normal);        // Vertex normal in View space.
    vec3 L = normalize((LightPosition - gl_Position).xyz); // Light vector in View space.

    gl_FrontColor = DiffuseMaterial * 0.4;              // Ambient component.
    float diffuse = dot(N, L);
    if (diffuse>0)
    {
    gl_FrontColor += DiffuseMaterial * diffuse;    // Diffuse component.

    vec3 R = reflect(-L, N);                       // Reflected light vector in View space.
    vec3 V = normalize(-gl_Position .xyz);      // View vector in View space.

    float specular = dot(V, R);
    if (specular>0)
          gl_FrontColor += SpecularMaterial * pow(specular, 40); // Specular component.
    }

    gl_Position = gl_ProjectionMatrix *  gl_Position;  // Vertex position in Screen space.
}
```

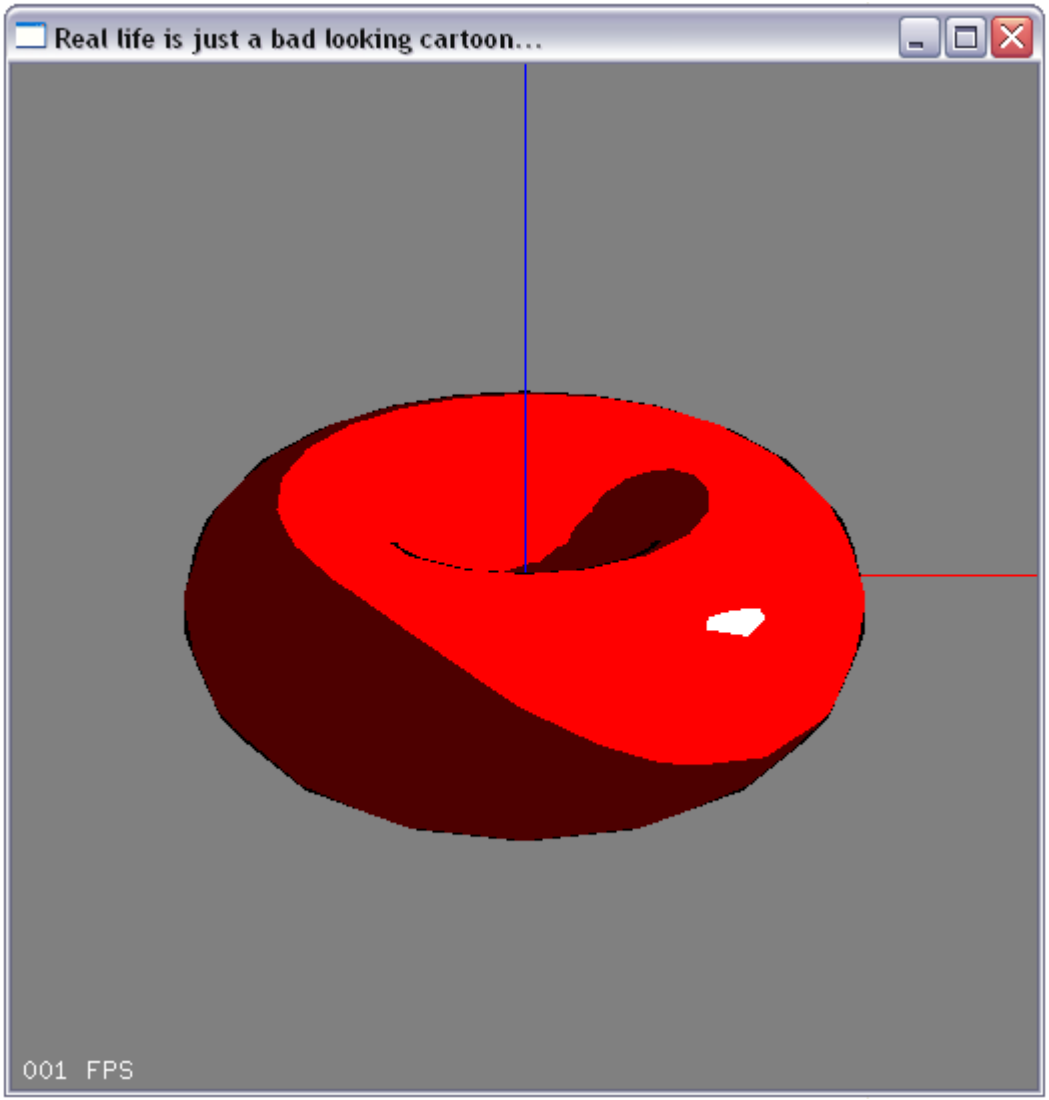
per_pixel_true_phong.vert [GLSL]

```
void main()
{
    // Vertex position in Screen space.
    gl_Position = ftransform();

    // Vertex position in View space.
    gl_TexCoord[0] = gl_ModelViewMatrix * gl_Vertex;

    // Vertex normal in View space.
    gl_TexCoord[1] = vec4(gl_NormalMatrix * gl_Normal, 0);
}
```

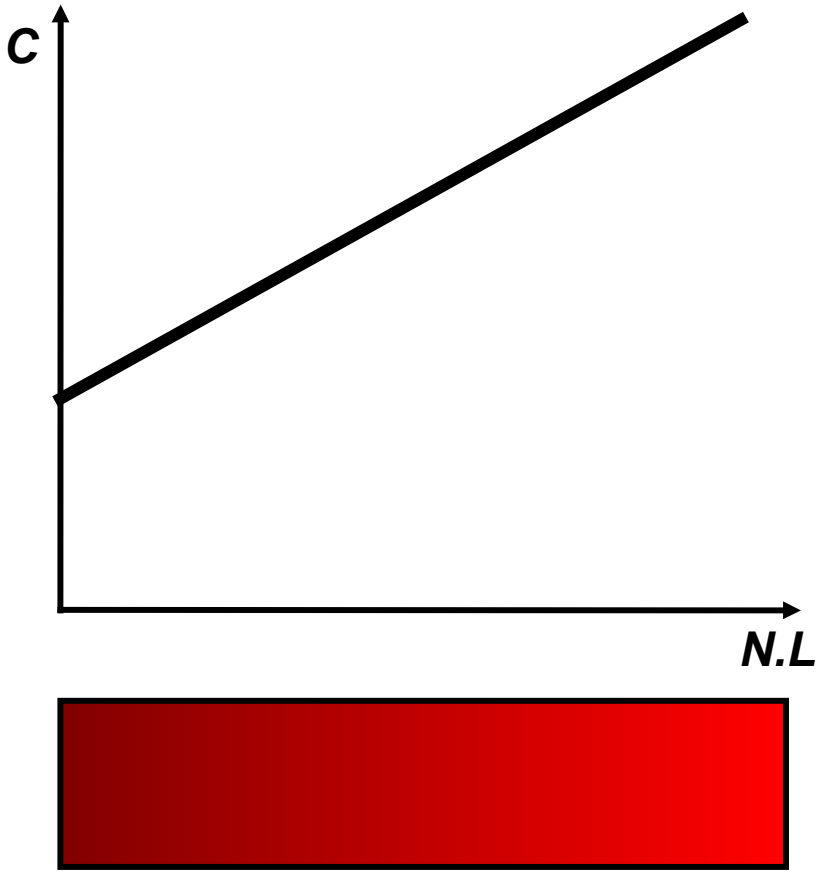
per_pixel_true_phong.frag [GLSL]

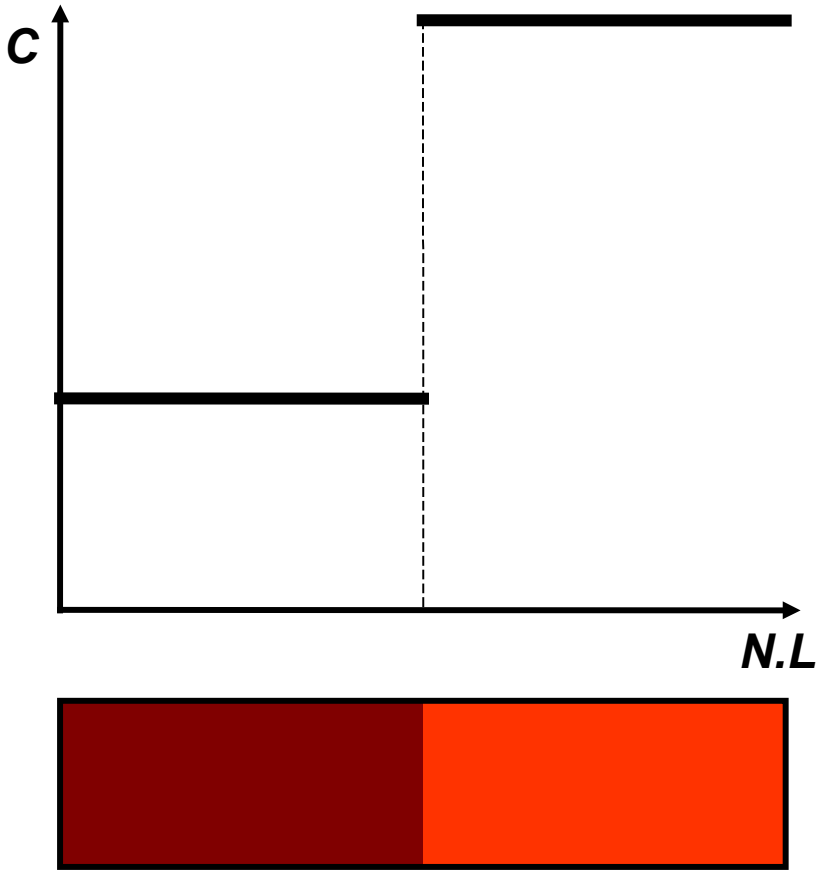
```
uniform vec4 LightPosition;   // Light position in View space.
uniform vec4 DiffuseMaterial; // Material.
uniform vec4 SpecularMaterial;

void main() {
    // Vertex normal in View space.
    vec3 N = normalize(gl_NormalMatrix * gl_TexCoord[1].xyz);
    // Light vector in View space.
    vec3 L = normalize((LightPosition - gl_ModelViewMatrix * gl_TexCoord[0]).xyz);

    // Ambient component.
    gl_FragColor  = DiffuseMaterial * 0.4;
    float diffuse = dot(N, L);
    if (diffuse>0) {
    // Diffuse component.
    gl_FragColor += DiffuseMaterial * diffuse;

    // Reflected light vector in View space.
    vec3 R = reflect(-L, N);

    // View vector in View space.
    vec3 V = normalize(-gl_TexCoord[0].xyz);

    // Specular component.
    float specular = dot(V, R);
    if (specular>0)
    gl_FragColor += SpecularMaterial * specular;
    }
}
```

Toon Shading



Toon Shading





toon_shading.vert [GLSL]

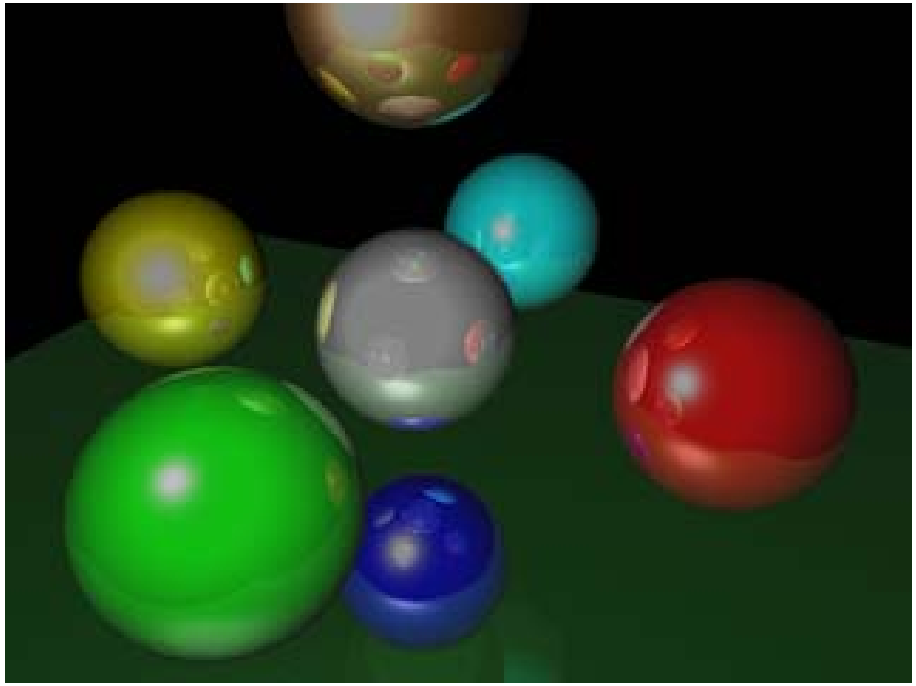
```
uniform vec4 LightPosition;

void main()  {
    gl_Position = gl_ModelViewMatrix * gl_Vertex; // Vertex position in View space.
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);        // Vertex normal in View space.
    vec3 L = normalize((LightPosition - gl_Position).xyz); // Light vector in View space.
    vec3 V = normalize(-gl_Position.xyz);                  // View vector in View space.

    gl_TexCoord[0].x = max(dot(N, V), 0.0);                // Perform edge detection.

    gl_TexCoord[0].y = dot(N, L);                          // Diffuse component.

    if (gl_TexCoord[0].y>0) {
          vec3 H = normalize(L + V);                       // Half vector in View space.
          gl_TexCoord[0].z = pow(max(dot(N, H), 0.0), 40);

    }
    else {
          gl_TexCoord[0].y = 0.0;gl_TexCoord[0].z = 0.0;
    }

    gl_Position = gl_ProjectionMatrix * gl_Position;   // Vertex position in Screen space.
}
```

toon_shading.frag [GLSL]

```
void main()  {
    float diffuse, specular;

    if (gl_TexCoord[0].x < 0.1)
        gl_FragColor = vec4(0, 0, 0, 1);
    else {
        // Compute final color with fixed diffuse and specular materials.
        if (gl_TexCoord[0].y < 0.2)
            diffuse = 0.3;
        else
            diffuse = 1.0;

        if (gl_TexCoord[0].z<0.5)
            specular = 0.0;
        else
            specular = 1.0;

        gl_FragColor = vec4(1, 0, 0, 1)*diffuse + vec4(1, 1, 1, 1)*specular;
    }
}
```

Rendu

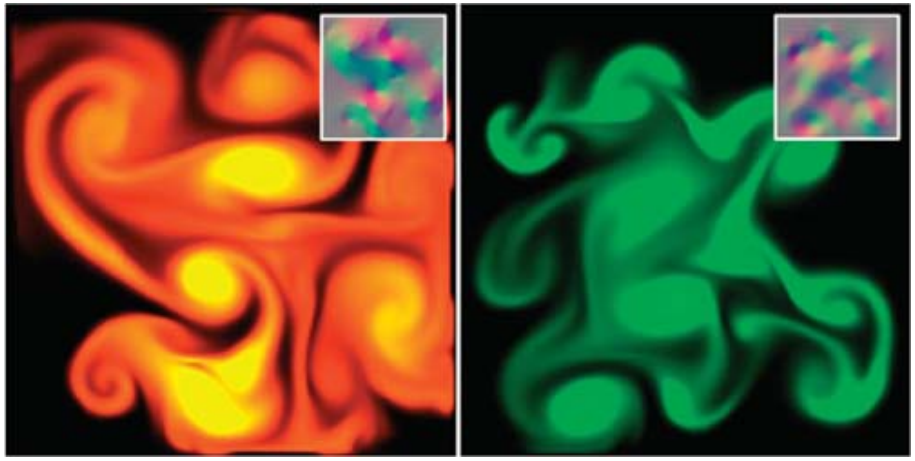
High Dynamic Range Rendering



Raytracing

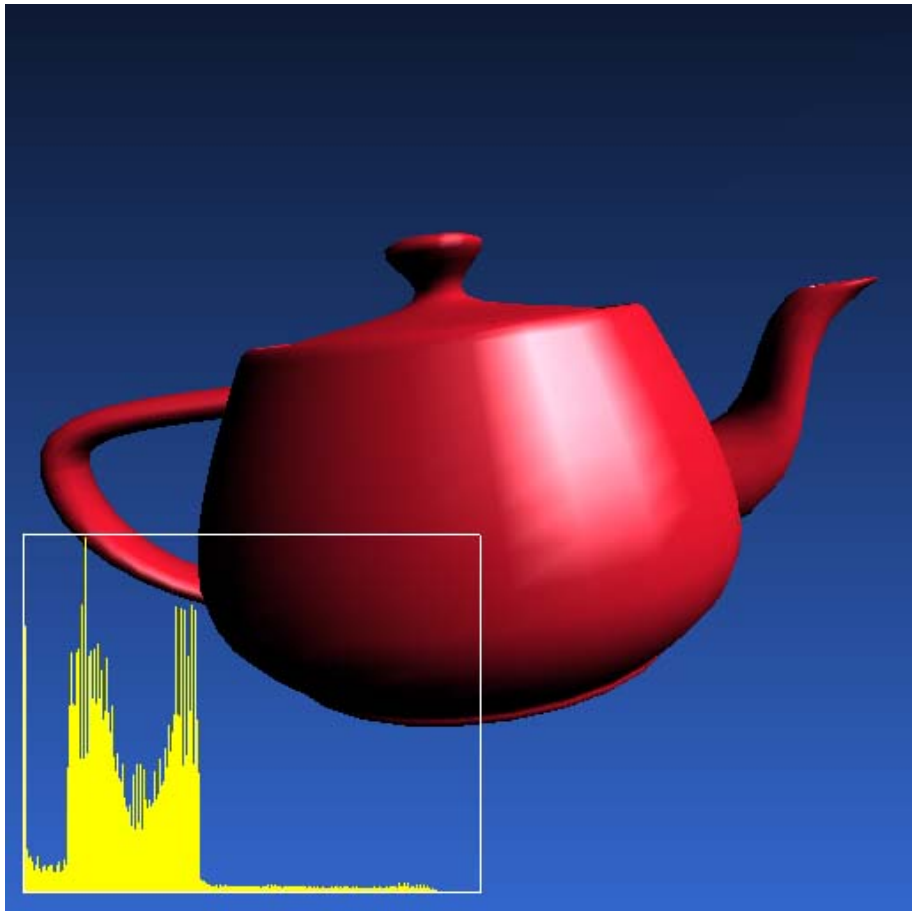
Simulation physique

Animation de vêtements



Simulation de fluides

Traitement Numérique des Images



Calcul d'histogrammes

Détection de contours