# M2-Images

J.C. Iehl

January 30, 2007

# Shaders ?

- A quoi ça sert ?
- Qu'est ce que c'est ?
- Comment ça marche ?

# Shaders : A quoi ça sert ?

## à faire mieux que les fonctions standards

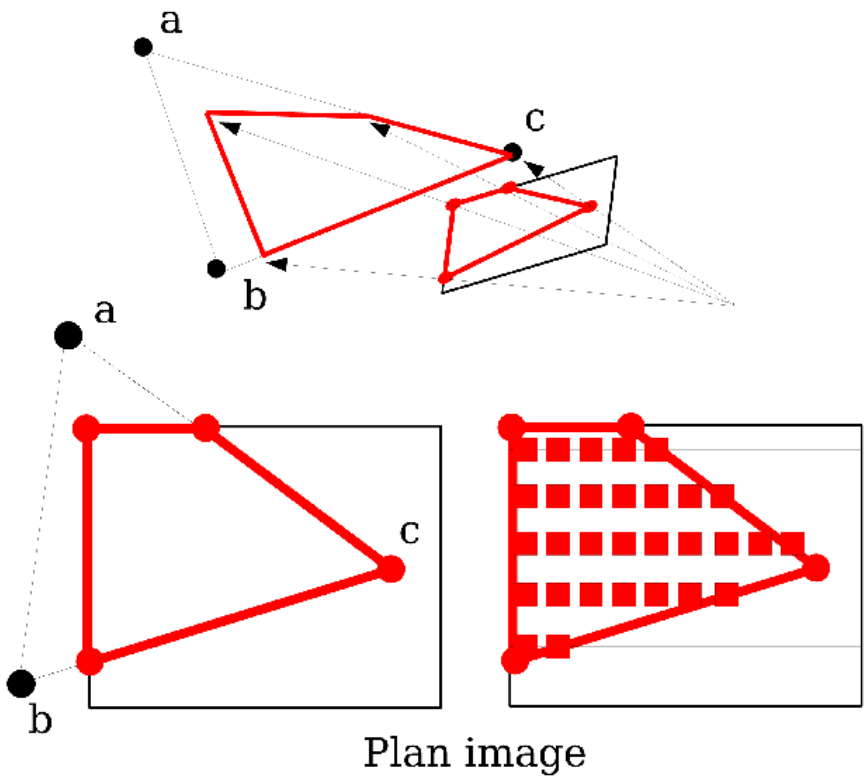
- matériaux réalistes (modèle local d'illumination),
- ajouter des détails géométriques,
- éclairement plus réaliste (ombres, pénombres, etc.),
- phénomènes naturels (feu, fumée, eau, nuages, etc.),
- matériaux non réalistes (rendu expressif),
- plus grande liberté pour accéder aux données (textures),
- traitement d'images,
- animation, déformation, etc.,

## à faire autre chose ...

# Shaders : Qu'est ce que c'est ?

- deux programmes éxécutés par les processeurs graphiques,
- *vertex shader* : permet de modifier la géométrie,
- *fragment shader* : permet de modifier l'image générée,

# Shaders : Qu'est ce que c'est ?



# Shaders : Qu'est ce que c'est ?

## vertex

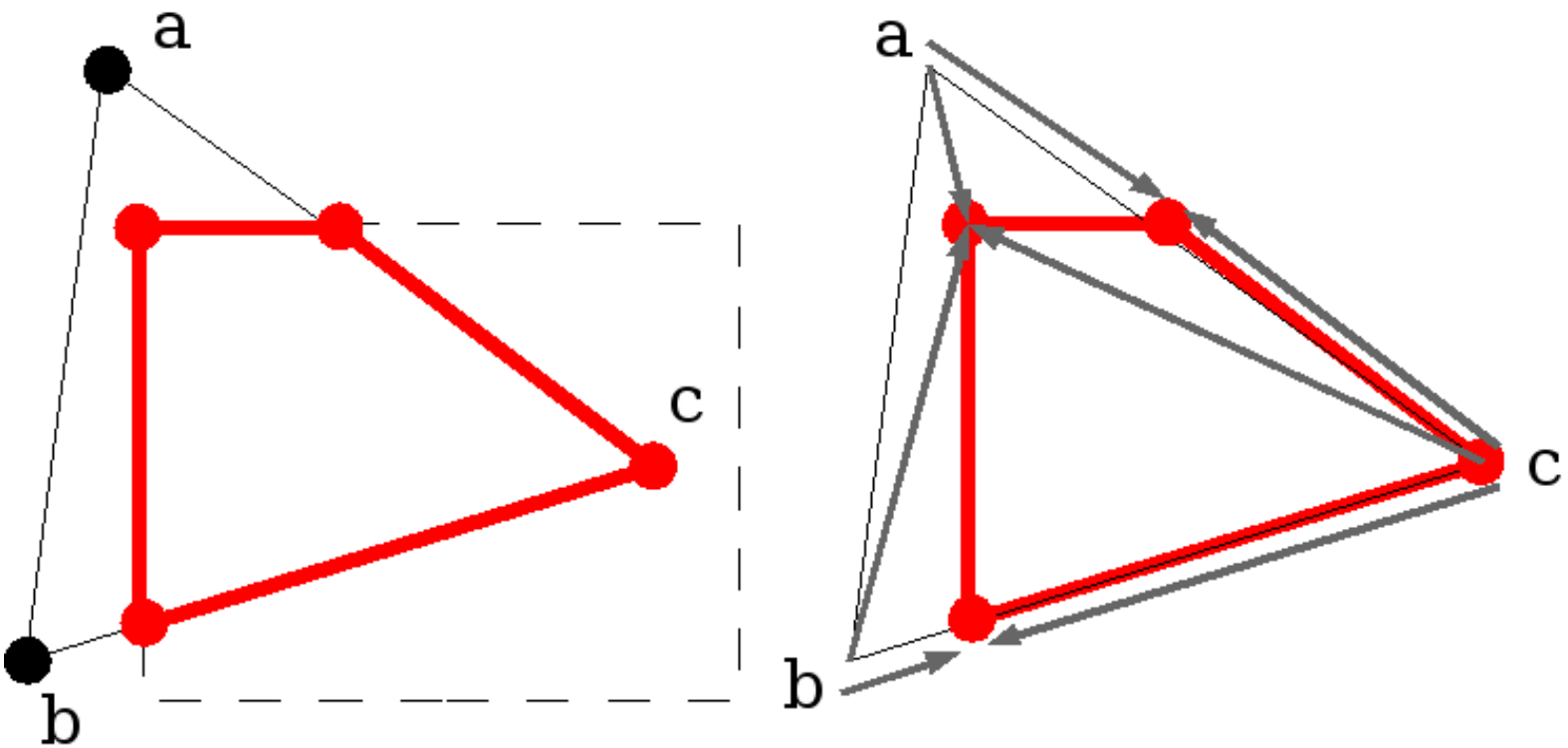
sommet de la primitive + tous ses attributs

- position,
- couleur,
- matière,
- attributs définis par l'application.

## Attention !

tous les attributs seront interpolés.

# Shaders : Qu'est ce que c'est ?



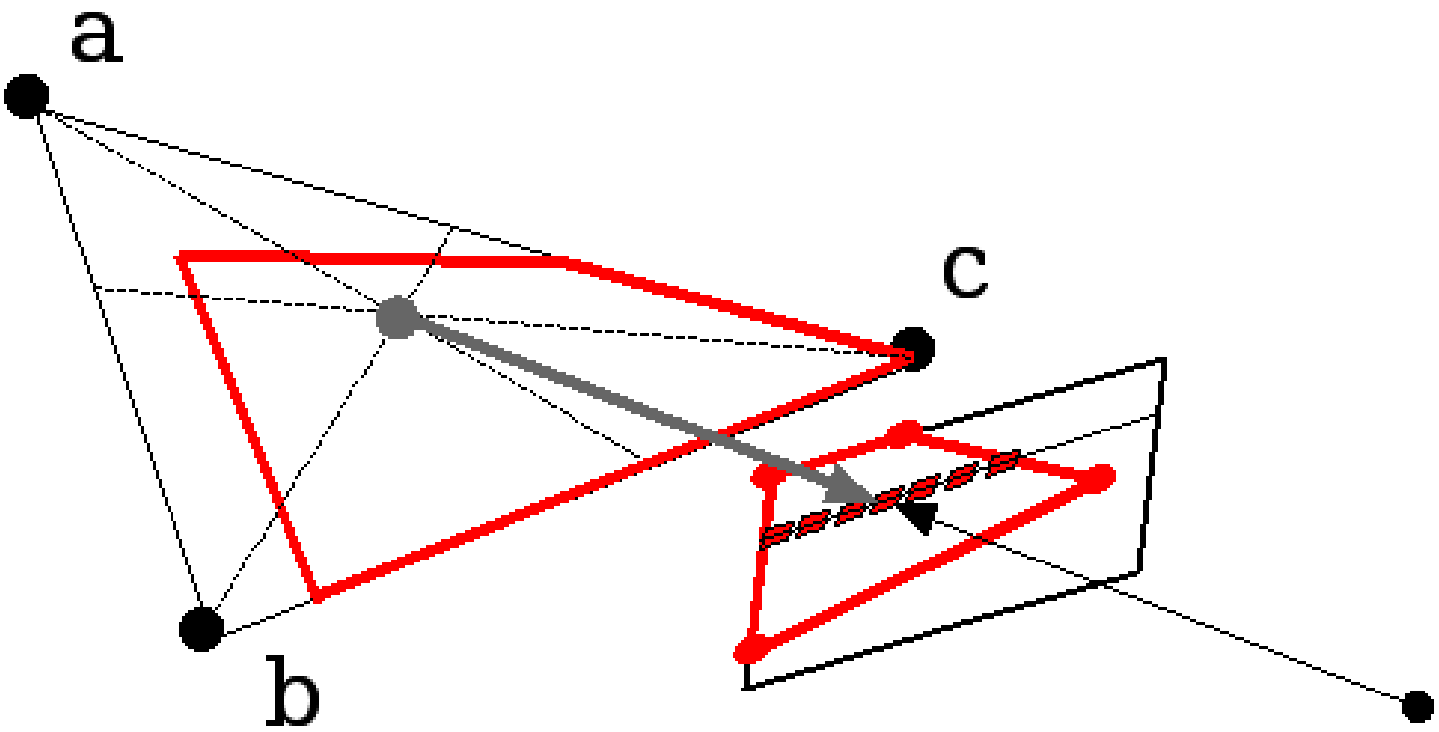
# Shaders : Qu'est ce que c'est ?

## fragment

élément de l'image + tous ses attributs

- position 3D, distance à la caméra,
- matière, couleur, transparence,
- attributs définis par l'application et le vertex shader puis interpolés lors de la fragmentation (rasterization).

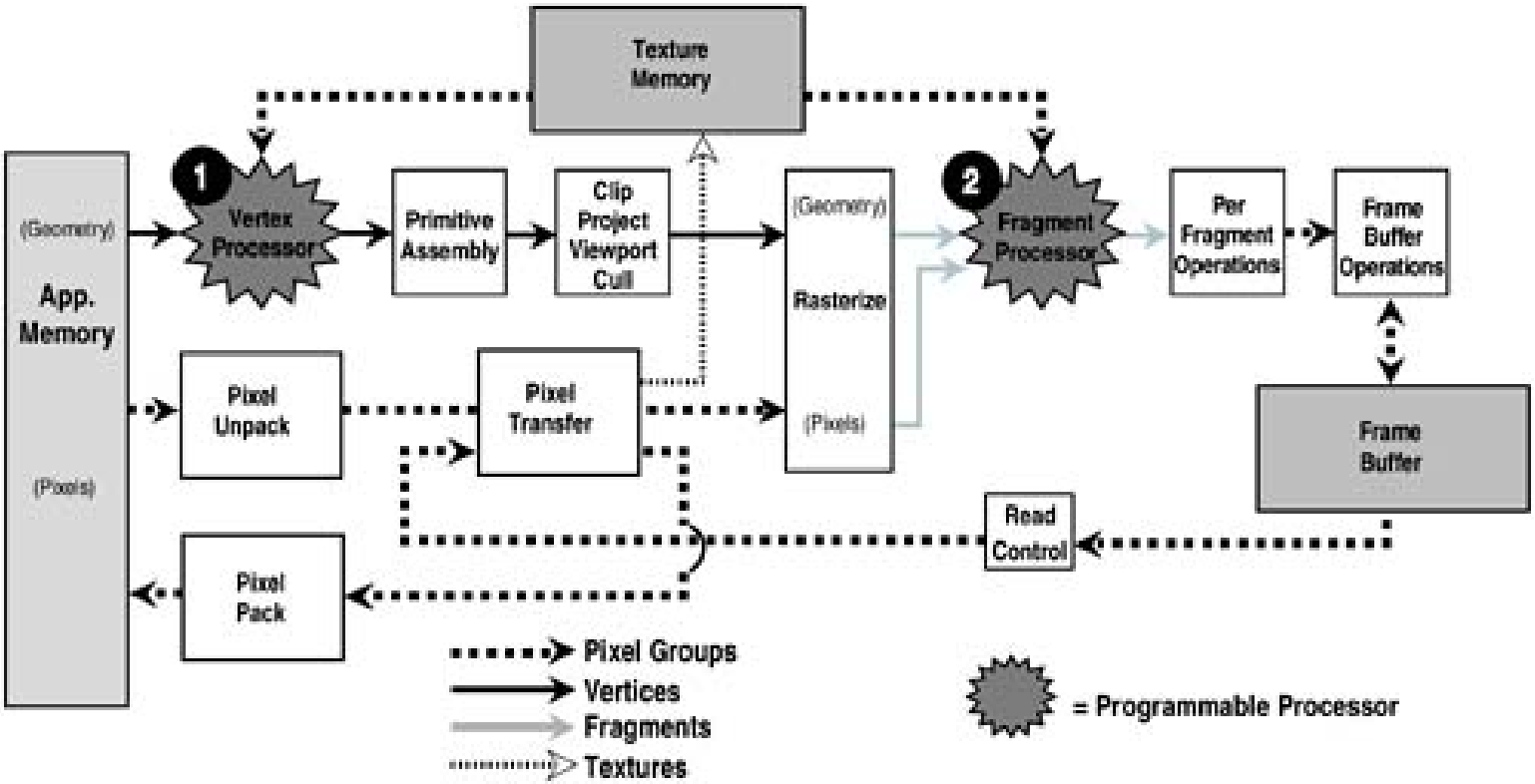
# Shaders : Qu'est ce que c'est ?



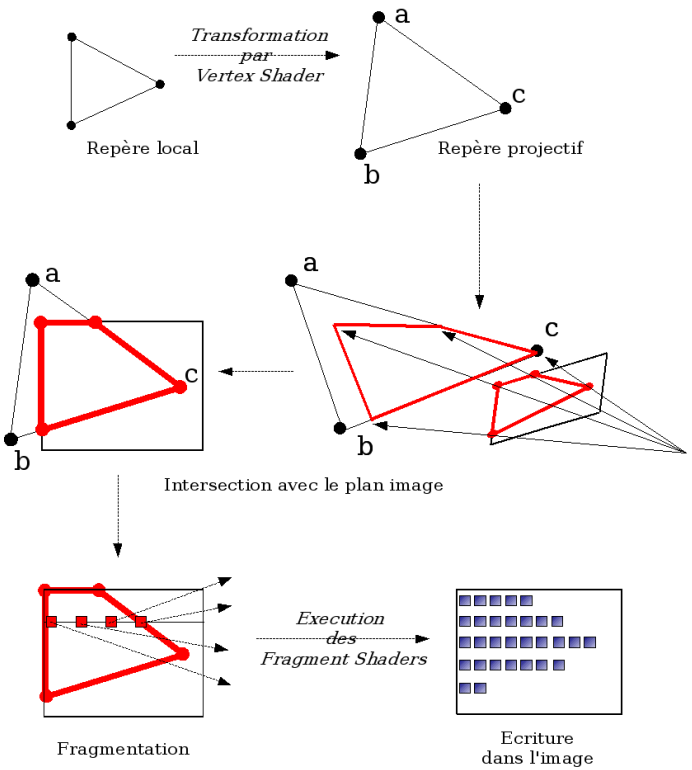
# Shaders : Qu'est ce que c'est ?

Ils remplacent les opérations standards dans le pipeline OpenGL

# Shaders : Qu'est ce que c'est ?



# Shaders : Qu'est ce que c'est ?



# Shaders : Comment ça marche ?

- programmes définis par l'application,
- paramètres passés par l'application,
- communication entre les vertex et les fragment shaders ?

## *OpenGL Shading Language*

- syntaxe proche du C / C++,
- types de base : scalaires, vecteurs, matrices,
- *samplers* (accès aux textures),
- accès au *contexte* OpenGL (constantes globales),
- accès au paramètres définis par l'application.

# Shaders : Comment ça marche ?

## Création des shaders

1. `glCreateShader()`
2. `glShaderSource()`
3. `glCompileShader()`

les shaders sont considérées comme des fonctions.

# Shaders : Comment ça marche ?

## Création du programme complet

1. `glCreateProgram()`
2. `glAttachShader()` `(vertex)`
3. `glAttachShader()` `(fragment)`
4. `glLinkProgram()`

il faut linker les fonctions pour obtenir un programme utilisable !

# Shaders : Comment ça marche ?

## Utilisation du programme

1. `glUseProgram()`

## Vérifications

- `glGetShaderInfoLog()`
- `glGetProgramInfoLog()`

détails sur : http://www.lighthouse3d.com/opengl/glsl/
http://www.opengl.org/documentation/specs/

# Shaders : Paramètres ?

## Paramètres uniform

définis par l'application, pour chaque primitive ou une image (paramètres généraux).

## Paramètres attribute

définis par l'application, pour chaque sommet (couleur, normale, etc.)

## Paramètres varying

définis par le vertex shader et interpolés lors de la fragmentation. L'application ne peut pas les définir explicitement.

## Déclaration

par le / les shader(s).

# Shaders : Valeur des Paramètres

## Paramètres uniform

- `location= glGetUniformLocation(program, xxx)`
- `glUniformXXX(location, xxx)`
- `glUniformMatrixXXX(location, xxx)`

## Paramètres attribute

- `location= glGetAttribLocation(program)`
- `glVertexAttribXXX(location, xxx)`

# Shaders : Exemple (vertex)

```
// Copyright (c) 2003-2004: 3Dlabs, Inc.
uniform float Time;              // updated each frame by the application
uniform vec4  Background;        // constant color equal to background
attribute vec3  Velocity;        // initial velocity
attribute float StartTime;       // time at which particle is activated
varying vec4 Color;

void main(void)
{
    vec4  vert;
    float t = Time - StartTime;

    if (t >= 0.0)
    {
        vert    = gl_Vertex + vec4 (Velocity * t, 0.0);
        vert.y -= 4.9 * t * t;
        Color   = gl_Color;
    }
    else
    {
        vert  = gl_Vertex;       // Initial position
        Color = Background;      // "pre-birth" color
    }
    gl_Position  = gl_ModelViewProjectionMatrix * vert;
}
```

# Shaders : Exemple (vertex)

```
// application

GLuint location, loc_Velocity, loc_Start;

glUseProgram(program);

location= glGetUniformLocation(program, "Background");
glUniform4f(location, 0.0, 0.0, 0.0, 1.0);
location= glGetUniformLocation(program, "Time");
glUniform1f(location, -5.0);

loc_Velocity= glGetAttribLocation(program, "Velocity");
loc_Start= glGetAttribLocation(program, "StartTime");

glBegin(GL_POINTS)
    glVertexAttrib3f(loc_Velocity, xxx, xxx, xxx);
    glVertexAttrib1f(loc_Start, xxx);
    glVertex3f(x, y, z);
glEnd();
```

# Shaders : Exemple (fragment)

```
// Copyright (c) 2003-2004: 3Dlabs, Inc.

varying vec4 Color;

void main (void)
{
    gl_FragColor = Color;
}
```

# Shaders : Demo !

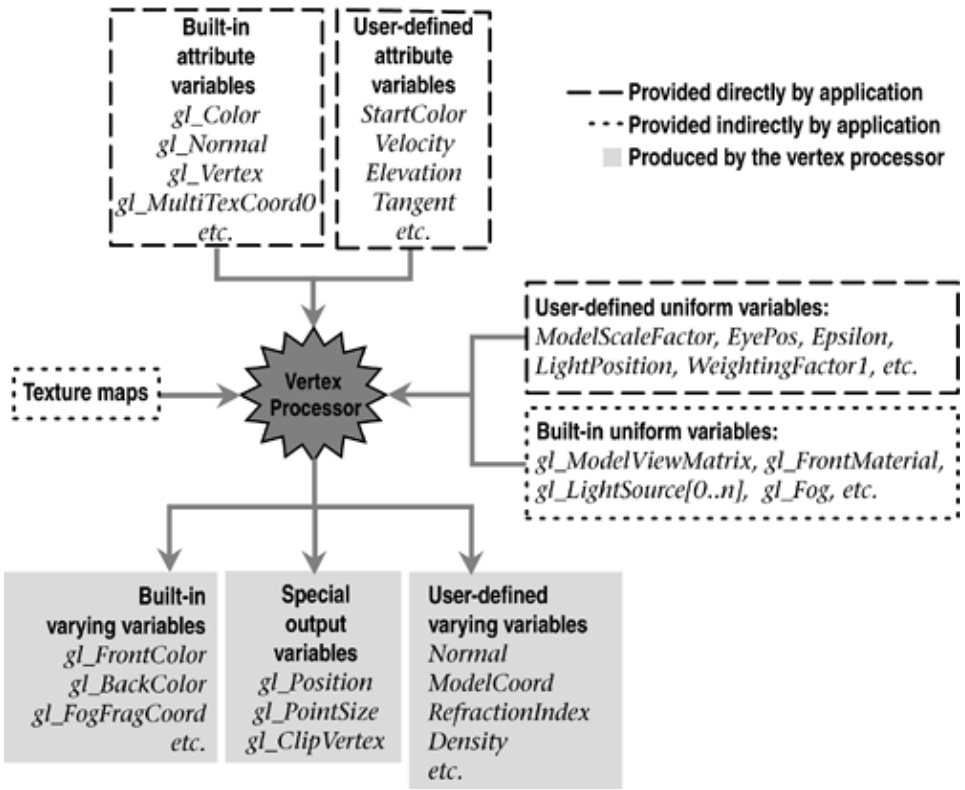
# Vertex Shaders : Comment ça marche ?

- éxécuté sur chaque sommet,
- doit calculer : `gl_Position`, dans l'espace projectif,

mais peut aussi calculer :

- la couleur : `gl_FrontColor`, `gl_BackColor`,
- la normale : `gl_Normal`,
- les coordonnées de textures : `gl_TexCoord[]`, etc.,
- et tous les `varying` utilisés par le fragment shader.

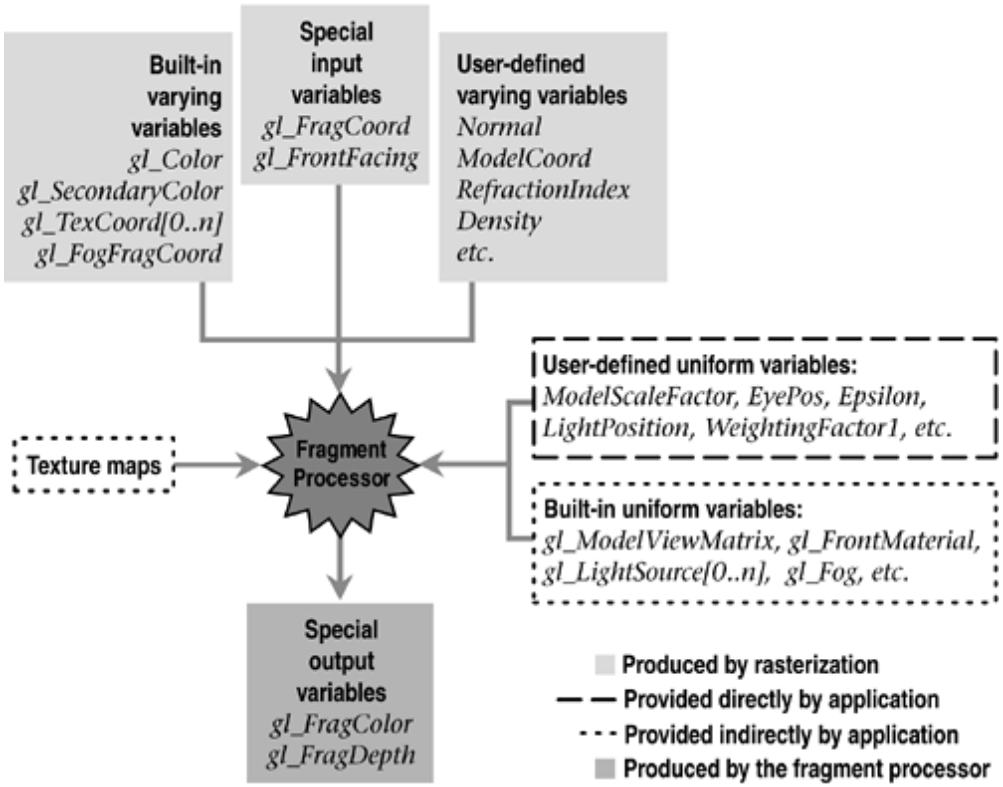
# Vertex Shaders : Paramètres (2)



# Fragment Shaders : Comment ça marche ?

- éxécuté sur chaque fragment dessiné,
- doit calculer : gl_FragColor.

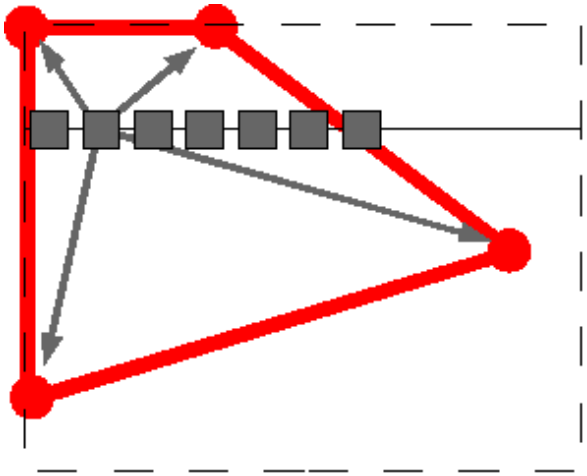
# Fragment Shaders : Paramètres (2)



# Shaders : Comment ça marche ?

## Ne pas oublier !

les paramètres `attribute`, `varying` sont interpolés lors de la fragmentation.



# Texture Shaders : Comment ça marche ?

```
uniform float compression;
uniform float saturation;

uniform sampler2DRect tex0;

void main(void)
{
  const vec3 rgby= vec3(0.3, 0.59, 0.11);
  vec3 color;
  float y, t;
  float k1= 1.0 / pow(saturation, 1.0 / compression);

  color= texture2DRect(tex0, gl_TexCoord[0].st).rgb;
  y= dot(color, rgby);

  if(y < saturation)
  {
    color/= y;
    t= k1 * pow(y, 1.0 / compression);
    color= color * t;
  }
  else
    color= vec3(1.0, 1.0, 1.0);
  gl_FragColor= vec4(color, 1.0);
}
```

# Texture Shaders : Comment ça marche ?

```
glUseProgram(program);

loc_tex= glGetUniformLocation(program, "tex0");
loc_compression= glGetUniformLocation(program, "compression");
loc_saturation= glGetUniformLocation(program, "saturation");

glActiveTexture(GL_TEXTURE0);
glUniform1i(loc_tex, 0);
glUniform1f(loc_compression, 2.2f);
glUniform1f(loc_saturation, 255.0f);

glBegin(GL_QUADS);
    glTexCoord2f(0.f, 0.f);
    glVertex3f(x, y+h, z);

    glTexCoord2f(0.f, v);
    glVertex3f(x, y, z);

    glTexCoord2f(u, v);
    glVertex3f(x+w, y, z);

    glTexCoord2f(u, 0.f);
    glVertex3f(x+w, y+h, z);
glEnd();
```

# Résumé : Pipeline

1. installation des shaders,
2. réception des primitives, des sommets et des paramètres,
3. opérations sur les sommets,
4. assemblage des primitives,
5. fragmentation des primitives,
6. opérations sur les fragments,
7. écriture des fragments dans l'image résultat.

# Etape 3 : opérations sur les sommets

## vertex shader

- responsable de transformer les sommets dans l'espace projectif de la caméra,
- peut définir des paramètres `varying` à destination des fragment shaders,
- peut utiliser les paramètres du contexte OpenGL.

```
// simple vertex shader

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

# Etape 6 : opérations sur les fragments

## fragment shader

- responsable de calculer la couleur du fragment,
- peut utiliser les paramètres `varying` crées par le vertex shader,
- peut utiliser les paramètres du contexte OpenGL.

```
// simple fragment shader

void main(void)
{
    gl_FragColor= vec4(0.0, 0.8, 0.0, 1.0);
}
```

# GLSL : Shading Language

## C/C++

- opérations sur les matrices, vecteurs,
- structures,
- fonctions (non récursives),
- passage de paramètres par copie (in, inout, out),
- fonctions spéciales.

# GLSL : types de base

## matrices

- mat2, mat3, mat4,
- mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, etc.
- mat4 m; m[1]= vec4(...);
- produits matrices, vecteurs.

# GLSL : types de base

## vecteurs

- vec2, vec3, vec4
- ivec234, bvec234
- sélection des composantes :
  vec3 v3; vec4 v4;
  v3= v4.xyz;
  v3.x= 1.0;
  v4= vec4(1.0, 2.0, 3.0, 4.0);

# Fonctions spéciales

- radians, degrees
- cos, sin, tan, acos, asin, atan,
- pow, exp, log, sqrt, inversesqrt,
- abs, sign, floor, ceil, fract, mod, etc.
- min, max, clamp
- mix, step, smoothstep,
- length(u), u= distance(p1, p0),
- dot, cross, normalize, etc,
- cf. GLSL specification, chapitre 8.

# Accès au contexte OpenGL

## Vertex Shader

- gl_Position, gl_PointSize, gl_ClipVertex,
- gl_Color, gl_Normal, gl_TexCoords[], gl_MultiTexCoords[], etc.
- gl_ModelViewMatrix, gl_ProjectionMatrix, + inverse,
- cf. GLSL specification, chapitre 7.

# Accès au contexte OpenGL

## Fragment Shader

- gl_FragColor, gl_FragDepth, gl_FragData, gl_FragCoord,
- etc.,
- cf. GLSL specification, chapitre 7.

# OpenGL 2 : Mise au point

## Mise au point de shader

- BuGLe
- gDEBugger
- + IDE spécialisés (windows) : FX Composer, RenderMonkey, etc.

# OpenGL 2 : Optimisation

## Qui est le maillon faible ?

- application / API / GPU ?
- le gpu est constitué de plusieurs "éléments" :
- traitement des primitives,
- vertex shaders,
- fragmentation,
- fragment shaders,
- tests et opérations sur l'image résultat.

# Questions ?

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