M2-Images

Rendu Temps Réel - Shaders

J.C. Iehl

April 11, 2012

Shaders ?

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

Shaders : à 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 ?

des programmes exécutés par les processeurs graphiques :

- *vertex shader* : permet de transformer un sommet d'une primitive,
- *geometry shader* : permet d'ajouter/supprimer des primitives,
- *pixel shader* : permet de modifier l'image générée,

ils ne sont exécutés que lors de l'affichage de primitives : points, lignes, triangles, etc.

écrits dans un langage spécial, proche du C / C++ :

- HLSL pour DirectX
- GLSL pour OpenGL

Shaders : qu'est ce que c'est ?

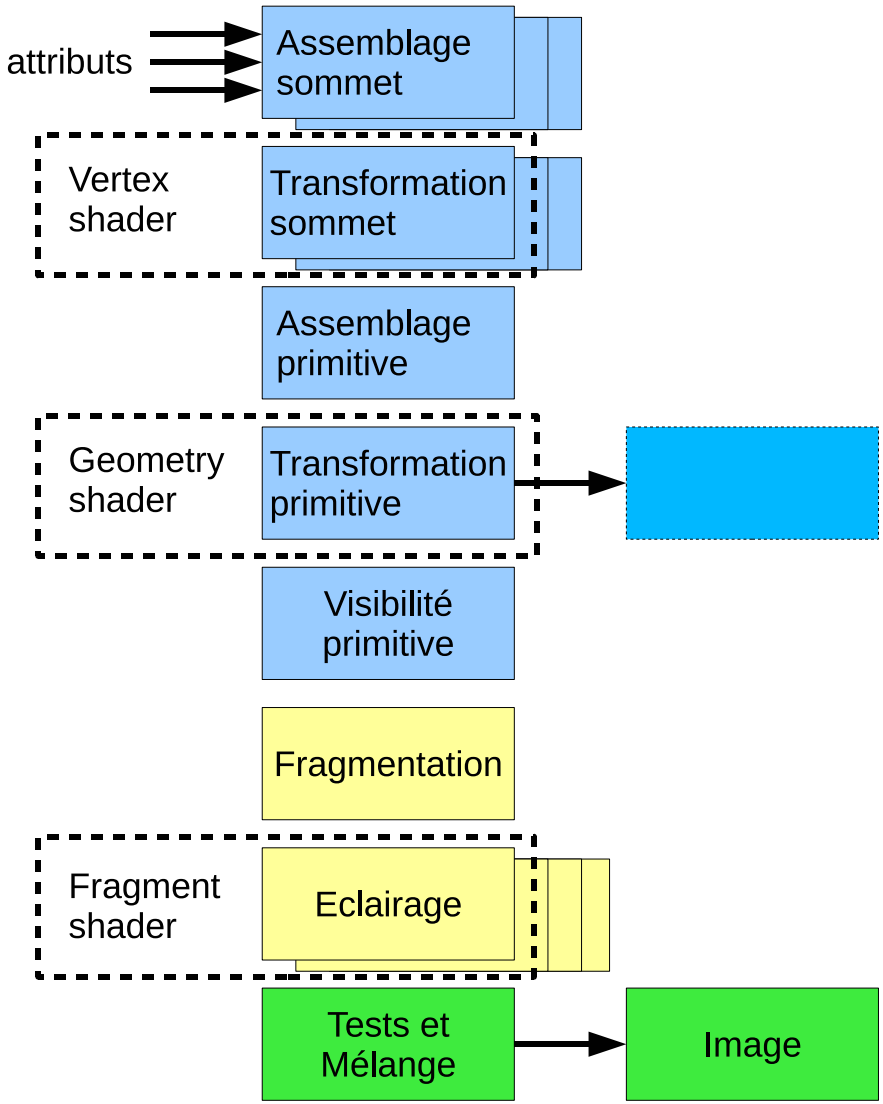
les shaders réalisent les opérations du pipeline graphique.

un shader a donc des obligations :

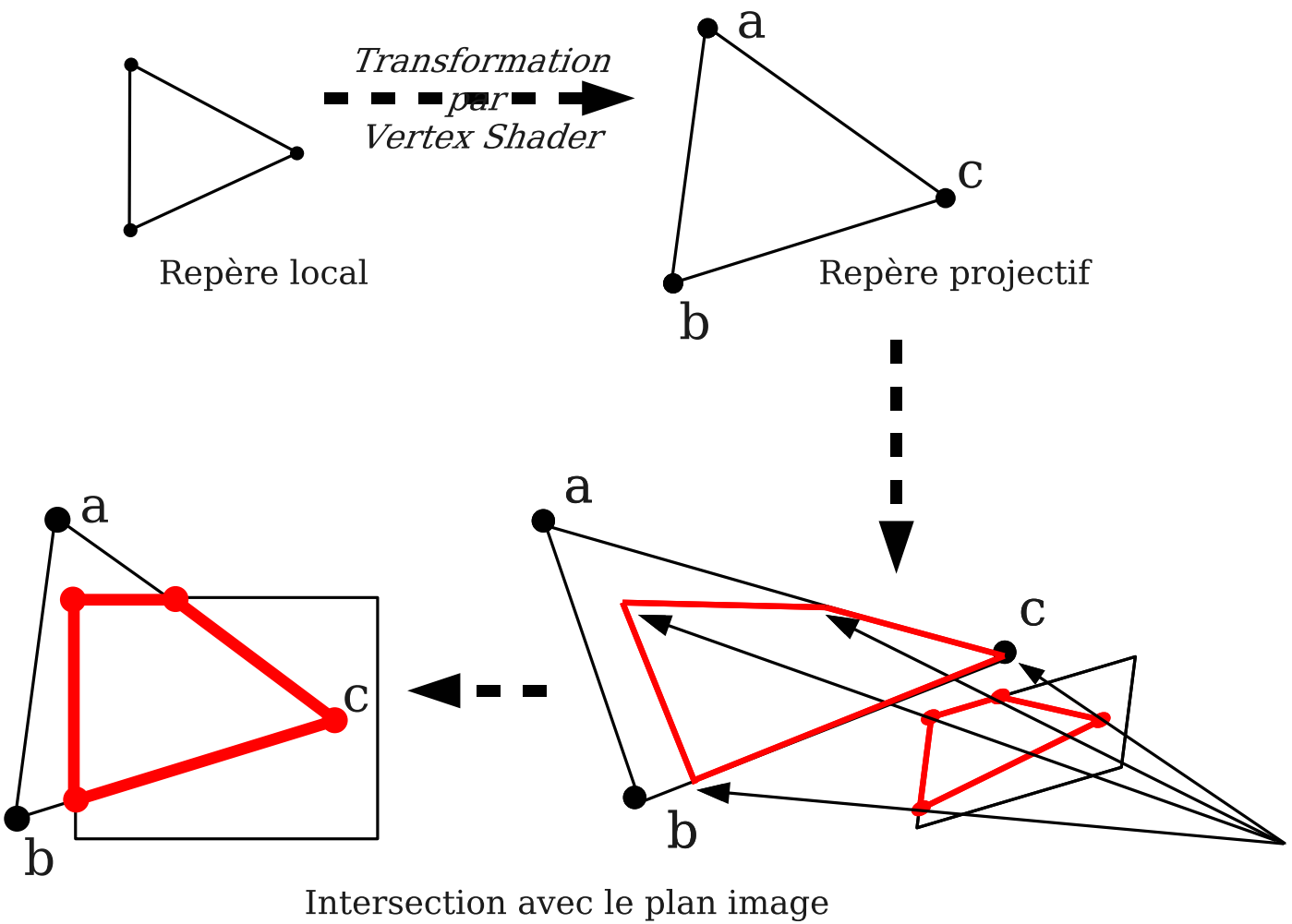
- un shader n'est qu'une étape d'un calcul,
- il doit traiter ses entrées (les résultats de l'étape précédente),
- il doit produire certains résultats (pour les étapes suivantes),

mais les shaders peuvent faire d'autres choses en plus . . .

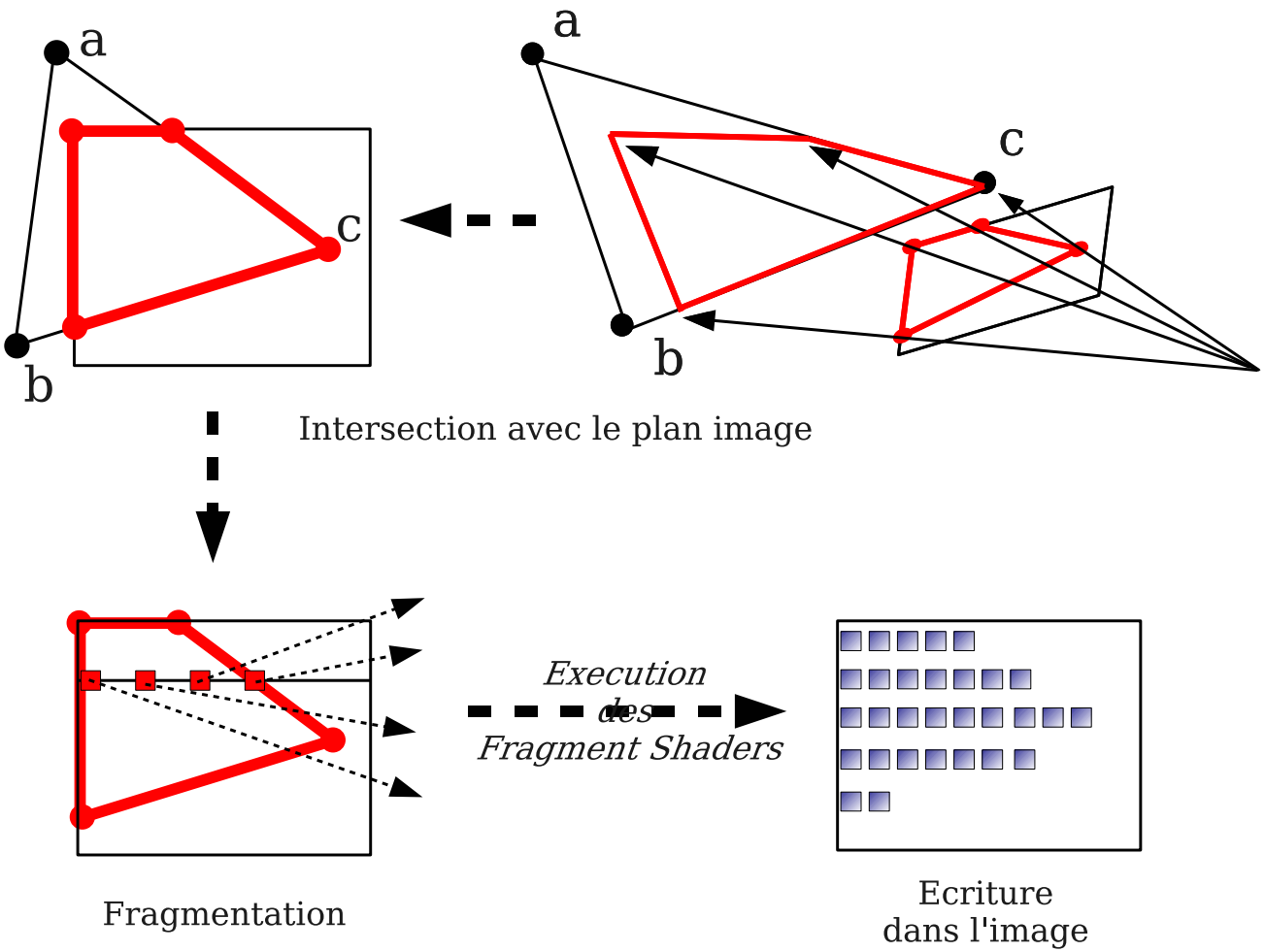
Shaders et pipeline graphique



Shaders et pipeline graphique



Shaders et pipeline graphique



Vertex Shaders : qu'est ce que c'est ?

vertex

sommet de la primitive + tous ses attributs :

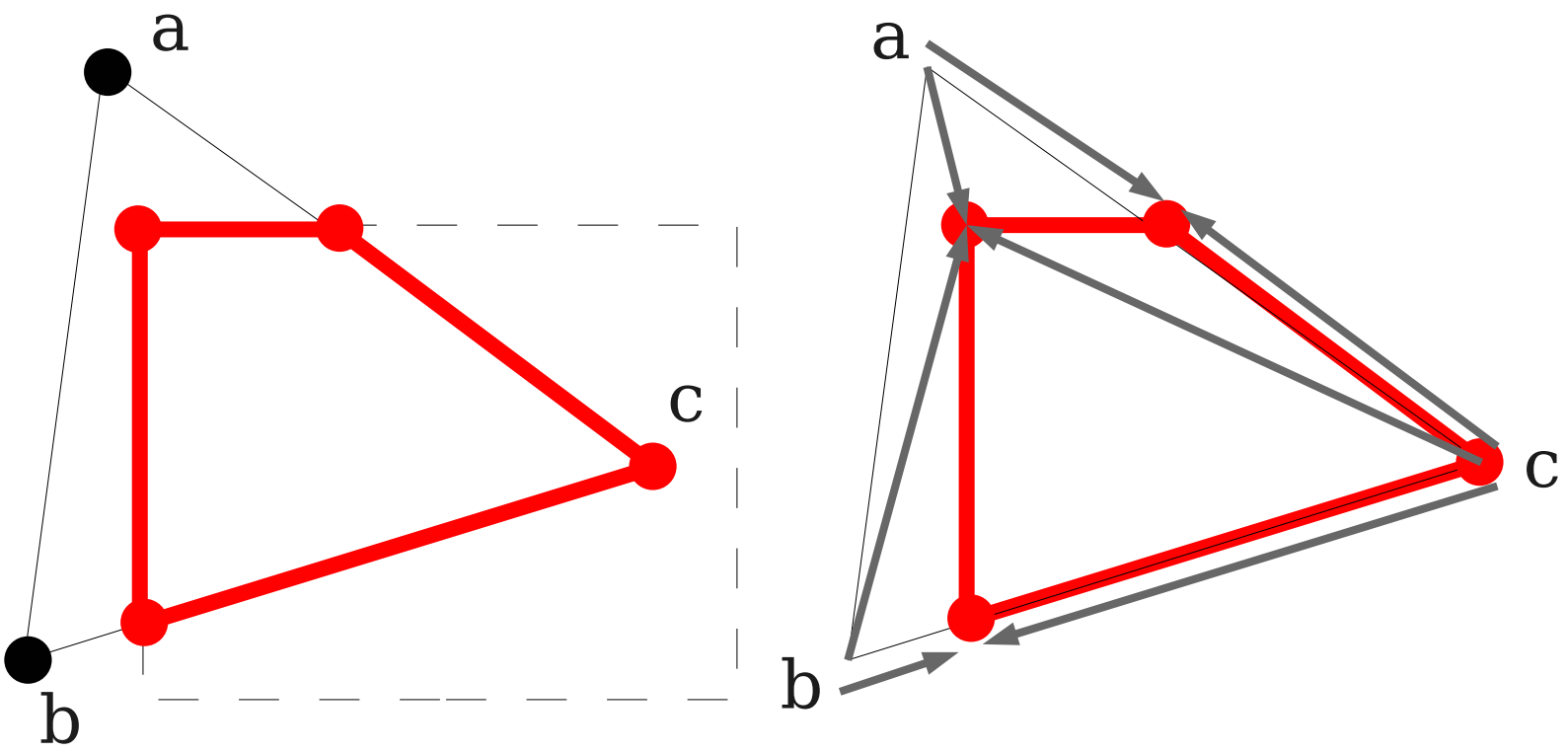
- position,
- couleur,
- matière,
- normale,
- attributs définis par l'application.

attention !

tous les attributs seront interpolés.

Interpolation des attributs : pourquoi ?

le pipeline peut créer de nouvaux sommets . . .



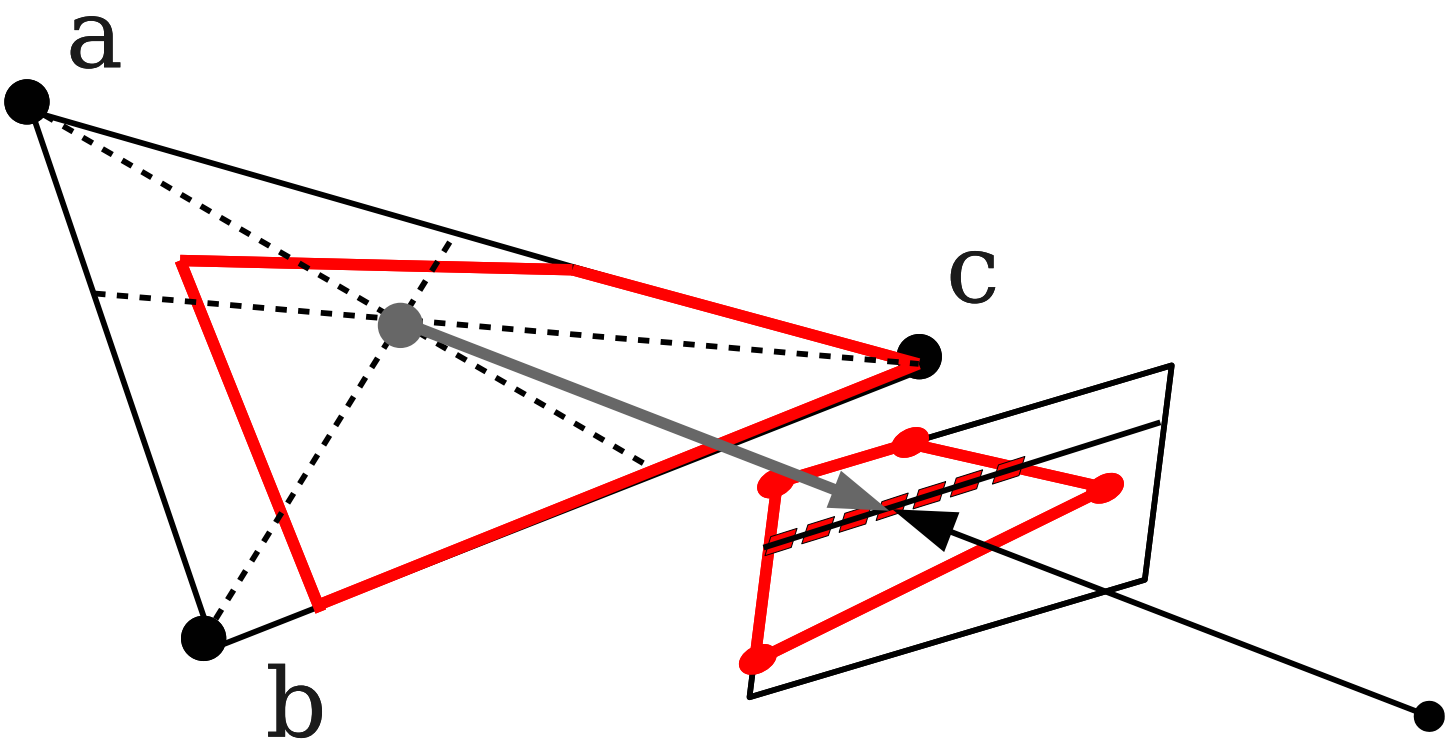
Fragment Shaders : qu'est ce que c'est ?

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).

Fragment 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 aux paramètres définis par l'application,
- accès aux attributs définis par l'application.

Shaders : comment ça marche ?

création des shaders :

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

les shaders ne sont finalement que des fonctions du pipeline.

Shaders : comment ça marche ?

création du programme complet :

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

il faut linker les fonctions pour obtenir un programme utilisable !

Shaders : comment ça marche ?

utilisation du programme :

1. `glUseProgram( )`
2. fixer les valeurs des paramètres,
3. dessiner la géométrie (attributs associés aux sommets ?).

vérifications :

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

Shaders : paramètres ?

paramètres uniform / constants :

paramètres généraux, constants pour tous les sommets d'un objet, initialisés par l'application.

paramètres attribute :

associé à chaque sommet (couleur, normale, etc.)
initialisés par l'application lors du `draw( )`.

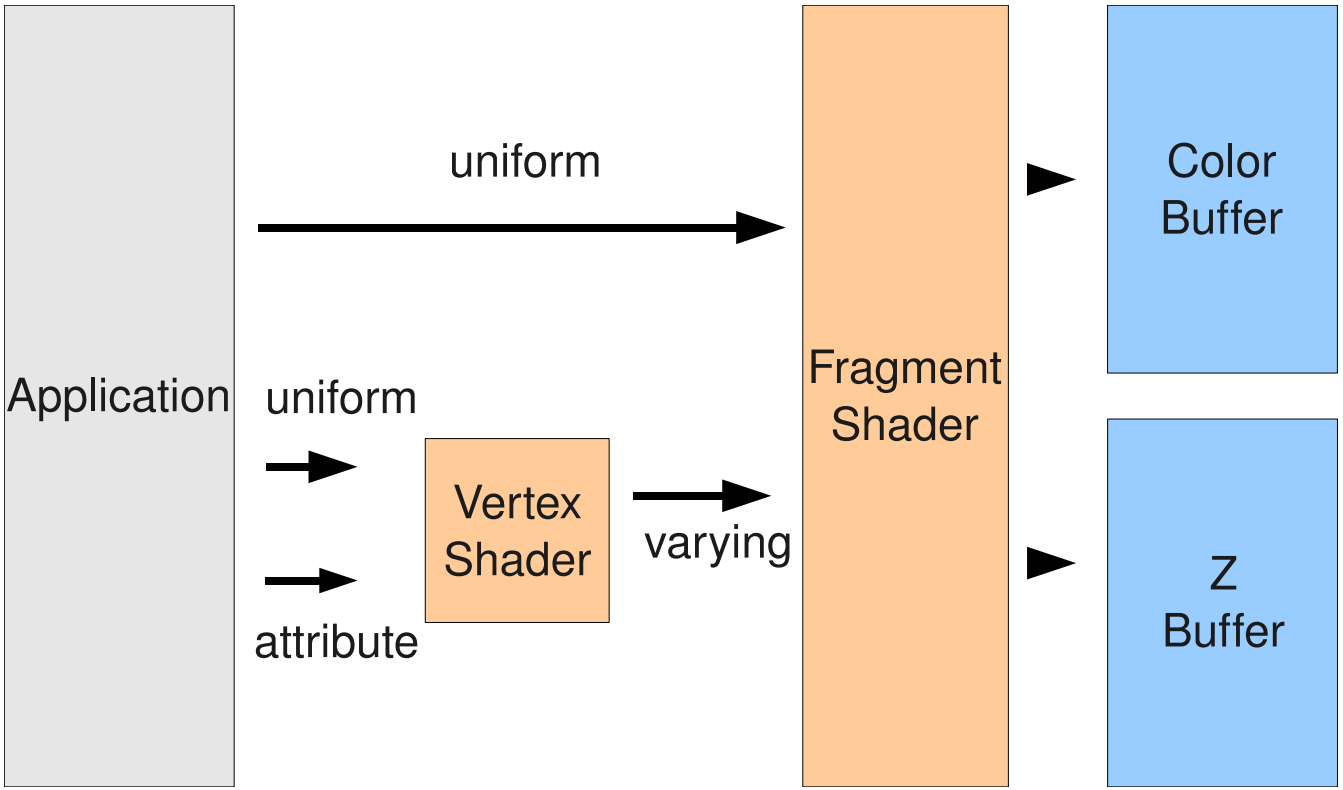
paramètres varying :

résultat d'un shader transmis au reste du pipeline (et aux autres shaders)
L'application ne peut pas les définir explicitement.

déclaration :

par le / les shader(s).

Shaders : paramètres et attributs



Shaders : valeur des paramètres

paramètres uniform :

- `location= glGetUniformLocation(program, "uu")`
- `glUniformXXX(location, valeur)`
- `glUniformMatrixXXX(location, valeur)`

paramètres attribute :

- `location= glGetAttribLocation(program, "aa")`
- `glVertexAttribXXX(location, xxx)`
- `glVertexAttribPointer( )`

Shaders : exemple (vertex)

```
// Copyright (c) 2003-2004: 3Dlabs, Inc.
uniform mat4 mvpMatrix; // model * view * projection
uniform float Time;             // updated each frame by the application
uniform vec4  Background;       // constant color equal to background
in vec4  Position;       // position
in vec3  Velocity;       // initial velocity
in float StartTime;      // time at which particle is activated
in vec3 Color;     // color
out vec4 vertexColor;

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

    if (t >= 0.0)
    {
        vertex    = Position + vec4 (Velocity * t, 0.0);
        vertex.y -= 4.9 * t * t;
        vertexColor   = Color;
    }
    else
    {
        vertex  = Position;       // Initial position
        vertexColor = Background;      // "pre-birth" color
    }
    gl_Position  = mvpMatrix * vertex;
}
```

Shaders : exemple (vertex)

```
// application
glUseProgram(program);

GLint background= glGetUniformLocation(program, "Background");
glUniform4f(background, 0.0, 0.0, 0.0, 1.0);
GLint time= glGetUniformLocation(program, "Time");
glUniform1f(time, -5.0);
GLint mvp= glGetUniformLocation(program, "mvpMatrix");
glUniformMatrix4fv(mvp, 1, GL_TRUE, gk::Transform().matrix());

// vertex buffers
GLint velocity= glGetAttribLocation(program, "Velocity");
glBindBuffer(GL_ARRAY_BUFFER, buffer1);
glVertexAttribPointer(velocity, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(velocity);

GLint start= glGetAttribLocation(program, "StartTime");
glBindBuffer(GL_ARRAY_BUFFER, buffer2);
glVertexAttribPointer(start, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(start);

GLint position= glGetAttribLocation(rogram, "Position");
glBindBuffer(GL_ARRAY_BUFFER, buffer3);
glVertexAttribPointer(position, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(position);

glDrawArrays(GL_POINTS, 0, n);
```

Shaders : exemple (fragment)

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

in vec4 vertexColor;   // interpolated from vertex shader

layout(location= 0) out vec4 fragmentColor; // output color

void main (void)
{
    fragmentColor = vertexColor;
}
```

Shaders : demo !

Vertex Shaders : comment ça marche ?

- éxécuté sur chaque sommet
  (données fournies par l'application),
- doit calculer :
- `gl_Position= ModelViewProjectionMatrix * Position`
- tous les `varying` utilisés par le reste du pipeline (fragment shader).

`gl_Position` doit être dans l'espace projectif normalisé de la caméra.

Fragment Shaders : comment ça marche ?

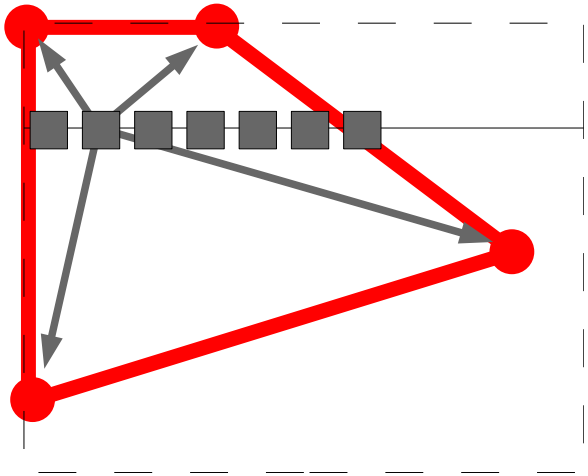
- éxécuté sur chaque fragment dessiné,
- doit calculer la couleur du fragment,
- travaille dans le repère de la fenêtre d'affichage (viewport).

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

in vec4 vertexColor;   // interpolated from vertex shader

layout(location= 0) out vec4 fragmentColor; // output color

void main (void)
{
    fragmentColor = vertexColor;
}
```

Shaders : comment ça marche ?

ne pas oublier !

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



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,

```
// simple vertex shader
#version 330

uniform mat4 mvpMatrix;
in vec4 position;

void main(void)
{
    gl_Position= mvpMatrix * position;
}
```

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,

```
// simple fragment shader
#version 330

layout(location= 0) out vec4 fragmentColor;

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

GLSL : Shading Language

proche du 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_ClipDistance[],

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 : mise au point

Mise au point de shader :

- gDEBugger http://www.gremedy.com/,
- glslDevil http://www.vis.uni-stuttgart.de/glsldevil/
- + IDE spécialisés (windows) : FX Composer, RenderMonkey,
- PIX (windows),
- apitrace https://github.com/apitrace/.

OpenGL : 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/