M2-Images

Rendu Temps Réel - Shaders

J.C. Iehl

September 23, 2015

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.),
- matériaux non réalistes (rendu expressif),
- plus grande liberté pour accéder aux données,
- traitement d'images,
- animation, déformation, etc.,

à faire autre chose ...

- *compute* shaders

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/modifier des primitives,
- *fragment 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 ?

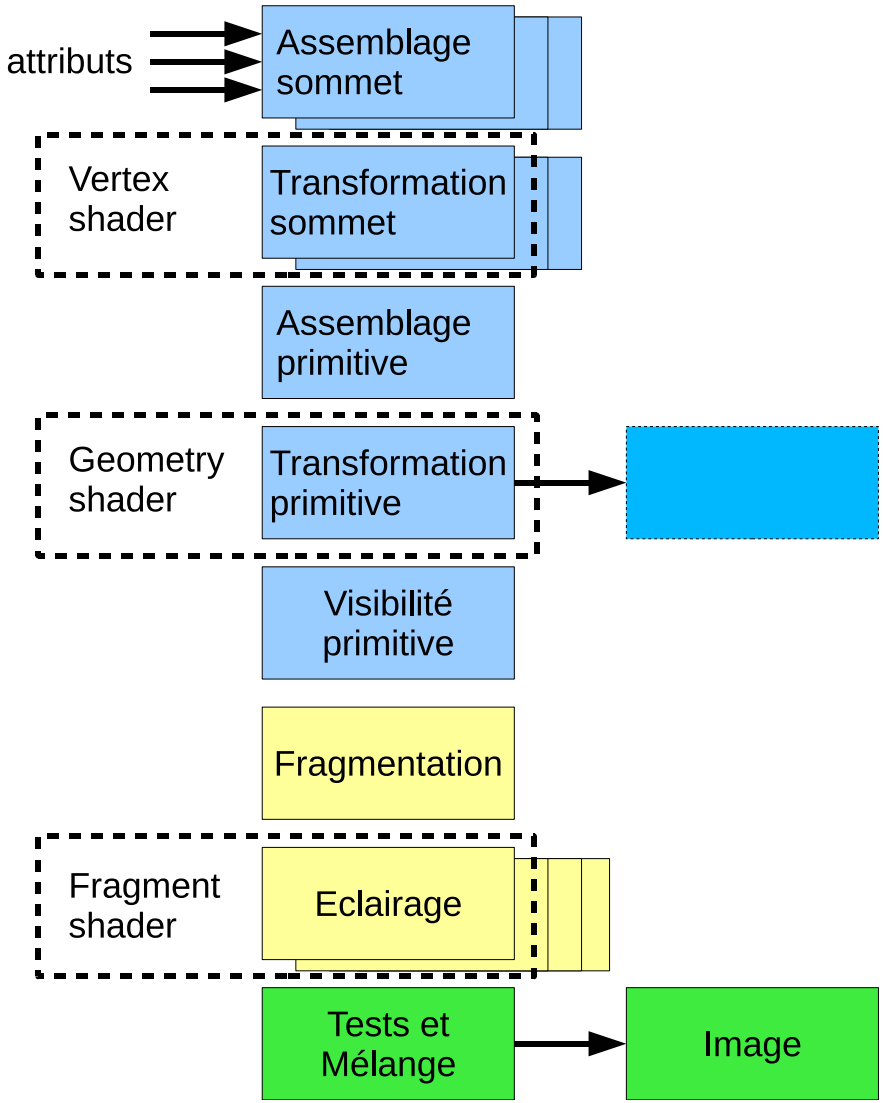
les shaders réalisent certaines étapes du pipeline graphique.

un shader a donc des obligations :

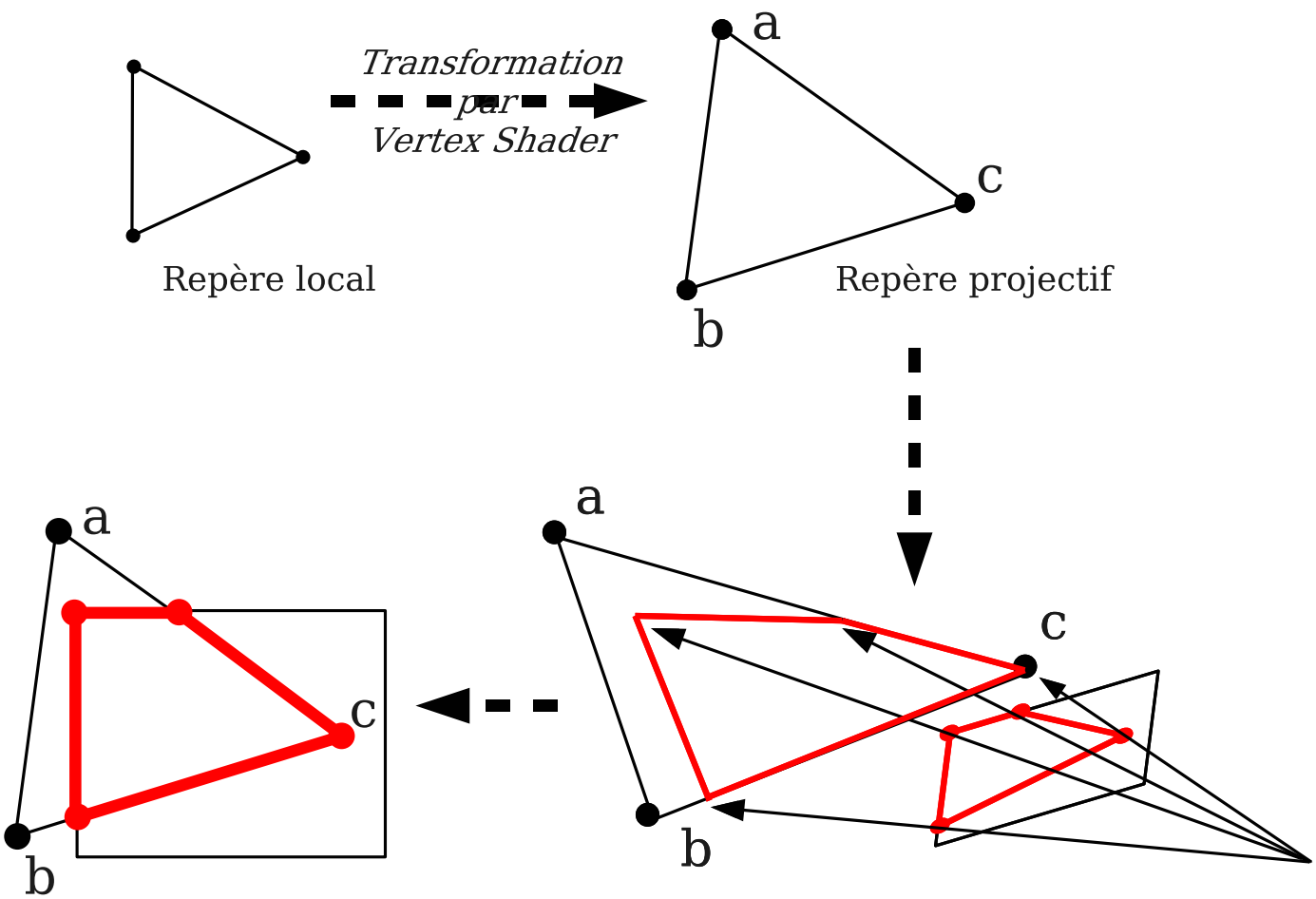
- un shader n'est qu'une étape du pipeline,
- il doit traiter ses entrées (les résultats de l'étape précédente),
- il doit produire certains résultats (pour les étapes suivantes).

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

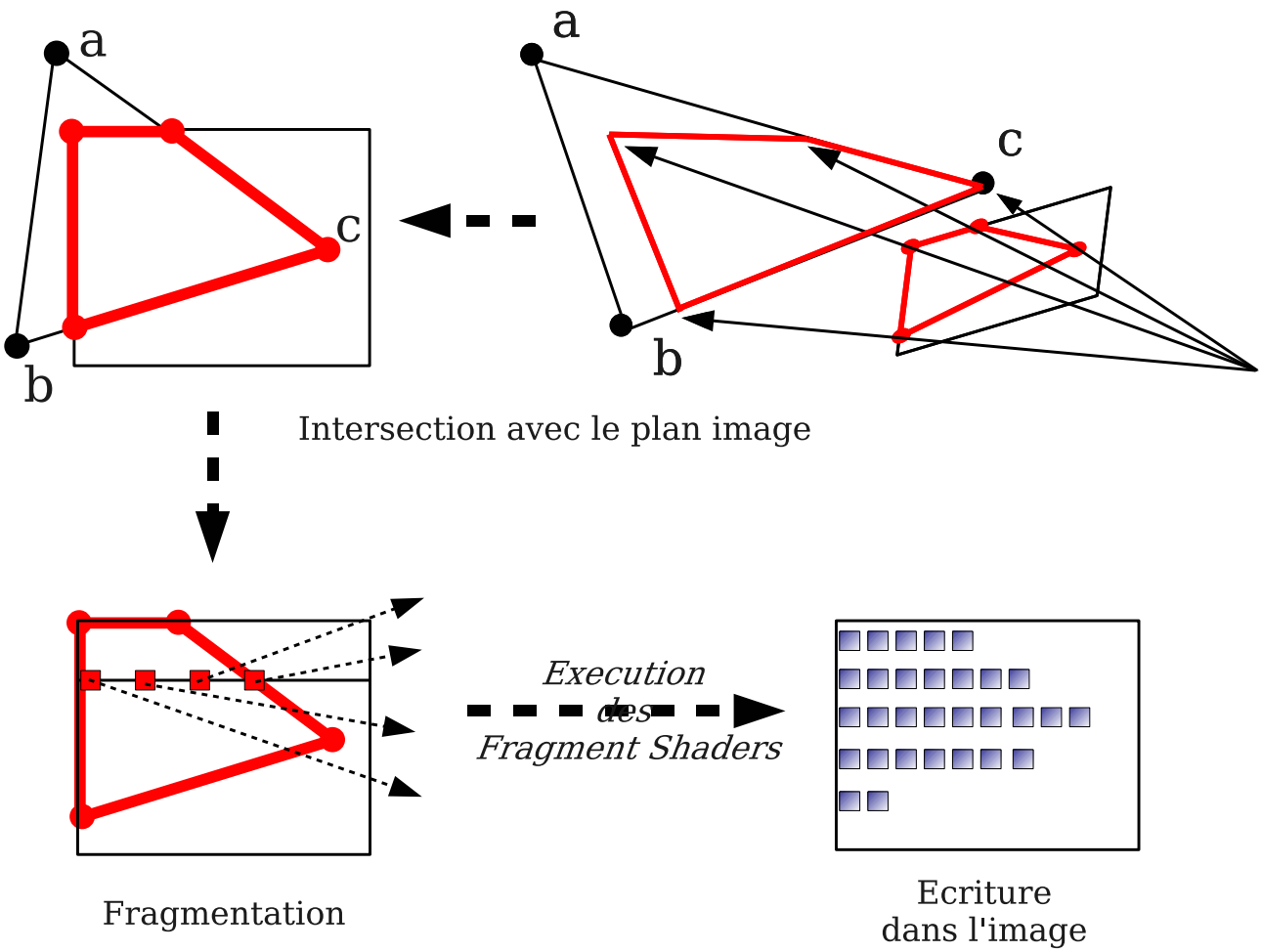
Shaders et pipeline graphique



Shaders et pipeline graphique



Shaders et pipeline graphique



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

vertex :

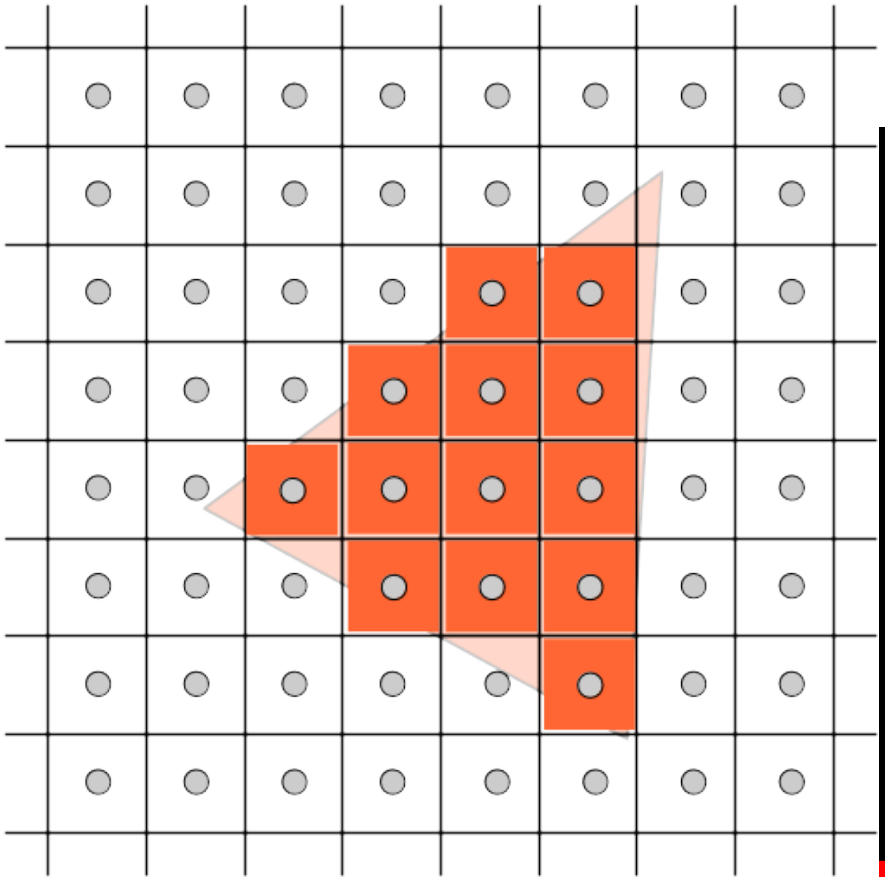
sommet de la primitive + tous ses attributs :

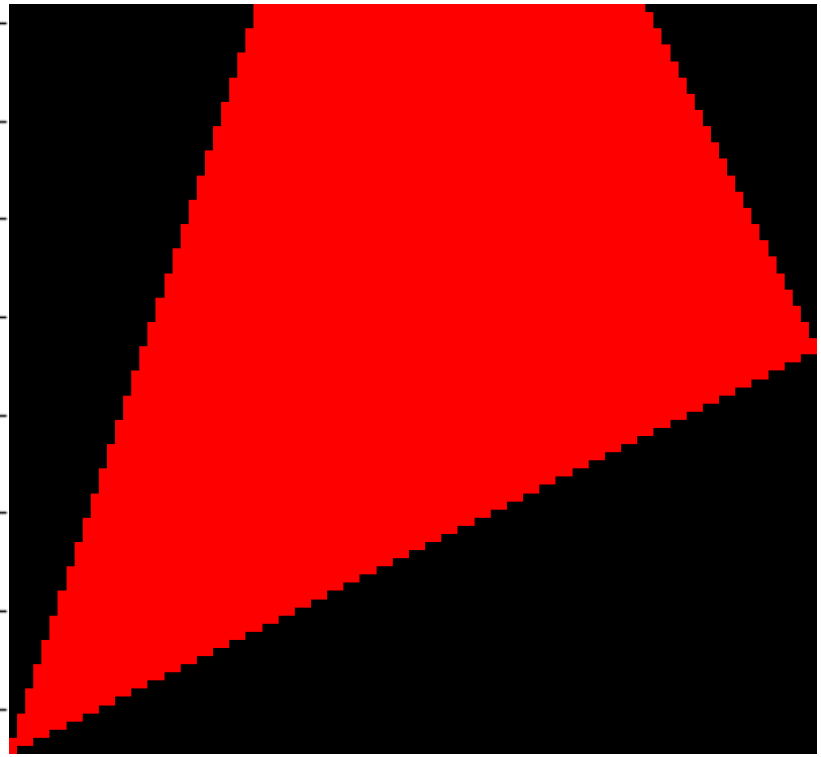
- position,
- couleur,
- matière,
- normale,
- attributs définis par l'application.

attention !

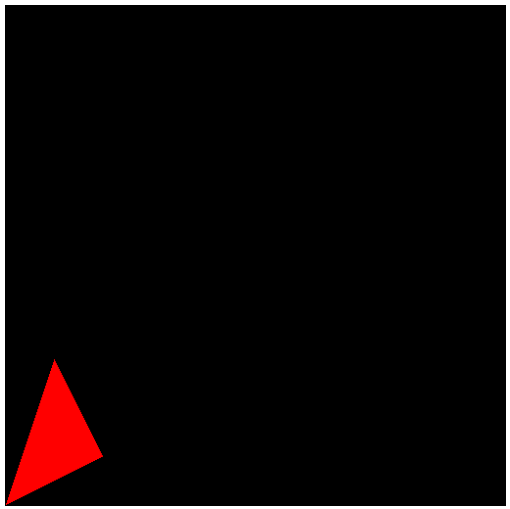
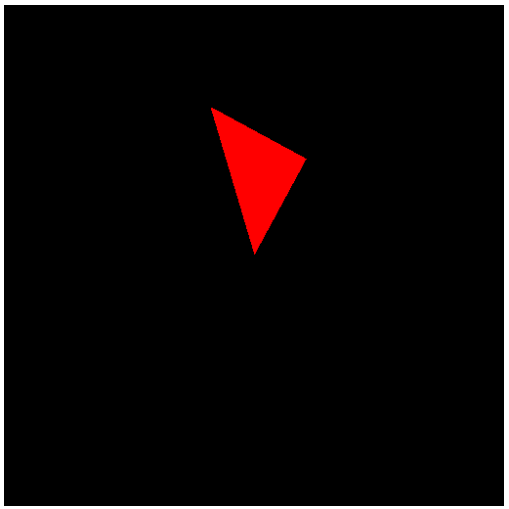
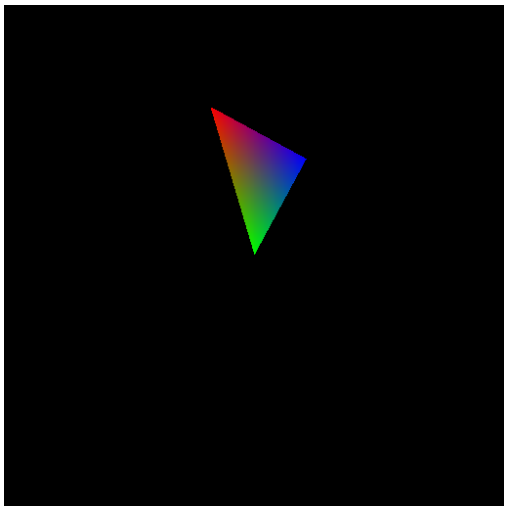
tous les attributs seront interpolés *linéairement* lors de la fragmentation.

Rappel : fragmentation





Rappel : fragmentation

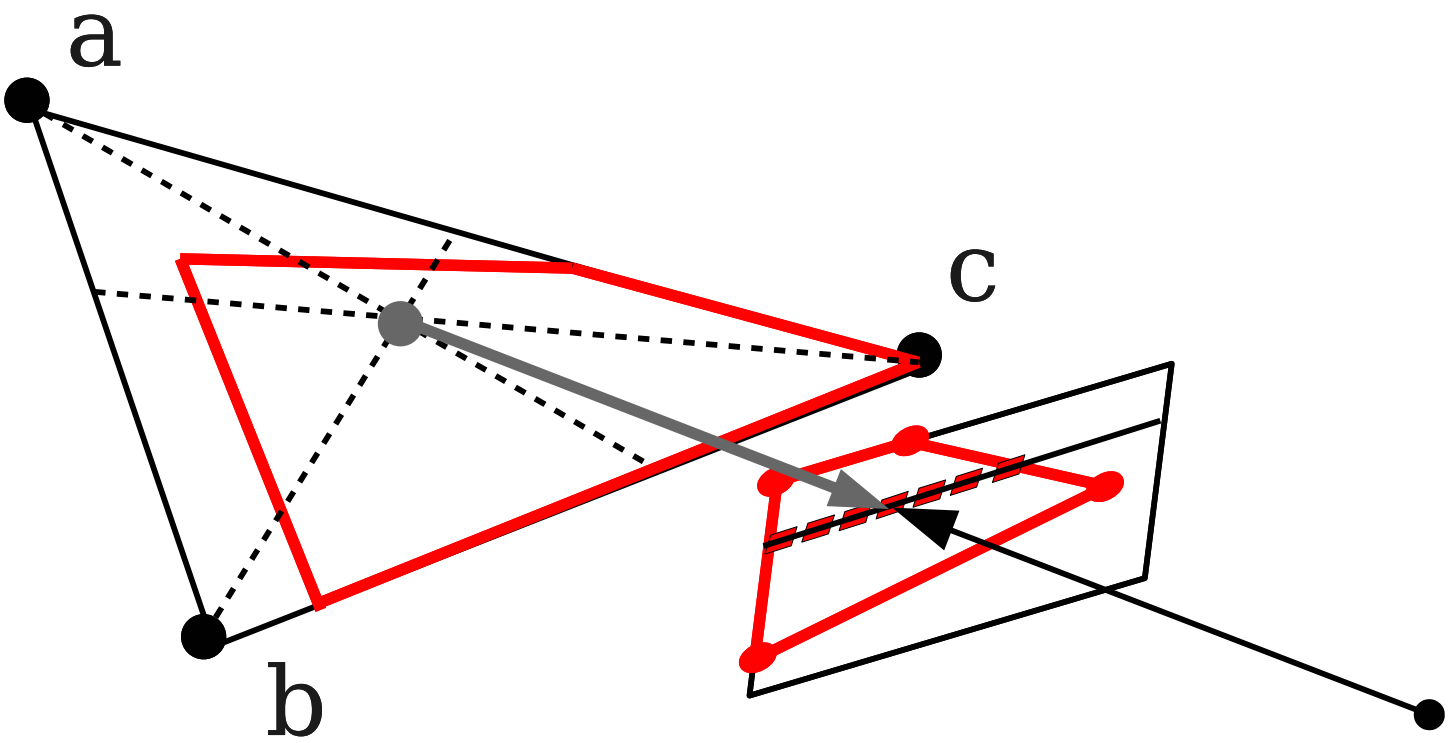
Fragment Shaders : qu'est ce que c'est ?

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.

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 : paramètres ?

paramètres uniform / constants :

paramètres généraux, constants pour toutes les primitives, initialisés par l'application.

paramètres attribute :

associés à chaque sommet (couleur, normale, etc.), initialisés par l'application.

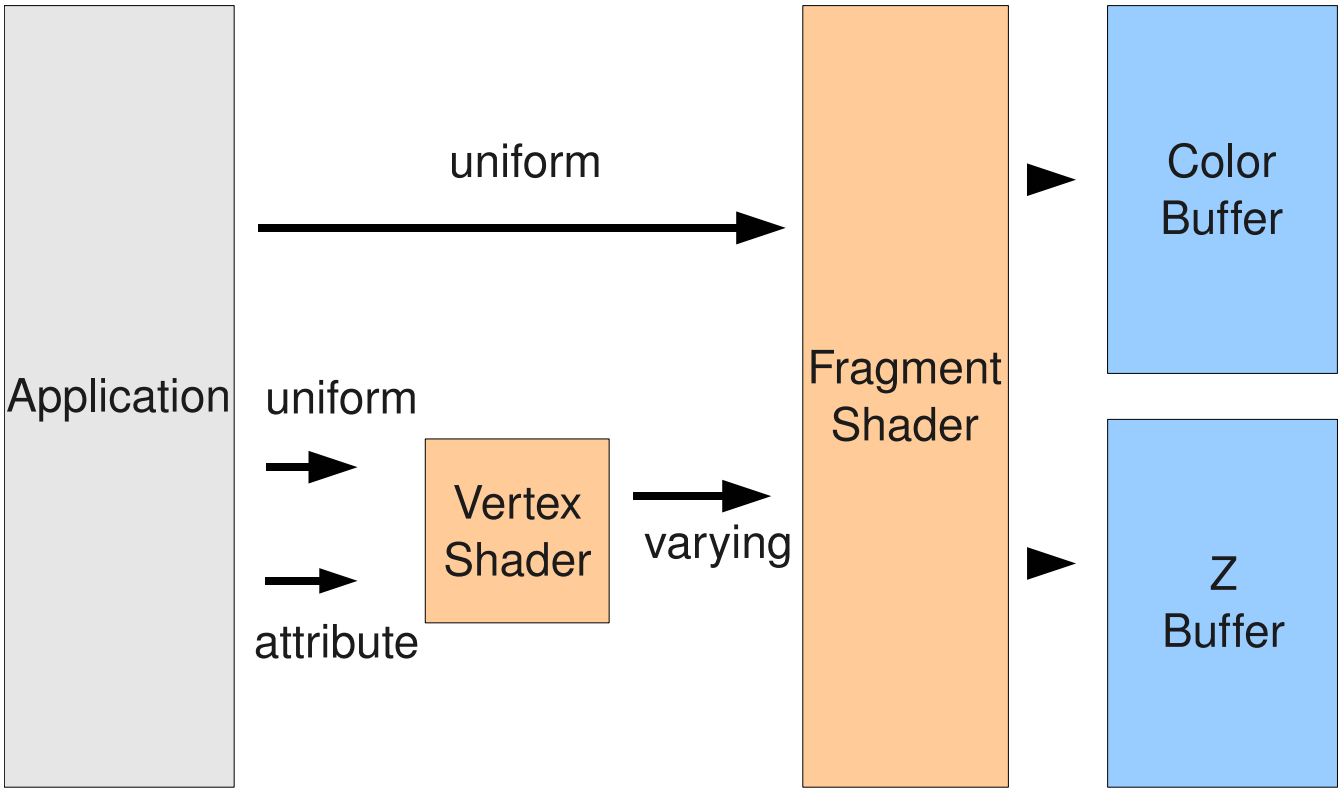
paramètres varying :

résultats 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 : 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;

    vertex = Position;
    vertexColor = Background;
    if (t >= 0.0)
    {
        vertex      = Position + vec4 (Velocity * t, 0.0);
        vertex.y   -= 4.9 * t * t;
        vertexColor = Color;
    }

    gl_Position     = mvpMatrix * vertex;
}
```

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;
}
```

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 homogène de la caméra.

Fragment Shaders : comment ça marche ?

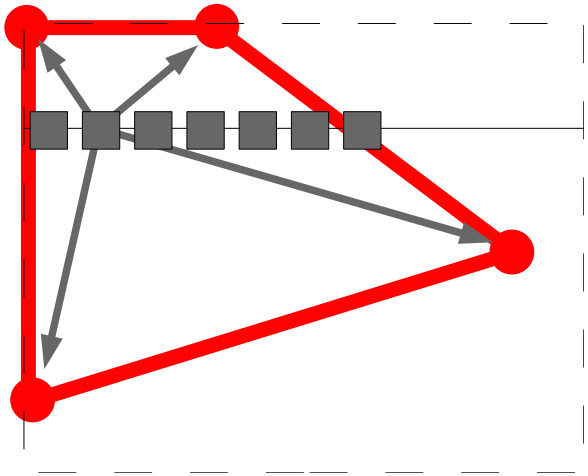
- éxécuté sur chaque fragment dessiné,
- doit calculer la couleur du fragment,
- travaille dans le repère de l'image (viewport).

```
// 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;
in vec3 normal;

out vec3 vertexColor;   // varying

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

Etape 6 : opérations sur les fragments

fragment shader :

- responsable de calculer la couleur du fragment,
- peut utiliser les paramètres `varying` déclarés par le vertex shader,

```
// simple fragment shader
#version 330

in vec3 vertexColor;     // varying, cf declaration vertex

layout(location= 0) out vec4 fragmentColor;

void main(void)
{
    fragmentColor= vec4( abs(vertexColor), 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_FragDepth, gl_FragCoord,
- etc.,

cf. GLSL specification, chapitre 7.

OpenGL : mise au point

Mise au point de shader :

- shader_kit dans gKit2
- IDE spécialisés (windows) : FX Composer, RenderMonkey,
- CodeXL (AMD), Nsight (Nvidia),
- Apitrace https://github.com/apitrace/,
- VOGL https://github.com/ValveSoftware/vogl.

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 ?

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