gamagora

LES FORMATIONS PROFESSIONNELLES AUX MÉTIERS DU JEU VIDÉO

UNIVERSITÉ LUMIÈRE LYON 2

# Shaders

## UE Rendu - Parcours Dev

*Jean-Philippe.Farrugia@LIRIS.CNRS.fr*

LIRIS

UCBL

# Sources

- Cours Alexandre Meyer et J.C Iehl.
- GLSL Programming Guide
- MSDN
- Sources web diverses


# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Pipeline théorique

- Fonctionnalité de base d'un GPU :
  - Calcule une image à partir d'un flot de données 3D.
- Principales étapes :
  - Transformation objet - monde - vue
  - Calcul éclairement
  - Assemblage des primitives
  - Rasterization ( = Discretisation)
  - Calcul couleur des pixels.

LIRIS

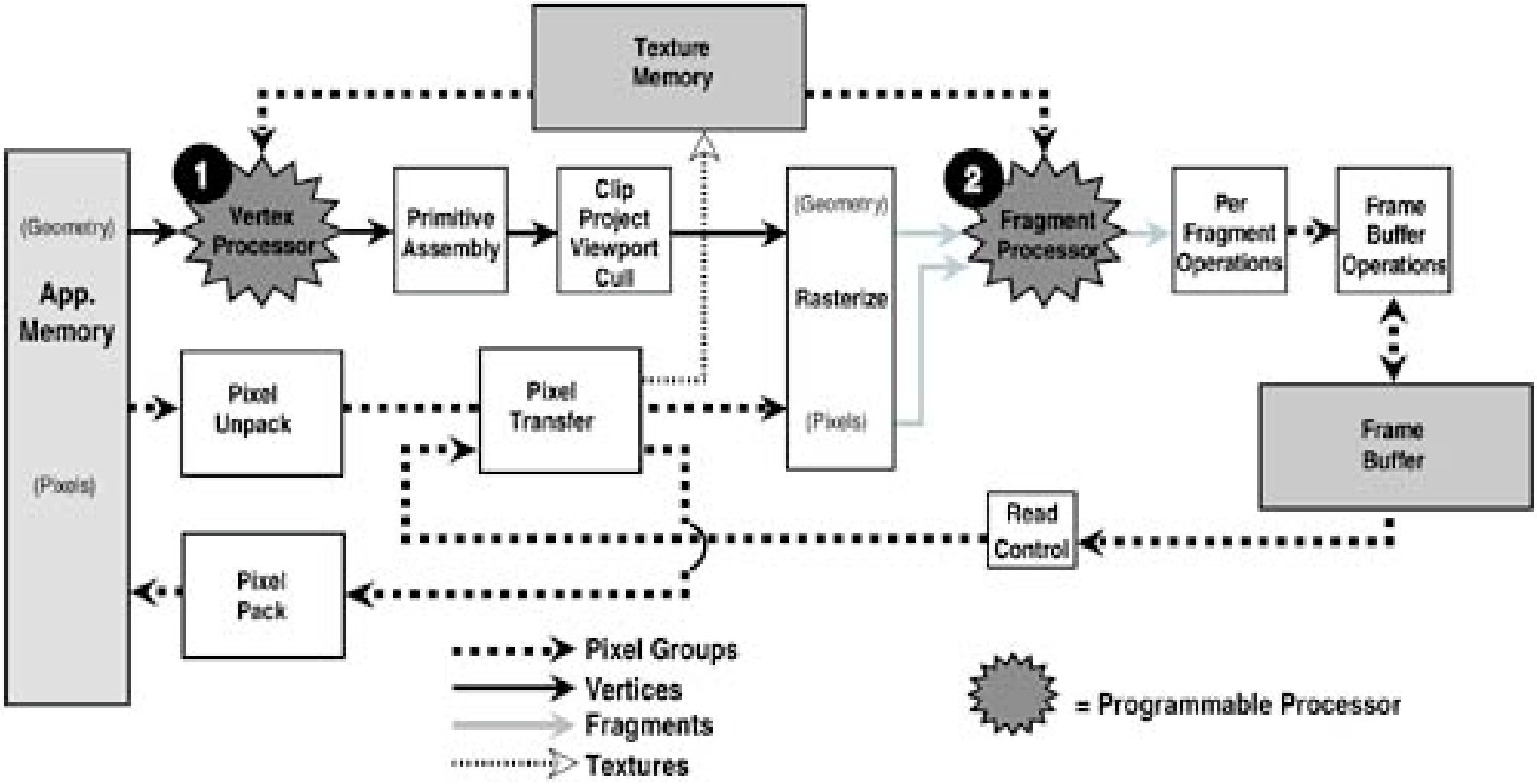
# Pipeline théorique

- Fonctionnalité de base d'un GPU :
  - Calcule une image à partir d'un flot de données 3D.
- Principales étapes :
  - Transformation objet - monde - vue
  - Calcul éclairement
  - Assemblage des primitives
  - Rasterization ( = Discretisation)
  - Calcul couleur des pixels.

# Pipeline théorique

- Fonctionnalité de base d'un GPU :
  - Calcule une image à partir d'un flot de données 3D.
- Principales étapes :
  - Transformation objet - monde - vue
  - Calcul éclairement
  - Assemblage des primitives
  - Rasterization ( = Discretisation)
  - Calcul couleur des pixels.

# Pipeline simplifié



# Unités programmables

- Transformations modelview et projection :
  - Vertex Processor.
- Couleur des fragments :
  - Fragment Processor.
- Assemblage des primitives :
  - Geometry processor.
- Programmes associés :
  - Vertex/Fragment/Geometry Shaders


# Comparaisons

- Programme CPU :
  - Traite l'ensemble de ses données en une fois.
  - L'ensemble des données est accessible n'importe quand, dans n'importe quel ordre.
  - Le format de sortie est libre.
- Programme GPU :
  - Transforme un flux de données en un autre, tous deux de formats imposés.

LIRIS

# Programmation GPU

- Penser Stream processing !
  - Une seule donnée (vertex - fragment) à la fois.
  - Le frame buffer n'est pas accessible directement.
  - Des paramètres peuvent être transmis depuis l'application mais...
  - ... rien n'est modifiable en dehors des valeurs de sorties imposées.


# Pourquoi ?

- Ouvre des possibilités considérables :
  - Matériaux réalistes.
  - Ajouts de détails.
  - Eclairements évolués.
  - Rendu expressif (toon shading...).
  - Traitement d'images / post processing.
  - Animation - Déformation.
  - ...

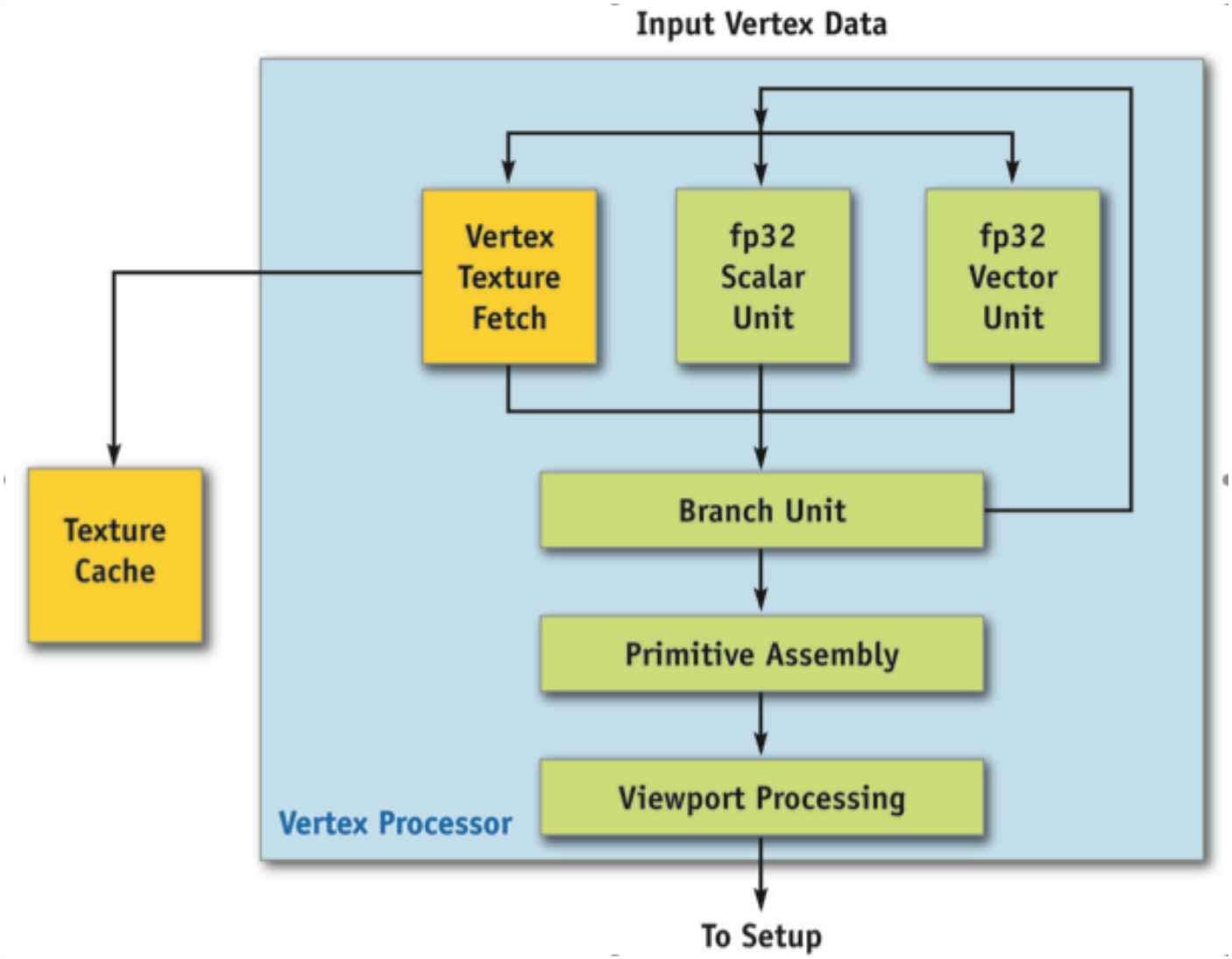
# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Plan

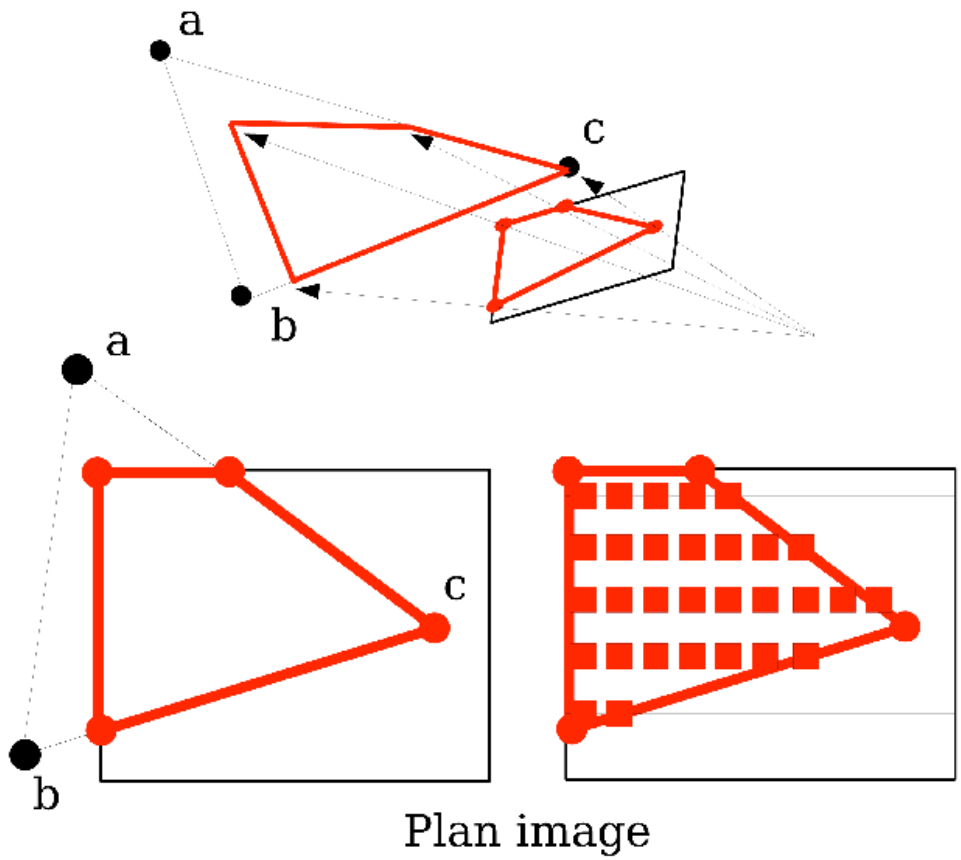
- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Vertex Processor



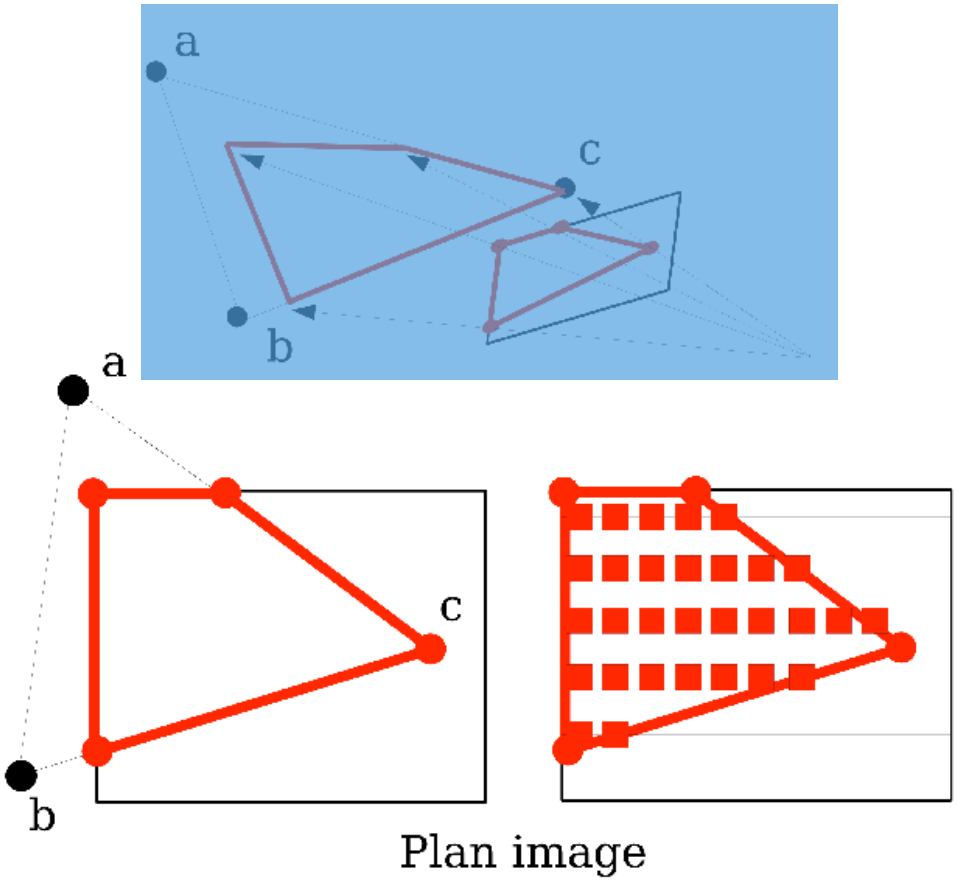
# Vertex Processor

- Reçoit les vertex de l'application.
- Transforme les vertex pour l'affichage.
- Calcule éventuellement leurs caractéristiques.



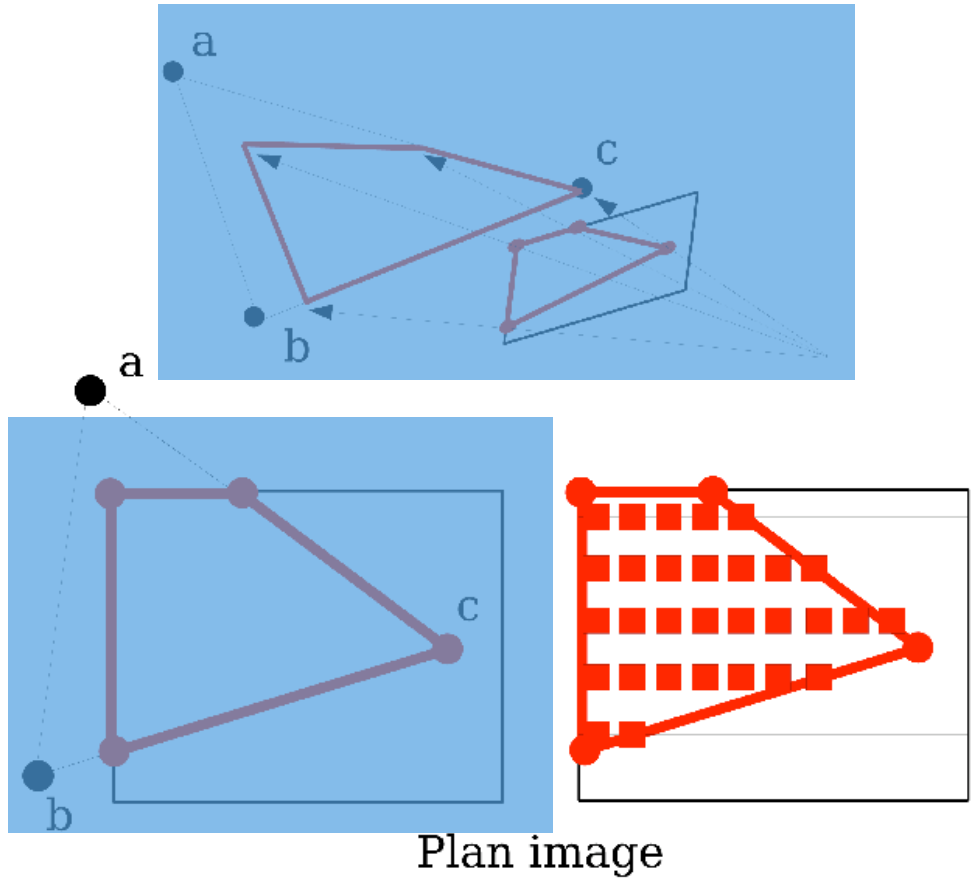
# Vertex Processor

- Reçoit les vertex de l'application.
- Transforme les vertex pour l'affichage.
- Calcule éventuellement leurs caractéristiques.



# Vertex Processor

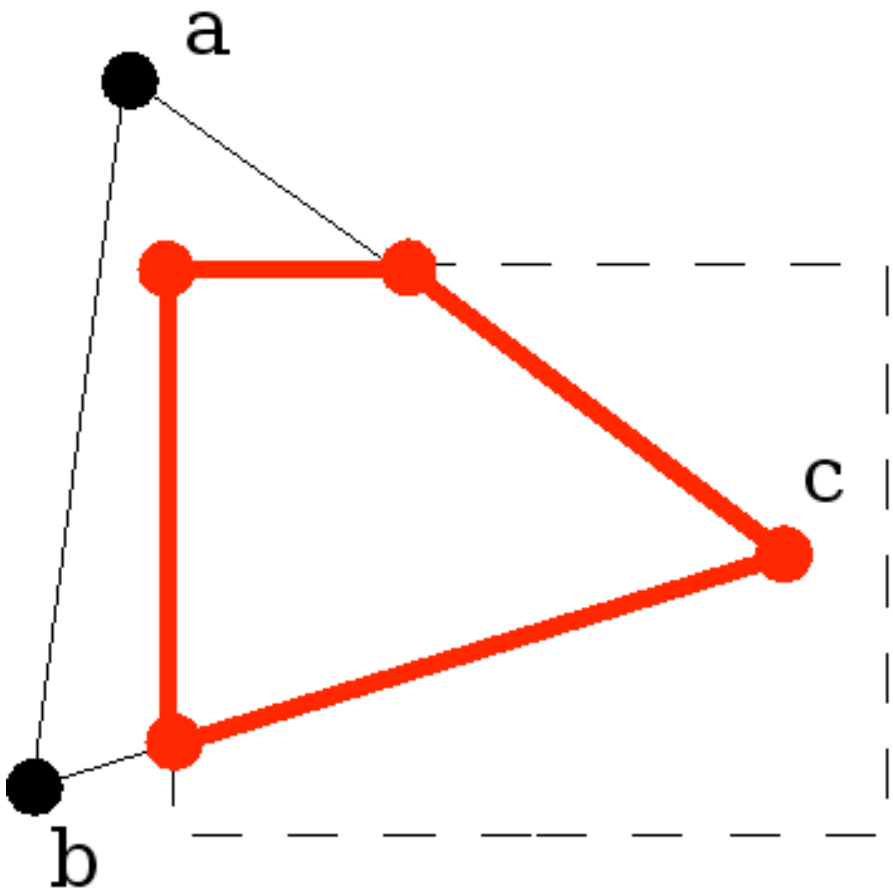
- Reçoit les vertex de l'application.
- Transforme les vertex pour l'affichage.
- Calcule éventuellement leurs caractéristiques.

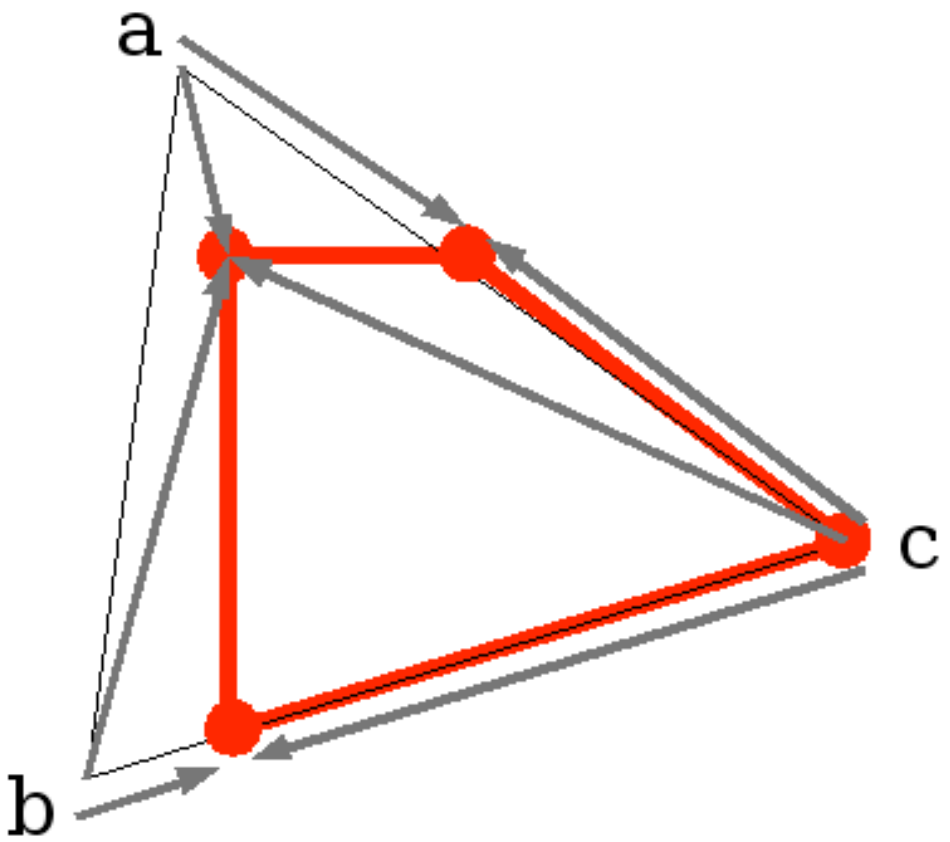


# Vertex

- Sommet de la primitive + attributs
  - Position
  - Couleur
  - Matière
  - Normale
  - Autres définis par l'application
- Ces attributs seront interpolés dans le pipeline.

# Interpolation des attributs



# Vertex Shader

- En entrée :
  - Un vertex avec ses attributs.
- En sortie :
  - Une liste d'attributs pour ce vertex :
    - Nouvelle position.
    - Normale.
    - Couleur.
    - + tout traitement nécessaire...



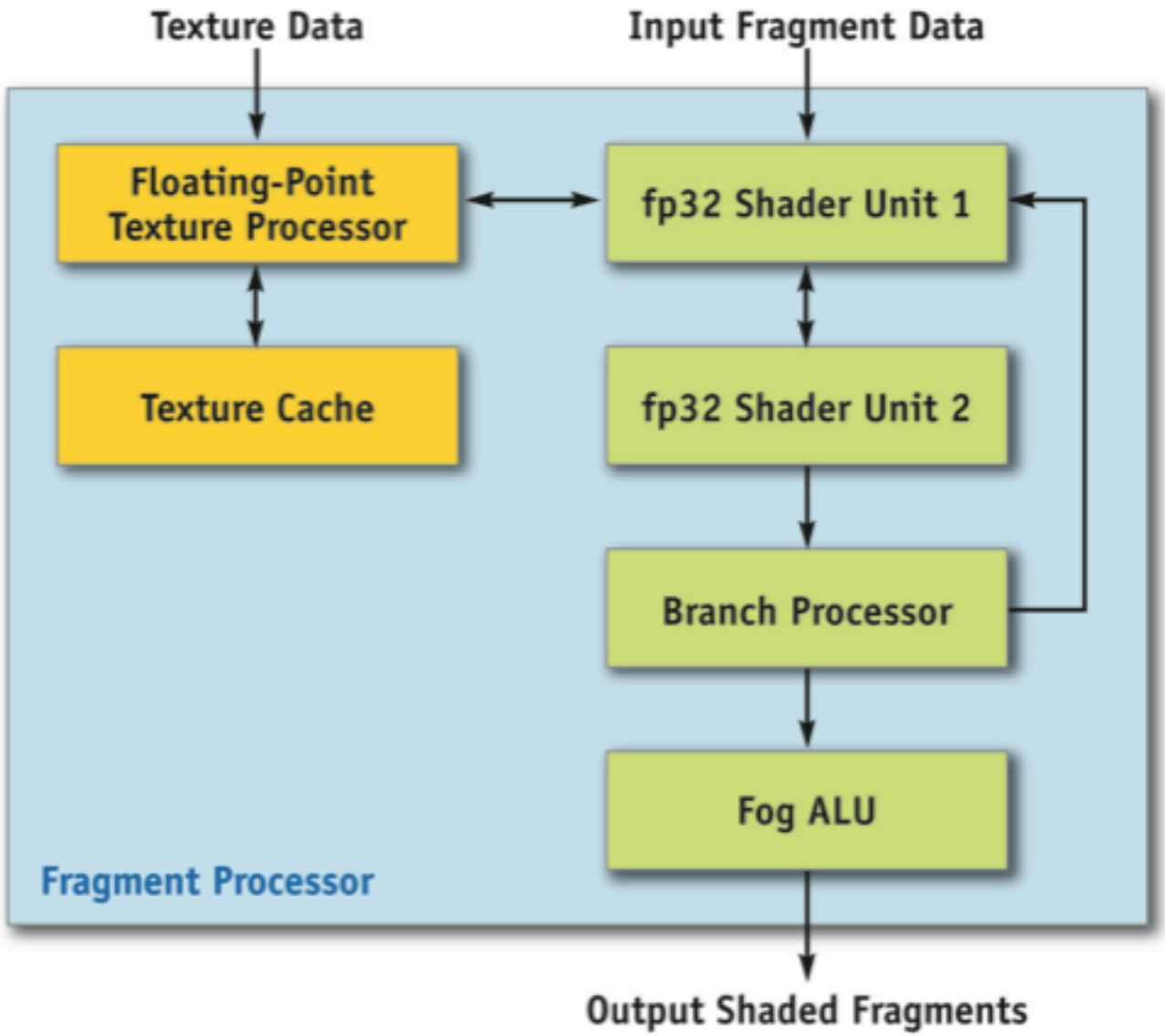
# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Plan

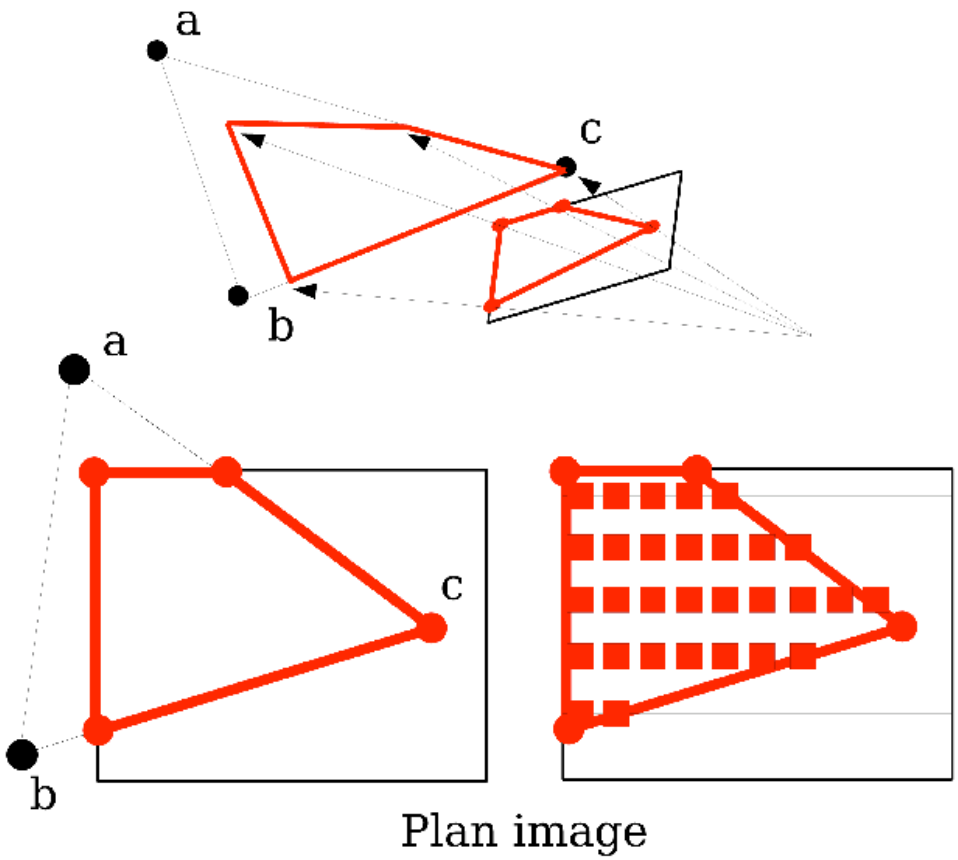
- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Fragment Processor



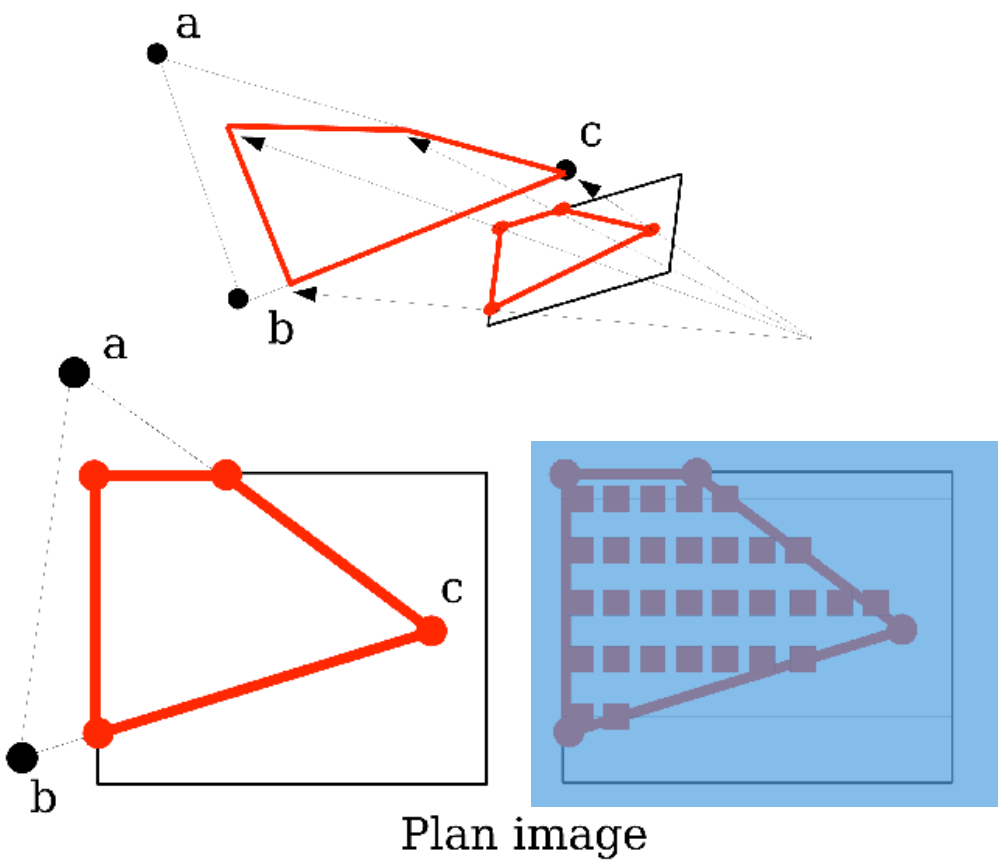
# Fragment Processor

- Reçoit les données du Rasterizer.
- Transforme le fragment en pixel
  - =>Calcule l'apparence finale.



# Fragment Processor

- Reçoit les données du Rasterizer.
- Transforme le fragment en pixel
  - =>Calcule l'apparence finale.



# Fragment

- Elément de l'image + tout ses attributs
  - Position 3D, distance à la caméra.
  - Matière, couleur, transparence.
  - Autres définis par l'application...
- Ces attributs ont été définis par le Vertex Shader puis interpolés par le Rasterizer.
- Attention : il est impossible affecter un attribut à un fragment directement !

# Fragment Shader

- En entrée :
  - Un fragment provenant du Rasterizer.
- En sortie :
  - Une liste de caractéristiques pour ce fragment.
    - Couleur.
    - Profondeur.
    - Transparence.
    - + Autres...


# Particularités

- Un Fragment shader peut avoir une sortie multiple :
  - Multiple Render Target.
  - Ecriture de 4 fragments dans 4 buffers distincts.
- Un Fragment shader n'écrit pas nécessairement dans le framebuffer visible.
  - RenderBuffers, FBOs...

# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Plan

- GPU - Pipeline graphique - rappels.
- Shaders
  - Vertex Shaders
  - Pixel Shaders
- API existantes
  - GLSL
  - Cg - HLSL

# Notions globales

- Syntaxe proche du C++.
- Types de base : scalaires, matrices.
- Samplers : accès au texture.
- Accès au contexte OpenGL.
- Accès aux paramètres définis par l'application.

# Création d'un shader

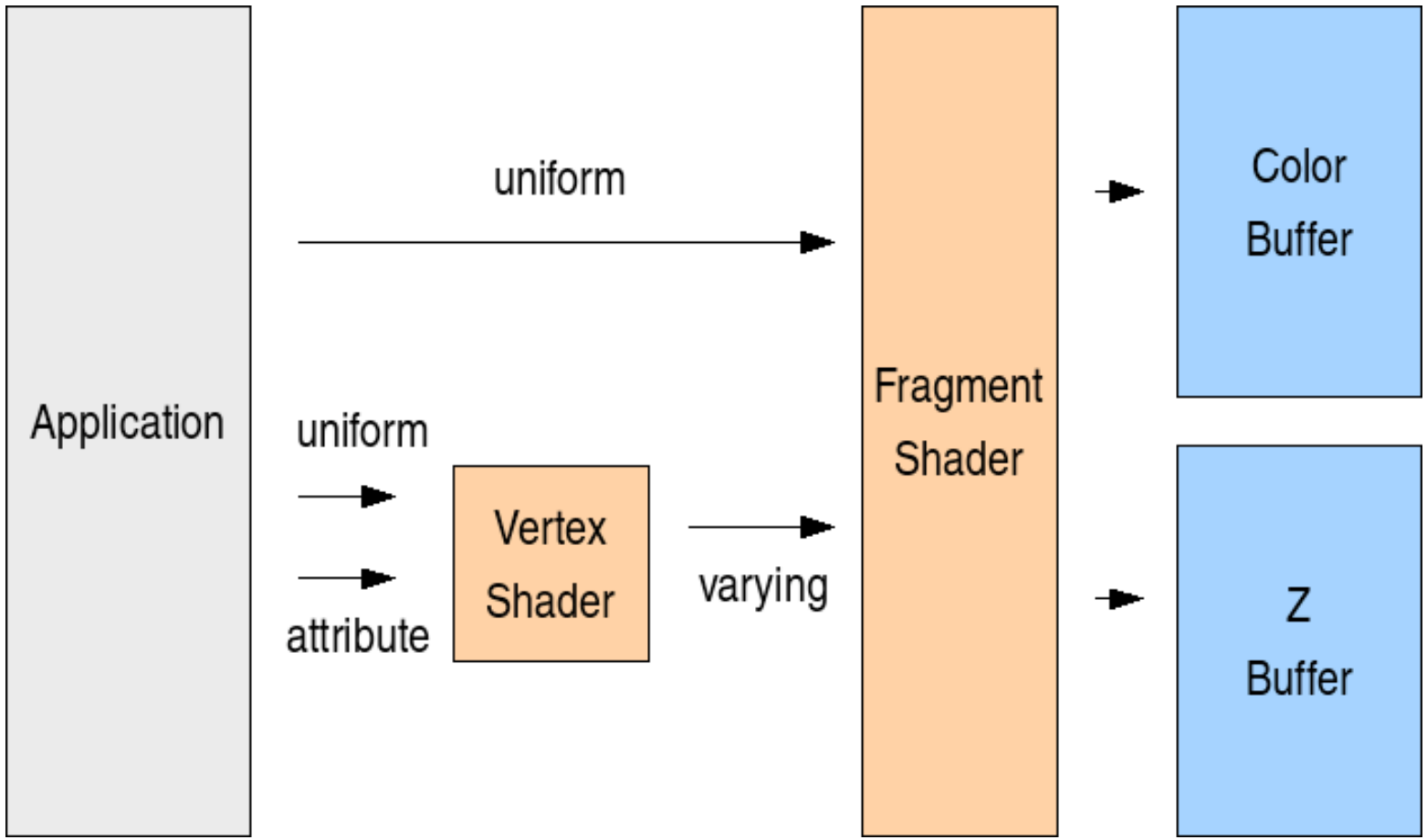
- Création d'un shader.
  - Un shader est une fonction.
  - Chargement du source depuis l'application.
  - Compilation.
- Création d'un programme
  - Il faut deux shaders pour un programme.
  - Edition de liens (link) des deux objets pour obtenir un progamme.



# Paramètres

- 4 types de paramètres :
  - Uniform : définis par l'application pour chaque primitive.
  - Attribute : définis par l'application pour chaque vertex.
  - Varying : définis par le vertex shader, puis interpolés lors de la fragmentation.

# Paramètres


Application
uniform
uniform
attribute
Vertex
Shader
varying
Fragment
Shader
Color
Buffer
Z
Buffer

# API existantes

- Cg (nVidia)
- GLSL (3DLabs)
- HLSL (Microsoft - nVidia)

# C for graphics

- Avantages :
  - Indépendant de l'API 3d (direct3D ou OpenGL).
  - Support actif.
  - Nombreux outils.
- Inconvénients :
  - Plus performant sur carte nVidia...
  - Outils annexes disponibles sous windows principalement.


# GL Shading Language

- Avantages :
  - Utilisation plus simple que Cg.
  - Multi-plateformes, multi-matériel.
- Inconvénients :
  - OpenGL pas toujours bien supporté...
  - Pas vraiment d'outils de conception.



# High Level SL

- Conçu par Microsoft et nVidia.
- De loin le plus utilisé en développement de jeux vidéos.
- Outils de création et de débug performants.
- Exclusif windows - directX...


# Exemple GLSL

```
// 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;
}
```


# Exemple GLSL

```
// 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();
```

# Exemple GLSL

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

varying vec4 Color;

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

# Exemple HLSL

```
float4x4 modelviewproj;  // World * View * Projection transformation

struct VS_OUTPUT
{
    float4 position    : POSITION;   // vertex position
    float4 diffuse     : COLOR0;     // vertex diffuse color
};

VS_OUTPUT main(in float4 position : POSITION)
{
      VS_OUTPUT output;

    output.position= mul(position, modelviewproj);
    output.diffuse= float4(1.f, 0.f, 0.f, 1.f);

    return output;
}
```


# TP

- Ecrivez un shader “vision thermique” :
  - Chaque vertex possède une température comprise entre 0 et 45 degrés.
  - Selon la valeur de la température, la couleur du pixel va changer :
    - de 0 à T_MIN : bleu à jaune
    - T_MIN à T_MAX : jaune à rouge
    - > T_MAX :  rouge.