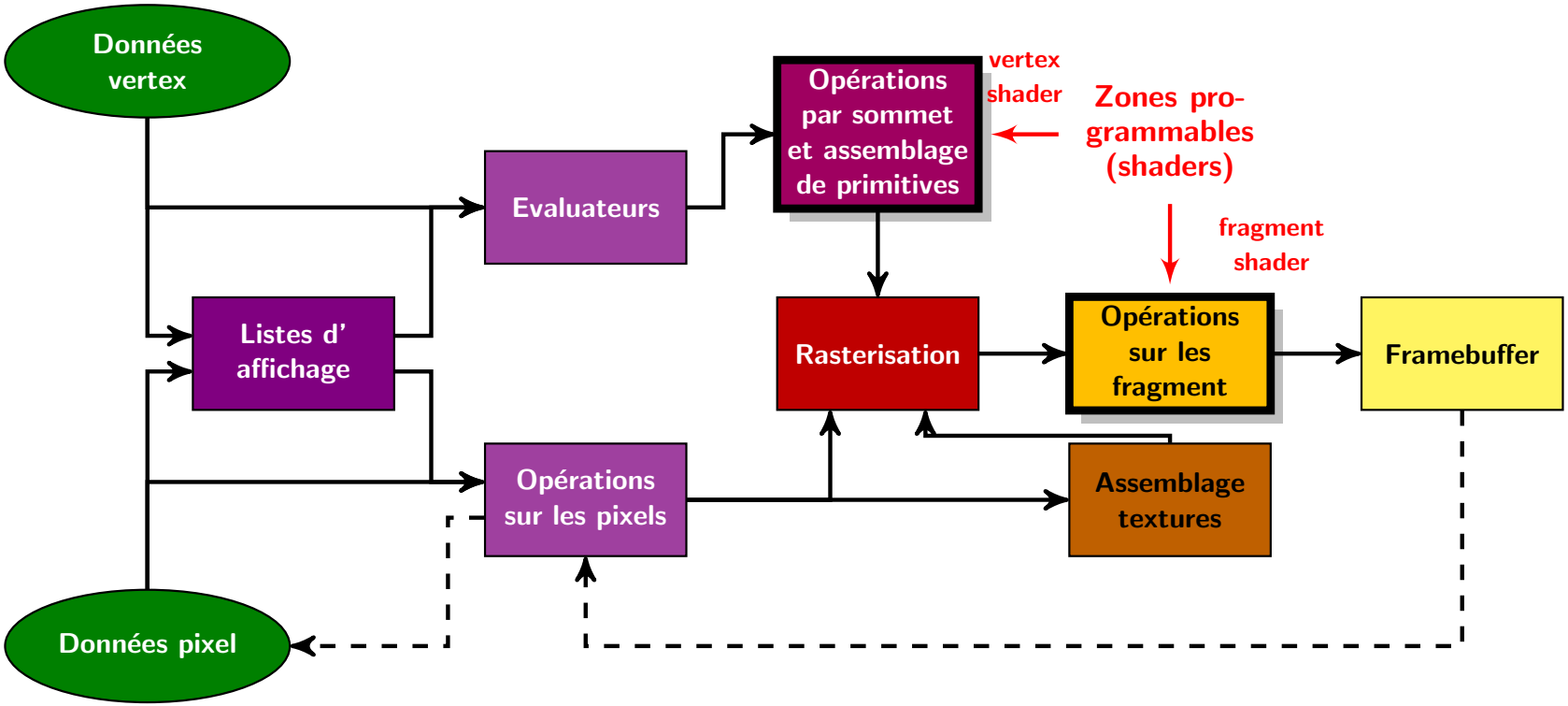
Les *shaders*

ENSIIE2 : Option RVIG

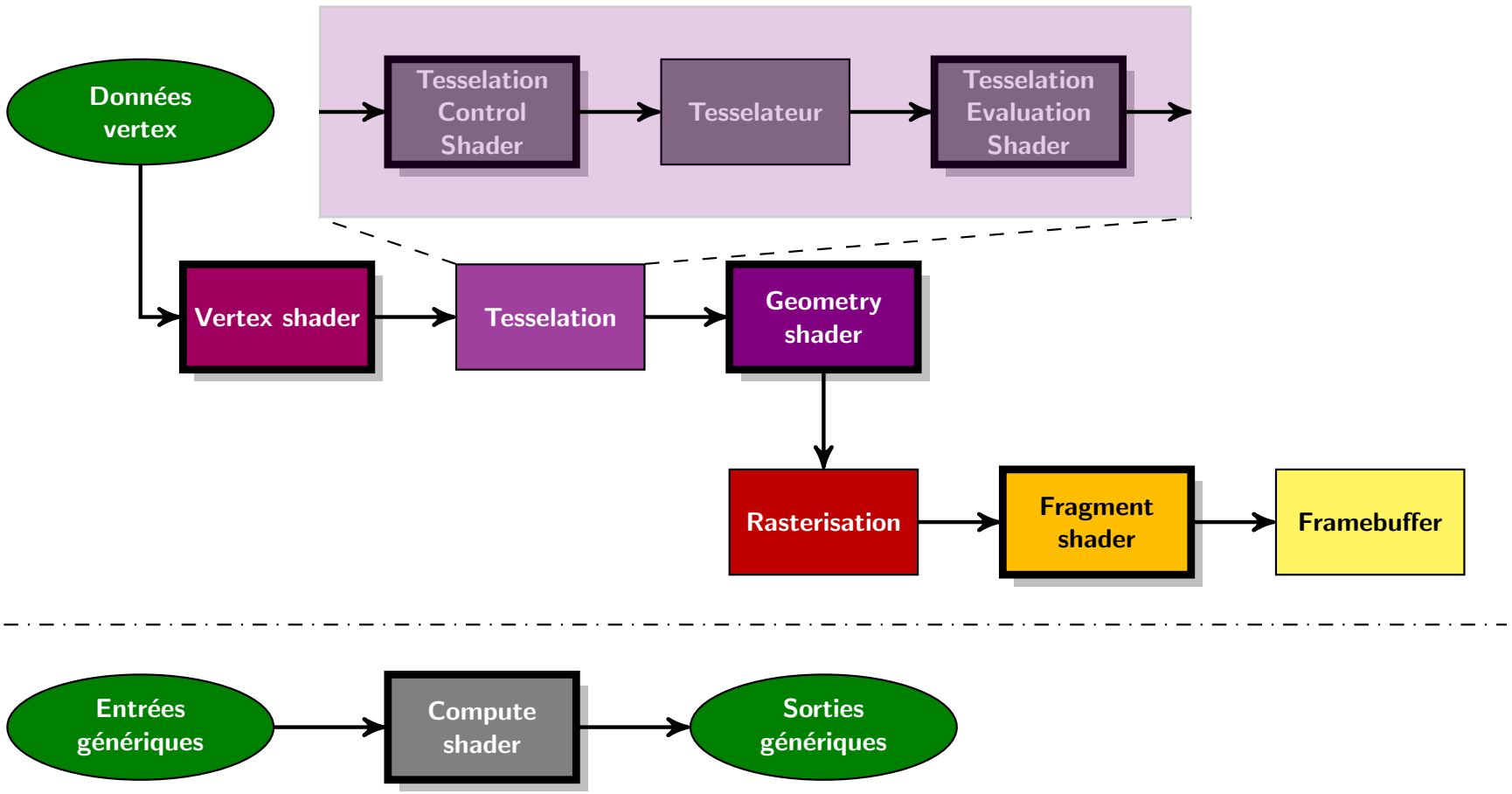
Jean-Yves Didier

didier@ufrst.univ-evry.fr

Carte graphique de génération II



Carte graphique de génération IV



Les *shaders* (1/2)

Principe

Morceaux de programmes interprétable par la carte graphique qui viennent s'intercaler dans le *pipeline* graphique.

Un changement de paradigme !

- Passage d'un *pipeline* **configurable** é un *pipeline* **programmable** !
- Les *shaders* se **substituent aux zones** configurables spécialisées.

Les *shaders* (2/2)

Plusieurs types

- *Compute shader* (2013) :
  shader généraliste de calcul ;
- *Tesselation control shader / hull shader* (2011) :
  transforme un ensemble de polygones en un autre ;
- *Tesselation evaluation shader / domain shader* (2011) :
  évalue la position et les attributs des sommets générés par un algorithme de tesselation ;
- *Geometry shader* (2007) :
  création *ex-nihilo* de primitives géométriques ;
- *Vertex shader* (2003) :
  opérations sur les sommets ;
- *Fragment shader / pixel shader* (2002) :
  opérations sur les fragments.

Languages de *shader*

shaders courants

- *ARB assembly language*, (ARB – 2002) ;
- Cg – *C for Graphics* , (Nvidia – 2003) ;
- HLSL – *High Level Shading Language*, (Microsoft – 2004) ;
- GLSL – *GL Shading Language*, (Khronos group – 2004) ;

shaders pour *flash*

- *Adobe Graphics Assembly Language* – surcouche pour *ARB assembly language* ;
- *Abobe Pixel Bender* – surcouche GLSL.

Exemples de *fragment shaders* (1/3)

Conversion BGRA à RGBA de la texture

ARB Assembly Language

```
!!ARBfp1.0
TEMP color;
TEX color, fragment.texcoord[0], texture[0], 2D;
MOV result.color.r, color.b ;
MOV result.color.b, color.r ;
MOV result.color.g, color.g ;
MOV result.color.a, color.a ;
END
```

Exemples de *fragment shaders* (2/3)

Conversion BGRA à RGBA de la texture

GLSL 1.0

```
uniform sampler2D myTexture;
void main() {
  vec4 color = texture2D(myTexture, gl_TexCoord[0]);
  gl_FragColor = color.bgra;
}
```

HLSL

```
sampler2D mytexture;
float4 MyShader( float2 Tex : TEXCOORD0 ) : COLOR0 {
  float4 Color;
  Color.bgra = tex2D(mytexture, Tex);
  return Color;
}
```

Exemples de *fragment shaders* (3/3)

Conversion BGRA à RGBA de la texture

Cg

```
struct pixel_out {
  float4 color : COLOR;
};
pixel_out main(float2 texCoord : TEXCOORD0, uniform
    sampler2D myTexture) {
  pixel_out OUT;
  OUT.color = tex2D(myTexture, texCoord).bgra;
  return OUT;
}
```

Vertex shader (1/2)

Applications/utilisations

- Transformation de la position du sommet par rapport à la matrice de projection ;
- Transformation des normales au sommet ;
- Génération de coordonnées de texture ;
- Illumination par sommets ;
- Calcul des couleurs.

Court-circuite une partie du *pipeline* graphique

Nécessaire de parfois reproduire certaines étapes de ce dernier.

Vertex shader (2/2)

Entrées/sorties

Entrées : données *vertex* : position, couleur, normales, . . . ;

Sorties : au minimum la position nouvellement calculée du *vertex*.

Calculs indépendants pour chaque sommets

Aucune connaissance des sommets voisins.

Fragment shader (1/2)

Applications/utilisations

- Calcul de couleur des coordonnées textures par rapport au pixel ;
- Application de texture ;
- Brouillard ;
- Calcul des normales si illumination par pixel.

Avertissements

Les mêmes que pour les *vertex shaders* :

- Tout est à faire lorsqu'on passe en *shader* ;
- Les calculs sur les fragments sont indépendants.

Fragment shader (2/2)

Entrées/sorties

Entrées : données fragment : coordonnées, profondeur, couleur, coordonées texture ;

Sorties : deux choix :

- discard : le fragment n'est pas rendu ;
- au minimum la couleur du fragment.

GLSL : Présentation

Un langage de *shader*

- Syntaxe *C-like* ;
- Dernière version en date : 4.50.6 ;
- Associé aux évolutions d'OpenGL ;
- Associé au standard HTML5 (WebGL) :
  - Plus particulièrement lié à OpenGL ES 2.0 ;
  - Version de GLSL employée : 1.10.59.

Une évolution paralléle à OpenGL

Année	Version GLSL	Version OpenGL
2016	4.50.6	4.5
2014	4.40.9	4.4
2013	4.30.8	4.3
2011	4.20.11	4.2
2010	4.10.6	4.1
2010	4.00.9	4.0
2010	3.30.6	3.3
2009	1.50.11	3.2
2009	1.40.08	3.1
2009	1.30.10	3.0
2006	1.20.8	2.1
2004	1.10.59	2.0

Eléments de langage

Types des données

Types autorisés

- Nombres réels et vecteurs associés : float, vec2, vec3, vec4
- Nombres entiers et vecteurs associés : int, ivec2, ivec3, ivec4
- Booléens et vecteurs associées : bool, bvec2, bvec3, bvec4
- Matrices de réels de taille $n \times n$ ($matn$) : mat2, mat3, mat4
- Textures : sampler1D, sampler2D, sampler3D
- Structures : comme en C, en utilisant les types de base ;
- Et, bien sûr : void

Champs des vecteurs

vecteurs : x,y,z,w, couleurs : r,g,b,a, textures : s,t,p,q.

Eléments de langage (1/2)

Modificateurs

- Concerne les variables globales des *shaders* ;
- Différents suivant les versions de GLSL.

GLSL 1.10

uniform	Donnée globale pour les *shaders* (lecture) ;
attribute	Donnée par primitive (vertex ou fragment) (lecture) ;
varying	Donnée produite par le *vertex shader* (lecture/écriture), interpolée et fournie au *fragment shader* (lecture) ;
const	Donnée constante lors de la compilation (lecture).

Eléments de langage (2/2)

Modificateurs

GLSL > 1.30

- Modificateurs classiques :

  in, centroid in : donnée venant d'un *shader* en amont ;
  out, centroid out : donnée pour un *shader* en aval du *pipeline* ;
  const, uniform .

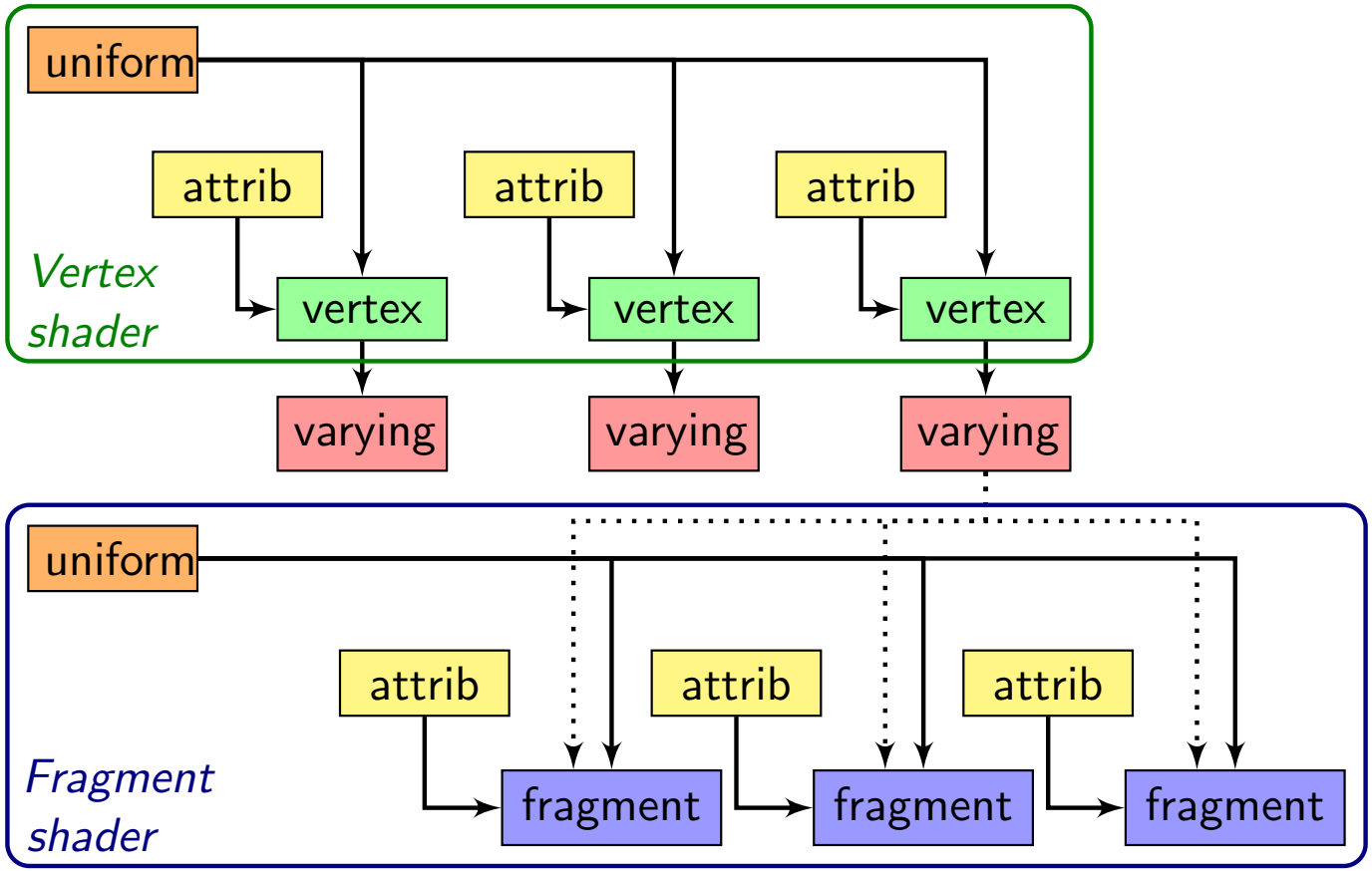
- Modificateurs d'interpolation :

  smooth interpolation avec correction perspective ;
  noperspective interpolation linéaire ;
  flat pas d'interpolation.

- Ordre d'application :
  - `[interp] [classique] type nom_variable.`

Eléments de langage

uniform, *varying* et *attribute*



Eléments de langage

Les fonctions

Déclarer une fonction : syntaxe

```
type_retour nom_fnct([mod_arg] type_arg nom_arg, ....) {
    ...
}
```

Les modificateurs de paramètres

in	paramètre copié à l'entrée seulement (défaut) ;
out	paramètre copié à la sortie seulement ;
inout	paramètre copié à l'entrée et la sortie ;
const	paramètre constant.

Fonction prédéfinies (1/3)

Fonctions trigonométriques

(Fonctionnent aussi sur des vecteurs et des matrices)

sin, cos, tan, asin, acos, atan, radians, degrees

Fonctions d'exponentiation

(Fonctionnent aussi sur des vecteurs et des matrices)

pow, exp, log, exp2, log2, sqrt, inversesqrt

Fonctions classiques

(Fonctionnent aussi sur des vecteurs et des matrices)

abs, ceil, clamp, floor, fract, max, min, mix, mod, sign, smoothstep, step

Fonction prédéfinies (2/3)

Fonctions géométriques

- vec3 cross(vec3, vec3) : produit vectoriel ;
- float distance(*,*) : distance entre deux variables ;
- float dot(*,*) : produit scalaire ;
- float length(*) : norme euclidienne de la variable ;
- * normalize(*) : variable normalisée (norme = 1) ;
- * reflect(* I,* N) : réflexion, I = incident, N = normale ;
- * refract(* I,* N,float eta) : réfraction, eta = angle.

Fonction prédéfinies (3/3)

Fonctions sur les textures

Récupération de la couleur du texel au point considéré :

- Texture 1 D :
  - vec4 texture1D(sampler1D, float)
  - vec4 texture1DProj(sampler1D, vec2)
  - vec4 texture1DProj(sampler1D, vec4)
- Texture 2 D :
  - vec4 texture2D(sampler2D, vec2)
  - vec4 texture2DProj(sampler2D, vec3)
  - vec4 texture2DProj(sampler2D, vec4)
- Et les variantes pour niveau de détail et texture3D :
  - vec4 textureXD[Lod][Proj](samplerXD, ...)

Le *vertex shader* GLSL

Variables spéciales de sortie

- gl_Position : coordonnées finales du sommet (obligatoire) ;
- gl_PointSize : taille du point en pixels ;
- gl_ClipVertex : coordonnées à utiliser avec les plans de coupe (*clipping*).

Minimum requis

Doit prendre en charge les transformations et l'illumination (*transform and lighting*) !

Vertex shader minimal (avec OpenGL)

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

Le *fragment shader* GLSL

Variables spéciales d'entrée

- gl_FragCoord : coordonnées du fragment dans la fenêtre (y compris profondeur) ;
- gl_FrontFacing : précise si le fragment fait face à la caméra.

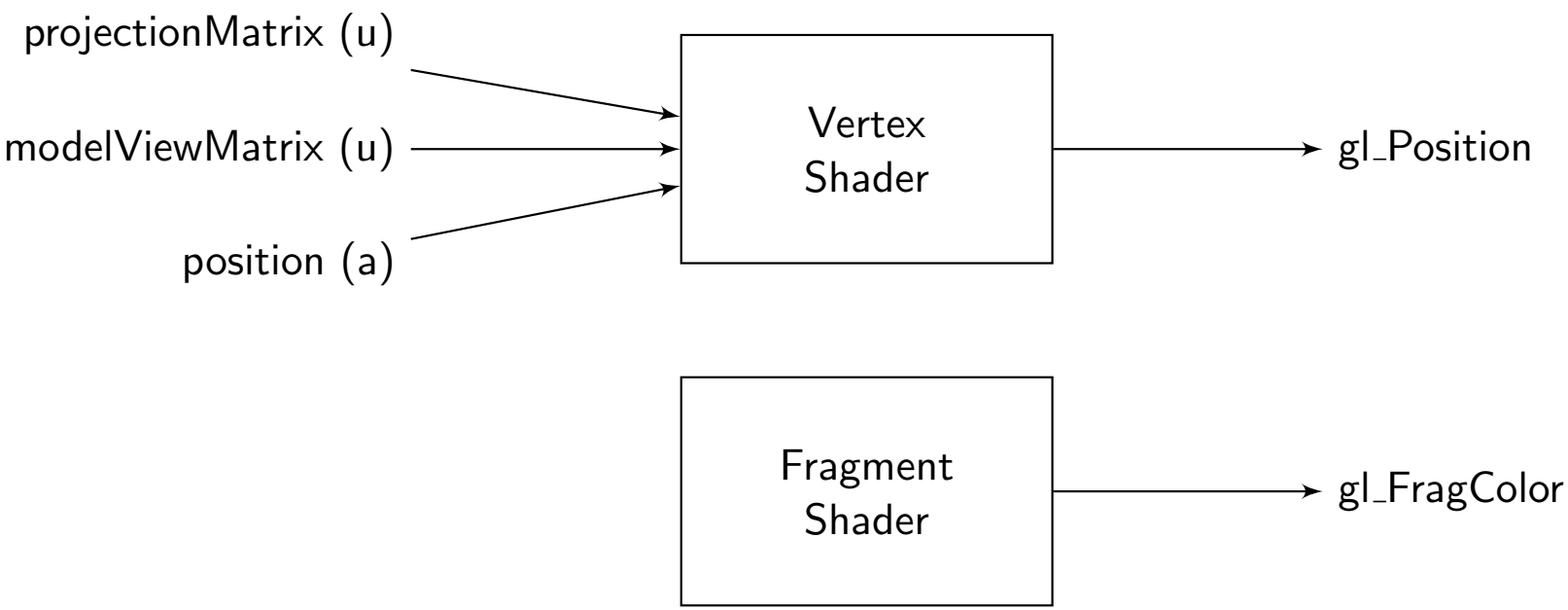
Variables spéciales de sortie

- gl_FragColor : couleur du fragment ;
- gl_FragData[] : données du fragment pour *buffers* suivants ;
- gl_FragDepth : profondeur du fragment.

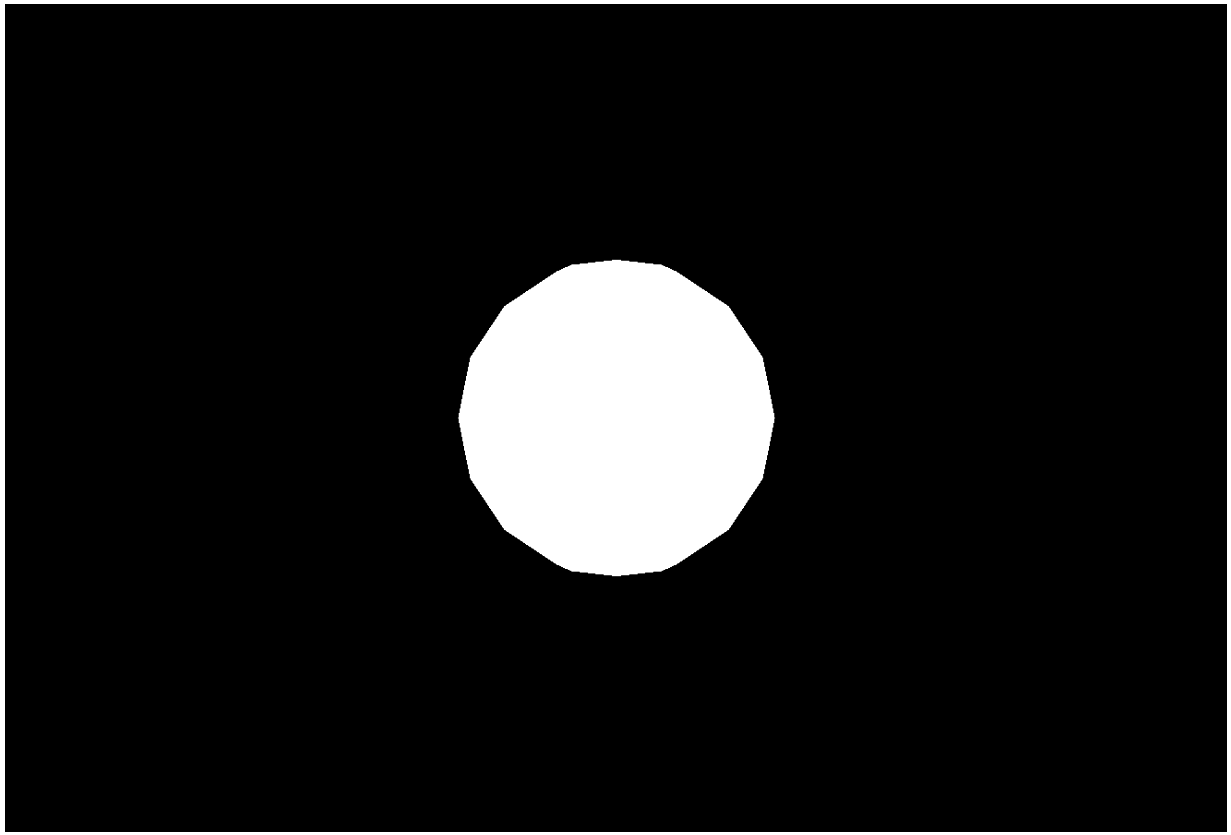
Fragment shader minimal

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

Résultat appliqué sur une sphère (1/2)



Résultat appliqué sur une sphère (2/2)



Exemple de passage de variable (1/3)

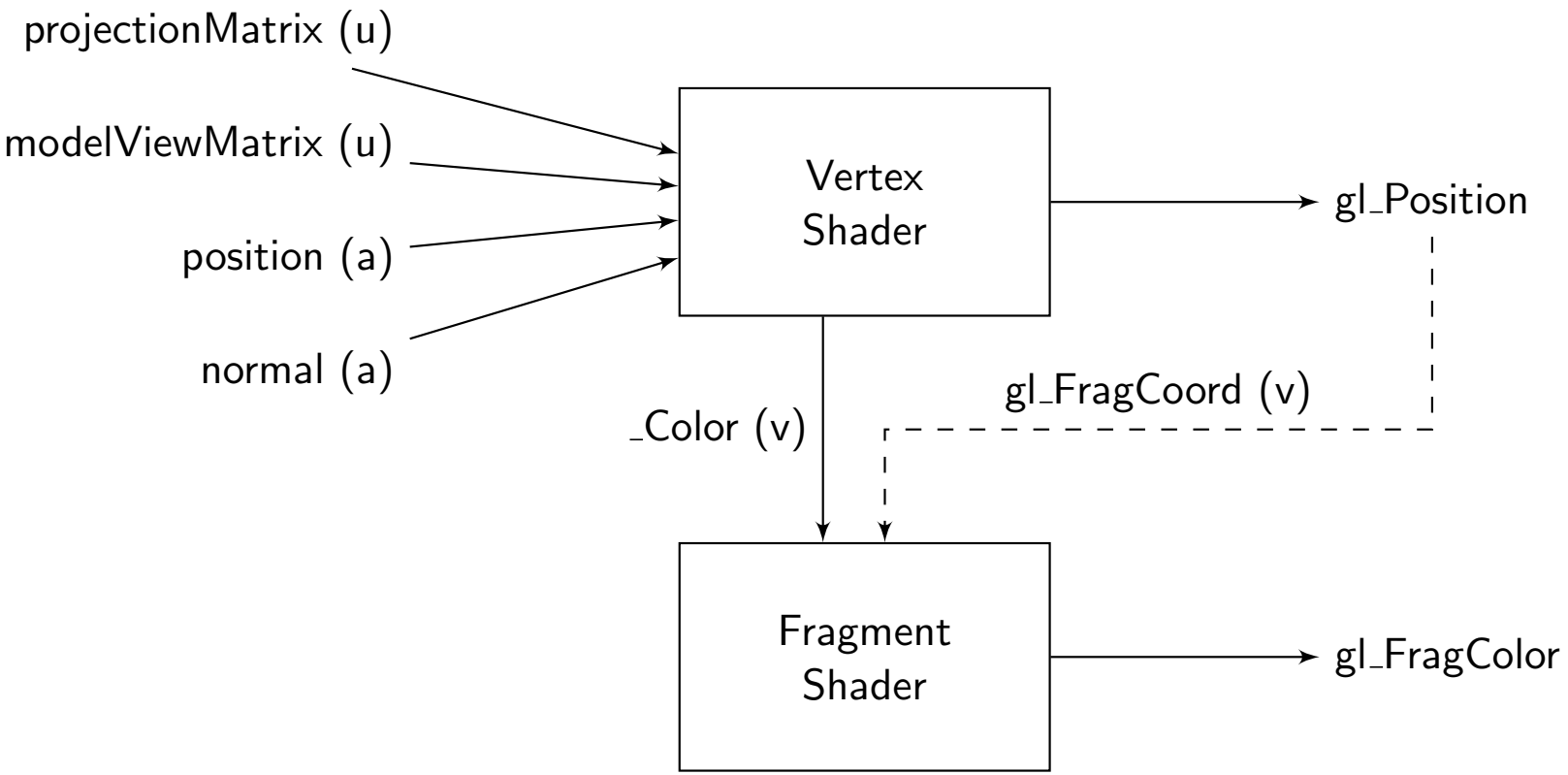
vertex shader

```
varying vec4 _Color;
void main() {
  gl_Position = gl_ProjectionMatrix *
      gl_ModelViewMatrix * gl_Vertex;
  _Color = vec4((1.0 + normalize(gl_Normal))/2.0,1.0);
}
```

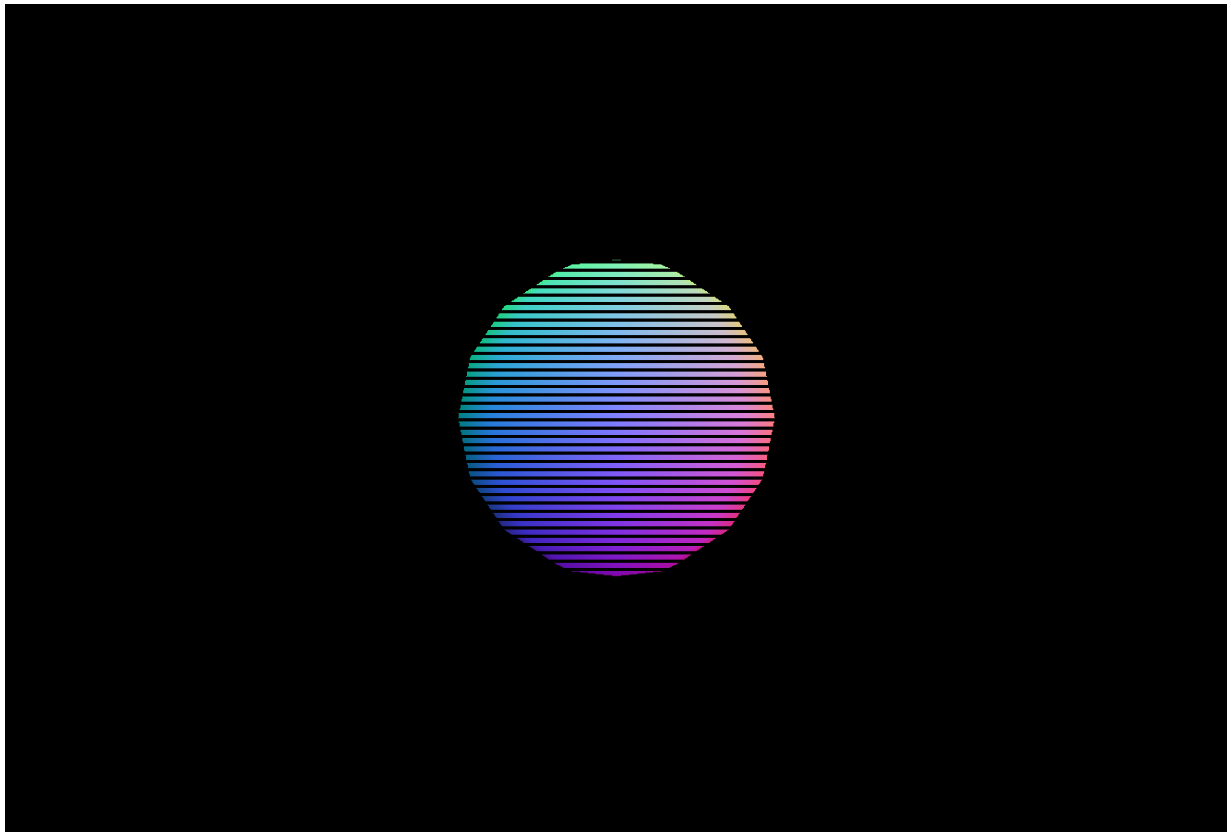
fragment shader

```
varying vec4 _Color;
void main() {
  if (mod(gl_FragCoord.y,9.0) < 3.0) discard;
  gl_FragColor    = _Color ;
}
```

Exemple de passage de variable (2/3)



Exemple de passage de variable (3/3)



Résultat obtenu.

Exemple de placage de texture (1/3)

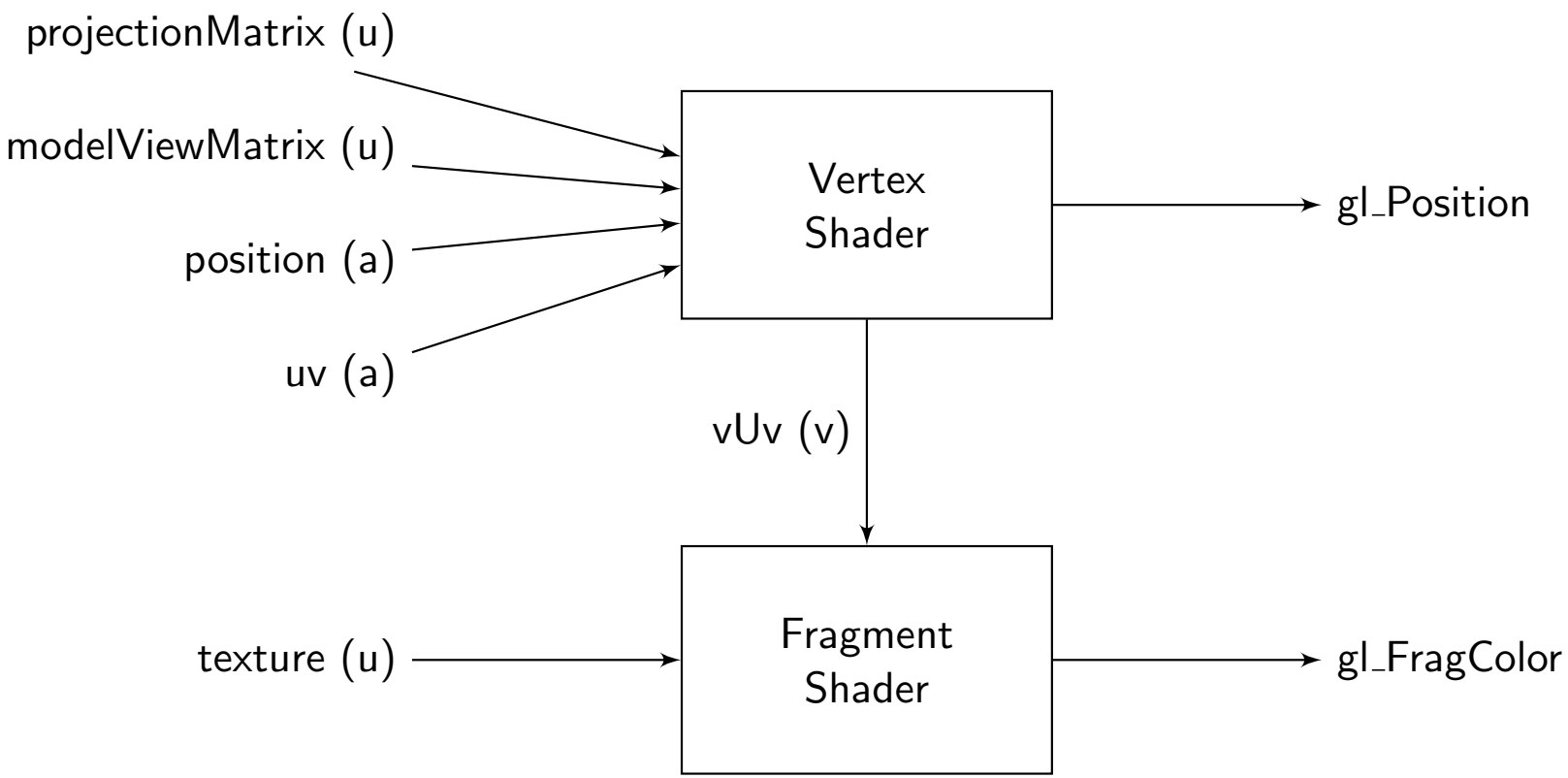
vertex shader

```
void main() {
  gl_Position = gl_ProjectionMatrix *
      gl_ModelViewMatrix * gl_Vertex;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

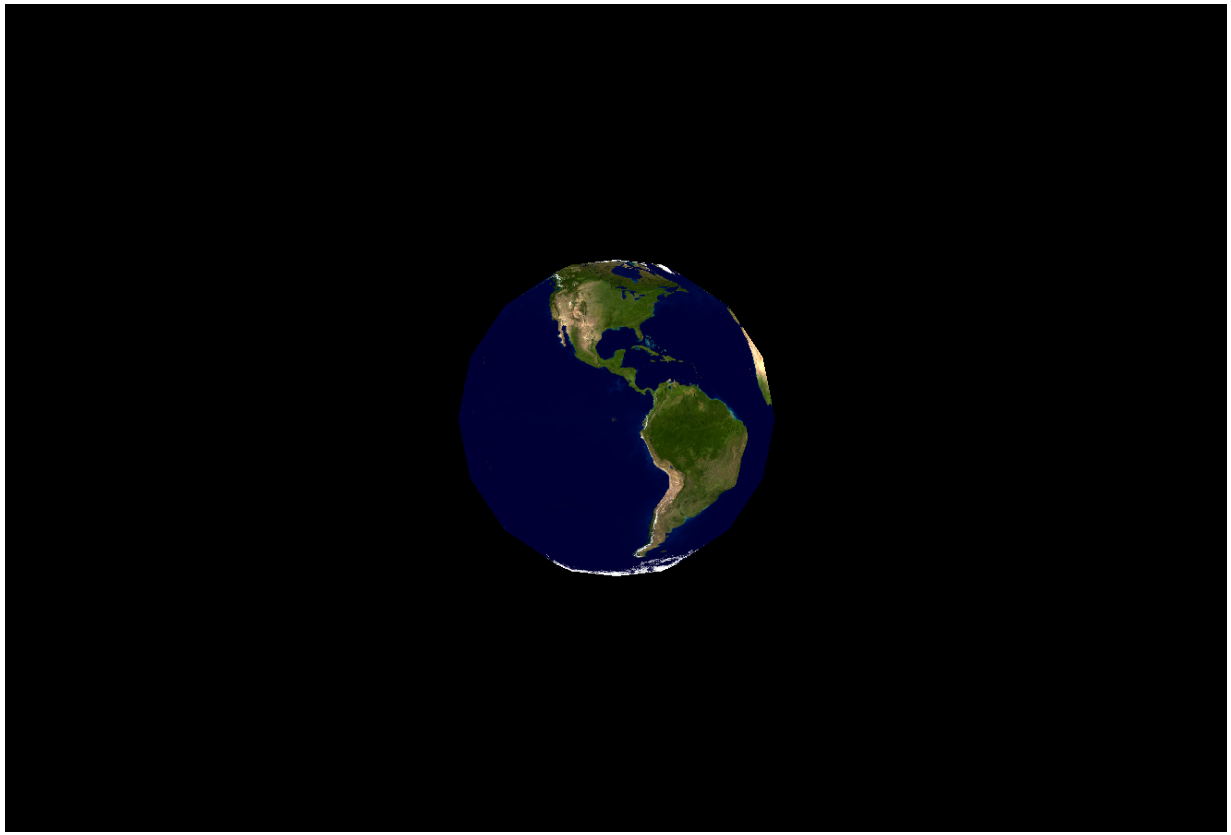
fragment shader

```
void main() {
  gl_FragColor =  texture2DProj(texture, gl_TexCoord
      [0]) ; ;
}
```

Exemple de placage de texture (2/3)



Exemple de placage de texture (3/3)



Résultat obtenu.

Concevoir son propre *shader*

Etapes de conception

1. Réfléchir aux traitements à déléguer au *vertex shader* puis au *fragment shader* ;
2. Identifier les variables uniformes nécessaires au deux *shaders* ;
3. Identifier les variables de type *varying* permettant aux shaders de communiquer entre eux ;
4. Ecrire les *shaders*.

GLSL et contexte d'utilisation

Contexte d'utilisation

Ensemble de paramètres uniformes et d'attributs prédéfinis :

- Dans OpenGL, beaucoup sont définis ;
- Dans WebGL, presque tout est à définir :
  - Pris en charge par certaines surcouches à WebGL.

GLSL et le contexte OpenGL (1/3)

Variables uniformes

- Matrices 4×4 : gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix, gl_TextureMatrix ainsi que inverses (gl_XxxxInverse) et transposées (gl_XxxxTranspose) ;
- Matrice 3×3 : gl_NormalMatrix ;
- Sources de lumière (structures) : gl_LightSource[] ;
- Matériaux (structures) : gl_FrontMaterial et gl_BackMaterial ;

GLSL et le contexte OpenGL (2/3)

Attributs pour le *vertex shader*

- gl_Vertex : coordonnées du sommet ;
- gl_Normal : normale au sommet ;
- gl_Color, gl_SecondaryColor : couleurs du sommet ;
- gl_FogCoord : coordonnées du brouillard ;
- gl_MultiTexCoord[0-7] : coordonnées de texture associées.

Sorties prédéfinies pour le *vertex shader*

- gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, gl_BackSecondaryColor : couleurs du sommet ;
- gl_TexCoord[] : coordonnées de texture ;
- gl_FogFragCoord : coordonnées du brouillard.

GLSL et le contexte OpenGL (3/3)

Entrées prédéfinies pour le *fragment shader*

- gl_Color, gl_SecondaryColor : couleur du fragment ;
- gl_TexCoord[] : coordonnées de texture associées au fragment ;
- gl_FogFragCoord : coordonnées du brouillard associé au fragment.

WebGL

Une partie du standard HTML5 (2014)

- Gère l'affichage 3D dans un navigateur ;
- Spécification maintenue par *Khronos group* ;
- Basé sur un sous-ensemble d'OpenGL ES :
  - 2.0 pour WebGL 1 ;
  - 3.0 pour WebGL 2.
- Programmation en Javascript.

Une écriture très bas niveau

Dans WebGL, tout est en réalité *shader* ! ! !

Support de WebGL dans les navigateurs

Un support établi pour WebGL1

- Safari 5.1+ ;
- Google Chrome 9+ ;
- Firefox 4+ ;
- Opera Next 12+ ;
- Internet Explorer : 12+ sinon plugin tiers : IEWebGL.

Un support récent pour WebGL2 (01/2017)

- Google Chrome 56+ ;
- Firefox 51+.

Structure d'un programme WebGL

Etapes

1. récupération du contexte WebGL ;
2. initialisation des *shaders* ;
3. préparation des données en mémoire tampon (*buffer*) ;
4. rendu de la scène.

Récupérer le contexte WebGL

Un objet de type WebGLObject

```
<!DOCTYPE html>
<html>
  <body>
    <canvas id="c"></canvas>
    <script type="text/javascript">
      var canvas = document.getElementById("c");
      var gl = canvas.getContext("webgl");
      gl.clearColor(1.0,0.0,0.0,1.0);
      gl.clear(gl.COLOR_BUFFER_BIT);
    </script>
<!-- ou experimental-webgl pour avoir le contexte -->
  </body>
</html>
```

Initialiser les *shaders* (1/4)

Déclarer ses *shaders* en WebGL

```
<script type="x-shader/x-vertex" id="v_shader">
  attribute vec3 _position;
  uniform mat4 _matrix;
  uniform mat4 _proj;
  void main() {
    gl_Position = _proj * _matrix * vec4(_position,1.0)
        ;
  }
</script>
<script type="x-shader/x-fragment" id="f_shader">
  void main() {
    gl_FragColor = vec4(1.0,1.0,1.0,1.0);
  }
</script>
```

Initialiser les *shaders* (2/4)

Compiler les *shaders*

```
// créer un objet de type shader
fShader = gl.createShader(gl.FRAGMENT_SHADER);
// accéder au contenu du shader.
fShaderNode = document.getElementById("f_shader");
// affecter les sources et compiler
gl.shaderSource(fShader, fShaderNode.text);
gl.compileShader(fShader);
// vérification du statut de compilation
if (!gl.getShaderParameter(fShader, gl.COMPILE_STATUS))
{
  // ... mettre un message d'alerte.
}
// faire de meme pour vShader
```

Initialiser les *shaders* (3/4)

Associer les shaders à un programme

```
shaderProgram = gl.createProgram();
// attacher les shaders à un programme
gl.attachShader(shaderProgram, vShader);
gl.attachShader(shaderProgram, fShader);
gl.linkProgram(shaderProgram);
// vérifier si tout se passe bien
if (!gl.getProgramParameter(shaderProgram, gl.
    LINK_STATUS))
{
  // ... mettre un message d'alerte
}
// utiliser le programme
gl.useProgram(shaderProgram);
```

Initialiser les *shaders* (4/4)

Récupérer les attributs et variables uniformes

```
// récupération d'un attribut et stockage dans variable
var positionAttrib = gl.getAttribLocation(shaderProgram
    , "_position");
// annonce que les sommets stockeront un attribut suppl
    émentaire
gl.enableVertexAttribArray(positionAttrib);
// récupération d'une variable uniforme
var matrixUniform = gl.getUniformLocation(shaderProgram
    , "_matrix");
// ... faire de meme pour _proj
```

Initialiser un *buffer*

Initialiser les données de la scène

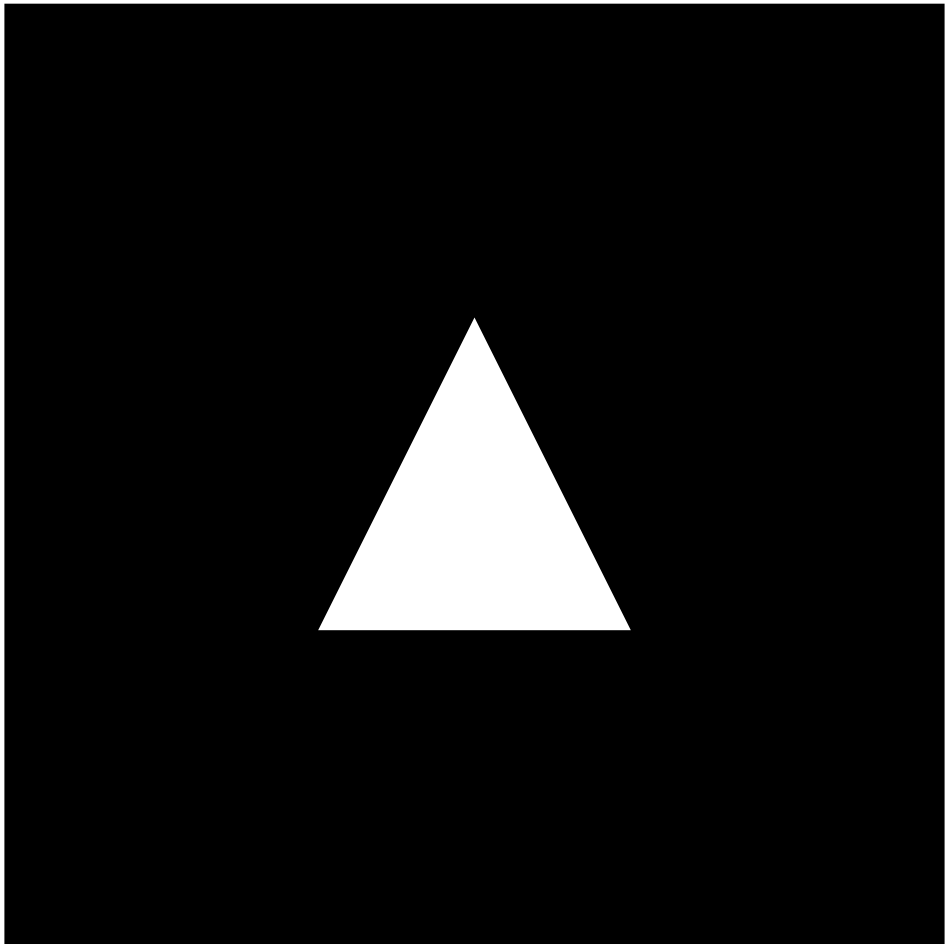
```
// création de l'objet
triangleBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleBuffer);
var vertices = [
   0.0,  1.0,  0.0,
  -1.0, -1.0,  0.0,
   1.0, -1.0,  0.0
];
// ajout des valeurs à l'objet
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(
    vertices), gl.STATIC_DRAW);
```

Effectuer le rendu de la scène

Rendu de la scène

```
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
// ... matrice projection à calculer
// ... matrice monde à calculer
gl.bindBuffer(gl.ARRAY_BUFFER, triangleBuffer);
gl.vertexAttribPointer(positionAttrib,3/*nb composants
    */, gl.FLOAT, false, 0, 0);
// passer les matrices calculées en paramètres
    uniformes
gl.uniformMatrix4fv(matrixUniform, false, /* matrice
    monde calculée */);
// affichage des primitives : 3 sommets regroupés en
    triangle
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Le rendu final



Un triangle blanc au centre de l'écran !

Surcouches à WebGL

Une programmation fastidieuse avec WebGL

Necessité d'avoir quelquechose de plus simple à manipuler !

Un certain nombre de bibliothèques

- glMatrix : manipulation de matrices ;
- Copperlicht : moteur 3D (commercial) pour WebGL ;
- SceneJS : gestion de graphe de scène ;
- three.js : gestion de graphe de scène ;
- BabylonJS : gestion de graphe de scène.

Utiliser les *shaders* avec *three.js* (1/4)

Fichier html

```
<!doctype html>
<html>
  <head>
    <title>scene three.js</title>
  </head>
  <body>
    <div id="container" style="background: #000"></div>
  </body>
  <!-- scripts -->
</html>
```

Utiliser les *shaders* avec *three.js* (2/4)

Scripts attachés

```
<script type="x-shader/x-vertex" id="vertex_shader">
  void main() {
    gl_Position = projectionMatrix * modelViewMatrix *
        vec4(position,1.0);
  }
</script>
<script type="x-shader/x-fragment" id="fragment_shader"
    >
  void main() {
    gl_FragColor = vec4(1.0,1.0,1.0,1.0);
  }
</script>
<!-- pour utiliser three.js -->
<script src="js/three.js"></script>
```

Utiliser les *shaders* avec *three.js* (3/4)

Script décrivant la scène (1/2)

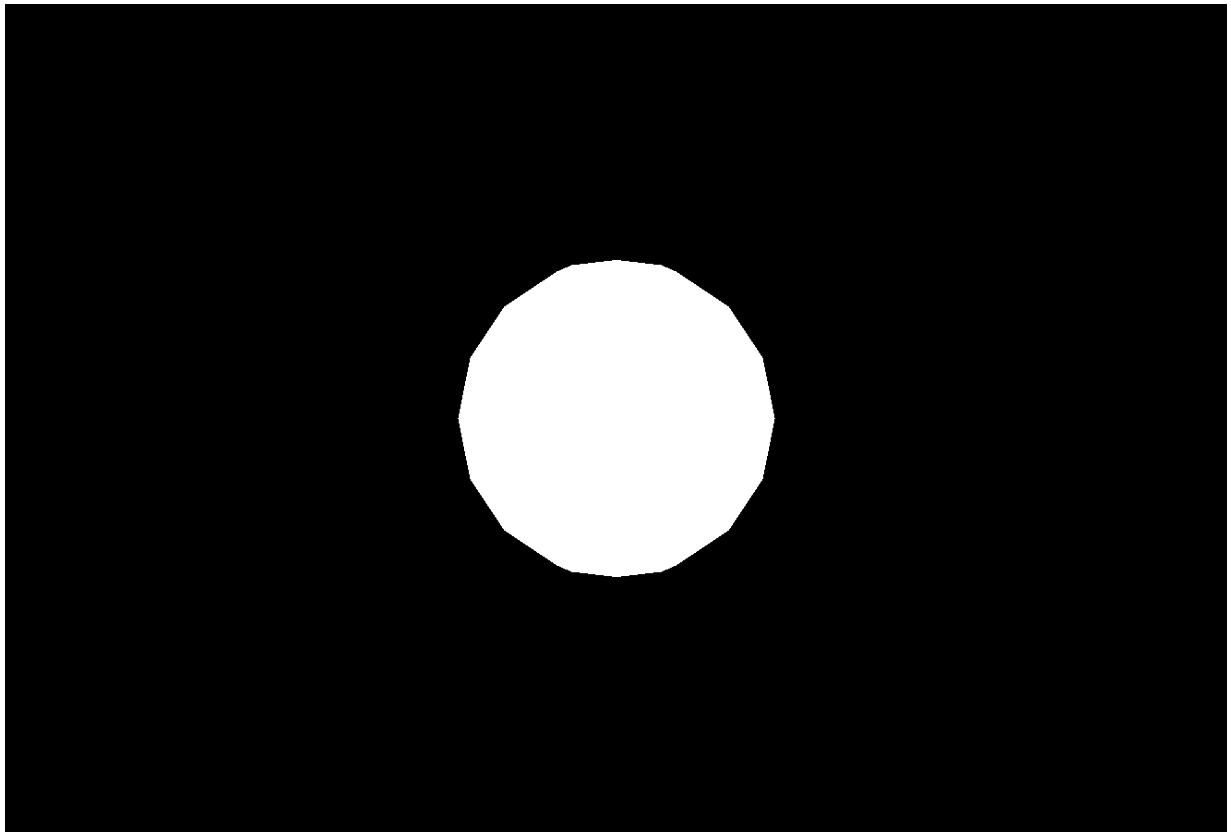
```
// récupérer le div où sera rendue la scène
var container = document.getElementById('container');
// création d'un contexte de rendu
var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth,window.innerHeight);
// création d'une scène (vide)
var scene = new THREE.Scene();
// création d'une caméra perspective
var camera = new THREE.PerspectiveCamera( 45,
  window.innerWidth/window.innerHeight,1,10000  );
// déplacement de la caméra
camera.position.z = 300;
```

Utiliser les *shaders* avec *three.js* (4/4)

Script décrivant la scène (2/2)

```
// création d'un matériau utilisant les shaders
var shaderMaterial = new THREE.ShaderMaterial({
  vertexShader:   document.getElementById( '
      vertex_shader').textContent,
  fragmentShader: document.getElementById( '
      fragment_shader').textContent });
// création d'un objet sphère : géométrie + matériau
var sphere = new THREE.Mesh(
  new THREE.SphereGeometry(50,8,8), shaderMaterial);
// ajouter la sphère à la scène
scene.add(sphere);
// effectuer le rendu de la scène
renderer.render(scene, camera);
// afficher dans le div
container.appendChild(renderer.domElement);
```

Résultat obtenu



Ressources

Spécifications du langage GLSL version 1.10.59 :
www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf

Spécification du langage GLSL version 4.30.8 :
www.opengl.org/registry/doc/GLSLangSpec.4.30.8.pdf

Tutoriel NVidia sur le langage Cg :
http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

GPU Gems (NVidia) – divers algorithmes de *shader* :
http.developer.nvidia.com/GPUGems/gpugems_part01.html

Spécification WebGL version 1.0 (Khronos group) :
https://www.khronos.org/registry/webgl/specs/1.0/

La bibliothèque de rendu three.js :
mrdoob.github.com/three.js/

Learning WebGL :
learningwebgl.com/blog/