

# WEBGL ET RENDU 3D

---

**Florent Grélard**

[florent.grelard@labri.fr](mailto:florent.grelard@labri.fr)

Licence Pro DAWIN, 2016-2017



# SOMMAIRE

Introduction

Premiers pas en WebGL

Shaders et rendu sur une page Web

Buffers

# RENDU 3D

Monde 3D  $\Leftrightarrow$  Affichage d'images 2D (écran)

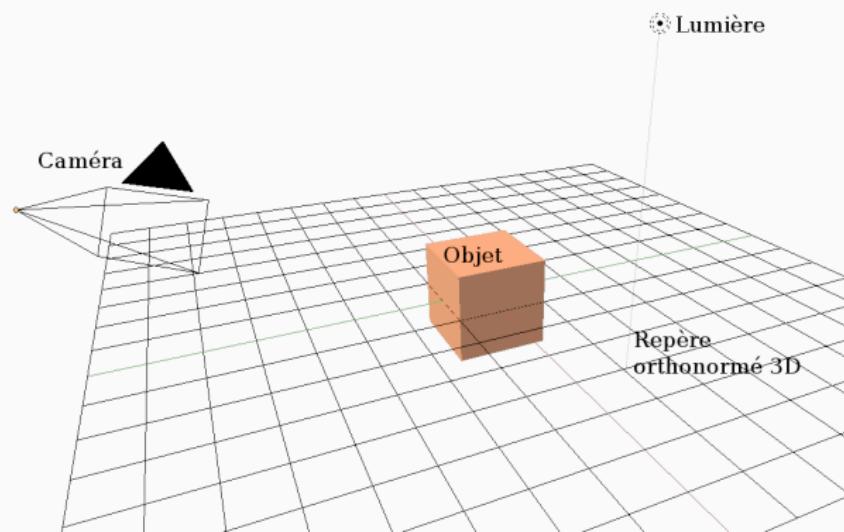
**Rendu 3D** = images 2D à partir de coordonnées 3D



# RENDU 3D

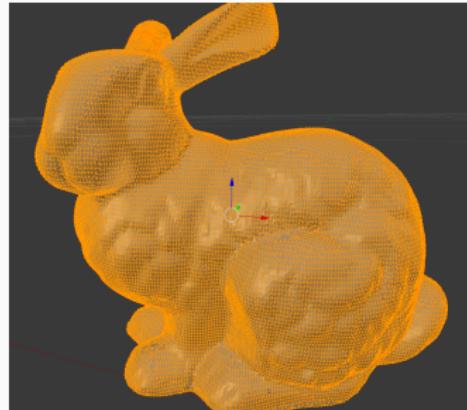
## Un peu de vocabulaire...

Scène, point de vue (=position de la **caméra**), source de lumière, objet(s).



# RENDU 3D

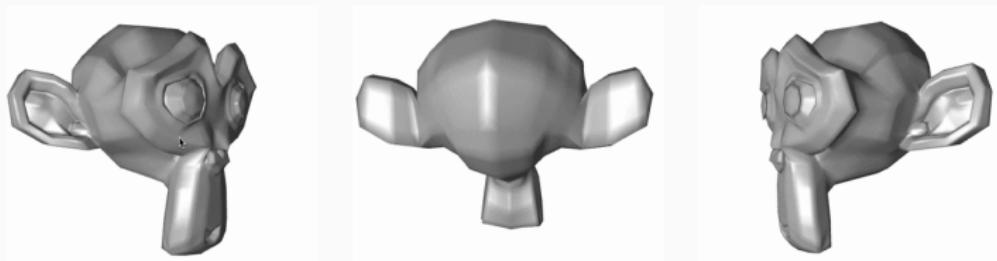
- Un objet est constitué de sommets (*vertex* en anglais).
- Triangle = 3 sommets, un carré = 4 sommets, etc.
- Un objet 3D est dessiné en utilisant une multitude de triangles. L'ensemble des triangles définit la surface de l'objet. On parle aussi de **maillage**.



# RENDU 3D

Le rendu se fait en fonction :

- du point de vue
- de la **lumière**
- de l'**objet** (matériau, texture...)



# RENDU 3D

## **Interactions caméra/objet**

Extraire la partie visible de l'objet

⇒ Projections

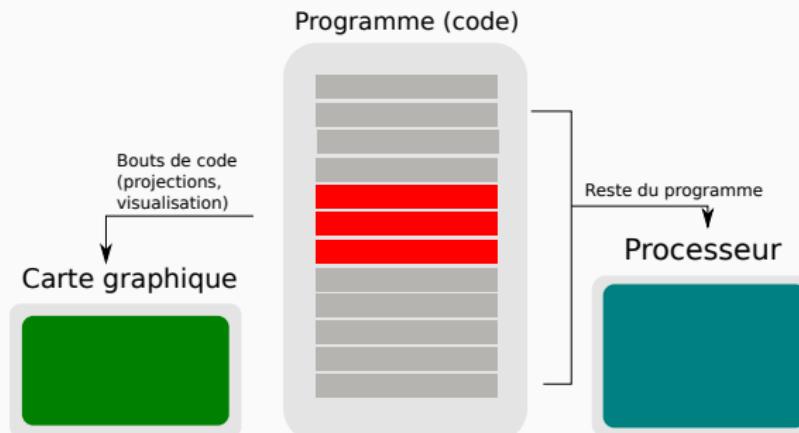
## **Interactions lumière/objet**

- Reflets, ombres sont décrits par de nombreux **modèles mathématiques**
- Lien avec physique (optique)

# RENDU 3D SUR ORDINATEUR

- Opérations coûteuses en temps de calcul
- Travail sur la carte graphique

⇒ Allège la charge du processeur



# PROGRAMME DU COURS

## Au programme

- Prise en main d'une scène 3D
- Programmation avec la carte graphique
- Rendu d'objets 3D simples
- Comprendre les interactions simples entre caméra et objet(s)

## Ce qui n'est pas au programme

- Ombrage et reflets (gestion de la lumière avancée)
- Rendu de courbes

# SOMMAIRE

Introduction

Premiers pas en WebGL

Shaders et rendu sur une page Web

Buffers

# PRÉSENTATION

- Rendu 3D dans une page Web
- Crée fin 2009 (toujours en version 1.0)
- API (interface) qui permet d'utiliser **OpenGL** (GLSL) via Javascript

## **OpenGL**

- Communication avec la carte graphique
- Portable (contrairement à DirectX de Microsoft)
- Codé en langage C

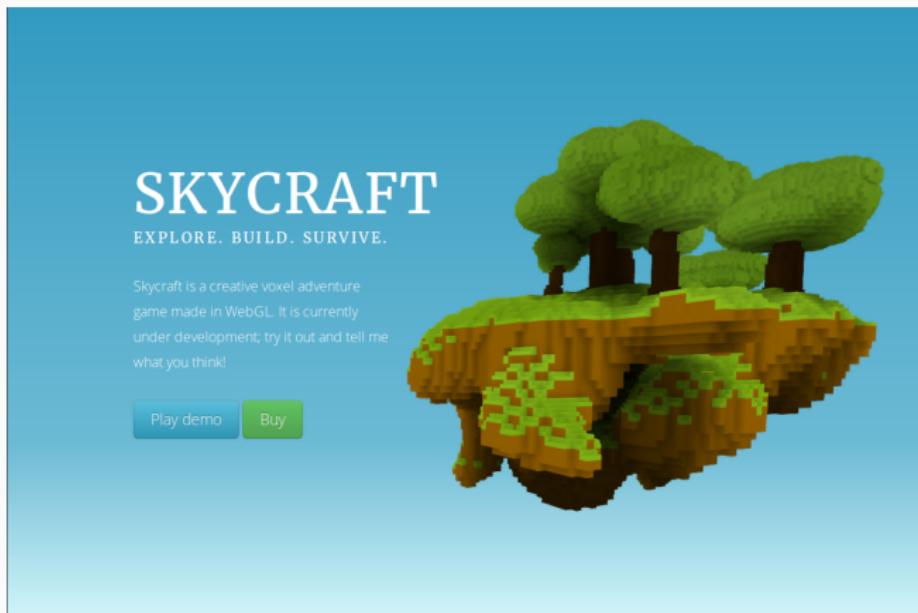
# QUELQUES EXEMPLES D'UTILISATION DE WEBGL

MapsGL: <https://www.google.fr/maps/>



## QUELQUES EXEMPLES D'UTILISATION DE WEBGL

Jeux Web 3D : <http://www.webglgames.com/>



## QUELQUES EXEMPLES D'UTILISATION DE WEBGL

Films d'animations semi-interactifs : <http://www.ro.me/>



# CONCEPTS DU WEBGL

Interface Javascript qui permet de programmer sur la carte graphique via OpenGL.

## HTML

- Utilisation de la balise <canvas>
- Récupération de l'**id** de ce <canvas> via Javascript
- Script .js externe

# PAGE HTML MINIMALE POUR WEBGL

Fichier html :

```
1 <html>
2   <body onload='main()>
3     <canvas id='dawin-webgl' width=800 height=800>
4       Utilisez un navigateur compatible avec WebGL
5     </canvas>
6     <script src='tp1.js'></script>
7   </body>
8 </html>
```

# RÉCUPÉRATION DU CONTEXTE EN JAVASCRIPT

Fichier javascript :

```
var canvas = document.getElementById('dawin-webgl');
var gl = canvas.getContext('webgl');
if (!gl) {
    console.log('ERREUR : echec du chargement du contexte');
    return;
}
```

A vous de jouer : commencez le TP1

# CONCEPTS DU WEBGL

## Généralités

- API relativement bas niveau
- Pas de structure de données : **machine à états**  
(activation/désactivation de modes, de paramètres)
- Notions de contexte, de **buffers** et de **shaders**

# SCÈNE

## Scène

- Coordonnées flottantes comprises entre -1.0 et 1.0
- Par exemple, pour un point 2D :

```
var point = [ 0.5, 0.5 ];
```

Pour un triangle :

```
var triangle = [ 0.5, 0.5,
                 -0.5, -0.5,
                 0.5, -0.5 ];
```

Dessiner un repère (O, x, y) dans l'intervalle [-1.0; 1.0] et les sommets du triangle ci-dessus.

# CONTEXTE

## Contexte

Assure le **dessin dans le canvas**

⇒ utilisation de la carte graphique :

- mémoire
- processeur interne (GPU)

- Dans le fichier Javascript : correspond à la variable `gl` rentrée par `canvas.getContext('webgl')`
- `gl` est un objet/interface permettant l'utilisation des fonctions OpenGL

# SOMMAIRE

Introduction

Premiers pas en WebGL

Shaders et rendu sur une page Web

Buffers

# SHADERS

## Définition

Shaders = bouts de code exécutés sur la carte graphique, vivant indépendamment du processeur (CPU). Ils sont directement liés au **rendu 3D**.

## Utilité

- Alléger la charge du processeur
- Rapidité d'exécution
- Exécuter des opérations complexes (rotations, projections, ombres...) avant que la scène ne soit dessinée

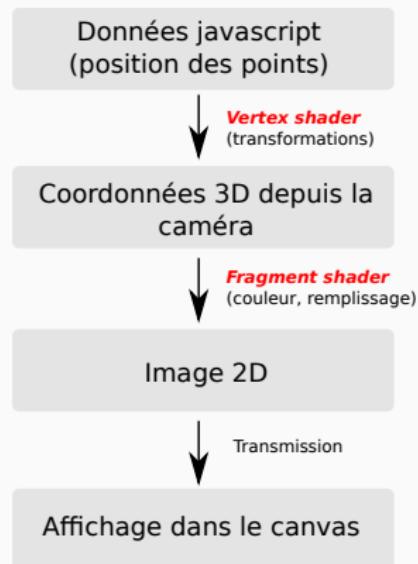
# SHADERS

Il existe deux types de shaders :

- le **vertex shader** : agit sur les **sommets** (vertex). Utilisé pour calculer la position des points en fonction de la caméra.
- le **fragment shader** : agit sur les **pixels** ⇒ permet d'obtenir une image 2D pouvant être dessinée dans le canvas. Utilisé pour donner de la couleur et la profondeur aux objets.

# SHADERS

Schéma simplifié de l'utilité des shaders dans le processus de rendu 3D :



# SHADERS

## Dans le code

- Les shaders sont écrits en GLSL (dérivé d'OpenGL), proche du **langage C**
- Doivent être **compilés, linkés**
- Sont initialisés après la récupération du contexte WebGL

On crée deux fichiers séparés pour le code des deux shaders (extension **.glsl**).

Au minimum, ils contiennent la fonction **void main() {}**

## VARIABLES EN GLSL

Machine à états  $\Rightarrow$  Il existe différents **descripteurs** de variables globales en GLSL :

1. **uniform** : peuvent être modifiées ( $\neq \text{const}$ ) mais restent les mêmes pour chaque vertex (généralement la couleur)
2. **attribute** : peuvent être toutes traitées individuellement, à chaque appel différent pour chaque sommet
3. **varying** : ce sont les valeurs interpolées passées aux shaders (typiquement dégradé de couleur)

## VARIABLES EN GLSL

Deux étapes pour changer la valeur d'une variable globale depuis Javascript :

1. `getAttribLocation` / `getUniformLocation` : permet de **récupérer** une variable globale GLSL dans le code javascript
2. `vertexAttrib[1234]f` / `uniform[1234]f` : permet de **changer** la valeur de la variable globale via javascript

## VARIABLES EN GLSL

- Les **types** sont les mêmes qu'en C (int, float, etc.). Une addition notable : vec2, vec3, vec4 pour les coordonnées et la couleur.
- Exemple de déclaration de variable globale pour la position dans le vertex shader :

```
attribute vec3 position;
```

## VARIABLES EN GLSL

Il existe des variables globales prédéfinies en GLSL très importantes car elles contrôlent l'**état des vertex/fragments** (il s'agit des variables de sortie) :

1. pour le **vertex shader** : **gl\_Position** (position des vertex),  
**gl\_PointSize** (taille des points)
2. pour le **fragment shader** : **gl\_FragColor** (couleur des vertex affichés à l'écran)

On les assigne dans le **main** des shaders.

## SHADERS

Les étapes de l'utilisation des shaders en javascript sont :

1. Récupération des fichiers .glsl par la fonction :

```
var shaderSource = loadText('shader.glsl');
```

2. Création du shader par :

```
var shader = gl.createShader(type);
```

3. Associer le shader au code du fichier .glsl :

```
gl.shaderSource(shader, shaderSource);
```

4. Compilation des shaders :

```
gl.compileShader(shader);
```

## SHADERS

Enfin, on peut combiner le vertex shader et le fragment shader en un seul programme :

1. Création du programme :

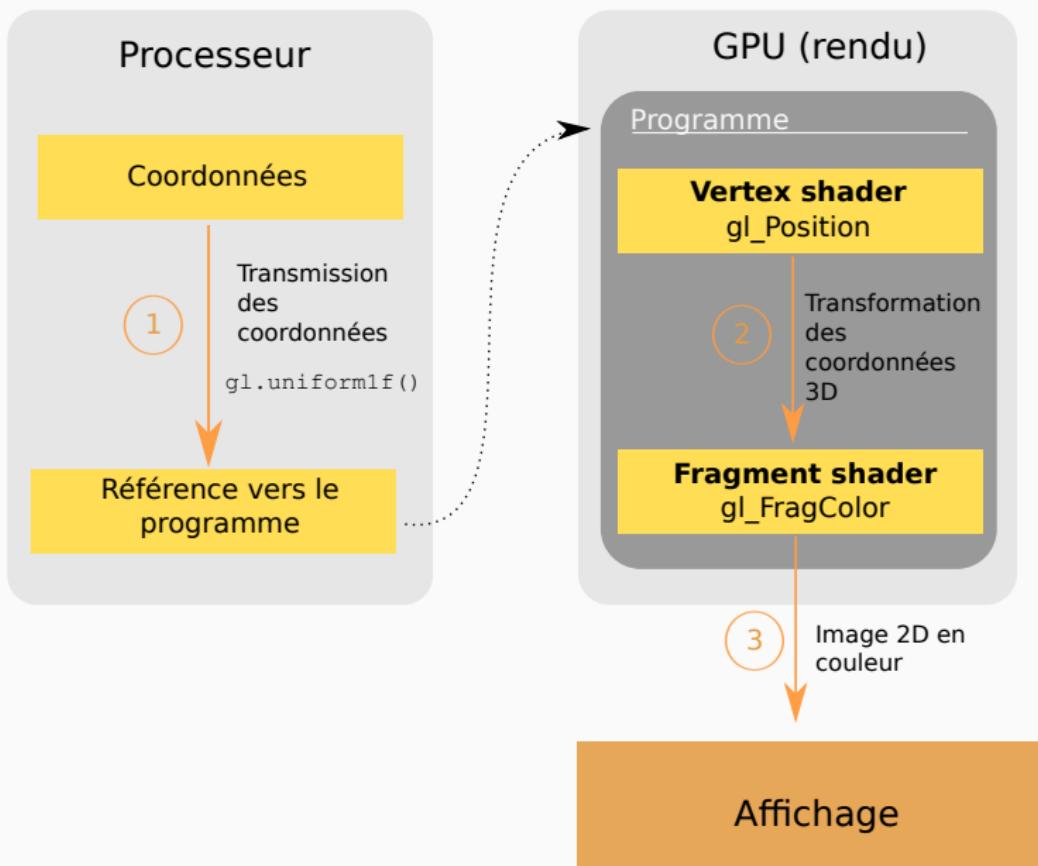
```
var program = gl.createProgram();
```

2. Combiner les deux fichiers de shaders (vertex et fragment) dans le programme :

```
gl.attachShader(program, vertexShader);  
gl.attachShader(program, fragmentShader);
```

3. Edition de liens (cf. compilation C/C++) :

```
gl.linkProgram(program)
```



# SOMMAIRE

Introduction

Premiers pas en WebGL

Shaders et rendu sur une page Web

Buffers

# BUFFER

**Problème :** stocker les données dans la RAM ⇒ rendu lent voire saccadé.

**Solution :** Buffer = espace mémoire réservé sur la carte graphique

Utiliser les buffers permet des échanges de données plus rapides avec le programme ⇒ rendu plus fluide

## Buffer vs shader

- Le buffer va contenir les **données** : sommets, objets...
- Les shaders correspondent au **code** (programme) sur la carte graphique : rendu des données.

Les étapes pour utiliser un buffer sont les suivantes en WebGL :

1. On réserve l'espace mémoire par :

```
var buffer = gl.createBuffer();
```

2. On choisit le buffer sur lequel on veut travailler avec :

```
gl.bindBuffer(type, buffer);
```

3. On remplit l'espace mémoire avec les points que l'on veut dessiner :

```
gl.bufferData(type, points, dessin)
```

A ce stade, l'espace mémoire **buffer** sur la carte graphique contient les données.

Pour dessiner les données contenues dans le buffer :

1. Créer un pointeur vers une variable attribute avec la fonction `gl.vertexAttribPointer(location, size, type, normalized, stride, offset)`.
2. Activer l'assignement du buffer au pointeur créé avec `gl.enableVertexAttribArray(location)`
3. Enfin, dessiner avec `drawArrays` (n'oubliez pas de spécifier le paramètre `count`).

### Ce qui change

`gl.attrib[1234]f` n'est plus nécessaire car les données contenues dans le buffer courant (buffer lié ou "bindé") sont passées directement vers le pointeur.

# BUFFER

Schéma simplifié du fonctionnement de la réservation d'un espace mémoire sur la carte graphique en WebGL :



