WEBGL ET RENDU 3D

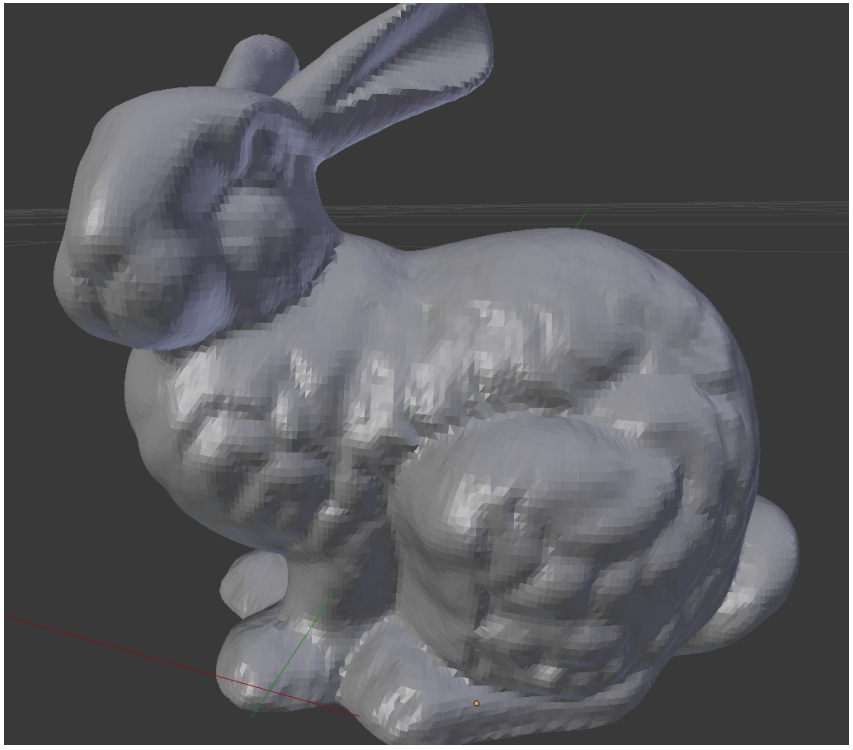
Florent Grélard
florent.grelard@labri.fr

Licence Pro DAWIN, 2016-2017

Sommaire

Rendu 3D

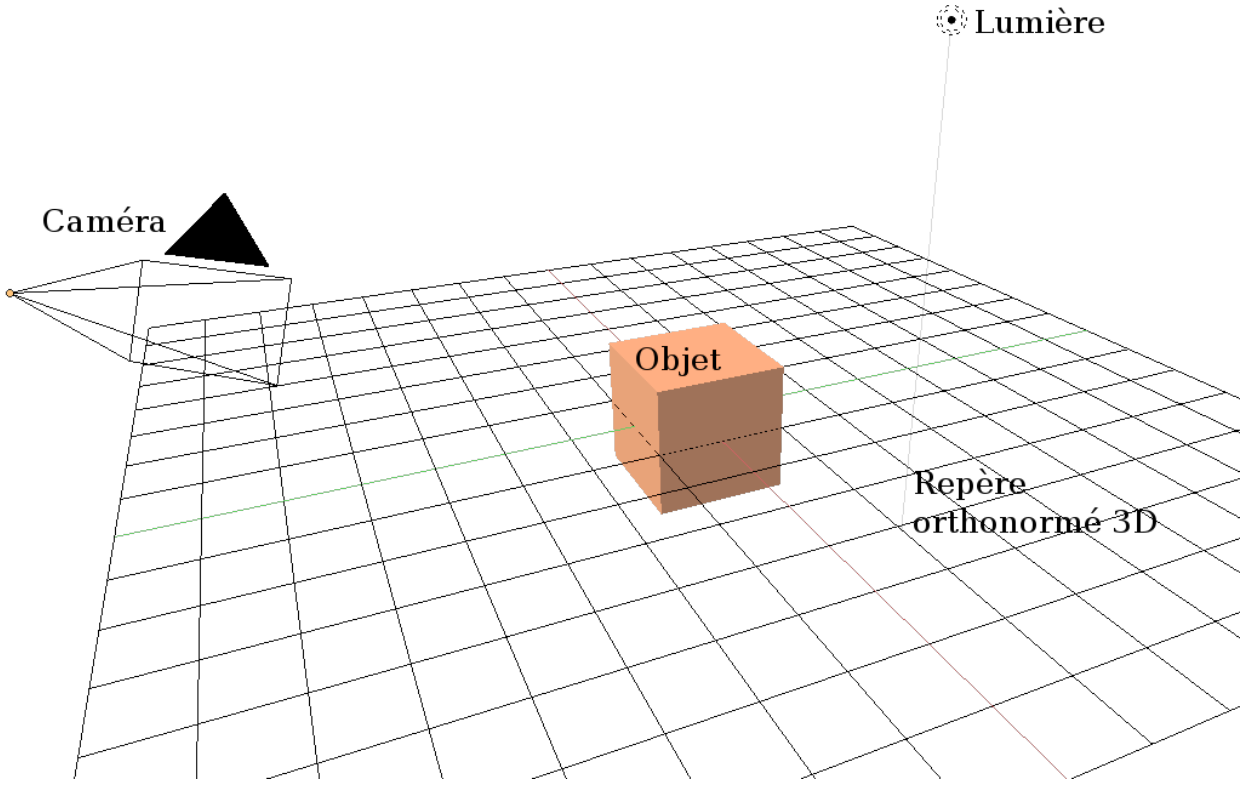
Rendu 3D = images 2D à partir de coordonnées 3D



Rendu 3D

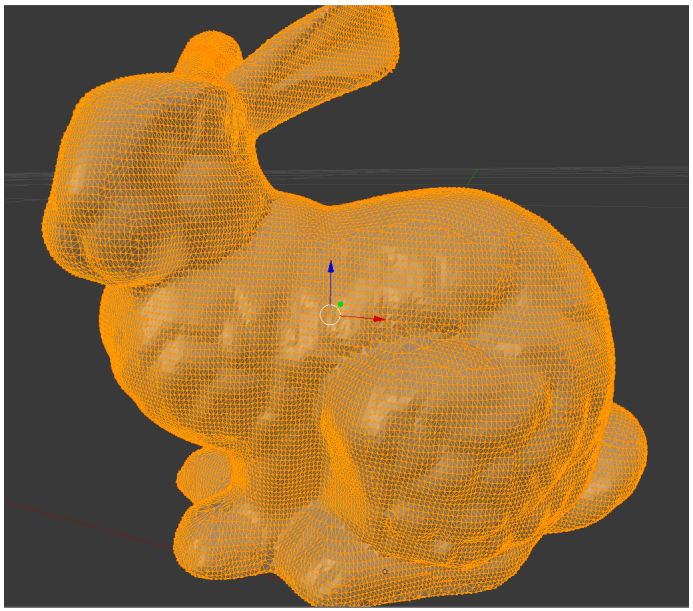
Un peu de vocabulaire...

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



Rendu 3D

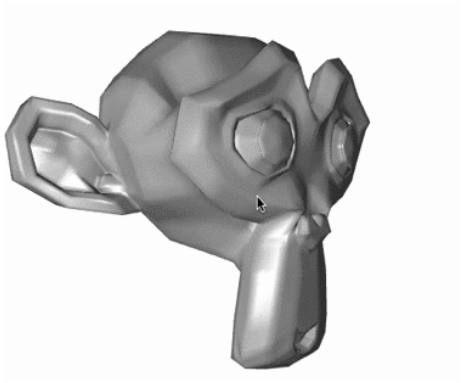
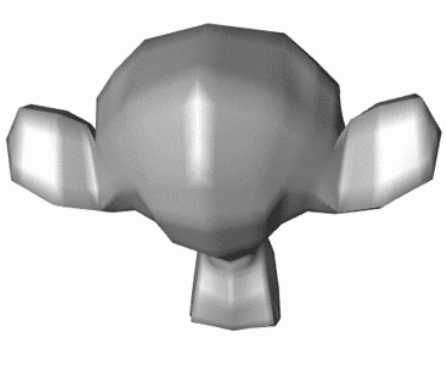
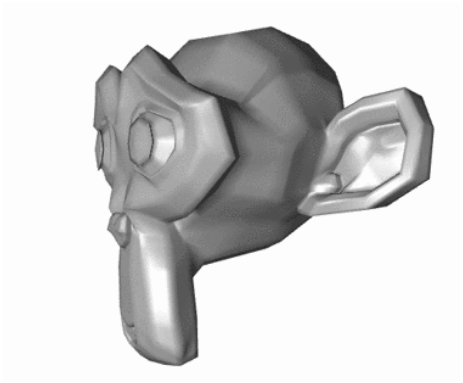
- Un objet est constitué de **sommets** (*vertex* en anglais).
- Triangle = 3 sommets, un carré = 4 sommets, etc.
- Un objet 3D est dessiné en utilisant une multitude de triangles. L'ensemble des triangles définit la surface de l'objet. On parle aussi de **maillage**.



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

$\Rightarrow$ **Projections mathématiques**

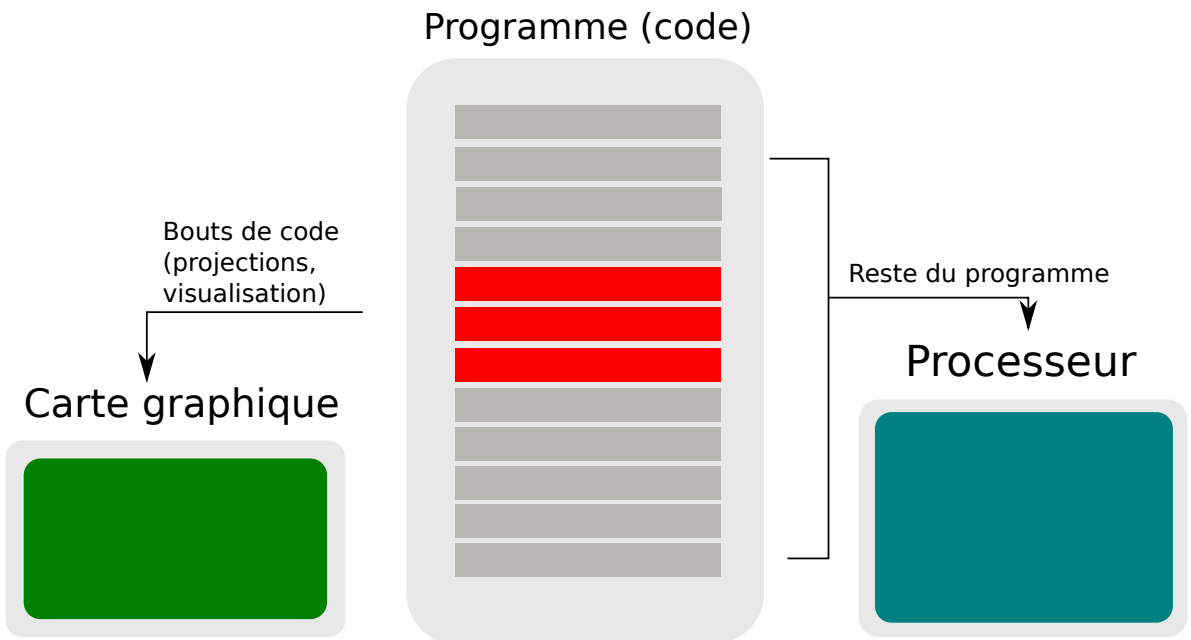
Interactions lumière/objet

- 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

- 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)

Sommaire

Présentation

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

OpenGL

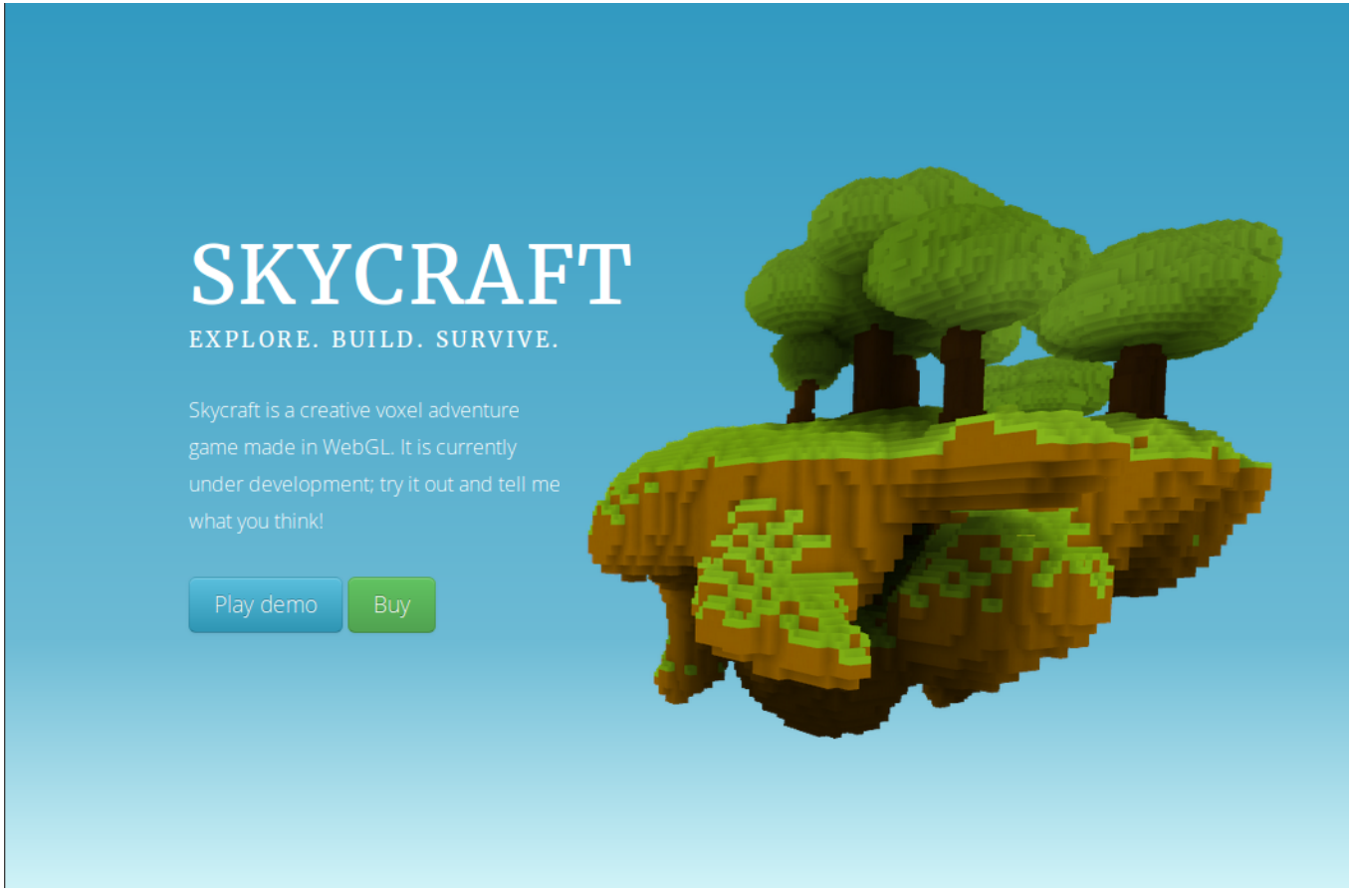
- Communication avec la carte graphique
- Portable (contrairement à DirectX de Microsoft)
- Codé en langage C

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␣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 réelles 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**

$\Rightarrow$ utilisation de la carte graphique :

- mémoire
- processeur interne (GPU)

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

Sommaire

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

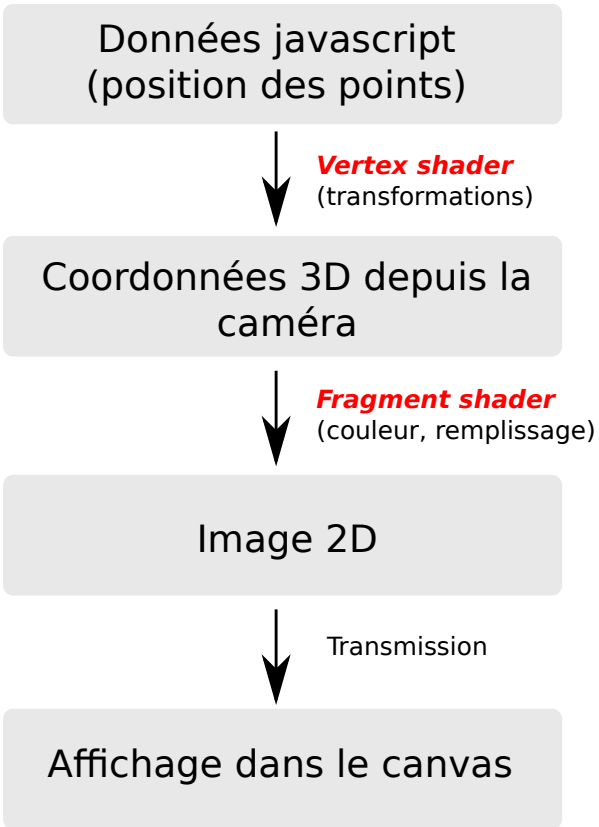
Shaders

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** $\Rightarrow$ 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() {}`

Utilisation des shaders

Les shaders sont des fichiers avec l'extension `glsl`. 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)
```

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 :

```
vec3 position;
```

Variables en GLSL

Des variables globales définies par le standard permettent de **modifier l'état** des points : il s'agit de variables de sortie.

1. Pour modifier la position d'un point : `gl_Position` (vertex shader) ;
2. Pour modifier la taille d'un point : `gl_PointSize` (vertex shader) ;
3. Pour modifier la couleur d'un point : `gl_FragColor` (fragment shader).

On les assigne dans le `main` des shaders.

Dans le code

Fichier javascript :

```
gl = canvas.getContext(...);
program = gl.createProgram();
var points = [ 0.1, 0.2,
              -0.3, -0.5,
               0.1, -0.5 ];
// ICI = Besoin d'assigner les
 coordonnees a gl_Position
gl.drawArrays(gl.POINTS, 0, 1);
```

vertexShader.glsl :

```
void main() {
 gl_Position = ???
}
```

Variables en GLSL

Pour “setter” les variables de sortie, on passe par des variables globales intermédiaires.
Besoin de **descripteurs** de variables globales :

1. `uniform` : peuvent être modifiées ($\neq$ const) mais restent les mêmes pour chaque sommet (généralement la **couleur**)
2. `attribute` : peuvent être toutes traitées individuellement, à chaque appel différent pour chaque sommet (**position**)
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 d'un shader 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

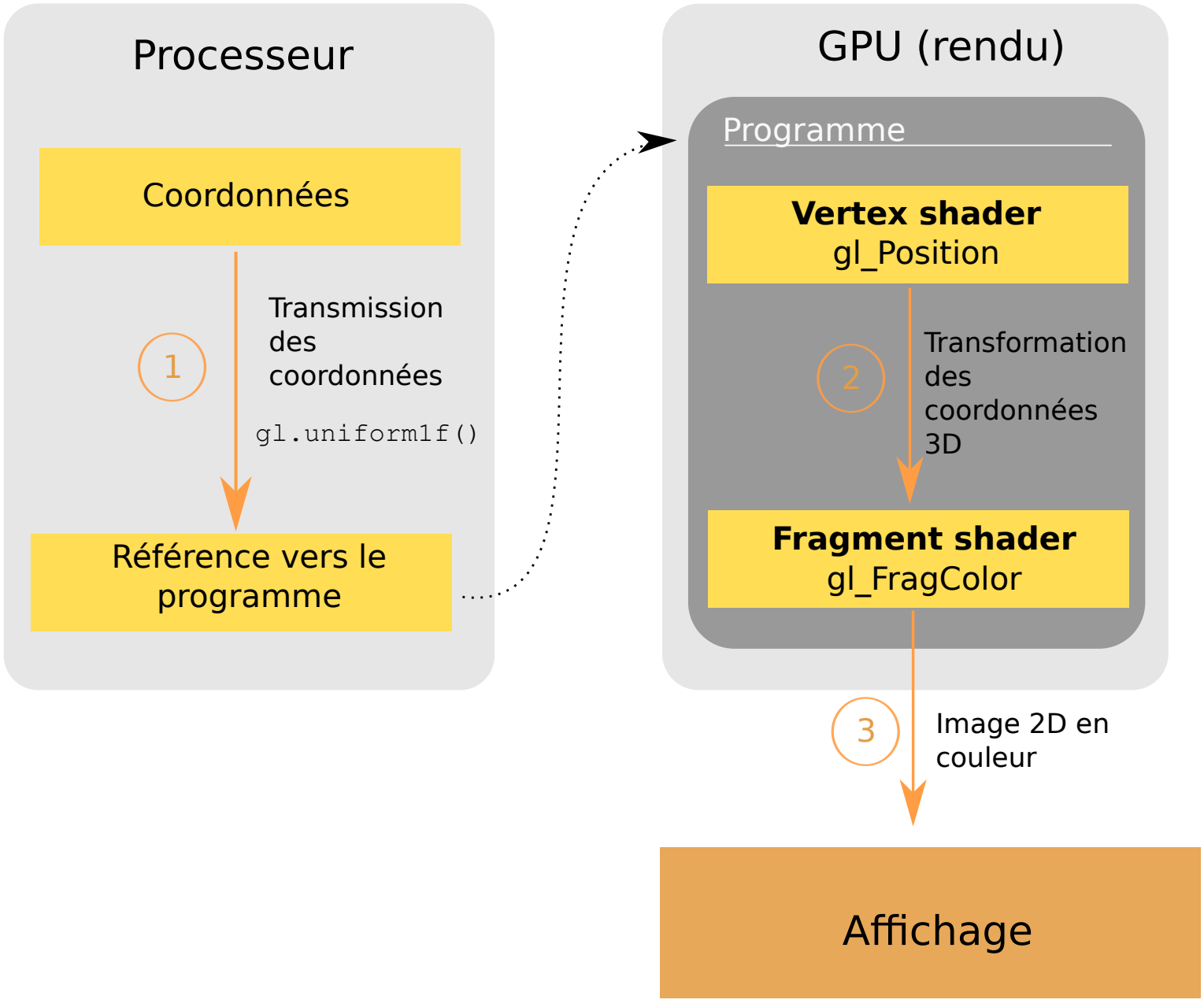
Dans le code

Fichier javascript :

```
1    gl = canvas.getContext(...);
2    program = gl.createProgram();
3    var points = [...];
4    var attr = gl.getAttribLocation(
       program, « position »);
5    gl.vertexAttrib4f(attr, points[0],
       points[1], 0, 1);
6    gl.drawArrays(gl.POINTS, 0, 1);
```

vertexShader.glsl :

```
attribute vec4 position;
void main() {
 gl_Position = position;
}
```


Processeur
Coordonnées
1
Transmission des coordonnées
gl.uniform1f()
Référence vers le programme
GPU (rendu)
Programme
Vertex shader
gl_Position
2
Transformation des coordonnées 3D
Fragment shader
gl_FragColor
3
Image 2D en couleur
Affichage

Sommaire

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

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

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

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

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

Pour **dessiner** les données contenues dans le buffer :

1. Autoriser la transmission des données contenues dans le buffer à la variable attribute `attribVar` : `gl.enableVertexAttribArray(attribVar)`
2. Passer les données contenues dans le buffer à la variable `attribVar` avec : `gl.vertexAttribPointer(attribVar, size, type, normalized, stride, offset)`.
3. Enfin, dessiner avec `drawArrays` (n'oubliez pas de spécifier le paramètre `count`).

Ce qui change

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

Sans buffer

Fichier javascript :

```
1    gl = canvas.getContext(...);
2    program = gl.createProgram();
3    var points = [...];
4    var attr = gl.getAttribLocation(
      program, « position »);
5    gl.vertexAttrib4f(attr, points[0],
      points[1], 0, 1);
6    gl.drawArrays(gl.POINTS, 0, 1);
```

vertexShader.glsl :

```
attribute vec4 position;
void main() {
 gl_Position = position;
}
```

Avec buffer

Fichier javascript :

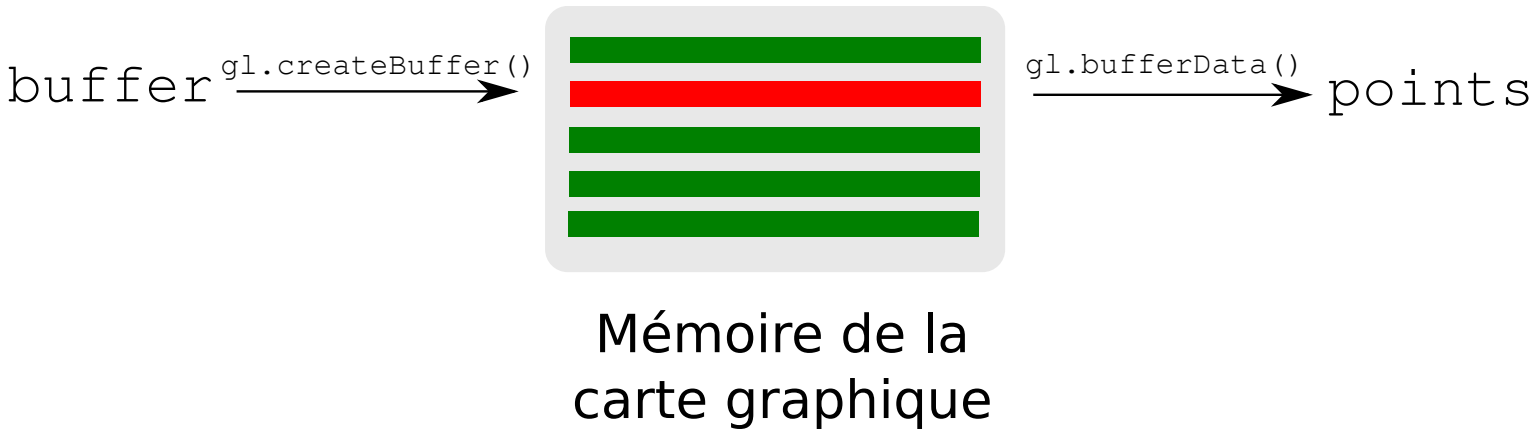
```
1     gl = canvas.getContext(...);
2     program = gl.createProgram();
3     var points = [...];
4     var attr = gl.getAttribLocation(
       program, « position »);
5     var buffer = gl.createBuffer();
6     gl.bufferData(buffer, points, ...);
7     gl.bindBuffer(buffer);
8     gl.enableVertexAttribArray(attr);
9     gl.vertexAttribPointer(attr, ...);
10    gl.drawArrays(gl.POINTS, 0, 1);
```

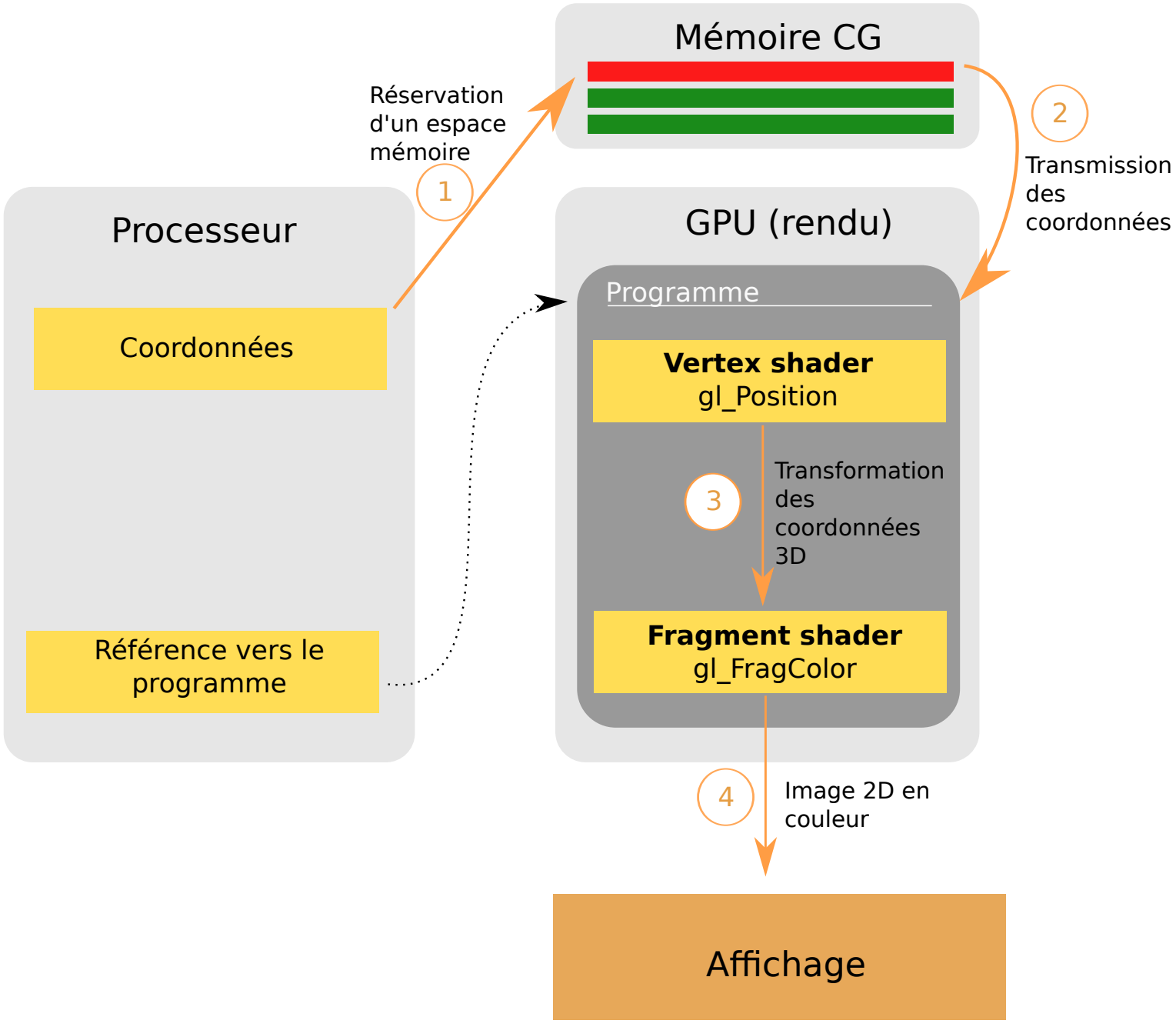
vertexShader.glsl :

```
attribute vec4 position;
void main() {
 gl_Position = position;
}
```

Buffer

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




Mémoire CG
Réservation d'un espace mémoire
1
2
Transmission des coordonnées
Processeur
GPU (rendu)
Programme
Coordonnées
Vertex shader
gl_Position
3
Transformation des coordonnées 3D
Référence vers le programme
Fragment shader
gl_FragColor
4
Image 2D en couleur
Affichage