

# TP 6 : Rendu Offscreen - FBO

## Application #1 : Procedural Texturing

### Introduction

OpenGL possède des buffers par défaut, comme la couleur (**color buffer**) et la profondeur (**depth buffer**) qui sont utilisés pour stocker le résultat de vos commandes de rendu GL. Chaque buffer fait la taille de votre écran et stocke les résultats de votre rendu par pixel. Par exemple, au début de vos méthodes `draw_wgl()`, vous trouverez une commande pour effacer le contenu précédent à chaque affichage d'une nouvelle frame :

```
// Clear the GL "color" and "depth" framebuffers (with OR)
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Pour information : Il y a en réalité d'autres buffers GL par défaut. Notamment pour gérer des animations fluides avec du double buffering : BACK, FRONT qui sont échangés à chaque nouveau rendu. On dessine dans un pendant que l'autre est affiché, puis on swap les 2. Il y a également des buffers pour gérer les images en stéréo (LEFT, RIGHT). Un buffer d'**accumulation** (pour accumuler des résultats. ex : blur, ...) et de **stencil** (pour gérer des masks. ex : reflets, sélection, etc...).

#### FBO : framebuffer object

Il est possible de créer vos propres buffers et de faire votre rendu dedans. On parle de rendu offscreen. Les résultats du rendu vont être dirigés vers des textures (généralement). Pour cela on doit créer et initialiser un objet particulier, un **FBO (framebuffer object)** sur lequel on va **attacher des textures** (ex : "couleurs" et "profondeur"). Cela permet de générer des textures **on-the-fly** (à la volée), à la demande. Par exemple, la texture peut être un pattern/motif généré par un algorithme (on parle de **texture procédurale**).

### Exercice 1 : Mise en place du FBO

Vous allez partir du fichier `tp6_01_fbo_basic.js` qui est disponible sur moodle.

[1] A l'initialisation, il vous faudra :

- Des variables globales :

```
var fbo = null; // le FBO
var tex = null; // texture attachée au FBO et dans laquelle ont fait le rendu
var fboTexWidth = 128; // la taille de la texture
var fboTexHeight = 128; // la taille de la texture
var fullscreen_shaderProgram = null; // shader spécifique pour dessiner un quad à l'écran
```

- Une texture :

1. Creez une texture d'une taille donnée (essayer une petite texture, par exemple: 64x64) sans données à l'intérieur (ici "data = null") avec :
  - `gl.createTexture()`
  - `gl.bindTexture()`
  - `gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, fboTexWidth, fboTexHeight, border, format, type, data);`
2. Spécifiez ses paramètres de filtrage

- `gl.texParameteri(..., gl.TEXTURE_MAG_FILTER)`
- `gl.texParameteri(..., gl.TEXTURE_MIN_FILTER)`

- Un FBO :

- créer un framebuffer object (FBO) : `gl.createFramebuffer()`
- binder le FBO : `gl.bindFramebuffer(...)`
- attacher la texture au FBO : `gl.framebufferTexture2D(target, attachment, texture, level);`
- spécifier la liste des buffers de couleurs dans laquelle on va faire le rendu : `gl.drawBuffers(...)`
- unbind le framebuffer (pour rétablir celui par défaut d'OpenGL) : `gl.bindFramebuffer()`

**NOTE** : dans ce TP, on ne va pas créer de "buffer de profondeur", mais simplement un buffer de couleur.

[2] Au rendu, il vous faudra :

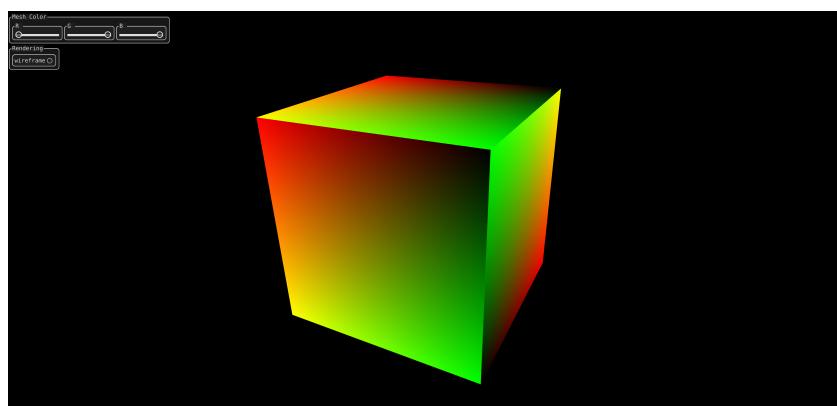
- **1st pass** : on génère le contenu de la texture en mode offscreen via le FBO
  - Bind le FBO : `gl.bindFramebuffer()`
  - Mettre à jour le viewport avec la taille de la texture attachée au FBO : `gl.viewport()`
  - Clear le(s) buffer(s) attaché(s) au FBO (colors, depth...) : `gl.clear()`
  - Bind le bon shader program => ici un quad fullscreen avec écriture dans la texture : `fullscreen_shaderProgram`
  - Appeler le rendu (par exemple avec : `gl.drawArrays(...)`)
- **2nd pass** : on fait le rendu du cube en appliquant la texture comme d'habitude
  - unbind le FBO (on revient au framebuffer par défaut d'OpenGL) : `gl.bindFramebuffer(...)`
  - reset le viewport (l'application WebGL conserve la taille originelle de la fenêtre dans `gl.canvas.width` et `gl.canvas.height`) : `gl.viewport()`
  - clear les buffers par défaut d'OpenGL : `gl.clear(...)`
  - faire le rendu de la scène => affichage du cube et application de la texture procédurale sur ses faces : `shaderProgram.bind()`

## Exercice 2 : Texture procédurale

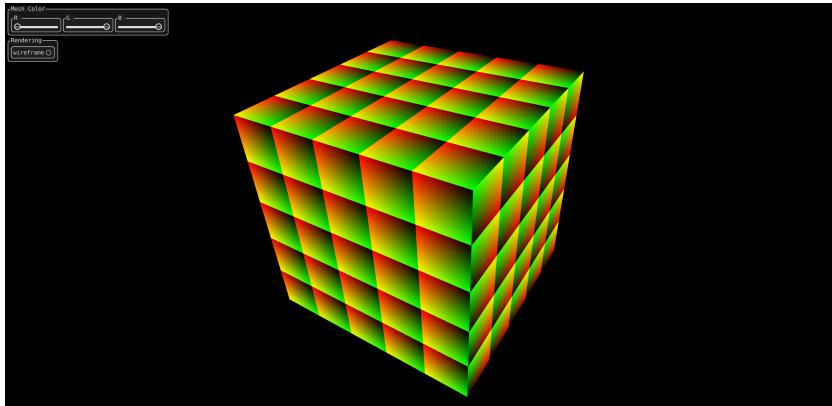
Remplacez la texture appliquée sur un cube, par une texture générée procéduralement dans le fragment shader, via un FBO en rendu offscreen.

Exemple de rendu :

- Afficher les coordonnées UV du cube (espace paramétrique entre 0.0 et 1.0, comme les coordonnées de texture).



- Avec un simple modulo on peut créer des tuiles par répétitions. Par exemple une répétition de 5 patterns avec une période de 1.0 => `vec2 uv = mod(5.0 * texCoord, 1.0);` :



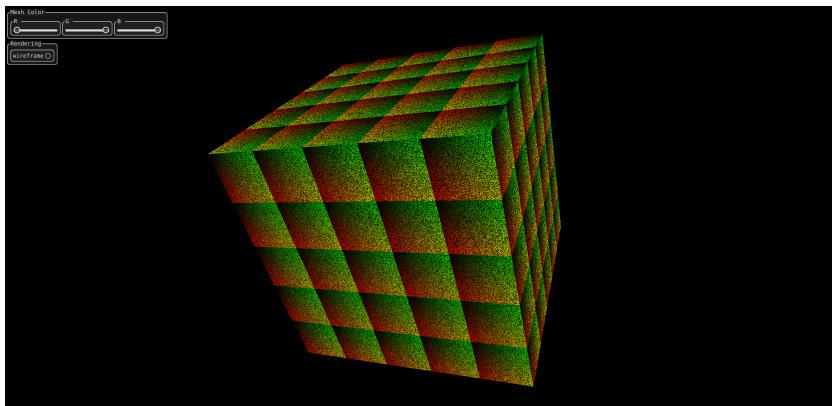
(Essayez des valeurs différentes sur x et y, ça créera des bricks allongées au lieu de cellules carrées)

**NOTE :** Avec cette approche par cellule (ou tuile), vous pourriez créer des mondes infinis par cellule. Voir même générer des valeurs par région afin de s'en servir plus tard pour la génération de monde => des valeurs de densité de forêt, etc... Il est même possible de générer des valeurs différentes par cellule grâce à un générateur de nombres aléatoires PRNG (pseudo-random number generator).

- On peut aussi ajouter du bruit avec une fonction `noise()` dans le fragment shader :

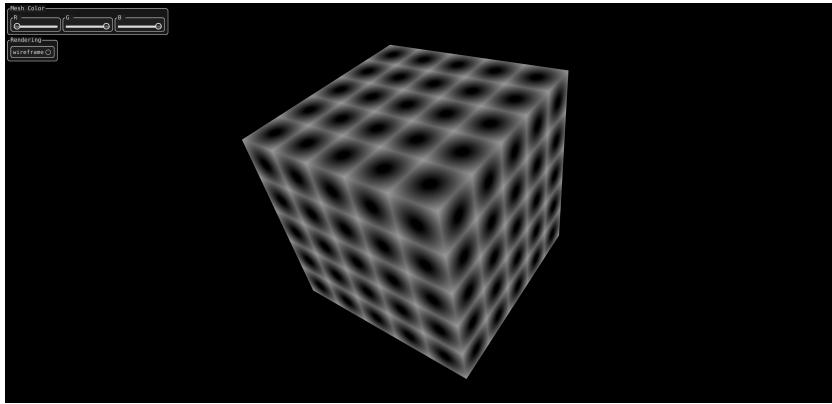
```
float noise(vec2 st)
{
    return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 43758.5453123);
}
```

On peut utiliser l'instruction `discard` (qui va stopper l'exécution du shader sans mettre à jour la couleur dans le framebuffer) si la valeur du bruit est inférieur à un certain seuil :



### Pour aller plus loin :

- Au lieu d'afficher les coordonnées UV en couleur, vous pourriez utiliser ces coordonnées pour dessiner des formes 2D procédurales par "tuile". Exemple : équation d'un cercle => créer des cercles centrés sur chaque tuile. Cela correspond à des **fonctions implicites** qui consistent à représenter des formes par des fonctions de distances (signed distance field) => 0 sur le bord, < 0 à l'intérieur et > 0 à l'extérieur de la forme. Idée : Rajoutez du noise sur les distances afin de déformer vos formes 2D.

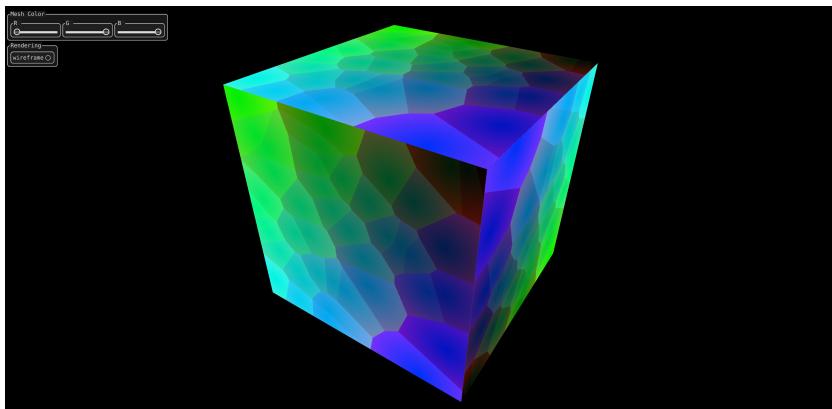


Catalogue de fonctions de distance 2D :

<https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm>

**NOTE** : les fonctions ci-dessus sont faites avec un repère en (0;0) au centre d'une image. Donc, par cellule, comme vos coordonnées variant localement de 0 à 1 avec (0;0) en bas à gauche, pensez à changer de repère comme ceci => "uv - vec2(0.5)" pour avoir un repère (0;0) au centre de vos cellules et envoyez cette valeur aux fonctions de distance précédentes.

- Pour quelque chose de moins régulier, on peut créer un motif avec le diagramme de **Voronoi**. Il s'agit de définir des centroïd (ici, des points en 2D générés aléatoirement dans les coordonnées de texture), puis, pour chaque fragment, de définir à quel centroïd il est le plus proche (distance euclidienne). Selon la position du centroïd le plus proche et de la distance du fragment à ce centroïd, on peut définir une couleur :



**NOTE** : Megakernel vs wavefront approach.

On aurait pu générer la texture procédurale dans le shader du cube. Mais il est parfois préférable de séparer un gros shader (megakernel) en plusieurs shaders (wavefront approach). Par exemple, un gros shader avec plein de code, des **if/else**, **for**, **while**, va consommer de nombreux registres hardware qui sont en nombre limité. Des délais d'attente vont alors apparaître pour obtenir les données, qui vont devoir transiter par de la mémoire globale très lente (pour stocker des résultats/variables intermédiaires) au lieu de registres locaux très rapides. De manière générale, moins de threads pourront être exécutés en parallèle par le scheduler du GPU afin de masquer les temps d'attente aux ressources (accès à une texture, etc...).

**Encore une note ...** : UBER-Shaders "avoid branching".

Essayer de coder les différentes fonctionnalités (types de rendu) en utilisant des macro du préprocesseur GLSL : **#define**, **#if**, **#else**, **#elif**. Cela permet de faire des shaders génériques en incluant tous les effets possibles. À la compilation du shader, les programmes peuvent parser les fichiers "glsl" et activer des fonctionnalités en ajoutant des **#define** au code puis en compilant autant de shaders qu'il y a d'options. De cette manière, le maximum de registres hardware sont utilisés pour une fonctionnalité spécifique, au lieu d'utiliser une variable de type **uniform** pour sélectionner une fonctionnalité particulière et faire plein de **if/else** qui monopoliseraient des ressources hardware pour rien.

Exemple :

```
void main()
{
#define MODE 2
#if MODE == 0
oColor = ...
#elif MODE == 1
oColor = ...
#else
oColor = ...
#endif
}
```