TP Textures OpenGL
malek.bengougam@gmail.com

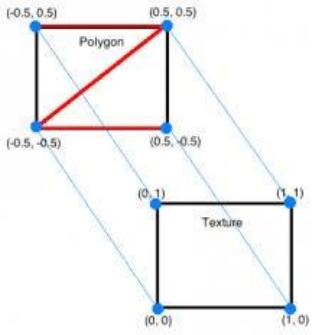
Voici un petit panorama des fonctionnalités de *texturing* en OpenGL classique et ES2/webgl1.

Nomenclature

Bitmap : représentation binaire d'une image caractérisée par ses dimensions (longueur x hauteur), le nombre de composantes couleurs (R, G, B, A) et le type de donnée de chaque composante (byte, short, int, float).

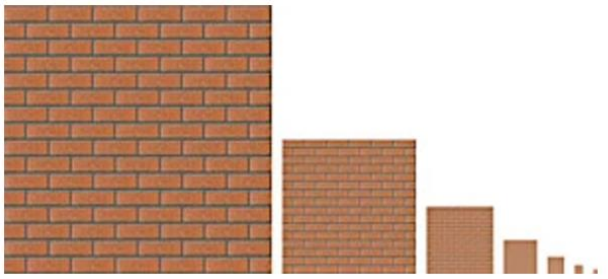
Texture : image appliquée à une surface par le rasterizer. Une texture est caractérisée par un « format » composé du type de donnée et des composantes couleurs. Le type de donnée et le format peuvent être différent du type de donnée de la bitmap (sauf sur mobile/web).

Sampler : « échantillonneur » en français, permet d'accéder au contenu d'une texture. Le sampler est configurable et permet d'indiquer comment appliquer la texture sur un polygone en fonction de différents paramètres tels que l'enroulement (wrapping) des coordonnées de texture, le filtre à utiliser lorsque la texture est plaquée sur une surface plus grande que l'image, le filtre à utiliser lorsque la texture est plaquée sur une surface plus petite que l'image, et d'autres réglages plus particuliers.



Coordonnées de texture : attribut indiquant quelle partie de la texture doit être attachée à un sommet d'un polygone. Afin de rendre le mécanisme de « sampling » générique (essentiellement indépendant des dimensions en pixels) les coordonnées de texture sont exprimées de manière normalisée entre [+0,+1]. Il est possible de définir des coordonnées négatives ou supérieures à 1.

Niveau de détail (Level of Detail, LoD) : correspond à différents niveaux de détail des textures en suite "pyramidale". La base représente le plus haut niveau de détail (niveau 0), qui correspond à la texture originelle. Les niveaux suivants représentent la même image mais à chaque fois deux fois plus petite que le niveau de détail précédent. On parle alors généralement de "mip maps" : mip étant l'acronyme du latin "*multum in parvo*".



Créer et configurer une texture OpenGL.

Avant OpenGL 3.x un Texture Object représente tout à la fois la texture et le sampler. C'est-à-dire qu'un Texture Object gère le stockage ainsi que les paramètres de lecture/sampling.

Un Texture Object étant un objet OpenGL classique on utilise donc des fonctions glGen*/glDelete* pour créer et détruire un objet (cf slides OpenGL Partie 03 pour le détail).

```
// A l'initialisation
glGenTextures(1, &texID);

// Pour détruite le Texture Object et tout son contenu
glDeleteTextures(1, &texID);
```

Nécessairement pour indiquer à OpenGL que l'on va éditer le Texture Object il faut le lier à un mode d'utilisation. Pour nous, dans la majeure partie des cas ce sera en mode texture à 2 dimensions.

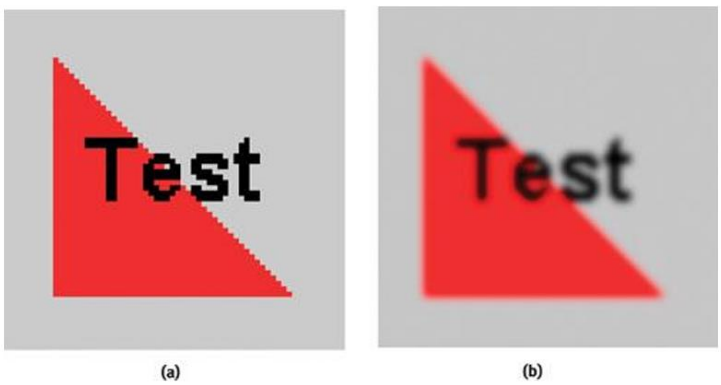
```
glBindTexture(GL_TEXTURE_2D, texID);
```

Par défaut, du fait qu'OpenGL est une API 3D, le mode de sampling en minification (polygone plus petit que la texture) **GL_TEXTURE_MIN_FILTER** suppose l'utilisation du mipmapping (valeur **GL_NEAREST_MIPMAP_LINEAR**, filtrage bilineaire et mipmap la plus proche).
Dans le cas où seul le niveau de détail de base est fourni (level 0) ce filtre risque de ne pas afficher de texture dans la majeure partie des cas car les autres niveaux de détail sont manquants.

```
// specifiez un filtrage bilineaire ou plus proche si pas de mipmap disponibles
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Les modes d'enroulement usuels (wrapping):

Filtrage : différence entre plus proche voisin (a) et bilinéaire (b)



Alternativement il est possible de demander à OpenGL de créer les mipmaps mais c'est relativement coûteux et le filtre de base du pilote risque de ne pas convenir aux artistes.

Dans notre cas cela reste cependant la méthode la plus simple pour obtenir ces niveaux de détail (LoD) de texture:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Le chargement des données d'une bitmap 2D vers une texture 2D s'effectue via la fonction

```
glTexImage2D(target, lod, internal format, width, height, border, format, datatype, pointer)
```

Les paramètres de glTexImage2D sont les suivants :
- 1, la "cible" (***target***) : ce sera majoritairement GL_TEXTURE_2D pour nous
- 2, **le niveau de détail actuel** : on se contente de charger le plus haut niveau de détail (niveau 0)
- 3, **le format interne** : format de stockage des texels en VRAM, ici RGBA8 le plus souvent c'est-à-dire RGBA sur 8 bits
- 4 et 5, les **dimensions de la texture** (identiques pour la bitmap et la texture)
- 6, l'identifiant d'une couleur de **bordure**, mettez zéro ici, rarement utile
- 7 et 8, il s'agit du **format des données source**. Par exemple, GL_RGBA indique 4 composants et GL_UNSIGNED_BYTE le type de chacun des composants contenus dans le dernier paramètre
- 9, l'**adresse mémoire** : spécifiez NULL si vous ne souhaitez qu'allouer (malloc) de la mémoire ou bien une adresse réelle si vous souhaitez en plus effectuer une copie (memcpy).

Exemple 1 : création d'une texture blanche de taille 1x1 en RGBA

```
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);

// Filtrage bilineaire dans tous les cas (Minification et Magnification)
// les coordonnees de texture sont limitees a [0 ; 1[
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

uint8_t data[] = { 255, 255, 255, 255 };
// Notez que le format interne GL_RGBA8 est typé, chaque composant est de 8 bits
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 1, 1, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

Exemple 2 : creation d'un texture 2x2 avec Mipmap auto générée et filtrage trilinéaire

```
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);

// Filtrage trilinéaire en minification et bilineaire en magnification
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Si rien n'est specifie pour GL_TEXTURE_WRAP_* c'est GL_REPEAT par defaut

uint8_t data[] = { 255,0,0,255, 0,255,0,255, 0,0,255,255, 255,255,255,255 };
// Notez que le format interne GL_RGBA8 est typé, chaque composant est de 8 bits
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 2, 2, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

Charger une bitmap.

Nous allons utiliser la bibliothèque **stb** https://github.com/nothings/stb qui supporte l'ensemble des formats que nous allons étudier lors du cours d'OpenGL. Après avoir téléchargé la lib stb et configuré le chemin des includes, tapez le code suivant :

```
// ATTENTION! Le define suivant ne doit se trouver que dans un seul et unique .cpp
#define STB_IMAGE_IMPLEMENTATION
#include "stb/stb_image.h"
```

Lorsque les canaux de couleurs des images sont sur 8 bits (1 byte) on utilise les fonctions suivantes:

```
uint8_t *data = stbi_load(filename, &width, &height, &comp, desiredComp);
```

'`filename`' est le nom + chemin du fichier, et `desiredComp` indique les composantes finales de la texture (valeur usuelles : `STBI_rgb ou STBI_rgb_alpha`).
Les informations retournées par la fonction sont les dimensions '`width, height`' et le nombre composantes '`comp`' couleur de l'image.

Généralement il n'est pas nécessaire de préserver la bitmap une fois que la texture a été créée. On libère la mémoire allouée pour la bitmap à l'aide de la fonction :

```
stbi_image_free(data) // data = pointeur retourné par stbi_load()
```

Exemple 3 : chargement de la texture par le contenu de la bitmap

```
int w, h;
uint8_t *data = stbi_load(filename, &w, &h, nullptr, STBI_rgb_alpha);
if (data) {
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
   stbi_image_free(data);
}
```

Utilisation des textures

Comme pour tout objet OpenGL il faut faire un appel à glBind*, ici glBindTexture().
Le GPU est capable d'accéder à plusieurs textures en même temps, il faut donc indiquer à OpenGL et aux shaders sur quel canal on accède à notre texture. Par défaut c'est le canal 0 (GL_TEXTURE0).
On utilise la fonction glActiveTexture() pour spécifier cela. In extenso cela donne :

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureA);
// une autre texture mais sur le canal 1
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textureB);
```

Côté shader, on a besoin de deux informations pour 'sampler' une texture. Premièrement il faut indiquer où lire la texture, et deuxièmement il faut fournir les coordonnées de texture du fragment.

Les coordonnées de texture sont récupérées dans le ***vertex shader*** sous forme d'attribut. Il suffit de les passer en varying pour que le rasterizer produise les coordonnées de texture de chaque fragment.

```
attribute vec2 a_texcoords;
varying vec2 v_texcoords;
…dans le main …
v_texcoords = a_texcoords;
```

il existe un type de donné spécifique à chaque type de texture supporté. Dans le cas GL_TEXTURE_2D on doit définir une variable de la manière suivante dans le ***fragment shader*** :

```
// dans le fragment shader (ou le vertex shader pour certains types de texture)
uniform sampler2D u_sampler;
```

Par défaut le shader va lire l'unité de texture/canal 0. Dans le cas où l'on souhaite qu'il accède à un canal différent on doit modifier la variable uniform. Le type sampler2D hérite de int donc on utilise :

```
// en C++ on indique que le shader 'sample' l'unite de texture 1
auto locationTexture = glGetUniformLocation(program, "u_sampler");
glUniform1i(locationTexture, 1);
```

Le *fragment shader* utilise alors la fonction **vec4** texture2D(**sampler2D, vec2**) afin de sampler la texture aux coordonnées de texture calculées par le rasterizer. Voici un exemple complet :

```
varying vec2 v_texcoords;
uniform sampler2D u_sampler;
void main(void) {
    vec4 color = texture2D(u_sampler, v_texcoords);
    gl_FragColor = color;
}
```

Exercices

- Exercice 1 : ajoutez un attribut supplémentaire à vos vertices correspondant à des coordonnées de textures (entre 0 et 1). Utilisez la texture de l'exemple 2 et modifiez le filtrage en minification et magnification de **GL_LINEAR** à **GL_NEAREST** et vice-versa, que constatez-vous ?
Optionnel: modifiez les coordonnées de texture pour qu'elles soient en dehors de [0;1] et jouez cette fois-ci avec les paramètres de **WRAP_S** et **WRAP_T**, que constatez-vous ?

- Exercice 2 : Utilisez l'exemple 3 à la place de l'exemple 2 afin de charger une texture depuis le disque.
N'oubliez pas de specifier les paramètres de texture, et optionnellement, activez le filtrage trilinéaire (nécessite la presence de mip-maps).

- Exercice 3 : chargez deux textures différentes sur deux slots différents et mélangez les dans votre fragment shader avec les operations qui vous passent par la tête (+, * etc…).

Note: Il n'est ni necessaire ni utile ici d'avoir deux jeux de coordonnées de texture. Vous pouvez donc utiliser les mêmes UV pour les deux textures.