TD OPENGL MODERNE

- LES BASES -

Malek.bengougam@gmail.com

Ce cours suppose l'utilisation de la classe GLShader qui encapsule les fonctionnalités de chargement, compilation et linkage des shaders dans un shader program.

Nous utiliserons également glew afin de pouvoir facilement charger les extensions qui nous intéressent (ceci n'est pas utile sous MacOS X car elles sont déjà exposées).

Je vous renvoie aux cours pour le détail des fonctionnalités ainsi qu'à votre moteur de recherche, mais n'hésitez pas à me poser des questions si nécessaire.

La méthodologie choisie est une approche du bas vers le haut, en abordant en premier lieu les commandes effectives de rendu avant d'aborder la configuration du rendu.

Dans le cadre de ce document nous allons nous limiter à trois fonctions que sont l'initialisation, la terminaison et le rendu.Pour un bon fonctionnement de l'application il faut également une fonction de rappel (*callback*) lorsque la fenêtre est redimensionnée, ainsi qu'une fonction de mise à jour de la simulation, (Update).

A. Fonctionnalités de base

I. Création et compilation des shaders

Voici un premier exemple d'utilisation de la classe GLShader :

```
GLShader g_BasicShader;

bool Initialise()
{
	g_BasicShader.LoadVertexShader("basic.vs");
	g_BasicShader.LoadFragmentShader("basic.fs");
	g_BasicShader.Create()

	// cette fonction est spécifique à Windows et permet d'activer (1) ou non (0)
	// la synchronization vertical. Elle necessite l'include wglew.h
	#ifdef WIN32
	wglSwapIntervalEXT(1);
	#endif
	return true;
}

void Terminate() {
	g_BasicShader.Destroy();
}
```

Le code précédent fonctionne avec toutes les versions des pilotes OpenGL (hormis les versions pré-OpenGL 2.0 mais cela ne nous concerne plus).

Par défaut, dans les cours qui vont suivre nous spécifions toujours la version minimale du shader à la version 120 (OpenGL 2.1) qui est la plus répandue des versions modernes d'OpenGL. Lorsque cela s'avèrera nécessaire nous indiquerons une version spécifique..

Nous avons vu précédemment, la syntaxe des shaders en OpenGL ES 2.0 (#version 100 es) et en OpenGL 2.0 (#version 110) et OpenGL 2.1 (#version 120):

Basic.vs

```
attribute vec2 a_position;
attribute vec3 a_color;

varying vec4 v_color;

void main(void) {
	gl_Position = vec4(a_position, 0.0, 1.0);
	v_color = vec4(a_color, 1.0);
}
```

Basic.fs

```
varying vec4 v_color;

void main(void) {
	gl_FragColor = v_color;
}
```

II. Rendu à base de triangles

Nous allons maintenant nous intéresser à la fonction de rendu principale.
Une passe de rendu peut se résumer aux étapes suivantes :

a. définition du viewport
b. effacement des buffers du framebuffer (généralement au moins le back buffer)
c. spécification du shader à utiliser
d. définition d'une géométrie
e. paramétrer le rendu (optionnelle)
f. rendu d'une géométrie

Nous verrons par la suite comment se passer de l'étape d dans la boucle de rendu.

```
void Render()
{
	// etape a. A vous de recuperer/passer les variables width/height
	glViewport(0, 0, width, height);

	// etape b. Notez que glClearColor est un etat, donc persistant
	glClearColor(0.5f, 0.5f, 0.5f, 1.f);
	glClear(GL_COLOR_BUFFER_BIT);

	// etape c. on specifie le shader program a utiliser
	Auto basicProgram = g_BasicShader.GetProgram();
	glUseProgram(basicProgram);

	// etape d.

	// etape e.

	// etape f. dessin de triangles dont la definition provient d'un tableau
	// le rendu s'effectue ici en prenant 3 sommets a partir du debut du tableau (0)
	glDrawArrays(GL_TRIANGLES, 0, 3);

	// on suppose que la phase d'echange des buffers front et back
	// le « swap buffers » est effectuee juste apres
}
```

III. Spécifications des attributs (étape d)

L'étape 'd' est cruciale car c'est ce qui permet à OpenGL d'indiquer au GPU comment interpréter les données à fournir aux variables de type 'attribute' dans le Vertex Shader.

Il s'agit d'utiliser les fonctions glVertexAttrib**() mais comme nos données sont sous formes de tableau il faudra donc le spécifier à OpenGL et qui plus est spécifier comment les données sont structurées en mémoire. Prenons un exemple simple, 3 sommets en 2D:

```
	static const float triangle[] = {
		-0.5f, -0.5f,
		0.5f, -0.5f,
		0.0f, 0.5f
	};
```

Chaque vertex est composé d'un seul attribut qui est la position du sommet et chaque position est composée de 2 float-s. Cela implique que les attributs d'un sommet sont séparés des attributs du sommet suivant par une distance en octet de sizeof(float) * 2.

Ce descriptif est exactement ce dont a besoin OpenGL (et le GPU)afin d'interpréter correctement les données et produire le rendu escompté.

Ceci se traduit par le code OpenGL suivant :

```
// le premier parametre = zero correspond ici au canal/emplacement du premier attribut
// glEnableVertexAttribArray() indique que les donnees sont generiques (proviennent
// d'un tableau) et non pas communes a tous les sommets
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 2, triangle);
```

glVertexAttribPointer(index, taille, type, normalisation, écart, adresse)
https://www.khronos.org/opengl/wiki/GLAPI/glVertexAttribPointer

index : il s'agit du canal ou registre correspondant a une variable 'attribute' dans le shader
taille: il s'agit en fait du nombre de composantes par attribut, donc 1, 2, 3 ou 4
type: un valeur énumérateur indiquant le type des données
normalisation : les données doivent-elles être normalisées durant le transfert (généralement non)
écart : 'stride' en anglais, l'écart en octet séparant les données N d'un attribut des données N+1
adresse : l'adresse de la première donnée d'un attribut

Le code est encore imparfait en OpenGL (ES) 2. En effet, il n'est pas garanti que le canal / location 0 corresponde bien à l'attribut de a_position que nous avons déclaré dans le vertex shader.
Pour s'en assurer il faut utiliser une commande OpenGL pour faire une requête sur le shader actuellement actif –après glUseProgram() donc.

int location = **glGetAttribLocation**(GLuint program, const char* name)

Cette fonction retourne le canal/location de l'**attribut** 'name' d'un shader program qui s'estlié correctement (sans erreurs). Le code précédent devient alors :

```
int loc_position = glGetAttribLocation(program, "a_position");
glEnableVertexAttribArray(loc_position);
glVertexAttribPointer(loc_position, 2, GL_FLOAT, false, sizeof(float) * 2, triangle);
```

Exercice A.1 –

Ajoutez un attribut couleur au tableau triangle. Les couleurs des sommets doivent être différentes de manière à ce que votre objet (triangle ici) s'affiche sous la forme d'un dégradé. Il est donc impératif de communiquer l'information de couleur à l'attribut « a_color ».

Prêtez bien attention à l'écart ainsi qu'à l'adresse de la première donnée de chaque attribut.

Exercice A.2 –

Remplacez votre tableau de float par un tableau de Vertex où Vertex est une structure composée des attributs position (2D) et color (3D). Il est très fortement recommandé de créer une structure/classe pour chacun des types fondamentaux que l'on utilisera pour les points, vecteurs et couleurs en 2D et 3D (voire 4D).