OpenGL - TD 01

Initiation à OpenGL et GLFW

L'objectif de cette première partie des TP est de vous initier à OpenGL (en 2D, cette année). Ce premier TP va également vous permettre de prendre en main GLFW, un système de fenêtrage et d'Interface Homme Machine (**IHM**), afin de créer une fenêtre dans laquelle un rendu OpenGL pourra être effectué. Nous verrons également comment intégrer la gestion des événements de type clavier, souris, etc…

Exercice 01 – Késako

Avant de commencer à travailler, il serait bon de savoir quels outils nous utilisons.

Questions :

À l'aide d'internet, définissez rapidement :

01. Qu'est-ce qu'OpenGL ?
02. Qu'est-ce que GLFW ?

Exercice 02 – Votre première fenêtre

Maintenant que vous savez ce que sont OpenGL et GLFW, il est temps de s'en servir. Cet exercice ne vous demandera pas d'écrire de code, il vise à vous montrer la manière dont nous allons travailler, notamment via l'outil de compilation **make**.

L'archive *TD01.zip* est disponible sur le site de votre chargé de TD.

Le fichier *minimal.c* situé dans le répertoire *TD01/doc/*, constitue la base de code sur laquelle nous travaillerons avec OpenGL et GLFW.

Vous trouverez également le fichier *makefile*, à la racine du répertoire *TD01/*

Questions :

01. Que signifient les directives `#include` au début de *minimal.c* ?
02. Quelles sont les bibliothèques utilisées pour OpenGL et GLFW ?
03. Comment sont elles appelées à la compilation dans le *makefile* ?

Le fichier source pour ce programme est donné dans *TD01/src/ex02/* (Il s'agit d'une simple copie de *minimal.c*)

Pour compiler à partir du makefile, ouvrez un terminal dans *TD01/*, et entrez la commande :

```
make ex02
```

Le programme *td01_ex02.out* est crée dans le répertoire *TD01/bin/*. Pour l'exécuter , entrez la commande (toujours à partir de *TD01/*) :

```
bin/td01_ex02.out
```

Note – Organisation d'un répertoire de TD

En faisant l'exercice 02, vous avez pu prendre en main la compilation et l'exécution d'un programme utilisant OpenGL et GLFW. Vous aurez sans doute remarqué que les fichiers de travail se trouvent à différents endroits. Et peut être même remarqué l'apparition d'un répertoire *TD01/obj/*

Un répertoire de TD est pensé comme un objet indépendant, vous pourrez l'extraire du .zip sur votre machine la où vous le souhaitez, et travailler immédiatement dedans (ou presque).

Le répertoire pour chaque TD sera divisé en sous-répertoires :

- *bin/* : les exécutables crées via **make** seront placés ici
- *doc/* : contient divers fichiers de référence, notamment *minimal.c* et l'énoncé du TD
- *src/ :* contient les fichiers sources sur lesquels vous travaillez pour chaque exercice
- *obj/* : contient les fichiers **.o* générés lors de la compilation

Avant de démarrer un nouvel exercice, vous aurez besoin de faire ceci :

- créer un répertoire pour votre exercice dans */src*, comprenant un nouveau fichier source
- mettre a jour le *makefile* du TD, pour qu'il puisse trouver ce fichier et le compiler

A faire :

01. Créez le répertoire *TD1/src/ex03/*

```
mkdir -p src/ex03
```

02. Copiez le fichier *minimal.c* dans ce répertoire en le renommant *td01_ex03.c* (Une autre option est de repartir du fichier source de l'exercice précédent)

```
cp doc/minimal.c src/ex03/td01_ex03.c
```

```
cp src/ex02/td01_ex02.c src/ex03/td01_ex03.c
```

03a. Ouvrez *TD1/makefile*

```
gedit makefile
```

03b. Ajoutez les lignes suivantes, dans la partie `# Fichiers TD 01` :

```
# Fichiers exercice 03
OBJ_TD01_EX03= ex03/td01_ex03.o
EXEC_TD01_EX03= td01_ex03.out
```

Ajoutez les lignes suivantes, dans la partie `# Regles compilation TD 01` :

```
ex03 : $(OBJDIR)$(OBJ_TD01_EX03)
    $(CC) $(CFLAGS) $(OBJDIR)$(OBJ_TD01_EX03) -o $(BINDIR)$(EXEC_TD01_EX03)
$(LDFLAGS)
```

Vous êtes désormais en mesure de compiler les fichiers pour l'exercice 03 et de générer l'exécutable *td01_ex03.out* dans le répertoire *TD01/bin*, en entrant simplement :

```
make ex03
```

Exercice 03 – Le domptage de la fenêtre

La fonction `glfwCreateWindow` permet d'ouvrir une fenêtre et de fixer ses paramètres. Elle à la signature suivante :

```
GLFWwindow* glfwCreateWindow (int width, int height,
            const char *      title,
            GLFWmonitor *     monitor,
            GLFWwindow *      share);
```

Les paramètres `width` et `height` permettent de fixer les dimensions de la fenêtre. Le second `title` est simplement le titre de la fenêtre. Les deux autres paramètres n'ont pas d'importance pour le moment (monitor sert à créer un fenêtre fullscreen et le dernier à partager un contexte avec d'autres fenêtres).

A faire :

01. À partir du fichier minimal.c et de la fonction ci-dessus, créez une fenêtre compatible OpenGL, de taille 600x600 et ayant pour titre : « TD 01 Ex 03 ».

Pour pouvoir dessiner une scène dans une fenêtre, il est nécessaire d'indiquer à OpenGL quelle zone de la scène sera visible via cette fenetre. Lorsqu'on voudra dessiner un point, il faudra fournir à OpenGL les coordonnées de ce point. Elles devront etre passées dans le repère virtuel de la scène. (qui peut s'exprimer en 2D ou en 3D). OpenGL ira ensuite convertir ces coordonnées dans un repère 2D lors de l'afficage à l'écran.

Cette étape de conversion entre l'espace virtuel de la scène et l'écran s'appelle la **projection**. Une fois que ces coordonées ont été obtenues, OpenGL peut travailler pour déterminer quels pixels devront etre coloriés (une étape connue sous le nom de **rasterization**).

Pour indiquer la taille de l'espace de que vous souhaitez représenter dans une fenetre, nous utiliserons une variable constante statique `GL_VIEW_SIZE` .

```
/* Espace fenetre virtuelle */
static const float GL_VIEW_SIZE = 1.;

// Nous representons ici les points situés en -0.5 et 0.5 en x et en y
```

Evidemment, lorsque la fenêtre est redimmensionnée, il faut modifier l'opération de projection vue précédemment. Nous allons également donc ajouter une fonction `onWindowResized()` pour garantir le bon fonctionnement de la projection tout au long de l'exécution.

A faire :

02a. Ajouter les lignes ci-dessus dans votre code (où à votre avis?). Puis ajoutez la fonction `onWindowResized()` également :

```
static float aspectRatio;

void onWindowResized(GLFWwindow* window,int width,int height)
{
    aspectRatio = width / (float) height;

    glViewport(0, 0, width, height);
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if( aspectRatio > 1)
    {
        gluOrtho2D(
        -GL_VIEW_SIZE / 2. * aspectRatio, GL_VIEW_SIZE / 2. * aspectRatio,
        -GL_VIEW_SIZE / 2., GL_VIEW_SIZE / 2.);
    }
    else
    {
        gluOrtho2D(
        -GL_VIEW_SIZE / 2., GL_VIEW_SIZE / 2.,
        -GL_VIEW_SIZE / 2. / aspectRatio, GL_VIEW_SIZE / 2. / aspectRatio);
    }
}
```

Le viewport définit la taille en pixels de la fenêtre réelle. Puis on passe en mode projection, et après avoir chargé la matrice identité, on recrée la projection orthogonale qui projette les coordonnées virtuelles en coordonnées pixels.

02b. Pour faire en sorte que cette fonction soit appelée à chaque redimmensionnement de fenêtre, nous allons demander à ce que cette fonction soit appelée dans ce cas à chaque fois grâce à l'instruction GLFW `glfwSetWindowSizeCallback` (voir documentation !) où le premier argument représente votre fenêtre, et le second le nom de votre fonction soit `onWindowResized`. Nous verrons dans la question suivante comment ce mécanisme opère dans GLFW. Ajoutez l'appel à la fonction `glfwSetWindowSizeCallback` juste avant la création de la fenêtre.

02c. Pour finir, pensez aussi à appeler `onWindowResized()` après la création de la fenêtre et avant la boucle while :

```
onWindowResized(window,WINDOW_WIDTH, WINDOW_HEIGHT);
```

Note – Le contexte d'une fenêtre (OpenGL)

Seulement pour les curieux.ses. On a parlé (très) brièvement de la notion de contexte (sur le dernier argument de la fonction de création de fenêtre). Lorsqu'une application de type IHM crée une fenêtre, celle-ci doit posséder un certain nombre d'état « graphique » qui sont soit demandés par l'utilisateur (je veux utiliser OpenGL 4.0, je veux de la transparence…) soit imposés par le serveur graphique de la machine. Autrement dit le contexte dépend à la fois de l'application demandée, mais également des possibilités de la machine.

Ainsi vous pourriez peut être demander telle ou telle version d'OpenGL mais si GLFW ne le peut pas, il ne pourra créer le contexte et donc ne pourra créer la fenêtre.

Pour les (encore plus) curieux, vous pouvez voir un certain nombre de ces états (appelés « hint » dans GLFW) à cet endroit. Comme vous le verrez cela va de la version d'OpenGL à si la fenêtre doit être redimmensionnable ou pas. Notez qu'il faut fixer ces « hints » **avant** la création de la fenêtre.

Exercice 04 – Des événements …

La gestion des évènements (utilisateurs ou pas) est un point très important des IHM. Les bibliothèques créant des fenêtres comme GLFW - mais également (en vrac) la SDL 1 et 2, SFML ou encore GLUT – ont globalement **deux types de stratégie** au regard de la gestion des évènements.

La première stratégie, plus flexible et probablement un peu plus efficace, est de gérer l'ensemble des évènements sous forme de liste à traiter à chaque instant. Lorsque l'application – ou plus exactement la bibliothèque – détecte un événement, elle le place dans une file interne avec des informations décrivant l'événement. Ce comportement est utilisé par exemple par les bibliothèque SDL ou SFML. Nous n'aborderons donc pas cette technique ici mais sachez qu'elle nécessite de traiter tous les événements reçus à chaque tour de la boucle d'affichage.

La seconde stratégie est une technique dite de *callback*, beaucoup plus simple à appréhender. Dans cette technique, le programmeur implémente, pour chaque type d'évènement auquel il souhaite réagir, une fonction dite de *callback*. Puis il demande à la bibliothèque de gestion des évènements (ici GLFW) d'appeler cette fonction. Cette fonction sera alors appelée à chaque fois qu'un évènement de ce type sera déclenché par l'utilisateur. Pour chaque type d'évènement, la fonction de callback doit avoir une signature spécifique. C'est dans cette signature que l'on retrouvera toutes les informations importantes liées à l'évènement.

Ca paraît flou ? Peut-être, mais vous l'avez déjà fait ! Dans l'exercice 3, nous avons défini une fonction `onWindowResized` avec une signature particulière (notamment deux paramètres qui définissent la nouvelle taille de la fenêtre) et nous avons demander à GLFW de l'appeler à chaque fois que la fenêtre est redimmensionnée via la fonction `glfwSetWindowSizeCallback`. Notez que nous gérons également les erreurs GLFW via une fonction de callback (saurez vous deviner laquelle?).

A faire :

01. Créez le nécessaire pour l'exercice 4 comme vu précédemment (dans la partie Notice)

02. Modifiez le programme pour qu'il s'arrête (utilisez `glfwSetWindowShouldClose`) lorsque l'utilisateur appuie sur la touche Q. Pour ce faire, trouver la fonction d'enregistrement de callback (et donc la signature de la fonction de callback) qui vous intéresse dans la documentation sur les évènements.

Note 1 : Préférez ici les *key input* au lieu des *text input*.

Note 2 : La signature propose 4 arguments concernant la touche pressée. Le premier de ceux-ci indique la touche pressée (la liste possible est ici mais attention **elle concerne un clavier US**), le second est un code unique pour la touche dépendant du clavier physique, le troisième indique l'action réalisée sur la touche, alors que le dernier vous indique si certaine touche sont pressées (en même temps que la touche qui a déclenché l'action) comme *Shift*, *Control* ou *Alt*.

Note 3 : Si vous souhaitez avoir directement la 'lettre' de la touche pressée au regard du clavier que vous utilisez, vous pouvez utiliser la fonction `glfwGetKeyName`.

Pour les plus chevroné.es, essayez maintenant de regarder la documentation de GLFW pour mettre votre application en plein écran. L'appui sur la touche Q est alors la seule façon 'propre' de quitter l'application.

03. Faites en sorte que lorsque l'utilisateur clique en position (x, y), la fenêtre soit redessinée avec la couleur (rouge = x mod 255, vert = y mod 255, bleu = 0) au prochain tour de la boucle d'affichage, plus précisément au prochain appel de `glClear(GL_COLOR_BUFFER_BIT)`. Notez que vous pouvez récupérer la position du curseur grâce à `glfwGetCursorPos`, définie à partir du coin supérieur gauche (qui a pour coordonnées 0,0 du coup).

Pour changer la couleur de fond, il faut utiliser la fonction `glClearColor(r, g, b, a)` qui définit (l'état de) la couleur de remplissage de la fenêtre. Ici `r`, `g`, `b` et `a` doivent être compris entre 0 et 1, il faut donc diviser les valeurs `r`, `g`, `b` entre 0 et 255 par 255.0, avant de les passer à OpenGL. Pour `a` il suffit de passer 1.

Attention : Assurez vous que votre division implique au moins un nombre flottant, sans quoi vous effectuerez une division euclidienne qui vous renverra 0, et non la valeur décimale voulue. Utilisez l'opérateur de cast `()` pour changer le type d'une expression. (ex : `(float) x`)

04. Faites de même mais lorsque la souris bouge (trouver la fonction de callback idoïne). Pour distinguer du précédent événement, faites en sorte que la fenêtre soit redessinée avec la couleur (rouge = x / largeur_fenetre, vert = 0, bleu = y / hauteur_fenetre). **Note :** Vous pouvez récupérer la largeur et hauteur de la fenêtre via la fonction `glfwGetWindowSize`.