I – Introduction

GLFW est une bibliothèque de gestion de fenêtre au sein d’OpenGL proposé à la base en alternative à GLUT jusqu’ici enseigné dans l’option Réalité Virtuelle mais qui aujourd’hui est considéré comme obsolète. Notre projet consistait donc en l’étude de cette bibliothèque et de proposé une solution pour remplacer les cours et TP de l’option RV sur GLUT par des solutions implantant la bibliothèque GLFW.

II – Présentation de GLFW

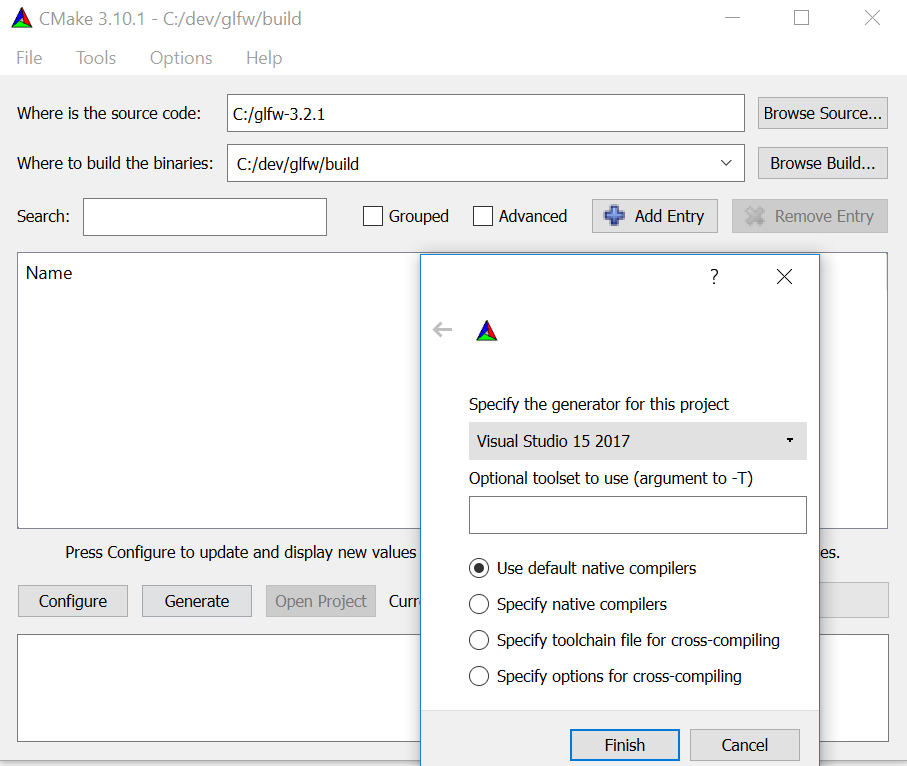
GLFW pour Graphics Library Framework est donc une bibliothèque C implantant la création et la gestion des fenêtres dans OpenGL. Elle ne fournit que le nécessaire : la création de la fenêtre, du contexte et la gestion des entrées utilisateur. Elle offre un maximum de contrôle sur la création du contexte OpenGL.

III – Installation de GLFW

Pour installer cette bibliothèque il faut dans un premier temps télécharger le code source de la bibliothèque, afin de générer la solution compatible avec le compilateur de notre choix (dans cet exemple on prendra VisualStudio 15 2017, voilà.).

Vous pouvez télécharger le code source via le lien suivant : <http://www.glfw.org/>

Une fois le code sourcé téléchargé, il nous faut le compiler pour le compilateur de notre choix (VS15) pour cela, nous utilisons Cmake (<https://cmake.org/download/>).

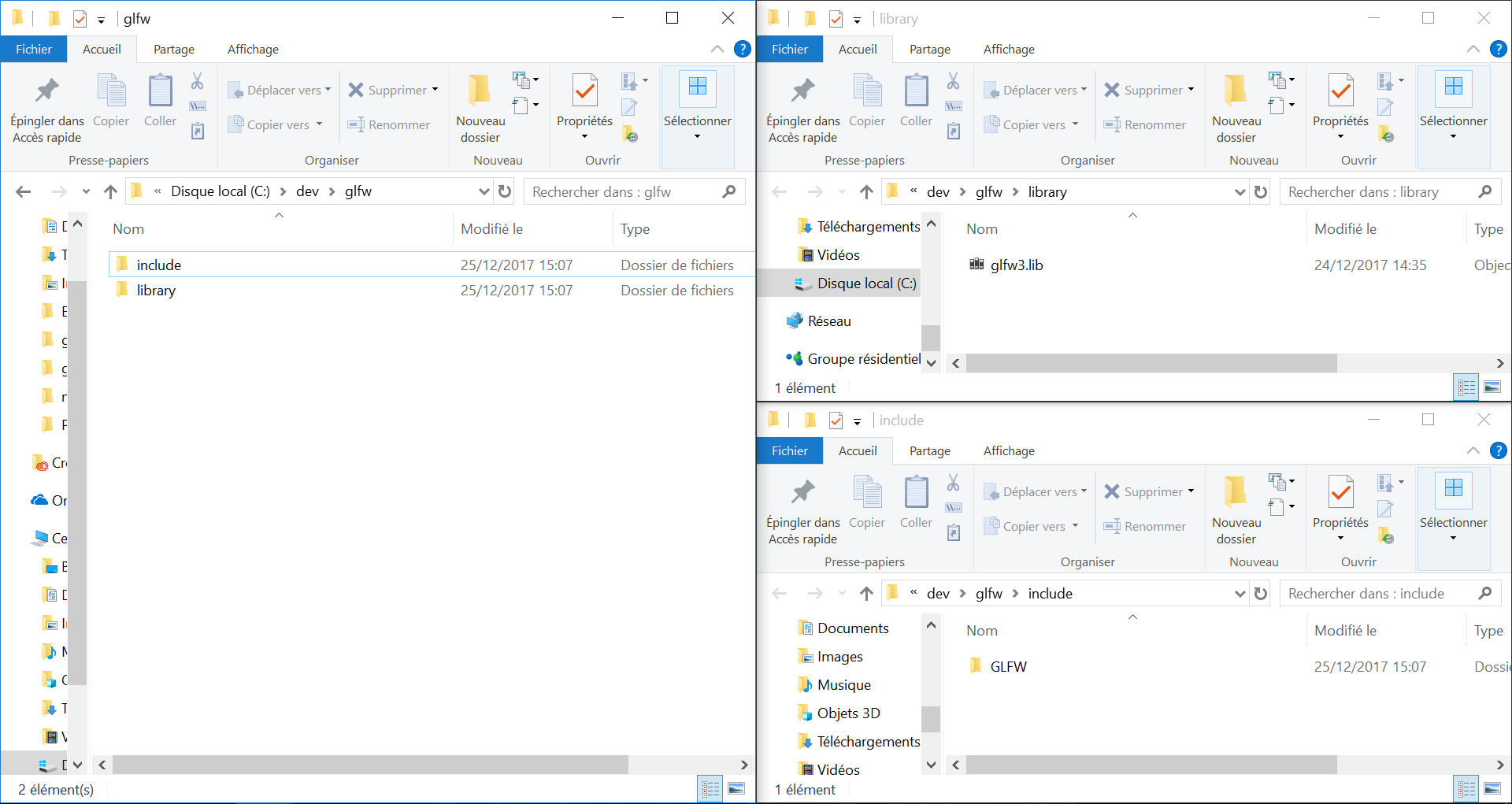


**Image 3.1 : compilation avec Cmake**

Comme sur le screen précédent, on renseigne l’emplacement du code source, ainsi que puis on renseigne l’endroit où l’on veut que la solution soit générée puis on clique sur générer et on choisit enfin le compilateur que l’on souhaite.

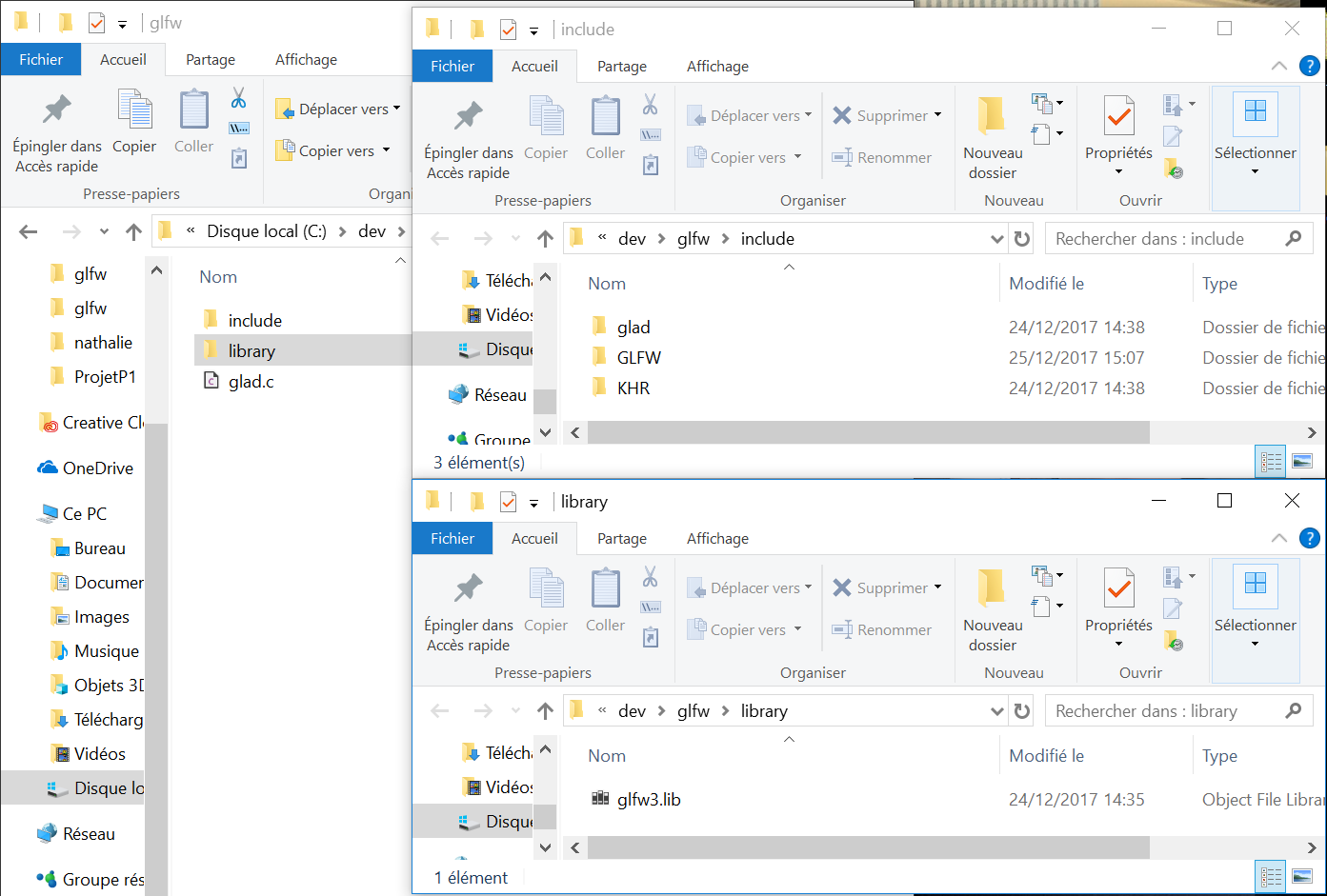
Cmake nous crée une solution compatible avec VS15 dans le dossier indiqué, nous faut maintenant générer la bibliothèque à partir de la solution crée. Il suffit pour cela de générer la solution ce qui vas créer la solution dans le dossier : …/DossierDeVotreBuild/src/debug

Maintenant pour être un minimum propre et organisé, nous allons créer un dossier dans lequel nous mettrons tous les fichiers importants pour créer un projet GLFW. Par exemple un fichier GLFW à dans le dossier c:/dev et nous allons y crée deux sous dossier : « include » et « library », pour justifier le TOEIC 850. Nous allons donc copier la bibliothèque précédemment crée dans le dossier c:/dev/GLFW/library. Ensuite, nous allons copier le dossier GLFW dans les include des fichier sources de GLFW (celui téléchargé au début de cette partie sur le site de GLFW.org) dans le dossier c:/dev/GLFW/include.



**Image 3.2 : Situation actuelle**

Afin de se faciliter la vie et ne pas avoir à taper des commande superflue pout gérer les différentes versions d’OpenGL et de carte graphique, nous allons installer GLAD, qui est une bibliothèque open source qui se chargera de la compatibilité pour nous. Pour cela il nous faut télécharger GLAD sur le site <http://glad.dav1d.de/> en précisant la version d’OpenGL supporté par notre carte graphique (recherche sur internet des caractéristiques ladite carte). Le langage étant C/C++ et le profile « core ». Téléchargeons le contenue du dossier « include » et copions le dans le dossier c:/dev/GLFW/include. Attention, le contenue du dossier « src » n’est pas une bibliothèque, mais un fichier source que l’on devra inclure dans notre projet GLFW. Il nous faut donc le télécharger et le mettre par exemple dans le dossier c:/dev/GLFW.



**Image 3.3 : Dossier final**

IV – Création d’un projet GLFW

Maintenant que l’on a installé GLFW, il est temps d’entrer dans le vif du sujet et de créer notre premier projet. Nous continuerons de travailler avec Visual Studio 15 2017 car ce compilateur est déjà installé sur les ordinateurs de la salle B114. Il nous faut donc créer un projet VS15 et le paramétrer pour utiliser GLFW et GLAD pour cela il faut aller ajouter nos répertoires d’include et de bibliothèques de GLFW dans les paramètres du projet :

* Projet 🡪 paramètres de NomDuProjet 🡪 Répertoires VC++
* Il faut ajouter le chemin de nos includes (c:/dev/GLFW/include) dans le répertoire d’include
* Il faut ajouter le chemin de notre bibliothèque (c:/dev/GLFW/library) dans le répertoire de bibliothèque

Il faut aussi ajouter quelques lignes à l’éditeur de liens du projet :

* Projet 🡪 paramètres de NomDuProjet 🡪 Editeur de Liens 🡪 entrée 🡪dépendances supplémentaires
* Il faut ensuite ajouter ces deux lignes ;
  + Opengl32.lib
  + Glfw3.lib

Pour finir, il nous reste à ajouter le fichier glad.c dans les fichiers sources du projet (un simple glissé déposé fera l’affaire). Le projet est enfin prêt pour utiliser la bibliothèque GLFW, pour voir si c’est bien le cas, on peut ajouter les include suivant au projet et vérifier si ce dernier compile bien (si ces lignes se sous-lignent en rouge, c’est qu’il y a un problème)

#include <glad/glad.h>

#include <GLFW/glfw3.h>

*\*Attention, il faut impérativement inclure glad.h avant glfw.h car glad.h contient les includes nécessaires pour le bon fonctionnement de GLFW, s’il est include après, il y aura forcément des erreurs de compilation.*

V – Première code : afficher une fenêtre

La création et l’écriture dans une fenêtre OpenGL se passe généralement de la façon suivante :

#include <entetes\_bibliothèque>

int main()

{

creerFenetre(titre, largeur, hauteur);

creerContextOpenGL(parametres);

while (fenetreOuverte)

{

while (evenement = nouvelEvenement())

gererEvenement(evenement);

misAJourScene();

dessineGraphismes();

afficheGraphismes();

}

return 0;

}

Passons maintenant au vif du sujet, et commençons à coder avec GLFW et donc à traduire l’algorithme précédant dans le langage qui nous concerne. Notre premier code consistera en une simple fenêtre dont on changera pour le moment seulement la couleur du fond. Pour commencer, il nous faut initialiser GLFW. Pour cela on utilise la fonction glfwInit.

int main()

{

// glfw: initialisation

glfwInit();

(…) //notre code

// glfw: iterruption

glfwTerminate();

}

Ensuite, il nous faut le configurer GLFW, pour cela, on utilise la fonction glfwWindowHint pour connaitre les différentes configurations modifiable, nous vous incitons à visiter la documentation suivante :

* La fonction glfwWindowHint :

<http://www.glfw.org/docs/latest/group__window.html#ga7d9c8c62384b1e2821c4dc48952d2033>

* Les différents « Hint »

<http://www.glfw.org/docs/latest/window_guide.html#window_hints_ctx>

Pour notre cas, nous allons nous concentrer sur la version 3.3 d’OpengGL, il nous faut donc configurer GLFW afin de créer le bon contexte OpenGL par la suite.

// configuration à la version 3.X

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

// configuration à la version x.3

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

// Ces deux lignes ont donc configure GLFW

// pour la version 3.3 d’OpenGL

Nous souhaitons un contexte qui ne supporte que les fonctionnalités du profil principal et pas l’ancien OpenGL.

//Telling GLFW explicitly that we want to use the core-profile means //we'll get access to a smaller subset of OpenGL features

glfwWindowHint(GLFW\_OPENGL\_PROFILE,GLFW\_OPENGL\_CORE\_PROFILE);

*Notons que les personnes qui travail sur Mac OS doivent ajouter une configuration de GLFW :*

// pour les utilisateurs Mac

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

Maintenant, il nous faut créer la fenêtre, pour cela, on utilisera la fonction glfwCreateWindow don voici la documentation :

<http://www.glfw.org/docs/latest/group__window.html#ga5c336fddf2cbb5b92f65f10fb6043344>

*Astuce de débogage : il est possible de créer une condition pour renvoyer un message d’erreur dans la commande si la fenêtre n’est pas correctement crée.*

Ensuite nous devons créer le contexte OpenGL avec la fonction glfwMakeContextCurrent :

<http://www.glfw.org/docs/latest/group__context.html#ga1c04dc242268f827290fe40aa1c91157>

// création de la fenêtre

GLFWwindow\* window = glfwCreateWindow(800, 600, "Première fenêtre", NULL, NULL);

// boucle de débogage

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return -1;

}

// activation du contexte

glfwMakeContextCurrent(window);

Ensuite arrive la boucle événementielle, GLFW utilise une boucle événementielle appelée « fermée ». Cela signifie que vous n'avez qu'à gérer les événements que lorsque vous en avez besoin. En clair, votre boucle événementielle semblera très simple. Les seules fonctions nécessaires dans la boucle sont glfwSwapBuffers() pour échanger le tampon arrière avec le tampon avant après avoir fini le rendu et la fonction glfwPollEvents() pour mettre à jour les événements.

while (!glfwWindowShouldClose(window))

{

glfwSwapBuffers(window);

glfwPollEvents();

}

On peut également rajouter une gestion des entrées clavier afin de pouvoir fermer la fenêtre en appuyant sur la touche Echap.

while (!glfwWindowShouldClose(window))

{

(…)

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS){

glfwSetWindowShouldClose(window, true);

}

(…)

}

*Notons qu’il est conseillé de faire une fonction regroupant toutes les entrées clavier et de la mettre ensuite dans notre fonction d’affichage*

void processInput(GLFWwindow \*window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

}

A ce niveau, il est possible d’afficher une fenêtre. Cependant, elle n’affiche rien, est complètement noir et n’est pas redimensionnable, bref, elle est moche.

Avant de faire appelle aux fonctions OpenGL pour les fonctions de dessin, il ne faut pas oublier de demander à GLAD d’initialiser OpenGl pour notre OS et notre carte graphique.

// glad: load all OpenGL function pointers

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);

*Il faut ajouter cette ligne avant toute instruction OpenGL donc avant notre boucle événementielle.*

Avant de pouvoir dessiner, Nous devons dire à OpenGL la taille de la fenêtre de rendu afin qu'OpenGL sache comment nous voulons afficher les données, pour cela on utilise la fonction glViewport (<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glViewport.xhtml>)

glViewport(0, 0, 800, 600);

Pour gérer le redimensionnement on vas utiliser la fonction glfwSetFramebufferSizeCallback (<http://www.glfw.org/docs/latest/group__window.html#ga3203461a5303bf289f2e05f854b2f7cf>)

Afin de rendre le code plus propre, il est possible de faire une fonction dans laquelle on modifie directement le viewport comme suit :

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height)

{

glViewport(0, 0, width, height);

}

int main()

{

(…)

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

(…)

// boucle événementielle

(…)

}

Maintenant, pour afficher des couleurs, il nous faut activer le color buffer via la fonction glClear (<https://www.khronos.org/registry/OpenGL-Refpages/es3.1/html/glClear.xhtml>)

Pour l’instant, nous n’avons besoin de rafraichir que le color buffer, mais cette fonction permet aussi d’activer le Z-buffer et le Stencil buffer (tampon mémoire permettant de filtrer les endroits où l'image doit être dessinée) utile notamment pour les scènes 3D.

while (!glfwWindowShouldClose(window))

{

(…)

glClearColor(0.3f, 0.2f, 0.3f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

(…)

}

VI – Premier Triangle (2D)

OpenGL 3 facilite l’écriture des choses compliquées, mais possède l’inconvénient de rendre l’affichage d’un simple triangle relativement difficile. En effet, il n’est plus possible de passer par de la 2D simple (ou en tout cas je n’ai pas trouvé) du coup il nous faut passer par la 3D et les shaders même pour afficher de la 2D simple.

Pour afficher notre triangle, nous allons utiliser 3 types de Shaders :

* Un Vertex Array Object (VAO)
* Un Vertex Shader (exécuté pour chaque sommet)
* Un Fragment Shader (exécuté pour chaque fragment)

Nous n’allons pas rentrer dans les détails des Shaders, pour initialiser le VAO, il suffit d’ajouter ces quelques lignes après la création du contexte OpenGL :

GLuint VertexArrayID;

glGenVertexArrays(1, &VertexArrayID);

glBindVertexArray(VertexArrayID);

Nous

<https://open.gl>

<https://learnopengl.com>

<https://fr.wikipedia.org>

<http://www.glfw.org/>

<https://www.opengl.org/documentation/>

<https://www.developpez.com/>

<http://www.opengl-tutorial.org>