

Apprendre OpenGL moderne

Par Joey de Vries - Jean-Michel Fray (traducteur)

Date de publication : 3 mai 2018

TOUT PUBLIC

Developpez.com a la joie d'accueillir une traduction en français du célèbre cours de grande qualité **Learn OpenGL**. Au cours de celui-ci, vous apprendrez à programmer des applications graphiques 3D grâce à la bibliothèque OpenGL.

Ces tutoriels sont accessibles aux débutants, mais les connaisseurs ne s'ennuieront pas non plus grâce aux chapitres avancés.

Cette page vous amène à la première partie du tutoriel, c'est-à-dire : l'introduction, la mise en place des outils et la mise en place de sa première application OpenGL.

Dans quelque temps, les parties suivantes seront mises en ligne :

- **éclairage** ;
- **chargement de modèle** ;
- **OpenGL avancé** ;
- éclairage avancé ;
- PBR ;
- **mise en pratique**.

Commentez

I - Introduction.....	5
I-A - Prérequis.....	5
I-B - Structure.....	5
I-C - Notes du traducteur.....	5
II - Bienvenue dans OpenGL.....	5
II-A - Pourquoi lire ces tutoriels ?.....	6
II-B - Qu'apprendrez-vous ?.....	6
II-C - Remerciements.....	6
III - OpenGL.....	7
III-A - Core-profile et Immediate mode.....	7
III-B - Les extensions.....	8
III-C - Machine à états.....	8
III-D - Les objets.....	9
III-E - On peut commencer.....	10
III-F - Ressources supplémentaires.....	10
III-G - Remerciements.....	10
IV - Créer une fenêtre.....	10
IV-A - GLFW.....	10
IV-B - Compiler GLFW.....	10
IV-B-1 - CMake.....	11
IV-B-2 - Compilation.....	11
IV-C - Notre premier projet.....	12
IV-D - Édition des liens (linking).....	12
IV-D-1 - Bibliothèque OpenGL sur Windows.....	13
IV-D-2 - Bibliothèque OpenGL sur Linux.....	13
IV-E - GLAD.....	14
IV-E-1 - Installer GLAD.....	14
IV-F - Suppléments.....	14
IV-G - Remerciements.....	15
V - Hello window.....	15
V-A - GLAD.....	16
V-B - Viewport.....	16
V-C - Prêts pour le départ.....	17
V-D - Une dernière chose.....	17
V-E - Les entrées.....	18
V-F - Le rendu.....	19
V-G - Remerciements.....	20
VI - Hello Triangle.....	20
VI-A - Sommets.....	22
VI-B - Le vertex shader.....	24
VI-C - Compiler un shader.....	25
VI-D - Le fragment shader.....	25
VI-E - Le program shader.....	26
VI-F - Lier les attributs de sommets.....	27
VI-G - Le Vertex Array Object.....	28
VI-H - Le triangle que nous attendions.....	29
VI-I - Les Element Buffer Objects.....	30
VI-J - Ressources supplémentaires.....	33
VI-K - Exercices.....	33
VI-L - Remerciements.....	34
VII - Les shaders.....	34
VII-A - GLSL.....	34
VII-A-1 - Les types.....	35
VII-A-2 - Les vecteurs.....	35
VII-A-3 - Entrées et sorties.....	35
VII-A-4 - Variables uniformes.....	37
VII-B - Plus d'attributs.....	39
VII-C - Notre propre classe shader.....	41

VII-C-1 - Lire dans les fichiers.....	42
VII-C-2 - Compilation.....	42
VII-C-3 - Fonctions.....	43
VII-C-4 - Utilisation.....	43
VII-D - Exercices.....	43
VII-E - Remerciements.....	44
VIII - Textures.....	44
VIII-A - Texture Wrapping.....	46
VIII-B - Interpolation de texture (texture filtering).....	46
VIII-B-1 - Mipmaps.....	48
VIII-C - Charger et créer les textures.....	49
VIII-C-1 - stb_image.h.....	49
VIII-C-2 - Générer une texture.....	49
VIII-D - Appliquer les textures.....	50
VIII-E - Unités de texture.....	53
VIII-F - Exercices.....	55
VIII-G - Remerciements.....	56
IX - Transformations.....	56
IX-A - Les vecteurs.....	56
IX-B - Opérations entre vecteur et scalaire.....	57
IX-C - Négation.....	58
IX-D - Addition et soustraction entre vecteurs.....	58
IX-E - Longueur.....	59
IX-F - Produit de vecteurs.....	60
IX-F-1 - Produit scalaire.....	60
IX-F-2 - Produit vectoriel.....	61
IX-G - Matrices.....	62
IX-G-1 - Addition et soustraction.....	62
IX-G-2 - Produit par un scalaire.....	63
IX-G-3 - Multiplication de deux matrices.....	63
IX-H - Multiplication d'un vecteur par une matrice.....	64
IX-I - La matrice identité.....	64
IX-J - Matrice de mise à l'échelle d'un vecteur.....	65
IX-K - Matrice de translation d'un vecteur position.....	66
IX-L - Matrice de rotation d'un vecteur.....	66
IX-M - Combiner les matrices.....	68
IX-N - En pratique.....	69
IX-O - GLM.....	69
IX-P - Lectures.....	72
IX-Q - Exercices.....	72
IX-R - Remerciements.....	72
X - Systèmes de Coordonnées.....	72
X-A - Le schéma global.....	72
X-B - L'espace local.....	73
X-C - Espace monde.....	73
X-D - Espace de vue.....	73
X-E - L'espace de clipping.....	74
X-E-1 - Projection orthogonale.....	74
X-E-2 - Projection en perspective.....	75
X-F - Synthèse.....	78
X-G - Passons en 3D.....	78
X-H - Plus de 3D.....	80
X-H-1 - Z-buffer.....	81
X-H-2 - Plus de cubes !.....	81
X-I - Exercices.....	82
X-J - Remerciements.....	82
XI - Caméra.....	82
XI-A - L'espace Caméra/Vue.....	83

XI-A-1 - Position de la caméra.....	83
XI-A-2 - Direction de la caméra.....	83
XI-A-3 - Axe de droite.....	83
XI-A-4 - Axe vers le haut.....	84
XI-B - Look At.....	84
XI-C - Se balader dans la scène.....	85
XI-D - Vitesse du mouvement.....	85
XI-E - Naviguer dans la scène.....	86
XI-E-1 - Les angles d'Euler.....	86
XI-F - Entrées avec la souris.....	88
XI-G - Zoom.....	91
XI-H - Une classe Camera.....	91
XI-I - Exercices.....	92
XI-J - Remerciements.....	92
XII - Glossaire.....	92
XII-A - Remerciements.....	93

I - Introduction

Puisque vous êtes ici, vous souhaitez sans doute apprendre le fonctionnement interne de la programmation graphique et faire des applications 3D comme les pros. Faire les choses soi-même est très sympa, formateur et vous permettra une bonne compréhension de ce sujet. Cependant, certains points doivent être pris en considération avant de débiter.

I-A - Prérequis

Puisqu'OpenGL est une API graphique et non une plateforme en soi, cela demande d'utiliser un langage de programmation, qui se trouve être le C++. Ainsi une connaissance correcte du C++ est nécessaire pour comprendre ces tutoriels. La plupart des concepts utilisés seront expliqués, il n'est pas indispensable d'être un expert en C++, mais au moins de savoir écrire un programme « Hello World ». Pour compléter vos connaissances, vous pouvez consulter des tutoriels gratuits sur [Developpez.com](http://developpez.com).

Nous utiliserons aussi quelques notions mathématiques (algèbre linéaire, géométrie et trigonométrie) que j'essaierai d'expliquer au moment voulu. N'étant pas mathématicien, mes explications seront assez simplistes et sans doute incomplètes. Je vous donnerai alors des liens vers de bonnes ressources pour compléter vos connaissances. Cependant, pas d'inquiétude, la plupart des notions mathématiques utilisées peuvent être comprises avec quelques notions de base et j'essaierai de limiter la difficulté au minimum. Il est surtout nécessaire de savoir utiliser ces notions.

I-B - Structure

LearnOpenGL est divisé en un certain nombre de parties générales. Chaque partie contient plusieurs chapitres qui expliquent les différents concepts en détail. Chaque chapitre est présenté de façon linéaire (mieux vaut donc les lire de haut en bas), où chaque page détaille la théorie et les aspects pratiques.

I-C - Notes du traducteur

Ce site est une traduction du site LearnOpenGL de **Joey de Vries**. Je ne suis pas un professionnel de la traduction, juste un utilisateur qui souhaite mettre le contenu du site d'origine à disposition des francophones pour leur faciliter la lecture de cet excellent tutoriel. La traduction peut sans doute être meilleure et j'invite les lecteurs qui auraient des améliorations à apporter à me contacter. Certains termes techniques sont laissés en anglais car utilisés tels quels par les développeurs. Cette traduction ne saurait remplacer le site d'origine et je pense que la bonne façon de travailler est de s'y référer en parallèle, ne serait-ce que pour se familiariser avec les termes anglais. J'ai ajouté dans le chapitre « Résumé » les termes techniques anglais utilisés dans ces tutoriels.

Je ne saurais trop vous conseiller de regarder aussi les discussions présentes sur le site LearnOpenGL.com, à la fin de chaque chapitre, mais qui restent en anglais. On y trouve plein de choses intéressantes, des questions (et réponses) que l'on se pose éventuellement en cours de lecture ou de test des concepts exposés. Je vous invite à utiliser le **forum OpenGL** de Developpez.com si vous rencontrez des soucis et pour avoir une aide en français, ou, si vous avez un commentaire spécifique au tutoriel, d'utiliser [cette discussion](#).

II - Bienvenue dans OpenGL

Bienvenue dans mon humble contribution pour une plateforme d'enseignement bien conçue d'une API graphique appelée OpenGL. Que vous souhaitiez apprendre OpenGL pour des besoins académiques, pour votre travail, ou par intérêt personnel, ce site vous donnera les connaissances de base, intermédiaires et avancées en utilisant la version moderne d'OpenGL (modern (core-profile) OpenGL). Le but de ce site est de vous montrer ce que permet OpenGL d'une façon pédagogique avec des exemples clairs, mais aussi de mettre à disposition une référence utile pour la suite.

II-A - Pourquoi lire ces tutoriels ?

Sur internet, on peut trouver moult documents et ressources pour apprendre OpenGL, cependant la plupart de ces documents sont écrits pour le mode immédiat, souvent décrit comme l'ancien OpenGL, ou bien sont incomplets, ou manquent de documentation précise et ne sont pas adaptés à votre façon d'apprendre. Ainsi, mon but est de produire une plateforme à la fois complète et facile à comprendre.

Si vous appréciez la lecture de ces tutoriels qui fournissent des instructions pas-à-pas, des exemples clairs et qui ne se perdent pas dans une foule de détails, ce site est fait pour vous. Ces tutoriels sont conçus pour des lecteurs n'ayant pas d'expérience en programmation graphique, mais sont aussi intéressants pour des utilisateurs expérimentés. On présentera des concepts pratiques, qui, avec l'aide de votre créativité, vous permettront de développer de réelles applications 3D. Si tout cela vous parle, alors je vous invite à continuer votre lecture.



II-B - Qu'apprendrez-vous ?

L'objet de ces tutoriels est l'OpenGL moderne. Apprendre (et utiliser) OpenGL requiert une solide connaissance de la programmation graphique et il est nécessaire de comprendre comment OpenGL fonctionne pour en acquérir la meilleure expérience possible. Ainsi, nous discuterons des aspects du noyau graphique, de savoir comment OpenGL affiche les pixels sur votre écran et comment mettre ces connaissances à profit pour créer des effets visuels intéressants.

En plus de cette connaissance du noyau d'OpenGL, nous présenterons plusieurs techniques utiles que vous pourrez utiliser dans vos propres applications : se promener dans une scène, créer de beaux éclairages, charger des objets complexes conçus avec un programme de modélisation, réaliser des opérations de post-processing et bien plus encore. Nous présenterons aussi un tutoriel progressif pour développer un petit jeu basé sur nos connaissances d'OpenGL, et ainsi vous pourrez appréhender ce qu'est le développement d'une application graphique.

Si vous souhaitez rester à jour quant aux modifications du site ou encore à propos de news concernant OpenGL, suivez-moi sur [Twitter](#) ou [Facebook](#).

II-C - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

III - OpenGL

Avant de commencer, il nous faut présenter ce qu'est OpenGL. OpenGL est considéré comme une API (*Application Program Interface*) qui offre un vaste ensemble de fonctions permettant de manipuler des objets graphiques et des images. Cependant, OpenGL en lui-même n'est pas une API, mais simplement une spécification, développée et maintenue par le **Khronos Group**.



La spécification OpenGL décrit exactement ce que sont les entrées et sorties de chaque fonction et comment elles doivent s'exécuter. C'est ensuite aux développeurs d'implémenter ces fonctions pour produire une solution. Puisque la spécification d'OpenGL ne donne pas les détails d'une implémentation, les implémentations peuvent présenter des différences tant que les résultats correspondent à la spécification (et apparaissent identiques pour l'utilisateur).

Les développeurs des bibliothèques OpenGL sont généralement les constructeurs de cartes graphiques. Chaque carte graphique présente ainsi sa propre version d'OpenGL, développée spécialement pour ce matériel. Lorsque l'on utilise OpenGL sur un système Apple, la bibliothèque OpenGL est maintenue par Apple lui-même ; sous Linux, il existe un mélange de versions fournies par les constructeurs, mais aussi des adaptations par des développeurs particuliers. Si l'on constate un comportement curieux d'OpenGL, cela est probablement la faute du fournisseur de la carte graphique (ou de ceux chargés de son développement ou de sa maintenance).

i Lorsque l'on constate un bogue, cela se résout très généralement en mettant à jour les pilotes (drivers) de la carte graphique ; ces pilotes incluent les dernières versions d'OpenGL pour votre carte. Voilà pourquoi il est conseillé de mettre à jour régulièrement ses pilotes.

Khronos propose publiquement tous les documents de spécification pour toutes les versions d'OpenGL. Le lecteur intéressé peut trouver la spécification de la version 3.3 (celle que nous utiliserons) **ici** : une bonne lecture pour se plonger dans les détails d'OpenGL (notez que cela ne donne pas l'implémentation). Les spécifications donnent aussi la définition exacte de ce que doit réaliser chaque fonction.

III-A - Core-profile et Immediate mode

Avant, utiliser OpenGL impliquait de développer en mode immédiat (aussi appelé le *fixed function pipeline*). C'était une méthode assez facile pour créer des graphiques. La plupart des fonctionnalités étaient cachées dans les bibliothèques et les développeurs n'avaient pas beaucoup de liberté sur la façon d'opérer d'OpenGL. Les développeurs ont souhaité plus de flexibilité et ainsi les spécifications se sont assouplies, donnant aux développeurs plus de contrôle. Le mode immédiat est facile à comprendre et à utiliser, mais reste peu efficace. Pour cette raison, la spécification a commencé à déprécier le mode immédiat à partir de la version 3.2 et on a encouragé les développeurs à travailler avec le *core-profile* qui constitue une rupture dans la spécification d'OpenGL et qui supprime toutes les fonctionnalités à bannir.

Dans la version *core-profile* d'OpenGL, on est obligé d'utiliser des méthodes modernes. Si l'on essaie d'utiliser une fonction obsolète, OpenGL lève une erreur et s'arrête. L'avantage d'apprendre la méthode moderne est d'avoir un résultat plus flexible et efficace, mais cela est aussi plus difficile. Le mode immédiat cache bon nombre d'opérations, alors qu'avant, bien que plus facile à appréhender, il était difficile de savoir comment travaillait OpenGL. L'approche moderne demande au développeur de vraiment comprendre OpenGL et la programmation graphique. Bien que

cela soit un peu difficile, cela autorise plus de flexibilité, d'efficacité, et surtout une meilleure compréhension de la programmation graphique.

Pour ces raisons, nos tutoriels sont tournés vers la version 3.3 d'OpenGL Core-Profile. Bien que plus compliqué, cela vaut vraiment l'effort.

Aujourd'hui, des versions plus élaborées d'OpenGL sont disponibles (4.5 à la date de ce document) et vous pouvez donc vous demander : pourquoi apprendre OpenGL 3.3 alors qu'on en est à 4.5 ? La réponse est assez simple. Toutes les versions futures d'OpenGL à partir de 3.3 ajouteront des fonctionnalités sans changer le mécanisme du noyau. Les versions récentes seront simplement légèrement plus efficaces ou fourniront des moyens plus faciles pour certaines tâches, mais tous les concepts et les techniques resteront les mêmes dans les versions d'OpenGL moderne ; il est ainsi légitime d'apprendre avec la version 3.3. Avec plus d'expérience, vous pourrez profiter sans difficultés des dernières améliorations d'OpenGL.



Si vous utilisez les fonctionnalités des dernières versions d'OpenGL, seules les cartes graphiques les plus récentes pourront faire fonctionner votre application. C'est pourquoi les développeurs utilisent des versions moins récentes et ne présentent les fonctionnalités plus récentes qu'en option.

Dans certains chapitres, vous pourrez trouver des fonctionnalités plus récentes, elles seront alors indiquées comme telles.

III-B - Les extensions

Une caractéristique importante d'OpenGL est sa capacité à supporter des extensions. Quand un fabricant de cartes propose une nouvelle technique ou une optimisation de rendu, cela se trouve souvent dans une extension, implémentée dans le pilote (driver). Si le matériel supporte cette amélioration, le développeur peut utiliser cette fonctionnalité pour un meilleur résultat, sans attendre qu'OpenGL l'inclue dans une version ultérieure. Il suffit de vérifier que cette extension est supportée par la carte graphique. Lorsqu'une extension devient courante ou se révèle utile, OpenGL l'intègre dans sa nouvelle version.

Le développeur devra donc interroger OpenGL pour savoir si l'extension est disponible (ou encore utiliser une bibliothèque d'extension d'OpenGL) :

```
if(GL_ARB_extension_name)
{
    // Utiliser les trucs récents supportés par le matériel
}
else
{
    // Extension non supportée : on utilise la vieille méthode
}
```

Avec OpenGL 3.3, nous aurons rarement besoin d'une extension, mais lorsque ce sera nécessaire, le moyen de l'utiliser sera présenté.

III-C - Machine à états

OpenGL est une énorme machine à états : un ensemble de variables qui définissent comment OpenGL doit fonctionner, nommé contexte OpenGL. On modifie souvent cet état au moyen d'options, en manipulant des tampons (buffers), pour finalement effectuer un rendu dans le contexte en cours.

Par exemple, si l'on souhaite afficher des lignes plutôt que des triangles, on changera l'état d'OpenGL en modifiant une variable de contexte qui impose à OpenGL sa façon de dessiner.

Nous verrons plusieurs fonctions qui changent le contexte OpenGL et d'autres fonctions qui réalisent des opérations en fonction de ce contexte. Si vous gardez à l'esprit qu'OpenGL est une machine à état, la plupart des fonctionnalités vous apparaîtront plus claires.

III-D - Les objets

Les bibliothèques OpenGL sont écrites en C et permettent d'utiliser d'autres langages, mais cela reste un noyau en C. Comme beaucoup de fonctionnalités du C ne se traduisent pas si bien dans d'autres langages, OpenGL a été développé en tenant compte de plusieurs abstractions. L'une d'entre elles est la notion d'objet.

Un objet dans OpenGL est un ensemble d'options qui représente un sous-ensemble de l'état d'OpenGL. Par exemple, on trouvera un objet qui représente les paramètres d'affichage d'une fenêtre. On peut ainsi définir sa taille, le nombre de couleurs, etc. Un objet peut être visualisé comme une structure en C :

```
struct object_name {  
    float  option1;  
    int    option2;  
    char[] name;  
};
```

Pour utiliser un objet, cela ressemblera à ceci (le contexte d'OpenGL étant vu comme une grande structure) :

```
// Etat d'OpenGL  
struct OpenGL_Context {  
    ...  
    object* object_Window_Target;  
    ...  
};  
  
// créer l'objet  
unsigned int objectId = 0;  
glGenObject(1, &objectId);  
  
// relier cet objet à GL_WINDOW_TARGET  
glBindObject(GL_WINDOW_TARGET, objectId);  
  
// définir les options voulues pour cet objet  
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH, 800);  
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);  
  
// remettre les options par défaut  
glBindObject(GL_WINDOW_TARGET, 0);
```



Ce petit bout de code est un schéma que l'on trouvera souvent en travaillant avec OpenGL. On crée d'abord un objet et l'on mémorise une référence à cet objet dans un entier (les données réelles de l'objet sont cachées). On relie ensuite cet objet à l'endroit cible du contexte (ici `GL_WINDOW_TARGET`). Ensuite, on positionne les options voulues et finalement on détache cet objet en spécifiant 0 comme id de l'objet. Les options sont mémorisées dans l'objet référencé par `objectId` et sont réappliquées dès lors que l'on relie à nouveau cet objet à `GL_WINDOW_TARGET`.

Ce qui est important avec l'utilisation de ces objets est la possibilité de définir plusieurs objets dans notre application, de définir les options qu'il contiendra, et lorsque l'on fera une opération utilisant l'état d'OpenGL, il suffira de relier cet objet à la cible correspondante pour disposer des options contenues dans cet objet. Il y a des objets qui sont par exemple des conteneurs pour les données d'un modèle 3D (une maison ou un simple caractère), et lorsque l'on voudra l'afficher, il suffira de relier cet objet. Disposer de plusieurs objets nous permettra de référencer plusieurs modèles et pour les utiliser nous relierons l'objet souhaité avant d'effectuer le rendu, sans avoir besoin de préciser à nouveau toutes les options.

III-E - On peut commencer

Maintenant, vous savez ce qu'est OpenGL, comment cela fonctionne à peu près et connaissez quelques trucs pour l'utiliser. Pas de souci si vous n'avez pas tout compris, nous avancerons pas à pas dans ces tutoriels et vous disposerez d'exemples pour vous familiariser avec OpenGL. Nous allons créer une première fenêtre dans le **chapitre suivant**.

III-F - Ressources supplémentaires

-  **opengl.org** : site officiel d'OpenGL.
-  **OpenGL registry** : spécifications et extensions pour toutes les versions d'OpenGL.

III-G - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

IV - Créer une fenêtre

La première chose à faire pour créer une application avec de superbes graphismes est de créer un contexte OpenGL et la fenêtre de l'application pour l'affichage. Cependant, ces opérations dépendent du système sur lequel vous travaillez et par conséquent, OpenGL rend ces opérations abstraites. Cela signifie que nous devons créer une fenêtre, définir un contexte et gérer les entrées utilisateur nous-mêmes.

Heureusement, il existe quelques bibliothèques qui permettent de faire cela, certaines étant spécifiques à OpenGL. Ces bibliothèques vont nous économiser ce travail qui dépend du système d'exploitation utilisé. Les plus connues de ces bibliothèques sont GLUT, SDL, SFML et GLFW. Pour nos tutoriels, nous utiliserons **GLFW**.

IV-A - GLFW



GLFW est une bibliothèque, écrite en C, spécifiquement destinée à OpenGL, fournissant les outils de base requis pour afficher des choses sur l'écran. Cela va nous permettre de créer un contexte OpenGL, définir les paramètres de la fenêtre et gérer les entrées utilisateur.

Le sujet de ce tutoriel et du suivant est de configurer GLFW, de le faire fonctionner, s'assurant qu'il crée un contexte OpenGL correct et qu'il affiche une fenêtre pour les rendus. Ce chapitre est une approche pas à pas pour obtenir, construire et lier la bibliothèque GLFW. Dans ce tutoriel, nous utilisons l'IDE Microsoft Visual Studio 2015 (la démarche est la même pour les versions plus récentes de l'IDE). Si vous n'utilisez pas Visual Studio (ou une version plus ancienne), pas de souci, la démarche est semblable sur la plupart des autres outils.

IV-B - Compiler GLFW

Vous pouvez télécharger GLFW sur le **site officiel**. GLFW propose déjà des binaires précompilés et des fichiers d'en-tête pour Visual Studio 2013/2015, mais par souci de complétude, nous compilerons nous-mêmes le code source. Téléchargez le « **Source package** ».



Si vous utilisez les binaires précompilés, assurez-vous de télécharger les versions 32 bits (à moins de savoir exactement ce que vous faites). Les versions 64 bits ont causé pas mal d'erreurs curieuses à la plupart des lecteurs.

Une fois téléchargé, extrayez les fichiers et ouvrez-les. Nous nous intéresserons seulement à ces deux choses :

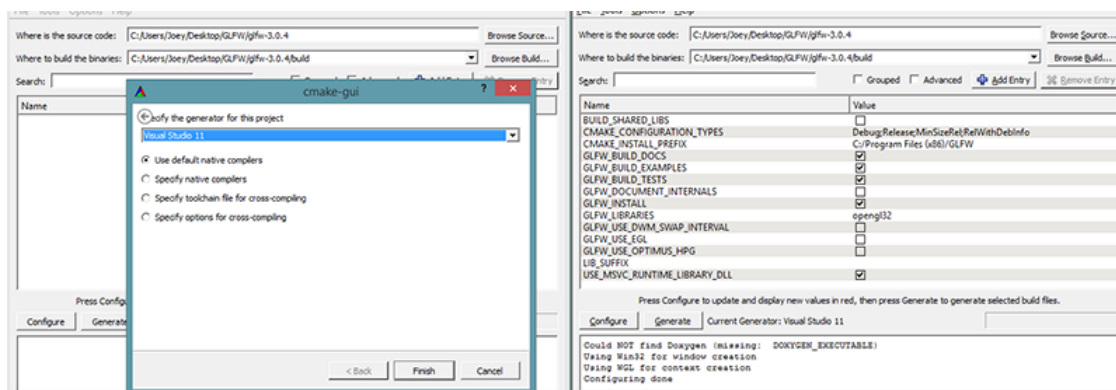
- la bibliothèque obtenue après compilation ;
- le dossier *include*.

Compiler la bibliothèque à partir des sources nous garantit que celle-ci sera parfaitement adaptée à notre CPU et OS, ce que les bibliothèques précompilées ne fournissent pas toujours. Le problème quand on fournit le code source, est que les utilisateurs n'ont pas tous le même IDE ce qui implique qu'ils doivent chacun construire leur projet avec les fichiers *.cpp*, *.h*, *.hpp*, ce qui est pénible. Pour cette raison, il existe un outil appelé CMake.

IV-B-1 - CMake

CMake est un outil pour générer les fichiers de projet selon le choix de l'utilisateur (par exemple Visual Studio, Code::Blocks, Eclipse) à partir d'un ensemble de fichiers source et utilisant des scripts CMake prédéfinis. Cela nous permet de générer un fichier projet Visual Studio 2012, à partir des sources de GLFW, que nous pourrions ensuite compiler pour créer la bibliothèque. Nous devons donc charger et installer CMake que l'on peut [trouver ici](#). Pour ma part, j'ai utilisé l'installateur Win32.

Une fois CMake installé, vous pouvez le lancer à partir d'une ligne de commande ou au moyen du GUI (*Graphical User Interface*). Pour simplifier, nous utiliserons le GUI. CMake demande de préciser un répertoire pour le code source et un répertoire de destination pour les binaires. Notre répertoire source sera la racine du package GLFW et nous créons un nouveau répertoire nommé build avant de le sélectionner pour CMake.



Une fois ces deux répertoires définis, cliquez sur le bouton Configure pour que CMake lise le code source et les paramètres. Nous choisissons ensuite le générateur pour ce projet, ici l'option Visual Studio 14 (Visual Studio 2015 s'appelle aussi Visual Studio 14). CMake va ensuite afficher les différentes options de compilation, nous gardons les valeurs par défaut et cliquons à nouveau sur Configure. Puis nous cliquons sur Generate et les fichiers produits seront placés dans le répertoire build.

IV-B-2 - Compilation

On trouvera dans le répertoire build un fichier nommé GLFW.sln, ouvrons-le avec Visual Studio 2015. Puisque CMake a généré un fichier projet qui contient déjà la bonne configuration, il nous suffit de cliquer sur Build Solution et la bibliothèque compilée se trouvera dans src/Debug sous le nom glfw3.lib (notons que nous utilisons la version 3).

Cette bibliothèque étant générée, nous devons nous assurer que l'IDE sait où se trouvent cette bibliothèque et les fichiers à inclure. On peut faire cela de deux façons :

- 1 Trouver où est placé le répertoire include de notre compilateur et y copier le contenu du répertoire include de GLFW et aussi ajouter glfw3.lib dans le répertoire lib de l'IDE. Ça marche mais ce n'est pas la bonne méthode. Si vous réinstallez votre IDE, vous perdrez la trace de vos fichiers GLFW.
- 2 La bonne méthode consiste à créer un nouvel ensemble de répertoires à l'endroit de votre choix, contenant les fichiers d'en-tête et bibliothèque de GLFW et dire à votre IDE où ils se trouvent. Pour ma part, j'ai créé un répertoire qui contient un sous-répertoire lib et un autre de nom include, où je place mes fichiers GLFW, ces répertoires servant pour chaque projet OpenGL. Bien sûr, pour chaque nouveau projet, il faut indiquer où se trouvent ces fichiers.

Cette étape accomplie, nous pouvons commencer à créer notre premier projet avec GLFW !

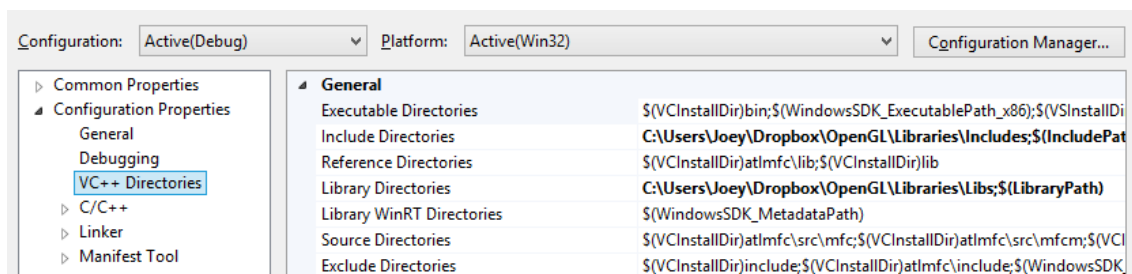
IV-C - Notre premier projet

Créons un nouveau projet avec Visual Studio. Choisir Visual C++ si plusieurs options se présentent ainsi qu'un projet vide (Empty Project) (donner aussi un nom correct à votre projet). Nous avons alors un espace de travail pour créer notre première application OpenGL.

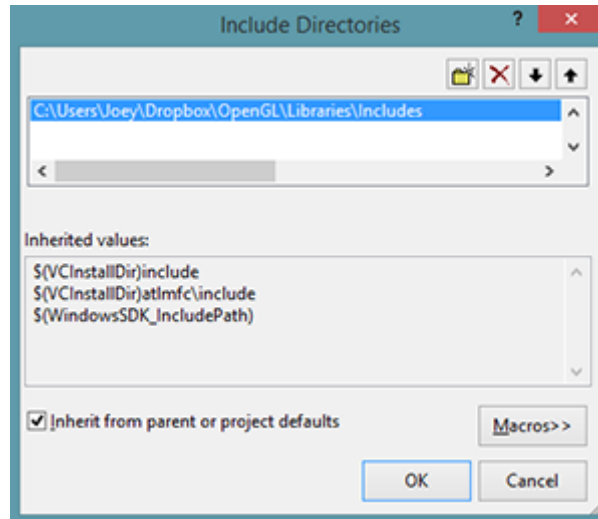
IV-D - Édition des liens (linking)

Pour utiliser GLFW, il faut lier la bibliothèque GLFW à notre projet. Il faut donc ajouter glfw3.lib dans la configuration de l'éditeur de liens du projet (*linker settings*), mais l'IDE ne sait pas où trouver glfw3.lib puisque nous l'avons placé dans le répertoire de notre choix. Il faut donc ajouter ce répertoire aux répertoires utilisés par Visual Studio pour trouver les bibliothèques.

Il faut modifier les propriétés du projet et trouver les répertoires utilisés avec VC++ comme indiqué ci-dessous :

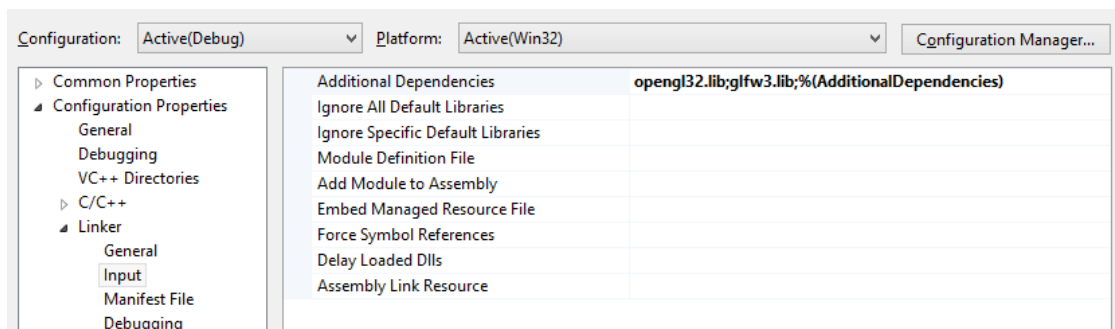


Dès lors, vous pouvez ajouter vos propres répertoires pour indiquer où trouver les fichiers. Cela peut être fait manuellement en insérant du texte ou bien en cliquant sur <Edit...> à la bonne ligne (lib ou include) :



Là, vous pouvez ajouter autant de répertoires que vous voulez et l'IDE cherchera vos fichiers aussi dans ces répertoires. Par exemple, il pourra trouver les fichiers d'en-tête nommés <GLFW/...>. C'est la même chose pour les bibliothèques.

Nous devons maintenant préciser quelles bibliothèques seront nécessaires pour générer le projet dans le menu Linker → input :



On spécifie le nom de la bibliothèque glfw3.lib que l'on ajoute aux Additional Dependencies, ce qui a pour effet de lier la bibliothèque GLFW à notre projet. On doit aussi ajouter la bibliothèque OpenGL, mais cela dépend des systèmes d'exploitation.

IV-D-1 - Bibliothèque OpenGL sur Windows

La bibliothèque opengl32.lib est installée avec Visual Studio, il nous suffit d'ajouter opengl32.lib dans les paramètres de l'éditeur de liens.

IV-D-2 - Bibliothèque OpenGL sur Linux

Sur Linux, il faut utiliser la bibliothèque libGL.so en ajoutant -lGL dans les paramètres de l'éditeur de liens. Si vous ne trouvez pas ce fichier (dans /usr/lib ou dans les sous-répertoires comme x86_64-linux-gnu), vous devrez probablement installer un de ces paquets : Mesa, NVidia ou AMD, mais je ne rentrerai pas dans ces détails (n'étant pas expert sur Linux).



Pour les utilisateurs de Linux, cette ligne de commande pourra vous aider à compiler le projet :
`-lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl`

Les bibliothèques ayant été bien configurées, on inclut le fichier d'en-tête de GLFW comme suit :

```
#include <GLFW\glfw3.h>
```

Voilà pour la configuration de GLFW.

IV-E - GLAD

Nous ne sommes pas encore arrivés, il reste une autre chose à faire. Puisqu'OpenGL est une spécification, il appartient au fabricant du pilote de la carte graphique d'implémenter cette API. Mais il existe beaucoup de versions différentes de pilotes OpenGL, l'emplacement de la plupart des fonctions n'est pas connu lors de la compilation et doit être retrouvé à l'exécution. Cela reste à la charge du développeur et cela **dépend du système d'exploitation utilisé**. Sur Windows cela ressemble à ça :

```
// Prototype de la fonction
typedef void (*GL_GENBUFFERS) (GLsizei, Gluint*);

// recherche de la fonction et mémorisation de son adresse
GL_GENBUFFERS glGenBuffers = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");

// la fonction peut ensuite être appelée normalement
unsigned int buffer;
glGenBuffers(1, &buffer);
```

Ce code est complexe et il serait pesant de l'écrire pour chaque fonction. Fort heureusement, une bibliothèque existe pour cela : GLAD, qui est bien connue et bien maintenue.

IV-E-1 - Installer GLAD

GLAD est une bibliothèque **open source**, qui s'installe un peu différemment de la plupart des bibliothèques. GLAD utilise un service **web** où nous pouvons préciser quelle est la version d'OpenGL utilisée et qui charge toutes les fonctions OpenGL correspondantes.

Rendez-vous sur le service **web**, choisissez le langage C++ et dans la section API, sélectionnez au minimum la version 3.3 (version utilisée dans les tutoriels, mais une version supérieure fonctionnera aussi). Assurez-vous de choisir le *core-profile* et que l'option **Generate a loader** est cochée. Cliquez sur **Generate** pour produire les fichiers de la bibliothèque.

GLAD devrait vous fournir un fichier zip contenant deux répertoires include et un fichier glad.c. Copiez ces deux répertoires include (glad et KHR) dans votre répertoire include (ou ajoutez un pointeur vers ces emplacements) et ajoutez le fichier glad.c à votre projet.

Il faut maintenant ajouter une directive include à votre fichier source :

```
#include <glad/glad.h>
```

Lancez la compilation. Elle devrait se produire sans erreur. Nous sommes prêts pour le prochain tutoriel où nous verrons comment utiliser GLAD et GLFW pour configurer un contexte OpenGL et afficher une fenêtre. Si vous avez un souci, vérifiez que toutes les ressources supplémentaires sont bien présentes et posez votre question dans le fil de discussion.

IV-F - Suppléments

-  **GLFW: Window Guide** : guide officiel GLFW pour afficher une fenêtre avec GLFW.

- **Compiler vos applications C** : informations sur les processus de compilation et éditions de liens avec une liste des erreurs possibles.
- **GLFW with Code::Blocks** : utiliser GLFW avec l'IDE Code::Blocks.
- **Running CMake** : aperçu de CMake sur Windows et Linux.
- **Writing a build system under Linux** : un tutoriel de Wouter Verholst pour écrire un système de construction de solution sur Linux, conçu pour ces tutoriels.
- **Polytonic/Glitter** : un projet simple préconfiguré avec toutes les bibliothèques.

IV-G - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

V - Hello window

Nous allons mettre en place GLFW et le faire fonctionner. Commençons par créer un fichier `.cpp` et ajoutons les fichiers d'en-tête :

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

Il faut inclure `glad.h` avant `glfw3.h` car le fichier `glad.h` inclut les fichiers d'en-tête adéquats d'OpenGL (`GL/gl.h`) nécessaires pour les autres dépendances.

Nous créons ensuite la fonction `main()` qui va instancier la fenêtre GLFW :

```
int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    return 0;
}
```

Dans la fonction `main()`, on initialise GLFW et on le configure avec `glfwWindowHint`. Le premier argument indique quelle option nous configurons parmi toutes les possibilités offertes, le second argument spécifie la valeur de cette option. Une liste de toutes ces options est disponible ici : [GLFW's window handling](#). Si l'on essaie de lancer l'application maintenant, on obtient des erreurs de type **undefined reference** signifiant que l'on n'est pas lié correctement à la librairie GLFW. Il faut préciser que nous travaillons avec la version 3.3 d'OpenGL. En indiquant la version majeure (3) et la version mineure (3), GLFW créera le contexte correct. Si l'ordinateur de l'utilisateur ne supporte pas cette version, le lancement de GLFW échouera. On doit aussi préciser que nous utilisons le core-profile. Cela signifie que nous aurons accès à un sous-ensemble plus petit des caractéristiques d'OpenGL (sans les fonctionnalités de compatibilité avec les anciennes versions).

Notons que sur Mac OS X vous aurez aussi besoin d'ajouter `glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);` pour un bon fonctionnement.



Assurez-vous d'avoir au moins la version 3.3 d'OpenGL sur votre machine sinon ça va planter ou produire un résultat non voulu. Pour trouver la version d'OpenGL installée sur votre machine, utiliser `glxinfo` sur Linux ou un utilitaire comme [OpenGL Extension Viewer](#) pour Windows. Si votre version n'est pas assez récente, vérifiez que votre carte graphique supporte OpenGL 3.3+ (sinon c'est vraiment ancien) et mettez à jour vos pilotes.

Ensuite, nous créons un objet fenêtre, laquelle contiendra toutes les données de fenêtrage et qui sera très souvent utilisée par les autres fonctions GLFW :

```
GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

Cette fonction requiert de passer la largeur et la hauteur de la fenêtre, le nom de cette fenêtre, et l'on peut ignorer le reste. Elle retourne un objet dont nous aurons besoin par la suite. Après cela, on indique à GLFW que le contexte de notre application sera le contexte de notre fenêtre.

V-A - GLAD

Nous avons indiqué précédemment que GLAD gère les pointeurs des fonctions OpenGL, il faut donc initialiser GLAD avant d'appeler la première fonction OpenGL :

```
if (!gladLoadGLLoader((GLADloadproc) glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

Nous passons à GLAD la fonction permettant de charger les adresses des pointeurs de fonctions OpenGL, lesquelles dépendent du système d'exploitation. GLFW offre une fonction générique `glfwGetProcAddress()` à cet effet.

V-B - Viewport

Avant d'effectuer le premier rendu, une chose reste à faire. Nous devons donner à OpenGL la taille de la fenêtre de rendu pour qu'OpenGL effectue ce rendu conformément à cette fenêtre. On spécifie ces dimensions avec la fonction `glViewport()` :

```
glViewport(0, 0, 800, 600);
```

Les deux premiers paramètres définissent la place du coin en bas à gauche de la fenêtre. On donne ensuite les dimensions de la fenêtre.

Nous pourrions définir des dimensions plus petites pour la zone de rendu (viewport), et OpenGL effectuerait ses rendus dans une zone plus petite, laissant la place restante pour d'autres affichages indépendants d'OpenGL.



En interne, OpenGL utilise les données spécifiées avec `glViewport()` pour transformer les coordonnées 2D qu'il calcule en coordonnées écran. Par exemple, un point situé en $(-0.5, 0.5)$ serait affiché en $(200, 450)$ en coordonnées écran. Notons que les coordonnées OpenGL sont entre -1 et $+1$, et ainsi on projette l'intervalle $[-1, +1]$ sur $[800, 600]$.

Cependant, si l'utilisateur redimensionne la fenêtre, la zone de rendu doit être ajustée à une nouvelle valeur. On réalise cela au moyen d'une fonction callback qui sera appelée à chaque fois que la fenêtre est redimensionnée. Le prototype de cette fonction est le suivant :

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

Cette fonction prend en premier argument une fenêtre de type `GLFWwindow` et ensuite deux valeurs pour les nouvelles dimensions. À chaque redimensionnement de la fenêtre, GLFW appelle cette fonction callback pour ajuster la taille du port d'affichage.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

Il faut indiquer cette fonction callback pour qu'elle soit appelée à chaque redimensionnement de la fenêtre :

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

Lorsque la fenêtre est affichée une première fois, `framebuffer_size_callback()` est exécutée, produisant ainsi les bonnes dimensions. Notons que pour les écrans Retina, largeur et hauteur donneront des valeurs plus grandes que prévu.

Il existe beaucoup de fonctions callback pour enregistrer le comportement de l'application. Par exemple, pour traiter les entrées joystick, gérer les messages d'erreur, etc. Nous installerons ces fonctions après avoir créé la fenêtre, mais avant que la boucle de rendu ne soit initiée.

V-C - Prêts pour le départ

Nous ne voulons pas que l'application affiche une seule image et se termine aussitôt, mais plutôt qu'elle continue à afficher des images en tenant compte des entrées utilisateur, ceci jusqu'à ce qu'on la stoppe. Pour cela, il faut créer une boucle, que nous appellerons la boucle de rendu, qui tourne tant que l'on ne l'arrêtera pas. Voici une boucle de rendu très simple :

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

La fonction `glfwWindowShouldClose()` vérifie à chaque passage que l'on ne souhaite pas fermer la fenêtre, auquel cas il nous resterait à fermer l'application. La fonction `glfwPollEvents()` regarde si un événement est survenu (comme une entrée clavier ou un mouvement de la souris), met à jour l'état de la fenêtre et appelle les fonctions callback correspondantes. La fonction `glfwSwapBuffers()` change le tampon (buffer) des couleurs (un tampon qui contient la couleur de chaque pixel de la fenêtre), pour afficher le contenu du tampon calculé pendant cette itération, selon la méthode du double buffer.



Double buffer : Lorsqu'une application n'utilise qu'un seul tampon, l'image résultante peut donner un effet de scintillement (*flickering*), car l'image n'est pas affichée d'un seul coup mais pixel par pixel. Si l'image, du fait des calculs, est modifiée pendant l'affichage, on constate des effets indésirables. Pour résoudre cela, les applications graphiques utilisent deux tampons. Le tampon avant (*front buffer*) contient l'image finale affichée sur l'écran, tandis que la construction de l'image est réalisée dans le tampon arrière (*back buffer*). Lorsque l'image est prête, on intervertit (*swap*) les deux buffers, affichant ainsi une image cohérente.

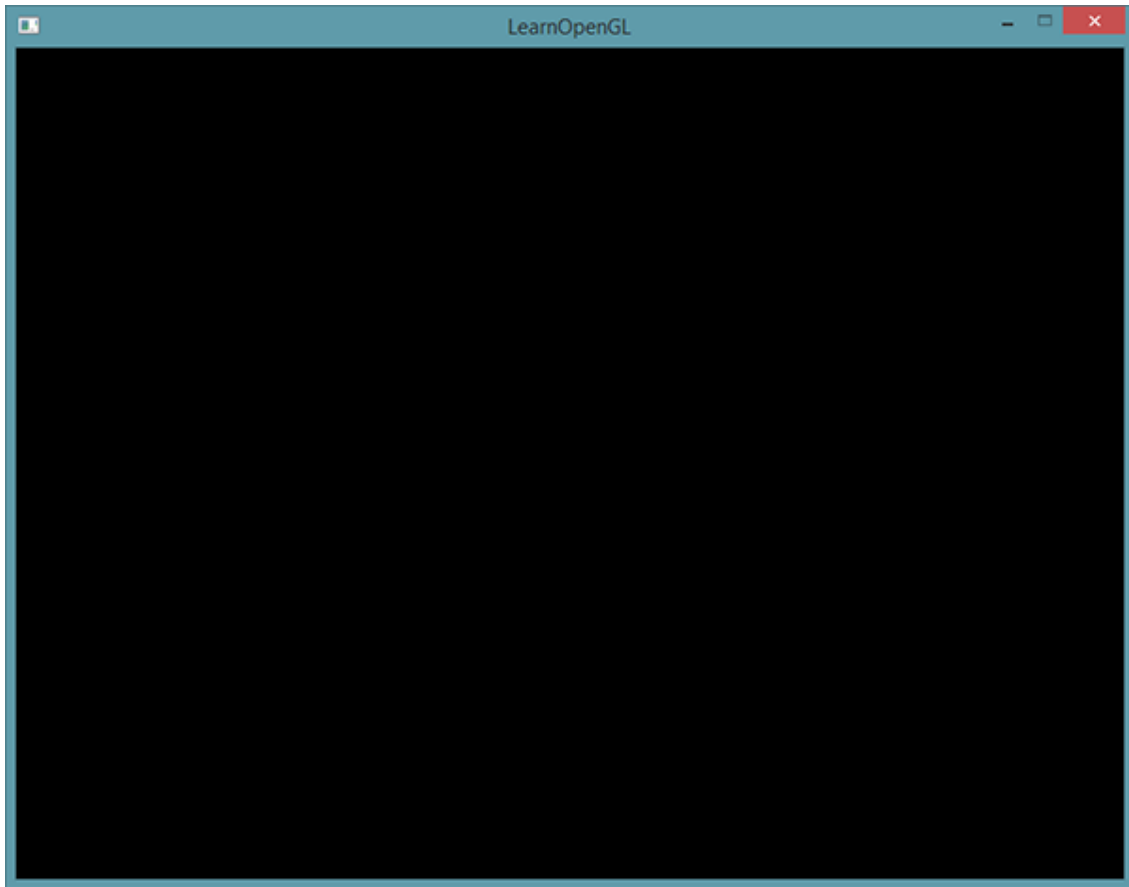
V-D - Une dernière chose

Lorsque l'on quitte la boucle de rendu, on souhaite fermer l'application proprement en effaçant ou supprimant les ressources que nous avons allouées :

```
glfwTerminate();
```

```
return 0;
```

Essayez maintenant de compiler votre application et vous devriez voir apparaître votre première fenêtre :



Si cette image noire est réellement ennuyeuse, vous avez réussi ! Sinon, vous pouvez consulter le code [ici](#).

En cas de souci, vérifiez vos options de compilation et notamment celles de l'éditeur de lien, et aussi que vous avez correctement inclus les fichiers d'en-tête. Vérifiez votre code en le comparant au code ci-dessus. Sinon, utilisez le fil de discussion pour poser une question ou voir les problèmes des autres lecteurs, et quelqu'un vous aidera.

V-E - Les entrées

Nous voulons aussi un minimum de contrôle au moyen des entrées utilisateur dans GLFW et, dans un premier temps, nous réaliserons cela avec `glfwGetKey()` qui prend en paramètre la fenêtre et la touche clavier utilisée. Cette fonction nous permet de savoir si cette touche est actuellement appuyée. Nous créons une fonction `processInput()` pour gérer cela :

```
void processInput(GLFWwindow *window)
{
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}
```

Ici, nous regardons si la touche Échap est utilisée (sinon, la fonction retourne `GLFW_RELEASE`). Si c'est le cas, nous fermons la fenêtre en positionnant la propriété `WindowShouldClose` à `true` grâce à la fonction `glfwSetWindowShouldClose()`. Le test de la boucle de rendu va échouer et ainsi l'application se fermera.

Ce test est effectué à chaque passage dans la boucle de rendu :

```
while (!glfwWindowShouldClose(window))
{
    processInput(window);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Cela nous donne un moyen simple de tester des touches du clavier et de réagir en conséquence.

V-F - Le rendu

Nous allons placer toutes les commandes de rendu dans la boucle, car nous souhaitons les exécuter à chaque passage :

```
// boucle de rendu

while(!glfwWindowShouldClose(window))
{
    // input
    processInput(window);

    // rendering commands here
    ...

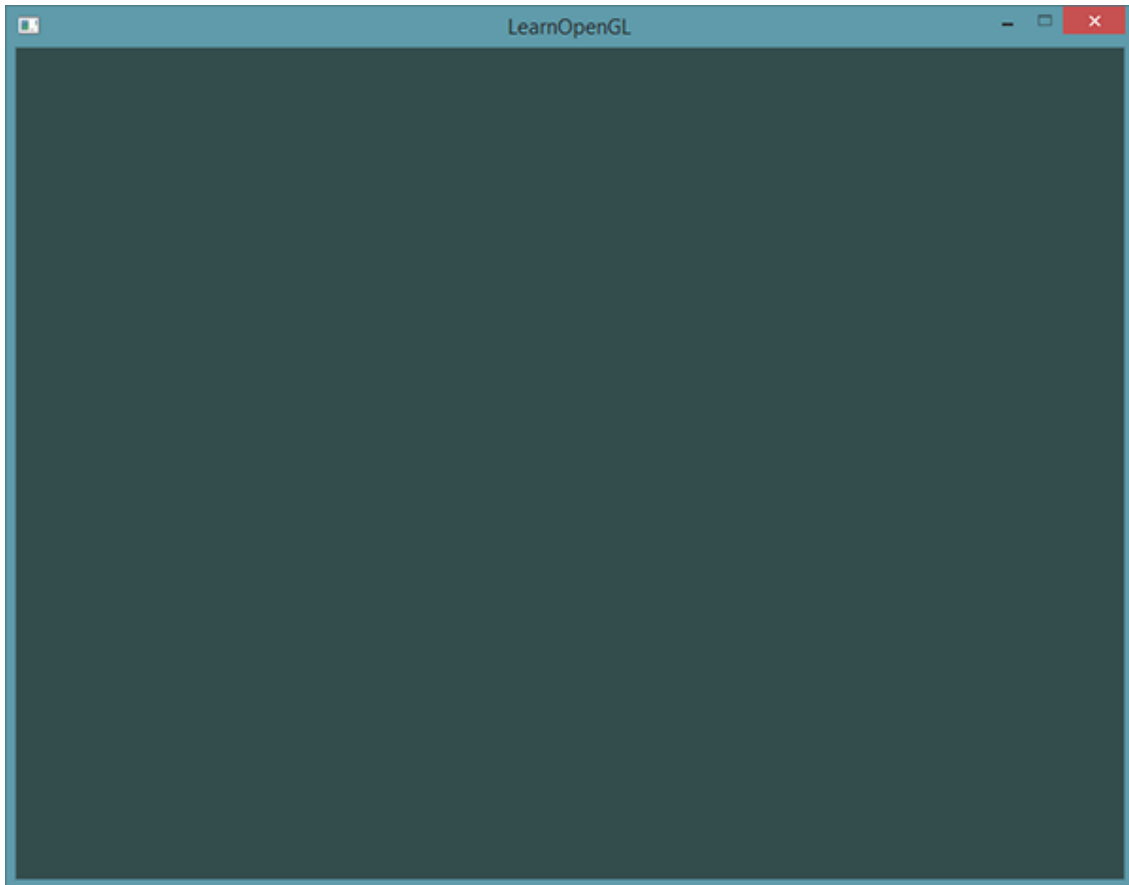
    // check and call events and swap the buffers
    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

Juste pour tester que les choses fonctionnent bien, nous allons colorer la fenêtre avec une couleur de notre choix. Au début de chaque rendu, nous effacerons l'écran, sinon nous verrions encore le résultat de l'itération précédente (cela peut être un effet voulu, mais souvent ce n'est pas le cas). On peut effacer le tampon de la couleur écran en utilisant la fonction `glClear()`, en spécifiant quels tampons nous voulons effacer. Les tampons possibles sont `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` et `GL_STENCIL_BUFFER_BIT`. Pour l'instant nous nous intéressons seulement aux couleurs :

```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

Nous avons choisi la couleur avec `glClearColor()`. À chaque appel de `glClear()`, le tampon sera entièrement rempli avec la couleur définie. On aura donc un fond bleu-gris ici.

Rappelez-vous le tutoriel OpenGL, la fonction `glClearColor()` est une fonction modifiant l'état et `glClear()` est une fonction pour mettre en service cet état.



Le code de cette application se trouve [ici](#).

On a désormais tout ce qu'il faut pour enrichir notre boucle de rendu, mais ce sera pour le tutoriel suivant.

V-G - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

VI - Hello Triangle

Dans OpenGL, tout est en 3D, mais l'écran et la fenêtre sont en réalité un tableau 2D de pixels. Par conséquent, une grande partie du travail d'OpenGL consiste à transformer les coordonnées 3D en coordonnées 2D sur l'écran. Ce travail est réalisé par le pipeline graphique. Celui-ci est divisé en deux grandes parties : la première transforme les coordonnées 3D en coordonnées 2D, la seconde transforme les coordonnées 2D en vrais pixels colorés. Dans ce tutoriel, nous discuterons rapidement du pipeline graphique et nous verrons comment l'utiliser pour afficher de jolis pixels.



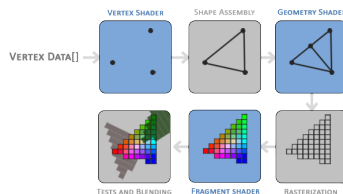
Une coordonnée 2D et un pixel sont deux choses différentes. Une coordonnée 2D est une représentation très précise d'un point de l'espace 2D, alors qu'un pixel est une approximation de ce point limitée par la résolution de l'écran ou de la fenêtre.

Le pipeline graphique utilise en entrée un ensemble de coordonnées 3D et les transforme en pixels colorés 2D sur l'écran. Le pipeline graphique se compose d'une suite d'étapes, où chaque étape utilise la sortie de l'étape précédente. Chacune de ces étapes est très spécialisée (chacune réalise une fonction très précise) et chacune peut

facilement être exécutée de façon parallèle. La plupart des cartes graphiques récentes possèdent des milliers de petits noyaux de calcul pour traiter rapidement les données du pipeline graphique en exécutant de petits programmes sur le processeur graphique (GPU), à chaque étape du pipeline. Ces petits programmes sont appelés *shaders*.

Certains de ces shaders sont configurables par le développeur, ce qui permet d'écrire ses propres shaders et de remplacer les shaders par défaut. Cela donne un contrôle bien plus fin sur certaines parties du pipeline, et permet d'économiser du temps CPU car ils s'exécutent sur le GPU. Les shaders sont écrits dans le langage GLSL (*OpenGL Shading Language*), que nous explorerons plus dans le prochain tutorial.

Une représentation de toutes les étapes du pipeline est figurée ci-dessous. Les parties bleues représentent les shaders qui peuvent être réécrits.



On remarque que le pipeline graphique contient un grand nombre de sections, chacune étant chargée d'une opération spécifique pour convertir les sommets (vertex) en pixels. Nous allons expliquer brièvement chacune de ces étapes, pour donner un bon aperçu du fonctionnement du pipeline.

En entrée du pipeline, on trouve une liste de trois coordonnées 3D qui formeraient ici un triangle dans un tableau appelé « Vertex Data ». Ces données forment un ensemble de sommets. Un sommet est un ensemble de données, associé à chacun des points que nous définirons par ses coordonnées 3D. Ces données sont représentées en utilisant des attributs de sommets, qui peuvent contenir n'importe quelles valeurs souhaitées, mais pour simplifier nous supposons que chaque sommet consiste juste en une position 3D et une couleur.

Pour qu'OpenGL puisse traiter ces collections de coordonnées et de couleurs, OpenGL doit être informé de quel type de rendu on souhaite obtenir avec ces données. Voulons-nous que ces données soient traitées comme de simples points, comme un ensemble de triangles, ou peut être juste comme une ligne ? Ces informations sont appelées primitives et sont transmises à OpenGL lors de l'appel des fonctions de dessin. Certains de ces attributs sont `GL_POINTS`, `GL_TRIANGLES` et `GL_LINE_STRIP`.

La première étape du pipeline est le vertex shader qui utilise en entrée un simple sommet. Son rôle principal est de transformer les coordonnées 3D en d'autres coordonnées 3D (plus de détails par la suite) et permet de réaliser des opérations de base sur les attributs des sommets.

L'étape de l'assemblage des primitives (*primitive assembly*) utilise en entrée tous les sommets (ou chaque sommet si `GL_POINTS` est choisi) issus du vertex shader qui forment une primitive, et assemble les points de cette forme primitive ; ici un triangle.

La sortie de cette étape est passée au geometry shader qui traite l'ensemble des sommets qui forment une primitive et qui a la capacité de générer d'autres formes en créant de nouveaux sommets pour créer de nouvelles primitives. Dans cet exemple, il générerait un second triangle à partir de la forme donnée.

La sortie du geometry shader est ensuite fournie à l'étape de rasterization qui fait correspondre aux primitives les pixels de l'écran final, pour donner des fragments qui sont traités par le fragment shader. Avant cela, on procède au clipping qui écarte les fragments situés en dehors de la vue, afin de ne pas traiter de données inutiles.



Dans OpenGL, un fragment contient toutes les données requises pour afficher un pixel.

Le but principal du fragment shader est de calculer la couleur d'un pixel et c'est en général l'étape au cours de laquelle tous les effets avancés d'OpenGL sont mis en œuvre. Le fragment shader contient des données sur la scène 3D, qu'il utilise pour calculer la couleur finale d'un pixel (comme les éclairages, les ombres, la couleur de la lumière, etc.).

Après que les couleurs ont été déterminées, l'objet final est ensuite passé à une ultime étape appelée le test alpha et la fusion (blending). Cette étape vérifie la profondeur (et le stencil) (détails plus loin) correspondante du fragment et l'utilise pour vérifier si le fragment est devant ou derrière d'autres objets pour éventuellement l'écarter. Cette étape vérifie aussi les valeurs alpha (valeur indiquant la transparence) et mélange les objets en conséquence. Ainsi, même si la couleur d'un pixel est calculée par le fragment shader, la couleur finale peut être très différente lorsque l'on affiche plusieurs triangles.

Comme on le voit, le pipeline graphique est un ensemble très complexe qui contient de nombreuses parties configurables. Cependant, dans la plupart des cas, on travaillera seulement avec le vertex shader et le fragment shader. Le geometry shader est optionnel et, en général, on utilise celui par défaut.

En OpenGL moderne, il **faut définir** au moins le vertex shader et le fragment shader soi-même (il n'en existe pas par défaut dans le GPU). Ainsi, il est souvent assez difficile de commencer l'apprentissage de l'OpenGL moderne, car des connaissances importantes sont nécessaires avant de pouvoir afficher un simple triangle. Lorsque vous aurez affiché un triangle à la fin de ce chapitre, vous en saurez déjà bien plus sur la programmation graphique.

VI-A - Sommets

Pour commencer à afficher quelque chose, nous devons donner à OpenGL des sommets en entrée. OpenGL est une bibliothèque graphique 3D, toutes les coordonnées sont donc en 3D (x, y, z). OpenGL ne fait pas que transformer les coordonnées 3D en pixels 2D sur l'écran ; OpenGL ne traite les coordonnées 3D que dans l'intervalle $[-1.0, +1.0]$ sur chacun des trois axes (x, y, z). Toutes les coordonnées de cet intervalle (appelées les coordonnées normalisées pour le périphérique) seront visibles sur l'écran (à l'exclusion de celles qui ne le seraient pas pour d'autres raisons).

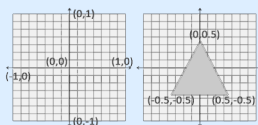
Puisque nous voulons afficher un seul triangle, nous allons définir trois sommets, chaque sommet ayant une position 3D. Nous les définissons en coordonnées normalisées dans un tableau de réels :

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

OpenGL opérant dans un espace 3D, un triangle 2D sera contenu dans le plan $z=0$. De cette façon, le triangle a la même apparence qu'un triangle en 2D.

Coordonnées normalisées

Une fois les coordonnées d'un sommet traitées par le vertex shader, elles doivent être en coordonnées normalisées, soit avec x, y et z dans l'intervalle $[-1.0, +1.0]$. Toute coordonnée en dehors de cet intervalle sera écartée et ne sera donc pas visible. On peut voir ci-dessous le triangle que nous avons défini (sans l'axe z) :



Contrairement aux coordonnées habituelles de l'écran, l'axe y pointe vers le haut, et le point (0, 0) est au centre du dessin, au lieu d'être en haut à gauche. Vous veillerez à ce que les coordonnées finales se trouvent dans cet espace, autrement les sommets ne seront pas visibles.

Les coordonnées normalisées seront transformées en coordonnées écran en utilisant la transformation de la zone d'affichage (viewport), que l'on aura spécifiée avec `glViewport()`. Les coordonnées écran obtenues sont ensuite transformées en fragments pour être passées au fragment shader.

Nous allons maintenant fournir ces données de sommets à la première étape du pipeline graphique : le vertex shader. Pour cela, on crée une zone mémoire dans le GPU pour y placer ces données, puis on configure la façon dont OpenGL interprétera ces données et comment ces données seront envoyées à la carte graphique. Le vertex shader traite ensuite les données qui lui ont été fournies.

Cette zone mémoire est représentée par un Vertex Buffer Object (VBO), un tampon qui peut contenir un grand nombre de sommets dans la mémoire du GPU. L'intérêt d'utiliser ces objets tampons est de pouvoir envoyer simultanément un grand ensemble de données à la carte graphique sans avoir à envoyer les sommets un par un. Envoyer des données du CPU vers la carte graphique est assez lent, ainsi on essaiera d'envoyer autant de données que possible en une seule fois. Une fois ces données stockées dans la mémoire de la carte graphique, le vertex shader a un accès direct aux données des sommets, ce qui est très rapide.

Le VBO est le premier objet OpenGL que nous rencontrons. Comme tout objet OpenGL, ce tampon a un identifiant unique que l'on peut générer avec la fonction `glGenBuffers()` :

```
unsigned int VBO;  
glGenBuffers(1, &VBO);
```

OpenGL dispose de beaucoup d'objets tampon et le type pour un VBO est `GL_ARRAY_BUFFER`. OpenGL permet d'utiliser plusieurs tampons à la fois, tant qu'ils ont des types différents. On peut attacher ce tampon nouvellement créé à la cible `GL_ARRAY_BUFFER` en utilisant la fonction `glBindBuffer()` :

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

Dès lors, chaque appel travaillant sur un tampon (dont la cible est `GL_ARRAY_BUFFER`), modifiera le tampon actuellement lié, c'est-à-dire le VBO. On peut ainsi utiliser la fonction `glBufferData()` qui copie les sommets définis dans la mémoire du tampon :

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

`glBufferData()` est une fonction dédiée pour copier les données utilisateur dans le tampon du type défini. Le premier argument spécifie le type de tampon dans lequel les données doivent être copiées : ici le VBO lié à la cible `GL_ARRAY_BUFFER`.

Le second argument précise la taille des données (en octets) à copier. Un simple `sizeof` des données des sommets suffit. Le troisième paramètre pointe vers les données à transférer.

Le dernier paramètre spécifie comment nous souhaitons que ces données soient traitées. Cela peut prendre trois formes :

- `GL_STATIC_DRAW` : les données ne seront pas modifiées (ou rarement) ;
- `GL_DYNAMIC_DRAW` : les données seront souvent modifiées ;
- `GL_STREAM_DRAW` : les données seront modifiées à chaque affichage.

Les sommets de notre triangle ne changent pas et restent identiques à chaque rendu, ainsi nous utiliserons l'option `GL_STATIC_DRAW`. Si nos sommets changeaient souvent, les autres options permettraient à la carte graphique de placer les données dans une zone mémoire ayant un accès plus rapide.

Nous avons jusqu'ici transféré nos données dans une zone mémoire de la carte graphique, gérées par un objet appelé le VBO. Nous allons maintenant créer un vertex shader et un fragment shader pour traiter ces données.

VI-B - Le vertex shader

Le vertex shader est l'un des shaders programmables par l'utilisateur. En OpenGL moderne, il faut au minimum configurer un vertex shader et un fragment shader pour effectuer un rendu ; nous allons ainsi présenter brièvement les shaders et configurer deux shaders très simples pour afficher notre premier triangle. Dans le [prochain tutorial](#), nous examinerons les shaders plus en détail.

La première chose à faire est d'écrire le vertex shader en langage GLSL (*OpenGL Shading Language*), puis le compiler pour l'utiliser dans notre application. Ci-dessous est présenté le code source d'un vertex shader très basique en langage GLSL :

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Comme on peut le voir, le GLSL ressemble au C. Chaque shader commence par la déclaration de la version utilisée. Depuis OpenGL 3.3 et pour les versions suivantes, les numéros de version de GLSL correspondent aux numéros de version d'OpenGL (la version 420 de GLSL correspond à la version 4.2 d'OpenGL par exemple). Nous mentionnerons aussi explicitement que nous utilisons les fonctionnalités core-profile.

Ensuite, nous déclarons tous les attributs des sommets comme entrées du vertex shader, avec le mot clé `in`. Pour l'instant, on se soucie seulement de la position des sommets, nous n'avons donc besoin que d'un seul attribut de sommet. GLSL a un type vecteur qui peut contenir de un à quatre nombres à virgule flottante, ce nombre étant spécifié par le dernier caractère du type. Puisque chaque sommet possède trois coordonnées, nous créons une variable de type `vec3`, que nous appelons `aPos`. Nous spécifions aussi l'emplacement de la variable d'entrée avec `layout (location = 0)` et nous verrons ensuite pourquoi nous utilisons cet emplacement.

Les Vecteurs

La programmation graphique utilise largement le concept mathématique de vecteur (voir [chapitre 8](#)), car il permet de représenter les positions et directions dans tout espace ; de plus les vecteurs ont des propriétés très utiles. Un vecteur GLSL a au plus quatre membres qui peuvent être accédés grâce à `vec.x`, `vec.y`, `vec.z` et `vec.w` respectivement, chacun d'eux pouvant représenter une coordonnée dans l'espace. `vec.w` n'est pas utilisé comme une coordonnée (ici nous travaillons en 3D, pas en 4D), mais trouve son utilité dans ce qu'on appelle la division de perspective. Nous parlerons des vecteurs plus en détail dans un tutorial ultérieur.

Pour définir la sortie du vertex shader, nous devons assigner la position des données à la variable prédéfinie `gl_Position` qui est un `vec4` prédéfini. À la fin de la fonction `main`, ce que nous aurons défini avec cette variable `gl_Position` sera utilisé en sortie du vertex shader. Puisque notre entrée est un `vec3`, nous devons le convertir en `vec4`. Cela est fait en insérant les valeurs de `vec3` dans un constructeur `vec4` et en assignant la valeur `1.0f` à `vec.w` (nous verrons pourquoi plus tard).

Le vertex shader que nous avons construit est des plus simples, car nous ne faisons aucun traitement sur les données entrées, nous ne faisons que les copier en sortie. Dans une application réelle, les données entrées ne sont pas déjà des coordonnées normalisées et nous aurons à les modifier pour qu'elles soient dans la région visible d'OpenGL.

VI-C - Compiler un shader

Nous avons écrit le code source du vertex shader (archivé dans des chaînes de caractères C), mais pour qu'OpenGL utilise le shader, il doit être compilé dynamiquement lors de l'exécution.

La première opération consiste à créer un objet shader, à nouveau référencé par son identifiant. Nous mémorisons le shader avec un unsigned int et créons le shader avec `glCreateShader()` :

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

Nous précisons le type de shader que nous voulons créer en passant comme argument `GL_VERTEX_SHADER` pour un vertex shader.

Ensuite, nous attachons le code source du shader à l'objet shader et compilons le shader :

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

Cette fonction prend l'objet shader à compiler en premier argument ; le second argument spécifie combien de chaînes nous utilisons pour le code source, ici seulement une. Le troisième paramètre est le code source lui-même du vertex shader et nous laissons le dernier paramètre à `NULL`.

On voudra sans doute tester si la compilation a réussi et sinon traiter les erreurs. Tester le résultat de la compilation s'effectue ainsi :

```
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

i Nous définissons un entier pour indiquer le résultat de l'opération, ainsi qu'un tampon pour les messages d'erreurs éventuels. Ensuite, nous testons le résultat. En cas d'erreur, nous affichons le message d'erreur :

```
if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
    infoLog << std::endl;
}
```

S'il n'y a pas d'erreur, le vertex shader est désormais compilé.

VI-D - Le fragment shader

Le fragment shader est le second et dernier shader que nous aurons à créer pour afficher le triangle. Ce shader calculera la couleur des pixels. Pour simplifier, ce shader donnera la couleur orange pour tous les pixels.

i Les couleurs en programmation graphique sont définies par un tableau de quatre valeurs : rouge, vert, bleu et alpha (transparence), abrégées en **RGBA**. Pour définir une couleur dans

OpenGL ou GLSL, on choisira pour chaque composante une valeur entre 0.0 et 1.0. Si par exemple on choisit 1.0 pour le rouge et le vert et 0.0 pour le bleu, on aura un mélange de rouge et de vert, soit du jaune. On peut ainsi générer 16 millions de couleurs différentes !

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Le fragment shader ne requiert qu'une seule variable de sortie, un vecteur de dimension 4 qui définira la couleur de sortie que nous devons calculer nous-mêmes. On peut déclarer les valeurs de sortie avec le mot clef `out`, que nous appelons `FragColor`. On assigne ensuite la couleur de sortie (orange) avec une transparence de 1.0, c'est-à-dire opaque.

Pour compiler le fragment shader, on procède de la même façon que pour le vertex shader, mais on utilise l'option `GL_FRAGMENT_SHADER` :

```
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

Les deux shaders sont compilés, il reste à lier ces deux objets shader dans un program shader que nous utiliserons pour effectuer le rendu.

VI-E - Le program shader

L'objet program shader est la version finale liée, combinaison des différents shaders. Pour utiliser les shaders compilés, nous devons les lier à un objet program shader, et activer celui-ci pour le rendu. Les shaders de ce programme seront ainsi utilisés lors des appels effectués pour le rendu.

Lors de l'édition de liens du program shader, chaque sortie de shader est utilisée comme entrée du shader suivant. Il peut y avoir des erreurs lors de l'édition des liens, si les sorties ne correspondent pas aux entrées.

La création du program shader est simple :

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

La fonction `glCreateProgram()` crée un program shader et retourne l'identifiant de ce nouvel objet. On doit ensuite attacher les shaders compilés à ce program shader, et les lier avec `glLinkProgram()` :

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Comme pour la compilation des shaders, on peut vérifier si l'édition de liens du programme a réussi, et retrouver les erreurs éventuelles. Cependant, à la place de `glGetShaderiv()`, nous utiliserons `glGetProgramiv()` :

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if(!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    ...
}
```

```
}
```

Le résultat est un objet programme que l'on peut activer en appelant `glUseProgram()` avec comme argument le nouveau programme :

```
glUseProgram(shaderProgram);
```

Chaque shader et chaque appel pour le rendu utilisera cet objet.

Ah oui, n'oublions pas de détruire les objets shader une fois intégrés au program shader, ils ne sont plus utiles :

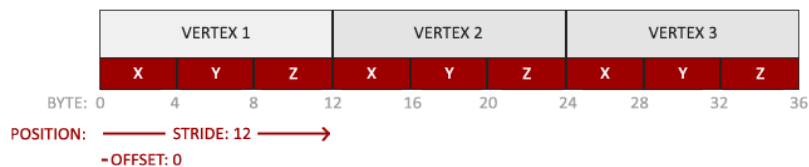
```
glDeleteShader(vertexShader);  
glDeleteShader(fragmentShader);
```

Maintenant, nous allons transmettre les sommets au GPU et lui spécifier comment il doit traiter ces sommets dans le vertex shader et le fragment shader. Nous y sommes, mais pas encore tout à fait. OpenGL ne sait pas comment interpréter les données sommets en mémoire et comment il doit connecter les données des sommets aux attributs du vertex shader. Nous allons lui dire comment faire.

VI-F - Lier les attributs de sommets

Le vertex shader nous permet de spécifier chaque entrée sous la forme d'attributs de sommets et bien que cela autorise une grande flexibilité, cela implique de spécifier quelles parties de nos données correspondent à quels attributs de sommets du shader. Il faut donc spécifier comment OpenGL interprétera les données avant le rendu.

Les données de nos sommets sont formatées ainsi :



- chaque coordonnée des positions est mémorisée dans un float codé sur 32 bits (4 octets) ;
- chaque position est composée de 3 valeurs ;
- il n'y a pas d'espace libre entre les données, qui sont consécutives dans un tableau ;
- la première valeur est au début du tampon.

Sachant cela, on peut dire à OpenGL comment il doit interpréter les données (par attribut de sommet) en utilisant `glVertexAttribPointer()` :

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);  
glEnableVertexAttribArray(0);
```

Cette fonction utilise plusieurs arguments qu'il faut examiner :

- Le premier paramètre spécifie quel attribut nous voulons configurer. Rappelons-nous que nous avons spécifié la position de l'attribut dans le vertex shader avec `layout (location = 0)`. Cela définit l'endroit de l'attribut à 0, et puisque nous souhaitons passer les données à cet attribut, nous passons 0 en argument.
- L'argument suivant spécifie la taille de l'attribut. Il s'agit d'un `vec3`, il est donc composé de trois valeurs.
- Le troisième argument spécifie le type de données qui est ici `GL_FLOAT` (une donnée de type `vec*` est constituée de nombres à virgule flottante dans GLSL).

- L'argument suivant spécifie si nous souhaitons que les données soient normalisées. Si nous le positionnons à `GL_TRUE`, toutes les données qui ne seront pas dans l'intervalle `[0.0, 1.0]` (`[-1.0, 1.0]` pour les données signées) seront projetées sur cet intervalle. Ici nous laissons cet argument à `GL_FALSE`.
- Le cinquième argument se nomme `stride`. Il détermine l'espace entre les différents ensembles consécutifs d'attributs. Puisque la position qui suit se trouve exactement à une distance de trois fois la taille d'un float, nous spécifions cette valeur pour le `stride`. Puisque les données dans le tableau sont consécutives les unes aux autres, nous aurions pu aussi spécifier 0 pour le `stride` et laisser OpenGL calculer lui-même la valeur du `stride`. À chaque fois que nous aurons plusieurs attributs, nous devrons définir précisément l'espace séparant ces attributs, nous en verrons des exemples plus loin.
- Le dernier paramètre est de type `void*` et requiert cet étrange transtypage. Il indique le décalage (offset) où les données de position commencent dans le tampon. Puisque nos données position sont placées au début du tampon, la valeur à mettre est 0. Nous verrons ce paramètre en détail dans la suite.



Chaque attribut tire ses données de la mémoire gérée par un VBO, en l'occurrence celui lié à `GL_ARRAY_BUFFER` (il peut y avoir plusieurs VBO). Puisque le VBO défini a été lié avant l'appel, l'attribut vertex 0 est donc associé avec les données des sommets.

Maintenant que nous avons spécifié comment OpenGL doit interpréter les données des sommets, nous devons aussi activer l'attribut en question avec `glEnableVertexAttribArray()` en passant en argument l'endroit où se trouve l'attribut (les attributs sont désactivés par défaut). Dès lors tout est en place : nous avons initialisé les données dans un tampon en utilisant un VBO, défini un vertex shader et un fragment shader et dit à OpenGL comment lier les données des sommets aux attributs du vertex shader. Afficher un objet dans OpenGL va donc ressembler à cela :

```
// 0. Copier nos sommets dans un tampon
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. Initialiser un pointeur vers les attributs des sommets
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. Utiliser notre program shader pour l'affichage d'un objet
glUseProgram(shaderProgram);
// 3. Afficher un objet
someOpenGLFunctionThatDrawsOurTriangle();
```

Nous aurons à répéter ces opérations à chaque fois que nous voudrions afficher un objet. Mais supposons que nous ayons cinq attributs et peut-être 100 objets différents, lier les tampons et configurer tous les attributs deviendrait rapidement très fastidieux. N'y aurait-il pas un moyen d'archiver ces configurations dans un objet et de lier simplement cet objet pour retrouver l'état voulu ?

VI-G - Le Vertex Array Object

Un Vertex Array Object (VAO) peut être lié exactement comme un VBO et tous les appels d'attributs à partir de ce point seront archivés dans le VAO. L'avantage est qu'après avoir configuré les attributs, on n'a qu'à effectuer l'appel une fois, et à chaque fois que l'on voudra afficher l'objet, on n'aura qu'à lier le VAO correspondant. Cela permet de passer de certaines données de sommets et attributs de sommets à d'autres en se liant seulement à différents VAO. L'état qui est défini est archivé dans le VAO.

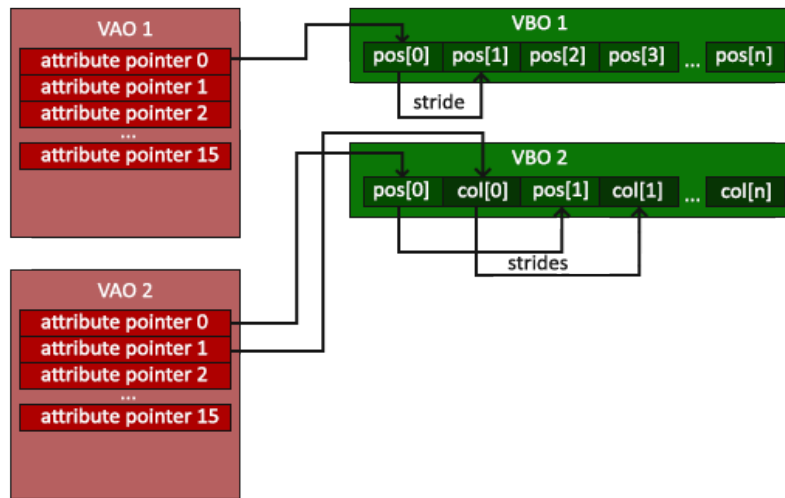


Le core-profile OpenGL requiert l'utilisation d'un VAO pour savoir que faire de nos sommets en entrée. Si l'on ne se lie pas à un VAO, OpenGL refuse d'afficher quoi que ce soit.

Un VAO archive les choses suivantes :

- les appels à `glEnable` ou à `glDisableVertexAttribArray()` ;
- les configurations des attributs via `glVertexAttribPointer()` ;

- les Vertex Buffer Objects (VBO) associés aux attributs par les appels à `glVertexAttribPointer()`.



Le processus pour générer un VAO est similaire à celui pour générer un VBO :

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

Pour utiliser un VAO, il suffit de le lier avec `glBindVertexArray()`. Dès lors, on peut lier/configurer le VBO correspondant et les pointeurs d'attribut et détacher ensuite le VAO pour un usage ultérieur. Dès que l'on veut afficher un objet, on lie simplement le VAO avec les options voulues avant d'afficher l'objet et c'est tout. Le code ressemble à cela :

```
// ...: Initialisation (à faire une seule fois, sauf si l'objet change souvent) :: ..
// 1. Lier le Vertex Array Object (VAO)
glBindVertexArray(VAO);
// 2. Copier les sommets dans un tampon pour qu'OpenGL les utilise
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. Initialiser les pointeurs d'attributs de sommets
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ...: Code pour l'affichage (dans la boucle de rendu) :: ..
// 4. Afficher l'objet
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
une_fonction_OpenGL_qui_dessine_notre_triangle();
```

Et c'est tout ! Toutes ces pages pour arriver à cela : un VAO qui mémorise notre configuration d'attributs et quel VBO utiliser. Lorsque l'on veut afficher de nombreux objets, on génère et configure tous les VAO (et aussi les VBO requis et les pointeurs d'attributs) puis on les archive pour un usage ultérieur. Au moment d'afficher un des objets, on prend le VAO correspondant, on le lie, on affiche l'objet et on détache le VAO.

VI-H - Le triangle que nous attendions

Pour afficher les objets souhaités, OpenGL fournit la fonction `glDrawArrays()` qui trace des primitives en utilisant le shader actuellement actif, la configuration d'attributs définie précédemment avec les données de sommets du VBO (liées indirectement par le VAO).

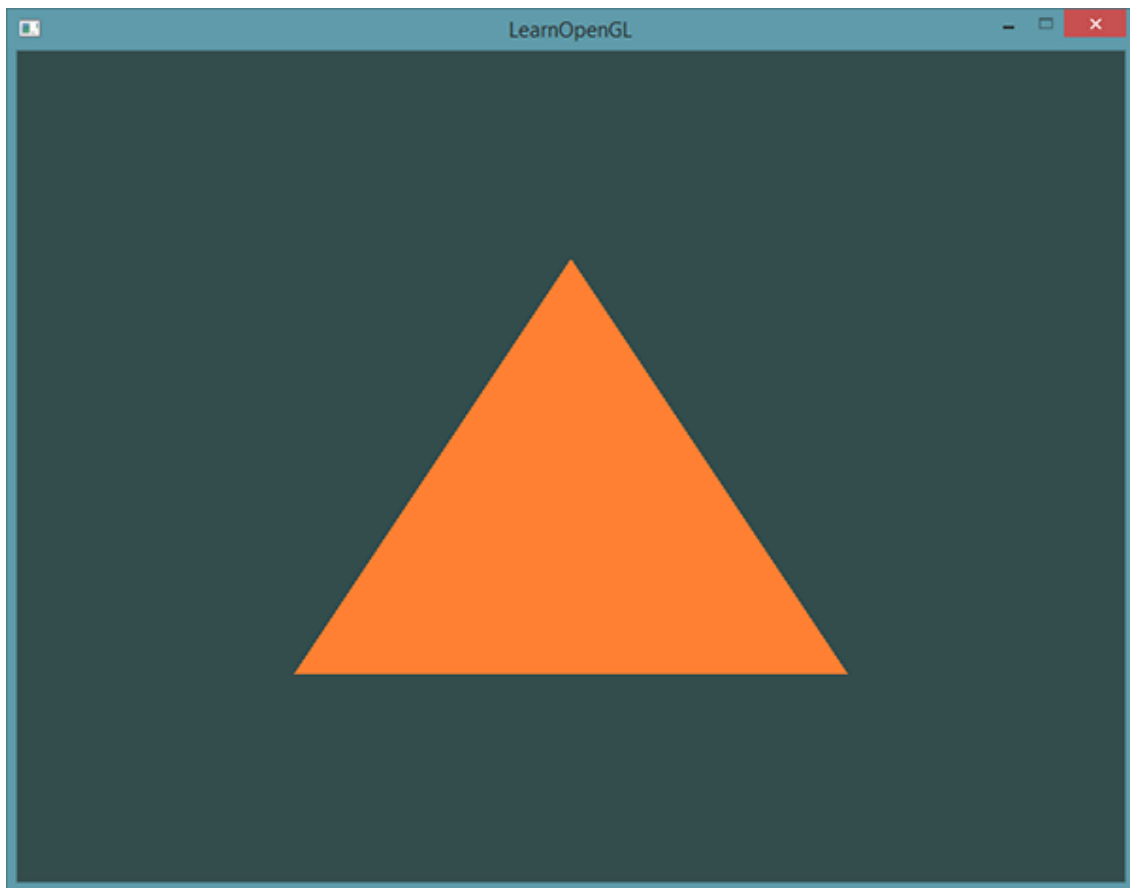
```
glUseProgram(shaderProgram);
```



```
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

La fonction `glDrawArrays()` utilise en premier argument le type de primitive que nous voulons tracer, ici `GL_TRIANGLES`. Le second argument spécifie l'index de début du tableau de sommets à afficher, ici 0. Le dernier argument précise combien de sommets doivent être tracés, trois dans notre cas (nous n'affichons qu'un seul triangle grâce à nos données qui contiennent exactement trois sommets).

Compilez le code et vérifiez-le si des erreurs se produisent. Puis vous devriez voir s'afficher le résultat suivant :



Le code source se trouve [ici](#).

En cas de problème, vous avez probablement commis une erreur, vérifiez le code entièrement, et si le problème persiste, posez une question sur [le forum](#).

VI-I - Les Element Buffer Objects

Une dernière chose à examiner pour le rendu des sommets est l'Element Buffer Objects (EBO). Pour expliquer le fonctionnement d'un EBO, le mieux est de donner un exemple. Supposons que nous voulions tracer un rectangle plutôt qu'un triangle. On peut afficher un rectangle en affichant deux triangles (OpenGL travaille essentiellement avec des triangles). Nous définirions cet ensemble de sommets :

```
float vertices[] = {
    // first triangle
    0.5f,  0.5f, 0.0f, // top right
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, 0.5f, 0.0f, // top left
    // second triangle
    0.5f, -0.5f, 0.0f, // bottom right
```

```
-0.5f, -0.5f, 0.0f, // bottom left
-0.5f, 0.5f, 0.0f // top left
};
```

Comme on peut le voir, les données se recouvrent. Nous avons spécifié le sommet bas droit et le sommet haut gauche deux fois ! Le rectangle peut être défini par quatre sommets, et non six. Cela deviendrait très complexe dès que des objets complexes sont à afficher. Une meilleure solution consiste à ne définir les sommets nécessaires qu'une seule fois, et de spécifier ensuite l'ordre dans lequel nous souhaitons afficher ces sommets. Nous aurions donc seulement quatre sommets pour le rectangle, et ensuite à définir dans quel ordre les tracer.

Heureusement un EBO sert à cela. Un EBO est un tampon, juste comme un VBO, qui mémorise des indices qu'OpenGL utilisera pour décider quels sommets sont à afficher. Tout d'abord, spécifions les sommets du rectangle, puis les indices :

```
float vertices[] = {
    0.5f, 0.5f, 0.0f, // top right
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, // bottom left
    -0.5f, 0.5f, 0.0f // top left
};
unsigned int indices[] = { // Notons que l'on commence à 0!
    0, 1, 3, // premier triangle
    1, 2, 3 // second triangle
};
```

Nous devons ensuite créer l'EBO :

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

De la même façon que pour le VBO, nous lions l'EBO et y recopions les indices avec `glBufferData()`. Ces appels sont faits entre l'établissement du lien et sa suppression.

Cette fois nous utilisons en paramètre `GL_ELEMENT_ARRAY_BUFFER` comme type de tampon.

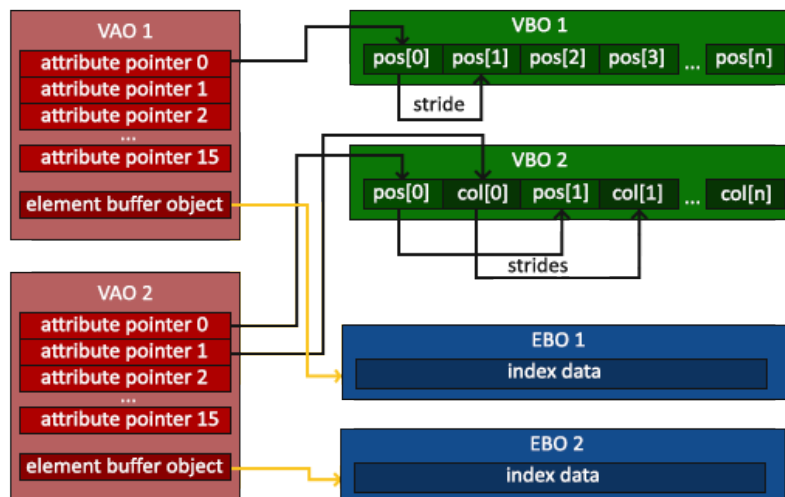
```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Notons que nous donnerons `GL_ELEMENT_ARRAY_BUFFER` comme cible de tampon. La dernière chose à faire est de remplacer l'appel pour indiquer que nous affichons les triangles à partir d'un EBO. L'affichage sera fait à partir des indices de l'EBO qui est lié à ce moment-là, grâce à `glDrawElements()` :

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Le premier argument spécifie le mode dans lequel nous voulons afficher, comme avec `glDrawArrays()`. Le second argument indique le nombre d'éléments à tracer (6 dans cet exemple). Le troisième argument est le type d'indice utilisé, ici `GL_UNSIGNED_INT`. Le dernier argument nous permet de préciser un décalage dans l'EBO (ou bien de passer un indice de tableau quand on n'utilise pas un EBO), ici nous indiquons 0.

La fonction `glDrawElements()` prend les indices dans l'EBO actuellement lié à la cible `GL_ELEMENT_ARRAY_BUFFER`. Cela implique d'attacher l'EBO correspondant à chaque rendu d'un objet avec les indices, ce qui semble fastidieux. Heureusement, un VAO garde la trace des liens avec les EBO. L'EBO en cours lorsqu'un VAO est attaché est archivé en qualité d'EBO de ce VAO. L'attachement du VAO attache donc automatiquement son EBO.



Un VAO archive les appels `glBindBuffer()` lorsque la cible est `GL_ELEMENT_ARRAY_BUFFER`. Cela signifie qu'il archive les appels de détachement pour être sûr que l'on ne détache pas l'EBO avant de détacher le VAO, sinon, l'EBO ne serait plus configuré.

L'initialisation et l'affichage ressemblent donc à ceci :

```
// ::: Initialisation :: ..
// 1. attacher le Vertex Array Object
glBindVertexArray(VAO);

// 2. copier les sommets dans un VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

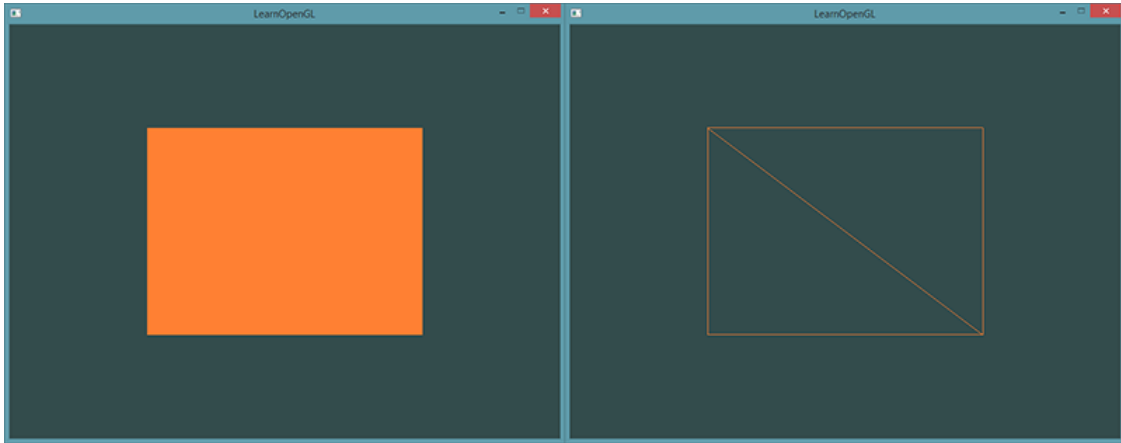
// 3. copier le tableau d'indices dans un tampon d'éléments
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

// 4. Établir les pointeurs d'attributs de sommets
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ::: Affichage (dans la boucle de rendu) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

L'exécution du programme doit donner l'image ci-dessous. Le rectangle vide est constitué de deux triangles.



Le mode fil de fer (wireframe)

Pour tracer un triangle en mode fil de fer, on peut configurer la façon dont OpenGL trace ses primitives, via `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. Le premier argument indique que nous voulons appliquer cela à la face avant et à la face arrière, le second argument que nous souhaitons tracer que les bords. Les appels suivants seront tracés en mode fil de fer, à moins que l'on spécifie l'autre mode, celui par défaut, avec

`glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)`.

Si vous avez réussi à tracer un triangle ou un rectangle comme indiqué, félicitations, vous avez passé l'une des phases les plus difficiles de l'apprentissage de l'OpenGL moderne. C'est difficile, car des connaissances poussées sont nécessaires avant de pouvoir afficher un premier triangle. Les tutoriels suivants seront plus faciles à aborder.

VI-J - Ressources supplémentaires

- antongerdelan.net/hellotriangle : explication d'Anton Gerdelan sur l'affichage du premier triangle.
- open.gl/dessin : explication d'Alexander Overvoorde sur l'affichage du premier triangle.
- antongerdelan.net/vertexbuffers : quelques explications supplémentaires sur les VBO.
- learnopengl.com/#!In-Practice/Debugging : ce tutoriel contient beaucoup d'étapes. Si vous êtes bloqué, vous voudrez peut-être lire comment déboguer une application OpenGL (du moins jusqu'à ce qu'on arrive à la section d'affichage de débogage).

VI-K - Exercices

Afin de bien appréhender les concepts précédents, nous proposons quelques exercices. On vous conseille de les traiter avant d'aborder la suite, pour vous assurer d'avoir bien assimilé cette partie.

- 1 Essayer de tracer deux triangles l'un à côté de l'autre en utilisant `glDrawArrays()` et en ajoutant d'autres sommets à vos données. **solution.**
- 2 Créer maintenant ces mêmes deux triangles en utilisant deux VAO et deux VBO différents. **solution.**
- 3 Créer deux program shader, le second program shader utilisant un fragment shader différent qui affiche un triangle jaune. Afficher ces deux triangles dont l'un est en jaune. **solution.**

VI-L - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

VII - Les shaders

Les shaders sont de petits programmes qui sont chargés dans le GPU. Ces programmes traitent chacun une étape spécifique du pipeline. Les shaders sont des programmes indépendants qui ne communiquent entre eux qu'au moyen de leurs entrées/sorties.

Dans le tutoriel précédent, nous avons approché ces shaders et leur utilisation. Nous allons ici les détailler et en particulier le langage OpenGL Shading Language (GLSL).

VII-A - GLSL

Les shaders sont écrits en langage GLSL, qui ressemble au C. Le langage est conçu pour le traitement graphique, et en particulier la manipulation des vecteurs et matrices.

Les shaders commencent avec une déclaration de version, suivie d'une liste de variables d'entrées et de variables de sortie, des déclarations de variables uniformes (avec le mot-clef `uniform`), puis la fonction `main()`. `main()` est le point d'entrée du shader et traite les entrées pour obtenir les sorties. Passons pour le moment sur les déclarations des variables uniformes.

Un shader a la structure suivante :

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
    // Traitement des entrées et calculs graphiques divers
    ...
    // Préparation de sorties
    out_variable_name = résultat des traitements pour les sorties;
}
```

Les variables d'entrées du vertex shader sont appelées attributs de sommets. Le nombre maximum d'attributs de sommets est limité par le matériel. OpenGL garantit au minimum la place pour 16 attributs de sommets 4D, mais certains matériels peuvent en offrir plus, ce que l'on peut vérifier avec `GL_MAX_VERTEX_ATTRIBS`:

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes << std::endl;
```

Cet appel retourne en général la valeur 16, ce qui est suffisant la plupart du temps.

VII-A-1 - Les types

Le GLSL offre des types de données pour spécifier quel genre de variable nous voulons représenter. On dispose ainsi des types C les plus courants : `int`, `float`, `double`, `uint` et `bool`. Le GLSL offre aussi deux types de structures couramment utilisés : les vecteurs et les matrices.

VII-A-2 - Les vecteurs

Un vecteur GLSL est une structure possédant 1, 2, 3 ou 4 composantes pour chacun des types simples possibles. Cela prend la forme suivante (n représente le nombre de composantes) :

- `vecn` : vecteur de n réels (`float`).
- `bvecn` : vecteur de n booléens.
- `ivec`n : vecteur de n entiers.
- `uvecn` : vecteur de n entiers non signés.
- `dvecn` : vecteur de n réels (`double`).

La plupart du temps, on utilisera le type basique `vecn`, car les réels (`float`) sont suffisants pour la plupart des applications.

Chaque composante d'un vecteur peut être accédée par `vec.x`, `vec.y`, `vec.z` ou `vec.w`. Le GLSL permet d'utiliser aussi les couleurs définies en `rgba` et les coordonnées des textures en `stpq`, accédant de la même façon aux quatre composantes.

Le type vecteur permet une sélection intéressante et flexible des composantes appelée *swizzling*. Le swizzling permet la syntaxe suivante :

```
vec2 someVec;  
vec4 differentVec = someVec.xyxx;  
vec3 anotherVec = differentVec.zyw;  
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

On peut utiliser toute combinaison des quatre lettres pour créer un nouveau vecteur (du même type) tant que le vecteur d'origine possède ces composantes. On ne peut pas accéder à la composante `z` d'un `vec2` par exemple. On peut aussi passer des vecteurs en argument à d'autres constructeurs de vecteurs, en réduisant le nombre d'arguments requis :

```
vec2 vect = vec2(0.5, 0.7);  
vec4 result = vec4(vect, 0.0, 0.0);  
vec4 otherResult = vec4(result.xyz, 1.0);
```

Le type vecteur est donc un type très flexible que l'on peut utiliser pour tous les types d'entrées/sorties. Nous verrons dans les exemples comment utiliser ce type de façon créative.

VII-A-3 - Entrées et sorties

Les shaders sont des petits programmes indépendants mais font partie d'un tout, il faut donc gérer les entrées et les sorties pour être compatibles avec leur environnement. Le GLSL utilise les mots-clés `in` et `out` à cet effet, pour définir les entrées et les sorties respectivement. Chaque fois qu'une variable de sortie correspond à une entrée du shader suivant, les données sont transmises d'un shader au suivant. Le vertex shader et le fragment shader sont cependant légèrement différents.

Le vertex shader doit recevoir des entrées sinon il ne servirait à rien, mais le vertex shader est quelque peu particulier, car ses entrées proviennent du tampon de données. Pour préciser comment les données du vertex shader sont organisées, nous spécifions les variables d'entrée avec la métadonnée **location** qui permet de configurer les attributs de sommets sur le CPU. Nous avons rencontré cela dans le tutoriel précédent avec `layout (location = 0)`.

Le vertex shader requiert ainsi une spécification supplémentaire pour ses entrées de façon à pouvoir les relier aux données des sommets.



Il est aussi possible d'omettre la spécification `layout (location = 0)` et de rechercher l'emplacement des attributs avec `glGetAttribLocation()`, mais on préférera les préciser directement dans le vertex shader, c'est plus facile à comprendre.

L'autre exception concerne le fragment shader qui requiert une variable de sortie `vec4` pour la couleur, car le fragment shader doit générer une couleur finale. Si l'on ne spécifie pas de couleur, OpenGL fera un rendu noir (ou blanc) de l'objet.

Pour transmettre les données d'un shader au suivant, nous devons déclarer les sorties de l'un de façon similaire aux entrées du suivant. Lorsque ces entrées et sorties correspondent (noms et types), OpenGL relie ces entrées et sorties, ce qui permet aux données de passer d'un shader au suivant (cela est réalisé lors de l'édition des liens du shader). Pour montrer cela en pratique, nous allons modifier les shaders du tutoriel précédent pour laisser le vertex shader décider de la couleur du triangle.

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // La variable position a l'attribut de position 0

out vec4 vertexColor; // Nous définirons la couleur dans cette variable

void main()
{
    gl_Position = vec4(aPos, 1.0); // un vec3 est utilisé pour construire un vec4
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // Couleur rouge foncé
}
```

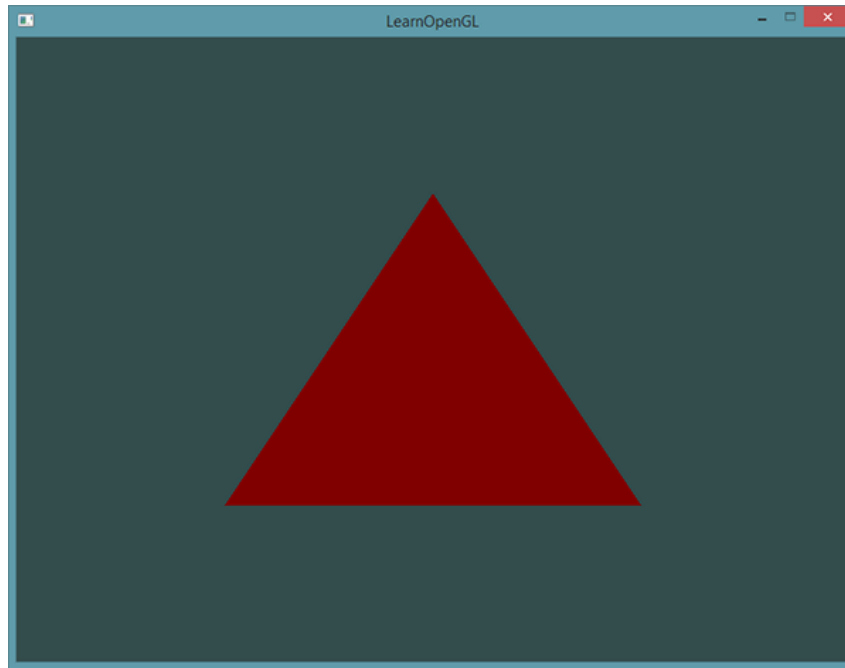
Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // Variable d'entrée identique à la sortie du vertex shader

void main()
{
    FragColor = vertexColor;
}
```

On peut remarquer que la variable `vertexColor` est déclarée comme `vec4` dans le vertex shader et de la même façon dans le fragment shader (nom et type). Ainsi ces deux variables sont liées et la couleur définie dans le vertex shader peut être utilisée dans le fragment shader, pour dessiner l'image suivante :



Compliquons un peu et voyons comment définir la couleur dans l'application pour ensuite la passer au fragment shader !

VII-A-4 - Variables uniformes

Les variables uniformes offrent un autre moyen de passer les données de notre application sur le CPU vers les shaders sur le GPU, mais les variables uniformes sont un peu différents des attributs de sommets. Tout d'abord, les variables uniformes sont globales, ce qui signifie qu'elles sont accessibles dans tout le shader et qu'elles ne peuvent être déclarées qu'une seule fois. De plus, les variables uniformes conservent leur valeur jusqu'à modification ou réinitialisation.

Pour déclarer une variable uniforme en GLSL, on la précise simplement avec le mot-clé `uniform`. Dès lors on peut utiliser la variable dans le shader. Utilisons cela pour déclarer la couleur :

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // nous affecterons cette variable dans le code OpenGL.

void main()
{
    FragColor = ourColor;
}
```

Nous déclarons une variable uniforme `vec4 ourColor` dans le fragment shader et assignons la sortie du fragment shader avec cette variable. Puisque cette variable est globale, on peut la déclarer dans n'importe quel shader, inutile d'y faire référence dans le fragment shader puisqu'on ne l'utilise pas dans celui-ci.



Si l'on déclare une variable uniforme sans l'utiliser, le compilateur l'enlèvera sans prévenir, ce qui peut être la cause d'erreurs difficiles à détecter.

La variable uniforme est pour l'instant non affectée. Pour ce faire, nous avons besoin de trouver son emplacement dans le shader. Après cela, nous pourrions lui affecter une valeur. Au lieu de passer simplement une couleur, modifions la couleur en fonction du temps :

```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Tout d'abord, nous obtenons la date courante en secondes avec `glfwGetTime()`. Ensuite, nous faisons varier l'intensité de la couleur verte avec la fonction sinus.

Puis, nous cherchons l'emplacement de la variable globale `ourColor` avec `glGetUniformLocation()`, en donnant le nom de cette variable et le shader considéré. Si `glGetUniformLocation()` retourne -1, cette variable n'a pas été trouvée. Enfin, nous initialisons cette variable avec la fonction `glUniform4f`. Notons que retrouver l'emplacement de la variable ne requiert pas d'utiliser le shader, mais par contre l'affectation de cette variable ne peut se faire que si l'on utilise effectivement le shader (avec `glUseProgram()`).

Le noyau d'OpenGL étant écrit en C, il ne supporte pas la surcharge de type, ainsi lorsqu'une fonction peut être utilisée avec des types différents, OpenGL définit une fonction par type de paramètre. `glUniform()` en est un bon exemple. Cette fonction requiert de préciser le type de la variable à affecter, grâce au suffixe :

- *f* : la fonction attend un float
- *i* : la fonction attend un int
- *ui* : la fonction attend un unsigned int
- *3f* : la fonction attend 3 float
- *fv* : la fonction attend un tableau de float

Chaque fois que vous souhaitez configurer une option dans OpenGL, choisissez la fonction qui correspond à votre type de variable. Dans notre cas, nous affectons quatre float indépendamment, et utilisons donc `glUniform4f()` (on aurait aussi pu utiliser la version `fv`).

Sachant comment affecter la valeur de la variable globale, on peut s'en servir dans la boucle de rendu, pour modifier la couleur de rendu à chaque itération. Nous calculons la couleur en fonction de la date, et ceci à chaque passage dans la boucle de rendu :

```
while(!glfwWindowShouldClose(window))
{
    // Entrées
    processInput(window);

    // rendu
    // effacement du tampon des couleurs
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // activation du program shader
    glUseProgram(shaderProgram);

    // Mise à jour de la couleur
    float timeValue = glfwGetTime();
    float greenValue = sin(timeValue) / 2.0f + 0.5f;
    int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

    // Rendu du triangle
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // échange des tampons et vérification des entrées utilisateurs
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Ce code découle simplement de la version précédente. Vous devriez voir la couleur du triangle changer graduellement du vert au noir.

Cliquer sur ce lien pour lancer l'animation

En cas de problème, le code est ici : [ici](#).

Comme on le voit, les variables uniformes sont pratiques pour modifier des attributs dans la boucle de rendu, permettant le transfert de données de l'application vers les shaders. Mais que faire si l'on souhaite avoir une couleur pour chaque sommet ? Nous pourrions avoir une variable uniforme pour chaque sommet, mais une meilleure solution est d'inclure des données supplémentaires dans les attributs de sommets ce que nous allons montrer dans la suite.

VII-B - Plus d'attributs

Nous avons vu dans le tutoriel précédent comment initialiser un VBO, configurer les pointeurs d'attributs de sommets et les mémoriser dans le VAO. Maintenant, nous voulons aussi ajouter une couleur aux données des sommets. Nous allons ajouter la couleur avec trois réels dans le tableau des sommets. Nous ajoutons une couleur différente pour chacun des trois sommets :

```
float vertices[] = {
    // positions      // colors
    0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, // bottom left
    0.0f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f  // top
};
```

Puisque nous devons fournir plus de données au vertex shader, nous devons ajuster le vertex shader pour y mémoriser la couleur comme attribut de sommets. Nous spécifions l'emplacement de l'attribut `aColor` avec un spécificateur `layout` :

```
#version 330 core
layout (location = 0) in vec3 aPos; // la variable aPos a l'attribut de position 0
layout (location = 1) in vec3 aColor; // la variable aColor a l'attribut de position 1

out vec3 ourColor; // transmettre une couleur au fragment shader

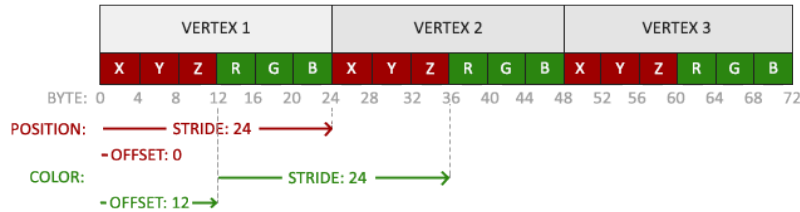
void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // affecter ourColor avec l'entrée color issue des données vertex
}
```

Nous n'utilisons plus de variable uniforme pour la couleur mais plutôt la variable de sortie `ourColor`, et nous devons aussi modifier le fragment shader.

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

Puisque nous avons ajouté un autre attribut de sommet et mis à jour la mémoire du VBO, il faut reconfigurer les pointeurs d'attributs de sommets. Les données du VBO en mémoire suivront ce schéma :



Connaissant l'emplacement en cours, on peut mettre à jour le format des vertex avec `glVertexAttribPointer()` :

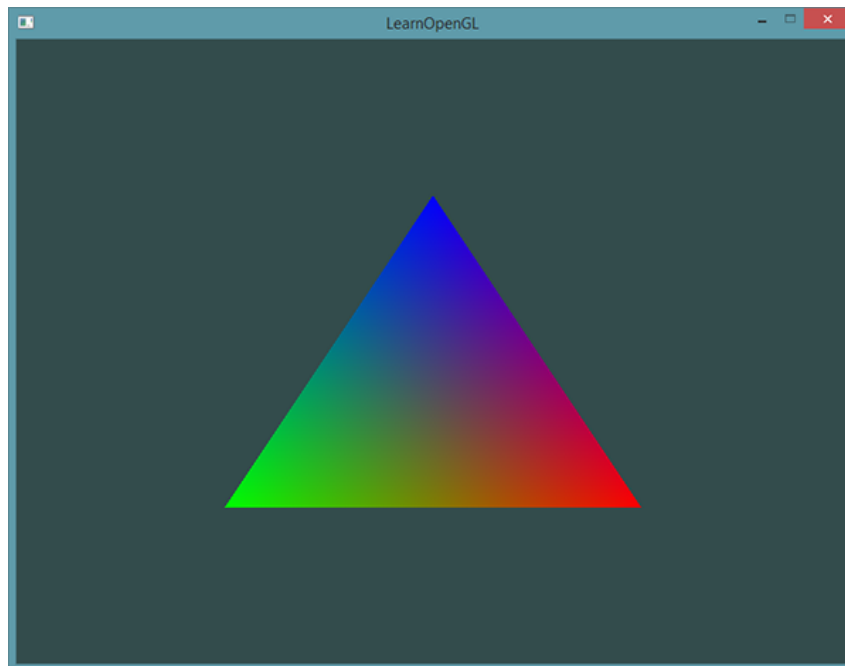
```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Les premiers arguments de `glVertexAttribPointer()` sont assez évidents. Cette fois, nous configurons les attributs de sommets de l'emplacement 1. Les couleurs ont une taille de 3 floats et nous ne normalisons pas les valeurs.

Puisqu'il y a deux attributs de sommets, nous devons recalculer le **stride**. Chacun des attributs de couleur est séparé du suivant de la taille de 6 floats (3 pour la position et 3 pour la couleur).

Cette fois nous devons également spécifier un décalage. La première position est à l'adresse 0, mais la première couleur est à l'adresse $3 * \text{sizeof(float)}$ en octets (= 12 octets).

Le résultat est le suivant :



Vérifiez le code [ici](#) en cas de pépin.

L'image n'est pas exactement ce que vous attendiez, puisque nous n'avons précisé que trois couleurs et non toute la palette que nous voyons. C'est le résultat de l'interpolation effectuée par le fragment shader. Lors du rendu, l'étape de rasterization produit bien plus de fragments que les sommets spécifiés au départ. Le rasterizer détermine la position de chacun de ces fragments en fonction de la forme du triangle. Ensuite, il interpole toutes les variables d'entrée du

fragment shader. Supposons que nous ayons une ligne partant d'un sommet vert et finissant sur un sommet bleu, un fragment situé à 70 % de la distance des deux sommets aura une couleur avec 70 % de vert et 30 % de bleu.

C'est ce qui se passe pour le triangle entier qui contient environ 50 000 fragments, la couleur de chacun étant une interpolation de la couleur des trois sommets définis. Cette interpolation est appliquée pour tous les attributs des données entrées.

VII-C - Notre propre classe shader

Écrire, compiler et gérer les shaders peut devenir fastidieux. Pour finir sur les shaders, nous allons construire une classe shader dont le code se trouve dans des fichiers texte, le compiler, le lier et vérifier les erreurs ; cela pour un usage plus simple des shaders.

Cela pourra vous donner aussi une idée de comment encapsuler les connaissances vues jusqu'ici dans des objets abstraits.

Nous créons la classe `Shader` entièrement dans un fichier d'en-tête, surtout pour des raisons pédagogiques et de portabilité. Commençons par ajouter les include et définissons la structure de la classe :

```
#ifndef SHADER_H // Évite d'inclure plusieurs fois ce fichier
#define SHADER_H

#include <glad/glad.h> // inclure glad pour disposer de tout en-tête OpenGL

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
    // the program ID
    unsigned int ID;

    // le constructeur lit et construit le shader
    Shader(const GLchar* vertexPath, const GLchar* fragmentPath);
    // Activation du shader
    void use();
    // fonctions utiles pour l'uniform
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};

#endif
```

i Nous avons utilisé plusieurs directives pour le préprocesseur au début de notre fichier. Ces petites lignes de code imposent au compilateur de n'inclure et compiler ce fichier d'en-tête que si cela n'a pas déjà été fait, évitant ainsi les inclusions multiples de ce fichier `shader.h`, et donc des erreurs de compilation.

La classe `Shader` contient l'identifiant du program shader. Le constructeur requiert le chemin d'accès au fichier contenant le code source du vertex shader et du fragment shader que nous aurons placés dans un fichier texte. Nous avons également prévu quelques fonctions pour nous simplifier la vie : `use()` active le program shader et les fonctions `set...` récupèrent l'emplacement des variables uniformes et les initialisent.

VII-C-1 - Lire dans les fichiers

Nous utilisons les flux de fichiers C++ (filestreams) pour lire le contenu du fichier dans des chaînes de caractères (string) :

```
Shader(const char* vertexPath, const char* fragmentPath)
{
    // 1. récupère le code du vertex/fragment shader depuis filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // s'assure que les objets ifstream peuvent envoyer des exceptions:
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        // ouverture des fichiers
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // lecture des fichiers et place le contenu dans des flux
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // fermeture des fichiers
        vShaderFile.close();
        fShaderFile.close();
        // conversions des flux en string
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch(std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char* fShaderCode = fragmentCode.c_str();
    [...]
```

VII-C-2 - Compilation

Ensuite nous devons compiler et lier les shaders. Notons que nous vérifions aussi les erreurs de compilation et d'édition des liens en affichant le cas échéant les erreurs, ce qui est très important pour déboguer.

```
// 2. compiler les shaders
unsigned int vertex, fragment;
int success;
char infoLog[512];

// vertex shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// affiche les erreurs de compilation si besoin
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
};

// de même pour le fragment shader
[...]
```

```
// program shader
ID = glCreateProgram();
glAttachShader(ID, vertex);
```

```
glAttachShader(ID, fragment);
glLinkProgram(ID);
// affiche les erreurs d'édition de liens si besoin
glGetProgramiv(ID, GL_LINK_STATUS, &success);
if(!success)
{
    glGetProgramInfoLog(ID, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}

// supprime les shaders qui sont maintenant liés dans le programme et qui ne sont plus
nécessaires
glDeleteShader(vertex);
glDeleteShader(fragment);
```

VII-C-3 - Fonctions

La fonction `use()` est très simple :

```
void use()
{
    glUseProgram(ID);
}
```

De la même façon les fonctions `set...` :

```
void setBool(const std::string &name, bool value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}
void setInt(const std::string &name, int value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}
void setFloat(const std::string &name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

VII-C-4 - Utilisation

Dès lors, nous avons une **classe complète**. Utiliser cette classe est très facile, nous créons un objet `Shader` et nous l'utilisons :

```
Shader ourShader("path/to/shaders/shader.vs", "path/to/shaders/shader.fs");
...
while(...)
{
    ourShader.use();
    ourShader.setFloat("someUniform", 1.0f);
    DrawStuff();
}
```

Ici, nous plaçons le code des shaders dans deux fichiers appelés *shader.vs* et *shader.fs*. Vous pouvez les appeler comme vous vous voudrez. J'utilise les extensions *.vs* et *.fs* pour plus de clarté.

Vous trouverez code source [ici](#), utilisé par la **classe Shader**.

VII-D - Exercices

1 Modifier le vertex shader pour inverser le triangle (bas en haut) : **solution**.

- 2 Spécifier un décalage horizontal avec une variable uniforme et déplacer le triangle vers la droite de la fenêtre en utilisant ce décalage dans le vertex shader : **solution**.
- 3 Faire passer la position des sommets au fragment shader par le mot-clé out et définir la couleur du fragment égal à la position du sommet (voir comment les valeurs position sont interpolées dans le triangle). Ceci réalisé, une question : pourquoi le côté en bas à gauche du triangle est-il noir ? **Solution**.

VII-E - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

VIII - Textures

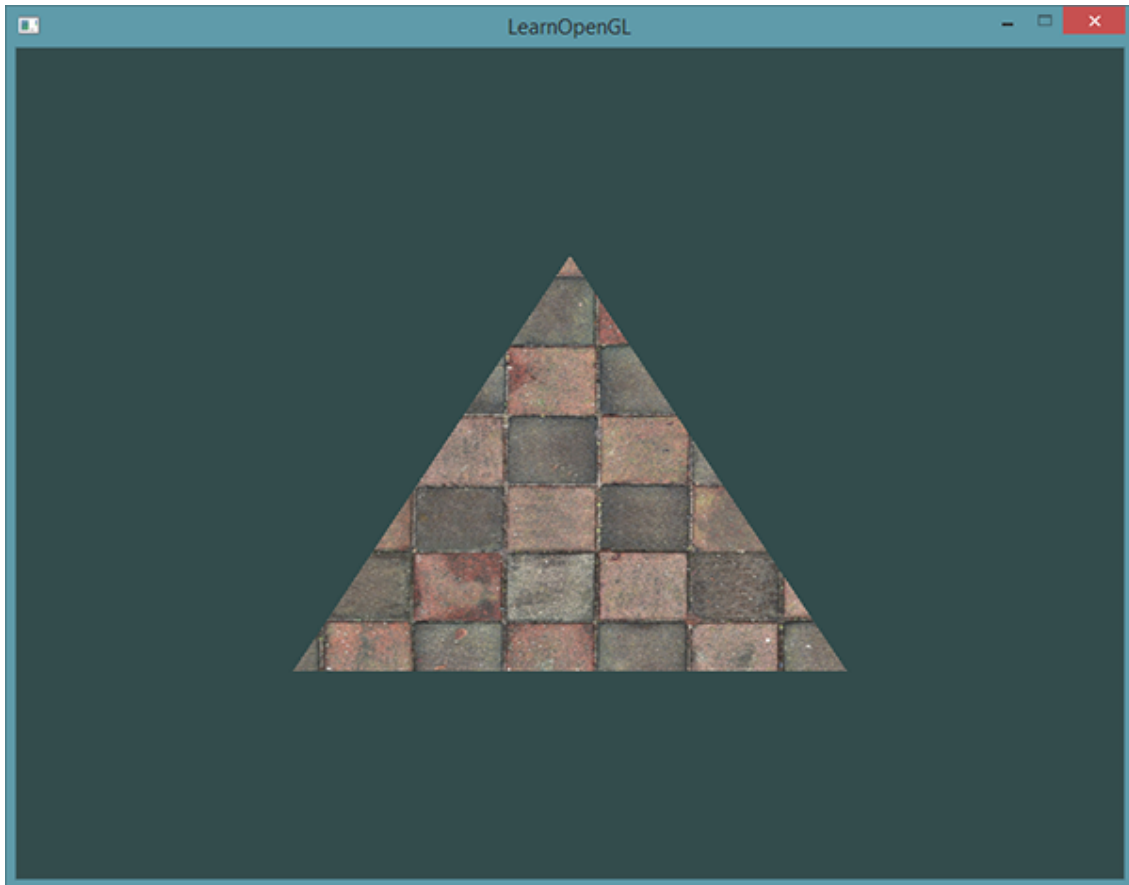
Nous avons vu comment utiliser la couleur pour chaque sommet, cependant pour des applications plus réalistes, nous aurons à traiter de nombreux sommets et de nombreuses couleurs, ce qui deviendra vite ingérable dans une scène réelle.

Les artistes et les programmeurs utilisent en général ce qu'on appelle une texture. Il s'agit d'une image 2D (les textures 1D et 3D existent aussi) que l'on peut voir comme un papier peint, par exemple l'image d'une brique, qui sera plaquée sur le modèle 3D d'une maison, ce qui donnera l'image d'un mur en briques. Une image permet de représenter des choses très variées et détaillées, ce qui donne l'illusion que l'objet 3D est lui-même très détaillé, sans pour autant ajouter des sommets.



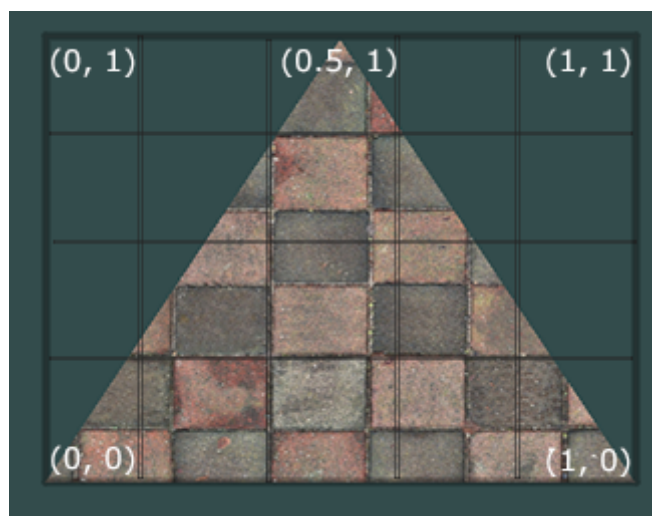
Les textures peuvent aussi être utilisées pour archiver de grandes quantités de données à transmettre aux shaders, mais nous verrons cela plus loin.

Ci-dessous, l'**image d'un mur de briques** projetée (map) sur notre triangle :



Pour appliquer une texture au triangle, nous devons préciser à quel sommet du triangle correspond quelle partie de la texture. Chaque sommet doit avoir une coordonnée de texture associée qui spécifie quel point de la texture lui correspond, l'interpolation entre fragments fera le reste pour les autres fragments.

Les coordonnées de texture sont dans l'intervalle $[0.0, 1.0]$ sur les axes x et y (pour une texture en 2D). Retrouver la couleur de la texture en utilisant les coordonnées texture s'appelle l'échantillonnage (sampling). $(0, 0)$ est le point en bas à gauche tandis que $(1, 1)$ est en haut à droite de l'image, comme le montre l'image ci-dessous :



Le coin en bas à gauche du triangle correspondra au point en bas à gauche de la texture ; de la même façon pour le point en bas à droite. Le point en haut du triangle correspond au point $(0.5, 1.0)$ de la texture. Nous passerons ces points au vertex shader, qui les passera au fragment shader qui interpolera ensuite les coordonnées pour tous les fragments.

Les coordonnées de la texture auront donc cette forme :

```
float texCoords[] = {
    0.0f, 0.0f, // côté bas gauche
    1.0f, 0.0f, // côté bas droit
    0.5f, 1.0f // côté haut
};
```

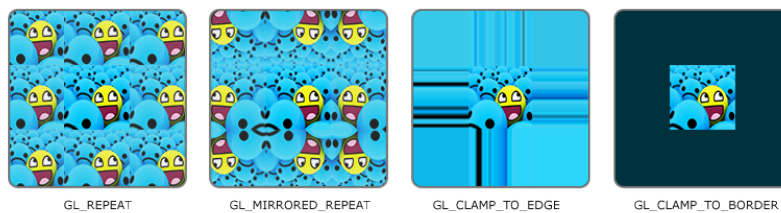
L'échantillonnage (sampling) peut se faire de différentes façons, il faudra donc préciser à OpenGL comment le faire.

VIII-A - Texture Wrapping

Que se passe-t-il si on donne des coordonnées de texture en dehors de l'intervalle [0, 1] ? le comportement par défaut d'OpenGL est de répéter les images de texture (on ignore la partie entière des réels représentant les coordonnées de la texture), mais d'autres options sont possibles :

- **GL_REPEAT**: par défaut, répétition de l'image.
- **GL_MIRRORED_REPEAT**: pareil que **GL_REPEAT** mais l'image est inversée à chaque répétition.
- **GL_CLAMP_TO_EDGE**: limite les coordonnées entre 0 et 1. les coordonnées plus grandes seront limitées aux bords, résultant en une forme étendue jusqu'aux bords.
- **GL_CLAMP_TO_BORDER**: les coordonnées en dehors de l'intervalle seront remplies avec une couleur spécifiée par l'utilisateur.

On peut voir sur les images ci-dessous le résultat de chacune de ces options :



Chacune de ces options peut être choisie pour chacun des axes de coordonnées (s, t (r si texture 3D)), avec la fonction `glTexParameter*` :

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Le premier argument spécifie la cible, ici **GL_TEXTURE_2D**. Le second argument spécifie l'option et l'axe à traiter (ici **WRAP** pour s et t). Le dernier argument donne le mode à utiliser, ici **GL_MIRRORED_REPEAT**.

Si on choisit l'option **GL_CLAMP_TO_BORDER**, il faudra indiquer aussi une couleur, en utilisant la version fv (float vector) de la fonction `glTexParameter()`. Nous passerons un vecteur de réels pour indiquer la couleur :

```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

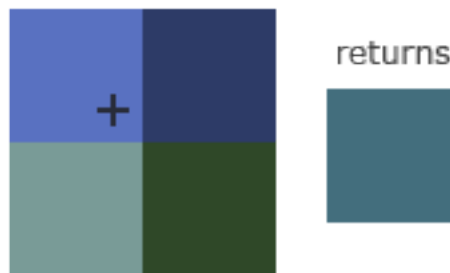
VIII-B - Interpolation de texture (texture filtering)

Les coordonnées de texture ne dépendent pas de la résolution, mais peuvent prendre toute valeur réelle, OpenGL doit donc décider quel pixel de la texture (appelé texel) doit être projeté sur le point considéré. Cela devient essentiel si vous avez un grand objet et une texture de faible résolution. OpenGL offre aussi des options pour cela. Nous présenterons les plus importantes : **GL_NEAREST** et **GL_LINEAR**.

`GL_NEAREST` (aussi appelée *nearest neighbour filtering*) est la méthode par défaut. Dans ce cas OpenGL choisit le pixel dont le centre est le plus proche de la coordonnée de texture. Dans l'exemple ci-dessous, la croix représente la coordonnée de texture ; le texel en haut à gauche est le plus proche de la croix, il est donc utilisé comme couleur :



`GL_LINEAR` (aussi appelée interpolation bilinéaire, *bilinear filtering*) choisit une valeur résultant de l'interpolation des texels voisins, donnant une couleur approchante. La distance des texels voisins donne le poids correspondant de la couleur dans le résultat final :



Quel est le résultat visuel de l'interpolation de texture ? Voyons cela avec une texture de faible résolution sur un grand objet (la texture est étirée et les texels sont visibles) :



`GL_NEAREST` donne un résultat où l'on peut clairement distinguer les pixels de la texture, tandis que `GL_LINEAR` produit une image plus réaliste, mais certains développeurs préféreront un aspect 8-bits et choisiront l'option `GL_NEAREST`.

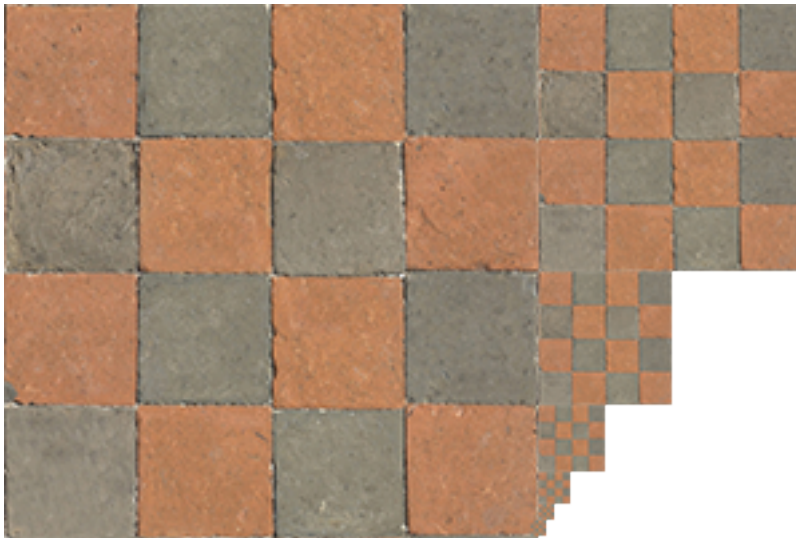
L'interpolation de texture peut être choisie pour les opérations de grossissement (*magnifying*) ou de réduction (*minifying*), indépendamment l'une de l'autre. Il faut spécifier l'option pour chacune des deux opérations, avec `glTexParameter()` :

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

VIII-B-1 - Mipmaps

Supposons que nous ayons une grande scène avec des milliers d'objets, chacun ayant une texture. Certains objets seront loin et auront une texture de grande résolution comme les objets qui sont au premier plan. Ces objets éloignés ne sont composés que de peu de fragments et OpenGL aura du mal à déterminer la bonne couleur pour ces fragments à partir d'une texture haute résolution. Cela produira des défauts visuels sur les petits objets, sans compter le gaspillage de la mémoire lié à une haute résolution inutile.

Pour traiter ce problème, OpenGL propose le concept des mipmaps : un ensemble de textures dans lequel chaque texture est deux fois plus petite que la précédente. Si l'objet se trouve plus loin qu'un certain seuil, OpenGL utilisera une texture plus petite, adaptée à l'éloignement de l'objet, sans que cela ne soit perceptible par le spectateur. On améliore aussi les performances. Voyons une texture mipmap :



Créer un ensemble de textures mipmap est fastidieux, mais OpenGL se charge de ce travail au moyen de l'appel `glGenerateMipmaps()` à effectuer après la création de la texture. Nous verrons cela plus loin dans le tutoriel.

Lorsque l'on change de niveaux de mipmaps pendant le rendu, OpenGL affiche certains effets gênants comme des bords tranchants entre deux niveaux. Complétant l'interpolation de texture, il est aussi possible de choisir la méthode d'interpolation entre les niveaux de mipmap :

- `GL_NEAREST_MIPMAP_NEAREST`: choisit la mipmap la plus proche pour la taille du pixel et utilise le voisin le plus proche pour la couleur du texel.
- `GL_LINEAR_MIPMAP_NEAREST`: choisit la mipmap la plus proche pour la taille du pixel et utilise un filtrage bilinéaire pour la couleur.
- `GL_NEAREST_MIPMAP_LINEAR`: interpolation linéaire entre les niveaux de mipmap et utilise le voisin le plus proche pour la couleur du texel.
- `GL_LINEAR_MIPMAP_LINEAR`: interpolation linéaire entre les niveaux de mipmap et utilise un filtrage bilinéaire pour la couleur.

Pour choisir la méthode d'interpolation, on a le choix entre les quatre méthodes, avec le suffixe `i` :

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Une erreur courante est de choisir une de ces options d'interpolation pour le grossissement des textures. Cela n'a pas d'effet puisque les mipmaps sont utiles lorsque les textures sont réduites en taille. La mise à l'échelle des textures n'utilise pas les mipmaps et fournir une option de mipmap donnera un code d'erreur OpenGL `GL_INVALID_ENUM`.

VIII-C - Charger et créer les textures

La première chose à faire pour utiliser une texture est de la charger dans l'application. Les images peuvent être mémorisées dans de nombreux formats, chacun ayant sa propre structure. Une solution serait de choisir un format comme *.png* et de convertir notre image en un grand tableau d'octets. Il faudrait alors écrire un module de chargement pour chaque format.

La meilleure solution est plutôt d'utiliser une bibliothèque de chargement qui supporte la plupart des formats, comme *stb_image.h*.

VIII-C-1 - stb_image.h

stb_image.h est un fichier en-tête très populaire, écrit par **Sean Barrett**. Ce fichier fait le travail et est facile à intégrer dans vos projets : on le trouve [ici](#). Téléchargez ce fichier et ajoutez-le à vos projets. Puis ajoutez le code suivant dans un nouveau fichier C++ :

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

Avec la définition de `STB_IMAGE_IMPLEMENTATION`, le préprocesseur modifie le fichier d'en-tête de façon à ce qu'il ne contienne que le code source approprié, modifiant le fichier d'en-tête en fichier *.cpp*. Il suffit ensuite d'inclure *stb_image.h* dans votre programme et de compiler.

Dans la suite, nous utilisons l'image d'un **conteneur en bois**. Pour charger l'image, nous utilisons la fonction `stbi_load()` :

```
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

On précise le chemin d'accès au fichier image, la fonction renvoyant trois entiers : la largeur de l'image, la hauteur et le nombre de canaux de couleur.

VIII-C-2 - Générer une texture

Comme tout objet dans OpenGL, une texture est accessible avec un identifiant :

```
unsigned int texture;
glGenTextures(1, &texture);
```

Cette fonction demande de préciser le nombre de textures à créer et archive leurs identifiants dans un tableau d'`unsigned int`. Comme nous l'avons déjà vu, nous devons attacher la texture et ainsi les commandes suivantes seront exécutées sur cette texture :

```
glBindTexture(GL_TEXTURE_2D, texture);
```

Lorsqu'une texture est attachée, nous pouvons initialiser cette texture avec l'image chargée, au moyen de la fonction `glTexImage2D()` :

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

Examinons les paramètres de cette fonction :

- Le premier argument spécifie la cible : `GL_TEXTURE_2D` implique que la texture visée sera la texture attachée à la cible 2D (les textures ayant pour cible `GL_TEXTURE_1D` ou `GL_TEXTURE_3D` ne seront pas affectées).
- Le second argument spécifie le niveau de mipmap pour lequel on souhaite créer la texture si l'on voulait établir chaque niveau de mipmap manuellement, mais ici nous laissons cela au niveau de base, soit 0.
- Le troisième argument précise le format d'archivage de la texture. Notre image n'a que des valeurs RGB, nous choisissons donc `GL_RGB`.
- Ensuite, nous passons la largeur et hauteur de l'image.
- L'argument suivant est toujours 0.
- Les deux arguments suivants spécifient le format et le type de données pour l'image source.
- Le dernier argument pointe sur les données effectives de l'image.

Après cet appel, la texture en cours possède une image qui lui est liée. Cependant, seul le niveau de base de l'image lui est attaché, et si nous voulons utiliser les mipmaps, nous devons spécifier manuellement les images, mais il est plus simple de générer automatiquement tous les mipmaps requis, avec `glGenerateMipmap()`.

Après cela, il est bien vu de libérer l'espace mémoire contenant l'image :

```
stbi_image_free(data);
```

Le code complet pour générer une texture va donc ressembler à ceci :

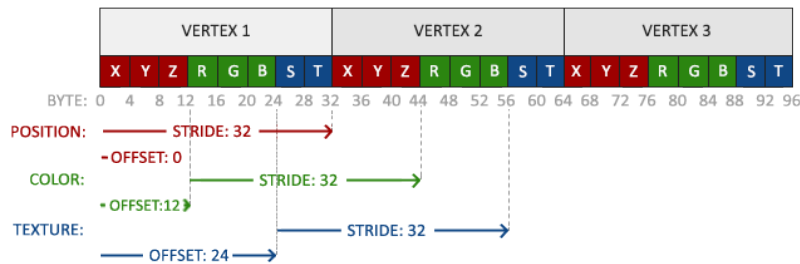
```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// définit les options de la texture actuellement liée
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// charge et génère la texture
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);
```

VIII-D - Appliquer les textures

Pour la suite, nous utiliserons le rectangle construit à la fin du chapitre **Hello Triangle**. Nous devons informer OpenGL sur la manière d'échantillonner la texture, pour cela nous ajoutons les coordonnées de la texture aux données de sommets :

```
float vertices[] = {
    // positions           // colors           // texture coords
    0.5f,  0.5f,  0.0f,    1.0f,  0.0f,  0.0f,    1.0f,  1.0f,    // top right
    0.5f, -0.5f,  0.0f,    0.0f,  1.0f,  0.0f,    1.0f,  0.0f,    // bottom right
    -0.5f, -0.5f,  0.0f,    0.0f,  0.0f,  1.0f,    0.0f,  0.0f,    // bottom left
    -0.5f,  0.5f,  0.0f,    1.0f,  1.0f,  0.0f,    0.0f,  1.0f    // top left
};
```

Il faut ensuite mettre à jour le format des sommets :



```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);
```

Notons qu'il faut ajuster le stride à la valeur $8 * \text{sizeof(float)}$ pour les deux autres attributs (position et couleur).

Il faut ensuite modifier le vertex shader pour prendre en compte les coordonnées de texture comme attributs de sommets et passer ces coordonnées au fragment shader :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = aTexCoord;
}
```

Le fragment shader prendra la variable de sortie TexCoord en entrée.

Le fragment shader doit aussi pouvoir accéder à l'objet texture, mais comment lui passer cet objet ? GLSL possède un type de donnée pour les textures appelé sampler (échantillonneur), précisant en suffixe de quel type de texture il s'agit : sampler1D, sampler3D ou comme ici sampler2D. On peut ainsi ajouter une texture au fragment shader en déclarant simplement une variable uniforme sampler2D et nous y placerons ensuite notre texture.

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```

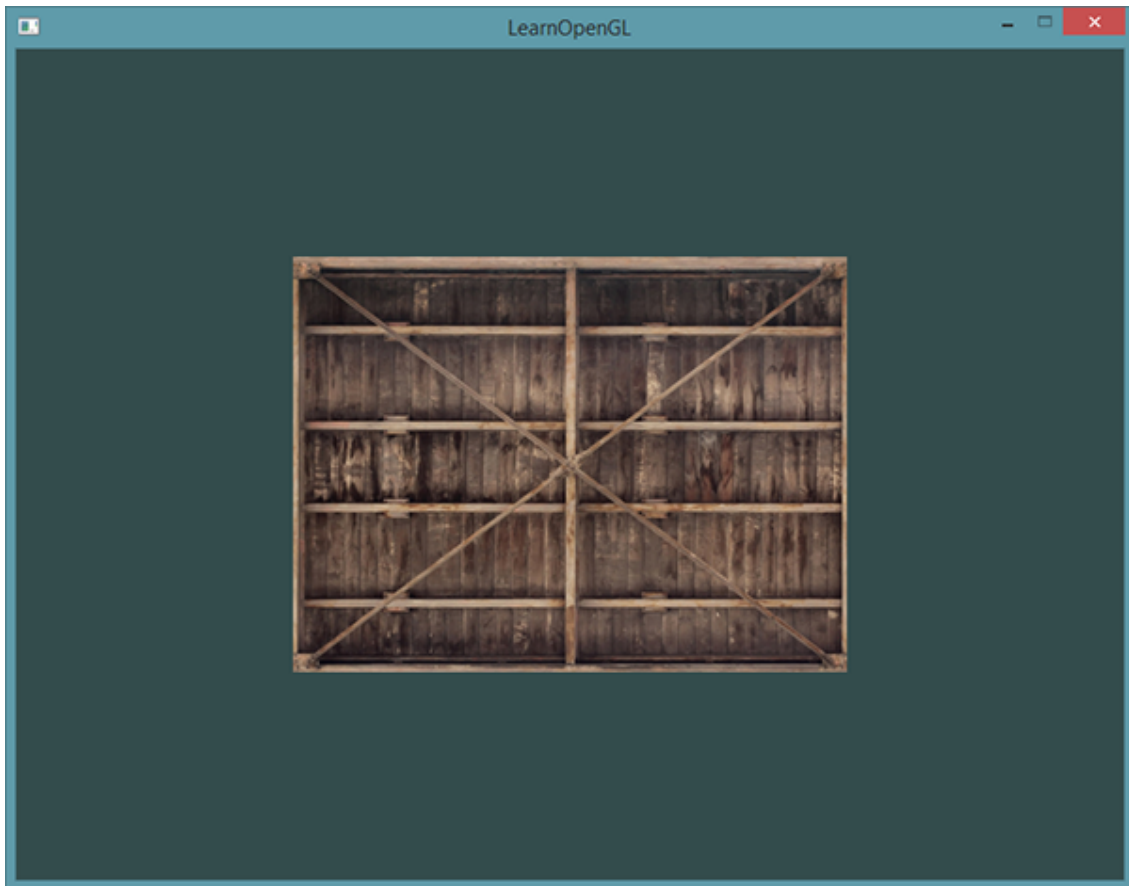
Pour échantillonner la couleur de la texture, nous utilisons la fonction GLSL texture() qui prend en premier argument un sampler et en second argument les coordonnées. La sortie du fragment shader sera ensuite déterminée par la texture.

Il reste à attacher la texture avant d'appeler glDrawElements(), cela assignera automatiquement la texture au sampler du fragment shader :

```
glBindTexture(GL_TEXTURE_2D, texture);
```

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Si tout a été fait correctement, voilà le résultat :



En cas de problème, vous pouvez consulter le code : [source code](#).

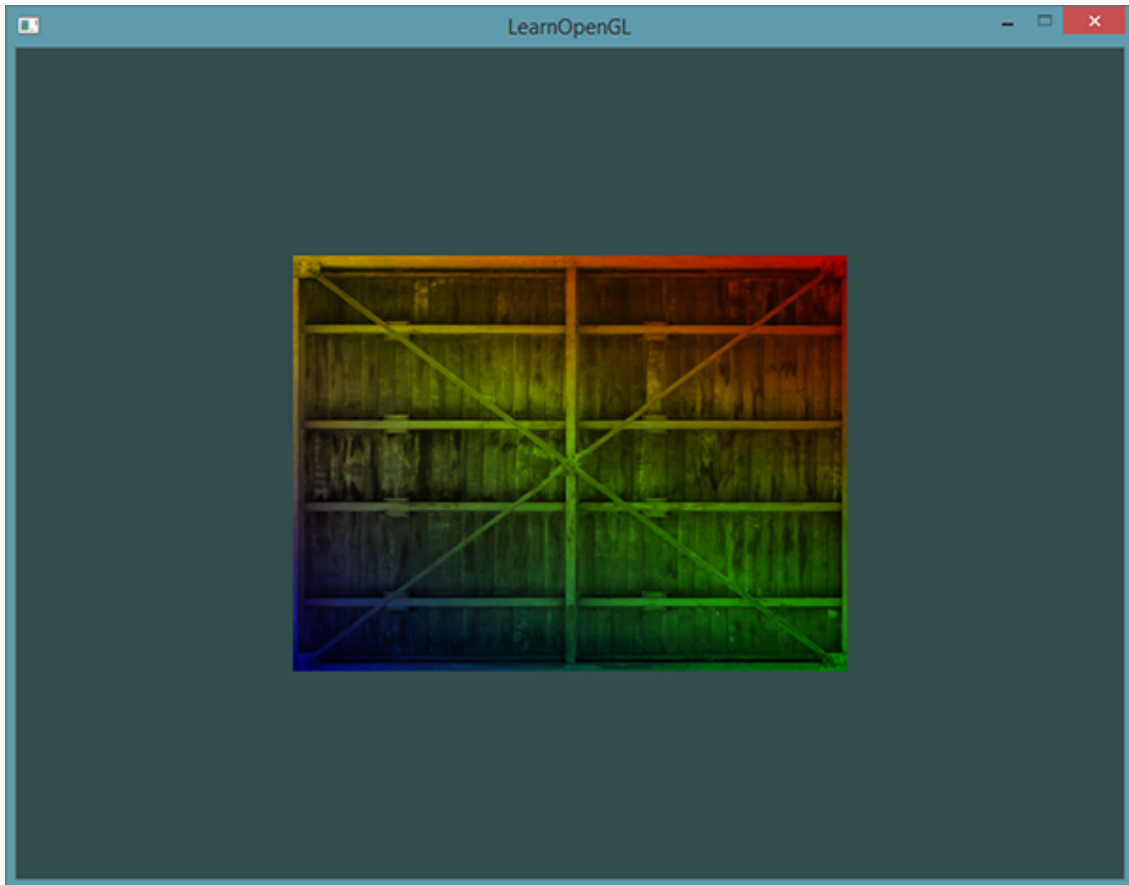


*Si votre programme ne fonctionne pas ou affiche un objet noir, continuez votre lecture et travaillez le dernier exemple qui **doit** fonctionner. Avec certains pilotes graphiques, il est nécessaire d'utiliser une unité de texture pour chaque variable uniforme de type sampler, ce que nous allons voir maintenant.*

Pour obtenir une image plus folklo, on peut mixer la texture avec les couleurs des sommets. On multiplie simplement la couleur de la texture avec la couleur du sommet dans le fragment shader :

```
FragColor = texture(ourTexture, TexCoord) * vec4(ourColor, 1.0);
```

Le résultat devrait être un mélange entre la couleur du sommet et la couleur de la texture :



VIII-E - Unités de texture

On peut se demander pourquoi la variable `sampler2D` est une variable uniforme, et que l'on ne lui a pas assigné de valeur. En utilisant `glUniform1i()`, on peut assigner un emplacement (location) au sampler pour utiliser plusieurs textures à la fois dans un fragment shader. L'emplacement d'une texture est appelé unité de texture (texture unit). L'unité par défaut est 0, qui est l'unité active par défaut, ce qui explique que nous ne l'avons pas spécifiée jusqu'ici ; tous les pilotes n'assignent pas de valeur par défaut, ce qui peut entraîner des dysfonctionnements si on ne traite pas ce point.

Le but principal des unités de texture est de permettre l'utilisation de plusieurs textures dans les shaders. En assignant une unité aux samplers, on peut attacher plusieurs textures à la fois, du moment que l'on active l'unité de texture correspondante juste avant. Il suffit de passer l'unité en paramètre de `glActiveTexture()` :

```
glActiveTexture(GL_TEXTURE0); // active l'unité de texture avant la liaison de la texture
glBindTexture(GL_TEXTURE_2D, texture);
```

Après avoir activé l'unité de texture, nous attachons la texture voulue à l'unité courante de texture. L'unité `GL_TEXTURE0` est toujours activée par défaut, nous n'avons donc pas à le faire dans le code précédent avant d'utiliser `glBindTexture()`.



OpenGL permet de disposer d'un minimum de 16 unités de texture, que l'on peut activer en utilisant `GL_TEXTURE0` jusqu'à `GL_TEXTURE15`. On peut accéder à ces objets par une opération d'addition : `GL_TEXTURE8` est identique à `GL_TEXTURE0 + 8` par exemple, ce qui est utile en cas d'itération.

Il faut cependant éditer le fragment shader pour accepter un autre sampler, ce qui assez direct :

```
#version 330 core
...

uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

La couleur finale sera ici un mélange de deux textures. La fonction `mix` de GLSL prend deux valeurs en entrée et interpole linéairement entre elles en fonction du troisième argument, qui indique le poids relatif de la seconde texture. 0.0 ne conserve que la première texture, 0.2 donnerait 80 % pour la première et 20 % pour la seconde.

Nous voulons charger et créer une deuxième texture ; nous utilisons ici une image de votre **expression faciale lorsque vous apprenez OpenGL**.

Pour utiliser cette nouvelle texture avec la première, nous devons modifier un peu la procédure en attachant les deux textures à leur unité :

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);

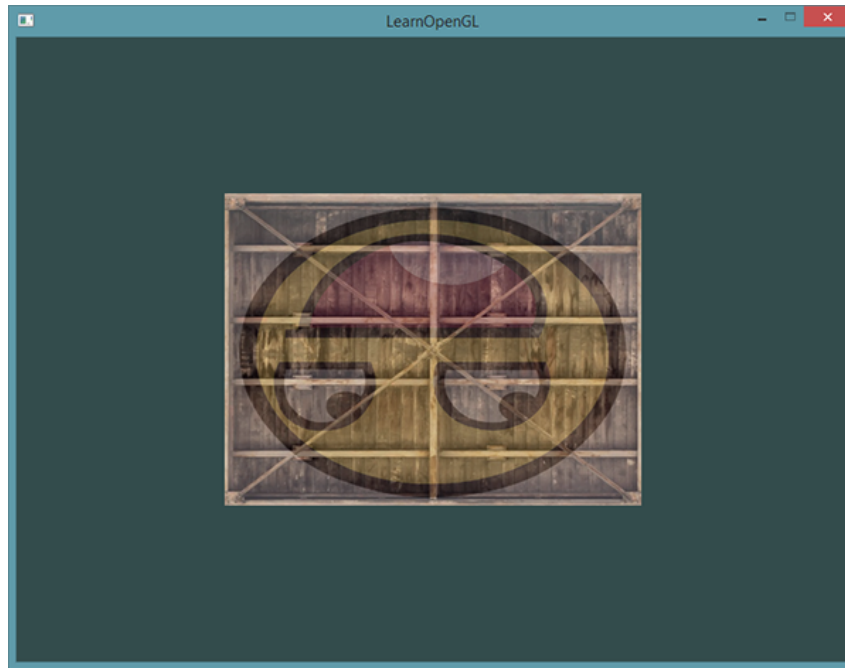
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Nous devons également dire à OpenGL à quelle unité de texture appartient chaque sampler, au moyen de `glUniform1i()`. Il suffit de faire cela une seule fois, on peut donc le faire avant la boucle de rendu :

```
ourShader.use(); // n'oubliez pas d'activer le shader avant de définir les variables uniformes
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0); // définition manuelle
ourShader.setInt("texture2", 1); // ou avec la classe de shader

while(...)
{
    [...]
}
```

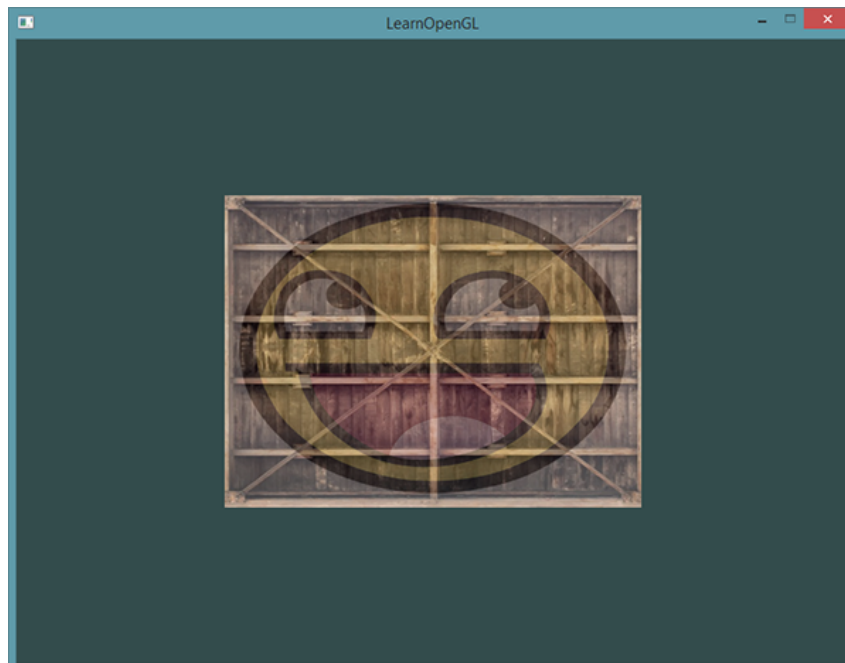
Le résultat est le suivant :



Vous avez remarqué que la texture est affichée à l'envers. Cela est dû à ce qu'OpenGL positionne le sens de l'axe Y vers le haut alors que les images sont codées avec le sens de l'axe Y vers le bas. *stb_image.h* permet d'inverser le sens de l'axe Y pendant le chargement de l'image, il faut lui préciser de cette façon avant le chargement :

```
stbi_set_flip_vertically_on_load(true);
```

Le résultat est alors le suivant :



En cas de problème le code est ici : [source](#).

VIII-F - Exercices

Pour mieux comprendre les textures, nous vous conseillons ces exercices :

- Faire en sorte que **seule** l'image « happy face » soit inversée en modifiant le fragment shader : **solution**.
- Essayer les différentes méthodes de wrapping en spécifiant les coordonnées de texture dans l'intervalle [0.0, 2.0]. Essayer d'afficher quatre smileys sur un seul container limité à son bord : **solution**, **resultat**. Essayer d'autres méthodes de wrapping.
- Essayer d'afficher seulement les pixels du centre de l'image sur le rectangle de façon à ce que les pixels deviennent visibles, cela en changeant les coordonnées de texture. Essayer la méthode d'échantillonnage `GL_NEAREST` pour mieux voir les pixels : **solution**.
- Utiliser une variable uniforme comme troisième paramètre de la fonction mix, pour faire varier le poids relatif de chaque texture. Utiliser les touches haut et bas pour afficher plus ou moins le smiley : **solution**.

VIII-G - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

IX - Transformations

On sait maintenant comment créer des objets, les colorer ou leur donner une apparence avec une texture, mais cela reste des objets statiques et donc peu intéressants. On pourrait essayer de les faire bouger en modifiant les sommets et en reconfigurant les tampons à chaque rendu, mais ce serait pénible et peu efficace. Il y a mieux à faire pour transformer un objet, en utilisant une (ou plusieurs) matrice de transformation.

Les matrices sont des outils mathématiques puissants qui peuvent effrayer au début, mais qui se révèlent très pratiques et utiles ensuite. Nous allons devoir parler un peu de maths, et pour les lecteurs qui souhaiteraient approfondir le sujet, je conseillerai des lectures additionnelles.

Pour bien comprendre les transformations, il nous faut présenter les vecteurs avant de parler des matrices. Le but de ce chapitre est de vous donner quelques bases de maths sur les sujets qui nous seront nécessaires par la suite. En cas de difficultés, essayez d'en comprendre le plus possible et revenez-y plus tard en cas de besoin.

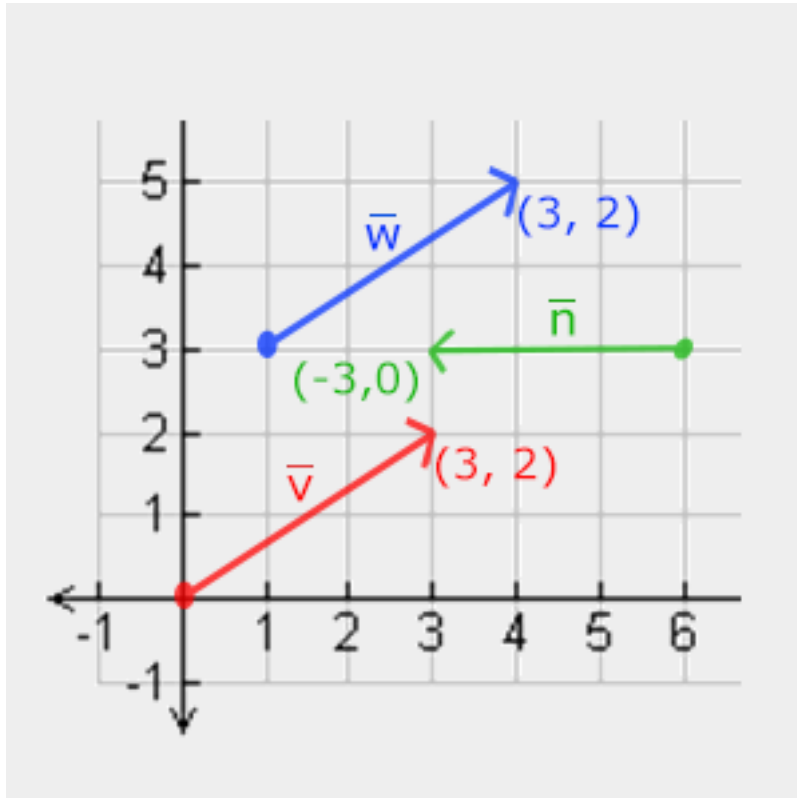
IX-A - Les vecteurs

Dans leur définition la plus basique, les vecteurs sont des directions, rien de plus. Un vecteur a une direction et une longueur (appelée aussi norme).

Par exemple, vous pouvez imaginer un vecteur comme une direction sur une carte au trésor : aller 10 pas vers la gauche, puis 3 pas vers le nord, puis 3 pas vers la droite. Ici « gauche » est la première direction, et « 10 pas » est la longueur du premier vecteur. Le chemin vers le trésor est constitué de 3 vecteurs.

Les vecteurs peuvent avoir n'importe quelle dimension, mais nous travaillerons en dimension 2, 3 ou 4. Si un vecteur est de dimension 2, il représente une direction dans un plan (2D) ; de dimension 3 il représente une direction dans l'espace (3D).

Ci-dessous sont représentés trois vecteurs où chacun est figuré avec une flèche et ses coordonnées. Vous pouvez imaginer un vecteur 2D comme un vecteur 3D ayant la coordonnée $z = 0$. Puisque ce sont des directions, l'origine d'un vecteur ne change pas sa valeur. Ici les vecteurs \vec{v} et \vec{w} sont les mêmes, bien qu'ils n'aient pas la même origine :



Pour nommer les vecteurs, les mathématiciens utilisent des flèches qu'ils placent au-dessus de la lettre pour bien montrer qu'il s'agit d'un vecteur. Par exemple si x, y, z sont les coordonnées du vecteur \vec{v} :

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Puisque les vecteurs représentent des directions, il est quelquefois difficile de les visualiser à une certaine position. Ce que l'on fait souvent consiste à les placer avec l'origine en $(0, 0, 0)$ et ensuite de les faire pointer dans la direction représentée, vers un point de coordonnées identiques à celles du vecteur. Le vecteur $(3, 2)$ pointe vers le point $(3, 2)$ si l'on place son origine en $(0, 0)$, comme le vecteur v de la figure précédente. On peut donc utiliser les vecteurs pour décrire une direction, mais aussi une position, en 2D ou en 3D.

Comme avec les nombres, on peut définir plusieurs opérations sur les vecteurs (nous avons déjà rencontré certaines opérations dans les chapitres précédents).

IX-B - Opérations entre vecteur et scalaire

Un scalaire est un simple nombre. On peut effectuer les quatre opérations $(+, -, *, /)$ entre un vecteur et un scalaire, consistant à faire l'opération donnée avec chacune des composantes de ce vecteur :

Pour l'addition par exemple :

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x = \begin{pmatrix} 1 + x \\ 2 + x \\ 3 + x \end{pmatrix}$$

On peut utiliser les quatre opérations, mais attention, pour $-$ et $/$, l'ordre est important, seules la soustraction et la division par un scalaire sont possibles (on ne peut pas diviser un scalaire par un vecteur).

IX-C - Négation

La négation d'un vecteur consiste à inverser sa direction en changeant de signe chacune de ses composantes (ou coordonnées). Cela revient à le multiplier par -1 :

$$-\vec{v} = - \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

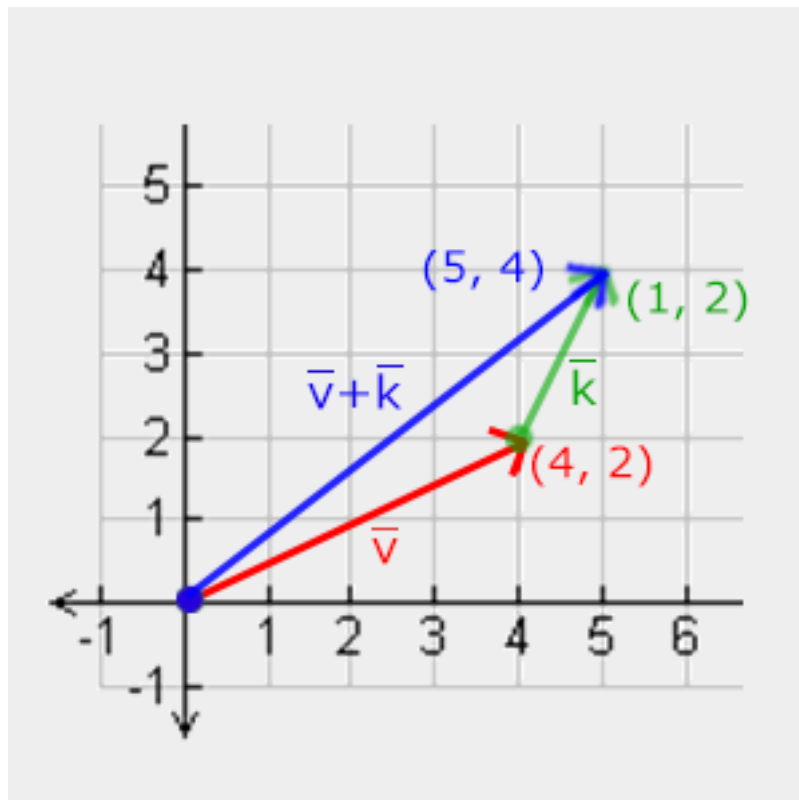
IX-D - Addition et soustraction entre vecteurs

L'addition de deux vecteurs consiste à additionner leurs composantes respectives :

$$\vec{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \vec{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \vec{v} + \vec{k} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

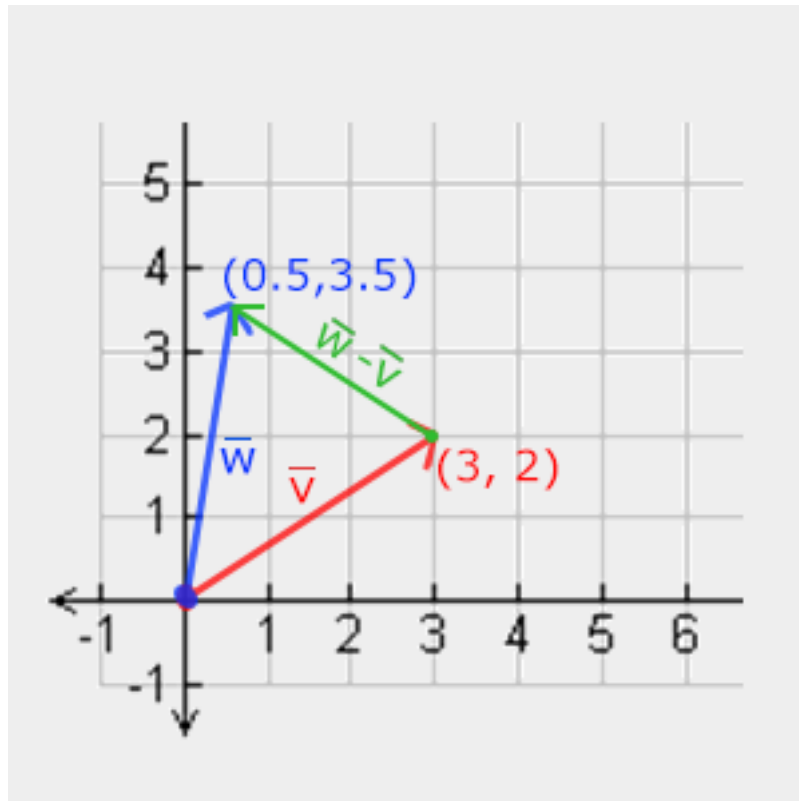
Visuellement, cela se représente comme dans la figure ci-dessous :

$$\vec{v} = (4, 2) \text{ et } \vec{k} = (1, 2) \text{ donc } \vec{v} + \vec{k} = (5, 4)$$



La soustraction suit le même procédé, mais la figure est différente. Le résultat est le vecteur qui va de l'extrémité d'un vecteur à l'extrémité de l'autre :

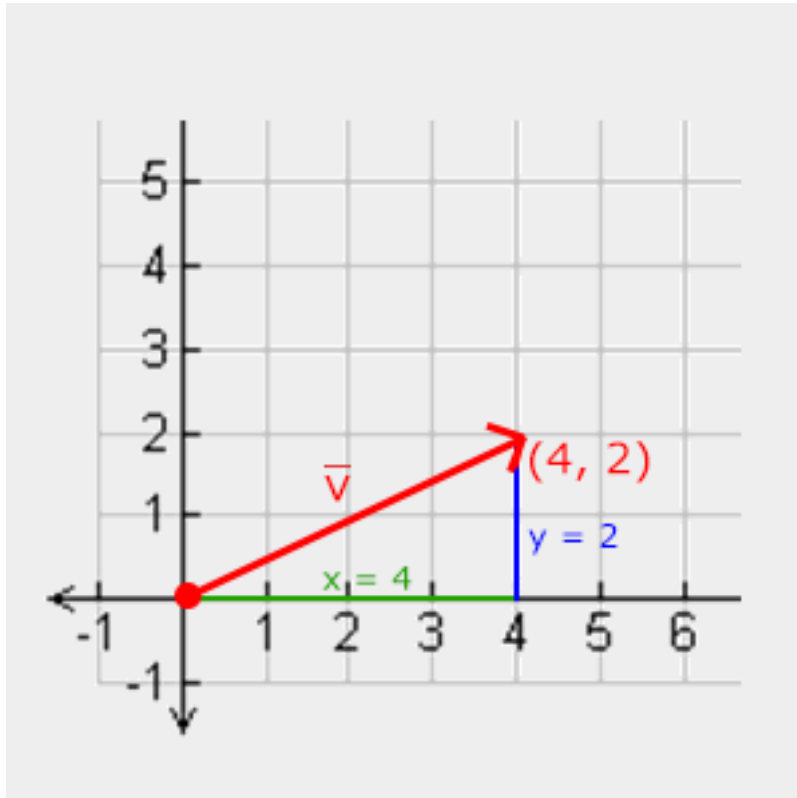
$$(0.5, 3.5) - (3, 2) = (-2.5, 1.5)$$



La différence entre deux vecteurs position peut s'avérer très utile pour calculer la direction définie par deux points de l'espace.

IX-E - Longueur

Pour retrouver la longueur d'un vecteur, on peut utiliser le théorème de Pythagore. Un vecteur de coordonnées (x, y) forme l'hypoténuse d'un triangle rectangle, les deux autres côtés ayant pour longueur x et y :



$$||\vec{v}|| = \sqrt{x^2 + y^2}$$

Où $||\vec{v}||$ est la longueur (ou norme) de \vec{v} .

Cela s'étend facilement au cas 3D en ajoutant la composante z à la formule.

Dans cet exemple, la longueur du vecteur (4, 2) se calcule ainsi :

$$||\vec{v}|| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$$

Un cas particulier de vecteur est le **vecteur unitaire** : sa longueur est égale à 1. On peut définir un vecteur unitaire dans la même direction que \vec{v} , simplement en le divisant par sa longueur. On appelle cela normaliser un vecteur. On utilise souvent ces vecteurs unitaires pour indiquer seulement une direction (la direction d'un vecteur ne change pas si l'on ne modifie que sa longueur).

IX-F - Produit de vecteurs

Multiplier deux vecteurs peut paraître étrange, mais on peut en fait définir deux opérations appelées l'une produit scalaire noté $\vec{v} \cdot \vec{k}$ et l'autre produit vectoriel noté $\vec{v} \times \vec{k}$

IX-F-1 - Produit scalaire

Le produit scalaire de deux vecteurs est égal au produit de leur longueur multiplié par le cosinus de l'angle qu'ils forment entre eux. L'angle étant noté θ , le produit scalaire est :

$$\vec{v} \cdot \vec{k} = ||\vec{v}|| \cdot ||\vec{k}|| \cdot \cos \theta$$

Ce produit est très utile. Supposons que \vec{v} et \vec{k} soient deux vecteurs unitaires, le produit scalaire vaut alors simplement $\cos(\theta)$. Cette valeur vaut 1 si l'angle vaut 0, et vaut 0 si l'angle vaut 90°. Cela permet par exemple de tester si deux vecteurs sont parallèles (0°) ou orthogonaux (90°). Pour en savoir plus sur les fonctions sin et cos, vous pouvez suivre les cours de la **Khan Academy** sur la trigonométrie.



On peut aussi s'en servir pour calculer l'angle entre deux vecteurs quelconques en divisant le produit scalaire par le produit des longueurs, on obtient alors $\cos(\theta)$.

Le produit scalaire se calcule simplement en multipliant entre elles les composantes des deux vecteurs et en ajoutant ces produits. Par exemple :

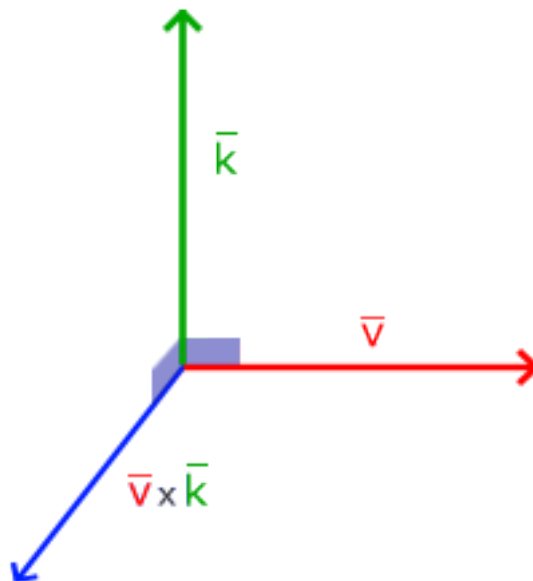
$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

Pour calculer l'angle, on calcule la longueur de chaque vecteur, qui dans cet exemple valent 1, ce qui signifie que $\cos(\theta) = -0.8$ et donc $\theta = 143$ degrés. Le produit scalaire est très utilisé en maths et en physique.

IX-F-2 - Produit vectoriel

Le résultat du produit vectoriel (utilisé en 3D) est un vecteur qui est orthogonal aux deux vecteurs considérés. Si ces deux vecteurs sont orthogonaux, ils formeront avec le vecteur produit un ensemble de trois vecteurs orthogonaux, ce qui nous sera très utile dans la suite.

L'image suivante donne un aperçu du résultat :



Le calcul des composantes du produit vectoriel n'est pas évident, et pour ne pas alourdir l'exposé, nous donnerons seulement la formule permettant de calculer ce produit vectoriel :

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$

Cette formule n'est pas très parlante, il faut surtout retenir que le résultat donne un vecteur orthogonal aux deux autres (si les deux vecteurs ne sont pas colinéaires, car dans ce cas, le produit vectoriel est le vecteur nul).

IX-G - Matrices

Nous avons présenté rapidement les vecteurs, voyons maintenant les matrices. Une matrice est basiquement un tableau rectangulaire de nombres. Chaque nombre est appelé élément de la matrice, voici un exemple de matrice 2x3 (2 lignes et 3 colonnes, ce sont les dimensions de la matrice) :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Les éléments sont repérés par l'indice de leur colonne et de leur ligne, on attribue l'indice 0 à la première ligne (en haut) et à la première colonne (à gauche). Dans cet exemple, $M[1, 0] = 4$.

Les matrices ne sont rien de plus que des tableaux de nombres, mais possèdent des propriétés mathématiques très utiles. On peut définir certaines opérations sur les matrices, parmi lesquelles l'addition, la soustraction et la multiplication.

IX-G-1 - Addition et soustraction

On peut ajouter un nombre (scalaire) à une matrice en ajoutant ce nombre à chacun des éléments de la matrice :

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 3 = \begin{bmatrix} 1+3 & 2+3 \\ 3+3 & 4+3 \end{bmatrix} = \begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix}$$

Idem pour la soustraction :

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} - 3 = \begin{bmatrix} 1-3 & 2-3 \\ 3-3 & 4-3 \end{bmatrix} = \begin{bmatrix} -2 & -1 \\ 0 & 1 \end{bmatrix}$$

L'addition et la soustraction entre matrices se fait élément par élément. Ces opérations n'étant possibles que pour deux matrices de mêmes dimensions :

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Le calcul est identique pour une soustraction :

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

IX-G-2 - Produit par un scalaire

Dans ce cas, on multiplie chaque élément de la matrice par le scalaire :

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

On peut noter que cette opération est en réalité une mise à l'échelle (ici une multiplication par 2), d'où l'origine du mot « scalaire » (scale en anglais).

IX-G-3 - Multiplication de deux matrices

Cette opération n'est pas très complexe mais peut apparaître comme peu évidente. On multiplie les éléments entre eux selon certaines règles. Cette opération exige cependant une condition sur leurs dimensions : on ne peut multiplier deux matrices que si le nombre de colonnes de la matrice de gauche est égal au nombre de lignes de la matrice de droite.

Attention : contrairement aux nombres, l'ordre de la multiplication est important : $A \cdot B \neq B \cdot A$ (opération non commutative).

Prenons un exemple avec deux matrices 2x2 :

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Les éléments de la matrice produit sont le résultat du produit d'une ligne de la matrice de gauche par une colonne de la matrice de droite, comme le montrent les couleurs employées dans la figure :

$$\begin{bmatrix} \boxed{1} & \boxed{2} \\ \boxed{3} & \boxed{4} \end{bmatrix} \cdot \begin{bmatrix} \boxed{5} & \boxed{6} \\ \boxed{7} & \boxed{8} \end{bmatrix} = \begin{bmatrix} \boxed{1 \cdot 5 + 2 \cdot 7} & \boxed{1 \cdot 6 + 2 \cdot 8} \\ \boxed{3 \cdot 5 + 4 \cdot 7} & \boxed{3 \cdot 6 + 4 \cdot 8} \end{bmatrix} = \begin{bmatrix} \boxed{19} & \boxed{22} \\ \boxed{43} & \boxed{50} \end{bmatrix}$$

Pour calculer le premier élément, d'indices (0,0), on utilise la première ligne de la matrice de gauche et la première colonne de la matrice de droite, en multipliant chacun des éléments deux à deux et en effectuant la somme de ces produits. On répète cette opération pour chacun des éléments de la matrice produit. Pour calculer l'élément de la ligne i et de la colonne j, on utilise la ligne i de la matrice de gauche et la colonne j de la matrice de droite.

La matrice produit est une matrice de dimensions (n,m), n étant le nombre de lignes de la matrice de gauche et m le nombre de colonnes de la matrice de droite.

Ne vous inquiétez pas si vous avez quelques difficultés avec ces calculs, essayez juste de les réaliser tranquillement à la main et revenez sur cette page en cas de difficultés. La multiplication des matrices vous deviendra vite familière.

Voici un autre exemple, plus compliqué, avec des matrices 3x3. Essayez de faire les calculs vous-mêmes et comparez vos résultats pour vous habituer à ces calculs :

$$\begin{bmatrix} 4 & 2 & 0 \\ 0 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 0 & 4 \\ 9 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 \cdot 4 + 2 \cdot 2 + 0 \cdot 9 & 4 \cdot 2 + 2 \cdot 0 + 0 \cdot 4 & 4 \cdot 1 + 2 \cdot 4 + 0 \cdot 2 \\ 0 \cdot 4 + 8 \cdot 2 + 1 \cdot 9 & 0 \cdot 2 + 8 \cdot 0 + 1 \cdot 4 & 0 \cdot 1 + 8 \cdot 4 + 1 \cdot 2 \\ 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 9 & 0 \cdot 2 + 1 \cdot 0 + 0 \cdot 4 & 0 \cdot 1 + 1 \cdot 4 + 0 \cdot 2 \end{bmatrix}$$

$$= \begin{bmatrix} 20 & 8 & 12 \\ 25 & 4 & 34 \\ 2 & 0 & 4 \end{bmatrix}$$

Pas d'inquiétude si cela ne vous semble pas simple, il faut surtout savoir à quoi servent ces multiplications de matrices. D'autre part, les bibliothèques mathématiques effectuent ces calculs pour nous. Pour une présentation plus détaillée, voir les vidéos de la **Khan Academy** à propos des matrices.

IX-H - Multiplication d'un vecteur par une matrice

Nous avons déjà utilisé les vecteurs pour représenter des positions, des couleurs et aussi des coordonnées de textures. On peut aussi considérer les vecteurs comme des matrices Nx1, où N est la dimension (3 en 3D), soit des matrices à 1 seule colonne. On peut donc multiplier une matrice par un vecteur de dimension N à condition que cette matrice possède N colonnes.

Mais quel est l'intérêt de cette multiplication ? En fait, nous allons pouvoir transformer notre vecteur au moyen d'une (ou plusieurs) matrices. Voyons ces possibilités en détail.

IX-I - La matrice identité

Dans OpenGL, on utilise en général des matrices 4x4 pour plusieurs raisons, mais essentiellement, car les vecteurs ont généralement quatre composantes. La matrice la plus simple que l'on peut concevoir est la matrice identité, où tous les éléments sont nuls sauf ceux de la première diagonale où tous les éléments valent 1. Si l'on multiplie un vecteur par cette matrice, on laisse le vecteur inchangé :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Le vecteur n'est pas modifié, ce qui devient clair lorsque l'on applique les règles de la multiplication. Pour le premier élément, le calcul est le suivant :

$$1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 0 \cdot 4 = 1$$

et il en est de même pour les autres éléments.

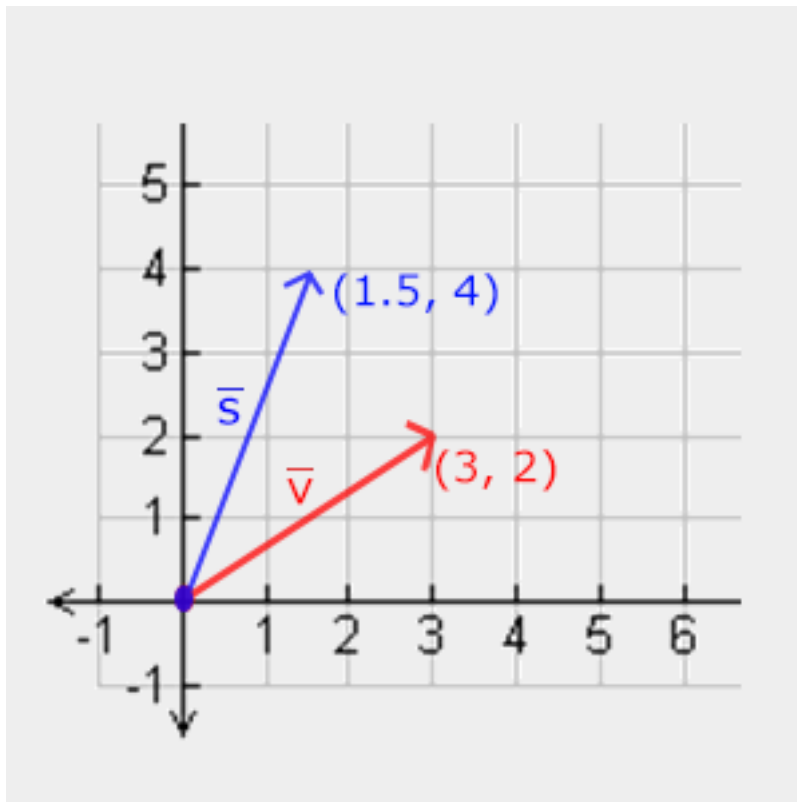


Mais pourquoi utiliser une matrice qui ne modifie pas le vecteur sur lequel elle opère ? En fait, cette matrice identité est souvent un point de départ pour générer d'autres matrices de transformation, et elle joue un rôle fondamental en algèbre linéaire.

IX-J - Matrice de mise à l'échelle d'un vecteur

Mettre un vecteur à une autre échelle consiste à modifier sa longueur sans modifier sa direction, en multipliant le vecteur par un scalaire (voir 8.2). Mais puisque l'on travaille en 2D ou en 3D, on peut utiliser une échelle différente pour chacune des dimensions, et utiliser un scalaire par dimension.

Par exemple, considérons le vecteur $\vec{v} = (3, 2)$. Nous pouvons choisir une mise à l'échelle de 0.5 en x et 2 en y, ce qui le rendra deux fois moins large et deux fois plus haut, pour donner le vecteur $\vec{s} = (1.5, 4)$:



Rappelons-nous qu'OpenGL opère en général en 3D, et donc pour un cas 2D il suffit d'utiliser l'échelle 1 pour la coordonnée z. Si l'on utilise le même scalaire pour chaque dimension, on parle alors de mise à l'échelle uniforme.

Construisons une matrice pour réaliser cette opération. En changeant les valeurs 1 de la matrice identité par les scalaires choisis pour la mise à l'échelle, on réalisera cette mise à l'échelle en multipliant le vecteur par la matrice.

Si les facteurs d'échelle sont définis par les variables (S_1, S_2, S_3) , on peut composer la matrice de mise à l'échelle avec ces valeurs et l'opération de mise à l'échelle sera ainsi définie pour tout vecteur (x, y, z) :

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Notons que la 4e valeur est fixée à 1, car nous ne modifions pas la composante w du vecteur 3D. On utilise w pour d'autres raisons que nous allons examiner dans la suite.

IX-K - Matrice de translation d'un vecteur position

Cette opération consiste à ajouter un autre vecteur au vecteur considéré pour obtenir une nouvelle position. Nous avons déjà rencontré l'addition de vecteurs, cela ne devrait pas vous paraître nouveau.

De la même façon que pour la mise à l'échelle, nous pouvons utiliser une matrice 4x4 pour réaliser cette opération d'addition. Dans ce cas, nous utilisons les trois premières valeurs de la 4e colonne.

Soient (T_x, T_y, T_z) les composantes du vecteur de translation, l'opération de translation se déroulera ainsi :

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Cela fonctionne car la composante w du vecteur est égale à 1, le calcul de la première valeur se déroulant ainsi : $1.x + 0.y + 0.z + T_x.1 = x + T_x$.

On voit l'intérêt de travailler en dimension 4, cette opération n'aurait pas été possible avec une matrice 3x3.

Coordonnées homogènes

La composante w est appelée coordonnée homogène. Pour obtenir le vecteur 3D à partir d'un vecteur homogène, on divise x, y et z par w. La plupart du temps, $w = 1.0$ et cela ne change rien. L'utilisation de coordonnées homogènes présente plusieurs avantages ; elles permettent d'effectuer des translations sur les vecteurs 3D et nous les utiliserons aussi pour créer des effets visuels dans le prochain chapitre.

Si la coordonnée w vaut 0, le vecteur est appelé vecteur de direction, on ne peut pas le traduire.

Avec les matrices de translation, on peut simplement déplacer des objets dans l'espace, cette opération se révèle très utile dans les applications 3D.

IX-L - Matrice de rotation d'un vecteur

Les rotations sont plus difficiles à comprendre. Pour une présentation détaillée, on pourra consulter les vidéos de la Khan Academy sur l'[algèbre linéaire](#).

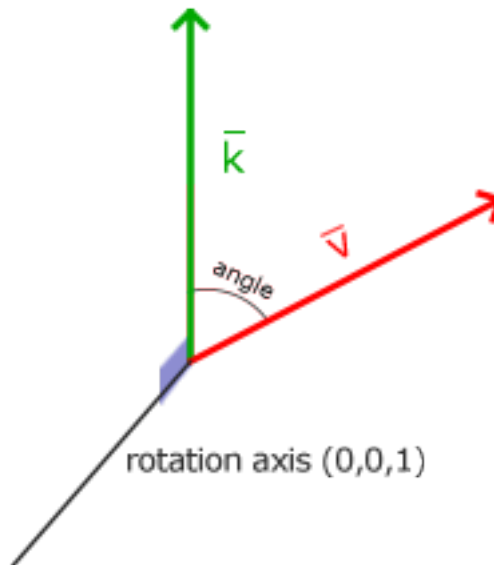
La rotation d'un vecteur consiste à le faire tourner. Une rotation est définie par son axe (autour duquel tourne le vecteur) et l'angle de rotation. Un angle peut être mesuré en radians ou en degrés (un tour entier vaut 360° ou 2π radians). Pour passer d'une mesure à l'autre :

La plupart des fonctions de rotation utilisent des angles définis en radians, mais on passe facilement d'un système à l'autre :

```
angle en degrés = angle en radians * (180.0f / PI)
angle en radians = angle en degrés * (PI / 180.0f)
```

Une rotation d'un demi-tour correspond à un angle de 180° ($360/2$), et une rotation de 1/5e de tour correspond à un angle de 72° ($360/5$).

Un exemple de rotation en 2D est montré dans la figure ci-dessous, le vecteur \vec{v} tourne de 72° autour de l'axe z (dirigé vers nous) pour donner le vecteur \vec{k} .



Les rotations en 3D requièrent de définir un angle mais aussi un axe de rotation. Essayez de visualiser cela en tournant votre tête d'un certain angle vers la droite ou la gauche tout en fixant votre regard vers le bas. Lorsque l'on fait tourner des vecteurs en 2D, l'axe de rotation est l'axe z (essayez de la visualiser).

En utilisant la trigonométrie, il est possible de transformer un vecteur par une rotation d'un angle donné, en utilisant les fonctions sinus et cosinus. Le calcul de ces matrices dépasse le cadre de ce tutoriel. Les matrices de rotation autour des axes x, y et z sont assez simples (θ est l'angle de rotation) :

Rotation autour de l'axe X :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Rotation autour de l'axe Y :

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x + \sin \theta \cdot z \\ y \\ -\sin \theta \cdot x + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Rotation autour de l'axe Z :

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x - \sin \theta \cdot y \\ \sin \theta \cdot x + \cos \theta \cdot y \\ z \\ 1 \end{pmatrix}$$

En utilisant ces matrices, on peut faire tourner un vecteur autour de l'un des trois axes X, Y et Z. On peut aussi les combiner avec plusieurs rotations successives, autour de l'axe X puis de l'axe Y par exemple. Cela introduit par contre le problème du *Gimbal lock*. Nous ne discuterons pas ce point délicat, mais une meilleure solution consiste à définir directement une rotation autour d'un axe quelconque, par exemple (0.662, 0.2, 0.722) – notons que l'axe est défini par un vecteur unitaire – au moyen d'une matrice de rotation. Si l'axe de rotation est défini par ses composantes

(R_x, R_y, R_z) , la matrice de rotation d'un angle θ autour de cet axe sera la suivante :

$$\begin{bmatrix} \cos \theta + R_x^2(1 - \cos \theta) & R_x R_y(1 - \cos \theta) - R_z \sin \theta & R_x R_z(1 - \cos \theta) + R_y \sin \theta & 0 \\ R_y R_x(1 - \cos \theta) + R_z \sin \theta & \cos \theta + R_y^2(1 - \cos \theta) & R_y R_z(1 - \cos \theta) - R_x \sin \theta & 0 \\ R_z R_x(1 - \cos \theta) - R_y \sin \theta & R_z R_y(1 - \cos \theta) + R_x \sin \theta & \cos \theta + R_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La démonstration est hors de propos ici. Cependant, cette matrice ne résout pas complètement le problème du gimbal lock. Le mieux est d'utiliser les quaternions, cela est plus sain, et plus facile à calculer. Nous réservons cela pour un autre tutoriel.

IX-M - Combiner les matrices

La puissance des matrices réside aussi dans la possibilité de combiner plusieurs transformations en une seule matrice grâce à la multiplication des matrices. Voyons cela sur un exemple. Supposons que l'on souhaite mettre à l'échelle 2 un vecteur (x, y, z) et ensuite le translater au moyen du vecteur (1, 2, 3). Nous définissons une matrice de translation et une matrice de mise à l'échelle et nous les multiplions :

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Noter que nous effectuons la multiplication dans un ordre précis : la translation puis le redimensionnement. La première opération qui sera effectuée sur le vecteur sera la mise à l'échelle, et ensuite la translation. On doit lire la multiplication des matrices de droite à gauche ! Il est conseillé d'effectuer d'abord les mises à l'échelle, puis les rotations et enfin les translations.

L'opération globale sur le vecteur donnera :

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x + 1 \\ 2y + 2 \\ 2z + 3 \\ 1 \end{pmatrix}$$

Voilà ! Le vecteur obtenu est d'abord mis à l'échelle, puis translaté. Si l'on inverse la multiplication, vous pourrez vérifier que l'on trouverait $2(x+1) = 2x + 2$ au lieu de $2x+1$...

IX-N - En pratique

Maintenant que nous avons expliqué les transformations en théorie, il est temps de voir comment utiliser ces connaissances. OpenGL n'offre pas d'objets vecteurs ou matrices, nous devons donc définir nous-mêmes ces objets mathématiques au moyen de classes et de fonctions. Nous ne réaliserons pas cette tâche dans ces tutoriels, nous ferons plutôt appel à des bibliothèques mathématiques. L'une d'entre elles est très adaptée à OpenGL, il s'agit de GLM.

IX-O - GLM



GLM est l'acronyme de OpenGL Mathematics, qui est une bibliothèque formée uniquement de fichiers d'en-têtes. Il suffira donc d'inclure ces fichiers d'en-têtes pour l'utiliser. Pas besoin de construire cette bibliothèque. GLM peut être téléchargée sur [ce site](http://glm.g-truc.net/) (version 0.9.8). Copiez le contenu du répertoire principal des fichiers d'en-têtes dans votre propre répertoire *include* et c'est tout.



Depuis la version 0.9.9, GLM initialise par défaut les matrices à 0, au lieu de les initialiser à la matrice identité. Depuis cette version, il est préférable d'initialiser les matrices de cette façon : `glm::mat4 mat = glm::mat4(1.0f);` Pour utiliser le code de nos tutoriels sans modifications, utilisez une version antérieure à 0.9.9, ou bien initialisez toutes les matrices par la matrice identité.

La plupart des fonctionnalités de GLM se trouvent dans seulement trois fichiers d'en-têtes que nous pouvons inclure comme ceci :

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

Voyons si nous pouvons utiliser nos connaissances pour traduire le vecteur (1, 0, 0) au moyen du vecteur (1, 1, 0). Notons que nous définissons notre vecteur avec le type `glm::vec4`, et la coordonnée w égale à 1.0f.

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans;
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

Nous commençons par définir un vecteur de nom `vec`, en utilisant la classe `vector` définie dans GLM. Ensuite nous définissons une matrice 4x4 au moyen du type `mat4`, initialisée à la matrice identité (attention : pour les versions récentes de GLM, utiliser `glm::mat4 trans(1.0);`). L'étape suivante consiste à créer une matrice de transformation à partir de la matrice identité, ce qui est réalisé par la fonction `glm::translate` dont le second paramètre est le vecteur de translation. Ensuite, nous utilisons cette matrice de translation qui ainsi opère sur notre vecteur `vec`, cela au moyen d'une multiplication. Le vecteur résultant devrait avoir comme coordonnées (1+1, 0+1, 0+0), soit (2, 1, 0), ce que l'on vérifiera aisément.

Effectuons maintenant une mise à l'échelle et une rotation de l'objet conteneur du tutoriel précédent :

```
glm::mat4 trans;
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

Nous commençons par une mise à l'échelle uniforme d'un facteur 0.5, puis nous effectuons une rotation de 90° autour de l'axe Z. GLM utilise des angles définis en radians, nous convertissons donc les degrés en radians au moyen de la fonction `radians()`. Notons que le rectangle texturé est dans le plan XY, et que l'axe de rotation est l'axe Z. Nous avons passé la matrice trans en argument de chacune des deux fonctions, le résultat sera donc le produit des deux matrices, combinant ainsi les deux transformations.

La question qui se pose alors : comment passer notre matrice de transformation au shader ? Nous avons vu que le GLSL possède un type `mat4`. Nous allons donc adapter le vertex shader en y intégrant une variable uniforme de type `mat4` et multiplier le vecteur position par cette matrice :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```



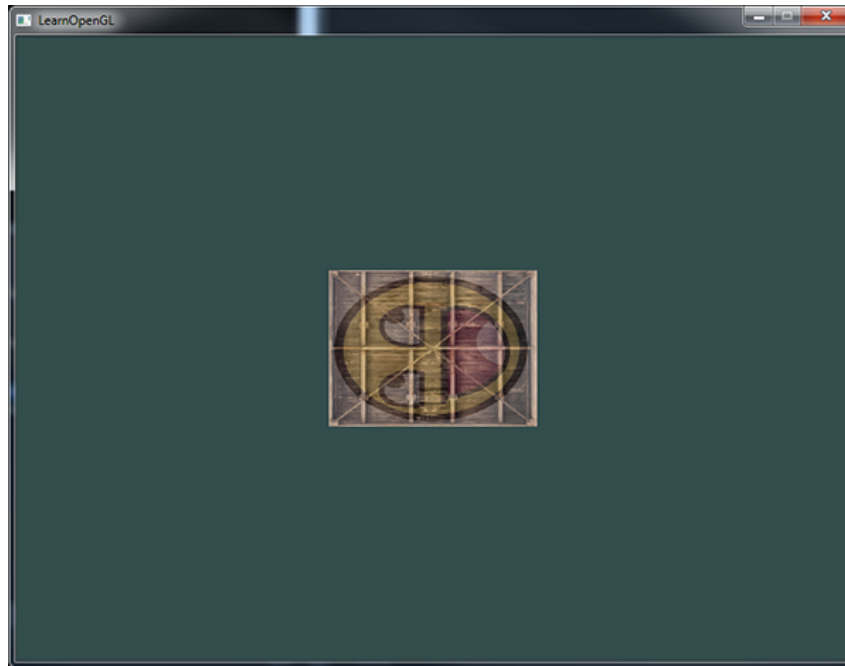
GLSL dispose aussi des types `mat2` et `mat3` qui permettent des opérations de transformation comme pour les vecteurs. Toutes les opérations que nous avons vues (comme la multiplication d'une matrice par un scalaire, d'un vecteur par une matrice, ou de deux matrices) sont permises sur ces types matrices. Lorsque des opérations particulières sur les matrices se présenteront, nous les expliquerons en détail.

Nous avons ajouté une variable uniforme et multiplié le vecteur position par cette matrice de transformation avant de l'affecter à la variable `gl_Position`. Notre conteneur devrait être deux fois plus petit et avoir tourné de 90° vers la gauche. Mais il nous reste à passer la matrice de transformation au shader :

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

Nous récupérons d'abord l'emplacement de la variable uniforme et transmettons ensuite la matrice au shader au moyen de la fonction `glUniform` munie du suffixe `Matrix4fv`. Le premier argument désigne l'emplacement de la variable uniforme, le second argument précise combien de matrices nous transmettons (une seule dans notre cas), le troisième argument permet de transposer la matrice, c'est-à-dire inverser lignes et colonnes. Les développeurs OpenGL utilisent souvent une configuration des matrices dans laquelle le premier indice désigne la colonne et le second la ligne, convention aussi utilisée par GLM, il n'est donc pas nécessaire de transposer les matrices, nous laissons cet argument à `GL_FALSE`. Le dernier paramètre contient la matrice elle-même, mais GLM ne mémorise pas les matrices exactement de la même façon qu'OpenGL souhaite les obtenir, ce qui impose d'utiliser la fonction `value_ptr()` de GLM.

Nous avons créé une matrice de transformation, déclaré une variable uniforme dans le vertex shader et transmis la matrice au shader dans lequel nous transformons nos coordonnées de sommets. Le résultat devrait ressembler à cela :



Parfait ! Notre conteneur est bien tourné vers la gauche et deux fois plus petit qu'avant, la transformation a réussi. Amusons-nous un peu et faisons tourner ce conteneur continuellement, tout en le repositionnant en bas à droite de la fenêtre. Pour qu'il tourne en continu, nous devons mettre à jour la matrice de transformation dans la boucle de rendu, afin de modifier l'angle de rotation constamment. Nous utilisons la fonction `glfwGetTime()` de GLFW à cet effet :

```
glm::mat4 trans;
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

Gardez à l'esprit que dans le cas précédent nous pouvions déclarer la matrice de transformation n'importe où, alors que maintenant, nous devons la recalculer à chaque itération de la boucle de rendu. Lorsque l'on affiche des scènes, il faut en général recalculer les matrices de transformation à chaque passage dans la boucle de rendu pour obtenir un effet de mouvement continu.

Ici, nous commençons par faire tourner le conteneur autour de l'origine (0, 0, 0), et ensuite nous le déplaçons en bas à droite de la fenêtre d'affichage. Rappelez-vous que l'ordre des transformations doit être lu dans l'ordre inverse : même si dans le code nous commençons par la translation et ensuite la rotation, la rotation sera bien effectuée en premier. Comprendre ces transformations et comment elles modifient les objets est assez difficile. Essayez différentes combinaisons de ces transformations et vous comprendrez mieux.

Si vous avez fait les choses correctement, voilà ce que vous obtiendrez :

Cliquer sur ce lien pour lancer l'animation

Un conteneur qui tourne régulièrement, simplement avec une matrice ! Vous voyez maintenant la puissance offerte par les matrices et pourquoi elles sont si utiles dans les applications graphiques. On peut définir une infinité de transformations et les combiner en une seule matrice que l'on peut réutiliser à loisir. Utiliser ces transformations dans le vertex shader nous évite de redéfinir les données des sommets et économise aussi du temps de calcul puisque nous n'avons pas besoin de transmettre les données à chaque fois (ce qui serait assez lent).

Si vous rencontrez quelques problèmes, consultez le [code source](#) ainsi que la nouvelle classe [shader](#).

Dans le tutoriel suivant, nous verrons comment utiliser ces matrices pour définir différents espaces de coordonnées pour nos sommets. Ce sera nos premiers pas dans l'univers du graphisme temps réel 3D !

IX-P - Lectures

-  **Essence of Linear Algebra** : excellente vidéo de Grant Sanderson sur les mathématiques des transformations et de l'algèbre linéaire.

IX-Q - Exercices

- En utilisant la dernière transformation du conteneur, essayez de changer l'ordre des transformations. Voyez ce qui se passe et essayez de comprendre : [solution](#).
- Essayez d'afficher un second conteneur avec un autre appel à `glDrawElements()` mais placez-le à un endroit différent en utilisant seulement les transformations. Assurez-vous de le placer en haut à gauche de la fenêtre et au lieu de le faire tourner, modifiez sa taille en fonction du temps (la fonction sinus peut s'avérer utile ici, notez aussi que les valeurs négatives de la fonction sinus vont inverser l'objet) : [solution](#).

IX-R - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

X - Systèmes de Coordonnées

Dans le tutoriel précédent, nous avons vu comment utiliser les matrices pour transformer tous les sommets au moyen de matrices de transformation. OpenGL requiert que tous les sommets que nous voulons voir à l'écran soient définis en coordonnées normalisées, c'est-à-dire dans l'intervalle $[-1.0, 1.0]$. Ce que l'on fait habituellement est de définir les coordonnées dans le système de notre choix, puis de normaliser ces valeurs dans le vertex shader. Ces coordonnées normalisées (NDC) sont ensuite transformées en coordonnées de pixels sur l'écran par le rasterizer.

La transformation des coordonnées se fait par étapes. Les coordonnées intermédiaires rendent certaines opérations plus faciles comme nous allons le voir. Nous trouverons ainsi cinq systèmes de coordonnées :

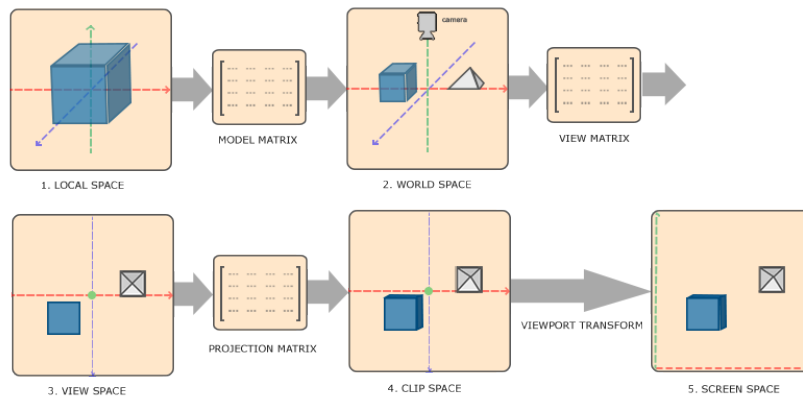
- espace local (ou espace objet) ;
- espace monde (world space) ;
- espace de vue (view space ou eye space) ;
- espace de clip (clip space) ;
- espace écran (screen space).

Ces différents espaces représentent les états successifs dans lesquels seront convertis nos sommets avant de devenir des fragments.

Explicitons cela avec un schéma de chaque étape.

X-A - Le schéma global

Pour passer d'un système de coordonnées au suivant, nous utiliserons des matrices de transformation. Les plus importantes sont les matrices de modèle, de vue et de projection. Les coordonnées des sommets sont d'abord exprimées dans l'espace local, puis transformées en coordonnées monde, puis de vue, puis de clip, et enfin d'écran, comme le montre la figure suivante :



- 1 Les coordonnées locales sont les coordonnées de votre objet, relatives à son origine.
- 2 L'étape suivante consiste à les transformer en coordonnées monde, qui sont relatives à un espace plus grand. Elles sont relatives à une origine globale à la scène et qui est partagée par tous les autres objets.
- 3 Ensuite, on transforme ces coordonnées monde en coordonnées de l'espace de vue de telle façon que chaque coordonnée soit comme elle serait si elles étaient vues par la camera, soit le point de vue du spectateur.
- 4 Ensuite, les coordonnées sont projetées sur l'espace de clipping, on ne conserve que les coordonnées dans l'intervalle $[0, 1]$.
- 5 Enfin, on transforme ces valeurs en coordonnées écran par la transformation de la zone de dessin (viewport), dans l'intervalle défini par `glViewport()`. Les coordonnées résultantes sont enfin envoyées au rasterizer pour en obtenir des fragments.

Certaines opérations sont plus faciles à réaliser selon l'espace de coordonnées utilisé. Par exemple, la modification d'un objet est plus simple dans l'espace local, tandis que le calcul de la position d'un objet par rapport à d'autres objets est plus simple dans l'espace monde. Cela autorise plus de flexibilité. Voyons ces différents espaces plus en détail.

X-B - L'espace local

C'est l'espace dans lequel est défini votre objet. Imaginons que nous créons un cube dans un logiciel de modélisation (comme Blender). L'origine du cube sera sans doute $(0, 0, 0)$, même si le cube doit se trouver finalement à une autre position. Tous les sommets du cube seront définis par rapport à cette origine, ces coordonnées sont locales à l'objet.

Les coordonnées de notre container ont toutes été spécifiées entre -0.5 et 0.5 , avec $(0, 0)$ comme origine, ce sont des coordonnées locales.

X-C - Espace monde

Si l'on importait tous les objets directement dans l'application, ils seraient sans doute empilés les uns sur les autres au point $(0, 0, 0)$, ce qui n'est pas souhaité ; nous voulons définir leur place dans l'espace monde, les objets étant ainsi répartis dans la scène à leur place respective. Cela est accompli par la transformation définie par la matrice du modèle.

La matrice du modèle translate, diminue ou agrandit et/ou fait tourner les objets pour les placer dans la scène. La matrice utilisée dans le tutoriel précédent pour placer le container est une sorte de matrice de modèle.

X-D - Espace de vue

L'espace de vue est ce qu'on appelle la caméra d'OpenGL (appelé encore l'espace de vue ou l'espace caméra). Les coordonnées de l'espace monde sont transformées en fonction de l'emplacement d'où est vue la scène. On obtient ces coordonnées par un ensemble de translations et rotations définies par la matrice de vue. Nous verrons comment définir une telle matrice dans le tutoriel suivant.

X-E - L'espace de clipping

Pour terminer chaque passage du vertex shader, OpenGL attend les coordonnées dans un intervalle spécifique. Les coordonnées en dehors de cet intervalle sont clippées, c'est-à-dire écartées de la scène, ces points ne seront pas visibles. Il est plus simple d'utiliser nos propres coordonnées, puis de les transformer en coordonnées normalisées.

Pour transformer les coordonnées de la vue vers l'espace de clipping, nous définirons une matrice de projection qui spécifie un intervalle de coordonnées, par exemple $[-1000, 1000]$ pour chacune des trois dimensions. La matrice de projection transforme les coordonnées de cet intervalle spécifique en coordonnées normalisées. Les coordonnées en dehors de cet intervalle spécifique ne seront pas projetées sur $[-1, +1]$ et seront donc écartées. Par exemple un sommet en $(1250, 500, 750)$ ne sera pas visible puisque $x > 1000$.



Si une partie d'une primitive (par exemple un triangle) est en dehors de l'intervalle, OpenGL reconstruit le triangle en un ou plusieurs triangles pour coller à l'espace de clipping.

Cette boîte de vue que crée la matrice de projection est appelée le frustum et chaque coordonnée comprise dans cette boîte sera affichée. Le processus qui convertit les coordonnées d'un intervalle spécifié en coordonnées normalisées pour être facilement transformées en coordonnées 2D de l'espace de vue est une projection.

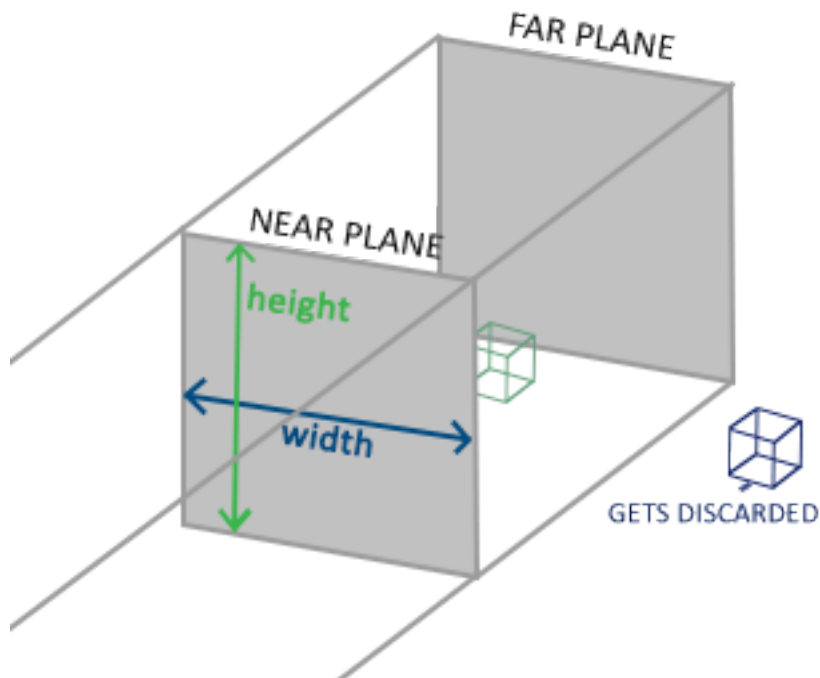
L'opération finale appelée division de perspective, divise les composantes x , y et z des vecteurs position par la composante homogène w ; la division de perspective transforme l'espace de clipping 4D en coordonnées normalisées 3D. Cela se fait automatiquement à la fin de chaque passage dans le vertex shader.

Après cette étape, les coordonnées résultantes sont projetées en coordonnées écran et transformées en fragments.

La matrice de projection qui transforme les coordonnées de vue en coordonnées écran peut prendre deux formes différentes. On peut utiliser une projection orthogonale ou une projection en perspective.

X-E-1 - Projection orthogonale

Une projection orthogonale définit un frustum en forme de cube. Dans ce cas, on spécifie les dimensions du cube (largeur, hauteur, longueur). Les coordonnées qui sont à l'intérieur de ce cube ne seront pas clippées. Ce cube ressemble à un container :



La composante w de chaque vecteur est inchangée. Si la composante w vaut 1.0, la division de perspective ne modifie pas les coordonnées.

Pour créer une matrice de projection orthogonale, on utilisera la fonction `glm::ortho()` :

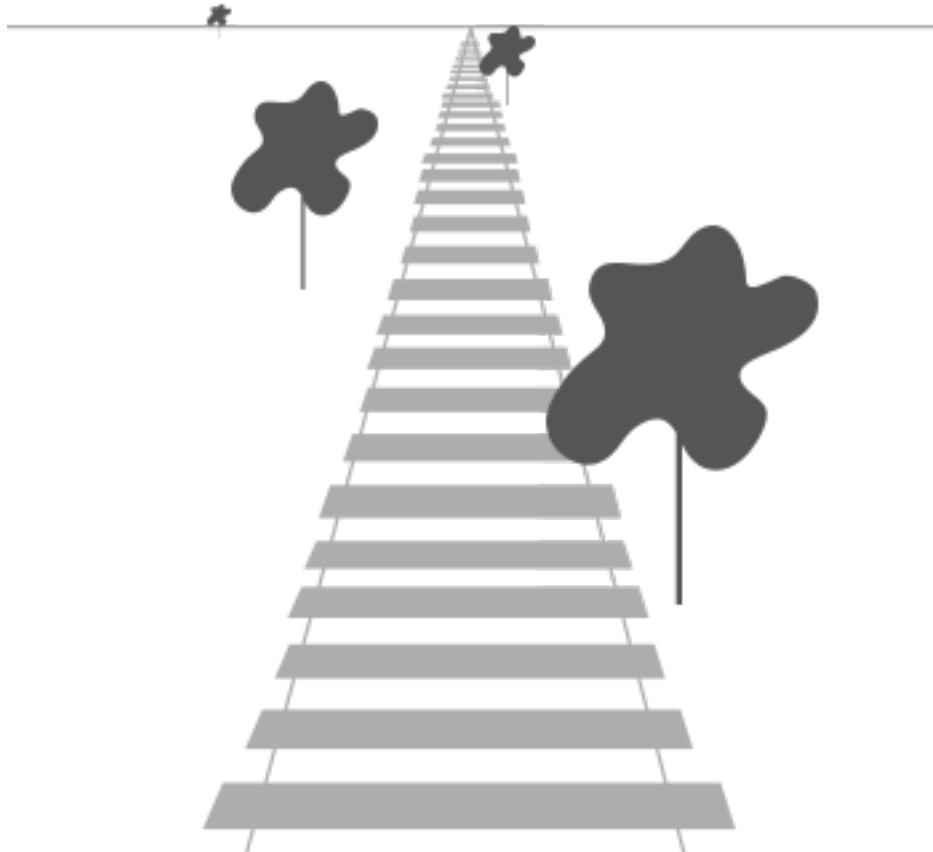
```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

Les deux premiers paramètres spécifient les coordonnées gauche et droite du frustum, ensuite on trouve les coordonnées bas et haut de ce frustum. Les paramètres 5 et 6 donnent la position du plan avant et du plan arrière. Cette matrice de projection transforme toutes les coordonnées en coordonnées normalisées.

Cette projection directe produit des effets peu réalistes car il n'y a aucun effet de perspective. Pour cela on utilisera plutôt la projection en perspective.

X-E-2 - Projection en perspective

Dans un dessin réaliste, les objets éloignés sont plus petits que les objets proches. C'est l'effet de perspective, que l'on voit bien sur l'image suivante :

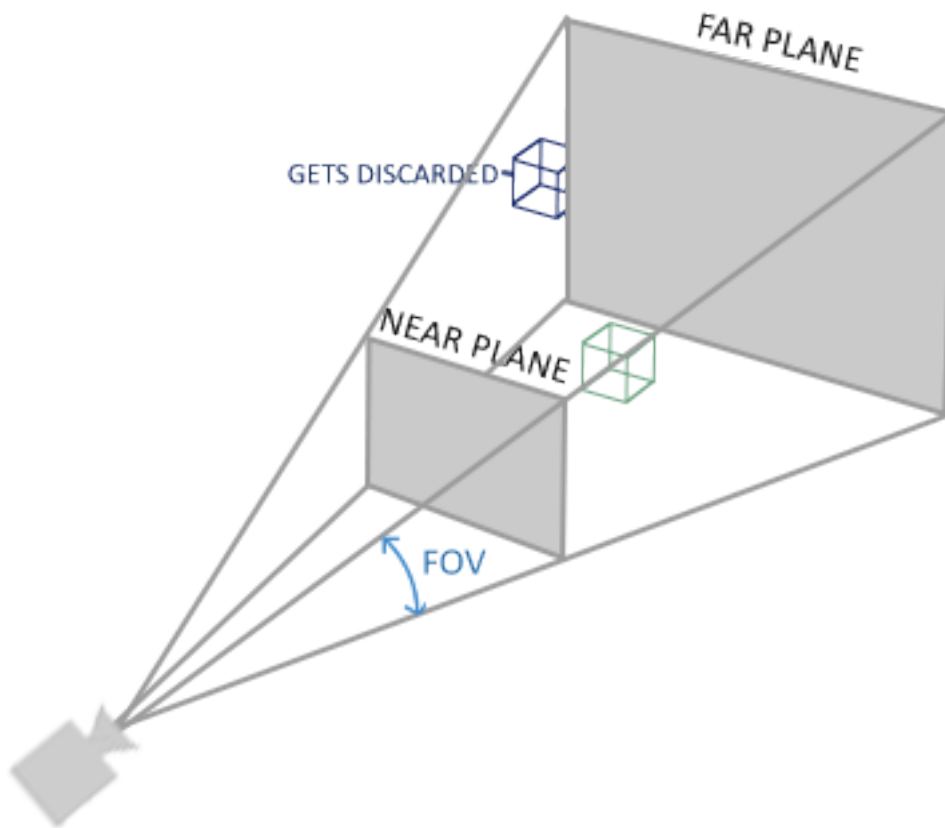


On voit que les travées du rail se rapprochent lorsque la distance augmente. Cet effet est rendu par une matrice de projection en perspective. Cette matrice modifie la composante `w` de chaque sommet de telle façon que plus le sommet sera loin, plus `w` sera grand. Une fois les coordonnées transformées dans l'espace de clipping, elles sont dans l'intervalle $[-w, +w]$. OpenGL requiert que les coordonnées visibles soient entre -1 et $+1$. Lorsque les coordonnées sont dans l'espace de clipping, la division de perspective est appliquée à ces coordonnées en les divisant par `w`, résultant en valeurs d'autant plus faibles que les points sont éloignés du spectateur. Les coordonnées sont ensuite normalisées. Cela est détaillé dans cet [excellent article](#) de Songho.

Une matrice de projection en perspective peut être créée comme suit :

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

Le rôle de `glm::perspective()` consiste à créer un frustrum qui définit l'espace visible, pouvant être vu comme une boîte de section variable :

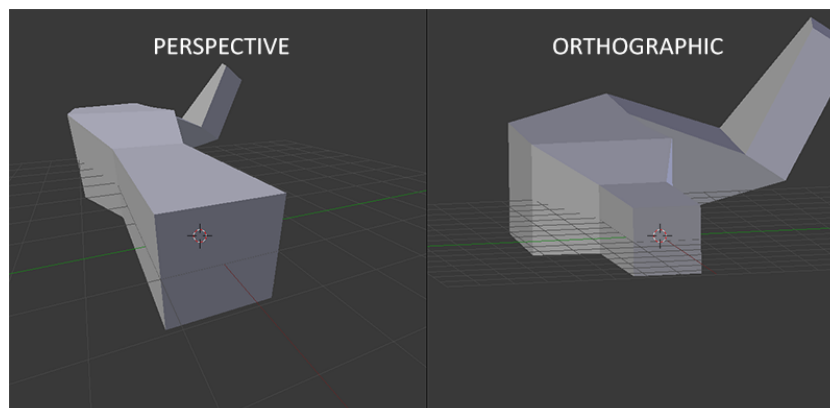


Le premier paramètre définit le champ de vision (field of view (fov)) définissant l'angle d'ouverture de la boîte, souvent choisi à 45°, mais on peut changer cette valeur. Le second paramètre définit le rapport largeur/hauteur, les autres paramètres sont relatifs à la position des plans avant et arrière de la boîte.



Si la valeur near (plan avant de la boîte) est un peu trop grande (par exemple 10.0), les objets proches deviendront invisibles ce qui donne l'impression de voir au travers de ces objets si l'on s'en approche trop.

La projection orthogonale est souvent utilisée pour les rendus 2D et dans certaines applications d'ingénierie ou d'architecture quand on ne souhaite aucune distorsion due à la perspective. Des applications comme Blender utilisent ce type de projection pour la modélisation. On peut voir sur l'image suivante la différence entre les deux types de rendu :



X-F - Synthèse

Nous allons créer une matrice pour chaque étape mentionnée : modèle, vue et projection. Une coordonnée de sommet sera donc transformée comme suit :

$$V_{clip} = M_{projection} \cdot M_{vue} \cdot M_{modele} \cdot V_{local}$$

Noter que l'ordre de multiplication des matrices est inversé (on lit la multiplication des matrices de droite à gauche). Le sommet résultant sera ensuite affecté à la variable `gl_Position` dans le vertex shader et OpenGL réalisera ensuite automatiquement la division de perspective et le clipping.

Et ensuite ?

La sortie du vertex shader requiert que les coordonnées soient dans l'espace de clipping, ce que nous avons réalisé avec les matrices de transformation. OpenGL réalise la division de perspective et le clipping puis utilise les paramètres de `glViewport()` pour obtenir les coordonnées écran à partir des NDC. Chaque coordonnée correspond donc à un point de l'écran. C'est la transformation de la zone de dessin.

Ce sujet peut vous paraître délicat, pas d'inquiétude, nous verrons comment utiliser ces concepts correctement, en particulier au travers des nombreux exemples.

X-G - Passons en 3D

Maintenant que nous savons comment transformer les coordonnées 3D en coordonnées 2D, nous allons commencer à montrer des objets plus réalistes que de simples plans 2D.

Commençons par créer une matrice de modèle, qui consistera ici en translations, homothéties et rotations permettant de placer nos sommets dans l'espace monde. Transformons notre plan en le faisant tourner selon l'axe X :

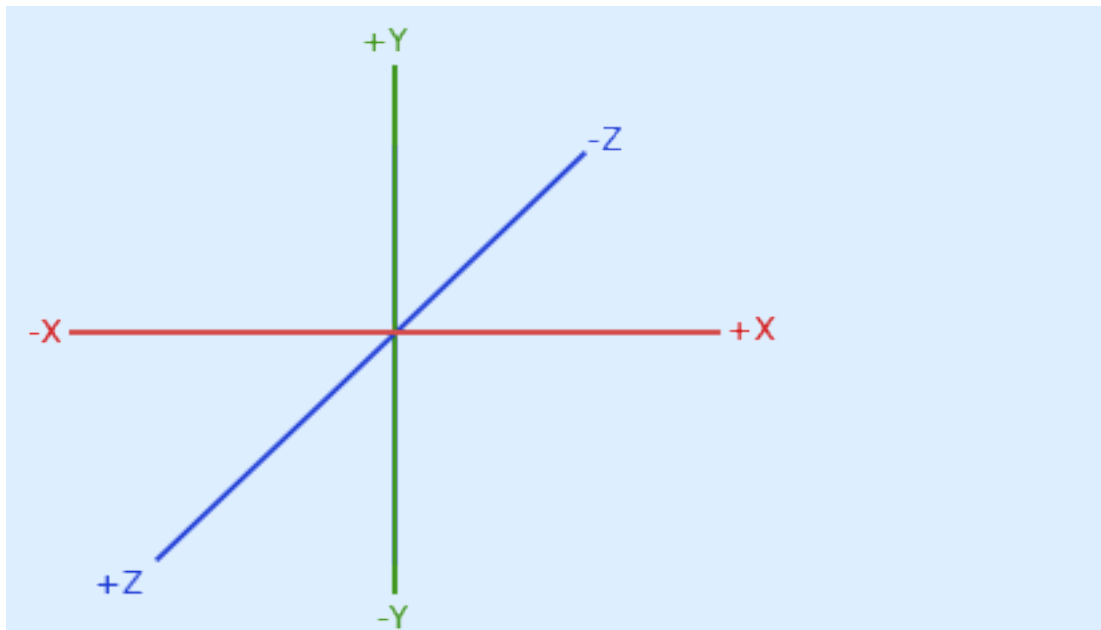
```
glm::mat4 model;
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

En multipliant les coordonnées des sommets par cette matrice de modèle, nous transformons les coordonnées des sommets en coordonnées monde. Notre plan qui est un peu tourné vers le sol représente le plan dans l'espace monde.

Nous créons ensuite une matrice de vue. Nous voulons reculer un peu dans la scène de façon à ce que les objets deviennent visibles (dans l'espace monde, nous sommes situés à l'origine (0, 0, 0)). Se déplacer en arrière revient à faire avancer la scène, et donc déplacer les objets vers les z négatifs.

Système de coordonnées direct

La figure montre le repère utilisé par OpenGL, l'axe Z est dirigé vers nous :



On peut se repérer de la façon suivante :

- placer le pouce de la **main droite** selon l'axe X ;
- placer l'index suivant l'axe Y ;
- le majeur vous indique alors le sens de l'axe Z.

Si vous utilisez la main gauche, ce sera le contraire..., le système inverse est utilisé dans DirectX. Notons que pour les coordonnées normalisées, OpenGL utilise le système inversé. (la matrice de projection inverse le système).

Nous verrons dans le prochain tutoriel comment déplacer le point de vue, mais pour l'instant notre matrice de vue sera la suivante :

```
glm::mat4 view;
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

La dernière chose à faire est de définir la matrice de projection en perspective :

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f, 100.0f);
```

Nous devons ensuite passer ces matrices de transformation au shader. Déclarons ces matrices en variables uniformes, et modifions les sommets au moyen de ces matrices :

```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // notez que nous lisons la multiplication de droite à gauche
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ...
}
```



```
}

```

Nous devons aussi transmettre ces matrices au shader, cela à chaque rendu, car ainsi nous pourrions les modifier au cours du temps :

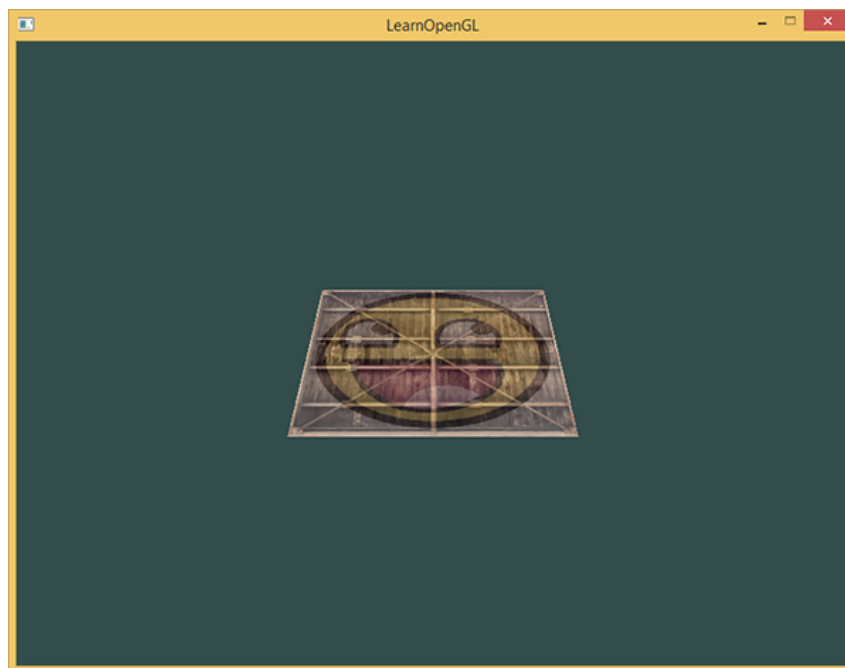
```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
... // pareil pour View Matrix et Projection Matrix

```

Nos sommets sont ainsi modifiés par ces trois matrices, l'objet final devrait être :

- penché vers le sol ;
- un peu éloigné de nous ;
- affiché en perspective.

Voyons si le résultat correspond à cela :



Cela ressemble à un plan reposant sur un sol imaginaire. Vous pouvez vérifier votre code ici : [source code](#).

X-H - Plus de 3D

Nous avons travaillé avec un plan 2D dans un espace 3D, passons maintenant à un cube en 3D. Pour cela nous utilisons 36 sommets (6 faces, 2 triangles par face, 3 sommets par triangle). Vous pouvez les retrouver [ici](#). Pour le fun, faisons tourner ce cube :

```
model = glm::rotate(model,
    (float)glfwGetTime() * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));

```

Et ensuite, il faut afficher le cube avec ses 36 sommets :

```
glDrawArrays(GL_TRIANGLES, 0, 36);

```

Vous devriez voir ceci :

Cliquer sur ce lien pour lancer l'animation

Cela ressemble à un cube mais il reste un problème. Certaines faces du cube sont tracées par-dessus d'autres faces, cela parce qu'OpenGL trace le cube triangle par triangle, effaçant certains triangles en affichant par-dessus, alors qu'ils devraient être cachés.

Heureusement, OpenGL mémorise l'information de profondeur dans ce qu'on appelle un z-buffer, ce qui permet d'afficher ou non un pixel. On peut donc configurer OpenGL pour effectuer des tests de profondeur.

X-H-1 - Z-buffer

OpenGL mémorise les informations de profondeur dans un z-buffer aussi appelé tampon de profondeur (depth-buffer). GLFW crée automatiquement ce tampon. La profondeur de chaque fragment est mémorisée, et à chaque affichage de fragment, la profondeur est comparée avec la valeur mémorisée dans le z-buffer ; si le fragment est derrière cette valeur, le fragment n'est pas affiché. Cette opération s'appelle le test de profondeur (depth testing).

Cependant, il faut activer ce test de profondeur, qui n'est pas activé par défaut. Cela se fait avec `glEnable()`. `glEnable()` et `glDisable()` permettent d'activer ou d'inhiber certaines fonctionnalités d'OpenGL. Voilà comment activer le test de profondeur :

```
glEnable(GL_DEPTH_TEST);
```

Il faut aussi effacer le tampon de profondeur avant chaque rendu, de la même façon que l'on efface le tampon de couleurs :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Voyons ce que cela donne :

Cliquer sur ce lien pour lancer l'animation

Ça fonctionne ! Voir le code [ici](#).

X-H-2 - Plus de cubes !

Supposons que nous voulions afficher 10 cubes sur l'écran. Chaque cube sera identique aux autres, mais placés différemment et tournant chacun selon un axe différent. Nous n'avons pas à modifier la définition du cube ni à modifier le tableau d'attributs pour dessiner plus d'objets. La seule chose à faire est de changer pour chaque objet la matrice de modèle pour chacun de ces cubes.

Définissons un vecteur de translation qui spécifie la position de chaque cube dans un tableau de `glm::vec3` :

```
glm::vec3 cubePositions[] = {
    glm::vec3( 0.0f, 0.0f, 0.0f),
    glm::vec3( 2.0f, 5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3( 2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f, 3.0f, -7.5f),
    glm::vec3( 1.3f, -2.0f, -2.5f),
    glm::vec3( 1.5f, 2.0f, -2.5f),
    glm::vec3( 1.5f, 0.2f, -1.5f),
    glm::vec3(-1.3f, 1.0f, -1.5f)
};
```

Utilisons une matrice de modèle différente pour chaque cube. Le cube défini sera ainsi rendu 10 fois avec une matrice de modèle différente (nous ajoutons aussi une petite rotation à chaque fois).

```
glBindVertexArray(VAO);
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model;
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```



En cas de souci, voir le code ici : [source](#).

X-I - Exercices

- Expérimenter les paramètres FoV et aspect-ratio de la fonction projection de GLM. Voir comment cela affecte le frustrum de la perspective.
- Jouer avec la matrice de vue dans différentes directions pour voir le changement de la scène. Imaginez cette matrice comme une caméra.
- Essayer de faire tourner les trois premiers containers en laissant les autres fixes, cela en utilisant la matrice de modèle : [solution](#).

X-J - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

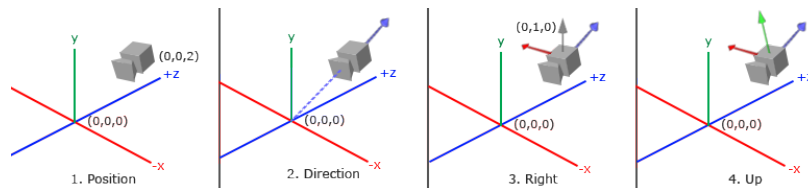
XI - Caméra

Dans le [tutoriel précédent](#), nous avons examiné la matrice de vue et comment l'utiliser pour se déplacer dans la scène. OpenGL ne connaît pas le concept de caméra, mais on peut le simuler en déplaçant la scène dans la direction inverse, donnant ainsi l'impression que nous nous déplaçons.

Nous allons voir comment mettre en place une caméra dans OpenGL. Nous présenterons une caméra permettant de se déplacer librement dans la scène. Nous présenterons aussi les entrées clavier et souris, et enfin nous concevrons une classe pour la caméra.

XI-A - L'espace Caméra/Vue

Lorsque nous parlons d'espace de vue, nous voyons toutes les coordonnées des sommets avec comme perspective celle de la caméra, prise comme origine. La matrice de vue transforme toutes les coordonnées en coordonnées de vue qui sont relatives à la position et à la direction de la caméra. Pour définir une caméra, nous devons connaître sa position, sa direction, un vecteur pointant vers la droite, un autre pointant vers le haut. Nous allons en fait créer un repère comprenant trois axes avec la caméra comme origine.



XI-A-1 - Position de la caméra

La position est donnée par un vecteur dans l'espace global :

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```



N'oublions pas que l'axe z est dirigé vers nous et donc pour faire reculer la caméra il faut lui affecter un $z > 0$.

XI-A-2 - Direction de la caméra

Un vecteur va définir la direction dans laquelle pointe la caméra. Pour l'instant, supposons que la caméra pointe vers le point (0, 0, 0). Cela peut se faire simplement en effectuant la différence entre les deux vecteurs : position et point de visée (voir chapitre sur les transformations). Ce vecteur va ainsi pointer vers les valeurs positives de z :

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```



Ce nom de direction n'est pas idéal puisque ce vecteur pointe dans la direction inverse de la direction visée par la caméra.

XI-A-3 - Axe de droite

Pour représenter l'axe x de l'espace caméra, nous utilisons une astuce consistant à faire le produit vectoriel entre un vecteur dirigé vers le haut (axe y>0) et le vecteur représentant la direction de visée :

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

XI-A-4 - Axe vers le haut

Cette direction est obtenue par le produit vectoriel entre la direction de visée et l'axe de droite :

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

Nous avons défini le repère de l'espace caméra, en utilisant le processus **Gram-Schmidt** de l'algèbre linéaire. Nous pouvons maintenant définir la matrice LookAt qui se révèle très utile pour créer la caméra.

XI-B - Look At

Une propriété très intéressante des matrices est la suivante : si l'on définit trois axes perpendiculaires, on peut créer une matrice au moyen de ces trois axes, et en y ajoutant un vecteur de translation, on peut simplement changer de repère en multipliant les coordonnées d'un vecteur par cette matrice. C'est ce que fait la matrice LookAt pour obtenir les coordonnées dans l'espace de la caméra :

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Où R est le vecteur pointant vers la droite, U le vecteur vers le haut, D le vecteur dans la direction de visée et P la position de la caméra. Notons que le vecteur est inversé puisque nous souhaitons déplacer l'espace dans la direction opposée au déplacement supposé du point de vue. En utilisant la matrice LookAt comme matrice de vue, nous transformerons toutes coordonnées de l'espace utilisé. Cette matrice crée ainsi une matrice de vue qui regarde dans la direction souhaitée.

En fait, GLM réalise déjà ce travail. Nous n'avons qu'à spécifier la position de la caméra ainsi qu'un point de visée et un vecteur qui pointe vers le haut et GLM fait le reste :

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                  glm::vec3(0.0f, 0.0f, 0.0f),
                  glm::vec3(0.0f, 1.0f, 0.0f));
```

La fonction `glm::LookAt()` requiert en paramètre la position, le point visé et le vecteur indiquant le haut. On crée ainsi une matrice de vue telle que définie dans le tutoriel précédent.

Avant de se plonger dans les entrées utilisateur, déplaçons notre caméra autour de la scène en gardant (0, 0, 0) comme point de visée.

Nous définissons comme position de la caméra un point situé sur un cercle, ce point étant modifié à chaque rendu, la caméra se déplace donc sur ce cercle de rayon radius. On recrée la matrice de vue à chaque fois, le rythme étant obtenu avec la fonction `glfwGetTime` de GLFW :

```
float radius = 10.0f;
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius;
glm::mat4 view;
view = glm::lookAt(glm::vec3(camX, 0.0, camZ),
                  glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

Voilà le résultat :

Cliquer sur ce lien pour lancer l'animation

Vous pouvez bien sûr essayer différentes valeurs de position et d'orientation pour mieux appréhender ces concepts. Le code se trouve [ici](#).

XI-C - Se balader dans la scène

Faire bouger la caméra autour de la scène est sympa, mais ce serait encore mieux si nous pouvions la déplacer nous-mêmes !

Créons d'abord une caméra en définissant les variables nécessaires au début du programme :

```
glm::vec3 cameraPos    = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront  = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp     = glm::vec3(0.0f, 1.0f, 0.0f);
```

La fonction LookAt devient :

```
view = glm::lookAt(cameraPos, cameraPos - cameraFront, cameraUp);
```

Nous avons d'abord défini la caméra comme dans le paragraphe précédent. La direction est la position courante — le point de visée. Cela assure que la caméra visera toujours le point de visée. Modifions la position de la caméra en modifiant le vecteur cameraPos en utilisant le clavier. Nous avons déjà utilisé la fonction processInput() pour gérer les entrées clavier, ajoutons quelques nouvelles commandes :

```
void processInput(GLFWwindow *window)
{
    ...
    float cameraSpeed = 0.05f; // adjust accordingly
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

En utilisant les touches WASD, la position de la caméra change : pour se déplacer en avant ou en arrière, on modifie le point de visée sur l'axe z, alors que les touches A et D permettent de se déplacer à droite ou à gauche de la position en cours.



Notons que nous normalisons le vecteur right. Sans cela nous obtiendrions un vecteur de norme variable, ce qui modifierait la vitesse de la caméra selon son orientation.

On peut désormais faire bouger la caméra, à cela près que la vitesse est dépendante de votre système.

XI-D - Vitesse du mouvement

Nous avons choisi une valeur constante pour le mouvement, mais selon la rapidité de la machine on aura un résultat différent. Il faut pouvoir fixer la vitesse de déplacement de la caméra indépendamment de la rapidité de la machine.

Les applications graphiques et les jeux mémorisent en général le temps écoulé depuis le dernier rendu. On multiplie ensuite ce temps par les valeurs de vitesse. Si ce temps est grand (système lent), on augmente ainsi le déplacement à

effectuer sur cette image et inversement, si le temps est petit (système rapide) on diminue le déplacement à effectuer pour cette image. Cela permet de s'affranchir de la vitesse de la machine.

Pour calculer ce temps, on utilisera deux variables globales :

```
float deltaTime = 0.0f; // Time between current frame and last frame
float lastFrame = 0.0f; // Time of last frame
```

On calcule ainsi pour chaque image le temps écoulé depuis le rendu précédent :

```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

Et on tient compte de ce temps pour la vitesse de déplacement :

```
void processInput(GLFWwindow *window)
{
    float cameraSpeed = 2.5f * deltaTime;
    ...
}
```

On obtient ainsi une caméra plus douce et cohérente pour se déplacer :

Cliquer sur ce lien pour lancer l'animation

Le code se trouve [ici](#).

XI-E - Naviguer dans la scène

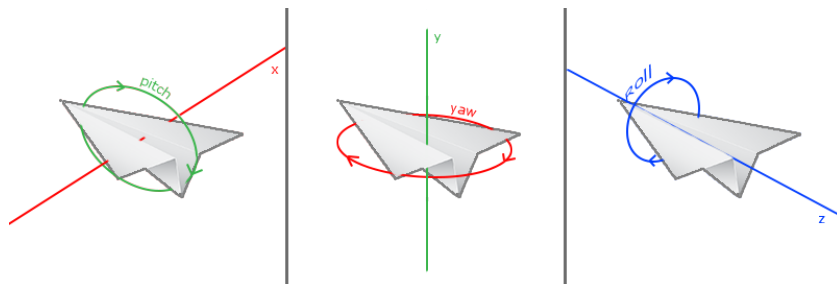
Utiliser les touches pour diriger son regard n'est pas très pratique. Mieux vaut utiliser la souris !

Nous allons modifier la direction visée en utilisant la souris. Cela est plus compliqué et requiert un peu de trigonométrie. Si vous n'êtes pas à l'aise avec ces notions, vous pouvez simplement recopier le code pour le moment.

XI-E-1 - Les angles d'Euler

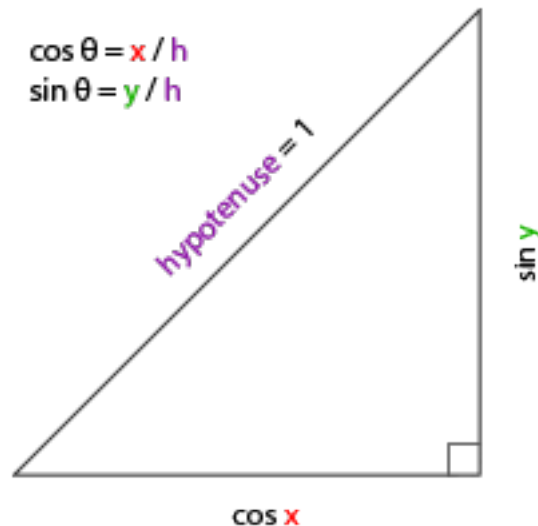
Les **angles d'Euler** sont trois valeurs qui permettent de représenter toute rotation en 3D. Pour mémoire, yaw = précession, pitch = nutation, roll = rotation propre.

Ces trois angles sont représentés sur la figure suivante :



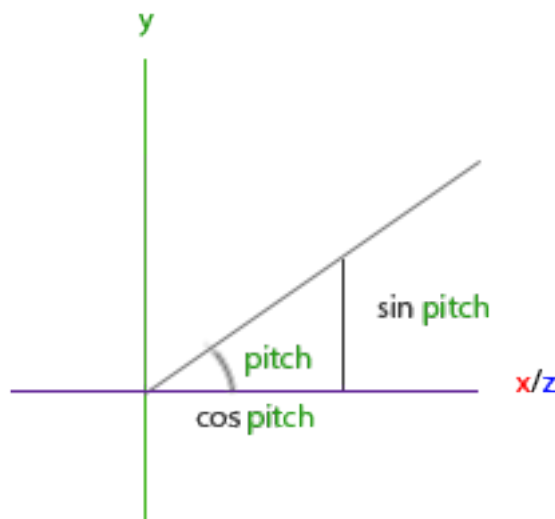
Le pitch précise l'orientation vers le haut ou le bas, le yaw indique la rotation vers la droite ou la gauche. Le roll est par exemple utilisé dans les jeux de combat aérien, nous ne l'utiliserons pas ici. La combinaison de ces angles permet de calculer notre vecteur de rotation en 3D.

Pour notre système de caméra, nous n'avons besoin que du yaw et du pitch. Grâce à ces valeurs, nous pouvons obtenir un vecteur 3D représentant notre vecteur de direction. Le processus pour convertir le yaw et le pitch en un vecteur demande un peu de trigonométrie :



Si nous définissons l'hypoténuse à 1, nous savons grâce à la trigonométrie (cosinus, sinus, tangente) que la longueur du côté adjacent est $\cos x / h = \cos x / 1 = \cos x$ et

$\sin y / h = \sin y / 1 = \sin y$ pour le côté opposé. Cela nous donne des formules génériques pour obtenir la longueur de la direction sur l'axe des X et Y à partir d'un angle donné. Utilisons-les pour calculer les composantes du vecteur de direction :



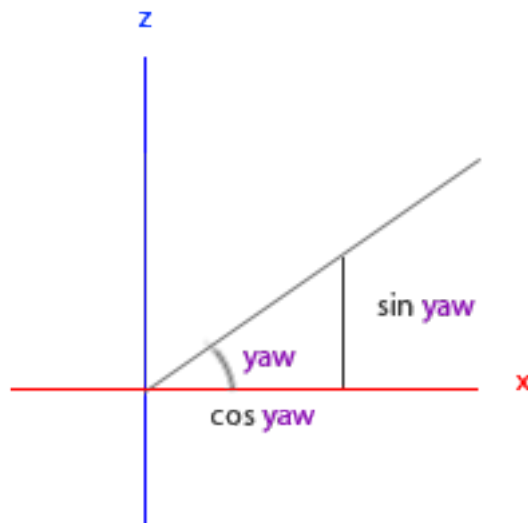
Le triangle ressemble au précédent. En le visualisant à partir du plan XZ et en regardant en direction de l'axe Y nous pouvons calculer la longueur de la direction Y (orientation verticale du regard). À partir de l'image, nous pouvons déduire que la valeur Y pour un pitch donné correspond à $\sin \theta$:

```
direction.y = sin(glm::radians(pitch)); // Notez que nous devons convertir les angles en radians
```

Ici, nous ne nous occupons que de la valeur Y, mais avec un peu plus de travail nous pouvons voir que les composantes X et Z sont aussi concernées. À partir du triangle, nous pouvons définir leurs valeurs comme suit :

```
direction.x = cos(glm::radians(pitch));  
direction.z = cos(glm::radians(pitch));
```

Voyons si nous pouvons aussi trouver les composantes à partir du yaw :



Tout comme avec le triangle pour le pitch, nous pouvons voir que la composante X dépend de $\cos(yaw)$ et que la composante Z dépend de $\sin(yaw)$. En ajoutant ce constat au calcul précédent, nous pouvons obtenir le vecteur direction grâce aux valeurs du pitch et du yaw :

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));  
direction.y = sin(glm::radians(pitch));  
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

Cela nous donne la formule pour convertir les valeurs de yaw et pitch en un vecteur direction que nous utiliserons pour inspecter la scène. Mais comment obtenir ces valeurs de pitch et yaw ?

XI-F - Entrées avec la souris

Les valeurs yaw et pitch vont être obtenues en bougeant la souris (ou un joystick). Les mouvements horizontaux déterminent le yaw et les mouvements verticaux le pitch. Le principe consiste à mémoriser la position de la souris lors du dernier rendu et de calculer la différence de position de la souris lors du rendu en cours. La variation de position sera ensuite convertie en variation des angles yaw et pitch, ce qui se traduira par le mouvement correspondant de la caméra.

Tout d'abord, imposons à GLFW de cacher le curseur de la souris et de le capturer (une fois que l'application a le focus, le curseur restera dans la fenêtre jusqu'à ordre contraire ou fin de l'application). Cela se fait simplement :

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Dès lors, la souris ne sera plus visible et ne quittera plus la fenêtre. C'est parfait pour une caméra à la première vue (comme dans les First Person Shooter (FPS)).

Nous devons imposer à GLFW d'enregistrer les événements correspondant aux mouvements de la souris, grâce à une fonction callback :

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

Ici, `xpos` et `ypos` représentent la position de la souris. La fonction callback, une fois enregistrée, est appelée dès que la souris bouge. Pour l'enregistrer :

```
glfwSetCursorPosCallback(window, mouse_callback);
```

Pour gérer une caméra FPS au moyen de la souris, plusieurs étapes sont nécessaires avant de calculer la nouvelle direction de la caméra :

- 1 Calculer le déplacement de la souris depuis le dernier rendu ;
- 2 Ajouter ce déplacement aux angles yaw et pitch ;
- 3 Contraindre les angles à certaines valeurs limites ;
- 4 Calculer le nouveau vecteur direction.

Calculons le déplacement de la souris depuis le dernier rendu. On supposera au début que la souris est au milieu de la fenêtre (800 x 600) :

```
float lastX = 400, lastY = 300;
```

Puis nous calculons à chaque rendu le déplacement :

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // il faut inverser car ypos est donné de haut en bas
lastX = xpos;
lastY = ypos;

float sensitivity = 0.05f;
xoffset *= sensitivity;
yoffset *= sensitivity;
```

Nous utilisons une variable de sensibilité pour adapter la valeur du déplacement de la souris au déplacement de la caméra, cette sensibilité pouvant être réglée à votre convenance.

Nous mettons à jour les angles en fonction de ce déplacement :

```
yaw   += xoffset;
pitch += yoffset;
```

Il faut aussi contraindre les angles pour ne pas obtenir de mouvements bizarres de la caméra. Ainsi le pitch sera limité à [-89, +89] degrés, permettant de regarder le ciel ou le sol, mais pas de retourner la caméra.

```
if(pitch > 89.0f)
    pitch = 89.0f;
if(pitch < -89.0f)
    pitch = -89.0f;
```

Le yaw n'a pas besoin d'être limité, car on peut tourner horizontalement autant que l'on veut.

La dernière opération consiste à mettre à jour le vecteur direction avec ces nouveaux angles :

```
glm::vec3 front;
front.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
front.y = sin(glm::radians(pitch));
front.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
cameraFront = glm::normalize(front);
```

Le vecteur `cameraFront` est déjà inclus dans la matrice `lookAt` de GLM, pas besoin d'en faire plus.

Si vous exécutez le code, vous remarquerez que la caméra fait un grand bond dès que l'application est activée. Cela est dû à ce que la position initiale du curseur est fautive puisque définie au centre de la fenêtre, point assez éloigné du bord de la fenêtre où apparaît le curseur lorsque l'on donne le focus à l'application. Il faut, pour résoudre cela, initialiser correctement la position du curseur et ne pas effectuer de déplacement lors du premier rendu :

```
if(firstMouse) // variable initialisée à true, puis laissée à false
{
    lastX = xpos;
    lastY = ypos;
    firstMouse = false;
}
```

Le code final devient donc :

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if(firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    float sensitivity = 0.05;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    if(pitch > 89.0f)
        pitch = 89.0f;
    if(pitch < -89.0f)
        pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}
```

Voilà, on peut se déplacer librement dans la scène !

XI-G - Zoom

Pour compléter le système de caméra, nous allons ajouter un zoom. Dans le tutoriel précédent, nous avons vu que le champ de vision (fov) définit quelle proportion de la scène serait visible. Lorsque l'on diminue le fov, l'espace rendu devient aussi plus petit, donnant l'illusion d'un zoom. Nous utiliserons la roulette de la souris (scroll) pour régler ce zoom. On utilise à nouveau une fonction callback :

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    if(fov >= 1.0f && fov <= 45.0f)
        fov -= yoffset;
    if(fov <= 1.0f)
        fov = 1.0f;
    if(fov >= 45.0f)
        fov = 45.0f;
}
```

La valeur `yoffset` représente la variation de la roulette, et nous modifions le fov en conséquence. 45 degrés est la valeur par défaut pour le fov, nous allons donc le limiter entre 1.0 et 45.0.

Il faut ensuite modifier la matrice de projection en perspective avec cette nouvelle valeur du fov :

```
projection = glm::perspective(glm::radians(fov), 800.0f / 600.0f, 0.1f, 100.0f);
```

Et n'oublions pas d'enregistrer notre fonction callback :

```
glfwSetScrollCallback(window, scroll_callback);
```

Nous avons maintenant une caméra qui permet de se déplacer dans la scène à notre guise.

Cliquer sur ce lien pour lancer l'animation

Essayez de coder ce système de caméra et de vous familiariser avec, en cas de souci le code est là : [source code](#).



*Ce système n'est pas parfait. En fonction de vos contraintes, vous pourrez introduire un **blocage de cardan** (Gimbal Lock). La meilleure caméra serait développée en utilisant les quaternions mais nous verrons cela plus loin.*

XI-H - Une classe Camera

Une caméra nécessite pas mal de lignes de code, que nous réutiliserons dans la suite. Pour cette raison, nous allons concevoir une classe Camera qui fera le travail demandé, et nous allons même y ajouter quelques trucs. Nous allons juste vous donner le code de cette classe si vous souhaitez vous y plonger.

De la même façon que pour la classe Shader, nous créons cette classe dans un fichier d'en-tête. Vous le trouverez : [ici](#). Vous pouvez comprendre le fonctionnement de cette classe, et on vous conseille de l'utiliser pour voir comment on peut créer un objet caméra de cette façon.



Cette caméra est de type FPS qui répond à la plupart des besoins et fonctionne bien avec les angles d'Euler, mais soyez prudent si vous créez une caméra pour un jeu de simulation de vol par exemple. Chaque type de caméra a ses propres limitations qui peuvent poser des problèmes. Par exemple une caméra FPS n'autorise pas d'angles de pitch > 90° et un vecteur up de (0, 1, 0) ne fonctionne pas si l'on prend en compte l'angle roll.

La version du code utilisant cette nouvelle caméra est [ici](#).

XI-I - Exercices

- Voyez si vous pouvez transformer la classe Camera pour réaliser une vraie caméra FPS avec laquelle on ne peut pas voler, mais qui permet seulement de regarder la scène en restant dans le plan xz : [solution](#).
- Essayez de créer votre propre fonction LookAt en créant manuellement une matrice de vue comme discuté au début de ce chapitre. Remplacer la fonction LookAt par la vôtre et voyez si cela fonctionne de la même façon : [solution](#).

XI-J - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](#).

XII - Glossaire

Bravo pour avoir atteint la fin de la section « Pour démarrer ». Vous savez maintenant créer une fenêtre, créer et compiler des shaders, transmettre les données des sommets aux shaders au moyen de tampons ou de variables uniformes, dessiner des objets, utiliser des textures, comprendre les vecteurs et les matrices et combiner tout cela pour créer une scène 3D avec une caméra pour s'y promener.

C'est vraiment un gros boulot d'accompli. Testez les différents tutoriels, et expérimentez vos propres idées. Lorsque vous serez à l'aise avec tout cela, [passez à la suite](#).

- **OpenGL** : une spécification formelle d'une API graphique qui fournit la définition et les sorties de chaque fonction.
- **GLAD** : une bibliothèque pour charger les extensions OpenGL et qui initialise les adresses de chaque fonction pour pouvoir les utiliser dans l'application.
- **Viewport** (zone d'affichage) : la fenêtre dans laquelle sera effectué le rendu.
- **Graphics Pipeline** : le processus complet que doit traverser un sommet avant de finir comme pixel sur l'écran.
- **Shader** : un petit programme qui tourne sur la carte graphique. Grâce aux shaders, plusieurs étapes du pipeline peuvent être définies par le développeur pour remplacer les fonctionnalités par défaut.
- **Vertex** (sommet) : un ensemble de données pour représenter un point.
- **Normalized Device Coordinates** (NDC, coordonnées normalisées) : le système de coordonnées que doivent utiliser les sommets après les opérations de clipping et la division de perspective. Les NDC entre -1.0 et 1.0 ne seront pas écartées et seront donc visibles.
- **Vertex Buffer Object** (VBO) : un tampon qui mémorise les données des sommets sur la carte graphique.
- **Vertex Array Object** (VAO) : mémorise les informations relatives à l'état des attributs de sommets et du tampon.
- **Element Buffer Object** (EBO) : un tampon qui mémorise des indices pour un tracé indexé.
- **Uniform** : un type spécial de variable GLSL globale (tous les shaders peuvent accéder à cette variable) et qui n'est affectée qu'une seule fois.
- **Texture** : une image appliquée sur les objets, donnant l'illusion d'un objet très détaillé.
- **Texture Wrapping** : définit la façon dont OpenGL utilise une texture lorsque les coordonnées de texture sortent de l'intervalle [0, 1].
- **Texture Filtering** (filtrage de texture) : définit la façon dont OpenGL échantillonne la texture quand on doit choisir entre plusieurs texels (pixels de texture). Cela est utile lorsque la texture doit être étirée.
- **Mipmaps** : ensemble de plusieurs versions d'une même texture de différentes tailles permettant d'utiliser des textures plus petites suivant la distance de l'objet avec le spectateur.
- **stb_image** : bibliothèque de chargement d'image.
- **Texture Units** : autorise l'utilisation de plusieurs textures pour un même objet en les mélangeant.
- **Vector** : un objet mathématique qui définit les directions ou positions dans chaque dimension.
- **Matrix** : un tableau mathématique de scalaires (réels).

- **GLM** : une librairie mathématique conçue pour OpenGL.
- **Local Space** : l'espace dans lequel est défini un objet, les coordonnées étant relatives à l'origine de l'objet.
- **World Space** : espace définissant une même origine pour tous les objets.
- **View Space** : les coordonnées sont calculées avec le point de vue de la caméra.
- **Clip Space** : les coordonnées sont vues depuis la caméra, mais la projection est appliquée. C'est dans cet espace que finissent les coordonnées des sommets, comme sorties du vertex shader. OpenGL fait le reste (clipping et division de perspective).
- **Screen Space** : les coordonnées sont relatives à l'écran (de 0 à la largeur/hauteur de l'écran).
- **LookAt** : une matrice de vue permettant à l'utilisateur de regarder où bon lui semble.
- **Angles d'Euler** : ces angles permettent de représenter toute direction en 3D.

XII-A - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site [Learn OpenGL](http://learnopengl.com).