

Développez vos applications 3D avec OpenGL 3.3

Par Boouh



www.openclassrooms.com

Sommaire

Sommaire	2
Lire aussi	4
Développez vos applications 3D avec OpenGL 3.3	6
Partie 1 : Les bases d'OpenGL 3	7
Introduction	7
Qu'est-ce qu'OpenGL ?	7
Installer les différents outils	8
Différences entre Windows et Linux	10
Différences entre OpenGL 2.1 et 3.3	10
Créer une fenêtre avec la SDL 2.0	11
Préparation de l'IDE	12
Premier programme	14
Les paramètres OpenGL	19
Premier affichage	25
Le fonctionnement d'OpenGL	26
Nomenclature des fonctions OpenGL	27
Boucle principale	27
Repère et Origine	30
Afficher un triangle	31
Afficher un triangle	31
Afficher plusieurs triangles	37
La classe SceneOpenGL	39
La classe SceneOpenGL	40
Implémentation de la classe	43
Exercices	49
Introduction aux shaders	56
Qu'est-ce qu'un Shader ?	56
Introduction	56
A quoi servent les shaders ?	57
Exemples de Shaders	57
Utilisation des shaders	59
Un peu de couleurs	63
Un peu de couleurs	63
Exercices	69
Les matrices	75
Introduction	75
Les matrices sous OpenGL 2.1	75
Partie théorique	76
Multiplication d'un vecteur par une matrice	77
Multiplication de deux matrices	81
Transformations au niveau matricielle	85
Transformations au niveau géométrique	86
Les Transformations sous OpenGL	88
Illustrations	89
Les matrices et la boucle principale	91
Accumulation de transformations	92
La troisième dimension (Partie 1/2)	93
La matrice de projection	94
Présentation	94
Utilisation de la librairie GLM	97
Utilisation de la projection	98
Inclusion des matrices	98
Interaction entre le programme et les matrices	101
Exemples de transformations	106
Introduction	106
Les transformations	107
La troisième dimension (Partie 2/2)	117
La caméra	118
Introduction	118
La méthode lookAt	118
Affichage d'un Cube	122
Introduction	122
Affichage d'un cube	124
La classe Cube	143
La classe Cube	143
Pseudo-animation	155
Le Frame Rate	157
Calcul du FPS	157
Programmation	157
Push et Pop	161
La problématique	161
Introduction	161
Les piles	161
Sauvegarde et Restauration	163

Substitution aux piles	163
Quand sauvegarder ?	164
Ce qu'il faut retenir	167
Les événements avec la SDL 2.0	167
Différences entre la SDL 1.2 et 2.0	168
La classe Input	170
Les méthodes indispensables	183
Méthodes	183
Exercices	188
Les textures	193
Introduction	194
Introduction	194
Chargement	196
Les Objets OpenGL	196
La méthode charger	199
Plaquage	214
Affichage d'une texture	214
Améliorations	228
L'inversion des pixels	228
Les méthodes additionnelles	237
Aller plus loin	242
Répéter une texture	242
Afficher une caisse	249
La caméra	263
Fonctionnement	263
Une caméra déplaçable	263
La gestion de l'orientation	264
La gestion du déplacement	268
En résumé	277
Implémentation de la caméra	278
La classe Camera	278
Les Constructeurs et le Destructeur	281
La méthode orienter	283
La méthode deplacer	291
Méthode lookAt	296
Implémentation de la caméra	297
Fixer un point	299
Fonctionnalités de la classe Camera	300
Améliorations	308
La méthode setPosition()	308
Les attributs sensibilité et rapidité	308
TP : Une relique retrouvée	313
Les consignes	313
Les consignes	313
Quelques conseils	320
Correction	323
La boucle principale	323
La cabane	325
Le sol	331
Le cristal	338
Afficher le tout	342
Téléchargement	345
Partie 2 : Les notions avancées	345
Les Vertex Buffer Objects	346
C'est quoi un VBO ?	346
Création	348
Le header	348
La méthode charger()	349
Le destructeur	359
Utilisation	359
Utilisation d'un VBO	359
Utilisation de plusieurs VBO	363
Mise à jour des données	365
La méthode updateVBO()	365
Un autre exemple	369
Le header	369
La méthode charger()	370
La méthode afficher()	373
Les Vertex Array Objects	377
Encore un objet OpenGL ?	377
Création et Utilisation	380
Création	380
Utilisation	386
Un autre exemple	387
Le header	387
La méthode charger()	387
La méthode afficher()	389
OpenGL 3.3 et Mac OS X	391
Passer à OpenGL 3.3	391
OpenGL 3 sous Mac OS X	393
Créer un projet SDL2 sous Mac OS X	394

La compilation de shaders	407
Piqûre de rappel	408
Introduction	408
Rappel	408
Compilation des sources	410
La classe Shader	410
Les constructeurs et le destructeur	413
La méthode getProgramID	414
La méthode charger (Partie 1/2)	414
La méthode compilerShader	416
Création du programme	427
La méthode charger (Partie 2/2)	427
Utilisation et améliorations	436
Utilisation	436
Améliorations	438
OpenGL Shading Language	442
Un peu d'histoire	442
L'histoire des shaders	442
Les variables	443
Le code source minimal	443
Les Variables	444
Les constructeurs	448
Les structures de contrôle	453
Les structures de contrôles	453
Les fonctions	456
Divers	458
Fonctions prédéfinies	460
Les fonctions trigonométriques	460
Les fonctions relatives aux vecteurs	461
Autres fonctions	462
Les shaders démystifiés (Partie 1/2)	462
Préparation	463
Préparation	463
Premier shader	471
Affichage simple	471
Utilisation de couleur	478
Gestion de la couleur	482
Préparation	482
Gestions des shaders	485
Exercices	490
Les shaders démystifiés (Partie 2/2)	497
Les variables uniform	497
Qu'est-ce qu'une variable uniform ?	497
Envoyer une variable simple	497
Envoyer un vecteur	500
Envoyer une matrice	502
Envoyer un tableau	504
Ce qu'il faut retenir	506
Exercices	506
Gestion de la 3D	512
Préparation	512
Intégrer la troisième dimension	514
Gestion des textures	519
Afficher une texture	519
Ce qu'il faut retenir	525
Bonnes pratiques	526
Optimisation du Vertex Shader	526
La méthode envoyerMat4()	528
Les Frame Buffer Objects	530
Qu'est-ce qu'un FBO ?	531
Définitions	531
Fonctionnement	532
Programme du chapitre	535
Le Color Buffer	536
Les tâches	536
Créer un Color Buffer	536
Le problème de la copie de texture	540
Les Render Buffers	544
Les tâches	544
La classe FrameBuffer	544
La méthode creerRenderBuffer()	547
Le Frame Buffer	551
La méthode charger()	551
Petit rajout	564
Le destructeur	565
Quelques getters	566
Utilisation	567
Quelques explications	567
Première utilisation	568
Ce qu'il faut retenir	587
Améliorations simples	587
Éviter la création du Stencil Buffer	588

Le constructeur de copie	591
Gérer plusieurs Color Buffers	592



Développez vos applications 3D avec OpenGL 3.3



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos commentaires pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.

Par



Boouh

Mise à jour : 10/06/2013

Difficulté : Intermédiaire



Salut à tous les Zéros,



Vous souhaitez réaliser un **jeu-vidéo**, mais vous ne savez pas par où commencer ? Eh bien bonne nouvelle, vous êtes au bon endroit. Nous allons apprendre à utiliser une librairie graphique puissante : **OpenGL** (version 3.3). Nous découvrirons ensemble les bases essentielles de la programmation 3D qui pourront vous servir plus tard dans un projet de jeu **multi-plateforme**.

Sachez qu'avant d'aller plus loin, il vous faut un PC soit sous **Windows** soit sous une distribution **UNIX/Linux**. Pour les utilisateurs de Mac, c'est un peu spécial, mais pour continuer il vous faudra au moins avoir le système d'exploitation **OS X Lion (10.7)** d'installé. Si vous possédez un de ces trois OS, il vous faudra aussi une carte graphique compatible avec l'API OpenGL 3.3. Les plus anciennes cartes compatibles sont les **GeForce** de la série **8000** chez **NVidia** et les **Radeon HD** chez **ATI**. Si vous possédez une carte inférieure à celles-ci, vous pouvez toujours suivre le tutoriel de Kayl sur OpenGL 2.1 ici :

- [Tutoriel sur OpenGL 2.1](#)



Pour suivre ce tutoriel vous devez connaître le langage **C++** (sauf la partie **Qt**). La lecture du cours sur la **SDL** (jusqu'à la gestion des événements) n'est pas obligatoire mais conseillée.

Un grand merci à [Coyote](#) pour ses corrections qui ne doit pas s'ennuyer chaque fois que je lui envoie un énorme pavé à lire. 😊

Partie 1 : Les bases d'OpenGL 3

Cette première partie est consacrée à l'apprentissage des notions de base d'*OpenGL*. Nous commencerons pas afficher de simples formes 2D pour vous familiariser avec l'affichage de données à l'aide de la carte graphique, puis nous partirons sur des concepts plus évolués avec l'utilisation de matrices, l'intégration de la 3D, des textures, ...

Nous aborderons également quelques notions complexes comme les **shaders**, je les survolerai rapidement pour le moment pour ne pas vous faire fuir tout de suite. 😊

Nous les développerons ensemble dans la deuxième partie du tutoriel une fois que vous aurez acquis un peu d'expérience.

On commence gentiment en somme. 😊

Introduction

Dans ce chapitre introductif, je vais vous parler rapidement d'OpenGL, de ses atouts et de l'installation des différents outils que nous utiliserons tout au long de ce tutoriel.

Qu'est-ce qu'OpenGL ?

OpenGL est une librairie exploitant la **carte graphique** d'un ordinateur permettant ainsi aux développeurs de programmer des applications 2D et 3D. Cela permet en autres de développer des jeux-vidéo. L'avantage principal d'OpenGL est qu'il est **multi-plateforme**, c'est-à-dire qu'un programme codé avec OpenGL sera compatible avec Windows, Linux et Mac, sous réserve que la gestion de la fenêtre et des inputs soient également multi-plateforme (comme la SDL 😊). Nous utiliserons une version récente d'OpenGL qui est la version 3.3 (sortie le 11 mars 2010). Cependant, nous n'utiliserons que la 3.1 pour la première partie du tuto, vous verrez pourquoi au fur et à mesure.

Actuellement, deux API existent pour exploiter notre matériel graphique : DirectX (uniquement utilisable sous Windows) et OpenGL (multi-plateforme). Vous l'aurez compris, OpenGL est dans un sens plus intéressant du fait de sa portabilité.



(Screenshot issu du jeu "Minecraft" proposant un affichage OpenGL)

Un des autres avantages d'OpenGL est que son utilisation est totalement gratuite, vous pouvez très bien coder des programmes gratuits voire commerciaux sans rendre de compte à personne. Vous pouvez aussi fournir votre code source pour en faire profiter

la communauté 😊.

En plus d'être gratuite, cette librairie met à disposition son propre code source. Chacun d'entre nous peut aller voir le code source d'OpenGL librement; et c'est d'ailleurs ce que nous allons faire pour comprendre le fonctionnement de certaines fonctions 😊.

Dans l'introduction du tutoriel, je vous ai parlé du langage C++. Ce langage de programmation est le langage le plus utilisé dans le monde du jeu vidéo et c'est pour cette raison que nous allons l'utiliser.



La version la plus récente d'OpenGL est la version 4.x. Mais il n'y a que les nouvelles cartes graphiques qui peuvent en profiter, donc inutilisable pour la plupart d'entre nous 😕. De plus, nous devrions en théorie être capable d'utiliser la version 3.3 mais à partir de la version 3.2, l'apprentissage de l'API se complexifie un peu. Cependant nous passerons à la version 3.3 dans le futur. 😊

Le but de ce tutoriel est de vous apprendre les bases de la programmation d'un jeu vidéo. Et qui dit jeu vidéo, dit aussi ... **mathématiques** ! Alors oui je comprends que certains maudissent les maths au plus haut point à cause des mauvaises notes à l'école ou de son incompréhensible logique 😕. Mais si vous souhaitez vraiment développer un jeu vous ne passerez pas à coté .

N'ayez cependant pas peur des maths, il n'y a rien de compliqué dans ce que nous allons voir. Il suffit d'apprendre par cœur. Et si vous avez un trou de mémoire, vous pourrez toujours revenir voir les formules sur le site du zéro. Pas de contrôle non plus, donc pas de pression. La plupart des notions sont déjà expliquées dans les tutos du SdZ. Nous aborderons pas mal de domaines comme les vecteurs, les matrices, les quaternions, la trigonométrie, ... Que du bon en somme !

Bien, j'espère que la pilule est passée. 😊



Que faut-il télécharger ?

Ah bonne question. Tout d'abord, comme dit dans l'introduction, il vous faut une carte graphique compatible avec OpenGL 3.x, soit au minimum les **GeForce 8000** chez **NVidia** et les **Radeon HD** chez **ATI**. Ensuite, il faut mettre à jour vos drivers pour être sûr de ne pas avoir de problèmes plus tard.

Deuxièmement, il vous faut la librairie SDL installée. M@teo explique comment installer cette librairie dans son tuto, cependant la version donnée est **incompatible** avec la version d'OpenGL que nous allons utiliser. Il faudra donc passer par une autre version.

Installer les différents outils



Bien, passons à l'installation 😊 et commençons par la SDL :

Pourquoi a-t-on besoin de la SDL me direz-vous ?

La raison est simple : OpenGL a besoin d'un contexte d'exécution dans lequel travailler, il faut d'abord créer ce contexte avant de pouvoir créer notre monde 3D. De plus, OpenGL ne sait pas gérer les inputs (clavier, souris, ...), il nous faut donc une librairie capable de savoir ce que fait l'utilisateur. Heureusement pour nous, la librairie SDL (que vous devez déjà connaître 😊) sait créer un contexte OpenGL et gérer les évènements. En combinant la SDL et OpenGL nous nous retrouvons avec un monde 3D interactif.

Il y a d'autres librairies capables de remplir ce rôle mais l'avantage de la SDL est qu'elle est, elle-aussi, multi-plateformes. Les programmes codés avec ces deux librairies fonctionneront aussi bien sous Linux que sous Windows (ainsi que toutes les plates-formes gérant les deux librairies).

Actuellement, la version la plus stable de la SDL est la version 1.2.x mais depuis quelques temps la compatibilité de la SDL tend à s'étendre sur toutes les plateformes : sur iPhone, Androïd, et même sur PS3 ! Cette nouvelle version de la SDL est pour le moment la version 2.0. Elle est encore en développement et n'est pas stable à 100% mais elle le sera dans l'avenir. D'ailleurs, elle n'est pas disponible officiellement, notre principal problème va être de devoir la compiler par nous-même afin de pouvoir l'utiliser. 😊



Ne vous inquiétez pas ça va être très facile, les développeurs de la SDL sont intelligents, il nous suffit d'exécuter quelques commandes dans le terminal et hop on a la librairie compilée. 😊

Mais avant cela pour les utilisateurs de **Windows** : vérifiez bien que les drivers de votre carte graphique sont à jour. Pour le reste, je vais vous fournir directement les fichiers compilés à inclure dans le répertoire de votre IDE, pourvu qu'il soit équipé du **compilateur MinGW**. Pour les utilisateurs de **Visual C++**, vous devrez compiler la librairie SDL et télécharger la librairie GLEW.

Pour les utilisateurs de **Linux**, vérifiez également que les drivers de votre carte graphique (*Pilotes Propriétaires*) sont à jour.

Pour Mac OS X, ça va être un peu spécial. Dans un premier temps, il vous faut être obligatoirement sous **OS X 10.7 Lion ou plus**. Cependant, pour utiliser OpenGL chez vous il faudra utiliser des notions assez complexes que l'on ne verra que dans la deuxième partie du tuto. Je vous conseille donc d'utiliser une version libre de Linux (comme Ubuntu) ou Windows si vous avez bootcamp pour suivre ce début de tutoriel. Un aparté est prévu pour vous quand nous aurons appris tout ce qui est nécessaire pour coder sous Mac.

Pour en revenir à Windows, si vous êtes utilisateur de Visual C++ vous devriez compiler la librairie vous-même, je vous donne le lien pour que le faisiez sans problème. Vous devrez aussi télécharger la librairie GLEW.

Télécharger : [Librairie SDL 2.0 + GLEW + GLM - Pour Windows MinGW](#)

Télécharger : [Code Source SDL 2.0 + GLM - Pour UNIX/Linux et Visual C++](#)

Télécharger : [Code Source GLEW - Pour Visual C++](#)

Télécharger : [Code Source GLM - Pour Visual C++](#)

Pour **MinGW** sous Windows : dézippez l'archive et placez le dossier "**SDL-2.0**" dans le répertoire de **MinGW**. Si vous utilisez CodeBlocks, ce répertoire se trouve probablement dans *C:\Program Files (x86)\CodeBlocks\MinGW*. Attention cependant, placez bien le dossier "**SDL-2.0**" et pas ceux qui se trouvent à l'intérieur. Les dossiers "**dll**", "**bin**", "**include**" et "**lib**" doivent rester à l'intérieur.

Pour les linuxiens, téléchargez l'archive contenant le code source de la SDL et dézippez son contenu dans votre home (Exemple : /home/Boouh). Ensuite, ouvrez votre terminal et exécutez les commandes suivantes. Elles vont vous permettre de compiler puis d'installer la SDL :

Code : Console

```
sudo apt-get install libgl1-mesa-dev build-essential  
  
cd  
cd SDL-2.0/SDL-2.0.0-6713/
```

```
chmod +x configure
sudo ./configure
make
sudo make install

sudo ln -s /usr/local/bin/sdl2-config /usr/bin/sdl2-config
cd ..
sudo cp -r GL3/ /usr/local/include/
sudo cp -r glm/ /usr/local/include/
sudo chmod -R 715 /usr/local/include/GL3/
```

Grâce à ces commandes, vous avez maintenant la librairie SDL 2.0 installée sur votre ordinateur. 😊

Différences entre Windows et Linux

Dans l'archive pour Windows, j'ai inclus les fichiers compilés de la librairie GLEW. Pour ceux qui ne connaissent pas, GLEW est une librairie permettant de charger des extensions pour OpenGL (un peu comme les extensions des jeux **Sims**). Cette librairie est à l'origine utilisée sur Windows ET sur Linux. Mais avec la version 3.0 d'OpenGL, une grande partie des extensions de la version précédente sont devenues officielles et sont donc déjà incluses avec la version de "base".

Cependant, cette officialisation ne s'est pas faite sous Windows, il faudra donc toujours utiliser GLEW sous Windows. Nous verrons cela en détails un peu plus tard.

Pour les linuxiens, vous trouverez dans votre archive l'include "gl3.h" qui vient remplacer "gl.h". Vous utiliserez donc ce nouvel include (et non glew.h) pour les futurs chapitres.

Différences entre OpenGL 2.1 et 3.3

Alors là, je conseille à ceux qui connaissent OpenGL 2.1 de bien s'assoir au risque de tomber dans les pommes 🍎. Cette partie ne concerne pas uniquement ceux qui ont déjà codé avec OpenGL, vous ne comprendrez pas tout mais ça vous concerne aussi.

Tout d'abord, ce qu'il faut savoir c'est qu'avec la version 3, une grande partie des fonctions ont été marquées comme dépréciées. C'est-à-dire que le programmeur était fortement invité à ne plus les utiliser. Un peu plus tard, avec la version 3.1, le groupe Khronos a finalement décidé de supprimer ces fonctions dépréciées afin que les développeurs ne soient plus tentés de les utiliser.

Pourquoi a-t-on supprimé des fonctions me direz-vous ? Et bien tout simplement parce qu'elles ne sont plus adaptées aux jeux de nos jours. Soit elles ne sont plus utilisées, soit elles sont trop lentes. Imaginez une course de voitures avec des voitures qui vont à 30km/h ... Passionnant !



Ok des fonctions ont été supprimées, mais je ne vois pas ce qui peut me faire tomber dans les pommes. 🍎

Détrompez-vous, car certaines de ces fonctions étaient très utilisées avant.

Je vais prendre un exemple : **glVertex(...)**

Cette fonction permet avec OpenGL 2.1 de définir la position d'un point dans un espace 3D. Nous de notre coté, nous spécifions la position du point dans l'espace et la carte graphique s'occupait du reste. C'est-à-dire qu'elle multipliait les coordonnées du point par la matrice "modelView" puis par la matrice de projection, puis elle définissait sa couleur, etc ...

Maintenant c'est **NOUS** qui allons devoir faire TOUTES ces opérations.



QUOI ??? Mais c'est nul ! On se retrouve avec quelque chose de plus compliqué maintenant. 😞

Non c'est très bien au contraire puisqu'on se débarrasse d'une fonction lente, puis surtout, ça nous permet de faire ce que l'on veut. Pour reprendre l'exemple de la course, sans ces fonctions nous pourrons "tuner" notre voiture comme nous le voulons. Elle sera plus rapide, plus maniable et on se débarrassera de tout ce qui nous ralentit. Toutes les fonctions telles que `glVertex`, `glColor`, `glLightv` ... sont désormais inutilisables (et c'est tant mieux).

Ah oui, j'allais oublier, vous connaissez les matrices de projection et tout le reste ? Et bien comme vous le pensez (même si vous espérez vous tromper), ces matrices sont supprimées elles-aussi. Nous devrons donc créer notre propre matrice de projection, `modelview`, ...

La suppression des matrices entraîne également la suppression des fonctions `gluPerspective` et `gluLookAt`. Heureusement pour nous, il existe une librairie parallèle à OpenGL qui s'appelle **GLM** (pour **OpenGL Mathematics**). Cette librairie permet de faire pas mal de calculs mathématiques et permet surtout d'utiliser les matrices sans avoir à tout coder nous-même. Elle est incluse dans le téléchargement que vous avais fait juste avant. 😊

Vous vous dites peut-être que toutes ces suppressions sont injustes, tout est fait pour vous décourager. Eh bien non, ces suppressions ne peuvent être que positives car elles nous obligent à personnaliser complètement notre programme, nous pourrons donc mieux exploiter notre matériel et créer des jeux plus puissants.

Vous savez désormais ce qu'est OpenGL et avec quels outils nous allons travailler. J'expliquerai plus en détails le fonctionnement d'OpenGL dans une autre partie. Dans le chapitre suivant, nous allons écrire nos premières lignes de code, mais attention la 3D ce sera pour un peu plus tard. 😊

Créer une fenêtre avec la SDL 2.0

Dans ce chapitre nous allons créer une fenêtre avec la librairie SDL. Pour ceux qui connaissent la SDL avec le tuto de M@teo, vous pourrez comparer les deux versions. 😊

Préparation de l'IDE

Pour coder à travers ce tutoriel, je vous recommande d'utiliser l'IDE **Code::Blocks** qui est un outil très utilisé sur le site du Zéro. 😊 Vous pourrez donc trouver plus facilement de l'aide en cas de problème. Pour ceux qui souhaitent l'utiliser, voici un lien pour le télécharger (Windows uniquement). Il existe deux versions de cet IDE, vous pouvez choisir celle que vous voulez, les projets donnés devraient fonctionner sur les deux sans problème.

Télécharger : [Code::Blocks 12.11 \(Windows uniquement\)](#)

Pour les **linuxiens** :

Code : Console

```
sudo apt-get install codeblocks
```

D'ailleurs, si vous utilisez cet IDE je vais pouvoir vous fournir directement le **template** nécessaire pour créer un projet *SDL 2.0*. Ce template vous permettra de linker les librairies automatiquement sans que vous ayez à faire des manips compliquées. 😊

Télécharger : [Template SDL 2.0 Code::Blocks 10.05 et 12.11 - Windows et Linux](#)

Pour Windows, dézippez l'archive et **fusionnez** le dossier "share" de l'archive avec le dossier "share" de Code::Blocks (chez moi : **C:\Program Files (x86)\CodeBlocks\share**).

Pour Linux, dézippez l'archive où vous voulez puis exécutez les commandes suivantes (pas dans le dossier "share" mais dans celui qui le contient) :

Code : Console

```
sudo cp -r share/CodeBlocks/* /usr/share/codeblocks/
sudo chmod -R 715 /usr/share/codeblocks/templates/wizard/sdl2/
sudo chmod 715 /usr/share/codeblocks/templates/wizard/config.script
```



Si vous avez un problème pour utiliser les espaces dans Code::Blocks c'est certainement parce que vous avez un problème de clavier. Faites un tour dans les paramètres généraux pour configurer votre clavier en **Français(France)**.

Nous pouvons maintenant créer notre premier programme en SDL 2.0. 😊

Pour cela, il faut d'abord créer un projet **SDL 2.0** et non pas **OpenGL**, ne vous trompez pas ! Sous Code::Blocks la procédure est la suivante :

. File -> New -> Project -> **SDL 2.0 project**

Vous vous souvenez du dossier "**SDL-2.0**" du chapitre précédent ? Si vous ne l'avez pas encore installé au bon endroit faites-le maintenant 😊 (reportez-vous au chapitre précédent).

Occupons-nous maintenant du linkage d'*OpenGL*, il faut dire à notre IDE que nous souhaitons utiliser cette librairie sinon il va nous mettre plein d'erreurs au moment de la compilation. Voici la procédure à effectuer sous Code::Blocks :

. Project -> Build options -> Linker settings -> Add

Un petit encadré vous demande quelle librairie vous souhaitez linker, cela dépend de votre système d'exploitation. Vous ne pouvez renseigner qu'une seule librairie par encadré. Ré-appuyez sur le bouton **Add** pour en ajouter une nouvelle.

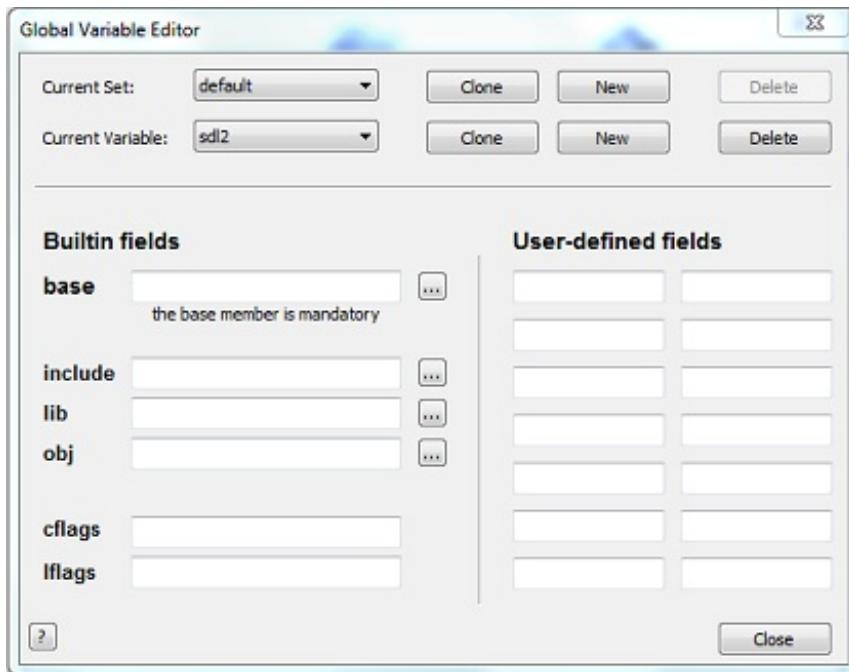
IDE	Option
Code::Blocks Windows	opengl32 glew32
Code::Blocks Linux	GL
DevC++	-lopengl32 -lglew32
Visual Studio	opengl32.lib glew32.lib

Selon votre OS, il suffit d'ajouter le ou les mot-clef(s) dans le petit encadré.



Pour ceux qui se demanderaient où est passée l'option de linkage pour GLU, rappelez-vous que nous n'utiliserons plus gluPerspective et gluLookAt, ces fonctions étant invalides avec la nouvelle version d'OpenGL.

Pour les utilisateurs de *CodeBlocks* sous **Windows** et uniquement sous **Windows**, Code::Blocks va certainement vous demander où se trouve la SDL avec cette popup :



Dans le champ **base** qui vous est proposé, renseignez le chemin vers le répertoire "**SDL-2.0**" que vous avez installé précédemment (chez moi : **C:\Program Files (x86)\CodeBlocks\MinGW\SDL-2.0**). Ne renseignez aucun autre champ et ça devrait être bon. Une popup vous indiquera sûrement un message du genre "Please select a valid location", mais ce sera une fausse alerte ne vous en faites pas, appuyez sur le bouton next. 😊

En parlant de ce dossier, si vous avez jeté un coup d'œil à l'intérieur vous vous apercevrez qu'il y a un sous-dossier nommé "**dll**". Ce dossier contient les fichiers **SDL2.dll** et **glew32.dll**, ils sont indispensables pour lancer vos futurs programmes.

Vous avez deux solutions pour les utiliser : soit vous les incluez dans le dossier de chaque projet, soit vous les copier dans le dossier "**bin**" du compilateur **MinGW** (chez moi : **C:\Program Files (x86)\CodeBlocks\MinGW\bin**). La première solution est la plus propre mais aussi la plus contraignante, faites comme bon vous semble mais il faut que vous utilisiez au moins une de ces techniques pour lancer vos programmes. 😊

Pour ceux qui utilisent d'autres IDE comme Visual C++ ou Eclipse vous allez devoir renseigner d'autres librairies car je n'ai pas de template tout fait à vous proposer. Il faudra que vous renseigneriez les dossiers où se trouvent les **librairies** et les **includes**. De plus, vous devrez linker manuellement deux librairies supplémentaires :



Premier programme

Nous avons tous les outils, ceux-ci sont tous configurés. Bien, commençons. 🎉

Code : C++

```
#include <SDL/SDL.h>
#include <iostream>

int main(int argc, char **argv)
{
    // Notre fenêtre et le clavier

    SDL_Surface *fenetre(0);
    SDL_Event evenements = {0};
    bool terminer(false);

    // Initialisation de la SDL

    if(SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return -1;
    }

    // Création de la fenêtre

    fenetre = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE);

    // Boucle principale

    while(!terminer)
    {
        SDL_WaitEvent(&evenements);

        if(evenements.type == SDL_QUIT)
            terminer = true;
    }
}
```

```
// On quitte la SDL  
SDL_Quit();  
return 0;  
}
```

Facile non ? Et bien maintenant, je vais vous demander d'oublier la moitié de ce code. 😊



Quoi ? C'est si différent que ça ?

Oui en effet, car souvenez-vous que nous travaillons avec la version 2.0. Et vu que l'on passe d'une version 1.x à une version 2.0, le code change beaucoup.

Commençons à coder avec la SDL 2.0 pour se mettre l'eau à la bouche :

Code : c++

```
#include <SDL2/SDL.h>  
#include <iostream>  
  
int main(int argc, char **argv)  
{  
    // Initialisation de la SDL  
  
    if(SDL_Init(SDL_INIT_VIDEO) < 0)  
    {  
        std::cout << "Erreur lors de l'initialisation de la SDL : "  
        << SDL_GetError() << std::endl;  
        SDL_Quit();  
  
        return -1;  
    }  
  
    // On quitte la SDL  
    SDL_Quit();  
  
    return 0;  
}
```

Bon jusqu'ici, pas de grand changement. Mis à part l'inclusion de la SDL qui passe de "**SDL/SDL.h**" à "**SDL2/SDL.h**". En même temps on ne peut pas changer grand chose. 😊

Voici le nouveau code qui permet de créer une fenêtre :

Code : c++

```
#include <SDL2/SDL.h>  
#include <iostream>  
  
int main(int argc, char **argv)  
{  
    // Notre fenêtre
```

```
SDL_Window* fenetre(0);

// Initialisation de la SDL

if(SDL_Init(SDL_INIT_VIDEO) < 0)
{
    std::cout << "Erreur lors de l'initialisation de la SDL : "
<< SDL_GetError() << std::endl;
    SDL_Quit();

    return -1;
}

// Création de la fenêtre

fenetre = SDL_CreateWindow("Test SDL 2.0",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,
SDL_WINDOW_SHOWN);

// On quitte la SDL

SDL_DestroyWindow(fenetre);
SDL_Quit();

return 0;
}
```

Nous voyons ici deux choses importantes, premièrement :

Code : C++

```
SDL_Window* fenetre;
```

Le pointeur **SDL_Window** remplacera désormais notre fenêtre, il n'y a donc plus de **SDL_Surface**. Maintenant notre fenêtre aura sa structure à part entière bien différente des surfaces classiques.

Deuxièmement :

Code : C++

```
fenetre = SDL_CreateWindow("Test SDL 2.0", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_SHOWN);
```

Vous l'aurez compris, cette fonction va nous permettre de créer notre fenêtre, elle vient donc remplacer notre bonne vieille **SDL_SetVideoMode**. Voici le prototype de cette fonction :

Code : C++

```
SDL_Window* SDL_CreateWindow(const char *title, int x, int y, int w,
int h, Uint32 flags);
```

- **title** : le titre de notre fenêtre
- **x** : l'abscisse de la position de la fenêtre, **SDL_WINDOWPOS_CENTERED** signifie que nous centrons la fenêtre par

rapport à l'axe x

- **y** : l'ordonnée de la position, `SDL_WINDOWPOS_CENTERED` signifie la même chose mais sur l'axe y
- **w** : (`width`) la largeur de la fenêtre
- **h** : (`height`) la hauteur de la fenêtre
- **flags** : ce paramètre est un peu spécial, il faudra toujours mettre `SDL_WINDOW_SHOWN`. Plus tard, ce sera ici que l'on indiquera à la `SDL` que nous souhaitons utiliser OpenGL

Cette fonction retourne notre fenêtre dans un pointeur de type **SDL_Window**.

Quant à la dernière fonction, son utilisation est simple : elle permet de détruire proprement notre fenêtre :

Code : C++

```
SDL_DestroyWindow(SDL_Window *window);
```

Nous lui donnerons la fenêtre créée au début du programme.

Code : C++

```
SDL_DestroyWindow(fenetre);
```

Puisque l'on parle de l'initialisation de la fenêtre, on va en profiter pour vérifier la valeur du pointeur **SDL_Window**. En effet dans de rares cas, celui-ci peut être nul à cause d'un éventuel problème logiciel ou matériel. On va donc tester sa valeur dans un **if**, s'il est nul alors on quitte la `SDL` en fournissant un message d'erreur, sinon c'est que tout va bien donc on continue :

Code : C++

```
// Cr ation de la fen tre

fenetre = SDL_CreateWindow("Test SDL 2.0", SDL_WINDOWPOS_CENTERED,
    SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_SHOWN);

if(fenetre == 0)
{
    std::cout << "Erreur lors de la creation de la fenetre : " <<
    SDL_GetError() << std::endl;
    SDL_Quit();

    return -1;
}
```

Ajoutons maintenant le code g rant les  v nements qui connaît, lui-aussi, son lot de modifications.

Avant, pour savoir si une fen tre devait se fermer, nous devions v rifier la variable type d'une structure `SDL_Event` comme ceci par exemple :

Code : C++

```
// Variables

SDL_Event evenements;
bool terminer(false);

// Gestion des  v nements

SDL_WaitEvent(&evenements);
```

```
if (evenements.type == SDL_QUIT)
    terminer = true;
```

Avec la SDL 2.0, la gestion des événements change quelques peu (nous verrons les différences un peu plus tard). Maintenant, pour savoir si on doit fermer une fenêtre, nous ferons ceci :

Code : C++

```
// Variables

SDL_Event evenements;
bool terminer(false);

// Gestion des évènements

SDL_WaitEvent(&evenements);

if (evenements.window.event == SDL_WINDOWEVENT_CLOSE)
    terminer = true;
```

Bien, résumons tout cela en un seul code. 😊

Code : C++

```
#include <SDL2/SDL.h>
#include <iostream>

int main(int argc, char **argv)
{
    // Notre fenêtre

    SDL_Window* fenetre(0);
    SDL_Event evenements;
    bool terminer(false);

    // Initialisation de la SDL

    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return -1;
    }

    // Création de la fenêtre

    fenetre = SDL_CreateWindow("Test SDL 2.0",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,
        SDL_WINDOW_SHOWN);

    if (fenetre == 0)
    {
        std::cout << "Erreur lors de la creation de la fenetre : "
        << SDL_GetError() << std::endl;
        SDL_Quit();
    }
}
```

```
    return -1;
}

// Boucle principale

while (!terminer)
{
    SDL_WaitEvent(&evenements);

    if (evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;
}

// On quitte la SDL

SDL_DestroyWindow(fenetre);
SDL_Quit();

return 0;
}
```

Les paramètres OpenGL

Tout ce code c'est bien mais il n'y a rien qui permet d'exploiter OpenGL, mais qu'attendons-nous ? 

Il y a pas mal d'attributs (des options de configuration) à paramétrer pour rendre notre programme compatible avec OpenGL, commençons par le plus important :

Code : C++

```
SDL_GLContext contexteOpenGL;
```

Cette structure va permettre à la SDL de créer un contexte OpenGL. Ce contexte est très important, pour utiliser l'API graphique et ses fonctions. Il faut tout d'abord le configurer.

Voici le prototype de la fonction qui permet de spécifier des attributs OpenGL à la SDL :

Code : C++

```
SDL_GL_SetAttribute(SDL_GLAttr attr, int value);
```

- **attr** : c'est notre attribut, nous verrons lesquels il faut spécifier
- **value** : c'est la valeur de l'attribut

Avec cette simple fonction, nous allons pouvoir spécifier à la SDL quelle version d'OpenGL on souhaite utiliser :

Code : C++

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);
```

Vous l'aurez compris :

- **SDL_GL_CONTEXT_MAJOR_VERSION** pour OpenGL 3.x
- **SDL_GL_CONTEXT_MINOR_VERSION** pour la version mineure : x.1 (Dans le futur, cette valeur sera de 3 lorsque nous

saurons manier OpenGL)



Petite question pour ceux qui connaissent la SDL 1.2, vous savez comment on utilise le double buffering ? 😊

Voici la réponse 😊 :

Code : C++

```
SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
```

Comme avec la SDL, on peut aussi activer le Double Buffering avec OpenGL. Cependant ça ne se passe pas de la même façon. Voici comment faire :

Code : C++

```
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
```

- **SDL_GL_DOUBLEBUFFER** : attribut activant le Double Buffering. Mettez 1 pour l'activer et 0 pour faire le contraire. 😊
- **SDL_GL_DEPTH_SIZE** : attribut de profondeur du Double Buffer. Nous mettons sa valeur à 24 pour une profondeur de 24 bits.



Bon on avance c'est bien 😊 . Mais tout à l'heure, tu as parlé d'une structure : **SDL_GLContext**, qu'est-ce qu'on en fait ?

Très bonne remarque, on va l'utiliser maintenant justement grâce à la fonction :

Code : C++

```
SDL_GLContext SDL_GL_CreateContext(SDL_Window *window);
```

Cette fonction demande un pointeur **SDL_Window** pour y attacher le contexte OpenGL. De plus, elle renvoie une structure **SDL_GLContext**, et justement c'est ce dont nous avons besoin 😊

Dans notre cas, nous lui donnerons notre fenêtre puis nous récupéreront la structure retournée :

Code : C++

```
contexteOpenGL = SDL_GL_CreateContext(fenetre);
```

Comme pour la fenêtre **SDL**, la création du contexte OpenGL peut lui aussi échouer. Le plus souvent ce sera parce que la version OpenGL demandée ne sera pas supportée par la carte graphique. Il faut donc tester la valeur de la variable **contexteOpenGL**. Si elle est égale à **0** c'est qu'il y a eu un problème. Dans ce cas on affiche un message d'erreur, on détruit la fenêtre puis on quitte la SDL :

Code : C++

```
// Création du contexte OpenGL

contexteOpenGL = SDL_GL_CreateContext(fenetre);

if(contexteOpenGL == 0)
{
    std::cout << SDL_GetError() << std::endl;
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return -1;
}
```

Si tout se passe bien lors de la création du contexte alors on continue.

Tout comme pour la fenêtre **SDL** encore une fois, il existe aussi une fonction qui permet de détruire le contexte lorsque nous n'en avons plus besoin. Son appel ressemble à ceci :

Code : C++

```
SDL_GL_DeleteContext(contexteOpenGL);
```

Bien, résumons tout cela :

Code : C++

```
#include <SDL2/SDL.h>
#include <iostream>

int main(int argc, char **argv)
{
    // Notre fenêtre

    SDL_Window* fenetre(0);
    SDL_GLContext contexteOpenGL(0);

    SDL_Event evenements;
    bool terminer(false);

    // Initialisation de la SDL

    if(SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return -1;
    }

    // Version d'OpenGL

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);

    // Double Buffer

    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
```

```
// Création de la fenêtre

fenetre = SDL_CreateWindow("Test SDL 2.0",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,
SDL_WINDOW_SHOWN);

if(fenetre == 0)
{
    std::cout << "Erreur lors de la creation de la fenetre : "
<< SDL_GetError() << std::endl;
    SDL_Quit();

    return -1;
}

// Création du contexte OpenGL

contexteOpenGL = SDL_GL_CreateContext(fenetre);

if(contexteOpenGL == 0)
{
    std::cout << SDL_GetError() << std::endl;
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return -1;
}

// Boucle principale

while(!terminer)
{
    SDL_WaitEvent(&evenements);

    if(evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;
}

// On quitte la SDL

SDL_GL_DeleteContext(contexteOpenGL);
SDL_DestroyWindow(fenetre);
SDL_Quit();

return 0;
}
```

On est presque au bout courage. 🎉 Il nous faut juste rajouter un paramètre dans la fonction `GL_CreateWindow`. Avec la SDL 1.2, ce paramètre ressemblait à ça :

Code : C++

```
SDL_SetVideoMode(..., ..., ..., SDL_OPENGL);
```

Avec la SDL 2.0, le nom change légèrement :

Code : C++

```
SDL_WINDOW_OPENGL
```

Il se place également dans la fonction qui crée la fenêtre :

Code : C++

```
fenetre = SDL_CreateWindow("Test SDL 2.0", SDL_WINDOWPOS_CENTERED,  
SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_SHOWN |  
SDL_WINDOW_OPENGL);
```

Et voilà notre code final :

Code : C++

```
#include <SDL2/SDL.h>  
#include <iostream>  
  
int main(int argc, char **argv)  
{  
    // Notre fenêtre  
  
    SDL_Window* fenetre(0);  
    SDL_GLContext contextOpenGL(0);  
  
    SDL_Event evenements;  
    bool terminer(false);  
  
    // Initialisation de la SDL  
  
    if(SDL_Init(SDL_INIT_VIDEO) < 0)  
    {  
        std::cout << "Erreur lors de l'initialisation de la SDL : "  
        << SDL_GetError() << std::endl;  
        SDL_Quit();  
  
        return -1;  
    }  
  
    // Version d'OpenGL  
  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);  
  
    // Double Buffer  
  
    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);  
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);  
  
    // Création de la fenêtre  
  
    fenetre = SDL_CreateWindow("Test SDL 2.0",  
    SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,  
    SDL_WINDOW_SHOWN | SDL_WINDOW_OPENGL);  
  
    if(fenetre == 0)  
    {  
        std::cout << "Erreur lors de la creation de la fenetre : "  
        << SDL_GetError() << std::endl;
```

```
    SDL_Quit();

    return -1;
}

// Cr ation du contexte OpenGL

contexteOpenGL = SDL_GL_CreateContext(fenetre);

if(contexteOpenGL == 0)
{
    std::cout << SDL_GetError() << std::endl;
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return -1;
}

// Boucle principale

while(!terminer)
{
    SDL_WaitEvent(&evenements);

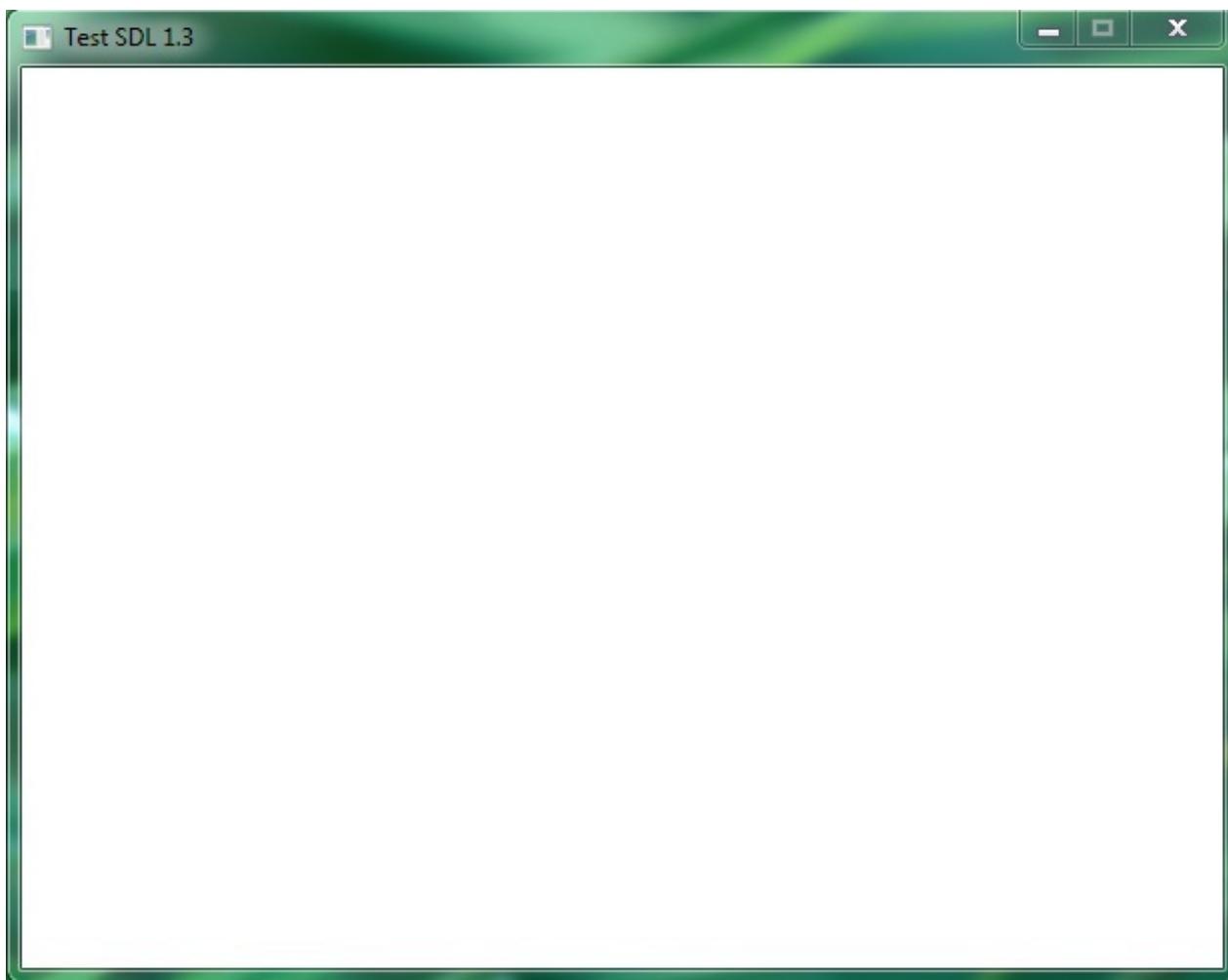
    if(evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;
}

// On quitte la SDL

SDL_GL_DeleteContext(contexteOpenGL);
SDL_DestroyWindow(fenetre);
SDL_Quit();

return 0;
}
```

Vous devriez obtenir quelque chose comme ceci :



Quoi ! Mais on a une fenêtre blanche c'est nul !

Du calme, vous n'imaginez pas ce que l'on vient de faire (enfin ce que la SDL vient de faire), nous venons de créer un contexte dans lequel OpenGL va évoluer. En gros nous venons de créer l'univers. 😊

Bref nous avons enfin une fenêtre SDL compatible avec OpenGL 3.1. Comparez le code avant et après, vous verrez la différence. Il y a plus de code certes, mais toutes ces instructions sont importantes. Ce nouvel environnement va nous permettre de créer des programmes plus paramétrables et donc plus puissants. 😊

Télécharger : [Code Source C++ du Chapitre 2](#)

Piouf, ça fait beaucoup de changements pour arriver grossièrement à la même chose. Mais au moins on passe à la nouvelle génération de programme.

Bon, on a une fenêtre c'est bien, mais pour le moment on a linké OpenGL pour rien. Passons maintenant à la partie OpenGL 😊.

Premier affichage

Nouveau chapitre, nouvelles notions.

Nous allons apprendre ici à faire nos premiers affichages dans la fenêtre **SDL**. Les notions que nous allons aborder nous seront utiles tout au long de ce tutoriel et constituent réellement la base du fonctionnement d'**OpenGL**. Vous aurez l'occasion de vous exercer à travers quelques exercices. Je ré-utiliserai ce procédé régulièrement car il n'y a rien de mieux que la pratique pour apprendre quelque chose.

Ceci étant dit, commençons dès maintenant ce nouveau chapitre. 😊

Le fonctionnement d'**OpenGL**

Nous allons commencer ce chapitre par un peu de théorie. 🧠

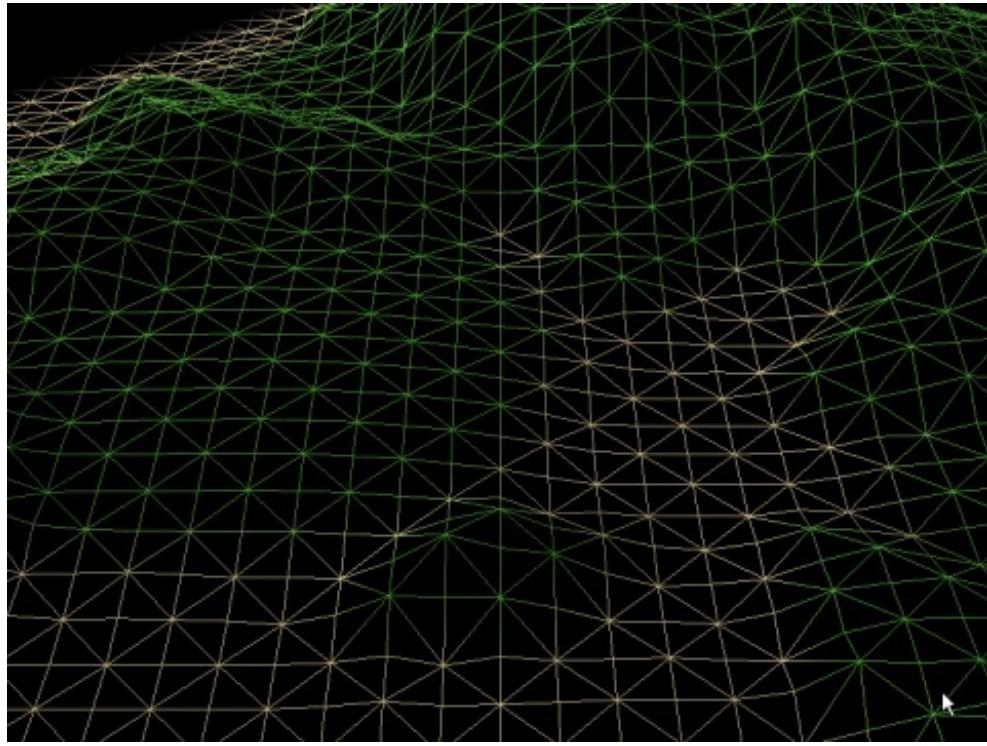


Comment fonctionne OpenGL ?

Basiquement, OpenGL fonctionne avec un système de "points" que nous plaçons dans un monde 3D. Nous appelons un point, un **vertex** (au pluriel : **vertices**), en français on peut dire un **sommet**. La première chose à faire avec ces vertices, c'est leur donner une position.

Une position dans un monde 3D est constituée de 3 coordonnées : **x** (la largeur), **y** (la hauteur) et **z** (la profondeur). Chaque vertex doit avoir ses propres coordonnées. Les vertices sont à la base de tout, ils forment tout ce que vous voyez dans un jeu vidéo : le héros, une arme, une voiture, des chaussettes ...

Une fois les vertices définis, on les relie entre eux pour former des formes géométriques simples : des carrés, des lignes, des triangles, ...



Voici comment OpenGL "voit" une map : une succession de vertices reliés entre eux pour former un terrain en mode filaire.

Maintenant que l'on a nos formes géométriques, il faut les "colorier", sinon nous n'aurions que des fils difformes qui ne ressembleraient à rien. Imaginez si on laissait tout comme ça, le jeu serait un peu particulier.

On peut distinguer deux formes de "coloriage": les **textures** et les **couleurs** (que nous spécifions nous-même). Les textures sont utilisées dans 99% des cas. En effet, nous ne spécifions que très rarement la couleur de nos objets directement, en général on le

fait pour des tests ou dans un shader (retenez bien ce mot, nous verrons cela plus tard). Dans les premiers chapitres nous n'utiliserons pas les textures, nous accorderons un chapitre entier pour cela.

Ce qu'il faut retenir c'est qu'au final nous définissons les coordonnées de nos vertices (sommets) dans l'espace, puis nous les relierons entre eux pour former une forme géométrique, enfin nous colorions cette forme pour avoir quelque chose de "consistant".

Nomenclature des fonctions OpenGL

Comme vous l'aurez remarqué avec la SDL, chaque fonction de la librairie commence par "SDL_". Bonne nouvelle, avec OpenGL c'est pareil, chaque fonction commence par la même chose : "gl".

Autre point, il se peut que vous soyez surpris en voyant le nom de certaines fonctions se répéter plusieurs fois, c'est normal.

Prenons un exemple : glUniform2f(...) et glUniform2i(...)

Nous verrons l'utilité de ces fonctions plus tard. Vous voyez la différence ? Dans la première, la dernière lettre est un "f" et dans la deuxième c'est un "i".

En général, la dernière lettre d'une fonction OpenGL sert à indiquer le type du paramètre à envoyer :

- **i** : integer (entier)
- **f** : float (réel)
- **d** : double (réel plus puissant)
- **ub** : unsigned byte (octet entre 0 et 255)
- ...

Selon le type de paramètre que vous enverrez il faudra utiliser la fonction correspondant au type de vos variables. 😊

Vous remarquez aussi que l'avant-dernière lettre est un chiffre, ce chiffre peut lui aussi changer. En général, si le chiffre est 1 on envoie un seul paramètre, si c'est 2 on en envoie deux, ...

Autre point que vous remarquerez plus tard : les types de variables OpenGL. Vous tomberez souvent sur des fonctions demandant des paramètres de types GLfloat, GLboolean, GLchar, ... Ce sont simplement des variables de type float, char, int, ... Le type GLboolean ne pourra prendre que deux valeurs : soit GL_FALSE (faux) soit GL_TRUE (vrai).

Boucle principale

Comme avec la SDL, OpenGL fonctionne avec une boucle principale. Tous les calculs se feront dans cette boucle. De plus, à chaque affichage il va falloir effacer l'écran car la scène aura légèrement changée, puis ré-afficher chaque élément un à un. Voici la fonction permettant d'effacer l'écran :

Code : C++

```
glClear(GLbitfield mask)
```

Dans un premier temps on effacera uniquement ce qui se trouve à l'écran grâce au paramètre : GL_COLOR_BUFFER_BIT.

Avec la SDL, pour actualiser l'affichage on utilisait la fonction `SDL_Flip()`, mais avec OpenGL on utilisera la fonction :

Code : C++

```
SDL_GL_SwapWindow(SDL_Window* window);
```

Le paramètre **window** étant notre structure `SDL_WindowID`. 😊

Malheureusement il existe encore une différence entre Linux et Windows. Comme vous l'avez vu dans le chapitre précédent, pour utiliser OpenGL, Windows est obligé de passer par une autre librairie du nom de GLEW. Cette librairie va nous permettre d'utiliser les fonctions d'OpenGL 3. Mais comme toute librairie, il va falloir l'initialiser. Sous Windows vous devrez inclure l'en-tête suivant :

Code : C++

```
#ifdef WIN32
#include <GL/glew.h>
```

Pour initialiser la librairie GLEW, il faudra ajouter la fonction : `glewInit()`. Comme toute librairie, l'initialisation peut échouer, il faut donc tester son initialisation :

Code : C++

```
// On initialise GLEW

GLenum initialisationGLEW( glewInit() );

// Si l'initialisation a échouée :

if(initialisationGLEW != GLEW_OK)
{
    // On affiche l'erreur grâce à la fonction :
    glewGetString(GLenum code)

    std::cout << "Erreur d'initialisation de GLEW : " <<
    glewGetString(initialisationGLEW) << std::endl;

    // On quitte la SDL

    SDL_GL_DeleteContext(contexteOpenGL);
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return -1;
}
```

Voilà pour Windows. Pour Linux c'est beaucoup plus simple, il suffit de placer une "define" spéciale pour indiquer que nous utiliserons les fonctions d'OpenGL 3 puis d'inclure le fichier d'en-tête "gl3.h" :

Code : C++

```
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>
```

Bien, récapitulons tout ceci :

Code : C++

```
#ifdef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
```

```
#include <GL3/gl3.h>

#endif

#include <SDL2/SDL.h>
#include <iostream>

int main(int argc, char **argv)
{
    /* *** Création de la fenêtre SDL *** */

    #ifdef WIN32

        // On initialise GLEW

        GLenum initialisationGLEW( glewInit() );

        // Si l'initialisation a échouée :

        if(initialisationGLEW != GLEW_OK)
        {
            // On affiche l'erreur grâce à la fonction :
            glewGetString(GLenum code)

                std::cout << "Erreur d'initialisation de GLEW : " <<
glewGetString(initialisationGLEW) << std::endl;

            // On quitte la SDL

            SDL_GL_DeleteContext(contexteOpenGL);
            SDL_DestroyWindow(fenetre);
            SDL_Quit();

            return -1;
        }

    #endif

    // Boucle principale

    while(!terminer)
    {
        // Gestion des évènements

        SDL_WaitEvent(&evenements);

        if(evenements.window.event == SDL_WINDOWEVENT_CLOSE)
            terminer = true;

        // Nettoyage de l'écran

        glClear(GL_COLOR_BUFFER_BIT);

        // Actualisation de la fenêtre

        SDL_GL_SwapWindow(fenetre);
    }

    // On quitte la SDL

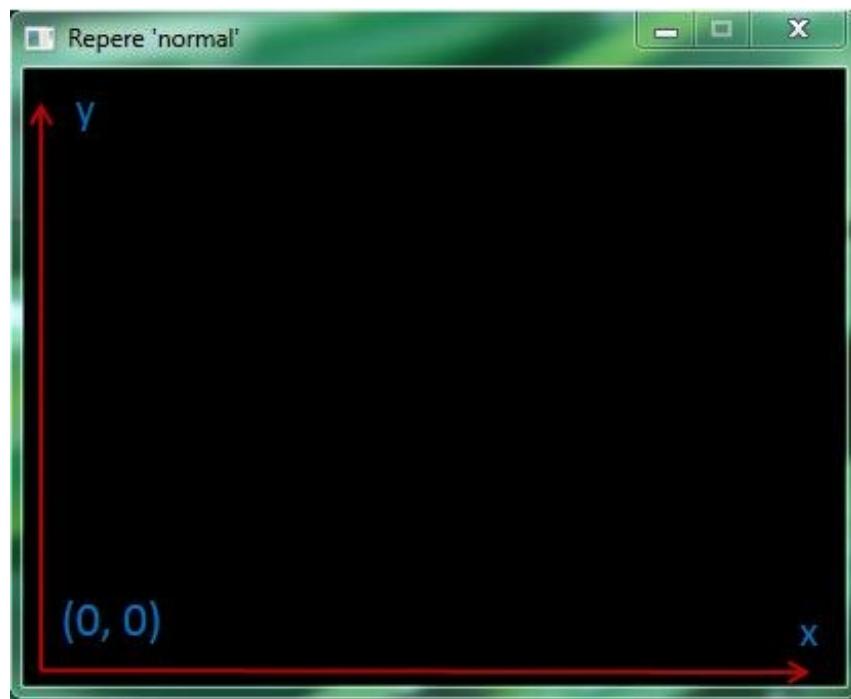
    SDL_GL_DeleteContext(contexteOpenGL);
```

```
    SDL_DestroyWindow(fenetre);  
    SDL_Quit();  
  
    return 0;  
}
```

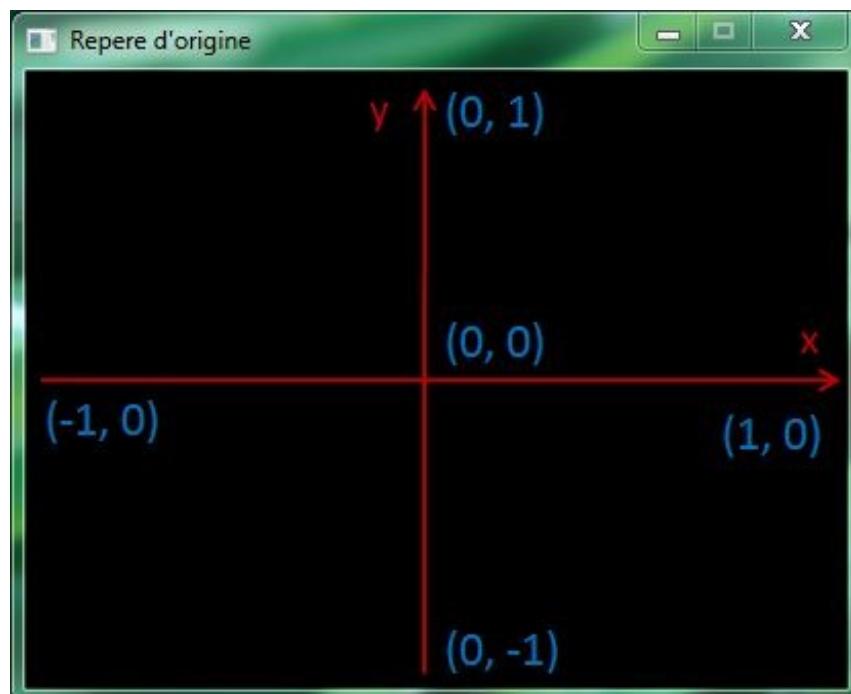
Repère et Origine

J'espère que vous connaissez la définition de ces deux termes. Un repère est un ensemble d'axes représentant au moins les axes X et Y, l'origine est le point de départ de ces axes.

En théorie voici ce que donne le repère d'OpenGL :



Mais dans un premier temps, nous utiliserons le repère par défaut d'OpenGL que voici :



L'origine du repère se trouve au centre de l'écran, et les coordonnées maximales affichables sont comprises entre (-1, -1) et (1, 1).

Nous utiliserons l'autre repère dès que nous y aurons inclus les matrices 😊. De plus, nous serons dans un premier temps dans un monde 2D, nous ne ferons pas de cube ou autre forme complexe pour commencer. Nous verrons cela un peu plus tard. 🍪

Afficher un triangle

Afficher un triangle

Maintenant que le blabla théorique est terminé, nous pouvons reprendre la programmation. 🎉

Dans le chapitre précédent, nous avons appris que tous les modèles 3D présents dans un jeu étaient constitués de sommets (vertices), puis qu'il fallait les relier pour former une surface. Et si on assemble ces formes, ça donne nos modèles 3D.

Notre premier exercice est simple : afficher un triangle. Pour pouvoir afficher n'importe quel modèle 3D il faut déjà spécifier ses vertices que nous allons ensuite donner à OpenGL. En général, on définit nos vertices dans un seul tableau, on place chaque coordonnée de vertex à la chaîne comme ceci :

Code : C++

```
float vertices = {X_vertex_1, Y_vertex_1, Z_vertex_1,     X_vertex_2,
                  Y_vertex_2, Z_vertex_2,     ...};
```

Dans un triangle il y a trois sommets, nous aurons donc 3 vertices.

Code : C++

```
float vertex1[] = {-0.5, -0.5};
float vertex2[] = {0.0, 0.5};
float vertex3[] = {0.5, -0.5};
```

N'oubliez pas que nous sommes en 2D pour l'instant, nous n'avons donc que les coordonnées x et y à définir. Ce début de code est bien mais si on utilise 3 tableaux, ça ne fonctionnera pas. Nous devons combiner les trois tableaux pour en former un seul :

Code : C++

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

Maintenant, il faut envoyer ces coordonnées à OpenGL. Pour envoyer des informations nous avons besoin d'un tableau appelé "**Vertex Attribut**". Pour ceux qui connaissent les "VertexArrays", c'est la même chose 😊. Sauf que maintenant ce n'est plus une optimisation mais belle et bien la méthode de base pour envoyer nos infos à OpenGL.

Ces "**Vertex Attributs**" sont déjà inclus dans l'API, il suffit de lui donner des valeurs (nos coordonnées) puis de l'activer. Voici le prototype de la fonction gérant les VertexAttributs :

Code : C++

```
void glVertexAttribPointer(GLuint index, GLuint size, GLenum type,
                           GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

- **index** : c'est le numéro du tableau, son identifiant.
- **size** : c'est le nombre de coordonnées par vertex. En 2D ce sera 2 et en 3D ce sera 3.
- **type** : c'est le type de données (float, int, ...).
- **normalized** : booléen permettant à OpenGL de normaliser les données (comme les vecteurs).
- **stride** : paramètre spécial, on le mettra à zéro tout le temps, nous ne l'utiliserons pas.
- **pointer** : c'est le pointeur sur nos données, ici nos coordonnées.

Voyons ce que ça donne avec les données de notre triangle :

Code : C++

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);
```

Malheureusement, en l'état, notre tableau est inutilisable, il faut d'abord l'activer avec la fonction :

Code : C++

```
void glEnableVertexAttribArray(GLuint index);
```

Le paramètre **index** étant notre identifiant de tableau, ici ce sera 0.

Bien, maintenant OpenGL sait quels vertices il doit afficher, il ne manque plus qu'à lui dire ... ce qu'il doit faire de ces points. 

Pour ça, on utilise une fonction (il y en a en fait deux, mais pour le début de ce tutoriel nous n'utiliserons que la première). Cette fonction permet de dire à OpenGL quelle forme afficher avec les points donnés précédemment. Voici le prototype de la fonction :

Code : C++

```
glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- **mode** : C'est la forme finale (nous verrons cela juste après).
- **first** : C'est l'indice de notre premier vertex à afficher (nous lui donnerons un int).
- **count** : C'est le nombre de vertices à afficher depuis **first** (nous lui donnerons également un int).

Alors Attention ! :

- Le paramètre **first** est un indice (comme un indice de tableau), si vous voulez utiliser le premier vertex vous lui donnerez la valeur "0" et pas "1".
- Pour **count**, c'est le contraire. Lui il veut le nombre de vertices à utiliser depuis **first**. Si vous utilisez 4 vertices vous lui donnerez la valeur "4" et pas "3".

Exemple : J'ai 4 vertices et je veux afficher un carré. Le premier vertex sera le vertex 0 (le premier vertex), et la valeur de "count" sera 4 car j'ai besoin du vertex 0 + les 3 vertices qui le suivent.

Passons au paramètre le plus intéressant : **mode**.

Ce paramètre peut prendre plusieurs formes donc voici les principales :

Valeur	Définition
GL_TRIANGLES	Avec ce mode, chaque groupe de 3 vertices formera un triangle. C'est le mode le plus utilisé.

GL_POLYGON	Tous les vertices s'assemblent pour former un polygone de type convexe (Hexagone, Heptagone, ...).
GL_LINES	Chaque duo de vertices formera une ligne.
GL_TRIANGLE_STRIP	Ici les triangles s'assembleront. Les deux derniers vertices d'un triangle s'assembleront avec un 4ème vertex pour former un nouveau triangle.

Toutes les valeurs possibles ne sont pas représentées mais je vous expose ici les plus utilisées. 😊

Revenons à notre code, nous avons désormais les différents modes d'affichage. Le but de ce chapitre est d'afficher un triangle, notre mode d'affichage sera donc : GL_TRIANGLES, ce qui donne :

Code : C++

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

La valeur "0" pour commencer l'affichage par le premier vertex, et le "3" pour utiliser les 3 vertices dont le premier sera le vertex "0".

Récapitulons tout ça :

Code : C++

```
// Vertices et coordonnées

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

// Boucle principale

while(!terminer)
{
    // Gestion des évènements

    ...

    // On remplit puis on active le tableau Vertex Attrib 0

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);
    glEnableVertexAttribArray(0);

    // On affiche le triangle

    glDrawArrays(GL_TRIANGLES, 0, 3);

    // On désactive le tableau Vertex Attrib puisque l'on en a plus
    // besoin

    glDisableVertexAttribArray(0);

    // Actualisation de la fenêtre

    ...
}
```

Vous avez dû remarquer la présence d'une nouvelle fonction : **glDisableVertexAttribArray**. Elle permet de désactiver le tableau VertexAttrib utilisé, le paramètre est le même que celui de la fonction **glEnableVertexAttribArray**. On désactive le tableau juste

après l'affichage des vertices.

Récapitulons tout avec le code SDL :

Code : C++

```
#ifdef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

#include <SDL2/SDL.h>
#include <iostream>

int main(int argc, char **argv)
{
    // Notre fenêtre

    SDL_Window* fenetre(0);
    SDL_GLContext contexteOpenGL(0);

    SDL_Event evenements;
    bool terminer(false);

    // Initialisation de la SDL

    if(SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return -1;
    }

    // Version d'OpenGL

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);

    // Double Buffer

    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);

    // Création de la fenêtre

    fenetre = SDL_CreateWindow("Test SDL 2.0",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,
        SDL_WINDOW_SHOWN | SDL_WINDOW_OPENGL);

    if(fenetre == 0)
    {
        std::cout << "Erreur lors de la creation de la fenetre : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return -1;
    }
```

```
// Création du contexte OpenGL

contexteOpenGL = SDL_GL_CreateContext(fenetre);

if(contexteOpenGL == 0)
{
    std::cout << SDL_GetError() << std::endl;
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return -1;
}

#ifndef WIN32

    // On initialise GLEW

    GLenum initialisationGLEW( glewInit() );

    // Si l'initialisation a échouée :

    if(initialisationGLEW != GLEW_OK)
    {
        // On affiche l'erreur grâce à la fonction :
        glewGetString(GLenum code)

            std::cout << "Erreur d'initialisation de GLEW : " <<
glewGetString(initialisationGLEW) << std::endl;

        // On quitte la SDL

        SDL_GL_DeleteContext(contexteOpenGL);
        SDL_DestroyWindow(fenetre);
        SDL_Quit();

        return -1;
    }

#endif

// Vertices et coordonnées

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

// Boucle principale

while(!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&evenements);

    if(evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT);

    // On remplit puis on active le tableau Vertex Attrib 0

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
```

```
vertices);
    glEnableVertexAttribArray(0);

    // On affiche le triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);

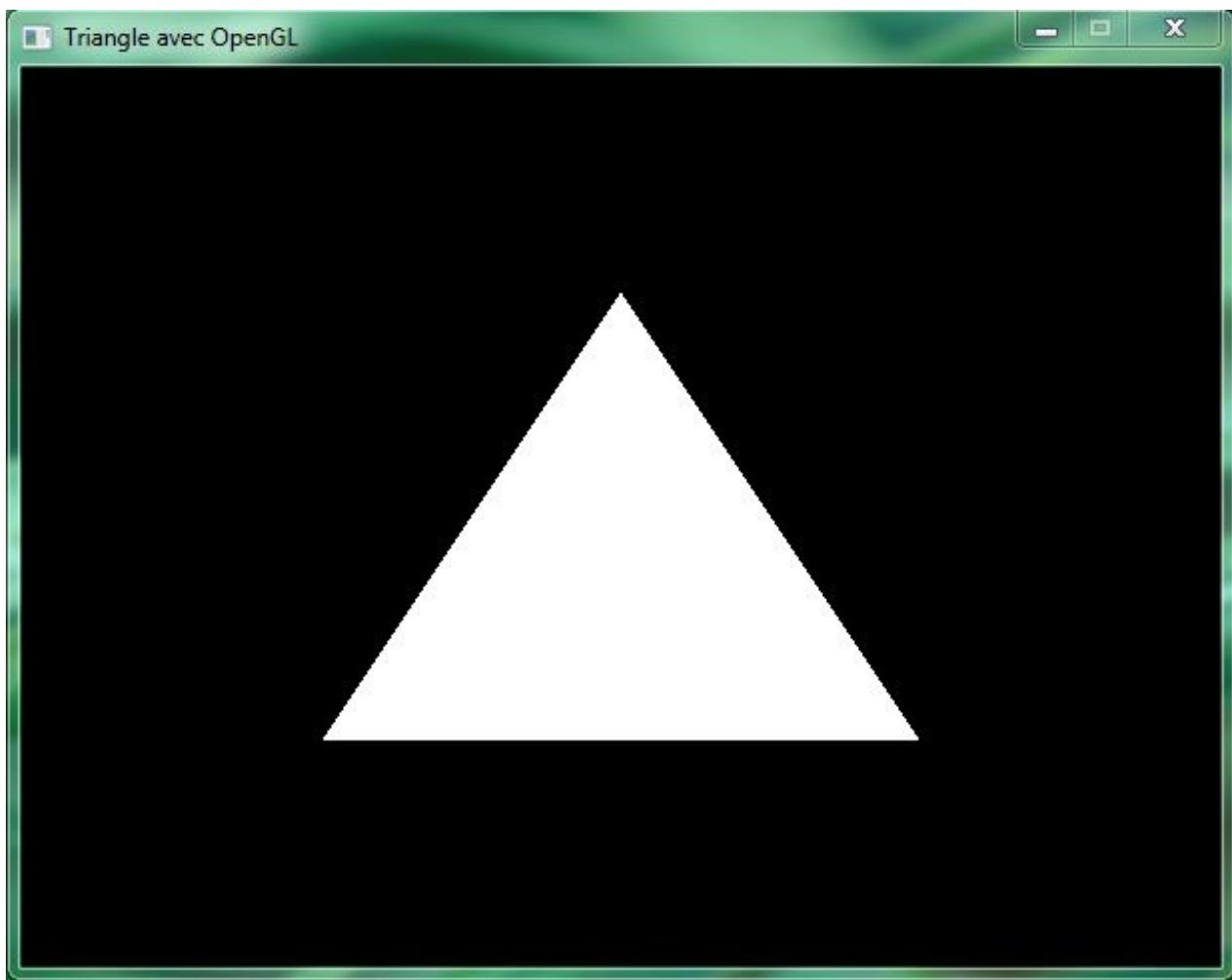
    // On désactive le tableau Vertex Attrib puisque l'on n'en
    // a plus besoin
    glDisableVertexAttribArray(0);

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow(fenetre);
}

// On quitte la SDL
SDL_GL_DeleteContext(contexteOpenGL);
SDL_DestroyWindow(fenetre);
SDL_Quit();

return 0;
}
```

Si tout se passe bien, vous devriez avoir une belle fenêtre comme celle-ci :



Afficher plusieurs triangles

Les vertices

Comme nous l'avons vu dans le tableau précédemment, le paramètre **GL_TRIANGLES** dans la fonction **glDrawArrays()** permet d'afficher un triangle pour chaque *triplet* de vertices.

Pour le moment, nous n'utilisons que **3** vertices, donc nous n'avons au final qu'un seul triangle. Or si nous en utilisions **6** nous aurons alors deux triangles.

On peut même aller plus loin et prendre 60, 390, ou 3000 vertices pour en afficher plein ! C'est d'ailleurs ce que font les décors et les personnages dans les jeux-vidéo, ils ne savent utiliser que ça. 😊

Enfin, prenons un exemple plus simple avant d'aller aussi loin. Si nous voulons afficher le rendu suivant ... :

Image utilisateur

... Nous devrons utiliser deux triplets de vertices.

Ce qu'il faut savoir, c'est qu'il ne faut surtout pas utiliser un tableau pour chaque triangle. On perdrait beaucoup trop de temps à tous les envoyer. A la place, nous devons inclure tous les vertices dans un seul et unique tableau :

Code : C++

```
// Vertices

float vertices[] = {0.0, 0.0, 0.5, 0.0, 0.0, 0.5, // Triangle 1
                     -0.8, -0.8, -0.3, -0.8, -0.8, -0.3}; // Triangle 2
```

Toutes nos données sont regroupées dans un seul tableau. Ça ne nous fait qu'un seul envoi à faire c'est plus facile à gérer, ça arrange même OpenGL. 😊

L'affichage

Au niveau de l'affichage, le code reste identique à celui du triangle unique. C'est-à-dire qu'il faut utiliser les fonctions :

- **glVertexAttribPointer()** : pour donner les vertices à OpenGL
- **glEnableVertexAttribArray()** : pour activer le tableau **Vertex Attrib**
- **glDrawArrays()** : pour afficher le tout

La seule différence notable va être la valeur du paramètre **count** (nombre de vertices à prendre en compte) de la fonction **glDrawArrays()**. Celui-ci était égal à **3** pour un seul triangle, nous la passerons désormais à **6** pour en afficher deux. L'appel à la fonction ressemblerait donc à ceci :

Code : C++

```
// Affichage des triangles

glDrawArrays(GL_TRIANGLES, 0, 6);
```

Ce qui donne le code source de la boucle principale suivant :

Code : C++

```
// Vertices

float vertices[] = {0.0, 0.0, 0.5, 0.0, 0.0, 0.5, // Triangle 1
                     -0.8, -0.8, -0.3, -0.8, -0.8, -0.3}; // Triangle 2

// Boucle principale

while (!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&evenements);

    if (evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

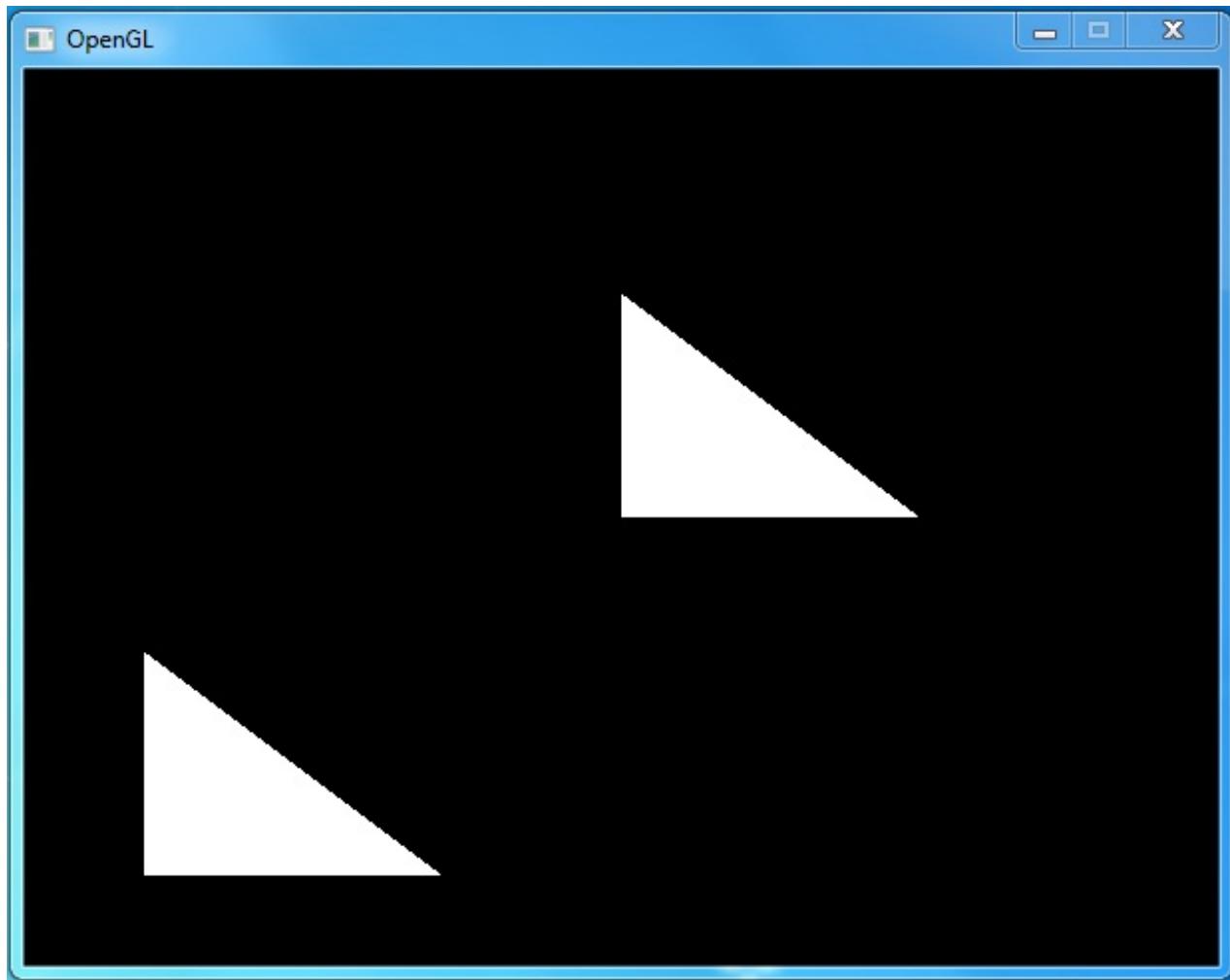
    glClear(GL_COLOR_BUFFER_BIT);

    // On remplit puis on active le tableau Vertex Attrib 0

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);
    glEnableVertexAttribArray(0);
```

```
// On affiche des triangles  
  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// On désactive le tableau Vertex Attrib puisque l'on n'en a  
plus besoin  
  
glDisableVertexAttribArray(0);  
  
// Actualisation de la fenêtre  
  
SDL_GL_SwapWindow(fenetre);  
}
```

Si vous compilez ce code, vous devriez obtenir :



Le tour est joué. 😊

La classe SceneOpenGL

Je ne sais pas si vous l'avez remarqué, mais depuis le début du tutoriel nous n'avons pas codé une seule classe. À vrai dire c'est normal, nous n'en avions pas vraiment besoin jusqu'ici. Cependant nos programmes vont commencer à se complexifier donc autant prendre les bonnes habitudes dès maintenant.

La classe SceneOpenGL

Le header

Pour le moment, nous n'aurons besoin que d'une seule classe dans nos programmes de test. Elle devra être capable de remplir plusieurs rôles :

- Créer la fenêtre SDL et le contexte OpenGL.
- Initialiser tous les objets d'une scène (personnages, caisses, sol, ... Nous verrons cela un peu plus tard 😊).
- Gérer l'interaction entre les objets tout au long du programme.

Pour remplir ces objectifs, nous allons créer plusieurs méthodes, mais avant cela nous devons déclarer la classe C++ qui s'occupera de tout ça. Cette classe s'appellera : **SceneOpenGL**. Commençons donc par créer deux fichiers **SceneOpenGL.h** et **SceneOpenGL.cpp**.

Une classe en C++ est composée de méthodes et d'attributs. Pour les méthodes, on voit déjà à peu près ce que l'on va faire. En revanche, on ne sait pas encore de quels attributs nous aurons besoin.

Si on regarde le début du code pour afficher le triangle blanc, nous voyons trois variables importantes :

Code : C++

```
// Variables

SDL_Window* fenetre();
SDL_GLContext contexteOpenGL();
SDL_Event evenements;
```



Le booléen **terminer** ne sera pas utilisé en tant qu'attribut mais en temps que simple variable dans une méthode.

On voit dans cette liste 3 variables correspondant :

- À la fenêtre
- Au contexte OpenGL
- Aux évènements SDL



On rajoutera le préfixe "**m_**" à chacune de ces variables pour bien différencier les attributs des variables normales.

Ces 3 variables deviendront les attributs de notre classe. On en rajoutera même trois supplémentaires qui correspondront :

- Au titre de la fenêtre.
- À sa largeur
- À sa hauteur

Si on résume tout ça, nous avons une classe **SceneOpenGL** avec **6 attributs**. Le header ressemblera donc à ceci :

Code : C++

```
#ifndef DEF_SCENEOPENGL
#define DEF_SCENEOPENGL

#include <string>
```

```
class SceneOpenGL
{
public:
    SceneOpenGL();
    ~SceneOpenGL();

private:
    std::string m_titreFenetre;
    int m_largeurFenetre;
    int m_hauteurFenetre;

    SDL_Window* m_fenetre;
    SDL_GLContext m_contexteOpenGL;
    SDL_Event m_evenements;
};

#endif
```

Et voici donc le squelette de tous nos futurs programmes. 😊

Bon pour le moment c'est un peu vide, on va habiller un peu tout ça. On va notamment ajouter deux éléments. Premièrement, il faut ajouter tous les includes concernant la **SDL** et **OpenGL** :

Code : C++

```
// Includes

#ifndef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

#include <SDL2/SDL.h>
#include <iostream>
```

Deuxièmement, il faut modifier le *constructeur* de la classe pour prendre en compte les paramètres de création de la fenêtre (titre, largeur et hauteur) :

Code : C++

```
SceneOpenGL(std::string titreFenetre, int largeurFenetre, int
hauteurFenetre);
```

Pour le moment, rien de bien compliqué. 😊

Les méthodes

Si on reprend la liste des objectifs de la classe, on retrouve trois points importants :

- Créer la fenêtre SDL et le contexte OpenGL.
- Initialiser tous les objets d'une scène (personnages, caisses, sol, ...).
- Gérer l'interaction entre les objets tout au long du programme.

Nous allons créer une méthode pour chaque point de cette liste.

La première méthode consistera donc à initialiser la fenêtre et le contexte OpenGL dans lequel nous allons évoluer :

Code : C++

```
bool initialiserFenetre();
```

Elle renverra un **booléen** pour confirmer ou non la création de la fenêtre. Nous mettrons à l'intérieur tout le code permettant de générer la fenêtre.

Pour le deuxième point, nous devrons initialiser tout ce qui concerne *OpenGL* (mis à part le contexte vu qu'il est créé juste avant). Pour le moment, nous n'avons que la bibliothèque **GLEW** à initialiser.

Généralement, la fonction qui s'occupe d'initialiser *OpenGL* s'appelle **initGL()**, nous appellerons donc notre méthode de la même façon :

Code : C++

```
bool initGL();
```

Comme la méthode précédente, elle renverra un **booléen** pour savoir si l'initialisation s'est bien passée.

Pour le troisième et dernier point, nous devons gérer la boucle principale du programme. Nous créerons donc une méthode **bouclePrincipale()** qui s'occupera de gérer tout ça :

Code : C++

```
void bouclePrincipale();
```

Elle ne renverra aucune valeur.

Résumé du Header

Si on met tout ce que l'on vient de voir dans le header, on a : une classe **SceneOpenGL**, des **includes** pour gérer la SDL et OpenGL, **6 attributs** et **3 méthodes** :

Code : C++

```
#ifndef DEF_SCENEOPENGL
#define DEF_SCENEOPENGL

// Includes

#ifndef WIN32
#include <GL/glew.h>
```

```
#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

#include <SDL2/SDL.h>
#include <iostream>
#include <string>

// Classe

class SceneOpenGL
{
public:

    SceneOpenGL(std::string titreFenetre, int largeurFenetre, int
hauteurFenetre);
    ~SceneOpenGL();

    bool initialiserFenetre();
    bool initGL();
    void bouclePrincipale();

private:

    std::string m_titreFenetre;
    int m_largeurFenetre;
    int m_hauteurFenetre;

    SDL_Window* m_fenetre;
    SDL_GLContext m_contexteOpenGL;
    SDL_Event m_evenements;
};

#endif
```

Implémentation de la classe

Dans cette dernière sous-partie, on va se reposer un peu. En effet, on a déjà tout codé avant, on n'a plus qu'à jouer au puzzle en coupant notre code et en mettant les bons morceaux au bon endroit. 🧩

Constructeur et Destructeur

Pour le constructeur rien de plus simple, on initialise nos attributs sans oublier de passer les 3 paramètres du constructeur concernant la fenêtre :

Code : C++

```
SceneOpenGL::SceneOpenGL(std::string titreFenetre, int
largeurFenetre, int hauteurFenetre) : m_titreFenetre(titreFenetre),
m_largeurFenetre(largeurFenetre),
m_hauteurFenetre(hauteurFenetre), m_fenetre(0), m_contexteOpenGL(0)
{}
```

Pour le destructeur, on détruit simplement le contexte et la fenêtre, puis on quitte la librairie SDL :

Code : C++

```
SceneOpenGL::~SceneOpenGL()
{
    SDL_GL_DeleteContext(m_contexteOpenGL);
    SDL_DestroyWindow(m_fenetre);
    SDL_Quit();
}
```



N'oubliez pas de rajouter le préfixe "**m_**" aux attributs quand vous copiez le code.

Les méthodes `initialiserFenetre()` et `initGL()`

Ici on sait déjà ce que l'on va mettre. On va implémenter le code permettant de créer la fenêtre et le contexte OpenGL, nous connaissons ce code depuis le chapitre précédent. Pour migrer celui-ci on va :

- Prendre tout le code gérant l'initialisation de la SDL et du contexte.
- Remplacer les deux occurrences de **fenetre** par "**m_fenetre**" et de **contexteOpenGL** par "**m_contexteOpenGL**".
- Remplacer les valeurs rentrées par "**true**" pour **0** et "**false**" pour **-1**.
- Remplacer le paramètre **title** de la fonction **SDL_CreateWindow()** par la chaîne de caractères de l'attribut **m_titreFenetre** soit **m_titreFenetre.c_str()**.

Encore une fois rien de bien compliqué. 😊

En code, ça nous donne ceci :

Code : C++

```
bool SceneOpenGL::initialiserFenetre()
{
    // Initialisation de la SDL

    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return false;
    }

    // Version d'OpenGL

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);

    // Double Buffer

    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);

    // Création de la fenêtre
```

```
m_fenetre = SDL_CreateWindow(m_titreFenetre.c_str(),
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, m_largeurFenetre,
m_hauteurFenetre, SDL_WINDOW_SHOWN | SDL_WINDOW_OPENGL);

if(m_fenetre == 0)
{
    std::cout << "Erreur lors de la creation de la fenetre : "
<< SDL_GetError() << std::endl;
    SDL_Quit();

    return false;
}

// Création du contexte OpenGL

m_contexteOpenGL = SDL_GL_CreateContext(m_fenetre);

if(m_contexteOpenGL == 0)
{
    std::cout << SDL_GetError() << std::endl;
    SDL_DestroyWindow(m_fenetre);
    SDL_Quit();

    return false;
}

return true;
}
```

On passe maintenant à la méthode `initGL()`. Pour le moment, nous n'avons pas grand chose à mettre à l'intérieur mise à part l'initialisation de la librairie **GLEW** pour Windows.

Et comme précédemment, il va falloir modifier le nom des attributs **fenetre** et **contexteOpenGL** en leur ajoutant le prefix "**m_**". Nous rajouterons également deux **return** dans la méthode :

- Un si l'initialisation échoue (donc **return false**).
- Et l'autre pour indiquer que l'initialisation s'est bien déroulée (**return true**).

Code : C++

```
bool SceneOpenGL::initGL()
{
#ifdef WIN32

    // On initialise GLEW

    GLenum initialisationGLEW( glewInit() );

    // Si l'initialisation a échoué :

    if(initialisationGLEW != GLEW_OK)
    {
        // On affiche l'erreur grâce à la fonction :
        glewGetString(GLenum code)

        std::cout << "Erreur d'initialisation de GLEW : " <<
        glewGetString(initialisationGLEW) << std::endl;

        // On quitte la SDL

        SDL_GL_DeleteContext(m_contexteOpenGL);
    }
}
```

```
        SDL_DestroyWindow(m_fenetre);
        SDL_Quit();

        return false;
    }

#endif

// Tout s'est bien passé, on retourne true

return true;
}
```

La méthode bouclePrincipale()

Allez on passe à la dernière méthode, c'est dans celle-ci que va se passer la quasi totalité du programme.

Dans cette méthode, on commence par déclarer le booléen **terminer** que vous devez déjà connaître. 🍪 On ne l'a pas déclaré en temps qu'attribut car il ne sert que dans la boucle **while**.

Après ce booléen, on va en profiter pour déclarer notre fameux tableau de vertices. Dans le futur, nous ferons des objets spécialement dédiés pour afficher nos modèles. Mais pour le moment, nous n'avons que de simples vertices à afficher, on peut donc se passer de classe.

Voici donc le début de la méthode **bouclePrincipale()** :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    bool terminer(false);
    float vertices[] = {-0.5, -0.5,     0.0, 0.5,     0.5, -0.5};
}
```

Pour le reste de la méthode, il suffit simplement d'ajouter la boucle **while** que nous avons déjà codé. 😊

Encore une fois, je me répète mais n'oubliez pas d'ajouter le préfixe "**m_**" aux attributs **evenements** et **fenetre** quand vous copiez le code :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    bool terminer(false);
    float vertices[] = {-0.5, -0.5,     0.0, 0.5,     0.5, -0.5};

    // Boucle principale

    while (!terminer)
    {
        // Gestion des évènements

        SDL_WaitEvent (&m_evenements);
```

```
    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = 1;

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT);

    // On remplit puis on active le tableau Vertex Attrib 0
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
    vertices);
    glEnableVertexAttribArray(0);

    // On affiche le triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // On désactive le tableau Vertex Attrib puisque l'on n'en
    a plus besoin
    glDisableVertexAttribArray(0);

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow(m_fenetre);
}
```

Le fichier `main.cpp`

Comme d'habitude en C++, la fonction `main()` sera la fonction la moins chargée du programme. Jusqu'à maintenant, nous codions tout à l'intérieur de cette fonction, ce qui la rendait un peu illisible.

Maintenant, nous allons juste déclarer un objet, l'initialiser et lancer sa boucle principale. 😊 Pour ça, il faut inclure le *header* de la classe `SceneOpenGL` dans le fichier `main.cpp` :

Code : C++

```
#include "SceneOpenGL.h"
```

Ensuite, on va déclarer un objet de type `SceneOpenGL` avec les bons paramètres :

Code : C++

```
int main(int argc, char **argv)
{
    // Création de la scène
    SceneOpenGL scene("Chapitre 3", 800, 600);
}
```

Maintenant, il faut appeler les deux méthodes qui permettent d'initialiser le programme correctement à savoir `initialiserFenetre()` et `initGL()`. De plus, il faut vérifier que ces méthodes retournent bien le booléen `true` et pas `false`. Si au moins une des deux initialisations échoue, alors on quitte le programme :

Code : C++

```
// Initialisation de la scène

if(scene.initialiserFenetre() == false)
    return -1;

if(scene.initGL() == false)
    return -1;
```

Enfin, on appelle la méthode `bouclePrincipale()` pour lancer la scène OpenGL :

Code : C++

```
// Boucle Principale

scene.bouclePrincipale();
```

N'oublions pas le `return 0` lorsque l'on quittera le programme :

Code : C++

```
// Fin du programme

return 0;
```

Si on résume tout ça :

Code : C++

```
#include "SceneOpenGL.h"

int main(int argc, char **argv)
{
    // Création de la scène

    SceneOpenGL scene("Chapitre 3", 800, 600);

    // Initialisation de la scène

    if(scene.initialiserFenetre() == false)
        return -1;

    if(scene.initGL() == false)
        return -1;

    // Boucle Principale

    scene.bouclePrincipale();
```

```
// Fin du programme  
return 0;  
}
```

Et voilà ! Il ne vous reste plus qu'à compiler tout ça. Vous devriez avoir le même résultat qu'au dessus mais vous avez codée proprement une classe en C++ avec du code OpenGL à l'intérieur. 😊

Nous ferons plein de classes au fur et à mesure de ce tuto, vous allez vite en prendre l'habitude. 😊

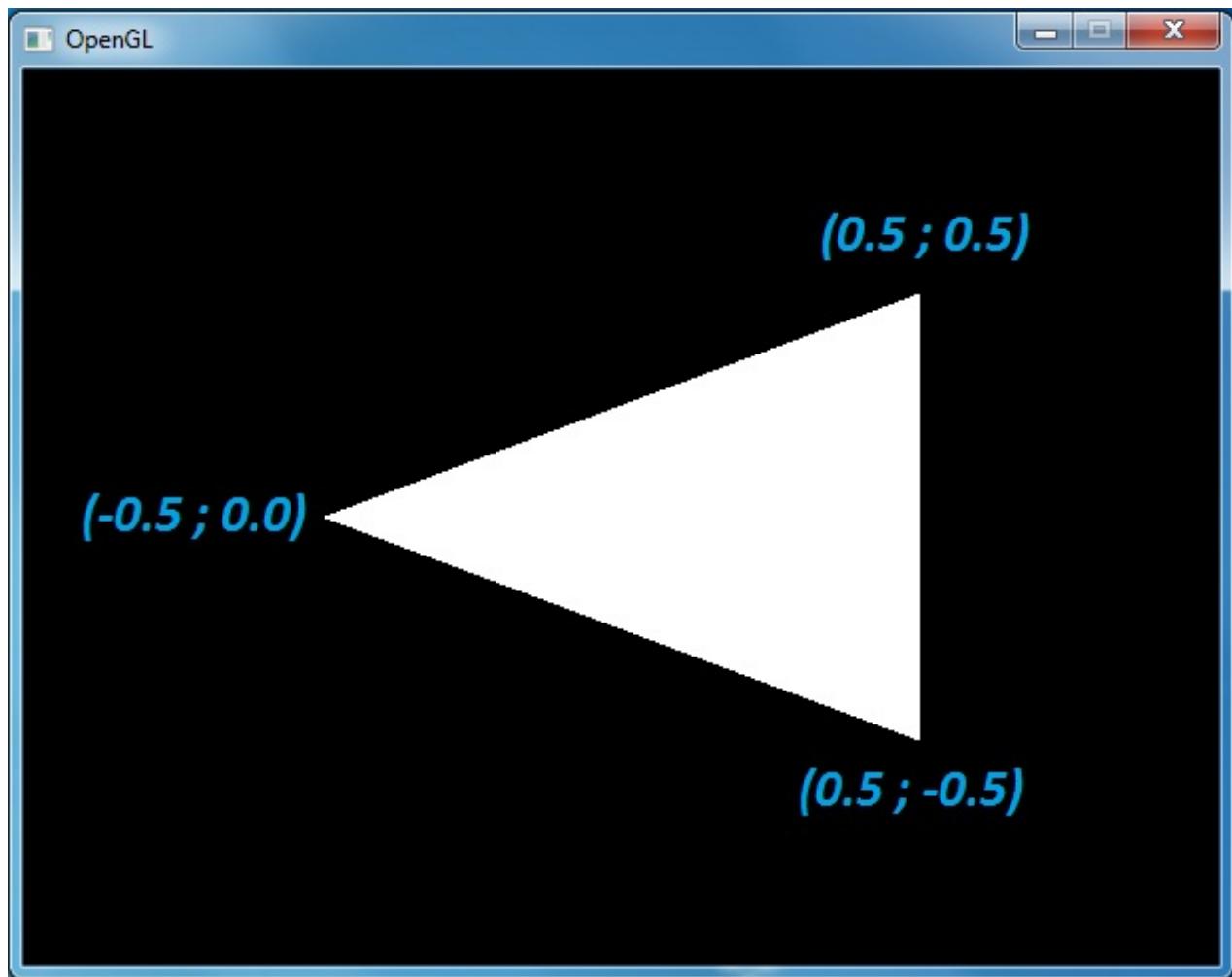
Télécharger : [Code Source C++ du Chapitre 3](#)

Exercices

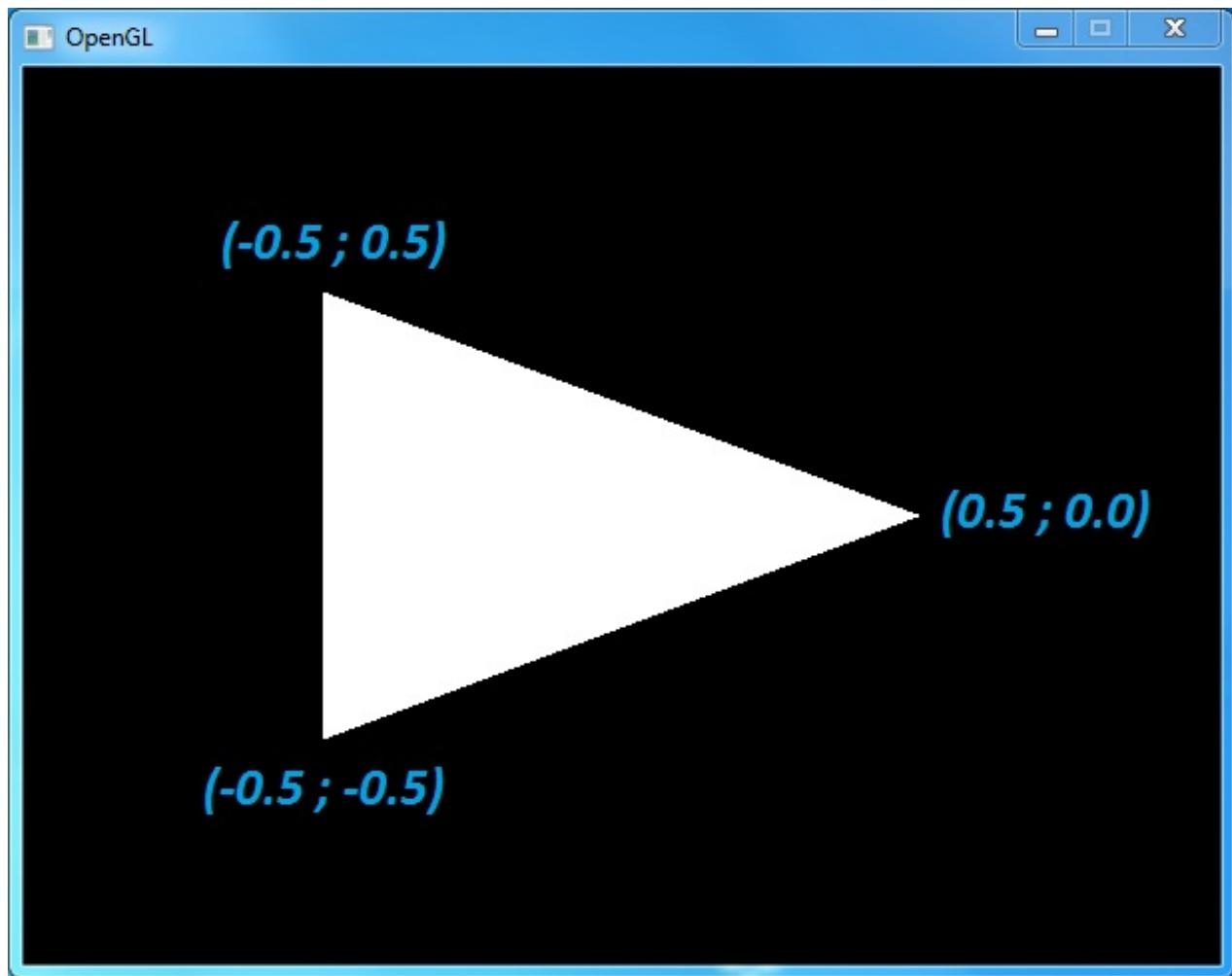
Énoncés

Tout au long de ce tutoriel, je vous ferai faire quelques petits exercices pour que vous appliquez ce que nous aurons vu dans les chapitres. Ce seront des exercices assez simples, il n'y aura rien de farfelu je vous rassure. Évidemment, vous avez le droit de vous aider du cours, je ne vous demande pas de tout retenir d'un coup à chaque fois. D'ailleurs, les solutions sont fournies juste après les énoncés, ne les regardez pas avant sinon ça n'a aucun intérêt. 😊

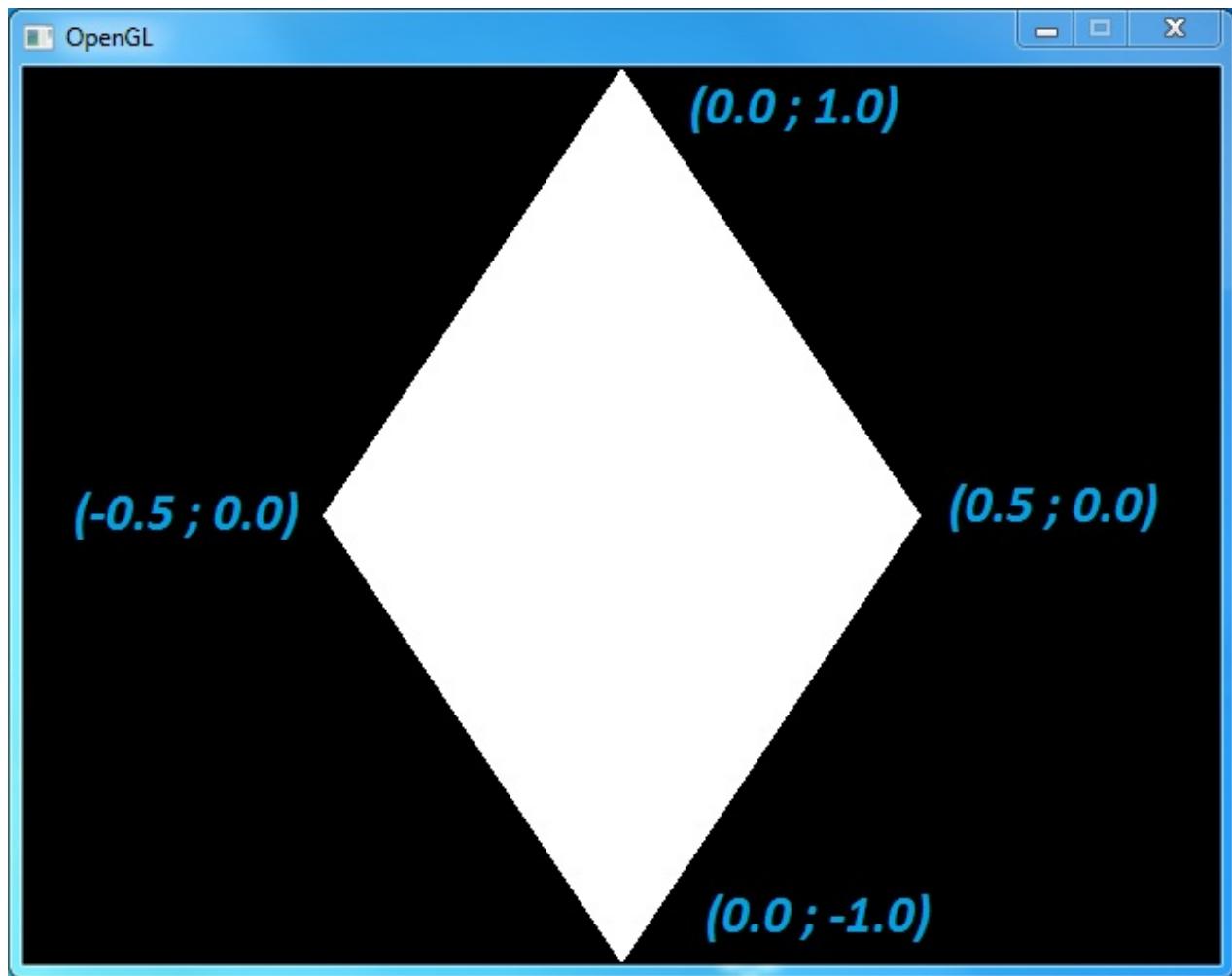
Exercice 1 : Avec les notions vues dans ce chapitre, affichez un triangle ayant les coordonnées présentes ci-dessous. Vous n'avez besoin de modifier que les vertices par rapport aux exemples du cours, inutile de toucher au reste.



Exercice 2 : Reprenez le même triangle que l'exercice précédent mais modifiez ses vertices pour l'inverser :



Exercice 3 : On passe un cran au-dessus maintenant. Je vous demande d'afficher la forme suivante en utilisant deux triangles distincts (donc pas avec le paramètre `GL_TRIANGLE_STRIP`) :



Solutions

Exercice 1 :

Secret (cliquez pour afficher)

La seule chose à modifier ici c'est le tableau de vertices. L'énoncé demandait un triangle avec des coordonnées spécifiques, le tableau de valeurs ressemble donc à ceci :

Code : C++

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

Vous remarquerez que j'ai délimité les coordonnées de façon à bien différencier les vertices. 😊

Je devrais en théorie m'arrêter là pour la correction, mais pour ceux qui le souhaitent je donne en détail le code source de l'affichage des vertices. Celui-ci ne change absolument pas par rapport aux exemples du cours, mais si ça peut vous aider à comprendre un peu mieux je vais le ré-expliquer rapidement.

Premièrement, on reprend le code de la boucle principale avec sa gestion d'évènements et son actualisation de fenêtre :

Code : C++

```
// Boucle principale
```

```
while (!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent (&m_evenements);

    if (m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow (m_fenetre);
}
```

Le code d'affichage consiste simplement à appeler les fonctions :

- **glClear()** pour nettoyer ce qui était présent avant
- **glVertexAttribPointer()** pour donner les vertices à OpenGL
- **glDrawArrays()** pour les afficher

Code : C++

```
// Nettoyage de l'écran
glClear(GL_COLOR_BUFFER_BIT);

// Tableau Vertex Attrib 0 pour envoyer les vertices
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);

// Affichage du triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Il ne faut bien sûr pas oublier d'activer le tableau **Vertex Attrib** au moment d'envoyer les vertices, puis de le désactiver quand on n'en a plus besoin :

Code : C++

```
// Nettoyage de l'écran
glClear(GL_COLOR_BUFFER_BIT);

// Tableau Vertex Attrib 0 pour envoyer les vertices
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);

// Activation du tableau Vertex Attrib
glEnableVertexAttribArray(0);

// Affichage du triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
```

```
// Désactivation du tableau Vertex Attrib  
glDisableVertexAttribArray(0);
```

Ce qui donne le code source suivant pour la boucle principale :

Code : C++

```
// Vertices  
float vertices[] = {-0.5, 0.0, 0.5, -0.5, 0.5, 0.5};  
  
// Boucle principale  
  
while (!terminer)  
{  
    // Gestion des évènements  
  
    SDL_WaitEvent(&m_evenements);  
  
    if (m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)  
        terminer = true;  
  
    // Nettoyage de l'écran  
  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    // Tableau Vertex Attrib 0 pour envoyer les vertices  
  
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);  
  
    // Activation du tableau Vertex Attrib  
  
    glEnableVertexAttribArray(0);  
  
    // Affichage du triangle  
  
    glDrawArrays(GL_TRIANGLES, 0, 3);  
  
    // Désactivation du tableau Vertex Attrib  
  
    glDisableVertexAttribArray(0);  
  
    // Actualisation de la fenêtre  
  
    SDL_GL_SwapWindow(m_fenetre);  
}
```

Exercice terminé. 😊

Exercice 2 :

Secret (cliquez pour afficher)

Pour cet exercice, il suffit juste de modifier notre tableau de vertices. On prend donc notre schéma pour en tirer les coordonnées suivantes :

Code : C++

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

Exercice 3 :**Secret (cliquez pour afficher)**

Cet exercice est un poil plus compliqué que les deux autres mais il n'y a rien de dur si on regarde dans le fond. 😊

La forme demandée est évidemment constituée de deux triangles, il faut donc commencer par déclarer deux tableaux de vertices qui contiendront les coordonnées de ces deux triangles :

Code : C++

```
// Vertices  
  
float vertices[] = {-0.5, 0.0, 0.0, 1.0, 0.5, 0.0, //  
Triangle 1  
-0.5, 0.0, 0.0, -1.0, 0.5, 0.0}; //  
Triangle 2
```

Il faut évidemment modifier l'appel à la fonction `glDrawArrays()` pour qu'elle prenne en compte les **6** vertices et non uniquement **3** :

Code : C++

```
// Affichage du triangle  
  
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Le reste du code ne change pas.

Ah, nous avons enfin pu afficher quelque chose (même si ce n'est un simple triangle), nous avons enfin pu faire bosser notre carte graphique qui commençait légèrement à s'endormir. 😴

Rappelez-vous bien de ce que nous avons vu ici, nous réutiliserons tout ça dans les futurs chapitres.

Introduction aux shaders

Ah c'est un vaste sujet que nous allons aborder aujourd'hui 😊. Tellement vaste que nous allons le découper en 4 chapitres dont le premier sera celui-ci. Nous verrons les trois autres beaucoup plus tard, vous comprendrez pourquoi. Bref, il est temps de découvrir ce que sont ces mystérieux Shaders. 🎉😊

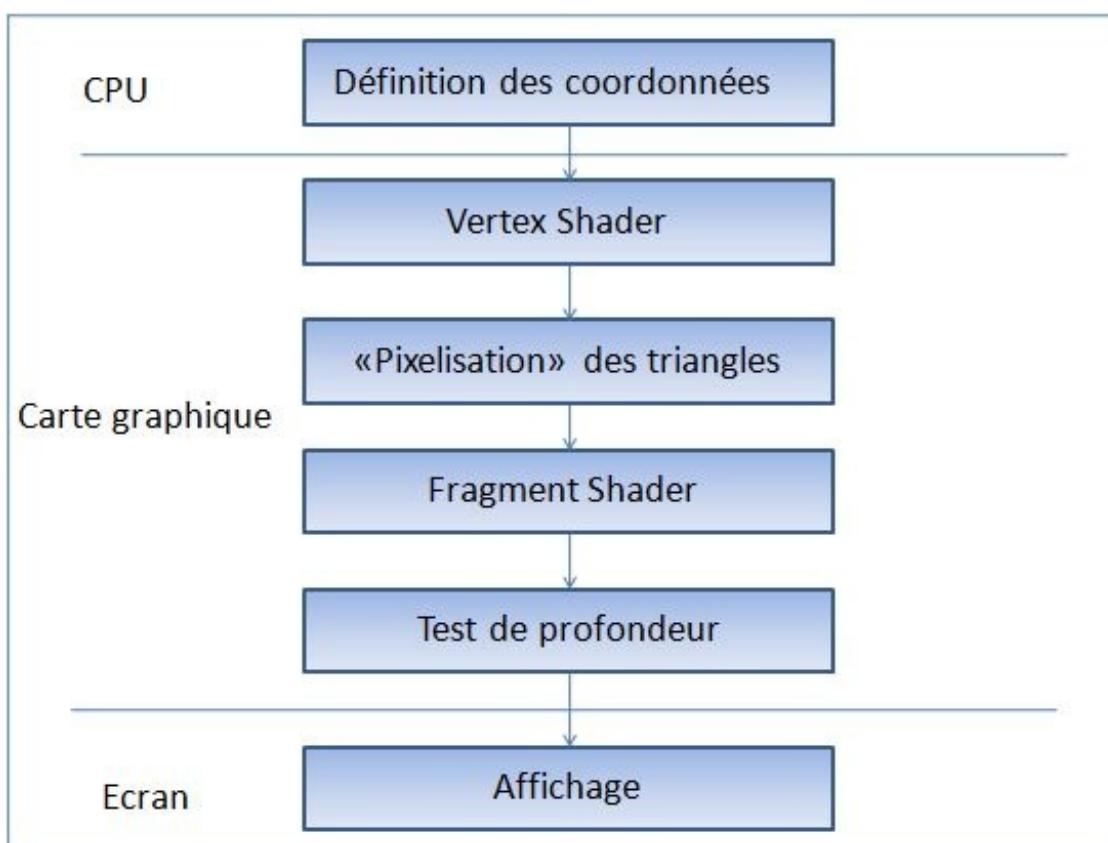
Qu'est-ce qu'un Shader ?

Introduction

Commençons par un peu de théorie. 😐

Comme vous le savez, OpenGL est une librairie qui tire sa puissance de la carte graphique. Or lorsque nous programmons, nous n'avons pas accès à cette carte mais uniquement au processeur et à la RAM. Comment fait notre programme pour l'exploiter me direz-vous ?

En réalité, OpenGL va bidouiller pas mal de choses dans la carte lorsque nous appelons certaines fonctions comme `glDrawArrays()`. Voyons d'ailleurs ce qui se passe entre le moment où l'on appelle cette fonction et le moment où la scène s'affiche à l'écran :



- **Définition des coordonnées** : Dans un premier temps, nous définissons les coordonnées de vertices. Ça on sait le faire grâce à la fonction `glVertexAttribPointer()`.
- **Vertex Shader** : Nous verrons cette étape dans quelques minutes. 😊
- **Pixelisation des triangles** : C'est le moment où une forme géométrique est convertie en pixels.
- **Fragment Shader** : Nous verrons également cela un peu plus loin.
- **Test de profondeur** : Test permettant de savoir s'il faut afficher tel ou tel pixel. Cette notion sera développée un peu plus tard.
- **Affichage** : C'est la sortie de la carte graphique, la plupart du temps ce sera notre écran.

Cette suite d'opérations constitue ce que l'on appelle : **le pipeline 3D**. Les parties qui nous intéressent dans ce chapitre sont : le **Vertex Shader** et le **Fragment Shader**.



Euh qu'est-ce que c'est que ça ? 😐

Un shader est simplement un programme exécuté non pas par le processeur mais par la carte graphique. Il en existe deux types (enfin 3 mais seuls 2 sont réellement importants) :

- **Vertex Shader** : C'est l'étape qui va nous permettre soit de valider les coordonnées de nos sommets, soit de les modifier. Cette étape prend un vertex à part pour travailler dessus. S'il y a 3 vertices (pour un triangle) alors le vertex shader sera exécuté 3 fois.
- **Fragment Shader** (parfois appelé Pixel Shader) : C'est l'étape qui va définir la couleur de chaque pixel de la forme délimitée par les vertices. Par exemple, si vous avez défini un rectangle de 100 pixels par 50 pixels, alors le fragment shader se chargera de définir la couleur des 5000 pixels composant le rectangle.



Ne vous inquiétez pas, 5000 pixels c'est une promenade de santé pour la carte graphique 🍩. Elle est conçue pour faire ce genre d'opérations à répétition.

A quoi servent les shaders ?

Étant donné la complexité de cette notion, nous ne devrions étudier les shaders que beaucoup plus tard dans le tutoriel. Dans les anciennes versions d'OpenGL, ils étaient optionnels, c'était l'API elle-même qui se chargeait d'effectuer ces opérations. Mais avec la version 3.0, le groupe Khronos a voulu introduire une nouvelle philosophie : **le tout shader**. En clair, ils nous imposent leur utilisation.



Euh ... Pourquoi nous imposer un truc aussi compliqué alors que c'était géré automatiquement avant ?

Ce qu'il faut comprendre, c'est que les jeux-vidéo ont beaucoup évolué depuis qu'OpenGL existe. Il y a quelques années, il était plus simple de laisser l'API faire tout le boulot, les jeux ne prenaient pas énormément de ressources et surtout il y avait moins de choses à calculer. Mais de nos jours, les graphismes, le réalisme et la vitesse sont devenus les principales préoccupations des développeurs (et des joueurs 😊). L'API n'est plus capable de gérer tout ça rapidement car elle le fait avec ses vieilles méthodes (trop lentes pour les jeux d'aujourd'hui).

L'avantage d'une gestion personnalisée est que l'on peut faire ce que l'on veut et surtout on peut y mettre uniquement ce dont on a besoin, ce qui peut nous faire gagner parfois pas mal de vitesse.



En théorie lors du chapitre précédent, nous n'aurions pas dû afficher un triangle sans utiliser les shaders. Cependant, OpenGL nous autorise à afficher des petites formes blanches avec le repère de base sans avoir à les utiliser. Avouez tout de même qu'un jeu comme ça ce n'est pas passionnant. A moins de faire un Pong en 2D mais je pense honnêtement que ce n'est pas votre but. 😊

Exemples de Shaders

Ah c'est certainement la partie qui va le plus vous intéresser. 🍩

Les shaders ne sont pas uniquement des étapes embêtantes qui sont là pour vous compliquer la vie 😞. Leur première utilité vient du fait qu'ils permettent d'afficher nos objets dans un monde 2D ou 3D. En gros, voilà ce qui se passe lorsque nous voulons afficher un objet :

- On définit ses coordonnées dans l'espace.
- L'objet est ensuite passé au Vertex Shader (n'oubliez pas que c'est un programme exécuté par la carte graphique).

- Puis au Fragment Shader (c'est un programme aussi).
- Il est maintenant prêt à être affiché.

Ça c'est la première utilité, passons maintenant à la deuxième qui est certainement la plus importante (et la plus intéressante 😊) : ils permettent de faire tous les effets 3D magnifiques que vous voyez dans un jeu-vidéo tels que : les lumières, l'eau, les ombres, les explosions ... Ce sont les shaders qui font d'un jeu un jeu plus réaliste.



Alors ne vous emballez pas, ce n'est pas maintenant que nous allons apprendre à faire tous ces effets mais nous y viendrons.



Le principal problème avec les shaders est que lorsque l'on débute dans la programmation 3D, il est très difficile d'apprendre à les utiliser du fait de leur complexité. C'est pour cela que dans un premier temps, je vais vous fournir le code source pour la création des shaders. Nous consacrerons trois chapitres entiers pour apprendre à les gérer une fois que vous serez plus habitués avec OpenGL.

En bref ce qu'il faut retenir c'est que :

- Un **shader** est un programme exécuté par la carte graphique.
- Il en existe de deux types : les **vertex** et les **fragment**.
- Chaque chose que nous voulons afficher passera d'abord entre les mains de ces deux shaders.

Utilisation des shaders

Passons maintenant à la partie programmation. 

Pour commencer, je vais vous demander de télécharger l'archive ci-dessous (pour Linux et Windows), elle contient un fichier « **Shader.h** », « **Shader.cpp** » et un dossier « **Shaders** » contenant plein de petits fichiers. Vous placerez tout ça dans le répertoire de chaque projet que vous ferez (donc dans chaque chapitre que nous ferons).

Télécharger : [Code Source C++ Shaders - Windows & Linux](#)

Une fois les fichiers ajoutés à votre dossier, il vous suffit simplement d'ajouter le header « **Shader.h** » et le fichier source « **Shader.cpp** » à votre projet.



N'oubliez pas d'inclure le header quand vous en avez besoin.

Comme vous le savez maintenant, un shader est un programme, différent d'un programme normal certes mais un programme quand même. Il doit donc respecter plusieurs règles :

- Un code source
- Une compilation

Nous ne verrons pas ces deux étapes maintenant mais sachez au moins que ce n'est pas si différent d'un programme normal. Les codes sources sont dans le dossier « **Shaders** » que vous devriez avoir placé dans le répertoire de votre projet.

Avant toute chose, pour pouvoir utiliser les shaders il va falloir utiliser la classe **Shader** dont voici le constructeur :

Code : C++

```
Shader(std::string vertexSource, std::string fragmentSource);
```

- **vertexSource** : C'est le chemin du code source de notre Vertex Shader.
- **fragmentSource** : C'est le chemin du code source de notre Fragment Shader.

Alors attention, appeler le *constructeur* ne suffit pas. Si vous nappelez que lui, votre shader ne sera pas exploitable. Pour le rendre exploitable, il faut utiliser la méthode **charger()** qui permet en gros de lire les fichiers sources, de les compiler, ...

Code : C++

```
bool charger();
```

Cette méthode retourne un *booléen* pour savoir si la création du shader s'est bien passée.



On ne va pas vérifier la valeur renournée ici mais vous êtes libre de le faire si vous le voulez. Si un shader ne réussit pas à se charger, le programme ne va pas s'arrêter pour autant. OpenGL continuera de s'exécuter comme si le shader n'existe pas.

En bref, voici un petit exemple de création de shader :

Code : C++

```
// Création du shader

Shader shaderBasique("Shaders/basique_2D.vert",
"Shaders/basique.frag");
shaderBasique.charger();

// Début de la boucle principale

while(!terminer)
{
    // Utilisation
}
```

Bien, passons à l'utilisation qui est ma foi assez simple puisque nous n'utilisons qu'une seule fonction :

Code : C++

```
glUseProgram(GLuint program) ;
```

 Attention, cette fonction commence par le préfixe "gl", ce n'est donc pas une méthode de la classe **Shader**.

Elle prend un paramètre : l'ID d'un certain "**program**", nous lui donnerons l'attribut : "**m_programID**" de la classe **Shader** grâce à la méthode :

Code : C++

```
GLuint getProgramID() const;
```

Ne vous posez pas de question sur ça pour le moment 😊

Cette fonction a deux utilités :

- Lorsqu'on lui donne l'attribut "**m_programID**", OpenGL va comprendre "Je prends le shader que tu me donnes pour l'utiliser dans mon pipeline".
- Une fois qu'on a affiché ce qu'on voulait afficher, on va faire comprendre à OpenGL de ne plus utiliser le shader. Dans ce cas, le paramètre ne sera pas "**m_programID**" mais nous lui donnerons la **valeur 0**.

Cette fonction se place juste avant **glDrawArrays()** lorsque nous voulons activer notre shader, puis après avec le paramètre 0 pour le désactiver :

Code : C++

```
// Activation du shader

glUseProgram(shaderBasique.getProgramID());

// Affichage du triangle

glDrawArrays(GL_TRIANGLES, 0, 3);

// Désactivation du shader
```

```
glUseProgram(0);
```



Je vous conseille d'indenter le code qui se trouve entre les appels de la fonction `glUseProgram()`. Vous verrez que nous mettrons de plus en plus de chose à l'intérieur, l'indentation rendra le tout plus lisible.

Simple non ? 🤔 D'ailleurs on peut même intégrer, entre ces deux appels de la fonction `glUseProgram()`, le code relatif à l'envoi des vertices au tableau **Vertex Attrib**. En général, on fait cela pour bien différencier le code d'affichage du reste du programme, ce qui donnerait pour nous :

Code : C++

```
// Activation du shader
glUseProgram(shaderBasique.getProgramID());

// On remplit puis on active le tableau Vertex Attrib 0
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, vertices);
 glEnableVertexAttribArray(0);

// Affichage du triangle
 glDrawArrays(GL_TRIANGLES, 0, 3);

// On désactive le tableau Vertex Attrib puisque l'on n'en a plus besoin
 glDisableVertexAttribArray(0);

// Désactivation du shader
glUseProgram(0);
```

Dans le chapitre précédent, nous nous sommes permis de ne pas utiliser ces fameux shaders. Mais comme je vous l'ai dit tout à l'heure, on ne peut plus continuer ainsi. Reprenons notre ancien code pour y ajouter les fonctions que nous venons de voir :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    bool terminer(false);
    float vertices[] = {-0.5, -0.5, 0.0, 0.5, 0.5, -0.5};

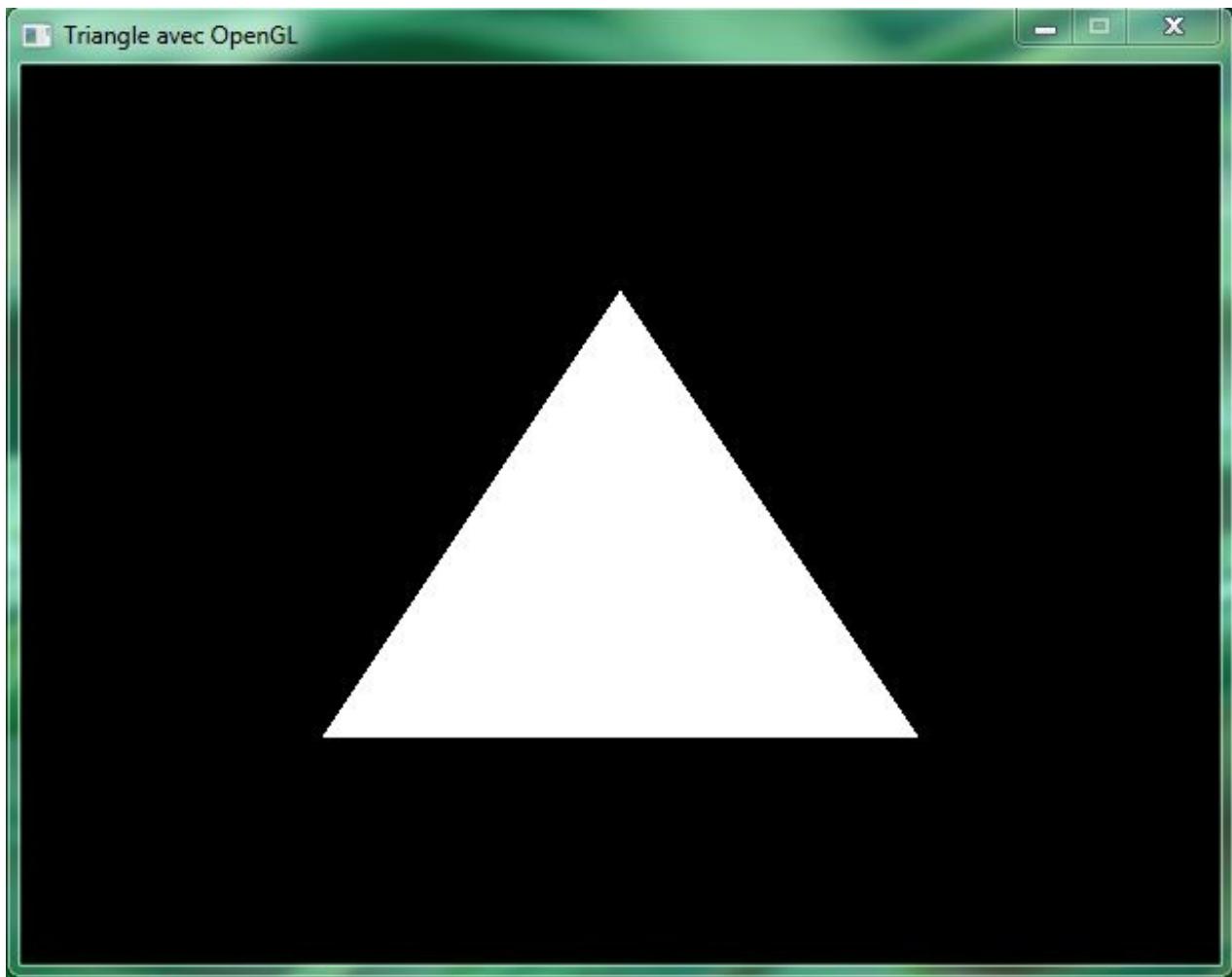
    Shader shaderBasique("Shaders/basique_2D.vert",
    "Shaders/basique.frag");
    shaderBasique.charger();

    // Boucle principale

    while (!terminer)
    {
        // Gestion des évènements
        SNT_WaitEvent(&event, etatEvenement);
        if (event.type == SDL_QUIT)
            terminer = true;
        else if (event.type == SDL_MOUSEMOTION)
            updatePosition(event.motion.x, event.motion.y);
        else if (event.type == SDL_MOUSEBUTTONDOWN)
            updatePosition(event.button.x, event.button.y);
        else if (event.type == SDL_MOUSEBUTTONUP)
            updatePosition(event.button.x, event.button.y);
        else if (event.type == SDL_KEYDOWN)
            updateKey(event.key.keysym.scancode);
        else if (event.type == SDL_KEYUP)
            updateKey(event.key.keysym.scancode);
    }
}
```

```
SDL_WAITEVENT(m_evenements);  
  
if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)  
    terminer = true;  
  
// Nettoyage de l'écran  
glClear(GL_COLOR_BUFFER_BIT);  
  
// Activation du shader  
glUseProgram(shaderBasique.getProgramID());  
  
// On remplit puis on active le tableau Vertex Attrib 0  
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,  
vertices);  
glEnableVertexAttribArray(0);  
  
// Affichage du triangle  
glDrawArrays(GL_TRIANGLES, 0, 3);  
  
// On désactive le tableau Vertex Attrib puisque l'on  
n'en a plus besoin  
glDisableVertexAttribArray(0);  
  
// Désactivation du shader  
glUseProgram(0);  
  
// Actualisation de la fenêtre  
SDL_GL_SwapWindow(m_fenetre);  
}  
}
```

Vous devriez vous retrouver avec une fenêtre comme celle-là :



Quoi mais c'est nul ! C'est la même chose que dans le chapitre précédent mais en plus compliqué.

Oui c'est la même chose mais c'est comme cela que les choses doivent être faites ;). Pour le moment vous ne voyez pas trop l'intérêt d'utiliser les shaders mais vous allez vite voir que c'est indispensable.

D'ailleurs pourquoi attendons-nous ? Voyons dès maintenant ce qu'ils ont à nous offrir. 

Un peu de couleurs

Un peu de couleurs

Il est maintenant temps de faire passer notre triangle à la couleur, je le trouve un peu trop pâlot. 



Comment fonctionnent les couleurs avec OpenGL ?

La réponse est simple, vous savez qu'avec la SDL, pour créer un rectangle coloré il fallait utiliser cette fonction :

Code : C

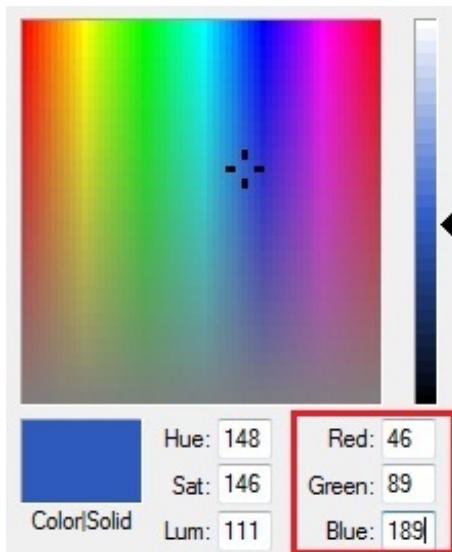
```
SDL_MapRGB(SDL_Surface *surface, Uint8 red, Uint8 green, Uint8 blue);
```

Les trois paramètres étaient les composantes RGB (Rouge, Vert, Bleu). Il suffisait de combiner ces trois couleurs pour en former

une seule au final. Bonne nouvelle, avec OpenGL c'est pareil. Pour fabriquer une couleur vous devez donner :

- Une quantité de **rouge**
- Une quantité de **vert**
- Une quantité de **bleu**

Si vous voulez faire des tests pour obtenir différentes couleurs, essayez la palette de couleurs Windows (ou équivalent sur Linux) :



Il y a deux façons de représenter les couleurs :

- Soit avec des valeurs comprises entre 0 et 255 (la plus compréhensible)
- Soit avec des valeurs comprises entre 0 et 1 (un peu difficile d'imaginer une couleur entre 0 et 1)

Malheureusement pour nous, nous allons devoir utiliser la seconde méthode. Mais pas de panique, nous allons utiliser une petite combinaison pour utiliser la première.

Juger une couleur entre 0 et 255 est plus facile à comprendre, pour pouvoir utiliser cette méthode nous allons **diviser** la valeur de la couleur (par exemple : 128) **par 255**. De cette façon on se retrouve avec une valeur comprise entre 0 et 1 :

Code : C

```
float rouge = 128.0/255.0; // = 0.50
float vert = 204.0/255.0; // = 0.80
float bleu = 36.0/255.0; // = 0.141
```

La quantité de rouge sera de 128, le vert de 204 et le bleu de 36. De plus faites attention, nous travaillons avec des float donc n'oubliez pas de préciser les décimales même s'il n'y a en pas.



Lorsque vous voulez mettre une quantité au maximum (255), il est inutile de faire 255.0/255.0, mettez simplement 1.0. Même chose pour une quantité nulle (0.0)

Au niveau du shader, on va changer le shader **basique** par le shader **couleur2D** car ce premier ne faisait qu'afficher ce que nous lui donnions en blanc. Maintenant que nous voulons de la couleur, il faut charger un autre shader gérer la couleur :

Code : C++

```
// Shader
```

```
Shader shaderCouleur("Shaders/couleur2D.vert",
"Shaders/couleur2D.frag");
shaderCouleur.charger();
```

Vu que nous avons changé le nom du shader (**shaderCouleur**), il faut effectuer le même changement de nom lorsque l'on récupère le **programID** :

Code : C++

```
// Activation du shader
glUseProgram(shaderCouleur.getProgramID());

// Envoi des données et affichage
. . .

// Désactivation du shader
glUseProgram(0);
```

Au niveau du code d'affichage, vous connaissez déjà presque tout. Les fonctions utilisées sont les mêmes que celles des coordonnées de vertex. Les couleurs se gèrent également de la même façon : on place la valeur de chaque couleur (RGB) les unes à la suite des autres dans un tableau.

Le seul changement sera le numéro du tableau, au lieu de placer nos couleurs dans le tableau **0** nous les placerons dans le tableau **1**. Le premier tableau (indice **0**) servira à stocker tous nos vertices et le deuxième tableau (indice **1**) servira à stocker nos couleurs.

Code : C++

```
// On définit les couleurs
float couleurs[] = {0.0, 204.0 / 255.0, 1.0, 0.0, 204.0 / 255.0,
1.0, 0.0, 204.0 / 255.0, 1.0};

while(...)

. . .

// Activation du shader
glUseProgram(shaderCouleur.getProgramID());

// Envoi des vertices
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
vertices);
 glEnableVertexAttribArray(0);

// On rentre les couleurs dans le tableau Vertex Attrib 1
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, couleurs);
 glEnableVertexAttribArray(1);
```

```
    ....  
  
    // Désactivation du shader  
    glUseProgram(0);  
  
    ....  
}
```



Faites attention au paramètre **size** du tableau `VertexAttrib`, ici nous lui donnons **3** car une couleur est composée de 3 couleurs.

Voyons ce que donne toutes ces petites modifications sur un exemple concret :

Code : C++

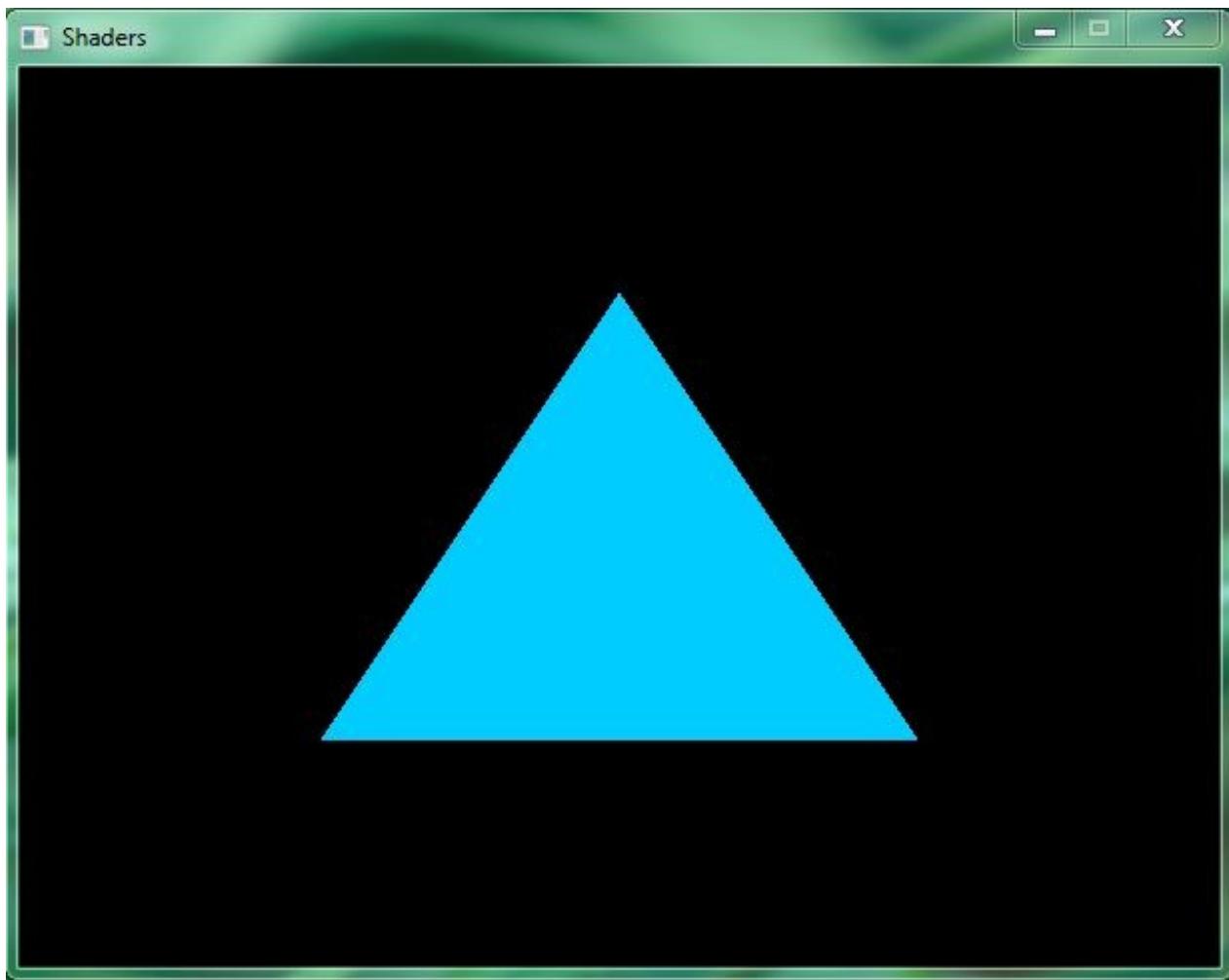
```
void SceneOpenGL::bouclePrincipale()  
{  
    // Variables  
  
    bool terminer(false);  
    float vertices[] = {-0.5, -0.5, 0.0, 0.5, 0.5, -0.5};  
    float couleurs[] = {0.0, 204.0 / 255.0, 1.0, 0.0, 204.0 /  
    255.0, 1.0, 0.0, 204.0 / 255.0, 1.0};  
  
    // Shader  
  
    Shader shaderCouleur("Shaders/couleur2D.vert",  
    "Shaders/couleur2D.frag");  
    shaderCouleur.charger();  
  
    // Boucle principale  
  
    while (!terminer)  
    {  
        // Gestion des événements  
  
        SDL_WaitEvent(&m_evenements);  
  
        if (m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)  
            terminer = true;  
  
        // Nettoyage de l'écran  
  
        glClear(GL_COLOR_BUFFER_BIT);  
  
        // Activation du shader  
  
        glUseProgram(shaderCouleur.getProgramID());  
  
        // Envoi des vertices  
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,  
        vertices);  
        glEnableVertexAttribArray(0);
```

```
// Envoi des couleurs  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,  
couleurs);  
glEnableVertexAttribArray(1);  
  
// Affichage du triangle  
glDrawArrays(GL_TRIANGLES, 0, 3);  
  
// Désactivation des tableaux Vertex Attrib  
glDisableVertexAttribArray(1);  
glDisableVertexAttribArray(0);  
  
// Désactivation du shader  
glUseProgram(0);  
  
// Actualisation de la fenêtre  
SDL_GL_SwapWindow(m_fenetre);  
}  
}
```



Petit détail : Vous désaktiverez vos tableaux dans l'ordre inverse où vous les avez activés. Par exemple : si j'active les tableaux dans l'ordre 1, 2, 3 alors je les désactiverai dans l'ordre 3, 2, 1.

Vous devriez obtenir un triangle coloré :



Vous commencez à voir l'intérêt des shaders ? 😊 Bon si vous n'êtes pas convaincus je vais vous montrer une autre façon de colorier notre triangle.

Si chaque sommet possède sa propre couleur alors OpenGL (avec l'aide des shaders) nous fera un joli petit dégradé entre les différentes couleurs. Prenons un exemple, nous allons définir une couleur différente pour chaque sommet :

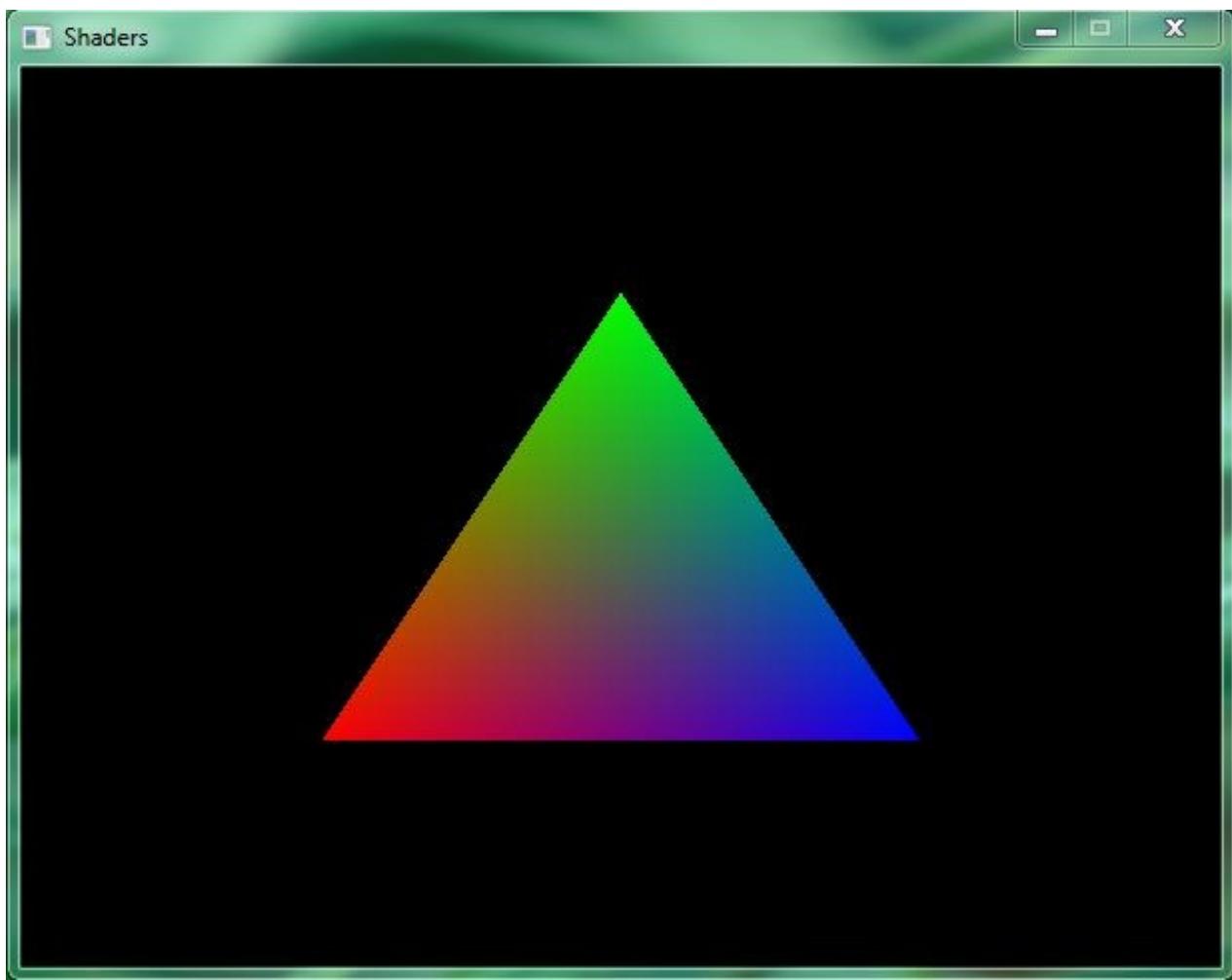
Code : C++

```
// Remplaçons les couleurs par les suivantes ...

float couleurs[] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0};

//... et voyons ce que ça donne;
```

Vous devriez avoir ceci :



Comme vous le constatez, les shaders ont calculé automatiquement la couleur de chaque pixel se trouvant entre les différents sommets.

Cette opération s'appelle **l'interpolation**. C'est-à-dire qu'OpenGL est capable de trouver la couleur de chaque pixel entre deux sommets ayant une couleur différente. Le gros avantage de l'interpolation c'est que nous ne nous en occupons pas 😊. En effet, même si nous définissons nos shaders, nous ne nous occuperons pas de trouver la couleur de chaque pixel. Il nous suffira juste de donner notre code source pour seulement un et OpenGL se chargera de faire la même opération pour tous les autres.

Magique n'est-ce pas ? 🎩

Télécharger : [Code Source C++ du Chapitre 4](#)

Exercices

Énoncés

Au même titre que le chapitre précédent, vous avez maintenant le droit à votre petite série d'exercice. Votre objectif va être de colorier différents triangles avec une couleur spécifique.

Exercice 1 : Coloriez le triangle du haut avec les valeurs données ci-dessous :

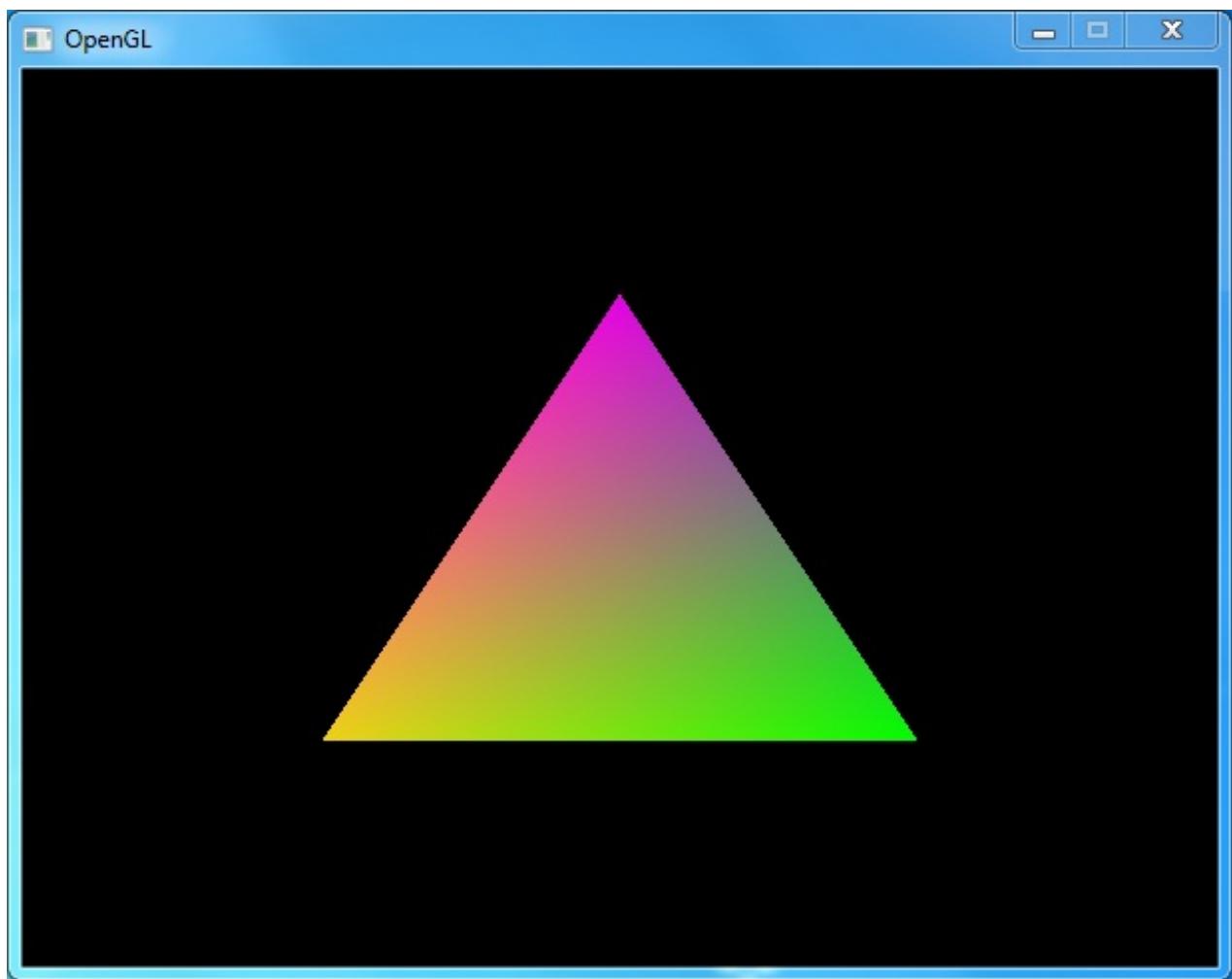
- **rouge** : 240.0
- **vert** : 210.0
- **bleu** : 23.0

Exercice 2 : Même consigne que précédemment mais avec les valeurs :

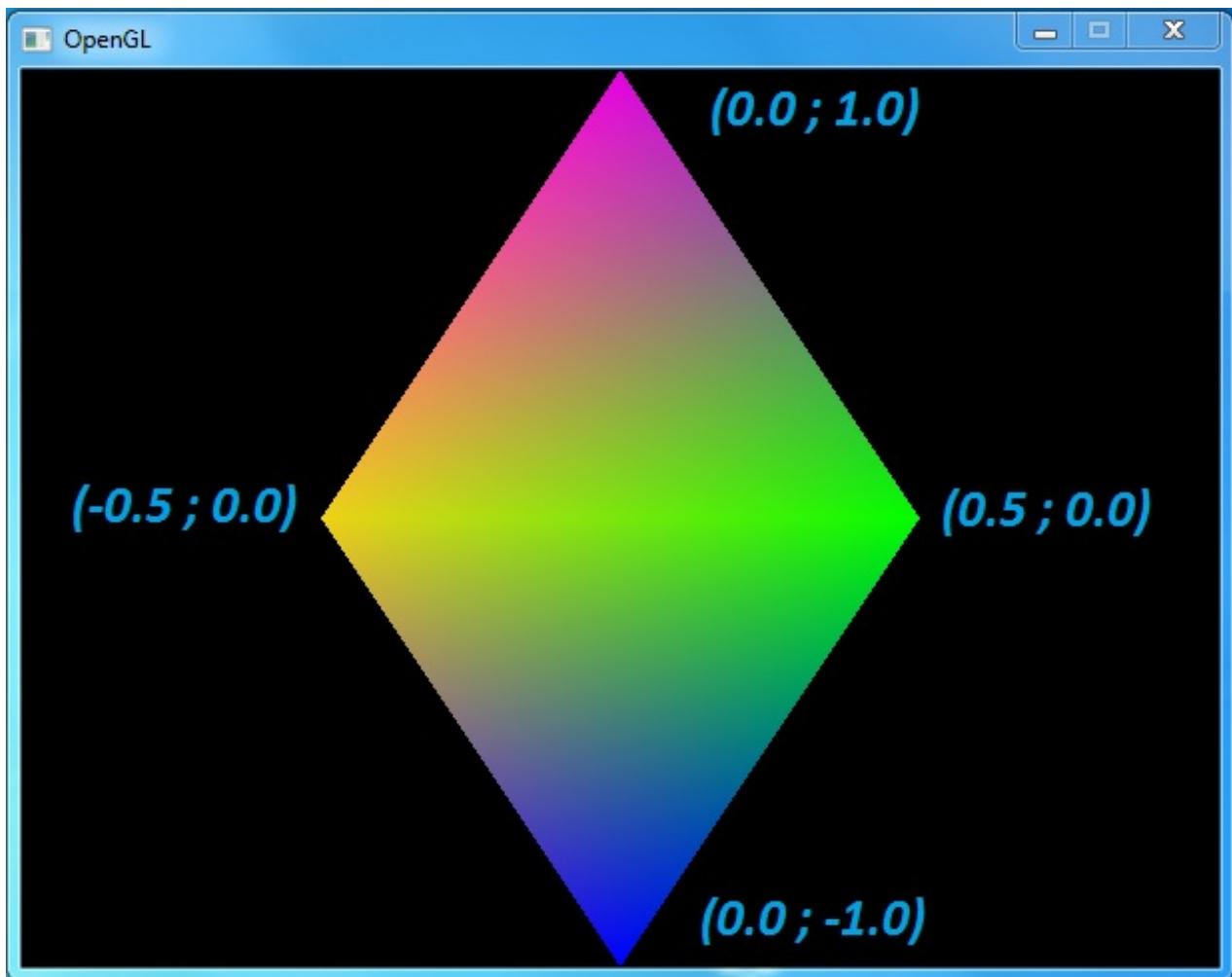
- **rouge** : 230.0
- **vert** : 0.0
- **bleu** : 230.0

Exercice 3 : Coloriez le triangle uniquement avec la couleur vert (valeur **255.0**). Simplifiez le tableau si possible.

Exercice 4 : Reprenez les 3 couleurs précédentes, au lieu de les appliquer au triangle appliquez en **une** pour **chaque vertex**. Le résultat final doit ressembler à ceci (ce n'est pas grave si les couleurs ne sont pas dans le même ordre) :



Exercice 5 : Le dernier exercice va allier ce que nous avons vu dans le chapitre précédent et celui-ci. L'objectif est d'afficher un losange multicolore avec les couleurs de l'exercice 4, le dernier vertex doit utiliser le bleu (**255.0**) :



Solutions

Exercice 1 :

Secret (cliquez pour afficher)

Le but des exercices est de colorier un triangle, il n'y a donc que le tableau **couleurs** à modifier. Le reste du code ne change pas, on n'envoie toujours ce tableau à OpenGL grâce au **VertexAttrib 1**. Pour ce premier exercice, il fallait trouver :

Code : C++

```
// Couleurs  
  
float couleurs[] = {240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0,  
// Vertex 1  
                    240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0,  
// Vertex 2  
                    240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0};  
// Vertex 3
```

Il ne fallait pas oublier de diviser les valeurs que je vous avais données par **255.0**, sinon le code n'aurait pas fonctionné. 😊

Exercice 2 :

Secret (cliquez pour afficher)

Le principe reste le même que le premier exercice, il suffit juste de modifier le tableau **couleurs** :

Code : C++

```
// Couleurs

float couleurs[] = {230.0 / 255.0, 0.0, 230.0 / 255.0,      //
Vertex 1
                    230.0 / 255.0, 0.0, 230.0 / 255.0,      //
Vertex 2
                    230.0 / 255.0, 0.0, 230.0 / 255.0}; // //
Vertex 3
```

Exercice 3 :**Secret (cliquez pour afficher)**

Encore une fois, on conserve le même principe que les exercices 1 et 2. La couleur demandée n'était que le vert ici, ce qui allège un peu notre tableau :

Code : C++

```
// Couleurs

float couleurs[] = {0.0, 255.0 / 255.0, 0.0,      // Vertex 1
                     0.0, 255.0 / 255.0, 0.0,      // Vertex 2
                     0.0, 255.0 / 255.0, 0.0}; // Vertex 3
```

Cependant il y avait un moyen de simplifier cette déclaration. En effet, je vous ai précisé dans le cours que l'on pouvait simplifier les divisions des couleurs brutes (rouge, vert et bleu). Donc au lieu de faire l'opération **255.0 / 255.0**, on pouvait directement mettre la valeur **1.0**. Ce qui allégeait encore plus le tableau final :

Code : C++

```
// Couleurs

float couleurs[] = {0.0, 1.0, 0.0,      // Vertex 1
                     0.0, 1.0, 0.0,      // Vertex 2
                     0.0, 1.0, 0.0}; // Vertex 3
```

Exercice 4 :**Secret (cliquez pour afficher)**

Le but de ce dernier exercice était bien évidemment d'appliquer chacune des couleurs mises en place précédemment sur un vertex en particulier. Nous avions déjà vu ce principe avec l'exemple du triangle *multicolore* :

Code : C++

```
float couleurs[] = {240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0,
// Vertex 1
                    230.0 / 255.0, 0.0, 230.0 / 255.0,
// Vertex 2
```

```
    0.0, 1.0, 0.0};  
// Vertex 3
```

L'ordre des couleurs n'est pas important ici, ce n'est pas quelque chose de demandé.

Exercice 5 :

Secret (cliquez pour afficher)

La seule difficulté ici était qu'il fallait reprendre la couleur de deux vertices pour le second triangle. En effet, ceux-ci sont doublés pour pouvoir afficher le triangle du bas, il fallait donc doubler leur couleur :

Code : C++

```
// Couleurs  
  
float couleurs[] = {240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0,  
// Vertex 1  
                    230.0 / 255.0, 0.0, 230.0 / 255.0,  
// Vertex 2  
                    0.0, 1.0, 0.0,  
// Vertex 3  
                    240.0 / 255.0, 210.0 / 255.0, 23.0 / 255.0,  
// Vertex 4 (Copie du 1)  
                    0.0, 0.0, 1.0,  
// Vertex 5  
                    0.0, 1.0, 0.0};  
// Vertex 6 (Copie du 3)
```

Il faut bien faire correspondre votre couleur avec votre vertex pour avoir un bon affichage. 😊

D'ailleurs, le tableau de vertices ne change absolument pas par rapport à avant.

Code : C++

```
// Vertices  
  
float vertices[] = {-0.5, 0.0, 0.0, 1.0, 0.5, 0.0, //  
Triangle 1  
                  -0.5, 0.0, 0.0, -1.0, 0.5, 0.0}; //  
Triangle 2
```

Pensez à bien affecter la valeur **6** au paramètre **count** de la fonction **glDrawArrays()** pour afficher vos deux triangles, sinon elle n'en prendra en compte qu'un seul :

Code : C++

```
// Affichage des triangles  
  
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Ce chapitre (un peu compliqué je vous l'accorde) est enfin terminé. Vous savez maintenant ce que sont les shaders et à quoi ils

servent. Je pense faire une partie entièrement consacrée aux effets assez sympas que l'on peut réaliser, mais bon ce n'est pas pour maintenant. 😊

Les matrices

Alors là, nous allons aborder un chapitre assez compliqué. Nous allons découvrir ce que sont les matrices avec OpenGL, leur utilité, etc... C'est un chapitre assez technique et surtout rempli de mathématiques . Je vous conseille de bien le relire plusieurs fois jusqu'à que vous ayez tout compris, car les matrices sont indispensables avec OpenGL 

Introduction

Lançons nous tout de suite dans le monde des mathématiques. Je vous rappelle que si vous voulez développer des jeux-videos vous ne pouvez pas éviter les maths, elles sont indispensables au bon fonctionnement du jeu. Les premières notions que nous allons apprendre sont les **transformations**.

Qu'est-ce qu'une transformation vous allez me dire ? Au lieu d'une définition barbante que nous n'allons point retenir () je vais vous donner des exemples de transformations et vous comprendrez tout de suite ce que c'est :

- **Une translation** : consiste à faire "glisser" un ensemble de points sur un "rail" (un vecteur).
- **Une rotation** : consiste à faire pivoter un ensemble de points d'un angle Thêta par rapport à un point.
- **Une homothétie** : consiste simplement à agrandir ou à réduire une forme géométrique.

Une transformation est donc grossièrement une modification apportée à un ensemble de points ou repère dans un espace donné (soit 2D, soit 3D, etc ...). Les 3 transformations que vous voyez là sont les principales transformations utilisées dans OpenGL, en particulier la **translation**.



Ok, on sait ce qu'est une transformation maintenant, mais à quoi ça sert ?

Elles vont nous servir à placer tous nos objets (personnages, arbres, maisons, ...) dans notre monde 3D. A chaque fois que nous voudrons placer un objet nous utiliserons les transformations. Elles seront essentiellement utilisées sur le repère.

Les matrices sous OpenGL 2.1

Avant de parler du fonctionnement des matrices dans la version 3, je vais vous parler de celui des anciennes versions, ce sera plus facile à comprendre. De plus le principe ne change absolument pas d'une version à l'autre.

Tout d'abord, qu'est-ce qu'une **matrice** ? C'est un tableau de nombres ordonnés en lignes et en colonnes entourés par des parenthèses. Sa syntaxe est semblable à celle d'un vecteur mais avec plus de nombres :

$$\begin{pmatrix} 7 & 3 & 10 \\ 6 & 0 & 6 \\ 4 & 8 & 12 \end{pmatrix}$$

Une matrice n'est pas forcément un tableau de 9 cases, elle peut en contenir jusqu'à l'infini. Cependant ce n'est pas qu'un simple tableau, c'est une sorte de **super-vecteur** qui permet de faire pas mal de choses intéressantes. Un vecteur est en général utilisé en 3D pour gérer les points, les directions, les normales... Les matrices permettent de faire bien plus que ça. Elles servent principalement à convertir des données géométriques en données numériques. Il est plus facile de travailler avec des nombres qu'avec des compas et des équerres placés sur notre écran.  Voyons d'ailleurs pourquoi OpenGL a besoin des matrices :

- **Pour la projection** : grâce à cela nous allons pouvoir "**transformer**" un monde 3D en un monde 2D (jusqu'à nouvel ordre, un écran ne dispose que de deux dimensions).
- **Pour les transformations** : regroupant les transformations que je vous ai énumérées à l'instant.

Ce sont les besoins fondamentaux d'OpenGL pour utiliser la 3D. Dans les anciennes versions, les matrices étaient gérées automatiquement, on n'utilisait que quelques fonctions pour les créer et les utiliser. Depuis la version 3.1, ces fonctions sont supprimées, car l'approche des jeux d'aujourd'hui est différente. Heureusement pour nous, nous avons en notre possession une librairie mathématique du nom de **GLM** qui permet de gérer les matrices (et bien plus), mais avant de commencer à les manipuler nous allons faire un peu de théorie pour bien comprendre ce que nous faisons.

En définitif, nous aurons besoin de deux matrices pour coder avec OpenGL :

- **La matrice de projection** : qui sert à transformer notre monde 3D en un monde 2D affichable sur l'écran.
- **La matrice modelview** (ou visualisation de modèle) : qui sera la matrice principale, c'est sur elle que nous allons appliquer nos transformations.

Chaque vertex sera multiplié par ces deux matrices pour pouvoir être affiché sur notre écran, le tout en nous donnant une impression de 3D (magique n'est-ce pas ).

Partie théorique

Dans cette partie nous allons apprendre ce que sont les matrices et comment les utiliser pour nos programmes. Pour simplifier cet apprentissage, je ne vais vous montrer que la multiplication de matrices car c'est une notion que nous retrouverons assez souvent dans ce tuto. De plus, cela vous donnera une bonne idée des calculs matriciels sans pour autant voir les choses les plus compliquées (inversion, déterminant, etc.). 

Matrice carrée

On commence tout de suite cette partie par les **matrices carrées**.

Une **matrice carrée** est une matrice ayant le même nombres de colonnes et de lignes :

$$\begin{pmatrix} 7 & 3 & 10 \\ 6 & 0 & 6 \\ 4 & 8 & 12 \end{pmatrix}$$

Il y a 3 rangées de **colonnes** et 3 rangées de **lignes**, c'est donc ce que l'on appelle une matrice carrée. Son nom complet est d'ailleurs : **matrice carrée d'ordre 3**. Le chiffre à la fin permet de spécifier la taille. Si on avait mis le chiffre 4, alors la matrice aurait eu 4 colonnes et 4 lignes. Nous n'utiliserons que les matrices carrées dans OpenGL. Et tant mieux, car ça simplifie grandement les choses .

Matrice d'identité

Sous ce nom barbare se cache en réalité la matrice la plus simple qu'il soit. Elle a la particularité d'avoir toutes ses valeurs égales à 0 hormis les valeurs de sa diagonale qui, elles, sont égales à 1. Quelle que soit la taille de la matrice, on retrouvera toujours cette particularité :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Si nous multiplions un vecteur par une telle matrice, le vecteur ne sera absolument pas changé  :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 5 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 4 \end{pmatrix}$$

Nous allons étudier la multiplication matricielle dans un instant. Mais vous pouvez déjà retenir ce principe : si un vecteur est multiplié par une matrice d'identité, alors il ne sera pas modifié.

Multiplication d'un vecteur par une matrice

Partie 1 : la vérification

Dans ce premier exemple, je vais prendre une matrice carrée d'ordre 3 pour simplifier les choses. Sachez cependant que le principe est le même quelle que soit la taille de la matrice.

$$\begin{pmatrix} 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

Voyons comment arriver à ce résultat.

Premièrement, il faut vérifier que le **nombre de colonnes** de la matrice soit égal au **nombre de coordonnées** du vecteur. Si ce n'est pas le cas alors la multiplication est impossible.

$$\begin{pmatrix} 5 \\ 1 \\ 4 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 6 \\ 7 \\ 1 \end{pmatrix} \quad \times \quad \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} \quad \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

La matrice possède 3 colonnes et le vecteur possède 3 coordonnées. La multiplication est donc possible.



Attention ! La matrice sera toujours à **GAUCHE** de la multiplication et le vecteur sera toujours à **DROITE** !

Prenons un autre exemple :

$$\begin{pmatrix} 7 \\ 6 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 0 \end{pmatrix} \quad \times \quad \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} \quad \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

La matrice possède 2 colonnes et le vecteur 3 coordonnées. La multiplication est impossible.
Vous avez compris le principe ? Bien, passons à la suite.

Partie 2 : La multiplication

Attaquons la partie la plus intéressante 😊. La multiplication d'une matrice et d'un vecteur s'effectue comme ceci : pour une seule coordonnée (x, y, ou z) du vecteur résultat, nous allons faire la **somme des multiplications** de chaque **nombre d'une ligne de la matrice** par **chaque nombre correspondant du vecteur**. Un peu barbant comme explication, rien ne vaut un bon exemple :

Commençons déjà par la première ligne, vous devriez comprendre le principe une fois cet exemple fini :

$$\begin{pmatrix} 5 & 1 & 6 \\ \square & \square & \square \\ \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} (5 * 3) + (1 * 4) + (6 * 2) \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 15 + 4 + 12 \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 31 \\ \square \\ \square \end{pmatrix}$$

Il faut multiplier les nombres de même couleur sur le schéma, puis additionner les différents résultats. Une ligne de la matrice ne donnera qu'une seule coordonnée du vecteur résultat, ici la **coordonnée x**. Passons à la deuxième ligne :

$$\begin{pmatrix} \square & \square & \square \\ 1 & 4 & 7 \\ \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} (1 * 3) + (4 * 4) + (7 * 2) \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 3 + 16 + 14 \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 33 \\ \square \\ \square \end{pmatrix}$$

Vous comprenez le principe ? La première ligne nous a donné la première coordonnée, la deuxième ligne nous donne la deuxième coordonnée. Nous multiplions chaque ligne de la matrice par toutes les valeurs du vecteur. Passons à la troisième ligne :

$$\begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} \square \\ (4 * 3) + (2 * 4) + (1 * 2) \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 12 + 8 + 2 \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 22 \\ \square \end{pmatrix}$$

La troisième ligne de la matrice nous donne la troisième coordonnée. Si la matrice avait eu 4 lignes (et donc 4 colonnes) alors il y aurait eu une quatrième opération du même type pour la quatrième coordonnée. Si on récapitule tout ça, on a :

$$\begin{pmatrix} 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} (5 * 3) + (1 * 4) + (6 * 2) \\ (1 * 3) + (4 * 4) + (7 * 2) \\ (4 * 3) + (2 * 4) + (1 * 2) \end{pmatrix} = \begin{pmatrix} 15 + 4 + 12 \\ 3 + 16 + 14 \\ 12 + 8 + 2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

Vous avez compris ? Je vous donne un autre exemple pour comparer les différents résultats :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} (3 * 2) + (7 * 4) + (2 * 1) \\ (1 * 2) + (5 * 4) + (7 * 1) \\ (4 * 2) + (2 * 4) + (8 * 1) \end{pmatrix} = \begin{pmatrix} 6 + 28 + 2 \\ 2 + 20 + 7 \\ 8 + 8 + 8 \end{pmatrix} = \begin{pmatrix} 36 \\ 29 \\ 24 \end{pmatrix}$$

N'hésitez pas à bien relire et essayez de comprendre ces deux exemples. C'est la base de la multiplication matricielle.



Je vous rappelle que la matrice est toujours à **gauche** et le vecteur est toujours à **droite**.

Passons à un pseudo-exemple, vous n'avez pas oublié ce qu'est la matrice d'identité j'espère 🤔, on sait qu'un vecteur multiplié par cette matrice ne changera pas. Maintenant que l'on connaît un peu la multiplication on va voir pourquoi le vecteur n'est pas modifié :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 4 \\ 6 \\ 3 \end{pmatrix} = \begin{pmatrix} (1 * 4) + (0 * 6) + (0 * 3) \\ (0 * 4) + (1 * 6) + (0 * 3) \\ (0 * 4) + (0 * 6) + (1 * 3) \end{pmatrix} = \begin{pmatrix} 4 + 0 + 0 \\ 0 + 6 + 0 \\ 0 + 0 + 3 \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \\ 3 \end{pmatrix}$$

Oh magie, le résultat ne change pas. Vous comprenez pourquoi cette matrice est la plus simple à utiliser. 😊

Je vais maintenant vous donner un exemple de multiplication avec une matrice carrée d'ordre 4. Le principe ne change absolument pas, nous allons voir cet exemple étape par étape pour bien comprendre. Commençons :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 3 \\ 5 & 0 & 1 & 0 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix}$$

La matrice possède 4 colonnes et le vecteur 4 lignes, la multiplication est possible. Maintenant on passe à la première étape :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ \square & \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} (1 * 3) + (3 * 4) + (2 * 5) + (6 * 1) \\ \square \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 3 + 12 + 10 + 6 \\ \square \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 31 \\ \square \\ \square \\ \square \\ \square \end{pmatrix}$$

Vous ne voyez qu'une seule différence : on ajoute à la somme une **multiplication supplémentaire**. En effet, n'oubliez pas que l'on multiplie les nombres de la matrice avec les nombres correspondants du vecteur. Pour la deuxième ligne c'est pareil :

$$\begin{pmatrix} \square & \square & \square & \square \\ 2 & 1 & 2 & 3 \\ \square & \square & \square & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} (2 * 3) + (1 * 4) + (2 * 5) + (3 * 1) \\ \square \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 6 + 4 + 10 + 3 \\ \square \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 23 \\ \square \\ \square \\ \square \\ \square \end{pmatrix}$$

Pour la troisième ligne ça ne change pas :

$$\begin{pmatrix} \square & \square & \square & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \\ 5 & 0 & 1 & 0 \\ \square & \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} \square \\ (5 * 3) + (0 * 4) + (1 * 5) + (0 * 1) \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 15 + 0 + 5 + 0 \\ \square \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 20 \\ \square \\ \square \\ \square \end{pmatrix}$$

Ah, une quatrième ligne, eh bien oui c'est plus long 🤪. Mais ça ne change toujours pas :

$$\begin{pmatrix} \square & \square & \square & \square \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ (1 * 3) + (2 * 4) + (3 * 5) + (2 * 1) \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ 3 + 8 + 15 + 2 \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ 28 \\ \square \\ \square \end{pmatrix}$$

Ce qui nous donne au final :

$$\begin{pmatrix} \color{red}{1} & \color{teal}{3} & \color{green}{2} & \color{blue}{6} \\ \color{red}{2} & \color{teal}{1} & \color{green}{2} & \color{blue}{3} \\ \color{red}{5} & \color{teal}{0} & \color{green}{1} & \color{blue}{0} \\ \color{red}{1} & \color{teal}{2} & \color{green}{3} & \color{blue}{2} \end{pmatrix} \times \begin{pmatrix} \color{red}{3} \\ \color{teal}{4} \\ \color{green}{5} \\ \color{blue}{1} \end{pmatrix} = \begin{pmatrix} (\color{red}{1} * \color{teal}{3}) + (\color{teal}{3} * \color{blue}{4}) + (\color{green}{2} * \color{green}{5}) + (\color{blue}{6} * \color{blue}{1}) \\ (\color{red}{2} * \color{teal}{3}) + (\color{teal}{1} * \color{blue}{4}) + (\color{green}{2} * \color{green}{5}) + (\color{blue}{3} * \color{blue}{1}) \\ (\color{red}{5} * \color{teal}{3}) + (\color{teal}{0} * \color{blue}{4}) + (\color{green}{1} * \color{green}{5}) + (\color{blue}{0} * \color{blue}{1}) \\ (\color{red}{1} * \color{teal}{3}) + (\color{teal}{2} * \color{blue}{4}) + (\color{green}{3} * \color{green}{5}) + (\color{blue}{2} * \color{blue}{1}) \end{pmatrix} = \begin{pmatrix} \color{red}{3} + \color{teal}{12} + \color{green}{10} + \color{blue}{6} \\ \color{red}{6} + \color{teal}{4} + \color{green}{10} + \color{blue}{3} \\ \color{red}{15} + \color{teal}{0} + \color{green}{5} + \color{blue}{0} \\ \color{red}{3} + \color{teal}{8} + \color{green}{15} + \color{blue}{2} \end{pmatrix} = \begin{pmatrix} 31 \\ 23 \\ 20 \\ 28 \end{pmatrix}$$

Vous voyez, le principe de la multiplication ne change absolument pas.

Piouf, cette partie était un peu difficile à comprendre. Je vais vous donner quelques exercices pour vous entraîner à la multiplication. Faites-les sérieusement, vous avez besoin de comprendre ce que l'on vient de faire pour comprendre la suite. Sinon vous serez complètement largués 😕.

Exercices de multiplications

Exercice 1 :

$$\begin{pmatrix} 4 & 2 & 3 \\ 1 & 3 & 2 \\ 0 & 4 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 5 \\ 3 \end{pmatrix}$$

Exercice 2 :

$$\begin{pmatrix} 5 & 7 & 8 \\ 3 & 1 & 6 \\ 7 & 4 & 2 \end{pmatrix} \times \begin{pmatrix} 4 \\ 7 \\ 9 \\ 3 \end{pmatrix}$$

Exercice 3 :

$$\begin{pmatrix} 3 & 1 & 4 & 2 \\ 4 & 5 & 1 & 3 \\ 2 & 7 & 3 & 4 \\ 1 & 6 & 5 & 1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 0 \\ 3 \\ 5 \end{pmatrix}$$

Solutions

Exercice 1 :

$$\begin{pmatrix} 23 \\ 22 \\ 23 \end{pmatrix}$$

Exercice 2 :

La multiplication est impossible, la matrice possède 3 colonnes alors que le vecteur possède 4 coordonnées.

Exercice 3 :

$$\begin{pmatrix} 28 \\ 26 \\ 33 \\ 22 \end{pmatrix}$$

Multiplication de deux matrices

La multiplication de deux matrices peut sembler plus compliquée à première vue mais si vous avez compris ce que l'on a fait avant, alors vous savez déjà multiplier deux matrices.

Partie 1

Avant tout, pour pouvoir multiplier deux matrices carrées entre-elles il faut absolument que les deux matrices soient de la même taille (donc du même ordre), c'est-à-dire qu'elles doivent avoir le même nombre de lignes et le même nombre de colonnes.

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix}$$

Les deux matrices ont la même taille, on peut donc les multiplier.

Prenons un autre exemple :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Ici, les deux matrices n'ont pas la même taille nous ne pouvons pas les multiplier.

Il ne devrait y avoir rien de compliqué pour le moment. 😊

Partie 2

Pour pouvoir multiplier deux matrices carrées, nous allons utiliser une petite combine. Nous allons couper la deuxième matrice en 3 vecteurs (ou 4 selon la taille), puis nous appliquerons la multiplication que nous avons vue à l'instant pour chacun de ces vecteurs. Prenons deux matrices cobayes :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix}$$



Attention, l'ordre des matrices est important ! $M1 \times M2$ ne sera pas forcément égal à $M2 \times M1$. Faites attention à l'ordre de votre multiplication !

Bien, coupons la seconde matrice en 3 vecteurs :

$$\begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix}$$

Maintenant il nous suffit d'appliquer la multiplication d'une matrice et d'un vecteur pour chaque vecteur que nous venons de créer. Voici ce que ça donne pour le premier :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} (3 * 5) + (7 * 3) + (2 * 1) \\ (1 * 5) + (5 * 3) + (7 * 1) \\ (4 * 5) + (2 * 3) + (8 * 1) \end{pmatrix} = \begin{pmatrix} 15 + 21 + 2 \\ 5 + 15 + 7 \\ 20 + 6 + 8 \end{pmatrix} = \begin{pmatrix} 38 \\ 27 \\ 34 \end{pmatrix}$$

Au tour du deuxième :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} (3 * 4) + (7 * 0) + (2 * 2) \\ (1 * 4) + (5 * 0) + (7 * 2) \\ (4 * 4) + (2 * 0) + (8 * 2) \end{pmatrix} = \begin{pmatrix} 12 + 0 + 4 \\ 4 + 0 + 14 \\ 16 + 0 + 16 \end{pmatrix} = \begin{pmatrix} 16 \\ 18 \\ 32 \end{pmatrix}$$

Quand il y a un "0" dans une matrice ça nous facilite grandement le calcul 😊 . Bref, on fait la même chose avec le dernier vecteur :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} (3 * 3) + (7 * 1) + (2 * 5) \\ (1 * 3) + (5 * 1) + (7 * 5) \\ (4 * 3) + (2 * 1) + (8 * 5) \end{pmatrix} = \begin{pmatrix} 9 + 7 + 10 \\ 3 + 5 + 35 \\ 12 + 2 + 40 \end{pmatrix} = \begin{pmatrix} 26 \\ 43 \\ 54 \end{pmatrix}$$

Nous obtenons au final 3 vecteurs fraîchement calculés. Il nous suffit ensuite de les assembler dans **l'ordre de la division de la matrice !**

$$\begin{pmatrix} 38 \\ 27 \\ 34 \end{pmatrix} \quad \begin{pmatrix} 16 \\ 18 \\ 32 \end{pmatrix} \quad \begin{pmatrix} 26 \\ 43 \\ 54 \end{pmatrix} \Rightarrow \begin{pmatrix} 38 & 16 & 26 \\ 27 & 18 & 43 \\ 34 & 32 & 54 \end{pmatrix}$$

Au final, nous obtenons :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix} = \begin{pmatrix} 38 & 16 & 26 \\ 27 & 18 & 43 \\ 34 & 32 & 54 \end{pmatrix}$$



Je vous rappelle encore une fois que l'ordre de la multiplication compte. **M1 x M2 est différent de M2 x M1 !**

Vous voyez, une fois que vous avez compris la première multiplication, vous savez déjà multiplier deux matrices. Évidemment, la taille de la matrice ne change toujours pas le principe, voyons ensemble un exemple de multiplication de deux matrices carrées d'ordre 4.

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 3 & 2 & 0 \\ 4 & 1 & 2 & 3 \\ 3 & 0 & 1 & 3 \\ 1 & 3 & 0 & 2 \end{pmatrix}$$

Coupons la seconde matrice en 4 vecteurs (et oui car c'est une matrice carrée d'ordre 4 et pas 3) :

$$\begin{pmatrix} 2 & 3 & 2 & 0 \\ 4 & 1 & 2 & 3 \\ 3 & 0 & 1 & 3 \\ 1 & 3 & 0 & 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 4 \\ 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \\ 3 \\ 2 \end{pmatrix}$$

Il nous suffit maintenant d'appliquer la multiplication "**Matrice - Vecteur**" sur les 4 vecteurs que nous venons de créer. Voici le premier résultat :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 \\ 4 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} (1 * 2) + (3 * 4) + (2 * 3) + (6 * 1) \\ (2 * 2) + (1 * 4) + (2 * 3) + (0 * 1) \\ (5 * 2) + (0 * 4) + (1 * 3) + (3 * 1) \\ (1 * 2) + (2 * 4) + (3 * 3) + (2 * 1) \end{pmatrix} = \begin{pmatrix} 2 + 12 + 6 + 6 \\ 4 + 4 + 6 + 0 \\ 10 + 0 + 3 + 3 \\ 2 + 8 + 9 + 2 \end{pmatrix} = \begin{pmatrix} 26 \\ 14 \\ 16 \\ 21 \end{pmatrix}$$

Puis le second :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\ 1 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} (1 * 3) + (3 * 1) + (2 * 0) + (6 * 3) \\ (2 * 3) + (1 * 1) + (2 * 0) + (0 * 3) \\ (5 * 3) + (0 * 1) + (1 * 0) + (3 * 3) \\ (1 * 3) + (2 * 1) + (3 * 0) + (2 * 3) \end{pmatrix} = \begin{pmatrix} 3 + 3 + 0 + 18 \\ 6 + 1 + 0 + 0 \\ 15 + 0 + 0 + 9 \\ 3 + 2 + 0 + 6 \end{pmatrix} = \begin{pmatrix} 24 \\ 7 \\ 24 \\ 11 \end{pmatrix}$$

Le troisième :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 \\ 2 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} (1 * 2) + (3 * 2) + (2 * 1) + (6 * 0) \\ (2 * 2) + (1 * 2) + (2 * 1) + (0 * 0) \\ (5 * 2) + (0 * 2) + (1 * 1) + (3 * 0) \\ (1 * 2) + (2 * 2) + (3 * 1) + (2 * 0) \end{pmatrix} = \begin{pmatrix} 2 + 6 + 2 + 0 \\ 4 + 2 + 2 + 0 \\ 10 + 0 + 1 + 0 \\ 2 + 4 + 3 + 0 \end{pmatrix} = \begin{pmatrix} 10 \\ 8 \\ 11 \\ 9 \end{pmatrix}$$

Et enfin le quatrième et dernier vecteur :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 3 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} (1 * 0) + (3 * 3) + (2 * 3) + (6 * 2) \\ (2 * 0) + (1 * 3) + (2 * 3) + (0 * 2) \\ (5 * 0) + (0 * 3) + (1 * 3) + (3 * 2) \\ (1 * 0) + (2 * 3) + (3 * 3) + (2 * 2) \end{pmatrix} = \begin{pmatrix} 0 + 9 + 6 + 12 \\ 0 + 3 + 6 + 0 \\ 0 + 0 + 3 + 6 \\ 0 + 6 + 9 + 4 \end{pmatrix} = \begin{pmatrix} 27 \\ 9 \\ 9 \\ 19 \end{pmatrix}$$

Maintenant on réunit les vecteurs résultats **dans le bon ordre** :

$$\begin{pmatrix} 26 \\ 14 \\ 16 \\ 21 \end{pmatrix} \quad \begin{pmatrix} 24 \\ 7 \\ 24 \\ 11 \end{pmatrix} \quad \begin{pmatrix} 10 \\ 8 \\ 11 \\ 9 \end{pmatrix} \quad \begin{pmatrix} 27 \\ 9 \\ 9 \\ 19 \end{pmatrix} \Rightarrow \begin{pmatrix} 26 & 24 & 10 & 27 \\ 14 & 7 & 8 & 9 \\ 16 & 24 & 11 & 9 \\ 21 & 11 & 9 & 19 \end{pmatrix}$$

Voici donc le résultat final :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 0 \\ 5 & 0 & 1 & 3 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 3 & 2 & 0 \\ 4 & 1 & 2 & 3 \\ 3 & 0 & 1 & 3 \\ 1 & 3 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 26 & 24 & 10 & 27 \\ 14 & 7 & 8 & 9 \\ 16 & 24 & 11 & 9 \\ 21 & 11 & 9 & 19 \end{pmatrix}$$

Vous remarquez que la multiplication se passe exactement de la même façon que l'on soit en présence de matrices à 3 colonnes ou à 4 colonnes ou même à 1000 colonnes 🎉. Passons maintenant à quelques exercices pour vous entraîner. C'est important je le répète, essayez de faire ces exercices sérieusement.

Exercices de multiplications

Exercice 1 :

$$\begin{pmatrix} 3 & 1 & 4 & 2 \\ 4 & 5 & 1 & 3 \\ 2 & 7 & 3 & 4 \\ 1 & 6 & 5 & 1 \end{pmatrix} \times \begin{pmatrix} 4 & 2 & 3 \\ 1 & 3 & 2 \\ 0 & 4 & 1 \end{pmatrix}$$

Exercice 2 :

$$\begin{pmatrix} 3 & 1 & 0 \\ 1 & 2 & 4 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix}$$

Exercice 3 :

$$\begin{pmatrix} 1 & 3 & 2 & 6 \\ 2 & 1 & 2 & 3 \\ 5 & 0 & 1 & 0 \\ 1 & 2 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 3 & 1 & 4 & 2 \\ 4 & 5 & 1 & 3 \\ 2 & 7 & 3 & 4 \\ 1 & 6 & 5 & 1 \end{pmatrix}$$

Solutions

Exercice 1 :

La multiplication est impossible, la première matrice possède 4 lignes et 4 colonnes alors que la seconde matrice ne possède que 3 lignes et 3 colonnes.

Exercice 2 :

$$\begin{pmatrix} 18 & 12 & 10 \\ 15 & 12 & 25 \\ 27 & 18 & 19 \end{pmatrix}$$

Exercice 3 :

$$\begin{pmatrix} 25 & 66 & 43 & 25 \\ 17 & 39 & 30 & 18 \\ 17 & 12 & 23 & 14 \\ 19 & 44 & 25 & 22 \end{pmatrix}$$

Et voilà, la partie la plus compliquée de ce chapitre est enfin terminée ! La multiplication matricielle est une notion difficile à comprendre (même s'il en existe bien d'autres plus complexes) mais vous êtes maintenant capables de la maîtriser. 😊

Transformations au niveau matricielle

Le but de la partie précédente était d'apprendre à multiplier deux matrices. J'ai volontairement inclus des exercices avec des matrices carrées d'ordre 4 car OpenGL aura besoin la plupart du temps de ce genre de matrices (et heureusement !). Dans cette partie, nous allons faire passer les transformations de la géométrie à l'algèbre (les nombres).

Comme vous le savez les transformations sont des outils géométriques, pour calculer une rotation par exemple nous avons besoin d'un rapporteur. Or c'est un peu compliqué de poser notre rapporteur sur l'écran, surtout si l'angle se trouve derrière le dos d'un personnage 😊. Pour pouvoir utiliser les transformations numériquement nous devons utiliser... les matrices. 😊

La translation

La translation permet de déplacer un ensemble de points ou une forme dans un espace donné. En gros, on prend un vecteur qui servira de "rail" puis on fera glisser nos points sur ce rail. La forme finale ne sera pas modifiée, elle sera juste déplacée. 😊

Une translation en 3 dimensions se traduit par la matrice suivante :

$$\begin{pmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- **X** : C'est la coordonnée x du vecteur de translation.
- **Y** : C'est la coordonnée y du vecteur de translation.
- **Z** : C'est la coordonnée z du vecteur de translation.

Vous remarquerez que cette matrice ressemble beaucoup à la matrice d'identité, il n'y a que les coordonnées du vecteur en plus.

L'homothétie

Une homothétie permet d'agrandir ou de réduire une forme géométrique, voici la matrice correspondante :

$$\begin{pmatrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- **X** : C'est le facteur multiplicatif de l'axe x.
- **Y** : C'est le facteur multiplicatif de l'axe y.
- **Z** : C'est le facteur multiplicatif de l'axe z.

La rotation

Attention ! La matrice que vous allez voir est certainement la matrice la plus compliquée de tout le tutoriel 🍪. Cependant, il est inutile de la retenir, elle est trop complexe nous la verrons jamais directement :

$$\begin{pmatrix} xx(1 - \cos(\theta)) + \cos(\theta) & xy(1 - \cos(\theta)) - z\sin(\theta) & xz(1 - \cos(\theta)) + y\sin(\theta) & 0 \\ xy(1 - \cos(\theta)) + z\sin(\theta) & yy(1 - \cos(\theta)) + \cos(\theta) & yz(1 - \cos(\theta)) - x\sin(\theta) & 0 \\ xz(1 - \cos(\theta)) - y\sin(\theta) & yz(1 - \cos(\theta)) + x\sin(\theta) & zz(1 - \cos(\theta)) + \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Je vous avais dit que cette matrice était... particulière (pour vous dire, je ne la connais pas par cœur moi-même). Grâce à elle, nous pouvons faire pivoter en ensemble de points d'un angle **thêta** autour d'un axe défini par les **coordonnées (x, y, z)**. Nul besoin de retenir cette matrice, je le répète. 😊

Transformations au niveau géométrique

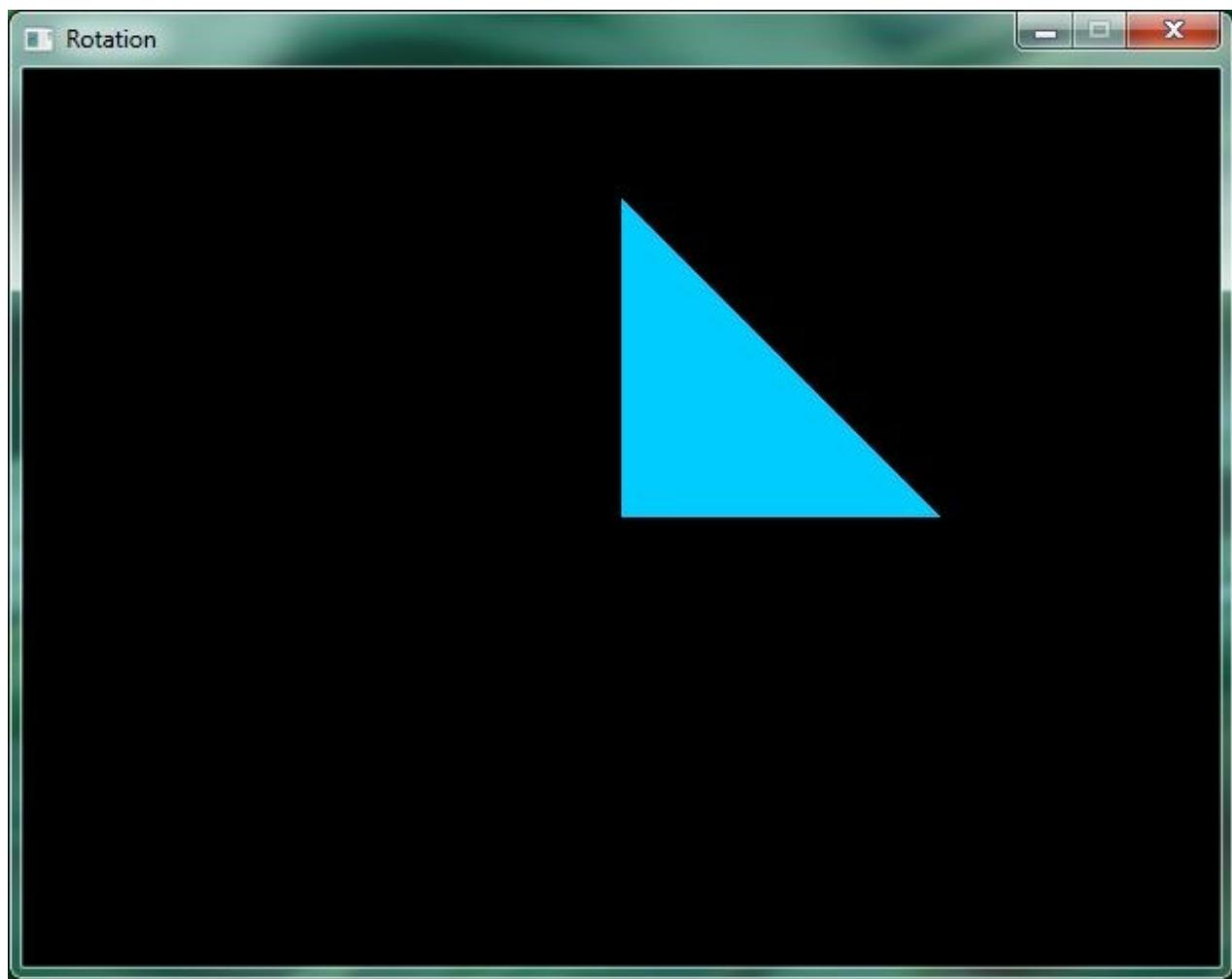
Durant l'introduction, je vous ai brièvement parlé de deux matrices : la **projection** (qui permet de convertir un monde 3D en un monde 2D affichable sur notre écran) et la **modelview** (qui permet de placer nos objets dans ce même monde 3D).

C'est sur cette dernière matrice que l'on effectuera toutes les transformations que l'on a vues, à savoir : la translation, la rotation et l'homothétie. Voyons d'ailleurs comment placer un objet dans un monde 3D :

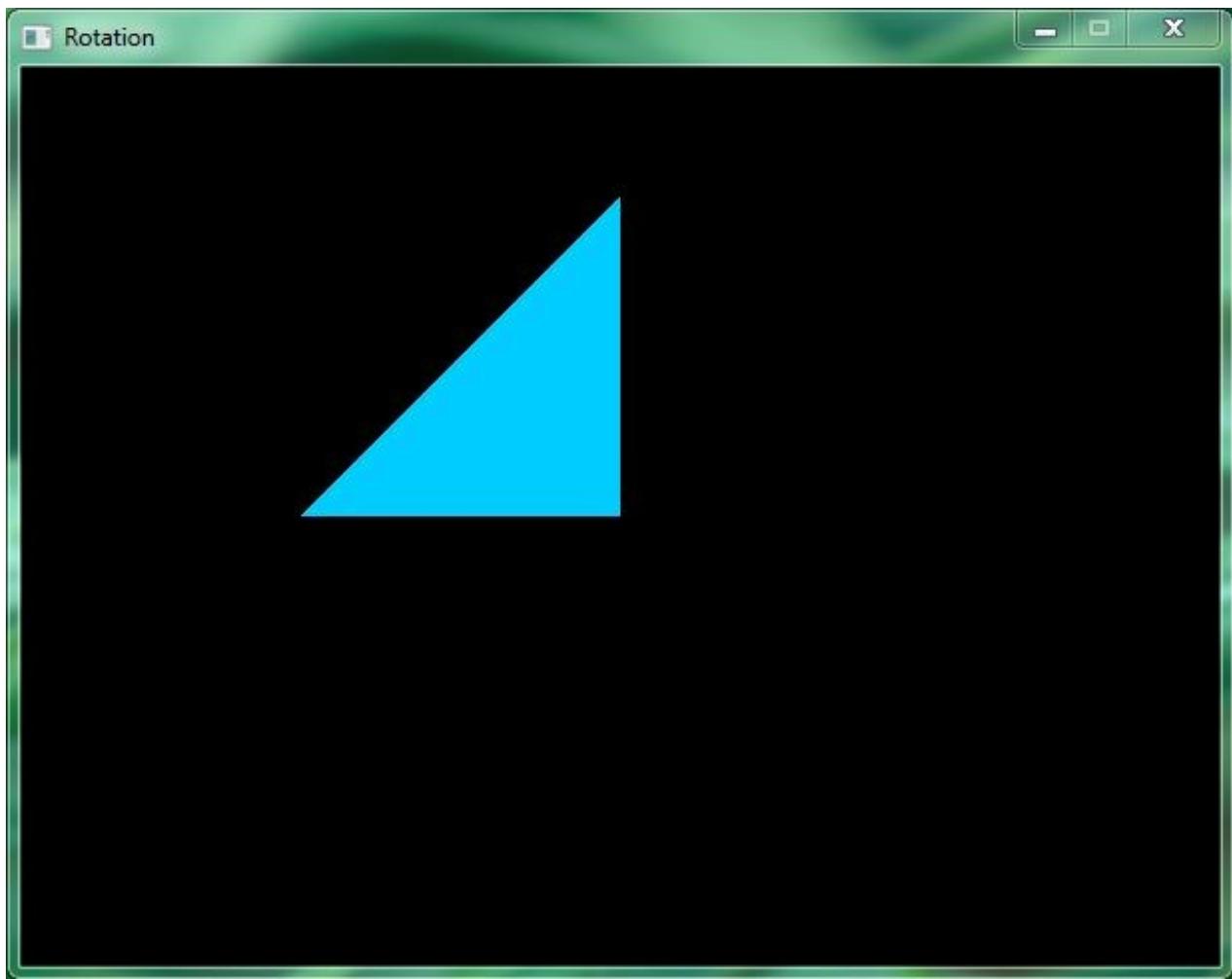
- On crée la matrice **modelview** (une seule fois pour tout le programme, pas une seule fois par objet).
- Puis on effectue une transformation sur cette matrice (par exemple une rotation de 90° sur l'axe Z).
- Enfin on dessine ce que l'on veut dessiner et vu que l'on a effectué une rotation de 90°, alors notre objet sera penché de 90°.

Voyons ce que cela donne si je fais pivoter un triangle de 90°...

Voici un triangle avant la rotation :



Et le revoilà après :



Vous voyez ce qui c'est passé ? Le triangle a pivoté de 90° grâce à une transformation.



La rotation se fait selon le sens trigonométrique et non selon le sens horaire.

Les Transformations sous OpenGL

Aaahh... on commence à relier OpenGL et les mathématiques (enfin !). Il est temps de voir le comportement des transformations dans un programme.

La première chose à savoir est que chaque transformation (rotation, ...) sera effectuée non pas sur un objet mais sur le **REPÈRE** du monde, c'est-à-dire que si l'on veut faire pivoter un objet de 120° alors il faudra faire pivoter non pas l'objet en lui-même mais son **REPÈRE**.

Par exemple, si vous voulez afficher un objet à 300 mètres d'un autre, vous n'allez pas ajouter 300 unités de longueur à chaque vertex de l'objet, autant garder les vertices déjà définis.



Hein ??? J'ai rien compris. 🤔

Bon, mettons que vous ayez un méchant soldat ennemi de 500 vertices. Vous n'allez pas modifier ses 500 vertices pour afficher un autre ennemi deux mètres plus loin. Ça serait trop couteux en ressources, surtout si vous affichez une armée de 200 soldats...

À la place, on modifera la position du repère (qui est en fait la matrice **modelview**) pour chaque objet que l'on veut dessiner. Un personnage gardera ses 500 vertices intactes quelque soit sa position.

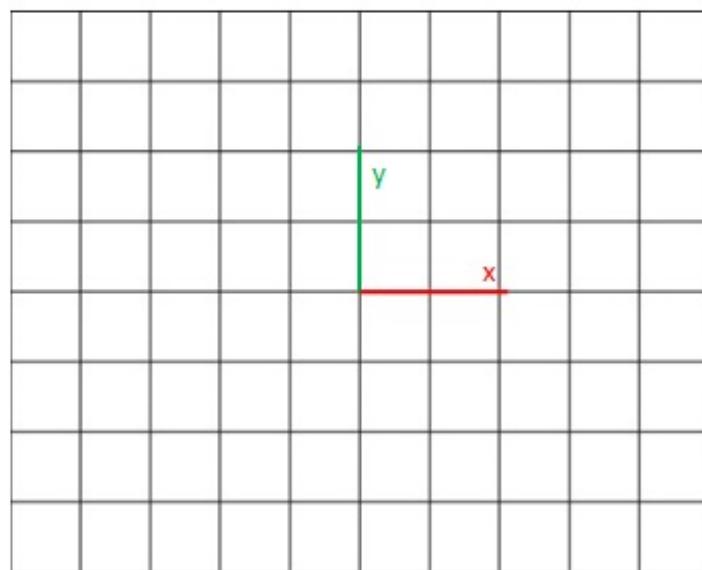
Vous voyez la différence entre modifier toute une scène de dizaines de milliers de vertices et modifier une matrice de 16 valeurs.

Illustrations

Je vais vous donner quelques exemples de transformations effectuées sur un repère, ce sera plus simple à comprendre si vous voyez ce que donne chaque transformation.

Le repère de base

Le repère de base est en fait un repère qui n'a pas encore été modifié, c'est sur celui-ci que l'on effectuera notre première transformation. Graphiquement, on représente un repère comme ceci :

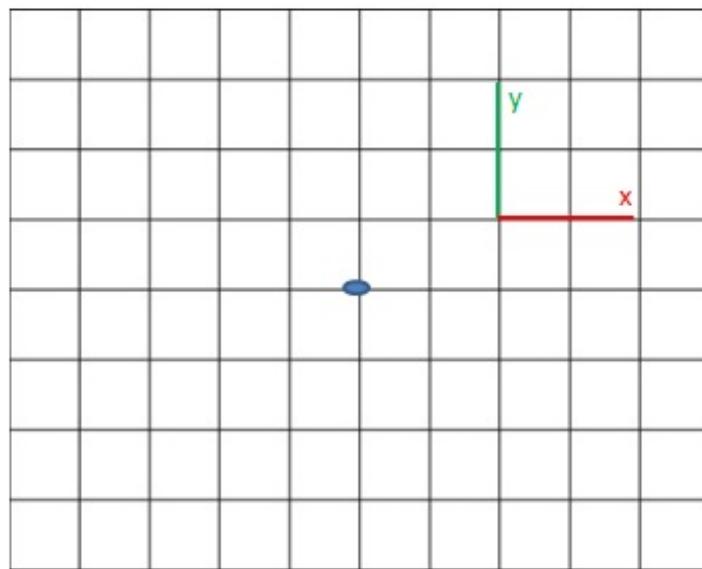


Au niveau des matrices, ce repère correspond simplement à une matrice d'identité d'ordre 4 :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La translation

La translation est la transformation la plus simple 😊. N'oubliez pas que c'est le REPÈRE qui est modifié. Ici, on translate le repère par rapport au vecteur V(2, 1), soit deux unités de longueur sur l'axe X et 1 unité sur l'axe Y :

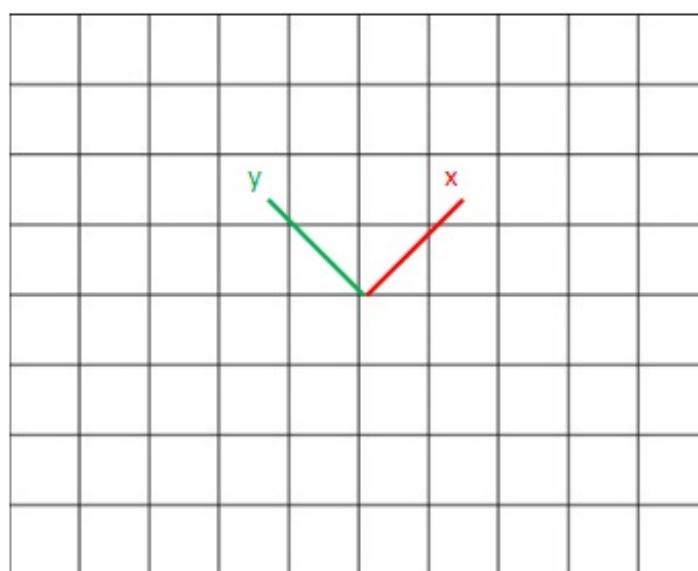


Avec les matrices on multiplierait la matrice **modelview** par la matrice de translation suivante :

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La rotation

Le principe ne change pas pour la rotation, on effectue la transformation sur le repère. Voici un exemple d'une rotation de 45° sur l'axe Z :

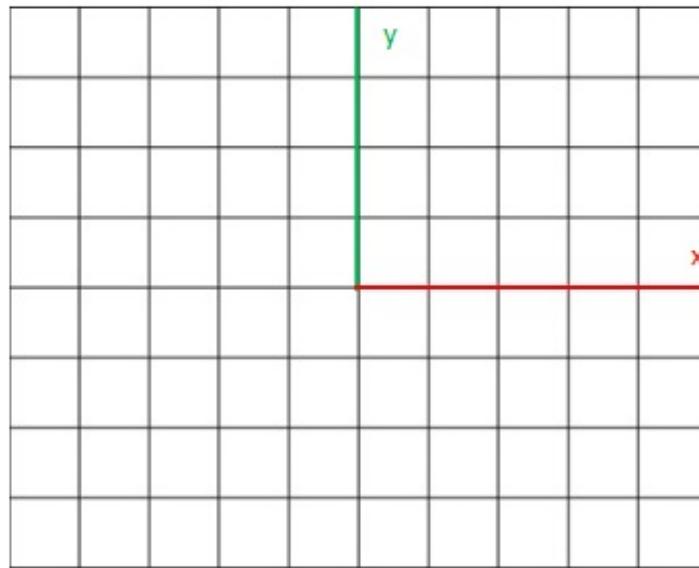


Pour la rotation, on multipliera la matrice modelview par la matrice de rotation (toujours aussi effrayante 😱) :

$$\begin{pmatrix} xx(1 - \cos(45)) + \cos(45) & xy(1 - \cos(45)) - z\sin(45) & xz(1 - \cos(45)) + y\sin(45) & 0 \\ xy(1 - \cos(45)) + z\sin(45) & yy(1 - \cos(45)) + \cos(45) & yz(1 - \cos(45)) - x\sin(45) & 0 \\ xz(1 - \cos(45)) - y\sin(45) & yz(1 - \cos(45)) + x\sin(45) & zz(1 - \cos(45)) + \cos(45) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

L'homothétie

En temps normal, avec une homothétie on modifie la taille d'une forme géométrique. Sauf qu'ici on modifie la taille du repère, donc on modifie la taille de chaque unité de longueur. Voici ce que donne une homothétie de coordonnées (2, 2) sur le repère :



Vous voyez que la taille du repère à changé, désormais si on fait une translation par un vecteur(2, 1) alors la translation sera plus grande et le repère se retrouvera plus loin.

Pour ce qui est de la matrice modelview, il suffira de la multiplier par la matrice suivante :

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Attention, la coordonnée Z est mis à 1 volontairement. Nous expliquerons pourquoi un peu plus loin.

Les matrices et la boucle principale

Vous n'êtes pas sans savoir qu'un jeu se passe en grande partie dans ce que l'on appelle la **boucle principale**, c'est une boucle qui va se répéter indéfiniment jusqu'à ce que le joueur arrête de jouer (enfin pas vraiment, mais partons de ce principe).

Il faudra à chaque tour de boucle réinitialiser la matrice modelview pour ne pas qu'elle garde les traces de transformations du tour précédent. Si c'était le cas, le jeu ne se redessinerait jamais au même endroit et serait totalement difforme.

Voici ce qui se passera à chaque tour de boucle :

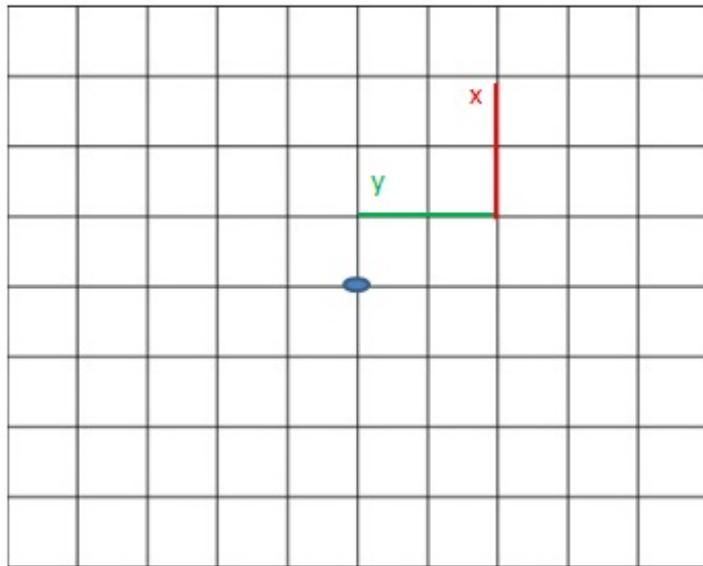
- On replacera notre **repère à sa position initiale**, grâce à la fonction `LoadIdentity`.
- On effectuera une ou des **transformation(s)** pour afficher un objet grâce aux fonctions `translate`, ...
- On répètera la deuxième étape jusqu'à que **tous les objets** du niveau soient **affichés**.

Toutes ces étapes se répéteront indéfiniment dans notre programme. En général, on affiche tous les objets d'un niveau 50 à 60 fois par seconde, imaginez le fourbi que l'ordinateur doit calculer. 🍪

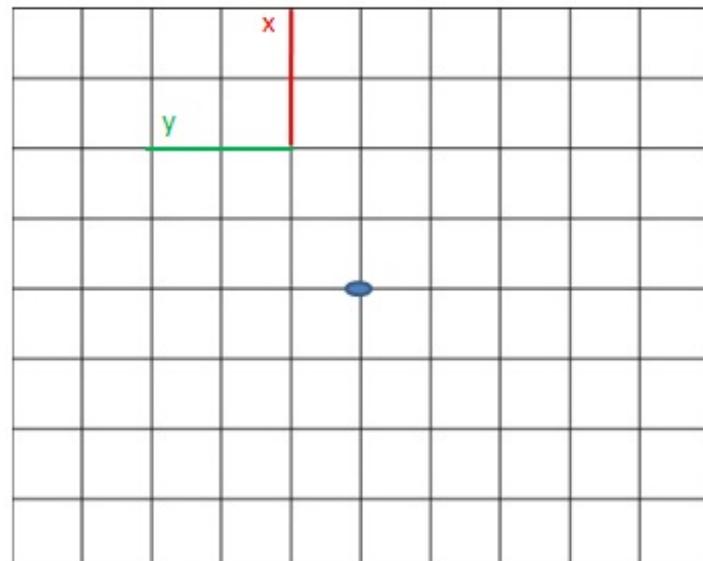
Accumulation de transformations

Attention, de même que l'ordre des matrices dans la multiplication, l'ordre des transformations a une importance capitale ! Si vous effectuez une translation puis une rotation, vous n'obtiendrez pas la même chose que si vous faisiez une rotation et une translation.

Comme d'habitude prenons un petit exemple. Dans un premier temps, je vais faire une translation du repère par rapport au vecteur $V(2, 1)$, puis une rotation de 90° sur l'axe Z (l'axe Z est en fait pointé vers nous, c'est pour ça que nous ne le voyons pas) :



Maintenant je vais l'inverse, une rotation de 90° puis une translation de vecteur $(2, 1)$:



Oh tiens ! Les repères ne sont pas les mêmes. Et oui, les repères sont différents car l'ordre des transformations est important. Faites donc bien attention à ce que vous voulez faire et à l'ordre dans lequel effectuer ces transformations. 😐

Enfin ce chapitre est terminé, il nous aura fallu du temps mais n'oubliez pas que tout ce que je vous enseigne est nécessaire pour comprendre et utiliser OpenGL. Récapitulons ce que nous savons faire :

- On sait afficher des **vertices** pour former des formes géométriques.
- On sait utiliser les **shaders** pour leur donner un peu de couleurs (même si l'on peut faire bien plus avec eux 😊).
- On sait utiliser des **matrices** pour effectuer des transformations.

Bonne nouvelle ! Nous connaissons tout ce qui est nécessaire pour ajouter une troisième dimension à nos programmes. Il est d'ailleurs temps mes amis, attaquons-nous à cette nouvelle dimension qui s'offre à nous ! 😎

La troisième dimension (Partie 1/2)

Le chapitre précédent était un peu compliqué. Mais grâce à lui, nous sommes maintenant capables d'utiliser les transformations pour façonner un monde 3D. Dans ce nouveau chapitre, nous allons mettre en commun ce que l'on sait sur OpenGL avec ce que l'on a appris concernant les **matrices**.

J'ai préféré diviser ce chapitre en deux parties, étant donné que nous allons aborder pas mal de nouvelles notions. Dans cette première partie, nous allons nous concentrer sur l'implémentation des matrices dans notre programme, puis dans la seconde partie, nous implémenterons la troisième dimension et surtout nous apprendrons à l'utiliser ! 

La matrice de projection

Présentation

Dans le chapitre précédent, nous avons vu ce qu'étaient les matrices ainsi que leur fonctionnement. Nous nous sommes surtout concentrés sur la matrice modelview qui permet de placer nos objets dans un monde 3D. Les transformations que nous avons vues (translation, rotation et homothétie) sont appliquées sur elle.

Dans ce chapitre-là maintenant, nous allons nous concentrer sur une autre matrice : la **matrice de projection**.

Sa seule utilité est de permettre l'affichage d'un monde 3D sur notre écran qui lui ne possède que 2 dimensions. Cependant, vous vous imaginez bien que ce processus n'est pas aussi simple que cela, on a quand même besoin de quelques paramètres pour permettre une telle opération.

Pour expliquer son fonctionnement, je vais prendre une bonne vieille fonction connue des habitués d'OpenGL : la fonction **gluPerspective()**.



Euh ouai, mais moi je viens ici pour apprendre OpenGL, je ne connais pas cette fonction. 

Ne vous inquiétez pas, je vais vous expliquer le fonctionnement de cette fonction en détails, et j'ai plutôt intérêt à le faire étant donné que nous allons la recoder entièrement. 

Comme je vous l'ai dit dans les chapitres précédents, une bonne partie des anciennes fonctions d'OpenGL a disparu avec la nouvelle version. La fonction **gluPerspective** a elle aussi disparu (indirectement). Il nous faudra donc la coder de nos propres mains. Et pour vous faciliter la tâche, je vais vous expliquer en détails son fonctionnement.

Premièrement, voici le prototype de la fonction :

Code : C++

```
void gluPerspective(double angle, double ratio, double near, double far);
```

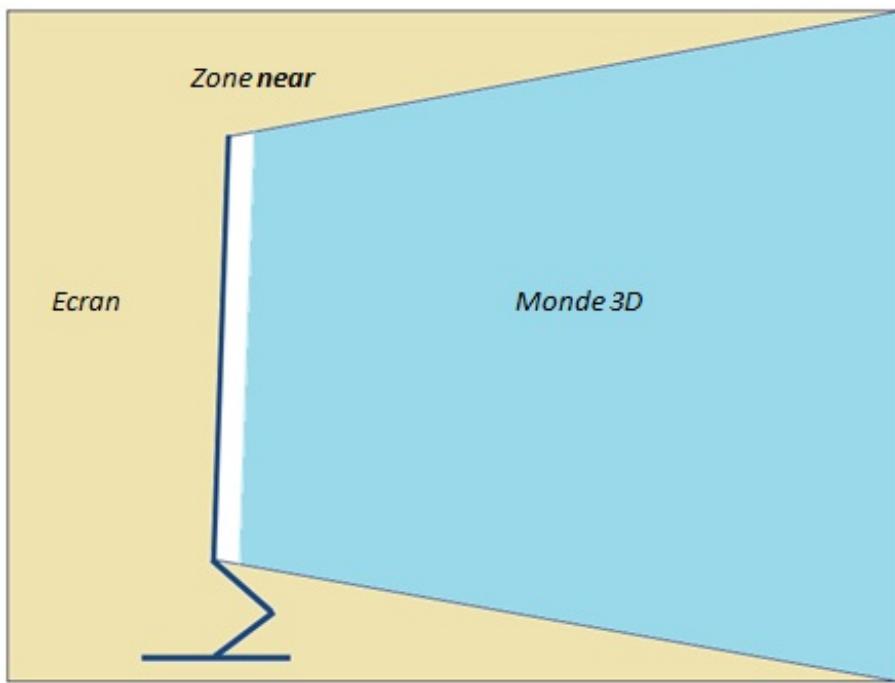


J'ai volontairement modifié le nom des paramètres pour une meilleure compréhension, cependant je garderai le nom original des fonctions recodées avec le préfixe (gl, glu, ...) en moins. 

Hum tout plein de paramètres intéressants  . Je vais commencer par les deux derniers paramètres : **near** et **far**.

Near

Le paramètre **Near** correspond à la distance entre votre écran et ce qui sera affiché.



La petite zone entre l'écran et la plaque est une zone où rien ne sera affiché. Même s'il y a un objet dans cet intervalle il ne sera pas affiché.

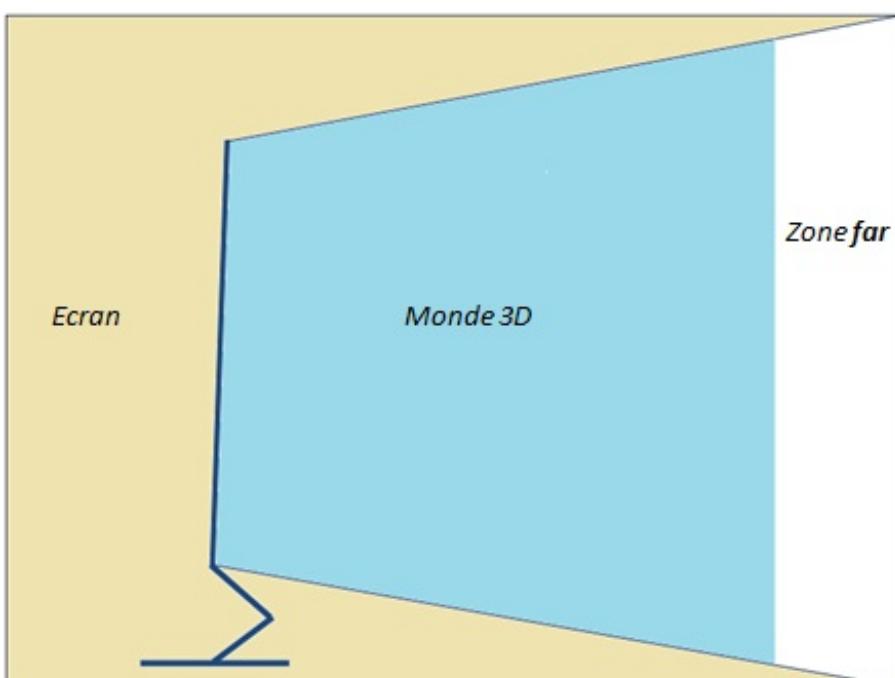
Pourquoi me direz-vous ? Simplement parce que les mathématiques nous l'imposent, il nous faut obligatoirement une zone non-affichable pour que le calcul de la projection puisse s'effectuer. Il se passe la même chose avec la division par zéro, c'est impossible. Il faut obligatoirement que le dénominateur soit au moins un peu plus grand (ou plus petit mais on s'en moque) que zéro.

Le paramètre **near** représente donc ce petit intervalle nécessaire au calcul de la projection.

Far

Le paramètre **far** est un peu plus simple, on peut dire que c'est l'inverse de **near**.

C'est également une distance entre votre écran et ce qui sera affiché. La différence est que tout objet se trouvant au delà de cette distance ne sera pas affiché.



Au final, pour qu'un objet puisse s'afficher sur l'écran, il faut qu'il se situe entre les zones **near** et **far**, sinon il ne sera pas affiché.

Ratio

Je pense que vous avez tous entendu parler des télévisions 4/3 (quatre tiers) et 16/9 (seize neuvièmes). Le **ratio** est le rapport entre la longueur de la télé et sa hauteur. Dans la plupart des cas nous avons soit un ratio de 4/3 soit un ratio de 16/9, voire 16/10 pour les jeux-vidéo.

Pour nous, le ratio s'appliquera à la fenêtre SDL que l'on a codée. Dans le chapitre 3, nous avions créé une fenêtre de 800 pixels par 600 pixels. Le ratio sera donc de 800/600, et si on simplifie la fraction on a un ratio de 4/3.

Dans les jeux-vidéo, on propose généralement plusieurs modes d'affichage afin de s'adapter à l'écran du joueur. En effet, tous les joueurs n'ont pas forcément le même écran. Nous donnerons au final comme paramètre la division entre la longueur de la fenêtre SDL et sa hauteur.

Angle

Ce paramètre est le plus spécial de la fonction. C'est en fait l'**angle de vue** avec lequel nous allons voir la scène. Plus cet angle sera petit, plus on aura l'impression de faire un effet de zoom sur la scène. Et à l'inverse, plus il sera grand, plus on aura l'impression que la scène s'éloigne et se déforme.

Voici trois exemples de la même scène avec **trois angles de vue** différents :





Merci à Kayl pour sa scène de test. 😊

Dans la première image, on utilise un angle de **70°**, dans la deuxième un angle de **30°** et dans la troisième un angle de **100°**.

L'avantage d'un angle plus petit est que l'on a l'impression de faire un zoom de jumelle ou de sniper 😎. A l'inverse, un angle plus grand créera une ambiance plus particulière, mais ce genre d'angle s'utilise rarement.

En général, l'angle de vision "**normal**" est de **70°** (70 degrés) et c'est d'ailleurs ce que l'on mettra dans notre code.

Utilisation de la librairie GLM

Dans les précédentes versions d'OpenGL, il existait une multitude de fonctionnalités mathématiques qui étaient gérées par OpenGL même, les matrices en faisait évidemment partie. Cependant, ces fonctionnalités sont maintenant supprimées et il faut trouver un autre moyen de gérer nos matrices nous-même.

Heureusement pour nous, il existe une librairie qui s'appelle **GLM** (pour **OpenGL Mathematics**). Vous l'avez déjà téléchargée

normalement donc vous n'avez rien à faire. Cette librairie nous permet d'utiliser les matrices, et bien plus encore, sans avoir à nous soucier de tout programmer à la main. Elle inclut même certaines fonctions dépréciées comme `gluPerspective()`. 😊

Nous allons donc utiliser **GLM** pour simuler toutes les fonctions dont nous aurons besoin.

Utilisation de la projection

Inclusion des matrices

Toutes ces belles matrices (que vous adorez j'en suis sûr 😊) ne nous servent pas à grand chose pour l'instant. Nous allons maintenant les intégrer dans notre code OpenGL pour enfin voir ce qu'elles ont véritablement dans le ventre.

Reprendons le code de la **boucle principale** que nous avons laissé au chapitre 4 :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    bool terminer(false);
    float vertices[] = {-0.5, -0.5, 0.0, 0.5, 0.5, -0.5};
    float couleurs[] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
    1.0};

    // Shader

    Shader shaderCouleur("Shaders/couleur2D.vert",
    "Shaders/couleur2D.frag");
    shaderCouleur.charger();

    // Boucle principale

    while (!terminer)
    {
        /* ***** Code d'affichage ***** */
    }
}
```

Je n'ai pas mis tout le code, seule cette partie nous intéresse pour le moment.

Les en-têtes de GLM

Vous savez maintenant que l'on a besoin de deux matrices pour faire fonctionner un jeu 3D : la matrice de **projection** et la **modelview**. Nous allons donc les déclarer dans notre code grâce à librairie **GLM**.

Pour commencer, veuillez inclure les en-têtes suivants dans le fichier **SceneOpenGL.h** qui permettent à notre programme d'utiliser les matrices :

Code : C++

```
// Includes OpenGL
...
// Includes GLM
#include <glm/glm.hpp>
```

```
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Autres includes

#include <SDL2/SDL.h>
#include <iostream>
#include <string>
#include "Shader.h"
```

La première inclusion permet d'utiliser les fonctionnalités principales de **GLM**, la deuxième les transformations et la dernière permet de récupérer les valeurs des matrices (nous verrons cela dans un instant).



Elle est bizarre ton extension c'est marqué .hpp c'est normal ?

Oui tout à fait. 😊

La particularité de **GLM** c'est qu'elle est codée directement dans les headers, il n'y a pas de librairie pré-compilée comme la **SDL** par exemple. Grâce à ce système, **GLM** est compatible avec toutes les plateformes et il n'y a pas besoin de fournir de fichier spécifique pour Linux, Windows, etc. Ni même pour compilateur (.a, .dll, etc.)

Inclusion des matrices

Après toute cette théorie nous pouvons enfin coder nos premières matrices. L'objectif est de créer deux matrices du nom de **projection** et **modelview**.

Pour cela, nous allons déclarer deux objets de type **mat4**, pour *matrice carrée d'ordre 4*. Il faut utiliser le **namespace glm** en même temps que la déclaration :

Code : C++

```
// Matrices projection et modelview

glm::mat4 projection;
glm::mat4 modelview;
```

Tous les objets et méthodes de **GLM** doivent utiliser le **namespace glm**. Pour éviter d'avoir à le taper à chaque fois, ajoutez la ligne de code suivante dans chaque classe où vous utiliser les matrices :

Code : C++

```
// Permet d'éviter la ré-écriture du namespace glm::

using namespace glm;
```

Ajoutez cette ligne dans le fichier **.cpp** et non **.h** faites attention.



Avec cette ligne de code, la déclaration des matrices devient :

Code : C++

```
// Matrices projection et modelview  
  
mat4 projection;  
mat4 modelview;
```

Maintenant que les matrices sont déclarées, nous allons pouvoir les initialiser.

La matrice projection

Occupons-nous d'abord de la projection. Dans la partie précédente, nous avons parlé d'une ancienne fonction du nom de **gluPerspective()**. Celle-ci a disparu avec OpenGL 3 mais **GLM** l'a gentiment recoder pour nous sous la forme d'une méthode, voici son prototype :

Code : C++

```
mat4 perspective(double angle, double ratio, double near, double  
far);
```

On remarque qu'elle contient exactement les mêmes paramètres, pratique n'est-ce pas ? La seule différence est qu'elle renvoie un objet de type **mat4**, cet objet doit être affecté à la matrice **projection**.

Nous appellerons cette méthode avec les paramètres suivants :

- Un **angle** de **70°** (qui correspond à un angle de vision normal).
- Un **ratio** en fonction de la taille de la fenêtre, ici un **ratio de m_largeurFenetre / m_hauteurFenetre**.
- Le paramètre **near** avec une valeur de 1.
- Le paramètre **far** avec une valeur de 100.

Voici son appel :

Code : C++

```
// Matrices  
  
mat4 projection;  
mat4 modelview;  
  
// Initialisation  
  
projection = perspective(70.0, (double) m_largeurFenetre /  
m_hauteurFenetre, 1.0, 100.0);
```



N'oubliez pas de faire un cast pour le **ratio**, sinon vous n'aurez que la partie entière de la division.

La matrice modelview

Pour la matrice **modelview** c'est un peu plus simple. Pour l'initialiser, nous allons leur donner les valeurs d'une matrice d'identité

(avec les **1** en diagonale). Elle ne doit pas avoir que des valeurs nulles car si nous multiplions des vecteurs par des **0**, il risquerait d'y avoir un gros problème avec notre scène finale. 😊

Pour cela, nous allons utiliser un constructeur qui ne demande qu'une seule valeur : la valeur **1.0**.

Code : C++

```
// Matrices

mat4 projection;
mat4 modelview;

// Initialisation

projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
modelview = mat4(1.0);
```

Le constructeur sait tout seul qu'il doit utiliser une matrice d'identité, cela éviter d'avoir à écrire toutes les valeurs à la main pour mettre des **1.0** en diagonal. 😊

La boucle principale

Vous vous souvenez qu'au début du chapitre je vous ai parlé du comportement des matrices avec la boucle principale du programme ? Je vous avais dit qu'à chaque tour de boucle, il fallait réinitialiser la matrice **modelview** pour ne pas avoir les anciennes valeurs du tour précédent.

Cette réinitialisation se fait exactement de la même façon que l'initialisation. C'est-à-dire que l'on affecte le résultat du constructeur **mat4(1.0)** à la matrice **modelview**. Nous devons faire ceci juste après le nettoyage de l'écran :

Code : C++

```
while (!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&m_evenements);

    if (m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT);

    // Réinitialisation de la matrice modelview

    modelview = mat4(1.0);
}
```

Interaction entre le programme et les matrices

Notre code OpenGL devient de plus en plus beau 😊. Mais les matrices ne nous servent toujours pas à grand chose elles

n'interagissent toujours pas avec notre programme.

Je ne vais pas m'étaler sur le sujet pour le moment, mais sachez que l'interaction entre les matrices et le programme se fait dans le **shader**. C'est lui qui va faire tous les calculs pour projeter le monde 3D sur notre écran.

Pour le moment, le shader que nous avons chargé (**couleur2D**) est incapable de gérer la projection, il faut en charger un autre. Si vous n'avez rien modifié dans le dossier "**Shaders**" du chapitre 4, vous devriez trouver des codes sources qui s'appellent **couleur3D.vert** et **couleur3D.frag** (à ne pas confondre avec **couleur2D**). Si vous ne trouvez pas ces fichiers, re-téléchargez le dossier complet depuis le chapitre 4.

À la différence du premier shader, **couleur3D** est, lui, capable de gérer la projection. Nous allons donc le charger à la place de son prédecesseur :

Code : C++

```
// Shader gérant la couleur et les matrices  
  
Shader shaderCouleur("Shaders/couleur3D.vert",  
"Shaders/couleur3D.frag");  
shaderCouleur.charger();
```

Ne vous inquiétez pas si le *fragment shader* ne change pas, c'est normal. 😊

Le shader n'est pas le seul à devoir changer, les vertices doivent elles aussi subir une modification. Il faut leur ajouter une troisième dimension pour les rendre compatibles avec la projection.



Ça veut dire que nos triangles seront en 3D ?

Oui et non ... En fait, ils sont en 3D mais pour le moment nous sommes mal placés pour les voir de cette façon. Nous verrons dans la deuxième partie de ce chapitre comment "bien se placer". Ne vous inquiétez pas ça arrivera très vite. 😊

Bref au final, il suffit d'ajouter la coordonnée **Z** à tous les vertices pour qu'ils soient en 3D. Nous mettrons la valeur **-1** pour le moment :

Code : C++

```
float vertices[] = {-0.5, -0.5, -1.0, 0.0, 0.5, -1.0, 0.5, -0.5,  
-1.0};
```

Vu que nous ajoutons une coordonnée à nos vertices, il va falloir modifier un paramètre dans la fonction **glVertexAttribPointer**. Le paramètre **size** permet de spécifier la taille d'un vertex, donc son nombre de coordonnées. On le passe désormais à **3** car nous avons **3 coordonnées** :

Code : C++

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);
```

Revenons à notre shader, vu que c'est lui qui gère les calculs de projection, il va falloir lui envoyer les matrices que nous avons déclarées pour qu'il puisse travailler correctement. L'envoi de matrice au shader se fait avec cette fonction :

Code : C++

```
glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value);
```

- **location** : permet de savoir où envoyer les matrices à l'intérieur même du shader.
- **count** : permet de savoir combien de matrice on envoie. On n'enverra qu'une seule matrice à la fois.
- **transpose** : booléen qui permet d'inverser ou non la matrice qu'on envoie. Dans notre cas, on lui donnera toujours la valeur **GL_FALSE**.
- **value** : est un pointeur sur le tableau de valeurs de la matrice. Nous utiliserons la méthode **value_ptr()** de la librairie **GLM**.

Je vous expliquerai en détails le fonctionnement de cette fonction dans le chapitre sur les shaders. Sachez juste qu'elle nous permet d'envoyer nos matrices au shader.

Nous lui enverrons ces valeurs :

- **location** : `glGetUniformLocation(shaderCouleur.getProgramID(), "Le_nom_de_la_matrice")`. Oui c'est bien une fonction (on la verra également dans le chapitre sur les shaders). Le paramètre en rouge est une chaîne de caractères, nous lui donnerons la valeur "**modelview**" et "**projection**".
- **count** : 1 (le chiffre 1) pour une matrice.
- **transpose** : **GL_FALSE**.
- **value** : Nous lui donnerons le résultat de la méthode **value_ptr()** de nos objets **mat4**.

Grâce à cette fonction, notre shader va pouvoir travailler avec les matrices que nous avons déclarées dans le main.

Nous appellerons cette fonction deux fois vu que nous avons deux matrices à envoyer. Ces appels se feront juste avant une fonction que vous connaissez bien : **glDrawArrays()**.

Dans notre cas, nous ferons ces appels toujours de la même façon :

Code : C++

```
// On envoie les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
    "modelview"), 1, GL_FALSE, value_ptr(modelview));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
    "projection"), 1, GL_FALSE, value_ptr(projection));
```

Si on les intègre au code d'affichage ça donne ceci :

Code : C++

```
// On spécifie quel shader utiliser

glUseProgram(shaderCouleur.getProgramID());

// Envoi des vertices et des couleurs

....

// On envoie les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
    "modelview"), 1, GL_FALSE, value_ptr(modelview));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
    "projection"), 1, GL_FALSE, value_ptr(projection));
```

```
// On affiche le polygone  
glDrawArrays(GL_TRIANGLES, 0, 3);  
  
// Désactivation des Vertex Attrib  
....  
  
// On n'utilise plus le shader  
glUseProgram(0);
```



Il faut envoyer les matrices au shader **APRÈS** avoir fait toutes les transformations ! Si vous les envoyez et que vous faites des transformations après, alors elles ne seront pas prises en compte.

De même, faites attention à les envoyer **APRÈS** la fonction `glUseProgram()`, sinon vous les enverrez dans le vide. 😊

Je pense vous avoir parlé de tout, compilons tout ça pour voir que ça donne :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Variables  
  
    bool terminer(false);  
    float vertices[] = {-0.5, -0.5, -1.0, 0.0, 0.5, -1.0, 0.5,  
-0.5, -1.0};  
    float couleurs[] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,  
1.0};  
  
    // Shader  
  
    Shader shaderCouleur("Shaders/couleur3D.vert",  
"Shaders/couleur3D.frag");  
    shaderCouleur.charger();  
  
    // Matrices  
  
    mat4 projection;  
    mat4 modelview;  
  
    projection = perspective(70.0, (double) m_largeurFenetre /  
m_hauteurFenetre, 1.0, 100.0);  
    modelview = mat4(1.0);  
  
    // Boucle principale  
  
    while(!terminer)  
    {  
        // Gestion des évènements  
  
        SDL_WaitEvent(&m_evenements);  
  
        if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)  
            terminer = true;  
  
        // Nettoyage de l'écran
```

```
glClear(GL_COLOR_BUFFER_BIT);

// Réinitialisation de la matrice modelview
modelview = mat4(1.0);

// On spécifie quel shader utiliser
glUseProgram(shaderCouleur.getProgramID());

// On remplit puis on active le tableau Vertex Attrib 0
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
vertices);
glEnableVertexAttribArray(0);

// Même chose avec le tableau Vertex Attrib 1
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
couleurs);
glEnableVertexAttribArray(1);

// On envoie les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));

// On affiche le polygone
glDrawArrays(GL_TRIANGLES, 0, 3);

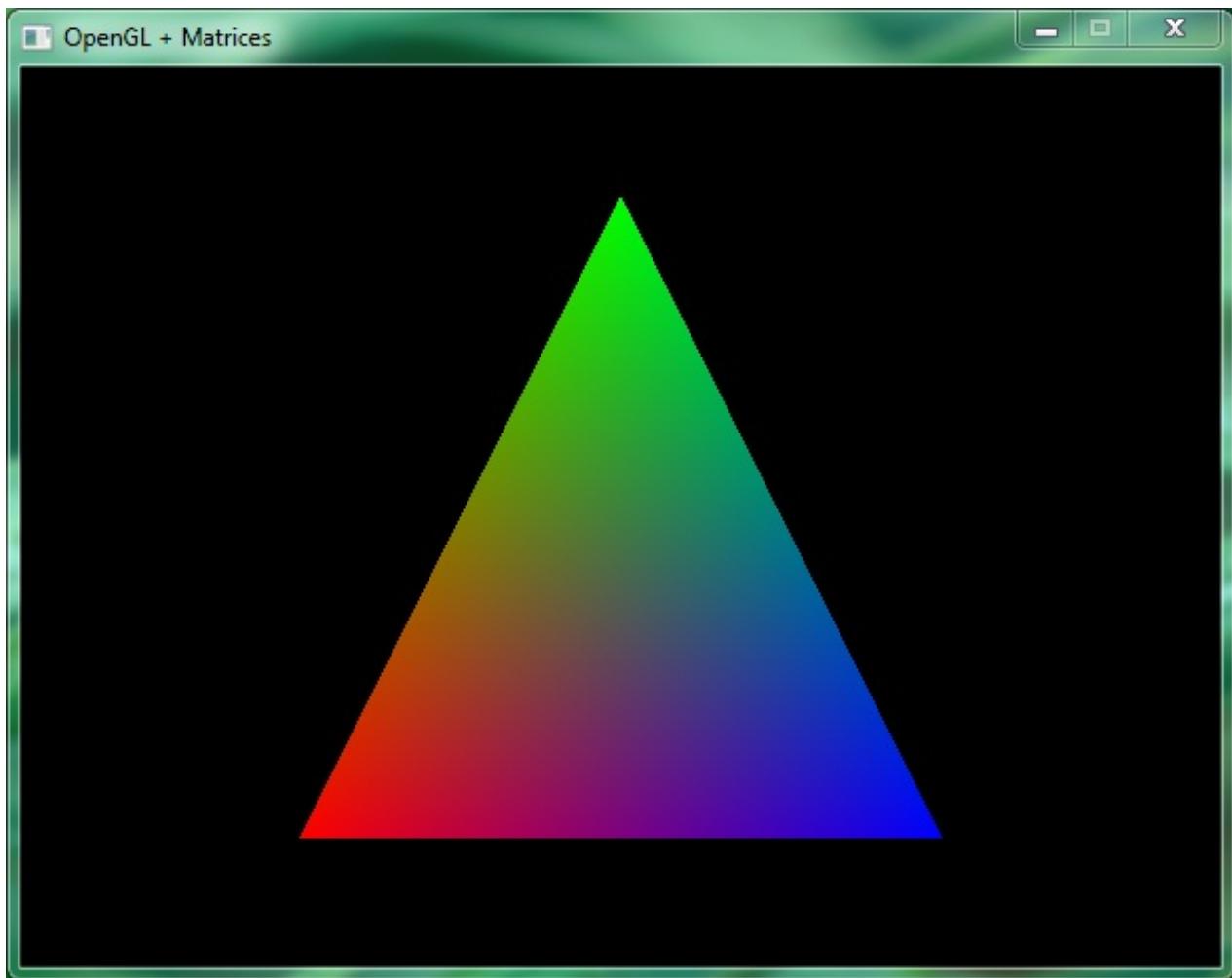
// On désactive les tableaux Vertex Attrib puisque l'on
n'en a plus besoin

glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);

// On n'utilise plus le shader
glUseProgram(0);

// Actualisation de la fenêtre
SDL_GL_SwapWindow(m_fenetre);
}
```

Vous devriez avoir cette fenêtre :



C'est tout ? C'est pratiquement la même chose qu'avant. 😕

À première vue oui, il ne se passe rien de plus sauf peut-être une légère déformation de notre triangle. Mais sachez que dans votre carte graphique il se passe pas mal de choses. De plus, nous pouvons maintenant utiliser les transformations dans nos programmes et ce sont justement elles qui forment la base d'un jeu-vidéo. Sans transformations, nous serions encore en train de jouer au Pong. 😊

Exemples de transformations

Introduction

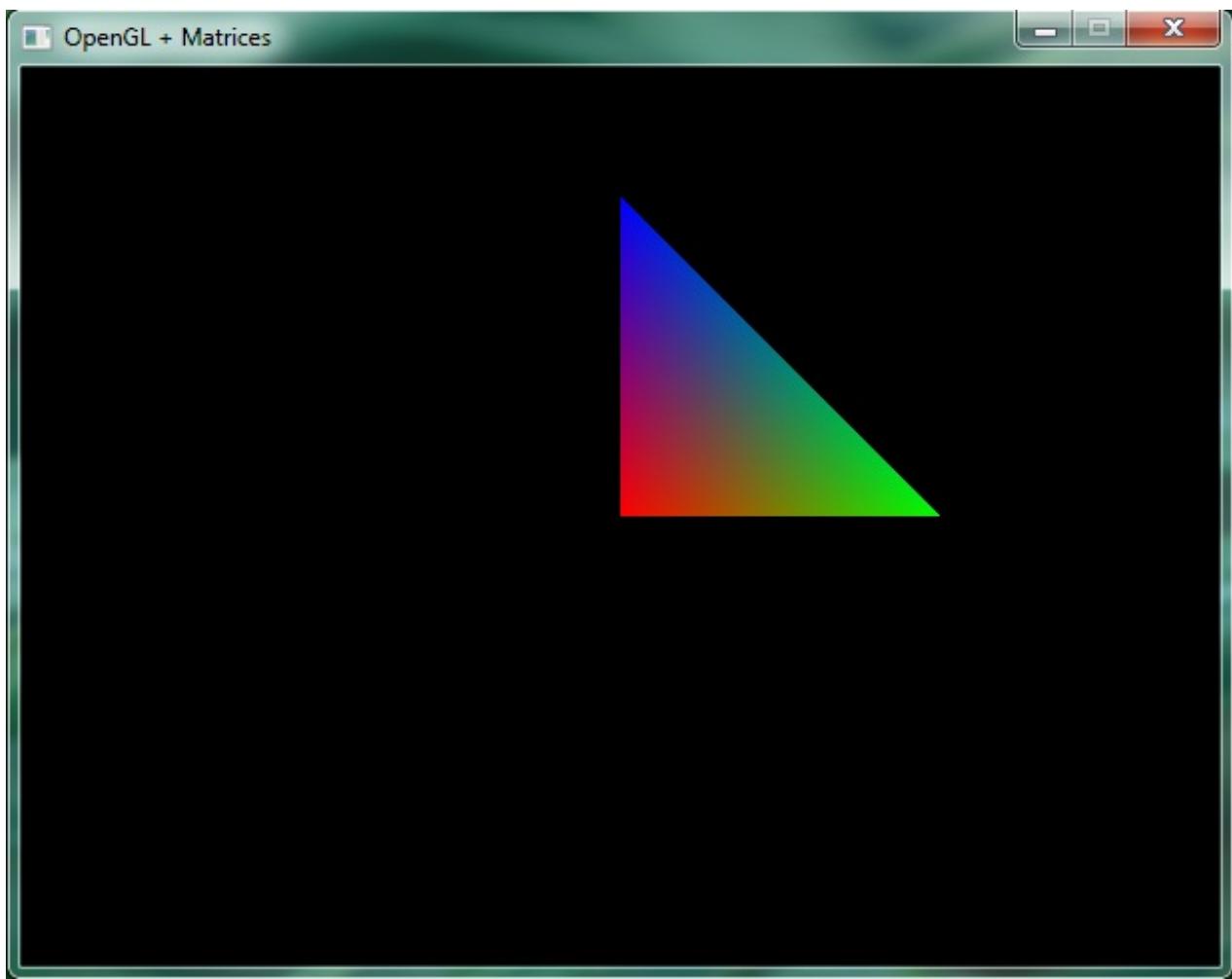
On va se reposer un peu dans cette partie, il n'y a rien de nouveau, on utilisera juste ce que l'on a déjà codé.

Tout d'abord, on va changer un peu nos vertices pour avoir un triangle plus petit. Sinon nous ne pourrions pas vraiment voir ce que donnent les transformations :

Code : C++

```
float vertices[] = {0.0, 0.0, -1.0, 0.5, 0.0, -1.0, 0.0, 0.5, -1.0};
```

D'origine, voici la position du triangle :



Les transformations

La translation

Prenons un vecteur V de coordonnées (0.4, 0.0, 0.0) pour effectuer une translation du repère (matrice **modelview**) sur 0.4 unité sur l'axe X. Normalement, le triangle devrait se retrouver sur la gauche.

Pour faire une translation avec **GLM**, nous allons utiliser la méthode **translate()** dont voici le prototype :

Code : C++

```
mat4 translate(mat4 matrice, vec3 translation);
```

- **mat4** : matrice qui sera multipliée par la matrice de translation. Il s'agit ici de **modelview**
- **translation** : objet vecteur à 3 coordonnées. Il correspond ici un point dans l'espace avec 3 coordonnées (x, y et z)

La méthode renvoie la matrice donnée en paramètre avec la translation ajoutée.

Pour contenter l'objet **vec3**, nous allons juste appeler le constructeur **vec3** avec les coordonnées de la translation que l'on veut faire, ici (0.4, 0.0, 0.0).

L'appel à la méthode ressemble au final à ceci :

Code : C++

```
// Translation  
modelview = translate(modelview, vec3(0.4, 0.0, 0.0));
```

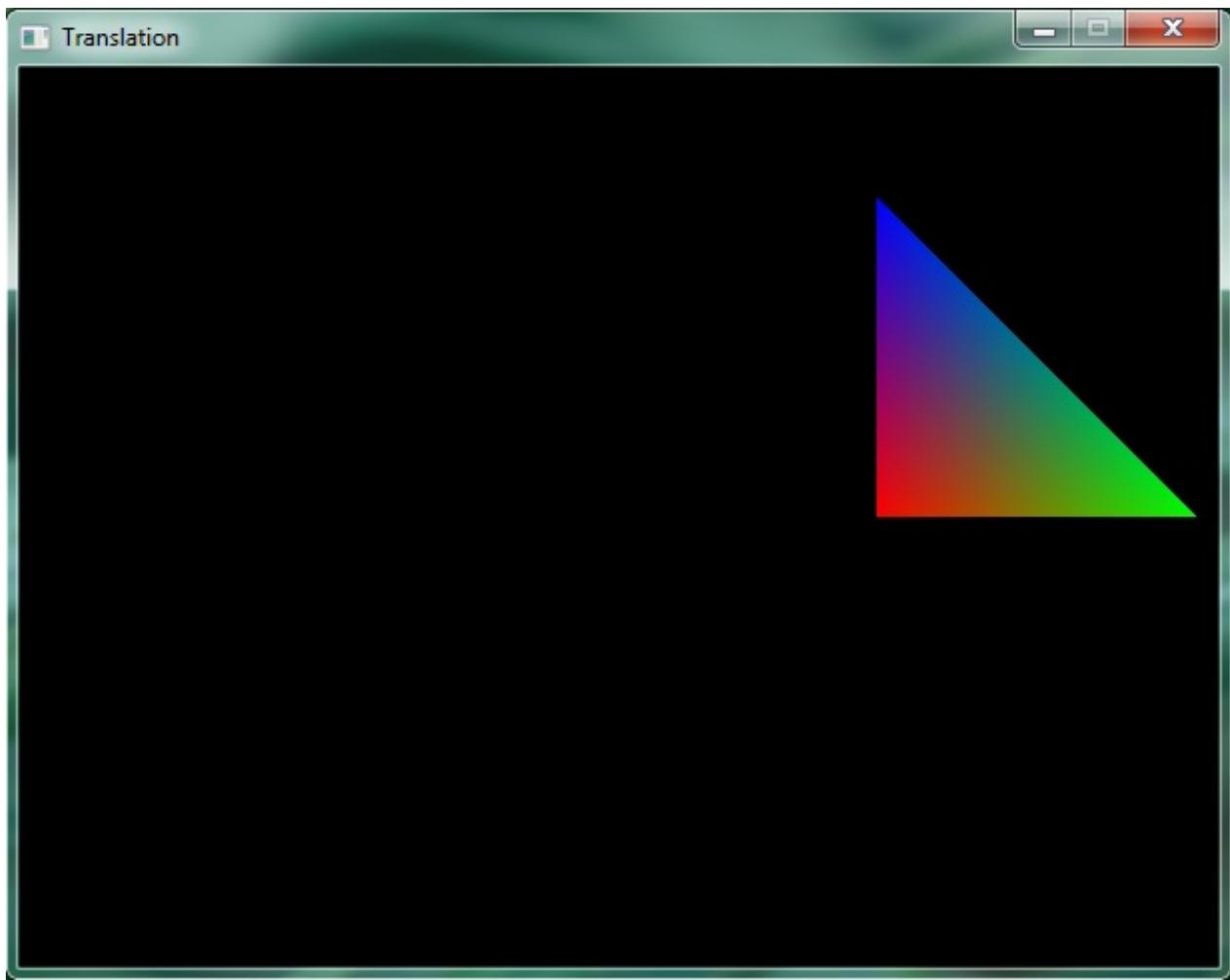
On identifie bien ce que fait cette ligne de code. On appelle la méthode **translate()** qui va modifier la matrice **modelview** à l'aide du **vecteur** de coordonnées (0.4, 0.0, 0.0).

Nous devons inclure cette ligne juste avant l'envoi des matrices au shader :

Code : C++

```
// On spécifie quel shader utiliser  
glUseProgram(shaderCouleur.getProgramID());  
  
// Remplissage des tableaux Vertex Attrib 0 et 1  
....  
  
// Translation  
modelview = translate(modelview, vec3(0.4, 0.0, 0.0));  
  
// On envoie les matrices au shader  
  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
  
// Affichage  
....  
  
// On n'utilise plus le shader  
glUseProgram(0);
```

Ce qui nous donne :



Le triangle ne connaît aucun changement, nous n'avons pas modifié ses **coordonnées**. En revanche, le repère lui a changé. On l'a translaté selon le vecteur V, le triangle se trouve donc un peu plus loin.

Bon, sur une si petite forme on ne voit pas trop l'intérêt, mais sur un personnage de plus de 1000 véctices on voit la différence croyez-moi. 😊

La rotation

Nous allons maintenant faire pivoter le triangle d'un angle de 60° selon l'axe Z. L'axe Z est pointé vers nous mais nous ne le voyons pas, nous verrons cela dans la deuxième partie du chapitre.

Nous utiliserons pour cela la méthode **GLM rotate()** :

Code : C++

```
mat4 rotate(mat4 matrice, double angle, vec3 axis);
```

Les paramètres sont sensiblement les mêmes :

- **matrice** : matrice qui sera multipliée par la rotation. Il s'agit ici de modelview
- **angle** : angle de la rotation
- **axis** : vecteur représentant l'axe de la rotation

Pour effectuer une rotation de 60° sur l'axe Z, il suffit donc d'appeler la méthode **rotate()** de cette façon :

Code : C++

```
// Rotation  
modelview = rotate(modelview, 60.0f, vec3(0.0, 0.0, 1.0));
```

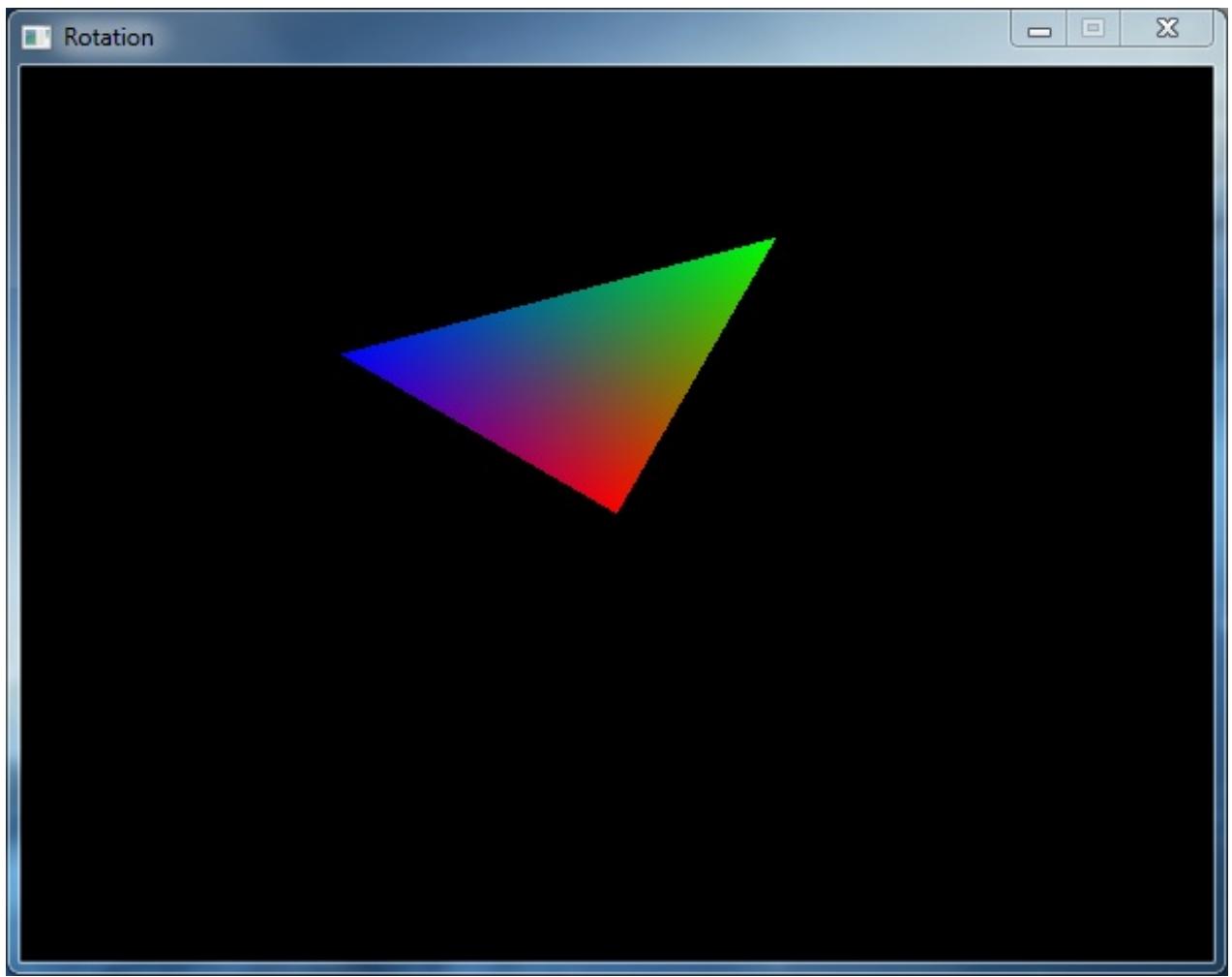


Attention, cette méthode est capricieuse et demande une variable **float**. Il faut donc ajouter le suffixe **f** à l'angle dans tous les cas.

On inclut cette ligne juste avant d'envoyer les matrices au shader :

Code : C++

```
// On spécifie quel shader utiliser  
glUseProgram(shaderCouleur.getProgramID());  
  
// Remplissage des tableaux Vertex Attrib 0 et 1  
....  
  
// Rotation  
modelview = rotate(modelview, 60.0f, vec3(0.0, 0.0, 1.0));  
  
// On envoie les matrices au shader  
  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
  
// Affichage  
....  
  
// On n'utilise plus le shader  
glUseProgram(0);
```



Hop, le triangle a pivoté de 60° (dans le sens trigonométrique). Encore une fois, on modifie le repère, on ne touche pas aux coordonnées du triangle.

L'homothétie

Allez encore une petit exemple. Nous allons faire une homothétie en multipliant par 2 les unités de mesure du repère. 1 unité sera plus longue, donc le triangle paraîtra plus gros.

On appellera pour cela la méthode **scale()** :

Code : C++

```
mat4 scale(mat4 matrice, vec3 factors);
```

- **matrice** : matrice qui sera multipliée par la rotation. Il s'agit ici de **modelview**
- **factors** : vecteur contenant les 3 facteurs de redimensionnement pour les X, Y et Z

On appellera cette méthode ainsi pour agrandir le repère de 2 unités sur les axes X et Y. L'axe Z n'est pas encore visible pour nous, on le laisse donc comme ça.

Code : C++

```
// Homothétie  
modelview = scale(modelview, vec3(2, 2, 1));
```

Et comme d'habitude, on l'appelle juste avant d'envoyer les matrices au shader :

Code : C++

```
// On spécifie quel shader utiliser
glUseProgram(shaderCouleur.getProgramID());

// Remplissage des tableaux Vertex Attrib 0 et 1
.....

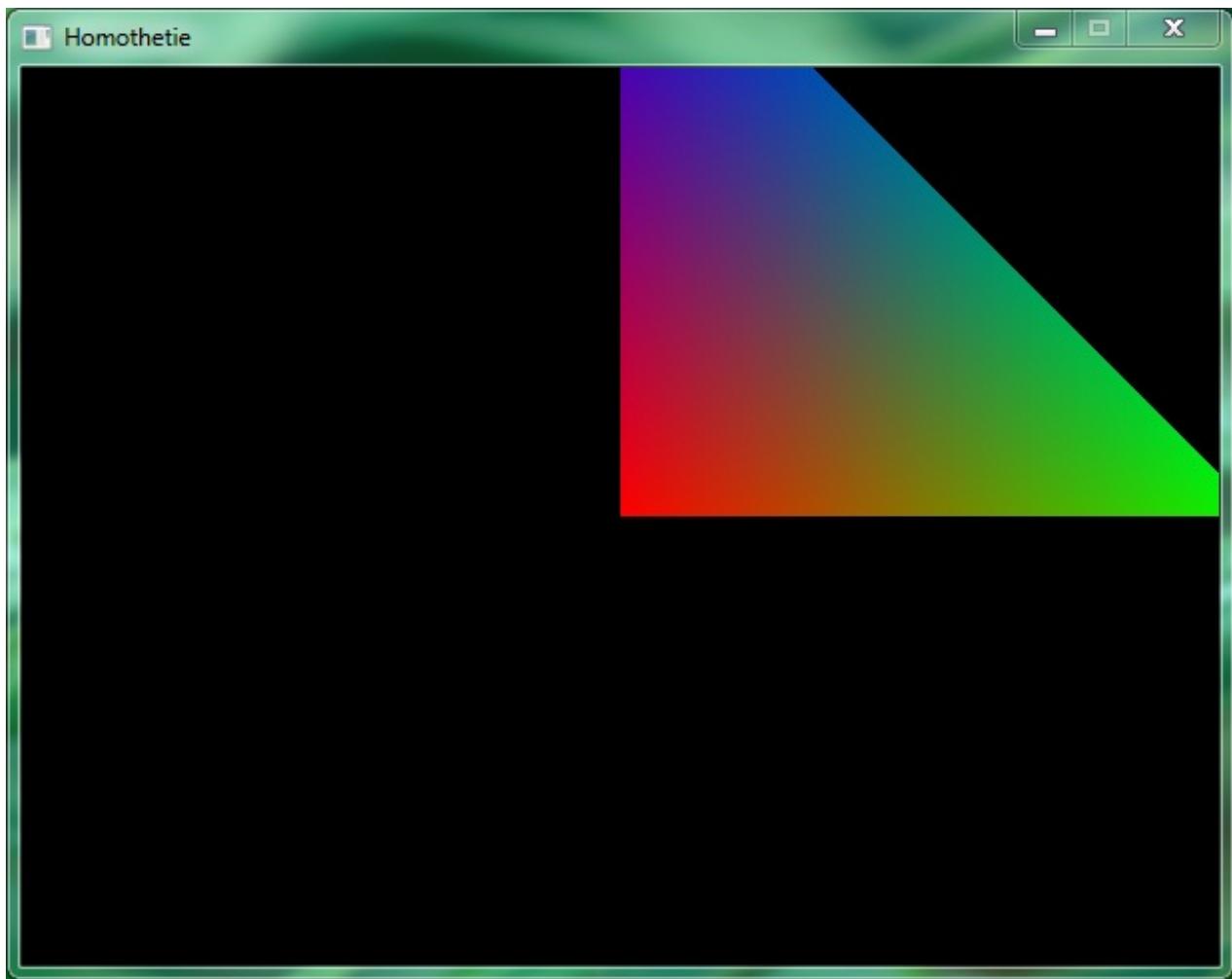
// Homothétie
modelview = scale(modelview, vec3(2, 2, 1));

// On envoie les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));

// Affichage
.....

// On n'utilise plus le shader
glUseProgram(0);
```



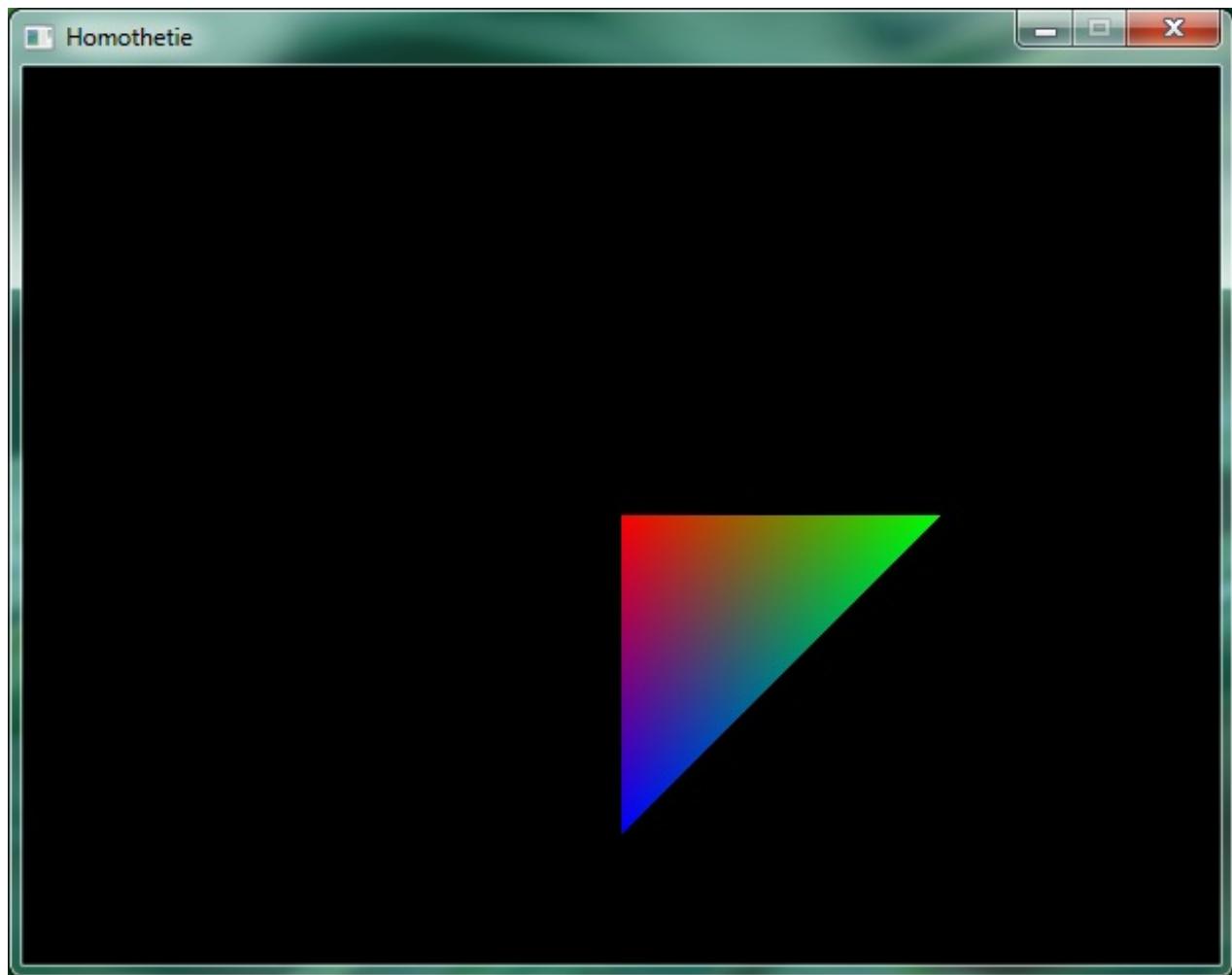
Le triangle est tellement gros qu'il sort de la fenêtre 😊. Bref vous avez compris le principe.

Je vais même vous montrer un truc sympa avec les homothéties, on va inverser le triangle. C'est une technique utilisée dans les jeux-vidéo pour les effets de reflets (comme un effet de miroir, d'eau, ...):

Code : C++

```
// Inversion du repère  
modelview = scale(modelview, vec3(1, -1, 1));
```

Les axes X et Z ne sont pas modifiés, ils sont multipliés par 1. Mais l'axe Y lui est inversé, voici ce que ça donne :



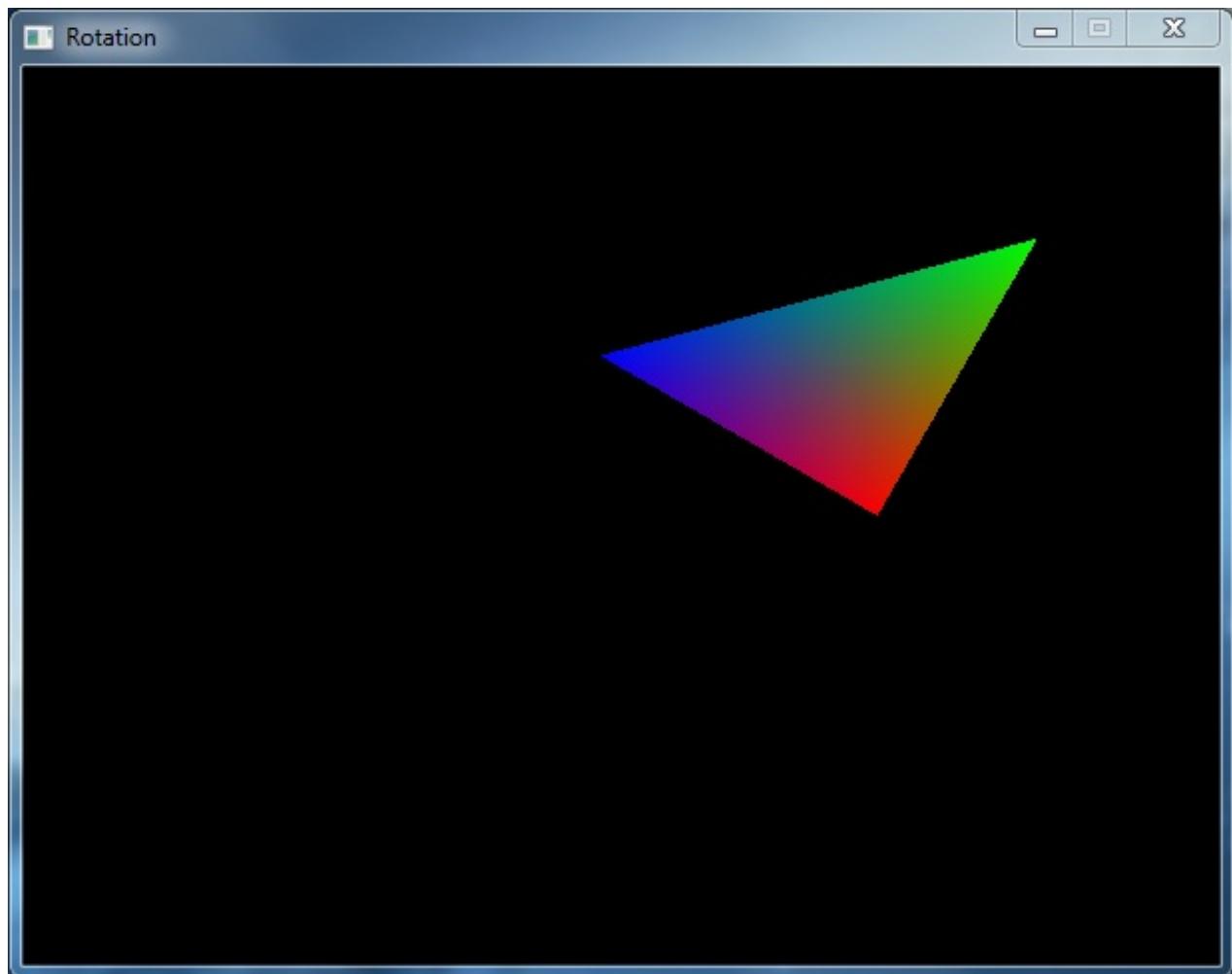
Ordre des transformations

Je vous ai dit au tout début que l'ordre des transformations était important. Vous pouvez désormais le constater par vous-même. Tout d'abord, on va translater notre triangle selon le vecteur V de coordonnées (0.4, 0.0, 0.0) puis on va le faire pivoter d'un angle de 60° sur l'axe Z :

Code : C++

```
// Translation puis rotation  
  
modelview = translate(modelview, vec3(0.4, 0, 0));  
modelview = rotate(modelview, 60.0f, vec3(0, 0, 1));
```

Voici le rendu :



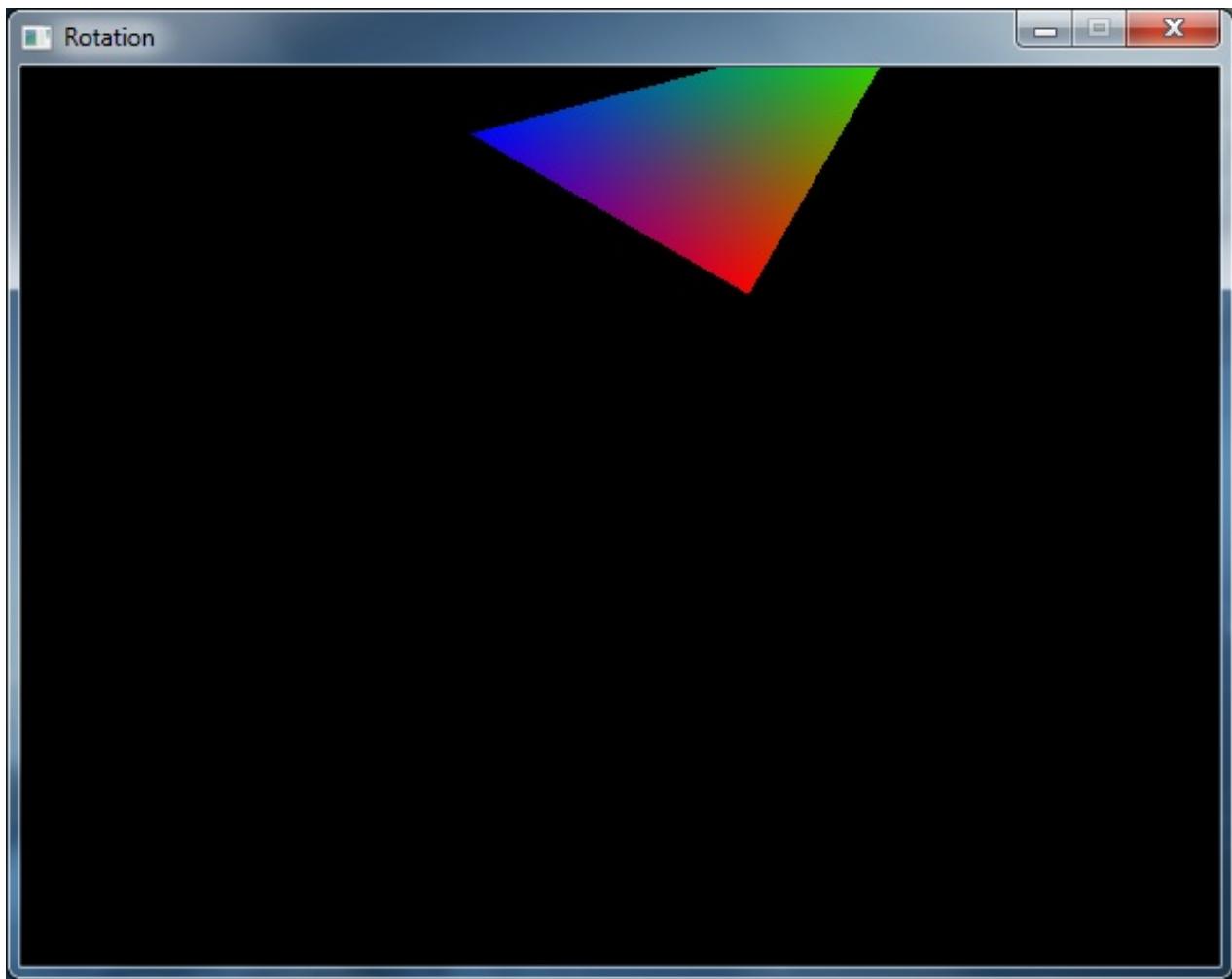
Maintenant, on fait l'inverse :

Code : C++

```
// Rotation puis translation

modelview = rotate(modelview, 60.0f, vec3(0, 0, 1));
modelview = translate(modelview, vec3(0.4, 0, 0));
```

Rendu :



On n'obtient pas la même chose dans les deux exemples. Faites attention à l'ordre des transformations c'est important. 😊

Multi-affichage

Courage on aborde le dernier point 😊. Dans le deuxième chapitre nous avons vu comment afficher plusieurs triangles sans les transformations. Maintenant que nous les avons, on va voir comment se faciliter la vie.

Si vous affichez plusieurs fois la même chose, pas besoin de remplacer les valeurs du tableau car les valeurs sont toutes les mêmes 😊. On ne fait qu'appliquer les transformations sur le repère puis on ré-affiche le triangle. Bien sûr, si on change les valeurs de la matrice modelview alors il faut la ré-envoyer au shader :

Code : C++

```
// On spécifie quel shader utiliser
glUseProgram(shaderCouleur.getProgramID());

// On translate le premier triangle
modelview = translate(modelview, vec3(0.4, 0, 0));

// On envoie les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));
```

```
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));

/* ***** Affichage du premier triangle **** */
glDrawArrays(GL_TRIANGLES, 0, 3);

// On fait pivoter le deuxième triangle puis on le translate
modelview = rotate(modelview, 60.0f, vec3(0, 0, 1));
modelview = translate(modelview, vec3(0.4, 0, 0));

// On envoie une deuxième fois les matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));

/* ***** Affichage du deuxième triangle **** */
glDrawArrays(GL_TRIANGLES, 0, 3);

// On n'utilise plus le shader
glUseProgram(0);
```



Il est inutile d'activer le shader plusieurs fois si vous l'utilisez sur plusieurs polygones. Une seule fois suffit et n'oubliez pas de le désactiver après le dernier affichage.

Je vous conseille de faire des petits tests avec les transformations, entraînez-vous avec avant de passer à la deuxième partie de ce chapitre. 😊

Télécharger : [Code Source C++ du Chapitre 6](#)

Nous arrivons à la fin de cette première partie. Nous avons vu pas mal de notions et il est important que vous les compreniez. Encore une fois, amusez-vous à faire des transformations pour vous familiariser avec OpenGL. Si vous vous sentez prêts, passez à la deuxième partie (qui est plus facile à comprendre que cette partie 😊).

La troisième dimension (Partie 2/2)

Dans cette deuxième partie, nous allons enfin faire ce que vous attendez tous : de la 3D. 😊 Cette partie sera plus facile que la première, il y aura de nouvelles notions à apprendre mais rien de bien compliqué. Encore une fois, si vous n'avez pas tout compris jusque là, je vous conseille de relire à tête reposée les chapitres précédents.

La caméra Introduction

Dans le chapitre précédent, nous avons appris à faire interagir les matrices avec OpenGL et nous avons par la même occasion créé la matrice de projection. La bonne nouvelle, c'est que l'on connaît déjà presque tout pour faire de la 3D. En effet, la troisième dimension est déjà implémentée dans nos programmes, nous l'avons vu en ajoutant la coordonnée Z à nos vertices. Seulement, nous sommes mal placés pour voir ce rendu en relief. Pour régler ce problème, il va falloir placer une chose indispensable à OpenGL : **la caméra**.

Eh oui, 🎂 OpenGL fonctionne avec une **caméra**, exactement comme les films. Il faut donc **placer** une caméra qui va **fixer** un point pour que le spectateur (ou le joueur dans notre cas) puisse voir la scène.

Dans les versions précédentes d'OpenGL, la caméra était gérée par la même librairie que celle qui gérait la projection : la librairie **GLU**. Mais comme vous vous en doutez, nous allons utiliser **GLM** pour la remplacer. 😊

La méthode lookAt

L'ancienne fonction

Avant d'utiliser la librairie **GLM**, nous allons étudier l'ancienne fonction **GLU** qui gérait la caméra. Celle-ci permettait de placer la caméra au niveau d'un point dans l'espace. Ce point de vue nous permet de voir une scène en 3 dimensions au lieu des 2 dimensions dont nous avons l'habitude depuis le début du tuto.

La fonction en question s'appelle **gluLookAt()** et voici son prototype :

Code : C++

```
void gluLookAt(double eyeX, double eyeY, double eyeZ, double
centerX, double centerY, double centerZ, double upX, double upY,
double upZ);
```



Hola y'a trop de paramètres. 😊

Si on les prend à part oui, mais vous remarquerez que les noms se ressemblent plus ou moins. En fait, il n'y a que 3 paramètres. Si on les prend par groupe de 3, on se retrouve avec 3 vecteurs bien distincts :

- Le **vecteur eye** (œil) qui est un vecteur permettant de placer la caméra.
- Le **vecteur center** (centre) qui est le point que la caméra doit fixer. Il y a 3 coordonnées, le point se trouve donc dans un espace 3D.
- Le **vecteur axe** qui est la verticale du repère.

Petite précision pour le dernier vecteur. En théorie, l'axe vertical est l'axe Y mais dans pas mal de jeux-vidéo on prend souvent l'axe Z. Personnellement, je fais de la résistance et je préfère utiliser l'axe Y comme axe vertical comme on nous l'a toujours appris depuis le collège. 😊

La nouvelle méthode

Pour remplacer cette ancienne fonction, nous allons utiliser une méthode de la librairie **GLM** qui s'appelle **lookAt()**. Son prototype est un peu plus compact que la fonction **GLU**:

Code : C++

```
mat4 lookAt(vec3 eye, vec3 center, vec3 up);
```

- **eye** : vecteur permettant de placer la caméra
- **center** : vecteur permettant d'indiquer le point fixé
- **up** : vecteur représentant la verticale du repère

Vous remarquez que la méthode prend bien les 9 paramètres de **gluLookAt()**, elle les place juste dans 3 objets de type *vec3*. 😊

Par ailleurs, elle renvoie une matrice toute neuve, elle ne modifie donc pas le contenu d'une matrice existante comme le font les méthodes de transformations.

Utilisation

Grâce à cette méthode, nous pouvons tout de même utiliser notre caméra.

On va d'ailleurs faire un petit test dès maintenant. Vous vous souvenez des deux triangles du chapitre précédent ? On va voir ce que ça donne si on adopte un "point de vue" en 3 dimensions. Bon je vous préviens, avec des triangles 2D ça va être moche mais ce sera déjà notre premier pas dans la 3D. 😊

Nous allons placer notre caméra au point de coordonnées **(1, 1, 1)** et celle-ci fixera le centre du repère, donc le point de coordonnées **(0, 0, 0)**. Enfin, nous utiliserons l'axe Y comme axe vertical (vous pouvez utiliser celui que vous voulez). L'appel à la méthode **lookAt()** sera donc :

Code : C++

```
// Placement de la caméra  
  
modelview = lookAt(vec3(1, 1, 1), vec3(0, 0, 0), vec3(0, 1, 0));
```

Nous ajoutons cette ligne de code juste après avoir ré-initialisé la matrice **modelview**:

Code : C++

```
// Boucle principale  
  
while(!terminer)  
{  
    /* Gestion des évènements .... */  
  
    // Nettoyage de la fenêtre  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    // Ré-initialisation de la matrice et placement de la caméra  
    modelview = mat4(1.0);  
    modelview = lookAt(vec3(1, 1, 1), vec3(0, 0, 0), vec3(0, 1, 0));
```

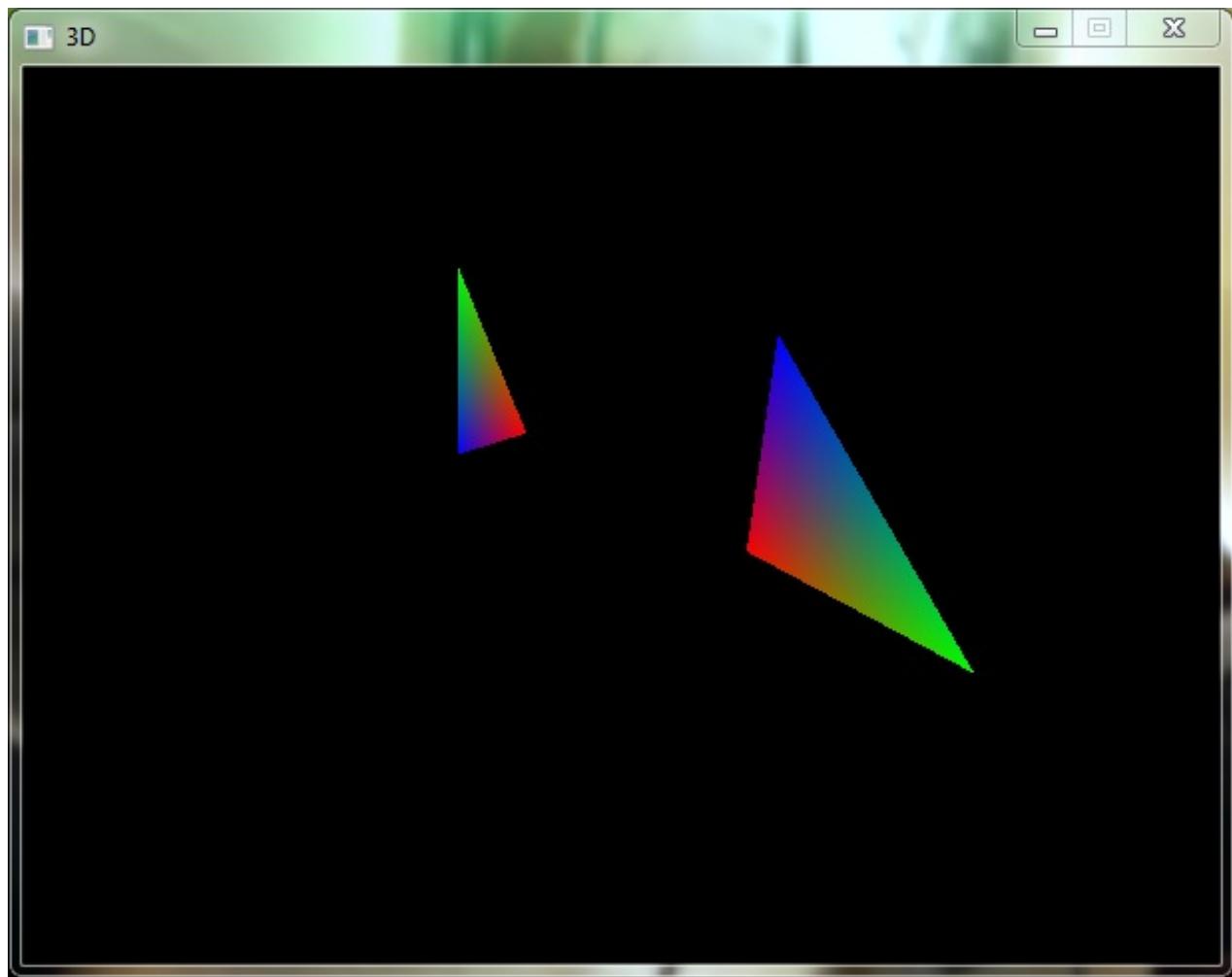
```
    /* Rendu ... */  
}
```

D'ailleurs, vous pouvez maintenant supprimer la ligne qui ré-initialise la matrice **modelview** avec les valeurs d'une matrice d'identité car la méthode **lookAt()** écrase complètement son ancien contenu et fait donc office de ré-initialisation. La ligne est donc inutile. 😊

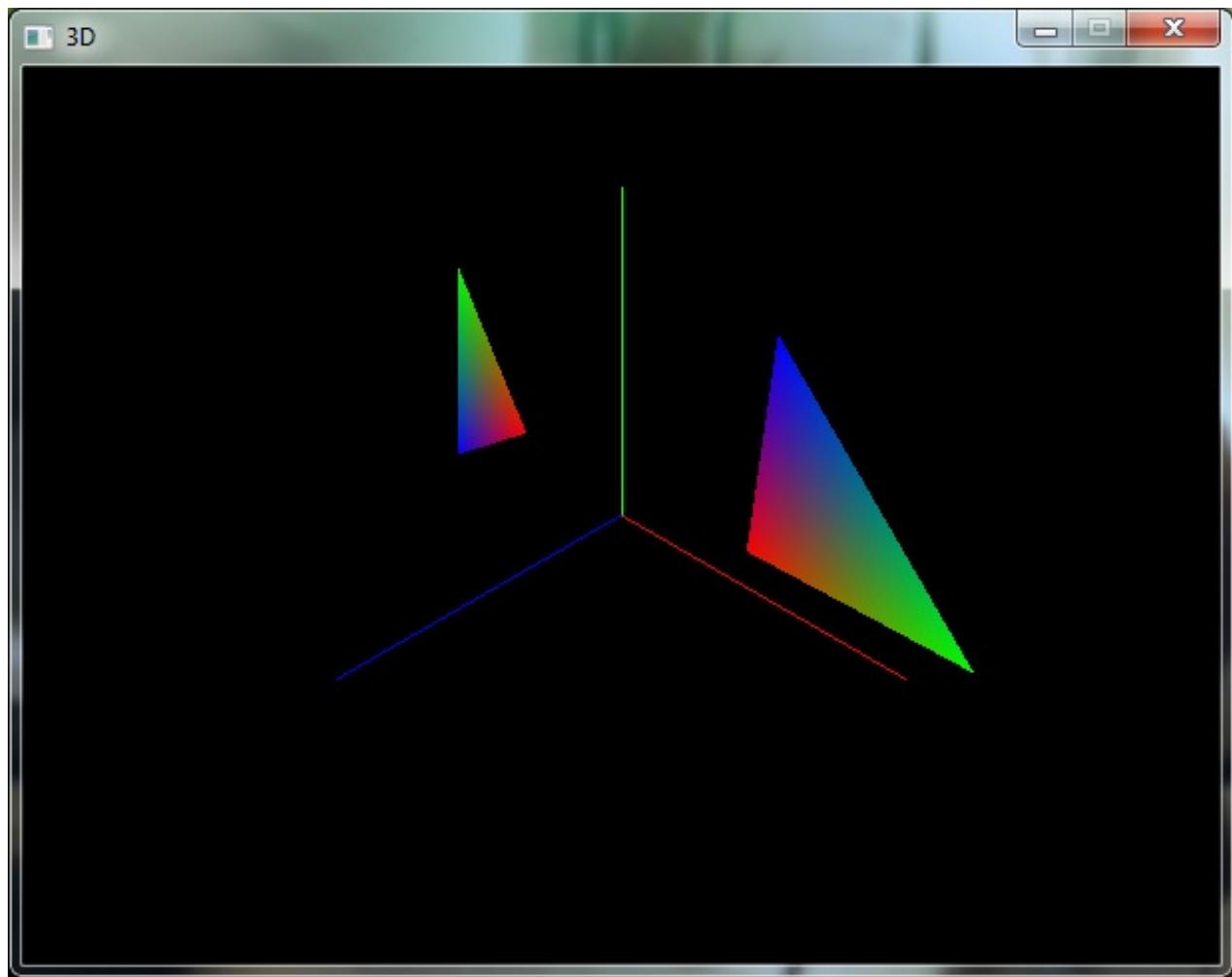
Code : C++

```
// Boucle principale  
  
while (!terminer)  
{  
    /* Gestion des évènements .... */  
  
    // Nettoyage de la fenêtre  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    // Placement de la caméra  
    modelview = lookAt(vec3(1, 1, 1), vec3(0, 0, 0), vec3(0, 1, 0));  
  
    /* Rendu ... */  
}
```

Voici ce que vous devriez obtenir :



Bon je vous avais prévenu c'est laid. 😞 Et pourtant on est bien en 3D :



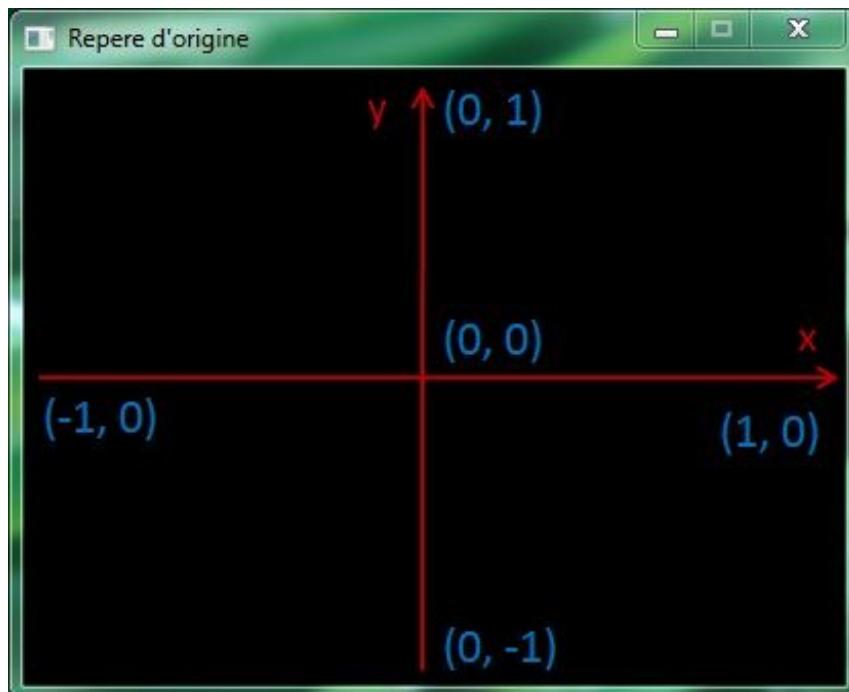
Avec des triangles 2D on n'ira pas très loin mais ça va vite changer. Commençons enfin la partie la plus intéressante. 😺

Affichage d'un Cube

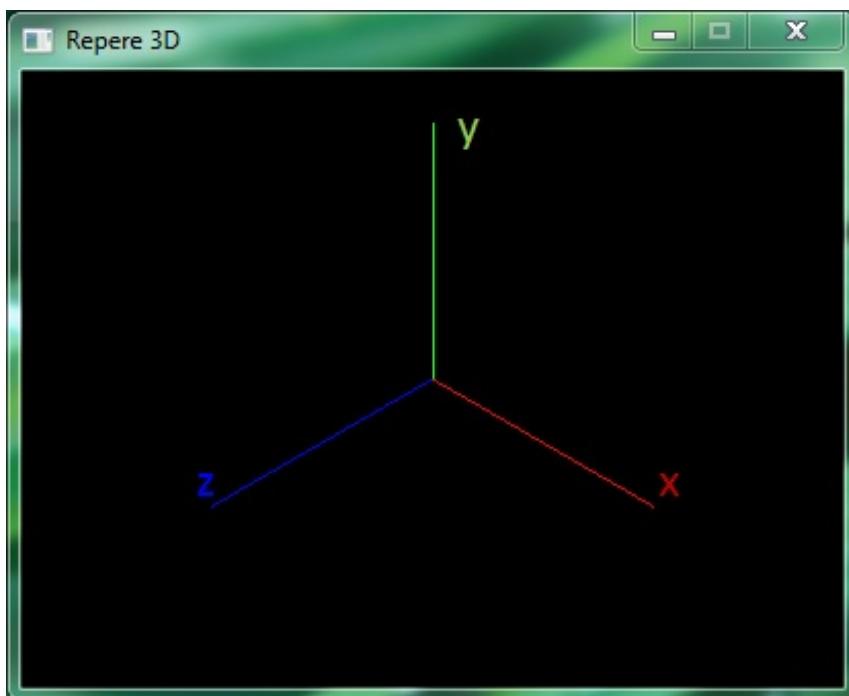
Introduction

Cette fois ça y est, enfin de la 3D ! 😊

La première bonne nouvelle est que l'on va se débarrasser ~~enfin~~ du repère que l'on a utilisé jusqu'à maintenant :



Nous allons désormais utiliser celui-ci :



Nos polygones ne seront plus limités à des coordonnées comprises entre 0 et 1.



Concrètement, qu'est-ce qu'il faut pour passer à la 3D ?

Vous ne vous en êtes peut-être pas rendu compte, mais depuis le début du tutoriel, vous avez déjà appris pas mal de choses. Petit à petit, vous avez appris tout ce qui est nécessaire pour commencer la programmation 3D :

- Un shader
- Des matrices
- La projection

- Une caméra

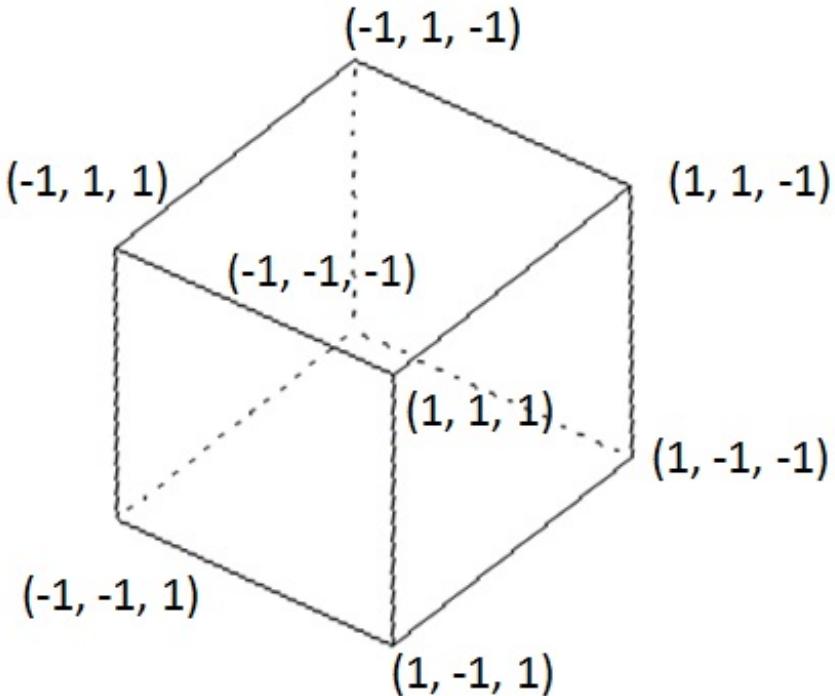
En réunissant intelligemment tout ça, on peut intégrer une troisième dimension à nos programmes. 😊

Affichage d'un cube

La partie théorique

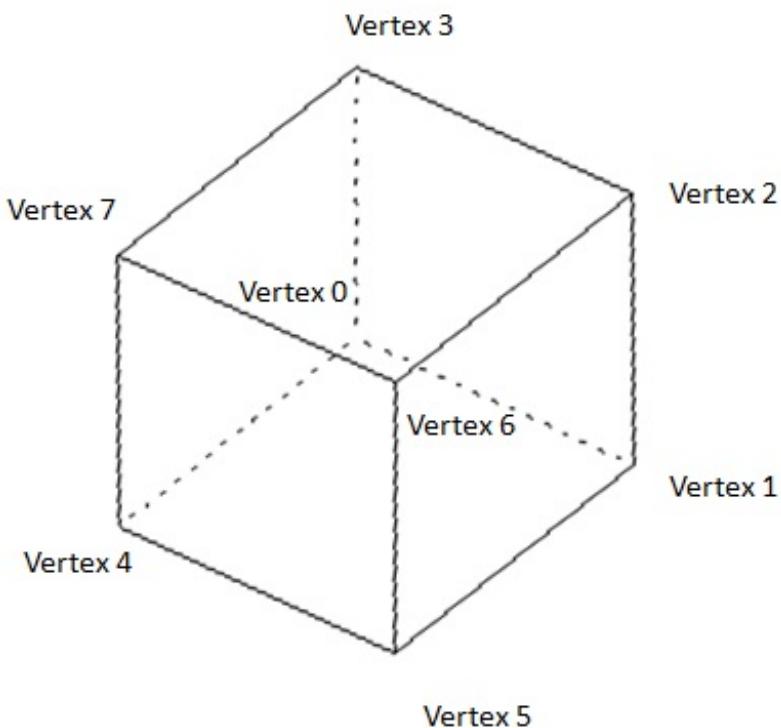
On va commencer par un exercice simple qui réunira tout ce que l'on connaît déjà ainsi que ce que l'on va voir maintenant. A la fin, on sera en mesure d'afficher notre premier modèle 3D : un cube (en couleur s'il vous plaît 😊).

Avant de s'attaquer à la programmation, il est essentiel de faire un peu de théorie en voyant de quoi est composé un cube. Il suffit d'ouvrir un manuel de géométrie pour y lire une des propriétés principales du cube : "un cube est composé de 8 sommets".



Les chiffres en parenthèses représentent les coordonnées de chaque sommet. Une arête mesure donc 2 unités.

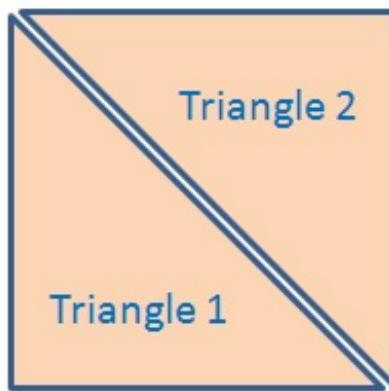
Hum intéressant, on retrouve le mot sommet (= vertex). Il y a 8 sommets, nous aurons donc besoin de 8 vertices :



Bien évidemment, il faudra les dédoubler pour afficher toutes nos faces comme nous le faisions avec le losange par exemple. Nous allons les étudier ensemble une par une en expliquant bien les étapes nécessaires.

D'ailleurs en parlant de ça, dans les anciennes versions d'OpenGL, on utilisait une primitive spécifique pour afficher un carré (comme **GL_TRIANGLES** pour les triangles) que l'on utilisait avec la fonction **glDrawArrays()**. Cependant, cette primitive a également été supprimée au même titre que les fonctions lentes vu que les cartes graphiques ne savent gérer **nativement** que des triangles.

Il existe heureusement une petite combinaison pour afficher des carrés sans cette primitive : il suffit de coller deux triangles rectangles entre eux :



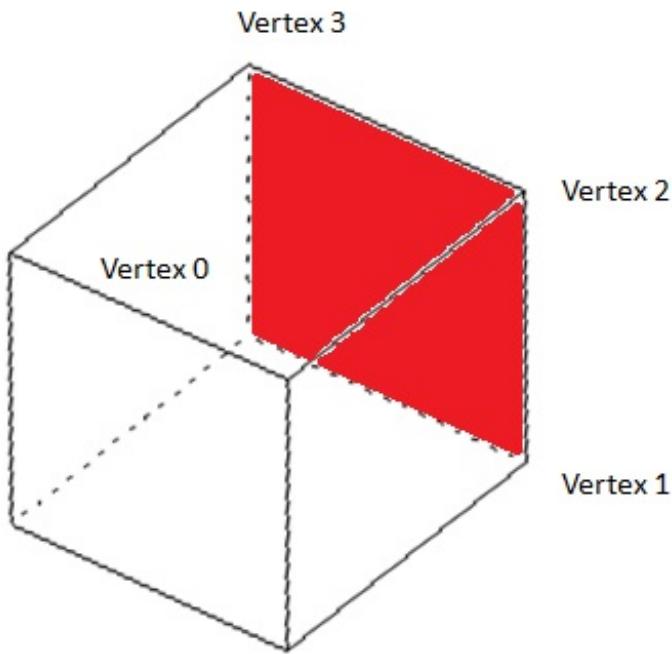
Ne vous inquiétez pas, ce n'est pas plus lent à l'affichage même si on affiche deux choses au lieu d'une. Pour vous dire, que ce soit des sphères, des cubes ou des personnages, absolument tous les modèles 3D ne sont composés uniquement que de triangles. 😊

La première face

Allez on attaque la partie programmation. 😊

Pour le moment, on va tout coder dans la boucle principale, ce sera plus simple à comprendre. Ensuite, nous migrerons proprement le code dans une classe dédiée.

On va commencer notre cube en affichant sa première face (celle du fond) :



Pour cela, nous aurons besoin de **6 sommets** vu que nous avons besoin de deux triangles pour faire un carré. Si on regarde le schéma ci-dessus, on remarque que l'on peut faire un triangle avec les vertices **0, 1 et 2**, et un autre avec les vertices **2, 3 et 0**. Le tableau dont nous avons besoin ressemblera donc à ceci :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Vertices

    float vertices[] = {-1.0, -1.0, -1.0,     1.0, -1.0, -1.0,     1.0,
1.0, -1.0,      // Triangle 1
                    -1.0, -1.0, -1.0,     -1.0, 1.0, -1.0,     1.0,
1.0, -1.0};      // Triangle 2
}
```

Avant d'intégrer ce tableau, nous allons reprendre ensemble la boucle principale pour voir ce que donnerait le nouvel affichage. On commence par évidemment par déclarer les matrices **projection** et **modelview** :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Booléen terminer

    bool terminer(false);

    // Matrices

    mat4 projection;
    mat4 modelview;
```

```

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Boucle principale

    while(!terminer)
    {
        /* Rendu */
    }
}

```

Nous pouvons maintenant intégrer le tableau de vertices que l'on a trouvé juste avant. On le place après la déclaration des matrices :

Code : C++

```

void SceneOpenGL::bouclePrincipale()
{
    // Booléen terminer

    bool terminer(false);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Vertices

    float vertices[] = {-1.0, -1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0,
-1.0, // Triangle 1
-1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0}; // Triangle 2

    // Boucle principale

    while(!terminer)
    {
        /* Rendu */
    }
}

```

Pour admirer le rendu final il ne manque plus qu'à colorier les deux triangles. Nous utiliserons le rouge vu qu'elle est présente dans le schéma un peu plus haut.

Le tableau à utiliser pour cela doit permettre d'affecter une couleur pour chaque vertex. Nous en avons **6** pour le moment donc nous aurons besoin de **6 couleurs** :

Code : C++

```

// Couleurs

float couleurs[] = {1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
// Triangle 1
1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,

```

```
0.0};           // Triangle 2
```

On en profite au passage pour déclarer le **shader** qui permettra de colorier nos faces :

Code : C++

```
// Couleurs

float couleurs[] = {1.0, 0.0, 0.0,    1.0, 0.0, 0.0,    1.0, 0.0, 0.0,
// Triangle 1
                    1.0, 0.0, 0.0,    1.0, 0.0, 0.0,    1.0, 0.0,
0.0};           // Triangle 2

// Shader

Shader shaderCouleur("Shaders/couleur3D.vert",
"Shaders/couleur3D.frag");
shaderCouleur.charger();
```

Une fois toutes ces déclarations faites, nous devrons nous occuper de la caméra. Nous devons la déclarer juste avant la boucle principale et la placer à chaque tour de boucle au point de coordonnées (0, 0, 1). Elle sera en mode '*affichage 2D*' temporairement pour nous permettre de voir le carré correctement :

Code : C++

```
// Boucle principale

while (!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&m_evenements);

    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT);

    // Placement de la caméra

    modelview = lookAt(vec3(0, 0, 1), vec3(0, 0, 0), vec3(0, 1, 0));

    // Rendu

    ...

    // Actualisation de la fenêtre

    SDL_GL_SwapWindow(m_fenetre);
}
```

Pour le rendu en lui-même, il n'y a pas de grand changement à faire. On commence par activer le shader puis on envoie nos données aux tableaux **VertexAttrib**, on sait le faire depuis un moment grâce à la fonction **glVertexAttribPointer()** :

Code : C++

```
// Activation du shader  
  
glUseProgram(shaderCouleur.getProgramID());  
  
// Envoi des vertices  
  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);  
 glEnableVertexAttribArray(0);  
  
// Envoi de la couleur  
  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, couleurs);  
 glEnableVertexAttribArray(1);  
  
....  
  
// Désactivation du shader  
  
glUseProgram(0);
```

Il ne nous reste plus qu'à envoyer les matrices **projection** et **modelview** au shader à l'aide des grosses fonctions. On pensera également à afficher le rendu grâce à la fonction **glDrawArrays()** et à désactiver les tableaux **Vertex Attrib** :

Code : C++

```
// Envoi des matrices  
  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
// Rendu  
  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Désactivation des tableaux  
  
glDisableVertexAttribArray(1);  
glDisableVertexAttribArray(0);
```



N'oubliez pas de mettre le paramètre **count** à **6** pour afficher les deux triangles.

Si on résume tout ça :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
 // Booléen terminer
```

```
bool terminer(false);

// Matrices

mat4 projection;
mat4 modelview;

projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
modelview = mat4(1.0);

// Vertices

float vertices[] = {-1.0, -1.0, -1.0,    1.0, -1.0, -1.0,    1.0,
1.0, -1.0,    // Triangle 1
                  -1.0, -1.0, -1.0,    -1.0, 1.0, -1.0,    1.0,
1.0, -1.0};    // Triangle 2

// Couleurs

float couleurs[] = {1.0, 0.0, 0.0,    1.0, 0.0, 0.0,    1.0, 0.0,
0.0,          // Triangle 1
                  1.0, 0.0, 0.0,    1.0, 0.0, 0.0,    1.0, 0.0,
0.0};          // Triangle 2

// Shader

Shader shaderCouleur("Shaders/couleur3D.vert",
"Shaders/couleur3D.frag");
shaderCouleur.charger();

// Boucle principale

while(!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&m_evenements);

    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT);

    // Placement de la caméra

    modelview = lookAt(vec3(0, 0, 1), vec3(0, 0, 0), vec3(0, 1,
0));

    // Activation du shader

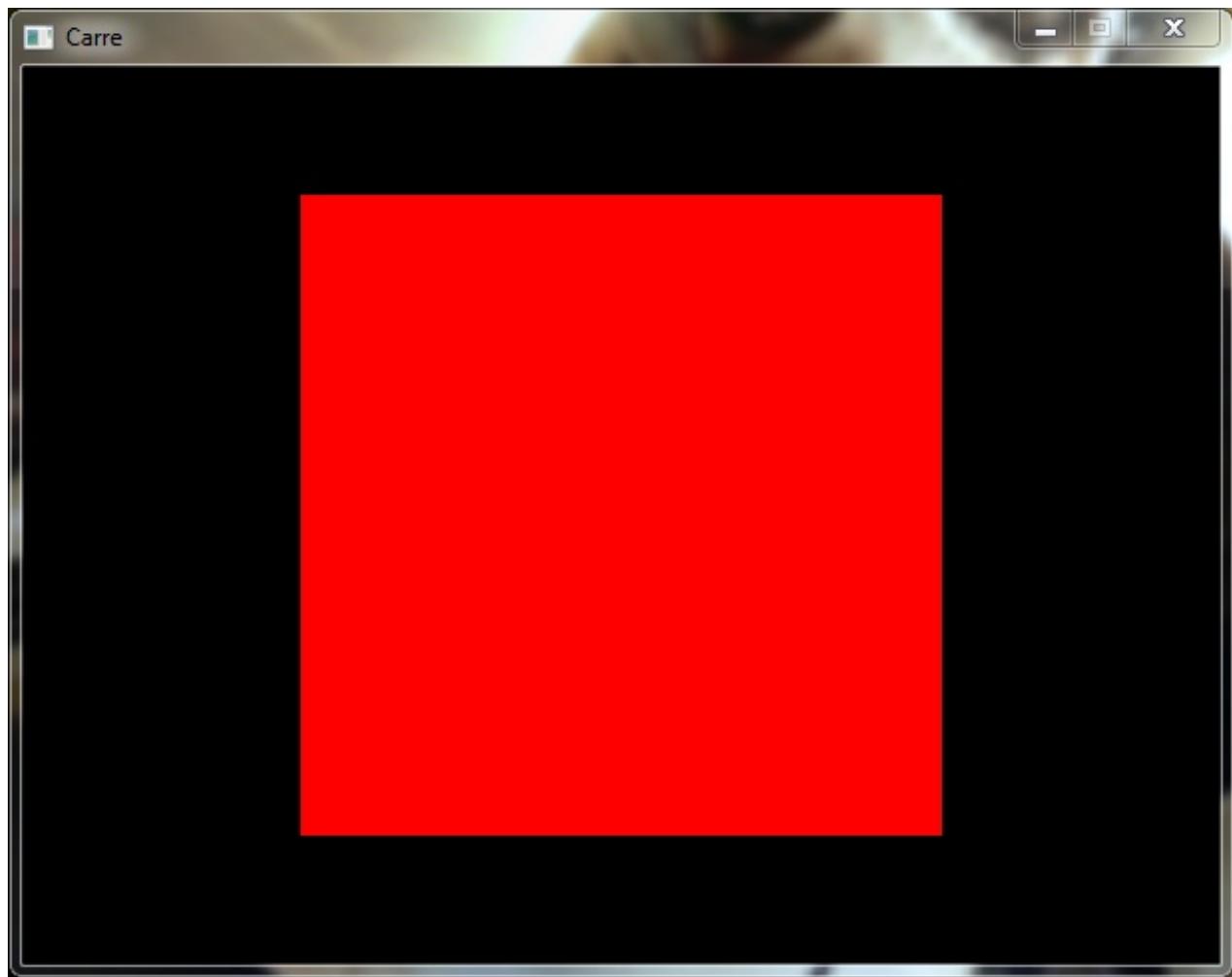
    glUseProgram(shaderCouleur.getProgramID());

    // Envoi des vertices

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
vertices);
    glEnableVertexAttribArray(0);
```

```
// Envoi de la couleur  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,  
couleurs);  
glEnableVertexAttribArray(1);  
  
// Envoi des matrices  
  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),  
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Désactivation des tableaux  
glDisableVertexAttribArray(1);  
glDisableVertexAttribArray(0);  
  
// Désactivation du shader  
glUseProgram(0);  
  
// Actualisation de la fenêtre  
SDL_GL_SwapWindow(m_fenetre);  
}  
}
```

Si vous compilez ce code, vous devriez obtenir votre premier carré avec OpenGL :



Notre première face du cube est maintenant prête. 😊

La deuxième face

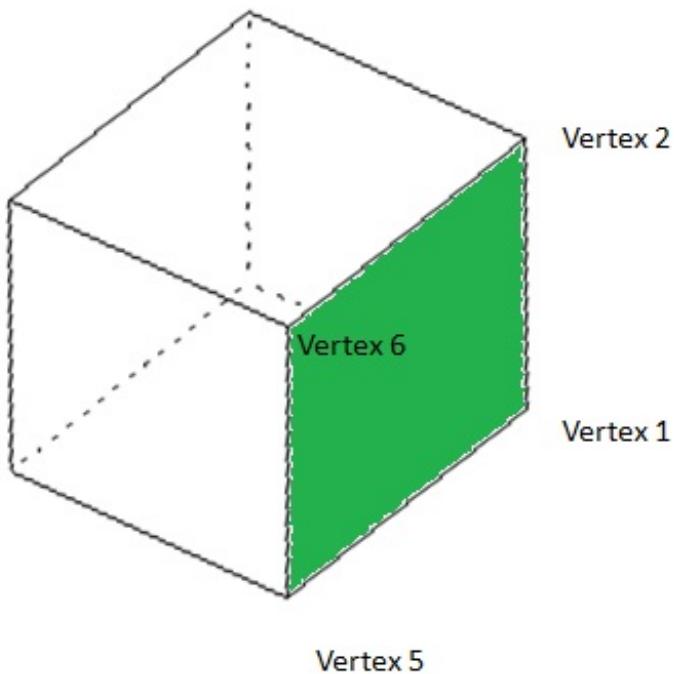
Passons maintenant à la deuxième face du cube.

Pour commencer, on va placer la caméra un peu différemment de façon à voir la scène en 3 dimensions. Nous la positionnerons au point de coordonnées (3, 3, 3) et la ferons fixer l'origine du repère (0, 0, 0) :

Code : C++

```
// Placement de la caméra
modelview = lookAt(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0));
```

La deuxième face que nous devons afficher doit ressembler à ceci :



On remarque qu'il faut prendre les vertices **5, 1 et 2** pour afficher le premier triangle, puis les vertices **2, 6 et 5** pour le second. Si on fait correspondre leurs coordonnées avec le schéma du début, on trouve les deux triangles suivants :

Code : C++

```
// Vertices

float vertices[] = {-1.0, -1.0, -1.0,     1.0, -1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 1           -1.0, -1.0, -1.0,     -1.0, 1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 1           1.0, -1.0, 1.0,     1.0, -1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 2           1.0, -1.0, 1.0,     1.0, 1.0, 1.0,     1.0, 1.0,
-1.0};      // Face 2
```



J'ai volontairement aéré un peu le tableau pour le rendre plus lisible.

Évidemment, si nous utilisons de nouveaux vertices il faut leur associer une couleur. On ajoute donc **6** triplets au tableau **couleurs** spécialement pour eux. Nous utiliserons le vert pour différencier les deux faces :

Code : C++

```
// Couleurs

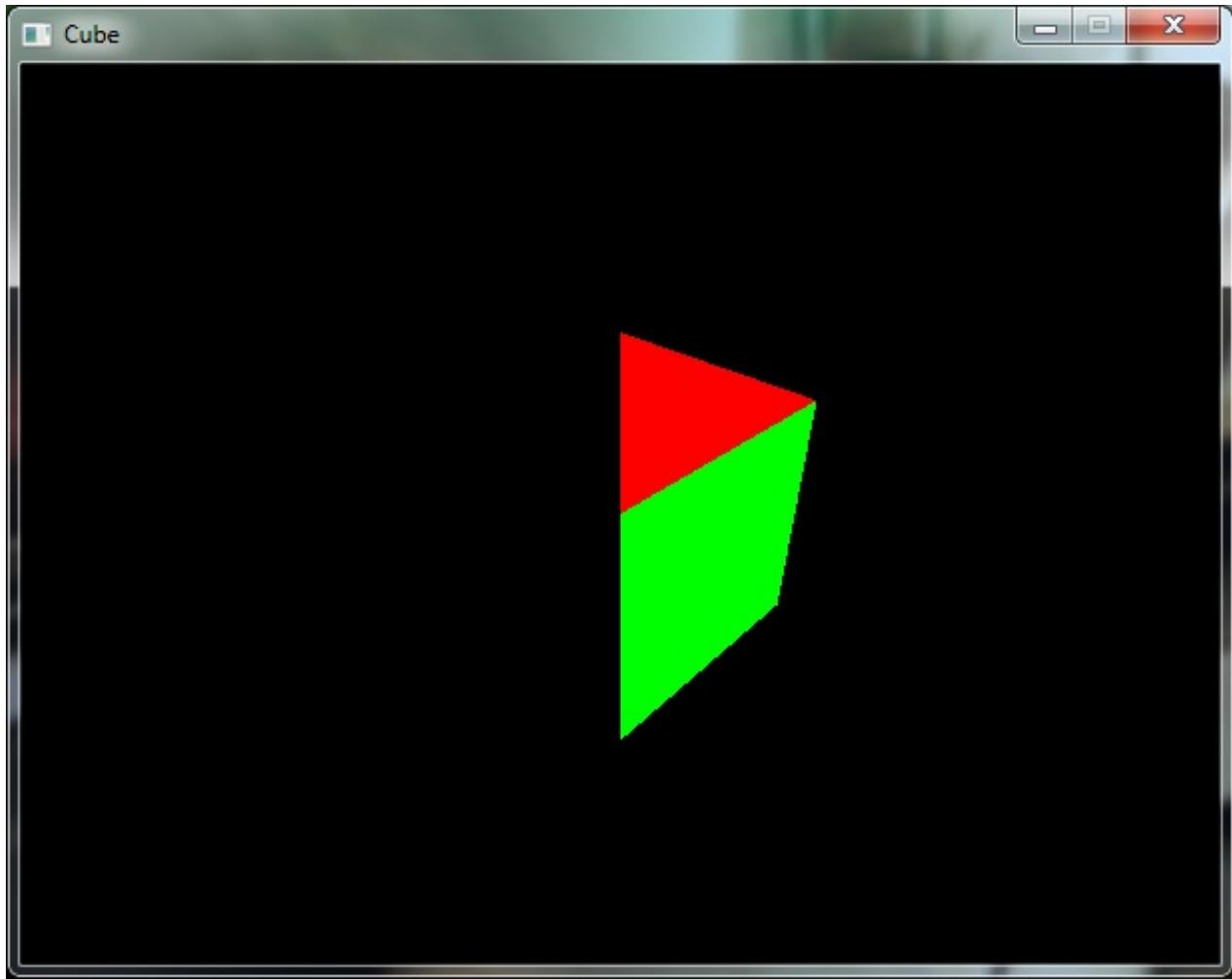
float couleurs[] = {1.0, 0.0, 0.0,     1.0, 0.0, 0.0,     1.0, 0.0, 0.0,
// Face 1           1.0, 0.0, 0.0,     1.0, 0.0, 0.0,     1.0, 0.0, 0.0,
// Face 1           0.0, 1.0, 0.0,     0.0, 1.0, 0.0,     0.0, 1.0, 0.0,
// Face 2           0.0, 1.0, 0.0,     0.0, 1.0, 0.0,     0.0, 1.0,
0.0};      // Face 2
```

Pour afficher la nouvelle face, il suffit de modifier le fameux paramètre **count** pour qu'il prenne en compte les 6 nouveaux sommets. On lui donne donc la valeur **6 + 6 = 12** :

Code : C++

```
// Affichage des triangles  
glDrawArrays(GL_TRIANGLES, 0, 12);
```

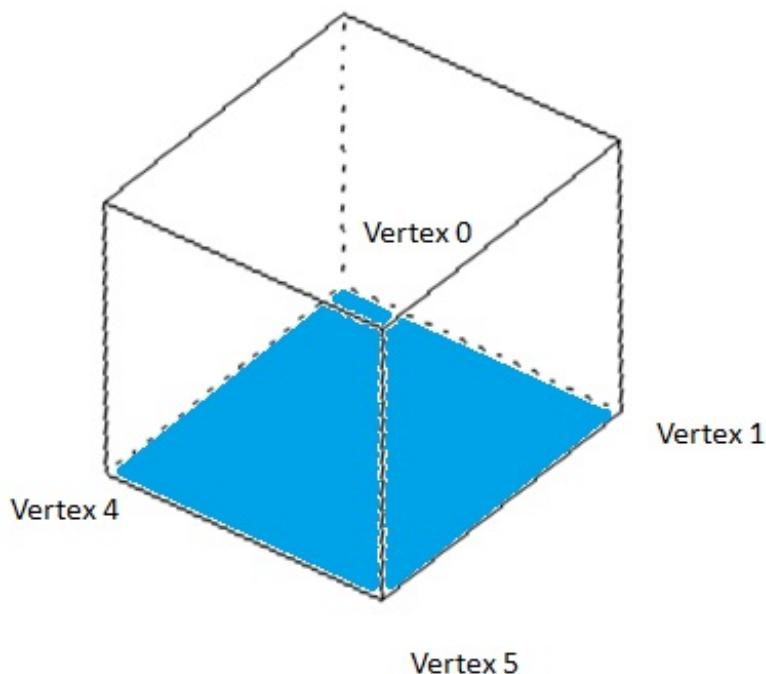
Si vous compilez tout ça, vous devriez obtenir :



On a maintenant la deuxième face, et en 3D s'il vous plaît. 😊

La troisième face

Allez, on continue avec la troisième face. On commence par définir les sommets dont nous aurons besoin :



Comme d'habitude, on fait correspondre ces sommets avec leurs coordonnées pour trouver le tableau suivant :

Code : C++

```
// Vertices

float vertices[] = {-1.0, -1.0, -1.0,     1.0, -1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 1           -1.0, -1.0, -1.0,     -1.0, 1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 1           1.0, -1.0, 1.0,     1.0, -1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 2           1.0, -1.0, 1.0,     1.0, 1.0, 1.0,     1.0, 1.0,
-1.0,      // Face 2           -1.0, -1.0, 1.0,     1.0, -1.0, 1.0,     1.0, -1.0,
-1.0,      // Face 3           -1.0, -1.0, 1.0,     -1.0, -1.0, -1.0,     1.0,
-1.0, -1.0}; // Face 3
```

Ensuite, on fait correspondre une nouvelle couleur aux deux nouveaux triangles formés par les vertices. On utilisera le bleu comme sur le schéma :

Code : C++

```
// Couleurs

float couleurs[] = {1.0, 0.0, 0.0,     1.0, 0.0, 0.0,     1.0, 0.0, 0.0,
// Face 1           1.0, 0.0, 0.0,     1.0, 0.0, 0.0,     1.0, 0.0, 0.0,
// Face 1           0.0, 1.0, 0.0,     0.0, 1.0, 0.0,     0.0, 1.0, 0.0,
// Face 2           0.0, 1.0, 0.0,     0.0, 1.0, 0.0,     0.0, 1.0, 0.0,
// Face 2           0.0, 0.0, 1.0,     0.0, 0.0, 1.0,     0.0, 0.0, 1.0,
// Face 3
```

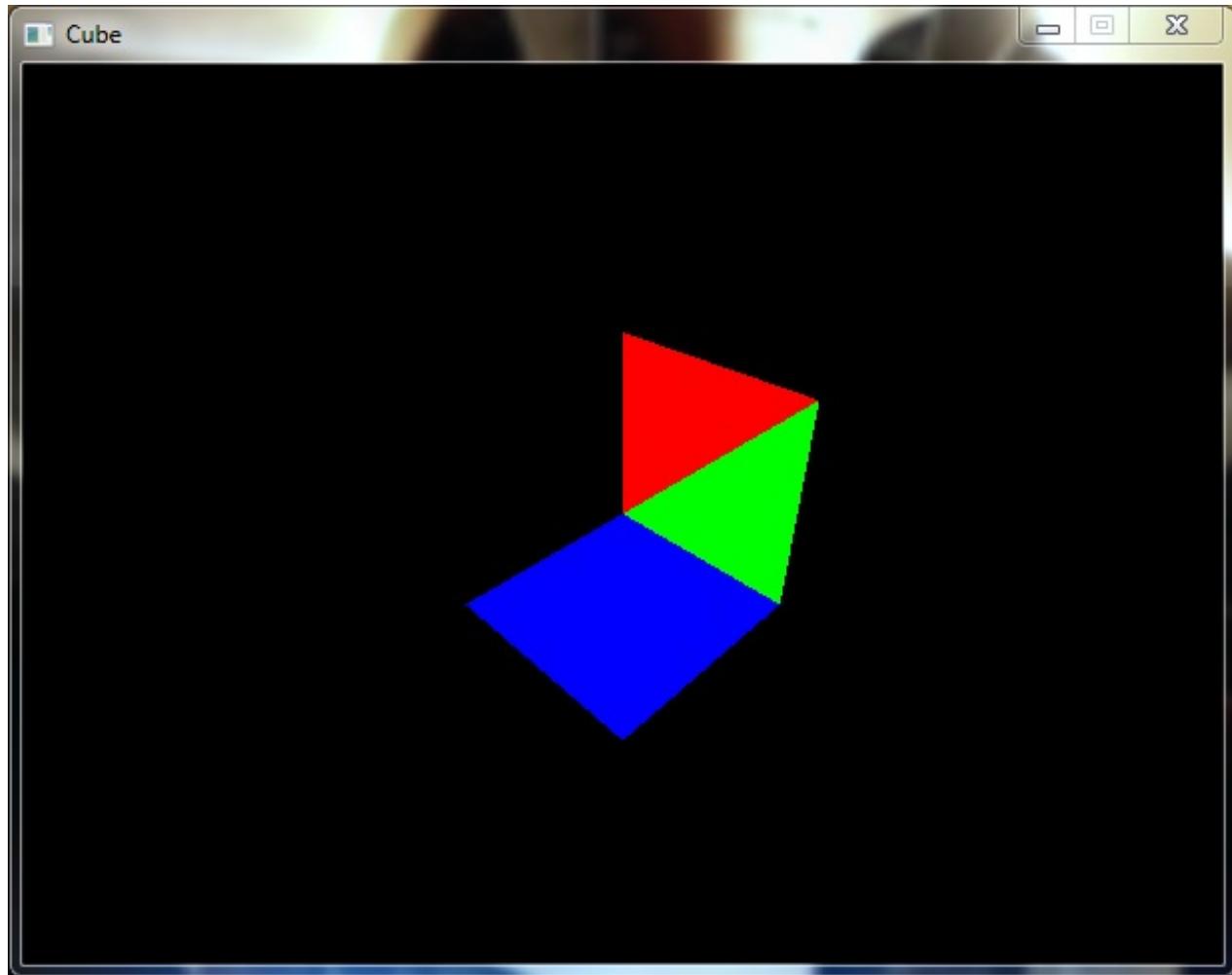
```
    0.0, 0.0, 1.0,    0.0, 0.0, 1.0,    0.0, 0.0,  
1.0};           // Face 3
```

Pour finir, on doit modifier une fois de plus le paramètre **count** pour prendre en compte les nouveaux vertices. Sa valeur passe de **12** à **18** :

Code : C++

```
// Affichage des triangles  
glDrawArrays(GL_TRIANGLES, 0, 18);
```

En compilant le nouveau code, on obtient :



Pourquoi ça s'affiche comme ça ? Y'a un bug ? 😊

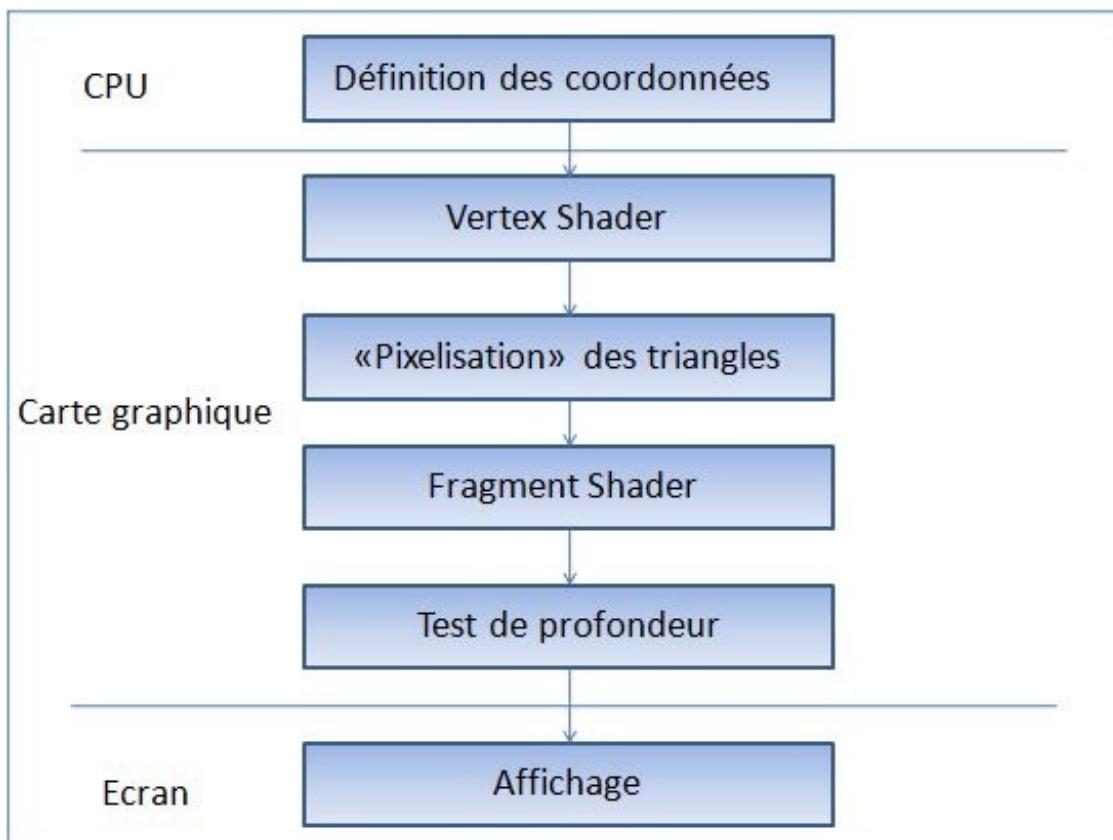
Non, ce n'est pas un bug, et vous allez vite comprendre pourquoi la face ne s'affiche pas correctement. 😊

Le Depth Buffer

C'est la première fois que nous avons un problème d'affichage, et c'est tout à fait normal puisqu'avant nous n'avions eu de formes superposées l'une sur l'autre. Ici, la face bleue et la face verte se superposent, et pour OpenGL c'est un problème car il ne sait pas quelle forme doit être visible et quelle forme doit être cachée. N'oubliez pas qu'un ordinateur est très bête, il ne sait rien faire à part calculer.

Le **Depth Buffer** (ou *Tampon de profondeur*) est ce qui va permettre à OpenGL de comprendre ce qu'il doit afficher et ce qu'il doit masquer. Si un pixel de modèle se trouve derrière un autre alors le Depth Buffer indiquera à OpenGL : "N'affiche pas ce pixel, mais affiche celui-ci car il est devant".

Je vous avais déjà parlé brièvement de cette notion dans le chapitre sur les shaders, notamment avec ce schéma :



La dernière étape du **pipeline** 3D était le **Test de profondeur**. C'est justement là qu'intervient le Depth Buffer. Heureusement pour nous, il ne faudra pas gérer ce tampon par nous-même, cette fonctionnalité n'a pas été supprimée avec la nouvelle version d'OpenGL. 😊 Avant que l'API puisse se servir de ce tampon, il faut l'activer grâce à la fonction `glEnable()`.

Code : C++

```
void glEnable(GLenum cap);
```

Nous reverrons plusieurs fois cette fonction qui permet d'activer certaines fonctionnalités d'OpenGL. Le paramètre `cap` est justement la fonctionnalité à activer. Pour le *Depth Buffer*, on lui donne le paramètre `GL_DEPTH_TEST`. On va donc appeler la fonction comme ceci dans la méthode `initGL()` juste après l'initialisation de la librairie `GLEW` :

Code : C++

```
bool SceneOpenGL::initGL()
{
#ifndef WIN32

    /* ***** Initialisation de la librairie GLEW ***** */

```

```
#endif

// Activation du Depth Buffer
glEnable(GL_DEPTH_TEST);

// Tout s'est bien passé, on retourne true
return true;
```



Même si vous n'êtes pas sous Windows, vous devez initialiser le *Depth Buffer* !

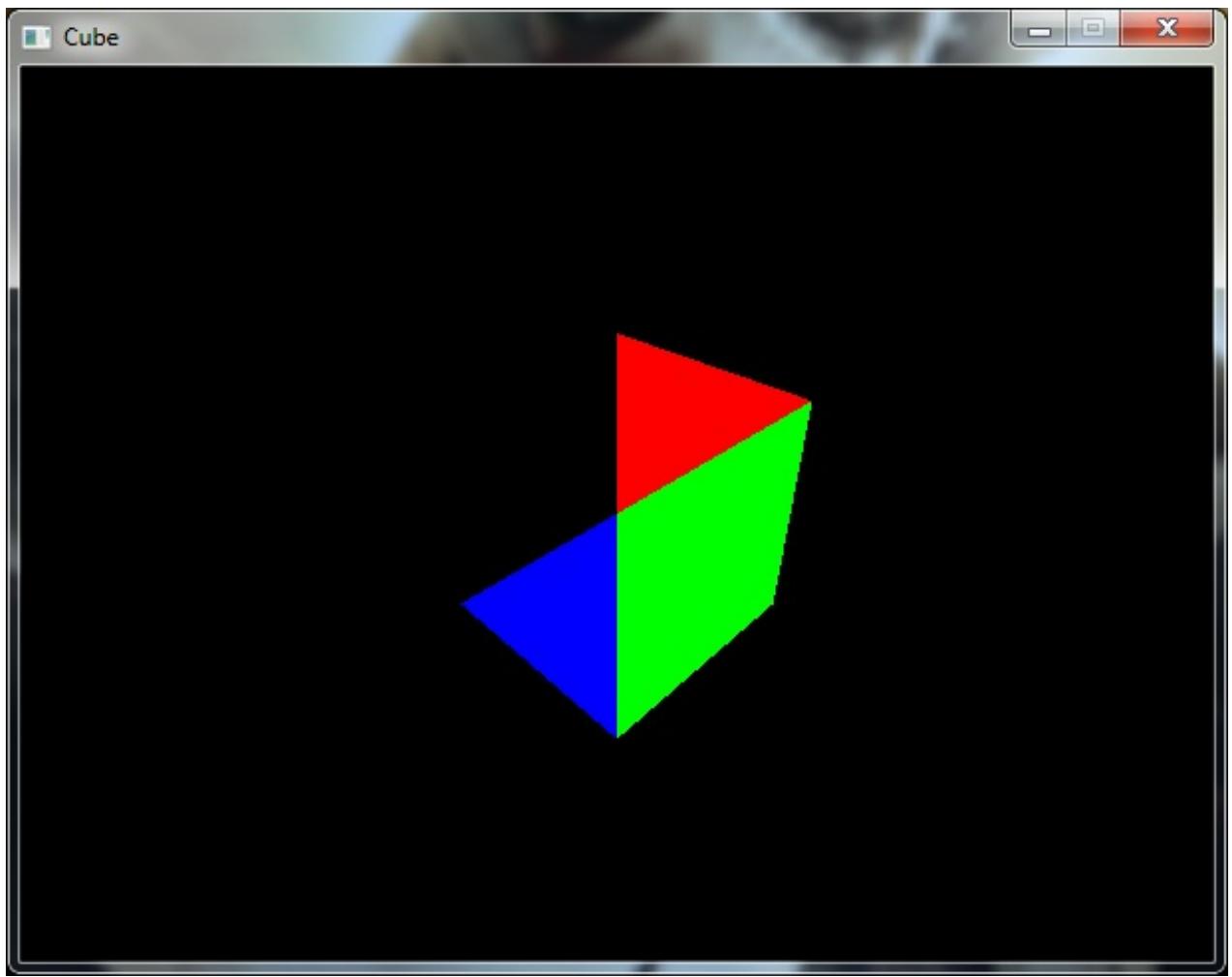
A chaque tour de boucle, il faudra (comme avec les couleurs et la matrice **modelview**) ré-initialiser le Depth Buffer afin de vider toute traces de l'affichage précédent. Pour ça, il suffit d'ajouter un paramètre à une fonction que l'on utilise déjà : **glClear()**. Rappelez-vous que cette fonction permet de vider les buffers qu'on lui donne en paramètre.

Pour le moment, on ne lui donne que le paramètre **GL_COLOR_BUFFER_BIT** pour effacer ce qui se trouve à l'écran. Maintenant, on va ajouter le paramètre **GL_DEPTH_BUFFER_BIT** pour effacer le Depth Buffer :

Code : C++

```
// Nettoyage de la fenêtre et du Depth Buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Maintenant que l'on sait ce qu'est le Depth Buffer, on peut régler notre problème d'affichage. Après avoir placé la fonction **glEnable()**, vous devriez obtenir ceci :

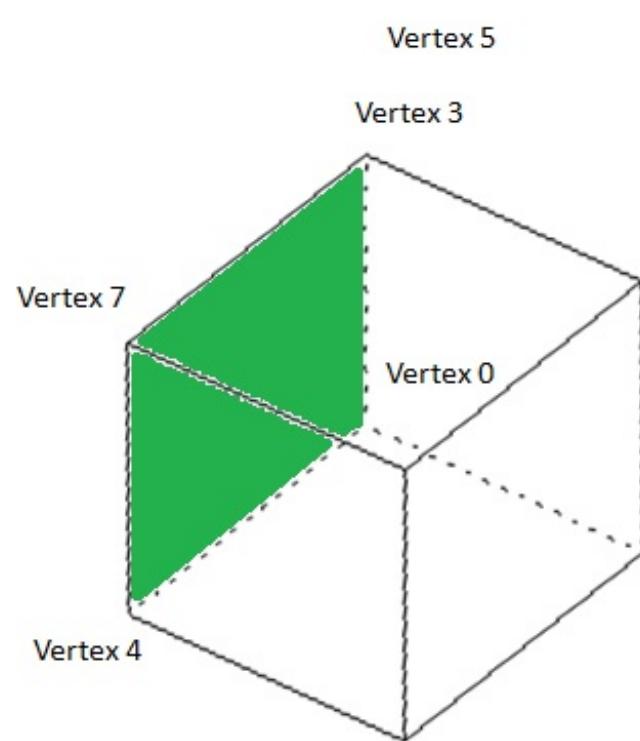
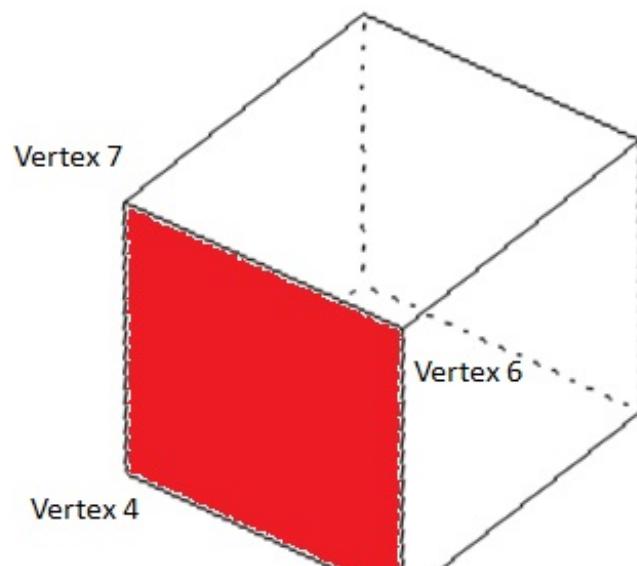


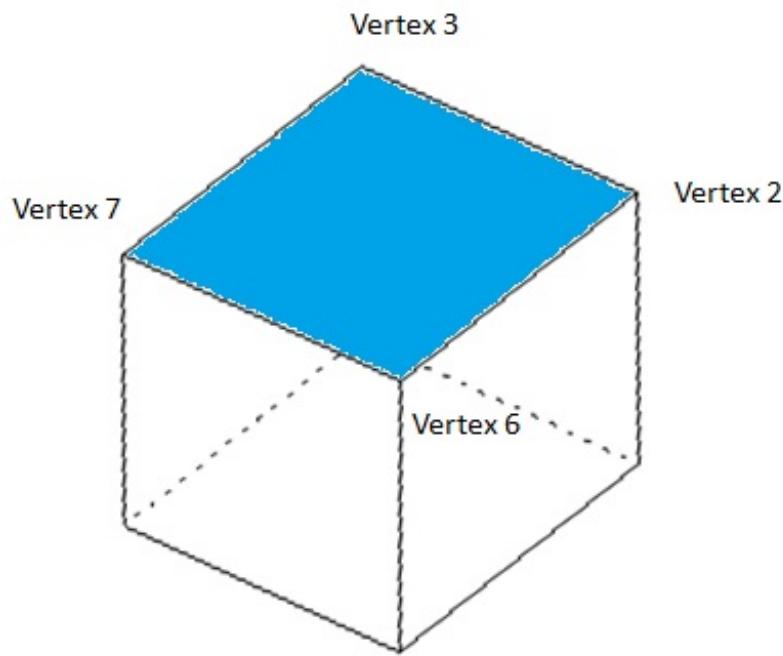
Les trois dernières faces

Courage nous sommes presque au bout, il ne reste plus qu'à afficher les 3 dernières faces. Et je fais bien de vous dire courage car je vais vous demander de terminer les trois dernières faces tous seuls. 🍪

Je vais vous donner les schémas contenant les vertices et la couleur dont vous aurez besoin pour chaque face, pour le reste ce sera à vous de le faire. Il n'y a rien de compliqué en plus, nous avons déjà fait la moitié du cube ensemble et vous n'aurez pas de surprise d'affichage car nous savons maintenant gérer le **Depth Buffer**.

Les tableaux finaux peuvent vous paraître gros et moches, c'est tout à fait normal. Pour vous dire, les vertices de personnages ou de décors sont infiniment plus moches au vu de leur nombre de données. 😊 Mais l'avantage avec eux c'est que nous n'avons pas à les coder à la main, nous ne les voyons même pas d'ailleurs. Mais bon, ça ça sera pour plus tard. Pour le moment, je vous demande de finir notre fameux cube à l'aide des schémas suivants :





Allez hop à votre clavier !

.....

On passe à la correction. Le principe reste le même, il faut juste faire correspondre les vertices et les couleurs. Voici ce que donne les tableaux finaux :

Secret ([cliquez pour afficher](#))

Vos vertices peuvent parfaitement être déclarés dans un ordre différent de celui que je donne. Ce n'est pas grave du moment que vous affichez des carrés correctement :

Code : C++

```
float vertices[] = {-1.0, -1.0, -1.0,     1.0, -1.0, -1.0,     1.0,
1.0, -1.0,      // Face 1
           -1.0, -1.0, -1.0,     -1.0, 1.0, -1.0,     1.0,
1.0, -1.0,      // Face 1
           1.0, -1.0, 1.0,      1.0, -1.0, -1.0,     1.0, 1.0,
-1.0,      // Face 2
           1.0, -1.0, 1.0,      1.0, 1.0, 1.0,      1.0, 1.0,
-1.0,      // Face 2
           -1.0, -1.0, 1.0,     1.0, -1.0, 1.0,     1.0,
-1.0, -1.0,      // Face 3
           -1.0, -1.0, 1.0,     -1.0, -1.0, -1.0,     1.0,
-1.0, -1.0,      // Face 3
           -1.0, -1.0, 1.0,     1.0, -1.0, 1.0,     1.0, 1.0,
1.0,      // Face 4
           -1.0, -1.0, 1.0,     -1.0, 1.0, 1.0,      1.0, 1.0,
1.0,      // Face 4
           -1.0, -1.0, -1.0,    -1.0, -1.0, 1.0,     -1.0,
```

```
    1.0, 1.0,      // Face 5
    -1.0, -1.0, -1.0,   -1.0, 1.0, -1.0,   -1.0,
    1.0, 1.0,      // Face 5
    -1.0, 1.0, 1.0,   1.0, 1.0, 1.0,   1.0, 1.0,
-1.0,           // Face 6
    -1.0, 1.0, 1.0,   -1.0, 1.0, -1.0,   1.0, 1.0,
-1.0};        // Face 6
```

L'ordre des couleurs peut, lui aussi, être différent. Ce qui compte, c'est que les lignes de couleur correspondent à leurs lignes de vertex :

Code : C++

```
// Couleurs

float couleurs[] = {1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0,           // Face 1
    1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0,           // Face 1
    0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0,           // Face 2
    0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0,           // Face 2
    0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0,           // Face 3
    0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0,           // Face 3
    1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0,           // Face 4
    1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0,           // Face 4
    0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0,           // Face 5
    0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0,           // Face 5
    0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0,           // Face 6
    0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0};        // Face 6
```

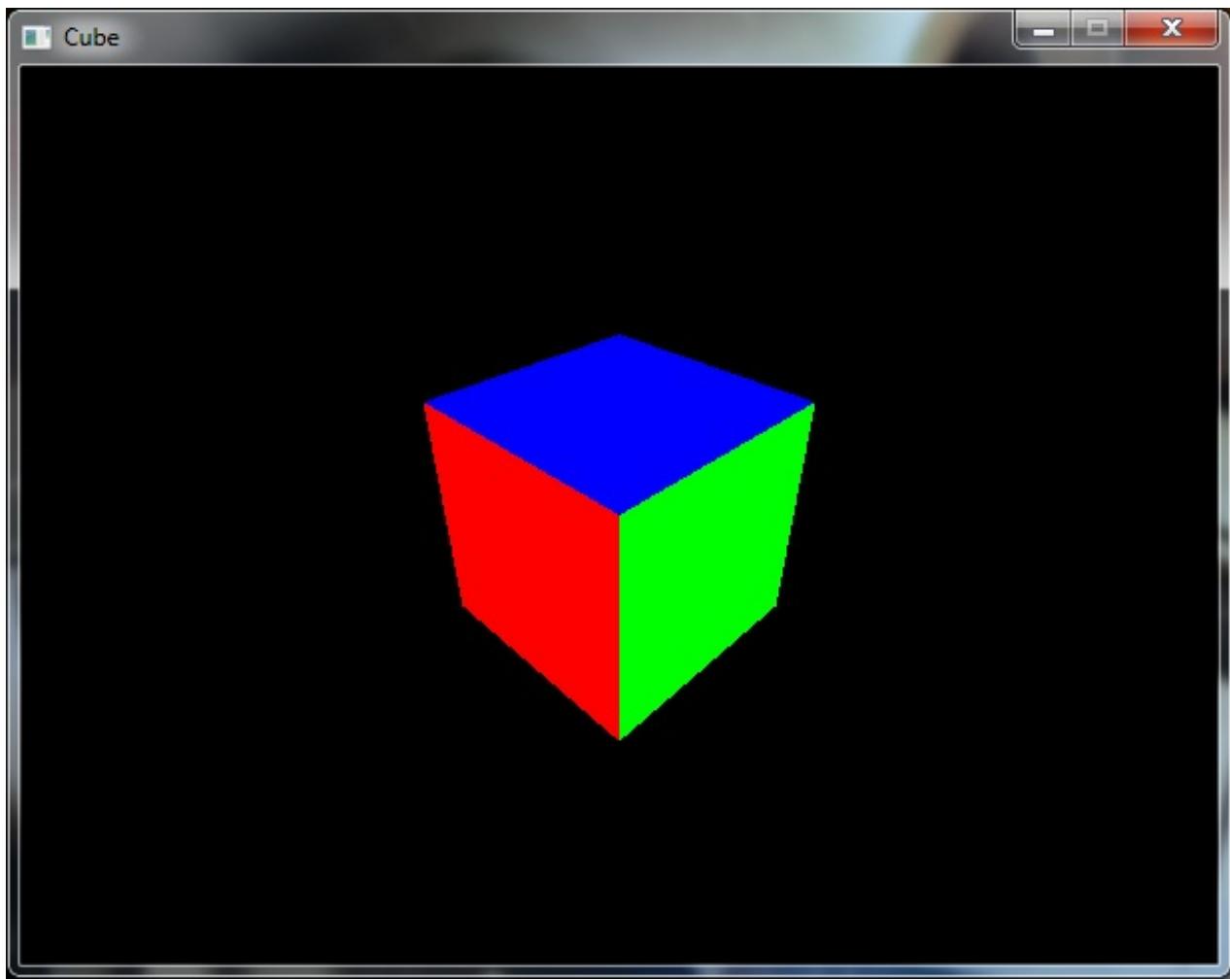
Au niveau de la fonction **glDrawArrays()**, vous devriez avoir la valeur du paramètre **count** à **36** :

Code : C++

```
// Affichage des triangles

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

A la fin, vous devriez avoir votre premier modèle 3D !



Magnifique n'est-ce pas ? 😊

La classe Cube

La classe Cube

Le header

Après tous les efforts que nous avons fournis dans la partie précédente, nous avons enfin pu afficher notre premier modèle 3D.



Vous avez remarqué que, mises à part les matrices, le processus était le même que pour les modèles 2D. C'est-dire-à qu'il nous a suffi d'activer un shader, d'envoyer les données aux tableaux **Vertex Attrib**, puis d'afficher le tout avec **glDrawArrays()**.

Ce que nous allons faire maintenant va nous permettre de nettoyer un peu la boucle principale. En effet, comme je vous l'ai précisé dans la correction du cube, les tableaux de vertices et de couleurs sont assez indigestes. Il serait donc judicieux de créer une classe dédiée au cube de façon à enfermer ces lignes de code à l'intérieur. Nous gagnerions en lisibilité et de plus, nous pourrions créer des cubes à l'infini en seulement quelques lignes de code !

Notre nouvel objectif va donc être la création d'une classe **Cube** dont on se servira pour la suite du tutoriel.

Et nous allons commencer tout de suite par le header. Celui-ci sera placé dans un fichier que nous appellerons **Cube.h** et devra contenir la déclaration de la classe ainsi que les inclusions nécessaires pour OpenGL, les shaders et les matrices :

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE
```

```
// Includes OpenGL

#ifndef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

// Includes GLM

#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Includes

#include "Shader.h"

// Classe Cube

class Cube
{
public:

private:
};

#endif
```



N'oubliez pas les en-têtes de **GLM** 😊

Au niveau des attributs, cette classe devra contenir tous les éléments dont nous avons eu besoin pour afficher notre modèle, à savoir :

- Un objet de type **Shader**
- Un tableau **flottant** de vertices
- Un tableau **flottant** de couleurs

Les matrices ne font pas partie de cette liste car nous n'en avons besoin qu'au moment de l'affichage, inutile donc de créer des attributs pour elles. Nous les enverrons en tant que paramètres dans une méthode.

Si on rassemble tout ça, on trouve :

Code : C++

```
// Attributs

Shader m_shader;
float m_vertices[108];
float m_couleurs[108];
```

La taille **108** des deux tableaux vient de la multiplication du nombre de vertices nécessaires pour un cube (**36**) par leur nombre de

coordonnées (3), ce qui fait **36 vertices x 3 coordonnées = 108 cases**.

Passons maintenant au constructeur, celui-ci aura besoin de trois paramètres : la **taille** du cube que l'on veut afficher ainsi que les deux **codes sources** du shader à utiliser.

Nous n'avons pas intégré la possibilité de choisir les dimensions avant afin d'éviter d'alourdir le code qui était déjà assez dense. Mais vu qu'à présent nous codons une classe, il serait quand même plus agréable de pouvoir créer des cubes de n'importe quelle taille en modifiant simplement une seule valeur. 😊

Nous prendrons une variable de type **float** pour gérer cette taille. Quant aux autres paramètres, vous savez déjà que ce seront des **string** :

Code : C++

```
Cube (float taille, std::string const vertexShader, std::string const fragmentShader);
```

On profite de ce passage pour déclarer le destructeur de la classe :

Code : C++

```
~Cube () ;
```

Après tous ces petits ajouts, nous nous retrouvons devant le header complet de la classe **Cube** :

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE

// Includes OpenGL
#ifndef WIN32
#include <GL/glew.h>
#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>
#endif

// Includes GLM
#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Includes
#include "Shader.h"

// Classe Cube
class Cube
{
    public:
```

```

        Cube(float taille, std::string const vertexShader, std::string
const fragmentShader);
~Cube();

private:
    Shader m_shader;
    float m_vertices[108];
    float m_couleurs[108];
};

#endif

```

Le constructeur

Passons maintenant à l'implémentation de la classe avec en premier lieu le constructeur.

Celui-ci débute avec l'initialisation des attributs. Nous en avons 3 mais seul le shader peut vraiment être initialisé ici car les vertices et les couleurs sont des tableaux, nous ne pouvons donc pas le faire directement après les deux points ":". Nous lui donnons les deux codes sources reçus en paramètres :

Code : C++

```

Cube::Cube(float taille, std::string const vertexShader, std::string
const fragmentShader) : m_shader(vertexShader, fragmentShader)
{
}

```

Ce qui se trouve entre les accolades commence également par le shader car nous devons appeler sa méthode **charger()** de façon à le charger complètement :

Code : C++

```

Cube::Cube(float taille, std::string const vertexShader, std::string
const fragmentShader) : m_shader(vertexShader, fragmentShader)
{
    // Chargement du shader
    m_shader.charger();
}

```

Le shader est maintenant initialisé et prêt à l'emploi.

On passe maintenant au plus délicat : l'initialisation des tableaux de vertices et de couleurs. Il y a deux manières de faire en C++ :

- Soit on initialise leurs valeurs une par une (donc on initialise séparément les 24 *valeurs* d'un tableau de 24 *cases* par exemple).
- Soit on déclare des tableaux temporaires contenant les valeurs désirées, puis on utilise des boucles pour les affecter aux vrais tableaux.

Si on utilise la première méthode, il nous faudrait **108** lignes de code juste pour initialiser les vertices, et le double si on s'occupe aussi des couleurs. Avouez tout de même que c'est méchamment fastidieux, surtout si on doit le faire deux fois. Je pense donc que vous serez d'accord pour utiliser la seconde méthode. 😊

Nous devons donc utiliser un tableau temporaire qui va contenir tous les vertices du cube, nous l'appellerons **verticesTmp[]** :

Code : C++

```
// Vertices temporaires

float verticesTmp[] = {-1.0, -1.0, -1.0,    1.0, -1.0, -1.0,    1.0,
1.0, -1.0,      // Face 1
                    -1.0, -1.0, -1.0,    -1.0, 1.0, -1.0,    1.0,
1.0, -1.0,      // Face 1
                    1.0, -1.0, 1.0,     1.0, -1.0, -1.0,    1.0,
1.0, -1.0,      // Face 2
                    1.0, -1.0, 1.0,     1.0, 1.0, 1.0,     1.0, 1.0,
-1.0,           // Face 2
                    -1.0, -1.0, 1.0,    1.0, -1.0, 1.0,     1.0,
-1.0, -1.0,      // Face 3
                    -1.0, -1.0, 1.0,    -1.0, -1.0, -1.0,    1.0,
-1.0, -1.0,      // Face 3
                    -1.0, -1.0, 1.0,    1.0, -1.0, 1.0,     1.0,
1.0, 1.0,       // Face 4
                    -1.0, -1.0, 1.0,    -1.0, 1.0, 1.0,     1.0,
1.0, 1.0,       // Face 4
                    -1.0, -1.0, -1.0,   -1.0, -1.0, 1.0,    -1.0,
1.0, 1.0,       // Face 5
                    -1.0, -1.0, -1.0,   -1.0, 1.0, -1.0,    -1.0,
1.0, 1.0,       // Face 5
                    -1.0, 1.0, 1.0,     1.0, 1.0, 1.0,     1.0, 1.0,
-1.0,           // Face 6
                    -1.0, 1.0, 1.0,     -1.0, 1.0, -1.0,    1.0,
1.0, -1.0};     // Face 6
```

Les vertices en l'état n'ont que bien peu d'intérêt car ils ne prennent pas en compte le paramètre **taille** du constructeur. Pour régler ce problème, nous allons simplement remplacer toutes les occurrences de la valeur **1.0** par le paramètre **taille** lui-même. C'est un peu long à faire mais la fonctionnalité "**Find and Replace**" de votre IDE devrait vous faciliter un peu la tâche. 😊

Le tableau remanié devrait ressembler à celui-ci :

Code : C++

```
// Vertices temporaires

float verticesTmp[] = {-taille, -taille, -taille,    taille, -taille,
-taille,    taille, taille, -taille,      // Face 1
                    -taille, -taille, -taille,    -taille, taille,
-taille,    taille, taille, -taille,      // Face 1
                    taille, -taille, taille,    taille, -taille,
-taille,    taille, taille, -taille,      // Face 2
                    taille, -taille, taille,    taille, taille,
taille,    taille, taille, -taille,      // Face 2
                    -taille, -taille, taille,    taille, -taille,
taille,    taille, -taille, -taille,      // Face 3
                    -taille, -taille, taille,    -taille, -taille,
-taille,    taille, -taille, -taille,      // Face 3
                    -taille, -taille, taille,    taille, -taille,
taille,    taille, taille, taille,      // Face 4
                    -taille, -taille, taille,    -taille, taille,
```

```

taille,    taille, taille, taille,           // Face 4
          -taille, -taille, -taille, -taille,
-taille, taille,  -taille, taille, taille, // Face 5
          -taille, -taille, -taille, -taille, taille,
-taille,  -taille, taille, taille,        // Face 5
          -taille, taille, taille, taille, taille,
taille,  taille, taille, -taille,         // Face 6
          -taille, taille, taille, -taille, taille,
-taille,  taille, taille, -taille};       // Face 6

```

Il reste encore une petite modification à faire. Si on regarde de plus près nos données, on remarque que les vertices vont de **-taille** à **+taille**. Cet intervalle fait que notre cube est multiplié par 2. 😕

Pour éviter cela, il faut diviser le paramètre **taille** par 2 avant de remplir le tableau. Ainsi, notre cube qui devait être multiplié par 2 ne le sera plus :

Code : C++

```

// Division du paramètre taille

taille /= 2;

// Vertices temporaires

float verticesTmp[] = {-taille, -taille, -taille,      taille, -taille,
                       -taille,  taille, taille, -taille, // Face 1
                           -taille, -taille, -taille, -taille, taille,
-taille,  taille, taille, -taille,        // Face 1
                           taille, -taille, taille,  taille, -taille,
-taille,  taille, taille, -taille, // Face 2
                           taille, -taille, taille,  taille, -taille,
taille,  taille, taille, -taille,        // Face 2
                           -taille, -taille, taille,  taille, -taille,
taille,  taille, -taille, -taille, // Face 3
                           -taille, -taille, taille,  -taille, -taille,
-taille,  taille, -taille, -taille, // Face 3
                           -taille, -taille, taille,  taille, -taille,
taille,  taille, taille, -taille, // Face 4
                           -taille, -taille, taille,  -taille, taille,
taille,  taille, taille, -taille,        // Face 4
                           -taille, -taille, -taille, -taille,
-taille,  taille, -taille, taille, // Face 5
                           -taille, -taille, -taille, -taille, taille,
-taille,  -taille, taille, taille, // Face 5
                           -taille, taille, taille,  taille, -taille,
taille,  taille, -taille, -taille, // Face 6
                           -taille, taille, taille,  -taille, taille,
-taille,  taille, -taille, -taille}; // Face 6

```

Piouf le plus dur est derrière nous. 😕

Le tableau de couleurs quant à lui est plus simple à faire puisqu'il suffit juste de reprendre celui que nous utilisions avant. Nous modifierons juste son nom en l'appelant **couleursTmp** vu qu'il s'agit de données temporaires :

Code : C++

```
// Couleurs temporaires

float couleursTmp[] = {1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, // Face 1
1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, // Face 1
0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0, // Face 2
0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0, // Face 2
0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0, // Face 3
0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0, // Face 3
1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, // Face 4
1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, // Face 4
0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0, // Face 5
0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
0.0, // Face 5
0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0, // Face 6
0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0}; // Face 6
```

Maintenant que nos données sont déclarées, il ne manque plus qu'à les transférer dans nos attributs.

Pour cela, nous allons utiliser une boucle qui va s'exécuter **108** fois, ce qui permettra donc de copier non seulement les coordonnées des sommets mais aussi les composantes des couleurs. En effet, les tableaux font tous les deux la même taille, on peut alors n'utiliser qu'une seule boucle. 😊

Code : C++

```
// Copie des valeurs dans les tableaux finaux

for(int i(0); i < 108; i++)
{
    m_vertices[i] = verticesTmp[i];
    m_couleurs[i] = couleursTmp[i];
}
```

Si on réunit tous les bouts de code :

Code : C++

```
Cube::Cube(float taille, std::string const vertexShader, std::string
const fragmentShader) : m_shader(vertexShader, fragmentShader)
{
    // Chargement du shader
    m_shader.charger();
```

```
// Division de la taille
taille /= 2;

// Vertices temporaires

float verticesTmp[] = {-taille, -taille, -taille,  taille,
-taille, -taille,  taille,  taille, // Face 1
                     -taille, -taille, -taille, -taille,
taille, -taille,  taille,  taille, -taille, // Face 1

                     taille, -taille, taille,  taille,
-taille, -taille,  taille,  taille, // Face 2
                     taille, -taille, taille,  taille,
taille, taille,  taille,  taille, -taille, // Face 2

                     -taille, -taille, taille,  taille,
-taille, taille,  taille, -taille, // Face 3
                     -taille, -taille, taille, -taille,
-taille, -taille,  taille, -taille, // Face 3

                     -taille, -taille, taille,  taille,
-taille, taille,  taille,  taille, // Face 4
                     -taille, -taille, taille, -taille,
taille, taille,  taille,  taille, // Face 4

                     -taille, -taille, -taille, -taille,
-taille, taille,  taille,  taille, // Face 5
                     -taille, -taille, -taille, -taille,
taille, -taille,  -taille,  taille, // Face 5

                     -taille, taille,  taille,  taille,
taille, taille,  taille, -taille, // Face 6
                     -taille, taille,  taille, -taille,
taille, -taille,  taille, -taille}; // Face 6

// Couleurs temporaires

float couleursTmp[] = {1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0,
0.0, 0.0,           // Face 1
                     1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0,
0.0, 0.0,           // Face 1

                     0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0,
1.0, 0.0,           // Face 2
                     0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0,
1.0, 0.0,           // Face 2

                     0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0,
0.0, 1.0,           // Face 3
                     0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0,
0.0, 1.0,           // Face 3

                     1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0,
0.0, 0.0,           // Face 4
                     1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0,
0.0, 0.0,           // Face 4

                     0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0,
1.0, 0.0,           // Face 5
                     0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0,
1.0, 0.0,           // Face 5

                     0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0,
0.0, 1.0,           // Face 6
                     0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0,
0.0, 1.0}; // Face 6
```

```
// Copie des valeurs dans les tableaux finaux

for(int i(0); i < 108; i++)
{
    m_vertices[i] = vertigesTmp[i];
    m_couleurs[i] = couleursTmp[i];
}
```

Le destructeur

Comme vous le savez déjà, un *destructeur* est une méthode appelée au moment de la *destruction* de l'objet. Il permet de libérer la mémoire prise par l'objet au cours de sa vie, en particulier la mémoire allouée dynamiquement.

Heureusement pour nous, dans notre cas nous ne faisons aucune allocation dynamique. 🎉 Le destructeur va donc être vide :

Code : C++

```
Cube::~Cube ()
{
```



L'objet de type **Shader** sera désalloué automatiquement vu que c'est un objet à part, inutile de tenter de le faire manuellement.

La méthode afficher

Comme son nom l'indique, la méthode **afficher()** va nous permettre ... d'afficher notre cube. 😊 Son prototype est assez simple :

Code : C++

```
void afficher(glm::mat4 &projection, glm::mat4 &modelview);
```

Elle prend en paramètre une référence sur les deux matrices que l'on connaît si bien maintenant. Elles sont indispensables pour afficher un modèle 3D. Dans cette méthode, nous en avons besoin pour les envoyer au shader.

Son implémentation va être ultra simple pour nous : il suffit juste de copier le code contenu entre l'activation et la désactivation du shader. Ce qui comprend :

- Le shader évidemment
- L'envoi des matrices
- L'utilisation des tableaux **Vertex Attrib**
- L'appel à la fonction **glUseProgram()**

Le code à copier est le suivant :

Code : C++

```
// Activation du shader
glUseProgram(shaderCouleur.getProgramID());

// Envoi des vertices
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);
 glEnableVertexAttribArray(0);

// Envoi de la couleur
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, couleurs);
 glEnableVertexAttribArray(1);

// Envoi des matrices
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
 "projection"), 1, GL_FALSE, value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shaderCouleur.getProgramID(),
 "modelview"), 1, GL_FALSE, value_ptr(modelview));

// Rendu
glDrawArrays(GL_TRIANGLES, 0, 36);

// Désactivation des tableaux
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);

// Désactivation du shader
glUseProgram(0);
```

Avant d'aller plus loin, il nous faut modifier le nom des variables anciennement utilisées. Ainsi :

- L'objet **shaderCouleur** devient **m_shader**
- Le tableau **vertices** devient **m_vertices**
- Le tableau **couleurs** devient **m_couleurs**

Une fois le nom des variables modifié, on se retrouve avec la méthode **afficher()** suivante :

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Envoi des vertices
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
 m_vertices);
    glEnableVertexAttribArray(0);
```

```

    // Envoi de la couleur

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
m_couleurs);
    glEnableVertexAttribArray(1);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));

    // Rendu

    glDrawArrays(GL_TRIANGLES, 0, 36);

    // Désactivation des tableaux

    glDisableVertexAttribArray(1);
    glDisableVertexAttribArray(0);

    // Désactivation du shader

    glUseProgram(0);
}

```



N'oubliez pas d'inclure la ligne "**using namespace glm;**" au début de votre fichier **.cpp** de façon à pouvoir vous passer du préfixe **glm::**:

J'adore quand le copier-coller fonctionne aussi facilement. 😊 Nous avions déjà fait le plus gros avant, il ne nous restait plus qu'à adapter le nom des attributs.

La boucle principale

Il ne reste plus qu'une seule chose à faire : déclarer un objet de type **Cube** et utiliser sa méthode **afficher()** dans la boucle principale. On efface donc tout ce qu'on a fait avant (vertices, couleurs, affichage, ...). On ne doit garder que ceci :

Code : C++

```

void SceneOpenGL::bouclePrincipale()
{
    // Variable

    bool terminer(false);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

```

```

// Boucle principale

while(!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&m_evenements);

    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra

    modelview = lookAt(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1,
0));

    // Rendu (Rien pour le moment)

    . . .

    // Actualisation de la fenêtre

    SDL_GL_SwapWindow(m_fenetre);
}
}

```



Pensez dès maintenant à ajouter le header "**Cube.h**" dans la classe *SceneOpenGL*.

Ensuite, on déclare notre objet de type **Cube** qui sera initialisé automatiquement avec le constructeur. Nous lui donnerons la valeur **2.0** pour le paramètre **taille** (ou une autre qui vous plaira 😊) ainsi que les *string* "**Shaders/couleur3D.vert**" et "**Shaders/couleur3D.frag**" pour le shader.

Code : C++

```

void SceneOpenGL::bouclePrincipale()
{
    // Variable

    bool terminer(false);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Déclaration d'un objet Cube

```

```
Cube cube(2.0, "Shaders/couleur3D.vert",
"Shaders/couleur3D.frag");

// Boucle principale

while(!terminer)
{
    ...
}
```

Enfin, on utilise la méthode **afficher()** dans la boucle principale en donnant les matrices en paramètres :

Code : C++

```
while(!terminer)
{
    // Gestion des évènements

    SDL_WaitEvent(&m_evenements);

    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;

    // Nettoyage de l'écran

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra

    modelview = lookAt(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0));

    // Affichage du cube

    cube.afficher(projection, modelview);

    // Actualisation de la fenêtre

    SDL_GL_SwapWindow(m_fenetre);
}
```

Si vous compilez tout ça, vous obtiendrez le même résultat que tout à l'heure mais cette fois-ci, vous avez un véritable objet C++ permettant d'afficher un cube. 😊

Pseudo-animation

Attention, cette sous-partie s'appelle bien "**pseudo-animation**" et pas **animation** tout court. On va apprendre à faire pivoter notre cube pour voir toutes ses faces, et ce grâce à la méthode **rotate()** de la librairie **GLM**.

Le principe est simple : on incrémente un angle à chaque tour de boucle puis on fait pivoter le cube avec cet angle qui change sans arrêt donnant ainsi une impression de mouvement. Pour le moment, il faudra bouger la souris dans la fenêtre SDL pour constater la rotation puisque nous utilisons la fonction **SDL_WaitEvent()** qui bloque le programme quand il n'y a pas d'événements.

On commence par déclarer un angle de type **float** que l'on incrémentera à chaque tour boucle :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    ...

    // Variable angle
    float angle(0.0);

    // Boucle principale
    while (!terminer)
    {
        ...
    }
}
```

Ensuite, on incrémente l'angle de rotation à chaque tour de boucle. Petite précision, l'angle atteindra forcément les 360° vu qu'on l'incrémentera sans arrêt. A chaque fois qu'il atteindra 360° , il faudra donc le remettre à zéro. Il est inutile d'avoir un angle incompréhensible de 1604° . 😊

Code : C++

```
// Incrémentation de l'angle
angle += 4.0;

if (angle >= 360.0)
    angle -= 360.0;
```

Une fois l'angle défini, il suffit de faire pivoter le repère en appelant la méthode **rotate()** de la matrice **modelview** :

Code : C++

```
// Rotation du repère
modelview = rotate(modelview, angle, vec3(0, 1, 0));
```

Si on place ce code au bon endroit :

Code : C++

```
while (!terminer)
{
    // Gestion des évènements
    ...

    // Placement de la caméra
    modelview = lookAt(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0));
```

```
// Incrémentation de l'angle
angle += 4.0;

if(angle >= 360.0)
    angle -= 360.0;

// Rotation du repère
modelview = rotate(modelview, angle, vec3(0, 1, 0));

// Affichage du cube
cube.afficher(projection, modelview);

// Fin de la boucle
...
}
```

Avec ce code, votre cube devrait tourner sur lui-même. 🎉

Le Frame Rate Calcul du FPS

Je profite de ce chapitre pour vous introduire une nouvelle notion : celle du **Frame Rate**.

Le **Frame Rate** est le nombre de fois par seconde où la boucle principale est exécutée. En France, on prend généralement une valeur 50 fps (**Frames Per Second** ou **Frames par seconde**). Pour le moment, notre jeu fonctionne avec 0 fps étant donné que l'on utilise une fonction qui bloque le programme : **SDL_WaitEvent()**. La boucle ne s'exécute que si on fait quelque chose, sinon le jeu est bloqué.

Or, dans un jeu-vidéo, si on ne touche pas à la souris il se passe quand même quelque chose. Nous allons régler ce problème en introduisant la notion de frame rate. La première chose à faire est de changer la fonction **SDL_WaitEvent()** par la fonction **SDL_PollEvent()** qui, elle, ne bloque pas le programme :

Code : C++

```
SDL_PollEvent(&m_evenements);
```

L'utilisation de cette fonction va cependant nous poser un problème : le CPU va être totalement surchargé. Heureusement, le frame rate est là pour nous aider. Grâce à lui, nous n'exécuterons pas la boucle principale des centaines de fois par seconde mais uniquement quelques dizaines de fois, ce qui est pour le CPU largement calculable. 😊 Pour imposer cette limitation, il suffit de bloquer le programme quelques millisecondes à un certain moment. Ce petit intervalle dépend du nombre de FPS que l'on souhaite afficher.

Pour calculer ce temps de blocage, il suffit de diviser 1000 millisecondes (soit une seconde) par le nombre de FPS que l'on veut. Par exemple, si on veut 50 FPS il faut faire : $1000 / 50 = 20$ millisecondes.

Pour 50 FPS, la boucle principale devra mettre 20 millisecondes à s'exécuter, quitte à mettre le programme en pause jusqu'à atteindre cet intervalle de 20 ms.

Programmation

La théorie c'est bien mais il faut maintenant adapter la notion de FPS dans notre code. Le principe est simple : on calcule le temps écoulé entre le début et la fin de la boucle. Si ce temps est inférieur à 20 ms alors on met en pause le programme jusqu'à atteindre les 20 ms.

L'implémentation se fait en plusieurs étapes :

- À chaque tour de boucle, on enregistre le temps où on commence la boucle.
- Puis on enregistre le temps une fois qu'elle est terminée.
- On soustrait le temps enregistré au début par le temps de fin de boucle.
- Si ce temps est inférieur à 20 ms, alors on met en pause le programme jusqu'à atteindre 20 ms.

Pour capturer le temps, on utilise une fonction de la SDL : **SDL_GetTicks()**. Elle retourne le temps actuel de l'ordinateur dans une structure de type **Uint32** :

Code : C++

```
Uint32 SDL_GetTicks(void);
```

Une autre fonction qui va nous être utile est la fonction **SDL_Delay()**. Cette fonction va nous permettre de mettre en pause le programme lorsque nous en aurons besoin. Elle prend un seul paramètre : le temps en millisecondes durant lequel elle va bloquer le programme :

Code : C++

```
void SDL_Delay(Uint32 ms);
```

Bref, commençons déjà par déclarer nos variables juste après le booléen **terminer** de la méthode **bouclePrincipale()** :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables relatives à la boucle

    bool terminer(false);
    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    /* ***** Reste du code **** */
}
```

N'oubliez pas que le temps de blocage est égal à : **1000 / Frame Rate**.

Occupons-nous maintenant de la limitation. La première chose à faire est de déterminer le temps où la boucle commence :

Code : C++

```
// Boucle principale

while (!terminer)
{
    // On définit le temps de début de boucle

    debutBoucle = SDL_GetTicks();
```

```
    /* ***** Boucle Principale ***** */  
}
```

Ensuite, il faut déterminer le temps qu'a mis la boucle pour s'exécuter. Pour ça, on enregistre le temps de fin de boucle que l'on soustrait par le temps du début :

Code : C++

```
while (!terminer)  
{  
    /* ***** Boucle Principale ***** */  
  
    // Calcul du temps écoulé  
  
    finBoucle = SDL_GetTicks();  
    tempsEcoule = finBoucle - debutBoucle;  
}
```

Enfin, si le temps est inférieur à 20 ms, on met en pause le programme jusqu'à atteindre les 20 ms :

Code : C++

```
// Si nécessaire, on met en pause le programme  
  
if (tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);
```

Si on résume tout ça :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Variables relatives à la boucle  
  
    bool terminer(false);  
    unsigned int frameRate (1000 / 50);  
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);  
  
    // Boucle principale  
  
    while (!terminer)  
    {  
        // On définit le temps de début de boucle  
  
        debutBoucle = SDL_GetTicks();  
  
        // Gestion des évènements  
  
        SDL_PollEvent(&m_evenements);  
  
        if (m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)  
            terminer = true;
```

```
/* ***** Rendu OpenGL ***** */  
  
// Calcul du temps écoulé  
  
finBoucle = SDL_GetTicks();  
tempsEcoule = finBoucle - debutBoucle;  
  
// Si nécessaire, on met en pause le programme  
  
if(tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);  
}  
}
```

Avec ce code, la boucle principale ne s'exécutera que 50 fois par seconde. Si vous observez l'activité de votre processeur, vous remarquerez que le programme ne le monopolise pas malgré l'utilisation de la fonction **SDL_PollEvent()**. Maintenant, nous ne sommes plus obligés de toucher la souris pour voir quelque chose bouger à l'écran. Essayez avec votre cube, il tourne même si vous ne faites rien. 😊

Télécharger : [Code Source C++ du Chapitre 7](#)

Et voilà, nous avons fait nos premiers pas dans la 3D. Nous avons vu les principes de base et nous sommes maintenant capables d'afficher des polygones en 3 dimensions avec OpenGL. 😊

Dans le chapitre suivant, nous allons nous reposer un peu et étudier quelques points divers dont je n'ai pas encore parlés. Ce chapitre sera un peu plus court que les autres mais tout aussi important. 😊

Push et Pop

Les chapitres les plus compliqués de la partie 1 sont enfin derrière nous, les suivants le seront moins. Aujourd'hui, on va s'intéresser aux piles de matrices.

Retenez bien ce que nous allons voir, et profitez de ce chapitre assez *soft* pour vous reposer. 

La problématique Introduction

Je profite de cette partie vous faire un aparté sur la matrice **modelview**. En effet, pour le moment il y a un petit problème avec notre façon de l'utiliser, nous allons étudier un cas pour voir ce qui ne va pas. Jusqu'à maintenant, nous ne nous en sommes pas vraiment rendus compte car nos modèles 3D sont assez simples. Mais maintenant que nous avons fait nos premiers pas dans la 3D, il vaut mieux prendre les bonnes habitudes dès le début. 

Un problème bien ennuyeux ...

Pour comprendre le problème, nous allons revenir un peu sur les transformations. Vous savez maintenant que chaque transformation que vous faites sera appliquée sur le repère de votre espace 3D et pas seulement sur l'objet que vous voulez afficher. Si vous faites une translation puis une rotation c'est le repère entier qui va être modifié.

Imaginez que vous voulez afficher un toit pour une maison. Il faudra tout d'abord se placer en haut de la maison, puis faire une rotation pour que le toit soit légèrement penché et afficher le tout.

Maintenant si vous voulez afficher le jardin, comment faites-vous pour revenir en bas de la maison ? Comment faites-vous pour annuler la translation et la rotation qui permettaient d'arriver jusqu'au toit ?

Pour ça, nous avons trois solutions :

- Refaire la rotation et la translation dans le sens inverse.
- Réinitialiser totalement la matrice **modelview**, puis la caméra et enfin se replacer en bas de la maison.
- Sauvegarder la matrice, afficher le toit en la modifiant, puis annuler les transformations en la restaurant à son état sauvegardé.

La première solution peut vous paraître la plus simple à mettre en œuvre et sur un exemple comme celui-ci on peut le penser. Mais imaginez une maison de 500 vertices, l'afficher nécessitera plusieurs transformations. Vous pensez vraiment refaire toutes ces transformations dans le sens inverse pour retrouver votre position ?

La réponse est bien sûr *non*, d'une part parce que se rappeler de toutes les transformations est trop fastidieux et d'autre part parce que ça vous coutera trop de ressources pour refaire tout dans le sens inverse. Surtout que votre maison sera affichée 50 fois par seconde, imaginez le fourbi. 

La deuxième solution nous pose les mêmes problèmes, réinitialiser la matrice et se replacer nous fait perdre du temps et des ressources.

Nous utiliserons donc la troisième solution qui nous permet de sauvegarder la matrice quand nous en avons besoin. De cette façon, on peut modifier le repère à volonté sans être obligé de retenir sur toutes les transformations. Une fois que nous voudrons revenir à la position sauvegardée, il suffira simplement de restaurer la sauvegarde du repère. 

Les piles

Avec OpenGL 2

Dans les précédentes versions d'OpenGL, il existait deux fonctions qui permettaient de sauvegarder et de restaurer les matrices facilement. Celles-ci s'appelaient :

- **glPush()** : pour la sauvegarde
- **glPop()** : pour la restauration

Ces deux fonctions fonctionnaient sur un système de **pile** qui permettait de stocker les sauvegardes les unes sur les autres.

Le principe d'une pile en programmation est d'entasser des variables ou des objets de même type les uns sur les autres comme une pile d'assiettes. Le but de la fonction **glPush()** était justement d'empiler des matrices entre elles pour former une pile de matrices :



On utilisait généralement les piles sur la matrice ***modelview***, étant donné que c'est elle la plus utilisée. Elles étaient basées sur le principe **LIFO (Last In First Out)**, littéralement sur le principe du dernier arrivé premier sorti. C'est-à-dire que la dernière sauvegarde était la première restaurée.

Prenons l'exemple des assiettes. Si vous empilez 18 assiettes et que vous voulez laver celle la plus en dessous. Il faudra d'abord laver les 17 premières assiettes qui sont au dessus. En revanche, si vous voulez laver la dernière arrivée, elle se trouvera en haut de la pile (l'endroit le plus simple à accéder).



Ok on sait ce qu'est une pile maintenant. 😊 Mais à quoi ça pouvait bien servir ?

Bonne question, les piles permettaient de sauvegarder la matrice ***modelview*** à un état donné pour pouvoir la restaurer plus tard. L'intérêt principal était de pouvoir empiler des matrices pour avoir un système de restauration assez simple : la dernière sauvegarde était la première restaurée.

De cette façon, nous n'avions plus besoin de réinitialiser la matrice ***modelview*** ou de retenir toutes les transformations pour

savoir où se trouvait le repère.

Avec OpenGL 3

Comme beaucoup d'autres fonctions, OpenGL a déprécié l'utilisation de `glPush()` et `glPop()` mais bon il n'y a rien de surprenant là-dedans. En revanche, ce qui est surprenant c'est que la librairie **GLM** n'inclut pas de méthodes de substitution à ces fonctions. Nous ne pouvons donc pas utiliser les piles de matrices.

Ceci est dû au fait que les matrices sont maintenant des objets au sens propre du terme. C'est-à-dire que nous pouvons les manipuler, utiliser l'allocation dynamique dessus, intégrer la **POO**, etc. L'usage des piles n'est donc plus utile. C'est assez perturbant pour ceux qui les ont toujours utilisées mais lorsque l'on connaît la puissance du C++, on se rend compte que ce n'est pas une si mauvaise idée que ça. 😊

Au final, nous allons utiliser une autre manière de faire pour sauvegarder nos matrices. Les piles ne sont plus indispensables mais les sauvegardes, elles, le sont toujours. On ne peut pas résoudre notre problème de tout sinon. 🤪

Sauvegarde et Restauration

Substitution aux piles

Comme nous l'avons vu à l'instant, les piles ne sont maintenant inutilisables, cependant les sauvegardes doivent quand même être faites.

Pour les faire, nous allons utiliser une des propriétés magiques du C++ : l'opérateur `=`. En effet, lorsque cet opérateur est surchargé, il permet de pouvoir copier un objet dans un autre objet. C'est ainsi que l'on peut, par exemple, copier deux voitures sans problème :

Code : C++

```
// Copie d'une voiture  
Voiture maCopie = voitureOriginale;
```



Sous réserve que la classe **Voiture** inclut la surcharge de l'opérateur `=`. 😊

Pour notre problème de matrices, nous allons faire exactement la même chose :

- Pour la **sauvegarde** : nous allons copier une matrice dans un objet **sauvegarde**
- Pour la **restauration** : nous allons faire l'inverse et copier la **sauvegarde** dans la matrice originale

En code, cela donnerait :

Code : C++

```
// Sauvegarde de la matrice  
mat4 sauvegardeModelview = modelview;  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;
```

Facile non ? Je dirais même que c'est plus facile à comprendre que les piles. 😊

Quand sauvegarder ?

Le problème des transformations

Maintenant que nous avons une méthode de substitution aux piles, nous allons pouvoir sauvegarder nos matrices dans nos programmes. En théorie, nous devrions faire cela à chaque fois que l'on fait une transformation. Par exemple, le cube du chapitre du chapitre précédent utilise une transformation, et plus précisément une rotation, qui nous permet de faire une pseudo-animation. Nous devons donc utiliser la sauvegarde de matrice ici.

Pourquoi me direz-vous ? Simplement parce que le prochain modèle que nous voudrons afficher (comme un autre cube) sera automatiquement affecté par la rotation. Ce qui fait qu'au lieu de faire pivoter le cube initial, nous les ferons pivoter tous les deux. Essayez ce code pour voir :

Code : C++

```
// Rotation du repère  
  
modelview = rotate(modelview, angle, vec3(0, 1, 0));  
  
// Affichage du premier cube  
  
cube.afficher(projection, modelview);  
  
// Affichage du second cube un peu plus loin  
  
modelview = translate(modelview, vec3(10, 0, 0));  
cube.afficher(projection, modelview);
```

Reculez votre caméra avant en la plaçant au point de coordonnées suivantes :

Code : C++

```
// Placement de la caméra  
  
modelview = lookAt(vec3(6, 6, 6), vec3(3, 0, 0), vec3(0, 1, 0));
```

Compilez pour voir.

Vous verrez que les deux cubes sont affectés par la rotation.

Dans un jeu-vidéo, ça serait assez problématique si une simple rotation de caméra faisait pivoter tout un bâtiment. 🎬

Pour éviter ça, il faut sauvegarder l'état de la matrice **modelview** avant la rotation, puis la restaurer une fois le modèle affiché :

Code : C++

```
// Sauvegarde de la matrice modelview  
  
mat4 sauvegardeModelview = modelview;  
  
// Rotation du repère
```

```
modelview = rotate(modelview, angle, vec3(0, 1, 0));  
  
// Affichage du premier cube  
cube.afficher(projection, modelview);  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;  
  
// Affichage du second cube plus loin  
modelview = translate(modelview, vec3(10, 0, 0));  
cube.afficher(projection, modelview);
```

Si vous essayez ce code, vous remarquerez que le second cube ne pivote plus. La rotation n'est valable que pour le premier car la matrice avait été sauvegardée avant.

Un autre exemple

Bien entendu, le système de sauvegarde/restauration ne doit pas être utilisé seulement pour les rotations mais bien pour toutes les transformations (translation et homothétie comprises).

Ce qui fait que le code précédent est encore incomplet car la translation du second cube affectera les objets affichés après. Il faut donc réutiliser la sauvegarde de matrice :

Code : C++

```
// Affichage du premier cube  
....  
  
// Sauvegarde de la matrice modelview  
mat4 sauvegardeModelview = modelview;  
  
// Affichage du second cube plus loin  
modelview = translate(modelview, vec3(2, 0, 0));  
cube.afficher(projection, modelview);  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;
```

Le cas des objets proches

Ce système de sauvegarde nous permet de revenir à chaque fois au centre du repère. Nos modèles sont donc placés sans erreur vu que nous partons toujours du point de coordonnées (0, 0, 0). Aucune transformation ne les affecte.

Le seul cas où vous pourrez vous permettre de ne pas restaurer la matrice immédiatement est le cas où des objets se situeraient près les uns des autres. Par exemple, imaginez que votre cube devienne une caisse et que vous souhaitez en afficher plusieurs

côte à côte. Vous n'allez pas sauvegarder votre matrice à chaque fois pour revenir quasiment au même point après.

Si elles sont assez proches, vous pourrez les afficher les unes à la suite des autres sans problème et ce même si vous utilisez d'autres transformations :

Code : C++

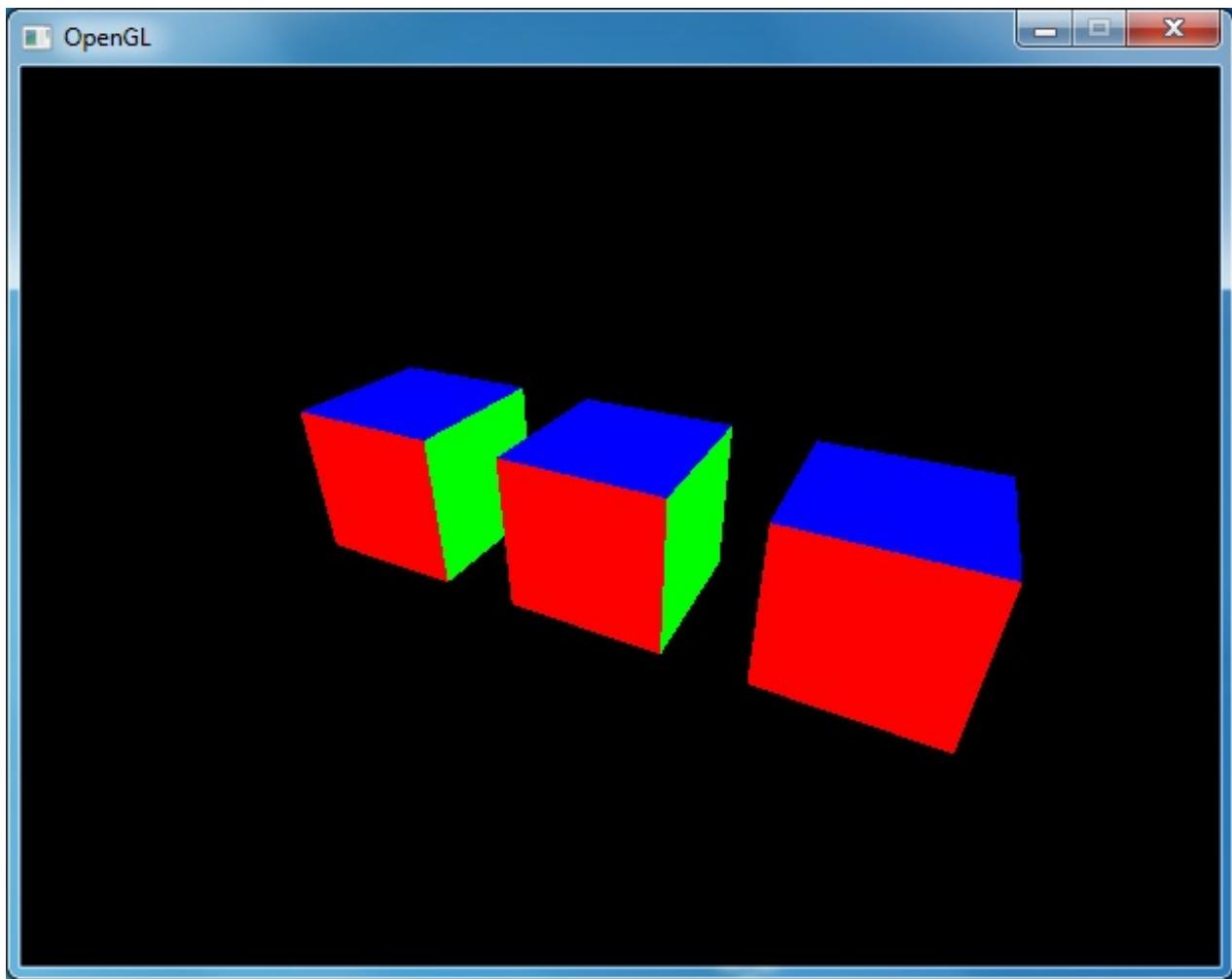
```
// Sauvegarde de la matrice modelview
mat4 sauvegardeModelview = modelview;

// Affichage du premier cube (au centre du repère)
cube.afficher(projection, modelview);

// Affichage du deuxième cube
modelview = translate(modelview, vec3(3, 0, 0));
cube.afficher(projection, modelview);

// Affichage du troisième cube
modelview = translate(modelview, vec3(3, 0, 0));
cube.afficher(projection, modelview);

// Restauration de la matrice
modelview = sauvegardeModelview;
```



Vous voyez ici que les cubes sont assez proches, il est donc inutile de revenir au centre du repère pour repartir afficher la caisse suivante.

Ce qu'il faut retenir

Pour résumer tout ce blabla un peu confus :

- Vous devez sauvegarder et restaurer la matrice **modelview** à chaque fois que vous faites une transformation
- Si vous affichez des objets assez proches, ne restaurez votre matrice qu'une fois toutes les transformations faites. Ceci afin d'économiser un peu de temps de calcul
- Si vous ne faites pas de transformation, ne sauvegardez rien

C'est tout ce qu'il faut retenir dans ce chapitre. 😊

Télécharger : [Code Source C++ du Chapitre 8](#)

Nous voici à la fin de ce chapitre. Nous avons appris à coder et à utiliser les sauvegardes de matrice. C'était un chapitre un peu technique mais il concernait une notion indispensable de la programmation avec OpenGL. N'oubliez pas qu'à partir de maintenant, nous utiliserons toujours les piles de matrices pour afficher nos modèles. 😊

Pour la suite du tuto je vous propose de vous reposer encore un peu, le prochain chapitre que nous allons aborder sera assez facile à comprendre. Il concerne les évènements avec la SDL 2.0. 😊

Les évènements avec la SDL 2.0

Depuis le début du tutoriel, les évènements sont gérés automatiquement et sans contrôle sur le clavier et la souris. Nous pourrions continuer à coder ainsi, sans se préoccuper de rien mais on va profiter de ce petit chapitre pour créer une classe à part entière qui s'occupera de gérer tout ça pour nous. Avec elle, nous pourrons retrouver facilement les évènements qui nous intéressent comme les touches ou les boutons de souris qui sont utilisés, ...

Bien entendu, notre code devra respecter la règle de l'encapsulation. Il faudra donc faire attention à protéger les attributs pour ne pas accéder directement aux évènements.

Différences entre la SDL 1.2 et 2.0

Nous avons déjà eu l'occasion de constater les différences qu'ils existaient entre l'ancienne et la nouvelle version de la SDL, notamment lors de la création de fenêtre. Nous allons voir maintenant celles qui concernent les évènements car ils sont indispensables au développement d'un jeu vidéo. D'ailleurs, on les a déjà utilisés lorsque nous voulions savoir si une fenêtre devait se fermer ou pas.

En effet, à chaque tour de la boucle principale, on vérifie si l'évènement **SDL_WINDOWEVENT_CLOSE** est déclenché, ce qui nous permet de continuer ou d'arrêter la boucle :

Code : C++

```
// Boucle principale

bool terminer(false);

while(!terminer)
{
    // Gestion des évènements

    SDL_PollEvent(&m_evenements);

    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        terminer = true;
}
```

Ce bout de code est pratique mais il ne nous permet pas de gérer la pression des touches. On pourrait par exemple appuyer sur la touche **ECHAP** pour terminer la boucle, ou utiliser un de souris.

Si vous rappelez du chapitre sur les évènements de la SDL 1.2 dans le [cours](#) de M@téo, vous vous souvenez sûrement de la façon de gérer les touches du clavier. Par exemple, pour gérer la pression des touches T ou ECHAP il fallait faire ceci :

Code : C++

```
// Structure

SDL_Event evenements;

// Récupération d'un évènement

SDL_PollEvent(&evenements);

// Switch sur le type d'évènement

switch(evenements.type)
{
    case SDL_KEYDOWN:

        // Gestion des touches

        switch(evenements.key.keysym.sym)
        {
            case SDLK_T:
```

```
    ...
    break;

    case SDLK_ESCAPE:
    ...
    break;
}

break;
}
```

Voilà comment on gérait les évènements avec la SDL 1.2, et la bonne nouvelle c'est qu'avec la version 2.0 ça se passe de la même façon. 😊

Seulement deux choses vont être modifiées :

- Premièrement, on ne vérifie plus le champ *sym*, mais le champ *scancode*. Ce qui donne au final : *m_evenements.key.keysym.scancode*.
- Ensuite, le *nom* des touches ne commencent plus par *SDLK_** mais par *SDL_SCANCODE_**. Ce qui nous donne ici : *SDL_SCANCODE_T* et *SDL_SCANCODE_ESCAPE*.

Si on modifie le code précédent on trouve :

Code : C++

```
// Structure

SDL_Event evenements;

// Récupération d'un évènement

SDL_PollEvent (&evenements);

// Switch sur le type d'évènement

switch (evenements.type)
{
    case SDL_KEYDOWN:

        // Gestion des touches

        switch (evenements.key.keysym.scancode)
        {
            case SDL_SCANCODE_T:
            ...
            break;

            case SDL_SCANCODE_ESCAPE:
            ...
            break;
        }

    break;
}
```

Aucun gros changement. 😊

Je ne peux vous parler plus explicitement des *scancodes* mais sachez simplement que mis à part le nom des constantes, le code restera le même pour nous. Si on veut aller plus loin au niveau du fonctionnement de la SDL on remarque que les *scancodes* sont

une modification majeur de la librairie car on change de norme pour identifier les touches en interne. Mais bon dans le fond on s'en moque un peu. 😊

Concernant la souris, il n'y a rien à signaler. Il n'y a pas de grands changements par rapport à la SDL 1.2. La gestion se fera de la même façon pour nous.

La classe Input

La classe Input

Maintenant que nous avons vu les différences qu'il existait entre les deux versions de la SDL, nous allons passer à l'implémentation d'une classe qui gèrera tous les événements toute seule. 😊



Ça veut dire qu'on va remplacer la structure **SDL_Event** par une **classe** ?

Alors oui et non, car la structure **SDL_Event** existera toujours quelque part. Mais au lieu de l'utiliser dans la classe **SceneOpenGL**, on l'utilisera dans une classe à part. Grâce à ça, les événements seront protégés par la règle de *l'encapsulation* et seront aussi plus simples à utiliser. Seul l'appel à une méthode nous permettra de savoir si telle ou telle touche est enfoncee. 😊

On appellera cette classe : la classe **Input**.

En informatique, les *inputs* sont des systèmes qui permettent d'apporter à l'ordinateur des actions venant de l'utilisateur. Elles peuvent prendre la forme d'un mouvement de souris, d'une pression sur une touche du clavier, d'un geste de doigt sur un écran tactile, ... D'où le nom **Input** pour la classe qui gèrera tous ces événements. 😊

Voici sa déclaration :

Code : C++

```
#ifndef DEF_INPUT
#define DEF_INPUT

// Include

#include <SDL2/SDL.h>

// Classe

class Input
{
public:
    Input();
    ~Input();

private:
};

#endif
```

Les attributs

La classe **Input** possèdera plusieurs attributs :

- Une structure **SDL_Event** : pour récupérer les événements SDL
- Un tableau de booléens **m_touches[]** : regroupant toutes les touches du clavier
- Un tableau de booléens **m_boutonsSouris[]** : regroupant tous les boutons de la souris

- Deux *entiers* (*int*) : représentant la position (x, y) du pointeur de la souris
- Deux *entiers* : représentant la position relative (x, y) du pointeur
- Un *booléen* **m_terminer** : qui permettra de savoir quand la fenêtre devra être fermée

On va directement s'intéresser aux tableaux de booléens. Chaque case de ces tableaux pourra prendre deux valeurs : si une touche, ou un bouton, est enfoncé(e) elle prendra la valeur **true**, dans le cas contraire ce sera la valeur **false**. Au début du programme, toutes les cases sont initialisées à **false**, vu que l'on appuie sur aucune des touches.

Pour définir la taille de ces tableaux on pourrait croire qu'il faille utiliser l'allocation dynamique - vu qu'il existe plusieurs types de claviers - mais pas du tout. En effet, la *SDL* gère toute seule les différents claviers. Elle est capable de nous fournir une seule et unique constante (**SDL_NUM_SCANCODES**) représentant le nombre maximal de touches d'un clavier, et par conséquent la taille du tableau **m_touches[]**. Grâce à elle, on pourra gérer toutes les touches automatiquement. 😊

Pour la souris, ce sera un peu différent. Il n'existe pas de constante que l'on peut donner au tableau. En revanche, on peut savoir que la **SDL** ne peut gérer que **7 boutons** pour la souris. On donnera donc une taille de **7** cases au tableau **m_boutonsSouris[]**.



En réalité, on donnera une valeur de **8** car l'indice du premier bouton de la souris commence à **1** et pas à **0**. Il faut donc prévoir une case supplémentaire.

Si on regroupe les attributs on a :

Code : C++

```
#ifndef DEF_INPUT
#define DEF_INPUT

// Include

#include <SDL2/SDL.h>

// Classe

class Input
{
public:
    Input();
    ~Input();

private:
    SDL_Event m_evenements;
    bool m_touches[SDL_NUM_SCANCODES];
    bool m_boutonsSouris[8];

    int m_x;
    int m_y;
    int m_xRel;
    int m_yRel;

    bool m_terminer;
};

#endif
```

Avec ces attributs, on pourra récupérer tous les événements dont on aura besoin. On verra en dernière partie de ce chapitre quelques méthodes qui nous permettront d'accéder à ces attributs.

Le constructeur et le destructeur

Maintenant qu'on a défini la classe **Input**, on doit s'occuper d'initialiser ses attributs avec le *constructeur*. Dans cette classe, on aura besoin que d'un constructeur : la *constructeur par défaut*.

Code : C++

```
Input();
```

Il s'occupera de mettre tous les attributs soit à 0 soit à *false*.

Son implémentation commence par l'initialisation de tous les attributs (sauf les tableaux) :

Code : C++

```
Input::Input() : m_x(0), m_y(0), m_xRel(0), m_yRel(0),  
m_terminer(false)  
{  
}
```

Pour initialiser les tableaux de *bool*, il faudra juste faire une boucle pour affecter la valeur **false** aux différentes cases :

Code : C++

```
Input::Input() : m_x(0), m_y(0), m_xRel(0), m_yRel(0),  
m_terminer(false)  
{  
    // Initialisation du tableau m_touches[]  
  
    for(int i(0); i < SDL_NUM_SCANCODES; i++)  
        m_touches[i] = false;  
  
    // Initialisation du tableau m_boutonsSouris[]  
  
    for(int i(0); i < 8; i++)  
        m_boutonsSouris[i] = false;  
}
```

Voilà pour le constructeur. 😊

On profite également de cette partie pour créer le *destructeur*. Même si on ne met rien dedans, on l'implémente quand même. C'est une bonne habitude à prendre. 😊

Code : C++

```
Input::~Input()  
{  
}
```

La boucle d'évènements

Pour la partie qui va suivre, je vais vous demander toute votre attention. Nous allons voir la notion la plus importante du chapitre : la boucle d'évènements.



Euh c'est quoi cette boucle ? Ça a un rapport avec les évènements SDL ?

Oui tout à fait. Il s'agit d'une boucle qui récupère tous les évènements à un moment donné, quelque soit leur nature (touche de clavier, mouvement de souris, ...).

Si je vous parle de cette boucle, c'est parce qu'à l'heure actuelle nous avons un gros problème : nous ne sommes pas capables de gérer plusieurs évènements à la fois. Je vais vous donner un exemple pour que vous compreniez bien le problème.

Dans votre vie, vous avez probablement déjà joué à un jeu-vidéo, comme Mario par exemple. Les jeux Mario sont en général des jeux de plateformes où notre petit plombier doit sauter et avancer sur des plateformes, tuyaux et autres éléments de l'environnement.

La plupart du temps, sans même vous en rendre compte, vous appuyez sur plusieurs touches (ou boutons) en même temps. Par exemple, un personnage saute et avance en même temps, il se passe deux actions : sauter et avancer. On peut même rajouter des actions à ça : tirer, sprinter, ... Dans ce cas, le programme doit gérer 4 évènements en même temps.

A l'heure actuelle, notre problème c'est qu'avec la SDL nous ne pouvons gérer qu'un seul et unique évènement à la fois. Si nous programmions un Mario avec la SDL, le personnage ne pourrait même pas avancer et sauter en même, ce qui est un peu contraignant ... 😱



Le problème vient de la SDL alors ? Elle est un peu nulle cette librairie 😞

Non pas du tout, car ça ne vient pas vraiment d'elle mais de notre code. Vous vous souvenez du **switch** au début du chapitre ? Celui-ci :

Code : C++

```
// Structure

SDL_Event evenements;

// Attente d'un évènement

SDL_PollEvent(&evenements);

// Switch sur le type d'évènement

switch(evenements.type)
{
    case SDL_KEYDOWN:

        // Gestion des touches

        switch(evenements.key.keysym.scancode)
        {
            case SDL_SCANCODE_T:
                ...
                break;

            case SDL_SCANCODE_ESCAPE:
                ...
                break;
        }
}
```

```
    }  
  
    break;  
}
```

Avec ce code, on ne peut gérer qu'un seul évènement par tour de boucle OpenGL.

Pour régler ce problème, on va se servir d'une petite astuce de la fonction **SDL_PollEvent()**. En effet, si on s'intéresse à son prototype on peut remarquer une chose :

Code : C++

```
int SDL_PollEvent(SDL_Event *event);
```

La fonction **SDL_PollEvent()** renvoie une **valeur** (un **int**).

On ne s'est jamais servi de cette valeur (on ne savait même pas qu'elle existait d'ailleurs) et pourtant c'est elle qui va régler notre problème. Cet **integer** retourné peut prendre deux valeurs :

- Soit **1** : ce qui veut dire qu'il reste encore des évènements à capturer dans la file d'attente
- Soit **0** : ce qui veut dire qu'il n'y en a plus

Pour capturer tous les évènements, il suffit de piéger la fonction **SDL_PollEvent()** dans une boucle. Tant qu'il reste quelque chose à récupérer dans la file d'attente, la fonction retourne **1**, donc on continue de récupérer les évènements jusqu'à qu'il n'y en ait plus.

Grâce à cette astuce, on sera capable de gérer plusieurs touches/boutons en même temps. Le petit Mario de tout à l'heure pourra donc avancer, sauter, sprinter, tirer des boules de feu ... et tout ça en même temps. 😊

D'où la *boucle d'évènements* dont je vous ai parlée tout à l'heure. 🎉

Au niveau du code, il suffit d'enfermer la fonction **SDL_PollEvent()** dans une boucle **while**. Tant que la fonction retourne **1** on continue la boucle :

Code : C++

```
// Structure  
  
SDL_Event evenements;  
  
// Boucle d'évènements  
  
while(SDL_PollEvent(&evenements) == 1)  
{  
    // Switch sur le type d'évènement  
  
    switch(evenements.type)  
    {  
        case SDL_KEYDOWN:  
  
            // Gestion des touches  
  
            switch(evenements.key.keysym.scancode)  
            {  
                case SDL_SCANCODE_T:  
                    ...  
                break;  
  
                case SDL_SCANCODE_ESCAPE:  
                    ...  
                break;
```

```
        }

    break;
}

}
```

On peut même simplifier le **while** en enlevant la condition '**== 1**' :

Code : C++

```
// Boucle d'évènements

while(SDL_PollEvent(&evenements))
{
    ...
}
```

Vous savez maintenant ce qu'est la *boucle d'évènements*. C'est une boucle qui permet de capturer toutes les actions qui concernent le clavier, la souris, ... en même temps.

Nous allons maintenant revenir à la classe **Input** pour intégrer cette boucle dans une méthode. 😊

La méthode **updateEvenements()**

La méthode qui va implémenter cette boucle s'appellera **updateEvenements()**. Voici son prototype :

Code : C++

```
void updateEvenements();
```

Commençons cette méthode en codant la fameuse boucle **while** qui prendra en "condition" la valeur renournée par la fonction **SDL_PollEvent()**.

D'ailleurs, on donnera à cette dernière l'adresse de l'attribut **m_evenements** - car oui la structure **SDL_Event** existe toujours quelque part, elle n'a pas disparue. 🤪

Code : C++

```
void Input::updateEvenements()
{
    // Boucle d'évènements

    while(SDL_PollEvent(&m_evenements))
    {
        ...
    }
}
```

La suite du code ne changera pas beaucoup par rapport à la *SDL 1.2*. On met en place un gros **switch** qui va tester la valeur du champ **m_evenements.type**. On commencera par gérer les conditions relatives au clavier.

Deux cas doivent être gérés par le clavier :

- Lorsqu'une **touche** est *enfoncée*, l'évènement **SDL_KEYDOWN** est déclenché (même chose qu'avec la *SDL 1.2*).
- Lorsqu'une **touche** est *relâchée*, l'évènement **SDL_KEYUP** est déclenché (même chose aussi)

Dans les deux cas, on actualisera l'état de la touche correspondante dans le tableau **m_touches[]**. D'ailleurs, on utilisera le champ **m_evenements.key.keysym.scancode** qui nous servira d'indice pour retrouver la bonne case :

Code : C++

```
// Actualisation de l'état de la touche  
m_touches[m_evenements.key.keysym.scancode] = true;
```



Nous avons vu que le champ **scancode** remplaçait désormais le champ **sym**. 😊

Le gros avantage du tableau de *booléens* c'est que, quelque soit la touche enfoncée, nous n'avons besoin que d'une seule ligne de code pour actualiser son état. Que ce soit la touche **A**, **B**, **C**, **ESPACE**, ... une seule ligne suffit. 😊

En définitif, pour gérer les touches du clavier nous devons :

- Tester le champ **m_evenements.type** pour savoir si une touche a été enfoncée ou relâchée
- Actualiser la touche dans le tableau de *booléens* avec une valeur **true** ou un **false**

Code : C++

```
void Input::updateEvenements()  
{  
    // Boucle d'évènements  
  
    while(SDL_PollEvent(&m_evenements))  
    {  
        // Switch sur le type d'évènement  
  
        switch(m_evenements.type)  
        {  
            // Cas d'une touche enfoncée  
  
            case SDL_KEYDOWN:  
                m_touches[m_evenements.key.keysym.scancode] = true;  
                break;  
  
            // Cas d'une touche relâchée  
  
            case SDL_KEYUP:  
                m_touches[m_evenements.key.keysym.scancode] = false;  
                break;  
  
            default:  
                break;  
        }  
    }  
}
```

Grâce à ce code, toutes les touches du clavier peuvent être mises à jour en même temps (enfin, en une boucle 😊).

Gestion de la souris

Nous avons déjà fait la plus grosse part du travail, on sait désormais gérer toutes les touches du clavier. Nous allons maintenant passer à la gestion de la souris.

La bonne nouvelle, c'est que les boutons de la souris se gèrent de la même façon que les touches du clavier. Il suffit de rajouter deux **cases** au **switch** : un pour les boutons qui seront pressés (ce case utilisera la constante **SDL_MOUSEBUTTONDOWN**) et un autre pour les boutons qui seront relâchés (il utilisera la constante **SDL_MOUSEBUTTONUP**).

Pour récupérer l'indice du bouton dans le tableau **m_boutonsSouris[]**, nous n'utiliserons pas le champ **m_evenements.key.keysym.scancode**, car ce champ ne concerne uniquement que le clavier. A la place, nous utiliserons le champ **evenements.button.button** qui lui est réservé à la souris.

On ajoute donc les deux **cases** suivants au **switch** :

Code : C++

```
// Cas de pression sur un bouton de la souris

case SDL_MOUSEBUTTONDOWN:

    m_boutonsSouris[m_evenements.button.button] = true;
    break;

// Cas du relâchement d'un bouton de la souris

case SDL_MOUSEBUTTONUP:

    m_boutonsSouris[m_evenements.button.button] = false;
    break;
```

Ça c'était la partie des boutons. Maintenant il faut s'occuper des mouvements de la souris.

Lorsque la souris est en mouvement, un évènement est déclenché dans la *SDL*. Cet évènement va permettre de mettre à jour les *coordonnées* (*x*, *y*) du pointeur ainsi que ses coordonnées relatives. Nous allons pouvoir détecter ces mouvements grâce à la constante **SDL_MOUSEMOTION**. Lorsque cet évènement sera déclenché, on mettra à jour les attributs qui concernent les coordonnées. On prendra les nouvelles valeurs dans le champ **m_evenements.motion** :

Code : C++

```
// Cas d'un mouvement de souris

case SDL_MOUSEMOTION:

    m_x = m_evenements.motion.x;
    m_y = m_evenements.motion.y;

    m_xRel = m_evenements.motion.xrel;
    m_yRel = m_evenements.motion.yrel;

    break;
```

Fermeture de la fenêtre

Il ne reste plus qu'un seul évènement à gérer dans cette boucle : le cas de la fermeture de la fenêtre (la croix rouge en haut à

droite sous *Windows*). Depuis le début du tutoriel, on utilise cet évènement dans la boucle principale d'OpenGL pour savoir si on doit quitter le programme :

Code : C++

```
if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
    terminer = true;
```

Il faut maintenant enlever ce code pour le migrer dans la méthode **updateEvenements()**. N'oubliez pas que c'est elle et uniquement elle qui doit mettre à jour tous les évènements.

Nous allons donc ajouter un nouveau **case** dans le **switch** pour gérer cette fermeture. On utilisera la constante **SDL_WINDOWSEVENT** pour savoir si cet évènement a été déclenché. N'oubliez pas de modifier la variable **terminer** en **m_terminer**, car on met à jour non plus une variable mais un attribut :

Code : C++

```
// Cas de la fermeture de la fenêtre

case SDL_WINDOWEVENT:
    if(m_evenements.window.event == SDL_WINDOWEVENT_CLOSE)
        m_terminer = true;

break;
```

Si réunie tous ces **cases** dans la méthode, on trouve :

Code : C++

```
void Input::updateEvenements()
{
    // Boucle d'évènements

    while(SDL_PollEvent(&m_evenements))
    {
        // Switch sur le type d'évènement

        switch(m_evenements.type)
        {
            // Cas d'une touche enfoncee

            case SDL_KEYDOWN:
                m_touches[m_evenements.key.keysym.scancode] = true;
            break;

            // Cas d'une touche relachée

            case SDL_KEYUP:
                m_touches[m_evenements.key.keysym.scancode] = false;
            break;

            // Cas de pression sur un bouton de la souris

            case SDL_MOUSEBUTTONDOWN:
                m_boutonsSouris[m_evenements.button.button] = true;
            break;
        }
    }
}
```

```

// Cas du relâchement d'un bouton de la souris

case SDL_MOUSEBUTTONUP:

    m_boutonsSouris[m_evenements.button.button] = false;

break;


// Cas d'un mouvement de souris

case SDL_MOUSEMOTION:

    m_x = m_evenements.motion.x;
    m_y = m_evenements.motion.y;

    m_xRel = m_evenements.motion.xrel;
    m_yRel = m_evenements.motion.yrel;

break;


// Cas de la fermeture de la fenêtre

case SDL_WINDOWEVENT:

    if(m_evenements.window.event ==
SDL_WINDOWEVENT_CLOSE)
        m_terminer = true;

break;


default:
break;
}
}

```

Problème des coordonnées relatives

Cette méthode est presque complète, il reste juste un petit point à régler qui concerne les coordonnées relatives. Ces coordonnées représentent la différence entre la position actuelle et l'ancienne position, elles seront très utiles dans le chapitre sur la caméra.

Le problème avec ces coordonnées c'est que : s'il n'y a aucun évènement alors elles ne sont pas mises à jour, elles conservent donc leurs anciennes valeurs. Ce qui veut dire que le programme considère que la souris continue de bouger, même si elle est inactive.

Pour régler ce problème, on va ré-initialiser les coordonnées avec la valeur **0** au début de la méthode. Ne vous inquiétez pas, si les coordonnées doivent être mises à jour avec de vraies valeurs elles le seront dans le **switch**.

Grâce à cette astuce, nous n'aurons aucun problème de mouvement fictif. On rajoute donc ces deux lignes de code au début de la méthode :

Code : C++

```
// Pour éviter des mouvements fictifs de la souris, on réinitialise  
les coordonnées relatives
```



Attention, il faut mettre ces deux lignes de code avant la boucle **while**. Si vous les mettez à l'intérieur, les coordonnées ne seront ré-initialisées que lorsqu'il y aura un évènement. C'est justement ce qu'on cherche à éviter. 😊

Notre méthode donne donc au final :

Code : C++

```
void Input::updateEvenements()
{
    // Pour éviter des mouvements fictifs de la souris, on
    // réinitialise les coordonnées relatives

    m_xRel = 0;
    m_yRel = 0;

    // Boucle d'évènements

    while(SDL_PollEvent(&m_evenements))
    {
        // Switch sur le type d'évènement

        switch(m_evenements.type)
        {
            // Cas d'une touche enfoncee

            case SDL_KEYDOWN:
                m_touches[m_evenements.key.keysym.scancode] = true;
                break;

            // Cas d'une touche relachée

            case SDL_KEYUP:
                m_touches[m_evenements.key.keysym.scancode] = false;
                break;

            // Cas de pression sur un bouton de la souris

            case SDL_MOUSEBUTTONDOWN:
                m_boutonsSouris[m_evenements.button.button] = true;
                break;

            // Cas du relachement d'un bouton de la souris

            case SDL_MOUSEBUTTONUP:
                m_boutonsSouris[m_evenements.button.button] = false;
                break;

            // Cas d'un mouvement de souris

            case SDL_MOUSEMOTION:
                m_x = m_evenements.motion.x;
                m_y = m_evenements.motion.y;

                m_xRel = m_evenements.motion.xrel;
```

```
m_yRel = m_evenements.motion.yrel;

break;
```

// Cas de la fermeture de la fenêtre

```
case SDL_WINDOWEVENT:
```

if(m_evenements.window.event ==
 SDL_WINDOWEVENT_CLOSE)
 m_terminer = **true**;

```
    break;
```

default:
 break;

```
}
```

```
}
```

Voilà ! Maintenant nous sommes capables de mettre à jour tous nos évènements, peu importe leur nombre ils seront tous gérés simultanément par la classe **Input**. 😊

La méthode `terminer()`

Il ne reste plus qu'une chose à ajouter dans la classe : une méthode qui permet de dire si oui ou non l'utilisateur veut quitter le programme. Jusqu'à maintenant, on utilisait un booléen **terminer** pour fermer la fenêtre, mais maintenant ce booléen se trouve dans la classe **Input**.

Or, avec la règle de l'encapsulation on ne peut pas directement vérifier cet attribut. Il nous faut donc coder un *accesseur* pour récupérer la valeur du booléen.

Le prototype est terriblement simple. 🤪

Code : C++

```
bool terminer() const;
```

Son implémentation l'est tout autant :

Code : C++

```
bool Input::terminer() const
{
    return m_terminer;
}
```

Modification de la classe `SceneOpenGL`

On passe maintenant à l'utilisation de la classe **Input** dans notre scène 3D. Pour cela, on va remplacer l'ancien code de gestion des évènements par nos nouvelles méthodes.

On commence donc par déclarer un objet de type **Input** dans la classe **SceneOpenGL** qui viendra remplacer l'ancien attribut **m_evenements** :

Code : C++

```
#include "Input.h"

class SceneOpenGL
{
public:
    /* *** Méthodes *** */

private:
    /* *** Attributs *** */

    // Objet Input pour la gestion des évènements
    Input m_input;
}
```

On n'oublie pas de l'initialiser dans le *constructeur* :

Code : C++

```
SceneOpenGL::SceneOpenGL(std::string titreFenetre, int
largeurFenetre, int hauteurFenetre) : m_titreFenetre(titreFenetre),
m_largeurFenetre(largeurFenetre),
m_hauteurFenetre(hauteurFenetre), m_fenetre(0), m_contexteOpenGL(0),
m_input()
```

Dans la boucle principale, on ne vérifie donc plus le booléen **terminer** mais la valeur retournée par la méthode **terminer()** :

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    /* ***** RENDU ***** */
}
```

Enfin, on supprime notre ancien code de gestion d'évènements que l'on remplace par la méthode **updateEvenements()** de la classe **Input** :

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();
```

```
// Gestion des évènements  
  
m_input.updateEvenements();  
  
/* *** Rendu *** */  
}
```

Voilà pour la classe **Input** ! Grâce à elle, tous nos évènements seront mis à jour automatiquement. 😊

Maintenant que l'on a fait cela, on va pouvoir passer à l'implémentation de plusieurs méthodes "*indispensables*" qui vont nous permettre de connaître l'état du clavier, de la souris, ... Nous devons passer par ces méthodes pour respecter la règle de l'encapsulation, sans quoi nous serions obligés d'accéder directement aux attributs pour avoir nos valeurs.

Les méthodes indispensables

Méthodes

Cette dernière partie sera consacrée à une série de méthodes qui seront utilisées tout au long du tuto. Je vais vous donner une petite liste des fonctionnalités que l'on va implémenter. Il nous faudra une méthode pour :

- Savoir si une touche est enfoncée
- Savoir si un bouton de la souris est enfoncé
- Savoir si le pointeur de la souris a bougé
- Récupérer les coordonnées (x, y) du pointeur
- Récupérer les coordonnées relatives (x, y) du pointeur
- Cacher le pointeur
- Capturer le pointeur dans la fenêtre



Houla on va devoir coder tout ça ? 😊

Et bien oui. Mais sachez pour vous rassurer que la plus grosse méthode de cette liste ne fera que 4 lignes. 😊

Ces méthodes ne se contenteront que de faire une simple action (renvoyer un booléen par exemple) avec parfois des bloc ***if else***. Rien de compliqué ne vous inquiétez pas.

La méthode `getTouche()`

On commence par la méthode la plus importante : celle qui permet de savoir si une touche a été enfoncée (ou non). En gros, on va coder un getter sur le tableau **m_touches[]**. 😊 Elle prendra en paramètre une variable de type **SDL_Scancode** correspondant à la touche demandée :

Code : C++

```
bool getTouche(const SDL_Scancode touche) const;
```

Cette méthode renverra **true** si la touche est pressée ou **false** si elle ne l'est pas. N'oubliez pas de la déclarer en tant que méthode constante, vu que l'on ne modifie aucun attribut :

Code : C++

```
bool Input::getTouche(const SDL_Scancode touche) const
```

```
{  
    return m_touches[touche];  
}
```

Fini. 😊

Bonus : On va utiliser cette nouvelle méthode dès maintenant. Désormais, je veux que chacune de vos fenêtres SDL de chaque projet que vous ferez puisse se fermer en appuyant sur la touche **ECHAP**.

Comment ferez-vous ça avec le getter que l'on vient de coder ? Je vous laisse réfléchir un peu. 🧐

Vous avez trouvé ?

Secret (cliquez pour afficher)

Code : C++

```
// Boucle principale  
  
while(!m_input.terminer())  
{  
    // On définit le temps de début de boucle  
  
    debutBoucle = SDL_GetTicks();  
  
    // Gestion des évènements  
  
    m_input.updateEvenements();  
  
    if(m_input.getTouche(SDL_SCANCODE_ESCAPE))  
        break;  
  
    /* *** Rendu *** */  
}
```

Il suffit simplement d'appeler le getter **getTouche()** avec le scancode **SDL_SCANCODE_ESCAPE**. Si le getter retourne **true**, on casse la boucle avec le mot-clé **break**.

Bonus 2 : Pour savoir si deux touches sont enfoncées simultanément, il suffira d'utiliser un bloc **if** avec les deux touches demandées :

Code : C++

```
// Est-ce que les touches Z et D sont pressées ?  
  
if(m_input.getTouche(SDL_SCANCODE_Z) &&  
m_input.getTouche(SDL_SCANCODE_D))  
{  
    ...  
}
```

La méthode `getBoutonSouris()`

La méthode, ou plutôt le getter, `getBoutonSouris()` fera la même chose que `getTouche()`. C'est-à-dire qu'elle permettra de savoir si un bouton spécifié est enfoncé ou pas. La seule différence avec la méthode précédente c'est que cette fois-ci, on lui donnera en paramètre non pas un `SDL_Scancode` mais une variable de type `Uint8` correspondant au bouton demandé (En réalité, ce sera une constante comme avec les `scancodes`).

Code : C++

```
bool getBoutonSouris(const Uint8 bouton) const;
```

Elle renverra l'état du bouton demandé dans le tableau `m_boutonsSouris[]` :

Code : C++

```
bool Input::getBoutonSouris(const Uint8 bouton) const
{
    return m_boutonsSouris[bouton];
}
```

La méthode `mouvementSouris()`

La méthode suivante nous permettra de savoir si le pointeur de la souris a bougé. Grâce à elle, nous pourrons déclencher une action dès que la souris bougera. On appellera cette méthode : `mouvementSouris()`, elle renverra un `booléen`.

Code : C++

```
bool mouvementSouris() const;
```

Pour détecter un mouvement de souris il suffit de comparer la position relative du pointeur grâce aux attributs `m_xRel` et `m_yRel`. Si ces deux attributs sont égals à 0, alors le pointeur n'a pas bougé. Si en revanche ils ont une valeur non nulle, alors c'est que le pointeur a bougé.

Code : C++

```
bool Input::mouvementSouris() const
{
    if(m_xRel == 0 && m_yRel == 0)
        return false;

    else
        return true;
}
```

Les Getters du pointeur

Les méthodes suivantes sont des getters, elles renvoient chacune un attribut qui concerne la position du pointeur. Vous savez déjà comment fonctionne un getter, je vous épargne donc les explications. 😊

Voici leur constructeur :

Code : C++

```
// Getters

int getX() const;
int getY() const;

int getXRel() const;
int getYRel() const;
```

Implémentation :

Code : C++

```
// Getters concernant la position du curseur

int Input::getX() const
{
    return m_x;
}

int Input::getY() const
{
    return m_y;
}

int Input::getXRel() const
{
    return m_xRel;
}

int Input::getYRel() const
{
    return m_yRel;
}
```

La méthode `afficherPointeur()`

Les méthodes suivantes peuvent vous paraître inutiles pour le moment, mais dès que nous utiliserons une caméra mobile pour comprendrez vite leur utilité.

La méthode `afficherPointeur()` va permettre d'afficher ou de cacher le pointeur à l'écran. Elle prendra en paramètre un **booléen** qui :

- S'il est à **true** : le pointeur est caché
- S'il est à **false** : le pointeur est affiché

Code : C++

```
void afficherPointeur(bool reponse) const;
```

Cette méthode va faire appel à une *fonction SDL* pour afficher ou cacher le pointeur :

Code : C++

```
int SDL_ShowCursor(int toggle);
```

Cette fonction prend en paramètre une constante qui peut être égale soit à **SDL_ENABLE** soit à **SDL_DISABLE**, ce paramètre se comporte un peu comme un booléen. On ne s'occupera pas de la valeur renournée par la fonction.

Voici l'implémentation de la méthode :

Code : C++

```
void Input::afficherPointeur(bool reponse) const
{
    if(reponse)
        SDL_ShowCursor(SDL_ENABLE);

    else
        SDL_ShowCursor(SDL_DISABLE);
}
```

La méthode `capturerPointeur()`

La dernière méthode va permettre d'utiliser le *Mode Relatif de la Souris*. Ce mode permet de piéger le pointeur dans la fenêtre, il ne pourra pas en sortir. C'est utile voir obligatoire dans un jeu-vidéo par exemple. On appellera cette méthode : **capturerPointeur()**.

Faites attention à cette méthode, si vous lappelez dans votre code vous devrez prévoir quelque chose pour fermer votre programme. Utilisez un `getTouche()` par exemple. Si vous ne faites pas ça, vous ne pourrez plus fermer votre fenêtre. Vous devrez alors utiliser les combinaisons **CTRL + MAJ + SUPPR** ou **WINDOWS + TAB** (si vous êtes sous Windows) pour pouvoir ré-utiliser la souris.

Le prototype de la méthode est le suivant :

Code : C++

```
void capturerPointeur(bool reponse) const;
```

Elle se comportera exactement de la même manière que la méthode précédente. Elle utilisera la fonction **SDL_SetRelativeMouseMode()** de la *SDL* pour activer le mode relatif de la souris :

Code : C++

```
int SDL_SetRelativeMouseMode(SDL_bool enabled);
```

Cette fonction prendra en paramètre un **SDL_bool**. Sa valeur peut être soit **SDL_TRUE** soit **SDL_FALSE**. Voici son implémentation :

Code : C++

```
void Input::capturerPointeur(bool reponse) const
{
    if(reponse)
        SDL_SetRelativeMouseMode(SDL_TRUE);
```

```
    else
        SDL_SetRelativeMouseMode (SDL_FALSE) ;
}
```

Voilà pour l'implémentation des méthodes "indispensables". 😊

Télécharger : [Code Source C++ du Chapitre 9](#)

Exercices

Énoncés

Je vais vous donner des exercices assez simples pour vous habituez à utiliser votre nouvelle classe. Ceux-ci seront axés sur la pseudo-animation du cube (celle qui permettait de le faire tourner). Vous n'aurez besoin d'utiliser que les méthodes qui sont déjà codées, vous n'aurez donc pas besoin d'en rajouter.

Il est préférable que vous incliez le code relatif aux événements avant la fonction `glClear()`. En effet, vous verrez dans le futur que nous aurons parfois de faire plusieurs affichages de la même scène. Pour économiser du temps de calcul les inputs doivent se gérer avant ces affichages. La seule chose qui peut se trouver après la fonction c'est la méthode `rotate()` car elle ne fait pas partie des événements mais des matrices (et donc de l'affichage).

Autant prendre les bonnes habitudes dès maintenant, surtout que ça ne mange pas de pain. 😊

Exercice 1 : L'animation initiale du cube permettait de le faire pivoter selon un angle et par rapport à l'axe Y. Votre objectif est de dé-automatiser cette rotation pour qu'elle réponde aux touches du clavier **Flèche Gauche et Droite (SDL_SCANCODE_LEFT et SDL_SCANCODE_RIGHT)**. Vous pouvez vous aider du code que l'on avait utilisé dans le chapitre 7 :

Code : C++

```
// Incrémentation de l'angle
angle += 4.0;

if(angle >= 360.0)
    angle -= 360.0;

// Sauvegarde de la matrice
mat4 sauvegardeModelview = modelview;

// Rotation du repère
modelview = rotate(modelview, angle, vec3(0, 1, 0));

// Affichage du cube
cube.afficher(projection, modelview);

// Restauration de la matrice
modelview = sauvegardeModelview;
```

Exercice 2 : Même exercice que précédemment sauf que l'axe concerné n'est plus **Y** mais **X** et les touches sont **Flèche Haut et Bas (SDL_SCANCODE_UP et SDL_SCANCODE_DOWN)**.

Exercice 3 : Rassemblez les deux animations précédentes pour le même cube. C'est-à-dire que vous devez pouvoir appuyer sur les 4 touches directionnelles pour le faire pivoter selon l'axe **X et Y** (Petit indice : vous aurez besoin de deux angles).

Solution

Exercice 1 :

Secret (cliquez pour afficher)

Le but de l'exercice était de "manualiser" la rotation en fonction des touches **Haut** et **Bas**. Pour cela, il fallait simplement appeler la méthode `getTouche()` de l'objet `m_input` deux fois avec les constantes `SDL_SCANCODE_LEFT` et `SDL_SCANCODE_RIGHT` :

Code : C++

```
// Gestion des événements
...
// Rotation du cube vers la gauche
if(m_input.getTouche(SDL_SCANCODE_LEFT))
{
}

// Rotation du cube vers la droite
if(m_input.getTouche(SDL_SCANCODE_RIGHT))
{
}

// Nettoyage de l'écran
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
...
```



Faites attention à bien placer ce code **avant** la fonction `glClear()`.

Une fois la vérification des touches pressées faite, il ne manquait plus qu'à copier le code de rotation. Bien entendu, il fallait additionner l'angle dans un cas et le soustraire dans l'autre :

Code : C++

```
// Gestion des événements
...
```

```
// Rotation du cube vers la gauche

if(m_input.getTouche(SDL_SCANCODE_LEFT))
{
    // Modification de l'angle

    angle -= 4.0;

    // Limitation

    if(angle >= 360.0)
        angle -= 360.0;
}

// Rotation du cube vers la droite

if(m_input.getTouche(SDL_SCANCODE_RIGHT))
{
    // Modification de l'angle

    angle += 4.0;

    // Limitation

    if(angle >= 360.0)
        angle -= 360.0;
}

// Nettoyage de l'écran

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

....
```

Enfin, il ne manquait plus qu'à appeler la méthode **rotate()** avec l'angle de rotation en paramètre et afficher le cube. Le tout encadré par la sauvegarde et la restauration de la matrice **modelview**. 😊

Code : C++

```
// Sauvegarde de la matrice modelview

mat4 sauvegardeModelview = modelview;

// Rotation du repère

modelview = rotate(modelview, angle, vec3(0, 1, 0));

// Affichage du premier cube

cube.afficher(projection, modelview);

// Restauration de la matrice

modelview = sauvegardeModelview;
```

Exercice 2 :**Secret (cliquez pour afficher)**

Cet exercice reprend le même principe que le précédent sauf qu'il fallait, d'une part, modifier les constantes utilisées :

Code : C++

```
// Rotation du cube vers le bas

if(m_input.getTouche(SDL_SCANCODE_DOWN) )
{
    // Modification de l'angle

    angle -= 4.0;

    // Limitation

    if(angle >= 360.0)
        angle -= 360.0;
}

// Rotation du cube vers le haut

if(m_input.getTouche(SDL_SCANCODE_UP) )
{
    // Modification de l'angle

    angle += 4.0;

    // Limitation

    if(angle >= 360.0)
        angle -= 360.0;
}
```

Et d'autre part, il fallait également modifier l'axe de rotation de la méthode **rotate()** :

Code : C++

```
// Rotation du repère

modelview = rotate(modelview, angle, vec3(0, 1, 0));
```

Exercice 3 :**Secret (cliquez pour afficher)**

Le dernier exercice était un poil plus dur que les deux autres, il fallait utiliser deux angles et gérer 4 touches du clavier.

Vous pouviez donner n'importe quel nom aux angles :

Code : C++

```
// Angles de la rotation  
  
float angleX(0.0);  
float angleY(0.0);
```

La gestion des 4 touches n'était pas compliquée du moment que vous utilisiez le bon angle avec le bon axe de rotation :

Code : C++

```
// Rotation du cube vers la gauche  
  
if(m_input.getTouche(SDL_SCANCODE_LEFT))  
{  
    angleY -= 5;  
  
    if(angleY > 360)  
        angleY -= 360;  
}  
  
// Rotation du cube vers la droite  
  
if(m_input.getTouche(SDL_SCANCODE_RIGHT))  
{  
    angleY += 5;  
  
    if(angleY < -360)  
        angleY += 360;  
}
```

Code : C++

```
// Rotation du cube vers le haut  
  
if(m_input.getTouche(SDL_SCANCODE_UP))  
{  
    angleX -= 5;  
  
    if(angleX > 360)  
        angleX -= 360;  
}  
  
// Rotation du cube vers le bas  
  
if(m_input.getTouche(SDL_SCANCODE_DOWN))  
{  
    angleX += 5;  
  
    if(angleX < -360)  
        angleX += 360;  
}
```

J'ai divisé le code en deux pour le rendre plus lisible. Il fallait évidemment tout coder au même endroit. 😊

Enfin, pour gérer les deux rotations, il fallait simplement appeler la méthode **rotate()** deux fois. Un appel prenait en compte l'angle **angleX** et l'autre l'angle **angleY**:

Code : C++

```
// Sauvegarde de la matrice modelview  
mat4 sauvegardeModelview = modelview;  
  
// Rotation du repère  
modelview = rotate(modelview, angleY, vec3(0, 1, 0));  
modelview = rotate(modelview, angleX, vec3(1, 0, 0));  
  
// Affichage du premier cube  
cube.afficher(projection, modelview);  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;
```

Encore une fois, n'oubliez pas d'utiliser la sauvegarde/restauration lorsque vous faites une transformation. 😊

Notre classe **Input** est maintenant complète et prête à l'emploi. Avec elle, nous pourrons gérer tous nos événements simplement. Nous serons capable de gérer la pression de plusieurs touches, des mouvements de la souris, etc. Et tout ça de manière simultanée.

Pour utiliser les événements, il suffira juste de passer un objet de type **Input** aux modèles que l'on veut afficher. De plus, l'avantage d'avoir une classe à part entière c'est que l'on pourra ajouter de nouvelles méthodes au fur et à mesure du tutoriel. Nous n'aurons pas besoin de revenir en arrière pour modifier le code, il suffira d'ajouter ce qu'il faut dans la classe **Input**.

Après ce chapitre repos, je vous propose de passer à un chapitre *hyper méga* important qui concerne les **Textures** avec OpenGL ! 😊

Les textures

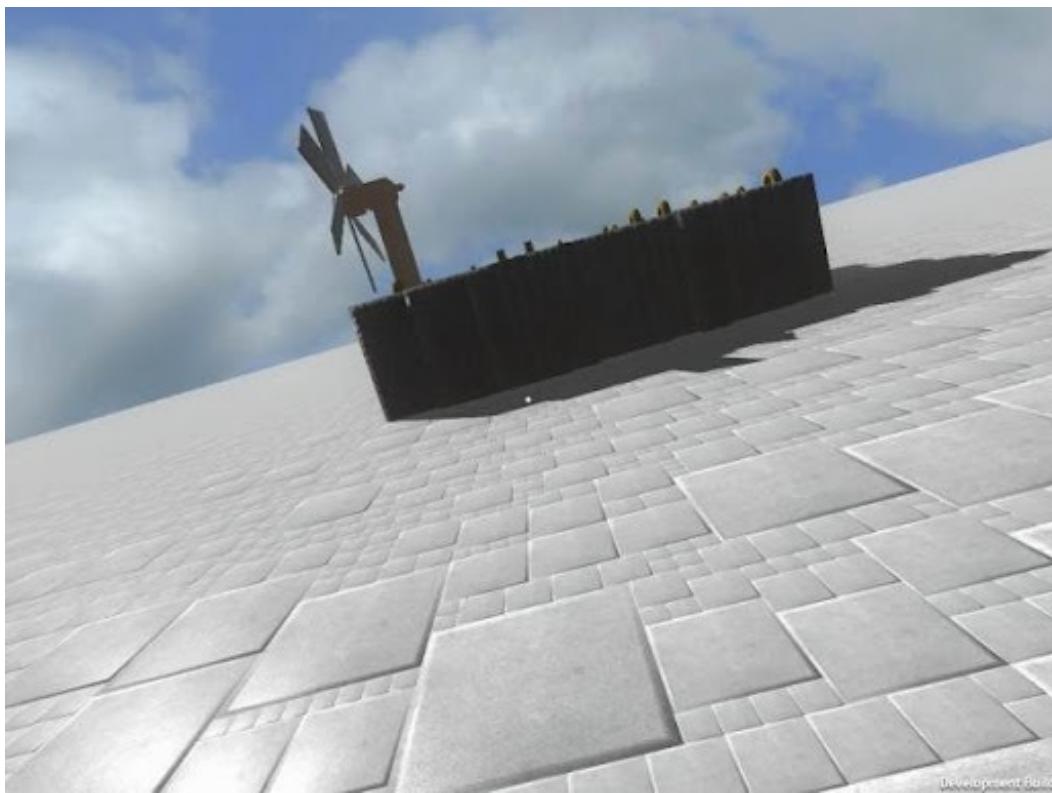
Après le chapitre quelque peu compliqué sur les piles de matrices, nous allons passer aujourd'hui à un chapitre beaucoup plus concret (et plus intéressant 😊) qui concerne le **Texturing** avec *OpenGL*. Avec ceci, nous pourrons rendre notre scène 3D plus réaliste. Nous verrons comment charger des textures stockées sur votre disque dur (ou SSD) et comment les appliquer à une surface 3D.

Introduction

Introduction

Avant de commencer ce chapitre très intéressant, on va définir ensemble un mot que vous avez déjà probablement entendu quelque part : le mot **Texture**.

Une *texture* est simplement une sorte de papier peint que l'on va "coller" sur les modèles (2D ou 3D). Dans un jeu vidéo, toutes les surfaces que vous voyez (sols, herbe, murs, personnages, ...) sont constituées de simples images collées sur des formes géométriques. Pour un mur par exemple, il suffit de créer une surface carrée puis de coller une image représentant des briques.



L'objectif de ce chapitre sera d'apprendre à créer une texture OpenGL de A à Z. Pour ceux qui auraient suivi le cours de M@téo sur la SDL, vous devriez déjà avoir une petite idée sur la façon de charger les images. Vous vous apercevrez cependant que le chargement de texture est assez différent avec OpenGL. Mais il permet de faire des choses beaucoup plus avancées.

L'une des premières difficultés va concerner la taille des textures à afficher. En effet, OpenGL ne sait gérer que les textures dont les dimensions sont des puissances de 2 (64 pixels, 128, 256, 512, ...).

Alors bon, ce n'est pas une obligation car OpenGL redimensionne de toute façon les images par lui-même mais si vous ne voulez pas vous retrouver avec une texture déformée, il vaut mieux prendre l'habitude des dimensions en puissance de 2. Le mieux est encore de modifier la taille de vos images avec des logiciels spécialisés. 😊

Avant d'aller plus loin, je vais vous demander de télécharger la librairie **SDL_image** (ou plutôt **SDL2_image**) qui nous permettra de réaliser la première étape dans la création de nos textures. Elle va nous faire gagner du temps en chargeant tous les bits d'une image en mémoire. Remarquez une fois de plus que toutes les librairies que nous utilisons sont portables. 😊

Télécharger : [SDL_image - MinGW pour Windows](#)

Télécharger : [Code Source SDL_image + Librairies - Visual C++ et Linux](#)

Pour MinGW sous Windows : fusionnez les dossiers **bin**, **dll**, **include** et **lib** avec ceux du dossier **SDL-2.0** que vous avez placé au tout début du tutoriel (chez moi : **C:\Program Files (x86)\CodeBlocks\MinGW\SDL-2.0**).

Pensez à rajouter les nouvelles dll soit dans le dossier de chaque projet, soit dans le dossier **bin** de MinGW selon la méthode que vous avez choisie au début du tuto.

Pour Linux ça va être folklo. 😊 Vu que tout le monde n'a pas forcément la même distribution, il faudra que vous compiliez vous-même **SDL_image**. Cependant, cette librairie fait appel à 5 autres librairies et il faudra également toutes les compiler ! Mais bon vous savez que je ne suis pas sadique (ah bon ?), je vais donc vous donner directement toutes les commandes à taper dans votre terminal.

Enfin, commencez par télécharger le code source de **SDL_image** et dézippez son contenu dans votre home. Ensuite, exécutez les commandes suivantes mais attention ! Si j'ai divisé les commandes en bloc ce n'est pas pour rien, je vous conseille d'exécuter les blocs un à un pour voir si tout compile normalement. Si vous copiez toute les commandes d'un coup vous risquez de zapper une erreur que vous regretterez plus tard. 😊

Code : Console

```
cd
cd SDL_image/tiff-4.0.3/
chmod +x configure
./configure
make
sudo make install
```

```
cd ../zlib-1.2.7/
chmod +x configure
./configure
make
sudo make install
```

```
cd ../libpng-1.5.13/
chmod +x configure
./configure
make
sudo make install
```

```
cd ../jpeg-8d/
chmod +x configure
./configure
make
sudo make install
```

```
cd ../libwebp-0.2.0/
chmod +x configure
./configure
make
sudo make install
```

```
cd ../SDL_image
chmod +x configure
./configure
make
sudo make install
```

Malgré le fait que vous compiliez toutes les librairies par vous-même, il semble qu'il y ait un problème avec le



chargement des images au format **PNG**. Tous les autres types de fichiers fonctionnent (jpeg, tiff, tga, bmp, ...) sauf celui-la. Je pense que ça vient du fait que la librairie est en plein développement, j'espère que le problème sera réglé rapidement.

On reprend au niveau des IDE **pour tout le monde**, il va falloir linker la librairie **SDL_image** avec vos projets. Voici un petit tableau avec le link à spécifier en fonction de votre IDE :

OS	Option
Code::Blocks Windows	SDL2_image
Code::Blocks Linux	SDL2_image
DevC++	-lSDL2_image
Visual Studio	SDL2_image.lib

Enfin pour terminer cette introduction, nous allons télécharger un pack de *textures* que l'on utilisera tout au long de ce tutoriel. Vous y verrez à l'intérieur plusieurs catégories d'images (sols, pierres, bois, ...). Je remercie au passage notre petit **Kayl** qui a fait découvrir ce pack dans son [tuto](#). Je reprends le même vu qu'il est assez complet.

[Télécharger : Pack de Textures Haute Résolution](#)

Chargement

Les Objets OpenGL

La librairie *SDL_image* va nous faciliter grandement la tâche dans le chargement de texture. Elle va nous faire gagner beaucoup de temps car elle sait charger une multitude de formats d'image, nous n'aurons donc pas à charger nos images manuellement. Cependant, elle ne peut pas tout faire pour nous. Les images chargées avec *SDL_image* seront, si on les laisse comme ça, inutilisables avec *OpenGL*.

En effet, la librairie permet de charger les textures uniquement pour la *SDL* et non pour les autres API. Il nous faut donc configurer *OpenGL* pour qu'il puisse reconnaître ces nouvelles textures. Dans cette partie nous allons voir pas mal de fonctions spécifiques à la librairie *OpenGL*, et vous verrez que vous retrouvez certaines d'entre elles tout au long du tutoriel. Je vous dirai celles qui sont importantes à retenir.

Les objets OpenGL

Avant d'aller plus loin, j'aimerais que l'on développe un point important de ce chapitre : les **objets OpenGL**. Les *objets OpenGL* sont semblables aux objets en C++ (même s'ils sont différents dans le fond), on peut les représenter par le laboratoire que l'on voit dans le chapitre sur les objets de [M@téo](#). Ce sont donc des sortes de laboratoires dont on ne connaît pas le fonctionnement, et d'ailleurs on s'en moque à partir du moment où ils fonctionnent.

Pourquoi je vous parle de ça ? Et bien simplement parce qu'une *texture* est un *objet OpenGL*. Vous verrez que l'on va apprendre à initialiser la texture mais vous n'aurez aucune idée de ce qui se passe à l'intérieur de la carte graphique, tout comme le laboratoire. Nous donnerons à la texture des pixels à afficher et *OpenGL* se chargera du reste. Bon je schématiserai un peu mais vous avez compris l'idée.



Comment on crée un *objet OpenGL* ? C'est dur à faire ?

Non pas du tout c'est en réalité très simple ! En effet, pour créer ces objets on utilise la plupart du temps la même fonction. Et cette fonction nous renverra toujours la même chose : un **ID** représentant l'objet créé. Cet ID est une variable de type **unsigned int** et va permettre à *OpenGL* de savoir sur quel objet il doit travailler.

En ce qui concerne la configuration de ces objets, nous procéderons ainsi :

- Chargement d'une image avec la librairie ***SDL_image***
- Création (ou plutôt de **génération**) de l'ID
- Verrouillage de l'ID (nous allons voir ce que c'est dans un instant)
- Configuration de l'objet
- Déverrouillage de l'ID

Toutes ces parties se gèrent avec les mêmes fonctions pour la plupart des *objets OpenGL* (que ce soit une texture ou autre). Il n'y a que l'étape de la configuration qui va varier.



Petite précision : Les shaders sont eux aussi des ***objets OpenGL*** (et oui encore eux !) mais ils se gèrent différemment des autres. Ce sont les seuls objets où nous aurons un contrôle plus global.

La classe Texture

On commence la partie programmation par le plus simple : la création d'une ***classe Texture***. Mis à part le constructeur et le destructeur, cette classe ne contiendra que la méthode **charger()** qui s'occupera de charger la texture demandée. Elle retournera un **booléen** pour confirmer ou non le chargement :

Code : C++

```
bool charger();
```

La classe contiendra également 2 attributs :

- **GLuint m_id** : Un *unsigned int* qui représentera le fameux ID
- **string m_fichierImage** : Le chemin vers le fichier contenant l'image

Au niveau du code, ça donne ça :

Code : C++

```
// Attributs  
  
GLuint m_id;  
std::string m_fichierImage;
```

Le constructeur prendra en paramètre une string qui représentera le chemin vers le **fichier image**. L'**ID** quant à lui sera généré par *OpenGL* :

Code : C++

```
// Constructeur  
  
Texture(std::string fichierImage);
```

On rajoutera au passage un accesseur pour l'attribut **m_id** et un mutateur pour **m_fichierImage** au cas où devrions spécifier une image après déclaration de l'objet. L'accesseur sera important pour la suite :

Code : C++

```
GLuint getID() const;
void setFichierImage(const std::string &fichierImage);
```

Sans oublier les includes de la *SDL*, de *SDL_image* et quelques autres voici ce que ça nous donne :

Code : C++

```
#ifndef DEF_TEXTURE
#define DEF_TEXTURE

// Include

#ifdef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <iostream>
#include <string>

// Classe Texture

class Texture
{
public:

    Texture(std::string fichierImage);
    ~Texture();
    bool charger();
    GLuint getID() const;
    void setFichierImage(const std::string &fichierImage);

private:

    GLuint m_id;
    std::string m_fichierImage;
};

#endif
```

Pour l'implémentation de ce début de code vous savez comment faire : 

Code : C++

```
#include "Texture.h"

// Constructeur

Texture::Texture(std::string fichierImage) : m_id(0),
m_fichierImage(fichierImage)
{
```

```
}

// Destructeur

Texture::~Texture()
{
}

// Méthodes

bool Texture::charger()
{
}

GLuint Texture::getID() const
{
    return m_id;
}

void Texture::setFichierImage(const std::string &fichierImage)
{
    m_fichierImage = fichierImage;
}
```

La méthode charger

Chargement de l'image dans une **SDL_Surface**

Maintenant que l'on a un squelette de classe propre, nous pouvons nous lancer dans la création de texture. La première étape consiste à charger un fichier image en mémoire grâce à la librairie *SDL_image*. Pour cela rien de plus simple, il existe une et unique fonction pour charger plus d'une dizaine de formats d'image différents ! Que demande le peuple. 😊

La fonction est la suivante :

Code : C++

```
SDL_Surface *IMG_Load(const char *file)
```

- **file** : Chemin du fichier image

La fonction renvoie une **SDL_surface** qui contiendra tous les pixels nécessaires.



Pensez bien à linker la librairie *SDL_image*, vous aurez une erreur de compilation dans le cas contraire.

Pour le chemin du fichier, nous donnerons l'attribut **m_fichierImage**, ou plutôt la *chaine C* de cet attribut car la fonction demande un tableau de caractère. 😊

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());
}
```

Attention cependant, la fonction peut renvoyer un pointeur sur 0. Il faut donc gérer cette erreur au cas où l'image n'existerait pas ou si le chemin donné contient une erreur. En cas de problème, on affiche alors un message d'erreur grâce à la fonction **SDL_GetError()** :

Code : C++

```
char* SDL_GetError(void);
```

Cette fonction permet de renvoyer la dernière erreur qu'a rencontrée la SDL (dans une chaîne de **char**). Donc en cas d'erreur de chargement, on inclut le résultat de cette fonction dans un flux **cout** :

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }
}
```

Génération de l'*ID*

On a vu tout à l'heure ce qu'étaient les objets *OpenGL* et on sait également que nous pouvons les gérer grâce à leur **ID**. Nous allons maintenant voir comment générer cet **ID**. Pour ce faire, il existe une fonction déjà toute prête dans OpenGL :

Code : C++

```
GLuint glGenTextures(GLsizei number, GLuint *textures);
```

- **number** : Le nombre d'*ID* à initialiser. Nous mettrons toujours la valeur 1
- **textures** : Un tableau de type **GLuint**. On peut aussi mettre l'adresse d'une variable **GLuint** pour initialiser un seul *ID* de texture (et c'est ce qu'on fera)



Les fonctions-générateurs d'**ID** commenceront toujours par le préfixe **glGenXXX()**. Les paramètres seront également les mêmes.

Pour générer un *ID* de texture, il suffit d'utiliser cette fonction en donnant en paramètre l'attribut **m_id** de notre classe *Texture* :

Code : C++

```
// Génération de l'ID
glGenTextures(1, &m_id);
```

On appelle cette fonction juste après avoir charger l'image en mémoire :

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());
    if (imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }

    // Génération de l'ID
    glGenTextures(1, &m_id);
}
```

Le verrouillage

Je vous ai parlé rapidement du verrouillage d'objet tout à l'heure, ceci permettait à *OpenGL* de verrouiller un objet pour travailler dessus. Tous les *objets OpenGL* doivent être verrouillés pour être configurés (et même pour être utilisés !) sinon vous ne pourrez rien faire avec.

On utilisera une fonction simple pour verrouiller nos objets. Ce sera d'ailleurs la même pour les déverrouiller. 😊 Voici son prototype :

Code : C++

```
void glBindTexture(GLenum target, GLuint texture);
```

- **target** : C'est un paramètre que vous retrouvez souvent avec tous les objets, nous le verrons même plusieurs fois dans ce chapitre. Il correspond au type de l'objet que l'on veut créer, nous lui affecterons la valeur **GL_TEXTURE_2D** en ce qui concerne les textures.
- **texture** : C'est l'*ID* de l'objet, nous lui donnerons la valeur de l'attribut **m_id**. **La valeur de m_id et non un pointeur cette fois ci!** 😊



Une fois de plus, cette fonction sera utilisée par tous les objets. Elle sera toujours de la forme **glBindXXX()**.

Voici donc comment utiliser la fonction dans notre cas :

Code : C++

```
// Verrouillage  
glBindTexture(GL_TEXTURE_2D, m_id);
```

Tiens au passage, vu que l'on a vu le *verrouillage* d'objets, nous allons voir maintenant le *déverrouillage* qui permet à OpenGL d'arrêter de se concentrer sur l'objet en cours, ce qui permet par extension d'empêcher les modifications.

Pour réaliser cette opération, on utilisera la même fonction mais avec le paramètre **target** non plus égal à la valeur de l'*ID* de la texture mais avec la valeur **0** (la valeur *nulle* quoi). En gros, on dit à OpenGL : "Verrouille l'objet possédant l'*ID* 0, soit rien du tout". 😊

Code : C++

```
// Déverrouillage  
glBindTexture(GL_TEXTURE_2D, 0);
```



Pensez bien à garder la valeur du paramètre **target** à **GL_TEXTURE_2D**, c'est important sinon le déverrouillage ne s'effectuera pas.

Petit récap :

Code : C++

```
bool Texture::charger()  
{  
    // Chargement de l'image dans une surface SDL  
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());  
  
    if(imageSDL == 0)  
    {  
        std::cout << "Erreur : " << SDL_GetError() << std::endl;  
        return false;  
    }  
  
    // Génération de l'ID  
    glGenTextures(1, &m_id);  
  
    // Verrouillage  
    glBindTexture(GL_TEXTURE_2D, m_id);  
  
    // Déverrouillage  
    glBindTexture(GL_TEXTURE_2D, 0);  
}
```

Configuration de la texture

Notre texture a un *ID* généré, elle est également verrouillée, on peut maintenant passer à sa configuration. 😊

Grossièrement parlant, pour avoir une texture dans OpenGL il suffit de copier les pixels d'une image dans la texture. C'est aussi simple que ça. Seulement voilà, il existe plusieurs formats d'image et certaines contiennent plus de données que d'autres, ...



Hein je croyais que la librairie **SDL_image** permettait justement de gérer tous ces formats ? 😕

Et bien oui vous avez raison, c'est bien **SDL_image** qui gère les différents formats de l'image. Il existe cependant une chose qu'elle ne peut pas nous dire automatiquement.

Vous n'êtes pas sans savoir qu'un pixel est composé de 3 couleurs (rouge, vert et bleu) ... Les pixels d'une image n'échappent pas à cette règle, chacun d'entre eux est composé de ces 3 couleurs. Seulement voilà, il existe, pour certains formats, une quatrième composante qui s'appelle la **composante Alpha**. Cette composante permet de stocker le "niveau de transparence" d'une image.

Pour charger correctement une texture, il faut savoir si cette valeur **alpha** est présente ou non, et heureusement pour nous, la librairie **SDL_image** est capable de nous le dire. En effet, dans la **structure SDL_Surface** utilisée au début de la méthode **charger()**, il existe un champ **BytesPerPixel** qui permet de dire s'il y a 3 ou 4 couleurs. Nous devrons donc d'abord récupérer cette valeur avant de copier les pixels dans la texture.

Bien on arrête là pour la théorie, on passe au code.

On veut savoir si une image possède 3 ou 4 couleurs, on récupère donc le champ **imageSDL->format->BytesPerPixel** pour le vérifier puis on met le tout dans un bloc **if**. Si on a une valeur inconnue, on arrête le chargement de la texture pour éviter de se retrouver avec une grosse erreur puis on n'oublie pas de libérer la *surface* **SDL** avant de quitter la méthode :

Code : C++

```
// Détermination du nombre de composantes

if(imageSDL->format->BytesPerPixel == 3)
{
}

else if(imageSDL->format->BytesPerPixel == 4)
{
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format de l'image inconnu" << std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}
```



On sait maintenant qu'il faut faire attention au bidule **alpha**, mais qu'est qu'on met à l'intérieur des **if**? Ils sont tout vides. 😕

C'est normal, il manque encore quelque chose. Comme on l'a vu plus tôt, OpenGL a besoin de savoir si la composante **alpha** existe ou pas. Seulement si on lui donne la valeur **3** ou **4** ça ne va pas lui suffire, il faudra envoyer une autre valeur qui sera un peu comme le paramètre **GL_TEXTURE_2D** que l'on a vu plus haut. Avec ce paramètre, il comprendra mieux ce qu'on lui enverra.

Il y aura deux cas à gérer :

- Soit l'image ne contiendra pas la composante *alpha* et dans ce cas on retiendra la constante **GL_RGB**
- Soit l'image contiendra la composante *alpha* et dans ce cas on retiendra la constante **GL_RGBA**

Au niveau du code, on utilisera une variable de type **GLenum** pour retenir cette valeur. On l'appellera **formatInterne**, vous verrez pourquoi juste après :

Code : C++

```
// Détermination du nombre de composantes

GLenum formatInterne(0);

if(imageSDL->format->BytesPerPixel == 3)
{
    formatInterne = GL_RGB;
}

else if(imageSDL->format->BytesPerPixel == 4)
{
    formatInterne = GL_RGBA;
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}
```

Il ne manque plus qu'une chose à faire. Selon le système d'exploitation ou même les images que vous utiliserez, les pixels ne seront pas stockés dans le même ordre. Par exemple sous Windows, la plupart des formats stockent leurs pixels selon l'ordre **Rouge Vert Bleu (RGB)** sauf les images au format **BMP**. Ceux-ci voient leurs pixels stockés selon l'ordre **Bleu Vert Rouge (BGR)**. C'est un problème que nous devons gérer car certains auront la belle surprise de voir leurs images avec des couleurs complètement inversées (Imaginez un Dark Vador en blanc ).

Il faut donc dire à OpenGL dans quel ordre les pixels sont stockés, et pour ça on va utiliser une autre variable de type **GLenum** que l'on appellera **format** :

Code : C++

```
// Détermination du format et du format interne

GLenum formatInterne(0);
GLenum format(0);
```

Pour connaître l'ordre des pixels, nous devons utiliser un autre champ de la structure **imageSDL**. Ce champ sera **imageSDL->format->Rmask**.

Il existe 4 champs similaires **Rmask**, **Gmask**, **Bmask** et **Amask** qui représente chacun la position de sa couleur à l'aide d'une valeur hexadécimal. Nous utiliserons le premier champ (**Rmask**), même si nous pouvions utiliser n'importe lequel. Sauf le dernier car il se trouve toujours à la fin quelque soit le format d'image.

Nous devons donc tester cette valeur pour connaître la position de la couleur rouge. Si sa valeur est égale à **0xff** alors elle est

placée au début, sinon c'est qu'elle se trouve à la fin :

Code : C++

```
// Format de l'image

GLenum formatInterne(0);
GLenum format(0);

// Détermination du format et du format interne

if(imageSDL->format->BytesPerPixel == 3)
{
    // Format interne

    formatInterne = GL_RGB;

    // Format

    if(imageSDL->format->Rmask == 0xff)
    {}

    else
    {}
}

else if(imageSDL->format->BytesPerPixel == 4)
{
    // Format interne

    formatInterne = GL_RGBA;

    // Format

    if(imageSDL->format->Rmask == 0xff)
    {}

    else
    {}
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}
```

Il faut maintenant affecter une valeur à la variable **format**. Il y a 4 cas à gérer :

- Rouge en **premier** pour une image à 3 composantes
- Rouge en **dernier** pour une image à 3 composantes
- Rouge en **premier** pour une image à 4 composantes
- Rouge en **dernier** pour une image à 4 composantes

Ces quatre cas seront représentés par les constantes suivantes :

- **GL_RGB**
- **GL_BGR**
- **GL_RGBA**
- **GL_BGRA**

Il n'y a plus qu'à affecter la bonne constante à la variable **format** :

Code : C++

```
// Format pour 3 couleurs

if(imageSDL->format->Rmask == 0xff)
    format = GL_RGB;

else
    format = GL_BGR;

.....

// Format pour 4 couleurs

if(imageSDL->format->Rmask == 0xff)
    format = GL_RGBA;

else
    format = GL_BGRA;
```

Ce qui donne au final :

Code : C++

```
// Format de l'image

GLenum formatInterne(0);
GLenum format(0);

// Détermination du format et du format interne pour les images à 3
// composantes

if(imageSDL->format->BytesPerPixel == 3)
{
    // Format interne

    formatInterne = GL_RGB;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGB;

    else
        format = GL_BGR;
}

// Détermination du format et du format interne pour les images à 4
// composantes

else if(imageSDL->format->BytesPerPixel == 4)
```

```

{
    // Format interne

    formatInterne = GL_RGBA;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGBA;

    else
        format = GL_BGRA;
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}

```

Pfiou tout ce gourbi pour déterminer deux valeurs ! 😅

On a fait le plus dur, il ne nous reste plus qu'à copier les fameux pixels dans la texture. Pour ça, on va utiliser la fonction suivante (ne soyez pas surpris du nombre de paramètres 🤪) :

Code : C++

```
void glTexImage2D(GLenum target, GLint level, GLint
internalFormat, GLsizei width, GLsizei height, GLint border,
GLenum format, GLenum type, const GLvoid * data);
```

- **target** : Comme on l'a vu précédemment, pour les textures on affectera toujours la valeur **GL_TEXTURE_2D**
- **level** : Paramètre que nous n'utiliserons pas, on le mettra à **0**
- **internalFormat** : Tiens ! On vient de le déterminer juste avant celui-là
- **width** : Largeur de l'image qui est contenue dans le champ **imageSDL->w**
- **height** : Hauteur de l'image qui est contenue dans le champ **imageSDL->h**
- **border** : Paramètre utile quand vous avez une bordure sur votre image. Nous donnerons la valeur **0** en général
- **format** : Oh lui aussi on l'a trouvé !
- **type** : Type de donnée des pixels (float, int, ...). Nous lui donnerons un type d'OpenGL : **GL_UNSIGNED_BYTE**
- **data** : Ce sont les pixels de l'image, on lui donnera l'adresse du tableau **imageSDL->pixels**

Ça en fait des paramètres tout ça ! Mais au moins on n'utilise qu'une seule fonction pour remplir notre texture.

Voyons son implémentation dans le code :

Code : C++

```

// Copie des pixels

glTexImage2D(GL_TEXTURE_2D, 0, formatInterne, imageSDL->w, imageSDL-
>h, 0, format, GL_UNSIGNED_BYTE, imageSDL->pixels);

```

On refait un petit récap :

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL

    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }

    // Génération de l'ID

    glGenTextures(1, &m_id);

    // Verrouillage

    glBindTexture(GL_TEXTURE_2D, m_id);

    // Format de l'image

    GLenum formatInterne(0);
    GLenum format(0);

    // Détermination du format et du format interne pour les images
    // à 3 composantes

    if(imageSDL->format->BytesPerPixel == 3)
    {
        // Format interne

        formatInterne = GL_RGB;

        // Format

        if(imageSDL->format->Rmask == 0xff)
            format = GL_RGB;

        else
            format = GL_BGR;
    }

    // Détermination du format et du format interne pour les images
    // à 4 composantes

    else if(imageSDL->format->BytesPerPixel == 4)
    {
        // Format interne

        formatInterne = GL_RGBA;

        // Format

        if(imageSDL->format->Rmask == 0xff)
            format = GL_RGBA;
```

```

    else
        format = GL_BGRA;
    }

    // Dans les autres cas, on arrête le chargement

    else
    {
        std::cout << "Erreur, format interne de l'image inconnu" <<
        std::endl;
        SDL_FreeSurface(imageSDL);

        return false;
    }

    // Copie des pixels

    glTexImage2D(GL_TEXTURE_2D, 0, formatInterne, imageSDL->w,
imageSDL->h, 0, format, GL_UNSIGNED_BYTE, imageSDL->pixels);

    // Déverrouillage

    glBindTexture(GL_TEXTURE_2D, 0);
}

```

Et voilà ! La texture contient enfin ses propres pixels issus de notre fichier image.

Les filtres

Allez courage, il ne nous manque plus qu'une chose à faire.

Je vais vous parler rapidement d'une notion que je connais très mal : la notion de *filtres*. Je ne m'y connais pas assez pour tenir tout un pavé alors je ferai vite. 😊

Un *filtre* permet de gérer la qualité d'affichage d'une texture. Il permet à OpenGL de savoir s'il doit afficher une image en mode **pixelisé** ou en mode **lisse**. On serait tenté de vouloir que toutes les textures soient lisses au moment de l'affichage mais sachez que ça joue sur la vitesse de votre programme. En effet, plus vous voudrez de belles textures à l'écran, plus votre matériel sera sollicité. Alors bon, aujourd'hui le problème est moins voyant qu'il y a 10 ans mais on est toujours aujourd'hui dans cette optique d'optimisation du rendu.

Pour vous éviter un gros bloc de théorie je vais résumer ma pensée en quelques lignes (en plus ça m'arrange, j'ai mal aux doigts 🤪).

Il existe deux types d'affichage pour une texture :

- Soit la texture est proche de l'écran et dans ce cas il vaut mieux la lisser, car le joueur a plus de chance de la voir
- Soit la texture est éloignée de l'écran et dans ce cas on peut se permettre de "l'afficher à l'arrache", le joueur ne s'en rendre même pas compte

Ces deux cas vont correspondre à deux *filtres* que nous allons créer.

Pour créer un *filtre*, on utilise la fonction suivante :

Code : C++

```
void glTexParameterI(GLenum target, GLenum pname, GLint param);
```

Cette fonction permet d'envoyer des paramètres supplémentaires à nos textures, dans notre cas des filtres. Voici ses paramètres :

- **target** : On ne le présente plus
- **pname** : Type de paramètre à envoyer, ici le filtre
- **param** : La valeur du paramètre

Avec cette fonction, nous pourrons créer deux filtres : un pour le cas où la texture est proche de l'écran et un autre pour le cas où elle en est éloignée.

Pour le premier filtre :

- Le filtre permettant de gérer les "*textures proches*" s'appelle **GL_TEXTURE_MIN_FILTER**, ce sera donc la valeur du paramètre **pname**.
- Nous voulons que les "textures proches" soient "lisses", nous donnerons donc la valeur **GL_LINEAR** au paramètre **param**

Code :

Code : C++

```
// Application des filtres  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Pour le second filtre :

- Le filtre permettant de gérer les "*textures éloignées*" s'appelle **GL_TEXTURE_MAG_FILTER**, ce sera donc la valeur du paramètre **pname**.
- Nous voulons que les "textures éloignées" soient "pixelisées", nous donnerons donc la valeur **GL_NEAREST** au paramètre **param**

Code :

Code : C++

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

Grâce à ces filtres, nous pourrons avoir un bon équilibre entre érgonomie pour le joueur et rapidité pour le CPU et la carte graphique.

Petit récap :

Code : C++

```
bool Texture::charger()  
{  
    // Chargement de l'image dans une surface SDL  
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());  
  
    if(imageSDL == 0)  
    {  
        std::cout << "Erreur : " << SDL_GetError() << std::endl;  
        return false;  
    }
```

```
// Génération de l'ID
glGenTextures(1, &m_id);

// Verrouillage
 glBindTexture(GL_TEXTURE_2D, m_id);

// Format de l'image
GLenum formatInterne(0);
GLenum format(0);

// Détermination du format et du format interne pour les images
à 3 composantes

if(imageSDL->format->BytesPerPixel == 3)
{
    // Format interne

    formatInterne = GL_RGB;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGB;

    else
        format = GL_BGR;
}

// Détermination du format et du format interne pour les images
à 4 composantes

else if(imageSDL->format->BytesPerPixel == 4)
{
    // Format interne

    formatInterne = GL_RGBA;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGBA;

    else
        format = GL_BGRA;
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}

// Copie des pixels
```

```
    glTexImage2D(GL_TEXTURE_2D, 0, formatInterne, imageSDL->w,
imageSDL->h, 0, format, GL_UNSIGNED_BYTE, imageSDL->pixels);

    // Application des filtres

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

    // Déverrouillage

    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Fin de la méthode

Pfffffiou, entre les *ID*, le *verrouillage*, *les filtres* et tout ça je n'en puis plus (comme dirait Hooper) !

Il est temps de terminer cette méthode. Il ne reste plus qu'à libérer la *surface SDL* qui contenait les pixels et à retourner la valeur **true** (car la méthode s'est normalement terminée sans erreur) :

Code : C++

```
// Fin de la méthode

SDL_FreeSurface(imageSDL);
return true;
```

Récap final :

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL

    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }

    // Génération de l'ID

    glGenTextures(1, &m_id);

    // Verrouillage

    glBindTexture(GL_TEXTURE_2D, m_id);

    // Format de l'image
```

```
GLenum formatInterne(0);
GLenum format(0);

// Détermination du format et du format interne pour les images
à 3 composantes

if(imageSDL->format->BytesPerPixel == 3)
{
    // Format interne

    formatInterne = GL_RGB;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGB;

    else
        format = GL_BGR;
}

// Détermination du format et du format interne pour les images
à 4 composantes

else if(imageSDL->format->BytesPerPixel == 4)
{
    // Format interne

    formatInterne = GL_RGBA;

    // Format

    if(imageSDL->format->Rmask == 0xff)
        format = GL_RGBA;

    else
        format = GL_BGRA;
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageSDL);

    return false;
}

// Copie des pixels

glTexImage2D(GL_TEXTURE_2D, 0, formatInterne, imageSDL->w,
imageSDL->h, 0, format, GL_UNSIGNED_BYTE, imageSDL->pixels);

// Application des filtres

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

```
// Déverrouillage  
glBindTexture(GL_TEXTURE_2D, 0);  
  
// Fin de la méthode  
SDL_FreeSurface(imageSDL);  
return true;  
}
```

Enfin ! Nous avons une classe **Texture** qui permet de charger des fichiers images pour les afficher dans nos scènes 3D. Le petit truc que je ne vous ai pas dit, c'est que cette méthode est incomplète, il manque encore quelque chose mais nous verrons ça dans la dernière sous-partie de ce chapitre. Je ne vais pas vous assassiner plus pour l'instant. 😊 Nous avons ce qu'il nous faut pour afficher une texture sur un modèle 3D et c'est le principal pour le moment. Je veux que vous vous familiarisez avec l'affichage de texture avant d'aller plus loin. 😊

Plaquette

Affichage d'une texture

Le plus dur est derrière nous, nous avons appris à déclarer et remplir un *texture* au sein d'OpenGL. On va maintenant apprendre à les afficher. Et la bonne nouvelle, c'est que vous savez déjà comment faire. 🎉 Ça se passe de la même façon qu'avec les vertices ou les couleurs.

Pour débuter, on va se contenter d'afficher notre première texture sur un simple carré afin de se concentrer sur l'essentiel. Nous utiliserons ensuite notre classe **Cube** pour afficher un modèle un peu plus élaboré. D'ailleurs, voyons ensemble la première texture que nous serons capables de traiter :



Elle nous sera utile tout au long du chapitre, vous la trouverez dans l'archive que vous avez téléchargée au début sous le nom de **crate13.jpg**. Au pire des cas, faites un clique droit sur l'image ci-dessus pour la récupérer. Nous aurons l'occasion d'en utiliser d'autres aussi.

Coordonnées de Texture

Je ne sais pas si vous lisez toujours les sous-titres oranges et bleues, mais si vous lisez une ligne plus haut vous tomberez sur le terme "**coordonnées de texture**".

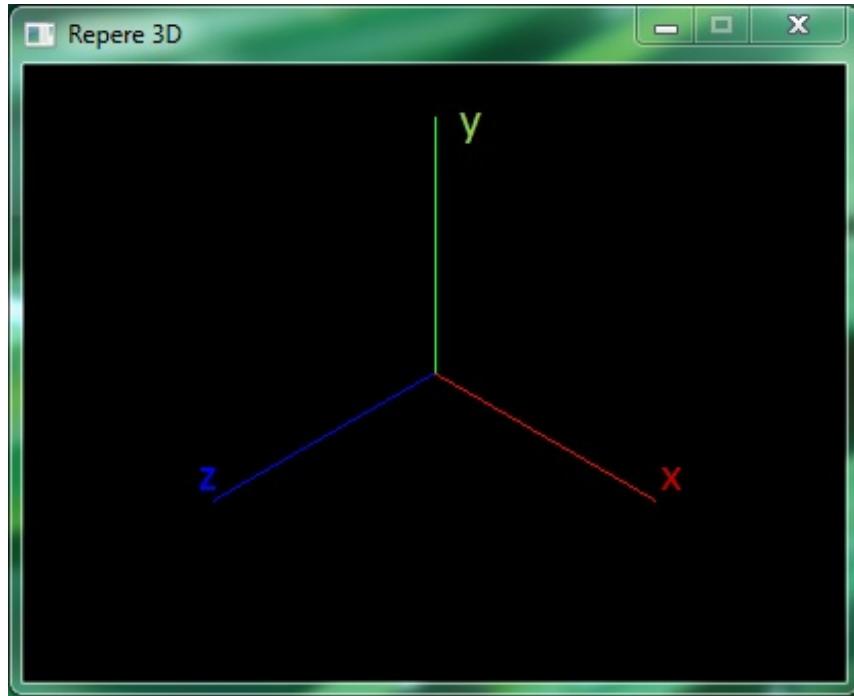


Oh ! C'est quoi ces coordonnées ? C'est similaire aux vertices et tout ça ?

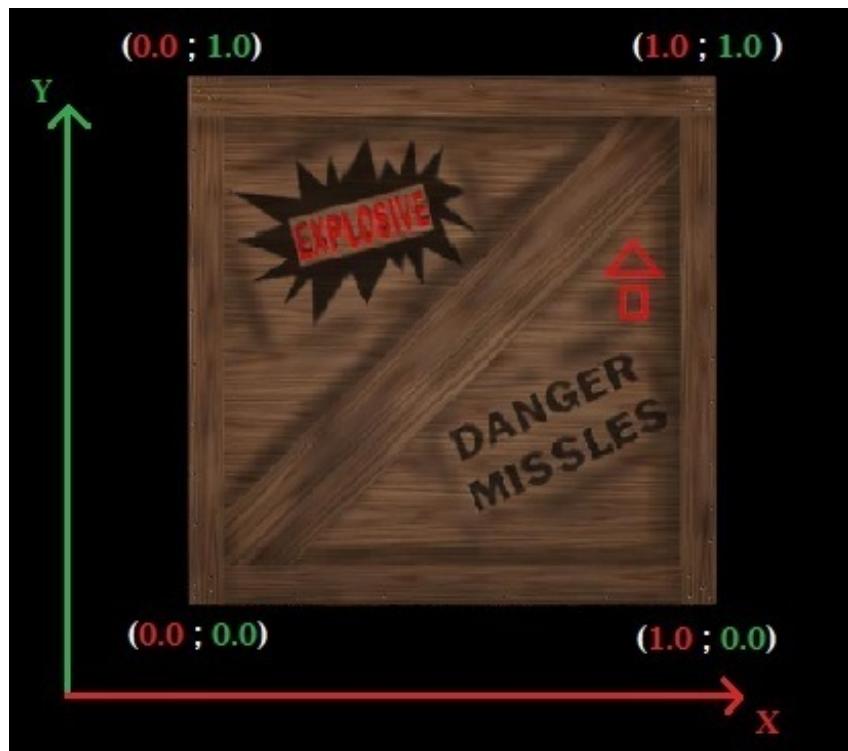
Et bien oui ! On peut faire une analogie avec les vertices car, pour afficher une texture, il faut lui donner une sorte de position. Seulement attention, on ne parle pas ici de coordonnées spatiales avec un repère 3D, une matrice et tout ça ...

Je vais vous expliquer ça avec deux schéma.

Vous savez depuis un moment que les vertices sont composés de coordonnées spatiales (X, Y et Z) :



Avec ce système, vous pouvez donner n'importe quelle taille à une forme 3D comme le cube des chapitres précédents par exemple. Et bien les textures se comportent de la même façon. A une exception près : les coordonnées de texture sont toujours comprises entre 0 et 1 :



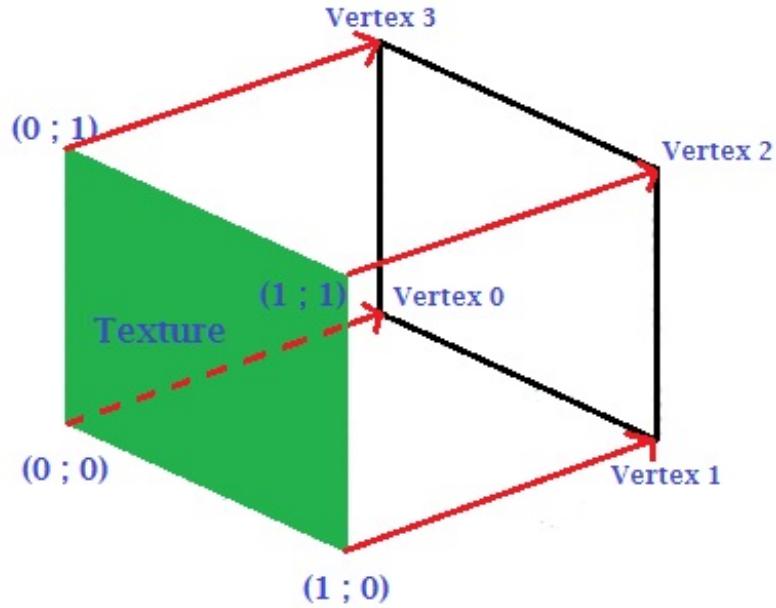
On peut voir sur le schéma le point d'origine de la texture en bas à gauche ainsi que 3 autres coordonnées. La plupart du temps,

nous n'utiliserons que ces 4 points vu qu'ils correspondent aux coins de la texture. Mais sachez qu'il est tout à fait possible d'avoir des coordonnées du type (0.27; 0.38) pour ne récupérer qu'une partie de la texture.

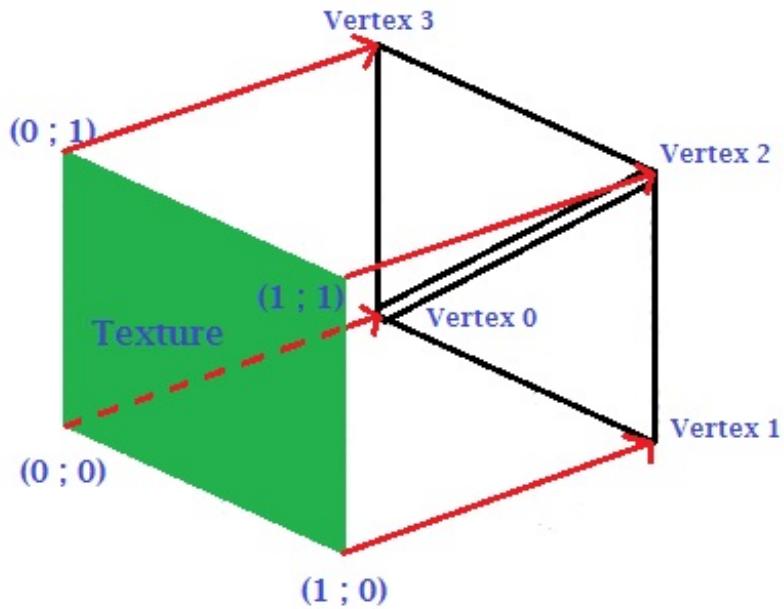


Attention, ces nombres n'ont aucun rapport avec le repère 3D. Elles sont propres à la texture.

Pour plaquer une texture sur une surface il faudra simplement faire correspondre les coordonnées de texture avec les bons vertices :



La chose se complexifie légèrement quand on n'oublie le carré et que l'on parle des deux triangles, mais le tout reste tout de même assez simple :



Simple n'est-ce pas ? 🍪

Vous savez maintenant ce que sont les coordonnées de texture : ce sont des points, correspondant généralement aux coins d'une texture, qui permettent de la plaquer sur une surface.

Affichage d'une texture

Nous avons toutes les cartes en main pour afficher notre première texture. Nous n'avons plus qu'à instancier la classe **Texture** et à utiliser ses coordonnées.

On reprend donc notre bonne vieille classe **SceneOpenGL** pour déclarer un objet de type **Texture** (n'oubliez pas d'inclure le header) qu'on appellera simplement **texture**, puis on la chargera en mémoire. Pensez également à effacer tout le code qui concernait le cube des chapitres précédents :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
    m_hauteurFenetre, 1.0, 100.0);
```

```
modelview = mat4(1.0);

// Texture

Texture texture("Textures/Caisse.jpg");
texture.charger();

// Boucle principale

while(!m_input.terminer())
{
    /* *** Boucle principale *** */
}
}
```

Je vous redonne au passage le code de base pour la boucle principale :

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if(m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra
    modelview = lookAt(vec3(0, 0, 2), vec3(0, 0, 0), vec3(0, 1, 0));

    /* *** Rendu *** */

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow(m_fenetre);

    // Calcul du temps écoulé
    finBoucle = SDL_GetTicks();
    tempsEcoule = finBoucle - debutBoucle;

    // Si nécessaire, on met en pause le programme
}
```

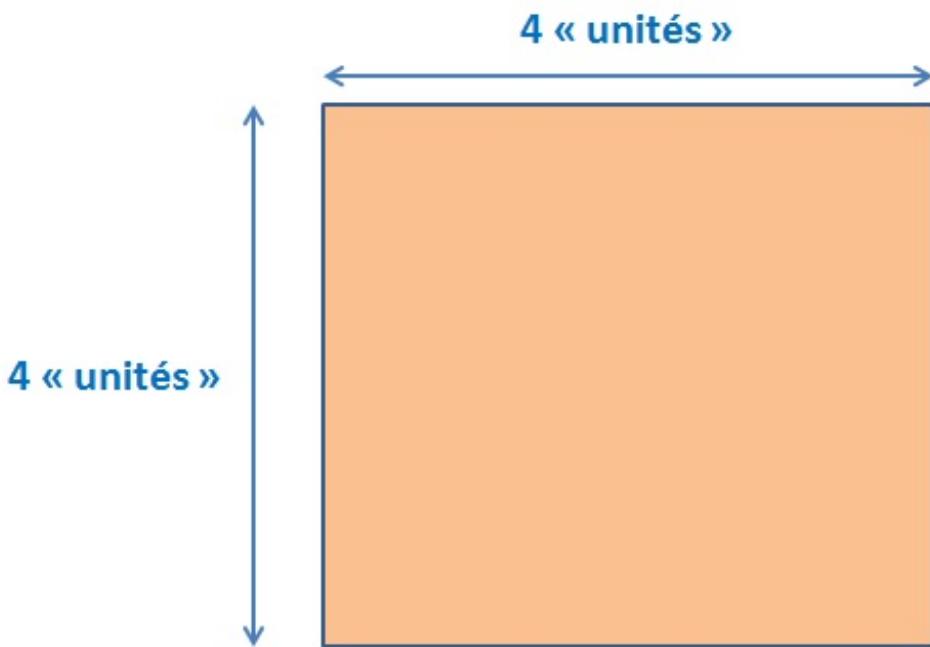
```
    if (tempsEcoule < frameRate)
        SDL_Delay(frameRate - tempsEcoule);
}
```



Remarquez que j'ai placé la caméra au point (0; 0; 2).

Bien, avec ce code propre on va pouvoir bosser tranquillement.

Notre objectif depuis le début de ce chapitre est l'affichage d'une texture sur une surface, et de préférence un carré. On va commencer par créer nos coordonnées pour les vertices. D'ailleurs, vous savez déjà depuis un moment comment faire un carré avec OpenGL. Je vais donc vous demander de me coder vous-même les vertices pour un carré avec des arrêtes mesurant 4 unités de longueur :



Cependant, je souhaite attirer votre attention sur deux points :

- Faites votre carré en 3 dimensions, il faut donc intégrer la coordonnée **Z**
- Faites attention au repère qui se trouve au milieu de la surface et non en bas à gauche.

Voilà pour les explications, en gros faites un carré à l'ancienne avec 3 dimensions. 😊

...

Fini ? Voici la solution :

Secret (cliquez pour afficher)

Code : C++

```
// Vertices

float vertices[] = {-2, -2, -2,     2, -2, -2,     2, 2, -2,     //
Triangle 1
-2, -2, -2,     -2, 2, -2,     2, 2, -2}; //
```

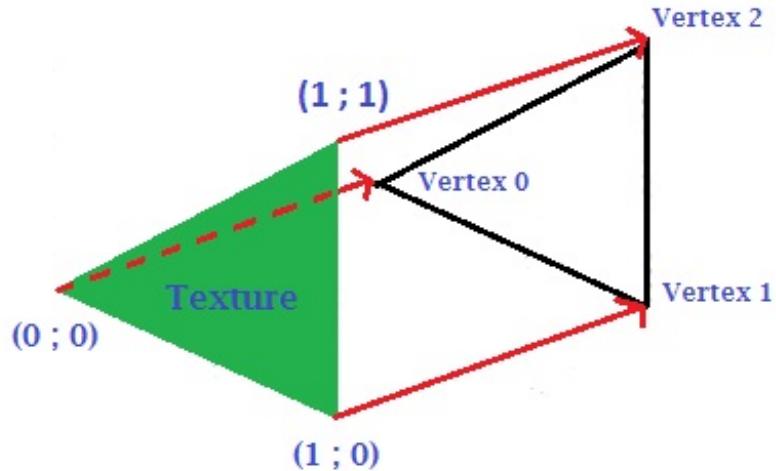
Triangle 2

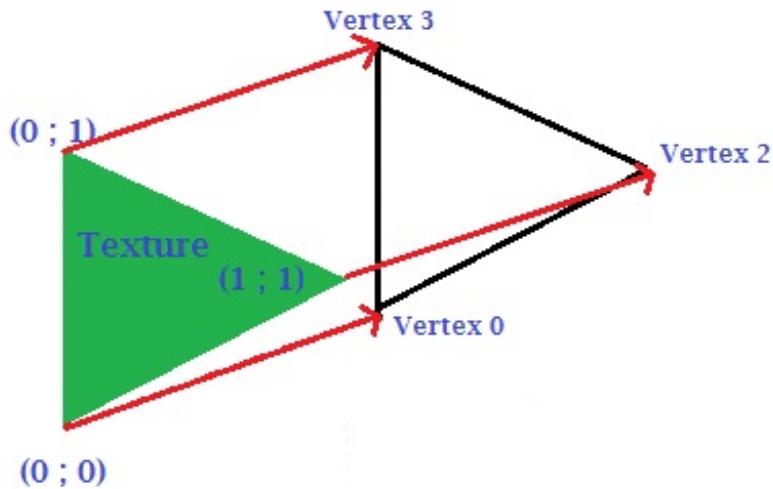
Aaaah, faire un carré c'est tout de même plus simple que de devoir faire un cube complet.

L'étape suivante consisterait normalement à colorier notre nouvelle forme. Cependant, nous n'en avons plus besoin vu que nous avons une texture qui va s'appliquer dessus.

Dans notre code, au lieu de déclarer un tableau de couleurs nous allons déclarer un tableau de coordonnées de texture. Et comme je suis sadique, je vais vous demander de me coder ce tableau vous-même comme des grands. 🍪 N'ayez pas peur c'est exactement le même principe qu'avec les vertices et les couleurs.

Pour vous aider dans cette tâche je vais vous donner deux petites schémas avec les informations nécessaires :





Rien de bien compliqué, vous devez simplement faire comme s'il s'agissait de vertices. Vous avez toutes les coordonnées (x, y) et vous devez en faire un tableau. Faites attention cependant à leur ordre à l'intérieur du tableau car celles-ci sont liées à vos vertices (au même titre que les couleurs).

... Pause café ...

Finis. Allez on passe à la correction.

Secret (cliquez pour afficher)

Code : C++

```
// Coordonnées de texture  
float coordTexture[] = {0, 0, 1, 0, 1, 1, // Triangle 1  
0, 0, 0, 1, 1, 1}; // Triangle 2
```

On a maintenant un carré, une texture et des coordonnées permettant de la plaquer dessus. On n'a plus qu'à donner tout ça à OpenGL et le tour est joué. 😊



Ah mais justement : comment on envoie les coordonnées à OpenGL ?

Bonne question, je vous ai dit tout à l'heure que les coordonnées de texture se géraient de la même façon que les vertices et les couleurs. On va donc utiliser donc un **Vertex Array** pour envoyer toutes les données à OpenGL.

Vous vous souvenez des **Vertex Array** quand même ? Ce sont des tableaux qui font le lien entre les coordonnées et OpenGL. Le tableau **VertexAttrib 0** permet d'envoyer les vertices et le tableau **VertexAttrib 1** permet d'envoyer les couleurs. Maintenant nous allons utiliser le tableau **VertexAttrib 2** qui va nous permettre d'envoyer nos coordonnées de texture.

Dans notre code, il nous faut appeler la fonction **glVertexAttribPointer()** avec l'indice **2** (et non **4**). Il faut également donner la valeur **2** au paramètre **size** car nos coordonnées de texture ne sont constituées que de couple de valeurs (**x, y**) et non de triplet (**x, y, z**). On pensera à donner le tableau **coordTexture** en dernier paramètres sinon on ne risque pas d'envoyer grand chose à OpenGL. 😊

Voici à quoi ressemblerait l'appel à **glVertexAttribPointer()** dans notre cas :

Code : C++

```
// Envoi des coordonnées de texture  
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);
```

Puisque nous envoyons des données à travers un tableau **VertexAttrib**, il faut penser à l'activer grâce à la fonction **glEnableVertexAttribArray()**. On appelle donc celle-ci avec la valeur **2** pour activer le tableau d'indice **2** :

Code : C++

```
// Envoi des coordonnées de texture  
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);  
glEnableVertexAttribArray(2);
```

On place ce bout de code au même endroit où on envoyait les couleurs, c'est-à-dire entre les deux appels à la fonction **glUseProgram()** :

Code : C++

```
// Activation du shader  
glUseProgram(shader.getProgramID());  
  
// Envoi des vertices  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);  
glEnableVertexAttribArray(0);  
  
// Envoi des coordonnées de texture  
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);  
glEnableVertexAttribArray(2);  
  
// Envoi des matrices et rendu  
....  
  
// Désactivation des tableaux
```

```
glDisableVertexAttribArray(2);
glDisableVertexAttribArray(0);

// Désactivation du shader

glUseProgram(0);
```



Pensez à appeler la fonction **glDisableVertexAttribArray()** avec la valeur **2** pour désactiver votre tableau.

Tiens en parlant de shader, nous allons devoir modifier les codes sources (le petits fichiers dans le dossier **Shaders**) qu'il utilise afin de pouvoir intégrer l'affichage de texture. En effet, ceux que nous avons utilisés jusqu'à maintenant ne gèrent que la couleur, nous en avons donc besoin de nouveaux pour gérer notre nouvelle façon d'afficher.

Ces nouveaux codes sources se nomment **texture.vert** et **texture.frag** et devraient être présents dans votre dossier **Shaders** :

Code : C++

```
// Shader gérant les texture

Shader shaderTexture("Shaders/texture.vert",
"Shaders/texture.frag");
shaderTexture.charger();
```

Faisons un petit récap (sans la boucle principale) pour voir où nous en sommes pour le moment :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Vertices

    float vertices[] = {-2, -2, -2,     2, -2, -2,     2, 2, -2,     // Triangle 1
                        -2, -2, -2,     -2, 2, -2,     2, 2, -2}; // Triangle 2

    // Coordonnées de texture

    float coordTexture[] = {0, 0,     1, 0,     1, 1,     // Triangle 1
                           0, 0,     0, 1,     1, 1}; // Triangle 2
```

```
// Texture  
  
Texture texture("Textures/Caisse.jpg");  
texture.charger();  
  
// Shader  
  
Shader shaderTexture("Shaders/texture.vert",  
"Shaders/texture.frag");  
shaderTexture.charger();  
  
// Boucle principale  
  
while (!m_input.terminer())  
{  
    /* *** Rendu *** */  
}  
}
```

Nous ferons un petit récap de la boucle principale dans un instant. Concentrons-nous d'abord sur le code d'affichage.

Celui-ci va commencer par l'activation du shader (qui gère les textures maintenant) suivie de l'envoi des vertices, des coordonnées de texture et des matrices ainsi que de l'appel à la fonction de rendu **glDrawArrays()** :

Code : C++

```
// Activation du shader  
  
glUseProgram(shaderTexture.getProgramID());  
  
// Envoi des vertices  
  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);  
 glEnableVertexAttribArray(0);  
  
// Envoi des coordonnées de texture  
  
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);  
 glEnableVertexAttribArray(2);  
  
// Envoi des matrices  
  
glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),  
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
// Rendu  
  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Désactivation des tableaux  
  
glDisableVertexAttribArray(2);  
glDisableVertexAttribArray(0);
```

```
// Désactivation du shader  
glUseProgram(0);
```

Ah encore une petite chose, normalement je devrais faire le sadique en vous demandant de compiler votre code maintenant. Seulement si vous le faites vous aurez au mieux un écran noir. 😊

Il manque donc encore une dernière chose à faire. Pour le moment, nous nous contentons d'envoyer les *coordonnées de texture* à OpenGL, mais à aucun moment nous lui disons *quelle texture* il doit afficher. Et comme OpenGL n'est pas devin, il va nous envoyer balader et afficher un carré noir.

Pour éviter cela, il faut faire une opération que vous savez déjà faire : le verrouillage de texture. En effet, grâce à ça, OpenGL saura ce qu'il doit afficher exactement comme au moment de la configuration où il savait sur quelle texture travailler.

Pour effectuer cette opération, nous allons réutiliser la fonction **glBindTexture()** avec le même paramètre que d'habitude soit l'**ID** de la texture. D'où l'utilité de la méthode **getID()** que je vous avais demandée de coder précédemment.

En définitif, nous devons appeler la fonction **glBindTexture()** deux fois avec la méthode **getID()** en tant que paramètre pour le premier appel et la valeur **0** pour le second :

Code : C++

```
// Verrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, texture.getID());  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);
```

Je vous conseille de verrouiller vos textures uniquement au moment de l'affichage. C'est-à-dire que pour **CHAQUE** texture vous devez :

- La verrouiller
- Afficher votre surface
- La déverrouiller immédiatement

Si vous ne le faites pas vous risquez d'afficher une mauvaise texture par la suite.

 Et si on veut afficher deux textures en même temps sur une surface on fait comment ? On les verrouille toutes les deux ?

Ah là on touche au domaine du *multi-texturing*, vaste sujet passionnant que nous ne verrons pas maintenant. Nous verrons cela quand nous saurons manipuler les shaders. 😊

On revient au code avec un petit récap de la boucle principale :

Code : C++

```
// Boucle principale

while (!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if (m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra
    modelview = lookAt(vec3(0, 0, 2), vec3(0, 0, 0), vec3(0, 1, 0));

    // Activation du shader
    glUseProgram(shaderTexture.getProgramID());

    // Envoi des vertices
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);
    glEnableVertexAttribArray(0);

    // Envoi des coordonnées de texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
coordTexture);
    glEnableVertexAttribArray(2);

    // Envoi des matrices
    glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));

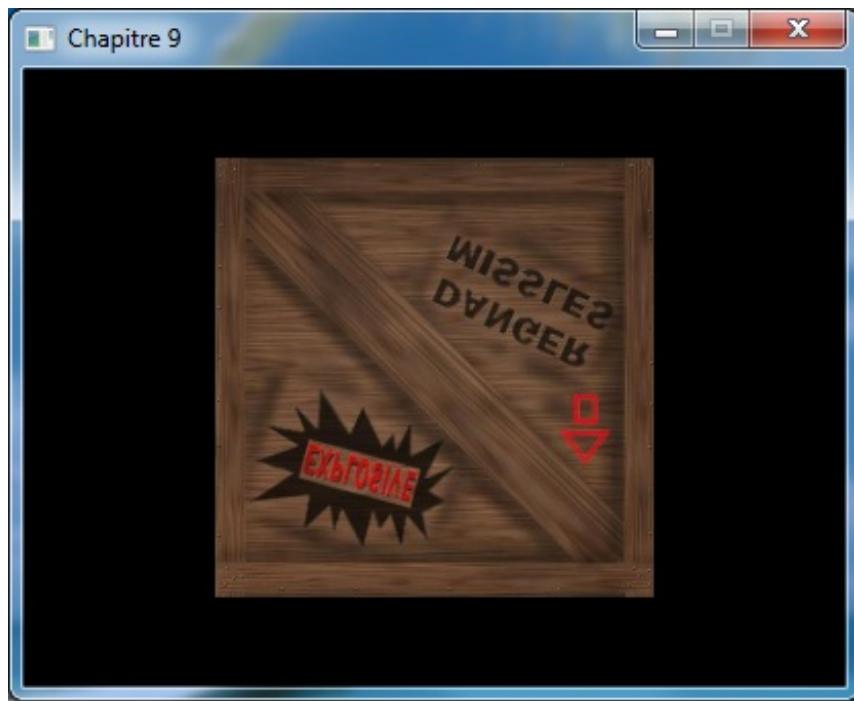
    // Verrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, texture.getID());

    // Rendu
    glDrawArrays(GL_TRIANGLES, 0, 6);

    // Déverrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, 0);
```

```
// Désactivation des tableaux  
glDisableVertexAttribArray(2);  
glDisableVertexAttribArray(0);  
  
// Désactivation du shader  
glUseProgram(0);  
  
// Actualisation de la fenêtre  
SDL_GL_SwapWindow(m_fenetre);  
  
// Calcul du temps écoulé  
finBoucle = SDL_GetTicks();  
tempsEcoule = finBoucle - debutBoucle;  
  
// Si nécessaire, on met en pause le programme  
if(tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);  
}
```

Cette fois-ci, notre code est complet. 😊 Affichons enfin notre première texture !



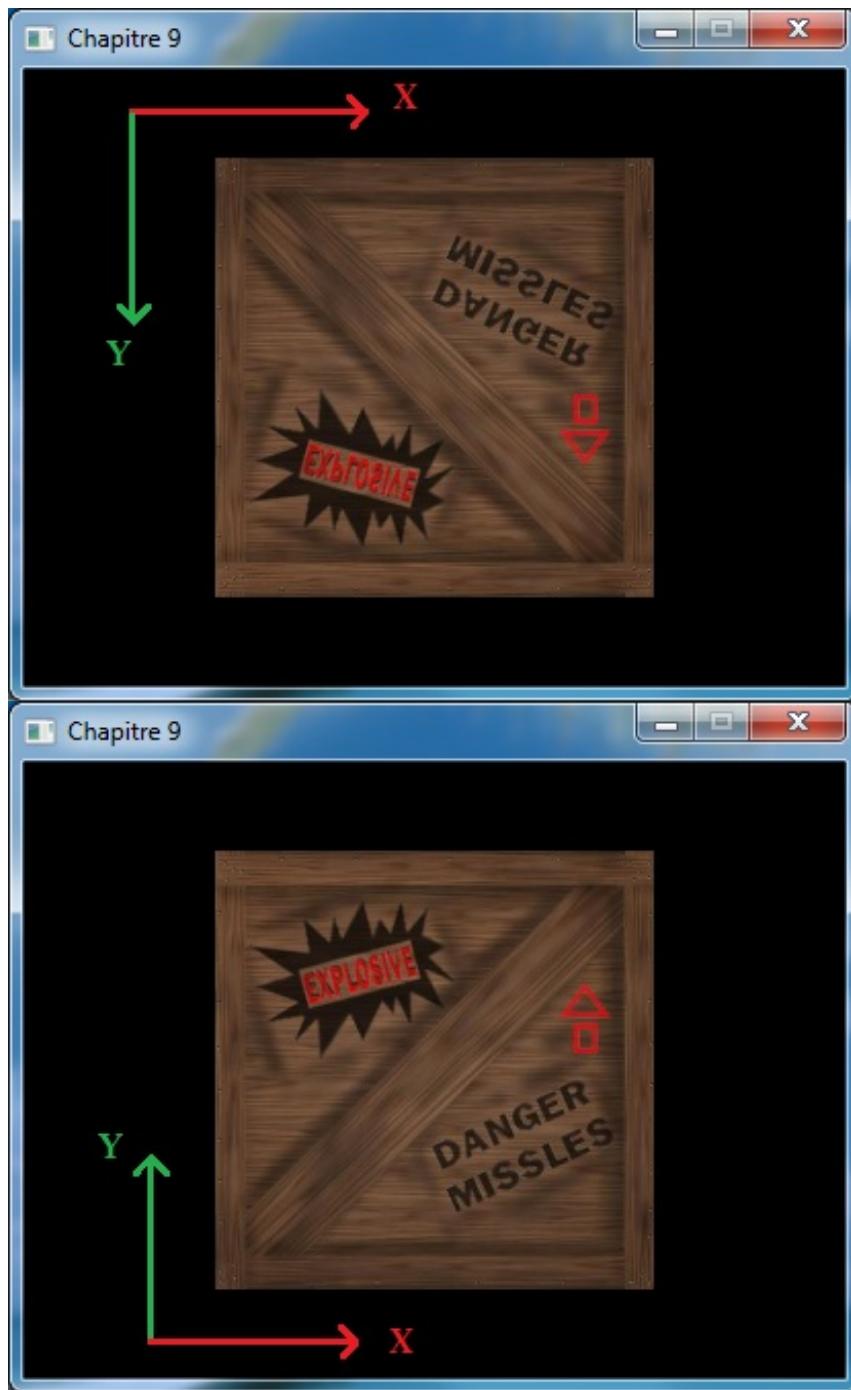
Heeeeein ! 😊😊😊



Pourquoi la texture est inversée ? C'est pourtant les bonnes coordonnées qu'on a envoyées non ? Qu'est-ce qui s'passe ? 😕

C'est tout à fait normal ne vous inquiétez pas. 😊 Tout à l'heure, je vous ai dit qu'on n'en arrêtait là avec le code de la classe **Texture**, et que du coup elle serait incomplète. En effet, la librairie **SDL_image** n'a pas le même repère que nous pour charger les

images. Son repère à elle lui se trouve en haut à gauche alors que le nôtre se situe en bas à droite :



Les textures se retrouvent donc la tête en bas pour nous. Pour corriger ce problème, il nous suffit d'*inverser* tous les pixels d'une image au moment de son chargement. C'est le fameux dernier point qu'il nous manquait dans notre méthode **charger()**. Nous allons régler ça dans un moment.

Mais avant cela, je vous conseille de vous entraîner à afficher quelques textures sur plusieurs carrés pour vous familiariser avec l'affichage de texture. 😊

Améliorations

Si vous n'êtes pas trop fatigués on va pouvoir passer à la dernière partie de ce chapitre. 😊

Les objectifs de cette dernière partie sont :

- De terminer l'implémentation de la méthode **charger()** pour afficher nos textures dans le bon sens.
- D'améliorer la classe **Texture** en y ajoutant des méthodes comme le *constructeur de copie*, l'opérateur `=`, ...

L'inversion des pixels

Théorie

Commençons tout de suite par nous n'occuper du premier point : l'inversion des pixels. Vous avez remarqué (avec effroi) que nos textures s'affichent la tête inversée. Il y a deux façons de régler ce problème :

- Soit on inverse l'ordre de toutes les coordonnées de texture pour afficher l'image à l'envers (mais qui du coup sera dans le bon sens)
- Soit on inverse les pixels de l'image *avant* de la charger côté OpenGL

Comme vous le savez déjà, on va opter pour la seconde solution.

En effet, la première pourrait fonctionner un temps mais une fois que nous chargerons des modèles depuis des fichiers externes il ne sera plus possible d'inverser les coordonnées de texture. Imaginez-vous en train de modifier 20 000 coordonnées de texture à la main. 😱

L'avantage d'inverser les pixels au moment du chargement c'est qu'il nous suffit d'une méthode pour inverser toutes nos textures. Avouez quand même que c'est un gain de temps énorme, si ce n'est astronomique !

Au niveau de la théorie, il faut juste de créer une copie conforme de la **première surface SDL** (celle qui contient l'image) puis d'inverser ses pixels. C'est grâce à une boucle que nous pourrons effectuer cette opération.

Cette boucle devra être capable de copier chaque ligne de la texture et d'inverser la position de celle-ci par rapport à l'axe des ordonnées. Et quand je dis copier chaque ligne, je parle de copier **TOUS** les pixels de la ligne **SANS CHANGER L'ORDRE**. En effet, les pixels présents sur l'axe des abscisses sont, eux, dans le bon ordre, il n'y que sur l'axe des ordonnées que ça ne va pas. Il faut conserver l'ordre des pixels sur chaque ligne mais inverser ces fameuses lignes par rapport à l'axe des ordonnées :



Essayez de bien comprendre cette notion. Revoyez le rendu de texture de la sous-partie précédente et vous verrez que sur l'axe des abscisses les pixels sont dans le bon ordre mais pas sur l'axe des ordonnées.

La méthode inverserPixels

Pour régler notre problème de texture, nous avons donc choisi d'inverser les pixels d'une image avant son chargement du côté OpenGL.

Pour faire ça proprement, on va déclarer une méthode **inverserPixels()** dans laquelle nous placerons notre code d'inversion. Cette méthode devra prendre en paramètre la surface SDL à inverser et elle renverra une autre surface SDL contenant **l'image inversée**.

Code : C++

```
SDL_Surface* inverserPixels(SDL_Surface *imageSource) const;
```



On déclare cette méthode en tant que méthode constante car elle ne modifie en rien les attributs de la classe **Texture**, elle ne fait que renvoyer une surface. 😊

L'implémentation de cette méthode va être rapide, on commence par créer une copie conforme de la surface SDL envoyée en paramètre. Pour créer les surfaces avec la SDL, on va utiliser une fonction que vous devez déjà connaître si vous avez suivi le tuto de M@téo dessus : la fonction **SDL_CreateRGBSurface()** :

Code : C++

```
SDL_Surface* SDL_CreateRGBSurface(Uint32 flags, int width, int height, int depth, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask)
```

Houla ! Ça en fait des paramètres tout ça. Si vous connaissez la SDL vous devriez au moins connaître les 4 premiers. 🍪 Faisons tout de même le point autour des paramètres :

- **flags** : Option permettant notamment de stocker l'image dans la carte graphique. On lui donnera la valeur **0**. Avec la **SDL 1.2**, nous aurions pu lui donner la valeur **SDL_HWSURFACE**, mais celle-ci n'est plus utilisée avec la version **2.0**
- **width** : Largeur de l'image. On lui donnera la valeur du champ **w**
- **height** : Hauteur de l'image. On lui donnera la valeur du champ **h**
- **depth** : Profondeur de l'image, on lui donnera la valeur du champ **BitsPerPixel**, qui n'est pas à confondre avec le champ **BytesPerPixel** attention !
- **Rmask**, **Gmask**, **Bmask**, **Amask** : Paramètres un peu spéciaux qui concerne le masque des couleurs. On leur donnera les champs respectifs de la structure **SDL_Surface**.

Avec cette fonction, nous disposerons d'une nouvelle surface identique à la première, excepté le fait qu'elle ne contiendra encore aucun pixel.

Si on implémente cette fonction, ça nous donne :

Code : C++

```
SDL_Surface* Texture::inverserPixels(SDL_Surface *imageSource) const
{
    // Copie conforme de l'image source sans les pixels

    SDL_Surface *imageInversee = SDL_CreateRGBSurface(0,
imageSource->w, imageSource->h, imageSource->format->BitsPerPixel,
imageSource->format->Rmask,
```

```
imageSource->format->Gmask, imageSource->format->Bmask, imageSource-
>format->Amask);
}
```



Je coupe volontairement les paramètres pour ne pas rendre le code trop long. Mais vous pouvez tout coder sur une seule ligne si ça vous chante.

On a vu dans la partie théorique que l'objectif de cette méthode est de pouvoir copier chaque ligne d'une texture afin d'inverser leur position sur l'axe des ordonnées, ceci à l'aide d'une boucle. En code "grossier" ça va nous donner ça :

Code : C++

```
for(int i(0); i < hauteur; i++)
{
    for(int j(0); j < largeur; j++)
        imageInversee->pixels[hauteur - i][j] = imageSource-
>pixels[i][j];
}
```

Pas de panique je vous explique. 😊

On remarque que grâce à cette boucle, chaque pixel présent sur la largeur de la texture (**à l'indice j**) sera copié exactement au même endroit dans l'image finale **à l'indice j** (vu qu'ils sont dans le bon ordre sur la largeur). Par exemple, un pixel présent en **j = 10ième position** dans l'image source sera copié au même endroit dans l'image finale en **j = 10ième position**.

En revanche, toute la ligne se retrouve inversée par rapport à sa position initiale : la ligne qui était présente en **position i** sur l'image source se retrouvera en **position hauteur - i**. Par exemple prenons une image 256x256, la ligne qui se trouvait en **5ième position** se retrouvera en **[256 - 5] = 251ième position**, la ligne s'est retrouvé en haut de l'image.

Si nous pouvions écrire directement cette boucle dans notre code ça serait le rêve. Cependant il existe un gros problème : le champ **pixels** de la structure *SDL_Surface* n'est pas un tableau à deux dimensions, il est donc impossible d'utiliser les doubles cases **[i][j]**. 😞

Néanmoins, il existe une façon de contrer cela dans les tableaux à une dimension. En effet :

- Un pixel du type **[i][j]** peut être remplacé par un indice du type **[(largeur * i) + j]** et
- Le pixel inverse de type **[hauteur - i][j]** peut être remplacé par **[(largeur * (hauteur - i)) + j]**

Par exemple, si on veut inverser un pixel situé à la position **[5][10]** sur l'image 256x256, on devra :

- Récupérer l'indice source en position **[256 * 5] + j] = [1280 + 10] = [1290]**
- Que l'on copiera dans l'image finale en position **[(256 * (256 - 5)) + 10] = [(256 * 251) + 10] = [64256] = [64266]**

Oui oui, ces chiffres donnent le tournis mais on a bien inversé notre pixel dans le bon sens. 😊

Avec cette méthode, on récupère bien notre pixel même si on est en présence d'un tableau à une dimension. 😊 Du coup, la boucle précédente devient :

Code : C++

```
for(int i(0); i < hauteur; i++)
{
    for(int j(0); j < largeur; j++)
        imageInversee->pixels[(largeur * i) + j] = imageSource-
>pixels[(largeur * (hauteur - i)) + j];
```

On est maintenant prêt pour coder notre méthode. 😊 Dernier petit détail cependant, je vous rappelle qu'en C++ il est malheureusement interdit d'utiliser un tableau contenu dans un pointeur (exemple : `imageSource->pixels[56]`), il va donc falloir passer par des tableaux intermédiaires pour manipuler les pixels :

Code : C++

```
SDL_Surface* Texture::inverserPixels(SDL_Surface *imageSource) const
{
    // Copie conforme de l'image source sans les pixels

    SDL_Surface *imageInversee = SDL_CreateRGBSurface(0,
    imageSource->w, imageSource->h, imageSource->format->BitsPerPixel,
    imageSource->format->Rmask,

    imageSource->format->Gmask, imageSource->format->Bmask, imageSource-
    >format->Amask);

    // Tableau intermédiaires permettant de manipuler les pixels

    unsigned char* pixelsSources = (unsigned char*) imageSource-
>pixels;
    unsigned char* pixelsInverses = (unsigned char*) imageInversee-
>pixels;
}
```



Pour plus de clarté, j'ai déclaré les tableaux sur deux lignes, vous pouvez très bien le faire sur une seule. 😊

Reprendons le grossier code de la boucle pour la convertir en code fonctionnel :

Code : C++

```
// Inversion des pixels

for(int i = 0; i < imageSource->h; i++)
{
    for(int j = 0; j < imageSource->w; j++)
        pixelsInverses[(imageSource->w * (imageSource->h - 1 - i)) +
j] = pixelsSources[(imageSource->w * i) + j];
```

Remarquez le **-1** dans l'indice de l'image inversée `[(image->w * (imageSource->h - 1 - i)) + j]`, je vous rappelle que l'on travaille avec des tableaux donc les indices partent de **0** pour arriver à **hauteur-1**. Je n'en parle que maintenant car je ne voulais pas surcharger la boucle précédente. Déjà que la notion n'est pas évidente à intégrer. 😊

Nous avons pratiquement terminé notre boucle, il ne manque plus qu'une seule chose : jusqu'à maintenant, cette boucle permettait d'échanger des pixels entre eux, mais je vous rappelle qu'un pixel est composé de 3 couleurs. Pour le moment, nous n'échangeons qu'un tiers de la texture, et encore cet échange est complètement buggé !

Pour pallier à ce problème, il suffit juste de *multiplier* par 3 ou par 4 (pour la composante alpha) la boucle qui parcourt la ligne, car chaque ligne du tableau fait en réalité 3 ou 4 fois la longueur de l'image :





Il ne faut surtout pas multiplier la boucle parcourant la hauteur ! Sinon vous vous retrouvez à charger des zones mémoires qui ne vous appartiennent pas. Il n'y a que la boucle de la largeur qui doit être modifiée.

Dans notre code, il suffit de rajouter un "`* imageSource->format->BytesPerPixel`" partout où vous trouvez le champ `imageSource->w`(y compris dans la boucle), soit 3 fois normalement :

Code : C++

```
// Inversion des pixels

for(int i = 0; i < imageSource->h; i++)
{
    for(int j = 0; j < imageSource->w * imageSource->format-
>BytesPerPixel; j++)
        pixelsInverses[(imageSource->w * imageSource->format-
>BytesPerPixel * (imageSource->h - 1 - i)) + j] =
pixelsSources[(imageSource->w * imageSource->format->BytesPerPixel *
i) + j];
}
```

Avec cette boucle, nous pourrons inverser tous les pixels d'une image et ainsi les afficher à l'endroit dans nos scènes 3D. 😊

Pour terminer la méthode, il ne nous manque plus qu'à retourner la nouvelle surface SDL :

Code : C++

```
// Retour de l'image inversée

return imageInversee;
```

Ce qui nous donne au final :

Code : C++

```
SDL_Surface* Texture::inverserPixels(SDL_Surface *imageSource) const
{
    // Copie conforme de l'image source sans les pixels

    SDL_Surface *imageInversee = SDL_CreateRGBSurface(0,
imageSource->w, imageSource->h, imageSource->format->BitsPerPixel,
imageSource->format->Rmask,

imageSource->format->Gmask, imageSource->format->Bmask, imageSource-
>format->Amask);

    // Tableau intermédiaires permettant de manipuler les pixels

    unsigned char* pixelsSources = (unsigned char*) imageSource-
>pixels;
    unsigned char* pixelsInverses = (unsigned char*) imageInversee-
>pixels;

    // Inversion des pixels

    for(int i = 0; i < imageSource->h; i++)
    {
        for(int j = 0; j < imageSource->w * imageSource->format-
>BytesPerPixel; j++)
```

```

        pixelsInverses[(imageSource->w * imageSource->format-
>BytesPerPixel * (imageSource->h - 1 - i)) + j] =
pixelsSources[(imageSource->w * imageSource->format->BytesPerPixel *
i) + j];
    }

    // Retour de l'image inversée

    return imageInversee;
}

```

Adaptation dans la méthode charger

Maintenant que nous sommes capables d'inverser le sens d'une image, nous pouvons intégrer la nouvelle méthode dans le chargement de texture. Il nous suffit d'appeler la méthode **inverserPixels()** juste après avoir chargé la surface SDL. Bien évidemment, on pense à détruire l'ancienne surface qui ne sert plus rien :

Code : C++

```

bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL

    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        printf("IMG_Load: %s\n", IMG_GetError());
        return false;
    }

    // Inversion de l'image

    SDL_Surface *imageInversee = inverserPixels(imageSDL);
    SDL_FreeSurface(imageSDL);

    // ....
}

```

Il ne reste plus qu'à remplacer toutes les occurrences du pointeur **imageSDL** par le pointeur **imageSDLInversee** (Vous devriez trouver 9 occurrences à remplacer). Ce qui donne au final :

Code : C++

```

bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL

    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }
}

```

```
// Inversion de l'image

SDL_Surface *imageInversee = inverserPixels(imageSDL);
SDL_FreeSurface(imageSDL);

// Génération de l'ID

glGenTextures(1, &m_id);

// Verrouillage

glBindTexture(GL_TEXTURE_2D, m_id);

// Format de l'image

GLenum formatInterne(0);
GLenum format(0);

// Détermination du format et du format interne pour les images
à 3 composantes

if(imageInversee->format->BytesPerPixel == 3)
{
    // Format interne

    formatInterne = GL_RGB;

    // Format

    if(imageInversee->format->Rmask == 0xff)
        format = GL_RGB;

    else
        format = GL_BGR;
}

// Détermination du format et du format interne pour les images
à 4 composantes

else if(imageInversee->format->BytesPerPixel == 4)
{
    // Format interne

    formatInterne = GL_RGBA;

    // Format

    if(imageInversee->format->Rmask == 0xff)
        format = GL_RGBA;

    else
        format = GL_BGRA;
}

// Dans les autres cas, on arrête le chargement

else
{
    std::cout << "Erreur, format interne de l'image inconnu" <<
    std::endl;
    SDL_FreeSurface(imageInversee);
```

```
        return false;
    }

    // Copie des pixels

    glTexImage2D(GL_TEXTURE_2D, 0, formatInterne, imageInversee->w,
imageInversee->h, 0, format, GL_UNSIGNED_BYTE, imageInversee-
>pixels);

    // Application des filtres

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

    // Déverrouillage

    glBindTexture(GL_TEXTURE_2D, 0);

    // Fin de la méthode

    SDL_FreeSurface(imageInversee);
    return true;
}
```

Et voilà nous pouvons enfin charger des textures dans le bon ordre. 😊 Essayez avec la texture de la caisse de missiles, vous verrez qu'elle s'affiche désormais à l'endroit :



Les méthodes additionnelles

On va terminer tranquillement ce chapitre par quelques méthodes importantes qu'il faut implémenter, à savoir : le constructeur par défaut, celui de copie, le destructeur et la surcharge de l'opérateur =.

Ces méthodes sont importantes pour assurer le bon cycle de vie de la classe **Texture**. Le constructeur par défaut par exemple est utile lorsque vous souhaitez initialiser tout un tableau de textures, l'opérateur = quant à lui sert à copier une texture facilement en une ligne de code, etc ...

Le constructeur par défaut et le destructeur

On va commencer tout de suite par implémenter le *constructeur par défaut* et le *destructeur*.

Je vous rappelle que le *constructeur par défaut* est appelé lorsque vous ne donnez aucun paramètre à votre objet. Son prototype est le suivant :

Code : C++

```
Texture();
```

Son rôle ne lui permet que d'initialiser les deux attributs de la classe. Il va affecter l'**ID 0** à l'attribut **m_id** et une chaîne vide à **m_fichierImage** :

Code : C++

```
Texture::Texture() : m_id(0), m_fichierImage("")  
{  
}
```

Occupons maintenant du *destructeur*. Cette "méthode" est appelée à chaque fois qu'un objet est détruit, c'est notamment ici que nous devrons détruire nos textures. Car oui, même les *objets OpenGL* doivent être détruits pour libérer la mémoire qu'ils occupent.

Les *objets OpenGL* sont alloués dynamiquement dans la carte graphique par la fonction **glGenTextures()**, il va nous falloir libérer l'espace occupé pour éviter les fuites de mémoire. Pour ça, OpenGL nous fournit une fonction qui reprend exactement les mêmes paramètres que **glGenTextures()**. Voici son prototype :

Code : C++

```
void glDeleteTextures(GLsizei number, const GLuint *textures);
```

- **number** : Le **nombre d'ID** à initialiser. Comme pour la génération, nous lui donnerons la valeur **1**
- **textures** : Un tableau de type **GLuint** ou une adresse d'**ID**. Nous lui donnerons l'adresse de l'**ID** à détruire

Nous utiliserons cette fonction dans notre *destructeur* dont j'espère que vous connaissez son utilité. 😊

Il est appelé au moment de la destruction d'un objet pour permettre au développeur de libérer toute la mémoire qu'il avait prise. D'ailleurs, nous avions déjà déclaré cette "méthode" précédemment, il ne nous reste donc plus qu'à utiliser la fonction **glDeleteTextures()** à l'intérieur :

Code : C++

```
Texture::~Texture()  
{  
    // Destruction de la texture  
  
    glDeleteTextures(1, &m_id);
```

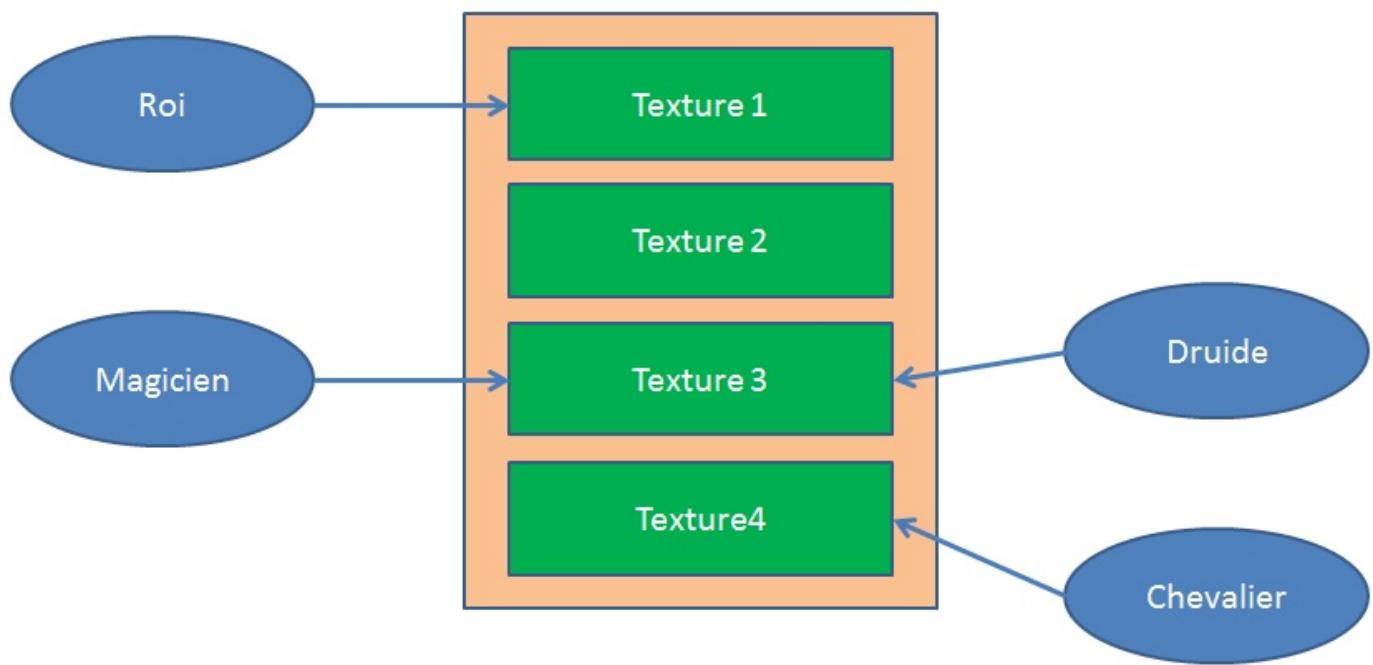
}

Et voilà, nos textures seront maintenant détruites proprement lorsqu'elles ne seront plus utilisées. 😊

Copier une texture

On passe maintenant à la copie de textures. Mais avant ça, je vais d'abord vous faire une petite parenthèse sur le chargement des textures dans un vrai jeu-vidéo.

Dans un jeu-vidéo lambda, le moteur 3D doit être capable de charger toutes les textures dont il a besoin dans un espèce de gros tableau qui doit être accessible par toutes les classes :



A partir de là, chaque modèle 3D (comme un personnage ou un arbre) va récupérer un pointeur sur la texture dont il aura besoin, ce n'est donc pas lui qui va charger sa propre texture. Et de plus, si vous avez une dizaine d'ennemis qui doit s'afficher à l'écran, vous n'allez pas charger 10 fois les mêmes textures pour les 10 ennemis. En théorie, chaque personnage devrait récupérer un pointeur sur la texture dont il a besoin dans le gros tableau pour l'utiliser. De cette façon, les textures ne sont chargées qu'une seule fois et ça économise pas mal de ressources.

Je vous parle de ça maintenant car c'est ce que nous "*devrions faire*" en réalité. Cependant nos scènes étant encore assez simples nous pouvons nous permettre de charger une texture plusieurs fois sans se préoccuper de créer le gros tableau. 😊

Bien, ceci étant dit on va pouvoir retourner à nos méthodes de copie.

Il existe grossièrement deux façons de copier une texture : soit on utilise le constructeur de copie, soit on surcharge l'opérateur =. La différence entre les deux est que si vous initialisez une texture à partir d'une autre texture, alors ce sera le constructeur de copie qui sera appelé. En revanche, si vous le faites après sa déclaration se sera l'opérateur = qui sera sollicité.

On va commencer l'implémentation de ces deux méthodes par le constructeur de copie dont voici le prototype :

Code : C++

```
Texture(Texture const &textureACopier);
```



Vu que l'on commence à bien manipuler les objets, nous utiliserons de plus en plus les références dans nos programmes. 😊

Le constructeur prend en paramètre une **référence constante** sur l'objet à copier, dans notre cas une texture.

Petit point important : si nous avons besoin de coder nous-même ce constructeur c'est parce qu'il y a, en général, un problème au niveau des pointeurs lors de la copie d'un objet. En effet, si on dispose de deux copies d'une même texture alors elles partageront le même **ID**. Or, si on détruit l'une des deux textures, la deuxième se retrouvera avec un **ID invalide** et donc une erreur d'affichage.

Alors bon, il est vrai que dans notre classe l'**ID** n'est qu'une simple variable. Mais pour votre carte graphique cet **ID** représente un pointeur pointant sur la texture chargée.

Par conséquent, pour copier proprement deux textures il va falloir créer un nouvel **ID** indépendant du premier. De cette façon, même si un objet est détruit, la copie ne sera pas affectée.

Dans la classe **Texture**, nous devrons donc :

- Copier l'attribut **m_fichierImage** qui contient le chemin vers le fichier image
- **Recharger** la texture pour avoir un nouvel **ID** indépendant du premier

Le constructeur de copie ressemblera donc à ceci :

Code : C++

```
Texture::Texture(Texture const &textureACopier)
{
    // Copie de la texture

    m_fichierImage = textureACopier.m_fichierImage;
    charger();
}
```

Rien de plus facile isn't it ? 😊

D'ailleurs on va s'occuper maintenant de coder la surcharge de l'opérateur = car le code ne change pas vraiment. Le prototype de cette surcharge sera le suivant :

Code : C++

```
Texture& operator=(Texture const &textureACopier);
```



La méthode retournera bien une référence sur un objet de type **Texture**. Cette référence concernera d'ailleurs l'objet lui-même (pointeur **this**).

Pour cette méthode, on copie simplement le même code que le constructeur de copie puis on renvoie le pointeur ***this** :

Code : C++

```
Texture& Texture::operator=(Texture const &textureACopier)
{
```

```
// Copie de la texture
m_fichierImage = textureACopier.m_fichierImage;
charger();

// Retour du pointeur *this
return *this;
}
```

Il manque encore un tout petit détail à cette méthode - un détail qui s'étend même au chargement en général. Imaginez qu'une texture soit déjà chargée et que l'on copie une autre texture dans celle-ci, que se passerait-il ?

Et bien le premier **ID** qui a été initialisé sera perdu, et du coup la texture qui a été chargée en mémoire sera elle aussi perdue. C'est encore une fuite. Pour régler ce petit problème, il faut simplement détruire l'ancien **ID** de la même façon qu'avec le destructeur. Dans le pire des cas, même si la texture n'était pas chargée avant, ce sera l'**ID 0** qu'OpenGL essaiera de détruire. Mais vu que c'est impossible il passera à autre chose sans nous renvoyer d'erreur.

Si j'ai dit que ce problème s'étendait aussi au chargement de texture en général, c'est parce que nous avons exactement le même problème lorsque nous utilisons la méthode **charger()**. En effet, si nous utilisons cette méthode deux fois sur le même objet, alors le premier chargement qui a été fait sera perdu en mémoire.

Pour régler ce problème, nous allons rajouter un petit bout de code dans la méthode **charger()** juste avant de générer l'identifiant. Ce code sera constitué d'une condition **if** qui permettra de savoir si une texture a déjà été chargée. Si oui, alors il faudra la détruire avant de la recharger.

La fonction qui permet de savoir ceci s'appelle **glIsTexture()** :

Code : C++

```
GLboolean glIsTexture(GLuint texture);
```

Elle ne prend en paramètre que l'identifiant de la texture à vérifier. Elle renvoie la valeur **GL_TRUE** si elle a déjà été chargée et **GL_FALSE** dans le cas contraire.

Nous devons donc appeler cette fonction avant de générer l'**ID**. Si elle renvoie **GL_TRUE** alors il faudra détruire la texture à l'aide de **glDeleteTextures()** :

Code : C++

```
bool Texture::charger()
{
    // Chargement de l'image dans une surface SDL
    SDL_Surface *imageSDL = IMG_Load(m_fichierImage.c_str());

    if(imageSDL == 0)
    {
        std::cout << "Erreur : " << SDL_GetError() << std::endl;
        return false;
    }

    // Inversion de l'image
    SDL_Surface *imageInversee = inverserPixels(imageSDL);
    SDL_FreeSurface(imageSDL);

    // Destruction d'une éventuelle ancienne texture
```

```
if(glIsTexture(m_id) == GL_TRUE)  
glDeleteTextures(1, &m_id);
```

.....

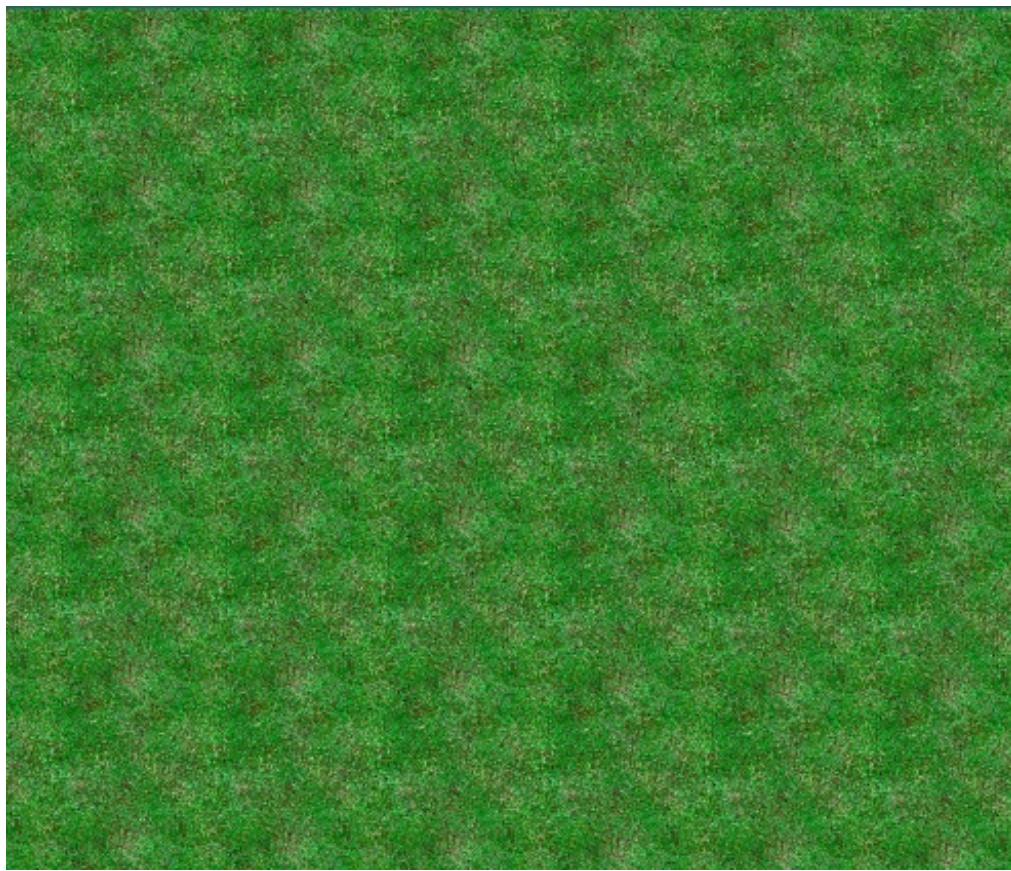
Et voilà, nos méthodes de copie sont maintenant complètes et prêtes à l'emploi. 😊

Aller plus loin Répéter une texture

Fonctionnement

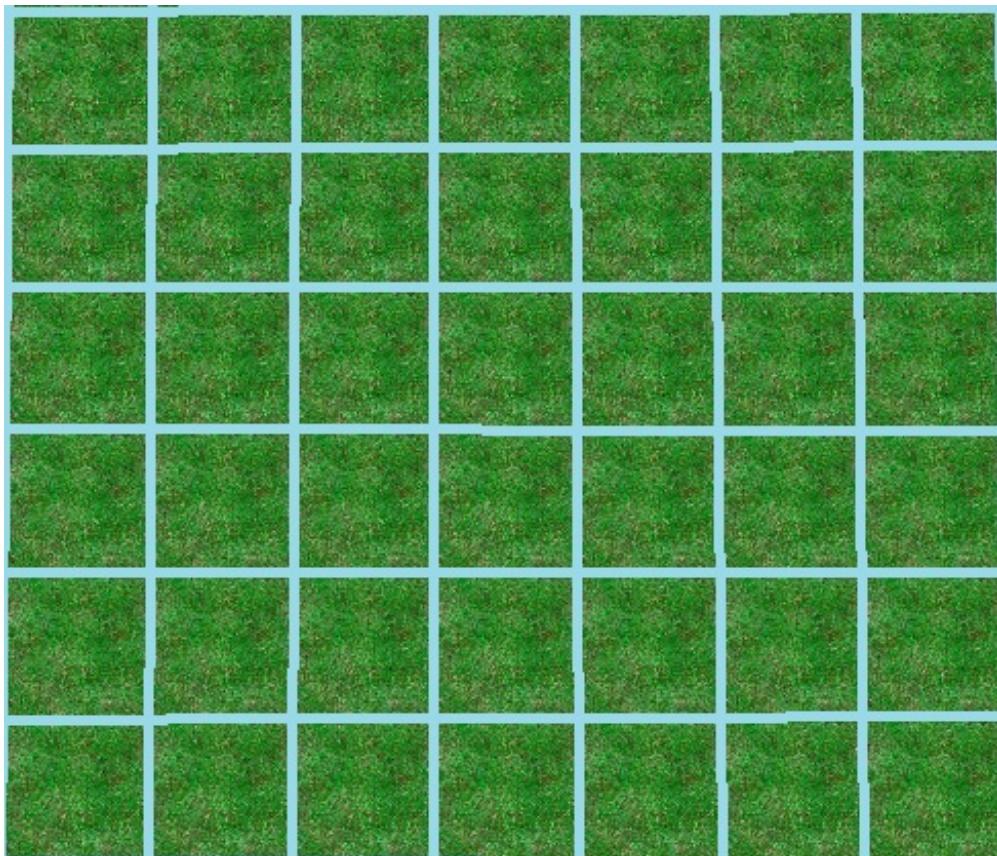
Maintenant que l'on peut afficher une texture dans le bon sens, je vais vous montrer la technique utilisée pour répéter une texture plusieurs fois. Si vous avez déjà joué à un jeu-vidéo, vous avez peut-être remarqué que les textures au sol se répétait en boucle.

On peut prendre l'exemple d'un sol composé d'herbe :



Si vous vous concentrez sur cette image, vous remarquerez qu'elle est composée d'une seule petite texture qui se répète en boucle.

Si un jeu-vidéo devait afficher ce sol et qu'il s'amuserait à charger la même texture pour chaque carré d'herbe, alors il deviendrait totalement injouable et serait beaucoup trop lent. A la place, il se contentera de la charger une seule fois puis il utilisera des coordonnées de texture '*spéciales*' pour la répéter :



Cette technique est utilisée par tous les jeux-vidéo. Sans elle, il n'y en aurait pas beaucoup d'ailleurs car ils seraient tous terriblement lents à l'affichage. 🍔

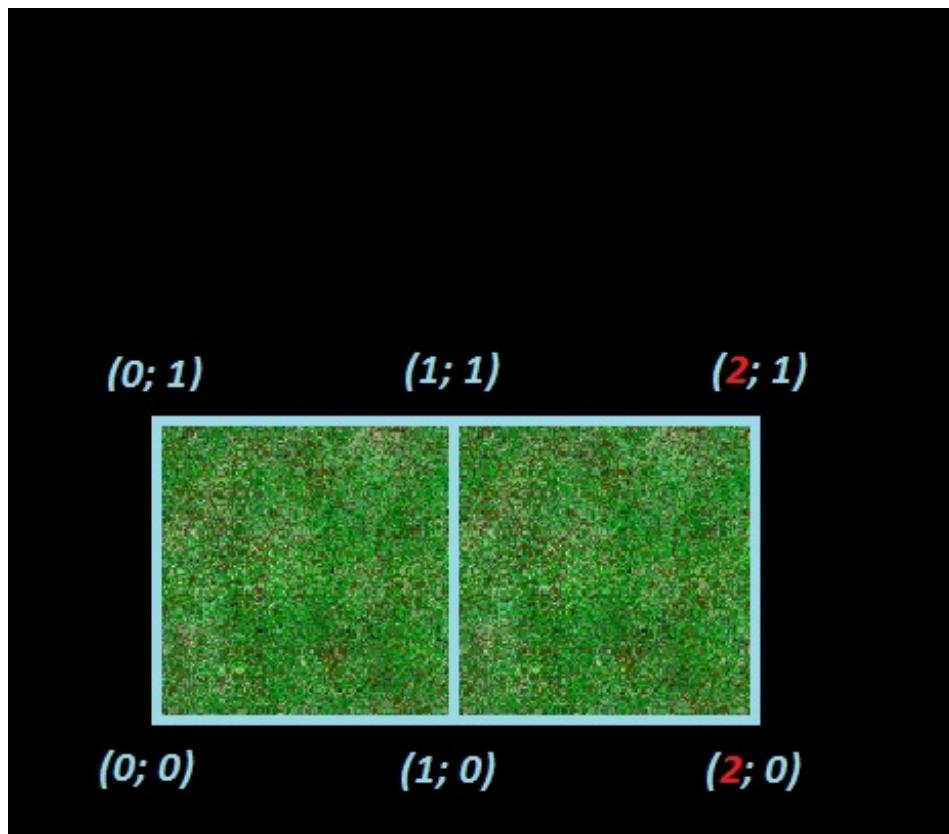
Bon en réalité les coordonnées utilisées n'ont rien de *spéciales*, elles ne sont juste pas comprises entre **0** et **1**.



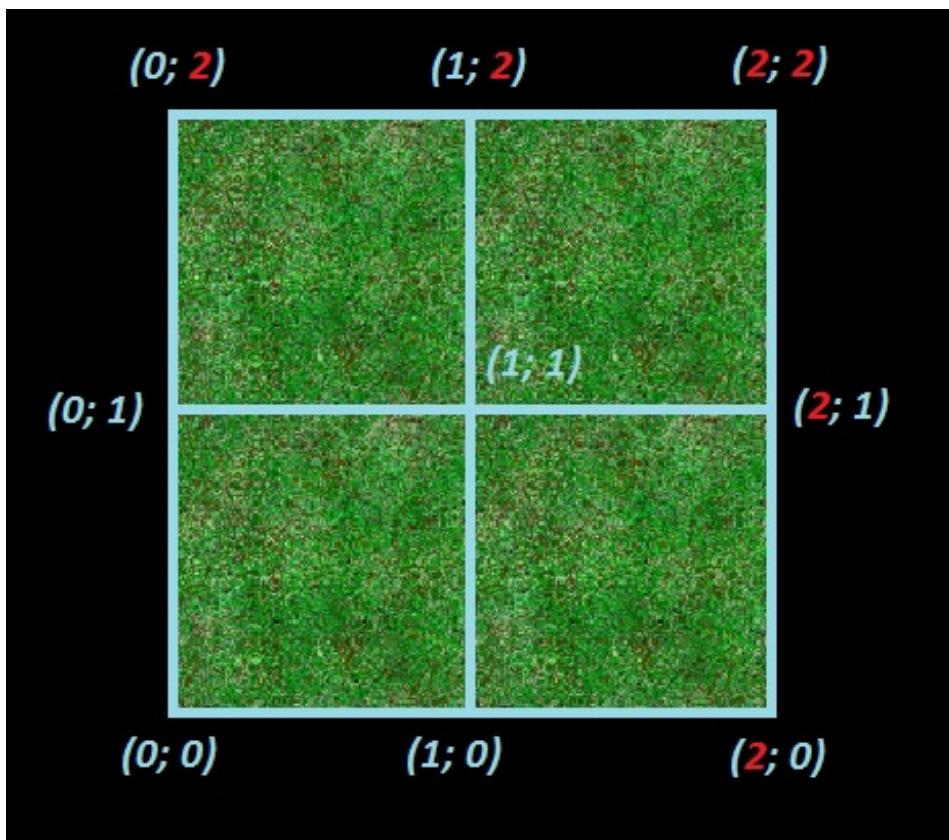
Hein ? Mais tu as dit que les coordonnées de texture devaient *toujours* être comprises entre **0** et **1** ?

Oui et c'est totalement vrai !

Si vous allez au-delà de **1** (**2** par exemple) ou en dessous de **0** alors vous répéterez la même texture :



Les coordonnées sont toujours comprises entre **0** et **1** mais au lieu de renvoyer une erreur, OpenGL recopiera la texture. Bien évidemment, plus vous fournissez des valeurs importantes, plus votre texture sera répétée :



Ce n'est absolument pas un bug puisque c'est prévu par OpenGL et c'est même **LA** technique à utiliser pour répéter vos textures

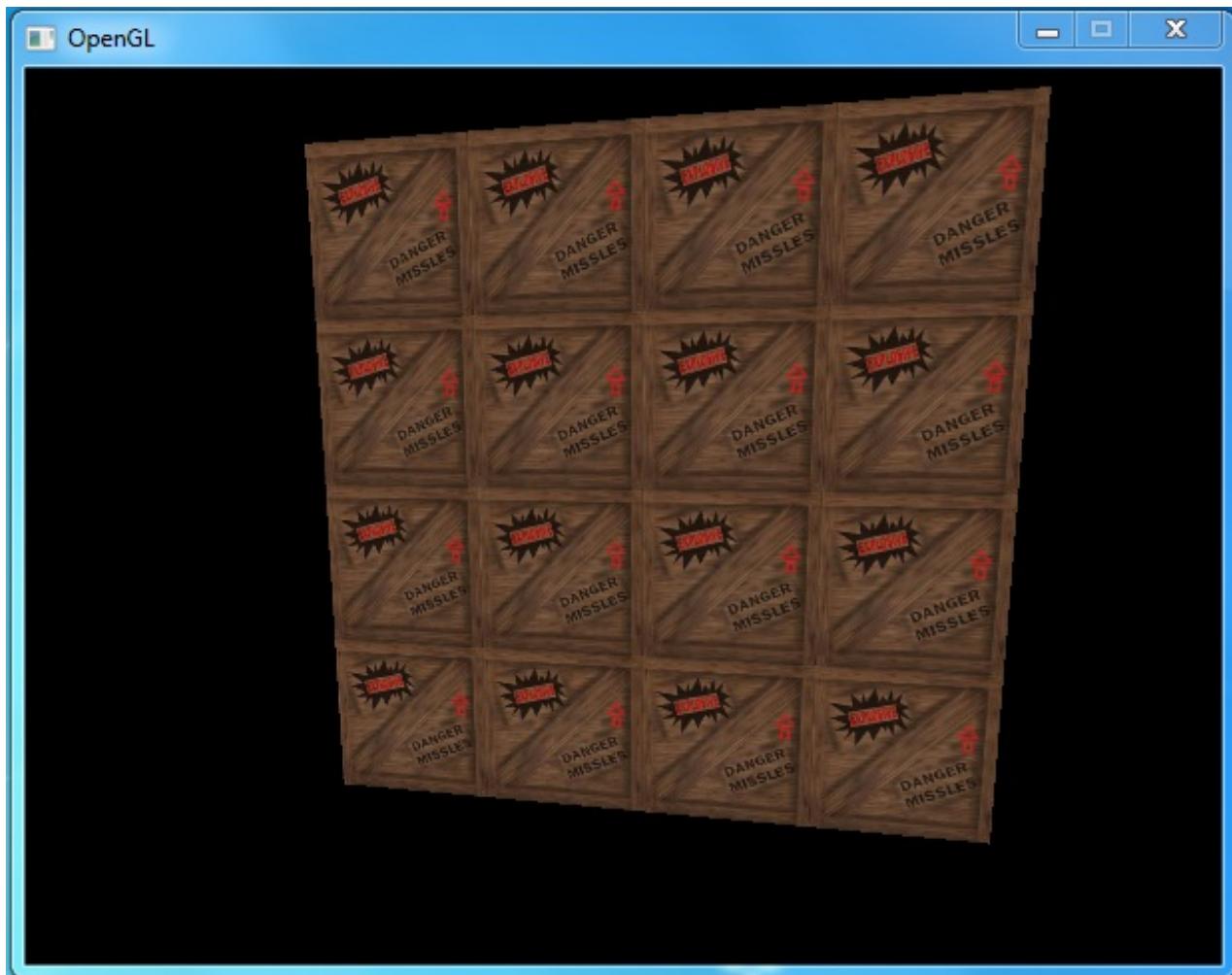
que ce soit sur les sols, les murs, ... Nous ferons comme ça dans le futur pour afficher de grands espaces. 😊

Le seul point à prendre en compte c'est qu'il faut que votre texture soit prévue pour la répétition, si elle ne l'est pas vous aurez un rendu assez moche et pas continu. En général, ce point concerne surtout le graphiste et non le développeur (une bonne raison de lui râler dessus si ça ne va pas 😱)

Vous pouvez essayer de voir ce que ça donne avec la texture que l'on utilise dans ce chapitre. Remplacez les coordonnées par celles-ci pour l'afficher 12 fois !

Code : C++

```
// Coordonnées de texture  
  
float coordTexture[] = {0, 0,     4, 0,     4, 4,      // Triangle 1  
                        0, 0,     0, 4,     4, 4};      // Triangle 2
```



Exercice

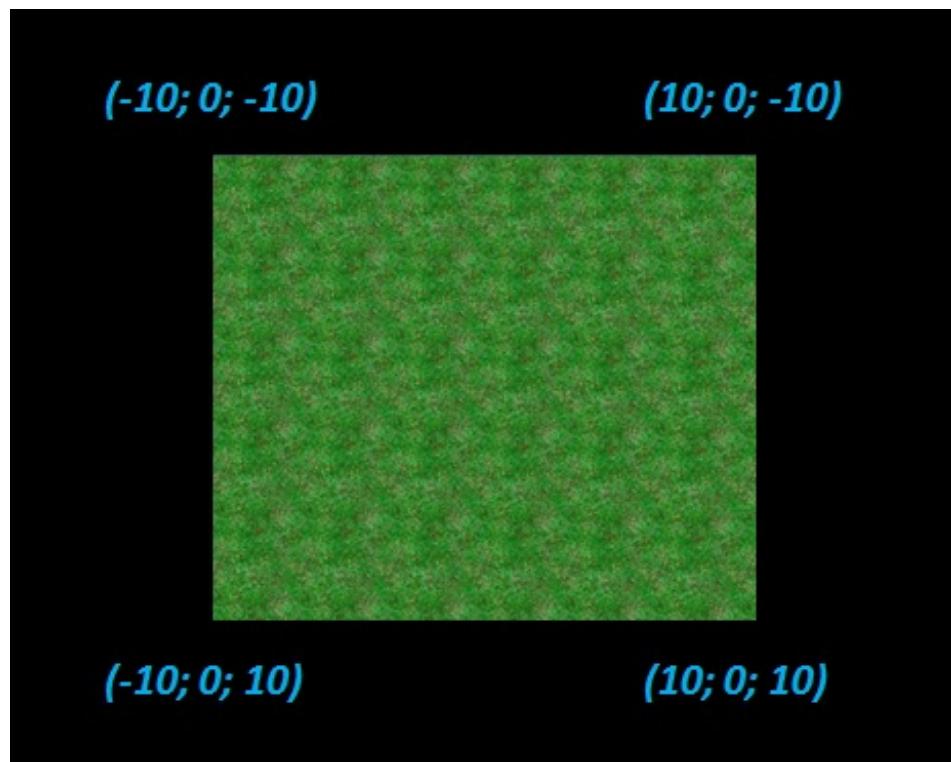
La répétition de texture est une occasion parfaite pour moi de vous proposer un petit exercice à faire.

En reprenant ce qu'on vient de voir, essayez de recréer le sol constitué d'herbe que je vous ai montré au-dessus. Les consignes sont les suivantes :

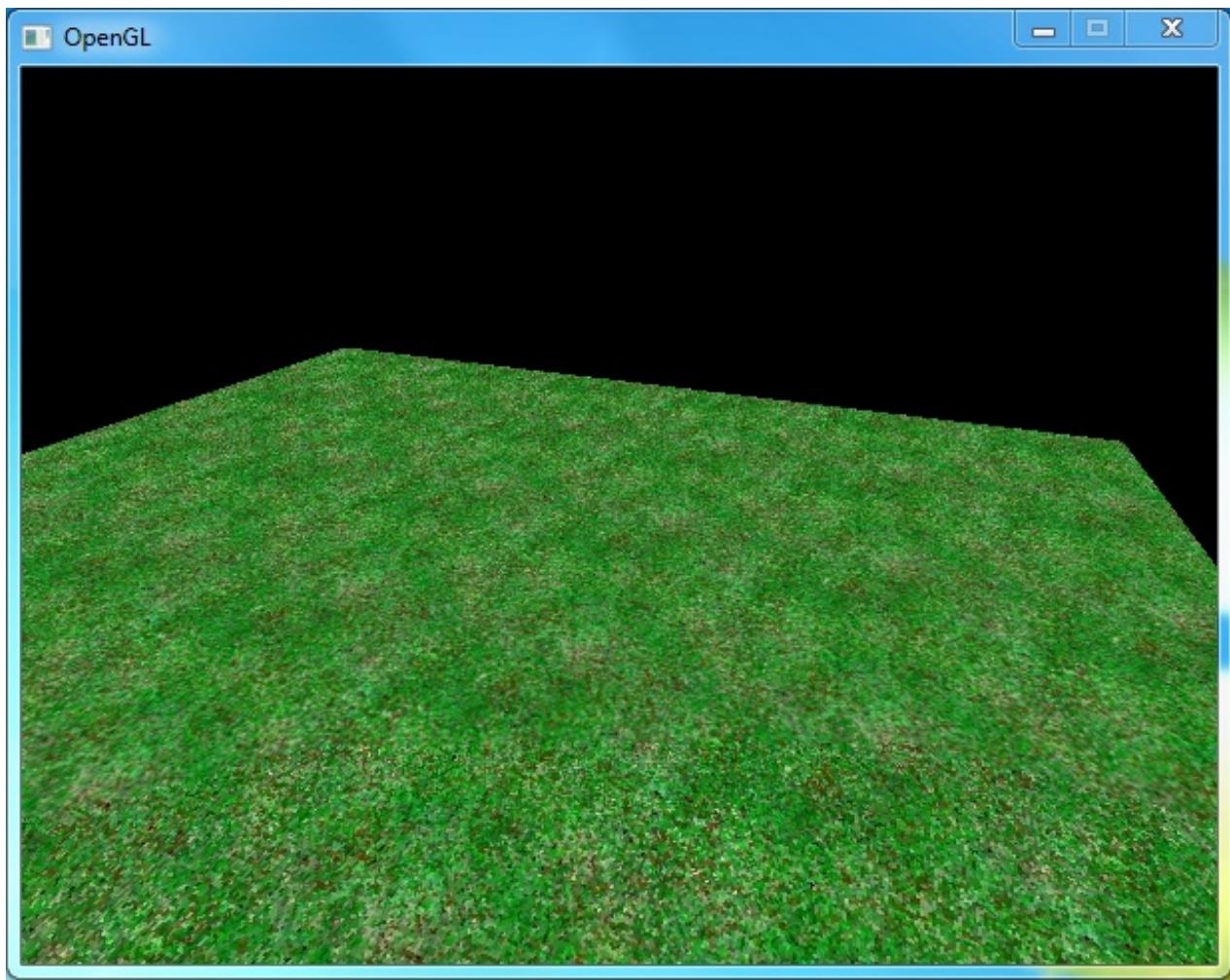
- Vous ne devez utiliser que **4** vertices (fournis dans le schéma ci-dessous)
- Répéter la texture **49** fois (7 fois en largeur et en hauteur)

Vous pouvez télécharger l'image à afficher un peu plus haut ou si vous avez téléchargé l'archive, vous la retrouverez sous le nom de "veg005.jpg".

Voici le schéma explicatif :



Et voici ce que vous devriez obtenir (en plaçant votre caméra un peu différemment pour mieux voir) :



Bonne chance. 😊 (et n'oubliez pas de verrouiller votre texture !)

Solution

Secret (cliquez pour afficher)

La solution de cette exercice est assez simple, nous n'avons que quelques lignes de code à modifier par rapport à l'exemple que l'on a vu.

En premier, il fallait définir les vertices. Nous faisons des carrés depuis un moment maintenant il n'aurait pas dû y avoir de problèmes :

Code : C++

```
// Vertices

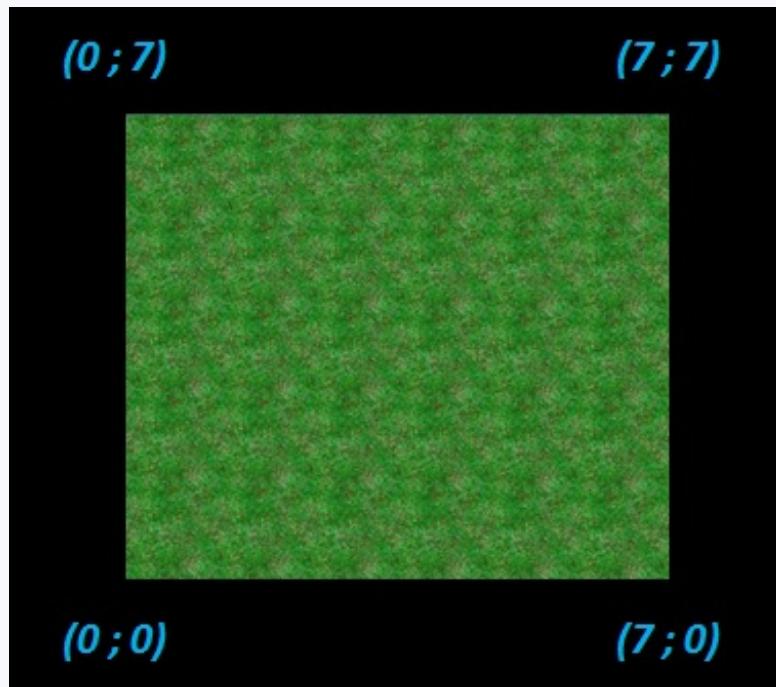
float vertices[] = {-10, 0, -10,    10, 0, -10,    10, 0, 10,    //
Triangle 1
                    -10, 0, -10,    -10, 0, 10,    10, 0, 10}; // 
Triangle 2
```

N'oubliez pas qu'il fallait afficher un sol, les vertices étaient donc tous à la hauteur 0.

Ensuite, il fallait définir les coordonnées de texture. Nous avons vu que si on allait plus que 1 dans leurs valeurs alors on attaquait un autre affichage de la même texture. Il fallait l'afficher 7 fois en largeur et en hauteur, donc les coordonnées

maximales à utiliser étaient respectivement [7 ; 0] et [0 ; 7].

On peut représenter la situation avec le schéma suivant :



Le tableau **coordTexture** à utiliser était donc le suivant :

Code : C++

```
// Coordonnées de texture  
float coordTexture[] = {0, 0,    7, 0,    7, 7,      // Triangle 1  
                         0, 0,    0, 7,    7, 7};      // Triangle 2
```

Une fois les données définies, il ne manquait plus qu'à déclarer un objet Texture qui permettait de charger l'image en mémoire :

Code : C++

```
// Vertices et coordonnées de texture  
....  
  
// Texture  
Texture texture("Textures/Herbe.jpg");  
texture.charger();
```

Au niveau du code d'affichage, il fallait juste reprendre celui que l'on utilisait précédemment en faisant attention de bien verrouiller la texture au moment d'appeler la fonction **glDrawArrays()**. Le reste du code permettait, entre autres, d'envoyer les vertices, les coordonnées de texture et les matrices à OpenGL :

Code : C++

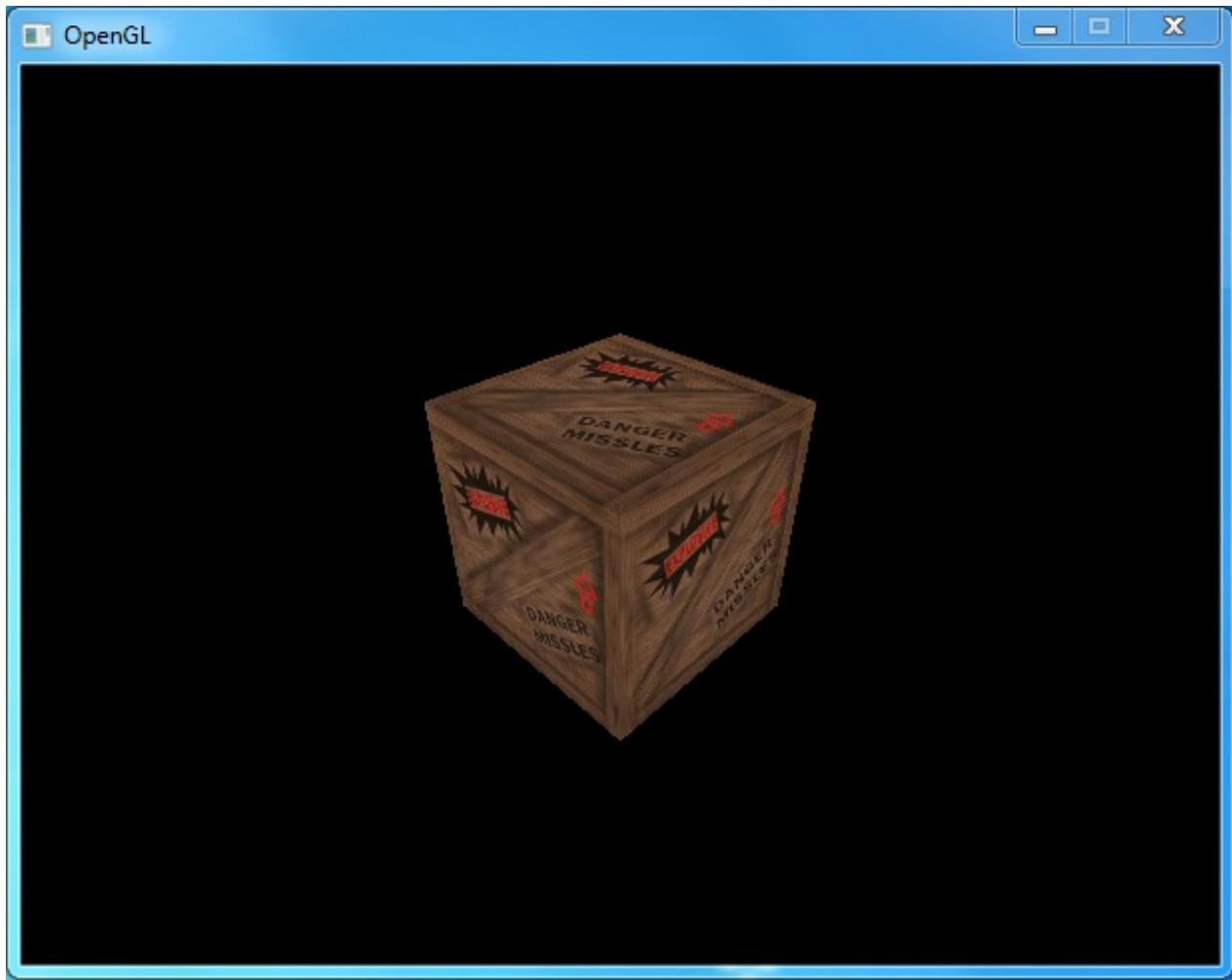
```
// Activation du shader
```

```
glUseProgram(shaderTexture.getProgramID());  
  
    // Envoi des vertices  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);  
    glEnableVertexAttribArray(0);  
  
    // Envoi des coordonnées de texture  
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);  
    glEnableVertexAttribArray(2);  
  
    // Envoi des matrices  
  
    glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),  
        "projection"), 1, GL_FALSE, value_ptr(projection));  
    glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),  
        "modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
    // Verrouillage de la texture  
    glBindTexture(GL_TEXTURE_2D, texture.getID());  
  
    // Rendu  
    glDrawArrays(GL_TRIANGLES, 0, 6);  
  
    // Déverrouillage de la texture  
    glBindTexture(GL_TEXTURE_2D, 0);  
  
    // Désactivation des tableaux  
    glDisableVertexAttribArray(2);  
    glDisableVertexAttribArray(0);  
  
    // Désactivation du shader  
    glUseProgram(0);
```

Afficher une caisse

Pour clore ce chapitre d'une façon intéressante, nous allons appliquer ce que l'on a vu sur les textures non plus sur une forme simple 2D mais bien sur un modèle 3D comme un cube. Vu que nous avons utilisé une texture de caisse jusqu'ici, nous pouvons légitimement l'utiliser pour transformer notre cube coloré à l'ancienne en super caisse de missiles. 😊

Notre objectif final est d'être capables d'avoir ce rendu :



Pour arriver à cela, nous allons coder une classe **Caisse** qui contiendra tout ce qu'il faut pour afficher un cube et le texturer. Pour nous faciliter la tâche, nous la ferons hériter de la classe **Cube** afin de ne pas avoir à refaire l'initialisation des vertices et du shader. 😊

L'héritage de la classe Cube

Avant de se lancer dans la classe **Caisse**, nous allons devoir régler un petit problème d'héritage au niveau du cube qui nous empêche de profiter de ses attributs. En effet, si on regarde son header, on remarque que ces derniers sont tous **privés** à cause de l'utilisation du mot-clef **private** :

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE

// Includes
...

// Classe Cube

class Cube
{
public:
    Cube(float taille, std::string const vertexShader, std::string
const fragmentShader);
```

```
~Cube () ;  
  
private:  
    Shader m_shader;  
    float m_vertices[108];  
    float m_couleurs[108];  
};  
  
#endif
```

Pour prétendre à un héritage, il faut changer ce mot-clé en le remplaçant par son frère **protected** :

Code : C++

```
#ifndef DEF_CUBE  
#define DEF_CUBE  
  
// Includes  
....  
  
// Classe Cube  
  
class Cube  
{  
public:  
    Cube(float taille, std::string const vertexShader, std::string  
const fragmentShader);  
    ~Cube();  
  
protected:  
    Shader m_shader;  
    float m_vertices[108];  
    float m_couleurs[108];  
};  
  
#endif
```

De cette façon, nous pouvons utiliser les trois attributs présents ici dans les classes filles.

Le gros avantage pour nous, c'est que le shader et surtout les vertices seront déjà initialisés sans même que nous ayons à faire quoi que ce soit. Il ne nous restera plus qu'à gérer les nouveaux attributs. Entre parenthèses, le tableau de couleurs ne sera pas utile pour la suite mais mieux vaut le garder sous la main au cas où.

Le header de la classe *Caisse*

Comme toute implémentation de classe, nous devons commencer par le header. Celui de la classe **Caisse** sera assez similaire à celui du **Cube** puisqu'il comportera les mêmes méthodes ainsi qu'un tableau à initialiser dans le constructeur.

Le début du fichier, que nous appellerons **Caisse.h**, contiendra la définition de la classe accompagnée du code "**public Cube**" qui lui permettra de profiter de l'héritage :

Code : C++

```
#ifndef DEF_CAISS
#define DEF_CAISS

// Includes

#include "Cube.h"

// Classe Caisse

class Caisse : public Cube
{
public:

private:
};

#endif
```

Pour agrémenter un peu ce code, nous allons ajouter les attributs nécessaires à la réalisation de notre caisse. Sachant que les vertices et le shader sont déjà fournis, nous n'en avons donc besoin que de deux supplémentaires :

- Un objet **Texture** qui représentera l'image à plaquer
- Un tableau de **float** qui représentera les coordonnées à associer aux vertices

Petit précision pour le tableau, celui-ci contiendra **72** cases car il faut associer **2** coordonnées à chaque vertex, ce qui donne **36 vertices x 2 coordonnées = 72 cases**.

Code : C++

```
// Attributs

Texture m_texture;
float m_coordTexture[72];
```

Passons maintenant aux méthodes dont nous aurons besoin. Hormis celle nécessaire pour afficher le rendu final, nous n'aurons besoin que du constructeur et du destructeur.

Le premier reprendra les mêmes paramètres que celui de la classe **Cube** afin de pouvoir lui donner les valeurs qu'il attend au moment de l'initialisation. Ces paramètres représentaient la taille désirée ainsi que les codes sources du shader à utiliser.

Nous en rajouterons un dernier, spécifique au constructeur **Caisse()**, qui représentera l'image à plaquer sur les faces de la caisse et qui permettra surtout d'initialiser l'attribut **m_texture**.

Le prototype du constructeur au final est le suivant :

Code : C++

```
Caisse(float taille, std::string const vertexShader, std::string
const fragmentShader, std::string const texture);
```

On en profite au passage pour déclarer le destructeur (qui est quand même plus simple à faire 😊) :

Code : C++

```
~Caisse();
```

Si on réunit tout ça dans le header, on devrait avoir :

Code : C++

```
#ifndef DEF_CAISS
#define DEF_CAISS

// Includes

#include "Cube.h"
#include "Texture.h"
#include <string>

// Classe Caisse

class Caisse : public Cube
{
public:
    Caisse(float taille, std::string const vertexShader, std::string
const fragmentShader, std::string const texture);
    ~Caisse();

private:
    Texture m_texture;
    float m_coordTexture[72];
};

#endif
```



Pensez à inclure les en-têtes **Texture.h** et **string** sinon votre compilateur risque de ne pas être d'accord avec vous.

Le constructeur

Le constructeur va être un poil différent de ceux que nous avons l'habitude d'implémenter. En effet, avant d'initialiser n'importe quel attribut, nous devons faire appel au constructeur parent de la classe **Caisse** comme nous le demande le C++.

Nous l'appelons donc juste après les deux points ":" en lui passant les trois premiers paramètres reçus (la taille et les codes sources shader) :

Code : C++

```
Caisse::Caisse(float taille, std::string const vertexShader,
std::string const fragmentShader, std::string const texture) :
Cube(taille, vertexShader, fragmentShader)
{}
```

Maintenant que la classe-mère est prête, nous pouvons nous occuper de nos attributs. Le seul que l'on peut initialiser après les deux points est la texture **m_texture**, l'autre étant un tableau il doit être rempli entre les accolades.

Nous donnerons à **m_texture** la **string** reçue en paramètre pour lui spécifier l'image à charger. Nous appellerons également sa méthode **charger()** sinon elle risque d'être un peu vide dans notre rendu. 

Code : C++

```
Caisse::Caisse(float taille, std::string const vertexShader,
    std::string const fragmentShader, std::string const texture) :
    Cube(taille, vertexShader, fragmentShader),
    m_texture(texture)
{
    // Chargement de la texture
    m_texture.charger();
}
```

Pour initialiser le second attribut, **m_coordTexture**, nous allons reprendre le principe que nous avons utilisé lors de la création du tableau de couleurs. C'est-à-dire que nous devrons faire correspondre un couple de coordonnées de texture à chaque vertex comme nous l'avons fait précédemment.

Pour la première face de la caisse par exemple, nous pouvons reprendre celles que nous avons utilisées pour le carré :

Code : C++

```
// Coordonnées de texture

float coordTextureTmp[] = {0, 0, 1, 0, 1, 1,      // Face 1
                           0, 0, 0, 1, 1, 1};      // Face 1
```



Nous ré-utiliserons aussi le principe des tableaux temporaires dont le contenu sera copié dans l'attribut final.

L'avantage d'un cube, c'est que toutes ses faces ne sont en fait que des carrés. Du coup, nous pouvons reprendre les coordonnées de texture de la première face pour les assigner à toutes les autres. Nous n'avons pas besoin de les déterminer manuellement. Le tableau temporaire devient donc :

Code : C++

```
// Coordonnées de texture temporaires

float coordTextureTmp[] = {0, 0, 1, 0, 1, 1,      // Face 1
                           0, 0, 0, 1, 1, 1,      // Face 1

                           0, 0, 1, 0, 1, 1,      // Face 2
                           0, 0, 0, 1, 1, 1,      // Face 2

                           0, 0, 1, 0, 1, 1,      // Face 3
                           0, 0, 0, 1, 1, 1,      // Face 3

                           0, 0, 1, 0, 1, 1,      // Face 4
                           0, 0, 0, 1, 1, 1,      // Face 4

                           0, 0, 1, 0, 1, 1,      // Face 5
                           0, 0, 0, 1, 1, 1,      // Face 5

                           0, 0, 1, 0, 1, 1}       // Face 6
```

```
0, 0, 0, 1, 1, 1}; // Face 6
```

On a l'impression de travailler avec des couleurs tellement que ça y ressemble. 😞

Une fois les coordonnées définies, il ne manque plus qu'à les copier dans l'attribut **m_coordTexture** à l'aide d'une boucle **for** :

Code : C++

```
// Copie des valeurs dans le tableau final  
  
for(int i (0); i < 72; i++)  
    m_coordTexture[i] = coordTextureTmp[i];
```

Si on résume tout ça :

Code : C++

```
Caisse::Caisse(float taille, std::string const vertexShader,  
std::string const fragmentShader, std::string const texture) :  
Cube(taille, vertexShader, fragmentShader),  
  
m_texture(texture)  
{  
    // Chargement de la texture  
  
    m_texture.charger();  
  
    // Coordonnées de texture temporaires  
  
    float coordTextureTmp[] = {0, 0, 1, 0, 1, 1, // Face 1  
                                0, 0, 0, 1, 1, 1, // Face 1  
  
                                0, 0, 1, 0, 1, 1, // Face 2  
                                0, 0, 0, 1, 1, 1, // Face 2  
  
                                0, 0, 1, 0, 1, 1, // Face 3  
                                0, 0, 0, 1, 1, 1, // Face 3  
  
                                0, 0, 1, 0, 1, 1, // Face 4  
                                0, 0, 0, 1, 1, 1, // Face 4  
  
                                0, 0, 1, 0, 1, 1, // Face 5  
                                0, 0, 0, 1, 1, 1, // Face 5  
  
                                0, 0, 1, 0, 1, 1, // Face 6  
                                0, 0, 0, 1, 1, 1}; // Face 6  
  
    // Copie des valeurs dans le tableau final  
  
    for(int i (0); i < 72; i++)  
        m_coordTexture[i] = coordTextureTmp[i];  
}
```

Le constructeur initialise maintenant tous nos attributs, même ceux de sa classe-mère puisqu'il fait appel aussi à son constructeur.

Le destructeur

On passe rapidement sur le destructeur qui restera vide au même titre que celui du cube :

Code : C++

```
Caisse::~Caisse()
{
}
```

La méthode afficher()

La méthode **afficher()** va faire exactement la même chose que sa méthode "parente" à savoir afficher notre caisse à l'écran. Elle aura besoin des matrices **projection** et **modelview** en paramètres afin de pouvoir les envoyer à son shader :

Code : C++

```
void afficher(glm::mat4 &projection, glm::mat4 &modelview);
```

Petit détail : Nous faisons ici ce qu'on appelle le *masquage de méthode*, c'est une notion dont **M@téo** parle dans son tuto sur le C++. 😊

Son implémentation ne va pas nous poser de problème puisqu'il suffit de recopier le code que nous avons utilisé pour afficher le carré. Nous avons juste une petite modification à faire au niveau de la fonction **glDrawArrays()**. Celle-ci ne prendra pas en compte **6** mais **36 vertices**, soit le nombre requis pour un cube comme nous avons eu l'occasion de le voir.

Il faut également changer le nom des variables utilisées comme :

- Le shader qui devient **m_shader**
- Les tableaux qui deviennent **m_vertices** et **m_coordTexture**
- La texture qui devient **m_texture**

Le code de base à recopier est le suivant :

Code : C++

```
// Activation du shader
glUseProgram(shaderTexture.getProgramID());

// Envoi des vertices
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vertices);
 glEnableVertexAttribArray(0);

// Envoi des coordonnées de texture
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, coordTexture);
 glEnableVertexAttribArray(2);

// Envoi des matrices
```

```
glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),
    "projection"), 1, GL_FALSE, value_ptr(projection));

glUniformMatrix4fv(glGetUniformLocation(shaderTexture.getProgramID(),
    "modelview"), 1, GL_FALSE, value_ptr(modelview));

// Verrouillage de la texture
glBindTexture(GL_TEXTURE_2D, m_texture.getID());

// Rendu
glDrawArrays(GL_TRIANGLES, 0, 36);

// Déverrouillage de la texture
glBindTexture(GL_TEXTURE_2D, 0);

// Désactivation des tableaux
glDisableVertexAttribArray(2);
glDisableVertexAttribArray(0);

// Désactivation du shader
glUseProgram(0);
```

Une fois remanié et intégré dans la méthode **afficher()**, il ressemble à ceci :

Code : C++

```
void Caisse::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Envoi des vertices
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
        m_vertices);
    glEnableVertexAttribArray(0);

    // Envoi des coordonnées de texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
        m_coordTexture);
    glEnableVertexAttribArray(2);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
        "projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
        "modelview"), 1, GL_FALSE, value_ptr(modelview));
```

```
// Verrouillage de la texture
glBindTexture(GL_TEXTURE_2D, m_texture.getID());

// Rendu
glDrawArrays(GL_TRIANGLES, 0, 36);

// Déverrouillage de la texture
glBindTexture(GL_TEXTURE_2D, 0);

// Désactivation des tableaux
glDisableVertexAttribArray(2);
glDisableVertexAttribArray(0);

// Désactivation du shader
glUseProgram(0);
}
```

La classe **Caisse** est maintenant terminée. Nous n'avons plus qu'à la tester.

Afficher une caisse

Le plus dur est derrière nous maintenant, on peut facilement afficher un cube texturé en seulement 2 lignes de code. 😊

Nous devons premièrement déclarer un objet **Caisse** dans la méthode **bouclePrincipale()**, pensez à inclure l'en-tête "**Caisse.h**". Nous lui donnerons en paramètre une taille de **2.0**, le chemin vers les fichiers **texture.vert** et **texture.frag** ainsi vers la texture **Caisse.jpg** :

Code : C++

```
// Objet Caisse
Caisse caisse(2.0, "Shaders/texture.vert", "Shaders/texture.frag",
"Textures/Caisse.jpg");
```

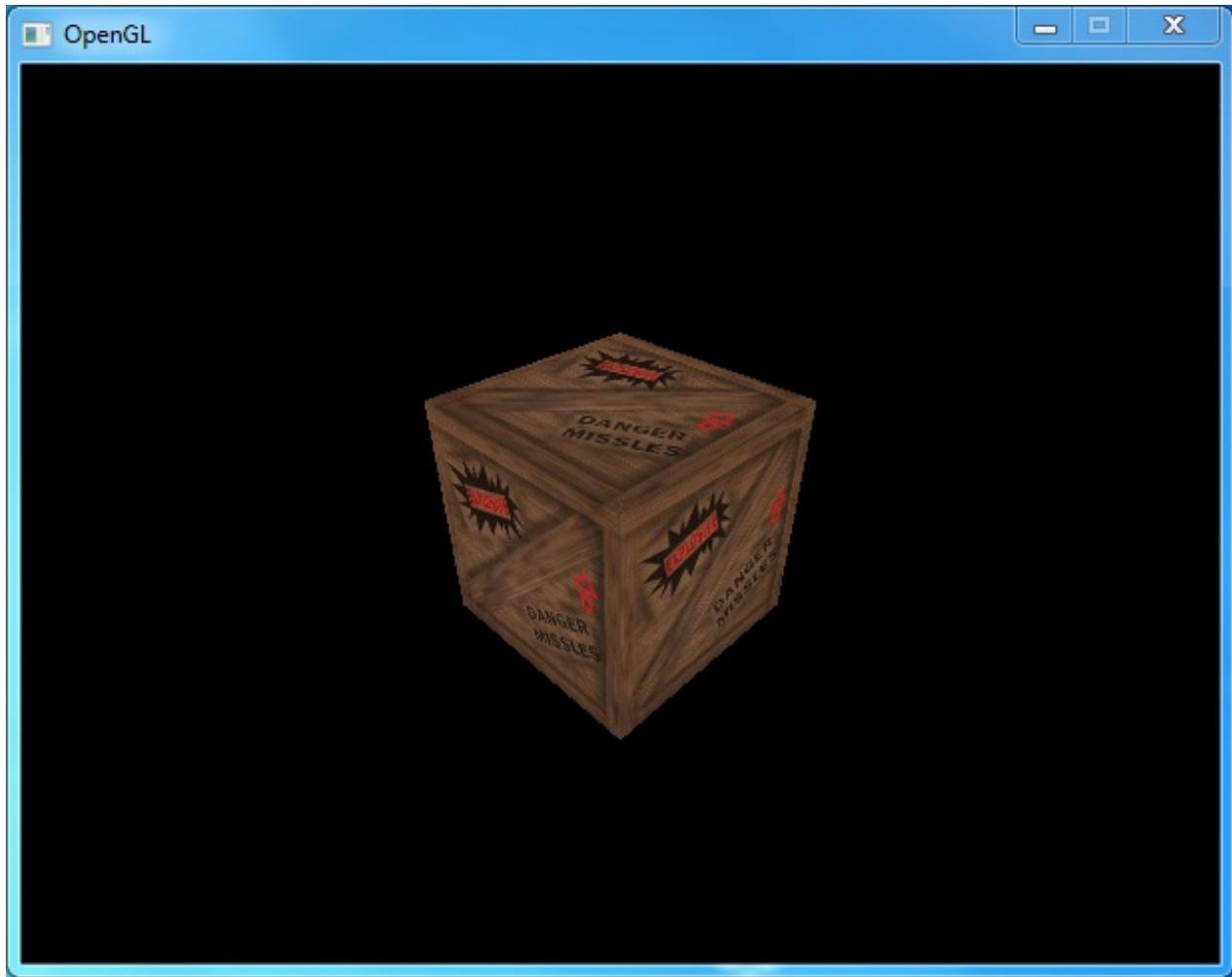
Ensuite, nous appelons sa méthode **afficher()** dans la boucle **while** en n'oubliant pas de lui donner les matrices **projection** et **modelview**:

Code : C++

```
// Boucle principale
while(!terminer)
{
    // Gestion des évènements, nettoyage de l'écran, ...
    ...
    // Affichage de la caisse
```

```
caisse.afficher(projection, modelview);  
  
// Actualisation de la fenêtre  
  
....  
}
```

Si vous compilez votre projet, vous devriez avoir :



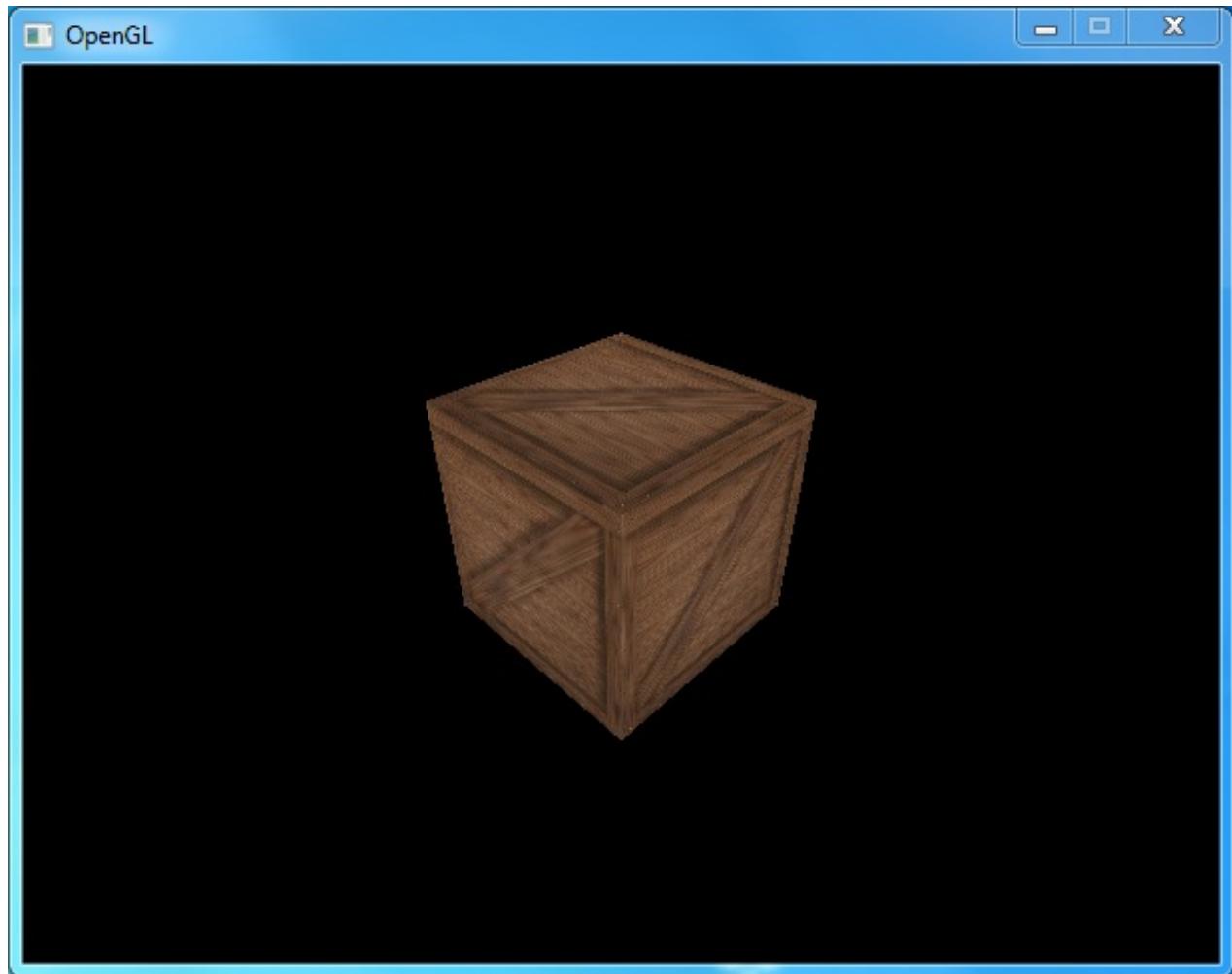
Vu que nous avons superbement bien coder notre classe (😊), nous pouvons changer l'aspect de notre caisse en ne modifiant qu'un seul paramètre lors de sa création. Par exemple, si vous voulez avoir une caisse avec la texture suivante (**crate12.jpg** dans l'archive) :



Il vous suffit juste de copier cette image dans votre dossier "**Textures**" puis de l'utiliser dans votre déclaration d'objet :

Code : C++

```
// Objet Caisse  
Caisse caisse(2.0, "Shaders/texture.vert", "Shaders/texture.frag",  
"Textures/Caisse2.jpg");
```



Vous pouvez même combiner la répétition de texture et notre caisse pour voir ce que ça donne. Vous devrez cependant faire translater votre cube pour ne pas qu'il transperce l'herbe. La valeur de la translation dépend de la taille de votre cube, celle-ci doit en faire la moitié. Si vous avez donné une taille de **2.0** unités alors la valeur sera de **1.0** :

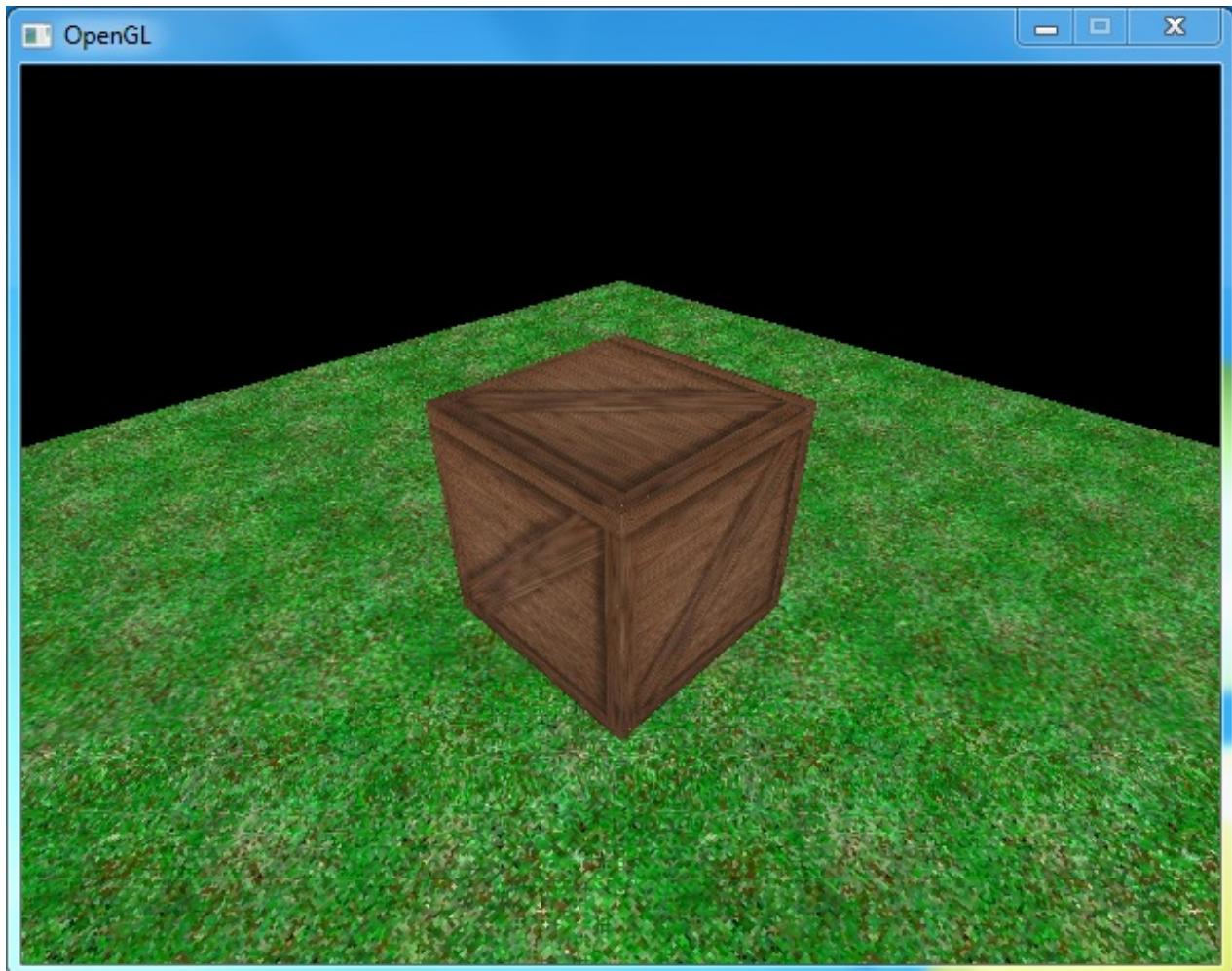
Code : C++

```
// Sauvegarde de la matrice modelview
mat4 sauvegardeModelview = modelview;

// Translation pour positionner le cube
modelview = translate(modelview, vec3(0, 1, 0));

// Affichage du cube
caisse.afficher(projection, modelview);

// Restauration de la matrice
modelview = sauvegardeModelview;
```



Télécharger : [Code Source C++ du chapitre 10](#)

Pfiou il était long ce chapitre. Je me demande encore si je n'aurais pas dû le couper en deux. 😊

Au moins ça nous a permis d'en apprendre pas mal sur les *textures* car nous savons maintenant charger des images depuis le disque dur et les afficher dans nos scènes 3D. Nous allons abandonner petit à petit les couleurs au profit de ces nouvelles *textures*, cela nous permettra d'avoir un monde plus réaliste.

Nous avons également vu dans ce chapitre comment manipuler des *objets OpenGL*. Je vous ai mis un commentaire en vert en dessous de chaque fonction que nous reverrons plus tard dans le tuto. Vous verrez que ces objets se manipulent tous de la même façon. 😊

La caméra

J'ai une bien mauvaise nouvelle très chers compagnons, nous arrivons à la fin de la première partie de ce tutoriel. 😞
(Super façon d'introduire un chapitre 😊)

En effet, nous avons déjà vu presque toutes les notions essentielles *d'OpenGL*. Nous maîtrisons l'affichage de formes simples, l'utilisation basique de Shaders, l'affichage de textures, la gestion des événements, ... Il ne reste plus qu'une seule chose à voir pour compléter ces essentielles : **une caméra déplaçable**.

Nous avons déjà eu l'occasion d'approcher *la caméra*, notamment dans le chapitre sur la 3D, mais jusqu'à maintenant on se contentait de la positionner au début du programme et du coup elle restait immobile. Notre objectif, avec ce chapitre, va être de lui donner la capacité de **se déplacer**. Ainsi nous pourrons nous balader librement dans nos scènes 3D.

Après ce chapitre, nous passerons à un TP qui rassemblera toutes les notions ce que nous avons vu. Nous aurons un début de monde 3D réaliste. Mais bon pour le moment, nous devons encore étudier la caméra.

Commençons ! 🎉😊

Fonctionnement Une caméra déplaçable

Les types de caméra

On a déjà eu l'occasion d'approcher le système **caméra** avec *OpenGL* qui permet d'avoir une représentation d'un monde virtuel sur notre écran. Ce système fait exactement la même chose qu'un tournage de film, nous ne sommes pas présents sur un plateau et pourtant nous voyons les scènes qui y sont tournées.

Le problème de notre *caméra* actuelle c'est qu'elle est totalement immobile, nous n'avons aucun moyen pour la faire bouger. L'objectif de ce chapitre va donc être de lui ajouter de la mobilité afin que nous puissions nous déplacer dans notre monde virtuel. Nous utiliserons une méthode de déplacement particulière basée sur le vol en mode libre (**FreeFly** en anglais). Ce mode permet à la caméra de voler, tel un avion, de façon totalement libre. On pourra donc aller à gauche, à droite, en haut et en bas comme on voudra. 😊

On retrouve ce type de caméra dans les jeux-vidéo où on pilote un hélicoptère, un avion, ...



Image issue du jeu *GTA San Andreas*

Cette caméra représentera une base avec laquelle nous pourrons travailler. Dans l'avenir, il suffira de quelques modifications pour la faire passer en vue à la 1ière ou à la 3ième personne. Il existe aussi d'autres types caméras *en vue de dessus* que l'on remarque dans les jeux de stratégie par exemple. Mais pour le moment, nous nous contenterons de développer une caméra libre que nous utiliserons dans les futurs chapitres. 😊

Les contrôles

Pour pouvoir se déplacer dans notre monde 3D, nous devrons utiliser le clavier et la souris. Grâce à la classe **Input** (que l'on a déjà codée) nous allons pouvoir lier un évènement à un déplacement, ces évènements seront divisés en deux catégories. Nous utiliserons :

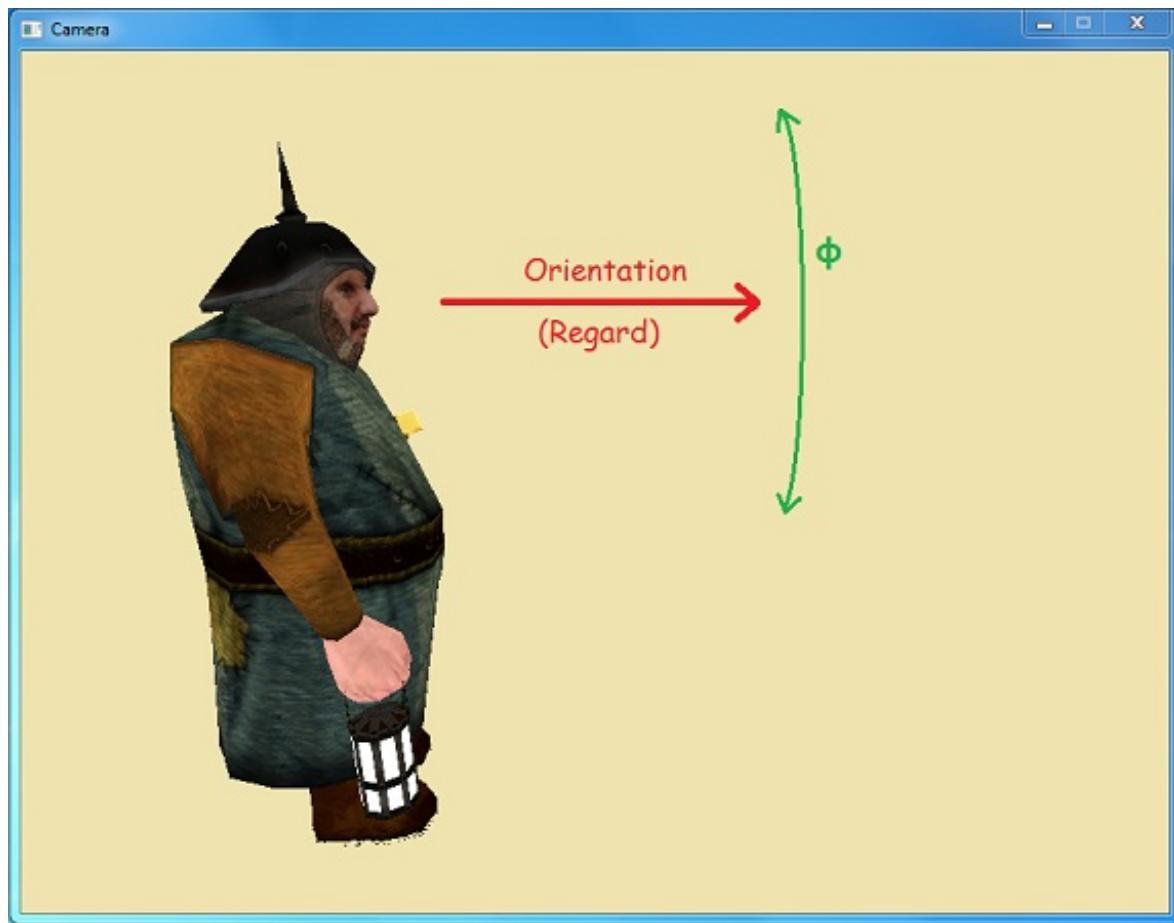
- **La souris** : pour orienter la caméra en fonction du mouvement effectué
- **Le clavier** : pour déplacer la caméra selon 4 axes (gauche, droite, haut et bas)

Ces deux catégories d'évènements vont se gérer de manière totalement différente, nous aurons besoin de deux méthodes distinctes. Et croyez-moi, elles ne feront pas du tout la même chose. 🤪

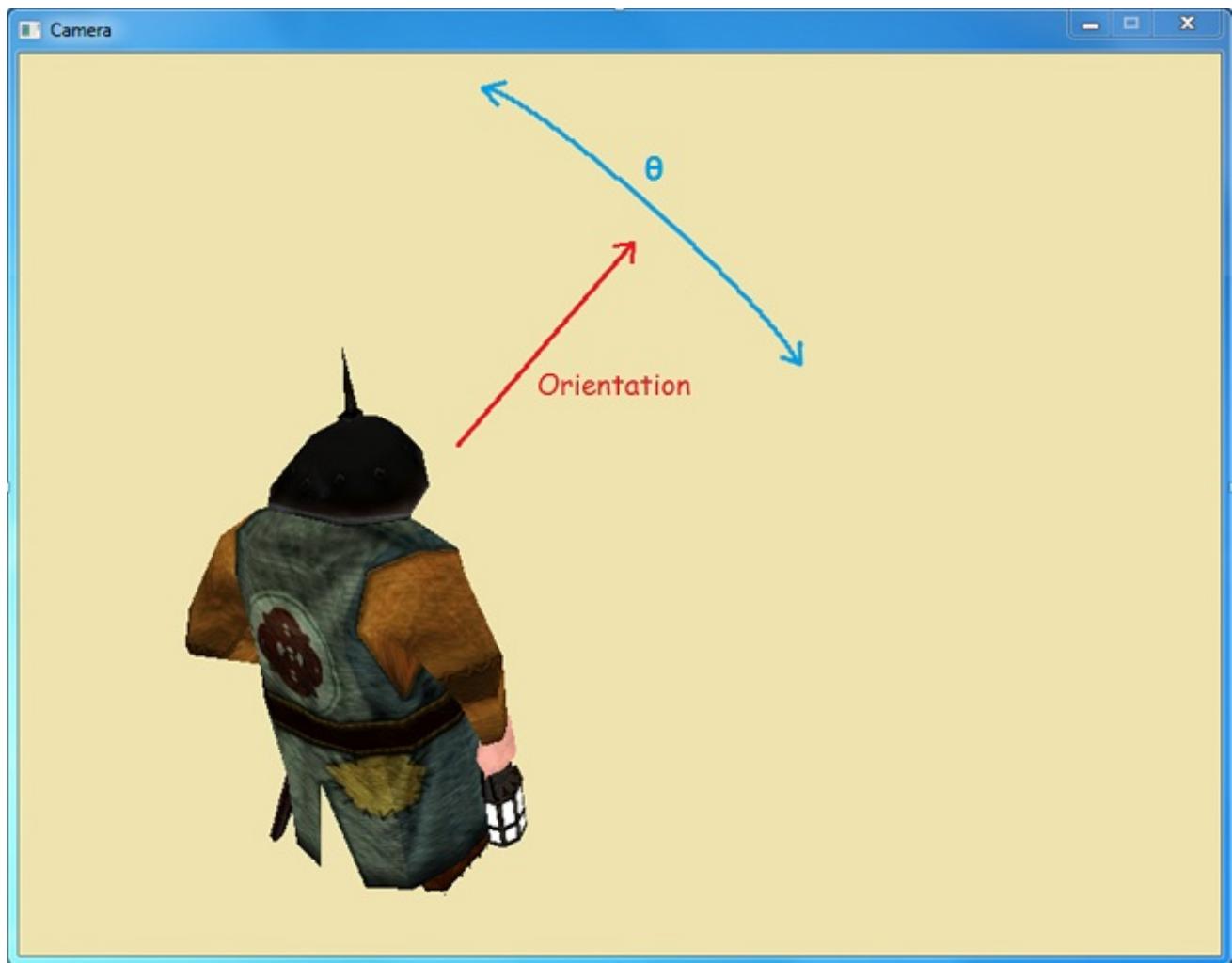
La gestion de l'orientation

On vient de voir le principe d'une *caméra libre* et je vous ai même précisé la façon dont on va la gérer. Je vais maintenant aller plus loin dans la théorie afin que vous puissiez comprendre ce que nous allons faire par la suite. Ouvrez grand les oreilles et n'hésitez pas à relire plusieurs fois ce début de chapitre. Il est un peu technique mais il est absolument fondamental que vous compreniez les notions que l'on va aborder. D'ailleurs, on en ré-utilisera certaines dans de futurs chapitres.

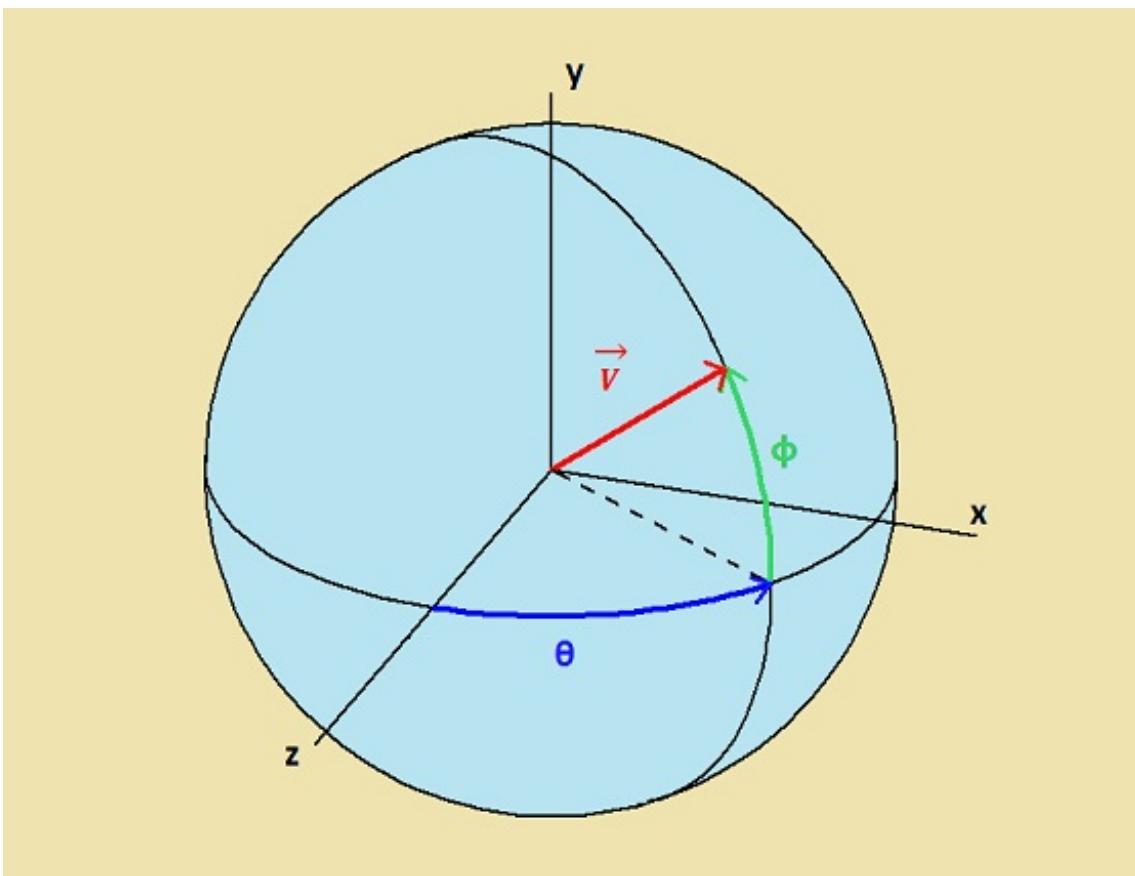
La première chose que l'on va voir concerne l'orientation de la caméra. Les évènements déclenchés par la souris permettront de l'orienter en fonction du mouvement effectué. Lorsque nous bougerons la souris vers le haut ou vers le bas, la caméra s'orientera de la même façon :



Et lorsque nous bougerons la souris vers la gauche ou vers la droite, la caméra s'orientera également de la même façon :



Bien entendu, on peut combiner ces deux mouvements pour que la caméra s'oriente dans n'importe quelle direction. Voici un schéma résumant toutes les informations dont nous avons besoin sur l'orientation :



Les angles **Phi** et **Theta** représentent les éléments mathématiques indispensables qui vont nous permettre de calculer l'orientation de la caméra. Le premier représente l'orientation sur l'axe **horizontal**, le second l'orientation sur l'axe **vertical**. Ces deux angles sont absolument primordiaux. Sans eux, on pourrait toujours se déplacer vers la gauche ou la droite, ... mais notre regard resterait totalement fixe. Imaginez-vous en train de marcher dans la rue sans pouvoir tourner la tête (un peu embêtant n'est-ce pas ?).

Les angles ne sont pas les seuls éléments importants sur ce schéma, le **vecteur V (orientation)** est tout aussi important. Il représente la direction de notre regard (ce qui sera affiché sur l'écran).



Mais avec quoi va-t-on pouvoir calculer ce vecteur **orientation** ? Je sais pas le calculer moi. 🤔

Rassurez-vous, je ne vais pas vous demander de trouver une formule par vous-même. 🎉

En fait, il existe une formule toute prête pour calculer ce vecteur, enfin plus précisément pour calculer ses **coordonnées (x, y, z)**. Cette formule permet de trouver ce que l'on appelle les coordonnées sphériques :

$$\begin{aligned} x &= r \times \cos(\phi) \times \sin(\theta) \\ y &= r \times \sin(\phi) \\ z &= r \times \cos(\phi) \times \cos(\theta) \end{aligned}$$

Ah tiens ! On retrouve les angles de tout à l'heure !

Et oui ah ah ! Je vous avais dit que ces angles étaient super importants. 🤓 Et encore, on peut simplifier cette fonction en

enlevant le paramètre R . Ce paramètre correspond à la longueur du rayon de la sphère précédente. Mais vu que nous travaillerons avec des vecteurs normalisés, cette longueur sera toujours égale à 1, ce qui donne au final :

$$\begin{aligned}x &= \cos(\phi) \times \sin(\theta) \\y &= \sin(\phi) \\z &= \cos(\phi) \times \cos(\theta)\end{aligned}$$

Grâce aux angles **Theta** et **Phi**, nous pourrons calculer le vecteur **orientation** les doigts dans le nez. 

 La formule des **coordonnées sphériques** n'est pas compliquée à démontrer. Si vous êtes curieux, vous pouvez voir comment la démontrer grâce à l'annexe sur la trigonométrie de **Kayl**. Attention cependant, Kayl utilise la coordonnée Z comme repère vertical et non l'axe Y. Les résultats sont donc inversés entre les coordonnées. Mais au moins, grâce à cette annexe vous pourrez constater que je n'ai pas sorti la formule de mon chapeau. 

 Et ces deux angles, comment va-t-on les calculer ?

Il n'y aura pas besoin des les calculer. Au niveau de la programmation, ces angles correspondront au mouvement de la souris. L'angle Phi représentera l'axe Y et Théta l'axe X.

En définitif, grâce à ces deux angles et aux coordonnées sphériques nous pourrons gérer l'orientation de notre caméra très facilement.

La gestion du déplacement

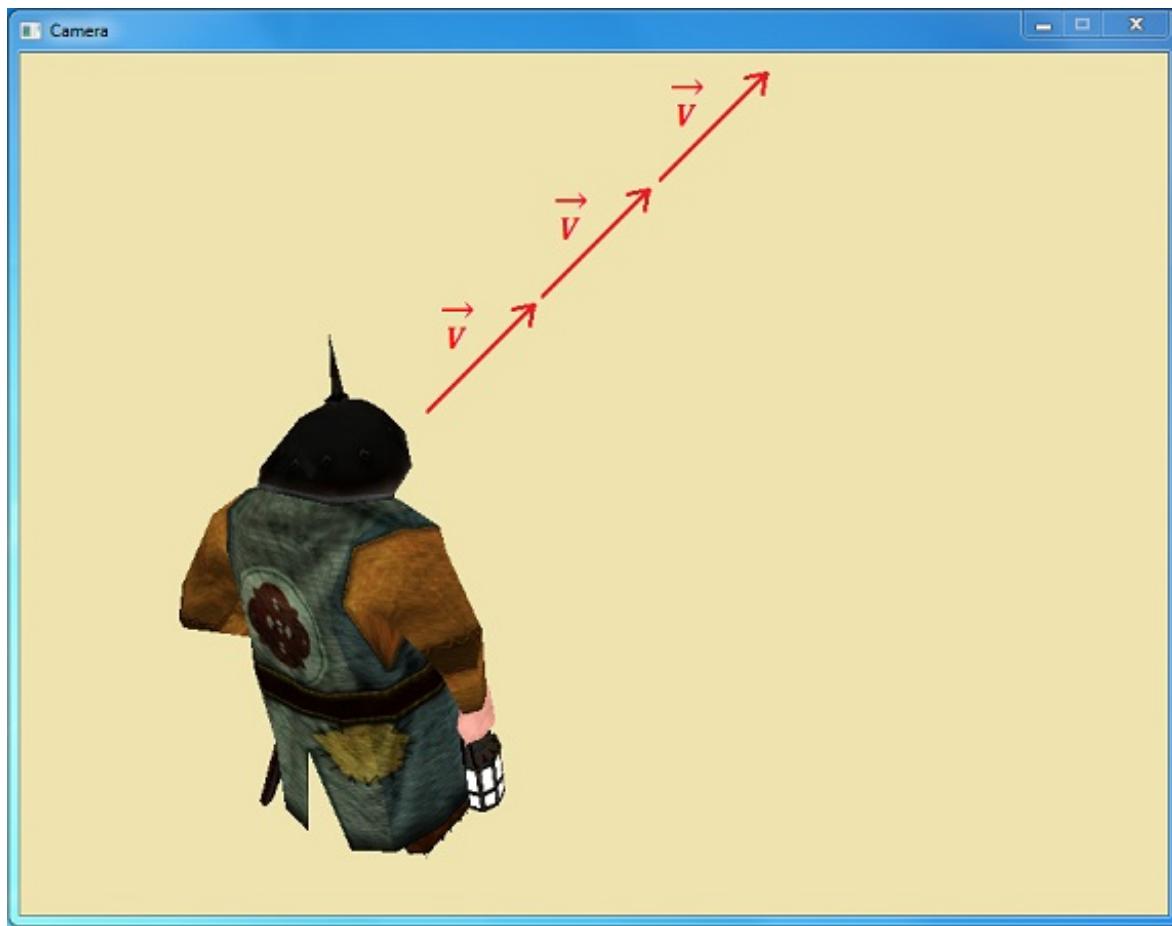
Avancer et reculer

La gestion du **déplacement** de la caméra va être différente de celle de l'**orientation**. Nous n'aurons ni besoin d'angles, ni de coordonnées sphériques, mais uniquement de vecteurs. On va diviser cette gestion en deux parties :

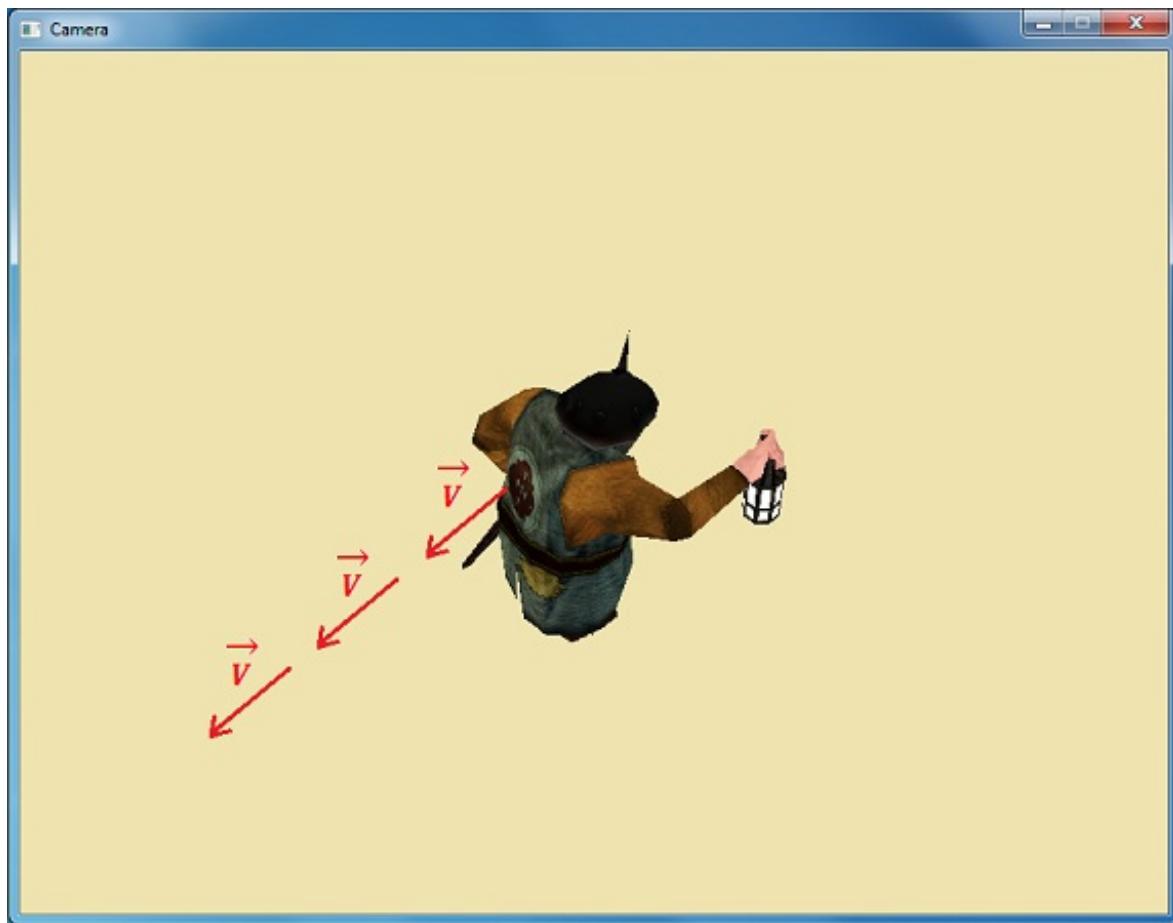
- Le déplacement '*vertical*' (à défaut de trouver meilleur terme) pour avancer et reculer
- Le déplacement *latéral* pour aller vers la gauche ou la droite

On commence par la gestion du déplacement vertical car ce sera le plus facile. Pour faire avancer notre caméra nous n'aurons besoin que d'une simple *addition*. Oui oui vous avez bien entendu, nous n'aurons que d'une simple *addition*. 

En effet, quand on se déplace vers l'avant (en marchant par exemple) on ne fait en réalité qu'une simple *translation*. Or quand on translate, on ne modifie pas notre orientation. C'est comme si l'on additionnait un vecteur par lui-même :



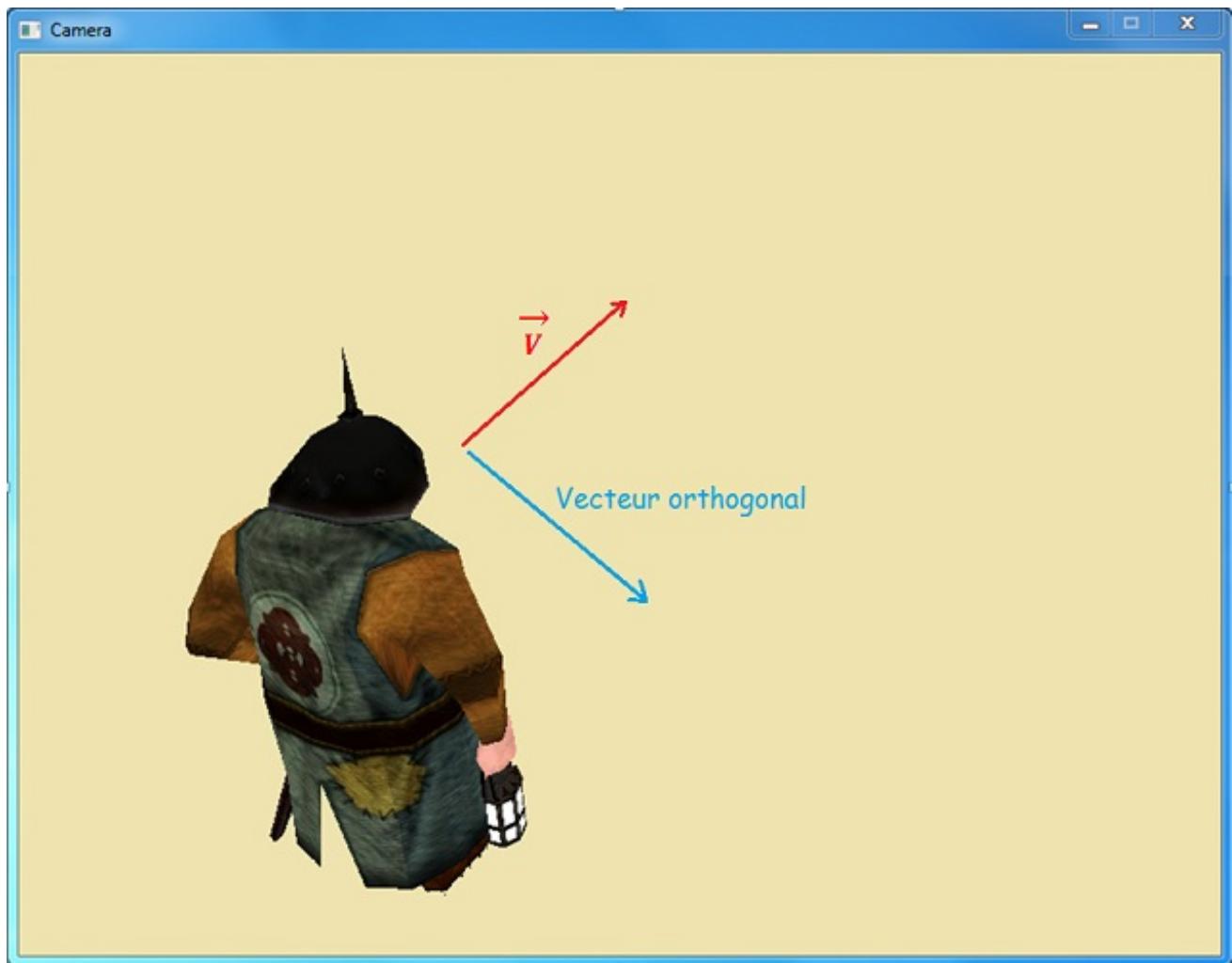
Pour faire avancer notre caméra, il suffira d'additionner le vecteur ***orientation*** par ***lui-même*** autant de fois que nécessaire. Et évidemment la faire pour reculer, il suffira de faire l'inverse d'une addition, c'est-à-dire ... une soustraction !



Je vous avais dit que la gestion verticale était simple. 😊

Le déplacement latéral (1/2)

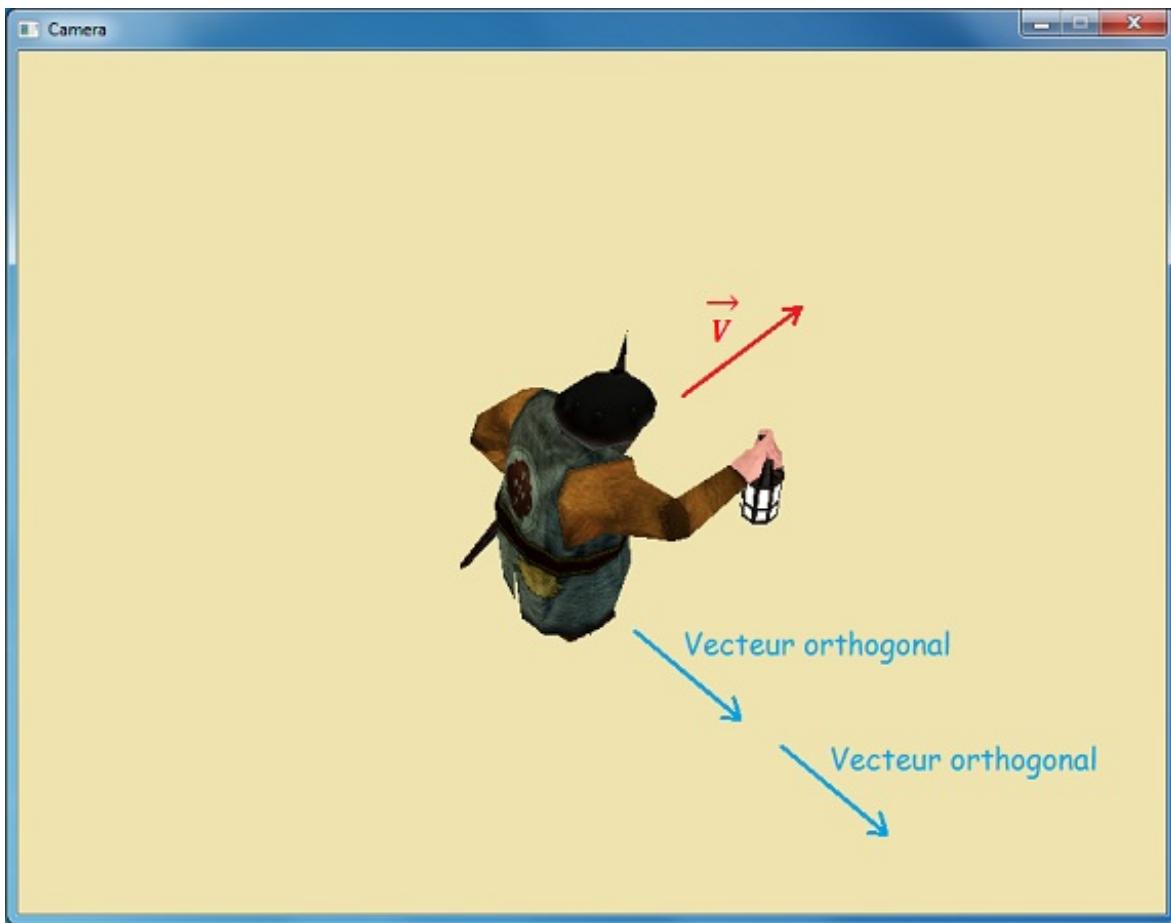
Le déplacement latéral va être un peu plus délicat à gérer. Son principe est cependant simple : il faut trouver un vecteur '*perpendiculaire*' (on dira plutôt *orthogonal*) au vecteur ***orientation*** :



Ce vecteur orthogonal va nous permettre de déplacer la caméra de façon latéral.

D'ailleurs, une fois que nous l'aurons trouvé il suffira de faire la même chose que pour avancer :

- **Pour aller vers la gauche** : on additionnera ce vecteur par lui-même autant de fois que nécessaire
- **Pour aller vers la droite** : on soustraira ce vecteur par lui-même autant de fois que nécessaire



Ah d'accord, on fait la même chose que pour avancer sauf qu'ici on avance '*perpendiculairement*' à l'orientation ?

Vous avez compris. 😊

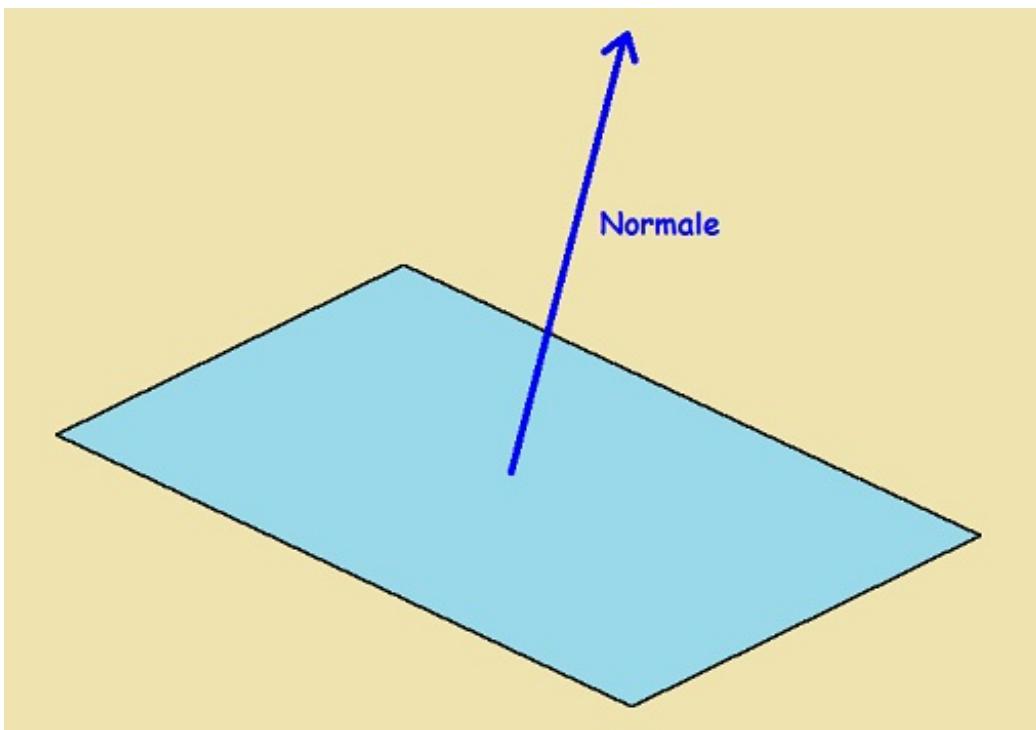
Il faut donc calculer ce vecteur pour avancer latéralement. Mais avant cela, il va falloir que l'on parle de la notion de **normale d'un plan**. Cette notion est importante car c'est elle qui va nous permettre de trouver ce fameux vecteur.

La normale d'un plan

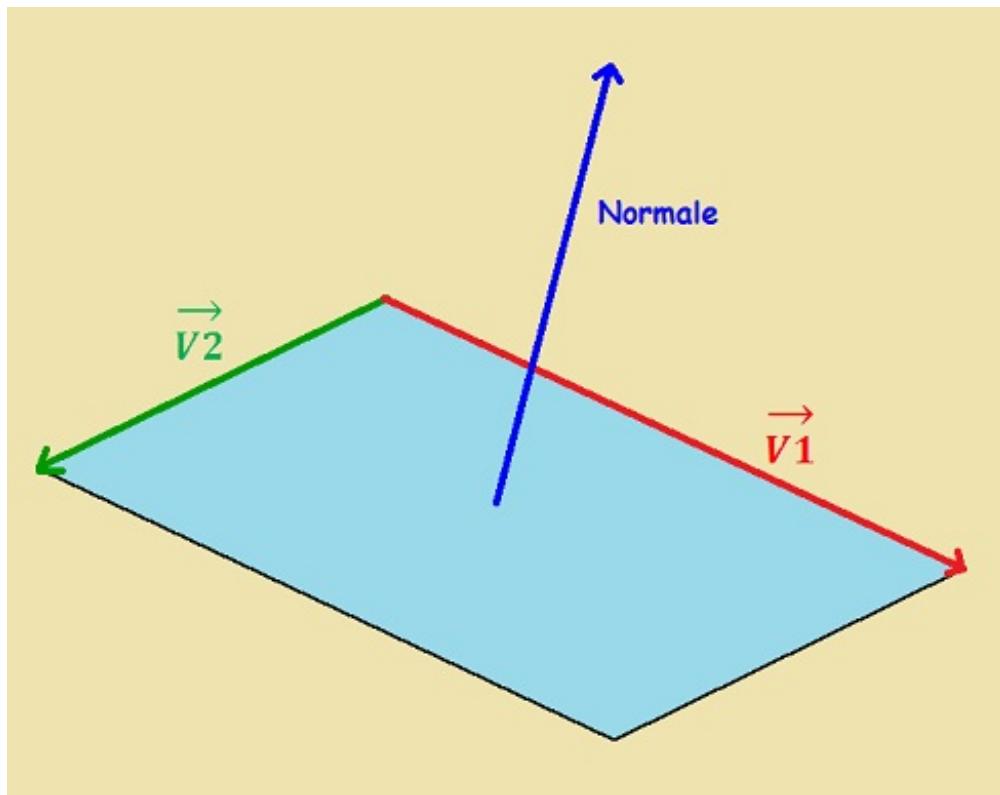
Les normales sont très importantes avec *OpenGL*. On retrouvera cette notion plusieurs fois dans le tuto alors soyez attentifs.



Une **normale** est une sorte de droite *perpendiculaire* à une surface plane, on appellera cette surface un **plan**. Si vous prenez un cahier (qui est une surface plane) la *normale* de son plan partira de la couverture pour aller vers le haut.



Cette *normale* peut être représentée par un vecteur. Et pour le calculer, il suffit de multiplier deux vecteurs appartenant à votre cahier :



Ça veut dire que si on multiplie deux vecteurs appartenant au cahier on trouve la normale du plan ?

Oui tout à fait, car il existe une propriété de la géométrie dans l'espace qui dit que : lorsque l'on **multiplie** deux vecteurs appartenant à un même plan, le résultat de cette multiplication sera un vecteur **perpendiculaire** au plan.

Ce vecteur résultat se nomme la **normale du plan**.

Cependant, pour pouvoir valider cette multiplication il va falloir respecter deux règles :

- Premièrement, il faut absolument que les deux vecteurs à multiplier ne soient pas parallèles.
- Deuxièmement, les vecteurs doivent partir vers des **directions opposées**. Il ne faut pas que leur flèche puissent se croiser.

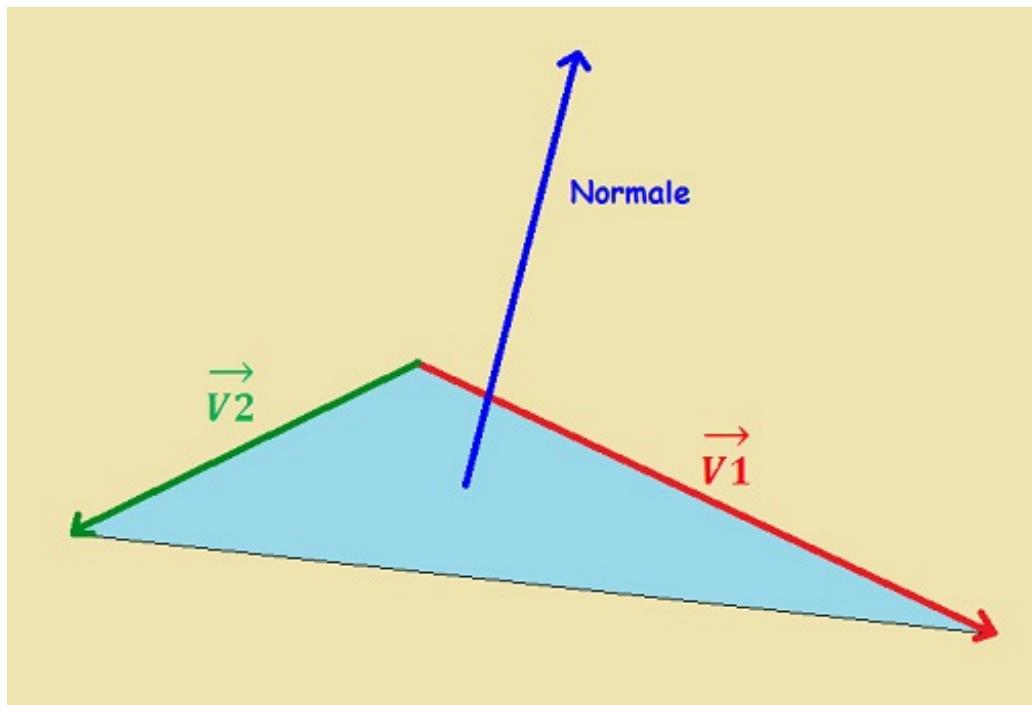
Si une de ces deux règles n'est pas respectée alors votre calcul sera faux et vous ne trouverez pas la normale.

Cependant je vous rassure, nous utiliserons toujours de bons vecteurs, je ne vais pas être sadique et vous faire faire des calculs foireux. 😊 Mais sachez au moins que ces règles existent.

Si on reprend le schéma précédent, on remarque que les vecteurs **V1** et **V2** ne sont pas parallèles et que leur flèche ne se rejoignent pas, les deux règles sont donc respectées. Si on multiplie ces deux vecteurs entre eux on trouvera la **normale** du plan.



Bien évidemment, les normales ne se limitent pas aux surfaces carrées. D'ailleurs, on les utilisera presque toujours sur des triangles, ce qui compte c'est que la surface utilisée soit plane :

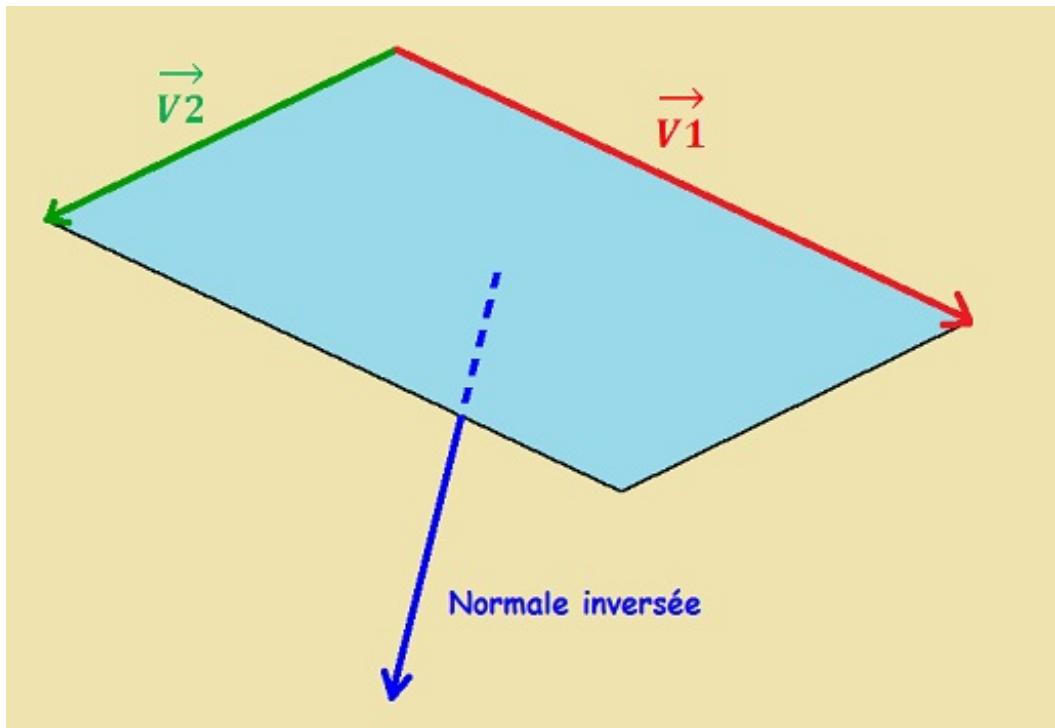


Petite question : comment est-ce qu'on multiplie deux vecteurs ?

Ce n'est pas à proprement parler une multiplication, il s'agit plus d'un produit vectoriel. Nous n'aurons pas à le faire nous même donc ne vous prenez pas la tête avec ça. 😊

D'ailleurs en parlant de ça, je vous avais également dit qu'il fallait faire attention à l'ordre dans la multiplication, que **V1 x V2** n'était pas forcément égal à **V2 x V1**.

Cette remarque est toujours d'actualité. Si vous inversez la multiplication de deux vecteurs vous trouverez bien la **normale**, mais le problème c'est qu'elle sera inversée :



Faites juste attention à l'ordre des multiplications que nous ferons et tout ira bien. 😊

Le déplacement latéral (2/2)

Pour en revenir à ce qui nous intéresse, je vous rappelle que l'on recherche un vecteur *orthogonal* au vecteur **orientation** pour que l'on puisse se déplacer latéralement.

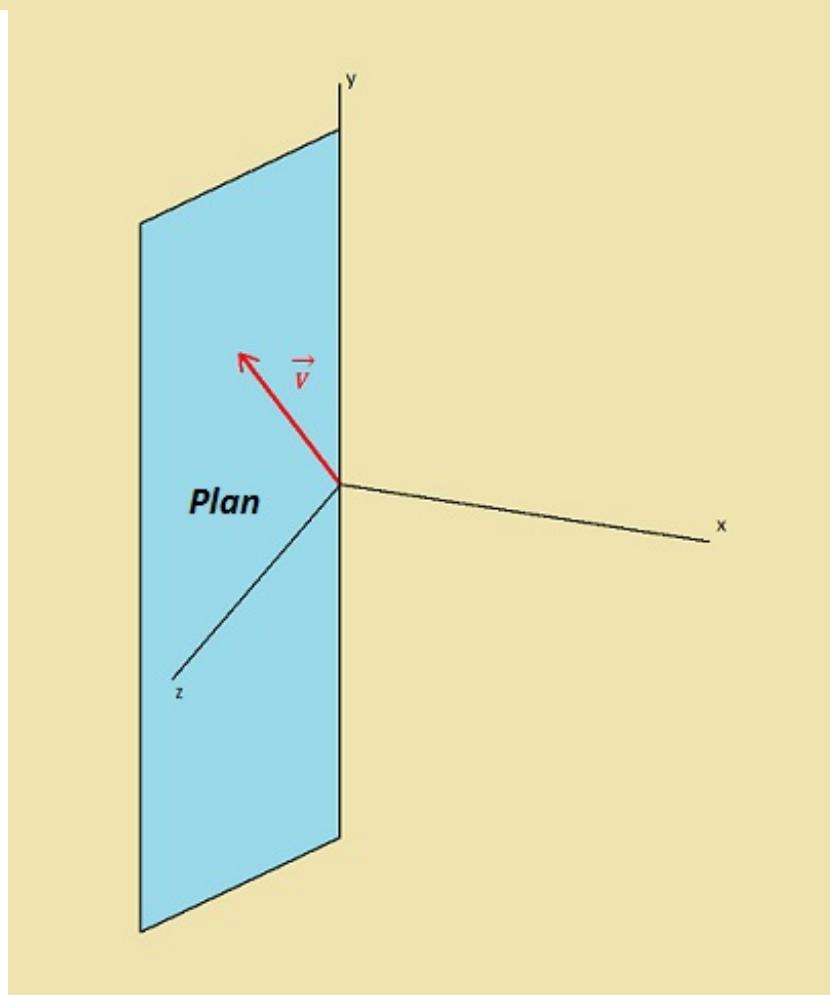
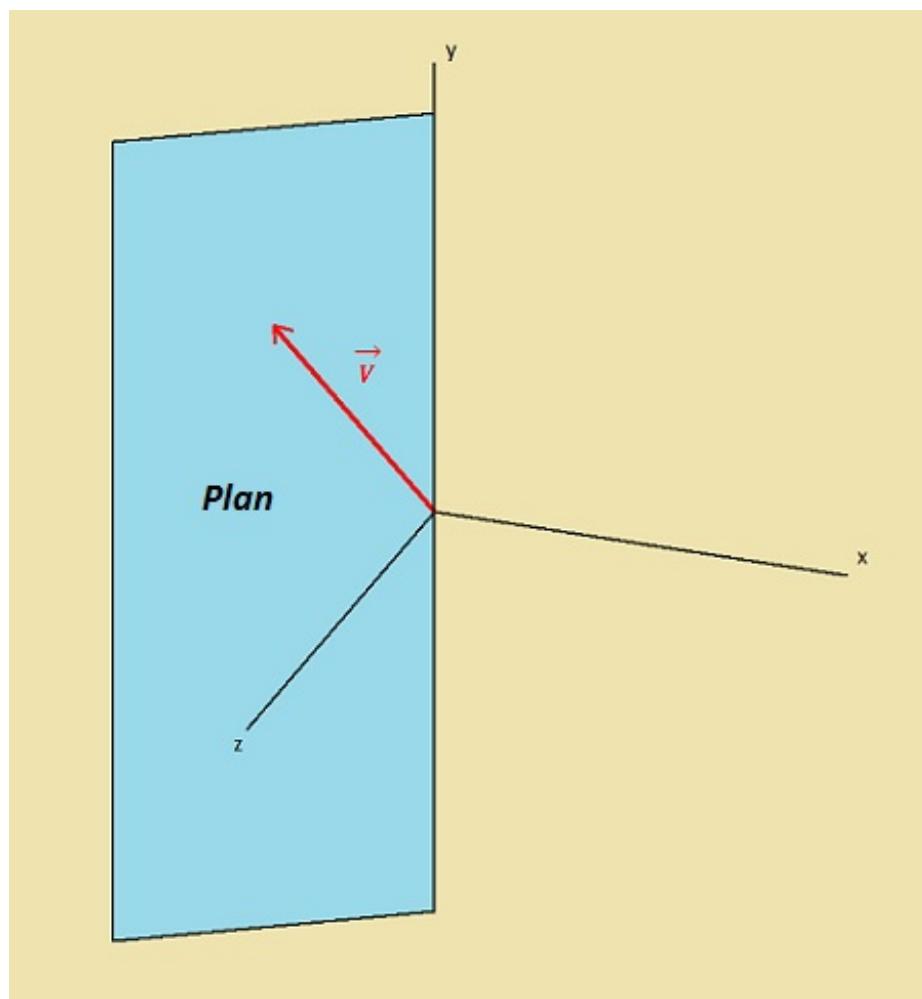
Par chance, nous venons d'apprendre ce que sont les *normales* et je vous annonce que le vecteur que nous recherchons en est justement une !

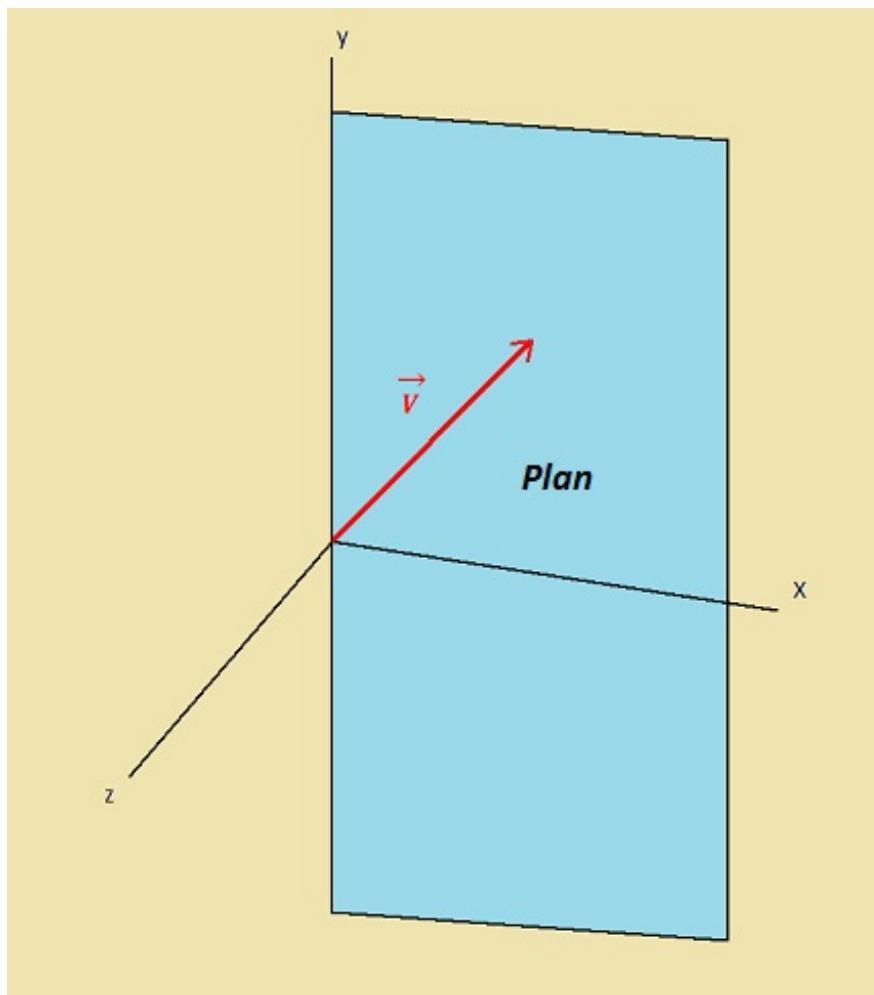


Hein ? Mais pour calculer une normale il faut un plan non ? Or là on n'a que le vecteur **orientation**.

Oui tout à fait, nous avons besoin d'un plan dans lequel se trouve le vecteur **orientation**. Et pour le trouver, nous allons utiliser une petite astuce : **l'axe y du repère 3D**.

Cet axe formera le deuxième vecteur dont on a besoin pour la multiplication. En effet, quelle que soit l'orientation de la caméra, le vecteur **orientation** et le vecteur de **l'axe y** feront toujours partie du même plan :





Ces vecteurs font partie du même plan et sont orientés vers des directions différentes, on peut donc les utiliser pour calculer la *normale du plan*. 😊

Cette astuce est efficace mais il existe un cas où elle ne fonctionnera pas. En effet, souvenez-vous que la normale est calculable tant que les deux vecteurs utilisés ne sont pas parallèles. Or, si la caméra s'aligne parfaitement à l'axe y alors l'une des deux règles sur les normales sera violée.



Pour éviter ce problème, il va falloir interdire à la caméra de se pencher à **90°** vers le haut (et **-90°** pour le bas). Ce qui veut dire qu'à une orientation de **89°**, la caméra ne pourra plus aller plus haut. Cependant, elle restera toujours libre au niveau de ses autres mouvements.

En définitif, pour trouver le vecteur de déplacement latéral il va falloir multiplier le vecteur **orientation** par le vecteur de **l'axe y**. Ensuite, il suffira de l'additionner ou de le soustraire par lui-même pour se déplacer vers la gauche ou vers la droite.

Pfiouuu ... Tout ce gourbi pour calculer un vecteur !

Et oui c'est toute une histoire, mais au moins vous savez maintenant ce que sont les *normales*. C'est une notion hyper-importante qu'on aura l'occasion de retrouver lorsque l'on fera de l'éclairage dynamique avec les shaders. 😞

En résumé

Nous venons de voir pas mal de mathématique, je peux comprendre votre probable mal de tête. 😥 Je vais faire un résumé des points importants qu'il faut retenir.

Pour gérer l'orientation :

- Nous avons besoin de deux angles : **Theta** et **Phi**
- Ces deux angles permettent de calculer des **coordonnées sphériques**
- Ces coordonnées sphériques formeront un vecteur qui représentera **l'orientation** de la caméra

Pour gérer le déplacement :

- Si on additionne le vecteur **orientation** par lui-même alors la caméra avance. Quand le soustrait par lui-même, elle recule
- Le vecteur **orientation** et le vecteur de l'**axe y** font partie du même plan
- Si on multiplie ces vecteurs entre eux, on trouve la **normale** du plan. Cette normale sera le vecteur de **déplacement latéral**

Voilà ce qu'il faut retenir.

N'hésitez pas à relire encore et encore cette première partie de chapitre. 😊

Lorsque vous vous sentirez prêt, nous passerons à la suite du chapitre.

Implémentation de la caméra

La classe Camera

Le header

J'espère que vous avez eu le temps de digérer tout ce que l'on vient de voir. Si ça peut vous rassurer, sachez que c'était la partie la plus compliquée. Nous allons pouvoir passer à l'implémentation de la caméra. 😊

Comme d'habitude, on va commencer par le header en déclarant une nouvelle classe : la classe **Camera**. Elle possèdera pas mal d'attributs, mais on les connaît déjà tous. Voici le header de base :

Code : C++

```
#ifndef DEF_CAMERA
#define DEF_CAMERA

// Classe

class Camera
{
    public:
        Camera();
        ~Camera();

    private:
};

#endif
```

Dans la première partie de ce chapitre, nous avons vu que le déplacement de la caméra se divisait en 2 catégories : une qui s'occupait de gérer l'**orientation** et l'autre du **déplacement** pur et dur.

Pour gérer ce premier point, nous aurons besoin de 3 choses :

- Un angle **Theta** : qui représentera l'orientation sur l'axe horizontal
- Un angle **Phi** : représentera l'orientation sur l'axe vertical
- Un vecteur **orientation** : qui représentera la direction dans laquelle on regarde



Ah mais comment on fait pour un avoir un vecteur en C++ ? Ça n'existe pas ?

En C++ non, mais n'oubliez pas que nous disposons d'une librairie mathématique complète. 😊 **GLM** ne sert pas qu'aux matrices mais aussi aux vecteurs, quaternions, etc. Plein de choses en somme.

Il existe un objet que nous avons déjà utilisé et qui permet de gérer des vecteurs à 3 dimensions. Cet objet s'appelle **vec3**. Vous vous en souvenez j'espère. 🍪 Il a l'avantage de posséder des méthodes simples qui permettent d'accéder à ses valeurs. Ainsi, pour retrouver sa coordonnée **x** par exemple, il nous suffit de faire :

Code : C++

```
monVecteur.x;
```

En définitif, nous allons utiliser un objet de type **vec3** pour gérer l'orientation de la caméra. 😊

Avec ça, nous avons donc nos 3 premiers attributs :

Code : C++

```
// Attributs d'orientation

float m_phi;
float m_theta;
glm::vec3 m_orientation;
```

Pour ne pas avoir d'erreur de compilation, il faut inclure les en-têtes relatifs à la librairie **GLM** :

Code : C++

```
#ifndef DEF_CAMERA
#define DEF_CAMERA

// Includes GLM

#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Classe

class Camera
{
public:

    Camera();
    ~Camera();

private:
};

#endif
```

Au niveau du deuxième point (le déplacement de la caméra), nous avons vu que nous aurons besoin de 3 choses :

- Un vecteur **orientation** (le même que le précédent)
- Un vecteur représentant l'**axe vertical** (l'axe y dans notre cas) : afin de pouvoir former un plan avec le vecteur **orientation**
- Un vecteur de **déplacement latéral** : qui représente la *normale* du plan formé

Vu que nous avons déjà le vecteur **orientation**, on ne rajoute que les deux derniers vecteurs aux attributs :

Code : C++

```
// Attributs de déplacement  
  
glm::vec3 m_axeVertical;  
glm::vec3 m_deplacementLateral;
```

Petite info au passage : souvenez-vous que la méthode **lookAt()** prends 3 paramètres de type **vec3**. Pour les alimenter, il nous suffira juste de donner nos attributs qui sont eux-mêmes de objets de type **vec3** 😊

En parlant de cette méthode, on va refaire un petit tour au niveau de son prototype :

Code : C++

```
mat4 lookAt(vec3 eye, vec3 center, vec3 up);
```

- Vecteur **eye** : Position de la caméra
- Vecteur **center** : Point fixé par la caméra
- Vecteur **up** : Axe vertical utilisé (**x**, **y** ou **z**)

Si je vous remémore cette méthode c'est parce que nous en aurons besoin dans la classe **Caméra**. En effet, nous avons beau créer une classe toute neuve pour gérer la caméra, nous aurons toujours besoin d'utiliser les *matrices* et notamment la méthode **lookAt()** de la matrice **modelview**. Nous aurons donc besoin de 3 vecteurs correspondant aux 3 vecteurs demandés par cette méthode.

Comme on vient de le voir on a déjà l'axe vertical, on ne va donc ajouter que les deux autres dans les attributs. Cependant, on va faire une petite modification au niveau de leur nom de façon à les rendre plus compréhensibles. Ainsi, le vecteur **eye** deviendra le vecteur '**position**' et **center** deviendra le vecteur '**pointCible**' (point ciblé):

Code : C++

```
glm::vec3 m_position;  
glm::vec3 m_pointCible;
```

Si on regroupe tous les attributs :

Code : C++

```
// Attributs  
  
float m_phi;  
float m_theta;  
glm::vec3 m_orientation;  
  
glm::vec3 m_axeVertical;  
glm::vec3 m_deplacementLateral;
```

```
glm::vec3 m_position;
glm::vec3 m_pointCible;
```

Notre header est presque complet, il ne manque plus qu'à ajouter un constructeur - non pas un constructeur par défaut mais un constructeur qui prendra exactement les mêmes paramètres que la méthode **lookAt()**, à savoir :

- Une position
- Un point à cibler
- Un vecteur représentant l'axe vertical

Code : C++

```
Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3
axeVertical);
```

Header final :

Code : C++

```
#ifndef DEF_CAMERA
#define DEF_CAMERA

// Includes GLM

#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Classe

class Camera
{
public:

    Camera();
    Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3
axeVertical);
    ~Camera();

private:

    float m_phi;
    float m_theta;
    glm::vec3 m_orientation;

    glm::vec3 m_axeVertical;
    glm::vec3 m_deplacementLateral;

    glm::vec3 m_position;
    glm::vec3 m_pointCible;
};

#endif
```

Les Constructeurs et le Destructeur

Vous commencez à avoir l'habitude avec les constructeurs et tout ça. 😊 Le principe ne change pas, on initialise toujours les attributs.

Au niveau du constructeur par défaut, il suffit simplement d'initialiser les angles avec la valeur **0** et les vecteurs avec leur constructeur par défaut.

Il faut cependant faire attention à l'attribut **m_axeVertical**. En effet, celui-ci permet de savoir quel axe parmi **X**, **Y** ou **Z** représentera l'axe vertical. Si nous n'utilisons le constructeur par défaut il faudra quand même affecter une valeur *non nulle* sinon la caméra ne pourra pas faire ses calculs correctement. Vu que l'axe **Z** représente (~~malheureusement~~) souvent l'axe vertical alors nous donnerons cette valeur par défaut à l'attribut **m_axeVertical**.

Code : C++

```
Camera::Camera() : m_phi(0.0), m_theta(0.0), m_orientation(),
m_axeVertical(0, 0, 1), m_deplacementLateral(), m_position(),
m_pointCible()
{

}
```

Le second constructeur est identique au premier sauf que l'on initialise ici les vecteurs **m_position**, **m_pointCible** et **m_axeVertical** avec les paramètres fournis :

Code : C++

```
Camera::Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3
axeVertical) : m_phi(0.0), m_theta(0.0), m_orientation(),
m_axeVertical(axeVertical),
m_deplacementLateral(), m_position(position),
m_pointCible(pointCible)
{



}
```

En théorie, ce constructeur ne devrait pas être modifié mais nous allons quand même y apporter une modification temporaire. Je vais vous demander d'affecter une certaine valeur aux angles, je vous expliquerai pourquoi en dernière partie de chapitre (je ne vais pas alourdir les explications pour le moment 😊).

Ainsi, affectez la valeur **-35.26** à l'angle **Phi** et **-135** à l'angle **Theta**.

Code : C++

```
Camera::Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3
axeVertical) : m_phi(-35.26), m_theta(-135), m_orientation(),
m_axeVertical(axeVertical),
m_deplacementLateral(), m_position(position),
m_pointCible(pointCible)
{



}
```

Pour finir, l'implémentation du destructeur se passe de commentaires. 😊

Code : C++

```
Camera::~Camera()
{
}
```

La méthode orienter

Alimentation des angles

La méthode que nous allons coder va être la plus importante du chapitre, c'est grâce à elle que l'on va pouvoir calculer les coordonnées sphériques du vecteur **orientation**. Nous l'appellerons la méthode **orienter()**, elle prendra en paramètres 2 **integer** représentant les *coordonnées relatives* de la souris.



Je vous rappelle que les coordonnées relatives représentent le mouvement effectué par la souris depuis l'affichage précédent, elles sont modifiées à chaque affichage (60 fois par second pour nous).

Voici le prototype de cette méthode :

Code : C++

```
void orienter(int xRel, int yRel);
```

Avant de commencer l'implémentation de cette méthode, nous allons revoir un petit peu les angles **Phi** et **Theta**. Je vous avais dit que ces angles allaient être alimentés par les mouvements de la souris :

- L'angle **Phi** : représentant l'orientation sur l'*axe vertical* sera alimenté par la coordonnée *y* de la souris
- L'angle **Theta** : représentant l'orientation sur l'*axe horizontal* sera alimentée par la coordonné *x* de la souris

A chaque tour de la boucle principale, nous devons modifier ces angles en fonction du mouvement de la souris.

Par exemple, si on bouge la souris vers le haut alors l'angle **Phi** s'agrandira. Au niveau du code, on additionne l'angle **Phi** par le petit mouvement vertical **yRel** :

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
    // Modification des angles

    m_phi += -yRel;
}
```



Hey tu t'es trompé non ? Tu as rajouté un signe - devant le paramètre **yRel** ?

Le sens trigonométrique

Non pas du tout. 🍪

C'est quelque chose que vous ne pouviez pas forcément savoir, mais lorsque l'on travaille avec des angles il y a un certain sens à respecter qui s'appelle le **sens trigonométrique** (sens inverse des aiguilles d'une montre). Ce sens étant inversé, les angles se retrouvent donc eux-aussi inversés.

Ainsi, un mouvement de souris allant vers le haut (donc une **augmentation** de l'angle **Phi**) sera interprété par une augmentation de son opposé (donc **-Phi**). Même chose pour les mouvements horizontaux, l'angle **Theta** sera inversé.



Mais avant on a jamais inversé nos angles ?

Dans la plupart des cas, nous n'aurons pas à inverser nos angles. Cependant ici, il y a un contre-sens entre le *sens trigonométrique* et le *sens normal* de la souris.

Si vous bougez votre souris vers le haut alors sa position **Y** va augmenter, elle utilise (grossièrement parlant) le **sens horaire**. Or un angle, lui, va utiliser le **sens anti-horaire**. Donc pour compenser cette inversion, il faut additionner **l'opposé de l'angle**.

D'ailleurs en parlant de ça, vous avez probablement remarqué que dans certains jeux, on vous offrait la possibilité d'inverser le sens des axes **X** et **Y**. Cette option spécifie simplement à votre caméra s'il faut utiliser le sens horaire ou le sens trigonométrique, donc additionner l'angle normal ou son opposé. Si vous avez envie d'implémenter la même option un jour vous savez maintenant comment faire. 😊

En définitif, nous devons additionner l'opposé des mouvements générés par la souris à cause du sens trigonométrique.

L'angle **Theta** ne fait d'ailleurs pas exception :

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
    // Modification des angles

    m_phi += -yRel;
    m_theta += -xRel;
}
```

Nous allons rajouter encore une toute petite chose à ces calculs (ne vous inquiétez pas, il n'y a rien de tordu cette fois 🍪). Vous verrez tout à l'heure que si nous utilisons les coordonnées relatives directement comme ça, la caméra va bouger trop vite et nous fera des mouvements bizarres. Pour régler ce problème, on va abaisser les coordonnées relatives en les multipliant par **0.5**. De cette façon, les mouvements de la caméra seront deux fois moins rapides, ce qui sera utile pour mieux voir. 😊

Dans la dernière partie, nous rajouterons un attribut pour gérer cette vitesse avec un setter :

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
    // Modification des angles

    m_phi += -yRel * 0.5;
    m_theta += -xRel * 0.5;
}
```

Calcul des coordonnées sphériques

Nous avons maintenant des angles actualisés en fonction de la souris, c'est bien mais il faut encore imposer une limite à un angle

en particulier. En effet, dans la première partie du chapitre je vous avais dit que le vecteur **orientation** ne devait jamais être parallèle avec l'axe **Y** vous vous souvenez ? Car dans ce cas, nous ne pouvions plus utiliser la formule des coordonnées sphériques.

Pour éviter de se retrouver devant cette alignment, il faut limiter l'angle **Phi** à une valeur maximale de **89°** (ou **-89°** dans l'autre sens) sinon l'une des deux règles sur le calcul de la normale sera violée. Donc en imposant cette limite, nous pourrons toujours utiliser le plan formé par les deux vecteurs pour le calcul du déplacement latéral. 😊

On implémente donc deux conditions pour limiter l'attribut **m_phi** à une valeur de **89°** ou **-89°** :

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
    // Récupération des angles

    m_phi += -yRel * 0.5;
    m_theta += -xRel * 0.5;

    // Limitation de l'angle phi

    if(m_phi > 89.0)
        m_phi = 89.0;

    else if(m_phi < -89.0)
        m_phi = -89.0;
}
```

Nous avons maintenant des angles parfaitement utilisables, nous allons pouvoir passer au calcul des coordonnées sphériques de l'orientation. Et pour cela on va utiliser une formule que nous avons déjà vue. 🍪

$$\begin{aligned} \mathbf{x} &= \cos(\phi) \times \sin(\theta) \\ \mathbf{y} &= \sin(\phi) \\ \mathbf{z} &= \cos(\phi) \times \cos(\theta) \end{aligned}$$

Nous devons donc transposer cette formule en code source. On utilisera le setter tout bête de chaque coordonnée du vecteur **m_orientation** pour affecter le résultat :

Code : C++

```
// Calcul des coordonnées sphériques

m_orientation.x = cos(m_phi) * sin(m_theta);
m_orientation.y = sin(m_phi);
m_orientation.z = cos(m_phi) * cos(m_theta);
```

....

....

Vous ne voyez pas une petite erreur dans ce code ?



Euh non je pense pas ... Il y en a une ?

Hum bon c'est un peu sadique je l'avoue, mais je veux que vous intégrez bien cette notion : le problème ici c'est qu'on envoie des angles exprimés en **degrés** alors que les fonctions **sin()** et **cos()** attendent des angles exprimés **radians**. Si on ne modifie pas ça, on risque d'avoir une sacrée surprise au moment de déplacer la caméra. 😊

Pour corriger le problème, nous allons convertir les angles **Phi** et **Theta** en radian. On stockera les nouveaux angles dans des variables temporaires que l'on utilisera pour calculer les coordonnées sphériques.

Je vous rappelle que pour convertir un angle en radian il faut le multiplier par **Pi** puis le diviser par **180** :

Code : C++

```
// Conversion des angles en radian

float phiRadian = m_phi * M_PI / 180;
float thetaRadian = m_theta * M_PI / 180;

// Calcul des coordonnées sphériques

m_orientation.x = cos(phiRadian) * sin(thetaRadian);
m_orientation.y = sin(phiRadian);
m_orientation.z = cos(phiRadian) * cos(thetaRadian);
```

Cette fois le calcul est fonctionnel car nous utilisons bien des angles exprimés en **radians**. Cependant et malgré ça, il subsiste encore un petit problème.

En effet, la formule avec laquelle je vous rabâche la tête depuis tout à l'heure n'est valable que si l'on utilise l'axe **Y**. Or tout le monde n'utilise pas forcément cet axe, il faut gérer les axes **X** et **Z** également.

Pour cela, il n'y a rien de compliqué puisqu'il suffit juste d'interchanger les *setters* du vecteur **orientation** entre eux. Je vous donne les formules pour les 3 axes **X**, **Y** et **Z** :

Coordonnées sphériques pour l'axe X

$$\begin{aligned}x &= \sin(\phi) \\y &= \cos(\phi) \times \cos(\theta) \\z &= \cos(\phi) \times \sin(\theta)\end{aligned}$$

Coordonnées sphériques pour l'axe Y

$$\begin{aligned}x &= \cos(\phi) \times \sin(\theta) \\y &= \sin(\phi) \\z &= \cos(\phi) \times \cos(\theta)\end{aligned}$$

(Celle-là on la connaît)

Coordonnées sphériques pour l'axe Z

$$\begin{aligned}x &= \cos(\phi) \times \cos(\theta) \\y &= \cos(\phi) \times \sin(\theta) \\z &= \sin(\phi)\end{aligned}$$

Pour gérer ces 3 formules nous allons simplement utiliser 3 blocs **if()** qui vont tester les coordonnées de l'attribut **m_axeVertical**. Celui qui possède la valeur **1.0** représentera l'axe vertical :

Code : C++

```
// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
{
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
}

// Sinon c'est l'axe Z

else
{}
```

Si le développeur n'a pas rentré la valeur **1.0** à une coordonnée alors ce sera l'axe **Z** qui sera choisi. Le développeur doit faire attention à la valeur qu'il donne pour ce vecteur, c'est sa responsabilité. 😊

Une fois les blocs créés, il suffit d'associer la bonne formule aux bonnes coordonnées :

Code : C++

```
// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
```

```

{
    // Calcul des coordonnées sphériques

    m_orientation.x = sin(phiRadian);
    m_orientation.y = cos(phiRadian) * cos(thetaRadian);
    m_orientation.z = cos(phiRadian) * sin(thetaRadian);
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
    // Calcul des coordonnées sphériques

    m_orientation.x = cos(phiRadian) * sin(thetaRadian);
    m_orientation.y = sin(phiRadian);
    m_orientation.z = cos(phiRadian) * cos(thetaRadian);
}

// Sinon c'est l'axe Z

else
{
    // Calcul des coordonnées sphériques

    m_orientation.x = cos(phiRadian) * cos(thetaRadian);
    m_orientation.y = cos(phiRadian) * sin(thetaRadian);
    m_orientation.z = sin(phiRadian);
}

```

Cette fois, le code est totalement fonctionnel. 😊

Dernier point à préciser, on devrait en théorie *normaliser* le vecteur **orientation**, mais vu que nous utilisons des fonctions trigonométriques le vecteur est déjà normalisé. En effet, lorsqu'on utilise la trigonométrie, le vecteur qui représente le rayon a toujours une norme égale à 1 donc il est normalisé. 😊

Calcul du vecteur de déplacement latéral

Nous avons maintenant le fameux vecteur **orientation**, nous pouvons donc l'utiliser pour le calcul du vecteur de **déplacement latéral**.

Vous vous souvenez que pour le trouver, il fallait multiplier le vecteur **orientation** par le vecteur représentant l'**axe vertical**. On trouve ainsi la normale du plan formé par ces deux vecteurs.

Quand je parle de la multiplication c'est un peu un abus de langage car il s'agit en fait d'un produit vectoriel (en comparaison du produit d'une multiplication). Le produit vectoriel permet de trouver le fameux vecteur orthogonal que nous cherchons.

Ce produit ne se fait pas avec un signe * classique, il faut plutôt utiliser une méthode de la librairie **GLM** qui s'appelle **cross()** :

Code : C++

```
glm::vec3 cross(glm::vec3 vector1, glm::vec3 vector2);
```

Cette méthode retourne le vecteur orthogonal des deux paramètres donnés.

Nous l'appelons donc en lui donnant les attributs **m_axeVertical** et **m_orientation** pour qu'elle puisse le calculer :

Code : C++

```
// Calcul de la normale  
  
m_deplacementLateral = cross(m_axeVertical, m_orientation);
```

On pensera cette fois à normaliser le résultat car on n'utilise pas la trigo, ce qui fait que rien n'est normalisé tant qu'on ne l'a pas demandé. Mais encore une fois, la librairie **GLM** nous fournit directement une méthode pour normaliser les vecteurs. Nous n'avons rien à faire décidément. 😊

Cette méthode s'appelle **normalize()** :

Code : C++

```
glm::vec3 normalize(glm::vec3 vector);
```

Elle prend en paramètre le vecteur à normaliser et renvoie le résultat.

Nous l'appellerons en lui donnant le vecteur que l'on vient de calculer, à savoir **m_deplacementVertical** :

Code : C++

```
// Calcul de la normale  
  
m_deplacementLateral = cross(m_axeVertical, m_orientation);  
m_deplacementLateral = normalize(m_deplacementLateral);
```

Actualisation du point cible

Enfin pour terminer cette méthode, il ne reste plus qu'à actualiser le *point fixé* par la matrice **modelview** (l'un des trois paramètres dont à besoin la matrice).

Pour trouver ce point, on va utiliser une petite astuce : on va additionner le vecteur **position** avec le vecteur **orientation**. Le résultat sera le point qu'attend la matrice, soit le point pile en face de la position de la caméra.



On peut additionner deux objets **vec3** ? Ça doit être compliqué non ?

Non pas du tout car les développeurs de **GLM** ont pensé à tout et ont intégré, au même titre que les matrices, les surcharges d'opérateurs. Nous pouvons donc les additionner sans problème juste en utilisant le signe +. Génial non ? 😊

Pour additionner le vecteur **position** et **orientation**, il nous suffit donc d'utiliser le signe + :

Au niveau du code, on additionne simplement ces deux vecteurs :

Code : C++

```
// Calcul du point cible pour OpenGL  
  
m_pointCible = m_position + m_orientation;
```

Addition terminée. 🎉



Et ce vecteur on ne le normalise pas ?

Nan, les vecteurs **position** et **pointCible** ne doivent jamais être normalisés car ils représentent des positions dans l'espace. Ils permettent à *OpenGL* de savoir où l'on se trouve dans un monde 3D, et un monde 3D ne se limite pas à des positions de 1 unité maximum.

Les vecteurs normalisés servent pour les calculs diverses comme l'**orientation** et le vecteur **déplacementLateral**.

Récapitulation

Si on récapitule tout ce que l'on vient de coder :

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
    // Récupération des angles

    m_phi += -yRel * 0.5;
    m_theta += -xRel * 0.5;

    // Limitation de l'angle phi

    if(m_phi > 89.0)
        m_phi = 89.0;

    else if(m_phi < -89.0)
        m_phi = -89.0;

    // Conversion des angles en radian

    float phiRadian = m_phi * M_PI / 180;
    float thetaRadian = m_theta * M_PI / 180;

    // Si l'axe vertical est l'axe X

    if(m_axeVertical.x == 1.0)
    {
        // Calcul des coordonnées sphériques

        m_orientation.x = sin(phiRadian);
        m_orientation.y = cos(phiRadian) * cos(thetaRadian);
        m_orientation.z = cos(phiRadian) * sin(thetaRadian);
    }

    // Si c'est l'axe Y

    else if(m_axeVertical.y == 1.0)
    {
        // Calcul des coordonnées sphériques

        m_orientation.x = cos(phiRadian) * sin(thetaRadian);
        m_orientation.y = sin(phiRadian);
        m_orientation.z = cos(phiRadian) * cos(thetaRadian);
    }
}
```

```
// Sinon c'est l'axe Z

else
{
    // Calcul des coordonnées sphériques

    m_orientation.x = cos(phiRadian) * cos(thetaRadian);
    m_orientation.y = cos(phiRadian) * sin(thetaRadian);
    m_orientation.z = sin(phiRadian);
}

// Calcul de la normale

m_deplacementLateral = cross(m_axeVertical, m_orientation);
m_deplacementLateral = normalize(m_deplacementLateral);

// Calcul du point ciblé pour OpenGL

m_pointCible = m_position + m_orientation;
}
```

Grâce à cette méthode, nous pouvons calculer non seulement l'*orientation* de la caméra mais aussi son vecteur *orthogonal* qui sera utilisé pour le déplacement latéral. 😊

La méthode déplacer

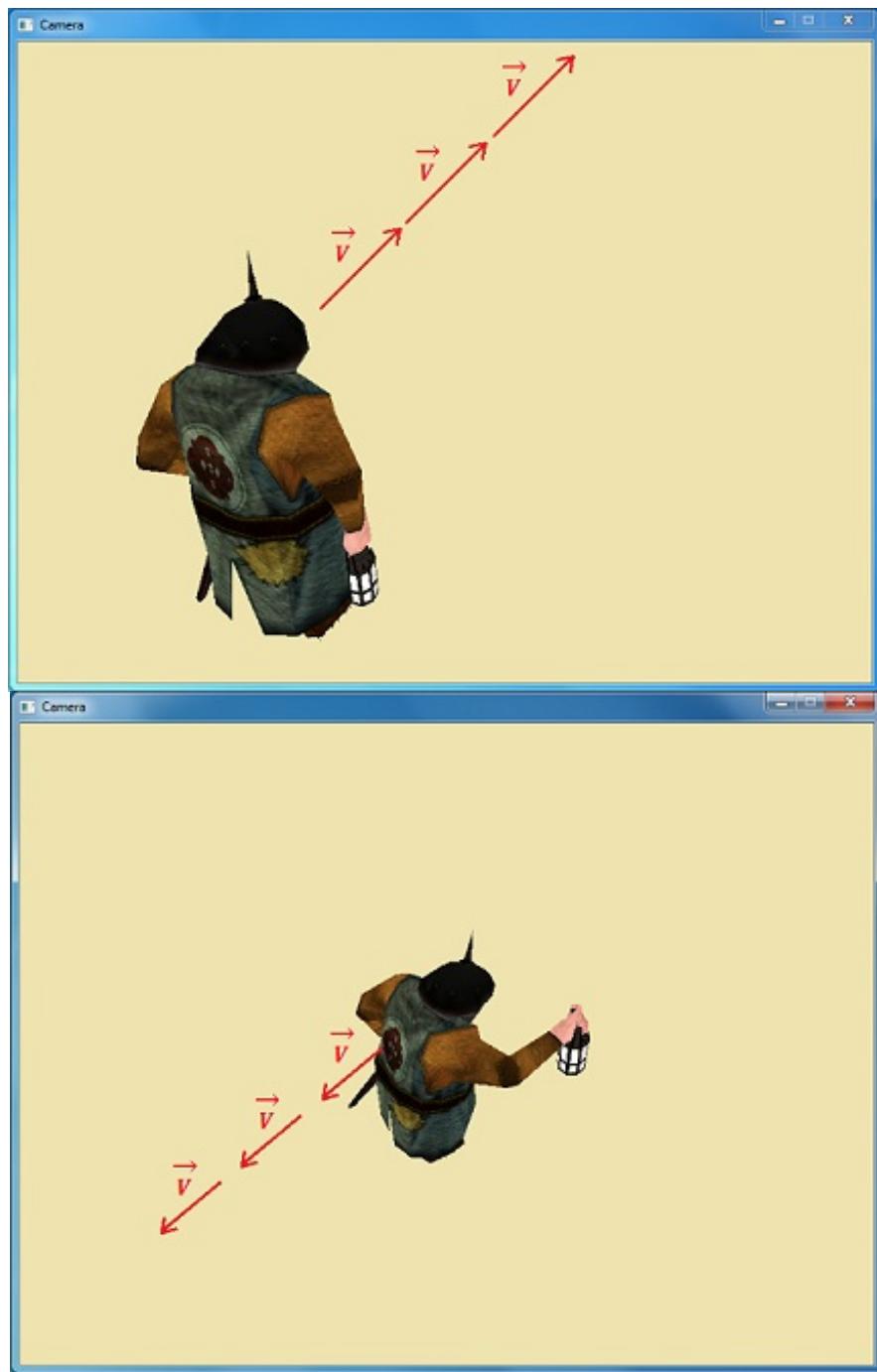
La méthode suivante va permettre de s'occuper du déplacement pur et dur de la caméra. Nous savons déjà comment faire en plus ça ne sera pas compliqué. 😊

Nous appellerons cette méthode la méthode **deplacer()**, elle prendra en paramètre une *référence constante* sur un objet de type **Input**. Car oui, nous aurons besoin de savoir si les touches de déplacement sont pressées ou non, nous avons donc besoin d'un objet de ce type.

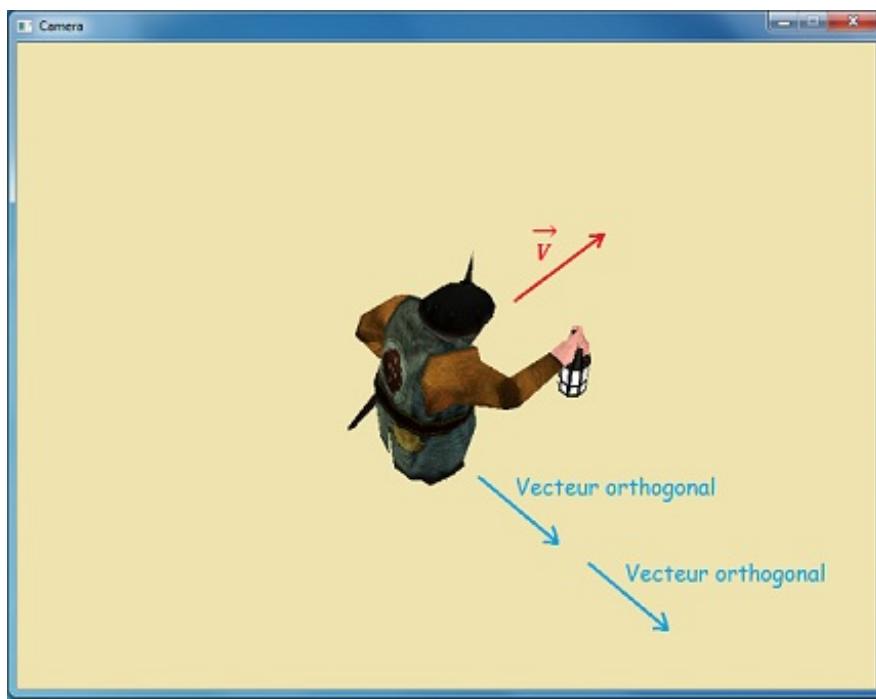
Code : C++

```
void deplacer(Input const &input);
```

Pour faire avancer ou reculer la caméra on additionnera ou on soustraira le vecteur **orientation** :



Et pour la déplacer latéralement on additionnera ou on soustraira le vecteur ***orthogonal*** au vecteur ***orientation*** :



Au niveau du code, on va encapsuler les quatre touches de déplacement dans des *blocs if*. Si une touche est pressée, alors on effectue l'action désirée (addition ou soustraction de vecteurs).

Par exemple, si la touche Haut (**SDL_SCANCODE_UP**) est pressée alors on additionnera le vecteur **orientation** avec le vecteur **position** pour faire avancer la caméra. On ajoutera ici aussi une contrainte de vitesse exactement comme pour l'orientation de la caméra (on créera également un attribut pour gérer ça plus tard). Nous diviserons la vitesse de déplacement par **2** en multipliant le vecteur **orientation** par **0.5** (un vecteur multiplié par un nombre est différent d'un calcul vectoriel. On multiplie juste les coordonnées par 0.5 dans ce cas 😊) :

Code : C++

```
void Camera::deplacer(Input const &input)
{
    // Avancée de la caméra

    if(input.getTouche(SDL_SCANCODE_UP))
        m_position = m_position + m_orientation * 0.5f;
}
```

Lorsque la position de la caméra est modifiée alors le point ciblé par *OpenGL* doit également être modifié. Pour cela, on fait exactement ce que l'on a fait dans la méthode **orienter()** :

Code : C++

```
void Camera::deplacer(Input const &input)
{
    // Avancée de la caméra

    if(input.getTouche(SDL_SCANCODE_UP))
    {
        m_position = m_position + m_orientation * 0.5f;
        m_pointCible = m_position + m_orientation;
    }
}
```

Grâce à cette condition, notre caméra peut avancer quand nous voulons. 😊

D'ailleurs pour la faire reculer ce n'est pas plus compliqué, on fait exactement la même chose sauf que cette fois on soustrait le vecteur **position** et le vecteur **orientation**. On utilisera le scancode **SDL_SCANCODE_DOWN** pour la touche du clavier :

Code : C++

```
// Recul de la caméra

if(input.getTouche(SDL_SCANCODE_DOWN))
{
    m_position = m_position - m_orientation * 0.5f;
    m_pointCible = m_position + m_orientation;
}
```

Le principe ne change pas pour le déplacement latéral : si on veut aller vers la gauche on additionne le vecteur **m_deplacementLateral** à la position, et si on veut aller vers la droite on le soustrait. On utilisera respectivement les scancodes **SDL_SCANCODE_LEFT** et **SDL_SCANCODE_RIGHT** :

Code : C++

```
// Déplacement vers la gauche

if(input.getTouche(SDL_SCANCODE_LEFT))
{
    m_position = m_position + m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}

// Déplacement vers la droite

if(input.getTouche(SDL_SCANCODE_RIGHT))
{
    m_position = m_position - m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}
```

Si on récapitule tout ça :

Code : C++

```
void Camera::deplacer(Input const &input)
{
    // Avancée de la caméra

    if(input.getTouche(SDL_SCANCODE_UP))
    {
        m_position = m_position + m_orientation * 0.5f;
        m_pointCible = m_position + m_orientation;
    }

    // Recul de la caméra

    if(input.getTouche(SDL_SCANCODE_DOWN))
    {
        m_position = m_position - m_orientation * 0.5f;
        m_pointCible = m_position + m_orientation;
    }
}
```

```
// Déplacement vers la gauche

if(input.getTouche(SDL_SCANCODE_LEFT))
{
    m_position = m_position + m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}

// Déplacement vers la droite

if(input.getTouche(SDL_SCANCODE_RIGHT))
{
    m_position = m_position - m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}
```



Petite question : pourquoi n'utilise-t-on pas des **else if** ?

Simplement pour être capable d'utiliser plusieurs touches à la fois. 😊 Si on utilisait des blocs **else if**, il n'y aurait qu'une seule touche qui serait gérée pour se déplacer.

Il ne manque plus qu'une petite chose à cette méthode. En effet, vu qu'on a l'objet **input** sous la main on peut en profiter pour savoir si il y a eu un mouvement de souris. Et s'il y a mouvement de souris alors il y a une modification de l'orientation et donc un appel à la méthode **orienter()**. 😊

On va donc rajouter une condition qui sera déclenchée lors d'un mouvement de la souris. A l'intérieur, on appellera la méthode **orienter()** à laquelle on donnera en paramètres les attributs **m_xRel** et **m_yRel** de l'objet **input**.

Code : C++

```
// Gestion de l'orientation

if(input.mouvementSouris())
    orienter(input.getXRel(), input.getYRel());
```

Méthode finale :

Code : C++

```
void Camera::deplacer(Input const &input)
{
    // Gestion de l'orientation

    if(input.mouvementSouris())
        orienter(input.getXRel(), input.getYRel());

    // Avancée de la caméra

    if(input.getTouche(SDL_SCANCODE_UP))
    {
        m_position = m_position + m_orientation * 0.5f;
        m_pointCible = m_position + m_orientation;
    }
```

```
// Recul de la caméra

if(input.getTouche(SDL_SCANCODE_DOWN))
{
    m_position = m_position - m_orientation * 0.5f;
    m_pointCible = m_position + m_orientation;
}

// Déplacement vers la gauche

if(input.getTouche(SDL_SCANCODE_LEFT))
{
    m_position = m_position + m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}

// Déplacement vers la droite

if(input.getTouche(SDL_SCANCODE_RIGHT))
{
    m_position = m_position - m_deplacementLateral * 0.5f;
    m_pointCible = m_position + m_orientation;
}
```

Avec cette méthode, nous pouvons gérer complètement le déplacement de la caméra. 😊

Méthode lookAt

On arrive à la dernière méthode de notre caméra. 🤔

Jusqu'à maintenant, nous avons utilisé pas mal de vecteurs mais vous savez que seuls 3 d'entre eux sont vraiment importants : la **position**, le **point ciblé** et l'**axe vertical**. Ces trois vecteurs sont indispensables pour la matrice **modelview**, nous devons donc les lui envoyer.

Nous allons coder une nouvelle méthode pour s'occuper de ça, on l'appellera **lookAt()** en référence à la méthode du même nom chez **GLM**. Elle prendra en paramètre une référence (non constante) sur la matrice **modelview** :

Code : C++

```
void lookAt(glm::mat4 &modelview);
```

Pour son implémentation, on appelle simplement la méthode **lookAt()** de la matrice **modelview**, on lui donne au passage les 3 vecteurs qu'elle demande. Faites attention à utiliser le namespace **glm::** ici même si vous utilisez le **using** dans votre fichier. Vu que nous avons deux méthodes portant le même nom, il faut utiliser le namespace pour les différencier :

Code : C++

```
void Camera::lookAt(glm::mat4 &modelview)
{
    // Actualisation de la vue dans la matrice

    modelview = glm::lookAt(m_position, m_pointCible,
    m_axeVertical);
}
```

Implémentation de la caméra

Notre caméra est maintenant totalement opérationnelle ! Il ne manque plus qu'à l'implémenter dans notre scène. 😊

On ajoute donc l'en-tête **Camera.h** dans le header **SceneOpenGL.h**, puis on déclare un objet de type **Camera** dans la méthode **bouclePrincipale()**. On placera cette caméra au point de coordonnées **(3, 3, 3)** et elle ciblera le point de coordonnées **(0, 0, 0)**. Bien entendu, l'axe vertical reste l'**axe y (0, 1, 0)**.

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    Uint32 frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Caméra mobile

    Camera camera(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0));

    ...
}
```

On en profite au passage pour piéger et cacher le pointeur de la souris grâce aux méthodes suivantes :

Code : C++

```
// Capture du pointeur

m_input.afficherPointeur(false);
m_input.capturerPointeur(true);
```

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    Uint32 frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Matrices
```

```

mat4 projection;
mat4 modelview;

projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
modelview = mat4(1.0);

// Caméra mobile

Camera camera(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0));
m_input.afficherPointeur(false);
m_input.capturerPointeur(true);

...
}

```

Pour les **linuxiens**, il semble qu'il y ait un problème avec la méthode **capturerPointeur()**. La fonction **SDL_SetRelativeMouseMode()** qui est utilisée à l'intérieur ne semble pas fonctionner chez vous. C'est un problème qui existe depuis la **SDL 1.3**, j'espère que la librairie va corriger ce problème rapidement pour que vous n'ayez plus de soucis. 😞

Ce qui est bizarre c'est que tout fonctionne correctement sous Windows mais pas sous Linux...

Ensuite, on appelle la méthode **deplacer()** de l'objet **camera** sans oublier de donner l'attribut **m_input** pour qu'il puisse travailler avec :

Code : C++

```

// Boucle principale

while (!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if (m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    camera.deplacer(m_input);

    ...
}

```

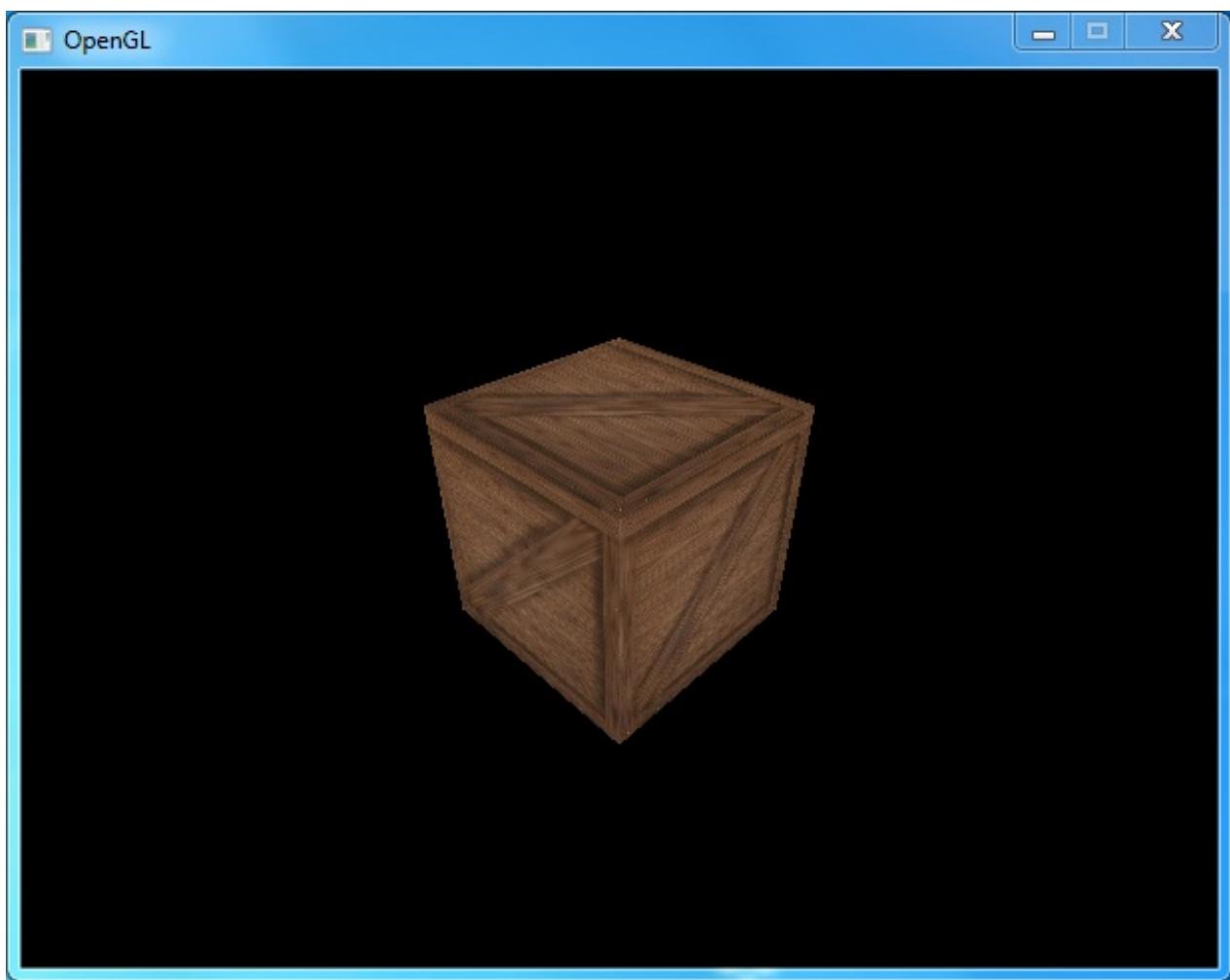
Enfin, on remplace la méthode **lookAt()** de la matrice **modelview** par la méthode **lookAt()** de la **caméra**. On n'oublie pas de lui donner le paramètre qu'elle attend soit la matrice **modelview** elle-même :

Code : C++

....

```
// Nettoyage de l'écran  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  
// Gestion de la caméra  
camera.lookAt(modelview);  
  
....
```

Et là, vous pouvez compiler joyeusement votre code source. 😊



Essayez votre nouvelle caméra, vous pouvez vous déplacer dans toutes les directions sans aucun problème. 😊

Notre caméra est maintenant quasi-complète, il manque en réalité encore une méthode. En effet, tout à l'heure je vous ai demandé de mettre des valeurs par défaut pour les angles dans le constructeur. Or cette solution ne fonctionne que dans le cas où la caméra se trouve au point de coordonnées (3, 3, 3). Dans la dernière partie de ce chapitre, nous allons coder une nouvelle méthode qui permettra de régler ce problème.

Fixer un point

Dans ces deux dernières parties, nous allons ajouter quelques petites fonctionnalités à la classe **Camera**. Nous nous occuperons :

- D'ajouter une **méthode** pour cibler un point

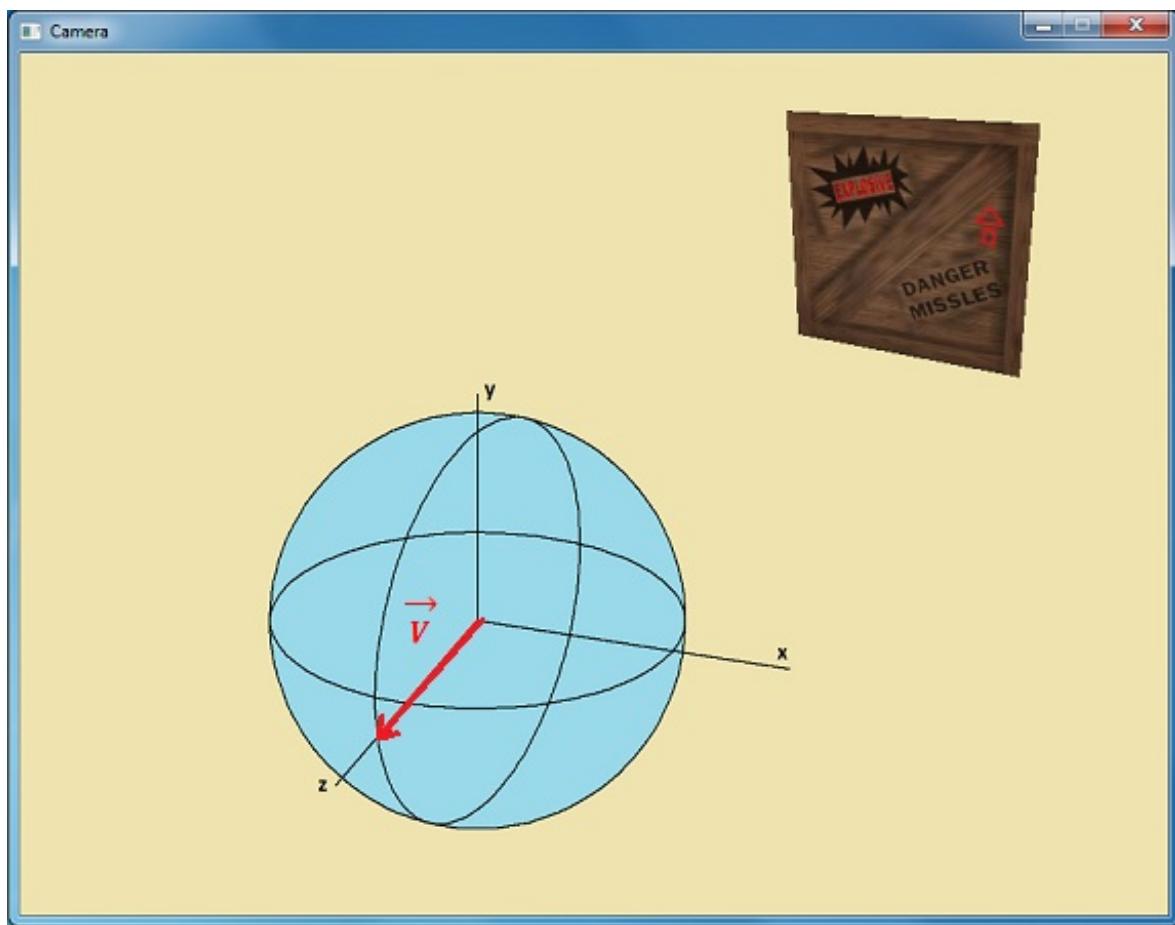
- D'ajouter un **setter** pour la position de la caméra
- D'ajouter deux **attributs** pour gérer sa sensibilité (orientation) et sa vitesse (déplacement)

Nous commencerons évidemment par le premier point. 😊 Les autres seront développés dans la dernière partie.

Fonctionnalités de la classe Camera

La méthode `setPointable()`

Tout à l'heure, je vous avais demandé d'affecter certaines valeurs aux angles **Theta** et **Phi** dans le constructeur. Si vous n'aviez pas fait cela, vos angles auraient été initialisés avec la valeur **0** et votre regard aurait été complètement inversé par rapport au point à cibler :



Vous auriez bien placé votre caméra mais elle ne serait pas du tout orientée de la bonne façon malgré le fait que vous auriez donné en paramètre le **point à cibler**. Elle serait orientée selon la valeur des angles soit la valeur **0** à cause du constructeur. Rappelez-vous qu'à chaque fois on ne fait qu'additionner les changements de position et les angles partent avec leur valeur d'origine :

Code : C++

```
// Modification des angles  
  
m_phi += -yRel * 0.5;  
m_theta += -xRel * 0.5;
```

Pour régler ce problème, il faut que les angles soient initialisés avec la bonne orientation dès le départ. Et pour ça, nous allons coder une méthode qui fera l'inverse de la méthode **orienter()**. Elle devra être capable de calculer la valeur initiale des angles à partir des vecteurs qu'on lui envoie.

Cette méthode va être facile à coder, il suffit juste de reprendre la formule des coordonnées sphériques et d'en inverser les opérations pour trouver la valeur des angles. 🧐 On commence avec la formule valable pour l'axe Y :

$$x = \cos(\phi) \times \sin(\theta)$$

$$y = \sin(\phi)$$

$$z = \cos(\phi) \times \cos(\theta)$$

$$y = \sin(\phi)$$

$$z = \cos(\phi) \times \cos(\theta)$$

$$y = \sin(\phi)$$

$$z = \cos(\phi) \times \cos(\theta)$$

$$\sin^{-1}(y) = \sin^{-1}(\sin(\phi))$$

$$\frac{z}{\cos(\phi)} = \frac{\cos(\phi) \times \cos(\theta)}{\cos(\phi)}$$

$$\sin^{-1}(y) = \phi$$

$$\frac{z}{\cos(\phi)} = \cos(\theta)$$

$$\cos^{-1}\left(\frac{z}{\cos(\phi)}\right) = \cos^{-1}(\cos(\theta))$$

$$\cos^{-1}\left(\frac{z}{\cos(\phi)}\right) = \theta$$

Calcul des angles Phi et Theta pour l'axe Y

$$\phi = \sin^{-1}(y)$$

$$\theta = \cos^{-1}\left(\frac{z}{\cos(\phi)}\right)$$

Avec cette formule, on peut trouver les angles **Theta** et **Phi** à partir des coordonnées (x, y, z) - coordonnées qui appartiennent au vecteur orientation dans le cas où on utilise l'axe Y. Pour trouver les angles avec les autres axes, il suffit juste d'interchanger les formules de départ pour tomber sur les résultats suivants :

Calcul des angles Phi et Theta pour l'axe X

$$\begin{aligned}\Phi &= \sin^{-1}(x) \\ \theta &= \cos^{-1}\left(\frac{y}{\cos(\phi)}\right)\end{aligned}$$

Calcul des angles Phi et Theta pour l'axe Z

$$\begin{aligned}\Phi &= \sin^{-1}(x) \\ \theta &= \cos^{-1}\left(\frac{z}{\cos(\phi)}\right)\end{aligned}$$

Vous voyez que la formule reste la même, il n'y a que la coordonnée utilisée qui va changer. 😊



Oui c'est bien tout ça mais on a besoin des coordonnées du vecteur **orientation** pour calculer les angles non ? Or on ne l'a pas celui-là à l'initialisation de la caméra ?

Oui tout à fait nous ne l'avons pas à l'initialisation de la caméra. Mais encore une fois, nous pouvons le trouver en inversant une autre formule mathématique.

Et oui, souvenez-vous que pour trouver le vecteur **pointCible**, on additionne le vecteur **position** et le vecteur **orientation** :

Code : C++

```
// Calcul du point ciblé pour OpenGL
m_pointCible = m_position + m_orientation;
```

pointCible = position + orientation

Donc pour trouver le vecteur **orientation** (et surtout ses coordonnées), il suffit de soustraire le vecteur **pointCible** par le vecteur **position** :

$$\text{pointCible} = \text{position} + \text{orientation}$$

$$\text{pointCible} - \text{position} = \text{position} + \text{orientation} - \text{position}$$

$$\text{pointCible} - \text{position} = \text{orientation}$$

$$\text{orientation} = \text{pointCible} - \text{position}$$

Grâce à cette soustraction, on va pouvoir calculer la valeur initiale des angles au moment de l'initialisation de la caméra. 😊

Pour effectuer tous ces calculs, nous utiliserons une méthode, ou plutôt un setter sur l'attribut **m_pointCible** que nous appellerons simplement **setPointcible()**. Il prendra en paramètre le vecteur à cibler :

Code : C++

```
void setPointcible(glm::vec3 pointCible);
```

On commence son implémentation en soustrayant les vecteurs **pointCible** et **position** pour trouver le vecteur **orientation**. Comme d'habitude, on normalisera ce vecteur :

Code : C++

```
void Camera::setPointcible(glm::vec3 pointCible)
{
    // Calcul du vecteur orientation
    m_orientation = m_pointCible - m_position;
    m_orientation = normalize(m_orientation);
}
```

Une fois le vecteur **orientation** trouvé, on peut maintenant calculer les angles **Phi** et **Theta** grâce aux formules inversées que l'on a trouvées précédemment :

Calcul des angles Phi et Theta pour l'axe X

$$\begin{aligned}\Phi &= \sin^{-1}(x) \\ \theta &= \cos^{-1}\left(\frac{y}{\cos(\Phi)}\right)\end{aligned}$$

Calcul des angles Phi et Theta pour l'axe Y

$$\begin{aligned}\Phi &= \sin^{-1}(y) \\ \theta &= \cos^{-1}\left(\frac{z}{\cos(\Phi)}\right)\end{aligned}$$

Calcul des angles Phi et Theta pour l'axe Z

$$\begin{aligned}\Phi &= \sin^{-1}(x) \\ \theta &= \cos^{-1}\left(\frac{z}{\cos(\Phi)}\right)\end{aligned}$$

Comme pour la méthode **orienter()**, on commence par faire 3 blocs **if** qui vont permettre de gérer le cas des trois axes verticaux :

Code : C++

```
// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
{
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
}

// Sinon c'est l'axe Z

else
{}
```

Une fois que c'est fait, on associe la bonne formule au bon axe. Je vous donne l'exemple du cas où on utilise l'axe **Y**:

Code : C++

```
// Calcul des angles pour l'axe Y

m_phi = asin(m_orientation.y);
m_theta = acos(m_orientation.z / cos(m_phi));
```

Implémentation du bon calcul dans les blocs **if** :

Code : C++

```
// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.y / cos(m_phi));
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
    // Calcul des angles

    m_phi = asin(m_orientation.y);
    m_theta = acos(m_orientation.z / cos(m_phi));
}

// Sinon c'est l'axe Z

else
{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.z / cos(m_phi));
}
```

En temps normal nous pourrions nous arrêter là mais il existe un petit problème avec la coordonnée utilisée pour calculer l'angle **Theta**. En effet, les formules que nous utilisons ne sont valables que si cette coordonnée est supérieure à **0**. Si elle est inférieure à **0**, les formules vont donner une valeur opposée de l'angle (**-Theta**), ce qui inversera le résultat final.

Pour régler ce problème, il suffit d'ajouter une sous-condition dans chaque bloc **if** pour vérifier le **signe** de la coordonnée utilisée : s'il est négatif, alors on multiplie l'angle **Theta** par **-1** pour avoir l'opposée de l'opposée, et donc la vraie valeur. S'il est positif alors l'angle est déjà correcte, on ne fait rien.

Voici ce que ça donne dans le cas où on utilise l'axe **Y**:

Code : C++

```
// Calcul des angles

m_phi = asin(m_orientation.y);
m_theta = acos(m_orientation.z / cos(m_phi));

if(m_orientation.z < 0)
    m_theta *= -1;
```

Implémentation pour tous les blocs **if** :

Code : C++

```
// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
```

```

{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.y / cos(m_phi));

    if(m_orientation.y < 0)
        m_theta *= -1;
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
    // Calcul des angles

    m_phi = asin(m_orientation.y);
    m_theta = acos(m_orientation.z / cos(m_phi));

    if(m_orientation.z < 0)
        m_theta *= -1;
}

// Sinon c'est l'axe Z

else
{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.z / cos(m_phi));

    if(m_orientation.z < 0)
        m_theta *= -1;
}

```

On arrive à la fin de la méthode, il ne reste plus qu'une chose à faire : convertir les angles **Theta** et **Phi** en degrés pour pouvoir les utiliser avec les coordonnées relatives de la souris. Les fonctions **cos()** et **sin()** ne prennent et ne renvoient que des angles exprimés en **radians** (je ne l'ai pas déjà dit  alors que nous, nous avons besoin d'angles exprimés en **degrés** pour pouvoir travailler avec la souris. Il faut donc convertir les résultats en multipliant les angles par **180** puis en les divisant par **Pi** :

Code : C++

```

// Conversion en degrés

m_phi = m_phi * 180 / M_PI;
m_theta = m_theta * 180 / M_PI;

```

Si on résume tout ça :

Code : C++

```

void Camera::setPointCible(glm::vec3 pointCible)
{
    // Calcul du vecteur orientation

    m_orientation = m_pointCible - m_position;
    m_orientation = normalize(m_orientation);
}

```

```

// Si l'axe vertical est l'axe X

if(m_axeVertical.x == 1.0)
{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.y / cos(m_phi));

    if(m_orientation.y < 0)
        m_theta *= -1;
}

// Si c'est l'axe Y

else if(m_axeVertical.y == 1.0)
{
    // Calcul des angles

    m_phi = asin(m_orientation.y);
    m_theta = acos(m_orientation.z / cos(m_phi));

    if(m_orientation.z < 0)
        m_theta *= -1;
}

// Sinon c'est l'axe Z

else
{
    // Calcul des angles

    m_phi = asin(m_orientation.x);
    m_theta = acos(m_orientation.z / cos(m_phi));

    if(m_orientation.z < 0)
        m_theta *= -1;
}

// Conversion en degrés

m_phi = m_phi * 180 / M_PI;
m_theta = m_theta * 180 / M_PI;
}

```

Et voilà ! Grâce à cette méthode nous pouvons enfin régler le problème d'initialisation de la caméra. Et de plus, nous pouvons l'orienter quand on le veut (pour une cinématique par exemple). 😊

Implémentons sans plus tarder cette méthode dans le constructeur. Pour ça, on ré-initialise les angles **Theta** et **Phi** à 0 puis on appelle la méthode **setPointCible()** :

Code : C++

```

Camera::Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3
axeVertical) : m_phi(0.0), m_theta(0.0), m_orientation(),
m_axeVertical(axeVertical), m_deplacementLateral(),
m_position(position), m_pointCible(pointCible)
{
    // Actualisation du point cible

    setPointcible(pointCible);
}

```

}

Améliorations

Dans cette dernière partie, nous allons finir en douceur et voir les dernières petites fonctionnalités que nous allons ajouter à notre classe **Camera**. Je vous les rappelle :

- Ajouter un **setter** pour la position de la caméra
- Ajouter deux **attributs** pour gérer sa sensibilité (orientation) et sa vitesse (déplacement)

On commencera par le premier point. 😊

La méthode `setPosition()`

Bon je pense que vous avez l'habitude des setters maintenant, surtout que celui-ci va être plus simple que le précédent. 🍀

Nous allons coder un petit setter `setPosition()` qui nous permettra de positionner la caméra quand on le souhaitera :

Code : C++

```
void setPosition(glm::vec3 position);
```

Ce setter permet de mettre à jour l'attribut **m_position**. On utilisera le signe `=` qui nous permet de copier deux objets de type `vec3` facilement. Le seul point auquel il faut faire attention est le fait qu'il faut mettre à jour le **point cible** à chaque fois que l'on change de **position** :

Code : C++

```
void Camera::setPosition(glm::vec3 position)
{
    // Mise à jour de la position
    m_position = position;

    // Actualisation du point cible
    m_pointCible = m_position + m_orientation;
}
```

Bien entendu on ne normalise pas ce vecteur-là, il fait partie des deux exceptions. 😊

Les attributs sensibilité et rapidité

On passe à la fonctionnalité suivante, nous allons créer deux nouveaux paramètres qui vont nous permettre de modifier la vitesse de déplacement de la caméra.

Vous vous souvenez que je vous avais demandé de multiplier les **angles** et le vecteur **orientation** par **0.5** pour réduire la vitesse de la caméra ?

Code : C++

```
void Camera::orienter(int xRel, int yRel)
{
```

```
// Récupération des angles  
  
m_phi += -yRel * 0.5f;  
m_theta += -xRel * 0.5f;  
  
....  
}
```

Code : C++

```
void Camera::deplacer(Input const &input)  
{  
    ....  
  
    // Avancée de la caméra  
  
    if(input.getTouche(SDL_SCANCODE_UP))  
    {  
        m_position = m_position + m_orientation * 0.5f;  
        m_pointCible = m_position + m_orientation;  
    }  
  
    ....  
}
```

Le problème avec ces valeurs c'est qu'elles sont figées, on ne peut pas les moduler. Tous les jeux-vidéo n'utilisent pas forcément la même vitesse de déplacement et de plus, les valeurs que nous avons mises sont dignes des plus grands championnats de courses d'escargots !

En effet, pour un véritable jeu il faut multiplier ces vitesses par **10** voire plus et non **0.5**.

Pour gérer tout ça, nous allons implémenter deux attributs **m_sensibilite** et **m_vitesse** qui correspondront respectivement à la sensibilité de la souris (l'**orientation**) et à la vitesse de déplacement (la **position**) :

Code : C++

```
class Camera  
{  
public:  
    // Méthodes  
    ....  
  
private:  
    float m_phi;  
    float m_theta;  
    glm::vec3 m_orientation;  
  
    glm::vec3 m_axeVertical;  
    glm::vec3 m_deplacementLateral;  
  
    glm::vec3 m_position;  
    glm::vec3 m_pointCible;  
  
    float m_sensibilite;
```

```
float m_vitesse;  
};
```

Bien entendu, on modifie nos deux constructeurs pour qu'ils puissent prendre en paramètre des valeurs pour ces deux attributs.

Voici le constructeur par défaut :

Code : C++

```
Camera::Camera() : m_phi(0.0), m_theta(0.0), m_orientation(),  
m_axeVertical(0, 0, 1), m_deplacementLateral(), m_position(),  
m_pointCible(), m_sensibilite(0.0), m_vitesse(0.0)  
{  
  
}
```

Et voici le second constructeur :

Code : C++

```
Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3  
axeVertical, float sensibilite, float vitesse);
```

Code : C++

```
Camera::Camera(glm::vec3 position, glm::vec3 pointCible, glm::vec3  
axeVertical, float sensibilite, float vitesse) : m_phi(0.0),  
m_theta(0.0), m_orientation(),  
m_axeVertical(axeVertical), m_deplacementLateral(),  
m_position(position), m_pointCible(pointCible),  
m_sensibilite(sensibilite), m_vitesse(vitesse)  
{  
    // Actualisation du point cible  
    setPointcible(pointCible);  
}
```

Ensuite, on inclut l'attribut **m_sensibilite** dans la méthode **orienter()**:

Code : C++

```
void Camera::orienter(int xRel, int yRel)  
{  
    // Modification des angles  
  
    m_phi += -yRel * m_sensibilite;  
    m_theta += -xRel * m_sensibilite;  
  
    ....  
}
```

Et enfin, on ajoute l'attribut **m_vitesse** à chaque déplacement possible de la caméra :

Code : C++

```
void Camera::deplacer(Input const &input)
{
    // Gestion de l'orientation

    if(input.mouvementSouris())
        orienter(input.getXRel(), input.getYRel());

    // Avancée de la caméra

    if(input.getTouche(SDL_SCANCODE_UP))
    {
        m_position = m_position + m_orientation * m_vitesse;
        m_pointCible = m_position + m_orientation;
    }

    // Recul de la caméra

    if(input.getTouche(SDL_SCANCODE_DOWN))
    {
        m_position = m_position - m_orientation * m_vitesse;
        m_pointCible = m_position + m_orientation;
    }

    // Déplacement vers la gauche

    if(input.getTouche(SDL_SCANCODE_LEFT))
    {
        m_position = m_position + m_deplacementLateral * m_vitesse;
        m_pointCible = m_position + m_orientation;
    }

    // Déplacement vers la droite

    if(input.getTouche(SDL_SCANCODE_RIGHT))
    {
        m_position = m_position - m_deplacementLateral * m_vitesse;
        m_pointCible = m_position + m_orientation;
    }
}
```

Pensez à renseigner ces deux paramètres lorsque vous initialisez la caméra dans la classe **SceneOpenGL** :

Code : C++

```
// Caméra mobile

Camera camera(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0), 0.5,
0.5);
```

Petit conseil pour terminer, je vous conseille vivement de faire des getters et setters sur ces deux attributs. Ca pourrait être utile dans vos futurs développements :

Code : C++

```
// Getters et Setters

float getSensibilite() const;
float getVitesse() const;

void setSensibilite(float sensibilite);
void setVitesse(float vitesse);
```

Code : C++

```
float Camera::getSensibilite() const
{
    return m_vitesse;
}

float Camera::getVitesse() const
{
    return m_vitesse;
}

void Camera::setSensibilite(float sensibilite)
{
    m_sensibilite = sensibilite;
}

void Camera::setVitesse(float vitesse)
{
    m_vitesse = vitesse;
}
```

Télécharger : [Code Source C++ du chapitre 11](#)

Nous sommes enfin arrivés à la fin de ce chapitre, j'espère que vous êtes toujours en vie. 

Nous avons vu pas mal de notions mathématiques mais elles sont indispensables dans le développement 3D. Si ça peut vous rassurer, nous n'en reverrons pas beaucoup avant longtemps. Je ne vous ai pas encore parlé des quaternions ça sera un grand moment quand nous aborderons ce chapitre-la. 

Enfin, nous avons maintenant une caméra totalement opérationnelle et nous pouvons nous déplacer allégrement dans notre monde 3D. Que diriez-vous maintenant si nous faisions un gros TP récapitulatif ?

TP : Une relique retrouvée

Piouf, nous en avons fait du chemin depuis le début du tutoriel. Nous avons vu toutes les notions de base de la programmation OpenGL depuis l'affichage de triangles simples jusqu'à la gestion des caméras mobiles.

Pour conclure cette première partie, nous allons faire un gros TP récapitulatif qui reprendra tout ce que l'on a vu à travers les différents chapitres. Il y aura donc des matrices, des textures, la caméra, etc. Je vous donnerai toutes les indications nécessaires ainsi que quelques conseils pour que votre TP se déroule dans de bonnes conditions.

Si vous vous sentez prêt, vous pouvez continuer. 😊

Les consignes

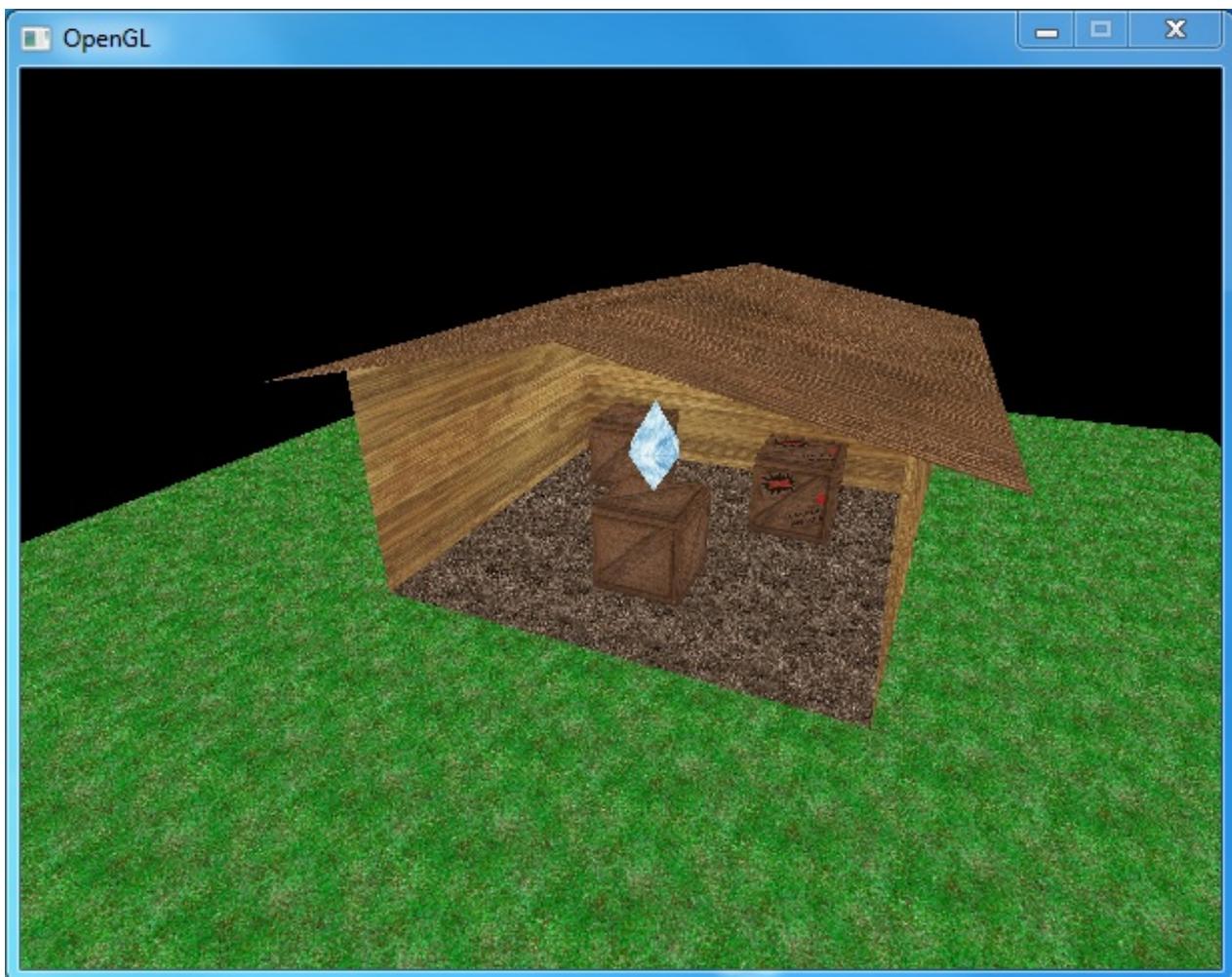
Les consignes

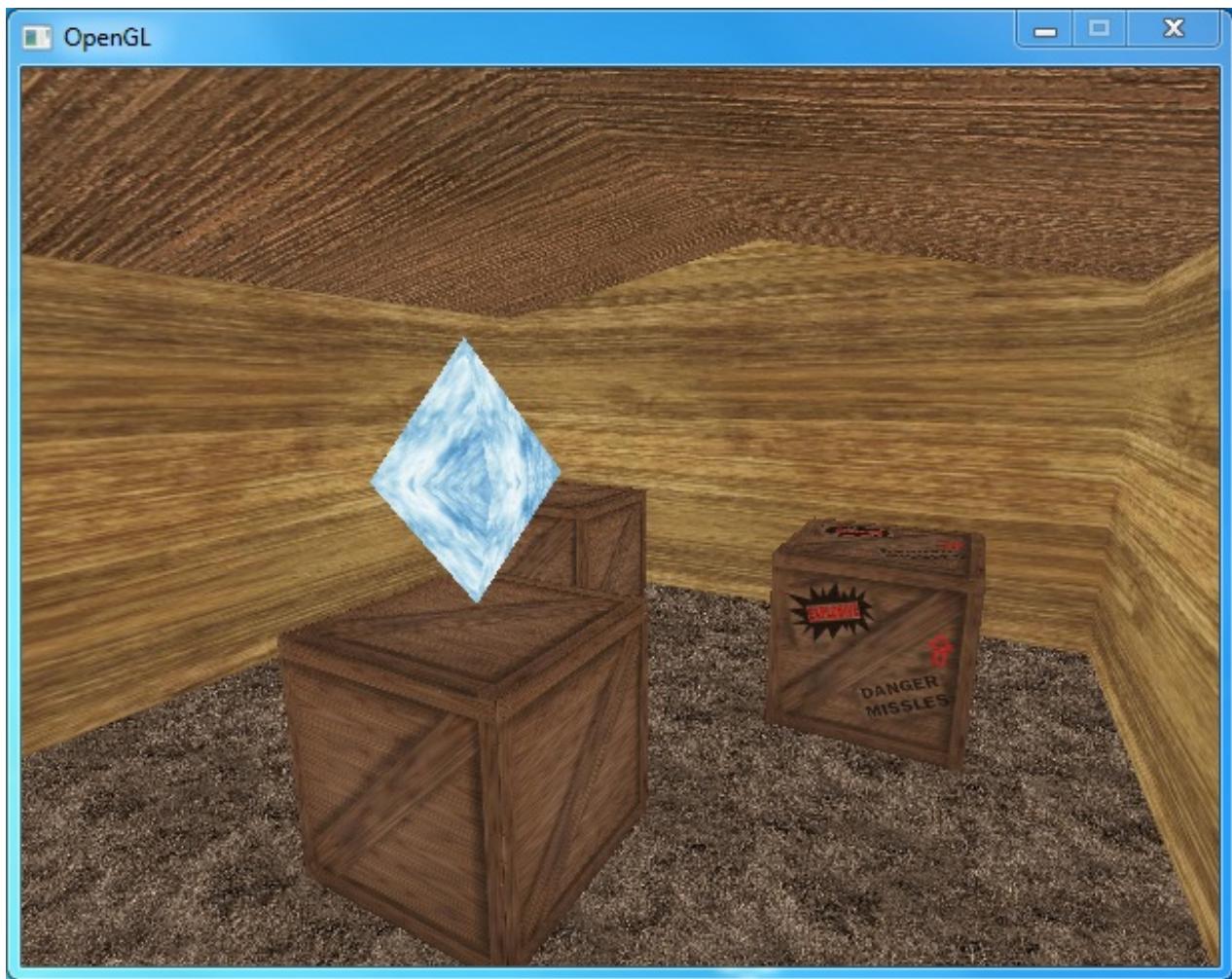
Objectif

Ce premier TP va vous permettre de mettre en pratique toutes les notions que nous avons abordées au cours de cette première partie. L'objectif est de faire une petite scène 3D dans laquelle vous pourrez vous balader à l'aide de votre caméra mobile.

Bien entendu, c'est un projet simple, ne vous attendez pas à avoir de la physique ou des effets avancés. Nous ne sommes qu'à la première partie, nous faisons juste un petit programme pour débuter. 😊

Je vais vous donner les consignes et des petits conseils pour bien démarrer ce TP. Mais avant tout, voyons ensemble à quoi devra ressembler votre rendu final :





Vous voyez ici la présence d'une cabane ainsi que de trois caisses à l'intérieur. Sur l'une d'elles se trouve d'ailleurs une petite relique en forme de cristal (*tancer musique de Tomb Raider ici*) qui tourne sur elle-même un peu à la manière du cube dans le chapitre sur la troisième dimension.

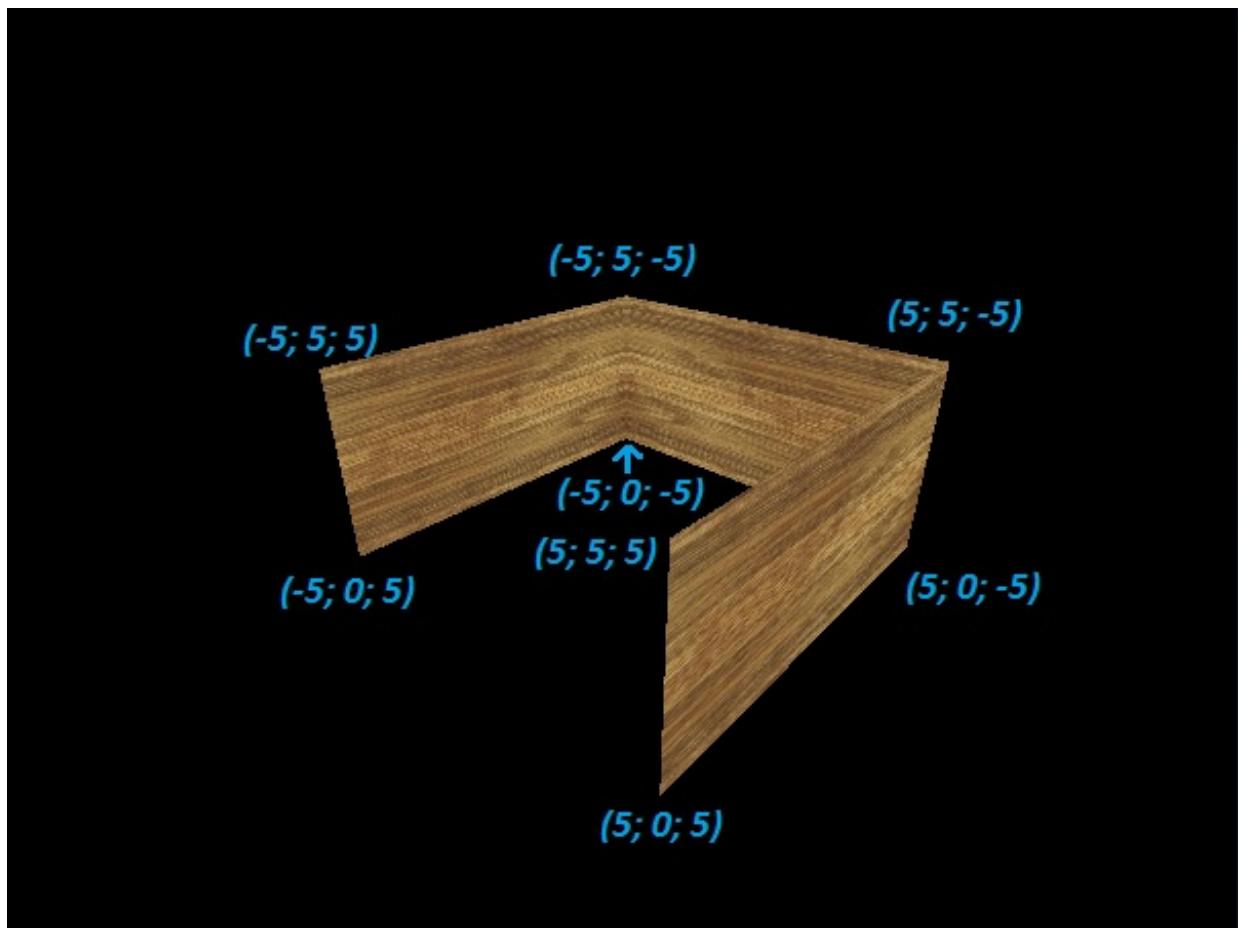
Vous devez donc recréer cette scène en incluant tous les éléments suivants :

- Une cabane placée au centre (0, 0, 0)
- Plusieurs caisses
- Un cristal
- Un sol herbeux entourant toute la scène
- Un sol terreux à l'intérieur de la cabane

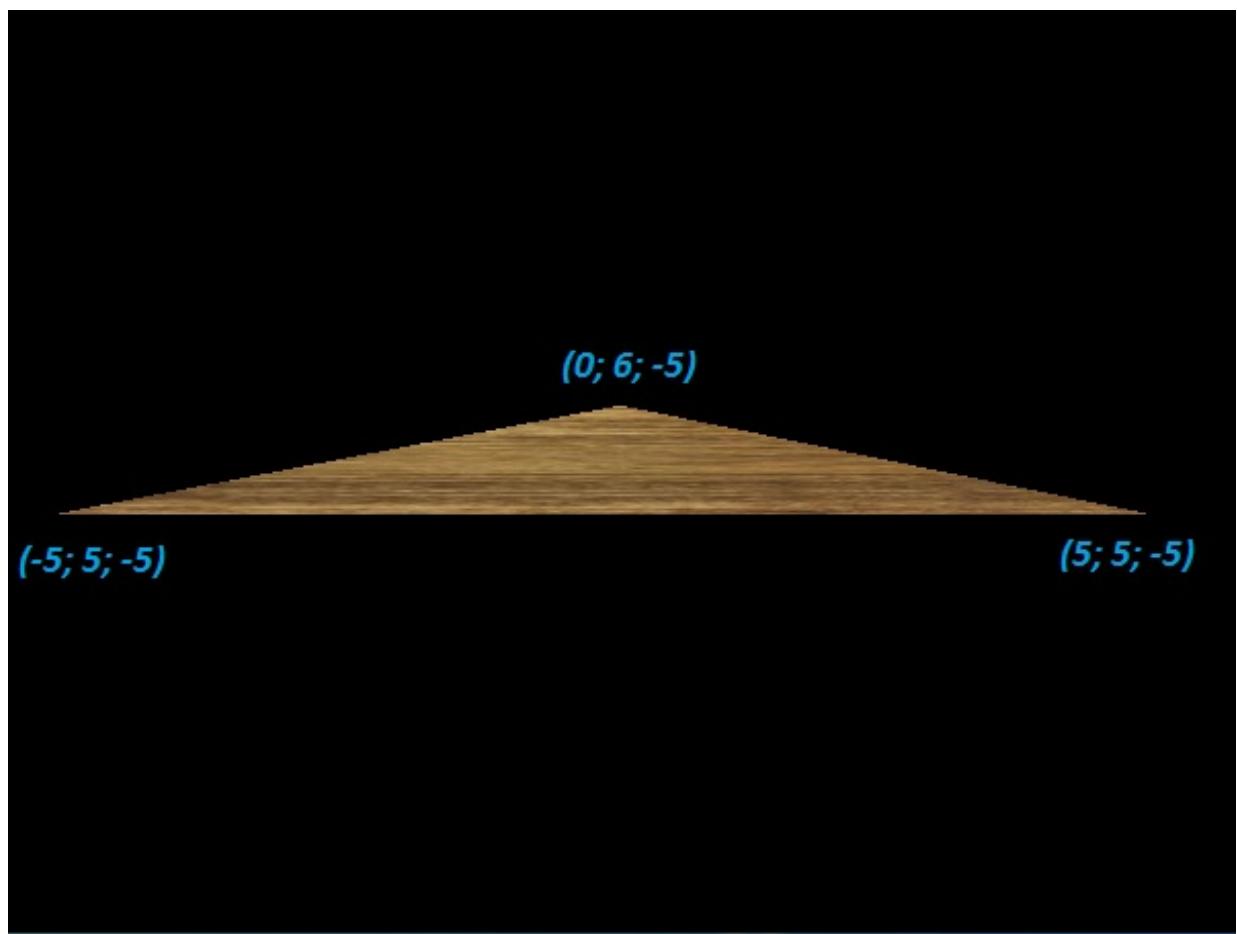
Les dimensions

Afin d'avoir un rendu similaire à celui des images précédentes, je vais vous donner toutes les dimensions nécessaires. Vous n'aurez pas à vous prendre la tête pour les définir vous-même. Je vous donnerai également la **répétition** de texture à utiliser. Au cas où vous n'auriez pas le gros pack d'images donné dans le chapitre **10**, je vous fournirai celles dont vous aurez besoin à la fin des explications.

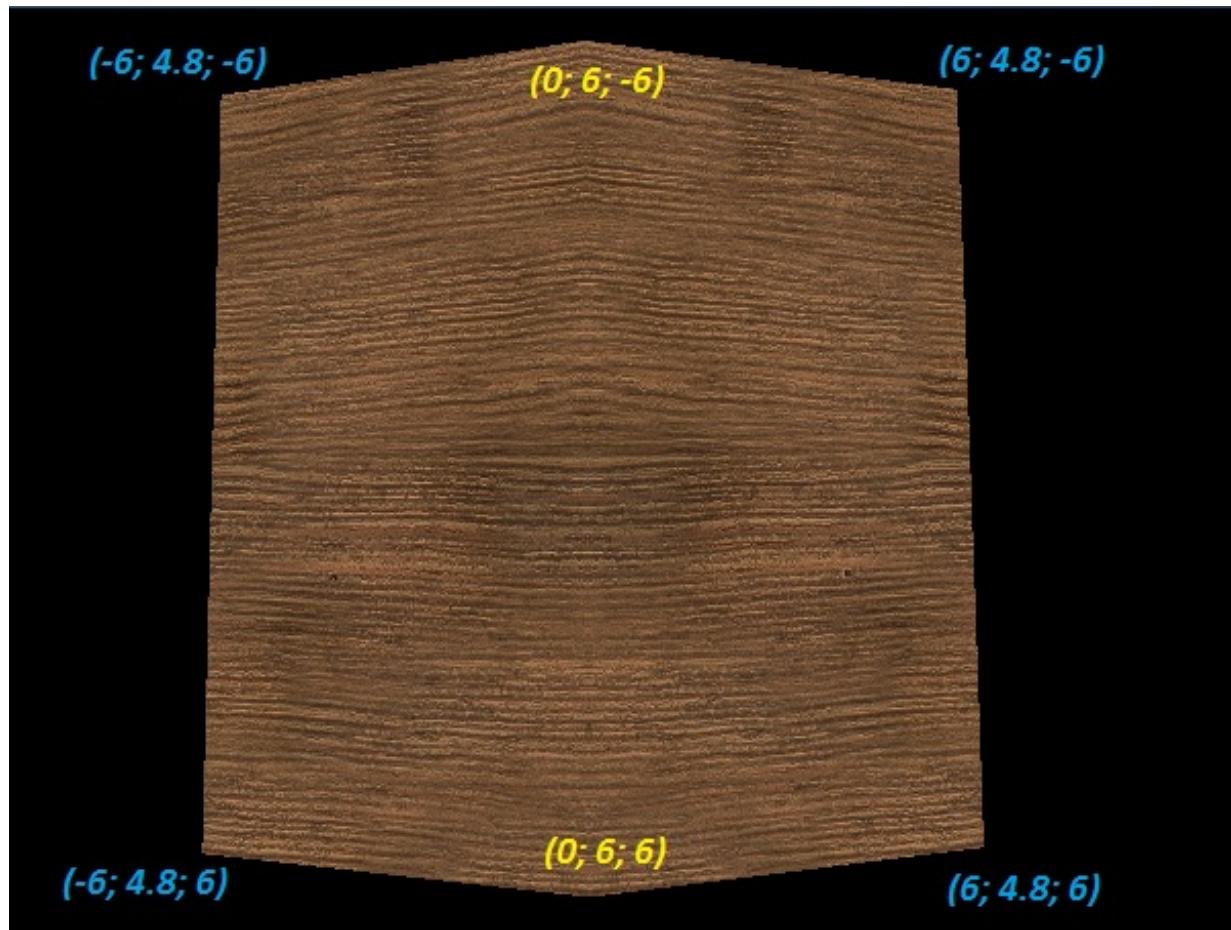
- On commence avec les données relatives à la cabane, et plus précisément à ses murs :



- Puis celles des combles (partie triangulaire située au fond de la cabane) :



- Et enfin, celles du toit (vu de haut) :

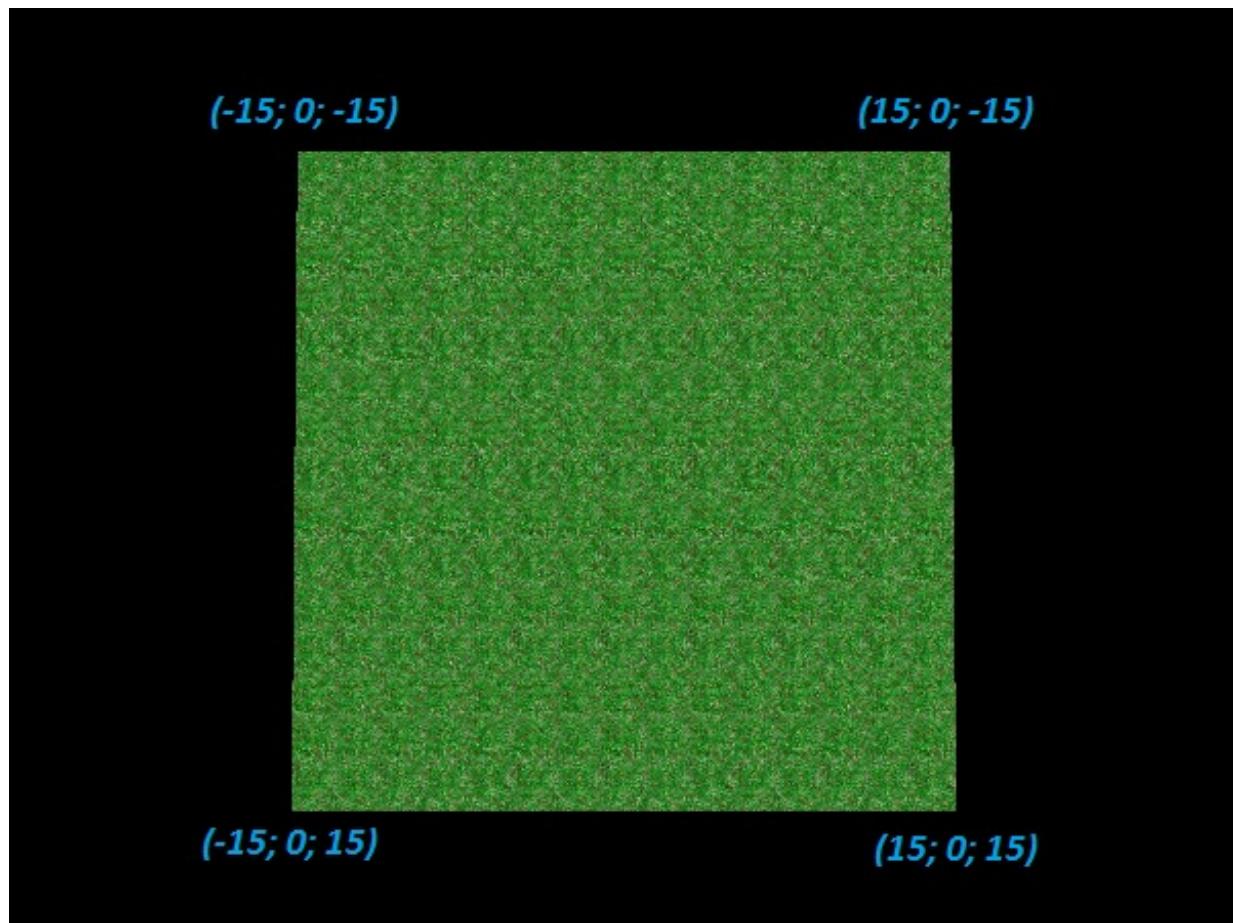


Les deux vertices jaunes sont de cette couleur uniquement pour mieux les voir sur la texture. Ils n'ont rien de spécial.

Aucune des parties de la cabane n'a besoin de la répétition de texture. Vos coordonnées ne doivent donc pas dépasser de l'intervalle [0; 1];

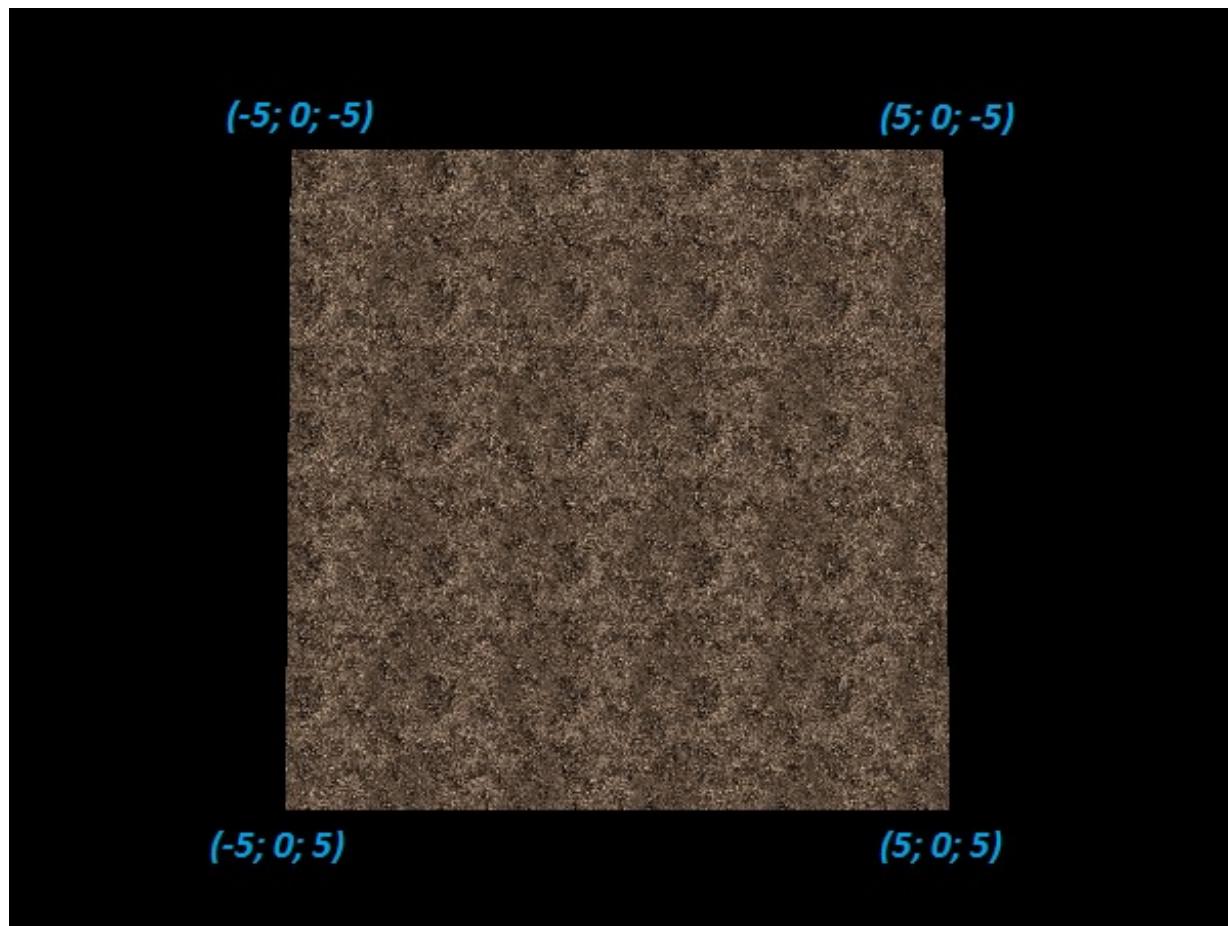
En ce qui concerne les dimensions du sol, on a :

- Le sol herbeux :



Cette fois, votre sol aura besoin de répéter sa texture. Vous devrez le faire **225** fois (**15** fois en longueur et en largeur). Ne vous inquiétez pas, l'image n'est chargée qu'une seule fois comme nous l'avons vu. Il n'y a donc rien de dangereux pour votre carte graphique. 😊

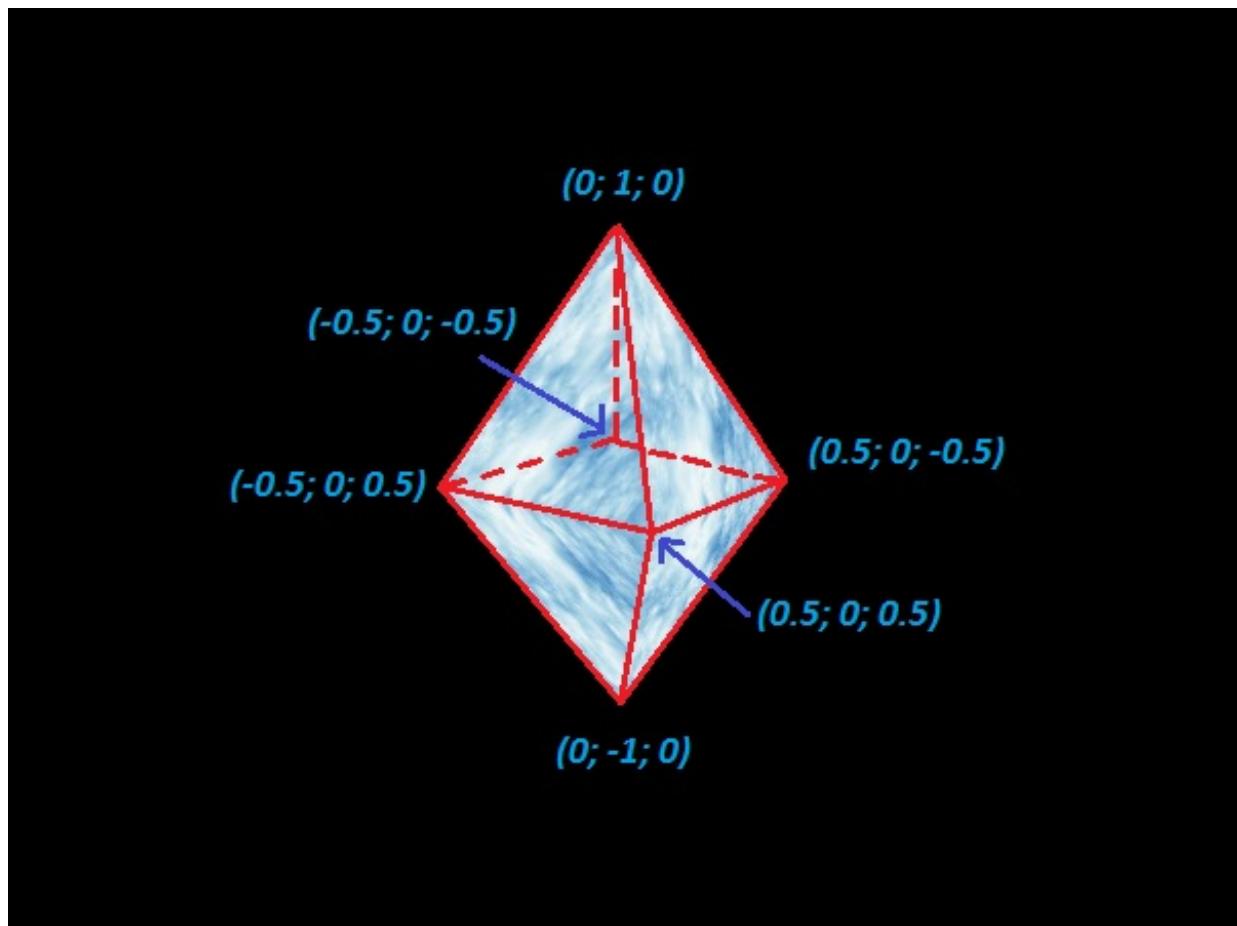
- Le sol terneux :



Même chose pour la texture, vous devrez la répéter. Faites le **25** fois ici (**5** fois en longueur et en largeur).

Au niveau des caisses, je pense que vous n'avez pas besoin de savoir grand chose. 🤪 Il en faut juste 3 avec un coté de 2.0 unités et les textures *Caisse.jpg* et *Caisse2.jpg*.

Enfin, en ce qui concerne le cristal, vous aurez besoin des dimensions suivantes :



Il s'agit d'une double pyramide à 4 cotés, il y a donc 8 triangles à créer.

Quelques conseils

Maintenant que nous avons vu ensemble les consignes, je vais vous donner quelques conseils pour vous aider dans votre travail. 😊

Les classes

Premièrement, au niveau même de la base de votre projet, je vous conseille de faire différentes classes pour gérer les modèles que vous devez afficher. C'est-à-dire qu'il vaut mieux faire une classe pour la cabane, une autre pour le cristal, etc. Prenez exemple sur le **Cube** que nous avons codé ensemble mais en intégrant en plus la gestion des textures. Ne vous compliquez pas la tâche en faisant de l'héritage cette fois. 😊

D'ailleurs, vous remarquerez que toutes vos classes contiendront les mêmes attributs : les vertices, les coordonnées de texture, un shader et une ou plusieurs textures. La méthode **afficher()** sera également identique, seuls les paramètres de la fonction **glDrawArrays()** vont varier.

La cabane

Au niveau de la cabane, je vous conseille de ne faire qu'une seule et unique classe. Celle-ci contiendrait toutes les données relatives aux murs, au toit et aux combles. Organisez votre tableau de vertices de cette façon pour avoir un code propre :

Code : C++

```

float verticesTmp[] = {1, 0, 1,    1, 0, 1,    1, 0, 1,      // Mur du
Fond
1, 0, 1,    1, 0, 1,    1, 0, 1,      // Mur du Fond
1, 0, 1,    1, 0, 1,    1, 0, 1,      // Mur Gauche
1, 0, 1,    1, 0, 1,    1, 0, 1,      // Mur Gauche
...
1, 0, 1,    1, 0, 1,    1, 0, 1,      // Toit Gauche
1, 0, 1,    1, 0, 1,    1, 0, 1};    // Toit Droit

```

Même chose pour le tableau de coordonnées de texture :

Code : C++

```

float coordTexture[] = {0, 0,      0, 0,      0, 0,      // Mur du Fond
0, 0,      0, 0,      0, 0,      // Mur du Fond
0, 0,      0, 0,      0, 0,      // Mur Gauche
0, 0,      0, 0,      0, 0,      // Mur Gauche
...
0, 0,      0, 0,      0, 0,      // Toit Gauche
0, 0,      0, 0,      0, 0};    // Toit Droit

```

Pour éviter d'avoir à chercher l'endroit où se situe votre erreur, testez vos vertices *triangle par triangle* au lieu de tout faire d'un coup. Croyez-moi, ça va vous faire gagner du temps.

Autre point important pour la cabane, il y a deux textures à gérer pour un seul tableau de vertices. Vous devrez jouer avec les paramètres de la fonction `glDrawArrays()` pour verrouiller vos textures au bon moment. Par exemple (et je dis bien *par exemple*), si vos **20 premiers** vertices concernent la texture du mur, alors vous devrez faire comme ceci :

Code : C++

```

// Verrouillage de la texture du mur
glBindTexture(GL_TEXTURE_2D, m_textureMur.getID());

// Affichage des murs (20 premiers vertices)
glDrawArrays(GL_TRIANGLES, 0, 20);

// Déverrouillage de la texture
glBindTexture(GL_TEXTURE_2D, 0);

```

Et si les **10 vertices suivants** concernent la texture du toit, alors vous devrez verrouiller la seconde texture et recommencer l'affichage en partant du **21ième** vertex :

Code : C++

```

// Verrouillage de la texture du toit
glBindTexture(GL_TEXTURE_2D, m_textureToit.getID());

```

```
// Affichage du toit (10 vertices suivants)  
glDrawArrays(GL_TRIANGLES, 20, 10);  
  
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);
```

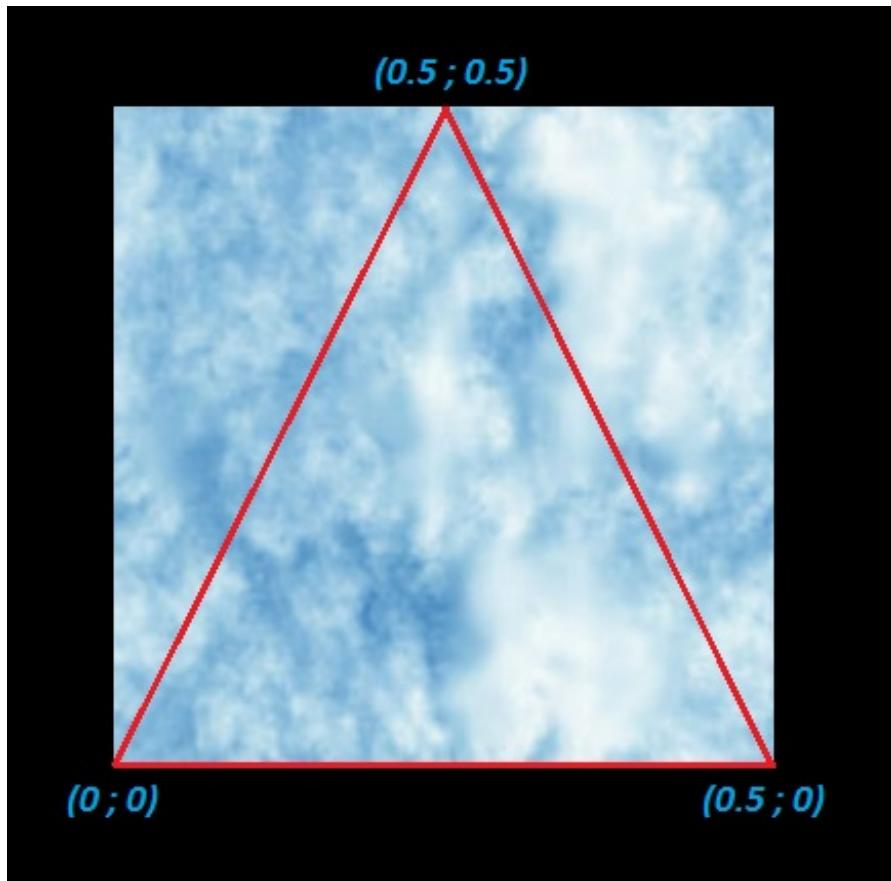
Les deux verrouillages doivent se passer dans la même méthode bien sûr.



Revoyez le chapitre 3 si vous ne comprenez pas trop pourquoi il faut mettre la valeur **20** et non **21** au deuxième paramètre pour commencer l'affichage à partir du **21^{ième}** vertex. 😊

Le cristal

Le cristal ne devrait pas trop vous poser de problème, il suffit de reprendre le même principe que le cube en y ajoutant la gestion de la texture. Faites juste attention à spécifier les coordonnées de texture de cette façon pour chaque triangle :



A part ça, n'oubliez pas de *l'animer* en le faisant tourner sur elle-même. Essayez de faire ça doucement en incrémentant votre angle de **1** degré à chaque tour de boucle au lieu de **4**. 😊

Les sols

Au niveau des deux types de sol, vous pouvez parfaitement faire deux classes pour les différencier, c'est plus facile et c'est efficace. Cependant, si vous pensez que vous pouvez le faire, je vous suggère de ne faire qu'une seule et unique classe. Vous pourrez l'instancier deux fois en modifiant juste la texture et sa répétition. Un constructeur que vous pouvez utiliser (sans être obligés) est le suivant :

Code : C++

```
Sol(float longueur, float largeur, int repitionLongueur, int  
repitionLargeur, std::string vertexShader, std::string  
fragmentShader, std::string texture);
```

Détail 1 : La terre doit évidemment se trouver au niveau de la cabane et pas autre part.

Détail 2 : Pour éviter de vous retrouvez avec un bug de texture, faites translater le sol herbeux de **0.01** sur l'axe **Y** :

Code : C++

```
// Sauvegarde de la matrice  
  
mat4 sauvegardeModelview = modelview;  
  
// Affichage du sol herbeux  
  
modelview = translate(modelview, vec3(0, 0.01, 0));  
solHerbeux.afficher(projection, modelview);  
  
// Restauration de la matrice  
  
modelview = sauvegardeModelview;
```

Vous pouvez tenter de faire sans cette translation si vous le souhaitez pour voir ce que ça donne.

A vos classes !

Bon, je pense vous en avoir assez dit, je ne vais pas vous donner tout le code non plus. 😊 Vous avez à disposition toutes les données dont vous avez besoin ainsi que quelques conseils pour commencer.

C'est à vous de jouer maintenant !

Télécharger : [Pack de textures à utiliser pour le TP](#)

Correction

Stop, arrêtez là ! Rendez-moi vos codes. 🤪

Bon, cette phrase ne sert pas à grand chose mais ça veut au moins dire que nous pouvons passer à la correction. Je vous propose une des solutions possibles, il y avait plusieurs façons d'aborder ce TP donc ne vous inquiétez pas si vous avez fait différemment.

Nous allons nous mettre sans plus tarder dans le bain en commençant par le code de base de la boucle principale.

La boucle principale

La méthode **bouclePrincipale()** est celle qui va nous permettre de mettre en scène tous nos objets. Elle devra donc contenir la cabane, les caisses, ... Mais pour le moment, il y a juste les matrices, la caméra, le nettoyage et l'actualisation de la fenêtre :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Matrices

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Caméra mobile

    Camera camera(vec3(3, 4, 10), vec3(0, 2.1, 2), vec3(0, 1, 0),
0.5, 0.5);
    m_input.afficherPointeur(false);
    m_input.capturerPointeur(true);

    // Boucle principale

    while(!m_input.terminer())
    {
        // On définit le temps de début de boucle

        debutBoucle = SDL_GetTicks();

        // Gestion des évènements

        m_input.updateEvenements();

        if(m_input.getTouche(SDL_SCANCODE_ESCAPE))
            break;

        camera.deplacer(m_input);

        // Nettoyage de l'écran

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Gestion de la caméra

        camera.lookAt(modelview);

        // Rendu

        ....
```

```
// Actualisation de la fenêtre
SDL_GL_SwapWindow(m_fenetre);

// Calcul du temps écoulé
finBoucle = SDL_GetTicks();
tempsEcoule = finBoucle - debutBoucle;

// Si nécessaire, on met en pause le programme
if(tempsEcoule < frameRate)
    SDL_Delay(frameRate - tempsEcoule);
}
```

Les objets à afficher seront ajoutés après.

La cabane

Le header

On passe maintenant aux différents modèles qui doivent être placés dans notre scène, on commence par la cabane.

La première chose avec elle est évidemment de créer une classe **Cabane**. Elle doit contenir tous les attributs nécessaires à l'affichage comme le tableau de vertices, celui des coordonnées de texture, le shader et les textures (une pour le toit et l'autre pour les murs) :

Code : C++

```
#ifndef DEF_CABANE
#define DEF_CABANE

// Includes OpenGL
#ifdef WIN32
#include <GL/glew.h>
#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>
#endif

// Includes GLM
#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Autres includes
#include "Shader.h"
#include "Texture.h"
```

```
// Classe Cabane

class Cabane
{
    public:

        Cabane(std::string const vertexShader, std::string const
fragmentShader);
        ~Cabane();

        void afficher(glm::mat4 &projection, glm::mat4 &modelview);

    private:

        Shader m_shader;
        Texture m_textureMur;
        Texture m_textureToit;

        float m_vertices[99];
        float m_coordTexture[66];
};

#endif
```

Remarquez la méthode **afficher()** qui est la même que celle du *Cube*.

Mis à part le fait qu'il faille deux textures ici, cette architecture de classe est la même pour tous les objets présents dans ce TP.



Le constructeur

Le constructeur est la *méthode* qui vous a probablement posés le plus de problèmes. Celui-ci doit initialiser tous les attributs et les premiers sont le shader et les deux textures :

Code : C++

```
Cabane::Cabane(std::string const vertexShader, std::string const
fragmentShader) : m_shader(vertexShader, fragmentShader),
{
    m_textureMur("Textures/Mur.jpg"), m_textureToit("Textures/Toit.jpg")
    // Chargement du shader
    m_shader.charger();

    // Chargement des textures
    m_textureMur.charger();
    m_textureToit.charger();
}
```

L'attribut le plus compliqué à initialiser était le tableau de vertices, il fallait créer tous les triangles nécessaires à l'aide des schémas fournis dans l'énoncé. Je vous avais donné quelques conseils pour y arriver comme le fait de vérifier votre affichage triangle par triangle par exemple.

La chose à ne pas faire était de créer une classe pour chaque partie de la cabane (Murs, Toit et Combles). C'est plus compliqué de gérer toutes ces parties indépendamment plutôt que tout réunir dans un seul tableau.

En prenant votre temps, vous avez pu trouver le tableau de vertices suivant :

Code : C++

```
// Vertices temporaires

float verticesTmp[] = {-5, 0, -5,      5, 0, -5,      5, 5, -5,      //
Mur du Fond
                  -5, 0, -5,      -5, 5, -5,      5, 5, -5,      //
Mur du Fond
                  -5, 0, -5,      -5, 0, 5,      -5, 5, 5,      //
Mur Gauche
                  -5, 0, -5,      -5, 5, -5,      -5, 5, 5,      //
Mur Gauche
                  5, 0, -5,      5, 0, 5,      5, 5, 5,      //
Mur Droit
                  5, 0, -5,      5, 5, -5,      5, 5, 5,      //
Mur Droit
                  -5, 5, -5,      5, 5, -5,      0, 6, -5,      //
Combles
                  -6, 4.8, -6,      -6, 4.8, 6,      0, 6, 6,      //
Toit Gauche
                  -6, 4.8, -6,      0, 6, -6,      0, 6, 6,      //
Toit Gauche
                  6, 4.8, -6,      6, 4.8, 6,      0, 6, 6,      //
Toit Droit
                  6, 4.8, -6,      0, 6, -6,      0, 6, 6};      //
```

J'ai aéré le code pour que l'on puisse bien distinguer les parties de la cabane. Je vous recommande de faire ça pour vos modèles.



Au niveau du tableau de coordonnées de texture, il faut juste faire correspondre les triangles aux bonnes coordonnées de la même façon que nous le faisions dans les chapitres précédents :

Code : C++

```
// Coordonnées de texture temporaires

float coordTexture[] = {0, 0,    1, 0,    1, 1,      // Mur du Fond
                        0, 0,    0, 1,    1, 1,      // Mur du Fond
                        0, 0,    1, 0,    1, 1,      // Mur Gauche
                        0, 0,    0, 1,    1, 1,      // Mur Gauche
                        0, 0,    1, 0,    1, 1,      // Mur Droit
                        0, 0,    0, 1,    1, 1,      // Mur Droit
                        0, 0,    1, 0,    0.5, 0.5, // Combles
                        0, 0,    1, 0,    1, 1,      // Toit Gauche
                        0, 0,    0, 1,    1, 1,      // Toit Gauche
                        0, 0,    1, 0,    1, 1,      // Toit Droit
                        0, 0,    0, 1,    1, 1};     // Toit Droit
```

Attention cependant aux combles qui ne sont représentés que par un seul triangle.

Une fois les tableaux définis, il ne manque plus qu'à copier leurs valeurs dans les attributs finaux :

Code : C++

```
// Copie des vertices  
  
for(int i(0); i < 99; i++)  
    m_vertices[i] = verticesTmp[i];  
  
// Copie des coordonnées  
  
for(int i(0); i < 66; i++)  
    m_coordTexture[i] = coordTexture[i];
```

Si on résume :

Code : C++

```
Cabane::Cabane(std::string const vertexShader, std::string const  
fragmentShader) : m_shader(vertexShader, fragmentShader),  
m_textureMur("Textures/Mur.jpg"), m_textureToit("Textures/Toit.jpg")  
{  
    // Chargement du shader  
  
    m_shader.charger();  
  
    // Chargement des textures  
  
    m_textureMur.charger();  
    m_textureToit.charger();  
  
    // Vertices temporaires  
  
    float verticesTmp[] = {-5, 0, -5,      5, 0, -5,      5, 5, -5,  
    // Mur du Fond  
                    -5, 0, -5,      -5, 5, -5,      5, 5, -5,  
    // Mur du Fond  
                    -5, 0, -5,      -5, 0, 5,      -5, 5, 5,  
    // Mur Gauche  
                    -5, 0, -5,      -5, 5, -5,      -5, 5, 5,  
    // Mur Gauche  
                    5, 0, -5,      5, 0, 5,      5, 5, 5,  
    // Mur Droit  
                    5, 0, -5,      5, 5, -5,      5, 5, 5,  
    // Mur Droit  
                    -5, 5, -5,      5, 5, -5,      0, 6, -5,  
    // Combles  
                    -6, 4.8, -6,     -6, 4.8, 6,      0, 6, 6,  
    // Toit Gauche  
                    -6, 4.8, -6,     0, 6, -6,      0, 6, 6,  
    // Toit Gauche  
                    6, 4.8, -6,      6, 4.8, 6,      0, 6, 6,  
    // Toit Droit  
                    6, 4.8, -6,     0, 6, -6,      0, 6, 6};  
    // Toit Droit
```

```

    // Coordonnées de texture temporaires

    float coordTexture[] = {0, 0, 1, 0, 1, 1,           // Mur du
Fond
                           0, 0, 0, 1, 1, 1,           // Mur du
Fond

                           0, 0, 1, 0, 1, 1,           // Mur
Gauche
                           0, 0, 0, 1, 1, 1,           // Mur
Gauche

                           0, 0, 1, 0, 1, 1,           // Mur
Droit
                           0, 0, 0, 1, 1, 1,           // Mur
Droit

                           0, 0, 1, 0, 0.5, 0.5,      // Combles
Gauche
                           0, 0, 1, 0, 1, 1,           // Toit
Gauche
                           0, 0, 0, 1, 1, 1,           // Toit
Droit
                           0, 0, 0, 1, 1, 1};          // Toit

    // Copie des vertices

    for(int i(0); i < 99; i++)
        m_vertices[i] = verticesTmp[i];

    // Copie des coordonnées

    for(int i(0); i < 66; i++)
        m_coordTexture[i] = coordTexture[i];
}

```

La méthode afficher()

En ce qui concerne la méthode **afficher()**, on peut reprendre celle de la classe **Caisse** mais en y faisant quelques modifications. En effet, il faut gérer l'affichage de deux textures ici et donc faire attention à savoir quels vertices correspondent à quelle texture.

D'après le tableau **m_vertices**, les **21** premiers vertices, représentés par les **63** premières cases, correspondent aux murs et aux combles qui utilisent la texture **Mur.jpg**. Il faut donc appeler la fonction **glDrawArrays()** pour afficher les **21** vertices en partant de celui d'indice **0** (le premier) :

Code : C++

```

// Verrouillage de la texture du Mur

glBindTexture(GL_TEXTURE_2D, m_textureMur.getID());

// Rendu

glDrawArrays(GL_TRIANGLES, 0, 21);

```

```
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);
```

Bien entendu, on n'oublie pas de verrouiller la texture du mur. 😊

Ensuite, il faut s'occuper du toit qui occupe les **12** derniers vertices, représentés par les **36** dernières cases du tableau. On réappelle donc la fonction **glDrawArrays()** pour afficher les **12** vertices en partant de celui d'indice **21** (le **22^{ième}**) :

Code : C++

```
// Verrouillage de la texture du Toit  
glBindTexture(GL_TEXTURE_2D, m_textureToit.getID());  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 21, 12);  
  
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);
```

N'oubliez pas de verrouiller l'autre texture maintenant.

Ce qui donne la méthode final :

Code : C++

```
void Cabane::afficher(glm::mat4 &projection, glm::mat4 &modelview)  
{  
    // Activation du shader  
    glUseProgram(m_shader.getProgramID());  
  
    // Envoi des vertices  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,  
    m_vertices);  
    glEnableVertexAttribArray(0);  
  
    // Envoi des coordonnées de texture  
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,  
    m_coordTexture);  
    glEnableVertexAttribArray(2);  
  
    // Envoi des matrices  
  
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),  
    "projection"), 1, GL_FALSE, value_ptr(projection));  
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),  
    "modelview"), 1, GL_FALSE, value_ptr(modelview));
```

```
// Verrouillage de la texture du Mur  
glBindTexture(GL_TEXTURE_2D, m_textureMur.getID());  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 0, 21);  
  
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);  
  
// Verrouillage de la texture du Toit  
glBindTexture(GL_TEXTURE_2D, m_textureToit.getID());  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 21, 12);  
  
// Déverrouillage de la texture  
glBindTexture(GL_TEXTURE_2D, 0);  
  
// Désactivation des tableaux  
glDisableVertexAttribArray(2);  
glDisableVertexAttribArray(0);  
  
// Désactivation du shader  
glUseProgram(0);  
}
```

Le sol

Le header

La correction du sol va être un peu spécial. En fait, tout dépend de la manière dont vous l'avez codé. Je vais considérer le fait que vous n'ayez fait qu'une seule classe pour gérer les deux types de sol (herbeux et terreux). Bien entendu, si vous en avez fait deux ça fonctionne quand même, il n'y a pas qu'une seule solution je le répète. 😊

Commençons par le header qui ressemble quasiment trait pour trait à celui de la classe **Cabane**

Code : C++

```
#ifndef DEF_SOL  
#define DEF_SOL  
  
// Includes OpenGL  
  
#ifdef WIN32
```

```
#include <GL/glew.h>

#ifndef GL3_PROTOTYPES
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

// Includes GLM

#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Autres includes

#include "Shader.h"
#include "Texture.h"

// Classe Sol

class Sol
{
public:

    Sol(float longueur, float largeur, int repetitionLongueur, int
repetitionLargeur, std::string const vertexShader, std::string const
fragmentShader, std::string const texture);
    ~Sol();

    void afficher(glm::mat4 &projection, glm::mat4 &modelview);

private:

    Shader m_shader;
    Texture m_texture;

    float m_vertices[18];
    float m_coordTexture[12];
};

#endif
```

La seule différence par rapport au précédent est le fait que nous n'avons besoin qu'une d'une seule texture ici.

Le constructeur

Si vous n'avez utilisé qu'une seule classe pour le sol, vous vous êtes peut-être confrontés au problème de la répétition de texture. Je vous avais donné une petite piste avec le constructeur suivant :

Code : C++

```
Sol(float longueur, float largeur, int repetitionLongueur, int
repetitionLargeur, std::string const vertexShader, std::string const
fragmentShader, std::string const texture);
```

Celui-ci prend 7 paramètres :

- **longueur** et **largeur** : pour définir la taille du sol
- **repetitionLongueur** et **repetitionLargeur** : pour définir la répétition de la texture
- **vertexShader** et **fragmentShader** : qui représentent les fichiers sources du shader
- **texture** : qui représente le chemin vers l'image à utiliser

Je passe très rapidement sur le début du constructeur qui ne fait qu'initialiser le shader et la texture :

Code : C++

```
Sol::Sol(float longueur, float largeur, int repetitionLongueur, int
repetitionLargeur, std::string const vertexShader, std::string const
fragmentShader, std::string const texture) :
    m_shader(vertexShader, fragmentShader), m_texture(texture)
{
    // Chargement du shader
    m_shader.charger();

    // Chargement de la texture
    m_texture.charger();
}
```

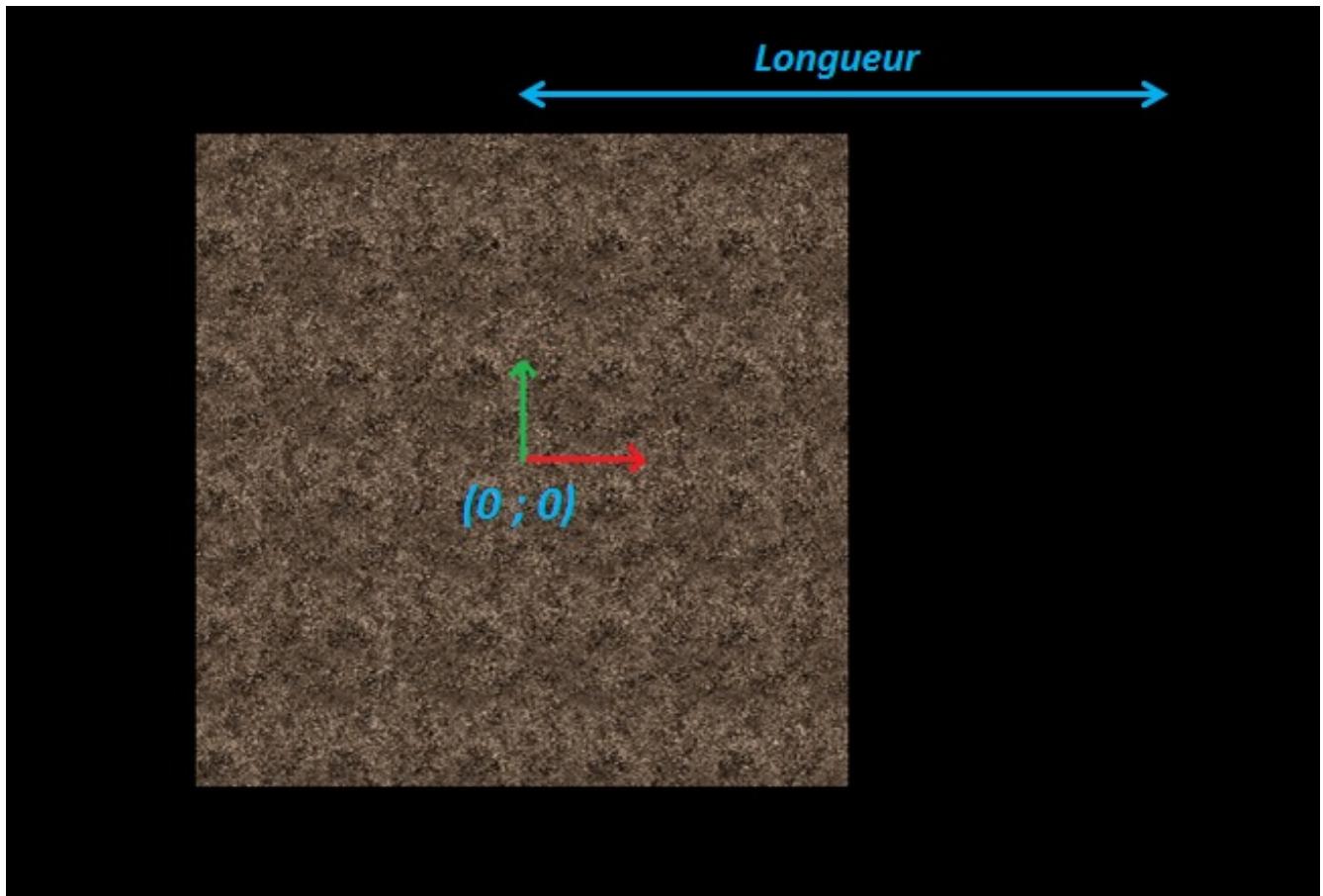
Les deux premiers paramètres de la liste ci-dessus servent à initialiser le tableau de vertices en créant deux triangles. Ces derniers formeront eux-mêmes un rectangle qui représentera le sol :

Code : C++

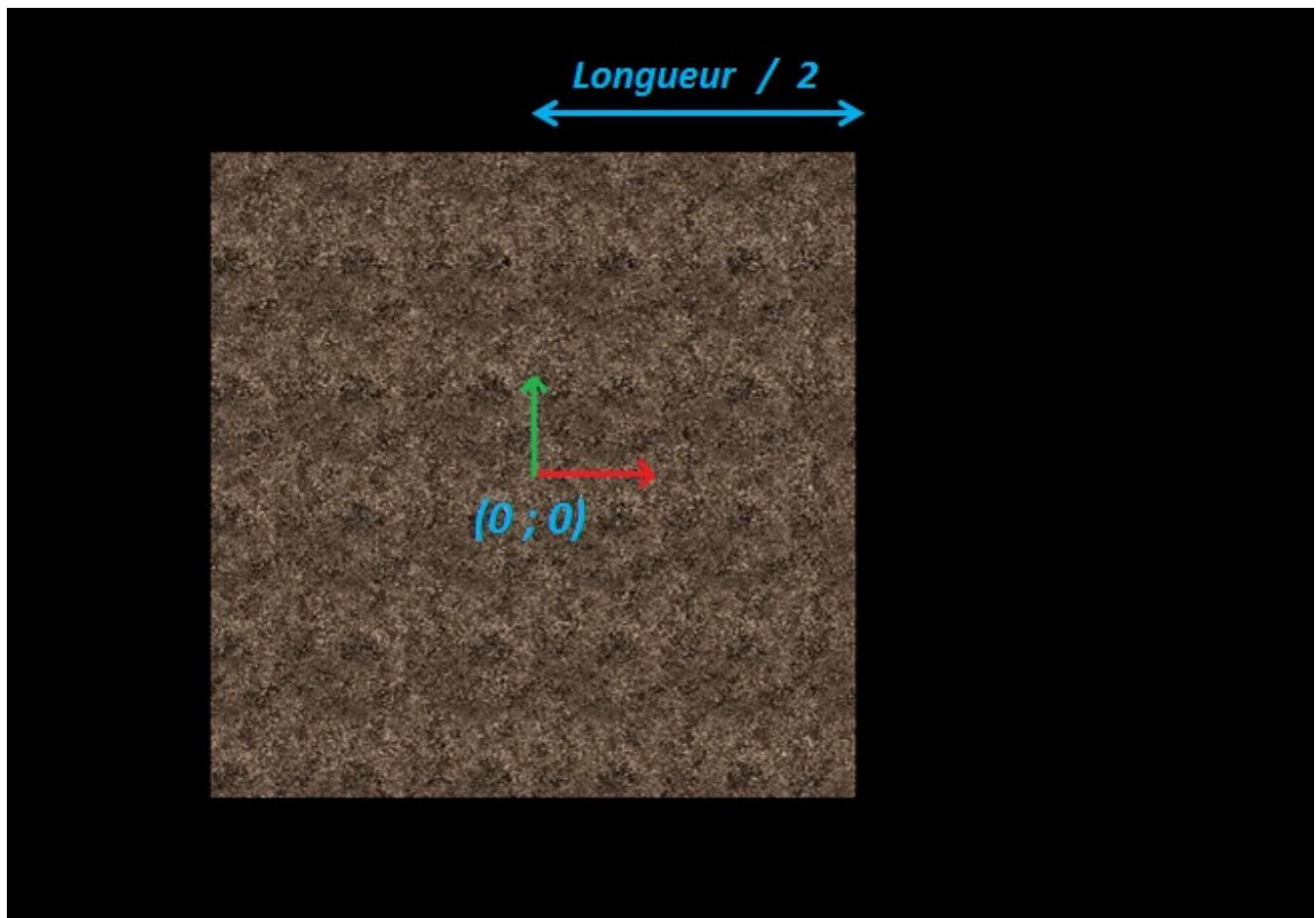
```
// Vertices temporaires

float verticesTmp[] = {-longueur, 0, -largeur, longueur, 0,
-lARGEUR, longueur, 0, largeur,      // Triangle 1
                     -longueur, 0, -largeur, -longueur, 0,
largeur,   longueur, 0, largeur};    // Triangle 2
```

Le petit piège ici serait de ne pas toucher aux paramètres **longueur** et **largeur** avant d'initialiser les vertices. En effet, si on ne le fait pas, le sol sera doublé :



Pour éviter cela, il faut diviser par **2** les paramètres **longueur** et **largeur** avant de les utiliser dans le tableau :

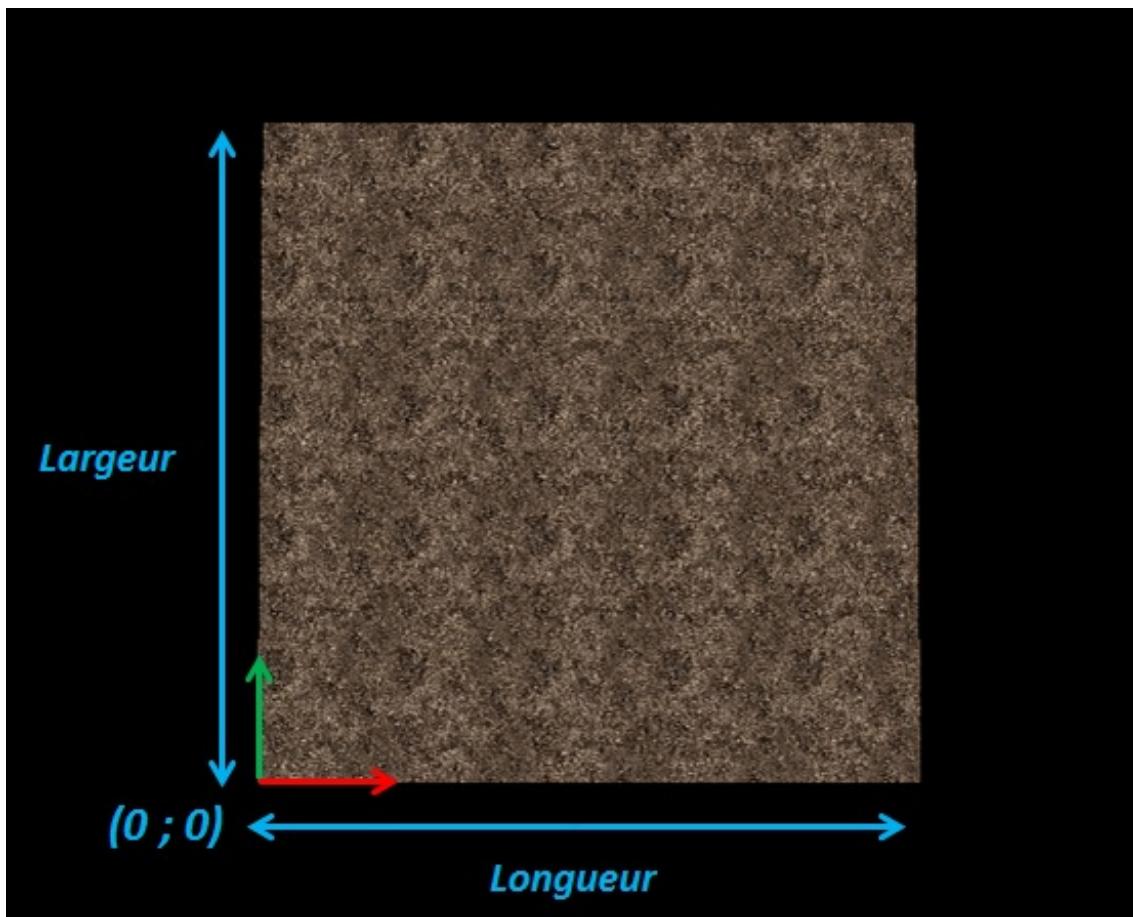


Nous avons fait la même chose dans la classe **Cube** pour éviter de doubler la taille de nos modèles.

Code : C++

```
// Division de la taille  
  
longueur /= 2.0;  
largeur /= 2.0;  
  
// Vertices temporaires  
  
float verticesTmp[] = {-longueur, 0, -largeur, longueur, 0,  
-largeur, longueur, 0, largeur, // Triangle 1  
-longueur, 0, -largeur, -longueur, 0,  
largeur, longueur, 0, largeur}; // Triangle 2
```

A noter que si vos vertices partent d'un coin, comme le coin inférieur gauche par exemple, alors vous ne devez pas faire ces divisions :



En ce qui concerne le tableau de coordonnées de texture, il suffit de reprendre celui que nous avons utilisé pour les carrés dans les chapitres précédents et d'y remplacer les valeurs 1 par les paramètres **repetitionLongueur** et **repetitionLargeur** :

Code : C++

```
// Coordonnées de texture temporaires

float coordTexture[] = {0, 0, repetitionLongueur, 0,
repetitionLongueur, repetitionLargeur,           // Triangle 1
                        0, 0, 0, repetitionLargeur,
repetitionLongueur, repetitionLargeur};           // Triangle 2
```

Une fois les tableaux définis, on copie leurs valeurs dans les attributs :

Code : C++

```
// Copie des vertices

for(int i(0); i < 18; i++)
    m_vertices[i] = verticesTmp[i];

// Copie des coordonnées

for(int i(0); i < 12; i++)
    m_coordTexture[i] = coordTexture[i];
```

Ce qui donne :

Code : C++

```

Sol::Sol(float longueur, float largeur, int repetitionLongueur, int
repetitionLargeur, std::string const vertexShader, std::string const
fragmentShader, std::string const texture) :
    m_shader(vertexShader, fragmentShader), m_texture(texture)
{
    // Chargement du shader
    m_shader.charger();

    // Chargement de la texture
    m_texture.charger();

    // Division de la taille
    longueur /= 2.0;
    largeur /= 2.0;

    // Vertices temporaires
    float verticesTmp[] = {-longueur, 0, -largeur, longueur, 0,
                           -largeur, longueur, 0, largeur,           // Triangle 1
                           -longueur, 0, -largeur, -longueur, 0,
                           largeur,   longueur, 0, largeur};      // Triangle 2

    // Coordonnées de texture temporaires
    float coordTexture[] = {0, 0, repetitionLongueur, 0,
                           repetitionLongueur, repetitionLargeur,           // Triangle 1
                           0, 0, 0, repetitionLargeur,
                           repetitionLongueur, repetitionLargeur};        // Triangle 2

    // Copie des vertices
    for(int i(0); i < 18; i++)
        m_vertices[i] = verticesTmp[i];

    // Copie des coordonnées
    for(int i(0); i < 12; i++)
        m_coordTexture[i] = coordTexture[i];
}

```

La méthode afficher()

Pour l'affichage, il suffit juste de reprendre la méthode **afficher()** de la classe **Caisse**, et non **Cube** cette fois-ci, et d'y modifier le nombre de vertices à afficher. Il y a **2** triangles dans notre cas, il en faut donc **6** :

Code : C++

```

void Sol::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

```

```
// Envoi des vertices  
  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,  
    m_vertices);  
    glEnableVertexAttribArray(0);  
  
    // Envoi des coordonnées de texture  
  
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,  
    m_coordTexture);  
    glEnableVertexAttribArray(2);  
  
    // Envoi des matrices  
  
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),  
    "projection"), 1, GL_FALSE, value_ptr(projection));  
  
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),  
    "modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
    // Verrouillage de la texture  
  
    glBindTexture(GL_TEXTURE_2D, m_texture.getID());  
  
    // Rendu  
  
    glDrawArrays(GL_TRIANGLES, 0, 6);  
  
    // Déverrouillage de la texture  
  
    glBindTexture(GL_TEXTURE_2D, 0);  
  
    // Désactivation des tableaux  
  
    glDisableVertexAttribArray(2);  
    glDisableVertexAttribArray(0);  
  
    // Désactivation du shader  
  
    glUseProgram(0);  
}
```

Le cristal

Le header

Le cristal est le modèle le plus simple du TP. Il suffit de reprendre le même principe que la classe **Cube** en ajoutant la gestion de texture. Voici le header :

Code : C++

```
#ifndef DEF_CRISTAL
#define DEF_CRISTAL


// Includes OpenGL

#ifdef WIN32
#include <GL/glew.h>

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif


// Includes GLM

#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>


// Autres includes

#include "Shader.h"
#include "Texture.h"


// Classe Cristal

class Cristal
{
public:

    Cristal(std::string const vertexShader, std::string const
fragmentShader, std::string const texture);
    Cristal();

    void afficher(glm::mat4 &projection, glm::mat4 &modelview);

private:

    Shader m_shader;
    Texture m_texture;

    float m_vertices[72];
    float m_coordTexture[48];
};

#endif
```

Le constructeur

Le tableau de vertices :

Code : C++

```
// Vertices temporaires

float verticesTmp[] = {-0.5, 0, -0.5, 0.5, 0, -0.5, 0, 1, 0,
// Triangle 1
0.5, 0, -0.5, 0.5, 0, 0.5, 0, 1, 0,
```

```
// Triangle 2           0.5, 0, 0.5, -0.5, 0, 0.5, 0, 1, 0,
// Triangle 3           -0.5, 0, 0.5, -0.5, 0, -0.5, 0, 1, 0,
// Triangle 4           -0.5, 0, -0.5, 0.5, 0, -0.5, 0, -1, 0,
// Triangle 5           0.5, 0, -0.5, 0.5, 0, 0.5, 0, -1, 0,
// Triangle 6           0.5, 0, 0.5, -0.5, 0, 0.5, 0, -1, 0,
// Triangle 7           -0.5, 0, 0.5, -0.5, 0, -0.5, 0, -1, 0,
// Triangle 8           -0.5, 0, 0.5, -0.5, 0, 0.5, 0, -1, 0};
```

Et le tableau de coordonnées de texture :

Code : C++

```
// Coordonnées de texture temporaires

float coordTexture[] = {0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
1           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
2           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
3           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
4           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
5           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
6           0, 0, 0.5, 0, 0.5, 0.5, 0.5, // Triangle
7           0, 0, 0.5, 0, 0.5, 0.5, 0.5}; // Triangle
8
```

Ce qui donne le constructeur :

Code : C++

```
Cristal::Cristal(std::string const vertexShader, std::string const
fragmentShader, std::string const texture) : m_shader(vertexShader,
fragmentShader), m_texture(texture)
{
    // Chargement du shader
    m_shader.charger();

    // Chargement de la texture
    m_texture.charger();

    // Vertices temporaires
    float verticesTmp[] = {-0.5, 0, -0.5, 0.5, 0, -0.5, 0, 1, 0,
// Triangle 1           0.5, 0, -0.5, 0.5, 0, 0.5, 0, 1, 0,
// Triangle 2           0.5, 0, 0.5, -0.5, 0, 0.5, 0, 1, 0,
```

```

// Triangle 3           -0.5, 0, 0.5, -0.5, 0, -0.5, 0, 1, 0,
// Triangle 4           -0.5, 0, -0.5, 0.5, 0, -0.5, 0, -1,
0,      // Triangle 5   0.5, 0, -0.5, 0.5, 0, 0.5, 0, -1, 0,
// Triangle 6           0.5, 0, 0.5, -0.5, 0, 0.5, 0, -1, 0,
// Triangle 7           -0.5, 0, 0.5, -0.5, 0, -0.5, 0, -1, 0,
0};     // Triangle 8

// Coordonnées de texture temporaires

float coordTexture[] = {0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 1           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 2           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 3           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 4           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 5           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 6           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 7           0, 0, 0.5, 0, 0.5, 0.5, // 
Triangle 8           0, 0, 0.5, 0, 0.5}; // 

// Copie des vertices

for(int i(0); i < 72; i++)
    m_vertices[i] = verticesTmp[i];

// Copie des coordonnées

for(int i(0); i < 48; i++)
    m_coordTexture[i] = coordTexture[i];
}

```

La méthode afficher()

Le reste de la classe est identique à celle du sol. La seule différence concerne la fonction **glDrawArrays()** qui doit afficher **24** vertices et non **6** :

Code : C++

```

void Cristal::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader

    glUseProgram(m_shader.getProgramID());

    // Envoi des vertices

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
m_vertices);

```

```
glEnableVertexAttribArray(0);

    // Envoi des coordonnées de texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
m_coordTexture);
    glEnableVertexAttribArray(2);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));

    // Verrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, m_texture.getID());

    // Rendu
    glDrawArrays(GL_TRIANGLES, 0, 24);

    // Déverrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, 0);

    // Désactivation des tableaux
    glDisableVertexAttribArray(2);
    glDisableVertexAttribArray(0);

    // Désactivation du shader
    glUseProgram(0);
}
```

Afficher le tout

Maintenant que nous avons toutes nos classes, il ne nous reste plus qu'à les instancier pour remplir notre scène. Celle-ci doit contenir une cabane, deux types de sol, un cristal et quelques caisses :

Code : C++

```
// Cabane
Cabane cabane("Shaders/texture.vert", "Shaders/texture.frag");

// Sols
Sol solHerbeux(30.0, 30.0, 15, 15, "Shaders/texture.vert",
"Shaders/texture.frag", "Textures/Herbe.jpg");
Sol solTerreux(10.0, 10.0, 5, 5, "Shaders/texture.vert",
"Shaders/texture.frag", "Textures/Sol.jpg").
```

```
shaders/texture.vert", "Shaders/texture.frag",  
"Textures/Caisse2.jpg");  
Caisse caisseDanger(2.0, "Shaders/texture.vert",  
"Shaders/texture.frag", "Textures/Caisse.jpg");  
  
// Cristal  
  
Cristal cristal("Shaders/texture.vert", "Shaders/texture.frag",  
"Textures/Cristal.tga");  
float angle(0.0);
```



Si vous souhaitez inclure plusieurs caisses ayant la même texture alors vous n'avez pas besoin d'en déclarer plusieurs, une seule suffit. Vous pouvez ensuite appeler sa méthode **afficher()** plusieurs fois.

Au niveau de l'affichage, il suffit d'appeler la méthode **afficher()** pour tous les objets. On commence avec la cabane et le sol :

Code : C++

```
// Affichage de la cabane  
  
cabane.afficher(projection, modelview);  
  
// Affichage du sol terreux  
  
solTerreux.afficher(projection, modelview);  
  
// Affichage du sol herbeux  
  
mat4 sauvegardeModelview = modelview;  
  
modelview = translate(modelview, vec3(0, -0.01, 0));  
solHerbeux.afficher(projection, modelview);  
  
modelview = sauvegardeModelview;
```

Il y a deux détails qui doivent retenir notre attention ici :

- La translation de **0.01** sur l'axe **Y** qui permet d'éviter un gros bug de texture.
- Les objets **cabane** et **solTerreux** qui ne sont pas encadrés une sauvegarde de la matrice. Vu que nous n'utilisons aucune transformation pour les afficher, nous n'en avons pas besoin.

Il ne manque plus qu'à placer les caisses et le cristal à l'aide de différentes translations. On commence avec les caisses sans oublier de sauvegarder et restaurer la matrice **modelview**. D'ailleurs, il n'y a pas besoin de redéclarer la sauvegarde vu qu'elle l'a déjà été avant. 😊

Code : C++

```
// Sauvegarde de la matrice  
  
sauvegardeModelview = modelview;
```

```
// Première caisse  
modelview = translate(modelview, vec3(-2.5, 1, -3));  
caisse.afficher(projection, modelview);  
  
// Deuxième caisse  
modelview = translate(modelview, vec3(5, 0, 1));  
caisseDanger.afficher(projection, modelview);  
  
// Troisième caisse  
modelview = translate(modelview, vec3(-2.5, 0, 4));  
caisse.afficher(projection, modelview);  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;
```

Vu que les caisses sont assez proches, nous pouvons nous permettre de ne sauvegarder et de ne restaurer la matrice **modelview** qu'une seule fois.

On termine l'affichage par le fameux cristal qui doit pivoter sur lui-même indéfiniment. On utilise pour cela la même technique que celle du cube dans le chapitre sur la 3D, à savoir l'utilisation d'un angle incrémenté de **1** à chaque tour de boucle. Si cet angle dépasse **360°** alors on lui soustrait justement **360°**:

Code : C++

```
// Sauvegarde de la matrice  
sauvegardeModelview = modelview;  
  
// Affichage des caisses  
....  
  
// Rotation du cristal  
angle++;  
  
if(angle > 360)  
    angle -= 360;  
  
// Affichage du cristal  
modelview = translate(modelview, vec3(0, 2.1, 0));  
modelview = rotate(modelview, angle, vec3(0, 1, 0));  
cristal.afficher(projection, modelview);  
  
// Restauration de la matrice  
modelview = sauvegardeModelview;
```

Faites attention à placer ce bout de code dans le bloc "sauvegarde/restauration de la matrice" des caisses, ne refaites pas de sauvegarde juste pour afficher le cristal. 😊

Téléchargement

Comme toute fin de chapitre, je vous propose de télécharger une archive contenant le code que nous avons fait. Ou plutôt que vous avez fait dans ce cas. 😊

Télécharger (Windows - UNIX/Linux) : Correction du TP - Une relique retrouvée

Nous venons de terminer notre premier TP, celui-ci nous a permis d'utiliser toutes les notions que avons vues sur OpenGL. 😊

Je vous conseille de mettre de coté le code que vous avez réalisé vous-même, le TP de la deuxième partie sera basé dessus. Bien évidemment, nous y rajouterons une multitude de nouveautés, nous les verrons toutes ensemble.

Je vous invite d'ailleurs à lire la deuxième partie de ce tutoriel qui sera consacrée aux notions avancées d'OpenGL. Je vous parlerai enfin des shaders en détails, vous saurez tout à leur sujet depuis la classe qui leur est dédiée jusqu'aux petits fichiers sources du dossier **Shaders**. Nous verrons également pas mal d'autres choses qui seront principalement axées autour de la carte graphique. 😊

Partie 2 : Les notions avancées

Bienvenue dans la deuxième partie de ce tutoriel, j'espère que vous avez survécu aux chapitres précédents. 😊

Vous avez intérêt en plus car nous allons voir ici les *notions avancées d'OpenGL*, notamment les fameux **shaders**. Je vous expliquerai tout de A à Z à leur sujet mais ce n'est pas la seule chose que nous verrons, nous aborderons aussi tout un tas de fonctionnalités qui se focaliseront principalement sur la carte graphique. Nous apprendrons à héberger des données dessus pour rendre nos programmes plus rapides, à faire des rendus dans des '**écrans cachés**', ...

L'objectif de cette partie est simple : maîtriser notre carte graphique. 😊

Les Vertex Buffer Objects

On attaque cette deuxième partie du tutoriel avec des *objets OpenGL* qui vont nous être très utiles par la suite : les **Vertex Buffer Objects**. Ils sont très liés à la carte graphique car ils permettent d'héberger des données directement dessus (vertex, coordonnées de texture, ...).

Vu que nous avons déjà eu l'occasion de manipuler les *objets OpenGL* avec les textures, vous retrouverez pas mal de notions similaires, et tant mieux parce que ça nous facilite la vie. 😊

C'est quoi un VBO ?

Définition

Je vous ai déjà donné une rapide définition des **Vertex Buffer Objects** (ou **VBO**) dans l'introduction mais faisons comme si vous n'avez rien vu. 😊



Un Vertex Buffer Object qu'est ce c'est ?

C'est un *objet OpenGL* qui contient des données relatives à un modèle 3D comme les vertices, les coordonnées de texture, les normales (pour les lumières), ... Sa particularité vient du fait que les données qu'il contient se trouvent non pas dans la **RAM** mais directement dans la carte graphique.

Lorsque vous achetez une carte graphique, je suis sûr que la majorité d'entre vous ne regarde qu'une seule caractéristique : la quantité de mémoire RAM (ou plutôt **VRAM** pour **Vidéo Random Access Memory**). Cette mémoire se comporte exactement comme la **RAM** classique, elle stocke des informations qui seront utilisées plus tard par un programme.

Un **Vertex Buffer Object** est donc une zone mémoire (**buffer**) appartenant à la carte graphique dans laquelle on peut stocker des données.

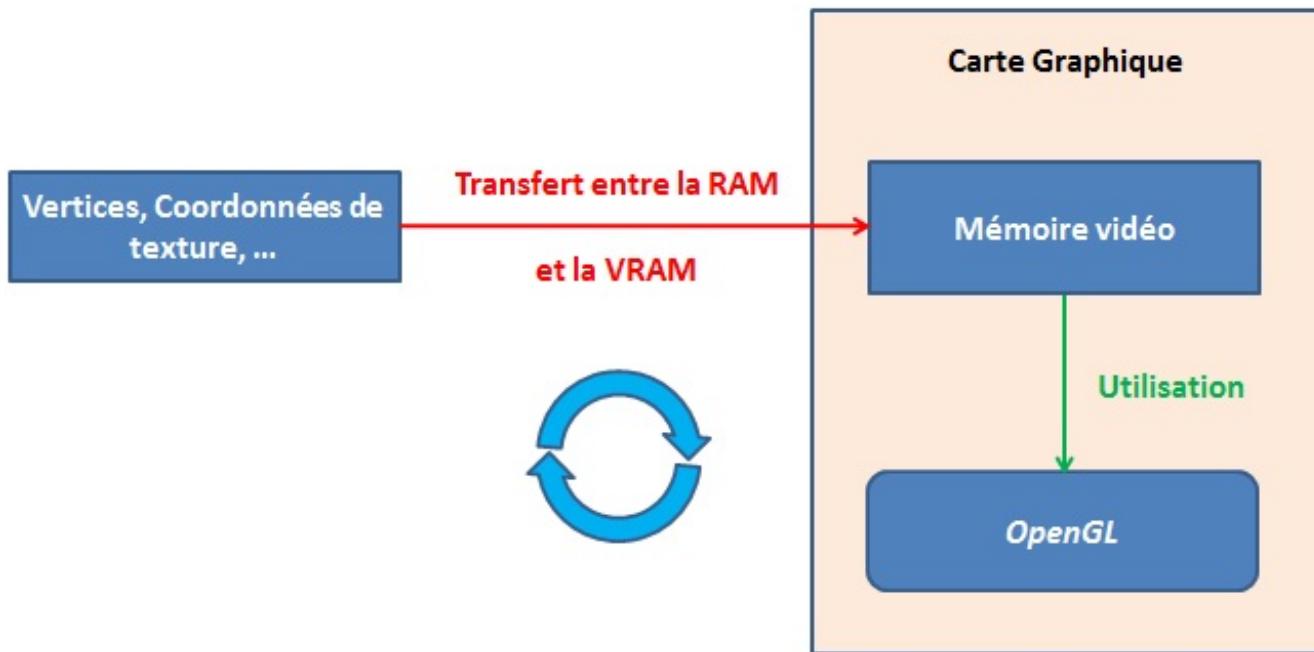
Utilité

Vous vous posez peut-être la question de savoir quel intérêt peut-on avoir à stocker des données directement dans la carte graphique ?

L'intérêt n'est peut-être pas évident pour le moment car nous n'avons pas de projets d'envergure pour le moment, mais sachez que les **VBO** permettent de gagner beaucoup de temps calculs en évitant à *OpenGL* des aller et retours inutiles.

En effet, lorsque vous affichez un modèle 3D (comme une maison par exemple), vous devez spécifier ses vertices, ses coordonnées de texture, ... Toutes ces données se trouvent automatiquement dans la **RAM**. Or, à chaque fois que vousappelez la fonction **glDrawArrays()** ou **glDrawElements()** pour afficher votre modèle, *OpenGL* va chercher toutes les données correspondantes dans la **RAM** pour les transférer dans la mémoire graphique pour qu'ensuite seulement il puisse travailler avec :

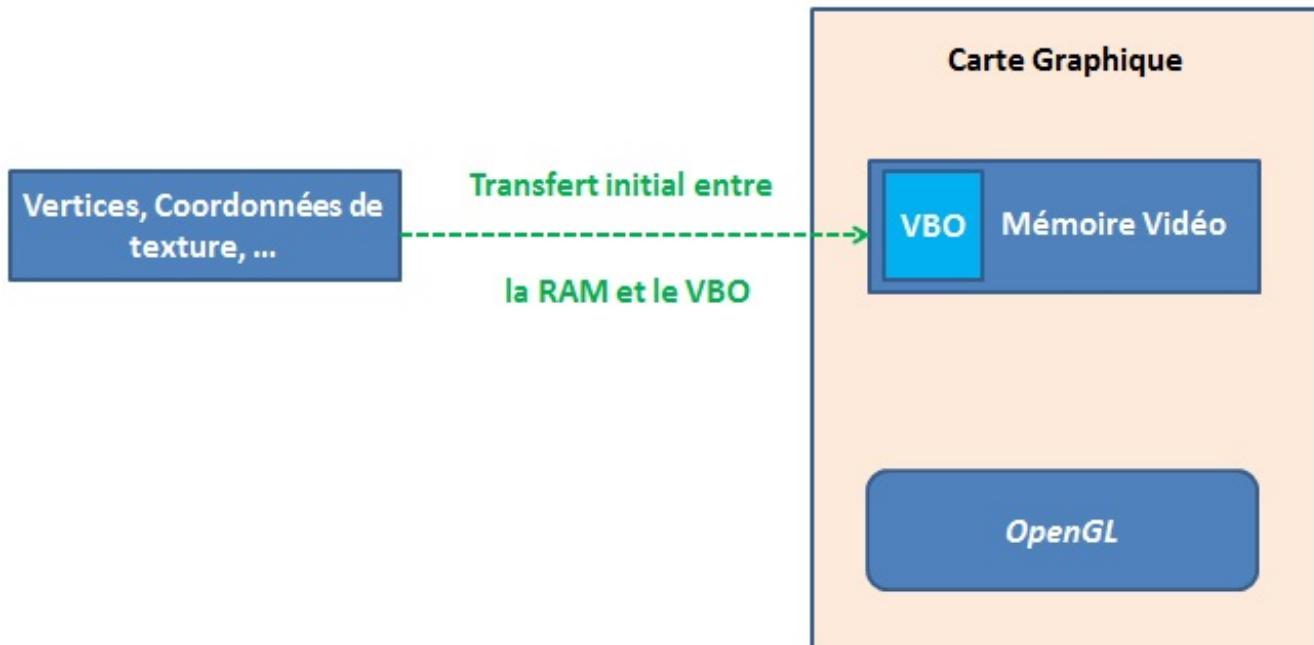
Boucle d'affichage



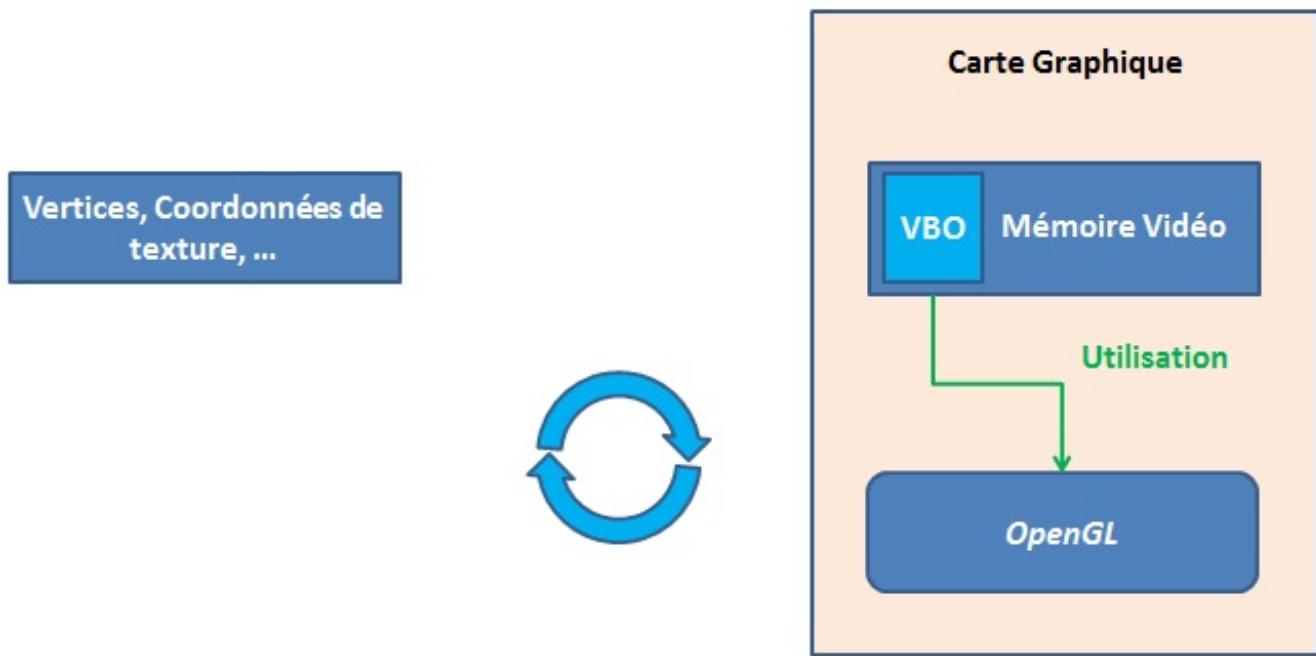
Je pense que vous aurez compris que ces transferts ralentissent pas mal le temps de calcul, surtout quand vous transférez des centaines de milliers de données 60 fois par seconde !

Pour éviter de perdre du temps avec ces transferts, *OpenGL* nous propose d'utiliser les **Vertex Buffer Objects** pour accéder directement à la mémoire vidéo. Au moment du chargement d'un modèle, on envoie toutes les données au VBO qui va se contenter de les stocker dans la carte graphique. Il n'y a donc plus qu'un seul transfert :

Chargement du modèle 3D



Boucle d'affichage



On économise ainsi le transfert de centaines de milliers de données à chaque affichage, la carte graphique peut travailler plus rapidement.

Attention cependant, on conserve tout de même une copie des données dans la **RAM**. Elles ne sont plus copiées à chaque affichage mais elles sont quand même là. On doit les converser pour pouvoir effectuer des calculs dessus, comme un déplacement de personnage par exemple.

D'ailleurs, lorsqu'on les mettra à jour il faudra les re-transférer dans le **VBO**, mais ça ce sera pour la fin du chapitre. 😊

Création Le header

Introduction

On sait maintenant ce que sont les **Vertex Buffer Objects**, on peut donc passer à leur implémentation. Mais avant tout je voudrais faire une petite comparaison avec les textures.

Souvenez-vous que les textures sont des *objets OpenGL* eux-aussi et que les fonctions qui permettent de les gérer sont quasiment les mêmes pour tous les *objets*. Ainsi, on retrouvera des fonctions identiques aux textures pour gérer les **VBO** comme `glGenXXX()`, `glBindXXX()` et `glDeleteXXX()`. De plus, on retrouvera également les variables de type **GLuint** qui les représentera et qu'on utilisera notamment dans le verrouillage.

Dans ce chapitre, nous travaillerons sur la classe **Cube** en intégrant un **Vertex Buffer Objet** pour l'afficher. Nous ferons de même par la suite pour la classe **Caisse** en prenant en compte l'héritage C++. 😊

Nouvel attribut

Pour commencer, nous allons ajouter à notre liste d'attributs une nouvelle variable de type **GLuint** que nous appellerons **m_vbOID** et qui représentera l'identifiant du **VBO**. Nous en aurons besoin pour sa configuration et son utilisation.

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE

// Includes
....  
  
// Classe Cube
class Cube
{
public:
    Cube(float taille, std::string const vertexShader, std::string
const fragmentShader);
    ~Cube();

    void afficher(glm::mat4 &projection, glm::mat4 &modelview);

protected:
    Shader m_shader;
    float m_vertices[108];
    float m_couleurs[108];

    GLuint m_vboID;
};

#endif
```

Comme tout nouvel attribut, il faut penser à l'initialiser dans le constructeur. Nous lui donnerons la valeur **0** vu qu'il est vide pour le moment :

Code : C++

```
Cube::Cube(float taille, std::string const vertexShader, std::string
const fragmentShader) : m_shader(vertexShader, fragmentShader),
m_vboID(0)
{
    // Initialisation
    ....
}
```

La méthode `charger()`

Génération de l'identifiant

Nous n'allons pas gérer la création du **VBO** dans le constructeur car cela poserait problème avec l'héritage dans les classes filles. A la place, nous allons créer une méthode à part que nous appellerons simplement `charger()` comme pour les textures et le shader. Son prototype ne sera pas compliqué vu qu'il n'y aura aucun paramètre à gérer :

Code : C++

```
void charger();
```

Ne lappelez pas **chargerVBO()** car nous mettrons autre chose à l'intérieur dans le chapitre suivant. 😊

Pour en revenir à notre nouvel attribut, rappelez-vous que les identifiants (ou **ID**) sont créés grâce à la fonction **glGenXXX()** et représentent l'*objet OpenGL* auquel ils sont rattachés. Pour créer celui des textures par exemple, nous utilisions la fonction **glGenTextures()**.

Pour les **VBO**, nous utiliserons la même fonction sauf que l'on remplacera le mot **Texture** par **Buffer** (ou zone mémoire). La fonction à utiliser devient donc **glGenBuffers()** :

Code : C++

```
GLuint glGenBuffers(GLsizei number, GLuint *buffers);
```

- **number** : Le nombre d'ID à initialiser. Nous lui donnerons toujours la valeur **1** comme pour les textures
- **buffers** : Un tableau de type **GLuint**. On peut également mettre l'adresse d'une variable **GLuint** pour n'initialiser qu'un seul ID

Donc pour générer un nouvel **ID**, nous devons appeler cette fonction en donnant en paramètre **1** (pour n'en créer qu'un seul) ainsi que l'adresse de l'attribut **m_vboID** pour récupérer la valeur renournée :

Code : C++

```
void Cube::charger()
{
    // Génération de l'ID
    glGenBuffers(1, &m_vboID);
}
```

Le verrouillage

Si je vous parle de la notion de **verrouillage**, vous vous souvenez de ce que ce que ça signifie ? 😊

A chaque fois que l'on veut configurer ou utiliser un *objet OpenGL* il faut le verrouiller car OpenGL a justement besoin de savoir sur quelle chose il doit travailler.

Pour configurer les textures par exemple, il fallait les verrouiller avant de leur donner les pixels d'une image. Si nous ne l'avions pas fait, nous aurions envoyé les pixels on ne sait où dans la mémoire.

Avec les **VBO** c'est la même chose, avant de les configurer ou de les utiliser il faut les verrouiller.

La fonction permettant de faire cette opération s'appelle **glBindXXX()**, avec les **VBO** ce sera **glBindBuffer()**.

Code : C++

```
void glBindBuffer(GLenum target, GLuint buffer);
```

- **target** : type de l'*objet* que l'on veut verrouiller. Comme d'habitude, à chaque fois que l'on verra ce paramètre on lui

donnera toujours la même valeur, et dans notre cas ce sera **GL_ARRAY_BUFFER**

- **buffer** : ID qui représente le **VBO**. On lui donnera la *valeur* de l'ID et pas son *adresse* cette fois. 😊

On appelle donc cette fonction en donnant en paramètre **GL_ARRAY_BUFFER** pour **target** et l'attribut **m_vboID** pour **buffer** :

Code : C++

```
void Cube::charger()
{
    // Génération de l'ID
    glGenBuffers(1, &m_vboID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);
}
```

Je profite également de cette partie pour vous montrer le déverrouillage d'objet. Vous savez qu'une fois que nous n'utilisons plus nos objets il faut les déverrouiller, pour éviter d'écrire par erreur dedans par exemple.

Pour déverrouiller un **VBO**, il suffit d'appeler la fonction **glBindbuffer()** avec un ID de **0** (valeur nulle).

Code : C++

```
void Cube::charger()
{
    // Génération de l'ID
    glGenBuffers(1, &m_vboID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Configuration
    ...

    // Déverrouillage de l'objet
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```



Je vous conseille d'indenter le code qui se trouve entre les appels à **glBindBuffer()** de la même manière que **glUseProgram()**, ça vous permettra de mieux voir ce que vous lui envoyez.

Allocation de la mémoire vidéo (Partie 1/2)

Maintenant que nous avons un **VBO** créé et verrouillé, nous allons pouvoir lui transférer les données stockées dans la RAM. Les transferts seront un peu déroutants car nous ne transférons pas directement des **float** ou des **unsigned int** mais des **bytes**.

Mais avant cela, il va falloir définir une zone mémoire à l'intérieur-même de la carte graphique. Les **VBO** se comportent un peu comme l'allocation dynamique, il faut allouer dynamiquement de la place dans la mémoire en fonction des données à transférer.

Pour allouer de la mémoire normale (RAM), nous devons utiliser le mot-clé **new[]** avec la taille désirée. Dans notre cas, il va falloir passer par une fonction dédiée qui s'appelle **glBufferData()** :

Code : C++

```
glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data,  
GLenum usage)
```

- **target** : On ne le présente plus, on lui donnera la valeur **GL_BUFFER_DATA**
- **size** : C'est la taille mémoire à allouer (en bytes), il faut prendre en compte **TOUTES** les données à envoyer (vertices, coordonnées de texture, ...) correspondant à un modèle 3D
- **data** : Ce sont les données à transférer. On n'enverra rien du tout pour le moment car on ne peut pas envoyer plusieurs tableaux en un seul paramètre, nous lui affecterons la valeur **0**
- **usage** : Permet de définir la fréquence de mise à jour des données

Le deuxième et le quatrième paramètres sont les plus importants, nous allons passer un peu plus de temps dessus.

Le paramètre **size** est un peu particulier, il correspond à la taille de la mémoire à allouer en **bytes** (ou **octets**). Pourquoi en bytes ? Parce que les variables primitives ne font pas forcément la même taille dans la **RAM** que dans la **VRAM**, une variable de type **float** par exemple peut prendre plus de place dans l'une et moins dans l'autre.

Pour transférer des variables correctement, il faut trouver un moyen de communication commun entre les deux mémoires, et ce moyen c'est le **byte**.

Il nous faut donc demander à la carte graphique une zone mémoire exprimée en bytes, elle devra être assez grande pour pouvoir contenir toutes les données. Dans la classe **Cube**, nous avons un tableau pour les vertices et un autre pour les couleurs. La zone devra donc faire la taille de ces deux-là.

Pour avoir la taille d'un tableau, il suffit de multiplier le nombre de ses cases par la taille en bytes du **type** de variable utilisée. Je vous rappelle que la fonction permettant d'avoir la taille d'un type de variable s'appelle **sizeof(type)**.

Pour calculer la taille des vertices par exemple, nous savons que nous en avons **108** donc nous faisons **108 * sizeof(float)** :

Code : C++

```
// Taille du tableau de vertices  
  
int tailleVerticesBytes = 108 * sizeof(float);
```

Pour le tableau de couleurs, on fait exactement le même calcul. Nous avons **108** cases donc nous faisons **108 * sizeof(float)** :

Code : C++

```
// Taille du tableau de couleurs  
  
int tailleCouleursBytes = 108 * sizeof(float);
```

La taille de la zone mémoire à allouer dans la carte graphique vaudra donc **tailleVerticesBytes + tailleCouleursBytes**. 😊

De nouveaux attributs

Je profite de cette partie pour faire une petite parenthèse. Nous aurons besoin des attributs précédents plusieurs fois dans la classe **Cube** (et ses filles). Il vaut donc mieux créer des attributs les représentant plutôt que de les redéclarer à la main à chaque fois.

Rajoutons donc deux nouvelles variables **m_tailleVerticesBytes** et **m_tailleCouleursBytes** dans le header :

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE

// Includes
....

// Classe Cube

class Cube
{
    public:

        Cube(float taille, std::string const vertexShader, std::string
const fragmentShader);
        ~Cube();

        void afficher(glm::mat4 &projection, glm::mat4 &modelview);

    protected:

        Shader m_shader;
        float m_vertices[108];
        float m_couleurs[108];

        GLuint m_vboID;
        int m_tailleVerticesBytes;
        int m_tailleCouleursBytes;
};

#endif
```

Initialisons-les ensuite dans le constructeur en leur donnant la taille des vertices et des couleurs en bytes :

Code : C++

```
Cube::Cube(float taille, std::string const vertexShader, std::string
const fragmentShader) : m_shader(vertexShader, fragmentShader),
m_vboID(0),

m_tailleVerticesBytes(108 * sizeof(float)),
m_tailleCouleursBytes(108 * sizeof(float))
{
    // Initialisation
    ....
}
```

Le paramètre **size** de la fonction **glBufferData()** prendra donc la valeur **m_tailleVerticesBytes + m_tailleCouleursBytes**.

Allocation de la mémoire vidéo (Partie 2/2)

On ferme la parenthèse et on revient à l'étude des autres paramètres. Le prochain sur la liste est **usage** et il sera plus rapide à comprendre. 😊

Il permet à OpenGL de savoir si les données que nous stockerons dans le **VBO** seront mises à jour rarement, fréquemment ou tout le temps.

Par exemple, un personnage qui bouge devra quasiment tout le temps mettre à jour ses vertices, une caisse en revanche ne devra jamais le faire. OpenGL à besoin de connaître cette fréquence de mise à jour, c'est pour ça qu'il nous donne le paramètre **usage**.

En théorie, il existe 9 valeurs possibles pour ce paramètre mais nous n'en retiendrons que 3 :

- **GL_STATIC_DRAW** : pour les données très peu mises à jour
- **GL_DYNAMIC_DRAW** : pour les données mises à jour fréquemment (plusieurs fois par seconde mais pas à chaque frame)
- **GL_STREAM_DRAW** : pour les données mises à jour tout le temps (A chaque frame cette fois-ci)

Vu que le cube ne bouge pas, nous mettrons la valeur **GL_STATIC_DRAW**. 😊

En définitif, avec tous les paramètres que l'on vient de voir, nous appellerons la fonction **glBufferData()** de cette façon :

Code : C++

```
void Cube::charger()
{
    // Génération de l'ID
    glGenBuffers(1, &m_vboID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Allocation de la mémoire
    glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +
    m_tailleCouleursBytes, 0, GL_STATIC_DRAW);

    // Déverrouillage de l'objet
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Avec ça, on vient de communiquer directement avec la carte graphique pour lui demander si elle pouvait nous réserver un petit espace mémoire. 😊

Transfert des données

La fonction **glBufferData()** nous a permis d'allouer un espace mémoire pour y stocker nos données, et vous avez vu que nous n'avons transféré aucune donnée même si elle nous le proposait. Nous ne l'avons pas fait car il est impossible d'envoyer plusieurs tableaux dans un seul paramètre.

Pour transférer les données, nous allons utiliser une autre fonction OpenGL dont le nom ressemble étonnamment à celui de la fonction précédente : **glBufferSubData()**.

Code : C++

```
glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size,  
const GLvoid *data);
```



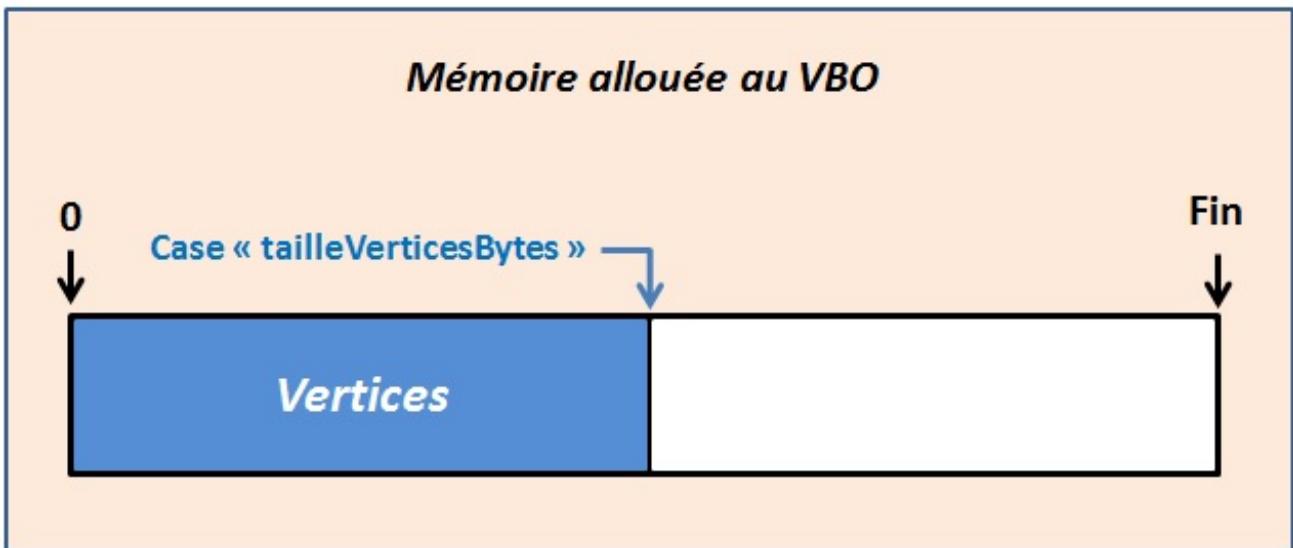
Petite précision au passage, cette fonction est un poil plus rapide que sa grande sœur **glBufferData()** pour transférer des données. Donc même si nous n'avons qu'un seul tableau à envoyer au **VBO** on préférera utiliser **glBufferSubData()**.

- **target** : Toujours le même, on lui donnera la valeur **GL_BUFFER_DATA**
- **offset** : Case en mémoire où on va commencer le transfert dans la **VRAM**
- **size** : La taille des données à copier (en bytes)
- **data** : Les données à copier, par exemple le tableau de vertices

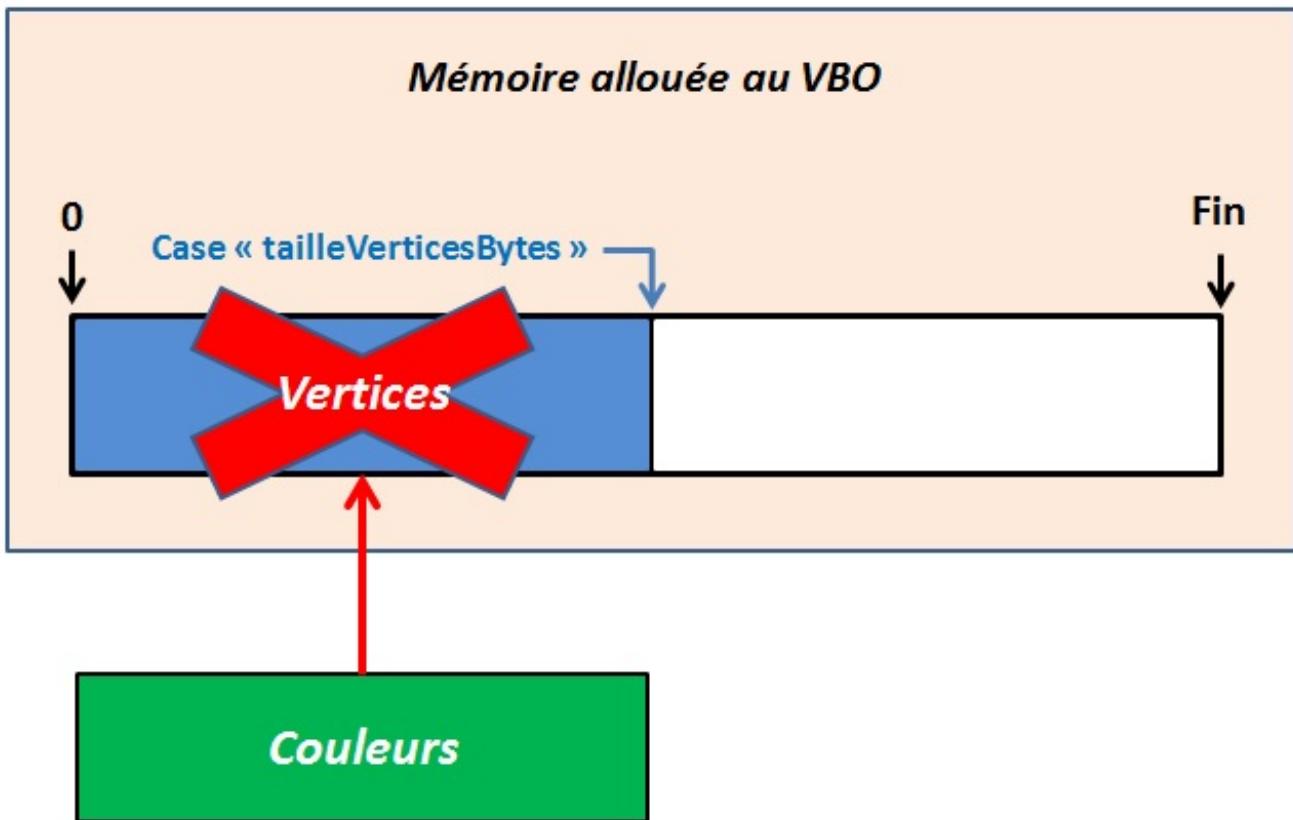
Les paramètres importants ici sont **l'offset** et **le size**.

Je commence par le paramètre **size**, il correspond à la taille (en bytes) des données à envoyer. Si nous envoyons le tableau de vertices, nous donnerons la taille du tableau que nous avons calculée juste avant. Même chose pour le tableau de couleurs.

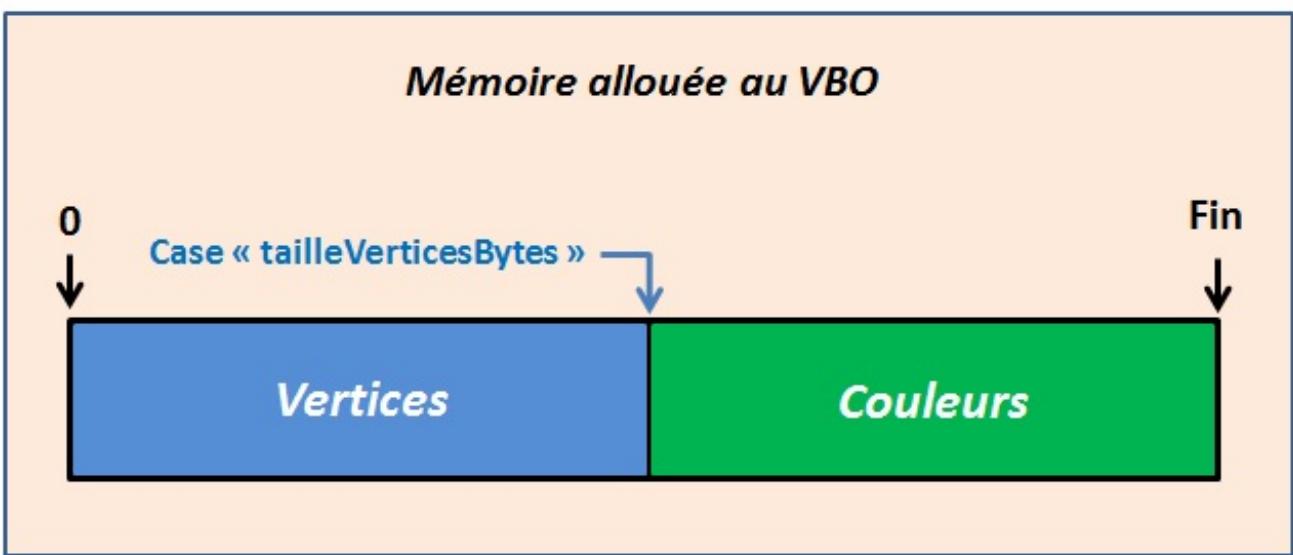
Le paramètre **offset** correspond quant à lui à la case mémoire dans laquelle va commencer la copie. Pour le transfert des vertices, ce paramètre sera de **0** vu que l'on commence à copier au début de la zone mémoire :



En revanche pour le transfert des couleurs, on ne va pas commencer la copie à la case **0** sinon on va écraser les valeurs transférées juste avant :



Il faudra commencer la copie à la fin du tableau de vertices, soit à la case **tailleVerticesBytes** :



A partir de cette case, on n'écrase plus rien, on copie au bon endroit.

D'ailleurs, **offset** signifie **décalage** en français, on renseigne un décalage dans la plage mémoire. 😊

Au final, pour envoyer toutes nos données à la carte graphique nous devons appeler la fonction **glBindSubData()** deux fois en faisant attention à copier le **bon volume** de données au **bon endroit**.

Le premier transfert s'occupera de copier le tableau de vertices dans la mémoire vidéo à partir de la case **0** :

Code : C++

```
// Transfert des vertices
```

```
glBufferSubData(GL_ARRAY_BUFFER, 0, m_tailleVerticesBytes,  
m_vertices);
```

Le second transfert s'occupera de copier le tableau de couleurs dans la mémoire vidéo juste après les vertices, soit à partir de la case **tailleVerticesBytes** :

Code : C++

```
// Transfert des couleurs  
  
glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,  
m_tailleCouleursBytes, m_couleurs);
```

On fait un petit récapitulatif de la création d'un **VBO** :

Code : C++

```
void Cube::charger()  
{  
    // Génération de l'ID  
  
    glGenBuffers(1, &m_vboID);  
  
    // Verrouillage du VBO  
  
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);  
  
    // Allocation de la mémoire vidéo  
  
    glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +  
    m_tailleCouleursBytes, 0, GL_STATIC_DRAW);  
  
    // Transfert des données  
  
    glBufferSubData(GL_ARRAY_BUFFER, 0, m_tailleVerticesBytes,  
    m_vertices);  
    glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,  
    m_tailleCouleursBytes, m_couleurs);  
  
    // Déverrouillage de l'objet  
  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
}
```

Éviter les fuites de mémoire

Si vous vous souvenez de ce que l'on a vu sur le changement de texture, vous vous souviendrez que nous avons ajouté une petite ligne de code au début de la méthode **charger()**. En effet, je vous avais parlé de ce qui se passait si on appelait deux fois cette méthode sur un même objet : cela entraînait une fuite de mémoire car le premier chargement était perdu dans la carte graphique.

Avec les **VBO**, nous avons le même problème : si on en charge un deux fois alors le premier chargement effectué sera perdu et

les ressources seront considérées comme étant "toujours utilisées". Pour éviter cette fuite de mémoire, nous allons faire la même chose que les textures en vérifiant d'une part si le **VBO** a déjà été chargé, puis en le détruisant si c'est le cas. Le tout au début de la méthode **charger()** avant l'initialisation.

La fonction permettant de savoir si un **VBO** a déjà été chargé s'appelle **glIsBuffer()**:

Code : C++

```
GLboolean glIsBuffer(GLuint buffer);
```

Elle ne prend en paramètre que l'identifiant à vérifier et renvoie la valeur **GL_TRUE** si le **VBO** a déjà été chargé ou **GL_FALSE** si ne l'a pas été.

La fonction de destruction quant à elle s'appelle **glDeleteBuffers()**:

Code : C++

```
void glDeleteBuffers(GLsizei number, const GLuint *buffers);
```

- **number** : Le nombre d'ID à initialiser. Nous lui donnerons la valeur **1**
- **buffers** : Un tableau de type **GLuint** ou l'adresse d'une variable **GLuint**. Nous lui donnerons l'ID du **VBO** à détruire.

Nous appellerons ces deux fonctions dans un bloc **if** au début de la méthode **charger()**:

Code : C++

```
void Cube::charger()
{
    // Destruction d'un éventuel ancien VBO

    if(glIsBuffer(m_vboID) == GL_TRUE)
        glDeleteBuffers(1, &m_vboID);

    // Génération de l'ID
    glGenBuffers(1, &m_vboID);

    ...
}
```

Ce qui donne le code source définitif :

Code : C++

```
void Cube::charger()
{
    // Destruction d'un éventuel ancien VBO

    if(glIsBuffer(m_vboID) == GL_TRUE)
        glDeleteBuffers(1, &m_vboID);

    // Génération de l'ID
    glGenBuffers(1, &m_vboID);
```

```
// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

// Allocation de la mémoire vidéo
glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +
m_tailleCouleursBytes, 0, GL_STATIC_DRAW);

// Transfert des données
glBufferSubData(GL_ARRAY_BUFFER, 0, m_tailleVerticesBytes,
m_vertices);
glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,
m_tailleCouleursBytes, m_couleurs);

// Déverrouillage de l'objet
glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Notre premier **VBO** est prêt à l'emploi. 😊

Le destructeur

Encore une fois, tout comme les textures, les **VBO** doivent être détruits lorsque nous n'en avons plus besoin. Vu que nous sommes dans une classe, nous devons gérer cette destruction dans la méthode prévue pour ça : le destructeur.

Cette tâche se fait avec la fonction **glDeleteBuffers()** que nous avons vue précédemment. Nous l'appellerons dans le destructeur en lui donnant l'adresse de l'attribut **m_vboID** :

Code : C++

```
Cube::~Cube ()
{
    // Destruction du VBO
    glDeleteBuffers(1, &m_vboID);
}
```

Utilisation

Utilisation d'un VBO

Maintenant que nous avons un **VBO** créé et initialisé, on ne va pas se priver du plaisir de l'utiliser dans nos programmes. 😊

Pour utiliser un **VBO** on fait exactement la même chose qu'avec les textures. D'ailleurs, vous vous souvenez de ce qu'on doit faire pour envoyer une texture à OpenGL ?

....

On la verrouille !

On va faire exactement la même chose avec les **VBO**, pour les utiliser on va les verrouiller. Comme tous les objets OpenGL j'ai envie de dire. 🎉



Ok mais euh ... On le verrouille où ?

Ah très bonne question. 😊

Jusqu'à ce chapitre, pour afficher un modèle 3D nous devions envoyer les données à l'aide de la fonction **glVertexAttribPointer()** puis nous affichions le tout avec la fonction **glDrawArrays()**. Nous continuerons à procéder ainsi sauf que nous ajouterons le verrouillage du **VBO** juste avant la fonction **glVertexAttribPointer()**.

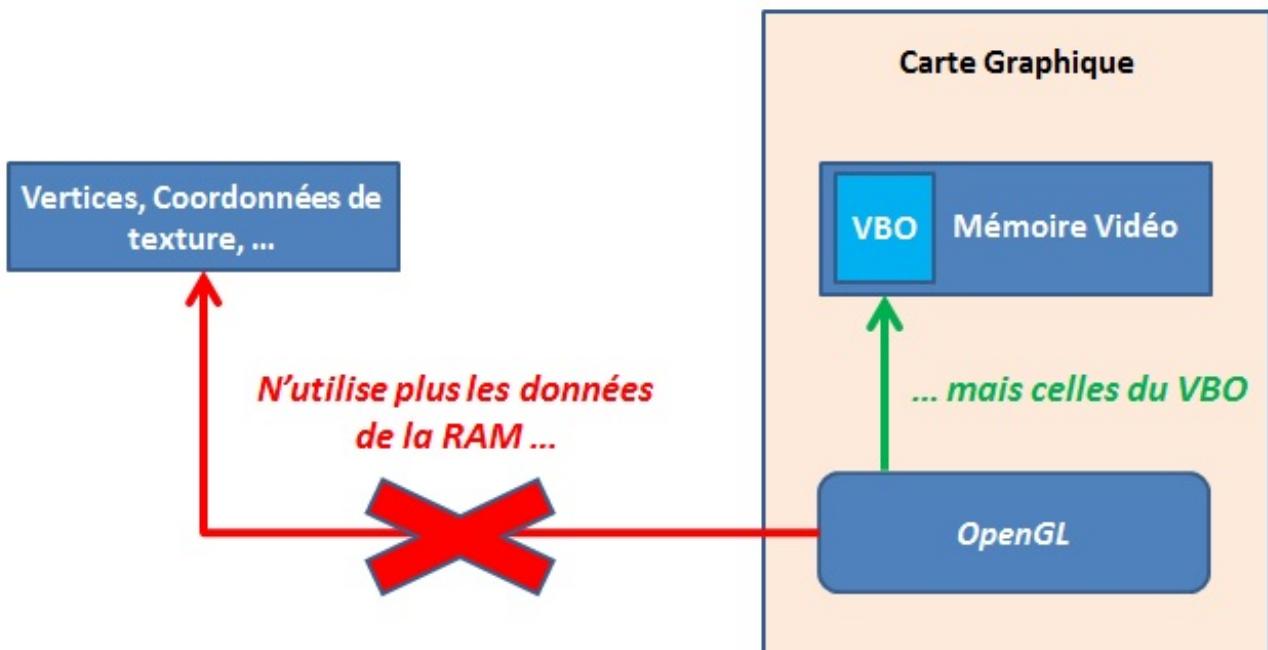
En effet même si les données se trouvent dans la mémoire vidéo, OpenGL ne sait pas où elles se situent exactement, c'est à nous de lui dire. Pour ça, on va toujours utiliser la fonction **glVertexAttribPointer()** sauf qu'on va modifier son dernier paramètre. Je vous redonne son prototype pour que vous visualisiez le paramètre en question :

Code : C++

```
void glVertexAttribPointer(GLuint index, GLuint size, GLenum type,
GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

Le paramètre **pointer** permet à OpenGL de savoir où se trouvent nos données (vertices et toute la clique je le rappelle). Avant nous donnions directement les tableaux à ce paramètre, ils étaient stockés dans la RAM ça ne posait pas de problème. Mais maintenant nous devons lui donner l'adresse des tableaux à l'intérieur de la mémoire vidéo.

Boucle d'affichage



Lorsque le **VBO** est verrouillé, OpenGL sait qu'il doit aller chercher les données dans la plage mémoire qui lui est consacrée. A partir de maintenant, le paramètre **pointer** n'attend plus un tableau mais un **offset**.

Offset ça ne vous rappelle pas un paramètre ? C'est celui de la fonction **glBufferSubData()**, il représente la case (le *décalage* en mémoire) où le transfert doit commencer.
Et bien c'est justement ce qu'attend OpenGL maintenant. Nous lui donnerons donc l'*offset* correspondant aux tableaux de données dans la mémoire vidéo.

Pour les vertices, cet offset sera de **0** car ils se situent au début de la zone mémoire.

Pour les couleurs, cet offset sera égal à l'attribut **m_tailleVerticesBytes** car elles sont situées juste après les vertices dans la zone mémoire.

L'appel à la fonction **glVertexAttribPointer()** devient donc :

Code : C++

```
// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

// Accès aux vertices dans la mémoire vidéo
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(0);

// Accès aux couleurs dans la mémoire vidéo
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
m_tailleVerticesBytes);
 glEnableVertexAttribArray(1);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Au niveau de la méthode **afficher()**, ça donnerait ceci :

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    // Accès aux couleurs dans la mémoire vidéo
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
m_tailleVerticesBytes);
    glEnableVertexAttribArray(1);
```

```
// Déverrouillage du VBO  
glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
....  
  
// Désactivation du shader  
glUseProgram(0);  
}
```

Techniquement, on ne peut pas compiler ce code car le paramètre **pointer** attend un décalage (représenté par une adresse et non une variable). Or nous, nous lui donnons une variable de type **integer**. Alors bon, pour le **0** ça passe encore mais pour **tailleVerticesBytes** notre compilateur va nous râler dessus. 😞

Pour utiliser les variables **tailleXXX** en décalage, OpenGL nous demande de les encadrer avec la macro suivante :

Code : C++

```
// Macro utile au VBO  
#ifndef BUFFER_OFFSET  
#define BUFFER_OFFSET(offset) ((char*)NULL + (offset))  
#endif
```

Ce code permet de spécifier un décalage **offset** en partant de l'adresse **NULL** (ou **0**) qui indique le début du **VBO**. Le mot-clé **char** permet simplement de faire le lien avec les bytes, je vous rappelle que les variables de type **char** sont codées sur **1** byte.



 Pensez à inclure ce code dans chaque fichier où vous utiliserez les **VBO**.

En utilisant cette macro, le compilateur comprend bien qu'on veut lui envoyer une adresse à l'aide d'une variable. Nous encadrions donc le paramètre **pointer** par **BUFFER_OFFSET()** :

Code : C++

```
// Verrouillage du VBO  
glBindBuffer(GL_ARRAY_BUFFER, m_vboID);  
  
// Accès aux vertices dans la mémoire vidéo  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,  
BUFFER_OFFSET(0));  
 glEnableVertexAttribArray(0);  
  
// Accès aux couleurs dans la mémoire vidéo  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,  
BUFFER_OFFSET(m_tailleVerticesBytes));  
 glEnableVertexAttribArray(1);
```

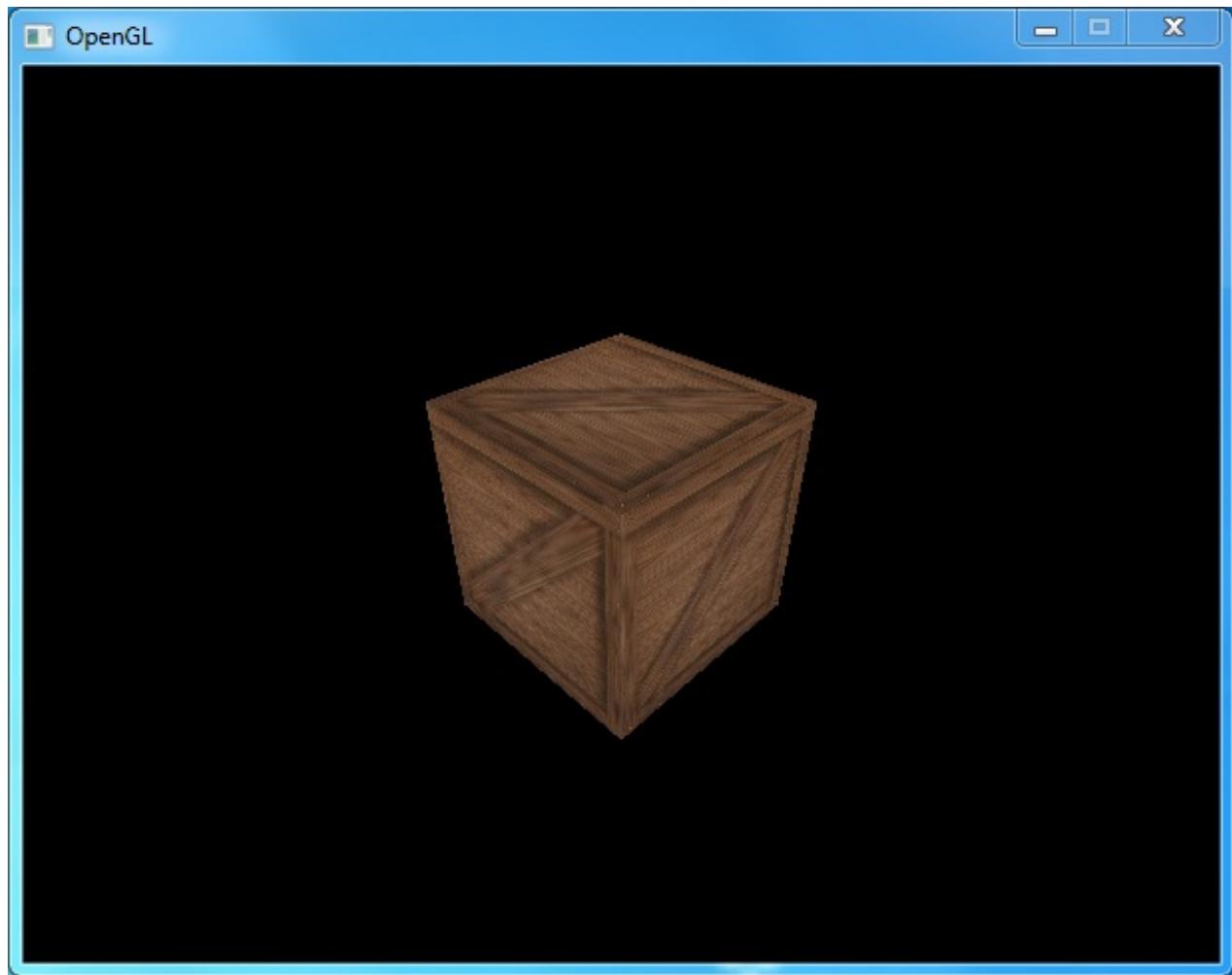
```
// Déverrouillage du VBO  
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Petit point important avant de compiler tout ça, faites attention à bien appeler la méthode **charger()** au moment de créer votre cube. Si vous ne le faites pas, votre application va joliment planter devant vos yeux ébahies. 😊

Code : C++

```
// Déclaration d'un objet Cube  
Cube cube(2.0, "Shaders/couleur3D.vert", "Shaders/couleur3D.frag");  
cube.charger();
```

Cette fois, vous pouvez compiler tranquillement.



Vous avez le même rendu qu'avant sauf que maintenant OpenGL ne fait plus d'aller-retours pour chercher les données dans la RAM. Tout se trouve déjà dans la carte graphique et ça lui fait gagner du temps. 😊

Utilisation de plusieurs VBO

On termine cette sous-partie avec une petite problématique ? Comment je fais si je veux utiliser plusieurs **VBO** pour afficher quelque chose ?

En temps normal, je vous dirais qu'il est inutile d'en utiliser plusieurs pour afficher un unique modèle, un seul fait parfaitement l'affaire. Cependant, il existe des cas où il est utile de séparer les données dans plusieurs **VBO**, en général ce sera pour faire de l'optimisation.

Si ça vous arrive un jour, sachez qu'il suffit simplement de verrouiller le premier **VBO** avant votre premier envoi de données, puis de verrouiller le deuxième avec le deuxième envoi, et ainsi de suite :

Code : C++

```
// Accès au premier VBO
glBindBuffer(GL_ARRAY_BUFFER, premierVBO);

// Accès à la mémoire vidéo
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
glEnableVertexAttribArray(0);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Accès au deuxième VBO
glBindBuffer(GL_ARRAY_BUFFER, deuxiemeVBO);

// Accès à la mémoire vidéo
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(m_tailleVerticesBytes));
glEnableVertexAttribArray(2);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Petit info au passage, si vous enchainez **plusieurs verrouillages** sur des **objets OpenGL de même type** (et j'insiste bien dessus) alors vous n'êtes pas obligés de tous les déverrouiller à chaque fois, ils le seront automatiquement. Seul le **dernier** déverrouillage est important.

On peut donc enlever le déverrouillage du premier objet dans ce cas :

Code : C++

```
// Accès au premier VBO
glBindBuffer(GL_ARRAY_BUFFER, premierVBO);

// Accès à la mémoire vidéo
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
glEnableVertexAttribArray(0);

// Accès au deuxième VBO
```

```
glBindBuffer(GL_ARRAY_BUFFER, deuxiemeVBO);

// Accès à la mémoire vidéo

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(m_tailleVerticesBytes));
glEnableVertexAttribArray(2);

// Déverrouillage du VBO

glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Mise à jour des données La méthode updateVBO()

Pourquoi mettre à jour les données ?

Transférer les données dans la carte graphique c'est bien, mais les mettre à jour quand un objet bouge c'est quand même mieux.



Notre caisse n'en a pas vraiment besoin car elle ne bouge pas. Mais lorsque nous animons des personnages, il faudra bien mettre à jour leurs vertices pour voir leurs déplacements.

La modification de données dans la mémoire vidéo se fait un peu comme l'écriture de texte dans un fichier.

Avec les fichiers, on commence par en ouvrir un avec un mode d'accès (lecture, écriture ou les deux), puis on écrit les données, et une fois qu'on a fini on ferme le fichier.

Avec les **VBO** c'est un peu la même chose, on va commencer par récupérer l'espace mémoire qu'on a allouée, puis on écrira les données à l'intérieur et enfin on fermera l'accès à la zone mémoire.

La seule véritable différence avec les fichiers c'est que les **VBO** il faut les verrouiller. 😊

Pour concrétiser ce qu'on va voir sur ça, nous allons coder une méthode qui nous permettra de mettre à jour nos données. Celle-ci s'appellera **updateVBO()** :

Code : C++

```
void updateVBO(void *donnees, int tailleBytes, int decalage);
```

- **donnees** : Un pointeur sur les données à envoyer, comme un tableau par exemple. Il est de type **void*** car on ne connaîtra pas forcément le type de donnée que l'on enverra
- **tailleBytes** : La taille en bytes des données
- **decalage** : Le décalage en mémoire où commencer la copie, nous allons voir ça un peu plus bas

Le code

Pour accéder à la mémoire vidéo, nous devons utiliser une fonction qui permet de retourner un pointeur un peu spéciale. Celui-ci forme en quelque sorte une *passerelle* entre le **VBO** et la **RAM**. Grâce à lui, nous pourrons accéder aux données de la mémoire vidéo comme s'il s'agissait de données présentes dans la **RAM**.

La fonction en question s'appelle **glMapBuffer()**.

Code : C++

```
void* glMapBuffer(GLenum target, GLenum access);
```

- **target** : Toujours le même, on lui donne la valeur **GL_ARRAY_BUFFER**
- **mode** : Mode d'accès aux données du **VBO** (lecture, écriture ou les deux)

Contrairement à ce qu'on pourrait penser, la fonction renvoie bien quelque chose. L'utilisation du type **void*** permet de spécifier un pointeur. Un pointeur qui, ici, forme la passerelle dont nous avons parlée à l'instant.

Le paramètre **mode** peut prendre 3 valeurs :

- **GL_READ_ONLY** : Lecture seulement
- **GL_WRITE_ONLY** : Écriture seulement
- **GL_READ_WRITE** : Lecture et écriture

Dans la plupart des cas, nous utiliserons la deuxième valeur car on ne fera que transférer des données, pas besoin de lecture dans ce cas. 😊

Ainsi, pour mettre à jour les informations contenues dans un **VBO** on commence par le verrouiller puis on récupère l'adresse de sa zone mémoire :

Code : C++

```
void Cube::updateVBO(void *donnees, int tailleBytes, int decalage)
{
    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Récupération de l'adresse du VBO
    void *adresseVBO = glMapBuffer(GL_ARRAY_BUFFER,
        GL_WRITE_ONLY);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Il peut arriver que la récupération de l'adresse échoue, ce n'est pas très courant mais ça peut arriver.

Si c'est le cas alors la fonction **glMapBuffer()** renvoie un pointeur **NULL**, il faut donc vérifier sa valeur avant de continuer. Si elle est nulle, alors on déverrouille le **VBO** et on annule le transfert. S'il n'y a aucune erreur on peut continuer :

Code : C++

```
void Cube::updateVBO(void *donnees, int tailleBytes, int decalage)
{
    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Récupération de l'adresse du VBO
    void *adresseVBO = glMapBuffer(GL_ARRAY_BUFFER,
        GL_WRITE_ONLY);

    // Si l'adresse retournée est nulle alors on arrête le
    // transfert
}
```

```

transfert

    if(adresseVBO == NULL)
    {
        std::cout << "Erreur au niveau de la récupération du
VBO" << std::endl;
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        return;
    }

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

```

Une fois notre adresse trouvée et validée nous pouvons commencer le transfert, et comme d'habitude avec les **VBO** les transferts se font en bytes. Le seul problème ici, c'est qu'on n'a pas de fonction OpenGL pour mettre à jour nos données, il va falloir le faire par nous-même avec comme seul outil l'adresse de destination dans la mémoire vidéo.

Heureusement pour nous, nous n'aurons pas à faire ça à la main. Il existe une fonction **C** (oui oui **C** pas **C++**) qui permet de copier tout un tas de bytes d'un coup. Cette fonction s'appelle **memcpy()** :

Code : C++

```
void* memcpy(void *destination, const void *source, size_t num);
```

- **destination** : L'adresse où on écrira les données, ici ce sera l'adresse mémoire que l'on a récupérée
- **source** : La source de données, ici ce sera les tableaux de vertices et de coordonnées de texture
- **num** : La taille des données à copier (en bytes), c'est la même chose que le paramètre **size** de la fonction **glBufferSubData()**



Vous devez inclure l'en-tête **Cstring** et je dis bien **Cstring** pour pouvoir utiliser cette fonction.

Le seul problème avec cette fonction, c'est qu'elle ne prend pas en compte le *décalage* dans une zone mémoire, elle n'est capable d'écrire qu'à partir du début.

Pour contourner ceci, nous allons reprendre la macro **BUFFER_OFFSET()** que nous utilisons pour l'affichage :

Code : C++

```

// Macro utile au VBO

#ifndef BUFFER_OFFSET
#define BUFFER_OFFSET(offset) ((char*)NULL + (offset))
#endif

```

Le code qui nous intéresse ici c'est la définition de la macro :

Code : C++

```
(char*)NULL + (offset)
```

Ce code permet de spécifier un décalage *offset* à partir de l'adresse **NULL**. Ce que nous voulons nous, c'est spécifier un décalage à partir de l'adresse **adresseVBO**.

Pour ce faire, nous devons simplement remplacer le mot-clé **NULL** par le pointeur **adresseVBO** et l'*offset* par la variable **decalage** :

Code : C++

```
(char*) adresseVBO + decalage
```

Simple n'est-ce pas ? 😊

Alors bon, nous n'allons pas recréer une macro pour ça, ce ne serait pas utile et ça alourdirait le header. A la place, nous allons juste donner ce code au paramètre **destination** de la fonction **memcpy()**. Quant aux autres, nous leur donnerons en valeur la taille des données à copier et le pointeur source.

L'appel à la fonction ressemblera au final à ceci :

Code : C++

```
// Mise à jour des données  
memcpy( (char*) adresseVBO + decalage, donnees, tailleBytes);
```

Avec ça, nous pouvons mettre à jour n'importe quel **VBO**.

Il ne manque plus qu'une seule chose à faire. Pour sécuriser le **VBO**, il faut invalider le pointeur retourné par **glMapBuffer()**. Si on ne le fait pas il y a un risque d'écraser les données présentes à l'intérieur, on peut se retrouver avec un magnifique bug d'affichage avec ça. 😱

La fonction permettant d'invalider ce pointeur s'appelle **glUnmapBuffer()** :

Code : C++

```
GLboolean glUnmapBuffer(GLenum target);
```

- **target** : Encore celui-là. 🍺 On lui donnera comme d'hab la valeur **GL_ARRAY_BUFFER**

La fonction renvoie un **GLboolean** pour savoir si tout s'est bien passé.

On l'appelle juste après les transferts précédents. On en profite au passage pour affecter la valeur **0** au pointeur pour une double sécurisation :

Code : C++

```
// Annulation du pointeur  
glUnmapBuffer(GL_ARRAY_BUFFER);  
adresseVBO = 0;
```

Si on récapitule tout ça, on a un beau code de mise à jour de **VBO** :

Code : C++

```

void Cube::updateVBO(void *donnees, int tailleBytes, int decalage)
{
    // Verrouillage du VBO

    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Récupération de l'adresse du VBO

    void *adresseVBO = glMapBuffer(GL_ARRAY_BUFFER,
GL_WRITE_ONLY);

    // Si l'adresse retornnée est nulle alors on arrête le
transfert

    if(adresseVBO == NULL)
    {
        std::cout << "Erreur au niveau de la récupération du
VBO" << std::endl;
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        return;
    }

    // Mise à jour des données

    memcpy((char*)adresseVBO + decalage, donnees, tailleBytes);

    // Annulation du pointeur

    glUnmapBuffer(GL_ARRAY_BUFFER);
    adresseVBO = 0;

    // Déverrouillage du VBO

    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

```

L'avantage avec cette méthode, c'est qu'elle n'aura pas besoin d'être réécrite dans les classes filles, elle est universelle. 😊

D'ailleurs, comme promis au début du chapitre, nous allons maintenant passer à la gestion du VBO dans la classe fille *Caisse*.

Un autre exemple

Le header

Comme promis, nous nous occupons maintenant de l'implémentation des **VBO** dans la classe *Caisse*. Il n'y aura rien de nouveau à apprendre, nous connaissons déjà tout. 🍻 Le seul point auquel il faudra faire attention est le fait qu'elle hérite déjà des attributs de sa classe mère.

Ainsi, nous ne devrons pas redéclarer une variable **GLuint** pour le **VBO** car nous héritons naturellement de celui de la classe *Cube*. Même chose pour la taille du tableau de vertices en bytes.

En revanche, nous devrons déclarer une variable **m_tailleCoordTextureBytes** car nous en aurons besoin pour définir la taille de la mémoire à allouer. En effet, nous n'enverrons pas des couleurs mais des coordonnées de texture, il faudra donc utiliser une autre variable. Nous devrons par ailleurs redéfinir la méthode **charger()** pour les envoyer à la place des couleurs.

Au final, on commence par déclarer, dans le header de la classe *Caisse*, une variable **m_tailleCoordTextureBytes** de type **int** :

Code : C++

```
#ifndef DEF_CAISSA
#define DEF_CAISSA

// Includes
...

// Classe Caisse

class Caisse : public Cube
{
public:
    Caisse(float taille, std::string const vertexShader, std::string const fragmentShader, std::string const texture);
    ~Caisse();

    void afficher(glm::mat4 &projection, glm::mat4 &modelview);

private:
    Texture m_texture;
    float m_coordTexture[72];
    int m_tailleCoordTextureBytes;
};

#endif
```

Une fois déclarée, nous l'initialisons dans le constructeur en lui donnant la taille du tableau de coordonnées de texture. Celui-ci fait **72** cases donc sa taille bytes fera **72 * sizeof(float)** :

Code : C++

```
Caisse::Caisse(float taille, std::string const
vertexShader, std::string const fragmentShader, std::string const
texture) : Cube(taille, vertexShader, fragmentShader),
m_texture(texture),
m_tailleCoordTextureBytes(72 * sizeof(float))
{
    // Initialisation
    ...
}
```

La méthode charger()

La méthode **charger()** ne va pas beaucoup changer par rapport à son parent. En fait, nous avons juste à modifier les attributs utilisés pour définir la taille du **VBO**. Son prototype sera aussi le même :

Code : C++

```
void charger();
```

Pour son implémentation, nous n'allons pas nous prendre la tête et recopier simplement tout le contenu de la méthode **charger()** précédente. 🤪 Il faudra évidemment faire des petites modifications ensuite mais au moins, nous aurons déjà le plus gros du code.

En copiant ce contenu, on se retrouve avec la méthode suivante :

Code : C++

```
void Caisse::charger()
{
    // Destruction d'un éventuel ancien VBO

    if(glIsBuffer(m_vboID) == GL_TRUE)
        glDeleteBuffers(1, &m_vboID);

    // Génération de l'ID
    glGenBuffers(1, &m_vboID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Allocation de la mémoire vidéo
    glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +
    m_tailleCouleursBytes, 0, GL_STATIC_DRAW);

    // Transfert des données
    glBufferSubData(GL_ARRAY_BUFFER, 0, m_tailleVerticesBytes,
    m_vertices);
    glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,
    m_tailleCouleursBytes, m_couleurs);

    // Déverrouillage de l'objet
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Ce qu'il faut modifier ici ce sont les appels aux fonctions **glBufferData()** et **glBufferSubData()**.

Nous savons que la première des deux permet d'allouer une zone mémoire dans la carte graphique pour le **VBO**. Elle a besoin de connaître la taille de la mémoire qu'elle doit allouer. Nous devons, ici, lui donner le résultat de l'addition de la taille des vertices et des coordonnées de texture. Nous utiliserons pour cela les attributs :

- **m_tailleVerticesBytes** (qui est héritée)
- **m_tailleCoordTextureBytes** (que nous avons créée il y a quelques instants)

Les autres paramètres eux ne changent pas :

Code : C++

```
// Allocation de la mémoire vidéo
```

```
glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +  
m_tailleCoordTextureBytes, 0, GL_STATIC_DRAW);
```

La fonction **glBufferSubData()**, quant à elle, permet de les remplir le **VBO** avec les données que l'on veut envoyer. Nous l'appelons deux de façon à envoyer les vertices d'une part et les coordonnées de texture de l'autre.

Code : C++

```
// Transfert des coordonnées de texture  
  
glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,  
m_tailleCoordTextureBytes, m_coordTexture);
```

Vu que les coordonnées de texture se trouvent juste après les vertices dans la mémoire, nous devons donc laisser le paramètre **offset** (le deuxième de la fonction) à la case **m_tailleVerticesBytes**.

La méthode **charger()** ressemble au final à ceci :

Code : C++

```
void Caisse::charger()  
{  
    // Destruction d'un éventuel ancien VBO  
  
    if(glIsBuffer(m_vboID) == GL_TRUE)  
        glDeleteBuffers(1, &m_vboID);  
  
    // Génération de l'ID  
    glGenBuffers(1, &m_vboID);  
  
    // Verrouillage du VBO  
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);  
  
    // Allocation de la mémoire vidéo  
  
    glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes +  
m_tailleCoordTextureBytes, 0, GL_STATIC_DRAW);  
  
    // Transfert des données  
  
    glBufferSubData(GL_ARRAY_BUFFER, 0, m_tailleVerticesBytes,  
m_vertices);  
    glBufferSubData(GL_ARRAY_BUFFER, m_tailleVerticesBytes,  
m_tailleCoordTextureBytes, m_coordTexture);  
  
    // Déverrouillage de l'objet  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
}
```

N'oubliez pas d'appeler cette méthode lorsque vous souhaitez créer une caisse :

Code : C++

```
// Objet Caisse

Caisse caisse(2.0, "Shaders/texture.vert", "Shaders/texture.frag",
"Textures/Caisse2.jpg");
caisse.charger();
```

La méthode afficher()

La méthode **afficher()** est plus simple à modifier que la précédente car il suffit juste de verrouiller le **VBO** et utiliser la macro **BUFFER_OFFSET()**.

On commence donc par encadrer le code relatif aux tableaux **Vertex Attrib** par la fonction de verrouillage **glBindBuffer()** :

Code : C++

```
void Caisse::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
    m_vertices);
    glEnableVertexAttribArray(0);

    // Envoi des coordonnées de texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
    m_coordTexture);
    glEnableVertexAttribArray(2);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    ....

    // Désactivation du shader
    glUseProgram(0);
}
```

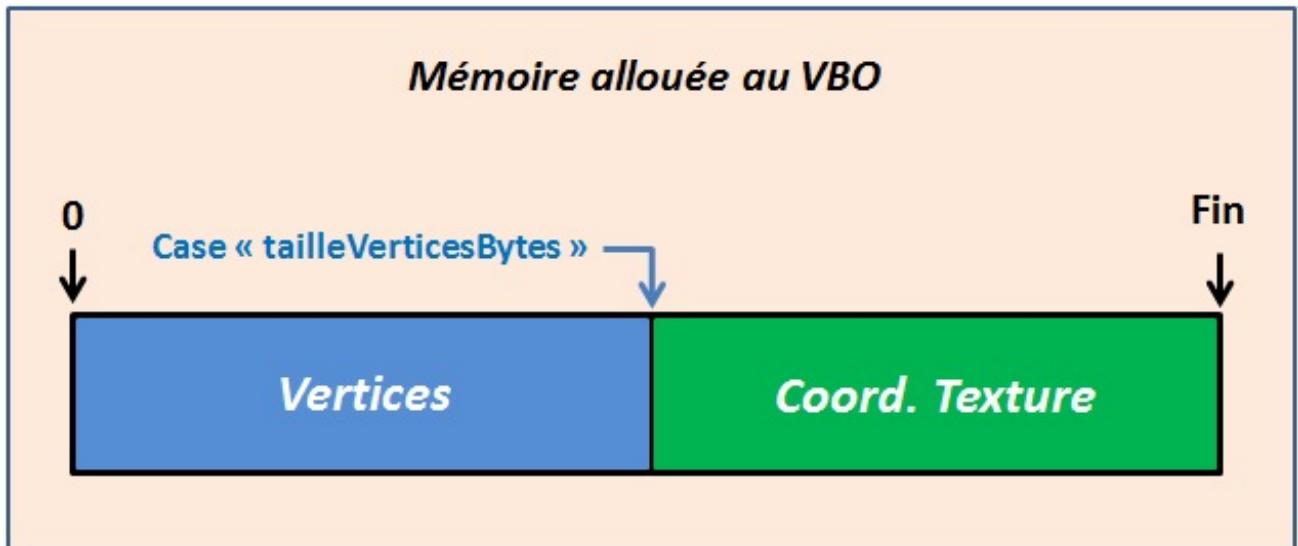
Ensuite, on modifie le paramètre **pointer** pour utiliser la macro avec le bon décalage. Pour l'envoi des vertices, on lui donnera la valeur **0** vu qu'ils se trouvent au début de la zone mémoire :

Code : C++

```
// Accès aux vertices dans la mémoire vidéo

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);
```

Les coordonnées de texture, quant à elles, sont situées après les vertices dans la mémoire. On peut représenter cette situation par un schéma :



Il faut donc utiliser la case "***tailleVerticesBytes***" représentée par l'attribut **m_tailleVerticesBytes** afin d'accéder aux coordonnées de texture. C'est donc cet attribut qu'il faut donner à la macro pour trouver le début des coordonnées de texture :

Code : C++

```
// Accès aux coordonnées de texture dans la mémoire vidéo

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(m_tailleVerticesBytes));
 glEnableVertexAttribArray(2);
```

Avec ces modifications, la méthode **afficher()** devient :

Code : C++

```
void Caisse::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Envoi des vertices
```

```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);

    // Envoi des coordonnées de texture

    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(m_tailleVerticesBytes));
    glEnableVertexAttribArray(2);

    // Déverrouillage du VBO

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));

    // Verrouillage de la texture

    glBindTexture(GL_TEXTURE_2D, m_texture.getID());

    // Rendu

    glDrawArrays(GL_TRIANGLES, 0, 36);

    // Déverrouillage de la texture

    glBindTexture(GL_TEXTURE_2D, 0);

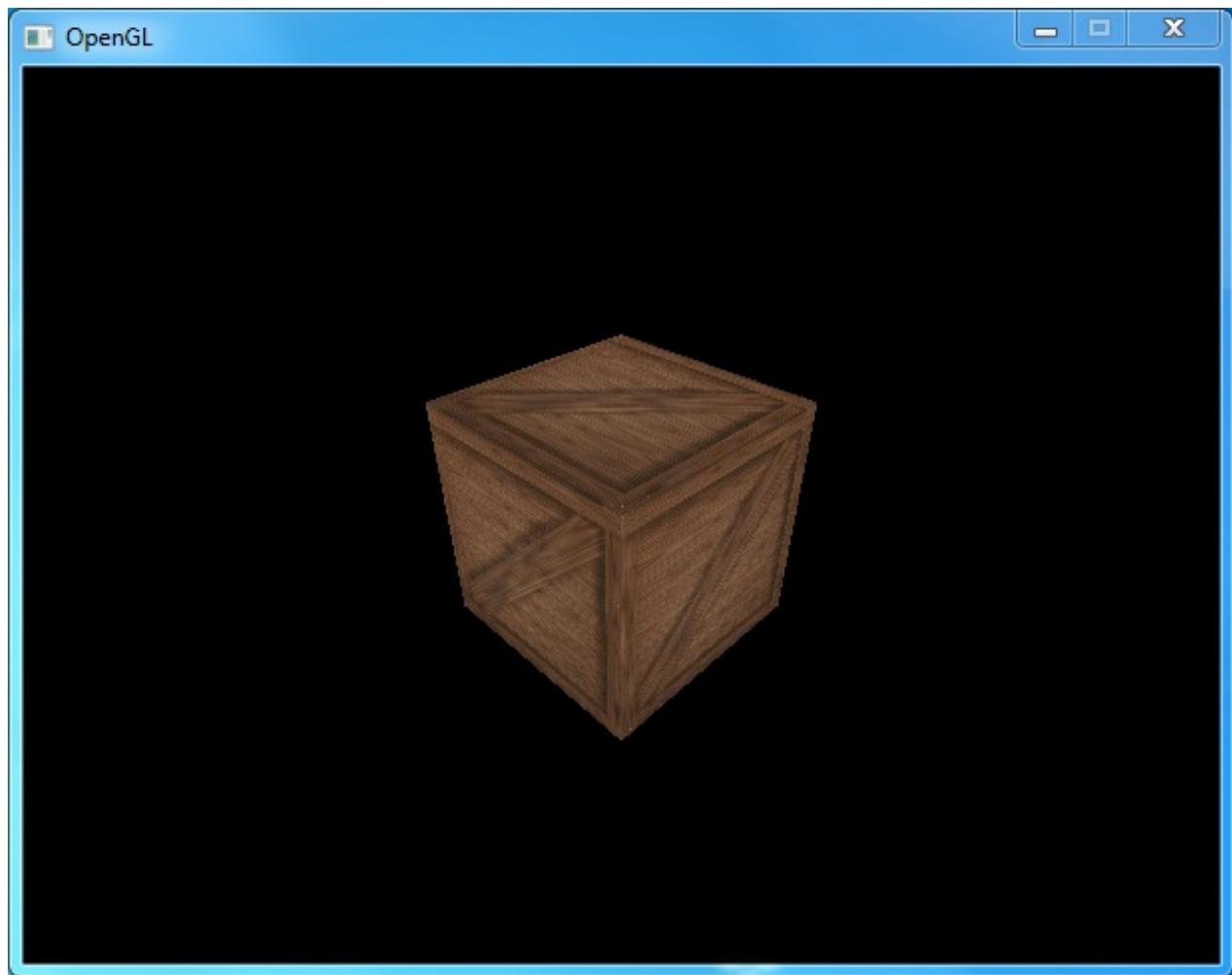
    // Désactivation des tableaux

    glDisableVertexAttribArray(2);
    glDisableVertexAttribArray(0);

    // Désactivation du shader

    glUseProgram(0);
}
```

Vous pouvez compiler pour admirer votre caisse chargée directement depuis votre carte graphique. 😊

**Télécharger : [Code Source C++ des Vertex Buffer Objects](#)**

Dans ce chapitre, nous avons vu ce que sont les **VBO** et la manière dont ils fonctionnent. Ils permettent d'augmenter la vitesse de nos applications en évitant à *OpenGL* des transferts de données inutiles. Ils sont très utilisés dans toutes les applications un tant soit peu développées, ce qui en font donc des incontournables dans la programmation 3D. 😊

A partir de maintenant, nous utiliserons toujours les **VBO** dans nos codes sources. Après tout, la deuxième partie du tuto a pour but de nous aider à maîtriser notre carte graphique. L'utilisation directe de la mémoire vidéo en est la première étape. 😊

Les Vertex Array Objects

Nous avons vu dans le chapitre précédent comment utiliser directement la mémoire de notre carte graphique en y stockant les données de nos modèles 3D. Aujourd'hui, nous allons continuer sur cette lancée en stockant autre chose dans cette mémoire, ce qui permettra de faire gagner encore plus de temps à OpenGL. 😊

Ne pensez pas que l'on maltraite notre carte graphique en la sollicitant ainsi, elle est justement faite pour ça et préfère même gérer ces informations elle-même plutôt que de devoir aller les chercher dans la RAM.

Ce chapitre sera très court car il se rapproche beaucoup du précédent, vous n'allez pas être dépayrés. 🍪

Encore un objet OpenGL ?

Les **Vertex Array Object** (ou VAO) sont des objets OpenGL relativement proches des **Vertex Buffer Object** (VBO) et font donc à peu près la même chose : stocker des informations sur la carte graphique.

Avec les **VBO**, nous facilitons la vie à *OpenGL* en lui évitant des transferts de données inutiles avec la RAM. Nous lui donnons toutes nos données une seule fois pendant le chargement et nous n'en parlons plus.

Avec les **VAO**, c'est un peu la même chose sauf que nous allons stocker des appels de fonctions.



Hein ? Ça veut dire qu'on va pouvoir stocker du code source dans la carte graphique ?

Alors oui et non. Nous n'allons pas lui envoyer directement du code, elle ne comprendrait pas et nous enverrait gentiment pâtre.



En fait, nous allons "*enregistrer*" certains appels de fonctions à l'intérieur de la carte graphique de façon à ce qu'*OpenGL* ne fasse pas des aller-retours répétitifs (à chaque frame donc 60 fois par seconde) avec l'application. Les appels concernés sont ceux qui s'occupent de l'envoi de données (vertices, ...):

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    .....

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);

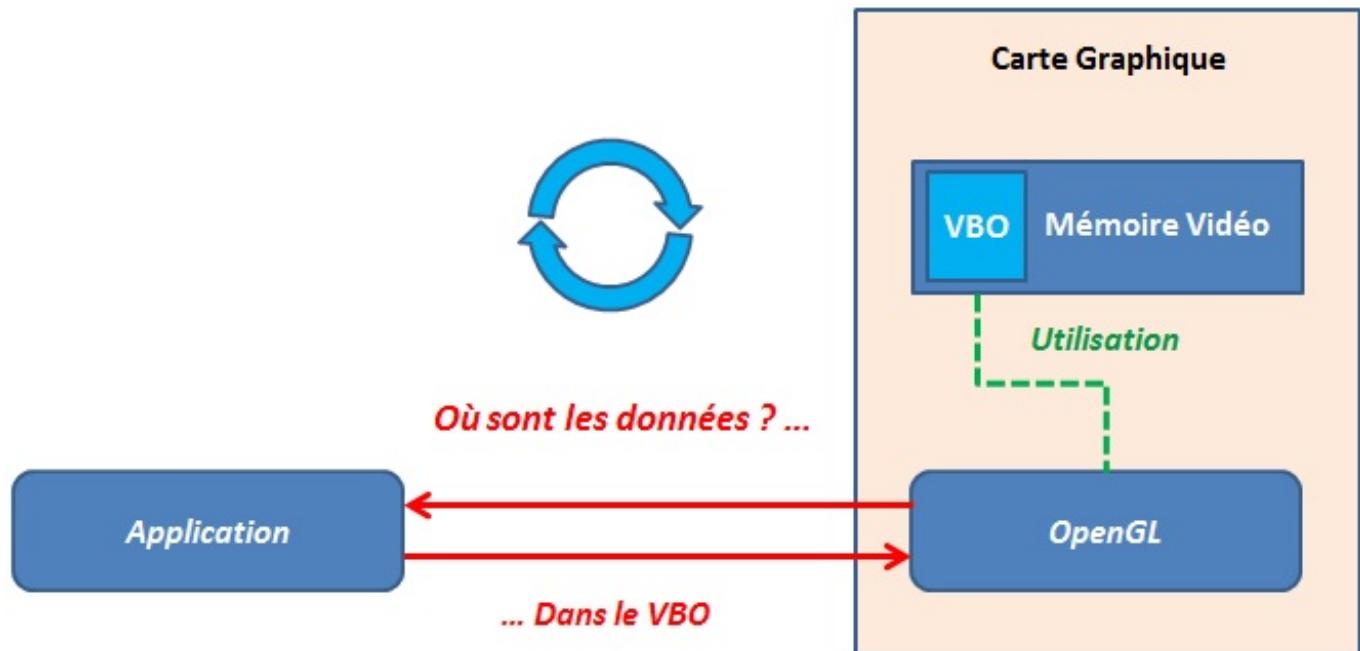
    // Accès aux couleurs dans la mémoire vidéo
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(m_tailleVerticesBytes));
    glEnableVertexAttribArray(1);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    .....
}
```

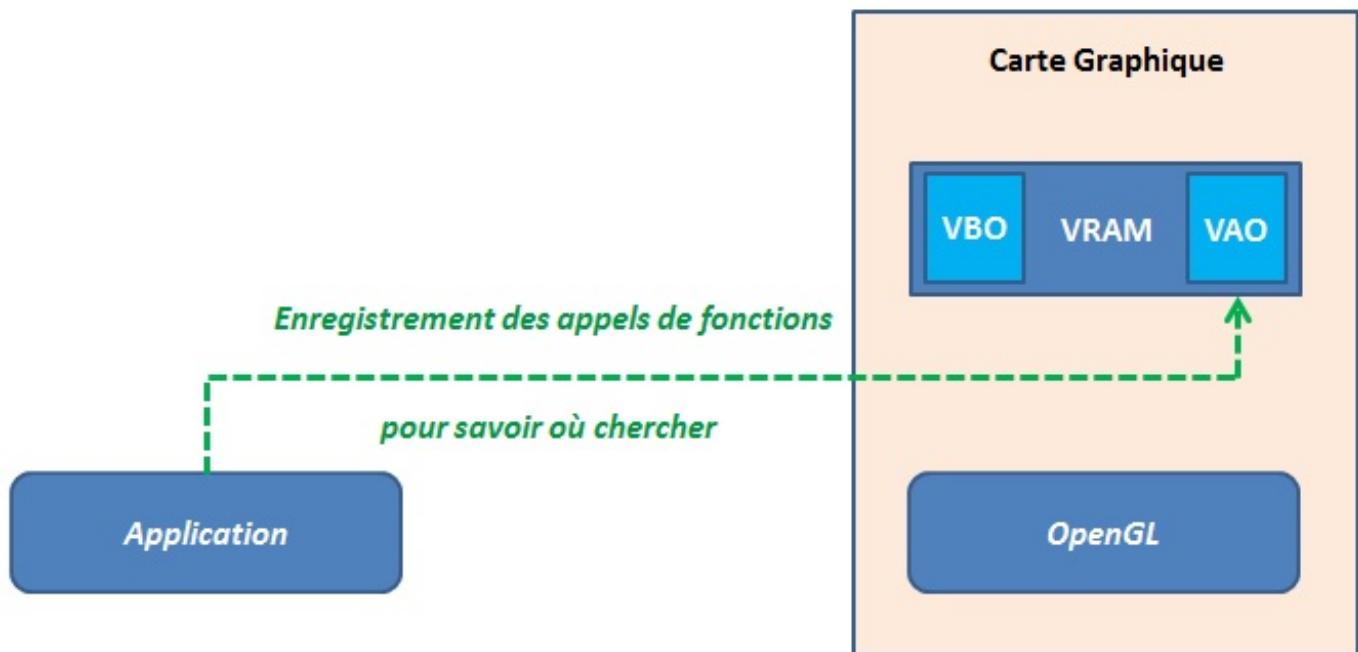
Malgré l'utilisation de VBO, OpenGL fait toujours quelques aller-retours entre la carte graphique et l'application pour savoir où chercher les données. Certes, il n'y a plus le transfert de millions de données mais il y existe quand même des appels répétitifs qui ne font que spécifier un emplacement dans la mémoire vidéo.

Boucle d'affichage

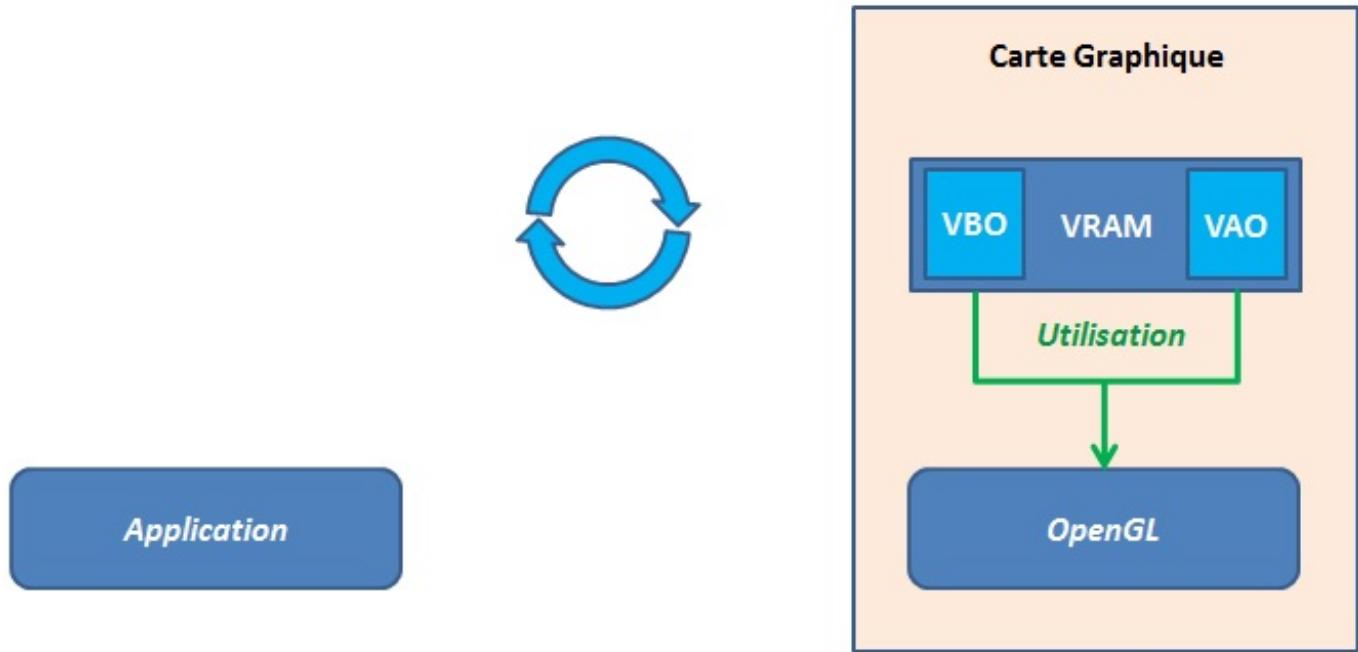


Pour éviter ça, OpenGL propose d'utiliser les **Vertex Array Objects** pour enregistrer un petit bout de code dans la carte graphique, ce qui lui permet d'aller chercher ses données dans la mémoire vidéo sans avoir à demander à l'application où elles se trouvent.

Chargement du modèle 3D



Boucle d'affichage



Grâce aux **VAO**, OpenGL sait exactement où aller chercher ses données sans avoir à le mander à personne. Tout se passe directement dans la carte graphique.

On ne sait pas vraiment ce qu'il se passe à l'intérieur des **VAO** et d'ailleurs on s'en moque on veut seulement que ça fonctionne,

peu importe comment. 😊

Ceux qui connaissent **OpenGL 2.1** feront sûrement l'analogie avec les **Display List**, on peut dire que les **VAO** sont leurs petits frères.

Pour vous donner une petite info, l'utilisation combinée des **VBO** et des **VAO** nous permettra de passer à **OpenGL 3.3**. Dans la dernière partie de ce chapitre, nous modifierons notre contexte pour profiter de la dernière version d'**OpenGL 3**. 😊

Les Mac-Users sont eux-aussi concernés car vous pourrez utiliser directement **OpenGL 3** sur votre Mac, vous n'aurez plus besoin de passer par un autre OS.

Création et Utilisation

Création

Le header

Contrairement aux **VBO**, les **VAO** seront très simples à configurer et à utiliser. Nous n'aurons pas besoin d'allouer de mémoire vidéo ni de faire des transferts de données, il suffira juste de faire des opérations que l'on connaît déjà.

Nous aurons tout d'abord besoin d'une variable de type **GLuint** pour représenter notre **VAO** au sein du code C++. Comme d'habitude, nous travaillerons avec la classe **Cube** (puis la classe **Caisse**) pour illustrer ce que nous verrons dans ce chapitre. Nous commençons donc pas déclarer un attribut de type **GLuint** dans la header, on l'appellera **m_vaoID** :

Code : C++

```
#ifndef DEF_CUBE
#define DEF_CUBE

// Includes
...

// Classe Cube
class Cube
{
public:
    Cube(float taille, std::string const vertexShader, std::string
const fragmentShader);
    ~Cube();

    void charger();
    void afficher(glm::mat4 &projection, glm::mat4 &modelview);

protected:
    Shader m_shader;
    float m_vertices[108];
    float m_couleurs[108];

    GLuint m_vboID;
    int m_tailleVerticesBytes;
    int m_tailleCouleursBytes;
    GLuint m_vaoID;
};

#endif
```

Bien entendu, on pense à l'initialiser dans le constructeur :

Code : C++

```
Cube::Cube(float taille, std::string const vertexShader, std::string const fragmentShader) : m_shader(vertexShader, fragmentShader),  
m_vboID(0),  
m_tailleVerticesBytes(108 * sizeof(float)),  
m_tailleCouleursBytes(108 * sizeof(float)), m_vaoID(0)  
{  
    // Initialisation  
  
    ...  
}
```

Chargement

Dans l'introduction, nous avons vu que les **VAO** permettent d'enregistrer certains appels de fonction dans la mémoire de la carte graphique. Ces appels concernent surtout l'affichage avec les tabl qui sont concernés. Les appels qui sont concernés sont ceux relatifs aux tableaux VertexAttrib. Ils vont donc disparaître de la méthode afficher() pour se retrouver dans la configuration des VAO. Nous verrons cela en détail dans un instant car avant, nous devons créer un objet OpenGL valide et le verrouiller pour le configurer.

Cela se fera dans la méthode **charger()** exactement comme le **VBO**. Nous placerons notre code après sa configuration.

On commence d'ailleurs avec la création du **VAO** grâce l'habituelle fonction de génération d'identifiant **glGenXXX()**. Elle s'appelle ici **glGenVertexArrays()** :

Code : C++

```
void glGenVertexArrays(GLsizei number, GLuint *arrays);
```

- **number** : Le nombre d'ID à initialiser, nous lui donnerons la valeur **1**
- **arrays** : Un tableau de **GLuint** ou l'adresse d'une seule variable du même type. Nous lui donnerons l'adresse du **VAO**

Nous devons l'appeler juste après la configuration du **VBO** avec les paramètres donnés ci-dessus :

Code : C++

```
void Cube::charger()  
{  
    // Création du VBO  
  
    ...  
  
    // Génération de l'ID du VAO  
  
    glGenVertexArrays(1, &m_vaoID);  
}
```

Comme toujours avec les *objets OpenGL*, nous devons les verrouiller lorsque nous voulons les configurer ou les utiliser. Tout ceux que nous vus jusqu'à maintenant (texture, **VBO**, etc.) nécessitaient cette opération, les **VAO** n'échappent pas à cette règle. Nous utiliserons pour cela la fonction **glBindVertexArray()** :

Code : C++

```
void glBindVertexArray(GLuint array);
```

- **array** : Le **VAO** à verrouiller

Remarquez que cette fonction ne prend pas de paramètre **target** contrairement aux **textures** et aux **VBO**. En temps normal, il y en a toujours un mais dans certains cas il arrive qu'il n'y en ait pas. 😊

On verrouille donc notre **VAO** en appelant la fonction **glBindVertexArray()**, on lui donnera en paramètre l'attribut **m_vaoID**. On en profitera au passage pour le déverrouiller immédiatement en utilisant la même fonction mais avec le paramètre **0** :

Code : C++

```
void Cube::charger()
{
    // Gestion du VBO
    ...

    // Génération de l'identifiant du VAO
    glGenVertexArrays(1, &m_vaoID);

    // Verrouillage du VAO
    glBindVertexArray(m_vaoID);

    // Vide pour le moment
    ...

    // Déverrouillage du VAO
    glBindVertexArray(0);
}
```

Notre **VAO** est maintenant prêt à être configuré.

Nous savons que celui-ci devra contenir le code relatif aux tableaux **Vertex Attrib**, il devra enregistrer tous les appels de fonctions qui y sont associés comme **glVertexAttribPointer()**, **glEnableVertexAttribArray()**, **glBindBuffer()**, etc.

Pour ce faire, il suffit simplement de couper le code d'envoi de données se trouvant dans la méthode **afficher()** et de le placer au moment de configurer le **VAO** (pendant le verrouillage).



Ça veut dire qu'on n'envoie plus les vertices et tout dans la méthode **afficher()** ?

Oui tout à fait. 😊

Le but est de sauvegarder ces appels dans la carte graphique, on les place donc dans le **VAO**. Nous devons donc les supprimer de la méthode **afficher()** pour les placer chez lui. Ne vous inquiétez pas cependant, nous ajouterons un petit bout de code dans cette méthode pour qu'OpenGL sache où chercher les données.

Au niveau de code, on coupe simplement le code suivant :

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    .....

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);

    // Accès aux couleurs dans la mémoire vidéo
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(m_tailleVerticesBytes));
    glEnableVertexAttribArray(1);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    .....
}
```

Que l'on colle ici (entre les lignes surlignées) :

Code : C++

```
void Cube::charger()
{
    // Gestion du VBO
    .....

    // Génération de l'identifiant du VAO
    glGenVertexArrays(1, &m_vaoID);

    // Verrouillage du VAO
    glBindVertexArray(m_vaoID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
glEnableVertexAttribArray(0);

// Accès aux couleurs dans la mémoire vidéo

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(m_tailleVerticesBytes));
glEnableVertexAttribArray(1);

// Déverrouillage du VBO

glBindBuffer(GL_ARRAY_BUFFER, 0);

// Déverrouillage du VAO

glBindVertexArray(0);
}
```

De cette façon, le processus d'envoi de données est enregistré sur la carte graphique et OpenGL n'a pas besoin d'interroger l'application C++ pour savoir ce qu'il doit faire.

Petit détail, je vous conseille d'indenter le code qui se trouve dans le **VAO** tout comme les shaders et les **VBO**. 😊



Attention cependant, nappelez pas d'autres fonctions à l'intérieur pour tenter de les enregistrer sur la carte graphique. Il n'y a que celles qui concernent les tableaux **Vertex Attrib** qui le peuvent, le reste ne fonctionnera pas.

Éviter les fuites de mémoire

Vous vous souvenez de ce que nous avons vu dans le chapitre précédent sur les fuites de mémoire des **VBO** ? Que si on en chargeait un deux fois alors le premier chargement était perdu et les ressources considérées comme étant "toujours utilisées".

Le problème est le même ici, si on charge un **VAO** plusieurs fois alors on gâche de la mémoire. Pour éviter cela, nous allons utiliser la même astuce que celle que nous avons utilisée pour les textures et les **VBO**, à savoir : appeler une fonction du type **glIsXXX()** pour savoir si une initialisation a déjà eu lieu, puis en appeler une autre du type **glDeleteXXX()** si c'est le cas.

La première de ces fonctions s'appelle dans notre cas :

Code : C++

```
GLboolean glIsVertexArray(GLuint array);
```

Elle prend en paramètre l'identifiant à vérifier et renvoie comme d'habitude **GL_TRUE** ou **GL_FALSE** pour confirmer ou non une précédente initialisation.

La fonction de destruction de **VAO** quant à elle s'appelle **glDeleteVertexArrays()** :

Code : C++

```
void glDeleteVertexArrays(GLsizei number, const GLuint *arrays);
```

- **number** : Le nombre d'ID à initialiser, nous lui donnerons la valeur **1**
- **arrays** : Un tableau de **GLuint** ou l'adresse d'une seule variable du même type. Nous lui donnerons l'adresse de l'attribut **m_vaoID**

Nous devons les appeler dans un bloc **if** juste avant la génération d'ID :

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Gestion du VBO
    ....
    // Destruction d'un éventuel ancien VAO
    if(glIsVertexArray(m_vaoID) == GL_TRUE)
        glDeleteVertexArrays(1, &m_vaoID);

    // Génération de l'identifiant du VAO
    glGenVertexArrays(1, &m_vaoID);

    // Verrouillage du VAO
    glBindVertexArray(m_vaoID);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

    // Accès aux vertices dans la mémoire vidéo
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);

    // Accès aux couleurs dans la mémoire vidéo
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(m_tailleVerticesBytes));
    glEnableVertexAttribArray(1);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Déverrouillage du VAO
    glBindVertexArray(0);
}
```

De cette façon, on évite tout éventuel chargement perdu dans la mémoire.

Le destructeur

Tout comme le **VBO**, nous devons détruire le **VAO** dans le destructeur (qui lui-même est appelé au moment de détruire un objet). Nous utiliserons une fois de plus la fonction `glDeleteVertexArrays()`, je pense que vous n'avez pas besoin de plus d'explication.



Code : C++

```
Cube::~Cube()
{
    // Destruction du VBO
    glDeleteBuffers(1, &m_vboID);

    // Destruction du VAO
    glDeleteVertexArrays(1, &m_vaoID);
}
```

Utilisation

L'utilisation des **VAO** est encore plus simple que leur création puisqu'il suffit juste de les verrouiller exactement comme si on utilisait une texture. C'est-à-dire que nous devons le verrouiller **avant** et le déverrouiller **après** la fonction `glDrawArrays()`. Bien évidemment, il faut supprimer toute référence aux tableaux **Vertex Attrib** maintenant qu'ils sont utilisés autre part :

Code : C++

```
void Cube::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Verrouillage du VAO
    glBindVertexArray(m_vaoID);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
        "projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
        "modelview"), 1, GL_FALSE, value_ptr(modelview));

    // Rendu
    glDrawArrays(GL_TRIANGLES, 0, 36);

    // Déverrouillage du VAO
    glBindVertexArray(0);

    // Désactivation du shader
}
```

```
    glUseProgram(0);  
}
```

Grâce à ce verrouillage, OpenGL saura automatiquement où se trouveront les données dont il a besoin.

Petit détail : vous n'êtes pas obligés d'insérer l'envoi des matrices pendant le verrouillage du **VAO**, ça fonctionnera très bien si vous le faites avant. Personnellement, j'aime bien tout mettre à l'intérieur (textures comprises). 😊



Tout code faisant référence aux **Vertex Attrib** dans la méthode **afficher()** doit totalement disparaître et doit maintenant se trouver dans la méthode **charger()**.



Au fait, on ne déverrouille pas les tableaux **Vertex Attrib** ? On les a activés pendant la configuration du **VAO** mais on ne les désactive à aucun moment.

C'est une question que je me suis aussi posée la première fois. Apparemment, les appels aux fonctions **glDisableVertexAttribArray()** sont inutiles avec les **VAO**, ils se désactivent en même temps que le déverrouillage du **VAO**. Pas besoin de se triturer la tête avec ça.

Un autre exemple

Le header

Nous avons vu, dans la partie précédentes, comment implémenter un **VAO** dans la classe **Cube**, nous allons maintenant faire la même chose dans sa classe fille (**Caisse**). Ça va nous donner l'occasion de voir un autre exemple pour mieux intégrer les notions que nous avons abordées.

Avant de commencer, nous savons maintenant que nous avons besoin d'une variable de type **GLuint** pour gérer le **VAO**. Cependant, vu que la classe **Caisse** hérite de celui de la classe **Cube** alors il est inutile d'en utiliser une nouvelle. De plus, contrairement aux **VBO**, nous n'avons pas besoin d'attribut supplémentaire pour définir la taille des tableaux de données.

Il n'y a donc aucune modification à faire dans le header. 😊

La méthode **charger()**

Au niveau de la méthode **charger()**, on commence par générer un nouvel ID pour le **VAO** grâce à la fonction **glGenVertexArrays()**. On le verrouille ensuite à l'aide de **glBindVertexArray()** :

Code : C++

```
void Caisse::charger()  
{  
    // Gestion du VBO  
  
    ....  
  
    // Destruction d'un éventuel ancien VAO  
  
    if(glIsVertexArray(m_vaoID) == GL_TRUE)  
        glDeleteVertexArrays(1, &m_vaoID);  
  
    // Verrouillage du VAO  
  
    glBindVertexArray(m_vaoID);  
  
    // Vide pour le moment
```

```
    . . .  
  
    // Déverrouillage du VAO  
    glBindVertexArray(0);  
}
```

Il ne faut pas oublier de rajouter les fonctions qui permettent d'éviter les fuites de mémoire en détruisant un éventuel ancien VAO :

Code : C++

```
void Caisse::charger()  
{  
    // Gestion du VBO  
  
    . . .  
  
    // Destruction d'un éventuel ancien VAO  
  
    if(glIsVertexArray(m_vaoID) == GL_TRUE)  
        glDeleteVertexArrays(1, &m_vaoID);  
  
    // Génération de l'ID du VAO  
    glGenVertexArrays(1, &m_vaoID);  
  
    // Verrouillage du VAO  
    glBindVertexArray(m_vaoID);  
  
    // Vide pour le moment  
  
    . . .  
  
    // Déverrouillage du VAO  
    glBindVertexArray(0);  
}
```

Comme nous l'avons vu depuis le début du chapitre, le **VAO** doit contenir le code relatif aux tableaux **Vertex Attrib**. On coupe donc le code concerné dans la méthode **afficher()** pour le transférer ici. Ce qui donne la méthode **charger()** suivante :

Code : C++

```
void Caisse::charger()  
{  
    // Gestion du VBO  
  
    . . .  
  
    // Destruction d'un éventuel ancien VAO  
  
    if(glIsVertexArray(m_vaoID) == GL_TRUE)  
        glDeleteVertexArrays(1, &m_vaoID);
```

```
// Génération de l'ID du VAO
glGenVertexArrays(1, &m_vaoID);

// Verrouillage du VAO
 glBindVertexArray(m_vaoID);

// Verrouillage du VBO
 glBindBuffer(GL_ARRAY_BUFFER, m_vboID);

// Accès aux vertices dans la mémoire vidéo
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
 BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

// Accès aux coordonnées de texture dans la mémoire
// vidéo
 glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
 BUFFER_OFFSET(m_tailleVerticesBytes));
 glEnableVertexAttribArray(2);

// Déverrouillage du VBO
 glBindBuffer(GL_ARRAY_BUFFER, 0);

// Déverrouillage du VAO
 glBindVertexArray(0);
}
```

Configuration terminée. 😊

La méthode afficher()

Pour afficher notre caisse au final, il suffit simplement de remplacer ce que l'on a supprimé dans la méthode **afficher()** par le verrouillage du **VAO**. La seule différence avec la classe **Cube** c'est qu'ici on utilise ici une texture :

Code : C++

```
// Verrouillage du VAO
 glBindVertexArray(m_vaoID);

// Envoi des matrices
 glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
 "projection"), 1, GL_FALSE, value_ptr(projection));
 glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
 "modelview"), 1, GL_FALSE, value_ptr(modelview));
```

```
// Verrouillage de la texture
glBindTexture(GL_TEXTURE_2D, m_texture.getID());

// Rendu
glDrawArrays(GL_TRIANGLES, 0, 36);

// Déverrouillage de la texture
glBindTexture(GL_TEXTURE_2D, 0);

// Déverrouillage du VAO
glBindVertexArray(0);
```

Le code final de la méthode **afficher()** est maintenant :

Code : C++

```
void Caisse::afficher(glm::mat4 &projection, glm::mat4 &modelview)
{
    // Activation du shader
    glUseProgram(m_shader.getProgramID());

    // Verrouillage du VAO
    glBindVertexArray(m_vaoID);

    // Envoi des matrices

    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
    "projection"), 1, GL_FALSE, value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
    "modelview"), 1, GL_FALSE, value_ptr(modelview));

    // Verrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, m_texture.getID());

    // Rendu
    glDrawArrays(GL_TRIANGLES, 0, 36);

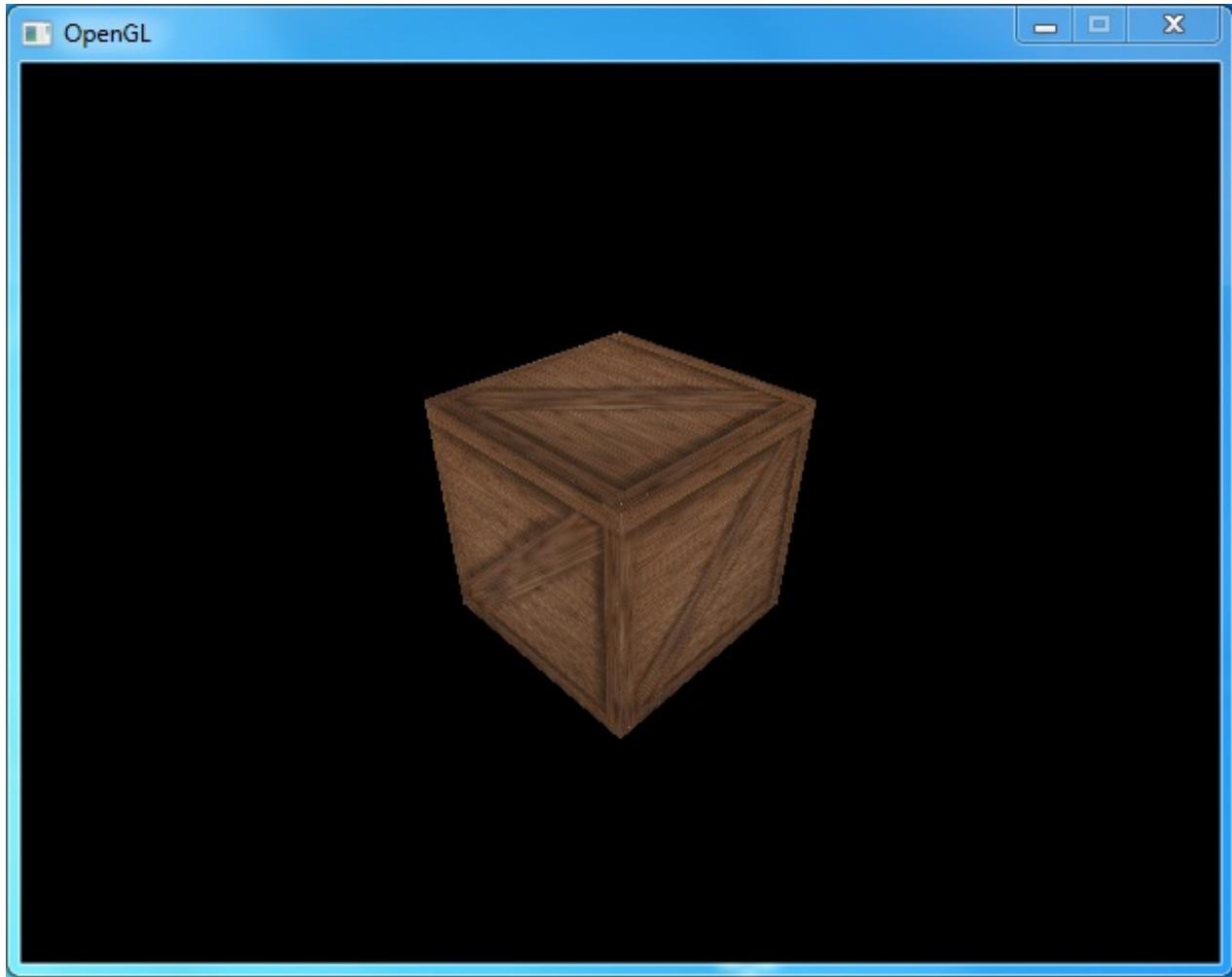
    // Déverrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, 0);

    // Déverrouillage du VAO
    glBindVertexArray(0);

    // Désactivation du shader
```

```
    glUseProgram(0);  
}
```

Vous pouvez maintenant compiler votre projet.



Ce chapitre est maintenant terminé. 🎉

Enfin juste en théorie, car nous allons voir maintenant comment passer à la version **3.3 d'OpenGL**. Comme je vous l'ai dit tout à l'heure, l'utilisation combinée des **VBO** et des **VAO** nous permettent de profiter pleinement d'**OpenGL3**. Les mac-users sont également concernés car vous pourrez maintenant coder vos applications directement sur votre Mac.

OpenGL 3.3 et Mac OS X

Passer à OpenGL 3.3

Mine de rien, nous avons fait pas mal de chemin depuis le début de ce tutoriel. Nous avons vu les principes de base d'OpenGL et nous venons de voir deux fonctionnalités avancées (ou plutôt **extensions**) exploitant directement notre carte graphique. Et croyez-moi ce n'est pas fini. 😊

Grâce aux deux derniers chapitres, nous connaissons tout ce qui est nécessaire pour mettre à jour notre version d'OpenGL. Vous n'êtes pas sans savoir que depuis le début du tutoriel, nous utilisons la version **3.1**. Cependant, je vous avais dit que nous passerions à **OpenGL3.3** dans le futur, et c'est justement ce que nous allons faire.

J'ai préféré utiliser la **3.1** jusqu'à maintenant car avec cette version, nous n'étions pas obligé d'utiliser les **VBO** et les **VAO** pour

chaque rendu. Imaginez si je vous avais montré ça dès le début, vous auriez fuis en courant. 😊
Mais maintenant que nous connaissons tout ça, nous pouvons enfin passer à la **3.3**.

Pour faire cela, la SDL va beaucoup nous aider puisqu'elle fera tout pour nous. Il suffira de lui demander la version qui nous intéresse. Vous vous souvenez du moment où nous initialisions la SDL avec des paramètres OpenGL ? Nous utilisons le code suivant dans la méthode **initialiserFenetre()** :

Code : C++

```
bool SceneOpenGL::initialiserFenetre()
{
    // Initialisation de la SDL

    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        std::cout << "Erreur lors de l'initialisation de la SDL : "
        << SDL_GetError() << std::endl;
        SDL_Quit();

        return false;
    }

    // Version d'OpenGL

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);

    // Double Buffer

    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);

    // Création de la fenêtre

    m_fenetre = SDL_CreateWindow(m_titreFenetre.c_str(),
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, m_largeurFenetre,
        m_hauteurFenetre, SDL_WINDOW_SHOWN | SDL_WINDOW_OPENGL);
    m_contexteOpenGL = SDL_GL_CreateContext(m_fenetre);

    return true;
}
```

Les lignes intéressantes dans ce code sont celles-ci :

Code : C++

```
// Version d'OpenGL

SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);
```

Elles permettent de demander à la SDL la version **3.1 d'OpenGL**. 😊

Pour passer à la **3.3**, il suffit de changer la valeur du paramètre **SDL_GL_CONTEXT_MINOR_VERSION** pour lui affecter la valeur 3 :

Code : C++

```
// Version d'OpenGL  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
```

Mais ce n'est pas tout. En effet, à partir d'**OpenGL 3.2** nous pouvons demander deux profils OpenGL : le profil de **compatibilité** (aussi appelé **Legacy** ou **Compatibility**) et le profil **Core**.

Le premier permet de conserver une compatibilité avec **OpenGL 2.1**. Il a été créé à l'origine pour ceux qui utilisent de gros moteurs 3D pour qu'ils n'aient pas à tout recoder.

Le second, lui, purge toutes les fonctionnalités inutiles pour ne garder que celles qui sont intéressantes. Il bénéficie en plus de pas mal de nouveautés spécifiques à **OpenGL 3** dont un nouveau pipeline plus souple et plus performant. Nous travaillerons toujours avec ce profil.

Pour le dire à la SDL, il suffit juste d'appeler la fonction **SDL_GL_SetAttribute()** avec en premier paramètre la valeur **SDL_GL_CONTEXT_PROFILE_MASK**. Le second quant à lui pourra prendre deux valeurs :

- **SDL_GL_CONTEXT_PROFILE_COMPATIBILITY** : Pour le profil de **Compatibilité**
- **SDL_GL_CONTEXT_PROFILE_CORE** : Pour le profil **Core**

Vous l'aurez compris, nous utiliserons la seconde valeur. 😊

Au final, l'appel à la fonction **SDL_GL_SetAttribute()** ressemble à ceci :

Code : C++

```
// Utilisation du profil Core  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,  
SDL_GL_CONTEXT_PROFILE_CORE);
```

Ce qui donne en somme les paramètres suivants :

Code : C++

```
// Version d'OpenGL  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);  
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,  
SDL_GL_CONTEXT_PROFILE_CORE);
```

Maintenant, nous sommes totalement à jour avec **OpenGL 3**. 😊

OpenGL 3 sous Mac OS X

Si vous n'êtes pas utilisateurs de Mac OS X, vous n'êtes pas obligés de lire cette partie mais je vous conseille quand même de le faire (au moins à partir du titre "**Version d'OpenGL**") car si vous voulez créer des jeux multiplateformes, il vous suffira juste d'inclure quelques lignes de code pour les rendre compatibles avec l'OS d'Apple.

Si vous êtes utilisateurs de Mac OS X vous avez dû vous sentir frustrés depuis le début du tutoriel vu qu'il était impossible d'utiliser **OpenGL 3**. 😊

Le problème vient du fait qu'Apple n'utilise que deux versions pour OpenGL : la **2.1** et la **3.2**, donc pas de **3.1**. De plus, pour profiter de la dernière version vous devez au moins être sous **OS X Lion (10.7)** vu qu'Apple ne soutient plus ses anciens OS.

Mais bon, si vous êtes au moins sous Lion et que vous avez suivi le tutoriel jusqu'ici, vous êtes donc capables d'utiliser **OpenGL3** directement sur votre Mac. Vous n'aurez pas la dernière version mais ce n'est pas grave, la plus importante étant la **3.2**. 😊

Je vais vous expliquer comment faire en utilisant **Xcode** qui est un IDE beaucoup plus adapté pour vous. Pour commencer, je vais vous demander de télécharger les deux DMG suivants qui contiennent respectivement les **Frameworks** de **SDL2** et de **SDL2_image** ainsi que la librairie **GLM**:

- [Télécharger : Framework **SDL2**](#)
- [Télécharger : Framework **SDL2_image**](#)
- [Télécharger : **GLM**](#)

Copiez les **Frameworks** se trouvant dans ces deux DMG dans le répertoire **/Library/Frameworks/** (ou **/Bibliothèque/Frameworks/**).

Copiez le dossier **glm** (en minuscules) se trouvant dans l'archive **OpenG Mathematics.zip** dans le répertoire **/usr/local/include**. Si vous ne voyez pas ce dossier, regardez dans le menu du **Finder** et cliquez sur "Aller" puis sur "Aller au dossier" et renseignez le chemin que je vous ai donné.

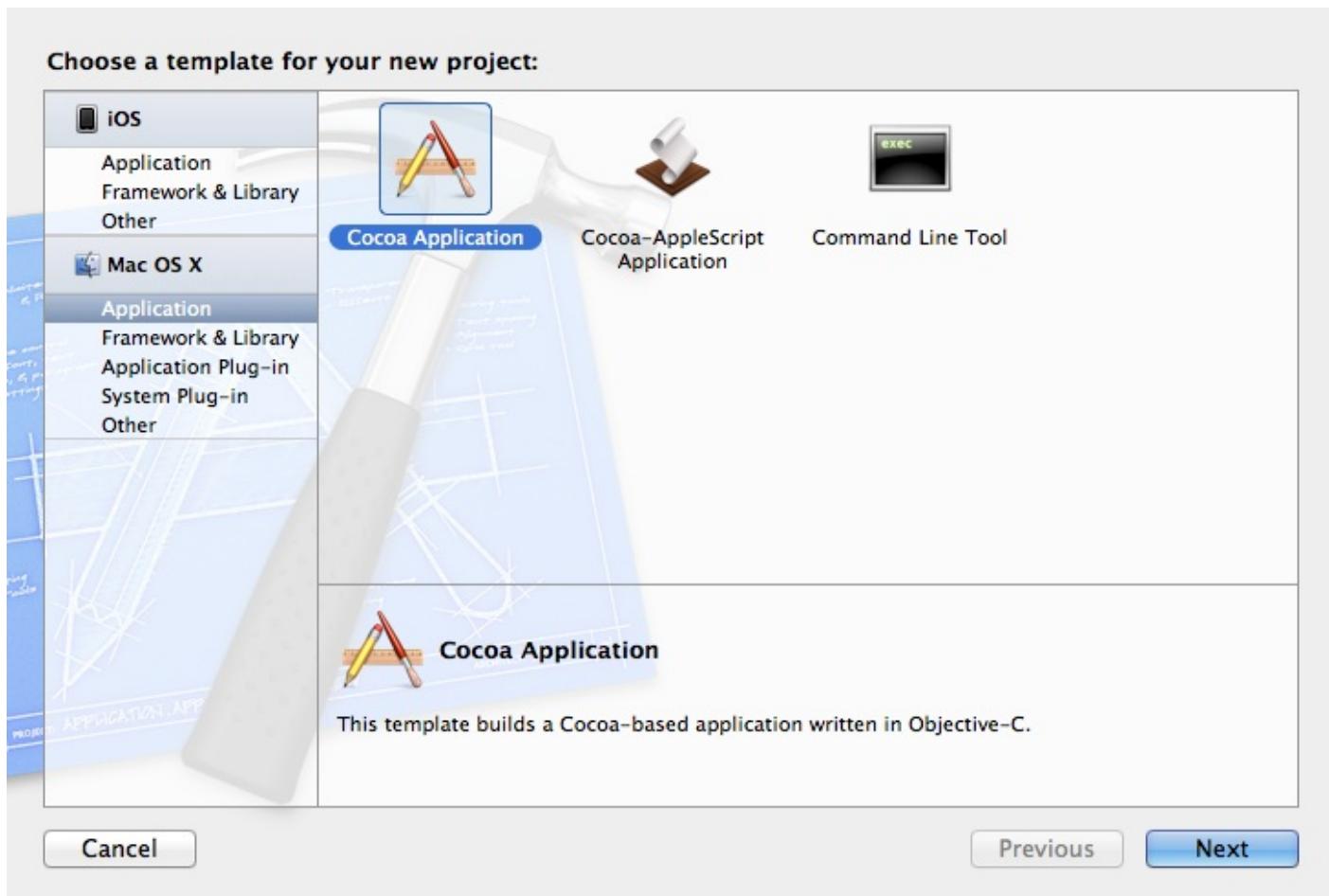
Créer un projet **SDL2** sous Mac OS X

Le projet

La création de projet **SDL2** avec Xcode est quelque chose d'assez folklo vu les changements fréquents de cet IDE. En effet, en temps normal vous devriez utiliser un **template** spécial qui crée votre projet **SDL2** automatiquement. Cependant, Apple change tellement le modèle de ses **templates** qu'il est devenu ingérable de fournir les fichiers nécessaires pour tout le monde.

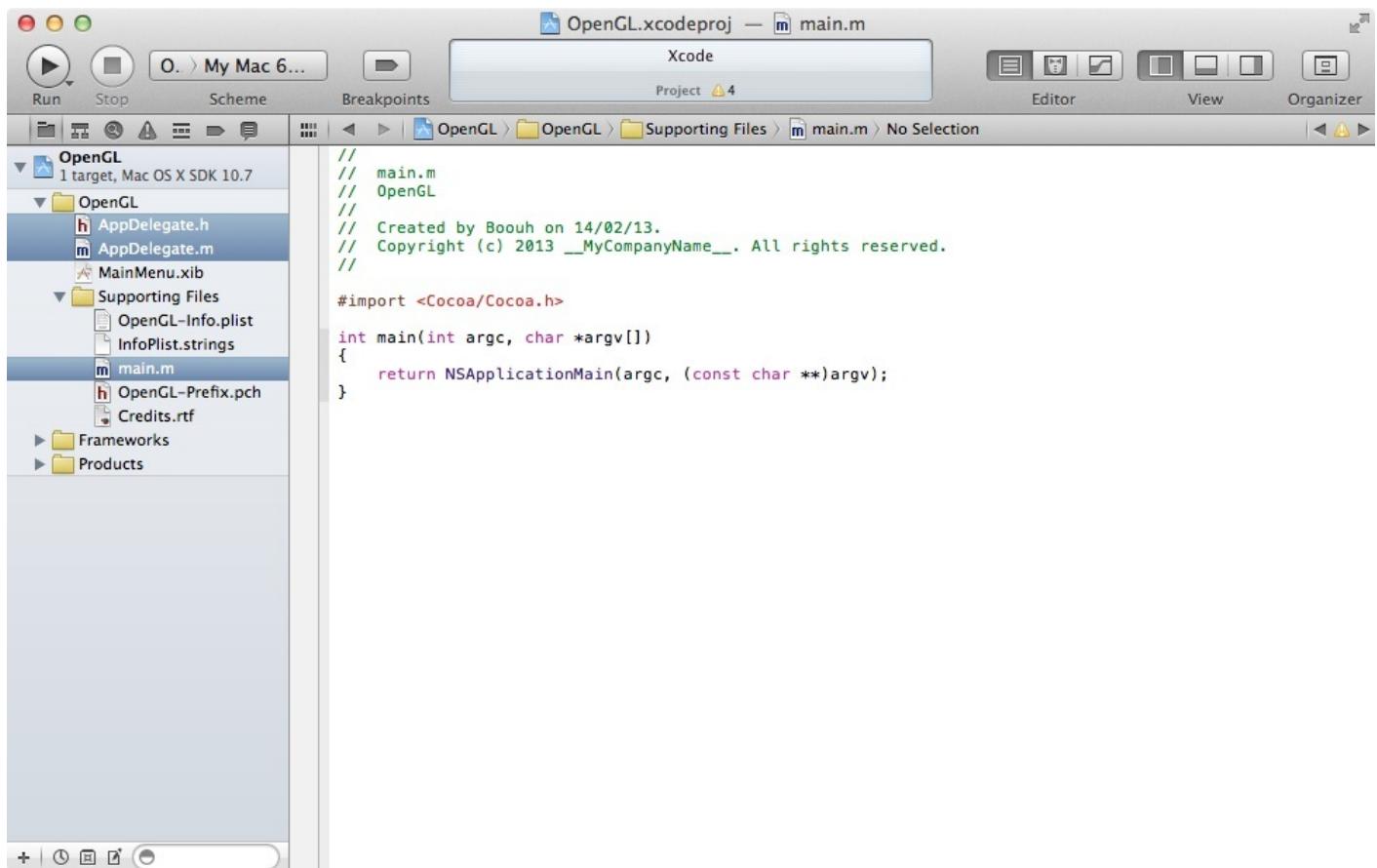
A la place, nous allons utiliser une petite astuce qui, au moins, fonctionnera pour tout le monde. 😊

Pour commencer, vous devez créer un projet "**Cocoa Application**" :

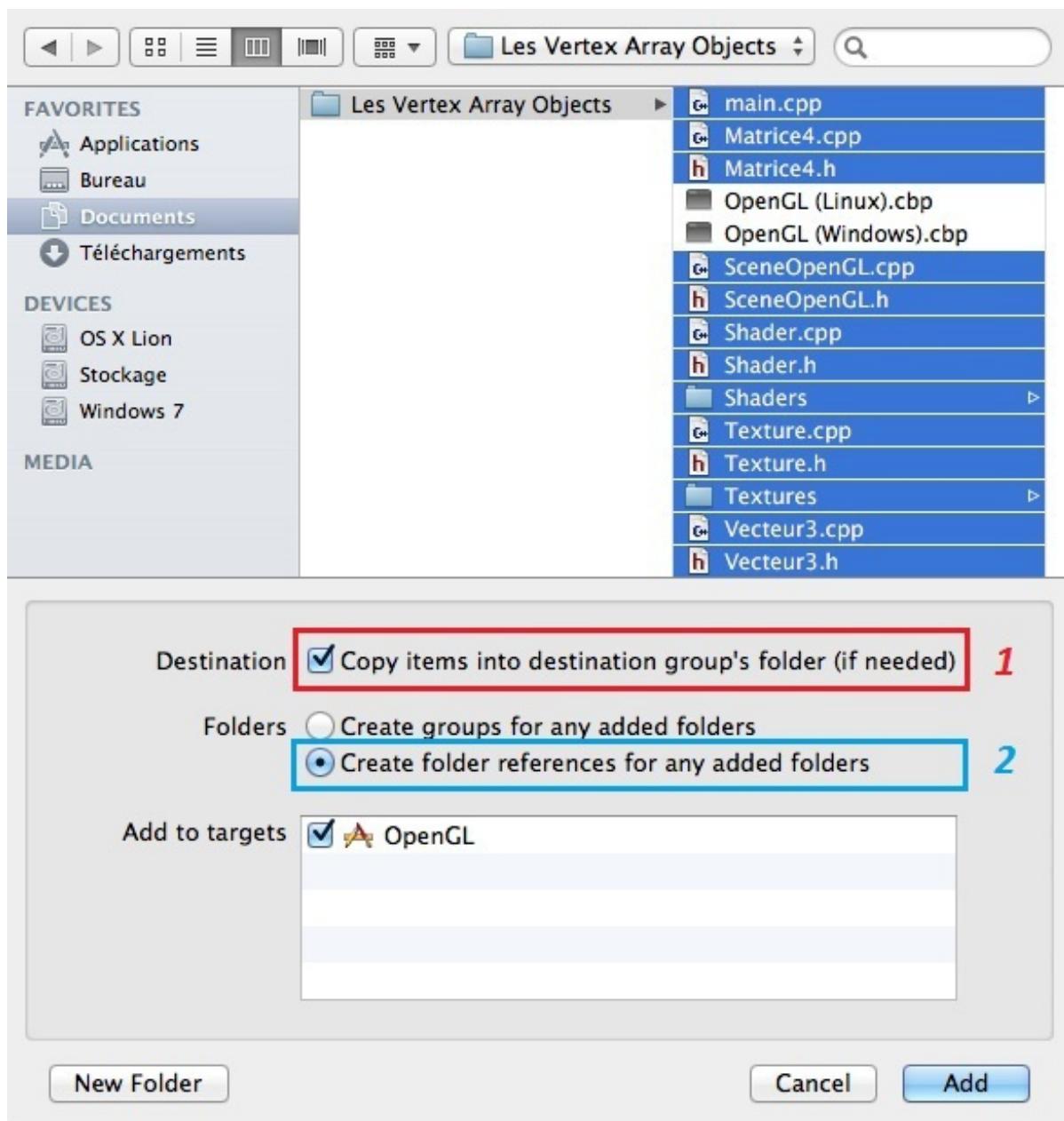


Une fois le projet créé, vous devez supprimer les fichiers suivants :

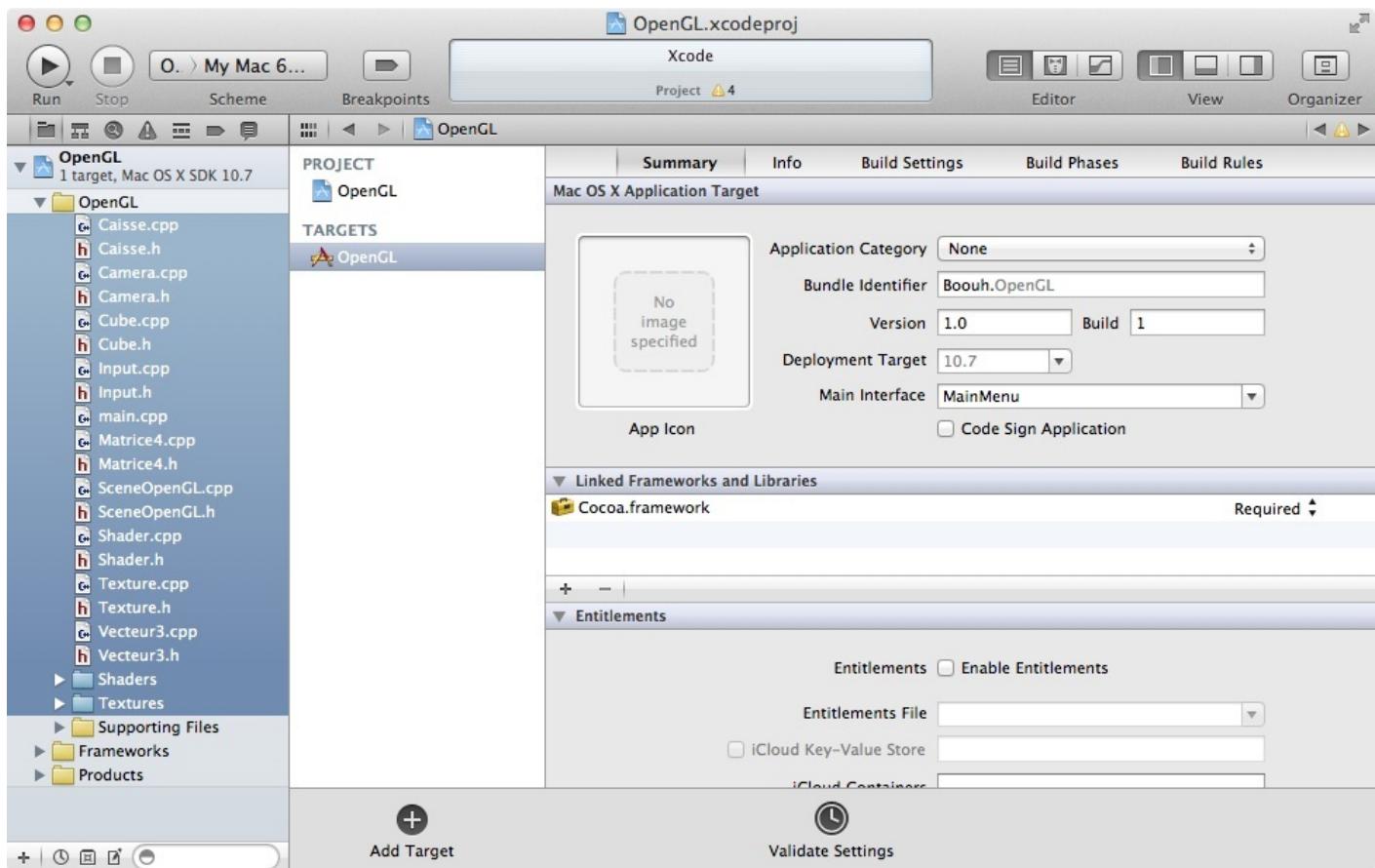
- **AppDelegate.h**
- **AppDelegate.m**
- **main.m**



Ensuite, ajoutez toutes les sources à votre projet (dossiers **Shaders** et **Textures** inclus). Au moment de cet ajout, faites attention à bien spécifier les options suivantes :



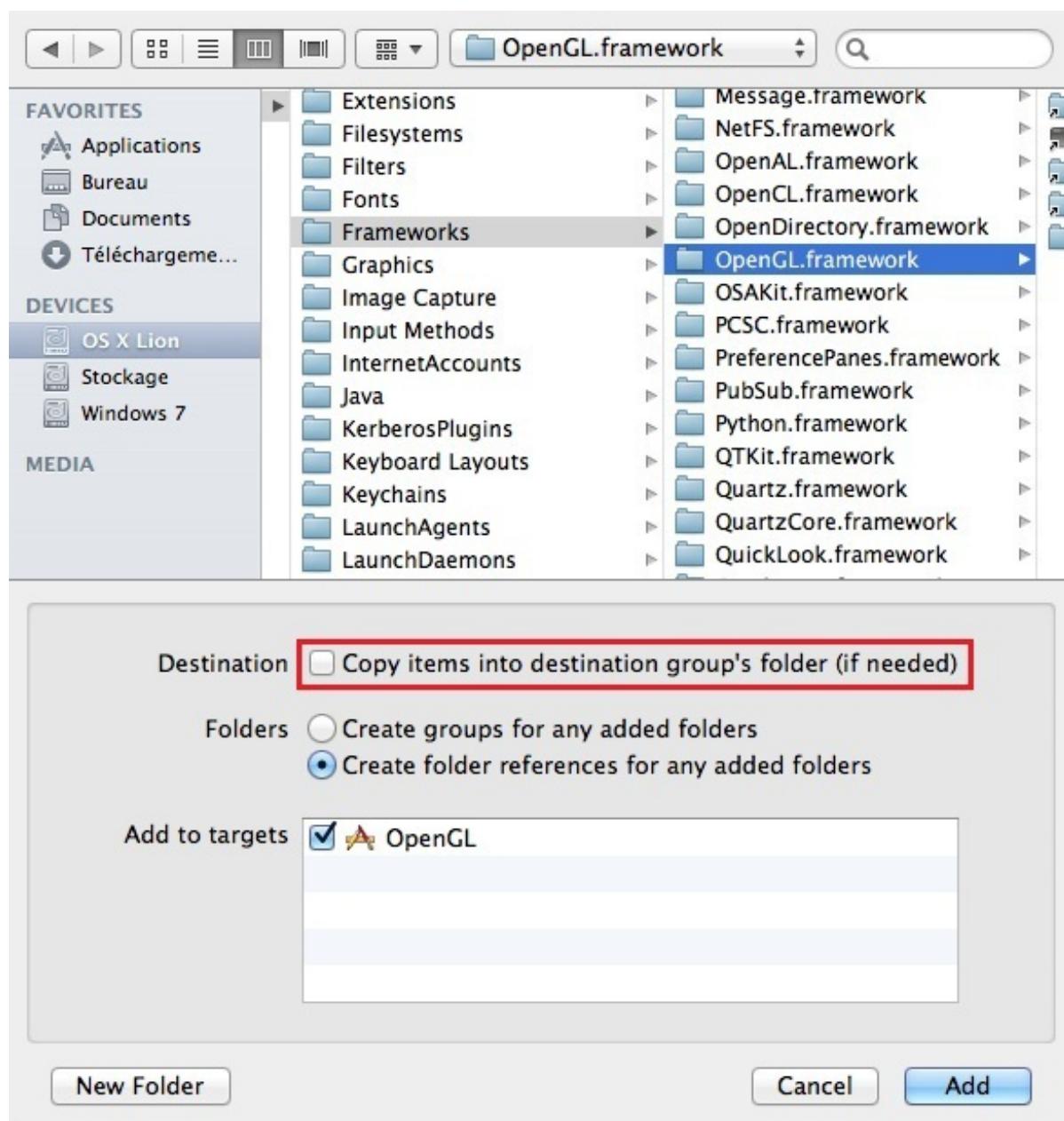
Une fois les fichiers ajoutés, vous devriez avoir :

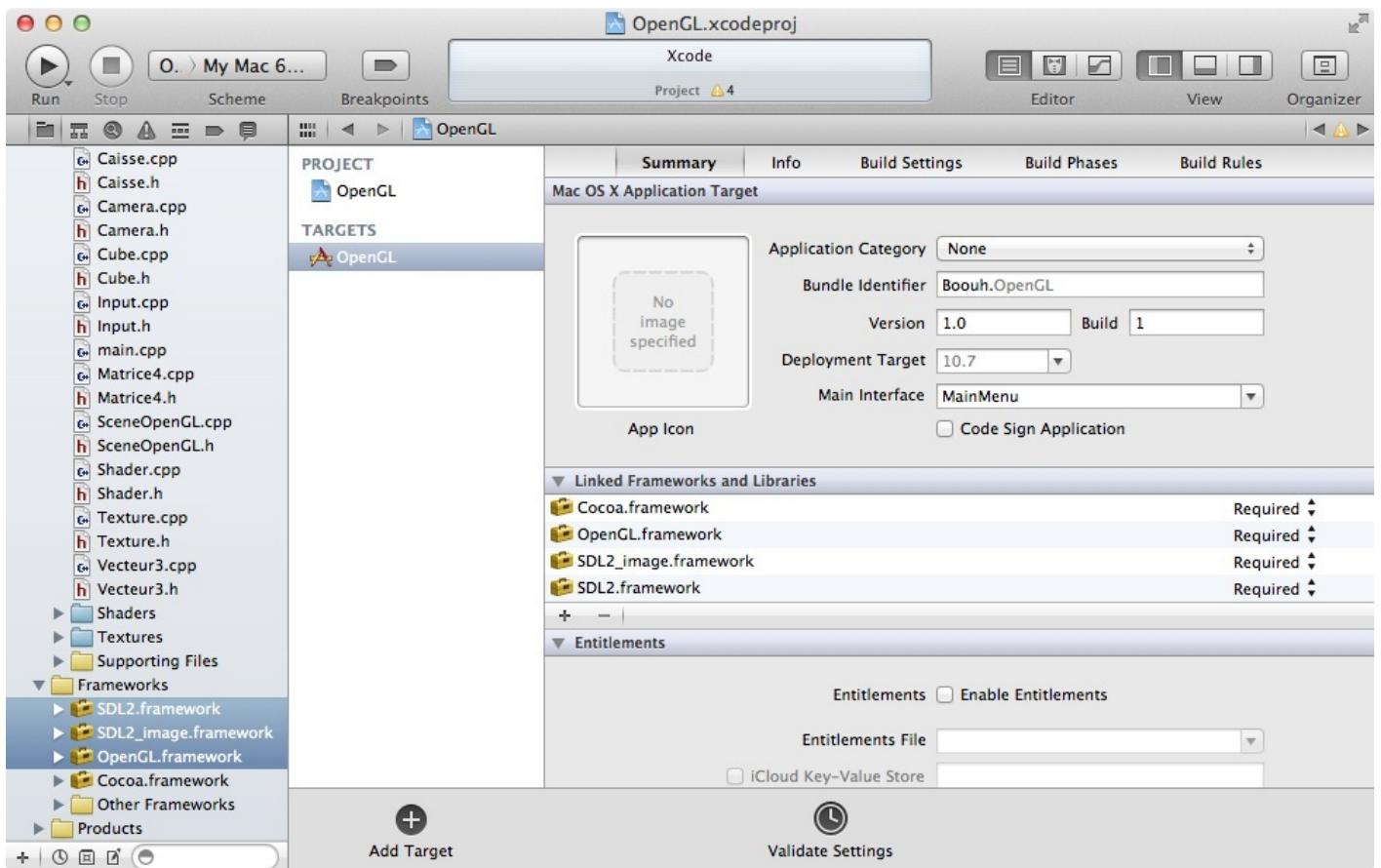


Une fois que vous avez importé les sources, vous devez importer les **Frameworks** qui vous permettront de compiler vos applications :

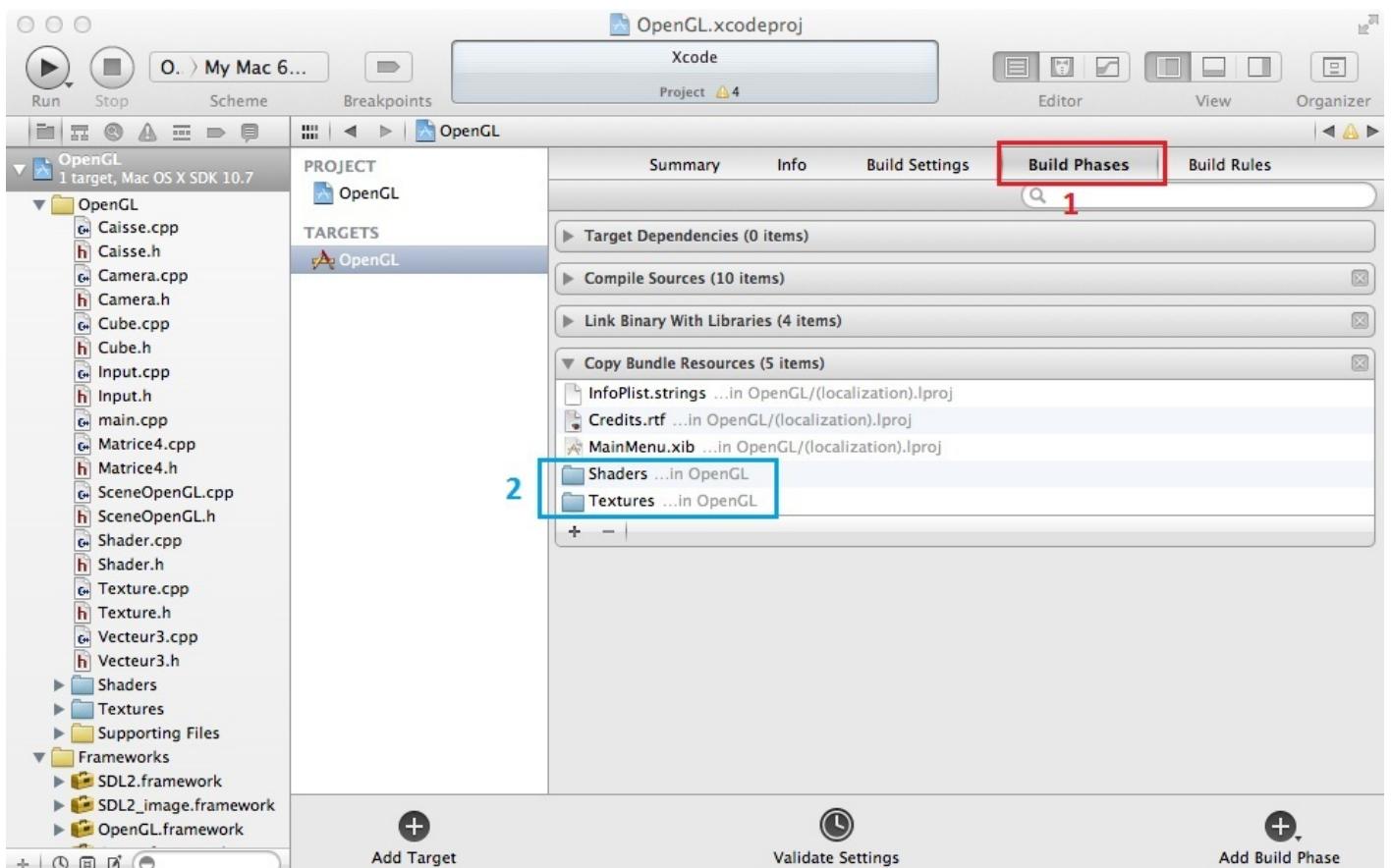
- **SDL2.framework** : Vous l'avez normalement installé dans le dossier **/Library/Frameworks**
- **SDL2_image.framework** : Même chose
- **OpenGL.framework** : Lui, il se trouve dans le répertoire **/System/Library/Frameworks**

Là par contre, il faut décocher la case "**Copy items into destination group's folder**" (Laissez le paramètre **Folders** au deuxième choix). Ceci est valable uniquement pour les Frameworks, pour le reste il faut qu'elle soit cochée.





Enfin dernier point pour l'importation, vérifiez la présence des dossiers **Textures** et **Shaders** dans la liste "Copy Bundle Resources" dans l'onglet **Build Phases** :

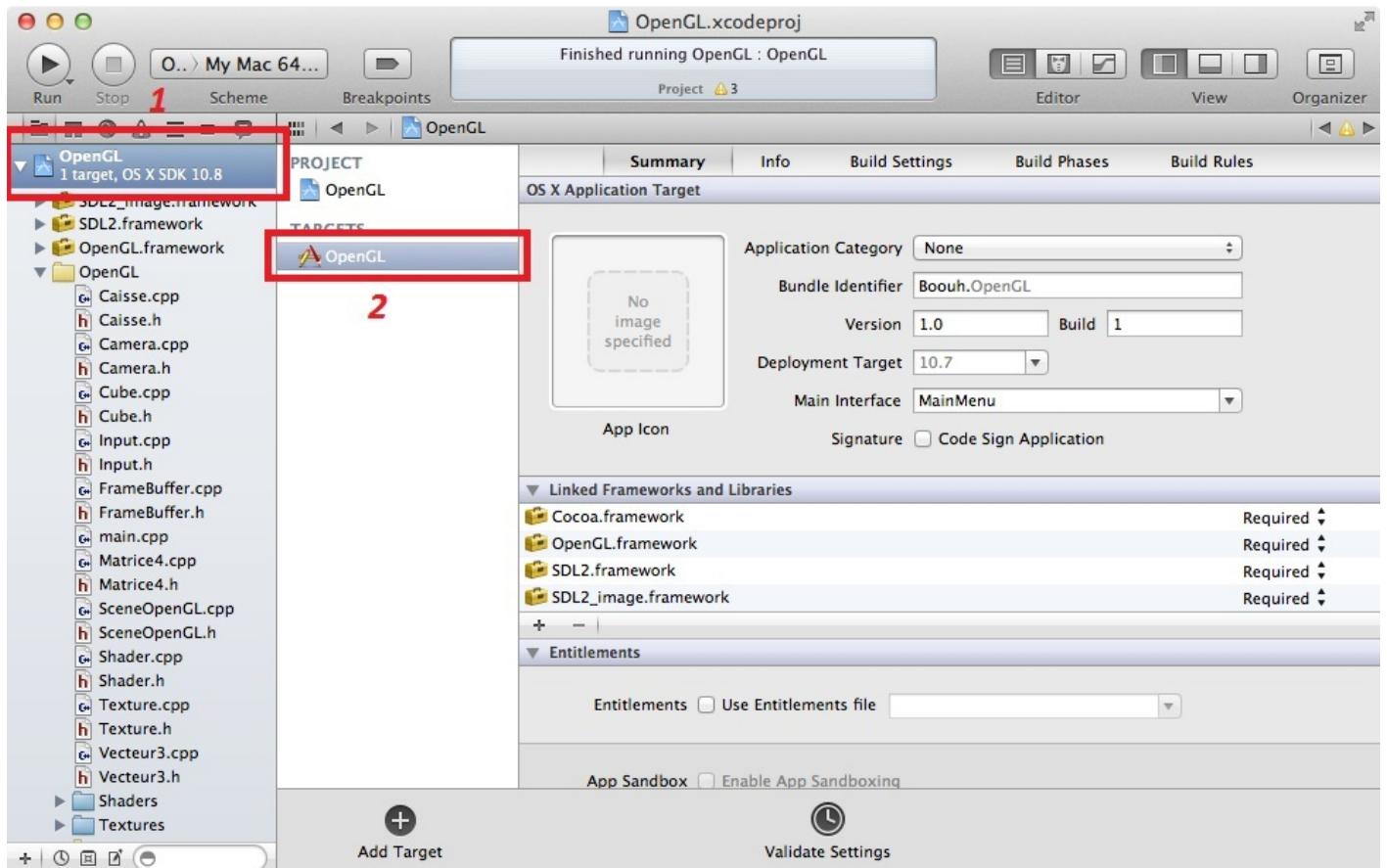


S'ils n'y sont pas, faites un glisser-coller depuis le menu de navigation à gauche.

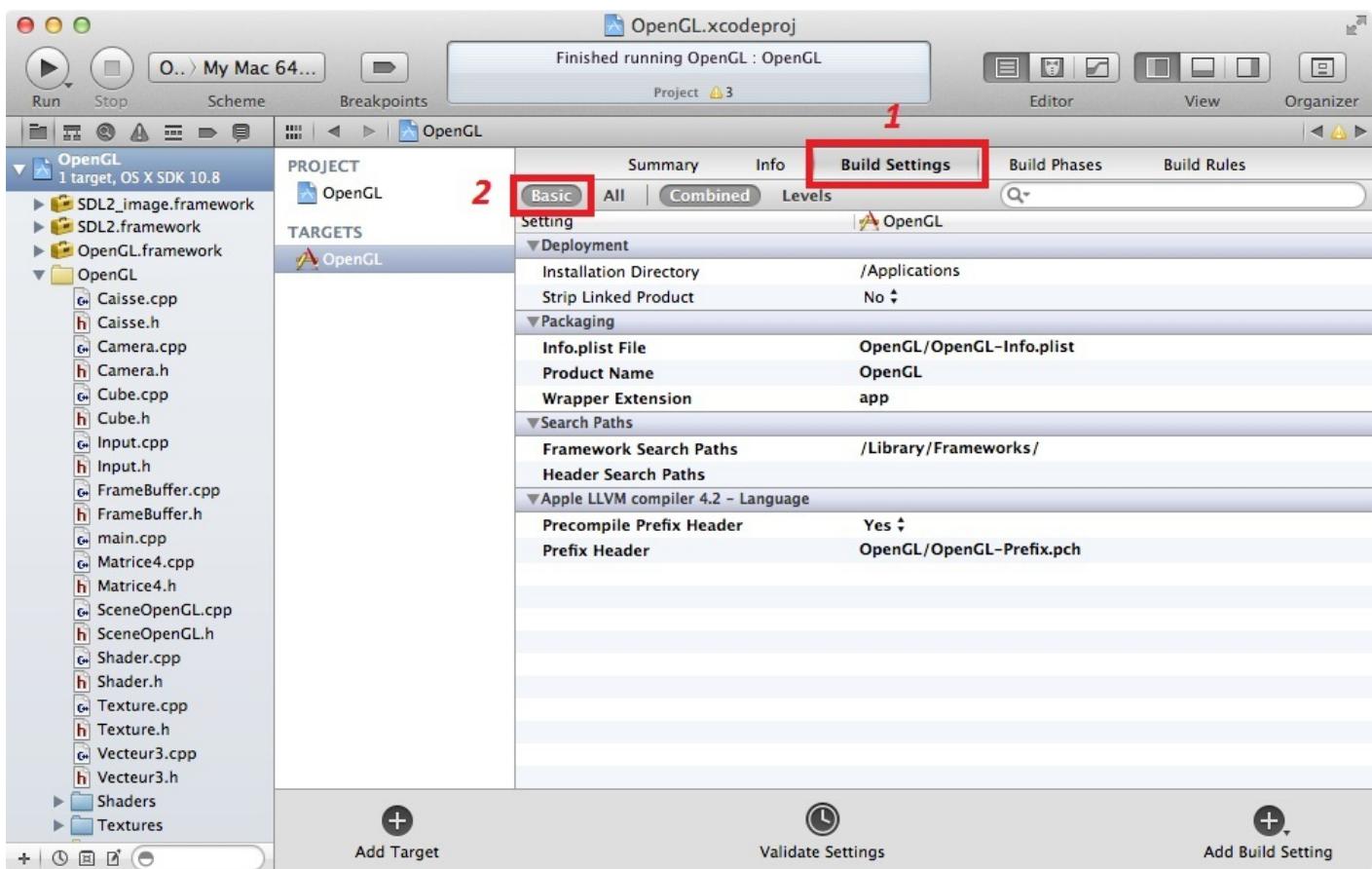
La librairie GLM

Pour vous permettre d'utiliser la librairie **GLM** dans vos projets, il va falloir ajouter un paramètre dans le fichier de votre projet.

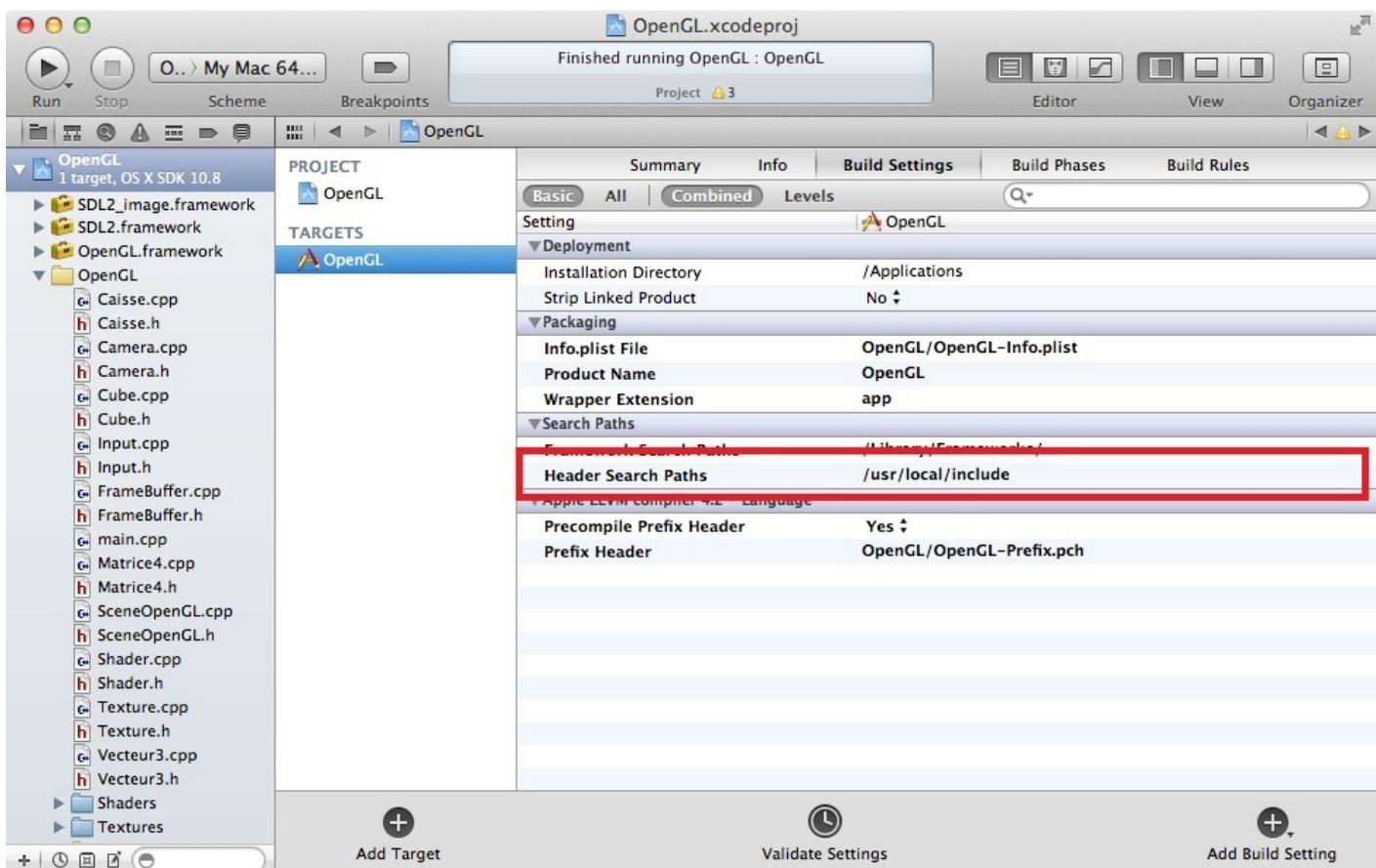
Commencez par cliquer sur le nom de votre projet sous le menu **Targets** :



Puis cliquez sur le menu "Build Settings" et sur le bouton "Basic" :



Repérez ensuite la ligne contenant le paramètre "**Header Search Paths**" et renseignez le chemin où vous avez copié le dossier **glm** tout à l'heure (**/usr/local/include**) :



Version d'OpenGL

N'oubliez pas que votre version d'OpenGL est la **3.2**, vous devez ajouter un bloc de type **#ifdef** dans votre code pour gérer le cas des Mac. Si la define **_APPLE_** est définie, alors l'application doit passer en mode OpenGL **3.2**, sinon elle reste en **3.3** :

Code : C++

```
#ifdef __APPLE__  
    // Version d'OpenGL  
  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,  
        SDL_GL_CONTEXT_PROFILE_CORE);  
  
#else  
    // Version d'OpenGL  
  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);  
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,  
        SDL_GL_CONTEXT_PROFILE_CORE);  
  
#endif
```



Notez que si vous souhaitez faire une application multiplateforme (Mac compris), il est préférable de créer un contexte **3.2** pour tout le monde.

Les ressources

A ce stade vous devriez être capables de compiler votre application. Cependant, vous aurez encore un problème pour trouver les ressources. En effet, Mac OS X ne se place pas dans le bon dossier lorsque vous lancez une application, ce qui fait qu'il est impossible pour lui de trouver les ressources que nous lui demandons.

Pour corriger ce problème, nous allons ajouter un petit bout de code spécifique à Mac OS X dans le même bloc **#ifdef** que précédemment.

Ce code est un peu spécial parce qu'il s'agit à l'origine d'un code **Objective-C** qui a été transposé en C. L'objective-C est le langage de base utilisé par les appareils Apple. Je ne vous expliquerai pas ce code car il faudrait faire un cours sur le **Cocoa** mais sachez simplement qu'il permet à Mac OS X de se placer dans le bon dossier des ressources.

Mais avant de copier ce code, vous devez inclure l'en-tête **CoreFoundation.h** toujours dans un bloc **#ifdef** à l'intérieur de la classe **SceneOpenGL** :

Code : C++

```
#ifdef __APPLE__  
#include <CoreFoundation/CoreFoundation.h>  
#endif
```

Maintenant, copiez le code suivant ...

Code : C++

```
// Récupération du Bundle

CFURLRef URLBundle =
CFBundleCopyResourcesDirectoryURL(CFBundleGetMainBundle());
char *cheminResources = new char[PATH_MAX];

// Changement du 'Working Directory'

if(CFURLGetFileSystemRepresentation(URLBundle, 1,
(UInt8*)cheminResources, PATH_MAX))
    chdir(cheminResources);

// Libération de la mémoire

delete[] cheminResources;
CFRelease(URLBundle);
```

...juste après la spécification de la version d'OpenGL :

Code : C++

```
#ifdef __APPLE__

// Version d'OpenGL

SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE);

// Récupération du Bundle

CFURLRef URLBundle =
CFBundleCopyResourcesDirectoryURL(CFBundleGetMainBundle());
char *cheminResources = new char[PATH_MAX];

// Changement du 'Working Directory'

if(CFURLGetFileSystemRepresentation(URLBundle, 1,
(UInt8*)cheminResources, PATH_MAX))
    chdir(cheminResources);

// Libération de la mémoire

delete[] cheminResources;
CFRelease(URLBundle);

#else

// Version d'OpenGL

SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE);

#endif
```

Grâce à ce bout de code, vous n'aurez plus de problème pour trouver vos ressources. 😊

Les headers

Pour terminer, faites attention aux en-têtes de la **SDL** et d'**OpenGL** qui diffèrent des autres OS :

Code : C++

```
// Include d'OpenGL  
  
#include <OpenGL/gl3.h>  
  
// Include de la SDL  
  
#include <SDL2/SDL.h>  
  
// Include de la SDL_image  
  
#include <SDL2_image/SDL_image.h>
```

Ainsi, l'inclusion d'OpenGL ressemblera maintenant à ceci :

Secret (cliquez pour afficher)

Code : C++

```
// Include Windows  
  
#ifdef WIN32  
#include <GL/glew.h>  
  
// Include Mac  
  
#elif APPLE  
#define GL3_PROTOTYPES 1  
#include <OpenGL/gl3.h>  
  
// Include UNIX/Linux  
  
#else  
#define GL3_PROTOTYPES 1  
#include <GL3/gl3.h>  
  
#endif
```

Celui de la classe **SceneOpenGL** va légèrement changer par rapport aux autres puisqu'il faudra inclure l'en-tête **CoreFoundation.h** en plus :

Secret (cliquez pour afficher)

Code : C++

```
// Include Windows  
  
#ifdef WIN32
```

```
#include <GL/glew.h>

// Include Mac

#elif __APPLE__
#define GL3_PROTOTYPES 1
#include <OpenGL/gl3.h>
#include <CoreFoundation/CoreFoundation.h>

// Include UNIX/Linux

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif
```

Et enfin, celui de la classe **Texture** sera :

Secret ([cliquez pour afficher](#))

Code : C++

```
// Include Windows

#ifndef WIN32
#include <GL/glew.h>
#include <SDL2/SDL_image.h>

// Include Mac

#elif __APPLE__
#define GL3_PROTOTYPES 1
#include <OpenGL/gl3.h>
#include <SDL2_image/SDL_image.h>

// Include UNIX/Linux

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>
#include <SDL2/SDL_image.h>

#endif
```

Voilà pour les explications. 😊

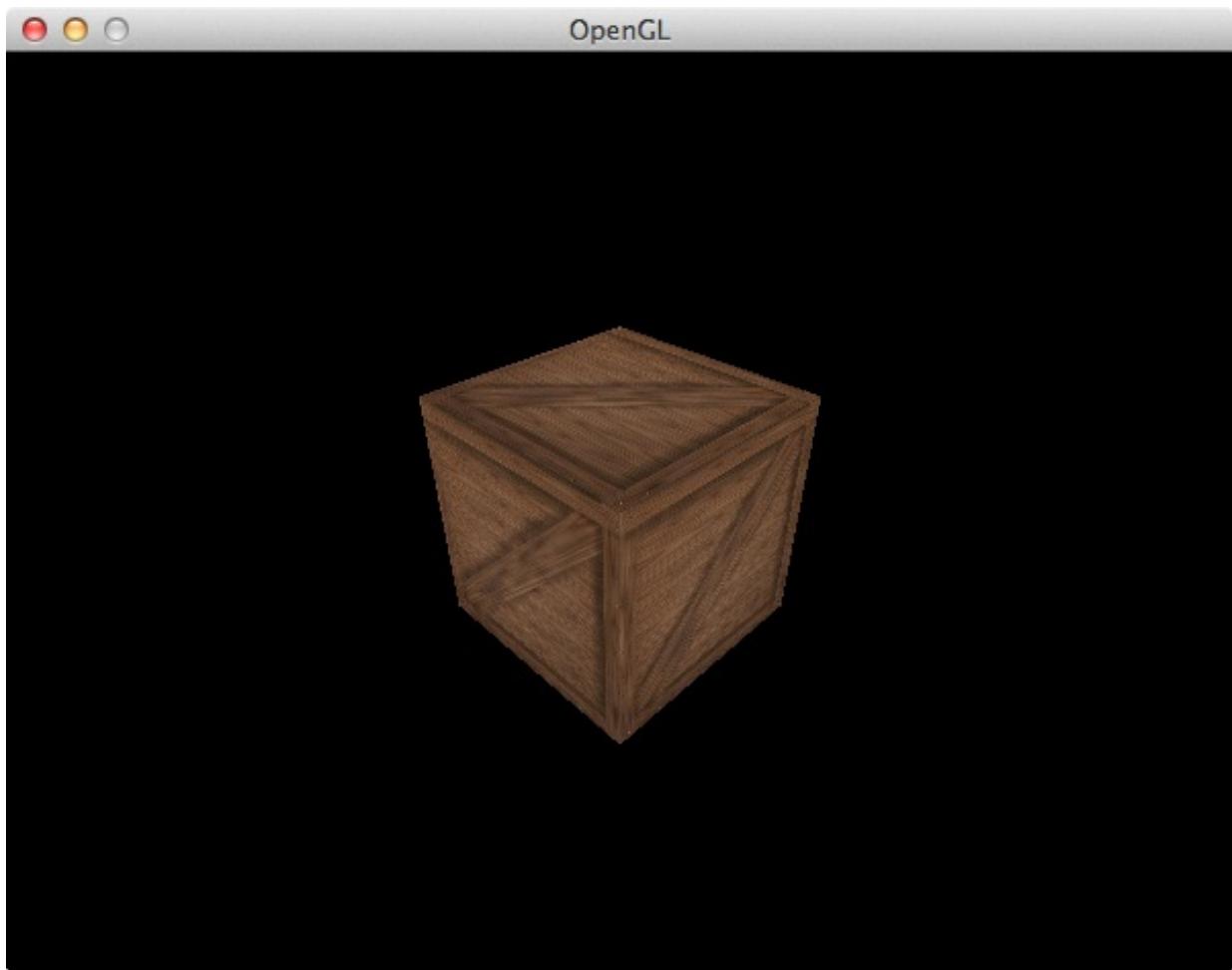
Ce qu'il faut retenir

Si on résume la création de projet **SDL2** sous Mac OS X :

- Création d'un projet **Cocoa Application**
- Suppression de l'**AppDelegate** et du **main.m**
- Ajout des **sources C++**
- Ajout du code concernant les ressources dans le fichier **SceneOpenGL.cpp**

- Modification des **en-têtes**

Vous êtes maintenant capables de coder avec **OpenGL 3.2** sur votre Mac. 😊



Télécharger (Windows, UNIX/Linux et Mac) : [Code Source C++ des Vertex Array Objects](#)

Nous avons vu ce que sont les **VAO** et comment ils fonctionnent. Ils sont très proches des **VBO** et s'utilisent même conjointement avec eux.

Grâce à ces deux notions, *OpenGL* possède tout ce qu'il faut à portée de main vu que tout se trouve dans la carte graphique. Les seules choses qu'il a besoin d'aller chercher dans la RAM, ce sont les textures. Cependant nous pouvons régler ce problème en utilisant les **Texture Buffer Objects (TBO)** qui permettent de stocker des textures directement dans la carte graphique. Mais bon, nous ne verrons pas ça pour le moment vous allez me faire une overdose sinon. 😊

Si je peux vous donner une bonne nouvelle c'est que vous avez acquis assez d'expérience avec OpenGL pour pouvoir apprendre les **shaders**. Nous commencerons à les attaquer en profondeur dès le prochain chapitre. Vous serez alors capables de créer votre propre classe **Shader**. 😊

La compilation de shaders

Re-bonjour à tous !

Aujourd'hui, nous allons attaquer un gros morceau de la programmation OpenGL, je vous conseille de préparer le café, le chocolat, les biscuits, etc. 😊

Les **shaders** sont quelque chose d'assez monstrueux, nous allons les étudier à travers 4 chapitres (et encore sans parler des effets que l'on pourra faire avec) dont le premier est consacré à une classe que l'on utilise presque depuis le début du tutoriel sans jamais avoir vu son fonctionnement. 😊

Piqûre de rappel Introduction

Depuis quasiment le début du tutoriel, nous utilisons une classe dans tous nos programmes sans même que nous sachions ce qu'il y a à l'intérieur. Je parle évidemment de la classe **Shader**.

En effet, que ce soit des couleurs ou des textures nous utilisons toujours cette classe pour afficher quelque chose à l'écran :

Code : C++

```
// Shader pour la colorisation
Shader shaderCouleur("Shaders/couleurs.vert",
"Shaders/couleurs.frag");
shaderCouleur.charger();

.....

// Shader pour les textures
Shader shaderTexture("Shaders/texture.vert",
"Shaders/texture.frag");
shaderTexture.charger();
```

La question que vous vous êtes déjà probablement posée est : que fait cette mystérieuse classe ?

Je vous rassure elle ne fait rien de compliqué. En fait, elle ne fait qu'une seule chose : compiler les fichiers sources qu'on lui envoie.

L'objectif de ce chapitre va être la **reprogrammation** complète de cette classe de façon à ce que vous appreniez comment créer et utiliser un **shader**. A la fin, vous serez en mesure de comprendre totalement votre code source, il n'y aura plus de code mystère. 😊 Mais avant cela, nous allons faire un petit rappel sur ce que nous savons déjà sur le **shaders**.

Rappel

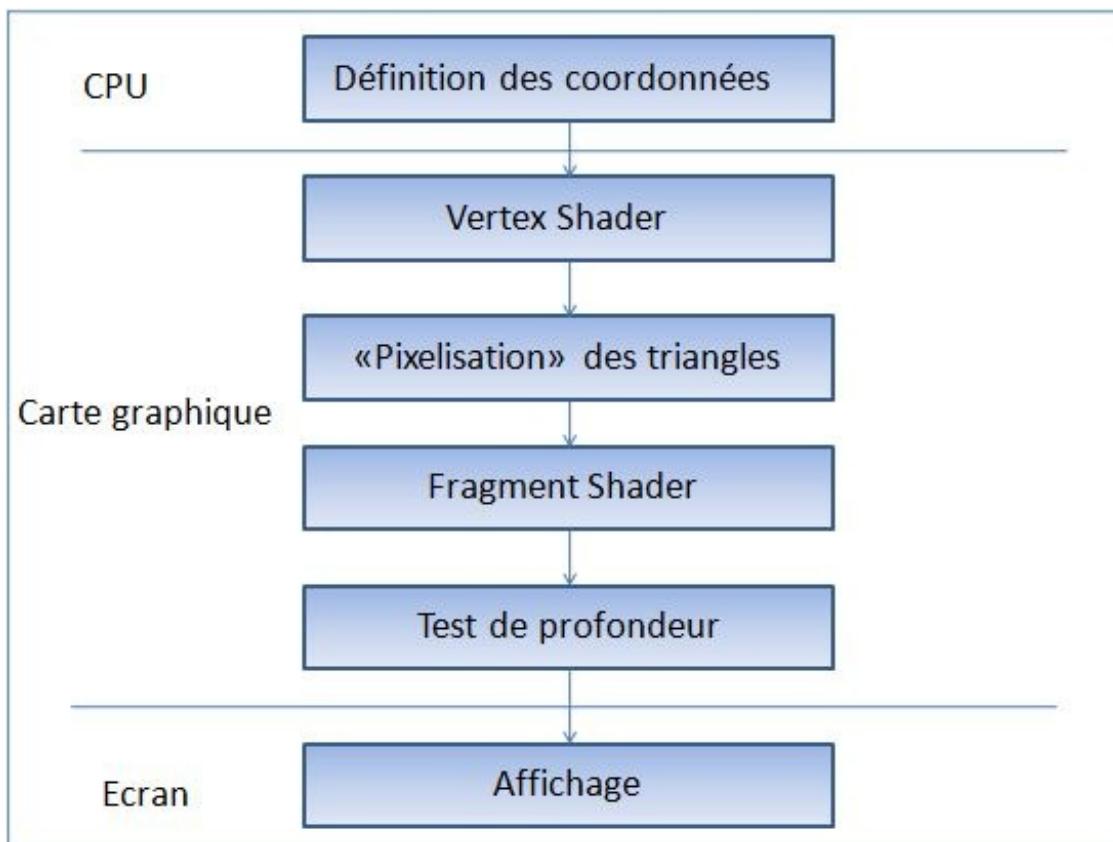
Vertex et Fragment Shader

Comme nous l'avons vu dans le chapitre 4, nous savons que les **shaders** sont des programmes qui sont exécutés non pas par le CPU mais par la carte graphique. Il en existe deux types :

- **Le Vertex Shader** : qui prend chaque vertex à part pour calculer sa position à l'écran. Si un vertex possède 3 coordonnées, alors le **Vertex Shader** aura besoin des matrices **projection** et **modelview** pour faire le calcul.
- **Le Fragment Shader** (ou **Pixel Shader**) : qui prend à part chaque pixel des triangles formés par les vertices pour définir

sa couleur. Si on utilise une texture, alors le **Fragment Shader** va chercher à l'intérieur de cette texture la couleur dont il a besoin.

Ces deux types de **shader** sont utilisés à deux moments différents du **pipeline 3D** :



Pour rappel :

- **Définition des coordonnées** : Ce sont les vertices et toutes les données qui y sont associées. Depuis peu, tout ça se trouve dans la carte graphique grâce aux **VBO**.
- **Vertex Shader** : Shader qui travaille sur les vertices
- **Pixelisation des triangles** : Moment où une forme géométrique (formée par les vertices) est convertie en pixels
- **Fragment Shader** : Shader qui travaille sur les pixels
- **Test de profondeur** : Test qui permet d'afficher ou de cacher un pixel
- **Affichage** : Sortie de la carte graphique (en général, il s'agit de l'écran)

Au final, nous voyons que les shaders sont exécutés à deux moments : une fois pour transformer les vertices et une autre pour définir leur couleur.

Geometry Shader

En réalité, les shaders devraient être exécutés 3 fois car il existe un troisième type qui s'appelle le **Geometry Shader** qui vient se placer juste entre les deux premiers. Celui-ci permet de modifier les primitives (triangles, ...) formées par les vertices. Cependant, nous ne l'utiliserons pas d'une part parce qu'il est optionnel et d'autre part car il est quasiment inutile pour nous. 🍪

Soit dit en passant, ce nouveau type a été introduit avec la version **3.2 d'OpenGL**.

Utilité

Les shaders ne se limitent pas au positionnement de vertices ou à la colorisation de pixels, ils permettent aussi de créer une multitude d'effets réalistes comme l'eau, la lumière, le feu, ... (ce que vous n'avez sûrement pas oublié je pense 😊). D'ailleurs, c'est certainement leur première utilité dans le monde du jeu-vidéo. Nous consacrerons une partie entière de ce tutoriel à l'élaboration de ces effets.

Depuis la version **3.1** d'**OpenGL**, l'utilisation des shaders est devenue obligatoire car les développeurs de l'API ont introduit une nouvelle philosophie : celle du '**tout shader**'. Dans les précédentes versions, OpenGL gérait lui-même tout le **pipeline 3D** sans rien demander à personne mises à part les données de base comme les vertices. Seulement, cette gestion est devenue trop lourde aujourd'hui, les développeurs de l'API préfèrent nous laisser gérer le maximum de tâches de façon à ce que nous les optimisions mieux en fonction de ce que nous voulons faire.

Cette philosophie complique grandement l'apprentissage d'OpenGL, c'est pour ça que nous n'avons pas vu les shaders tout de suite, sinon le tuto aurait été beaucoup plus complexe à suivre. Vous m'auriez sûrement fait une overdose avant même d'avoir atteint le chapitre sur les matrices. 😊

La classe Shader

Enfin, maintenant que vous avez acquis assez d'expérience avec la programmation OpenGL, nous allons pouvoir voir en détails le fonctionnement des shaders. Nous commencerons en premier par étudier la fameuse classe **Shader**. Et comme je vous l'ai dit précédemment, elle ne fait rien de magique, elle ne fait que compiler du code source.

Car oui, comme tout programme qui se respecte, les shaders possèdent bel et bien un code source. La seule différence avec les programmes classiques c'est qu'il est compilé juste avant d'être utilisé et pas avant, il n'y a donc pas d'application de type .exe à donner à votre carte graphique. D'ailleurs en parlant de ça, ces codes sources sont écrits avec un langage proche du C++ (ou plutôt du C) qui s'appelle l'**OpenGL Shading Language** (ou **GLSL**). Nous l'étudierons dans le prochain chapitre, nous devons savoir avant comment le compiler. 😊



Il faut bien faire la différence entre la **COMPIRATION** de shaders et leur **PROGRAMMATION**. Ce que nous voyons dans ce chapitre concerne uniquement la compilation, ne confondez pas.

Compilation des sources

La classe Shader

Bien, après cette brève introduction j'espère vous avoir mis un peu l'eau à la bouche. 😊

Il est temps de passer à la reprogrammation complète de la classe **Shader**, je vais vous demander de supprimer les fichiers **Shader.h** et **Shader.cpp** que vous utilisez actuellement et d'en recréer deux nouveaux (avec le même nom bien sûr). Si vous essayez de compiler votre code après ça, vous devriez avoir une belle petite erreur de compilation. 😊

Bref, on commence cette reprogrammation par le header en incluant tous les en-têtes nécessaires : ceux d'**OpenGL**, d'**iostream** et de **string**. Nous n'aurons pas besoin de ceux de la **SDL** car ils n'ont aucun rapport avec les shaders :

Code : C++

```
#ifndef DEF_SHADER
#define DEF_SHADER

// Include Windows

#ifndef WIN32
#include <GL/glew.h>

// Include Mac

#ifndef __APPLE__
#define GL3_PROTOTYPES 1
#include <OpenGL/g13.h>
```

```
// Include UNIX/Linux

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

// Includes communs

#include <iostream>
#include <string>

#endif
```



Remarquez l'en-tête pour Mac OS X qui permet de rendre nos projets compatibles avec cet OS. 😊

Après les inclusions, on passe à la classe en elle-même. Elle possèdera 3 attributs majeurs : des variables de type **GLuint** représentant le Vertex Shader, le Fragment Shader et un programme. Nous ajouterons également le constructeur par défaut ainsi que le destructeur :

Code : C++

```
// Classe Shader

class Shader
{
public:

    Shader();
    ~Shader();

private:

    GLuint m_vertexID;
    GLuint m_fragmentID;
    GLuint m_programID;
};
```



Ah **programID** ? C'est le truc qu'on utilise dans la fonction **glUseProgram()** non ?

Oui tout à fait. 😊

Comme vous le savez un shader est un programme, c'est pour cette raison que nous avons cet attribut. Remarquez aussi que les 3 variables présentées sont en fait des *objets OpenGL* car elles ont le type **GLuint** au même titre que les textures, les VBO ou les VAO.

Nous allons également rajouter un autre constructeur à cette classe. C'est un constructeur que vous connaissez par cœur car c'est celui qui demande en paramètre le chemin vers les deux codes sources shaders. Son prototype est le suivant :

Code : C++

```
Shader(std::string vertexSource, std::string fragmentSource);
```

Il prend en paramètre deux string représentant le chemin vers les codes sources spécifiés. Nous aurons besoin de deux attributs supplémentaires pour les gérer dans la classe. 😊

Nous rajoutons donc deux variables **m_vertexSource** et **m_fragmentSource** de type **string** dans le header :

Code : C++

```
// Attributs

GLuint m_vertexID;
GLuint m_fragmentID;
GLuint m_programID;

std::string m_vertexSource;
std::string m_fragmentSource;
```

Si on récapitule tout ça :

Code : C++

```
#ifndef DEF_SHADER
#define DEF_SHADER

// Include Windows

#ifdef WIN32
#include <GL/glew.h>

// Include Mac

#elif __APPLE__
#define GL3_PROTOTYPES 1
#include <OpenGL/gl3.h>

// Include UNIX/Linux

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

// Includes communs

#include <iostream>
#include <string>

// Classe Shader

class Shader
{
public:

    Shader();
    Shader(std::string vertexSource, std::string fragmentSource);
```

```
~Shader();  
  
private:  
  
GLuint m_vertexID;  
GLuint m_fragmentID;  
GLuint m_programID;  
  
std::string m_vertexSource;  
std::string m_fragmentSource;  
};  
  
#endif
```

Header terminé. 😊

Les constructeurs et le destructeur

Maintenant que le header de la classe est terminé, on va pouvoir passer à son implémentation. Et comme d'habitude, on commence par les constructeurs et le destructeur.

Le premier est le constructeur par défaut, qui ne fait rien de particulier d'ailleurs :

Code : C++

```
// Constructeur par défaut  
  
Shader::Shader() : m_vertexID(0), m_fragmentID(0), m_programID(0),  
m_vertexSource(), m_fragmentSource()  
{  
}
```

Le second, lui, fera la même chose sauf qu'il affectera les sources avec les **string** donnés en paramètres :

Code : C++

```
// Constructeur  
  
Shader::Shader(std::string vertexSource, std::string fragmentSource)  
: m_vertexID(0), m_fragmentID(0), m_programID(0),  
m_vertexSource(vertexSource), m_fragmentSource(fragmentSource)  
{  
}
```

Quant au destructeur, vous savez maintenant ce qui se trouve à l'intérieur. Mais attention ! Celui-ci ne restera pas vide longtemps. 😱

Code : C++

```
// Destructeur  
  
Shader::~Shader()  
{  
}
```

La méthode getProgramID

L'attribut **m_programID** est quelque chose que l'on utilise assez souvent, notamment pour activer les shaders. Nous allons recoder le getter qui permet de le récupérer et qui s'appelle **getProgramID()** :

Code : C++

```
GLuint getProgramID() const;
```

Son implémentation se passe de commentaire :

Code : C++

```
GLuint Shader::getProgramID() const
{
    return m_programID;
}
```

La méthode charger (Partie 1/2)

La méthode **charger()** va être la principale méthode de cette classe. En effet, vous avez sûrement remarqué qu'on l'utilise à chaque fois que l'on veut initialiser un **shader** :

Code : C++

```
// Shader Texture

shaderTexture("Shaders/texture.vert", "Shaders/texture.frag");
shaderTexture.charger();
```

Son prototype est assez simple vu qu'elle ne demande rien en paramètre :

Code : C++

```
bool charger();
```

Elle renverra un booléen pour confirmer ou non le chargement du shader.

Cependant, contrairement à son prototype, la méthode **charger()** ne sera pas aussi simple à programmer. Elle devra être capable de faire plusieurs choses :

- **Compiler** le code source du Vertex Shader
- **Compiler** le code source du Fragment Shader
- **Réunir** les fichiers compilés dans un seul et même programme

Pour effectuer les deux premières étapes, nous allons créer une autre méthode qui permettra de compiler le code source d'un shader. Il vaut mieux séparer la compilation du reste du code car vous allez voir que c'est un peu ... long.  Cette nouvelle méthode s'appellera simplement **compilerShader()** :

Code : C++

```
bool compilerShader(GLuint &shader, GLenum type, std::string const &fichierSource);
```

Elle aura besoin de 3 paramètres :

- **shader** : Le shader à compiler
- **type** : Un paramètre semblable au paramètre **target** des *objets OpenGL* (comme **GL_TEXTURE_2D**) pour définir le type de shader. Il peut prendre la valeur **GL_VERTEX_SHADER** ou **GL_FRAGMENT_SHADER**
- **fichierSource** : Le code source associé au shader

Un booléen sera également retourné pour confirmer ou non la réussite de la compilation.

Nous devons appeler cette nouvelle méthode deux fois afin de compiler nos deux codes sources. Le premier appel devra compiler un shader contenu dans l'attribut **m_vertexID** et dont le code source sera évidemment identifié par **m_vertexSource**. Le paramètre **type** sera quant à lui égal à la constante **GL_VERTEX_SHADER**. On n'oublie pas de vérifier le retour dans un bloc **if** :

Code : C++

```
bool Shader::charger()
{
    // Compilation du Vertex Shader

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,
m_vertexSource))
        return false;
}
```

Le second appel à cette méthode s'occupera cette fois du Fragment Shader. On lui donnera en paramètres les attributs **m_fragmentID** et **m_fragmentSource**. Le paramètre **type** sera égal à la constante **GL_FRAGMENT_SOURCE** :

Code : C++

```
bool Shader::charger()
{
    // Compilation des shaders

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,
m_vertexSource))
        return false;

    if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,
m_fragmentSource))
        return false;
}
```



On ne retourne pas de message d'erreur ?

Si justement, les messages d'erreurs seront en fait affichés par la méthode **compilerShader()** car c'est à elle de dire si quelque chose ne va pas. Il est très important de vérifier les erreurs de compilation avec les shaders car nous n'avons pas d'IDE pour

nous dire où elles seraient. Ce peut devenir vite embêtant si vous perdez trop de temps à chercher les erreurs par vous-même. D'ailleurs la vérification de messages d'erreurs est quelque chose ... d'assez folklo à programmer vous verrez. 😊

La méthode `compilerShader`

Créer un shader

Nous connaissons déjà le prototype de cette méthode et nous savons également qu'elle a besoin de 3 paramètres :

Code : C++

```
bool compilerShader(GLuint &shader, GLenum type, std::string const &fichierSource);
```

Cette méthode devra être capable de faire plusieurs choses :

- Générer un ID de shader
- Lire son code source
- Le compiler

La première étape va être très simple à réaliser car elle est très similaire à la création des autres *objets OpenGL*. Si je dis similaire, c'est parce que nous n'allons pas utiliser les fonctions classiques du type `glGenXXX()`, `glBindXXX()`, ... comme nous avons l'habitude de faire. Les shaders sont des *objets* un peu particuliers qui vont utiliser leurs propres fonctions. De plus, ils ne doivent pas être verrouillés pendant leur configuration mais seulement au moment de leur utilisation.

En définitif, nous ne devons pas utiliser la fonction `glGenShaders()` pour générer un ID mais la fonction `glCreateShader()` :

Code : C++

```
GLuint glCreateShader(GLenum shaderType);
```

Elle prend en paramètre une constante qui peut prendre la valeur `GL_VERTEX_SHADER` ou `GL_FRAGMENT_SHADER`. Nous lui donnerons la valeur du paramètre `type` de la méthode `compilerShader()`.

Nous appellerons cette fonction en affectant la valeur renournée au paramètre `shader` :

Code : C++

```
bool Shader::compilerShader(GLuint &shader, GLenum type, std::string const &fichierSource)
{
    // Création du shader
    shader = glCreateShader(type);
}
```

Vu que nous utilisons un paramètre externe à la méthode pour définir le `type` de shader, il est donc possible que ce paramètre contienne une valeur erronée. Nous devons donc vérifier l'`ID` retourné par la fonction `glCreateShader()`. S'il est égal à `0` c'est qu'il y a eu un problème, il faut donc arrêter le chargement :

Code : C++

```
// Vérification du shader

if(shader == 0)
{
    std::cout << "Erreur, le type de shader (" << type << ")"
n'existe pas" << std::endl;
return false;
}
```

Lecture du code source

La lecture du code source est quelque chose de capital dans la création d'un shader. Sans lui, notre programme ne sait pas quoi faire et la carte graphique nous enverrait gentiment paître. 

Nous devons lire les codes sources dont les chemins sont contenus dans les attributs **m_vertexSource** et **m_fragmentSource**. Pour cela, nous allons utiliser la bibliothèque **fstream** qui permet de la lecture et l'écriture de fichier en C++, je suppose que vous la connaissez déjà.  Pensez à inclure l'en-tête **fstream** dans le fichier **Shader.h**.

Pour utiliser cette bibliothèque, nous allons commencer par déclarer un objet de type **ifstream** que nous appellerons simplement **fichier**. Son constructeur demande une chaîne C représentant le chemin vers le fichier à lire, nous lui donnerons donc la chaîne C du paramètre **fichierSource** de la méthode **compilerShader()** :

Code : C++

```
// Flux de lecture

std::ifstream fichier(fichierSource.c_str());
```

Avant de continuer, nous devons tester l'ouverture du fichier au cas où celui-ci n'existerait pas. Si c'est le cas alors on arrête le chargement, on affiche un message d'erreur et on détruit le shader que l'on a créé juste avant.

La fonction permettant de détruire un shader ressemble fortement aux fonctions du type **glDeleteXXX()** sauf qu'elle ne prend pas deux paramètres mais un seul, elle n'a également pas de **s** à la fin de son nom :

Code : C++

```
glDeleteShader(GLuint shader);
```

Bien évidemment, elle prend en paramètre le shader à détruire.

Avec cette fonction et le code précédent, notre test d'ouverture ressemble donc à ceci :

Code : C++

```
// Flux de lecture

std::ifstream fichier(fichierSource.c_str());

// Test d'ouverture

if(!fichier)
{
    std::cout << "Erreur le fichier " << fichierSource << " est "
introuvable" << std::endl;
```

```
    glDeleteShader(shader);  
  
    return false;  
}
```

Si le fichier s'est ouvert correctement alors nous pouvons copier son contenu sans problème. Pour cela, nous allons utiliser la fonction **getline()** qui permet de copier une ligne entière d'un fichier. Nous l'utiliserons à l'intérieur d'une boucle **while**, ce qui permettra de copier le code source ligne par ligne. 😊

Nous aurons besoin de deux **string** pour cette boucle : une qui contiendra chaque ligne lue, et une autre qui contiendra le code source final :

Code : C++

```
// Strings permettant de lire le code source  
  
std::string ligne;  
std::string codeSource;  
  
// Lecture  
  
while(getline(fichier, ligne))  
    codeSource += ligne + '\n';
```

N'oubliez pas l'inclure le caractère de saut de ligne '\n' à la fin de chaque copie, sinon votre code source ne sera constitué que d'une gigantesque ligne incompréhensible. 🤪

Une fois la lecture terminée, on n'oublie pas de fermer le fichier pour éviter les fuites de mémoire :

Code : C++

```
// Strings permettant de lire le code source  
  
std::string ligne;  
std::string codeSource;  
  
// Lecture  
  
while(getline(fichier, ligne))  
    codeSource += ligne + '\n';  
  
// Fermeture du fichier  
  
fichier.close();
```

On fait un petit récap de ce que nous avons fait jusqu'à maintenant :

Code : C++

```
bool Shader::compileShader(GLuint &shader, GLenum type, std::string  
const &fichierSource)  
{  
    // Création du shader  
  
    shader = glCreateShader(type);
```

```
// Vérification du shader

if(shader == 0)
{
    std::cout << "Erreur, le type de shader (" << type << ")"
    n'existe pas" << std::endl;
    return false;
}

// Flux de lecture

std::ifstream fichier(fichierSource.c_str());

// Test d'ouverture

if (!fichier)
{
    std::cout << "Erreur le fichier " << fichierSource << " est "
    introuvable" << std::endl;
    glDeleteShader(shader);

    return false;
}

// Strings permettant de lire le code source

std::string ligne;
std::string codeSource;

// Lecture

while(getline(fichier, ligne))
    codeSource += ligne + '\n';

// Fermeture du fichier

fichier.close();
}
```

Compilation du shader

Faisons le point sur ce que nous avons fait jusqu'à présent : nous avons créé un shader et lu un code source qui se trouve maintenant en mémoire. Le problème c'est que les deux sont totalement dissociés pour le moment. Notre prochain objectif est donc d'envoyer ce code source à notre shader pour qu'il puisse enfin recevoir ses instructions.

Pour cela, nous allons utiliser une fonction OpenGL qui s'appelle **glShaderSource()** :

Code : C++

```
void glShaderSource(GLuint shader, GLsizei count, const GLchar
**string, const GLint *length)
```

- **shader** : Le shader concerné
- **count** : Paramètre indiquant le nombre de chaînes de caractère à envoyer. Nous lui donnerons la valeur **1** car notre code

source n'est composé que d'une seule **string**

- **string** : Sorte de "**double pointeur**" représentant un tableau de sous-chaines de caractère. Nous utiliserons une petite astuce pour ne lui donner qu'une seule chaîne
- **length** : Tableau de taille des sous-chaines. Heureusement pour nous, nous n'aurons pas à utiliser d'astuce car OpenGL nous autorise à envoyer la valeur **0**. Nous n'allons donc pas nous en priver 😊

Cette fonction est un peu spéciale vous l'aurez remarqué.

Cependant, nous pouvons contourner son trop-plein de tableaux en ne fournissant qu'une seule et unique chaîne de caractère. Il suffira d'envoyer l'adresse d'une **chaîne C**. En effet, si on envoie l'adresse d'une chaîne alors on se retrouve avec un pointeur du type ****chaîne**. La fonction **glShaderSource()** demande justement cette sorte de double pointeur. 😊

On commence donc pas récupérer la **chaîne C** du code source dans un pointeur grâce à la méthode **c_str()** :

Code : C++

```
// Récupération de la chaîne C du code source  
const GLchar* chaîneCodeSource = codeSource.c_str();
```

Ensuite, nous appelons la fonction **glShaderSource()** avec les paramètres que nous venons de voir. Bien entendu, on pense à donner l'adresse du pointeur **chaîneCodeSource** pour contenter le paramètre **string** :

Code : C++

```
// Récupération de la chaîne C du code source  
const GLchar* chaîneCodeSource = codeSource.c_str();  
  
// Envoi du code source au shader  
glShaderSource(shader, 1, &chaîneCodeSource, 0);
```

Notre shader vient enfin de récupérer son code source. 😊 Il ne manque plus qu'à le compiler.

Et pour ça, on va utiliser une fonction toute simple (ce qui fait du bien par rapport à la précédente) qui s'appelle **glCompileShader()** :

Code : C++

```
void glCompileShader(GLuint shader);
```

Elle ne prend qu'un seul paramètre : le shader à compiler. Nous l'appelons juste après avoir envoyé le code source :

Code : C++

```
// Récupération de la chaîne C du code source  
const GLchar* chaîneCodeSource = codeSource.c_str();  
  
// Envoi du code source au shader  
glShaderSource(shader, 1, &chaîneCodeSource, 0);
```

```
// Compilation du shader  
glCompileShader(shader);
```

Vérification de la compilation

La vérification de la compilation est certainement l'étape la plus importante et la plus chiante de cette partie. Imaginez que vous compiliez un code source et qu'il y ait une erreur à l'intérieur, comment pouvez-vous savoir où elle se trouve ? C'est un problème qui peut devenir vite énervant surtout quand votre erreur n'est qu'un bête oubli de point-virgule. 😠 Pour éviter d'avoir à relire vos codes sources à chaque fois, nous allons vérifier l'état de leur compilation grâce à quelques fonctions OpenGL.

La première de ces fonctions va nous permettre de renvoyer pas mal d'informations sur un shader donné, la vérification d'erreur au moment de la compilation en fait justement partie. Elle s'appelle `glGetShaderiv()` :

Code : C++

```
void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);
```

- **shader** : Comme toujours, le shader sur lequel on travaille
- **pname** : Le nom de l'information demandée
- **params** : L'adresse d'une variable qui accueillera cette information

Pour connaître l'état de la compilation, nous allons créer une variable de type **GLint** que l'on appellera **erreurCompilation**. Nous donnerons son adresse à la fonction `glGetShaderiv()` pour qu'elle puisse stocker l'information que l'on recherche. D'ailleurs, cette information se nomme **GL_COMPILE_STATUS** et sera la valeur du paramètre **pname** :

Code : C++

```
// Vérification de la compilation  
  
GLint erreurCompilation(0);  
  
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);
```

L'appel à cette fonction seule ne sert pas à grand chose, il faut maintenant vérifier la valeur de la variable **erreurCompilation**. 😊 Si elle différente de la constante **GL_TRUE** c'est qu'il y a eu une erreur, sinon c'est que tout va bien on peut retourner le booléen **true** pour clore le chargement :

Code : C++

```
// Vérification de la compilation  
  
GLint erreurCompilation(0);  
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);  
  
// S'il y a eu une erreur  
  
if(erreurCompilation != GL_TRUE)  
{  
}
```

```
// Sinon c'est que tout s'est bien passé  
  
else  
    return true;
```

Si la fonction `glGetShaderiv()` a détecté une erreur alors il faut la récupérer.

Malheureusement, nous ne pouvons pas utiliser les objets **string** pour cette récupération car les fonctions OpenGL ne gère pas les objets C++. Il faut donc faire à l'ancienne et allouer de la mémoire à la main pour une chaîne de caractère. 😞

Mais avant ça, nous devons connaître la taille du message d'erreur pour pouvoir allouer assez de mémoire. Pour ce faire, nous allons à nouveau utiliser la fonction `glGetShaderiv()` sauf que l'on demandera cette fois le paramètre **GL_INFO_LOG_LENGTH** qui correspond à la taille recherchée. Nous utiliserons la variable **tailleErreur** pour stocker la valeur qui sera retournée :

Code : C++

```
// Vérification de la compilation  
  
GLint erreurCompilation(0);  
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);  
  
// S'il y a eu une erreur  
  
if(erreurCompilation != GL_TRUE)  
{  
    // Récupération de la taille de l'erreur  
  
    GLint tailleErreur(0);  
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &tailleErreur);  
}  
  
// Sinon c'est que tout s'est bien passé  
  
else  
    return true;
```

Une fois que l'on connaît la taille de l'erreur, nous pouvons allouer de la mémoire pour une chaîne de caractère grâce au mot-clé **new[]**. On ajoute au passage 1 case pour gérer le caractère de fin de chaîne '\0' qui n'est pas fourni avec le message d'erreur :

Code : C++

```
// Vérification de la compilation  
  
GLint erreurCompilation(0);  
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);  
  
// S'il y a eu une erreur  
  
if(erreurCompilation != GL_TRUE)  
{  
    // Récupération de la taille de l'erreur  
  
    GLint tailleErreur(0);  
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &tailleErreur);  
  
    // Allocation de mémoire  
  
    char *erreur = new char[tailleErreur + 1];
```

```
    }

    // Sinon c'est que tout s'est bien passé

    else
        return true;
```

Maintenant que l'on a une chaîne allouée, nous pouvons récupérer la fameuse erreur. Nous utiliserons pour cela la fonction `glGetShaderInfoLog()`:

Code : C++

```
void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei
*length, GLchar *infoLog);
```

- **shader** : Le shader sur lequel on travaille
- **maxLength** : Taille de la chaîne de caractère qui va accueillir l'erreur. Pour nous, il s'agit de la variable **tailleErreur**
- **length** : Adresse de la variable qui contiendra la taille précédente. Je n'ai jamais compris l'utilité de ce paramètre mais bon, nous lui donnerons l'adresse de la variable **tailleErreur**
- **infoLog** : La chaîne de caractère qui contiendra le message final. Nous lui donnerons la chaîne **erreur**

L'appel à cette fonction ressemblera à ceci :

Code : C++

```
// Récupération de l'erreur

glGetShaderInfoLog(shader, tailleErreur, &tailleErreur, erreur);
```

Il faut également penser à rajouter le caractère de fin de chaîne '**\0**' pour compléter le message :

Code : C++

```
// Récupération de l'erreur

glGetShaderInfoLog(shader, tailleErreur, &tailleErreur, erreur);
erreur[tailleErreur] = '\0';
```

Ce qui donne au final :

Code : C++

```
// Vérification de la compilation

GLint erreurCompilation(0);
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);

// S'il y a eu une erreur

if(erreurCompilation != GL_TRUE)
{
    // Récupération de la taille de l'erreur
```

```
GLint tailleErreur(0);
glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &tailleErreur);

// Allocation de mémoire
char *erreur = new char[tailleErreur + 1];

// Récupération de l'erreur
glGetShaderInfoLog(shader, tailleErreur, &tailleErreur, erreur);
erreur[tailleErreur] = '\0';
}

// Sinon c'est que tout s'est bien passé

else
    return true;
```

On arrive à la fin courage. 😊

Il ne nous reste plus qu'à afficher le message d'erreur et à libérer la mémoire prise par la chaîne de caractère. On n'oublie pas de détruire le shader car celui-ci est inutilisable puis on retourne le booléen **false** pour terminer la méthode :

Code : C++

```
// Vérification de la compilation

GLint erreurCompilation(0);
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);

// S'il y a eu une erreur

if(erreurCompilation != GL_TRUE)
{
    // Récupération de la taille de l'erreur

    GLint tailleErreur(0);
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &tailleErreur);

    // Allocation de mémoire

    char *erreur = new char[tailleErreur + 1];

    // Récupération de l'erreur

    glGetShaderInfoLog(shader, tailleErreur, &tailleErreur, erreur);
    erreur[tailleErreur] = '\0';

    // Affichage de l'erreur

    std::cout << erreur << std::endl;

    // Libération de la mémoire et retour du booléen false

    delete[] erreur;
    glDeleteShader(shader);

    return false;
```

```
}

// Sinon c'est que tout s'est bien passé

else
    return true;
```

Pfiou on est enfin arrivé au bout ... Tout ce code pour vérifier une simple erreur ! Et oui, je vous avais prévenus que le chargement des shaders était quelque chose d'assez folklo. 😊

Faisons un petit récapitulatif final de la méthode **compilerShader()** :

Code : C++

```
bool Shader::compileShader(GLuint &shader, GLenum type, std::string
const &fichierSource)
{
    // Création du shader

    shader = glCreateShader(type);

    // Vérification du shader

    if(shader == 0)
    {
        std::cout << "Erreur, le type de shader (" << type << ")"
n'existe pas" << std::endl;
        return false;
    }

    // Flux de lecture

    std::ifstream fichier(fichierSource.c_str());

    // Test d'ouverture

    if(!fichier)
    {
        std::cout << "Erreur le fichier " << fichierSource << " est"
introuvable" << std::endl;
        glDeleteShader(shader);

        return false;
    }

    // Strings permettant de lire le code source

    std::string ligne;
    std::string codeSource;

    // Lecture

    while(getline(fichier, ligne))
        codeSource += ligne + '\n';

    // Fermeture du fichier

    fichier.close();
```

```
// Récupération de la chaîne C du code source
const GLchar* chaîneCodeSource = codeSource.c_str();

// Envoi du code source au shader
glShaderSource(shader, 1, &chaîneCodeSource, 0);

// Compilation du shader
glCompileShader(shader);

// Vérification de la compilation
GLint erreurCompilation();
glGetShaderiv(shader, GL_COMPILE_STATUS, &erreurCompilation);

// S'il y a eu une erreur
if(erreurCompilation != GL_TRUE)
{
    // Récupération de la taille de l'erreur
    GLint tailleErreur();
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &tailleErreur);

    // Allocation de mémoire
    char *erreur = new char[tailleErreur + 1];

    // Récupération de l'erreur
    glGetShaderInfoLog(shader, tailleErreur, &tailleErreur,
erreur);
    erreur[tailleErreur] = '\0';

    // Affichage de l'erreur
    std::cout << erreur << std::endl;

    // Libération de la mémoire et retour du booléen false
    delete[] erreur;
    glDeleteShader(shader);

    return false;
}

// Sinon c'est que tout s'est bien passé
else
    return true;
}
```

Grâce à tout ce code, nous sommes maintenant capables de compiler tous nos shaders. Il valait mieux séparer ce code de la méthode **charger()** sinon il aurait fallu coder ça deux, imaginez un peu le truc ! 😊

Cependant nous n'avons pas encore fini, il nous faut encore terminer la méthode **charger()**. Continuons. 😊

Création du programme

La méthode charger (Partie 2/2)

Création du programme

Maintenant que nous avons codé une méthode permettant de compiler un shader, nous pouvons tranquillement retourner à la méthode **charger()** que nous avons laissée précédemment. D'ailleurs, elle est un peu vide pour le moment :

Code : C++

```
bool Shader::charger()
{
    // Compilation des shaders

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,
m_vertexSource))
        return false;

    if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,
m_fragmentSource))
        return false;
}
```

Grâce à ce mini bout de code, nous avons enfin notre duo de shader compilé et validé. Cependant, ces shaders ne sont pas encore utilisables pour le moment car ce ne sont que des objets intermédiaires qui ne peuvent pas être exécutés par la carte graphique, elle ne sait pas quoi faire avec. Pour régler le problème, il nous faut les réunir à l'intérieur d'un **programme** qui, lui, sera exécutable par la carte.

Bien entendu, il ne s'agit pas d'un programme classique que vous avez l'habitude d'exécuter sur votre ordinateur. 🤖 C'est en fait un *objet OpenGL* qui va faire la passerelle entre les shaders et leur exécution au sein de la carte graphique. C'est pour cette raison que nous avons besoin du fameux attribut **programID** - attribut qu'on utilise depuis un bout de temps maintenant dans la fonction **glUseProgram()**.

Pour la suite de la méthode **charger()**, il nous faut donc créer ce programme. Nous ferons cela grâce à une fonction qui ressemble fortement à **glCreateShader()** et qui s'appelle **glCreateProgram()** :

Code : C++

```
GLuint glCreateProgram();
```

C'est l'une des rares fonctions OpenGL qui ne demande aucun paramètre, profitez-en. 🎉 Elle retourne juste l'**ID** d'un nouveau programme.

Nous l'appellerons donc pour créer un programme qui sera accessible via l'attribut **m_programID** :

Code : C++

```
bool Shader::charger()
{
    // Compilation des shaders

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,
m_vertexSource))
        return false;
```

```

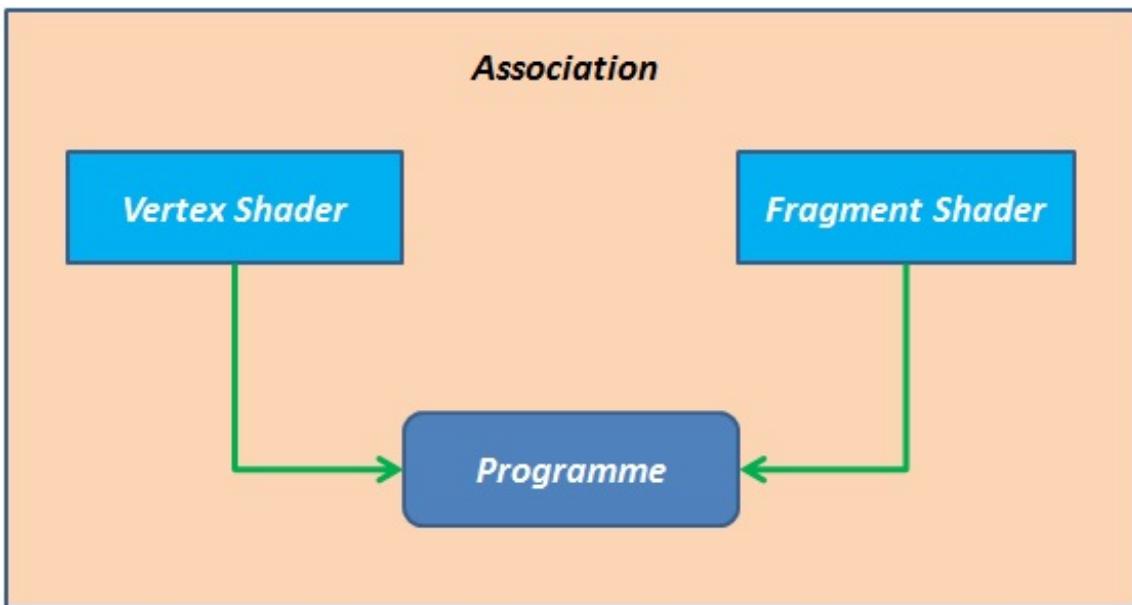
if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,
m_fragmentSource))
    return false;

// Création du programme
m_programID = glCreateProgram();
}

```

Association des shaders au programme

Comme je vous l'ai dit précédemment, le **Vertex** et **Fragment** Shader sont inutilisables en l'état car ils ne peuvent pas être exécutés par la carte graphique. En revanche, le programme lui peut l'être. Il nous faut associer les shaders avec celui-ci de façon à ce que nos fichiers compilés puissent servir à quelque chose.



Cette étape d'association se fait très simplement puisqu'il suffit d'utiliser une seule fonction OpenGL : **glAttachShader()**.

Code : C++

```
void glAttachShader(GLuint program, GLuint shader)
```

- **program** : Le programme sur lequel on travaille
- **shader** : Le shader à associer

Nous appellerons cette fonction deux fois pour associer nos deux shaders :

Code : C++

```

bool Shader::charger()
{
    // Compilation des shaders

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,

```

```
m_vertexSource))
    return false;

    if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,
m_fragmentSource))
        return false;

    // Création du programme
    m_programID = glCreateProgram();

    // Association des shaders
    glAttachShader(m_programID, m_vertexID);
    glAttachShader(m_programID, m_fragmentID);
}
```

Étape terminée. 😊

Vérouillage des entrées shader

Si je vous parle d'entrées shader, ça vous dit quelque chose ? Normalement vous devriez me répondre non. 😊 Et pourtant si je vous disais que vous savez déjà ce que c'est, vous me croiriez ?

En fait, les entrées shader sont tout simplement les tableaux **Vertex Attrib** que l'on utilise depuis le début du tuto ! 😊 Et oui, les shaders ont besoin d'eux pour travailler sinon ils ne pourraient pas afficher grand chose. Les tableaux **Vertex Attrib** constituent donc leurs sources de données (vertices, couleurs, ...) que l'on appelle plus communément leurs entrées. On parle bien d'**entrées** parce qu'il existe aussi des **sorties**, mais ça sera pour le chapitre suivant.

Le verrouillage des entrées consiste simplement à dire au shader :

- Le tableau **Vertex Attrib 0** correspondra aux **vertices**
- Le tableau **Vertex Attrib 1** correspondra aux **couleurs**
-

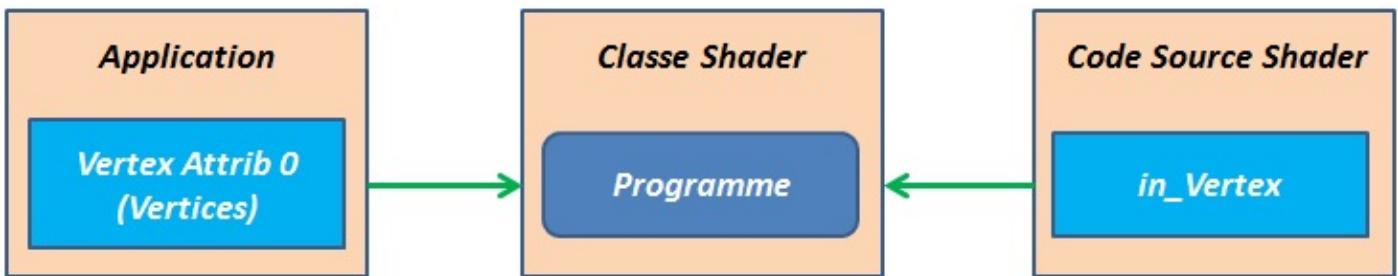
Pour faire ça, nous devons utiliser la fonction **glBindAttribLocation()** :

Code : C++

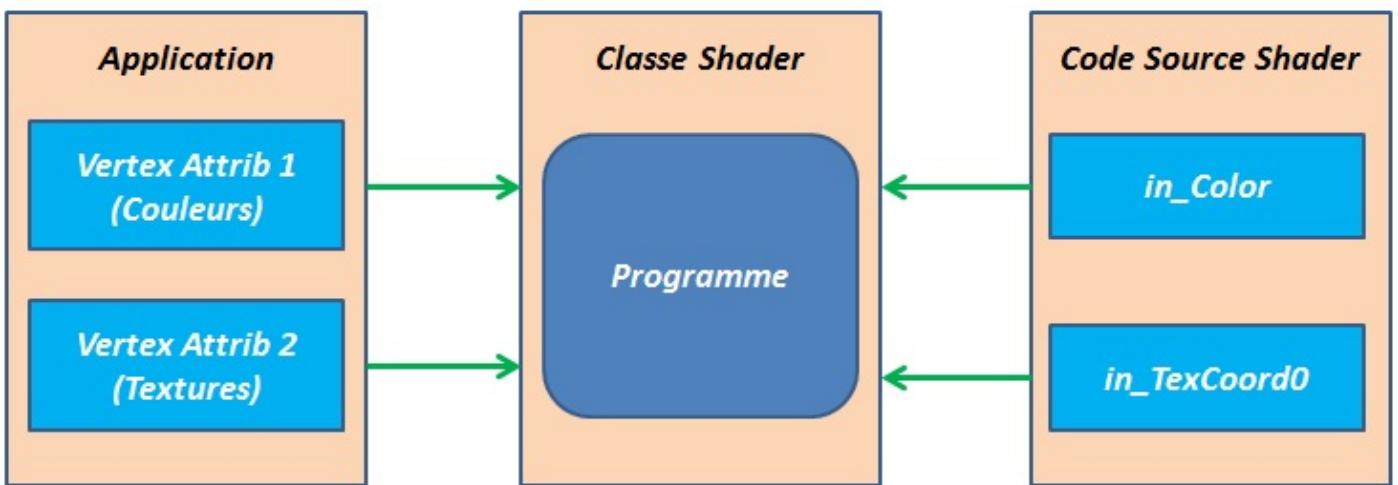
```
void glBindAttribLocation(GLuint program, GLuint index, const GLchar
*name);
```

- **program** : Le programme sur lequel on travaille
- **index** : Le numéro du tableau **Vertex Attrib** à verrouiller
- **name** : Le nom de la donnée dans le code source du shader

Le dernier paramètre peut vous sembler un peu confus pour le moment mais sachez juste qu'il s'agit d'une variable, dans le code source du shader, qui permet d'accéder aux données des tableaux **Vertex Attrib**. Par exemple, pour accéder aux *vertices* dans les shaders, nous utiliserons une variable qui s'appellera **in_Vertex** :



Pour les couleurs ce sera **in_Color** et pour les coordonnées de texture ce sera **in_TexCoord0** :



Vu que nous utilisons 3 tableaux **VertexAttrib** (vertices, couleurs et coordonnées de texture), nous devrons utiliser 3 fois la fonction **glBindAttribLocation()** pour verrouiller toutes les entrées. Pour le tableau d'indice **0**, l'appel ressemblera à ceci :

Code : C++

```
// Verrouillage des entrées shader (Vertices)
glBindAttribLocation(m_programID, 0, "in_Vertex");
```

Pour les deux autres, ce sera :

Code : C++

```
// Verrouillage des entrées shader (Couleurs et Coordonnées de
texture)

glBindAttribLocation(m_programID, 1, "in_Color");
glBindAttribLocation(m_programID, 2, "in_TexCoord0");
```



Euh j'ai une petite question, pourquoi il y a un **0** à la fin de la variable **in_TexCoord0** ?

Ça peut paraître un peu troublant mais le **0** est là pour indiquer au shader que les coordonnées de texture qu'il reçoit correspondent simplement à la première texture verrouillée (indice **0**). Quand nous utiliserons le multitexturing, nous enverrons

plusieurs textures au shader et celui-ci devra faire la différence entre les coordonnées reçues pour la première, celles reçues pour la deuxième, etc. Le chiffre de la variable `in TexCoord` permet de faire cette différence. Enfin, ne vous triturez pas la tête avec ça pour le moment, ce n'est pas important. 😊

Au final, nous plaçons les 3 appels à la fonction `glBindAttribLocation()` juste après l'association des shaders :

Code : C++

```
bool Shader::charger()
{
    // Compilation des shaders

    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,
m_vertexSource))
        return false;

    if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,
m_fragmentSource))
        return false;

    // Création du programme
    m_programID = glCreateProgram();

    // Association des shaders
    glAttachShader(m_programID, m_vertexID);
    glAttachShader(m_programID, m_fragmentID);

    // Verrouillage des entrées shader
    glBindAttribLocation(m_programID, 0, "in_Vertex");
    glBindAttribLocation(m_programID, 1, "in_Color");
    glBindAttribLocation(m_programID, 2, "in_TexCoord0");
}
```

Linkage

Le linkage constitue l'étape finale de la création d'un programme. Jusqu'à présent, nous avons compilé nos shaders séparément puis nous les avons associés à notre programme. Il ne reste plus maintenant qu'à le finaliser pour le rendre exécutable par la carte graphique.

Cette étape se fait grâce à la fonction `glLinkProgram()` :

Code : C++

```
void glLinkProgram(GLuint program);
```

Bien entendu, le paramètre `program` correspond au programme à linker. On appelle cette fonction juste après le verrouillage des entrées shader :

Code : C++

```
// Verrouillage des entrées shader
glBindAttribLocation(m_programID, 0, "in_Vertex");
```

```
glBindAttribLocation(m_programID, 1, "in_Color");
glBindAttribLocation(m_programID, 2, "in_TexCoord0");

// Linkage du programme

glLinkProgram(m_programID);
```

Vérification du linkage

Le linkage d'un programme est un processus qui peut malheureusement échouer, nous devons donc vérifier si tout s'est bien passé. La mauvaise nouvelle, c'est qu'il va falloir écrire un gros bloc de vérification comme pour la compilation de shader. 😱

L'avantage cependant c'est que cet énorme bloc de vérification ressemble exactement au premier, seules quelques fonctions OpenGL vont devoir changer. Ainsi, pour vérifier si le linkage s'est bien passé nous devrons :

- **Vérifier** s'il y a eu un problème
- Si oui, **récupérer** la taille du message d'erreur
- **Allouer** une chaîne de caractère grâce à cette taille
- **Récupérer** l'erreur et l'afficher

Pour la compilation de shader, nous avions utilisé la fonction **glGetShaderiv()** pour savoir s'il y avait eu une erreur. Pour le linkage d'un programme, la fonction est exactement identique, il n'y a que le nom qui change :

Code : C++

```
void glGetProgramiv(GLuint program, GLenum pname, GLint *params);
```

Les paramètres sont également identiques :

- **program** : Le programme sur lequel on travaille
- **pname** : Le nom de l'information demandée
- **params** : L'adresse d'une variable qui accueillera cette information

On doit utiliser cette fonction, dans un premier temps, pour vérifier s'il y a eu une erreur au moment du linkage. Le nom du paramètre **pname** qui nous permet cela sera la constante **GL_LINK_STATUS**. On doit aussi créer une variable de type **GLint** pour contenir la valeur renournée, on l'appellera **erreurLink**. On en profite au passage pour implémenter que le début du bloc **if** :

Code : C++

```
// Linkage du programme

glLinkProgram(m_programID);

// Vérification du linkage

GLint erreurLink(0);
glGetProgramiv(m_programID, GL_LINK_STATUS, &erreurLink);

// S'il y a eu une erreur

if(erreurLink != GL_TRUE)
{
```

```
// Sinon c'est que tout s'est bien passé  
  
else  
    return true;
```

Si on se retrouve dans le bloc **if**, c'est que quelque chose s'est mal passé. Nous devons donc récupérer la taille du message d'erreur grâce à la fonction **glGetProgramiv()**. Elle prendra exactement les mêmes paramètres que la fonction **glGetShaderiv()** soient : une constante **GL_INFO_LOG_LENGTH** et l'adresse d'une variable **tailleErreur** de type **GLint** :

Code : C++

```
// Vérification du linkage  
  
if(erreurLink != GL_TRUE)  
{  
    // Récupération de la taille de l'erreur  
  
    GLint tailleErreur(0);  
    glGetProgramiv(m_programID, GL_INFO_LOG_LENGTH, &tailleErreur);  
}
```

Une fois la taille de l'erreur récupérée, on peut allouer une chaîne de caractère pour la contenir sans oublier le caractère '**\0**' :

Code : C++

```
// Vérification du linkage  
  
if(erreurLink != GL_TRUE)  
{  
    // Récupération de la taille de l'erreur  
  
    GLint tailleErreur(0);  
    glGetProgramiv(m_programID, GL_INFO_LOG_LENGTH, &tailleErreur);  
  
    // Allocation de mémoire  
  
    char *erreur = new char[tailleErreur + 1];  
}
```

Maintenant que l'on a une chaîne de caractère assez grande, on peut récupérer le message d'erreur grâce à une fonction très similaire à **glGetShaderInfoLog()** qui s'appelle **glGetProgramInfoLog()** :

Code : C++

```
void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei  
*length, GLchar *infoLog);
```

Je vous rappelle que cette fonction permet de récupérer le message d'erreur. 😊 D'ailleurs, elle prend elle-aussi les mêmes paramètres que sa sœur-jumelle :

- **program** : Le programme sur lequel on travaille
- **maxLength** : Taille de la chaîne de caractère qui va accueillir l'erreur. Pour nous, il s'agit de la variable **tailleErreur**
- **length** : Adresse de la variable qui contiendra la taille précédente. Ce paramètre est encore une fois bizarre, nous lui donnerons l'adresse de la variable **tailleErreur**
- **infoLog** : La chaîne de caractère qui contiendra le message final. Nous lui donnerons la chaîne **erreur**

Nous appelons donc cette fonction avec les paramètres que nous venons juste de citer. N'oubliez pas de rajouter le caractère '\0' à la fin de la chaîne :

Code : C++

```
// Vérification du linkage

if(erreurLink != GL_TRUE)
{
    // Récupération de la taille de l'erreur

    GLint tailleErreur(0);
    glGetProgramiv(m_programID, GL_INFO_LOG_LENGTH, &tailleErreur);

    // Allocation de mémoire

    char *erreur = new char[tailleErreur + 1];

    // Récupération de l'erreur

    glGetShaderInfoLog(m_programID, tailleErreur, &tailleErreur,
erreur);
    erreur[tailleErreur] = '\0';
}
```

Pour terminer cette gestion d'erreur, il ne reste plus qu'à afficher le message tant convoité. Une fois que c'est fait, on libère la mémoire prise par la chaîne de caractère puis on détruit le programme vu que celui-ci est inutilisable.

La fonction permettant de détruire un programme s'appelle simplement **glDeleteProgram()** :

Code : C++

```
void glDeleteProgram(GLuint program);
```

Elle prend en paramètre le programme à détruire.

On appelle donc cette fonction en plus du mot-clé **delete[]** pour détruire la chaîne. On renverra également le booléen **false** pour indiquer que le chargement s'est mal passé :

Code : C++

```
// Vérification du linkage

if(erreurLink != GL_TRUE)
{
    // Récupération de la taille de l'erreur

    GLint tailleErreur(0);
    glGetProgramiv(m_programID, GL_INFO_LOG_LENGTH, &tailleErreur);

    // Allocation de mémoire

    char *erreur = new char[tailleErreur + 1];

    // Récupération de l'erreur
```

```
glGetShaderInfoLog(m_programID, tailleErreur, &tailleErreur,  
erreur);  
erreur[tailleErreur] = '\0';  
  
// Affichage de l'erreur  
  
std::cout << erreur << std::endl;  
  
// Libération de la mémoire et retour du booléen false  
  
delete[] erreur;  
glDeleteProgram(m_programID);  
  
return false;  
}
```

Récapitulatif de la méthode **charger()**:

Code : C++

```
bool Shader::charger()  
{  
    // Compilation des shaders  
  
    if (!compilerShader(m_vertexID, GL_VERTEX_SHADER,  
m_vertexSource))  
        return false;  
  
    if (!compilerShader(m_fragmentID, GL_FRAGMENT_SHADER,  
m_fragmentSource))  
        return false;  
  
    // Création du programme  
  
    m_programID = glCreateProgram();  
  
    // Association des shaders  
  
    glAttachShader(m_programID, m_vertexID);  
    glAttachShader(m_programID, m_fragmentID);  
  
    // Verrouillage des entrées shader  
  
    glBindAttribLocation(m_programID, 0, "in_Vertex");  
    glBindAttribLocation(m_programID, 1, "in_Color");  
    glBindAttribLocation(m_programID, 2, "in_TexCoord0");  
  
    // Linkage du programme  
  
    glLinkProgram(m_programID);  
  
    // Linkage du programme  
  
    glLinkProgram(m_programID);  
  
    // Vérification du linkage  
  
    GLint erreurLink(0);
```

```
glGetProgramiv(m_programID, GL_LINK_STATUS, &erreurLink);

// S'il y a eu une erreur

if(erreurLink != GL_TRUE)
{
    // Récupération de la taille de l'erreur

    GLint tailleErreur(0);
    glGetProgramiv(m_programID, GL_INFO_LOG_LENGTH,
&tailleErreur);

    // Allocation de mémoire

    char *erreur = new char[tailleErreur + 1];

    // Récupération de l'erreur

    glGetShaderInfoLog(m_programID, tailleErreur, &tailleErreur,
erreur);
    erreur[tailleErreur] = '\0';

    // Affichage de l'erreur

    std::cout << erreur << std::endl;

    // Libération de la mémoire et retour du booléen false

    delete[] erreur;
    glDeleteProgram(m_programID);

    return false;
}

// Sinon c'est que tout s'est bien passé

else
    return true;
}
```

Cette fois la méthode est bel et bien terminée, et je dirai même que l'implémentation de la classe **Shader** est également terminée. Vous êtes maintenant en mesure de comprendre tout le code C++ de vos applications, plus de classe mystère !

Après toutes ces péripéties, vous avez bien mérité une petite pause moi j'dis. 

D'ailleurs, vous pouvez continuer à lire la dernière partie de ce chapitre tout en faisant une pause car nous n'allons voir que quelques points mineurs. 

Utilisation et améliorations

Utilisation

La partie **utilisation** est vraiment quelque chose d'anecdotique car vous savez déjà comment utiliser un shader, mais nous allons tout de même faire un petit point dessus. Après tout, les rappels ça n'a jamais fait de mal. 

Enfin bref. Pour créer un objet de type **Shader**, vous devez simplement déclarer un nouvel objet en lui donnant en paramètre le chemin vers deux codes sources : celui du **Vertex Shader** et celui du **Fragment Shader**, puis d'appeler la méthode **charger()**. Par exemple, pour charger le shader de texture :

Code : C++

```
// Shader Texture

Shader shaderTexture("Shaders/texture.vert",
"Shaders/texture.frag");
shaderTexture.charger();
```

Une fois chargé, vous n'avez plus qu'à l'activer au moment d'*afficher* votre modèle grâce à la fonction **glUseProgram()** :

Code : C++

```
void glUseProgram(GLuint program);
```

La paramètre **program** correspond évidemment au shader à activer.

Cette fonction est vraiment similaire aux fonctions du type **glBindXXX()** puisqu'elle permet de dire à OpenGL : "Je souhaite utiliser ce shader-ci pendant toute la durée où il est activé". Ainsi, tous les modèles qui se trouvent à l'intérieur des deux appels à **glUseProgram()** seront obligés de l'utiliser pour être affichés.

Code : C++

```
// Activation du shader

glUseProgram(m_shader.getProgramID());

// Verrouillage du VAO

glBindVertexArray(m_vaoID);

// Envoi des matrices

glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));

// Verrouillage de la texture

glBindTexture(GL_TEXTURE_2D, m_texture.getID());

// Rendu

glDrawArrays(GL_TRIANGLES, 0, 36);

// Déverrouillage de la texture

glBindTexture(GL_TEXTURE_2D, 0);

// Déverrouillage du VAO

glBindVertexArray(0);
```

```
// Désactivation du shader  
glUseProgram(0);
```

D'ailleurs en parlant de ça, je vous conseille fortement d'activer votre shader avant de verrouiller tous vos autres *objets OpenGL*. Si vous ne le faites pas, vous risquerez d'envoyer des données dans le vide (comme les matrices ou les textures) car aucun shader ne sera là pour les recevoir. 😊



Les dernières lignes de code C++ qui peuvent vous gêner sont celles de l'envoi des matrices. Nous allons les voir dans le prochain chapitre car elles font partie intégrante de l'utilisation des shaders. 😊

Améliorations

Le destructeur

Dans cette petite partie **améliorations**, nous allons surtout parler de méthodes qui seront utiles si nous voulons copier des objets de type **Shader** entre eux. Mais avant ça, nous allons parler un peu de libération de mémoire car jusqu'à maintenant, lorsqu'un objet **Shader** est détruit, il ne libère à aucun moment les ressources qu'il a prises.

Pour l'aider à le faire, nous allons simplement détruire tous les attributs dans le destructeur. Sachant que les **string** savent le faire toutes seules, il ne reste plus qu'à gérer les attributs **m_vertexID**, **m_fragmentID** et **m_programID**. Les fonctions permettant de détruire ces objets OpenGL sont **glDeleteShader()** et **glDeleteProgram()**.

Nous les appelons donc le destructeur :

Code : C++

```
Shader::~Shader()  
{  
    // Destruction du shader  
  
    glDeleteShader(m_vertexID);  
    glDeleteShader(m_fragmentID);  
    glDeleteProgram(m_programID);  
}
```

Comme quoi, le destructeur ne reste pas vide à chaque fois. 😊

Le constructeur de copie

On continue cette dernière partie avec le *constructeur de copie*. J'espère que vous savez ce que c'est quand même, surtout qu'on l'utilise depuis un moment maintenant. 😊

En temps normal, il est inutile de le coder tant qu'on n'utilise pas de pointeurs car seuls eux peuvent poser des problèmes. Cependant, les *objets OpenGL* peuvent être rapprochés aux pointeurs car leur **ID** se comporte un peu de la même façon. Si vous essayez de copier un **ID** dans un autre objet, alors les deux copies auront accès exactement aux mêmes données alors qu'ils devraient avoir chacun les leurs.

Il nous faut ré-écrire ce constructeur pour pouvoir être capables de copier deux shaders correctement sans avoir à nous soucier de ce problème. Voici d'ailleurs le prototype de cette pseudo-méthode :

Code : C++

```
Shader(Shader const &shaderACopier);
```



La référence constante est importante ici, ne l'oubliez pas.

Pour la classe **Shader**, nous avons 5 attributs à gérer. Les deux **strings** peuvent être copiées directement sans problème :

Code : C++

```
Shader::Shader(Shader const &shaderACopier)
{
    // Copie des fichiers sources

    m_vertexSource = shaderACopier.m_vertexSource;
    m_fragmentSource = shaderACopier.m_fragmentSource;
}
```

Les 3 derniers attributs concernent les shaders et le programme. Ils ne peuvent pas être copiés de la même façon, il va falloir re-générer de nouvelles valeurs (de nouveaux **ID**) à partir des fichiers sources. Pour ce faire, il suffit juste d'appeler la méthode **charger()** car c'est elle qui s'occupe de ça :

Code : C++

```
Shader::Shader(Shader const &shaderACopier)
{
    // Copie des fichiers sources

    m_vertexSource = shaderACopier.m_vertexSource;
    m_fragmentSource = shaderACopier.m_fragmentSource;

    // Chargement du nouveau shader
    charger();
}
```

Le constructeur de copie est terminé. 😊

L'opérateur =

L'opérateur **d'affectation** = est un opérateur très pratique en C++ car il permet de copier un objet directement en utilisant le symbole = comme si on faisait une banale opération arithmétique. Nous allons l'implémenter, au même titre que le constructeur de copie, dans le cas où nous aurions à copier un shader.

Voici son prototype :

Code : C++

```
Shader& operator=(Shader const &shaderACopier);
```



Cette méthode prend et renvoie une référence constante.

Son contenu est strictement identique au constructeur de copie car elle fait exactement la même chose. 🤔 On rajoutera juste le retour du pointeur ***this** pour retourner proprement notre objet :

Code : C++

```
Shader& Shader::operator=(Shader const &shaderACopier)
{
    // Copie des fichiers sources

    m_vertexSource = shaderACopier.m_vertexSource;
    m_fragmentSource = shaderACopier.m_fragmentSource;

    // Chargement du nouveau shader
    charger();

    // Retour du pointeur this
    return *this;
}
```

Encore une fois, je vous repose la question pour ce chapitre (🤔) : vous vous souvenez de ce qui se passait si on chargeait deux fois un même objet OpenGL (texture, VBO, etc.) d'un coup ? Que cela entraînait une fuite de mémoire ? Et bien nous avons encore le même problème ici. Si vous chargez deux fois un shader, lors d'une copie par exemple, alors vous risquez de gâcher de la mémoire qui ne pourra pas être libérée avant la fermeture de votre application.

Pour éviter cela, nous devons appeler les fonctions de vérification d'objet OpenGL du type **glIsXXX()**, puis de les détruire si besoin est. Elles s'appellent ici **glIsShader()** et **glIsProgram()** :

Code : C++

```
// Vérification de shader
GLboolean glIsShader(GLuint shader);

// Vérification de programme
GLboolean glIsProgram(GLuint program);
```

Elles prennent en paramètre respectivement un identifiant de shader et un identifiant de programme pour savoir s'ils ont déjà été chargés. Elles renvoient toutes les deux la valeur **GL_TRUE** si c'est le cas, sinon c'est **GL_FALSE**.

Au final, pour les attributs **m_vertexID**, **m_fragmentID** et **m_programID**, nous devons faire un bloc **if** en vérifiant la valeur renournée par la fonction **glIsXXX()**. Si c'est **GL_TRUE**, alors on appelle leur fonction de destruction :

Code : C++

```
bool Shader::charger()
{
    // Destruction d'un éventuel ancien Shader
```

```
if(glIsShader(m_vertexID) == GL_TRUE)
    glDeleteShader(m_vertexID);

if(glIsShader(m_fragmentID) == GL_TRUE)
    glDeleteShader(m_fragmentID);

if(glIsProgram(m_programID) == GL_TRUE)
    glDeleteProgram(m_programID);

// Compilation des shaders

if(!compilerShader(vertexShader, GL_VERTEX_SHADER,
m_vertexSource)
    return false;

if(!compilerShader(fragmentShader, GL_FRAGMENT_SHADER,
m_fragmentSource)
    return false;

    ...
}
```

Grâce à ce bout de code, on évite toute fuite de mémoire car les éventuels anciens **ID** seront détruits avant d'être ré-initialisés.

Télécharger (Windows, UNIX/Linux, Mac OS X) : [Code Source C++ du chapitre sur la compilation de Shader](#)

A travers ce premier chapitre consacré aux *shaders*, nous avons pu voir comment compiler n'importe quel code source. Tous ceux que nous avons utilisés depuis la partie 1 sont passés par cette classe - classe dont nous sommes enfin capables de comprendre le fonctionnement.

Les notions que nous avons abordées sont assez costauds au niveau technique, je vous conseille de relire à tête reposée ceux qui vous paraissent encore un peu sombres.

Néanmoins, après le code de ce pseudo-compilateur, nous pouvons enfin passer aux codes sources mêmes des *shaders*. Les 3 prochains chapitres leur seront entièrement consacrés. Refait le stock de café ou de chocolat moi j'dis ! 

OpenGL Shading Language

Dans le chapitre précédent, nous avons vu ce qu'étaient les **shaders** et comment les implémenter au sein de nos applications. Nous avons même écrit, en quelque sorte, notre propre compilateur qui permet désormais de compiler absolument tous leurs codes sources. 

Aujourd'hui, nous allons justement passer aux codes sources mêmes et étudier le langage de programmation avec lequel ils sont faits : l'**OpenGL Shading Language**. Nous nous concentrerons principalement sur sa syntaxe qui se rapproche fortement de celle du C mais qui contient tout de même quelques différences.

Si vous vous sentez prêts alors on peut commencer. 

Un peu d'histoire L'histoire des shaders

La création

Ce qu'il faut savoir avec les **shaders**, c'est que leur intégration au sein des jeux-vidéo est une chose assez récente par rapport à leur création. En effet, ceux-ci ont été créés en **1988** par les studios d'animation **Pixar** qui avaient remarqué qu'il était possible de traiter tous les pixels un par un pour leur appliquer des instructions spécifiques. Cela leur permettait de manipuler leurs rendus non seulement pour les améliorer mais aussi pour les rendre plus réalistes sans surcharger les ressources des ordinateurs.

Le premier standard à utiliser les **shaders** est le **RenderMan (RIS)** développé par les studios du même nom. Depuis, ils se sont développés dans tous les domaines qui traitent la 3D tels que les films d'animation, le cinéma et évidemment les jeux-vidéo. Ils permettent aujourd'hui d'exploiter efficacement la carte graphique pour effectuer une multitude de calculs mathématiques. Leur plus grand intérêt est évidemment la réalisation d'effets réalistes que l'on retrouve aujourd'hui dans tous les jeux-vidéo à la mode. 

Au niveau de leur fonctionnement, vous savez maintenant que les **shaders** sont des petits programmes qui vont traiter les vertices et les pixels. Ils sont spécialement conçus pour faire ce genre d'opération, ils n'ont donc pas peur de traiter des millions de données à chaque seconde. 

Les différents langages

Si les **shaders** sont des programmes alors ils possèdent forcément un code source pour savoir ce qu'ils doivent faire. Et évidemment, ces codes sources doivent être écrits avec un langage de programmation. Depuis 1988, plusieurs entreprises ont tenté de sortir leur propre langage soit pour s'imposer dans le domaine, soit pour un type d'utilisation spécifique. Certains sont plus ou moins connus et même plus ou moins compliqués à utiliser.

Il existe deux types de langage utilisables pour la programmation des shaders. Jusqu'à récemment, il fallait en utiliser certains qui étaient proches de l'**Assembleur**, ils étaient donc considérés comme étant de **bas-niveau**. Pour ceux qui ne savent pas ce qu'est l'**Assembleur**, sachez qu'il s'agit d'un langage de programmation extrêmement difficile à manier pour les non-initiés. Si vous voulez faire un jeu-vidéo 3D avec ça je vous conseille de vous lever très tôt chaque matin pour avancer dans le développement. 

Mais heureusement pour nous, il existe aussi des langages dits de **haut-niveau**, donc plus faciles à utiliser, qui se rapprochent fortement de ceux que l'on connaît aujourd'hui. Nous pouvons en citer principalement 3 :

- **OpenGL Shading Language (GLSL)** : créé par le consortium [Architecture Review Board](#) qui gère *OpenGL*, il est évidemment compatible avec cet API et en est même le standard actuel
- **C for Graphics (Cg)** : créé par Nvidia, il est compatible non seulement avec *OpenGL* mais aussi avec son concurrent Microsoft *Direct3D*
- **High Level Shader Language (HLSL)** : créé par Microsoft, il est uniquement compatible avec *Direct3D*

Ces langages ne sont pas compatibles avec toutes les API 3D mais ils sont tous utilisés dans le monde du jeu-vidéo. Vu que nous, nous programmons avec *OpenGL*, nous allons donc utiliser l'**OpenGL Shading Language (GLSL)** pour développer nos shaders.

OpenGL Shading Language

Le **GLSL** est donc un langage de programmation créé par l'[Architecture Review Board](#) pour permettre aux programmeurs d'utiliser directement la puissance des cartes graphiques sans passer par l'ancien *pipeline 3D*. Cela permet de gagner en rapidité car on évite tous les calculs superflus qui existaient auparavant. On gagne également en flexibilité car on peut traiter chaque pixel individuellement.

L'avantage de ce langage c'est que d'une part il est multiplateforme, c'est-à-dire qu'il est supporté par la majorité des cartes graphiques, et d'autre part sa syntaxe se rapproche énormément de celle du **C**. Il est plus facile d'apprendre à l'utiliser lui plutôt que d'apprendre l'*Assembleur*. 

Malgré sa notoriété, le **GLSL** n'est né que très récemment. Comparé à la création des shaders qui date de 1988, il fait office de bambin puisqu'il n'a été accepté officiellement qu'en 2004 avec la sortie d'*OpenGL 2.0*. Pour information, la première version d'*OpenGL* est sortie en 1992, ce qui fait 12 ans de décalage entre les deux ! 

Cependant, durant sa courte histoire, le **GLSL** n'a pas chômé puisqu'il a fourni pas moins de 10 versions en l'espace de 8 ans. En voici quelques unes :

- **1.20** : sortie avec *OpenGL 2.1*, c'est la version la plus connue à ce jour.
- **1.30** : sortie avec *OpenGL 3.0*, elle introduit une **nouvelle façon de programmer** et déprécie certaines fonctionnalités (même s'il reste toujours possible de les utiliser).
- **1.40** : sortie avec *OpenGL 3.1*. Cette fois-ci, les fonctionnalités dépréciées sont totalement supprimées, la **nouvelle façon de programmer** devient le nouveau standard.
- **1.50** : sortie avec *OpenGL 3.2*, nous utiliserons celle-ci car c'est la plus haute version avec laquelle nous pourrons programmer de façon '*multiplateforme*'. Au-delà, certains OS comme Mac OS X ne supportent plus le langage.

Contrairement à ce qu'on pourrait penser, il n'existe pas de version 1.60, 1.70 ... A partir d'*OpenGL 3.3*, le numéro des versions devient le même que celui de l'API. Pour *OpenGL 3.3* par exemple, le **GLSL** passe en 3.30, pour la 4.0 il passe en 4.0, ...

Enfin, vous savez maintenant ce que sont les langages de programmation shader et vous avez même eu le droit à un petit cours d'histoire pour votre culture générale. 

Pour la suite du tutoriel, nous utiliserons le **GLSL** dans sa version **1.50**. C'est avec lui que nous serons bientôt capables de faire des effets réalistes que l'on pourra intégrer dans nos scènes 3D. D'ailleurs, nous allons tout de suite passer à l'étude de ce langage en commençant par sa syntaxe de base.

Les variables

Le code source minimal

Comme nous l'avons vu dans l'introduction, l'**OpenGL Shading Language** est fortement inspiré du **C**, vous n'avez donc pas à être effrayés par l'apprentissage d'un nouveau langage de programmation.

Dans cette partie, nous nous concentrerons principalement sur sa syntaxe avec la gestion des variables, les fonctions, etc. Je ferai en sorte de vous montrer des exemples à chaque fois et nous en profiterons pour faire quelques exercices.

Je n'ai malheureusement pas d'IDE à vous proposer pour coder vos shaders. Par habitude, j'utilise toujours celui pour coder les applications classiques et je ne connais que très peu les autres pour pouvoir vous les conseiller efficacement. Si vous voulez ne pas être embêtés à ouvrir plusieurs programmes, je vous propose de coder également sur le même IDE que vous utilisez en temps normal.

Enfin maintenant que l'on sait tout ce que l'on à savoir, nous pouvons enfin commencer l'étude du **GLSL**. La première chose que nous allons voir concerne le code source minimal demandé par un shader pour pouvoir être compilé. Celui-ci est assez simple mais mérite tout de même quelques explications :

Code : C

```
void main()
{
}
```

Un peu cours n'est-ce pas ? 😊

Le code minimal contient donc simplement une fonction qui **main()** qui ne prend et qui ne renvoie aucun paramètre. Remarquez que la syntaxe des fonctions est strictement identiques au **C**. Si vous essayez de compiler ce code vous n'aurez aucun message d'erreur, le shader l'acceptera sans problème mais il n'affichera pas grand chose.

La fonction **main()** a la même utilité qu'en **C**, c'est elle qui est appelée en première au lancement d'un programme. Nous placerons donc nos premières futures instructions à l'intérieur. Cependant, contrairement au **C** cette fonction n'a pas besoin de renvoyer une variable pour indiquer si tout s'est bien passé (le fameux **return 0;**).

En définitif, ce code constitue la source minimale pour compiler un shader, nous devrons donc implémenter la fonction **main()** à chaque fois. 😊

Les Variables

Les variables 'classiques'

Comme tout tutoriel sur les langages de programmation, nous allons nous attaquer en premier lieu aux *variables*.

Il existe plusieurs types de variable avec le **GLSL** dont certains sont assez déroutants car ils n'ont pas leur équivalent en **C**. On va commencer par voir ceux qu'on a l'habitude d'utiliser car ce sont les plus simples à comprendre :

- **bool** : booléen pouvant prendre la valeur **true** ou **false**
- **int** : les nombres entiers
- **uint** : les nombres entiers non-signés
- **float** : les nombres flottants (décimaux)



Les types **char**, **short**, **long** et **double** n'existent pas. Ne cherchez donc pas à les utiliser ou votre pseudo-compilateur va vous râler dessus. 😠

L'avantage des variables en **GLSL** c'est qu'elles se comportent exactement de la même façon qu'en **C**. Nous pouvons donc les additionner, les multiplier, leur affecter une valeur, etc. Le seul point auquel il faut faire attention c'est leur initialisation qui ne se fait pas à l'aide de parenthèses mais avec le signe **=**.

Prenons un petit exemple en déclarant deux variables de type **int**, puis additionnons-les :

Code : C

```
void main()
{
    // Déclaration de deux variables integer

    int mesChats = 5;
    int chatDuVoisin = 1;

    // Addition

    int resultat = mesChats + chatDuVoisin;
}
```

Ce code pourrait parfaitement servir à un programme normal. 😊

Vous voyez ici que la déclaration et l'utilisation de variable se passent de la même manière qu'en **C**, chaque instruction se termine même par un point-virgule. Tout ce que vous connaissez sur les variables s'applique au **GLSL**, à savoir :

- Les opérations arithmétiques
- Les affectations
- L'incrémentation et la décrémentation (++ et --)
- ...

Vous avez peut-être même remarqué l'utilisation des commentaires dans le code source. Leur syntaxe reste aussi la même :

Code : C

```
void main()
{
    // Petit commentaire

    /* Gros commentaire,
    Parce que j'ai beaucoup de chose à dire */
}
```

Les tableaux

Les tableaux, qui permettent de rassembler des données du même type en mémoire, sont aussi utilisables avec les shaders. Ils se comportent aussi de la même manière, ils possèdent donc une taille et ses cases sont accessibles à l'aide un indice.

Pour initialiser un tableau, il suffit de lui donner un nom ainsi qu'une taille entre crochets. Pour accéder ou affecter une valeur aux cases, on utilise également les crochets avec un indice qui peut commencer à **0** ou qui peut se terminer par la **taille - 1**.

Voici un petit exemple de l'utilisation d'un tableau de 3 flottants :

Code : C

```
void main()
{
    // Déclaration d'un tableau

    float tableau[3] = {1.0, 2.0, 3.0};

    // Utilisation

    float premiereCase = tableau[0];
}
```



Les tableaux multidimensionnels du type **monTableau[0][0]** ne peuvent pas être utilisés.

Les vecteurs

Les variables que nous venons de voir ont l'avantage d'être faciles à comprendre car nous avons l'habitude de les utiliser dans nos codes sources classiques. Cependant, il existe encore d'autres **types** propres au **GLSL** qui sont un peu spéciaux. D'ailleurs, je dois vous avouer que vous les connaissez déjà en fait. 😊 Ces derniers peuvent être comparés aux structures du **C** mais ce

sont bel et bien des *types de variable*.

Ils ont été créés pour faciliter la vie aux développeurs car ils permettent de faire pas mal de calculs mathématiques relatifs à la 3D sans se prendre la tête. Si je vous dis que vous les connaissez déjà c'est parce que la librairie **GLM** que l'on utilise depuis le début du tuto est directement inspiré du **GLSL** et donc de ces types de variable. C'est à dire que l'on retrouve des types communs entre les deux, même s'il s'agit d'un objet d'un côté et d'une variable de l'autre.

Le premier de ces types concerne les vecteurs, il se décline en plusieurs versions :

- **vec2** : vecteur à 2 coordonnées
- **vec3** : vecteur à 3 coordonnées
- **vec4** : vecteur à 4 coordonnées



Ces types-là sont ceux que nous utiliserons la plus avec les shaders, retenez donc bien ce que nous allons voir sur eux.

Le type **vec3** devrait vous dire quelque chose il me semble. 🍪 Et oui, l'objet **vec3** qu'on utilise dans l'application C++ est inspiré directement du type de variable **vec3**.

Ces variables en **GLSL** se rapproche plus à des structures **C** car ils contiennent des sortes de "*sous-variable*". Par exemple, le type **vec2** possède deux coordonnées (x, y) qui sont accessibles via l'utilisation du point ! tout comme les structures :

Code : C

```
void main()
{
    // Vecteur à 2 coordonnées

    vec2 monVecteur;

    // Accès aux coordonnées

    monVecteur.x = 1.0;
    monVecteur.y = 2.0;
}
```

Bien entendu, on peut affecter des valeurs à ces coordonnées. Elles n'auraient que bien peu d'intérêt sinon. 🍪 Ces valeurs doivent être des **float** et non des **int**.

Les types **vec3** et **vec4** ont eux-aussi les coordonnées (x, y) sauf qu'ils en possèdent respectivement 1 (**x, y, z**) et 2 (**x, y, z, w**) en plus.



Petit détail : le **w** se trouve en dernière position et ne vient donc qu'après le **z**, faites attention à ça. 😊

L'accès à ces nouvelles coordonnées se fait exactement de la même façon ben sûr :

Code : C

```
void main()
{
    // Vecteurs à 3 et 4 coordonnées

    vec3 monVecteur;
    vec4 monGrandVecteur;

    // Accès aux coordonnées du premier vecteur
```

```
monVecteur.x = 1.0;
monVecteur.y = 2.0;
monVecteur.z = 3.0;

// Accès aux coordonnées du second vecteur

monGrandVecteur.x = 1.0;
monGrandVecteur.y = 2.0;
monGrandVecteur.z = 3.0;
monGrandVecteur.w = 4.0;
}
```

Vu que ces vecteurs sont des variables comme les autres, nous pouvons donc utiliser les opérateurs arithmétiques sur eux comme le +, -, *, etc :

Code : C

```
void main()
{
    // Vecteurs

    vec3 position, orientation;

    // Affectation de valeur

    position.x = 10.0;
    orientation.x = 5.0;
    .....

    // Opérations

    vec3 resultat = position + orientation;
    resultat *= 0.5;
}
```

Comme d'habitude avec les vecteurs, vous devez faire attention au nombre de coordonnées avant d'effectuer des opérations entre eux. Une variable *vec2* ne peut pas être multipliée par un *vec4* par exemple.

Petit bonus : il est possible de faire des tableaux de vecteur vu que ce sont des variables (j'espère vous l'avoir assez rabâché 😊):

Code : C

```
void main()
{
    // Vecteurs

    vec4 vecteur1, vecteur2;

    // Tableau

    vec4 montableau[2] = {vecteur1, vecteur2};
}
```

Les matrices

Aaaaah les matrices, je suis sûr que vous êtes tous contents de les retrouver. 🍪 Surtout que vous pouvez l'être car le **GLSL** nous fournit très gentiment des types de variable qui devraient vous dire quelque chose.

Les matrices sont des outils mathématiques incontournables dans la 3D. Il était donc évident de les intégrer dans le langage de programmation des shaders, au même titre que les vecteurs. Elles sont tellement importantes qu'elles possèdent même plusieurs **types** de variable. Voici ceux que l'on utilisera le plus :

- **mat2** : matrice carrée d'ordre 2
- **mat3** : matrice carrée d'ordre 3
- **mat4** : matrice carrée d'ordre 4

Hum **mat4**, moi j'dis ça me rappellerait quelque chose. 🤔

Les matrices en **GLSL** se déclarent simplement : il suffit d'utiliser un des types précédents suivi du nom qu'on veut donner.

Code : C

```
void main()
{
    // Matrice carrée d'ordre 3

    mat3 matrice;
}
```

Leur principal avantage c'est qu'elles sont **AUSSI** considérées comme des variables tout comme les vecteurs. Il est donc possible de faire des opérations arithmétiques dessus :

Code : C

```
void main()
{
    // Matrices

    mat3 matrice1, matrice2;

    // Multiplication

    mat3 résultat = matrice1 * matrice2;
}
```

Simple n'est-ce pas ?

Je ne vous parle pas de la façon d'accéder aux valeurs des matrices car elles fonctionnent d'une manière différente de celle que l'on connaît. En fait, elles se lisent *colonne par colonne* plutôt que *ligne par ligne*. Nous verrons cela dans un prochain chapitre pour éviter de s'embrouiller. 😊 En revanche, nous allons voir dans cette partie la façon de les initialiser.

Enfin, vous connaissez maintenant tous les types de variable qu'il existe en **GLSL**. Ou plutôt presque tous car il en existe encore plein d'autres comme les vecteurs d'entier (**ivec**), les matrices non-carrées (**mat3x4**), etc. Je ne vous ai présenté ici que ceux que nous utiliserons le plus, le reste est plus anecdotique.

Les constructeurs

L'initialisation simple

Les constructeurs sont des outils très puissants et très flexibles qui vont nous permettre de faire gagner pas mal à temps à nos shaders. C'est une des seuls points que l'on peut rapprocher au C++, même s'il ne s'agit pas de réel constructeur de classe car la notion d'objet n'existe pas en GLSL.

Leur utilité vient du fait qu'ils permettent d'initialiser les variables (vecteurs et matrices compris) en une seule ligne de code. Il devient alors inutile d'affecter toutes les valeurs à la main. Le nom du constructeur à utiliser correspond au type de variable à initialiser.

Par exemple, si on veut initialiser une variable `vec3` avec les coordonnées (1.0, 2.0, 3.0), il suffit de faire :

Code : C

```
void main()
{
    // Initialisation d'un vecteur à 3 coordonnées

    vec3 vecteur = vec3(1.0, 2.0, 3.0);
}
```

C'est exactement la même chose qu'avec **GLM**. Ce qui est normal d'ailleurs puisque les développeurs de cette librairie ont fait en sorte que le comportement C++ des objets se rapproche au maximum de celui des variables en **GLSL**. Même l'utilisation faite des constructeurs est recopiée.

Ce constructeur nous évite au final d'avoir à faire :

Code : C

```
void main()
{
    // Initialisation d'un vecteur à 3 coordonnées

    vec3 vecteur;

    vecteur.x = 1.0;
    vecteur.y = 2.0;
    vecteur.z = 3.0;
}
```

Les constructeurs font très plaisir quand vous avez plusieurs vecteurs à initialiser dans votre shader, croyez-moi. 🍀

Petit bonus : Si vous voulez initialiser un vecteur nul, vous n'avez pas besoin de renseigner toutes les coordonnées. Seule la valeur **0.0** sera nécessaire, le constructeur saura qu'il doit tout initialiser avec :

Code : C

```
void main()
{
    // Initialisation d'un vecteur nul

    vec4 vecteur = vec4(0.0);
}
```

Les vecteurs ne sont pas les seuls concernés par les constructeurs, les matrices aussi ont le droit d'en profiter. Voici, par exemple, comment initialiser une matrice carrée d'ordre 3 :

Code : C

```
void main()
{
    // Initialisation d'une matrice

    mat3 matrice = mat3(0.0, 3.0, 6.0,
                        1.0, 4.0, 7.0,
                        2.0, 5.0, 8.0);
}
```

J'ai volontairement fait un retour à la ligne toutes les 3 valeurs pour que le code soit plus lisible, il est évidemment possible de tout mettre en une seule.

On remarque dans cet exemple la présence de **9** valeurs allant de **0.0** à **8.0** qui permettent d'initialiser la matrice. Si on avait utilisé un type **mat4**, alors il en aurait fallu **16**. On remarque aussi que les valeurs sont arrangees *en colonnes*. Si vous suivez les valeurs des yeux dans l'ordre croissant, vous remarquerez cette particularité.



C'est compliqué d'utiliser les matrices comme ça, je vais devoir m'adapter à chaque fois que je vais m'en servir ?

Non pas vraiment car elles seront la plupart du temps déjà données. Nous n'aurons donc pas à jouer avec cette lecture en colonne. 😊

Petit bonus : pour initialiser une matrice nulle, c'est-à-dire qu'avec des valeurs égales à **0.0**, il existe heureusement une astuce qui nous évite tous les petits soucis précédents. En fait, il suffit juste de mettre **une seule** valeur **0.0** dans le constructeur comme pour les vecteurs :

Code : C

```
void main()
{
    // Matrice nulle

    mat4 matrice = mat4(0.0);
}
```

Plus simple que l'exemple précédent n'est-ce pas ? 😊

Dernier bonus : si vous voulez initialiser une matrice d'identité, vous pouvez faire exactement la même chose qu'avec **GLM** en appelant le constructeur avec la valeur **1.0**. Ce dernier saura automatiquement qu'il doit affecter cette valeur à la diagonale :

Code : C

```
void main()
{
    // Matrice d'identité

    mat4 matrice = mat4(1.0);
}
```

Cette astuce ne fonctionne qu'avec les matrices. Pour les vecteurs, votre compilateur vous râlera dessus car il demandera la valeur des autres coordonnées.

L'initialisation à partir d'autres variables

L'utilité des constructeurs ne s'arrête pas avec des valeurs statiques, il est tout à fait possible d'initialiser une variable à partir d'autres variables pré-existantes. C'est même une chose que l'on fera très souvent dans nos codes sources. 

Le gros avantage également c'est que vous pouvez initialiser des variables à partir de n'importe quelle autre variable quelque soit son **type**. Par exemple, vous pouvez très bien utiliser un **vec2** pour construire un **vec4** à partir de ses coordonnées :

Code : C

```
void main()
{
    // Vecteur

    vec2 petitVecteur = vec2(1.0, 2.0);

    // Initialisation à partir d'un autre vecteur
    vec4 grandVecteur = vec4(petitVecteur);
}
```

Vous avez compris le principe ? Le constructeur **vec4()** va prendre automatiquement les coordonnées (x, y) du premier vecteur pour les copier dans le second. Nous aurions pu utiliser ce code pour faire la même chose :

Code : C

```
void main()
{
    // Vecteur

    vec2 petitVecteur = vec2(1.0, 2.0);

    // Initialisation à partir d'un autre vecteur
    vec4 grandVecteur;

    grandVecteur.x = petitVecteur.x;
    grandVecteur.y = petitVecteur.y;
}
```

Alors bon, je dois vous avouer que ce code n'est pas totalement correcte. En fait, si nous le compilions nous nous retrouverions avec un beau message d'erreur car le constructeur ne sait pas comment initialiser les dernières coordonnées (z, w). Il faut donc leur affecter une valeur à elles-aussi.

Pour cela, il suffit simplement de rajouter 2 autres valeurs en paramètres :

Code : C

```
void main()
{
    // Vecteur

    vec2 petitVecteur = vec2(1.0, 2.0);

    // Initialisation à partir de la variable
    vec4 grandVecteur = vec4(petitVecteur, 3.0, 4.0);
```

```
}
```



Faites attention cependant à toujours vérifier le nombre de vos paramètres. Il vaut mieux perdre un peu de temps à vérifier plutôt que d'attendre l'erreur de compilation pour vous en rendre compte.

Cette fois, notre compilateur est content car toutes les coordonnées ont le droit à une valeur.

D'ailleurs, il est même possible d'inverser les trois paramètres en plaçant les valeurs **3.0** et **4.0** au début :

Code : C

```
void main()
{
    // Vecteur position

    vec2 vecteur = vec2(1.0, 2.0);

    // Vecteur

    vec4 vecteur2 = vec4(3.0, 4.0, vecteur);
}
```

Si on fait ça en revanche, les données ne seront pas initialisées de la même façon. C'est-à-dire que la valeur **3.0** sera affectée à la coordonnée **x** et la valeur **4.0** à **y**. Pour **z** et **w**, le constructeur va prendre ce qu'il lui reste, à savoir respectivement les coordonnées (**x**, **y**) de la variable **vecteur**.

Ce code revient à faire la chose suivante :

Code : C

```
void main()
{
    // Vecteur

    vec2 petitVecteur = vec2(1.0, 2.0);

    // Autre vecteur

    vec4 grandVecteur;

    grandVecteur.x = 3.0;
    grandVecteur.y = 4.0;
    grandVecteur.z = petitVecteur.x;
    grandVecteur.w = petitVecteur.y;
}
```

Allez, on prend un dernier exemple pour être sûr que vous ayez compris. Si je vous montre le bout de code suivant, seriez-vous capable de trouver l'équivalent en utilisant un constructeur ?

Code : C

```
void main()
{
    // Vecteur
```

```
vec2 petitVecteur = vec2(5.0, 8.0);  
  
// Autre vecteur  
  
vec4 grandVecteur;  
  
grandVecteur.x = 7.0;  
grandVecteur.y = petitVecteur.x;  
grandVecteur.z = petitVecteur.y;  
grandVecteur.w = 2.0;  
}
```

J'ai volontairement changé les valeurs pour vous déstabiliser. 😜

Vous avez trouvé ?

Secret (cliquez pour afficher)

Code : C

```
void main()  
{  
    // Vecteur  
  
    vec2 petitVecteur = vec2(5.0, 8.0);  
  
    // Autre vecteur  
  
    vec4 grandVecteur = vec4(7.0, petitVecteur, 2.0);  
}
```

Bien évidemment, je passe à coté de toute la subtilité offerte par le **GLSL** au niveau des variables. Cependant, nous en avons vu assez pour pouvoir travailler nos shaders efficacement. 😊

Les structures de contrôle

Les structures de contrôles

Nous avons vu le plus gros du chapitre avec la partie sur les variables, il y avait pas mal de petites notions à voir. Nous allons maintenant nous attaquer à tout ce qui concerne les structures de contrôle comme les conditions, les boucles, etc. Il n'y aura pas de surprise cette fois, vous connaissez déjà tout. 😊

Les conditions

Les conditions permettent de déclencher une portion de code en fonction de la valeur d'une variable. Il y a deux manières de les utiliser :

- Soit avec les instructions : **if**, **else if**, **else**

Code : C

```
void main()  
{  
    // Variable
```

```
int variable = 1;

// Conditions

if(variable == 0)
    ...
else if(variable == 1)
    ...
else
    ...
}
```

- Soit avec les instructions : **switch, case**

Code : C

```
void main()
{
    // Variable

    int variable = 1;

    // Conditions

    switch(variable)
    {
        // Cas 1

        case 0:
            ...
        break;

        // Cas 2

        case 1:
            ...
        break;

        // Sinon

        default:
            ...
        break;
    }
}
```



Les mot-clefs **break** et **continue** sont utilisables en **GLSL**.

Les boucles

Les boucles permettent de répéter une portion de code en fonction d'une condition donnée. Ces conditions sont les mêmes que celles vues précédemment. Il y existe trois façons d'utiliser une boucle :

- Soit avec l'instruction **while**

Code : C

```
void main()
{
    // Variable

    int variable = 0;

    // Boucle

    while(variable < 10)
        variable++;
}
```



Cette boucle s'exécutera tant que la variable n'aura pas dépassé la valeur **10**.

- Soit avec les instructions **do, while**

Code : C

```
void main()
{
    // Variable

    int variable = 0;

    // Boucle

    do
    {
        variable++;

    }while(variable < 10);
}
```

- Soit avec l'instruction **for**

Code : C

```
void main()
{
    // Variable

    int variable = 0;

    // Boucle

    for(int i = 0; i < 10; i++)
        variable++;
}
```

Les fonctions

La déclaration

Les fonctions sont des fonctionnalités importantes avec les langages de programmation. Nous serions très malheureux sans elles puisqu'il faudrait tout coder au même endroit. Imaginez si nous devions réunir toutes les classes d'un projet C++ dans un seul fichier ! 😊

Heureusement pour nous, ça ne sera pas le cas avec le **GLSL** car il permet la création et l'utilisation de fonctions. L'avantage en plus c'est qu'elles se comportent exactement comme en C il n'y a aucun différence. Nous retrouvons donc leurs caractéristiques principales :

- Des paramètres
- Un nom
- Une variable renvoyée si besoin

Le prototype, qui représente la signature de la fonction, n'est pas obligatoire mais je vous conseille de l'utiliser pour simplifier la lecture de vos codes sources. Il se place juste avant la fonction **main()** pour que le compilateur puisse connaître leur existence avant d'attaquer le programme.

Voici un exemple de prototype de fonction :

Code : C

```
// Prototype  
vec4 convertirVecteur(vec2 vecteur);  
  
// Fonction main  
void main()  
{  
}
```

L'implémentation se fait après la fonction **main()** de façon à ce que ce soit plus lisible :

Code : C

```
// Implémentation de la fonction  
vec4 convertirVecteur(vec2 vecteur)  
{  
    vec4 resultat = vec4(vecteur, 0.0, 0.0);  
  
    return resultat;  
}
```

Notez la présence du mot-clé **return** qui permet de renvoyer une variable.

Ce qui donnerait un shader final :

Code : C

```
// Prototype  
  
vec4 convertirVecteur(vec2 vecteur);  
  
// Fonction main  
  
void main()  
{  
  
}  
  
// Implémentation  
  
vec4 convertirVecteur(vec2 vecteur)  
{  
    vec4 resultat = vec4(vecteur, 0.0, 0.0);  
  
    return resultat;  
}
```

L'utilisation

Les fonctions s'utilisent également de la même manière qu'en **C**. Il suffit de les appeler en leur donnant les paramètres qu'elles demandent.

Si on reprend l'exemple précédent :

Code : C

```
void main()  
{  
    // Vecteur  
  
    vec2 monPetitVecteur = vec2(1.0, 2.0);  
  
    // Appel à la fonction de conversion  
  
    vec4 monGrandVecteur = convertirVecteur(monPetitVecteur);  
}
```

Les surcharges

La surcharge de fonction est la deuxième notion du **GLSL** que l'on peut rapprocher au **C++**. Il est possible d'utiliser plusieurs fois le même nom pour des fonctions différentes. Les conditions qui permettent de faire cela sont les mêmes qu'avec le **C++**, à savoir que les paramètres doivent être différents en nombre ou en type.

On reprend une fois de plus l'exemple précédent en surchargeant la fonction **convertirVecteur()** afin qu'elle puisse prendre en paramètre des variables de type **vec3**:

Code : C

```
// Prototype original
```

```
vec4 convertirVecteur(vec2 vecteur);  
  
// Surcharge  
  
vec4 convertirVecteur(vec3 vecteur);
```

Il faut bien évidemment faire une seconde implémentation car les deux prototypes représentent des fonctions différentes :

Code : C

```
// Implémentation de la surcharge  
  
vec4 convertirVecteur(vec3 vecteur)  
{  
    vec4 resultat = vec4(vecteur, 0.0);  
  
    return resultat;  
}
```

Pour utiliser cette surcharge, il suffit aussi de l'appeler par son nom. Le shader s'occupera tout seul de savoir s'il doit prendre celle-ci ou la fonction originale :

Code : C

```
void main()  
{  
    // Vecteur  
  
    vec3 monPetitVecteur = vec3(1.0, 2.0, 3.0);  
  
    // Appel à la fonction de conversion  
    vec4 monGrandVecteur = convertirVecteur(monPetitVecteur);  
}
```

Les pointeurs

Malheureusement, les pointeurs n'existent pas en **GLSL**, et je ne vous parle même pas des références qui existent encore moins. Il n'est donc pas possible de modifier plusieurs variables dans une seule fonction, il faudra en utiliser plusieurs. Mais je vous rassure, nous n'aurions quasiment jamais eu besoin de pointeur de toute façon. 😊

Divers

Le préprocesseur

Le préprocesseur est un processus qui s'exécute juste avant le compilateur et qui permet de filtrer ou d'ajouter des instructions au code source original.

Sa particularité vient du fait qu'il ne lit pas les lignes de code '*normales*' mais uniquement celles que l'on appelle des **directives de préprocesseur**. On les reconnaît grâce à leur symbole #.

Avec le **GLSL**, il existe aussi un préprocesseur qui nous permet d'utiliser des directives comme **#define**, **#ifdef**, ... La plus importante pour nous va être celle qui permet de définir la version du **GLSL** avec laquelle nous coderons. Elle s'appelle **#version** et s'utilise de cette façon :

Code : C

```
// Version du GLSL
#version 150 core

// Fonction main()

void main()
{
```

}

Elle permet de dire à OpenGL que notre code source utilisera la version **1.50** du **GLSL** ainsi que le profil **core**. Notez qu'il est possible de remplacer le mot-clé **core** par **compatibility**. 😊

Nous utiliserons cette directive au début de chacun de nos codes sources à partir de maintenant.

Les structures

Les structures sont un peu comme les ancêtres des objets. Ils possèdent des champs, que l'on peut rapprocher aux attributs, mais ne possèdent pas de méthodes propres. La bonne nouvelle c'est qu'on peut quand même les utiliser pour programmer nos shaders. 😊

Leur déclaration se fait exactement comme en **C**, on utilise le mot-clé **struct** accompagné du nom souhaité. On inclut ensuite les champs à l'intérieur.

Prenons un petit exemple en déclarant une structure **Camera**. Structure qui ressemble étonnement à une classe que nous avons déjà codée. 😊

Code : C

```
// Structure Camera

struct Camera
{
    vec3 position;
    vec3 orientation;

    float rapidite;
};
```

L'avantage des structures en **GLSL** c'est qu'il n'y a pas besoin d'utiliser le mot clef **typedef** pour s'affranchir du mot-clé **struct**. Nous pouvons donc utiliser notre structure directement comme s'il s'agissait d'une variable normale :

Code : C

```
void main()
{
    // Déclaration d'une structure Camera
    Camera maCamera;
```

```
// Utilisation  
maCamera.position = vec3(2.0, 5.0, 2.0);  
maCamera.rapidite = 0.5;  
}
```

Fonctions prédéfinies

Nous avons fait le tour de la syntaxe de notre nouveau langage, il n'y a pas grand chose à dire de plus dessus. Je vous réserve le reste pour le chapitre suivant. 😊 Mais avant de passer à ça, j'aimerais vous montrer quelques fonctions prédéfinies qui nous seront utiles par la suite.

En effet, le **GLSL** a l'avantage d'intégrer nativement une bibliothèque mathématique assez complète qui permet d'aider le développeur dans ses calculs 3D. On retrouve ainsi plusieurs fonctions relatives à la trigonométrie, aux vecteurs, aux matrices, etc.

Les fonctions trigonométriques

Sinus, Cosinus et Tangente

Si je vous parle de trigonométrie vous devez forcément penser aux fonctions **sin()**, **cos()** et **tan()**. 🎉 Elles permettent de calculer respectivement le sinus, le cosinus et la tangente d'un angle. Leur prototype est le même qu'en C++ :

Code : C

```
float sin(float angle);  
float cos(float angle);  
float tan(float angle);
```

Voici un petit exemple qui permet de calculer le sinus d'un angle :

Code : C

```
#version 150 core  
  
void main()  
{  
    // Calcul du sinus d'un angle de 90°  
  
    float sinus = sin(90.0);           /* La valeur du sinus sera  
    de 1.0 */  
}
```

Les autres fonctions s'utilisent de la même façon.

Convertir un angle en radian et inversement

Il existe des fonctions permettant de convertir un angle exprimé en degrés vers les radians, et inversement bien sûr. Elles sont très utiles dans certains cas, d'ailleurs on aurait bien aimé les avoir en C++ quand nous avons utilisé la trigonométrie. 🎉

Ces fonctions s'appellent respectivement **radians()** et **degrees()** :

Code : C

```
float radians(float degrees);
float degrees(float radians);
```

Un petit exemple de conversion d'un angle exprimé en degrés :

Code : C

```
#version 150 core

void main()
{
    // Conversion de l'angle 90° en radians

    float radians = radians(90.0);                  /* La valeur de
l'angle sera de 1/2 Pi soit environ 1.57 */
}
```

Les fonctions relatives aux vecteurs

Normalisation

Ah la normalisation ... Ça devrait vous rappeler quelques souvenirs. Surtout que nous avons eu l'occasion d'en faire plein avec **GLM**. 

Vous devriez donc savoir que normaliser un vecteur revient à réduire sa *norme* (sa longueur) à **1.0**. Cela permet de faire pas mal d'opérations mathématiques et de profiter de la trigonométrie.

La fonction qui permet de normaliser un vecteur s'appelle **normalize()** exactement comme celle de **GLM** :

Code : C

```
vec normalize(vec vector);
```



Les fonctions prédéfinies sont en général toutes surchargées, ce qui évite d'avoir à utiliser des noms différents pour la même opération. Le type **vec** n'existe évidemment pas, je l'ai mis ici pour vous montrer que la fonction **normalize()** était surchargée.

Je ne vous fait pas d'exemple, je pense que vous avez compris le principe. 

Calculer la norme

Vu que l'on parle de la *norme* d'un vecteur, nous pouvons parler de la fonction qui permet de la calculer. Elle s'appelle **length()** :

Code : C

```
float length(vec vector);
```

Calculer le produit vectoriel

Nous avons déjà utilisé le produit vectoriel pour faire des calculs dans la classe **Camera** qui nous permettait de trouver le vecteur orthogonal à l'orientation. La fonction permettant de faire la même chose en **GLSL** s'appelle aussi **cross()** :

Code : C

```
vec cross(vec vector1, vec vertor2);
```

Cette fonction prend deux paramètres représentant les deux vecteurs à multiplier. Faites attention à utiliser le même **type** pour cette opération. Vous aurez une erreur de compilation si vous essayez d'en utiliser deux différents.

Autres fonctions

Il existe encore une multitude de fonctions prédéfinies dont je ne vous ai pas parlées mais il y en a tellement que mes pauvres doigts souffriraient trop si je devais toutes vous les énumérer. 😊

Sachez qu'il existe aussi des fonctions permettant d'effectuer des calculs sur les matrices comme le calcul du déterminant, d'inversion, etc. Il en existe aussi qui permettent de faire des opérations 'classiques' comme les arrondis, les valeurs absolues, les modulus, etc. Le **GLSL** contient une véritable bibliothèque couteau-suisse pour les calculs mathématiques dont est inspirée la librairie **GLM**.

Dernier point à préciser : rappelez-vous que les shaders s'exécutent autant de fois qu'il y a de vertices/pixels, donc potentiellement des millions de fois par seconde. Il est donc préférable d'utiliser les fonctions prédéfinies à la place des nôtres quand on le peut.

Nous avons fait le tour du langage de programmation qu'est l'**OpenGL Shading Language**.

Nous connaissons les types de variables que l'on peut utiliser, les structures de contrôle comme les conditions, les boucles, etc. Nous avons vu en somme sa syntaxe globale; syntaxe qui se rapproche beaucoup de **C/C++**, les développeurs ont tout fait pour faciliter son apprentissage.

Nous nous servirons de ce langage pour coder tous nos futurs codes sources. D'ailleurs, nous allons commencer tout de suite dans le chapitre suivant en programmant notre premier **shader** ! 😊

Les shaders démystifiés (Partie 1/2)

Nous voici arrivés (enfin !) au chapitre qui va nous permettre de créer nos propres *shaders*. Nous avons appris la syntaxe du langage de programmation **GLSL** et nous savons même comment compiler des codes sources l'utilisant.

Aujourd'hui, nous allons lever le voile sur les dernières zones d'ombres qui subsistent dans nos programmes, notamment en étudiant tous les petits fichiers sources que l'on utilisés depuis le début. Nous en profiterons pour jouer un peu avec afin de créer nos premiers effets personnalisés. Ils seront un peu basiques certes, mais il faut bien commencer par quelque chose. 😊

J'ai préféré couper ce chapitre en deux car il était trop lourd à suivre en une seule fois. A la place, vous aurez le droit à deux chapitres plus petits dans lesquels vous vous sentirez moins étouffés.

Préparation

Préparation

Dans ce chapitre, nous allons nous concentrer sur les premiers shaders que nous avons utilisés au début du tuto. Exit donc les matrices et la caméra, nous verrons ça dans le prochain chapitre. Il vaut mieux ne pas s'en occuper pour le moment car si je vous montre tout d'un coup, vous risquez de vous emmêler les pinceaux. 😊 Il vaut mieux y aller en douceur. Mais ne vous inquiétez pas, nous en verrons assez pour faire nos premiers essais dans la programmation **GLSL**.

Avant de commencer, il va nous falloir nettoyer un peu la boucle principale en enlevant la gestion des matrices et de la caméra. Cependant, nous allons devoir conserver l'utilisation du **VBO** et du **VAO** car ceux-ci sont obligatoirement demandés par la version **3.3** d'**OpenGL**.

Nous testerons nos premières sources sur un petit carré à 2 dimensions pour le moment vu que les matrices ne sont pas encore utilisables. Nous finirons sur des cubes et des pyramides dans le chapitre suivant.

Bref pour commencer, je vous demande de vider votre méthode **bouclePrincipale()** de tout son contenu précédent. Ajoutez-y ensuite les vertices de notre petit carré :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Vertices

    float vertices[] = {-0.5, -0.5,      0.5, -0.5,      0.5, 0.5,      //
Triangle 1
                           -0.5, -0.5,      -0.5, 0.5,      0.5, 0.5}; // //
Triangle 2
}
```

Ce carré ne sera pas minuscule ne vous inquiétez pas. Vu que nous n'utilisons pas la caméra il faut revenir à l'ancien repère où les coordonnées sont comprises dans l'intervalle [-1; 1]. Si vous faites un carré dépassant cette taille alors il prendra toute votre fenêtre.

Enfin, maintenant qu'on déclaré nos vertices, nous pouvons passer à l'implémentation du **VBO**. Il n'y a qu'un seul tableau à envoyer pour le moment donc le calcul de sa taille sera simple. L'espace mémoire à allouer fait donc **3 vertices x 2 coordonnées x 2 triangles = 12 cases** :

Code : C++

```
// VBO et taille des données

GLuint vbo;
int tailleVerticesBytes = 12 * sizeof(float);
```

Une fois la taille définie, on peut allouer **VBO** :

Code : C++

```
// Génération du VBO  
  
glGenBuffers(1, &vbo);  
  
// Verrouillage  
  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
  
// Allocation  
  
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes, 0,  
GL_STATIC_DRAW);  
  
// Déverrouillage  
  
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Puis, on peut le remplir avec les vertices :

Code : C++

```
// Allocation  
  
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes, 0,  
GL_STATIC_DRAW);  
  
// Remplissage  
  
glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes, vertices);
```

Nous avons vu ce code dans le chapitre sur les **VBO**, il ne devrait pas vous poser de problème. 😊

Ce qui nous donne pour le moment :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Vertices  
  
    float vertices[] = {-0.5, -0.5,      0.5, -0.5,      0.5, 0.5,      //  
Triangle 1  
                  -0.5, -0.5,      -0.5, 0.5,      0.5, 0.5}; //  
Triangle 2  
  
    /* ***** Gestion du VBO ***** */  
  
    GLuint vbo;  
    int tailleVerticesBytes = 12 * sizeof(float);  
  
    // Génération du VBO
```

```
glGenBuffers(1, &vbo);

// Verrouillage
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Remplissage
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes, 0,
GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes,
vertices);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

On passe maintenant au **VAO**. On commence évidemment par lui générer un **ID** :

Code : C++

```
// VAO

GLuint vao;

// Génération du VAO
glGenVertexArrays(1, &vao);
```

Nous devons mettre à l'intérieur tous nos appels aux tableaux **Vertex Attrib** (activation comprise). Pour le moment, nous n'avons que celui des vertices à appeler donc c'est assez simple.

D'ailleurs en parlant de ça, faites attention aux paramètres de ce tableau car nos vertices possèdent **2** coordonnées et non **3**. Affectez donc la valeur **2** au paramètre **size** :

Code : C++

```
// Verrouillage du VAO
glBindVertexArray(vao);

// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
glEnableVertexAttribArray(0);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
// Déverrouillage du VAO  
glBindVertexArray(0);
```

N'oubliez pas de verrouiller votre **VBO** quand vous utilisez le tableau **Vertex Attrib**.



Si on résume tout ça :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Vertices  
  
    float vertices[] = {-0.5, -0.5,      0.5, -0.5,      0.5, 0.5,      //  
Triangle 1  
                  -0.5, -0.5,      -0.5, 0.5,      0.5, 0.5}; //  
Triangle 2  
  
/* ***** Gestion du VBO ***** */  
  
GLuint vbo;  
int tailleVerticesBytes = 12 * sizeof(float);  
  
// Génération du VBO  
glGenBuffers(1, &vbo);  
  
// Verrouillage  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
  
// Remplissage  
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes, 0,  
GL_STATIC_DRAW);  
glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes,  
vertices);  
  
// Déverrouillage du VBO  
glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
/* ***** Gestion du VAO ***** */  
  
GLuint vao;  
glGenVertexArrays(1, &vao);  
  
// Verrouillage du VAO
```

~1 Bind Vertex Array (VAO) ~

```
glBindVertexArray(vao);

    // Verrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

        // Vertex Attrib 0 (Vertices)
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(0));
        glEnableVertexAttribArray(0);

    // Déverrouillage du VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Déverrouillage du VAO
    glBindVertexArray(0);
}
```

Pfiouuu, tout ce code pour envoyer deux simples triangles ! 😊

Et encore ce n'est pas fini, il nous manque toujours la boucle **while**. On commence à bien la connaître celle-là :

Code : C++

```
// Variables relatives au framerate

unsigned int frameRate (1000 / 50);
Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

// Boucle principale

while (!m_input.terminer())
{
    // Gestion des évènements, etc.

    ...
}
```

On inclut évidemment le code de gestion des évènements :

Code : C++

```
// Boucle principale

while (!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();
```

```
if (m_input.getTouche(SDL_SCANCODE_ESCAPE))
    break;

// Rendu
....  
  
// Calcul du temps écoulé
finBoucle = SDL_GetTicks();
tempsEcoule = finBoucle - debutBoucle;  
  
// Si nécessaire, on met en pause le programme
if (tempsEcoule < frameRate)
    SDL_Delay(frameRate - tempsEcoule);
}
```

Et enfin, on rajoute le code permettant de nettoyer et d'actualiser ce qui est affiché à l'écran :

Code : C++

```
// Boucle principale

while (!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if (m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* *** Rendu ***
       */

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow(m_fenetre);

    // Calcul du temps écoulé
    finBoucle = SDL_GetTicks();
    tempsEcoule = finBoucle - debutBoucle;
```

```
// Si nécessaire, on met en pause le programme  
  
if(tempsecoule < frameRate)  
    SDL_Delay(frameRate - tempsecoule);  
}
```

Il ne reste plus qu'à afficher notre rendu. Celui-ci va être très simple car il suffit de verrouiller le **VAO**, puis d'appeler la fonction **glDrawArrays()**:

Code : C++

```
// Verrouillage du VAO  
  
glBindVertexArray(vao);  
  
// Rendu  
  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Déverrouillage du VAO  
  
glBindVertexArray(0);
```

Ah dernier point, il faut penser à détruire le **VBO** et le **VAO** une fois la boucle terminée. 😊

Code : C++

```
// Destruction du VAO et du VBO  
  
glDeleteBuffers(1, &vbo);  
glDeleteVertexArrays(1, &vao);
```

Récap final :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Vertices  
  
    float vertices[] = {-0.5, -0.5, 0.5, -0.5, 0.5, 0.5, //  
    Triangle 1  
                    -0.5, -0.5, -0.5, 0.5, 0.5, 0.5}; //  
    Triangle 2  
  
    /* ***** Gestion du VBO ***** */  
  
    GLuint vbo;  
    int tailleVerticesBytes = 12 * sizeof(float);  
  
    // Génération du VBO  
  
    glGenBuffers(1, &vbo);
```

```
// Verrouillage
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Remplissage
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes, 0,
GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes,
vertices);

// Déverrouillage
glBindBuffer(GL_ARRAY_BUFFER, 0);

/* ***** Gestion du VAO **** */
GLuint vao;

// Génération du VAO
glGenVertexArrays(1, &vao);

// Verrouillage du VAO
glBindVertexArray(vao);

// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Déverrouillage du VAO
glBindVertexArray(0);

// Variables relatives au framerate
unsigned int frameRate (1000 / 50);
Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

// Boucle principale
while (!m_input.terminer())
{
    // On définit le temps de début de boucle
```

```
debutBoucle = SDL_GetTicks();  
  
// Gestion des évènements  
m_input.updateEvenements();  
  
if(m_input.getTouche(SDL_SCANCODE_ESCAPE))  
    break;  
  
// Nettoyage de l'écran  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  
// Verrouillage du VAO  
glBindVertexArray(vao);  
  
// Rendu  
glDrawArrays(GL_TRIANGLES, 0, 6);  
  
// Déverrouillage du VAO  
glBindVertexArray(0);  
  
// Actualisation de la fenêtre  
SDL_GL_SwapWindow(m_fenetre);  
  
// Calcul du temps écoulé  
finBoucle = SDL_GetTicks();  
tempsEcoule = finBoucle - debutBoucle;  
  
// Si nécessaire, on met en pause le programme  
if(tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);  
}  
  
// Destruction du VAO et du VBO  
glDeleteBuffers(1, &vbo);  
glDeleteVertexArrays(1, &vao);  
}
```

Nous avons maintenant un code propre sur lequel nous pouvons nous exercer. Nous augmenterons son contenu au fur et à mesure jusqu'à réintégrer les matrices, la caméra et les textures.

J'ai préféré faire une partie dédiée pour cela, en vous expliquant chaque étape, plutôt que de vous jeter le code d'un coup. Vous connaissez absolument tout ce qui se trouve à l'intérieur, il n'y a rien de nouveau (pour le moment). 😊

Premier shader Affichage simple

Chargement

Il est enfin temps de programmer notre premier shader, celui-ci sera très basique car il ne se contentera que d'afficher le carré à l'écran. Dans un premier temps nous lui donnerons la couleur blanche, nous gèrerons les autres couleurs petit à petit.

Pour commencer, je vais vous demander de vider complètement le dossier **Shaders** de votre projet afin d'enlever tous les précédents codes sources. Une fois fait, créez à la main deux nouveaux fichiers qui porteront le nom de **basique2D.vert** et **basique2D.frag**. Ensuite, déclarez un objet de type **Shader** dans la méthode **bouclePrincipale()** avec en paramètres le chemin vers ces fichiers :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Shader

    Shader shader("Shaders/basique2D.vert",
    "Shaders/basique2D.frag");
    shader.charger();

    // Vertices

    float vertices[] = {-0.5, -0.5,      0.5, -0.5,      0.5, 0.5,      //
Triangle 1
                           -0.5, -0.5,      -0.5, 0.5,      0.5, 0.5}; // //
Triangle 2

    ...

}
```

Activez-le ensuite au moment du rendu de façon à ce qu'il soit pris en compte par OpenGL :

Code : C++

```
// Activation du shader

glUseProgram(shader.getProgramID());

// Rendu

 glBindVertexArray(vao);

 glDrawArrays(GL_TRIANGLES, 0, 6);

 glBindVertexArray(0);

// Désactivation du shader

glUseProgram(0);
```

Le Vertex Shader

Nos outils sont en place et les fichiers sont créés ... Nous pouvons maintenant passer à la programmation du code source.

Nous commencerons par celui du **Vertex Shader** car c'est celui-ci qu'OpenGL appellera en premier dans le **pipeline 3D**. Son rôle

consiste à prendre un vertex (avec la couleur et les coordonnées de texture qui lui sont associées) pour travailler dessus.

Son code source commencera par l'utilisation de la balise `#version`, pour indiquer la version du **GLSL** utilisée, suivie par la fonction `main()` :

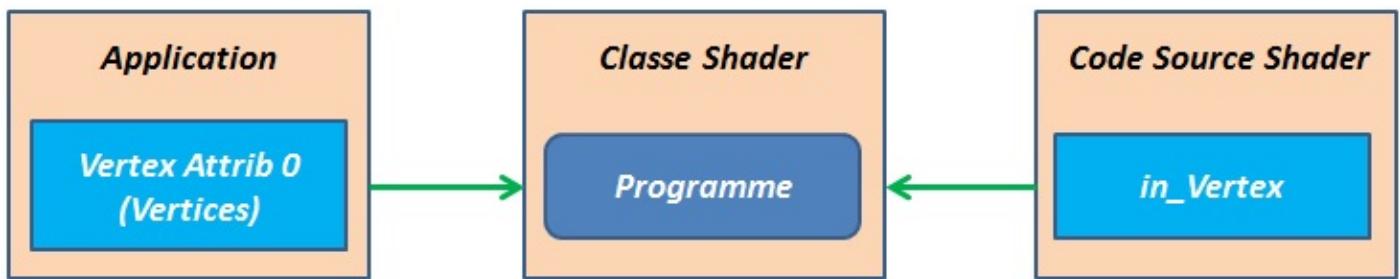
Code : C

```
// Version du GLSL
#version 150 core

// Fonction main
void main()
{}
```

Pour la suite, nous aurons besoin d'une notion que l'on a déjà vue précédemment : les *entrées shader*. Nous avons évoqué ce terme il y a à peine deux chapitres quand nous parlions du linkage du shader. Nous utilisons alors la fonction `glBindAttribLocation()` qui permettait de créer une passerelle entre un tableau **Vertex Attrib** et sa variable présente dans le **GLSL**.

Pour les vertices par exemple, le nom de cette variable était `in_Vertex` :



Nous avons utilisé la fonction `glBindAttribLocation()` pour verrouiller cette entrée, donc la variable `in_Vertex` est accessible dans notre code source. D'ailleurs, les autres variables comme `in_Color` sont elles aussi accessibles, nous verrons cela un peu plus loin. 😊

Pour utiliser `in_Vertex`, il faut déclarer une variable globale portant son nom. Je dis bien globale parce qu'elle doit être déclarée en dehors de la fonction `main()`. Elle doit être représentée par un type `vec` dont le nombre de coordonnées dépend de celui renseigné dans le tableau **Vertex Attrib**. Dans notre cas, nos vertices possèdent 2 coordonnées, donc `in_Vertex` sera de type `vec2` :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée Shader
vec2 in_Vertex;

// Fonction main
```

```
void main()
{
}
```

Cette déclaration n'est pas encore tout à fait complète. En effet, si nous la laissons ainsi, OpenGL croira qu'il s'agit d'une variable normale sans aucun rapport avec les tableaux **VertexAttrib**. Pour corriger ça, il faut lui ajouter le mot-clef **in** juste avant son type :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée Shader
in vec2 in_Vertex;

// Fonction main
void main()
{}
```

Grâce à ce mot-clef, OpenGL saura que cette variable est reliée à un tableau **VertexAttrib**.

Enfin, maintenant que l'on a récupéré le vertex en cours, il faut l'utiliser pour qu'il puisse servir à quelque chose. 😊 Pour cela, nous allons affecter son contenu à une variable prédéfinie dans le **GLSL** (comme les fonctions mathématiques) qui se nomme **gl_Position** :

Code : C

```
vec4 gl_Position;
```

Cette variable est prédéfinie dans le **GLSL**, nous n'avons donc pas besoin de la déclarer en **in**. Elle permet, entre autres, de définir la position finale du vertex à l'écran, elle demande pour ça le contenu de la variable **in_Vertex**.

La seule difficulté que l'on peut rencontrer lors de cette affectation concerne le conflit de type entre les deux variables. En effet, **in_Vertex** est de type **vec2** alors que **gl_Position** est de type **vec4**. Cependant, vous devriez savoir comment surmonter ce problème non ? 😐

Pour combler ce manque de coordonnées, il suffit simplement d'utiliser un constructeur ! Et plus précisément, le constructeur **vec4()**. Nous lui donnerons la variable **in_Vertex** ainsi que deux autres coordonnées égales à **0.0** pour **Z**, car le carré n'est qu'en 2D pour le moment, et **1.0** pour **W**.

La coordonnée **W** d'un vertex est un peu spéciale. Si vous mettez d'autres valeurs comme **2.0** par exemple, vous remarquerez que la taille de votre rendu sera divisée par deux. Pour conserver une taille normale, il faut laisser la valeur **1.0** à **W**. C'est ce que nous ferons pour tous nos shaders.

Code : C

```
// Position finale du vertex
```

```
gl_Position = vec4(in_Vertex, 0.0, 1.0);
```

Ce qui donne :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée Shader
in vec2 in_Vertex;

// Fonction main
void main()
{
    // Position finale du vertex
    gl_Position = vec4(in_Vertex, 0.0, 1.0);
}
```

Je vous avais dit que les constructeurs étaient utiles. 🎉

Et voilà ! Notre premier **Vertex Shader** est terminé ! Certes, il ne fait pas grand chose mais au moins il fait exactement ce qu'on veut : il définit la position d'un vertex à l'écran. Sachez que ce bout de code sera présent dans **TOUS** vos futurs shaders, quelque soit leur complexité.

Les Entrées-Sorties

Avant d'aller plus loin dans le développement des codes sources, j'aimerais que l'on voit ensemble le fonctionnement des entrées-sorties au niveau des shaders. Vous connaissez déjà le mot-clé **in** qui permet de définir des données entrantes. Sachez qu'il existe aussi son opposé, le mot-clef **out**, qui permet de définir des données sortantes.

Ces termes sont assez subtiles car ils ne signifient pas la même chose en fonction du shader où ils sont utilisés.

Pour le **Vertex Shader**, ils signifient :

- **in** : Variable représentant les données des tableaux **Vertex Attrib** (vertices, couleurs et tutti quanti)
- **out** : Variable qui sera envoyée au shader suivant, soit le **Fragment Shader** dans notre cas

Pour le **Fragment Shader**, ces termes signifient :

- **in** : Variable représentant les données reçues depuis le shader précédent, soit celles étant déclarées avec le mot-clef **in** dans le **Vertex Shader**
- **out** : Variable représentant la couleur finale d'un pixel

Vous voyez qu'il y a une différence selon le shader qu'on utilise, faites bien la différence.

La variable **gl_Position** est un peu spéciale et ne fait pas partie de ce système d'entrées-sorties, elle est spécifique au **Vertex Shader**. Il y en a quelques unes comme ça mais c'est la la seule vraiment importante.

Le Fragment Shader

On passe maintenant au **Fragment Shader**. Celui-ci permet de définir la couleur de chaque pixel d'une surface affichée. Si vous avez utilisé une résolution de 800x600 pour votre fenêtre alors votre carte graphique devra gérer 120 000 pixels pour votre carré ! Non non vous ne rêvez pas, votre Fragment Shader devra gérer 120 000 pixels différent et cela 60 fois par seconde. 

Mais ne vous inquiétez pas, elle est justement faite pour ça. Pour vous dire, 120 000 c'est un nombre extrêmement petit, elle en gère beaucoup plus dans des applications développées.

Enfin, le code source du **Fragment Shader** commence par la même chose que précédemment : la balise `#version` et la fonction `main()`.

Code : C

```
// Version du GLSL
#version 150 core

// Fonction main
void main()
{}
```

Contrairement au **Vertex Shader**, nous n'aurons pas (pour le moment) de variable d'entrée ici, nous n'aurons donc pas besoin d'utiliser le mot-clef `in`. En revanche, nous devrons gérer une variable de sortie, donc nous devrons utiliser le mot-clef `out`.

Comme nous l'avons vu juste avant, la sortie d'un **Fragment Shader** correspond à la couleur finale du pixel. Il s'agit d'une variable `vec4` tout comme `gl_Position` sauf que cette fois-ci elle n'est pas prédéfinie, nous devons la déclarer nous-même. Nous l'appellerons `out_Color` :

Code : C

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{}
```

Cette variable demande 4 valeurs représentant les composantes **RGBA** d'une couleur. Chaque valeur doit être comprise entre **0** et **1**, comme lorsque nous utilisions le tableau **VertexAttrib** au début du tuto.

Pour affecter ces composantes à la variable `out_Color`, nous aurons besoin une fois de plus du constructeur `vec4()`. Pour le moment, nous leur affecterons 4 fois la valeur **1.0** de façon à avoir la couleur blanche :

Code : C

```
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Ce qui donne :

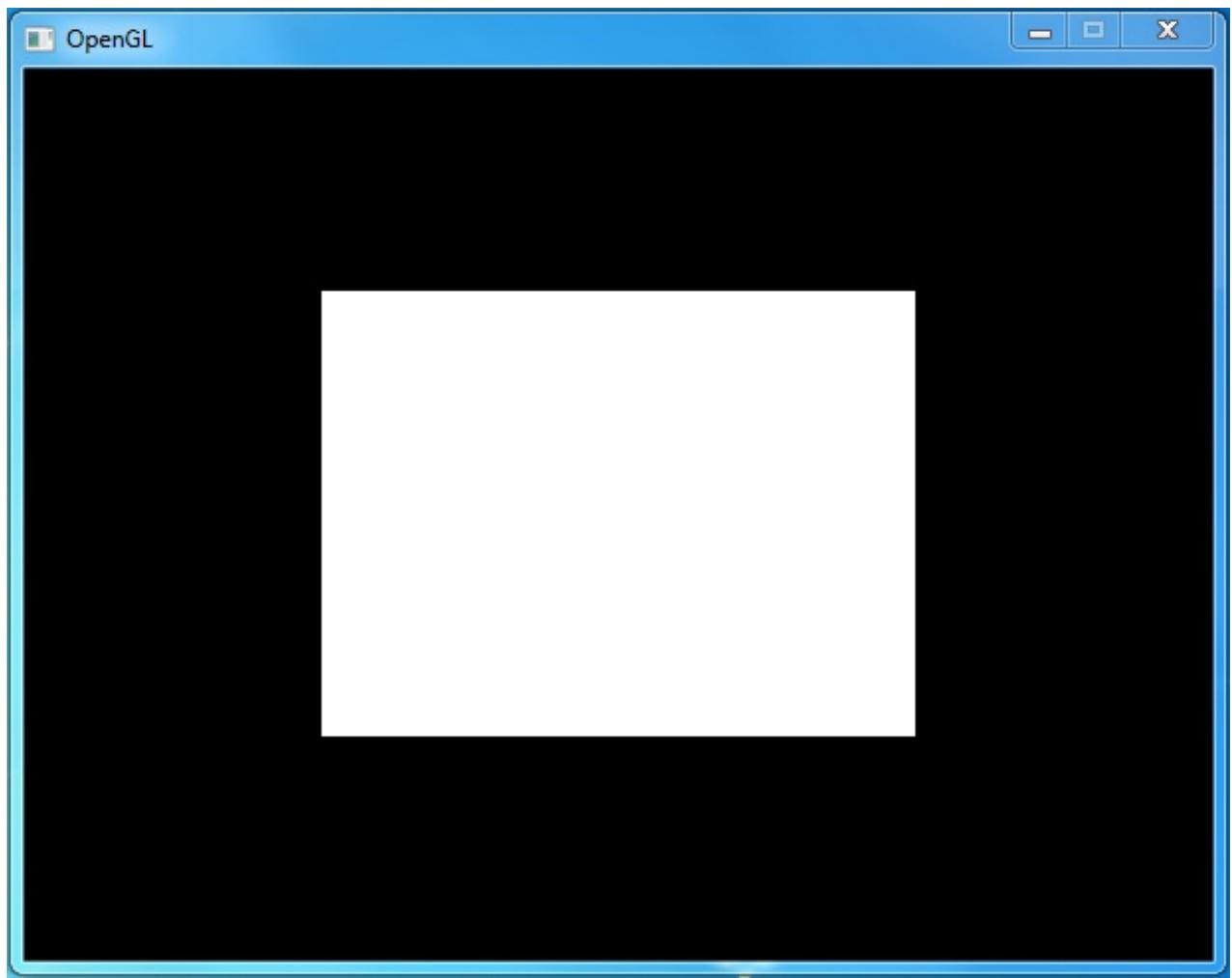
Code : C

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Et là, vous pouvez enfin compiler pour voir ce que ça donne :



Félicitation, vous venez de programmer votre premier shader ! Champagne ! 😊

Ce qu'il faut absolument retenir

Avant de passer à la suite, j'aimerais faire un petit point sur ce que vous devez absolument retenir de cette partie.

En ce qui concerne le **Vertex Shader**, vous devez savoir que :

- Les variables **in** représentent les tableaux **Vertex Attrib**
- Les variables **out** sont envoyées au shader suivant, soit le **Fragment Shader** dans notre cas
- La variable **gl_Position** doit être remplie avec la variable **in_Vertex**. Elle ne doit pas être déclarée en **out**

En ce qui concerne le **Fragment Shader**, vous devez savoir que :

- Les variables **in** représentent les variables **out** du shader précédent, soit le **Vertex Shader** dans notre cas
- La sortie du **Fragment Shader** représente la couleur finale du pixel
- Elle est représentée par une variable **out**

Il faut absolument que vous connaissez ces points **par cœur**. Ce sont vraiment les bases de la programmation **GLSL**. 😊

Utilisation de couleur

Changer de couleur

Le blanc n'est évidemment pas la seule couleur que l'on peut utiliser. Pour en prendre une autre, il suffit simplement de changer les valeurs données à la variable `out_Color`.

Par exemple, pour afficher le carré avec la couleur bleue, nous pouvons affecter les valeurs RGB : **0.0, 0.0, 1.0**. Pour la valeur Alpha, il vaut mieux la laisser à **1.0** pour le moment, quelque soit la couleur que l'on veut utiliser :

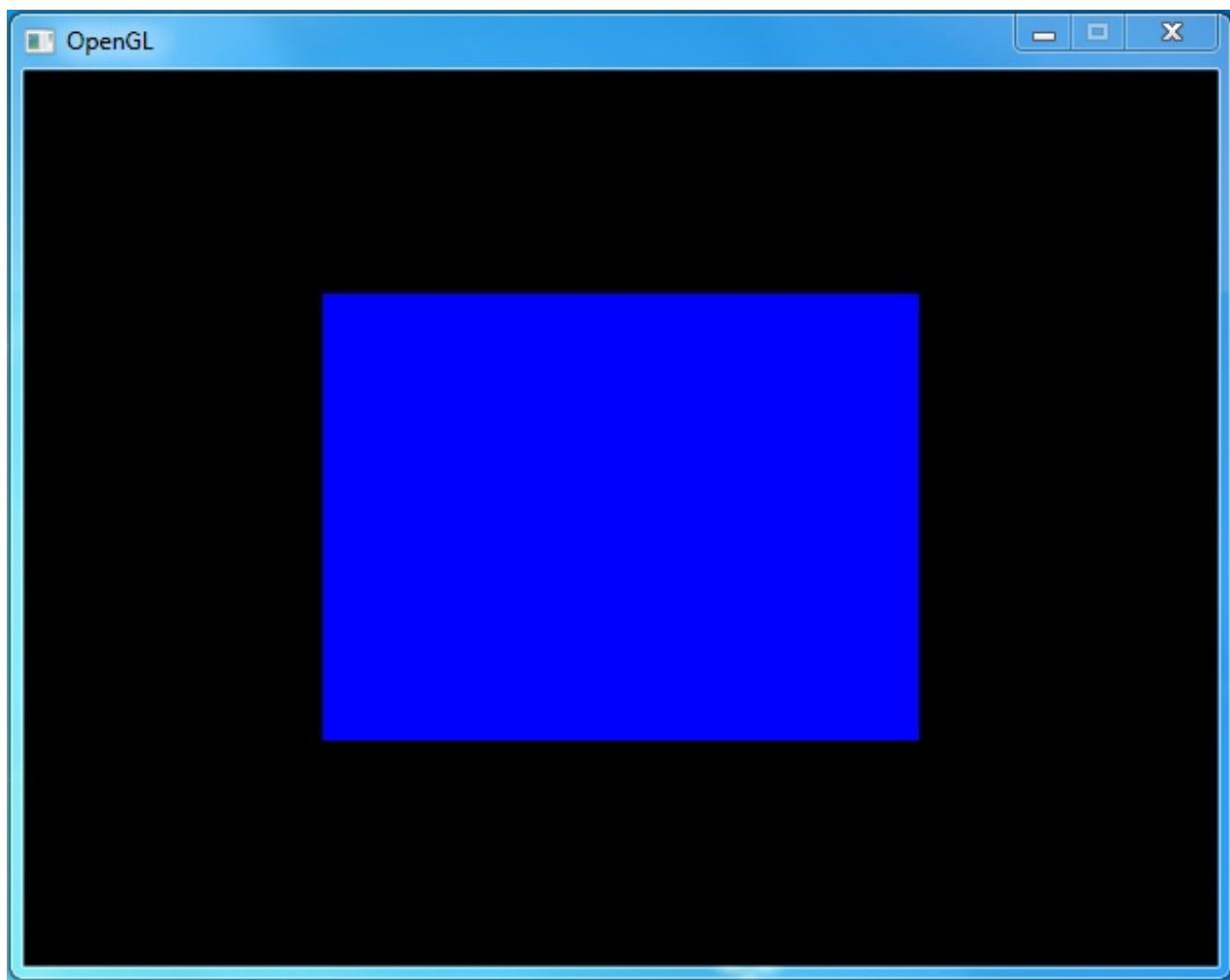
Code : C

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Si vous relancez le projet (sans le compiler 😊), vous devriez avoir ceci :



Exercices

Allez, je vais vous donner vos premiers exercices utilisant la programmation en **GLSL**. Ils sont très simples. 😊

Exercice 1 : Coloriez le carré en rouge.

Exercice 2 : Coloriez le carré en violet/rose (avec les composantes rouge et bleu).

Exercice 3 : Coloriez le carré en jaune.

Exercice 4 : Coloriez le carré en gris (n'importe quelle nuance).

Solutions

Exercice 1 :

Secret (cliquez pour afficher)

Il n'y a que le **Fragment Shader** à modifier (pour les 4 exercices), l'autre ne change absolument pas. 😊

Code : C

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Exercice 2 :**Secret** (cliquez pour afficher)**Code : C**

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 0.0, 1.0, 1.0);
}
```

Exercice 3 :**Secret** (cliquez pour afficher)**Code : C**

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;
```

```
// Fonction main

void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 1.0, 0.0, 1.0);
}
```

Exercice 4 :

Secret (cliquez pour afficher)

Code : C

```
// Version du GLSL
#version 150 core

// Sortie Shader
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(0.5, 0.5, 0.5, 1.0);
}
```

Gestion de la couleur

Préparation

Changement de shader

Avant toute chose, vu que nous ajoutons une nouvelle fonctionnalité à notre rendu il nous faut donc créer un nouveau shader. Créons donc deux nouveaux fichiers appelés **couleur2D.vert** et **couleur2D.frag** dans le dossier **Shaders**.

Il nous faut ensuite ajouter leur chemin dans notre objet **shader** :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Shader
    Shader shader("Shaders/couleur2D.vert",
    "Shaders/couleur2D.frag");
    shader.charger();

    ...
}
```

Déclaration du tableau de couleurs

Il y a bien longtemps, dans une galaxie loin... nous utilisions des tableaux de couleurs pour afficher nos modèles. Nous ne connaissions pas encore les textures, c'était alors le seul moyen d'afficher quelque chose de non-blanc à l'écran.

Pour fonctionner correctement, il fallait affecter une couleur pour chaque vertex et chacune d'elles possédait 3 composantes (Rouge Vert Bleu):

Code : C++

```
// Exemple de couleur  
float rouge = {1.0, 0.0, 0.0};
```

Dans notre code test actuel nous avons 6 vertices, nous aurons donc besoin d'un tableau de **6 couleurs x 3 composantes** soit **18 cases**. Nous utiliserons la couleur **bleu** pour le premier triangle et le **rouge** pour le second :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Shader  
    ...  
  
    // Vertices  
    float vertices[] = {-0.5, -0.5, 0.5, -0.5, 0.5, 0.5, //  
    Triangle 1  
                -0.5, -0.5, -0.5, 0.5, 0.5, 0.5}; //  
    Triangle 2  
  
    // Couleurs  
    float couleurs[] = {0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,  
    1.0, // Triangle 1  
          1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,  
    0.0}; // Triangle 2  
  
    ...  
}
```

Avec ce tableau, chacun de nos vertices aura sa propre couleur.

Gestion du VBO et du VAO

Bien évidemment, déclarer un tableau comme ça ne sert pas à grand chose, il faut l'intégrer au **VBO** et au **VAO**.

Pour ce qui est du premier, il va falloir le redimensionner car il ne peut accueillir que les vertices pour le moment. On commence donc pas créer une variable **tailleCouleursBytes** qui contiendra la taille du tableau de couleurs en bytes :

Code : C++

```
// VBO et taille des données

GLuint vbo;

int tailleVerticesBytes = 12 * sizeof(float);
int tailleCouleursBytes = 18 * sizeof(float);
```

On redimensionne ensuite le **VBO** en additionnant les deux tailles de données :

Code : C++

```
// Nouvelle taille du VBO

glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes +
tailleCouleursBytes, 0, GL_STATIC_DRAW);
```

Et enfin, on le remplit avec le tableau de couleurs :

Code : C++

```
// Envoi des données

glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes, vertices);
glBufferSubData(GL_ARRAY_BUFFER, tailleVerticesBytes,
tailleCouleursBytes, couleurs);
```

Au niveau du **VAO**, nous devons juste appeler puis activer le tableau **Vertex Attrib 1** qui correspond à l'envoi des couleurs :

Code : C++

```
// Verrouillage du VAO

glBindVertexArray(vao);

// Verrouillage du VBO

glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Vertex Attrib 0 (Vertices)

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

// Vertex Attrib 1 (Couleurs)

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(tailleVerticesBytes));
 glEnableVertexAttribArray(1);

// Déverrouillage du VBO

glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
// Déverrouillage du VAO  
glBindVertexArray(0);
```

Gestions des shaders

Le Vertex Shader

Contrairement à ce qu'on pourrait penser au premier abord, les couleurs ne sont absolument pas envoyées au **Fragment Shader** mais bel et bien au **Vertex Shader**. C'est lui qui récupère toutes les données issues des tableaux **Vertex Attrib**. Si nous voulons travailler sur les pixels avec une couleur envoyée, alors il faudra la transférer manuellement au **Fragment Shader**.

Cependant, avant de faire cela il va falloir la récupérer dans notre source, tout comme nous l'avons fait avec le vertex (**in_Vertex**). La variable qui permet d'accéder aux couleurs s'appelle **in_Color**, nous l'avions appelée ainsi lors du linkage du shader. Vu qu'elle possède 3 composantes RGB alors elle sera de type **vec3**. Si nous avions spécifié la valeur Alpha dans le tableau de couleurs, elle aurait été de type **vec4**.

Code : C

```
// Entrée Shader  
in vec3 in_Color;
```

Ce qui donne le code source suivant :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrées  
in vec2 in_Vertex;  
in vec3 in_Color;  
  
// Fonction main  
void main()  
{  
    // Position finale du vertex  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
}
```

Maintenant que l'on a récupéré la couleur du vertex, nous allons pouvoir l'envoyer au **Fragment Shader**. Pour cela, nous allons utiliser une variable **out** !

Et oui, rappelez-vous que les variables **out** du **Vertex Shader** sont automatiquement envoyées au shader suivant. La seule condition pour pouvoir les utiliser est qu'elles doivent être **strictement** identiques dans les deux codes sources. C'est-à-dire qu'elles doivent avoir le même **type** et surtout le même **nom**. Si vous ne respectez pas l'une de ces deux conditions, alors OpenGL ne fera pas le lien entre les deux. ☺

Pour envoyer notre couleur donc, nous devons utiliser une variable **out** qui sera du type **vec3** de façon à pouvoir envoyer toutes les composantes. Nous l'appellerons simplement **color** :

Code : C

```
// Sortie  
out vec3 color;
```

Ce qui donne le code suivant :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrées  
in vec2 in_Vertex;  
in vec3 in_Color;  
  
// Sortie  
out vec3 color;  
  
// Fonction main  
void main()  
{  
    // Position finale du vertex  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
}
```



Pourquoi on ne l'appelle pas **out_Color** ? Vu qu'elle sort du shader on peut la nommer ainsi ?

Vous pouvez certes, mais il y aura un petit problème de logique. En effet, je vous ai dit qu'elle doit avoir le même nom dans les deux codes sources, ce qui voudrait dire que la variable d'entrée **in** du **Fragment Shader** aurait un nom qui commencerait pas **out_** ? Ce n'est pas logique évidemment. 😊

Enfin bref, maintenant que nous avons un moyen de communiquer avec le **Fragment Shader**, nous allons pouvoir lui envoyer le contenu de la variable **in_Color**. Nous n'avons pas besoin de constructeur vu qu'elles sont de même type :

Code : C

```
void main()  
{  
    // Position finale du vertex  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
  
    // Envoi de la couleur au Fragment Shader
```

```
    color = in_Color;  
}
```

Ce qui donne le code source final :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrées  
  
in vec2 in_Vertex;  
in vec3 in_Color;  
  
// Sortie  
  
out vec3 color;  
  
// Fonction main  
  
void main()  
{  
    // Position finale du vertex  
  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
  
    // Envoi de la couleur au Fragment Shader  
  
    color = in_Color;  
}
```

Et voilà, **Vertex Shader** terminé. 😊

Le Fragment Shader

La **Fragment Shader** va être très simple à programmer car il ne fera que récupérer une couleur pour la renvoyer ensuite dans une variable de sortie **out**.

La première chose à faire va être de re-déclarer la variable **color** qui sort du **Vertex Shader** précédent. Et comme vous le savez maintenant, elle doit être exactement identique au code source précédent. On conserve donc le même **type** et le même **nom**, la seule différence va être le mot-clé utilisé car elle ne ~~s'sort~~ pas mais elle *rentre* dans le **Fragment Shader**. On utilise donc non pas le mot-clé **out** mais **in** :

Code : C

```
// Entrée  
  
in vec3 color;
```

Ce qui donne :

Code : C

```
// Version du GLSL

#version 150 core

// Entrée
in vec3 color;

// Sortie
out vec4 out_Color;

// Fonction main

void main()
{
    // Couleur finale du pixel
    out_Color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Le nom et le type sont respectés, la variable **color** est donc utilisable. D'ailleurs, nous allons l'utiliser tout de suite en enlevant le code de la couleur blanc pour affecter son contenu à la variable **out_Color**.

Si vous faites attention, vous remarquerez qu'il y a un conflit de type entre les deux variables. En effet, l'une est de type **vec3** alors que l'autre est de type **vec4**. Pour régler ce problème, il va falloir utiliser un constructeur, encore une fois ! 😊

Et pour mon plus grand plaisir, je vais vous demander de convertir la variable **color** vous-même. Ce n'est pas compliqué bien sûr, nous avons déjà fait cet opération plusieurs fois, vous devez juste convertir un **vec3** et **vec4** en rajoutant la composante Alpha (**1.0**).

.....

Vous avez trouvé ?

Voici la solution :

Secret (cliquez pour afficher)**Code : C**

```
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(color, 1.0);
}
```

Il ne manque qu'une coordonnée au **vec3** pour devenir un **vec4**. On appelle donc le bon constructeur en ajoutant la composante Alpha à la variable **color**. Le résultat sera une variable de type **vec4**.

Ce qui donne le code source final :

Code : C

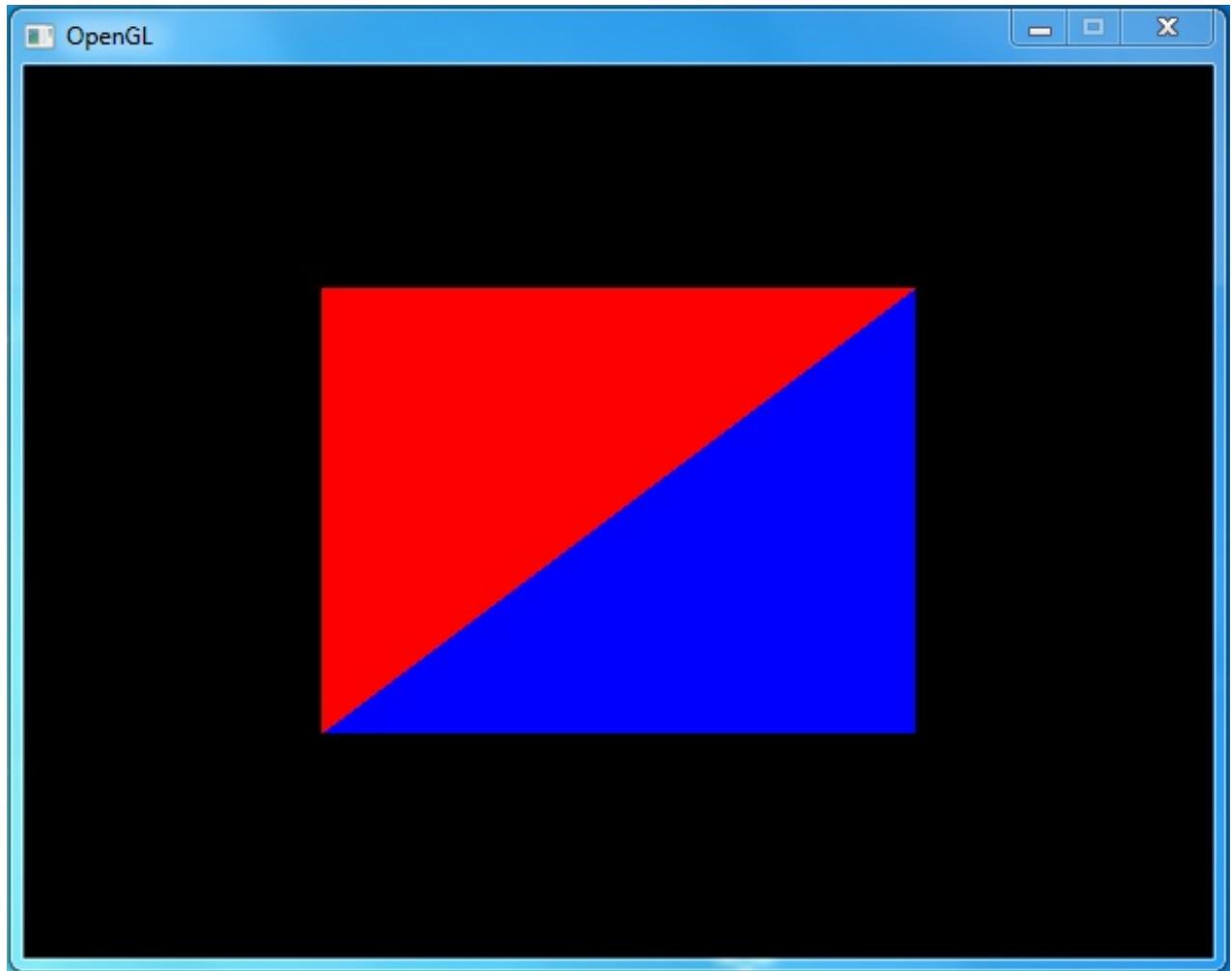
```
// Version du GLSL
#version 150 core

// Entrée
in vec3 color;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(color, 1.0);
}
```

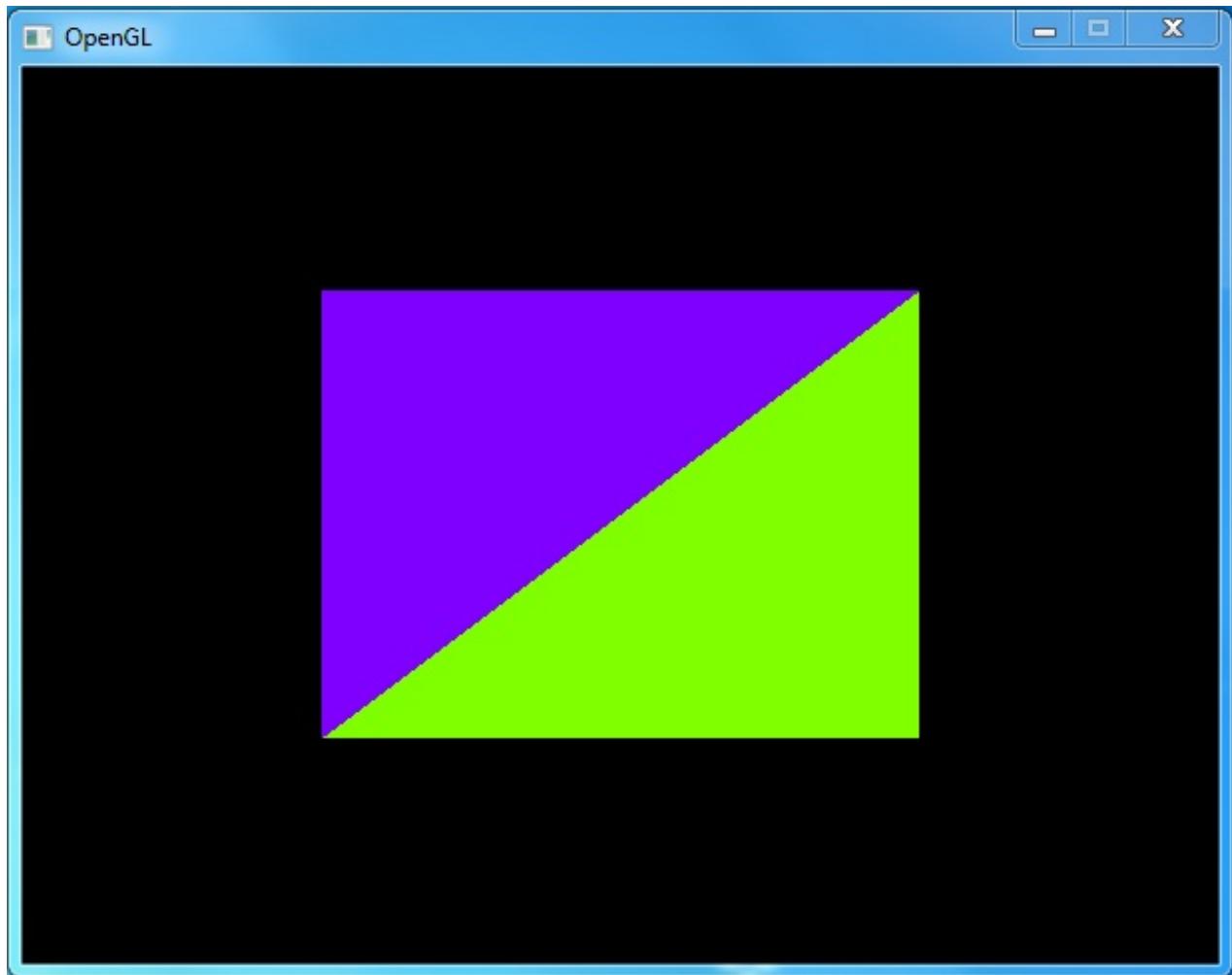
Vous pouvez maintenant relancer votre projet, vous devriez avoir le résultat suivant :



Notre shader est maintenant capable de gérer les couleurs. 😊 Vous pouvez même changer le tableau des composantes pour vérifier que tout fonctionne correctement :

Code : C++

```
float couleurs[] = {0.5, 1.0, 0.0,    0.5, 1.0, 0.0,    0.5, 1.0, 0.0,  
// Triangle 1           0.5, 0.0, 1.0,    0.5, 0.0, 1.0,    0.5, 0.0,  
1.0}; // Triangle 2
```



Exercices

Énoncés

On continue notre petite vague d'exercices à l'instar de la partie précédente. Je vais monter un peu le niveau cette fois, il faudra réfléchir un peu plus. 🤔

Exercice 1 : Ajoutez la composante Alpha pour chaque couleur dans le tableau couleurs. Modifiez ensuite le shader pour gérer les couleurs entrantes à 4 composantes. (Pensez à modifier le VBO et le VAO pour prendre en compte la nouvelle composante.)

Exercice 2 : Inversez la couleur entrante dans le **Fragment Shader** avant de l'affecter à la couleur sortante. Pour vous donner un indice : inverser une couleur revient à inverser l'ordre des composantes en passant de l'ordre RGB à BGR. La composante Alpha reste cependant toujours à la fin. Vous pouvez reprendre le code de l'exercice précédent, ou utiliser celui du cours.

Exercice 3 : Oubliez complètement le tableau de couleurs et créez une variable **maCouleur** (à la place de **Color**) de type **vec4** dans le **Vertex Shader**. Ses composantes doivent permettre d'afficher du bleu. L'objectif est d'envoyer cette variable au **Fragment Shader** à la place des données issues du tableau **Vertex Attrib**.

Exercice 4 : Reprenez le même principe que l'exercice précédent (voire correction si besoin) sauf que la couleur à envoyer doit dépendre de l'abscisse du vertex. Si la coordonnée **x** est supérieure à zéro alors vous devez envoyer la couleur bleu, si elle est inférieure ou égale à zéro alors vous devez envoyer la couleur rouge.

Solutions

Exercice 1 :

Secret (cliquez pour afficher)

Pour ajouter la composante Alpha aux couleurs, il suffit de rajouter la valeur **1.0** au tableau **couleurs[]** :

Code : C++

```
float couleurs[] = {0.5, 1.0, 0.0, 1.0,    0.5, 1.0, 0.0, 1.0,
0.5, 1.0, 0.0, 1.0,    // Triangle 1
                    0.5, 0.0, 1.0, 1.0,    0.5, 0.0, 1.0, 1.0,
0.5, 0.0, 1.0, 1.0}; // Triangle 2
```

Pensez à modifier la variable **tailleCouleursBytes** pour prendre en compte les nouvelles données soit **4 composantes x 3 coordonnées x 2 triangles = 24 valeurs** :

Code : C++

```
// Gestion du VBO

GLuint vbo;
int tailleVerticesBytes = 12 * sizeof(float);
int tailleCouleursBytes = 24 * sizeof(float);
```

Pensez également à modifier le paramètre **size** du tableau **VertexAttrib 1** pour le passer à **4** (pour **4** composantes) :

Code : C++

```
// Vertex Attrib 1 (Couleurs)

glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(tailleVerticesBytes));
 glEnableVertexAttribArray(1);
```

Dans le **Vertex Shader**, la variable entrante **in_Color** n'est plus de type **vec3** mais de type **vec4** maintenant. La variable sortante **color** devient donc elle-aussi une **vec4** :

Code : C

```
// Version du GLSL
#version 150 core

// Entrées
in vec2 in_Vertex;
in vec4 in_Color;

// Sortie
out vec4 color;

// Fonction main
void main()
{
    // Position finale du vertex
    gl_Position = vec4(in_Vertex, 0.0, 1.0);

    // Envoi de la couleur au Fragment Shader
    color = in_Color;
}
```

Dans le **Fragment Shader**, on commence évidemment par changer le type de la variable **color** en **vec4**. Puis on enlève le constructeur vu qu'elle possède maintenant le même type que la variable sortante **out_Color**. On peut donc l'affecter normalement :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée
in vec4 color;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = color;
}
```

Exercice 2 :

Secret (cliquez pour afficher)

Pour inverser une couleur, il suffit juste d'inverser les 'sous-variables' (**x**, **y**, **z**) avant d'affecter la variable **color** à **out_Color**. On peut appeler le constructeur **vec4()** ou affecter les valeurs à la main, comme vous voulez. Mais il est préférable d'utiliser la première solution :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée
in vec4 color;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(color.z, color.y, color.x, color.w);
}
```

Il y a un moyen de raccourcir encore plus ce code et même d'utiliser d'autres noms pour les composantes, mais je ne vous en parle pas maintenant pour éviter de nous embrouiller l'esprit. 😊

Le **Vertex Shader** n'a pas besoin d'être modifié car il ne fait qu'envoyer la couleur, il ne fait pas de traitement dessus.

Exercice 3 :**Secret** (cliquez pour afficher)

On commence par supprimer la variable **Color** dans le **Vertex Shader** pour la remplacer par une nouvelle. On la nomme **maCouleur** et on lui donne le type **vec4**. On utilise également le mot-clef **out** puisqu'elle sort du shader :

Code : C

```
// Sortie
out vec4 maCouleur;
```

Ensuite, on lui assigne les 4 composantes permettant d'afficher du bleu :

Code : C

```
void main()
{
    // Position finale du Vertex
```

```
gl_Position = vec4(in_Vertex, 0.0, 1.0);

// Envoi de la couleur au Fragment Shader

maCouleur = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Ce qui donne le code source :

Code : C

```
// Version du GLSL

#version 150 core

// Entrées

in vec2 in_Vertex;
in vec4 in_Color;

// Sortie

out vec4 maCouleur;

// Fonction main

void main()
{
    // Position du vertex

    gl_Position = vec4(in_Vertex, 0.0, 1.0);

    // Envoi de la couleur au Fragment Shader

    maCouleur = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Quant au **Fragment Shader**, on supprime simplement la variable **in Color** pour la remplacer par **maCouleur** afin de respecter les conditions des entrées-sorties :

Code : C

```
// Entrée

in vec4 maCouleur;
```

Il ne reste plus qu'à assigner le contenu de la variable **maCouleur** à **out_Color**. On n'utilise pas de constructeur vu qu'elles sont de même type :

Code : C

```
// Version du GLSL

#version 150 core
```

```
// Entrée  
in vec4 maCouleur;  
  
// Sortie  
out vec4 out_Color;  
  
// Fonction main  
void main()  
{  
    // Couleur finale du pixel  
    out_Color = maCouleur;  
}
```

Exercice 4 :

Secret (cliquez pour afficher)

On reprend le même code que précédemment auquel on va rajouter une condition pour choisir une couleur dans le **Vertex Shader**. Si la coordonnée **x** de la variable **in_Vertex** est supérieure à zéro alors on assigne la couleur **bleu** à **maCouleur**. Dans le cas contraire, on assigne la couleur **rouge** :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrées  
in vec2 in_Vertex;  
in vec4 in_Color;  
  
// Sortie  
out vec4 maCouleur;  
  
// Fonction main  
void main()  
{  
    // Position du vertex  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
  
    // Si la coordonnée x est supérieure à 0  
    if(in_Vertex.x > 0)  
        maCouleur = vec4(0.0, 0.0, 1.0, 1.0);  
  
    // Dans le cas inverse  
    else
```

```
    }  
    maCouleur = vec4(1.0, 0.0, 0.0, 1.0);
```

Il n'y a rien à changer au niveau du **Fragment Shader**.

Télécharger (Windows, UNIX/Linux, Mac OS X) : [code source C++ du chapitre sur les shaders \(Partie 1/2\)](#)

Nous avons vu pas mal de choses dans ce chapitre, nous avons même appris à programmer nos premiers effets ! 😊

Vous savez maintenant comment gérer les vertices du coté des shaders pour afficher quelque chose à l'écran. Vous savez même gérer les tableaux de couleur pour colorier toutes les surfaces à votre guise. Ces effets sont assez simples et ne gèrent pas la 3D mais au moins, vous avez fait vos premiers pas dans le développement de shaders.

Si vous êtes prêts, je vous invite à lire le prochain chapitre qui va nous permettre d'apprendre à intégrer la troisième dimension et afficher des textures. 😊

Les shaders démystifiés (Partie 2/2)

Précédemment, nous avons commencé à démystifier un peu les shaders en étudiant leur comportement dans un environnement en deux dimensions. Nous avons même vu comment gérer la couleur grâce aux tableaux **Vertex Attrib** et envoyer des variables d'un shader à un autre.

Ces échanges de données seront quasiment toujours présents quelque soit les effets que l'on voudra programmer. Si vous n'êtes pas à l'aise avec ça, je vous conseille vivement de relire le chapitre précédent. 😊

D'ailleurs, ce que nous allons voir aujourd'hui concerne également les échanges de données sauf qu'ici, nous parlerons d'échanges entre l'application principale (en **C++**) et les shaders (en **GLSL**). Grâce à eux, nous pourrons enfin envoyer nos matrices à nos codes sources et nous pourrons enfin gérer la 3D. 😊

Nous parlerons aussi des derniers petits points à connaître sur la programmation **GLSL**.

Les variables uniform

Qu'est-ce qu'une variable uniform ?

On commence ce chapitre par une notion que l'on retrouvera dans tous les shaders un tant soit peu développés. 😊

Les variables **uniform** sont des variables envoyées depuis une application classique (codée en **C++**) jusqu'à un shader. Elles peuvent être utilisées pour envoyer tous les types de données possibles au niveau du **GLSL** soit les variables classiques (**float**, **int**, etc.), les vecteurs et les matrices. Nous connaissons particulièrement ce dernier cas car nous avons déjà eu l'occasion d'envoyer les fameuses matrices **projection** et **modelview**, ce qui permettait d'intégrer la troisième dimension à nos applications.

D'ailleurs lorsque nous les avons vues, je vous avais fourni deux fonctions totalement incompréhensibles qui permettaient d'envoyer des matrices aux shaders. Elles ressemblaient à ceci :

Code : C++

```
// Envoi des matrices au shader
glUniformMatrix4fv(glGetUniformLocation(shader.getProgramID(),
    "projection"), 1, GL_FALSE, value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shader.getProgramID(),
    "modelview"), 1, GL_FALSE, value_ptr(modelview));
```

Complément incompréhensible bien sûr. 🤪

Vous remarquerez cependant l'utilisation du terme **uniform** dans le nom de la fonction **glUniformMatrix4fv()**. Ce qui signifie que les matrices sont bien considérées comme des variables **uniform**. Elles arrivent donc directement dans le shader. C'est un exemple d'utilisation que l'on rencontrera tout le temps, mais les matrices ne sont pas les seules valeurs à pouvoir être envoyées comme nous allons le voir.

Envoyer une variable simple

Le mot-clef uniform

Avant d'étudier en détails les fonctions que l'on vient de voir, nous allons faire un petit détour au niveau des codes sources **GLSL**. Les variables **uniform** se comportent un peu comme les **in** et les **out** dans le sens où ils possèdent un mot-clef qui leur est propre. Ce mot-clef est simplement **uniform**.

Prenons un petit exemple en déclarant une variable **float** en tant qu'**uniform** :

Code : C

```
// Version du GLSL

#version 150 core

// Variable uniform

uniform float maVariable;

// Fonction main

void main()
{
}
```

Ce nouveau mot-clef indique au shader que l'application enverra une variable au code source **GLSL**. Bien évidemment, celle-ci se comportera comme une variable classique sauf qu'on ne pourra pas la modifier. Nous pourrons toujours l'utiliser dans des opérations arithmétiques, dans des conditions, ... mais nous ne pourrons jamais modifier sa valeur directement.

Il faut savoir également qu'une variable **uniform** est accessible dans les deux types de shaders (**Vertex** et **Fragment**), nous n'avons donc pas besoin de les manipuler avec les entrées-sorties ni de spécifier dans quel code source les utiliser. Ça les rend plus simple à utiliser. 😊

La localisation et son utilisation

On revient maintenant aux fonctions de gestion des **uniform** au niveau de l'application. Nous en avons rapidement survolé une que nous connaissons déjà depuis un moment et qui s'appelait **glUniformMatrix4fv()**. Celle-ci permet d'envoyer une matrice directement au shader.

Cependant, ce n'est pas la seule fonction qui permet d'envoyer des données. Elle possède plusieurs 'œurs' qui prennent en compte d'autres types comme les variables simples (**float**, **int**, etc.) et les vecteurs. Tous ces cas se gèrent un peu de la même façon, il n'y a que les paramètres à utiliser qui vont changer.

D'ailleurs nous allons étudier en premier le cas des variables simples car c'est celui qui en requiert le moins. 🍪 Nous aurons besoin pour cela d'utiliser une fonction dont le nom se rapproche fortement de celui que nous connaissons déjà. Elle s'appelle **glUniform1f()** :

Code : C++

```
void glUniform1f(GLint location, GLfloat value);
```



A noter qu'il existe aussi les fonctions **glUniform1i()** et **glUniform1ui()** qui permettent d'envoyer respectivement des entiers et des entiers non-signés.

- **location** : paramètre permettant de retrouver la variable utilisant le mot-clef **uniform** dans le code source **GLSL**
- **value** : La valeur (de type **float**) que vous souhaitez envoyer

Le premier paramètre peut vous sembler un peu flou et c'est tout à fait normal. Pour le comprendre, nous allons faire une analogie avec la fonction **glBindAttribLocation()**. Celle-ci demandait, parmi ses paramètres, le nom correspondant à la variable de destination dans le code **GLSL**. Par exemple, nous l'utilisions pour faire le lien entre la variable **in_Vertex** et le tableau de vertices. Même chose avec la variable **in_Color** et le tableau de couleurs.

Cette fonction nous permettait donc de spécifier directement un nom pour retrouver son destinataire dans le code **GLSL**.

Le problème avec les **uniform**, c'est que ce système de nom n'existe pas directement. Il faut passer par une fonction intermédiaire

pour faire le lien entre le shader et l'application. Cette fonction intermédiaire s'appelle **glGetUniformLocation()** :

Code : C++

```
GLint glGetUniformLocation(GLuint program, const GLchar *name)
```

- **program** : l'ID du programme shader. Nous utiliserons la méthode **glGetProgramID()** pour lui affecter une valeur
- **name** : nom de la variable dans le code source **GLSL**

Grâce à elle, nous pouvons trouver facilement le paramètre **location** de **glUniform1f()**. 😊

On reprend notre petit exemple précédent qui contenait la variable **uniform maVariable**. Pour récupérer sa localisation au sein du shader, on appelle donc la fonction **glGetUniformLocation()** en donnant en paramètre l'ID du programme ainsi qu'une chaîne de caractère contenant le nom "**maVariable**" :

Code : C++

```
// Localisation de maVariable  
  
int localisation = glGetUniformLocation(shader.getProgramID(),  
"maVariable");
```

Maintenant que l'on connaît la localisation de **maVariable**, nous pouvons lui envoyer une valeur. On utilise pour cela la fonction **glUniform1f()** en donnant en paramètre cette fameuse localisation ainsi qu'une valeur (par exemple **8.5**) :

Code : C++

```
// Localisation de maVariable  
  
int localisation = glGetUniformLocation(shader.getProgramID(),  
"maVariable");  
  
// Envoi d'une valeur à maVariable  
  
glUniform1f(localisation, 8.5);
```



Vous ne devez envoyer vos variable **uniform** qu'après avoir activé votre shader, sinon il aura un peu de mal à les recevoir.

La variable **uniform** dans le code **GLSL** possèdera maintenant la valeur **8.5**. 😊

Petit détail, il est tout à fait possible de combiner les deux fonctions utilisées comme nous l'avons vu pour les matrices :

Code : C++

```
// Envoi d'une valeur à maVariable  
  
glUniform1f(glGetUniformLocation(shader.getProgramID(),  
"maVariable"), 8.5);
```

Point récapitulatif

Nous allons faire un petit point récapitulatif car ce que nous venons de voir constitue la base de l'envoi de variable au shader. Il n'y a que trois points à retenir alors ouvrez grand vos oreilles (ou plutôt vos yeux

- Premièrement, il faut déclarer une variable avec le mot-clef **uniform** dans le code source **GLSL**.
- Ensuite, on récupère sa localisation grâce à la fonction **glGetUniformLocation()** du côté de l'application **C++**
- Puis, on envoie la valeur souhaitée grâce à la fonction **glUniform1f()** ou ses variantes

Je le répète, ces 3 points constituent la base du fonctionnement des **uniform**. Retenez-les bien. Nous ferons quelques exercices à la fin de cette partie pour vous familiariser avec eux.

Envoyer un vecteur

Côté GLSL

Comme nous l'avons vu précédemment, il existe une multitude de variantes de la fonction **glUniform1f()** qui permettent d'envoyer d'autres types de variable que les **float** ou les **int**. Nous allons justement voir une de ces variantes qui nous servira à envoyer des vecteurs (à 2, 3, ou 4 coordonnées).

Pour illustrer cela, nous allons déclarer une variable **position** de type **vec3** avec le mot-clef **uniform** dans le **Vertex Shader** :

Code : C

```
// Version du GLSL
#version 150 core

// Variable uniform
uniform vec3 position;

// Fonction main
void main()
{}
```

Côté application

Le détail qui vous a peut-être frappé dans le nom de la fonction **glUniform1f()** est l'utilisation du *chiffre 1* qui, apparemment, n'avait rien à faire ici. Ce chiffre est en fait très important car il permet de déterminer le nombre de données à envoyer à la variable de destination.

Lorsqu'il s'agit d'un variable simple comme un **float**, nous n'avons besoin d'envoyer qu'une seule valeur. Cependant lorsque l'on parle de vecteur, le nombre de données à envoyer devient un peu plus important car il faut affecter une valeur à chacune de ses coordonnées.

Vu qu'il existe 3 types de vecteur (**vec2**, **vec3** et **vec4**), il existe donc 3 formes possibles pour la fonction **glUniform***(). La seule chose qui va être modifiée avec celle que l'on connaît déjà est le fameux *chiffre* utilisé à la fin du nom. Pour envoyer un vecteur à 2 coordonnées par exemple, il faudra mettre la valeur **2**, ce qui donnera la fonction suivante :

Code : C

```
// Vecteur à 2 coordonnées  
  
void glUniform2f(GLint location, GLfloat value0, GLfloat value1);
```

- **location** : localisation du vecteur au niveau **GLSL**
- **value0** : valeur pour la coordonnée **x**
- **value1** : valeur pour la coordonnée **y**

Pour les vecteurs à 3 ou 4 coordonnées, il faudra mettre le chiffre correspondant au nom de la fonction **glUniform()** :

Code : C++

```
// Vecteur à 3 coordonnées  
  
void glUniform3f(GLint location, GLfloat value0, GLfloat value1,  
GLfloat value2);  
  
// Vecteur à 4 coordonnées  
  
void glUniform4f(GLint location, GLfloat value0, GLfloat value1,  
GLfloat value2, GLfloat value3);
```

- **location** : localisation du vecteur au niveau **GLSL**
- **value0** : valeur pour la coordonnée **x**
- **value1** : valeur pour la coordonnée **y**
- **value2** : valeur pour la coordonnée **z**
- **value3** : valeur pour la coordonnée **w**

Si on reprend l'exemple du vecteur **vec3 position** déclaré dans le shader, la fonction à utiliser sera **glUniform3f()**. Il faudra lui donner en paramètre la localisation de la variable ainsi qu'une valeur pour chacune de ses 3 coordonnées :

Code : C++

```
// Localisation du vecteur position  
  
int localisation = glGetUniformLocation(shader.getProgramID(),  
"position");  
  
// Envoi d'une valeur au vecteur position  
  
glUniform3f(localisation, 1.0, 2.0, 3.0);
```

Le vecteur **position** est maintenant utilisable dans notre shader :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Variable uniform  
  
uniform vec3 position;
```

```
// Fonction main

void main()
{
    // Exemple d'utilisation

    if(position.x >= 0)
        ....;

    else
        ....;
}
```

Le fonctionnement reste évidemment le même pour les autres types de vecteur.

Envoyer une matrice

Côté GLSL

Avant de s'occuper de la partie application, nous allons déclarer une matrice dans le **Vertex Shader**. Nous l'appellerons **modelview** en référence avec une certaine matrice que l'on connaît déjà. 😊

Nous déclarons donc une matrice carrée d'ordre 4 (**mat4**) avec le mot-clé **uniform** :

Code : C

```
// Version du GLSL

#version 150 core

// Variable uniform

uniform mat4 modelview;

// Fonction main

void main()
{}
```

"Transposer" une matrice

Les envois de matrice se passent quasiment de la même façon que ceux que nous venons d'étudier. A vrai dire, il n'y a que deux paramètres supplémentaires à gérer. Pour envoyer une matrice carrée d'ordre 4 par exemple, la fonction à utiliser s'appelle **glUniformMatrix4fv()** :

Code : C++

```
void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);
```

- **location** : localisation de la matrice au niveau **GLSL**
- **count** : nombre de sous-tableaux utilisés par la matrice. Nous lui affecterons la valeur **1.0** car nous n'en enverrons qu'un seul.
- **transpose** : *booléen* pouvant prendre la valeur **GL_TRUE** ou **GL_FALSE** et qui permet de transposer une matrice. Nous verrons ce que cela signifie dans un instant
- **value** : pointeur sur les valeurs de la matrice. Nous lui donnerons le résultat de la méthode **getValeurs()**



A noter aussi qu'il existe les fonctions **glUniformMatrix2fv()** et **glUniformMatrix3fv()** qui permettent d'envoyer respectivement des matrices carrées d'ordre 2 et 3. Elles prennent les mêmes paramètres que celle que nous étudions actuellement.

Le paramètre **transpose** peut vous sembler un peu flou pour le moment. C'est parce que nous n'avons pas encore vu ce que voulait dire le terme "*transposer*" une matrice. C'est pourtant une notion assez simple qui signifie "*inverser*" les valeurs d'une matrice pour que les lignes se retrouvent à la place des colonnes et les colonnes à la place des lignes :

	[0]	[1]	[2]	[3]
[0]	3	1	4	2
[1]	4	5	1	3
[2]	2	7	3	4
[3]	1	6	5	1

Si vous utilisez **GLM**, ce qui est normalement le cas, vous n'avez pas à vous soucier de ce paramètre car la librairie est en parfaite adéquation avec le **GLSL**. C'est-à-dire que les objets **mat4** en C++ se lisent aussi en colonne et c'est exactement ce que veut le shader :

Lecture en colonne

3	1	4	2
4	5	1	3
2	7	3	4
1	6	5	1

Si vous utilisez une autre librairie mathématique pour gérer vos matrices, je vous conseille de vérifier leur ordre de lecture car elles doivent peut-être être transposées. C'est-à-dire qu'elles se lisent peut-être en ligne :

Lecture en ligne

3	1	4	2
4	5	1	3
2	7	3	4
1	6	5	1

Si c'est le cas, vous devez les transposer à l'aide du paramètre **transpose**.

Coté application

Ceci étant dit, nous pouvons maintenant repasser à la fonction . Nous allons pouvoir l'utiliser pour envoyer notre matrice **modelview** à la variable du même nom dans le code source **GLSL**.

Nous savons qu'elle prendra 4 paramètres :

- **location** : qu'il faudra déterminer comme nous savons déjà le faire
- **count** : auquel nous affecterons la valeur **1.0**
- **transpose** : auquel il faudra affecter la valeur **GL_FALSE** car nous n'avons pas besoin de transposer nos matrices
- **value** : qui demande un pointeur sur les valeurs à envoyer

Ainsi, pour envoyer la matrice **modelview** au sahder, nous devrons d'abord la localiser à l'aide de la fonction **glGetUniformLocation()** :

Code : C++

```
// Localisation de la matrice modelview

int localisation = glGetUniformLocation(shader.getProgramID(),
"modelview");
```

Puis, nous envoyons ses valeurs grâce à la fonction **glUniformMatrix4fv()** avec les paramètres cités précédemment :

Code : C++

```
// Localisation de la matrice modelview

int localisation = glGetUniformLocation(shader.getProgramID(),
"modelview");

// Envoi de la matrice

glUniformMatrix4fv(localisation, 1, GL_FALSE, value_ptr(modelview));
```

La variable **uniform** au niveau du shader contient maintenant les valeurs de la matrice **modelview**.

Vous êtes à présent capables de comprendre les fameux envois que nous utilisons depuis le chapitre sur la 3D. 😊

Code : C++

```
// Envoi des matrices au shader

glUniformMatrix4fv(glGetUniformLocation(shader.getProgramID(),
"projection"), 1, GL_FALSE, value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shader.getProgramID(),
"modelview"), 1, GL_FALSE, value_ptr(modelview));
```

Envoyer un tableau

Côté GLSL

L'envoi de tableaux est parfois utile lorsque vous souhaitez envoyer plusieurs variables en une seule fois. Nous savons déjà comment faire en plus car ce type d'envoi s'effectue exactement de la même façon que les matrices. Les paramètres sont eux-aussi identiques, il n'y a que le nom de la fonction à utiliser qui va changer.

Au niveau du **GLSL**, il suffit simplement de déclarer un tableau comme nous le ferions en temps normal mais précédé par le mot-cléf **uniform**. Pour déclarer un tableau de 10 cases par exemple, nous ferions ainsi :

Code : C

```
// Version du GLSL
#version 150 core

// Variable uniform
uniform float monTableau[10];

// Fonction main

void main()
{}
```

Côté application

La fonction à utiliser est très similaire à celle des matrices au niveau des paramètres. Elle s'appelle : **glUniform1fv()** :

Code : C

```
void glUniform1fv(GLint location, GLsizei count, const GLfloat *value);
```

- **location** : localisation du tableau au niveau **GLSL**
- **count** : taille du tableau
- **value** : pointeur sur les données



Attention, la fonction possède la lettre **v** à la fin de son nom, ce qui la différencie de **glUniform1f()**.

Pour envoyer un tableau de 10 **float** à l'**uniform** précédent, il suffit donc d'appeler la fonction **glUniform1fv()** avec les paramètres suivants :

Code : C++

```
// Tableau de 10 float
float tableau[10] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
1.0};
```

```
// Localisation du tableau monTableau  
  
int localisation = glGetUniformLocation(shader.getProgramID(),  
"monTableau");  
  
// Envoi des valeurs  
  
glUniform1fv(localisation, 10, tableau);
```

Ce qu'il faut retenir

Les variables **uniform** sont un peu complexes au premier abord mais vous avez remarqué qu'ils se gèrent tous de la même façon. La seule difficulté vient du nombre de variantes de la fonction **glUniform***() à utiliser. Nous allons faire un petit résumé de ce qu'il faut retenir pour éviter de vous emmêler les pinceaux dans le futur.

Tout d'abord, il faut savoir que les **uniform** sont des variables envoyées de l'application jusqu'au shader. Elles doivent être déclarées avec le mot-cléf **uniform** dans le code **GLSL**.

Pour envoyer des données, il faut commencer par localiser les variables contenues dans le shader grâce à la fonction **glGetUniformLocation()**. Puis, il faut appeler l'une des variantes suivantes :

- **glUniform1f()** : pour envoyer des variables simples
- **glUniform*f()** : pour envoyer des vecteurs. L'étoile dans le nom permet de définir le type utilisé dans le shader (**vec2**, **vec3** ou **vec4**).
- **glUniformMatrix*f()** : pour envoyer des matrices. Même remarque pour l'étoile mais pour les types de matrice cette fois
- **glUniform1fv()** : pour envoyer des tableaux

Enfin, la lettre *f* dans le nom des fonctions peut être remplacée par *i* ou *ui* pour envoyer respectivement des *entiers* ou des *entiers non-signés*. Si vous laissez la lettre *f*, alors vous enverrez des *flottants*.

C'est tout ce qu'il faut retenir. 😊

Exercices

Énoncés

Comme d'habitude, nous allons faire quelques exercices pour assimiler ce que nous avons vu dans cette partie. Vous avez évidemment le droit de revenir sur le cours pour vous aider. 😊

Exercice 1 : Créez une variable de type **float** dans votre application que vous appellerez **maCouleur**. Envoyez-la ensuite au **Fragment Shader** pour qu'elle puisse remplacer la composante **rouge** de la variable d'entrée **in Color** au moment de l'affecter à la variable de sortie **out Color**.

Exercice 2 : Créez un objet de type **Vecteur** dans votre application, ses attributs représenteront une couleur quelconque. Envoyez justement ces attributs au **Fragment Shader** pour définir la couleur du pixel.

Exercice 3 : Re-déclarez les matrices **projection** et **modelview** dans votre application (vous n'avez pas besoin de les initialiser). Envoyez-les ensuite dans le **Vertex Shader** pour les multiplier dans la fonction **main()**. Le résultat doit être contenu dans une variable de type **mat4**.

Solutions

Exercice 1 :

Secret (cliquez pour afficher)

On commence par créer la variable **maCouleur** dans l'application, puis on l'envoie au shader depuis la boucle principale grâce à la fonction **glUniform1f()** :

Code : C++

```
// Variable à envoyer  
  
float maCouleur(1.0);  
  
....  
  
// Boucle principale  
  
while (!m_input.terminer())  
{  
    ....  
  
    // Localisation de la variable maCouleur  
  
    int localisationMaCouleur =  
        glGetUniformLocation(shader.getProgramID(), "maCouleur");  
  
    // Envoi de la variable  
  
    glUniform1f(localisationMaCouleur, maCouleur);  
  
    ....  
}
```

Au niveau du **Fragment Shader**, on déclare la variable **maCouleur** avec le mot-clef **uniform** :

Code : C

```
// Uniform  
  
uniform float maCouleur;
```

Puis on modifie l'affectation de la variable de sortie **out_Color** en prenant en compte l'**uniform** et les composantes (y, z) de la variable d'entrée **Color** :

Code : C

```
void main()  
{  
    // Couleur du pixel  
  
    out_Color = vec4(maCouleur, color.y, color.z, 1.0);  
}
```

Ce qui donne le code source suivant :

Code : C

```
// Version du GLSL
#version 150 core

// Entrée
in vec3 color;

// Uniform
uniform float maCouleur;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur du pixel
    out_Color = vec4(maCouleur, color.y, color.z, 1.0);
}
```

Exercice 2 :

Secret (cliquez pour afficher)

On commence par créer un objet *Vecteur* dans l'application, puis on envoie ses coordonnées grâce à la fonction `glUniform3f()` :

Code : C++

```
// Vecteur à envoyer
Vecteur monVecteur(1.0, 0.0, 1.0);

....
```

// Boucle principale

```
while (!m_input.terminer())
{
    ....
```

// Localisation de la variable monVecteur

```
int localisationMonVecteur =
glGetUniformLocation(shader.getProgramID(), "monVecteur");
```

```
// Envoi de la variable  
glUniform3f(localisationMonVecteur, monVecteur.getX(),  
monVecteur.getY(), monVecteur.getZ());  
  
....  
}
```

Au niveau du **Fragment Shader**, on déclare la variable **monVecteur** de type **vec3** avec le mot-clé **uniform** :

Code : C++

```
// Uniform  
uniform vec3 monVecteur;
```

Puis on utilise le constructeur **vec4()** en donnant en paramètre **monVecteur** ainsi qu'une valeur représentant la composante Alpha (**1.0**) :

Code : C++

```
void main()  
{  
    // Couleur du pixel  
    out_Color = vec4(monVecteur, 1.0);  
}
```

Ce qui donne le code source suivant :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrée  
in vec3 color;  
  
// Uniform  
uniform vec3 monVecteur;  
  
// Sortie  
out vec4 out_Color;  
  
// Fonction main  
void main()  
{  
    // Couleur du pixel
```

```
    out_Color = vec4(monVecteur, 1.0);  
}
```

Exercice 3 :**Secret (cliquez pour afficher)**

On commence par déclarer les matrices **projection** et **modelview** comme nous savons le faire depuis un moment déjà. Puis on les localise toutes les deux grâce à la fonction **glGetUniformLocation()** :

Code : C++

```
// Matrices  
  
mat4 projection;  
mat4 modelview;  
  
....  
  
// Boucle principale  
  
while(!m_input.terminer())  
{  
    ....  
  
    // Localisation des matrices  
  
    int localisationProjection =  
    glGetUniformLocation(shader.getProgramID(), "projection");  
    int localisationModelview =  
    glGetUniformLocation(shader.getProgramID(), "modelview");  
}
```

Ensuite, on les envoie au shader grâce à la fonction **glUniformMatrix4fv()** en n'oubliant pas de les transposer :

Code : C++

```
// Localisation des matrices  
  
int localisationProjection =  
glGetUniformLocation(shader.getProgramID(), "projection");  
int localisationModelview =  
glGetUniformLocation(shader.getProgramID(), "modelview");  
  
// Envoi des matrices  
  
glUniformMatrix4fv(localisationProjection, 1, GL_FALSE,  
value_ptr(projection));  
glUniformMatrix4fv(localisationModelview, 1, GL_FALSE,  
value_ptr(modelview));
```

Au niveau du **Vertex Shader**, on déclare deux variables **uniform** de type **mat4** :

Code : C

```
// Uniform  
  
uniform mat4 projection;  
uniform mat4 modelview;
```

Enfin, il ne reste plus qu'à les multiplier en prenant soin de créer une variable pour contenir le résultat de l'opération :

Code : C

```
void main()  
{  
    // Multiplication des deux matrices  
  
    mat4 resultat = projection * modelview;  
  
    ....  
}
```

Ce qui donne le code source suivant :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrées  
  
in vec2 in_Vertex;  
in vec3 in_Color;  
  
// Uniform  
  
uniform mat4 projection;  
uniform mat4 modelview;  
  
// Sortie  
  
out vec3 color;  
  
// Fonction main  
  
void main()  
{  
    // Multiplication des deux matrices  
  
    mat4 resultat = projection * modelview;  
  
    // Position finale du vertex  
  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);  
  
    // Envoi de la couleur au Fragment Shader
```

```
        color = in_Color;  
    }
```

Gestion de la 3D Préparation

Dans la partie précédente, nous avons vu ce qu'étaient les variables **uniform** et comment elles fonctionnaient. Nous allons maintenant pouvoir en tirer profit en envoyant nos fameuses matrices à notre shader. Pour rappel, la matrice **modelview** permet de "placer" un modèle dans un monde en 3D. La matrice de **projection** quant à elle permet de transformer, ou plutôt projeter, ce monde vers notre écran; ce dernier n'étant qu'en 2 dimensions.

Ce qu'il faut comprendre c'est qu'elles ont chacune une utilité bien particulière. Elles sont complémentaires pour afficher quelque chose mais elles sont totalement différentes dans le fond.

Pour afficher des modèles en 3D, nous aurons donc besoin de les envoyer toutes les deux au shader que l'on utilisera grâce aux **uniform**.

Le gros avantage avec la préparation du code c'est qu'on va pouvoir supprimer le gros code tout moche de la boucle principale. En effet, nous n'avons plus besoin d'afficher un carré en 2D pour faire nos tests, nous avons besoin de modèles 3D. Et nous avons justement codé deux classes qui permettent d'en afficher (**Cube** et **Caisse**), nous n'allons donc pas nous priver de leur utilisation. 

On peut donc supprimer tout le code relatif au carré pour ne garder que ceci (sans la boucle **while**) :

Code : C++

```
void SceneOpenGL::bouclePrincipale()  
{  
    // Variables  
  
    unsigned int frameRate (1000 / 50);  
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);  
  
    // Matrices  
  
    mat4 projection;  
    mat4 modelview;  
  
    projection = perspective(70.0, (double) m_largeurFenetre /  
    m_hauteurFenetre, 1.0, 100.0);  
    modelview = mat4(1.0);  
  
    // Boucle principale  
  
    while (!m_input.terminer())  
    {  
    }  
}
```

Profitons-en dès maintenant pour réutiliser notre bonne vieille caméra en déclarant un objet Camera et en la positionnant en 3D au point de coordonnées (3, 3, 3) :

Code : C++

```
// Caméra mobile  
  
Camera camera(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0), 0.5,  
0.5);
```

Pour tester l'implémentation de la 3D, nous allons utiliser un objet **Cube**. Nous verrons par la suite la façon de gérer les textures avec un objet **Caisse**. Nous lui donnerons une taille de **2.0** ainsi que le chemin vers deux nouveaux fichiers que nous appellerons **couleur3D.vert** et **couleur3D.frag** (n'oubliez pas de les placer dans le dossier **Shaders** de votre projet) :

Code : C++

```
// Objet Cube

Cube cube(2.0, "Shaders/couleur3D.vert", "Shaders/couleur3D.frag");
cube.charger();
```

Au niveau de la boucle principale, on supprime tout le code relatif à l'ancien carré pour ne garder que ceci :

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if(m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    // Gestion du déplacement de la caméra
    camera.deplacer();

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra
    camera.lookAt(modelview);

    // Rendu

    // Actualisation de la fenêtre
    SDL_GL_SwapWindow(m_fenetre);

    // Calcul du temps écoulé
    finBoucle = SDL_GetTicks();
    tempsEcoule = finBoucle - debutBoucle;
```

```
// Si nécessaire, on met en pause le programme  
if (tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);  
}
```



Remarquez les appels aux méthodes **deplacer()** et **lookAt()** de la caméra. N'utilisez pas directement celle de la matrice **modelview**.

On y ajoute ensuite l'appel à la méthode **afficher()** de l'objet cube. On lui donnera au passage les matrices **projection** et **modelview**:

Code : C++

```
// Placement de la caméra  
....  
  
// Affichage du cube  
cube.afficher(projection, modelview);  
  
// Actualisation de la fenêtre  
....
```

La préparation du code est terminée. 😊

Intégrer la troisième dimension

Côté application

Pour programmer nos premiers shaders "3D", nous avons vu à l'instant qu'il fallait envoyer les matrices **projection** et **modelview** en tant que variable **uniform**. Nous aurons besoin pour cela de la fonction **glUniformMatrix4fv()** dont nous avons enfin appris son fonctionnement dans la partie précédente.

Vérifiez donc bien (même s'il ne devrait pas y avoir de problème) qu'elles sont bien présentes dans la méthode **afficher()** du cube :

Code : C++

```
// Verrouillage du VAO  
glBindVertexArray(m_vaoID);  
  
// Envoi des matrices  
glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),  
"projection"), 1, GL_FALSE, value_ptr(projection));  
glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
```

```
"modelview"), 1, GL_FALSE, value_ptr(modelview));  
  
    // Rendu  
    glDrawArrays(GL_TRIANGLES, 0, 36);  
  
    // Déverrouillage du VAO  
    glBindVertexArray(0);
```

Le Vertex Shader

La première chose à faire au niveau du **GLSL** va être la déclaration des matrices avec le mot-clé **uniform**. D'ailleurs, vu qu'elles sont relatives aux vertices, et non aux pixels, il faut les déclarer dans le **Vertex Shader**.

On reprend donc le code que nous avons laissé au chapitre précédent auquel on va rajouter les nouveaux **uniform** :

Code : C

```
// Uniform  
  
uniform mat4 projection;  
uniform mat4 modelview;
```

Ce qui donne :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrées  
  
in vec2 in_Vertex;  
in vec3 in_Color;  
  
// Uniform  
  
uniform mat4 projection;  
uniform mat4 modelview;  
  
// Sortie  
  
out vec3 color;  
  
// Fonction main  
  
void main()  
{  
    // Position finale du vertex  
  
    gl_Position = vec4(in_Vertex, 0.0, 1.0);
```

```
// Envoi de la couleur au Fragment Shader
color = in_Color;
}
```

Avant d'aller plus loin, pensez à modifier le type de la variable d'entrée **in_Vertex**. Vu qu'elle possède maintenant **3** coordonnées, son type devient donc **vec3** :

Code : C

```
// Entrées
in vec3 in_Vertex;
```

La variable **in_Color** ne change pas, elle a toujours ses 3 composantes.

Contrairement à ce qu'on pourrait penser, la gestion de la 3D est une chose d'assez simple à réaliser. Ce qu'il y a de plus dur à comprendre, c'est l'envoi des variables **uniform**. 🍪

En fait, il suffit juste de multiplier la variable **in_Vertex** par la matrice **modelview** et **projection**. De cette façon, on peut positionner le vertex dans le monde 3D et le projeter sur l'écran.

Le seul point sensible auquel il faut faire attention concerne l'ordre des membres dans la multiplication qui doivent se présenter de la manière suivante : **projection x modelview x vertex**. Si vous inversez un seul de ces membres alors votre calcul sera totalement faux. Dans le meilleur des cas, vous afficherez un bout de votre modèle et dans le pire, vous aurez un bel écran noir.



Euh au fait, c'est normal qu'on puisse multiplier une matrice par un vecteur ?

Oui tout à fait, nous avons même vu comment faire dans le chapitre sur les matrices :

$$\begin{pmatrix} 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

La seule condition à respecter était le fait que le nombre de **colonnes** de la matrice soit égal au nombre de **coordonnées** du vecteur :

$$\begin{pmatrix} 1 & 2 & 3 \\ 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

Pour le moment, notre vertex ne possède que 3 coordonnées, la multiplication est donc impossible. Cependant grâce aux constructeurs, nous allons pouvoir régler ce problème, il suffira juste d'ajouter la valeur **1.0** à la coordonnée **W** comme nous le faisions avant.

Pour résumer, nous devons multiplier les matrices que nous avons envoyées en **uniform** par la variable d'entrée correspondant au vertex. Vu qu'elles sont toutes considérées comme des variables en **GLSL**, nous allons pouvoir faire cette opération très simplement :

Code : C

```
void main()
{
    // Position finale du vertex en 3D
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);

    // Envoi de la couleur au Fragment Shader
    color = in_Color;
}
```

Ce qui donne le code source final :

Code : C

```
// Version du GLSL
#version 150 core

// Entrées
in vec3 in_Vertex;
in vec3 in_Color;

// Uniform
uniform mat4 projection;
uniform mat4 modelview;

// Sortie
out vec3 color;

// Fonction main
void main()
{
    // Position finale du vertex en 3D
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);

    // Envoi de la couleur au Fragment Shader
    color = in_Color;
}
```

Le Fragment Shader

Le code du **Fragment Shader** va être super simple à faire car il n'y a aucun modification à faire dessus. 😊

En effet, nous n'avons pas modifié le contenu de la variable de sortie **Color** dans le **Vertex Shader** précédent. Il n'y a donc aucune modification à apporter au **Fragment Shader** puisqu'il se contente simplement de recopier cette variable dans sa sortie. Nous pouvons reprendre sans problème le code source du chapitre précédent (même s'il était prévu pour une utilisation 2D à la base) :

Code : C

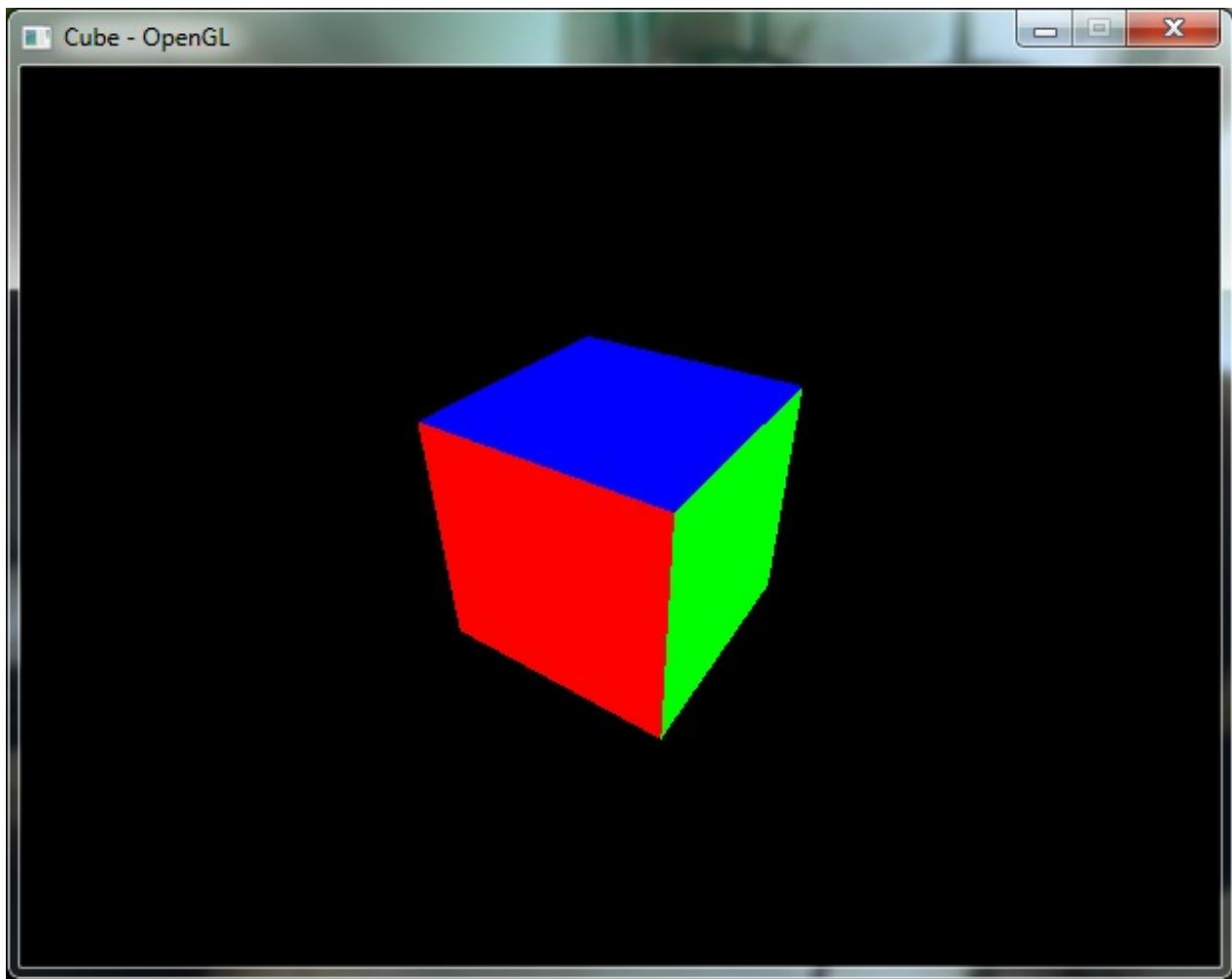
```
// Version du GLSL
#version 150 core

// Entrée
in vec3 color;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
    // Couleur finale du pixel
    out_Color = vec4(color, 1.0);
}
```

Vous pouvez compiler votre projet pour admirer le retour de la 3D dans vos programmes. 😊



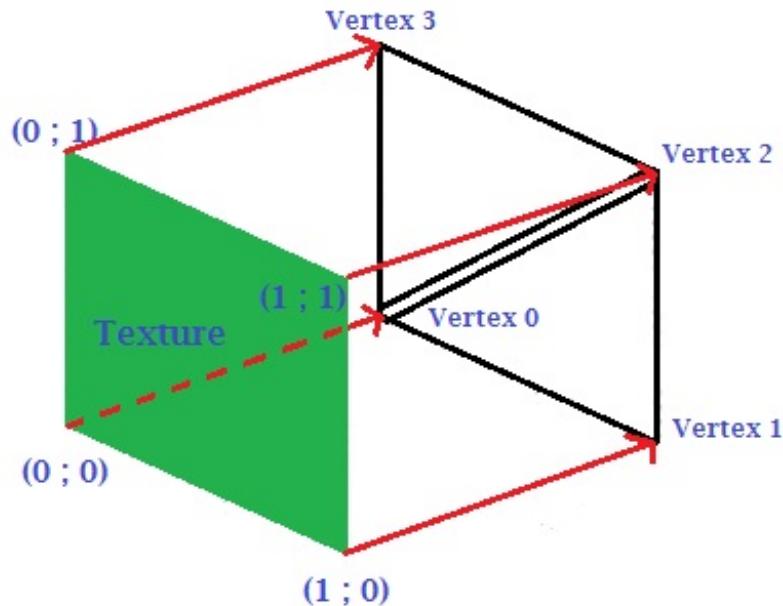
Gestion des textures

Afficher une texture

Côté application

Nous allons maintenant passer au dernier shader qui reste encore un peu mystérieux pour le moment : celui qui permet d'afficher des *textures*.

Depuis que nous les utilisons, nous n'avons plus besoin de couleur pour afficher quelque chose à l'écran. Ce qui rajoute un semblant de réalisme à nos scènes. Pour les intégrer dans l'application, nous devons utiliser un tableau (différent de celui des couleurs) qui contient des coordonnées de texture. Ces coordonnées permettent de savoir où "*plaquer*" l'image que l'on veut afficher :



Avant de commencer, veuillez créer deux nouveaux fichiers sources que vous appellerez **texture.vert** et **texture.frag**.

Pour la suite, nous allons supprimer le code relatif au cube pour le remplacer par celui de la classe **Caisse**. Nous déclarons donc un objet de ce type avec en paramètres le chemin vers le deux nouveaux fichiers créés ainsi qu'un chemin supplémentaire pour sa texture :

Code : C++

```
// Objet Caisse  
  
Caisse caisse(2.0, "Shaders/texture.vert", "Shaders/texture.frag",  
"Textures/Caisse2.jpg");  
caisse.charger();
```

Au niveau de la boucle principale, nous n'avons qu'à supprimer l'appel à la méthode **afficher()** du cube pour le remplacer par celui de notre nouvelle caisse :

Code : C++

```
// Placement de la caméra  
  
....  
  
// Affichage de la caisse  
caisse.afficher(projection, modelview);
```

```
// Actualisation de la fenêtre  
....
```

Préparation terminée. 😊

Le Vertex Shader

Pour intégrer les textures, nous n'avons pas de grande modification à faire dans le **Vertex Shader** car ce ne sera pas à lui de gérer leur affichage. En effet, ce n'est pas lui qui va prendre les pixels de l'image pour les mettre dans sa sortie, ça c'est le rôle du **Fragment Shader**. 😊 La seule chose que l'on va modifier, c'est l'envoi de données au **Fragment Shader**. On n'enverra plus de couleurs mais des coordonnées de texture.

Dans notre code, on commence par reprendre celui que l'on utilisait précédemment auquel on va supprimer la variable d'entrée **in_Color**, qui est devenue inutile ici, ainsi que la variable de sortie **color** :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrée  
  
in vec3 in_Vertex;  
  
// Uniform  
  
uniform mat4 projection;  
uniform mat4 modelview;  
  
// Fonction main  
  
void main()  
{  
    // Position finale du vertex en 3D  
  
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);  
}
```

Ensuite, on rajoute une nouvelle variable d'entrée qui va nous permettre de récupérer les coordonnées de texture. Nous l'avions appelée **in_TexCoord0** lorsque nous avons utilisé la fonction **glBindAttribLocation()** pour faire le lien avec le tableau **Vertex Attrib 2**. Nous n'avons donc qu'à déclarer cette variable accompagnée du type **vec2** (car il n'y a que 2 coordonnées) ainsi que du mot-clef **in** :

Code : C

```
// Entrées  
  
in vec3 in_Vertex;  
in vec2 in_TexCoord0;
```

On en profite également pour déclarer une variable de sortie qui copiera le contenu de **in_TexCoord0** pour l'envoyer au **Fragment Shader**. Nous l'appellerons **coordTexture** et sera elle-aussi du type **vec2**. Elle utilisera cependant le mot-clef **out** vu qu'elle sort du shader :

Code : C

```
// Sortie  
out vec2 coordTexture;
```

Enfin, pour terminer notre code source, nous devons simplement copier le contenu de la variable d'entrée dans celle de sortie. Le **Fragment Shader** se chargera du reste.

Code : C

```
void main()  
{  
    // Position finale du vertex en 3D  
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);  
  
    // Envoi des coordonnées de texture au Fragment Shader  
    coordTexture = in_TexCoord0;  
}
```

Ce qui donne le code source final :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrées  
in vec3 in_Vertex;  
in vec2 in_TexCoord0;  
  
// Uniform  
uniform mat4 projection;  
uniform mat4 modelview;  
  
// Sortie  
out vec2 coordTexture;  
  
// Fonction main  
void main()  
{  
    // Position finale du vertex en 3D  
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);
```

```
// Envoi des coordonnées de texture au Fragment Shader  
coordTexture = in_TexCoord0;  
}
```

Le Fragment Shader

Le **Fragment Shader** va subir un peu plus de modifications que son prédecesseur. Le premier sera évidemment la disparition de la variable d'entrée **color** qui n'est maintenant plus utilisée je vous le rappelle. 😊

A la place, nous mettrons la variable **coordTexture** que le **Vertex Shader** nous a gentiment envoyée à l'instant :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrée  
in vec2 coordTexture;  
  
// Sortie  
out vec4 out_Color;  
  
// Fonction main  
void main()  
{  
    // Couleur finale du pixel  
    out_Color = vec4(color, 1.0);  
}
```

Pour la suite, vous vous demandez peut-être comment intégrer une texture au sein d'un code source **GLSL** ? Vu que nous ne savons pas encore le faire, c'est une question qui peut paraître légitime.

Au risque de vous étonner, les textures sont en fait des variables **uniform** !



Euh des **uniform** ? Mais on a jamais utilisé la fonction **glUniform***() pour en envoyer ?

Alors oui certes, nous ne l'avons jamais fait. Mais nous aurions pu !

L'appel à la fonction **glUniform***() est facultatif si on n'envoie qu'une seule texture par shader. En revanche, si on en envoie plusieurs, il faut le faire pour pouvoir les différencier dans le code source **GLSL**. Nous aurons l'occasion de voir cela dans la partie sur les effets avancés. 😊

En attendant, cet appel reste facultatif. Cependant, nous devons quand même déclarer les textures en tant que variables **uniform** dans les shaders. D'ailleurs, elles possèdent un type particulier qui s'appelle **sampler2D** :

Code : C

```
// Uniform  
uniform sampler2D texture;
```

Vous pouvez donner n'importe quel nom à votre texture du moment que vous n'en envoyez qu'une. Le shader saura automatiquement laquelle utiliser.

L'objectif de cette variable en tout cas va être de récupérer la couleur du pixel recherché pour l'affecter à la variable de sortie **out_Color**. Pour le récupérer, nous allons utiliser notre première fonction prédéfinie dans le **GLSL**. Celle-ci s'appelle **texture()** :

Code : C

```
vec4 texture(sampler2D texture, vec2 textCoord);
```

- **texture** : la texture contenant les pixels. Nous lui donnerons la variable **uniform texture**
- **textCoord** : le couple de coordonnées permettant de retrouver un pixel. Nous lui donnerons notre variable d'entrée **coordTexture**

Cette fonction renvoie un **vec4** contentant la couleur du pixel que l'on recherche (composante Alpha comprise).

Nous devons donc faire appel à cette fonction pour trouver notre couleur finale. Le résultat sera affecté à la variable **out_Color** :

Code : C

```
void main()  
{  
    // Couleur du pixel  
    out_Color = texture(texture, coordTexture);  
}
```

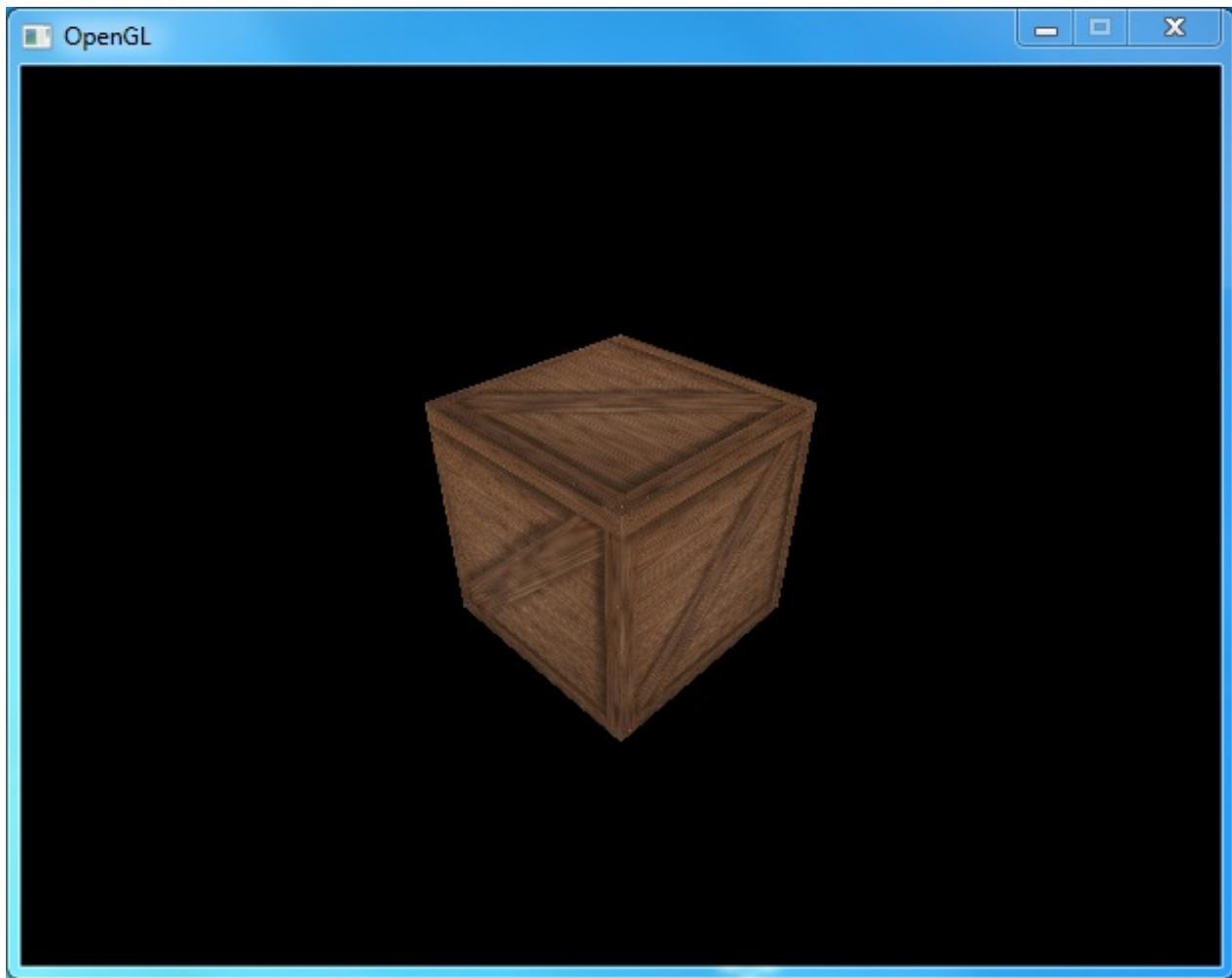
Ce qui donne le code source final :

Code : C

```
// Version du GLSL  
#version 150 core  
  
// Entrée  
in vec2 coordTexture;  
  
// Uniform  
uniform sampler2D texture;  
  
// Sortie  
out vec4 out_Color;  
  
// Fonction main
```

```
void main()
{
    // Couleur du pixel
    out_Color = texture(texture, coordTexture);
}
```

Si vous compilez votre code, vous devriez avoir une belle caisse (sans jeu de mot 😊) affichée sur votre écran.



Ce qu'il faut retenir

Point récapitulatif

Nous allons faire un dernier point récapitulatif sur ce que nous avons vu dans ces deux dernières parties. Il n'y aura pas eu beaucoup de code nouveau mais il vaut mieux synthétiser toute ça pour éviter d'éventuelles zones d'ombre.

Ainsi, pour intégrer la troisième dimension, nous devons :

- Envoyer les matrices **projection** et **modelview** au **Vertex Shader** grâce aux variables **uniform**
- Les multiplier toutes les deux par le vertex en cours dans l'ordre : **projection * modelview * vertex**. Le résultat final doit être contenu dans la variable prédéfinie **gl_Position**

Pour gérer l'affichage de texture, nous devons :

- Passer les coordonnées du tableau **Vertex Attrib 2** depuis le **Vertex Shader** jusqu'au **Fragment Shader** (avec les mots-clés **out** et **in**)
- Déclarer la texture en tant que variable **uniform sampler2D** dans le **Fragment Shader**
- Appeler la fonction **texture()** pour trouver la couleur du pixel cherché. On affecte le résultat à la variable de sortie **out_Color**

Quand faut-il "changer" de shader ?

Avant de terminer, nous allons faire un dernier point sur le *changement de shader*. Vous vous êtes déjà peut-être demandé quand est-ce que vous devez changer de shader pour afficher un modèle ? Faut-il que vous en reprogrammiez un ou pouvez-vous utiliser celui que vous avez déjà fait ?

Pour faire simple, vous devez changer de shader à chaque fois que :

- Vous touchez aux tableaux **Vertex Attribs**
- Vous voulez envoyer d'autres variables **uniform**

Nous avons vu dans ces deux derniers chapitres qu'à chaque fois que nous modifions nos tableaux **VertexAttrib** (pour les vertices, les couleurs, etc.), nous devons programmer un autre shader. Nous avions créé de nouveaux fichiers pour chaque changement.

Évidemment, si vous envoyez une variable **uniform** vous devrez créer un autre shader pour le prendre en compte.

Si vous ne souhaitez pas utiliser d'effets avancés dans vos applications alors les exemples que nous avons eus l'occasion de voir seront ceux que vous utiliserez **90%** du temps. Ce sont les shaders de base de la programmation **GLSL**. 😊

Bonnes pratiques

Optimisation du Vertex Shader

Actuellement, nos shaders souffrent d'un énorme manque d'optimisation. En effet si vous avez remarqué, nous multiplions les matrices **projection** et **modelview** dans le **Vertex Shader**. Le problème de cette multiplication c'est qu'elle s'effectue autant de fois qu'il y a de vertex à traiter. Donc si avons par exemple **5000** vertices à gérer alors elle s'effectuera également **5000 fois** !

En temps normal, je vous dirais que ça ne pose pas de problème, que la carte graphique est faite pour ça, etc. Mais là, nous pouvons quand même alléger les calculs en effectuant simplement la multiplication **avant** d'envoyer les matrices au shader. C'est une opération toute simple à faire car nous avons déjà surchargé l'**opérateur ***.

D'ailleurs, c'est de cette façon que ça se passait dans les anciennes versions d'OpenGL. 😊

Au niveau du code C++, il nous suffit juste de déclarer une nouvelle matrice que nous appellerons **modelviewProjection** (en référence à **gl_ModelviewProjectionMatrix** d'OpenGL 2.1) au moment de l'envoyer au shader qui contiendra le résultat de la multiplication de **projection** et **modelview**. Ensuite, nous n'aurons plus qu'à l'envoyer à la place d'envoyer les deux matrices séparément

Code : C++

```
// Verrouillage du VAO
glBindVertexArray(m_vao);

// Multiplication des matrices
mat4 modelviewProjection = projection * modelview;

// Envoi du résultat
```

```
    glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
"modelviewProjection"), 1, GL_FALSE,
value_ptr(modelviewProjection));  
  
    // Rendu  
  
    ....  
  
    // Déverrouillage du VAO  
    glBindVertexArray(0);
```

Au niveau du **Vertex Shader**, nous devons supprimer les deux anciens **uniform** pour les remplacer par un nouveau du nom de **modelviewProjection** :

Code : C

```
// Uniform  
  
uniform mat4 modelviewProjection;
```

Il ne manque plus qu'à prendre en compte la nouvelle matrice dans le calcul de la position du vertex :

Code : C

```
void main()  
{  
    // Position finale du vertex en 3D  
    gl_Position = modelviewProjection * vec4(in_Vertex, 1.0);  
  
    // Envoi des coordonnées de texture  
    coordTexture = in_TexCoord0;  
}
```

Notre **Vertex Shader** ne perd maintenant plus de temps à effectuer une multiplication à chaque fois qu'il doit traiter un vertex



Il n'y avait que deux lignes à changer au final. Voici à quoi ressemblerait le code permettant d'afficher une texture avec ces petits changements :

Code : C

```
// Version du GLSL  
  
#version 150 core  
  
// Entrées  

```

```
out vec2 coordTexture;

// Uniform

uniform mat4 modelviewProjection;

// Fonction main

void main()
{
    // Position finale du vertex en 3D
    gl_Position = modelviewProjection * vec4(in_Vertex, 1.0);

    // Envoi des coordonnées de texture
    coordTexture = in_TexCoord0;
}
```



Bien entendu, il peut arriver qu'on ait besoin de traiter les deux matrices séparément dans le shader. Si cela arrive, alors on enverra toutes les deux séparément au lieu, ou en plus, d'envoyer la multiplication seule.

La méthode `envoyerMat4()`

Nous terminons ce chapitre par une méthode simple qui va nous permettre d'économiser l'écriture d'une longue ligne de code répétitive. Je ne sais pas si ça vous fait le même effet mais personnellement, l'appel à la fonction `glUniformMatrix4fv()` m'est extrêmement pénible du fait de sa longueur :

Code : C

```
// Envoi de la matrice modelviewProjection

glUniformMatrix4fv(glGetUniformLocation(m_shader.getProgramID(),
    "modelviewProjection"), 1, GL_FALSE,
    value_ptr(modelviewProjection));
```

Étant donné que nous sommes obligés de faire ceci à chaque fois que l'on veut afficher un modèle, il serait judicieux de coder une méthode qui nous permettrait d'enfermer cette fonction sans avoir à la réutiliser tout le temps. Je pense que vous serez d'accord sur le principe. 🎉

Nous allons donc créer une méthode, dans la classe `Shader`, qui nous permettra de faire cette économie. Nous l'appellerons `envoyerMat4()` car elle n'enverra que des matrices carrées d'ordre 4. Elle prendra en paramètre le nom de la matrice dans le code source `GLSL` ainsi qu'une référence sur un objet de type `mat4` :

Code : C++

```
void envoyerMat4(std::string nom, glm::mat4 matrice);
```

Dans un premier temps, nous devons localiser la variable de destination grâce à la fonction `glGetUniformLocation()` :

Code : C++

```
void Shader::envoyerMat4(std::string nom, glm::mat4 matrice)
{
    // Localisation de la matrice

    int localisation = glGetUniformLocation(m_programID,
nom.c_str());
}
```

Ensuite, nous appelons la fameuse fonction **glUniformMatrix4fv()** pour envoyer les valeurs de la matrice :

Code : C++

```
void Shader::envoyerMat4(std::string nom, glm::mat4 matrice)
{
    // Localisation de la matrice

    int localisation = glGetUniformLocation(m_programID,
nom.c_str());

    // Envoi des valeurs

    glUniformMatrix4fv(localisation, 1, GL_FALSE,
value_ptr(matrice));
}
```

Méthode terminée. 😊

Elle va nous faire gagner un peu temps à chaque fois que nous devrons afficher un modèle. D'ailleurs, le code source deviendra un peu plus compréhensible également :

Code : C++

```
// Multiplication des matrices

mat4 modelviewProjection = projection * modelview;

// Envoi du résultat

m_shader.envoyerMat4("modelviewProjection", modelviewProjection);
```

Nous pouvons même inclure la multiplication directement dans l'appel à la méthode **envoyerMat4()**. Ce qui fait que l'envoi d'une matrice se résumera maintenant à une seule et unique ligne :

Code : C++

```
// Envoi d'une matrice au shader

m_shader.envoyerMat4("modelviewProjection", projection * modelview);
```

A partir de maintenant, on ne reverra plus les lignes d'envoi de matrice. 😊

L'avantage de cette méthode en plus, c'est qu'elle fonctionne aussi pour les envois séparés de **projection** et de **modelview**. En effet, elle ne fait qu'appeler la fonction **glUniformMatrix4fv()**, ce qui la rend parfaitement compatible avec les autres matrices carrées d'ordre 4.

Télécharger (Windows, UNIX/Linux, Mac OS X) : [Code Source C++ du chapitre sur les shaders \(Partie 2/2\)](#)

Nous sommes enfin arrivés à la fin de cette série de chapitres consacrés aux **shaders**. Nous avons vu pas mal de notions relatives à leur programmation de la compilation à l'aide d'une classe dédiée jusqu'à l'envoi de variable depuis l'application principale.

J'ai préféré vous occulter toute cette partie au début du tuto car la plupart d'entre vous aurait arrêté sa lecture dès que je vous aurais parlé de ça. C'était beaucoup trop indigeste à mon goût pour être balancé ni vu ni connu en même temps que la découverte d'OpenGL. Un chapitre introductif sur les shaders me semblait plus approprié.

Néanmoins, toutes les zones d'ombre sont maintenant levées et vous êtes seuls maîtres de vos applications. 😊

En ce qui concerne le **GLSL**, nous avons vu les principales notions à connaître pour faire une application, à savoir la gestion des couleurs, de la 3D et des textures. Si vous souhaitez réaliser une solution pour entreprise, vous n'avez besoin que de ça. En revanche, si vous souhaitez aller plus loin dans le développement, je serai heureux de vous retrouver dans la quatrième partie de ce tuto pour étudier les effets réalistes.

Mais avant cela, nous devons terminer la partie en cours avec encore un chapitre consacré aux **Frame Buffer Objects**. Nous en aurons besoin dans la quatrième partie alors ne faites pas l'impasse dessus. 🤪

Les Frame Buffer Objects

Nous voici arrivés au dernier chapitre de cette deuxième partie. Celui-ci concerne une fonctionnalité très utile pour les effets réalisés et constitue donc un incontournable de la programmation avancée OpenGL. Il y aura un peu de technique mais rien de vraiment insurmontable, nous verrons tout ce qu'il y a à voir pas à pas pour ne pas être perdu. 😊

Commençons d'ailleurs sans plus tarder ! 😎



Ce chapitre a eu une naissance assez folklorique, ne vous en faites pas s'il y a des petites erreurs, des images bizarres ou des fautes d'orthographe. Je préfère sortir ce long chapitre maintenant plutôt que d'attendre et lui faire prendre la poussière jusqu'à une validation totale.

Qu'est-ce qu'un FBO ?

Définitions

Les différents types de buffer

Avant de nous intéresser au sujet principal de ce chapitre, nous allons faire un petit retour en arrière sur une fonction que nous utilisons depuis pas mal de temps déjà : la fonction `glClear()` :

Code : C++

```
// Nettoyage de l'écran  
glClear(GL_COLOR_BUFFER | GL_DEPTH_BUFFER);
```

Celle-ci permet de nettoyer les *buffers* (zones mémoires ou tampons si vous préférez) qu'ils lui sont donnés en paramètres. Par exemple, nous lui demandons ici de nettoyer le **Color** ainsi que le **Depth Buffer**, le premier correspondant simplement aux pixels affichés sur notre fenêtre et le second à la position des objets 3D les uns par rapport aux autres pour pouvoir les cacher si besoin.

Si je vous parle de ça, c'est parce qu'il existe encore un autre *buffer* dont je ne vous ai pas encore parlé, ce qui est un peu normal vu que c'est celui qui est le moins utilisé. 😊 Il s'agit du **Stencil Buffer** ou en français **tampon du pochoir**. Si vous vous posez la question de savoir ce qu'est un *pochoir*, sachez que vous en avez déjà probablement utilisé un en maternelle pour dessiner une forme complexe comme un animal par exemple. Je vous laisse regarder sur internet des exemples de pochoirs, vous devriez vite comprendre à quoi ils correspondent en les voyant.

Le **Stencil Buffer** permet de faire exactement la même chose. Par exemple, au lieu d'afficher une texture carrée représentant un buisson, vous pouvez lui appliquer un pochoir (donc utiliser le **Stencil Buffer**) pour n'afficher que les pixels qui vous intéressent de façon à ne pas vous retrouver avec un buisson carré mais avec un buisson "réaliste". Je schématiserai un peu mais vous aurez compris le principe.

Le problème avec ce buffer, c'est qu'il est assez lourd à mettre en place, il faut activer un tableau **Vertex Attrib** dédié et envoyer ses données au shader. Nous ne l'utiliserons pas pour le moment. Si je vous en parle c'est parce que nous devrons gérer le cas où nous en aurions besoin plus tard, vous verrez pourquoi.

Ce qu'il faut retenir c'est qu'il existe 3 types de *buffers* :

- **Color** : qui contient la couleur de chaque pixel de votre fenêtre
- **Depth** : qui permet à OpenGL de gérer la profondeur et de cacher les objets
- **Stencil** : qui permet de filtrer un rendu avec des pochoirs

Les Frame Buffers

Cette parenthèse étant faite, nous allons pouvoir passer au sujet principal qui concerne les **Frame Buffer Objects** ou **FBO**. Vous

l'aurez deviné, nous allons encore voir des objets OpenGL. 😊

Pour donner une définition simple d'un **FBO**, on pourrait dire qu'il s'agit d'un *écran caché* qui utilise les 3 types de *buffers* que nous avons vus à l'instant :



Si j'utilise le terme *d'écran caché* c'est par les **FBO** permettent de faire ce que l'on appelle un rendu off-screen (en dehors de l'écran). C'est-à-dire qu'au lieu d'afficher votre scène dans votre fenêtre, vous le ferrez dans un endroit à part dans la carte graphique comme s'il y en avait une deuxième à l'intérieure. Le rendu sera exactement le même sauf que vous ne le verrez pas directement.



À quoi ça sert d'afficher quelque chose si on ne le voit pas ?

C'est quelque chose d'un peu tordu au premier abord mais je pense que vous allez vite aimer les **FBO** quand vous connaitrez les possibilités qu'ils offrent. En étant associés aux shaders, ils permettent de faire plein d'effets réalistes tels que :

- **Le flou** : pour la vitesse ou l'éblouissement
- **Les reflets** : pour les miroirs et l'eau
- **Les ombres dynamiques** : qui sont modifiées à chaque fois que leur "modèle" bouge
- **La capture vidéo** : pour filmer un endroit tout en étant dans un autre (caméra de surveillance, télévision, etc.)
- ...

Une bonne palette d'effets sympathiques en somme. 😊

Petit détail pour terminer : votre écran lui-même est considéré comme un **Frame Buffer**. Il reprend donc toutes les caractéristiques dont nous venons de parler, sauf pour les effets réalistes parce qu'il est impossible de modifier les pixels une fois qu'ils sont affichés sur l'écran.

Fonctionnement

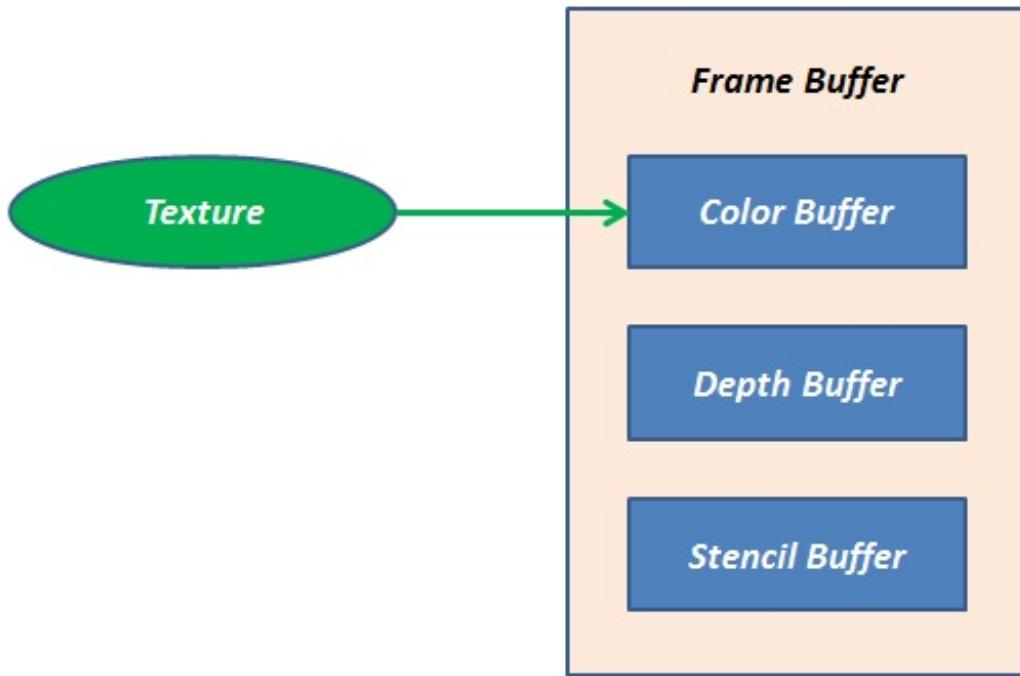
Nous allons aller un peu plus loin dans la structure des **FBO** car ils ont une façon bien particulière de fonctionner, en particulier au niveau de ses 3 *buffers* qui ne se gèrent pas tous de la même manière. On distinguera d'un côté le **Color Buffer** et de l'autre le **Depth** et le **Stencil Buffer**.

Le Color Buffer

Comme nous l'avons vu un peu plus haut, un **FBO** peut être considéré comme un *écran caché* au sein de la carte graphique, nous pouvons donc sans aucun problème faire n'importe quel rendu à l'intérieur. L'intérêt dans tout ça vient du fait que l'on peut modifier les pixels une fois qu'ils sont affichés, ce qui est impossible avec le véritable écran.

Ces fameux pixels sont contenus dans le **Color Buffer**, celui que nous nettoyons à chaque tour de boucle avec la constante **GL_COLOR_BUFFER**. Et si je peux vous apprendre quelque chose d'étonnant avec lui, c'est qu'il s'agit en fait d'une simple *texture* !

Oui, oui vous avez bien lu, le *Color Buffer* est une texture :



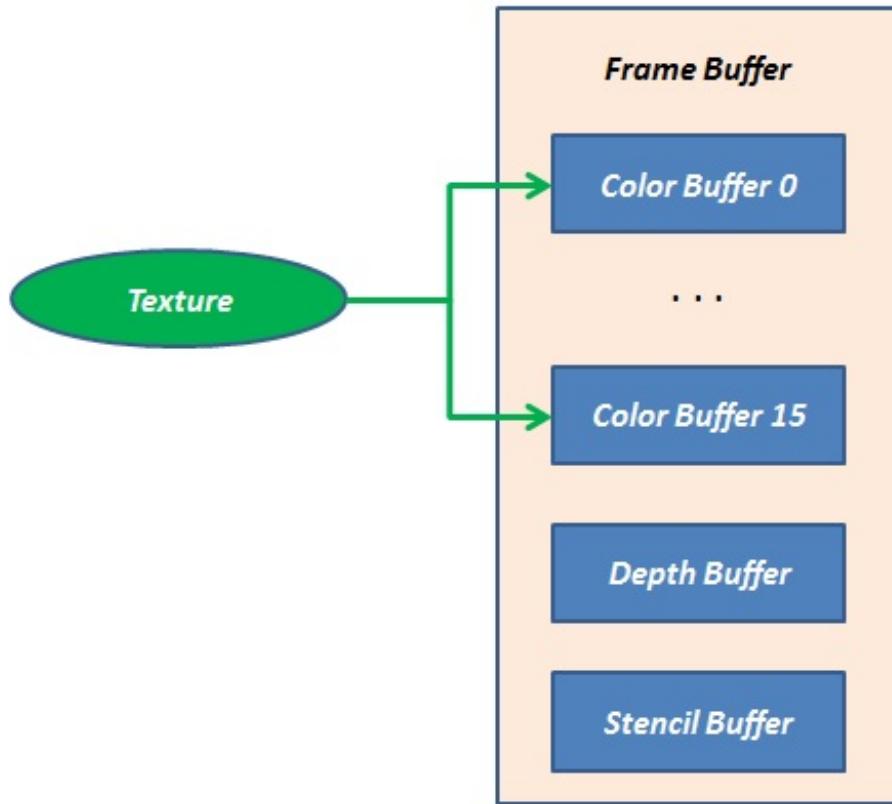
C'est un peu logique puisque le but d'OpenGL est d'afficher des pixels sur un écran, et une texture est justement faite pour en contenir plein. L'avantage avec elle, c'est que nous pourrons l'utiliser dans un shader pour la modifier très facilement. Rajouter un effet sera donc un jeu d'enfant. 😊



Ah j'ai une question : le **Color Buffer** de l'écran est lui-aussi une texture ?

Oui on peut dire ça comme ça. C'est comme si on avait une grosse texture, contenant l'affichage de notre scène, plaquée sur la fenêtre SDL.

Petite précision importante, les **FBO** peuvent contenir jusqu'à **16 Color Buffers**. C'est-à-dire que vous pouvez afficher ou modifier votre rendu **16** fois dans le **même FBO** :

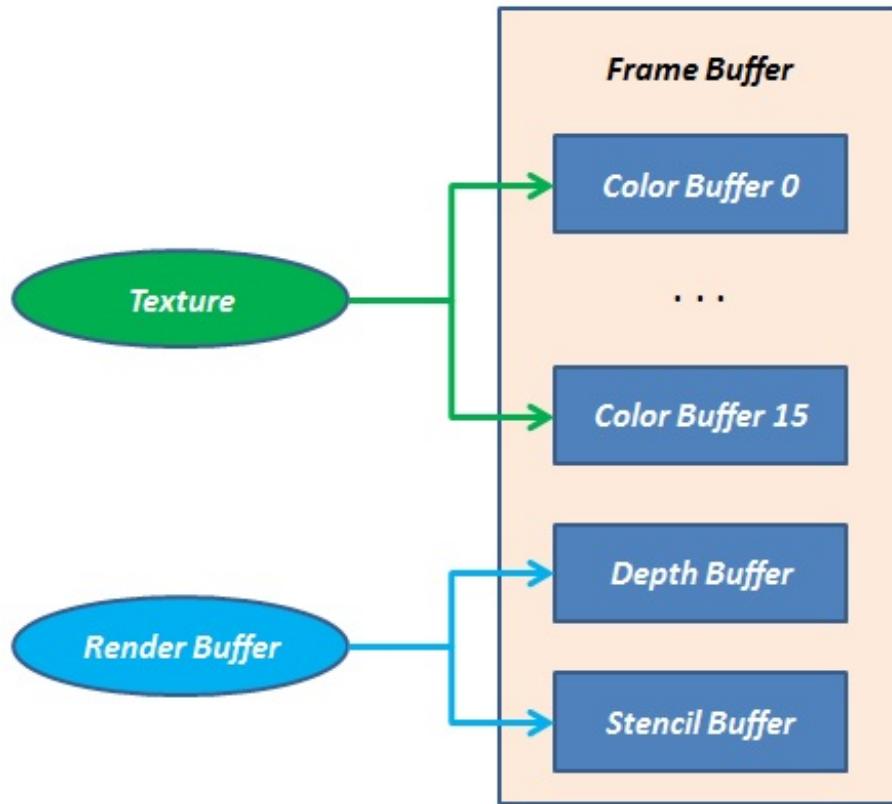


Ceci est particulièrement utile si l'on souhaite stocker des données autres que celles relatives à l'affichage. Les ombres en sont un exemple car elles utilisent plusieurs **Color Buffers** pour stocker leurs données.

Mais bon, pour le moment nous n'en utiliserons qu'un seul ne vous en faites pas, c'est déjà bien assez.

Le Depth et le Stencil Buffer

Les deux autres buffers (**Depth** et **Stencil**) se gèrent différemment du premier. Leurs données sont un peu plus complexes et ne peuvent pas être représentées par des textures. À la place, nous utiliserons ce que l'on appelle des **Render Buffers**, ce sont des espaces mémoires conçues pour accueillir ces types de données :



En résumé

Pour résumer un peu tout ce flot d'informations : un **Frame Buffer** est composé de 3 *sous-buffers* (**Color**, **Depth** et **Stencil**) :

- Le premier est représenté par une **texture** et contient l'affichage d'une scène. Il peut y en avoir jusqu'à **16**.
- Les deux autres sont représentés par des **Render Buffer**.

Ce sont les points à connaître par cœur en ce qui concerne les **FBO**.

Programme du chapitre

Après ce petit bout de théorie, nous allons pouvoir nous lancer dans la pratique pure et dure.

Je vous retiens encore un peu ici pour vous montrer le programme pour la suite. Nous allons éparpiller le code relatif aux **FBO** dans plusieurs classes, je préfère donc vous exposer clairement les différentes tâches qui nous attendent.

Celles-ci sont :

- La modification la classe **Texture** pour lui permettre de créer des **Color Buffers**
- La modification des méthodes de copie pour gérer ce nouveau type de texture
- L'implémentation des **Render Buffers**
- La création une classe **Frame Buffer** qui réunira tous les buffers

Nous avons donc pas mal de trucs à faire avec tout ceci.

Je vous invite naturellement à commencer par la première tâche qui concerne la modification de la classe **Texture**. 😊

Le Color Buffer

Les tâches

Pour reprendre très rapidement le programme précédent, notre objectif dans cette partie va être la réalisation des deux premières tâches :

- La modification la classe **Texture** pour lui permettre de créer des **Color Buffers**
- La modification des méthodes de copie pour gérer ce nouveau type de texture
- L'implémentation des **Render Buffers**
- La création une classe **Frame Buffer** qui réunira tous les buffers

Je parle bien des deux premières car nous modifierons la classe **Texture** pour toutes les deux, autant les faire dans la même partie.



Commençons par la première.

Créer un Color Buffer

Le header

La fameuse modification de la classe **Texture** concerne l'ajout d'une nouvelle méthode qui permettra de créer des *textures vides*. Je dis bien *vide* parce qu'elle n'est pas chargée depuis une image sur le disque dur, elle ne contient donc aucune donnée au moment de sa création. Ce n'est qu'à l'affichage qu'elle aura ses pixels.

Avant de créer ladite méthode, nous allons tout d'abord ajouter de nouveaux attributs à la classe **Texture** qui lui permettront de gérer les textures vides.

Dans les chapitres précédents, nous utilisions une structure de type **SDL_image** pour garder en mémoire pas mal d'informations sur une texture comme ses dimensions, le format interne des couleurs, etc. Vu que nous n'en utilisons pas ici, il faudra recréer toutes ces informations à la main à travers des attributs de classe.

Ces attributs sont les suivants :

- **int largeur** : La largeur de la texture
- **int hauteur** : Sa hauteur
- **Glenum format** : Le format des couleurs (3 ou 4 couleurs en comptant le canal Alpha)
- **Glenum formatInerne** : Le format interne (l'ordre des couleurs)
- **bool textureVide** : Ce booléen sera utilisé dans les méthodes de copie

En ajoutant le prefix **m_** à ces attributs, on a le header suivant :

Code : C++

```
#ifndef DEF_TEXTURE
#define DEF_TEXTURE

// Includes

....
```



```
// Classe Texture
```



```
class Texture
{
    public:
```



```
    // Méthodes
```

```
....  
private:  
GLuint m_id;  
std::string m_fichierImage;  
  
int m_largeur;  
int m_hauteur;  
GLenum m_format;  
GLenum m_formatInterne;  
bool m_textureVide;  
};  
  
#endif
```

Les anciens constructeurs

Les nouveaux attributs doivent évidemment être initialisés dans les différents constructeurs, nous leur donnerons tous la valeur **0**:

Code : C++

```
// Constructeur par défaut  
  
Texture::Texture() : m_id(0), m_fichierImage(""), m_largeur(0),  
m_hauteur(0), m_format(0), m_formatInterne(0), m_textureVide(false)  
{  
  
}  
  
// Autre constructeur  
  
Texture::Texture(std::string fichierImage) : m_id(0),  
m_fichierImage(fichierImage), m_largeur(0), m_hauteur(0),  
m_format(0), m_formatInterne(0), m_textureVide(false)  
{  
  
}
```

Notez que j'ai volontairement occulté le constructeur de copie, nous le verrons à la fin.

Un nouveau constructeur

Le problème avec les constructeurs précédents c'est qu'aucun d'entre eux n'est capable de donner de "vraies" valeurs aux attributs, ce qui est normal car ils n'ont aucun paramètres qui leur permettent de faire cela.

Pour combler ce manque, nous allons devoir rajouter un nouveau constructeur qui, lui, prendra en paramètres tout ce dont on a besoin pour initialiser nos attributs correctement :

Code : C++

```
Texture(int largeur, int hauteur, GLenum format, GLenum  
formatInterne, bool textureVide);
```

L'implémentation de ce constructeur est assez simple puisqu'il suffit juste de donner le bon paramètre au bon attribut :

Code : C++

```
Texture::Texture(int largeur, int hauteur, GLenum format, GLenum  
formatInterne, bool textureVide) : m_id(0), m_fichierImage(""),  
m_largeur(largeur),  
    m_hauteur(hauteur), m_format(format),  
m_formatInterne(formatInterne), m_textureVide(textureVide)  
{  
}  
}
```

Grâce à lui, nous pourrons donner de véritables valeurs à nos nouveaux attributs.

La méthode `chargerTextureVide()`

L'étape suivante consiste à coder la méthode qui nous permettra de créer des textures vides. Elle utilisera pour cela les attributs précédemment initialisés, elle s'appellera tout simplement `chargerTextureVide()` :

Code : C++

```
void chargerTextureVide();
```

Elle ne prendra aucun paramètre.

Son code sera beaucoup plus simple que celui de la méthode `charger()` car elle n'aura pas besoin de charger une image, d'inverser ses pixels et de déterminer le format des couleurs. Nous avons déjà toutes ces informations grâce à notre nouveau constructeur. 😊

On commence l'implémentation de la méthode en générant un nouvel identifiant d'objet OpenGL grâce à la fonction `glGenTextures()`. On en profitera au passage pour ajouter le code de vérification d'ID à l'aide la fonction `glIsTexture()` :

Code : C++

```
void Texture::chargerTextureVide()  
{  
    // Destruction d'une éventuelle ancienne texture  
  
    if(glIsTexture(m_id) == GL_TRUE)  
        glDeleteTextures(1, &m_id);  
  
    // Génération de l'ID  
  
    glGenTextures(1, &m_id);  
}
```

On ajoute ensuite le verrouillage a l'aide de la fonction `glBindTexture()` :

Code : C++

```
void Texture::chargerTextureVide()
{
    // Destruction d'une éventuelle ancienne texture

    if(glIsTexture(m_id) == GL_TRUE)
        glDeleteTextures(1, &m_id);

    // Génération de l'ID
    glGenTextures(1, &m_id);

    // Verrouillage
    glBindTexture(GL_TEXTURE_2D, m_id);

    // Déverrouillage
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Une fois le verrouillage enclenché, nous allons appeler la fonction **glTexImage2D()** qui permet de définir les caractéristiques d'une texture (dimensions, formats, etc.). Je vous redonne son prototype ainsi que sa floppé de paramètres :

Code : C++

```
void glTexImage2D(GLenum target, GLint level, GLint
internalFormat, GLsizei width, GLsizei height, GLint border,
GLenum format, GLenum type, const GLvoid * data);
```

- **target** : Le type de la texture auquel nous donnerons, comme toujours, la constante **GL_TEXTURE_2D**
- **level** : Paramètre que nous n'utiliserons pas, nous lui donnerons la valeur **0**
- **internalFormat** : Le format interne de la texture
- **width** : Sa largeur
- **height** : Sa hauteur
- **border** : Une bordure, nous ne l'utiliserons pas et donnerons la valeur **0**
- **format** : Le format de la texture
- **type** : Type de donnée des pixels (float, int, ...). Nous lui donnerons la constante **GL_UNSIGNED_BYTE**
- **data** : Pixels de la texture, nous n'en avons pas encore et nous donnerons encore la valeur **0**

Refaite une petite lecture lentement pour différencier correctement les paramètres. Remarquez aussi que nous utiliserons tous nos nouveaux attributs.

En appelant la fonction **glTexImage2D()** avec ces paramètres, on a :

Code : C++

```
// Définition des caractéristiques de la texture

glTexImage2D(GL_TEXTURE_2D, 0, m_formatInterne, m_largeur,
m_hauteur, 0, m_format, GL_UNSIGNED_BYTE, 0);
```

Pour terminer la méthode, il ne nous manque plus qu'à appliquer les *filtres* avec la fonction **glTexParameteriv()**. Vous vous souvenez de ce que sont les filtres ? Ils permettent à OpenGL de savoir s'il doit améliorer ou baisser la qualité de la texture en

fonction de notre distance par rapport à elle.

On utilisera les mêmes filtres que ceux de la méthode **charger()**:

Code : C++

```
// Application des filtres

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

Ces filtres permettent à OpenGL d'affiner l'affichage des textures proches et de pixeliser celui des textures éloignées, le premier étant plus gourmand en calcul que le second.

Si nous réunissons ces bouts de code, on trouve la méthode **chargerTextureVide()** suivante :

Code : C++

```
void Texture::chargerTextureVide()
{
    // Destruction d'une éventuelle ancienne texture

    if(glIsTexture(m_id) == GL_TRUE)
        glDeleteTextures(1, &m_id);

    // Génération de l'ID
    glGenTextures(1, &m_id);

    // Verrouillage
    glBindTexture(GL_TEXTURE_2D, m_id);

    // Définition des caractéristiques de la texture
    glTexImage2D(GL_TEXTURE_2D, 0, m_formatInterne, m_largeur,
    m_hauteur, 0, m_format, GL_UNSIGNED_BYTE, 0);

    // Application des filtres
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_NEAREST);

    // Déverrouillage
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Grâce à elle, nous pourrons créer tous les **Colors Buffers** que nous voudrons. 😊 Il suffira juste de faire attention aux paramètres que nous donnerons.

Le problème de la copie de texture

Étant donné que nous avons rajouté des attributs dans la classe, nous devons penser à modifier les méthodes de copie (*constructeur* et *opérateur =*) pour éviter les problèmes plus tard. C'est la deuxième tâche sur notre liste.

Les attributs

On commence par le constructeur.

En reprenant son code actuel, on remarque que celui-ci ne gère pas les nouveaux attributs :

Code : C++

```
Texture::Texture(Texture const &textureACopier)
{
    // Copie de la texture

    m_fichierImage = textureACopier.m_fichierImage;
    charger();
}
```

Nous devons donc corriger cela en utilisant l'opérateur = pour chacun d'entre eux :

Code : C++

```
Texture::Texture(Texture const &textureACopier)
{
    // Copie des attributs

    m_fichierImage = textureACopier.m_fichierImage;

    m_largeur = textureACopier.m_largeur;
    m_hauteur = textureACopier.m_hauteur;
    m_format = textureACopier.m_format;
    m_formatInterne = textureACopier.m_formatInterne;
    m_textureVide = textureACopier.m_textureVide;

    // Chargement de la copie

    charger();
}
```

Le chargement

Le chargement d'une texture copiée est ce qu'il y a de plus délicat à gérer, surtout dans notre situation car nous avons deux méthodes de chargement : l'une pour les images présentes sur le disque dur et l'autre pour les textures vides.

Le problème avec la copie ici c'est que la classe **Texture** est incapable de savoir laquelle des deux elle doit appeler. Pour le moment, elle appelle toujours la méthode `charger()` quelque soit le type de texture (vide ou image), ce qui va nous poser des problèmes quand on utilisera les **Color Buffer** car eux ont besoin de l'autre méthode.

Pour éviter cela, nous devons définir manuellement les cas de chargement pour appeler la bonne méthode. Pour nous aider, nous allons nous servir du seul attribut que nous n'avons pas encore utilisé : le booléen **m_textureVide**. Nous lui avons donné une valeur dans tous les constructeurs un peu plus haut.

Si sa valeur est égale à **true** (c'est-à-dire si la texture est vide) alors nous devons appeler la méthode `chargerTextureVide()`. Si elle est égale à **false** (image SDL), alors nous devons appeler la méthode `charger()`. Un petit bloc **if** sera nécessaire pour faire

cette vérification :

Code : C++

```
// Si la texture est vide, alors on appelle la méthode
chargerTextureVide()

if(m_textureVide)
    chargerTextureVide();

// Sinon, on appelle la méthode charger() par défaut

else
    charger();
```

Le constructeur complet :

Code : C++

```
Texture::Texture(Texture const &textureACopier)
{
    // Copie des attributs

    m_fichierImage = textureACopier.m_fichierImage;

    m_largeur = textureACopier.m_largeur;
    m_hauteur = textureACopier.m_hauteur;
    m_format = textureACopier.m_format;
    m_formatInterne = textureACopier.m_formatInterne;
    m_textureVide = textureACopier.m_textureVide;

    // Si la texture est vide, alors on appelle la méthode
    chargerTextureVide()

    if(m_textureVide)
        chargerTextureVide();

    // Sinon, on appelle la méthode charger() par défaut

    else
        charger();
}
```

Cette fois, la classe appellera la bonne méthode pour charger la texture. 😊

Problème de copie des textures non chargées

Courage, il ne manque plus qu'une condition à gérer et toute cette partie embêtante sera terminée. 😊

Celle-ci concerne les textures qui n'ont pas encore été chargées au moment de la copie, c'est-à-dire celles dont nous n'avons appelé ni la méthode **charger()**, ni la méthode **chargerTextureVide()**. Que se passerait-il à votre avis si on essayait d'en copier une dans cette situation ? Et bien la copie serait automatiquement chargée même si l'originale ne l'était pas. Cet automatisme peut causer des problèmes et engendrer des bugs dans certains cas.

Pour éviter cela encore une fois, nous devons rajouter une condition qui vérifiera si la copie doit être chargée ou pas. Nous utiliserons la fonction **glIsTexture()** qui nous permet justement de savoir ceci :

Code : C++

```
Texture::Texture(Texture const &textureACopier)
{
    // Copie des attributs

    m_fichierImage = textureACopier.m_fichierImage;
    m_largeur = textureACopier.m_largeur;
    m_hauteur = textureACopier.m_hauteur;
    m_format = textureACopier.m_format;
    m_formatInterne = textureACopier.m_formatInterne;
    m_textureVide = textureACopier.m_textureVide;

    // Si la texture est vide, alors on appelle la méthode
    chargerTextureVide()

    if(m_textureVide && glIsTexture(textureACopier.m_id) == GL_TRUE)
        chargerTextureVide();

    // Sinon, on appelle la méthode charger() par défaut

    else if(glIsTexture(textureACopier.m_id) == GL_TRUE)
        charger();
}
```

Désormais, pour déclencher un des deux chargements, il faudra que la texture originale ait un identifiant valide.

L'opérateur =

Il ne reste plus qu'une seule chose à faire : recopier le code précédent dans la méthode **operator=()**, en n'oubliant pas de rajouter le **return *this** bien sûr :

Code : C++

```
Texture& Texture::operator=(Texture const &textureACopier)
{
    // Copie des attributs

    m_fichierImage = textureACopier.m_fichierImage;
    m_largeur = textureACopier.m_largeur;
    m_hauteur = textureACopier.m_hauteur;
    m_format = textureACopier.m_format;
    m_formatInterne = textureACopier.m_formatInterne;
    m_textureVide = textureACopier.m_textureVide;

    // Si la texture est vide, alors on appelle la méthode
    chargerTextureVide()

    if(m_textureVide && glIsTexture(textureACopier.m_id) == GL_TRUE)
        chargerTextureVide();

    // Sinon, on appelle la méthode charger() par défaut

    else if(glIsTexture(textureACopier.m_id) == GL_TRUE)
        charger();
}
```

```
// Retour du pointeur *this  
return *this;  
}
```

Les modifications dans la classe **Texture** sont terminées. 😊

Les Render Buffers

Les tâches

Reprendons la petite liste de tâches que nous avons à faire pour créer un **FBO**. Dans cette partie, nous allons nous occuper des deux dernières :

- La modification la classe **Texture** pour lui permettre de créer des **Color Buffers**
- La modification des méthodes de copie pour gérer ce nouveau type de texture
- L'implémentation des **Render Buffers**
- La création une classe **Frame Buffer** qui réunira tous les buffers

Ces deux tâches sont liées entre elles car les **Render Buffers** sont créés à l'intérieur de la classe **FrameBuffer**. Nous commencerons par parler d'elle et de ses attributs avant d'attaquer les **Render Buffers**.

La classe FrameBuffer

Le header

La classe **FrameBuffer** constitue évidemment le cœur des **FBO**, elle devra gérer la création et l'utilisation de tous les *buffers*. Elle appellera par la méthode **chargerTextureVide()** que nous avons codée dans la partie précédente. 😊

Son header de base est celui-ci :

Code : C++

```
#ifndef DEF_FRAMEBUFFER  
#define DEF_FRAMEBUFFER  
  
// Include Windows  
  
#ifdef WIN32  
#include <GL/glew.h>  
  
// Include Mac  
  
#elif __APPLE__  
#define GL3_PROTOTYPES 1  
#include <OpenGL/g13.h>  
  
// Include UNIX/Linux  
  
#else  
#define GL3_PROTOTYPES 1  
#include <GL3/gl3.h>  
  
#endif
```

```
// Classe FrameBuffer

class FrameBuffer
{
    public:
    private:
};

#endif
```

Cette classe possédera pas mal d'attributs pour fonctionner correctement :

- **GLuint m_id** : identifiant OpenGL représentant le **FBO**
- **float m_largeur** : largeur du **FBO**. On peut comparer cet attribut à la largeur de la fenêtre SDL
- **float m_hauteur** : même chose pour la hauteur
- **vector m_colorBuffers** : tableau dynamique qui contiendra tous les **Colors Buffers** désirés
- **GLuint m_depthBufferID** : identifiant du **Depth Buffer**



On ne crée pas d'attribut pour le **Stencil Buffer** ?

Non nous n'en avons pas besoin, vous verrez pourquoi dans la partie suivante. 😊

Cette petite flopée d'attributs permettra de gérer tous les aspects de nos futurs **FBO**. Nous en rajouterons quelqu'uns dans la dernière partie consacrée aux améliorations.

Code : C++

```
// Attributs

GLuint m_id;
int m_largeur;
int m_hauteur;

std::vector<Texture> m_colorBuffers;
GLuint m_depthBufferID;
```

Le header comportera aussi son constructeur par défaut :

Code : C++

```
FrameBuffer();
```

Ainsi qu'un second constructeur qui prendra en paramètres la largeur et la hauteur du **FBO**. Nous passerons leur valeur aux attributs du même nom :

Code : C++

```
FrameBuffer(int largeur, int hauteur);
```

Ce sera ce constructeur que l'on utilisera principalement. 😊

Enfin, nous ajoutons aussi le destructeur qui ne change décidément pas de forme :

Code : C++

```
~FrameBuffer();
```

Si on résume tout ça :

Code : C++

```
#ifndef DEF_FRAMEBUFFER
#define DEF_FRAMEBUFFER

// Include Windows

#ifdef WIN32
#include <GL/glew.h>

// Include Mac

#elif __APPLE__
#define GL3_PROTOTYPES 1
#include <OpenGL/gl3.h>

// Include UNIX/Linux

#else
#define GL3_PROTOTYPES 1
#include <GL3/gl3.h>

#endif

// Includes communs

#include <vector>
#include "Texture.h"

// Classe

class FrameBuffer
{
public:

    FrameBuffer();
    FrameBuffer(int largeur, int hauteur);
    ~FrameBuffer();

private:

    GLuint m_id;
    int m_largeur;
    int m_hauteur;

    std::vector<Texture> m_colorBuffers;
    GLuint m_depthBufferID;
};

#endif
```

Les constructeurs et le destructeur

Passons tout de suite à l'implémentation des quelques pseudo-méthodes que nous avons déclarés, nous en seront débarrassés.



Le premier constructeur est assez simple (comme d'habitude) puisqu'il ne fait que mettre des valeurs nulles aux attributs. Nous leur donnerons tous la valeur **0** :

Code : C++

```
FrameBuffer::FrameBuffer() : m_id(0), m_largeur(0), m_hauteur(0),  
m_colorBuffers(0), m_depthBufferID(0)  
{  
}
```

Le second quant à lui sera un poil différent puisqu'il prendra **2** paramètres qui correspondent aux dimensions du **FBO**. Les autres attributs seront initialisés avec la valeur **0** :

Code : C++

```
FrameBuffer::FrameBuffer(int largeur, int hauteur) : m_id(0),  
m_largeur(largeur), m_hauteur(hauteur),  
m_colorBuffers(0), m_depthBufferID(0)  
{  
}
```

Pour ce qui est du destructeur, celui-ci ne va pas détruire grand chose pour le moment. Nous le remplirons une fois que nous aurons terminé l'implémentation de toutes les méthodes.

Code : C++

```
FrameBuffer::~FrameBuffer()  
{  
}
```

La méthode `creerRenderBuffer()`

Explications

Maintenant que nous avons un squelette de classe défini, nous allons pouvoir passer à la troisième tâche que nous attend sur la liste : la création des **Render Buffers**.

Comme je vous l'ai dit tout à l'heure, les **Render Buffers** ne seront pas codés dans une classe dédiée, ils ne sont pas aussi importants que les textures donc ils n'ont pas besoin d'être séparés du reste. Nous les utiliserons pour gérer le **Depth** et le

Stencil Buffer qui, contrairement au **Color**, ne sont pas représentables par des textures (souvenez-vous du petit schéma dans l'introduction).

Pour les créer, nous allons implémenter une méthode qui se chargera de générer et de configurer un **Render Buffer** à partir d'un identifiant OpenGL. De cette façon, nous initialiserons un buffer très simplement en utilisant seulement une ligne de code. 😊

La méthode en question s'appellera **creerRenderBuffer()** et prendra 2 paramètres :

Code : C++

```
void creerRenderBuffer(GLuint &id, GLenum formatInterne);
```

- **id** : L'identifiant qui représentera le **Render Buffer**
- **formatInterne** : Format interne du buffer. C'est un paramètre que l'on voit plus souvent avec les textures, nous verrons pourquoi nous l'utilisons ici aussi

Génération de l'identifiant

L'avantage des **Render Buffers** c'est que ce sont des objets OpenGL, ils se gèrent donc de la même façon que les **VBO**, les **VAO** et les **textures**. C'est-à-dire qu'ils ont besoin d'un **identifiant** et d'un verrouillage pour leur configuration (et à fortiori pour leur utilisation). Nous retrouverons ainsi, une fois de plus, toutes les fonctions du type **glGenXXX()**, **glBindXXX()** etc. 😊

Je vais passer rapidement sur ces deux notions que nous avons l'habitude de voir maintenant. La première est la génération d'identifiant, elle se fait ici avec la fonction **glGenRenderbuffers()** :

Code : C++

```
void glGenRenderbuffers(GLsizei number, GLuint *renderbuffers);
```

- **number** : Le nombre d'ID à initialiser. Nous lui donnerons toujours la valeur 1
- **renderbuffers** : Un tableau de type **GLuint** ou l'adresse d'une variable de même type représentant le ou les **Render Buffer(s)**

Nous appellerons cette fonction en donnant en paramètre la valeur 1 ainsi que l'adresse de la variable **id** (qui est elle-même un paramètre de la méthode **creerRenderBuffer()**) :

Code : C++

```
void FrameBuffer::creerRenderBuffer(GLuint &id, GLenum formatInterne)
{
    // Génération de l'identifiant
    glGenRenderbuffers(1, &id);
}
```

On profite de cette génération pour inclure la vérification d'un "précédent chargement" qui s'applique à tous les objets OpenGL, et donc aux **Render Buffers**, afin d'éviter d'éventuelles fuites de mémoire. Nous utiliserons pour cela la fonction **glIsRenderbuffer()** :

Code : C++

```
GLboolean glIsRenderbuffer(GLuint renderbuffer);
```

La fonction renvoie la valeur **GL_FALSE** si aucun chargement n'a été effectué sur l'objet donné ou **GL_TRUE** si c'est le cas.

Si nous tombons sur la première valeur, nous n'aurons rien à faire. En revanche, si la fonction nous renvoie **GL_TRUE** il faudra faire attention à détruire le **Render Buffer** avant de le charger à nouveau. La fonction permettant cette destruction s'appelle **glDeleteRenderbuffers()**:

Code : C++

```
void glDeleteRenderbuffers(GLsizei number, GLuint *renderbuffers);
```

Elle prend exactement les mêmes paramètres que **glGenRenderbuffers()**, à savoir :

- **number** : Le nombre d'ID à détruire
- **renderbuffers** : Un tableau de type **GLuint** ou l'adresse d'une variable de même type contenant le ou les objet(s) à détruire

Comme à l'accoutumé, nous appellerons les deux fonctions que nous venons de voir juste avant la génération d'ID :

Code : C++

```
void FrameBuffer::creerRenderBuffer(GLuint &id, GLenum formatInterne)
{
    // Destruction d'un éventuel ancien Render Buffer
    if(glIsRenderbuffer(id) == GL_TRUE)
        glDeleteRenderbuffers(1, &id);

    // Génération de l'identifiant
    glGenRenderbuffers(1, &id);
}
```

Configuration

Maintenant que nous avons un **Render Buffer** généré, il ne nous reste plus qu'à le configurer. Cette étape commence évidemment par le "verrouillage" de façon à ce qu'OpenGL sache sur quel objet il doit travailler. Nous utiliserons pour cela la fonction **glBindRenderbuffer()** :

Code : C++

```
void glBindRenderbuffer(GLenum target, GLuint renderbuffer);
```

- **target** : Le fameux paramètre **target** 🍩 qui correspond toujours au type de l'objet que l'on veut verrouiller. Dans notre cas, nous lui donnerons la constante **GL_RENDERBUFFER**
- **renderbuffer** : Identifiant représentant le **Render Buffer**

Nous appellerons cette fonction deux fois : une fois pour le verrouillage et une autre pour le déverrouillage. Le premier appel

prendra en paramètre l'**identifiant** à verrouiller tandis que le second prendra la valeur **0** :

Code : C++

```
void FrameBuffer::creerRenderBuffer(GLuint &id, GLenum formatInterne)
{
    // Destruction d'un éventuel ancien Render Buffer

    if(glIsRenderbuffer(id) == GL_TRUE)
        glDeleteRenderbuffers(1, &id);

    // Génération de l'identifiant
    glGenRenderbuffers(1, &id);

    // Verrouillage
    glBindRenderbuffer(GL_RENDERBUFFER, id);

    // Déverrouillage
    glBindRenderbuffer(GL_RENDERBUFFER, 0);
}
```

Le **Render Buffer** est à présent verrouillé et prêt à être configuré. Cette opération va d'ailleurs être beaucoup plus simple que celle des autres objets OpenGL car nous n'avons besoin ici que d'une seule et unique fonction.

La fonction en question peut d'ailleurs nous faire penser à **glTexImage2D()** car elle permet de définir le format et les dimensions du **Render Buffer**. Elle possède cependant moins de paramètres qu'elle, ce qui la rend tout de même plus agréable à utiliser. 😊

Cette fameuse fonction s'appelle **glRenderbufferStorage()** :

Code : C++

```
void glRenderbufferStorage(GLenum target, GLenum internalformat,
                           GLsizei width, GLsizei height);
```

- **target** : Type de l'objet que l'on veut verrouiller, donc **GL_RENDERBUFFER** ici
- **internalformat** : Format interne du buffer, nous allons faire un petit aparté dessus dans un instant
- **width** : largeur du buffer
- **height** : hauteur du buffer

Vous vous souvenez du paramètre **formatInterne** dont je vous parlé tout à l'heure ? Celui de la méthode **creerRenderBuffer()** ? Et bien c'est ici que nous allons l'utiliser.

Ce dernier permet de définir le "type" de donnée à stocker dans le **buffer**. Par exemple, les données relatives à la profondeur (**Depth**) ne sont pas du tout les mêmes que celles relatives aux pochoirs (**Stencil**), et pourtant ce sont tous les deux des **Render Buffers**. Pour marquer cette différence, il faut indiquer dès maintenant le type de donnée qu'accueillera le **buffer**.

Pour cela, nous allons donner le paramètre **formatInterne** à la fonction **glRenderbufferStorage()**. Nous lui donnerons également les dimensions du **FBO** à l'aide des attributs **m_largeur** et **m_hauteur** :

Code : C++

```
// Configuration du Render Buffer
```

```
glRenderbufferStorage(GL_RENDERBUFFER, formatInterne, m_largeur,  
m_hauteur);
```

Avec cet appel, nous demandons à créer un **Render Buffer** du type donné par le paramètre **formatInterne** (nous verrons ses valeurs possibles dans la partie qui suit) le tout avec les dimensions de notre **FBO** initial.

Si on ajoute ceci à notre méthode, on obtient :

Code : C++

```
void FrameBuffer::creerRenderBuffer(GLuint &id, GLenum  
formatInterne)  
{  
    // Destruction d'un éventuel ancien Render Buffer  
  
    if(glIsRenderbuffer(id) == GL_TRUE)  
        glDeleteRenderbuffers(1, &id);  
  
    // Génération de l'identifiant  
    glGenRenderbuffers(1, &id);  
  
    // Verrouillage  
    glBindRenderbuffer(GL_RENDERBUFFER, id);  
  
    // Configuration du Render Buffer  
    glRenderbufferStorage(GL_RENDERBUFFER, formatInterne,  
    m_largeur, m_hauteur);  
  
    // Déverrouillage  
    glBindRenderbuffer(GL_RENDERBUFFER, 0);  
}
```

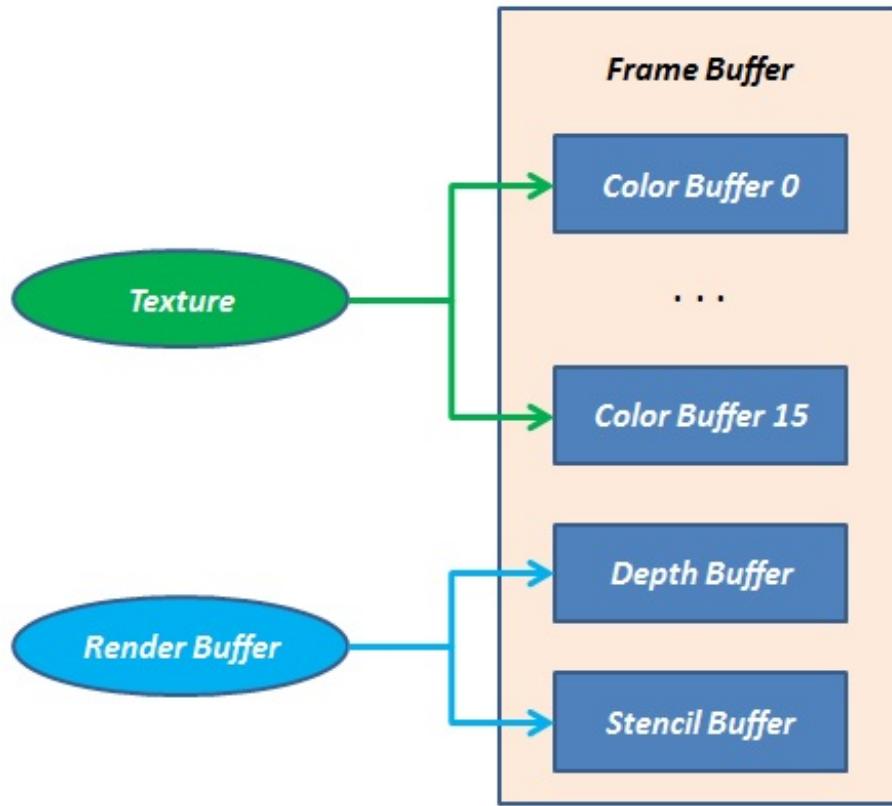
La méthode **creerRenderBuffer()** est maintenant terminée, nous pouvons l'utiliser quand bon nous semble. C'est d'ailleurs ce que nous allons faire tout de suite en utilisant tout ce que l'on vient de faire (**Color** et **Render Buffers**) dans le but de créer enfin notre premier **FBO**. 😊

Le Frame Buffer

La méthode **charger()**

Une impression de déjà-vu

Après avoir créé les deux types de buffers dans les parties précédentes, nous allons enfin pouvoir les associer au **FBO**. Si vous vous sentez un peu perdus entre tous ces buffers, vous pouvez faire une petite analogie avec les shaders : d'un côté vous avez les **Color** et **Render Buffers** (les **Vertex** et **Fragment Shaders**) et de l'autre le **Frame Buffer** qui va les utiliser (Le **Programme**). Je vous redonne le schéma du début pour vous résituer :



La comparaison avec les shaders ne s'arrête pas là d'ailleurs car nous retrouvons des notions similaires telles que l'association des buffers avec le **FBO**, le "**attachment**", ainsi que la vérification d'intégrité qui sera ici beaucoup moins longue à coder je vous rassure 😊

Nous ferons tout ça dans une nouvelle méthode qui s'appellera tout simplement **charger()** :

Code : C++

```
bool charger();
```

Elle retournera un booléen pour confirmer ou non la réussite du chargement. Pensez à inclure ce prototype dans le header de la classe **FrameBuffer**.

La génération d'identifiant

Allez, on se met dans le bain directement et on commence à coder notre nouvelle méthode dès maintenant. Le début sera très similaire à ce que nous avons fait jusqu'à présent car les **Frame Buffers** sont eux-aussi des objets OpenGL (et oui encore 😊). On retrouvera donc une fois de plus la génération d'identifiant, la vérification de double chargement et le verrouillage.

La première de ces opérations sera assurée par la fonction **glGenFramebuffers()** :

Code : C++

```
void glGenFramebuffers(GLsizei number, GLuint *ids);
```

- **number** : Le nombre d'ID à initialiser. Nous lui donnerons la valeur 1
- **ids** : Un tableau de type **GLuint** ou l'adresse d'une variable de même type représentant le ou les **FBO**

Le verrouillage quant à lui, sera effectué par la fonction **glBindFramebuffer()** :

Code : C++

```
void glBindFramebuffer(GLenum target, GLuint framebuffer);
```

- **target** : Type d'objet à verrouiller, ici nous lui donnerons toujours la valeur **GL_FRAMEBUFFER**
- **framebuffer** : Identifiant représentant le **FBO**

Nous utiliserons ces deux fonctions au début de la méthode **charger()**. L'identifiant à donner sera évidemment l'attribut **m_id** de notre classe **FrameBuffer** :

Code : C++

```
bool FrameBuffer::charger()
{
    // Génération d'un id
    glGenFramebuffers(1, &m_id);

    // Verrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, m_id);

    // Déverrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Remarquez le dernier appel qui permet de déverrouiller le **FBO**.

Comme d'habitude, on en profite au passage pour inclure le code qui permet d'éviter le double chargement. Nous utiliserons pour cela les fonctions **glIsFramebuffer()** et **glDeleteFramebuffers()** :

Code : C++

```
GLboolean glIsFramebuffer(GLuint framebuffer);
void glDeleteFramebuffers(GLsizei number, GLuint *framebuffers);
```

- La première renverra la valeur **GL_FALSE** si aucun chargement n'a été effectué ou **GL_TRUE** si c'est le cas.
- La seconde prendra les mêmes paramètres que la fonction de génération d'ID.

Nous appellerons ces fonctions avec un bloc **if** de la même façon que d'habitude. Je vous conseille cependant de mettre des accolades à votre bloc car nous rajouterons une instruction dans très peu de temps. 😊

Code : C++

```
// Vérification d'un éventuel ancien FBO
```

```
if(glIsFramebuffer(m_id) == GL_TRUE)
{
    glDeleteFramebuffers(1, &m_id);
}
```

Si on ajoute ceci au code que nous avions déjà :

Code : C++

```
bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO

    if(glIsFramebuffer(m_id) == GL_TRUE)
    {
        glDeleteFramebuffers(1, &m_id);
    }

    // Génération d'un id
    glGenFramebuffers(1, &m_id);

    // Verrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, m_id);

    // Déverrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Le Color Buffer

Comme dit un peu plus, la configuration du **FBO** va être d'une simplicité enfantine. Nous avons déjà tout codé dans les parties précédentes, nous n'avons juste qu'à appeler nos nouvelles méthodes pour créer nos différents buffers.

Le premier dont nous allons nous occuper est le **Color Buffer**. En théorie, nous devrions gérer le cas où nous en utiliserions **16** comme le permet OpenGL, cependant je ne veux pas vous alourdir encore plus les explications surtout avec tout ce que l'on a vu jusqu'à maintenant. Nous verrons donc cela dans la dernière partie de ce chapitre qui sera consacrée aux améliorations.

En attendant, nous allons gérer le cas où nous utilisons un seul et unique **Color Buffer** qui correspond, je vous le rappelle, à une texture. La première étape consistera à créer un nouvel objet de type **Texture** grâce au nouveau constructeur de cette classe.

Code : C++

```
Texture(int largeur, int hauteur, GLenum format, GLenum
formatInterne, bool textureVide);
```

Celui-ci prendra en paramètres :

- Les **dimensions du FBO** : parce qu'il faut bien que les **buffers** fassent la même taille que lui. Ces dimensions sont représentées par les attributs **m_largeur** et **m_hauteur** de notre classe
- Le **format** : qui correspond au format des couleurs (3 ou 4 couleurs avec le canal Alpha). Nous lui affecterons la constante **GL_RGBA**

- Le **format interne** des couleurs : qui correspond au interne de la texture (l'ordre des couleurs). Bizarrement, nous lui affecterons la constante **GL_RGBA**, celle-ci fonctionne pour les deux paramètres
- Le **booléen textureVide** : qui sera utile en cas de copie de texture pour savoir quelle méthode appeler (**charger()** ou **chargerTextureVide()**). Il faut lui affecter la valeur **true** évidemment

Nous créons donc un objet **Texture** en utilisant ce constructeur :

Code : C++

```
// Cr ation du Color Buffer  
  
Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA, true);
```

Ensuite, on appelle la méthode **chargerTextureVide()** de notre nouvel objet de façon à l'initialiser avec les bonnes dimensions et les bons formats :

Code : C++

```
// Cr ation du Color Buffer  
  
Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA, true);  
colorBuffer.chargerTextureVide();
```

Enfin, on l'ajoute au tableau dynamique **m_colorBuffers**. Ce dernier ne contiendra qu'une seule texture pour le moment mais n'oubliez pas que nous allons gérer les autre cas un peu plus tard :

Code : C++

```
// Cr ation du Color Buffer  
  
Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA, true);  
colorBuffer.chargerTextureVide();  
  
// Ajout au tableau  
  
m_colorBuffers.push_back(colorBuffer);
```

Petit récap :

Code : C++

```
bool FrameBuffer::charger()  
{  
    // V rification d'un  ventuel ancien FBO  
  
    if(glIsFramebuffer(m_id) == GL_TRUE)  
    {  
        glDeleteFramebuffers(1, &m_id);  
    }  
  
    // G n ration d'un id  
  
    glGenFramebuffers(1, &m_id);
```

```
// Verrouillage du Frame Buffer
glBindFramebuffer(GL_FRAMEBUFFER, m_id);

// Création du Color Buffer
Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA,
true);
colorBuffer.chargerTextureVide();

// Ajout au tableau
m_colorBuffers.push_back(colorBuffer);

// Déverrouillage du Frame Buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Le Depth et le Stencil Buffer

Bien, on passe maintenant aux **Render Buffers**.

Dans la partie précédente, nous avons codé une méthode permettant d'en créer un très facilement. Son prototype est le suivant :

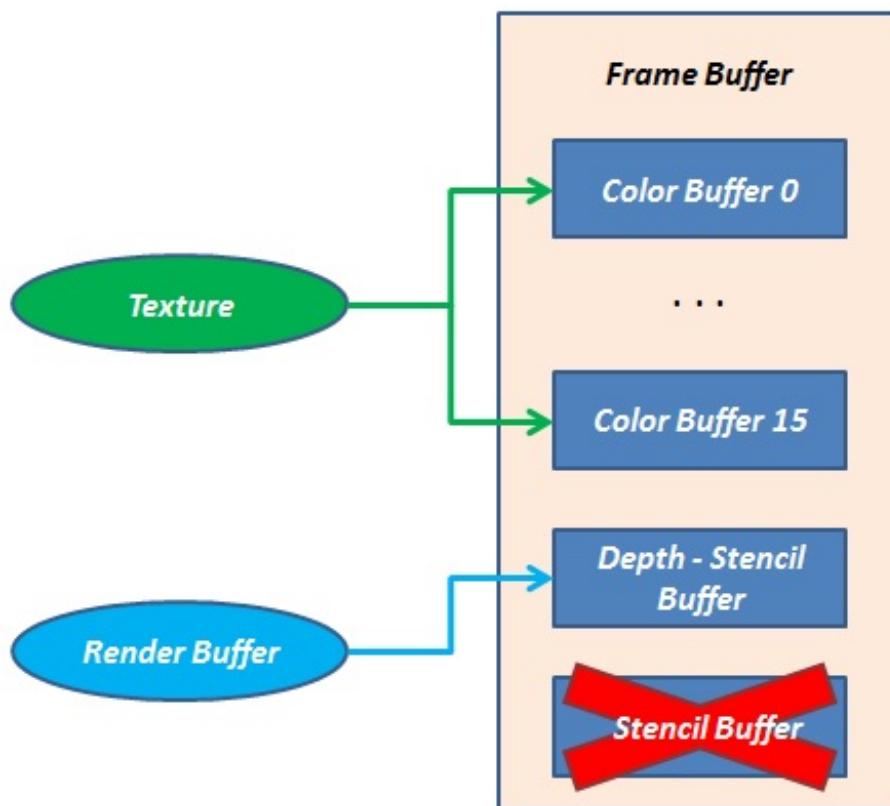
Code : C++

```
void creerRenderBuffer(GLuint &id, GLenum formatInterne);
```

- **id** : référence sur l'identifiant représentant le buffer créé
- **formatInterne** : Format interne du buffer

Fut un temps, nous aurions appelé cette méthode deux fois : une pour le **Depth Buffer** et l'autre pour le **Stencil**. Cependant, les choses ont changés et ce système ne semble plus fonctionner avec OpenGL 3. Au lieu d'avoir deux buffers séparés, nous n'en avons maintenant plus qu'un seul qui remplit le rôle du Depth et du Stencil.

Notre petit schéma n'est donc plus exact, il doit être modifié pour fusionner les deux **Render Buffers** :



Ce schéma est plus proche de la réalité. 😊

En définitif, nous n'appellerons pas la méthode **creerRenderBuffer()** deux fois mais une seule fois. Nous lui donnerons en paramètre l'attribut **m_depthBufferID** qui accueillera le "double-buffer" ainsi que la constante **GL_DEPTH24_STENCIL8** pour le format interne. Cette constante permet de :

- Définir une profondeur de **24bits** pour le **Depth Buffer**. Nous avons utilisé le même nombre au moment de configurer le contexte OpenGL avec la fonction **SDL_GL_SetAttribute()**
- Même chose pour le **Stencil Buffer** mais avec seulement **8 bits** cette fois-ci. Ce buffer prend moins de place en mémoire

L'appel final ressemble donc à ceci :

Code : C++

```
// Création du Depth et du Stencil Buffer
creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);
```



Attention, nous utilisons bien une **référence** pour le premier paramètre, ce n'est donc pas une copie qui est envoyée à la méthode.

Quelle simplicité, nous venons de créer un double-buffer en seulement une lignes de code. 🎉

Petite récap :

Code : C++

```
bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO
```

```

if(glIsFramebuffer(m_id) == GL_TRUE)
{
    glDeleteFramebuffers(1, &m_id);
}

// Génération d'un id
glGenFramebuffers(1, &m_id);

// Verrouillage du Frame Buffer
glBindFramebuffer(GL_FRAMEBUFFER, m_id);

// Création du Color Buffer
Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA,
true);
colorBuffer.chargerTextureVide();

// Ajout au tableau
m_colorBuffers.push_back(colorBuffer);

// Création du Depth et du Stencil Buffer
creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);

// Déverrouillage du Frame Buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

```

Association (Attachment)

Nos différentes buffers sont maintenant créés et prêts à être utilisés. Cependant ils ne sont pas d'une grande utilité pour le moment, ils sont tous éparpillés dans plusieurs attributs et OpenGL ne connaît même pas leur existence. Pour remédier à ce problème, nous allons les associer au **FBO**, un peu comme nous avons associé les shaders au programme.

Pour faire cela, nous aurons besoin de deux fonctions : l'une est dédiée aux **Color Buffers** et l'autre aux **Render Buffers**. Je vais vous demander un peu d'attention pour leur explication car ce sont certainement les fonctions OpenGL les plus importantes de ce chapitre.

La première s'appelle **glFramebufferTexture2D()** (un peu compliqué comme nom 😊) :

Code : C++

```

void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum
texttarget, GLuint texture, GLint level);

```

- **target** : Le paramètre target du **FBO**, soit **GL_FRAMEBUFFER** ici
- **attachment** : Paramètre SUPER important que nous allons développer dans un instant
- **texttarget** : Le paramètre target de la texture cette fois, soit **GL_TEXTURE_2D**
- **texture** : L'identifiant de la texture à associer. Nous utiliserons le getter **getID()** de notre **Color Buffer**

- **level** : Paramètre que l'on a déjà rencontré dans le chapitre sur les textures. Nous l'avions laissé à **0** et c'est ce que nous allons faire ici aussi

Le paramètre auquel il faut faire attention ici est évidemment le paramètre **attachment**. Ce dernier correspond au point d'attache, ou plus grossièrement à l'index du **Color Buffer**. N'oubliez pas que nous pouvons en créer jusqu'à **16**, il faut qu'OpenGL donne un index à chacun d'entre eux pour pouvoir les reconnaître au moment de l'affichage.

C'est précisément ce que fait le paramètre **attachment**, il permet de différencier les buffers au moment de les associer. Les valeurs qu'il peut prendre correspondent aux constantes allant de **GL_COLOR_ATTACHMENT0** à **GL_COLOR_ATTACHMENT15**. Dans notre cas, nous n'utiliserons que la première car nous n'avons qu'un seul buffer à gérer.

Au final, l'appel à la fonction **glFramebufferTexture2D()** ressemblera à ceci :

Code : C++

```
// Association du Color Buffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, m_colorBuffers[0].getID(), 0);
```

La seconde fonction d'association ressemble fortement à la première, ses paramètres sont assez semblables sauf qu'ils permettent de gérer les **Render Buffers**. La fonction s'appelle **glFramebufferRenderbuffer()** :

Code : C++

```
void glFramebufferRenderbuffer(GLenum target, GLenum attachment,
GLenum renderbuffertarget, GLuint renderbuffer);
```

- **target** : Le paramètre target du **FBO**, soit **GL_FRAMEBUFFER**
- **attachment** : Paramètre que l'on va développer dans un instant 😊
- **renderbuffertarget** : Le paramètre target du **Render Buffer** cette fois, celui que nous avons utilisé pour le configurer. Nous lui donnerons donc la constante **GL_RENDERBUFFER**
- **renderbuffer** : L'identifiant du **Render Buffer** à associer. Nous lui donnerons l'attribut **m_depthBufferID** qui contient le **Depth** et le **Stencil Buffer**

Le principe du paramètre **attachment** ne change pas vraiment par rapport au précédent on parle toujours du *point d'attache*, sauf qu'ici OpenGL n'attend plus un index mais le type de **Render Buffer** que l'on souhaite associer.

Là-aussi, nous devrions en théorie appeler cette fonction deux fois pour le **Depth** et le **Stencil Buffer**, cependant avec la fusion de ces derniers, nous n'aurons besoin que d'un seul appel.

Le paramètre **attachment** prendra une constante assez proche de celle que l'on a utilisée pour la méthode **creerRenderBuffer()** et qui s'appelle : **GL_DEPTH_STENCIL_ATTACHMENT**. L'appel à la fonction **glFramebufferRenderbuffer()** ressemblera donc à ceci :

Code : C++

```
// Association du Depth et du Stencil Buffer
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);
```

Avec cet appel, on associe le double-buffer que l'on a créé précédemment avec le point d'attache **GL_DEPTH_STENCIL_ATTACHMENT**. Le **FBO** possède maintenant tous les buffers dont il a besoin pour fonctionner. 😊

Si on récapitule tout ça :

Code : C++

```
bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO

    if(glIsFramebuffer(m_id) == GL_TRUE)
    {
        glDeleteFramebuffers(1, &m_id);
    }

    // Génération d'un id
    glGenFramebuffers(1, &m_id);

    // Verrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, m_id);

    // Création du Color Buffer
    Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA,
true);
    colorBuffer.chargerTextureVide();

    // Ajout au tableau
    m_colorBuffers.push_back(colorBuffer);

    // Création du Depth et du Stencil Buffer
    creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);

    // Association du Color Buffer
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, m_colorBuffers[0].getID(), 0);

    // Association du Depth et du Stencil Buffer
    glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);

    // Déverrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Vérification de la construction

En théorie, nous sommes maintenant capables d'utiliser notre classe **FrameBuffer** sans ajouter de code quelconque. Cependant, faire cela serait assez imprudent car nous ne savons absolument pas si notre **FBO** est valide ou non.

Pour éviter d'être surpris lors de notre prochain affichage, nous allons coder encore un petit bloc dans la méthode **charger()** qui

nous permettra de vérifier ce que l'on peut appeler *l'intégrité du FBO*. Si une erreur s'est produite au moment de sa construction (buffer, association, etc.) OpenGL nous le fera savoir, et nous devrons réagir en conséquence.



Mais euh comment on fait pour savoir si le **FBO** est mal construit ?

La réponse est très simple : il existe une fonction pour nous le dire, tout comme avec les shaders.

Heureusement pour nous, la fonction en question est moins compliquée à utiliser que celle des shaders une fois de plus. Vous vous souvenez qu'avec eux, il fallait vérifier s'il y avait une erreur, récupérer la taille du message et l'afficher. Avec les **FBO**, nous n'avons pas à faire tout ça. 😊 Mais en contrepartie, nous n'aurons pas de détails précis sur l'erreur remontée. Ce n'est pas trop grave ici car nous n'avons pas de code source à vérifier, l'erreur est donc moins susceptible de venir de nous.

Cette fonction de vérification s'appelle **glCheckFramebufferStatus()** :

Code : C++

```
GLenum glCheckFramebufferStatus(GLenum target);
```

- **target** : Le paramètre target des **FBO** ! La constante **GL_FRAMEBUFFER**. Oui c'est un peu spécial mais je vous rassure, la fonction fonctionne correctement.

Elle renvoie plusieurs constantes en cas d'erreur ou **GL_FRAMEBUFFER_COMPLETE** si tout s'est bien passé. Pour plus de simplicité, nous n'utiliserons que cette dernière. 😊

Nous commençons donc notre code en vérifiant la valeur renvoyée par la fonction **glCheckFramebufferStatus()**. Si elle est différente de **GL_FRAMEBUFFER_COMPLETE** alors on entre dans un bloc **if** :

Code : C++

```
// Vérification de l'intégrité du FBO

if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
```

```
}
```

Si le programme entre dans ce bloc, c'est qu'il s'est passé quelque chose de mauvais dans la méthode (un **Render Buffer** mal initialisé par exemple). Si cela arrive, il faudra libérer toute la mémoire prise par les différents objets.

Pour cela, nous allons appeler la fonction **glDeleteFramebuffers()** pour détruire le **FBO** et la fonction **glDeleteRenderbuffers()** pour détruire le double-buffer gérant le **Depth** et le **Stencil** :

Code : C++

```
// Vérification de l'intégrité du FBO

if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
    // Libération des buffers

    glDeleteFramebuffers(1, &m_id);
    glDeleteRenderbuffers(1, &m_depthBufferID);
}
```



Hum au fait, on ne libère pas tout là ... Il manque encore le **Color Buffer** non ?

Oui c'est exact ! Je l'ai gardé pour la fin celui-la.

En fait, ce buffer-là est un peu spécial vu qu'il s'agit d'une **texture**. Sa libération dépend donc entièrement de son destructeur. Pour l'appeler, il nous suffit simplement de supprimer la texture dans le tableau **m_colorBuffers**, le programme appellera automatiquement le destructeur concerné. C'est une des bases du C++.

Pour supprimer la texture, nous n'allons pas utiliser la méthode **pop_back()** contrairement à ce qu'on pourrait penser. A la place, nous allons prendre un peu d'avance et imaginer que nous ayons **7 Color Buffers** à détruire. Si nous étions dans ce cas, nous n'appellerions pas la même méthode 7 fois d'affilé. Ce serait une perte de temps, surtout que la classe **vector** nous fourni une jolie méthode qui permet de vider entièrement son contenu. Cette méthode s'appelle **clear()**.

Si nous l'utilisons, le tableau se videra entièrement et le programme appellera automatiquement les destructeurs de tous les objets qu'il contient.

Donc au final, pour libérer la mémoire prise par toutes les textures, nous appellerons la méthode **clear()** :

Code : C++

```
// Vérification de l'intégrité du FBO
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
    // Libération des buffers
    glDeleteFramebuffers(1, &m_id);
    glDeleteRenderbuffers(1, &m_depthBufferID);

    m_colorBuffers.clear();
}
```

Maintenant, tous nos objets sont détruits proprement et la mémoire est libérée.

Il ne reste plus qu'à afficher un message d'erreur pour conclure le tout et renvoyer la valeur **false** :

Code : C++

```
// Vérification de l'intégrité du FBO
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
    // Libération des buffers
    glDeleteFramebuffers(1, &m_id);
    glDeleteRenderbuffers(1, &m_depthBufferID);

    m_colorBuffers.clear();

    // Affichage d'un message d'erreur et retour de la valeur false
    std::cout << "Erreur : le FBO est mal construit" << std::endl;

    return false;
}
```

```
}
```

Si on récapitule toute notre méthode **charger()** :

Code : C++

```
bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO

    if (glIsFramebuffer(m_id) == GL_TRUE)
    {
        glDeleteFramebuffers(1, &m_id);
    }

    // Génération d'un id
    glGenFramebuffers(1, &m_id);

    // Verrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, m_id);

    // Création du Color Buffer
    Texture colorBuffer(m_largeur, m_hauteur, GL_RGBA, GL_RGBA,
true);
    colorBuffer.chargerTextureVide();

    // Ajout au tableau
    m_colorBuffers.push_back(colorBuffer);

    // Création du Depth et du Stencil Buffer
    creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);

    // Association du Color Buffer
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, m_colorBuffers[0].getID(), 0);

    // Association du Depth et du Stencil Buffer
    glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);

    // Vérification de l'intégrité du FBO

    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE)
    {
        // Libération des buffers

        glDeleteFramebuffers(1, &m_id);
        glDeleteRenderbuffers(1, &m_depthBufferID);

        m_colorBuffers.clear();
    }
}
```

```

        // Affichage d'un message d'erreur et retour de la
        valeur false

        std::cout << "Erreur : le FBO est mal construit" <<
        std::endl;

        return false;
    }

    // Déverrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

```

Il ne manque plus que la touche finale : le renvoi de la valeur **true** pour indiquer que tout s'est bien passé (tant que la condition précédente n'a pas été déclenchée) :

Code : C++

```

bool FrameBuffer::charger()
{
    // Création du FBO + Vérification
    . . .

    // Déverrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);

    // Si tout s'est bien passé, on renvoie la valeur true
    return true;
}

```

Cette fois, nous avons enfin terminé tout le codage des **Frame Buffer**. 😊 Nous avons créé tout ce dont ils avaient besoin pour fonctionner. Nous avons même inclus un code de vérification en cas d'erreur.

On passe maintenant aux derniers détails à régler avant d'utiliser notre premier **FBO** dans une scène 3D. 😊

Petit rajout

On commence tout de suite les petits détails par la première vérification de la méthode **charger()**, celle qui concerne le cas des double chargements. Je vous avais demandé de rajouter des accolades à votre bloc **if** car nous allions rajouter quelques lignes de code :

Code : C++

```

bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO
    if(glIsFramebuffer(m_id) == GL_TRUE)
    {
        glDeleteFramebuffers(1, &m_id);
}

```

```
}

// Génération d'un id

....
```

Les lignes de code concernées sont en fait celles qui permettent de détruire tous les objets OpenGL. Le but de cette vérification est de nettoyer un éventuel ancien chargement, il faut donc penser à détruire tous les objets qui étaient présents avant.

Cependant, nous n'ajouterons que la destruction des **Colors Buffers** car les **Render Buffers** sont détruits automatiquement au début de la méthode `creerRenderBuffer()`. Il est donc inutile de les détruire une seconde fois. 😊

Donc au final, nous n'ajoutons que l'appel à la méthode `clear()` pour l'attribut `m_colorBuffers` :

Code : C++

```
bool FrameBuffer::charger()
{
    // Vérification d'un éventuel ancien FBO

    if(glIsFramebuffer(m_id) == GL_TRUE)
    {
        // Destruction du Frame Buffer

        glDeleteFramebuffers(1, &m_id);

        // Libération des Color Buffers

        m_colorBuffers.clear();
    }

    // Génération d'un id

    ....
}
```

Le destructeur

Contrairement au début du chapitre, le destructeur va enfin pouvoir se remplir un peu pour lui permettre de détruire tous les objets OpenGL. Pour nous faciliter la vie en plus, nous allons faire les faignants et reprendre le code de destruction que nous déjà avons fait au moment de vérifier l'intégrité du **FBO**. 🍔 En effet, ce code-là détruit proprement les différents objets OpenGL utilisés, à savoir tous les **Color Buffers**, le **double Render Buffer (Depth et Stencil)** et le **FBO** en lui-même pour finir :

Code : C++

```
FrameBuffer::~FrameBuffer()
{
    // Destruction des buffers

    glDeleteFramebuffers(1, &m_id);
    glDeleteRenderbuffers(1, &m_depthBufferID);

    m_colorBuffers.clear();
}
```

Quelques getters

On termine cette partie avec quelques getters qui nous serviront à manipuler nos **FBO** sans problème. Nous en aurons exactement besoin de quatre : le premier permettra de récupérer l'identifiant du **FBO**, le deuxième permettra de récupérer ceux des différents **Color Buffer** et les deux derniers s'occuperont de renvoyer la largeur et la hauteur. Voici leur prototype :

Code : C++

```
GLuint getID() const;  
GLuint getColorBufferID(unsigned int index) const;  
  
int getLargeur() const;  
int getHauteur() const;
```



Remarquez que le second getter prend en paramètre une variable de type **unsigned int** qui permettra de récupérer un **Color Buffer** dans le tableau-attribut **m_colorBuffers**.

L'implémentation de la première méthode se passe de commentaire, il suffit de renvoyer la valeur de l'attribut **m_id**:

Code : C++

```
GLuint FrameBuffer::getID() const  
{  
    return m_id;  
}
```

Au niveau de la seconde méthode, on ne peut pas se contenter de renvoyer l'attribut **m_colorBuffers** car cela casserait la règle de l'encapsulation (il ne faut pas pouvoir accéder à un attribut en dehors de sa classe d'origine). A la place, nous allons utiliser un index pour récupérer une texture dans ce tableau et appeler ensuite sa propre méthode **getID()**. De cette façon, on peut renvoyer l'identifiant du **Color Buffer** souhaité sans avoir à accéder directement à l'attribut **m_colorBuffers** en dehors de la classe. 😊

Code : C++

```
GLuint FrameBuffer::getColorBufferID(unsigned int index) const  
{  
    return m_colorBuffers[index].getID();  
}
```

Les deux derniers getters renverront simplement les attributs **m_largeur** et **m_hauteur**:

Code : C++

```
int FrameBuffer::getLargeur() const  
{  
    return m_largeur;  
}  
  
int FrameBuffer::getHauteur() const  
{  
    return m_hauteur;
```

}

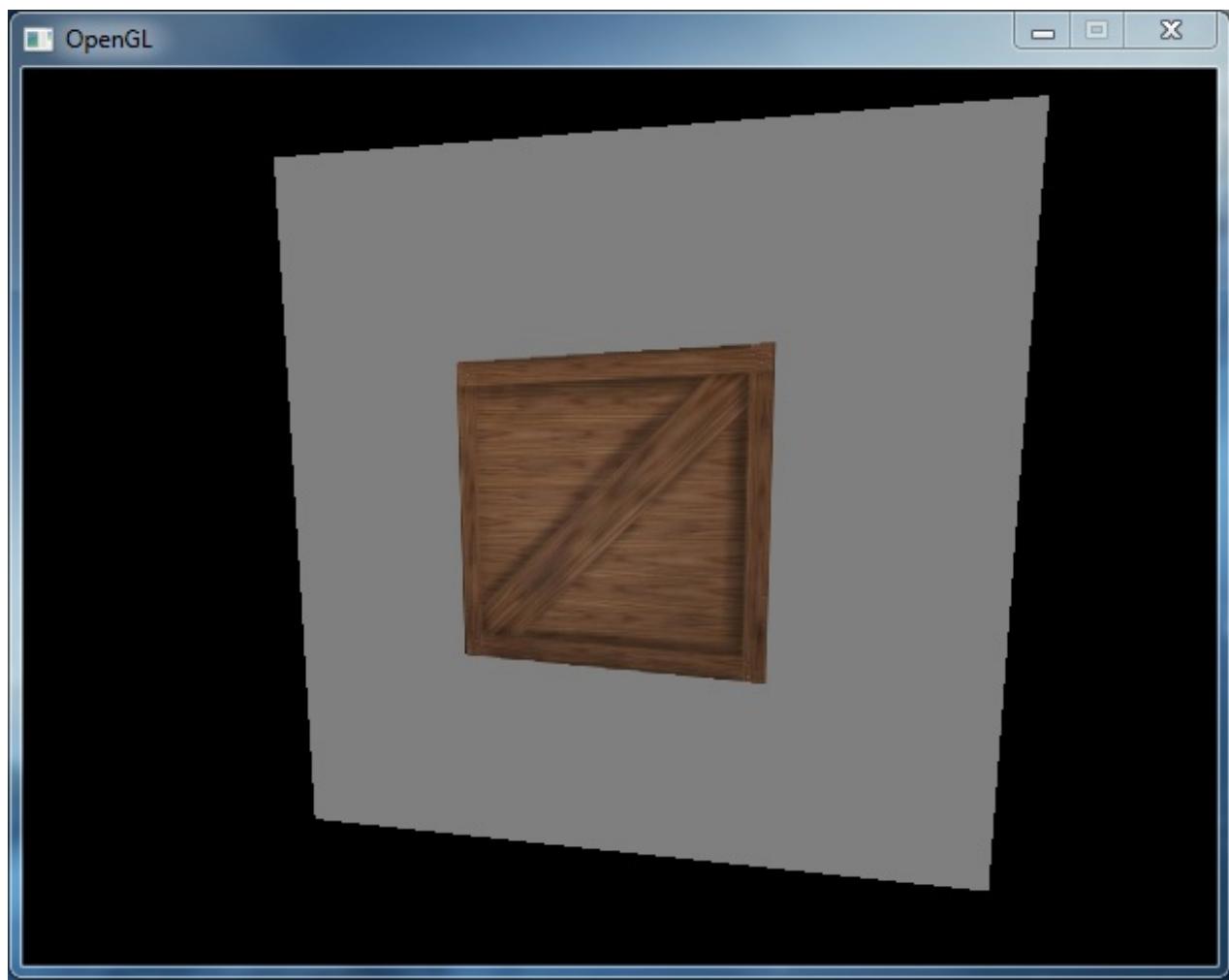
Grâce à ces getters, nous pourrons utiliser nos **FBO** facilement. 😊 Et c'est justement ce que nous allons faire maintenant.

Utilisation

Quelques explications

Après toutes les péripéties des précédentes parties, nous allons pouvoir nous reposer un peu. L'utilisation des **FBO** est une chose assez simple si on la compare à la configuration car il n'y a pas grand chose à faire.

L'objectif de cette partie va être de faire un rendu d'une caisse à l'intérieur d'un carré, que l'on peut comparer un à second écran :



Avec ceci, nous ne verrons plus la caisse comme un modèle 3D mais comme une simple image sur un écran de télévision. Nous ajouterons même une petite rotation pour prouver que le rendu se fait en temps réel.

Mais avant cela, nous allons voir ensemble quelques notions relatives à l'utilisation des **FBO**. Elles ne sont pas compliquées à comprendre mais il faut mieux les voir pour éviter d'être surpris par certains points plus tard.

Les passes

La première chose à savoir à propos de l'utilisation des **FBO** c'est qu'elle est axée autour de deux étapes, ou de deux *passes*.

La première *pass* consiste à effectuer le rendu de ce qui se trouvera dans le **FBO**. Dans notre cas, ce serait uniquement une

caisse. Si nous voulions faire une caméra de sécurité, il y aurait une salle, des personnages, des bureaux, etc. En gros tout ce qui se trouve dans le champ de vision de la caméra.

Tous ces rendus ne s'affichent pas sur l'écran évidemment mais dans le **Color Buffer** du **FBO** qui est une simple texture en deux dimensions.

La seconde **passe** quant à elle consiste à faire le rendu "*normal*" de la scène 3D, exactement comme nous l'avons toujours fait jusque là. La seule différence sera l'ajout d'un carré, ou tout autre surface, sur laquelle nous afficherons la texture du **FBO**. Pour reprendre l'exemple de la caméra, nous ajouterions une carré représentant une télévision quelque part dans notre scène sur laquelle viendrait s'afficher le rendu du **FBO**. Nous pourrions ainsi observer ce qui se trouve dans une salle tout en étant à un autre endroit.

Bien entendu, nous pouvons faire beaucoup plus de choses, notamment des ajouts d'effets sur la texture, mais vous avez au moins un aperçu de ce que sont les **passes**.

La résolution d'un FBO

La résolution des **FBO** est notion importante à prendre en compte car elle impacte directement les performances de votre application.

Il faut savoir qu'un **FBO** fera très rarement la taille de votre véritable écran. Il est tout à fait possible de procéder ainsi mais les ressources consommées seront trop importantes du fait des deux passes. C'est comme si vous travailliez avec des textures de 2048x2048 ...

Le choix de la résolution dépend de ce que vous voulez faire avec votre **FBO**. Un effet de miroir demandera plus de précision qu'un effet de flou par exemple. Il faudra jauger en fonction des situations.



Sachez que les options graphiques dans les jeux-vidéo permettent de déterminer cette précision (Qualité des ombres, des reflets, etc.)

Dans ce cas, nous utiliserons une résolution de 512x512. Cela nous rapprochera du pack de textures que nous utilisons.

Les matrices

Étant donné que nous avons deux affichages différents, nous allons devoir utiliser deux couples de matrices différents. Nous aurons ainsi une matrice **projection** et **modelview** dédiées à la première passe, et un autre couple pour le rendu normal.

Même si l'on fait deux fois le même affichage il vaut mieux séparer les matrices, en particulier la matrice **projection** car c'est elle qui gère la résolution finale. Souvenez-vous de ce que fait sa méthode **perspective()**.

Première utilisation

Après ces petites explications, nous pouvons passer à la pratique. Le but est d'afficher une caisse dans une sorte de télé, nous allons le faire en faisant attention aux quelques points que nous avons vus à l'instant.

Création du FBO

Premièrement, nous allons reprendre le code du dernier chapitre et ajouter quelques lignes de code pour créer notre **FBO**. Celui-ci sera contenu dans un objet **FrameBuffer** et sa résolution de 512x512 pixels :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables
```

```
unsigned int frameRate (1000 / 50);
Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

// Frame Buffer

FrameBuffer frameBuffer(512, 512);
frameBuffer.charger();

// Matrices

....
```

N'oubliez pas d'inclure le header **FrameBuffer.h** pour ne pas avoir d'erreur au moment de la compilation. N'oubliez pas non plus d'appeler la méthode **charger()**.

Les matrices

Maintenant que le **FBO** est créé, nous pouvons passer à la création du second couple de matrices, celles qui seront dédiées à la première passe. Ces matrices porteront quasiment le même nom que celles que nous avons l'habitude de manipuler :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Frame Buffer

    FrameBuffer frameBuffer(512, 512);
    frameBuffer.charger();

    // Matrices (première passe)

    mat4 projectionFBO, modelviewFBO;

    ....
}
```

L'initialisation de ces matrices sera elle-aussi quasiment identique. La seule chose qui va différer dans cette initialisation c'est le rapport des pixels pour la matrice de projection (le **ratio**).

De base, celui-ci est relatif à la taille de votre fenêtre, tout dépend de ce que vous avez mis avant. Cependant le rendu de la première passe s'effectue dans le **FBO**, le rapport doit donc être relatif à la taille de ce dernier. Vu que nous avons spécifié une taille de 512x512 pixels, le rapport sera donc de **512 / 512**. Nous utiliserons les getters sur la largeur et la hauteur du **FBO** pour avoir accès à ces valeurs, ce qui sera utile en cas de changement de ces dimensions :

Code : C++

```
// Matrices (première passe)

mat4 projectionFBO;
```

```
mat4 modelviewFBO;

// Initialisation

projectionFBO = perspective(70.0, (double)frameBuffer.getLargeur() /
frameBuffer.getHauteur(), 1.0, 100.0);
modelviewFBO = mat4(1.0);
```

Si vous vous posez la question, sachez que l'autre couple de matrices n'a pas besoin d'être modifié, elles sont très bien comme ça. 😊

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Frame Buffer

    FrameBuffer frameBuffer(512, 512);
    frameBuffer.charger();

    // Matrices (première passe)

    mat4 projectionFBO;
    mat4 modelviewFBO;

    projectionFBO = perspective(70.0,
(double)frameBuffer.getLargeur() / frameBuffer.getHauteur(), 1.0,
100.0);
    modelviewFBO = mat4(1.0);

    // Matrices (seconde passe)

    mat4 projection;
    mat4 modelview;

    projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
    modelview = mat4(1.0);

    // Caméra

    . . .
}
```

La première passe

Tous nos objets sont maintenant en place, il ne manque plus qu'à passer aux deux passes. Dans la première, nous allons juste afficher une caisse dans le **FBO**. Créez donc, si ce n'est déjà fait, un objet *Caisse*. Ajoutez également une variable de type **float**, elle nous permettra de le faire pivoter sur lui-même :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Frame Buffer, Matrices et Caméra
    ...

    // Objet Caisse

    Caisse caisse(2.0, "Shaders/textureMVP.vert",
    "Shaders/texture.frag", "Textures/Caisse2.jpg");
    caisse.charger();

    float angle = 0.0;

    // Boucle principale
    ...
}
```



La classe **Caisse** a été modifiée dans le chapitre précédent pour gérer l'envoi de la matrice **modelviewProjection** et non les matrices **projection** et **modelview** séparément. Si vous n'avez pas modifié les classes **Caisse** et **Cube** et bien faites-le () ou donnez le shader "texture.vert" à votre objet et non "textureMVP.vert".

Étant donné que les **FBO** se comportent comme des écrans, nous n'avons pas à modifier notre manière d'effectuer nos rendus. Il nous faut toujours nettoyer les buffers avec la fonction **glClear()**, repositionner la caméra, etc. La seule chose qui ne sera pas présente ce sera la *limitation Frame Rate* qui elle concerne l'application entière et pas juste un **FBO**.

Pour être plus précis, nous devons :

- Nettoyer les buffers
- Ré-initialiser la matrice **modelview**
- Remplacer la caméra
- Afficher la caisse, rotation comprise

Nous avons déjà vu ce code pas mal de fois, je passerai donc les explications pour celui-ci. () Faites juste attention à utiliser les matrices réservées au **FBO**.

Code : C++

```
// Boucle principale

while (!m_input.terminer())
{
    // Gestion des évènements
    ...

    /* ***** Première passe ***** */

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Placement de la caméra
```

```
modelviewFBO = lookAt(vec3(3, 0, 3), vec3(0, 0, 0), vec3(0, 1, 0));  
  
    // Gestion de la rotation de la caisse  
    angle += 2;  
    if(angle > 360)  
        angle -= 360;  
  
    mat4 sauvegardeModelviewFBO = modelviewFBO;  
    modelviewFBO = rotate(modelviewFBO, angle, vec3(0, 1, 0));  
    caisse.afficher(projectionFBO, modelviewFBO);  
    modelviewFBO = sauvegardeModelviewFBO;  
  
    // Actualisation de la fenêtre  
    ...  
}
```

Remarquez ici que l'on n'utilise pas la caméra mobile, on se place au niveau d'un point fixe. Si nous faisons cela, c'est pour éviter que le contenu du **FBO** ne change en fonction des mouvements de la caméra. Si vous faites un test avec elle, vous remarquez que votre cube n'apparaît qu'à certains endroits car votre point de vue est modifié en permanence.



Mais si on ne change rien ce code, le rendu va se faire sur l'écran non ?

Oui évidemment si on ne dit rien à OpenGL il va comprendre qu'il doit tout afficher sur l'écran.

Pour lui dire d'utiliser le **FBO**, nous allons devoir ajouter deux choses :

- Premièrement, nous allons devoir encadrer toutes les étapes précédentes par le verrouillage du **FBO** que l'on veut remplir. Il faudra donc appeler la fonction **glBindFramebuffer()** juste avant l'appel à **glClear()** et une autre fois après avoir affiché tous nos objets.
- Ensuite, nous allons redimensionner **virtuellement** la fenêtre, ou plutôt le contexte OpenGL. Nous devons faire cela car celui-ci considère toujours qu'il est dans une fenêtre dont les dimensions sont celles que nous avons spécifiées au début du programme. Une petite fonction lui permettra de redimensionner son espace d'affichage pour correspondre aux dimensions du **FBO**.

Le verrouillage va être très simple à faire car il nous suffit d'appeler la fonction **glBindFramebuffer()** au début et à la fin de l'affichage de la première passe :

Code : C++

```
// Boucle principale  
  
while(!m_input.terminer())  
{  
    // Gestion des évènements  
    ...  
}
```

```
/* ***** Première passe ***** */

// Verrouillage du Frame Buffer

glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer.getID());

// Nettoyage de l'écran

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Placement de la caméra

modelviewFBO = lookAt(vec3(3, 0, 3), vec3(0, 0, 0), vec3(0,
1, 0));

// Gestion de la rotation de la caisse

angle += 2;

if(angle > 360)
    angle -= 360;

// Affichage de la caisse

mat4 sauvegardeModelviewFBO = modelviewFBO;

modelviewFBO = rotate(modelviewFBO, angle, vec3(0, 1,
0));
caisse.afficher(projectionFBO, modelviewFBO);

modelviewFBO = sauvegardeModelviewFBO;

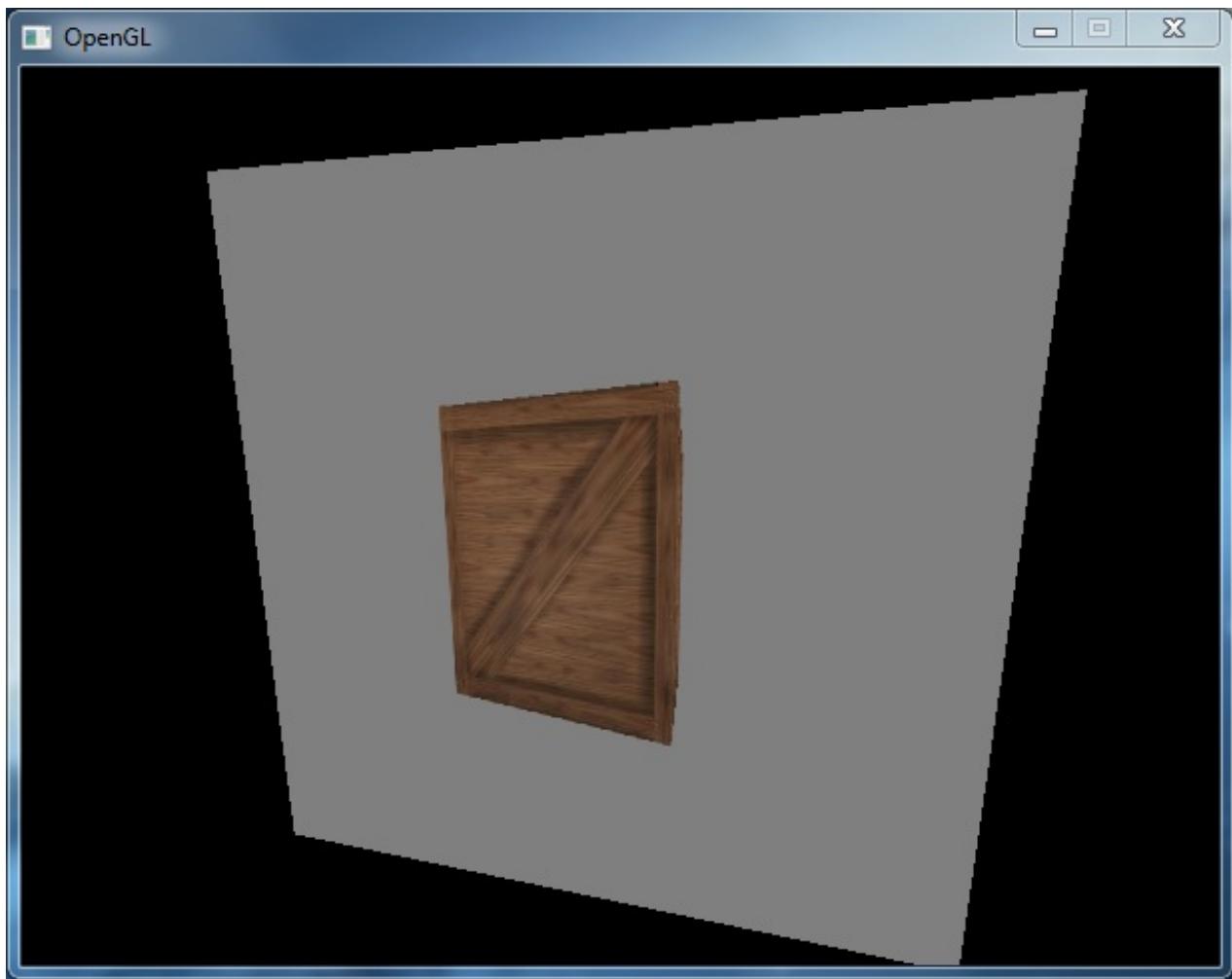
// Déverrouillage du Frame Buffer

glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Actualisation de la fenêtre

....
```

Le redimensionnement de la fenêtre est une notion qui peut vous paraître un peu bizarre, mais pour vous donner un exemple voici ce qui se passerait si nous ne faisions pas cette étape :



Dans ce cas, **OpenGL** et le **FBO** ne travaillent pas avec les mêmes dimensions, la caisse peut se retrouver alors déformée, tronquée, ou je ne sais quoi d'autre.

Pour éviter cela, nous allons dire à OpenGL de travailler temporairement avec les dimensions du **FBO** (512x512 ici). Ceci se fait grâce à la fonction **glViewport()** :

Code : C++

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

- **x** : abscisse où commence le redimensionnement. Le point de coordonnées (**0; 0**) correspond au coin inférieur gauche de votre fenêtre
- **y** : ordonnée où commence le redimensionnement
- **width** : nouvelle largeur de la zone d'affichage
- **height** : nouvelle hauteur

Cette fonction est à appeler juste après la fonction de nettoyage **glClear()** :

Code : C++

```
// Boucle principale  
  
while (!m_input.terminer())  
{  
    // Gestion des évènements
```

```
....  
  
/* ***** Première passe ***** */  
  
// Verrouillage du Frame Buffer  
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer.getID());  
  
// Nettoyage de l'écran  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  
// Redimensionnement de la zone d'affichage  
glViewport(0, 0, frameBuffer.getLargeur(),  
frameBuffer.getHauteur());  
  
// Placement de la caméra  
modelviewFBO = lookAt(vec3(3, 0, 3), vec3(0, 0, 0), vec3(0,  
1, 0));  
  
// Gestion de la rotation de la caisse  
angle += 2;  
if(angle > 360)  
    angle -= 360;  
  
// Affichage de la caisse  
mat4 sauvegardeModelviewFBO = modelviewFBO;  
modelviewFBO = rotate(modelviewFBO, angle, vec3(0, 1,  
0));  
caisse.afficher(projectionFBO, modelviewFBO);  
modelviewFBO = sauvegardeModelviewFBO;  
  
// Déverrouillage du Frame Buffer  
glBindFramebuffer(GL_FRAMEBUFFER, 0);  
  
// Actualisation de la fenêtre  
....  
}
```



Vu que nous avons redimensionné l'espace de travail d'OpenGL ici, nous allons devoir refaire la même chose pour le rendu normal.

Si tout s'est bien passé jusque là, vous devriez avoir le rendu de votre caisse à l'intérieur de votre **FBO**, et plus précisément à l'intérieur de la *texture* qu'est le **Color Buffer**.

Si vous pensez qu'il y a trop de choses à apprendre, ne vous inquiétez pas nous allons faire un point récapitulatif à la fin de cette

partie. 😊

La seconde passe

La seconde passe est la dernière étape d'implémentation d'un **FBO**, elle consiste simplement à afficher le contenu de son **Color Buffer** sur une surface. Si vous avez une scène 3D, c'est le moment de l'afficher également.

Il n'y a pas de nouvelles notions à apprendre ici, il faut effectuer le rendu comme d'habitude. Le seul petit ajout sera le redimensionnement de la zone d'affichage qui est réduite à 512x512 pour le moment. Il faudra remettre les dimensions de la fenêtre SDL.

Pour commencer, nous allons afficher un carré qui accueillera la texture du **FBO**. Ajoutez donc le code suivant après déclaré la caisse. Je passe un peu son explication, le chapitre est assez dense comme ça, il permet juste d'afficher un carré avec une gestion des **VBO/VAO**. Les coordonnées de textures sont également présentes, elles permettront de faire le lien avec la texture du **FBO**.

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Frame Buffer, Matrices et Caméra
    ...
    // Objet Caisse
    ...
    // Vertices
    float vertices[] = {0, 0, 0,     4, 0, 0,     4, 4, 0,      //
Triangle 1
                        0, 0, 0,     0, 4, 0,     4, 4, 0};      // Triangle 2
    float coordTexture[] = {0, 0,     1, 0,     1, 1,      //
Triangle 1
                           0, 0,     0, 1,     1, 1};      // Triangle 2

    /* ***** Gestion du VBO **** */
    GLuint vbo;
    int tailleVerticesBytes = 18 * sizeof(float);
    int tailleCoordTextureBytes = 12 * sizeof(float);

    // Génération du VBO
    glGenBuffers(1, &vbo);

    // Verrouillage
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    // Remplissage
    glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes +
tailleCoordTextureBytes, 0, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes, vertices);
    glBufferSubData(GL_ARRAY_BUFFER, tailleVerticesBytes,
tailleCoordTextureBytes, coordTexture);
```

```
// Déverrouillage
glBindBuffer(GL_ARRAY_BUFFER, 0);

/* ***** Gestion du VAO **** */
GLuint vao;

// Génération du VAO
glGenVertexArrays(1, &vao);

// Verrouillage du VAO
glBindVertexArray(vao);

// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(tailleVerticesBytes));
 glEnableVertexAttribArray(2);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Déverrouillage du VAO
glBindVertexArray(0);

// Shader
Shader shaderTexture("Shaders/textureMVP.vert",
"Shaders/texture.frag");
shaderTexture.charger();

// Boucle principale
while (!m_input.terminer())
{
    ....
}
```

Dans le prochain chapitre, nous créerons une classe dédiée pour les carrés. Cela évitera d'avoir à inclure des gros bouts de code comme celui-là. 😊

Une fois le carré déclaré, il me manque plus qu'à l'afficher. Pour cela nous allons répéter les mêmes opérations d'affichage que l'on

a l'habitude de faire, à savoir:

- Activer le shader
- Verrouiller le VAO
- Envoyer les matrices
- Verrouiller la texture
- Afficher le tout

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    // Gestion des évènements

    ....

    /* ***** Première passe ***** */

    ....

    /* ***** Seconde passe ***** */

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Gestion de la caméra
    camera.lookAt(modelview);

    // Activation du shader
    glUseProgram(shaderTexture.getProgramID());

    // Verrouillage du VAO
    glBindVertexArray(vao);

    // Envoi des matrices
    shaderTexture.envoyerMat4("modelviewProjection",
    projection * modelview);

    // Verrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, frameBuffer.getColorBufferID(0));

    // Rendu
    glDrawArrays(GL_TRIANGLES, 0, 6);

    // Déverrouillage de la texture
```

```
    glBindTexture(GL_TEXTURE_2D, 0);

    // Verrouillage du VAO
    glBindVertexArray(0);

    // Désactivation du shader
    glUseProgram(0);

    // Gestion du Frame Rate
    ....
}
```

N'oublions pas la touche finale qui consiste à redimensionner de la zone d'affichage. Nous utiliserons les attributs **m_largeurFenetre** et **m_hauteurFenetre** avec la fonction **glViewport()** pour régler ce petit détail :

Code : C++

```
/* ***** Seconde passe ***** */

// Nettoyage de l'écran
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Redimensionnement de la zone d'affichage
glViewport(0, 0, m_largeurFenetre, m_hauteurFenetre);

// Gestion de la caméra
camera.lookAt(modelview);
```

Récapitulatif de la méthode bouclePrincipale()

La méthode **bouclePrincipale()** accueille pas mal de code en ce moment, nous allons tout récapituler pour être sûr qu'il ne vous manque pas un bout quelque part.

Voici les objets à déclarer au début de votre méthode (attention au **shader de la caisse** je le répète 😊) :

Code : C++

```
void SceneOpenGL::bouclePrincipale()
{
    // Variables

    unsigned int frameRate (1000 / 50);
    Uint32 debutBoucle(0), finBoucle(0), tempsEcoule(0);

    // Frame Buffer
    FrameBuffer frameBuffer(512, 512);
```

```
frameBuffer.charger();

// Matrices (première passe)

mat4 projectionFBO;
mat4 modelviewFBO;

projectionFBO = perspective(70.0,
(double)frameBuffer.getLargeur() / frameBuffer.getHauteur(), 1.0,
100.0);
modelviewFBO = mat4(1.0);

// Matrices (seconde passe)

mat4 projection;
mat4 modelview;

projection = perspective(70.0, (double) m_largeurFenetre /
m_hauteurFenetre, 1.0, 100.0);
modelview = mat4(1.0);

// Caméra mobile

Camera camera(vec3(3, 3, 3), vec3(0, 0, 0), vec3(0, 1, 0), 0.5,
0.5);
m_input.afficherPointeur(false);
m_input.capturerPointeur(true);

// Objet Caisse

Caisse caisse(2.0, "Shaders/textureMVP.vert",
"Shaders/texture.frag", "Textures/Caisse2.jpg");
caisse.charger();

float angle = 0.0;

// Vertices

float vertices[] = {0, 0, 0, 4, 0, 0, 4, 4, 0, // Triangle 1
0, 0, 0, 0, 4, 0, 4, 4, 0}; // Triangle 2
float coordTexture[] = {0, 0, 1, 0, 1, 1, // Triangle 1
0, 0, 0, 1, 1, 1}; // Triangle 2

/* ***** Gestion du VBO ***** */

GLuint vbo;
int tailleVerticesBytes = 18 * sizeof(float);
int tailleCoordTextureBytes = 12 * sizeof(float);

// Génération du VBO

glGenBuffers(1, &vbo);

// Verrouillage

glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Remplissage
```

```
glBufferData(GL_ARRAY_BUFFER, tailleVerticesBytes +
tailleCoordTextureBytes, 0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, tailleVerticesBytes, vertices);
glBufferSubData(GL_ARRAY_BUFFER, tailleVerticesBytes,
tailleCoordTextureBytes, coordTexture);

// Déverrouillage
glBindBuffer(GL_ARRAY_BUFFER, 0);

/* ***** Gestion du VAO ***** */

GLuint vao;

// Génération du VAO
glGenVertexArrays(1, &vao);

// Verrouillage du VAO
glBindVertexArray(vao);

// Verrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

// Vertex Attrib 0 (Vertices)
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(tailleVerticesBytes));
 glEnableVertexAttribArray(2);

// Déverrouillage du VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Déverrouillage du VAO
glBindVertexArray(0);

// Shader
Shader shaderTexture("Shaders/textureMVP.vert",
"Shaders/texture.frag");
shaderTexture.charger();

// Boucle principale
while (!m_input.terminer())
{
    ...
}
```

Et voici ce que contient la boucle principale :

Code : C++

```
// Boucle principale

while(!m_input.terminer())
{
    // On définit le temps de début de boucle
    debutBoucle = SDL_GetTicks();

    // Gestion des évènements
    m_input.updateEvenements();

    if(m_input.getTouche(SDL_SCANCODE_ESCAPE))
        break;

    camera.deplacer(m_input);

    /* ***** Première Passe ***** */

    // Verrouillage du Frame Buffer
    glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer.getID());

    // Nettoyage de l'écran
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Redimensionnement de la zone d'affichage
    glViewport(0, 0, frameBuffer.getLargeur(),
frameBuffer.getHauteur());

    // Placement de la caméra
    modelviewFBO = lookAt(vec3(3, 0, 3), vec3(0, 0, 0), vec3(0,
1, 0));

    // Gestion de la rotation de la caisse
    angle += 2;
    if(angle > 360)
        angle -= 360;

    // Affichage de la caisse
    mat4 sauvegardeModelviewFBO = modelviewFBO;
    modelviewFBO = rotate(modelviewFBO, angle, vec3(0, 1,
0));
    caisse.afficher(projectionFBO, modelviewFBO);
    modelviewFBO = sauvegardeModelviewFBO;
```

```
// Déverrouillage du Frame Buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);

/* ***** Seconde Passe **** */
// Nettoyage de l'écran
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Redimensionnement de la zone d'affichage
glViewport(0, 0, m_largeurFenetre, m_hauteurFenetre);

// Gestion de la caméra
camera.lookAt(modelview);

// Activation du shader
glUseProgram(shaderTexture.getProgramID());

// Verrouillage du VAO
glBindVertexArray(vao);

// Envoi des matrices
    shaderTexture.envoyerMat4("modelviewProjection",
projection * modelview);

// Verrouillage de la texture
glBindTexture(GL_TEXTURE_2D, framebuffer.getColorBufferID(0));

// Rendu
glDrawArrays(GL_TRIANGLES, 0, 6);

// Déverrouillage de la texture
    glBindTexture(GL_TEXTURE_2D, 0);

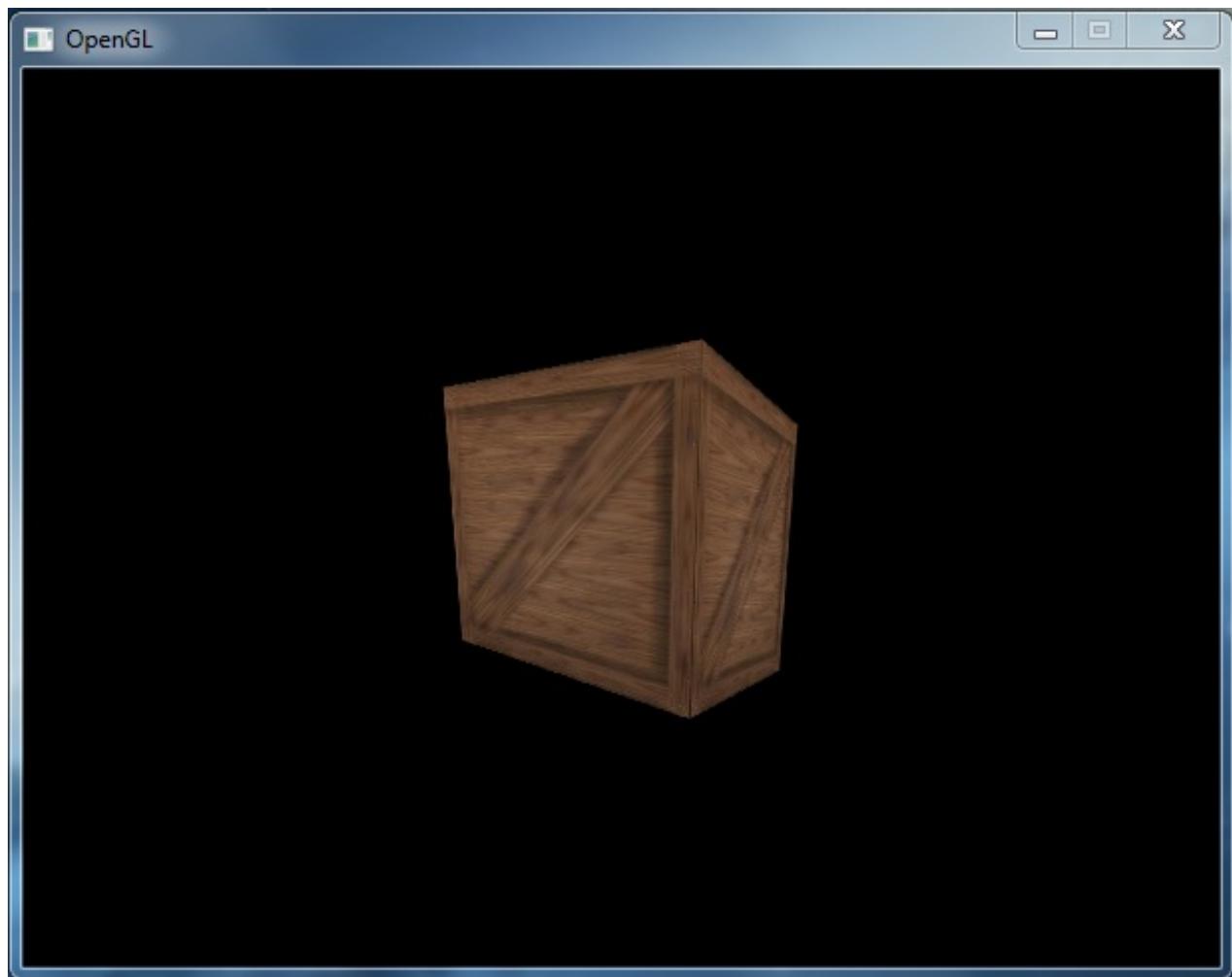
// Verrouillage du VAO
glBindVertexArray(0);

// Désactivation du shader
glUseProgram(0);

// Actualisation de la fenêtre
SDL_GL_SwapWindow(m_fenetre);
```

```
// Calcul du temps écoulé  
finBoucle = SDL_GetTicks();  
tempsEcoule = finBoucle - debutBoucle;  
  
// Si nécessaire, on met en pause le programme  
if(tempsEcoule < frameRate)  
    SDL_Delay(frameRate - tempsEcoule);  
}
```

Si vous êtes prêts, vous pouvez compiler tout ça pour voir ce que cela donne. 😊



C'est un peu bizarre ton truc on ne voit pas grand chose. 😕

Bon j'avoue, on ne voit pas grand chose.

On va ajouter un appel à la fonction **glClearColor()** juste avant d'appeler **glClear()** au niveau de la première passe. Cette fonction permet de donner une couleur par défaut à votre affichage au moment où les buffers sont nettoyés :

Code : C++

```
void glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)
```

- **red** : composante rouge de la couleur par défaut
- **green** : composante verte
- **blue** : composante bleue
- **alpha** : composante alpha

Appelons cette fonction dans la première passe juste avant **glClear()**, nous donnerons une couleur grise par défaut :

Code : C++

```
/* ***** Première passe ***** */

// Nettoyage de l'écran

glClearColor(0.5, 0.5, 0.5, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Le problème avec cette fonction est qu'elle applique la couleur par défaut à tous les **FBO**, écran compris. Pour remettre la couleur qu'il y avait avant il faut donc faire la même chose qu'avec **glViewport()** et rappeler la fonction une seconde fois au moment de la seconde passe.

C'est donc ce que nous allons faire en remettant cette fois la couleur noir :

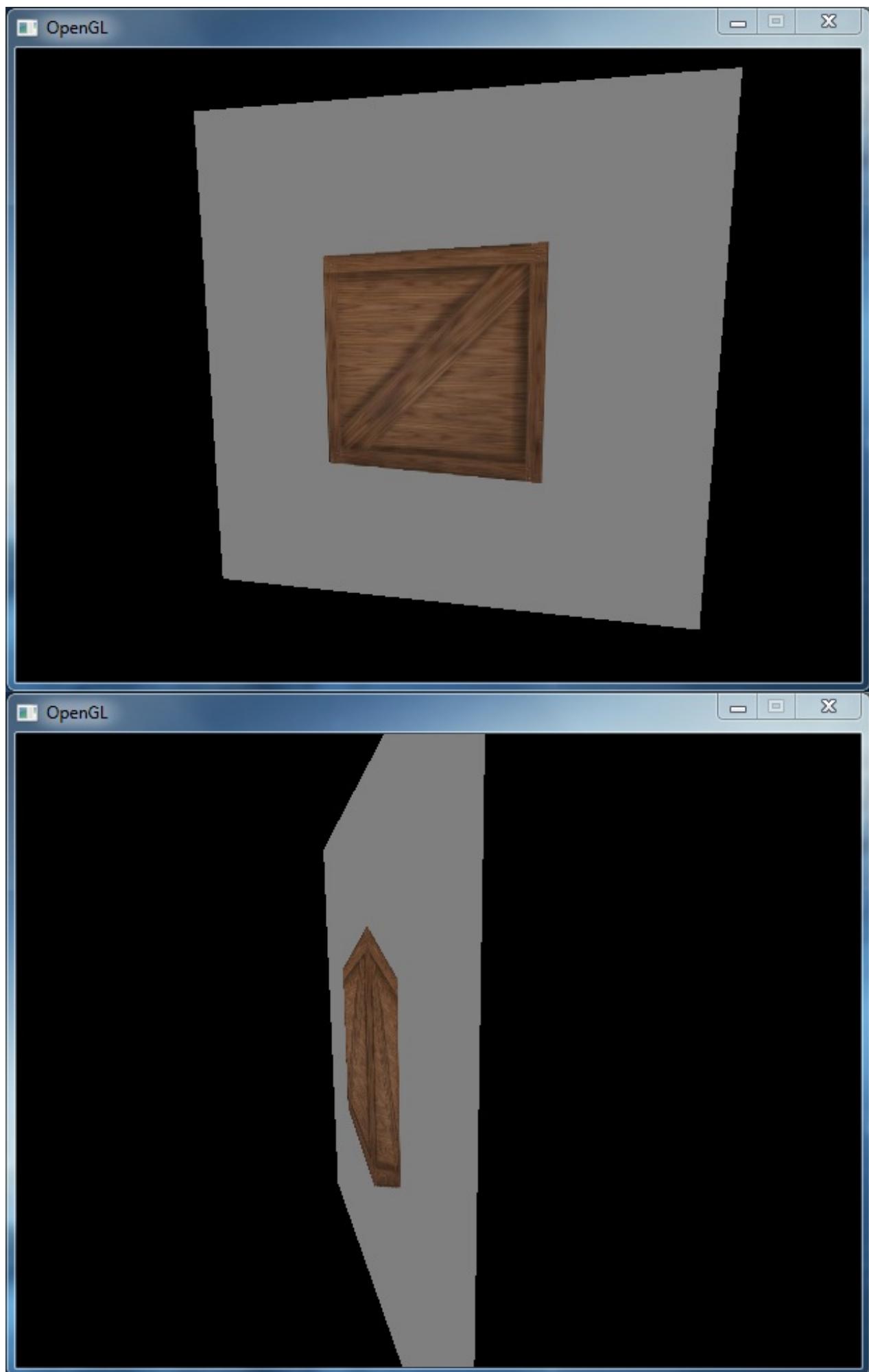
Code : C++

```
/* ***** Seconde passe ***** */

// Nettoyage de l'écran

glClearColor(0.0, 0.0, 0.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Recompilons maintenant tout cela :



C'est déjà un peu mieux.

Faites le tour de votre carré pour voir que votre caisse n'existe pas en 3D, elle est contenue dans une simple texture 2D. Grâce aux **FBO**, nous sommes en théorie capables d'appliquer n'importe quel effet sur cet affichage à l'aide des shaders (déformation, miroir, flou, ombre, etc.).

Cet exemple n'est pas bien parlant, mais le TP qui va arriver juste après ce chapitre, vous devrez en faire une utilisation plus utile.



Ce qu'il faut retenir

Piouf, nous aurons vu pas mal de choses dans ce chapitre. Entre la création d'un **FBO** et son utilisation finale, il y a plein de petites notions auxquelles il faut faire attention. Nous allons résumé tout cela en quelques points pour synthétiser toute cette partie.

Les préparatifs

Premièrement, l'utilisation d'un **FBO** est divisée en deux étapes que l'on appelle *passe*. La première contient les éléments affichés uniquement dans le **FBO**. La seconde quant à elle se concentre sur l'affichage normal de la scène (plus le contenu du **FBO**).

Avant même de commencer la première passe, il faut déterminer les *dimensions* du **FBO**. Plus elles seront grandes, meilleur sera votre rendu. Cependant cela impactera sur les performances de votre application final. Il est préférable de choisir des dimensions moins importantes pour économiser au maximum vos ressources.

Enfin, le dernier point à retenir pour les préparatifs est la création d'un second **couple de matrices** spécialement dédié à la première passe. On évite ainsi d'éventuelles erreurs en rapport avec la seconde. La matrice **projection** doit prendre en compte le rapport des dimensions du **FBO** dans la méthode **perspective()**.

La première passe

La première passe commence par le verrouillage du **FBO** à l'aide de la fonction **glBindFramebuffer()**. Incluez le déverrouillage immédiatement après pour ne pas oublier de le faire.

Le rendu se comporte ensuite de la même façon qu'un rendu classique. Cela va de la fonction **glClear()**, qui permet de nettoyer les buffers du **FBO** jusqu'au dernier objet que vous souhaitez afficher. Petite précision importante : Il est préférable de se positionner d'un point de vu **fixe**, et non mobile. Préférez donc la méthode **lookAt()** de la matrice **modelview**.

Pensez à appeler la fonction **glViewport()** pour redimensionner la zone d'affichage en fonction des dimensions du **FBO**.

La seconde passe

Le rendu de la second passe se concentre sur le rendu normal de la scène 3D. On retrouve ainsi tous les objets qui la composent, même ceux faisant partie du **FBO**.

Pensez la encore à appeler la fonction **glViewport()** pour redimensionner la zone d'affichage en fonction des dimensions de votre fenêtre.

Affichez enfin le contenu du **Color Buffer** du **FBO** sur une surface comme vous le feriez pour une texture classique.

Améliorations simples

Dans cette partie, nous allons faire plusieurs modifications dans notre classe **FrameBuffer**. Nous allons tout d'abord voir comment économiser un peu de ressources en évitant d'utiliser le **Stencil Buffer** lorsque l'on n'en a pas besoin. Nous implémenterons ensuite le constructeur de copie, vous savez déjà à quoi il peut bien servir. Enfin, nous verrons ensemble, dans une ultime partie, comment gérer plusieurs **Color Buffers**. Ce sera un peu technique, je ferai donc un point récapitulatif

dessus à la fin.

Éviter la création du Stencil Buffer

Un nouvel attribut

Le premier point que nous allons gérer concerne l'économie de ressources relatives au **Stencil Buffer**. En effet celui-ci n'est pas souvent utilisé, d'ailleurs nous ne l'avons jamais utilisé, nous pouvons donc éviter sa création dans la méthode **charger()** lorsque nous n'en avons pas besoin.

Pour cela, nous allons ajouter un nouvel attribut nommé **m_utiliserStencilBuffer** de type **bool** :

Code : C++

```
// Classe

class FrameBuffer
{
public:

    ...

private:
    GLuint m_id;
    int m_largeur;
    int m_hauteur;
    std::vector<Texture> m_colorBuffers;
    GLuint m_depthBufferID;
    bool m_utiliserStencilBuffer;
};
```

Celui-ci permettra de dire si oui ou non, nous devons créer le **Stencil Buffer**.

Modification du constructeur

Pour lui donner une valeur au moment au moment de créer un objet **FrameBuffer**, nous allons légèrement modifier le constructeur. Pour le moment, le prototype est le suivant :

Code : C++

```
FrameBuffer(int largeur, int hauteur);
```

Nous allons ajouter un autre paramètre pour lui permettre de donner une valeur à notre nouveau booléen. De préférence, nous l'ajouterais avec une valeur par défaut égale à **false** car le **Stencil Buffer** n'est pas souvent utilisé :

Code : C++

```
FrameBuffer(int largeur, int hauteur, bool utiliserStencilBuffer =
false);
```

Bien entendu, il faut modifier le constructeur dans le fichier **.cpp**. Il faut ajouter ce nouveau paramètre ainsi que l'initialisation du booléen :

Code : C++

```
FrameBuffer::FrameBuffer(int largeur, int hauteur, bool utiliserStencilBuffer) : m_id(0), m_largeur(largeur),  
m_hauteur(hauteur),  
m_colorBuffers(0), m_depthBufferID(0),  
m_utiliserStencilBuffer(utiliserStencilBuffer)  
{  
}
```

Pour compléter les modifications, pensons à initialiser le booléen dans le constructeur par défaut avec une valeur égale à **false** :

Code : C++

```
FrameBuffer::FrameBuffer() : m_id(0), m_largeur(0), m_hauteur(0),  
m_colorBuffers(0), m_depthBufferID(0),  
m_utiliserStencilBuffer(false)  
{  
}
```

Création des Render Buffers

Maintenant que le booléen possède une valeur (soit **false** par défaut, soit **true** par le programmeur) nous pouvons l'utiliser dans la méthode **charger()** pour nous permettre de bloquer ou non le chargement du **Stencil Buffer**.

Mais avant cela, nous allons revenir à la création du **double-Render Buffer** et se demander : comment peut-on séparer le **Depth** et le **Stencil Buffer** ? Car je vous rappelle que les deux se trouvent dans le même buffer.

La réponse est en fait très simple et se situe au niveau du *format interne*, celui que nous donnons à la méthode **creerRenderBuffer()** :

Code : C++

```
// Crédation du Depth et du Stencil Buffer  
creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);
```

Pour le moment, nous lui donnons la constante **GL_DEPTH24_STENCIL8** pour dire à OpenGL de créer un **double-RenderBuffer** contenant le **Depth** et le **Stencil**.

Pour ne garder que le **Depth**, il suffit juste de changer la constante en **GL_DEPTH_COMPONENT24**. Avec elle, OpenGL comprendrait qu'il ne doit créer que le **Depth Buffer**. L'appel à la méthode **creerRenderBuffer()** ressemblerait donc à ceci :

Code : C++

```
// Crédation du Depth Buffer
```

```
    creerRenderBuffer(m_depthBufferID, GL_DEPTH_COMPONENT24);
```

En définitif, si le booléen **m_utiliserStencilBuffer** est égal à la valeur **true** alors on conserve le **double-Render Buffer**, sinon on utilise la nouvelle constante pour ne garder que le **Depth Buffer** :

Code : C++

```
bool FrameBuffer::charger()
{
    // Début de la méthode

    ...

    // Création du Depth Buffer et du Stencil Buffer (si besoin)

    if(m_utiliserStencilBuffer == true)
        creerRenderBuffer(m_depthBufferID, GL_DEPTH24_STENCIL8);

    else
        creerRenderBuffer(m_depthBufferID, GL_DEPTH_COMPONENT24);

    // Association du Color Buffer

    ...
}
```

Association des Renders Buffers

L'appel à la fonction **glFramebufferRenderbuffer()** doit également être modifié. En effet, son paramètre **attachment** permet de spécifier le point d'attache du **Render buffer**, sa valeur est pour le moment égale à la constante **GL_DEPTH_STENCIL_ATTACHMENT** :

Code : C++

```
// Association du Depth et du Stencil Buffer

glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);
```

Dans le cas où le booléen est à **false**, il faudrait changer cette constante car le **Stencil Buffer** n'existe pas. Celle-ci deviendrait alors **GL_DEPTH_ATTACHMENT** :

Code : C++

```
// Association du Depth Buffer

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, m_depthBufferID);
```

En utilisant le booléen, l'association du **Render Buffer** ressemblerait donc à :

Code : C++

```
bool FrameBuffer::charger()
{
    // Début de la méthode

    ...

    // Association du Color Buffer

    ...

    // Création du Depth Buffer et du Stencil Buffer (si besoin)

    if(m_utiliserStencilBuffer == true)
        glFramebufferRenderbuffer(GL_FRAMEBUFFER,
        GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);

    else
        glFramebufferRenderbuffer(GL_FRAMEBUFFER,
        GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, m_depthBufferID);

    // Vérification de l'intégrité du FBO

    ...
}
```

Création d'un objet FrameBuffer

Au final, si vous souhaitez utiliser le **Stencil Buffer** avec un **FBO** vous devrez ajouter la valeur **true** au constructeur :

Code : C++

```
// Frame Buffer avec Stencil Buffer

FrameBuffer frameBuffer(512, 512, true);
```

Si vous ne souhaitez pas l'utiliser, soit vous ne mettez pas de booléen, soit vous mettez la valeur **false** :

Code : C++

```
// Frame Buffer sans Stencil Buffer

FrameBuffer frameBuffer(512, 512);
```

Le constructeur de copie

Le constructeur de copie est une pseudo-méthode que nous avons l'habitude de rencontrer maintenant. Vous savez que les objets OpenGL ne peuvent être copiés comme de simples variables, ils doivent être totalement rechargés dans leur copie exactement comme s'il s'agissait de pointeurs. On ne risque ainsi pas de perdre des données si un objet original est détruit.

Le constructeur de copie est assez simple puisqu'il ne prend en paramètre qu'une *référence constante* sur un objet de même type, ici **FrameBuffer** :

Code : C++

```
FrameBuffer(const FrameBuffer &frameBufferACopier);
```

Son implémentation commencera comme d'habitude par la copie des attributs variables. Si on regarde notre liste d'attributs, on remarque qu'il n'y en a que trois : **m_largeur**, **m_hauteur** et **m_utiliserStencilBuffer**.

Code : C++

```
FrameBuffer::FrameBuffer(const FrameBuffer &frameBufferACopier)
{
    // Copie de la largeur, de la hauteur et du booléen

    m_largeur = frameBufferACopier.m_largeur;
    m_hauteur = frameBufferACopier.m_hauteur;
    m_utiliserStencilBuffer =
frameBufferACopier.m_utiliserStencilBuffer;
}
```

Ces trois attributs peuvent être copiés directement. En revanche, les autres sont(ou contiennent) des objets OpenGL ils ne peuvent donc pas être copiés par le signe `=`. La méthode qui leur donne une valeur est la méthode **charger()**. C'est elle qui charge le **FBO** et qui permet de gérer un identifiant à tous ces attributs.

Nous l'appellerons donc cette méthode pour simuler une copie de ces attributs :

Code : C++

```
FrameBuffer::FrameBuffer(const FrameBuffer &frameBufferACopier)
{
    // Copie de la largeur, de la hauteur et du booléen

    m_largeur = frameBufferACopier.m_largeur;
    m_hauteur = frameBufferACopier.m_hauteur;
    m_utiliserStencilBuffer =
frameBufferACopier.m_utiliserStencilBuffer;

    // Chargement de la copie du Frame Buffer
    charger();
}
```

Le constructeur de copie est maintenant complet. 😊

Gérer plusieurs Color Buffers

Cette partie est en cours de ré-écriture, sa précédente version était trop lourde et inutilement complexe. Néanmoins, vous avez déjà pratiquement tout vu sur les **FBO**. 😊

Télécharger (Windows, UNIX/Linux, Mac OS X) : [Code Source C++ du chapitre sur les Frame Buffers Objects](#)

Ce chapitre était un peu compliqué et il y avait pas mal de notions à assimiler en une fois. Vous comprendrez pourquoi je l'ai placé à la fin des notions avancées.

Les **Frame Buffers** permettent de faire pas mal de choses mais il faut avoir un minimum de connaissances en OpenGL pour

connaitre leur fonctionnement. Je vous rassure tout de suite, si je vous ai fait un chapitre dessus c'est parce que vous êtes tout à fait capables de les utiliser. 😊

Après tout ce que vous avez vu jusqu'à présent, vous avez une bonne vu d'ensemble de ce que propose OpenGL. Il y a plein d'autres fonctionnalités à voir évidemment, mais à partir de maintenant nous n'allons pas vraiment apprendre de nouvelles notions pures et dures mais nous allons plutôt utiliser tout ce que nous avons vu pour faire de véritables rendus. Nous allons apprendre à charger des modèles 3D statiques et animés, utiliser des SkyBox, des height maps, et bien évidemment des effets réalistes avec les shaders.

Avant de passer à ce programme, je vous invite à faire un petit TP, basé sur le premier que vous avez déjà fait, pour mettre en pratique ce que nous avons vu dans cette deuxième partie. Il y aura des **VBO**, des **VAO**, des **shaders** et bien évidemment des **FBO**. 🎉

Ce tutoriel est loin d'être terminé, nous avons encore pas mal de choses à voir et à apprendre ensemble 😊. Pour vous donner un avant-goût des prochains chapitres, sachez que :

- La **troisième partie** sera consacrée aux techniques du jeu vidéo en général, comme le chargement de modèles 3D animés et non-animés, les polices d'écriture, ...
- La **quatrième partie** sera consacrée à l'élaboration d'effets avancés comme la lumière, le bump mapping, l'eau, ...

Si vous avez des remarques ou des éléments que vous aimeriez que je développe, merci de m'en faire part dans les commentaires de ce tutoriel. Toute critique (constructive) sera la bienvenue 😊.