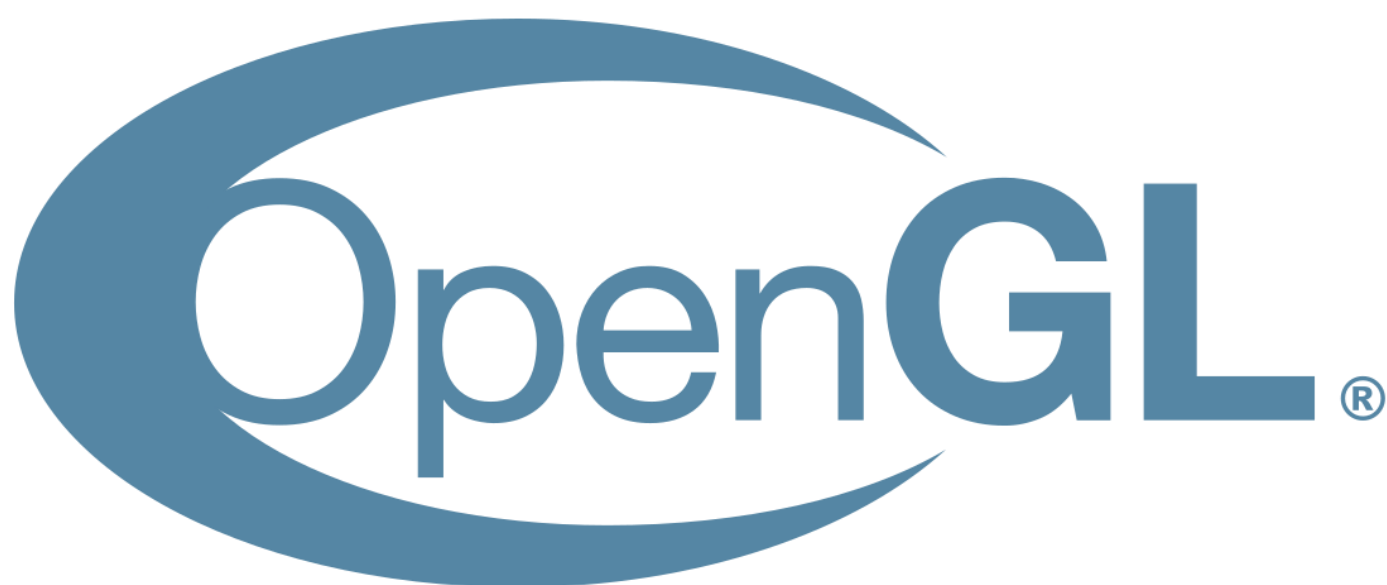


Apprendre OpenGL



Sommaire

Introduction	3
La Théorie : Les Bases d'OpenGL	4
<i>Qu'est-ce qu'OpenGL ?</i>	<i>4</i>
<i>Les différentes couches.....</i>	<i>5</i>
<i>Vertex Buffer.....</i>	<i>6</i>
<i>Vertex Buffer Layout.....</i>	<i>7</i>
<i>Vertex Array (ou Vertex Array Object).....</i>	<i>7</i>
<i>Shader.....</i>	<i>8</i>
<i>Index Buffer</i>	<i>9</i>
<i>Pour résumer</i>	<i>11</i>
La Pratique : Utiliser l'interface.....	12
<i>VertexBuffer</i>	<i>12</i>
<i>VertexBufferLayout.....</i>	<i>12</i>
<i>VertexArray.....</i>	<i>13</i>
<i>Shader.....</i>	<i>13</i>
<i>IndexBuffer</i>	<i>13</i>
<i>Renderer</i>	<i>14</i>
<i>Framework de Test.....</i>	<i>15</i>
Sources	22
<i>The Cherno.....</i>	<i>22</i>
<i>learnOpenGL.com</i>	<i>22</i>
<i>openGL-tutorial.com.....</i>	<i>22</i>
<i>openclassroom.com.....</i>	<i>22</i>

Introduction

Dans le cadre du cours « Introduction to Computer Graphics » de [Tobias Isenberg](#), vous allez découvrir un ensemble de techniques utilisées dans le monde de la programmation graphique. Vous allez aussi découvrir ce que l'on appelle le « Graphics Pipeline » ou l'ensemble des étapes que doit effectuer l'ordinateur avant de pouvoir afficher un pixel à l'écran.

Ce cours (assez théorique) tentera de vous faire mettre en pratique ces notions via des TP et un projet que vous devrez faire sous Visual Studio en C++. Tous les concepts théoriques que vous verrez sont implémentés dans différentes bibliothèques de différentes manières (comme DirectX, OpenGL, Vulkan, etc.). Pour tous les étudiants intéressés par le jeu vidéo, il est intéressant de comprendre ces techniques car elles sont celles mises en œuvre par les principaux moteurs de jeu (comme Unity ou l'Unreal Engine). Ici, dans le cadre de ce cours vous utiliserez une implémentation moderne d'OpenGL (OpenGL 3.3).

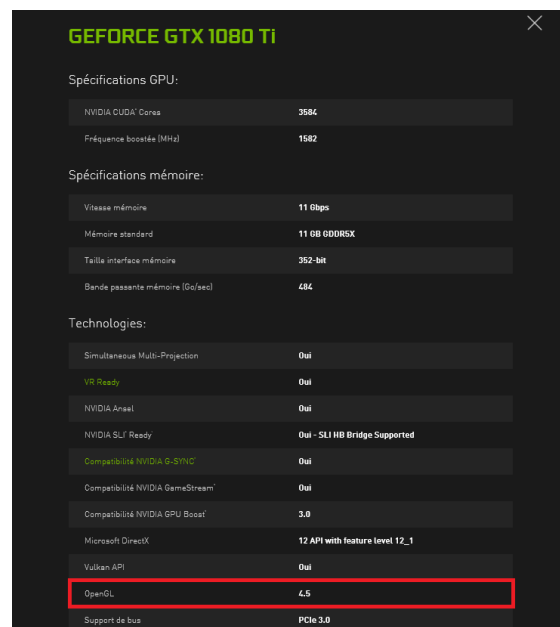
Ce document a été fait pour vous aider à comprendre OpenGL et la relation avec le cours.

La Théorie : Les Bases d'OpenGL

Qu'est-ce qu'OpenGL ?

Voilà la définition de Wikipédia : « *OpenGL (Open Graphics Library) est un ensemble normalisé de fonctions de calcul d'images 2D ou 3D. Cette interface de programmation est disponible sur de nombreuses plateformes où elle est utilisée pour des applications qui vont du jeu vidéo jusqu'à la CAO en passant par la modélisation.* ». Ce qu'il faut retenir de cette définition c'est qu'**OpenGL n'est** qu'une interface normalisée, c'est-à-dire **un ensemble de fonctions qui sont disponibles dans toutes les cartes graphiques** (modernes) pour afficher des éléments (2D ou 3D) à l'écran. Ainsi, chaque constructeur de carte graphique programme sa version compatible avec la norme d'OpenGL et indique, normalement, quelle version est prise en charge.

Figure 1 : Spécifications techniques d'une carte graphique



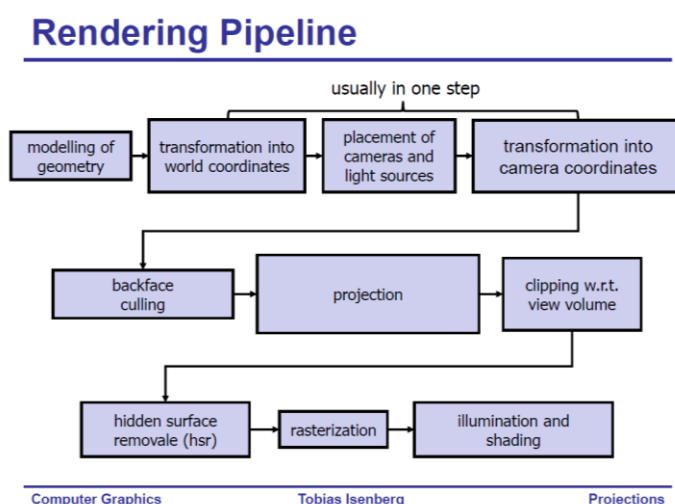
GEFORCE GTX 1080 Ti	
Spécifications GPU:	
NVIDIA CUDA® Cores	3584
Fréquence boostée (MHz)	1582
Spécifications mémoire:	
Vitesse mémoire	11 Gbps
Mémoire standard	11 GB GDDR5X
Taille interface mémoire	352-bit
Bande passante mémoire (Go/sec)	484
Technologies:	
Simultaneous Multi-Projection	Oui
VR Ready	Oui
NVIDIA Ansel	Oui
NVIDIA SLI Ready	Oui - SLI HB Bridge Supported
Compatibilité NVIDIA G-SYNC™	Oui
Compatibilité NVIDIA GameStream™	Oui
Compatibilité NVIDIA GPU Boost	3.0
Microsoft DirectX	12 API with feature level 12_1
Vulkan API	Oui
OpenGL	4.5
Support de bus	PCIe 3.0

Ainsi, tout ce que l'on va faire ici, c'est d'utiliser ces fonctions pour afficher à l'écran ce que l'on souhaite dessiner.

Les différentes couches

OpenGL est ce que l'on appelle **une machine d'état** : c'est-à-dire qu'à chaque étape du « graphics pipeline » elle utilise les valeurs qu'on lui a demandé de stocker (que l'on lui a liées) pour faire ces calculs et dessiner chaque pixel. En somme, en fonction de l'état (les valeurs qu'il a), OpenGL tracera d'une façon ou d'une autre les objets.

Figure 2: le Graphics Pipeline



L'ensemble des valeurs/états est appelé le Contexte OpenGL (OpenGL context). C'est en changeant les différentes valeurs d'OpenGL au bon moment que l'on obtient le résultat souhaité.

La façon de travailler avec OpenGL est toujours la même (ou presque) :

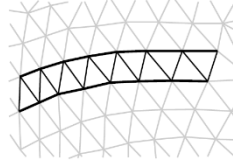
- 1) On génère un objet dans OpenGL du type que l'on veut et celui-ci nous met à jour l'ID (passé en paramètre par référence) : `glGenBuffer(&ID)` `glGenTexture(&ID)` (ID est un unsigned int)
- 2) On lie à OpenGL l'ID de l'objet que l'on souhaite en lui précisant à quoi il correspond : `glBindBuffer(GL_ARRAY_BUFFER, ID)`, `glBindTexture(GL_TEXTURE_2D, ID)`
- 3) Une fois que tout est lié comme on le souhaite, on dessine (draw call) et OpenGL dessine en utilisant ce qui est lié

Vertex Buffer

Pour travailler OpenGL a besoin de données. Comme vous l'avez vu en cours (ou vous le verrez), la primitive la plus simple à utiliser pour dessiner n'importe quelle forme est le triangle.

Figure 3 : les raisons du choix triangle

- polygons to define the surface of objects
- triangle meshes
 - polygon with fewest vertices
 - always convex & planar
→ defines unique surface
- triangle strips: faster rendering



Ainsi, il faut définir un ensemble de points pour OpenGL qui formeront l'ensemble des triangles à tracer. Dans les faits, **on stockera tous les points dans un seul gros tableau que l'on appellera le VertexBuffer (VB)**. Au fur et à mesure que l'on avance, on peut vouloir associer d'autres valeurs à chaque point (comme sa couleur ou sa normale), on les mettra dans ce même tableau juste après les coordonnées du point. S'il faut retenir quelque chose c'est que **le vertexBuffer représente les données brut pour OpenGL** (il stocke les valeurs mais ne sait pas à quoi correspond quoi pour le moment). On va par la suite essayer de faire comprendre de manière logique à OpenGL comment sont regroupées les données.

Figure 4 : Exemple d'un VertexBuffer

```
//Tableau contenant les positions des points (8 pour un cube) et leur UV coords (2)
float VB[] = {

    -50.0f, -50.0f, 50.0f, 0.0f, 0.0f, //0
    50.0f, -50.0f, 50.0f, 1.0f, 0.0f, //1
    50.0f, 50.0f, 50.0f, 1.0f, 1.0f, //2
    -50.0f, 50.0f, 50.0f, 0.0f, 1.0f, //3

    - 50.0f, -50.0f, -50.0f, 0.0f, 0.0f, //4
    50.0f, -50.0f, -50.0f, 1.0f, 0.0f, //5
    50.0f, 50.0f, -50.0f, 1.0f, 1.0f, //6
    -50.0f, 50.0f, -50.0f, 0.0f, 1.0f //7

};
```

Tout ce qui nous reste à faire pour l'instant, c'est de dire à OpenGL d'utiliser ce tableau pour créer son objet VertexBuffer et l'utiliser pour tracer les triangles au prochain draw call.

Figure 5 : exemple d'utilisation d'un VertexBuffer

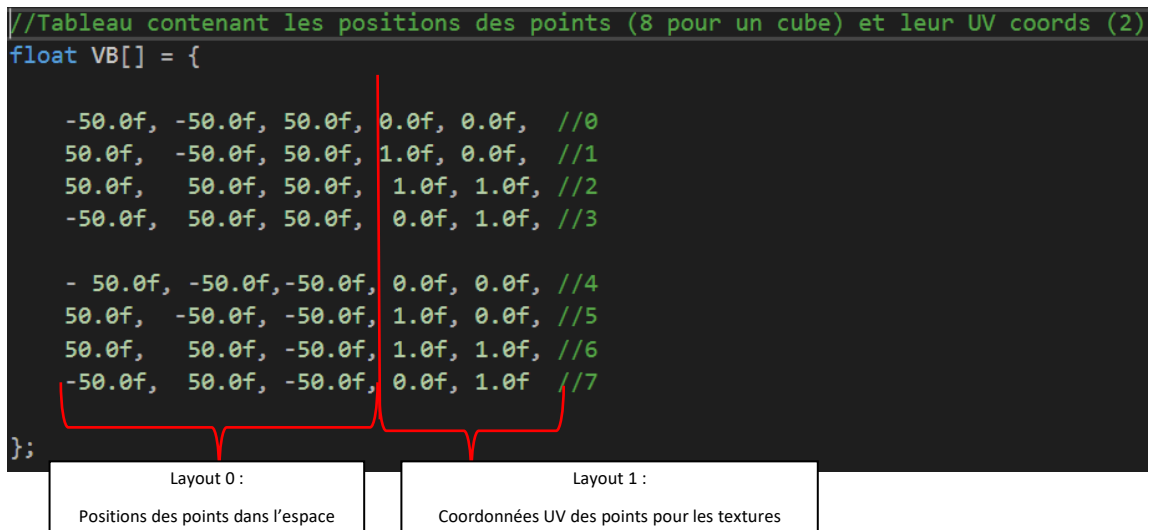
```
unsigned int ID;

//On génère un buffer (il va mettre l'indice du buffer dans la variable ID)
glGenBuffers(1, &ID);
//On lui précise ce que l'on va utiliser (ici un simple tableau), On lie notre buffer sur lequel on va travailler
glBindBuffer(GL_ARRAY_BUFFER, ID);
//On le remplit de données
glBufferData(GL_ARRAY_BUFFER, tailleVB, VB, GL_STATIC_DRAW);
```

Vertex Buffer Layout

Maintenant que l'on a stocké toutes les données, il faut les « **délimiter** ». C'est-à-dire expliquer à OpenGL comment est structuré notre vertexBuffer. Cela se fait via les fonctions `glEnableVertexAttribArray(num)`, `glVertexAttribPointer(...)`. La première indique à OpenGL le numéro du layout que l'on souhaite utiliser, la seconde explique à OpenGL comment est structuré le layout (combien de float, etc.). En somme cela revient à faire ça :

Figure 6 : Explication du VertexBuffer Layout

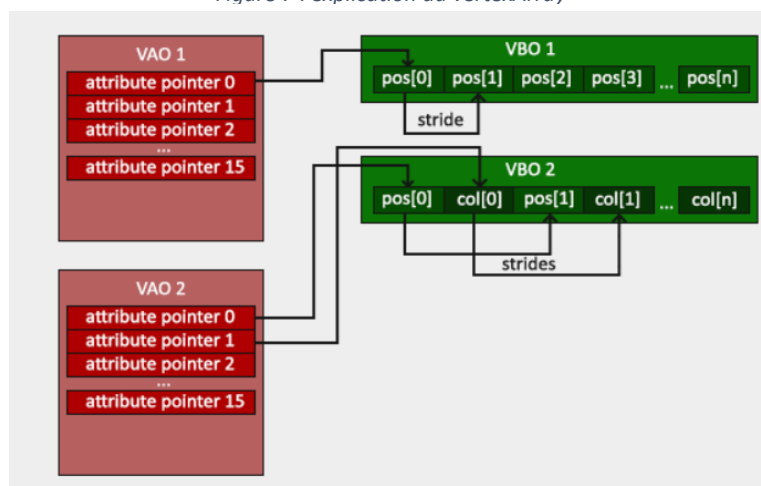


Ici, on ne parle que de la structure du vertexBuffer. Pour l'instant, OpenGL ne sait toujours pas à quoi correspond quoi (les données n'ont toujours pas de sens). C'est par la suite que l'on va lui expliquer.

Vertex Array (ou Vertex Array Object)

Le vertex Array, est juste un regroupement du vertexBuffer et de ses différents vertexBufferLayout (principalement l'appel aux fonctions de structure du layout : `glEnableVertexAttribArray(num)`, `glVertexAttribPointer(...)`). L'objectif ici est juste de **donner du sens à nos données**.

Figure 7 : explication du vertexArray



Shader

En programmation graphique, **un shader est juste un programme exécuté par la carte graphique (GPU)**. Ainsi, toutes les instructions dans un shader ne seront traitées que par la carte graphique et pas par le processeur (CPU). OpenGL nomme les shaders des « program ». Ainsi, dans un shader, on ne code pas en C++ : on utilise le langage de programmation des shaders (ici **l'OpenGL Shading Language (GLSL)**). On écrira le shader dans un autre fichier (en général NOMSHADER.shader). Il existe de nombreux shaders qui font pleins de choses différentes. Dans le cadre du cours, ceux que vous utiliserez sont **le Vertex Shader et le Fragment Shader**. Le premier est un shader (un programme pour la carte graphique) qui est appelé pour chaque point (vertex en anglais) :

Figure 8 : Structure d'un Shader

Déclaration du type de shader	<code>#shader vertex</code>
Déclaration de la version	<code>#version 330 core</code>

Récupération des VertexBufferLayout (selon les numéros passés en paramètre)	<pre>//On écrit les shaders //VertexShader ==> exécuté une fois par point //FragmentShader ==> exécuté une fois par pixel //on indique quel est l'id et le type d'entrée layout(location = 0) in vec4 position; layout(location = 1) in vec2 texCoords;</pre>
---	--

Uniform est une variable en GLSL (en général envoyée depuis le code C++ (CPU-side))	<pre>//Varying => passer des données du VS au FS (les in et out) //v_ pour varying out vec2 v_TextureCoords; uniform mat4 u_MVP; //ModelViewProjection Matrice => Matrice de projection</pre>
---	--

Corps du shader, le code que va exécuter le shader lorsqu'il sera appelé par la carte graphique	<pre>void main() { // /\ L'ordre est important dans un produit Matriciel /\ gl_Position = u_MVP * position; v_TextureCoords = texCoords; }</pre>
---	--

Le second est un shader (un programme pour la carte graphique) qui est appelé pour chaque pixel (ou fragment) :

Figure 9 : Exemple d'un Fragment Shader

```
#shader fragment
#version 330 core

in vec2 v_TextureCoords;

//Ce qui sort du fragment shader (ici que la couleur à afficher)
layout(location = 0) out vec4 color;

//Ce que l'on prend comme variable (sous forme d'uniform)
uniform vec4 u_Color;

//Pour spécifier que cet uniform est l'emplacement du slot => mot-clé sampler2D
uniform sampler2D u_Texture;

void main() {

    // fonction texture(slot,UV coords)
    vec4 texColor = texture(u_Texture, v_TextureCoords);
    color = texColor;

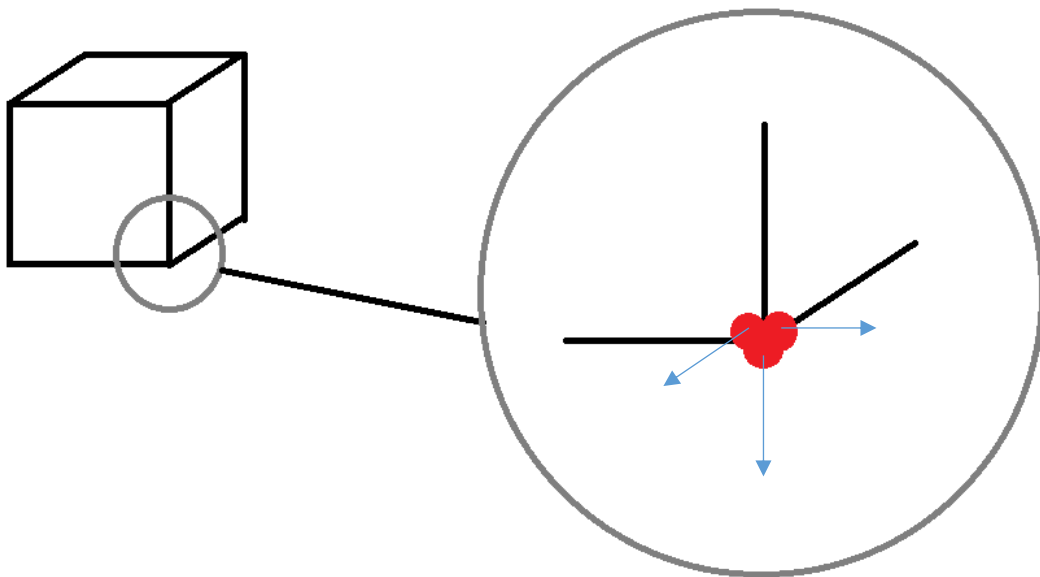
}
```

Pour utiliser un shader en OpenGL, il faut utiliser un ensemble de fonction que l'on ne détaillera pas ici.

Ainsi, on se rend compte que si l'on ne fait pas attention, on peut vite multiplier inutilement le nombre de points utilisés. Pour un simple cube ici, dans le pire des cas, on multiplie par 4 le nombre de points utilisés.

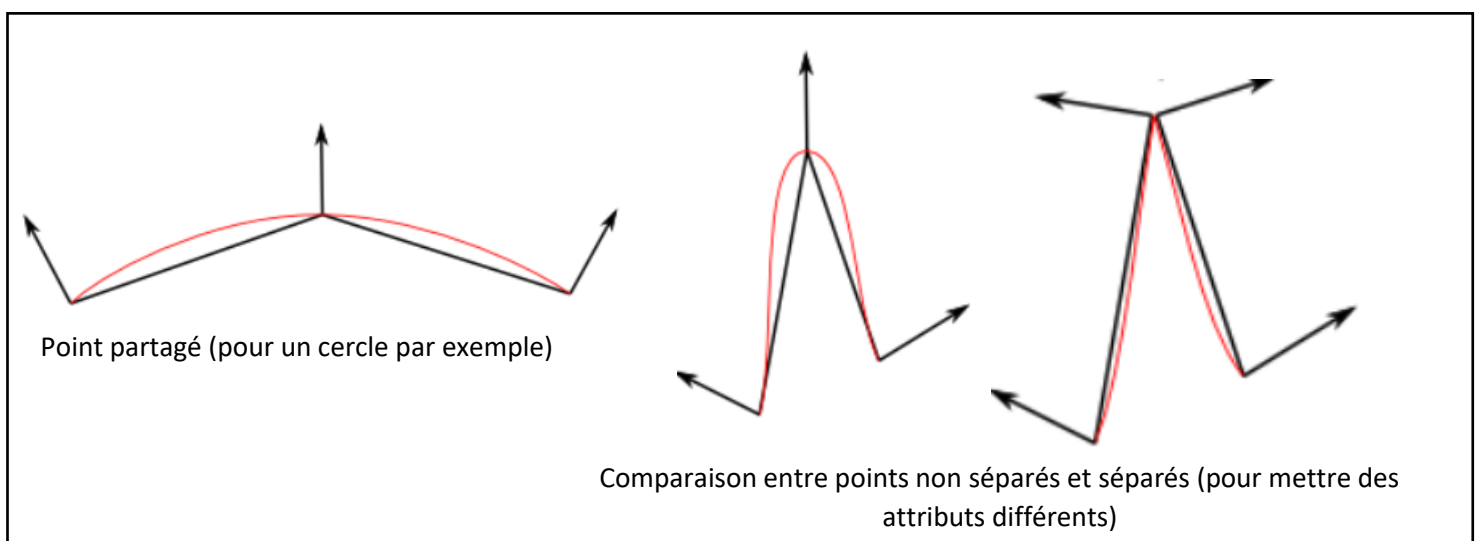
Ici l'utilisation de l'index buffer semble intéressante. Cependant, il faut se méfier ! Car si l'on veut rajouter des propriétés à chaque point, on est obligé d'utiliser la solution avec les 36 points différents. En effet, si on doit associer une normale à chaque point, on est embêté dans le premier cas car pour le même point on devrait associer 3 valeurs différentes (ce qui n'est pas possible). Ainsi, ici on est obligé de tracer 3 points à la même position.

Figure 12 : Problème avec l'indexBuffer



Cependant, le cube est un cas extrême dans le sens où chaque point ne peut être partagé. Dans la plupart des cas, il faut **réfléchir aux points que l'on peut partager ou non selon les propriétés que l'on souhaite utiliser** pour l'objet.

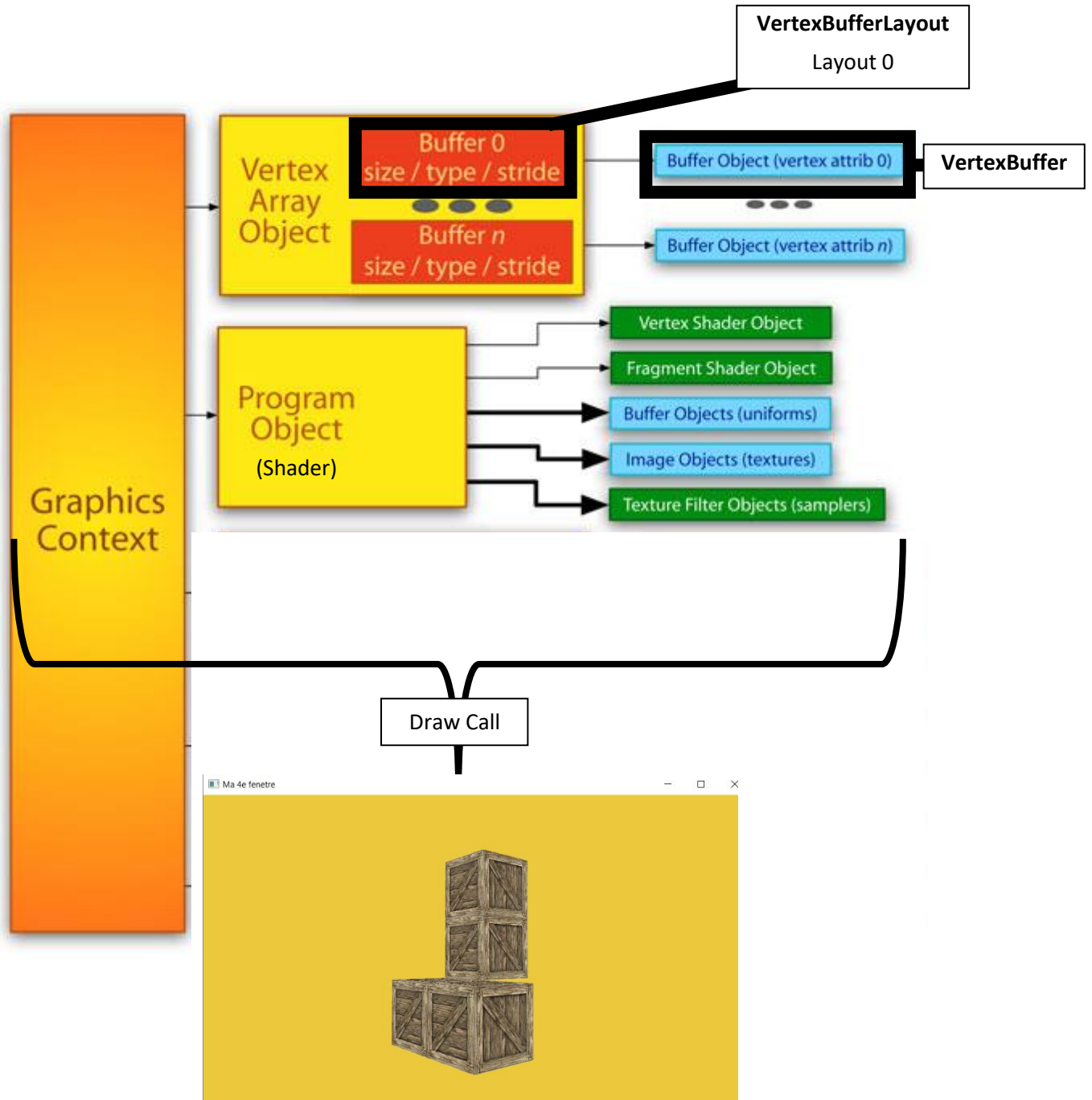
Figure 13 : Réflexion sur l'indexBuffer



Pour résumer

On peut résumer tout ce que l'on a vu avec le schéma suivant :

Figure 14 : Résumé



La Pratique : Utiliser l'interface

Pour vous aider, il vous est fourni un projet Visual Studio où la majorité du code OpenGL est abstrait dans des classes faisant la majorité du travail. Cela a pour but de vous **simplifier la compréhension et l'utilisation d'OpenGL** dans le cadre de ce cours.

De manière générale, on va procéder de la façon suivante : on crée nos données et on remplit notre VertexBuffer, puis on crée le VertexBufferLayout qui explique la structure du VertexBuffer. On regroupe le tout dans le VertexArray. Puis on crée le shader (en lui donnant le lien du fichier du shader), l'indexBuffer et les éventuelles textures. Finalement dans la boucle de rendu (rendering loop), on bind() tout avant le dessin et l'on demande au renderer de dessiner tout ce que on lui passe en paramètre.

REMARQUE : Etant donné que vous n'avez que très peu d'expérience en C++ et peut-être aucun concept de ce qu'est la programmation orientée-objet, ce guide tentera de vous expliquer simplement ce que fait chaque classe importante du projet Visual Studio. Pour faire simple, une classe est un regroupement de variables (attributs) et de fonction (méthodes) qui ne sont que partiellement utilisable depuis l'extérieur. L'idée est d'offrir à l'utilisateur de la classe (c'est-à-dire vous) une facilité d'usage pour des objets qui sont complexes à modéliser.

VertexBuffer

Comme son nom l'indique, cette classe représente l'objet VertexBuffer. On lui donne juste le tableau (ici data) avec nos positions (et autres attributs) et la taille en octet (nombre de points * nombre de floats par point * sizeof(float)), et la classe va s'occuper de faire ce qu'il faut avec OpenGL.

Figure 15 : Construction d'un VertexBuffer

```
VertexBuffer vb = VertexBuffer(data, sizeData);
```

VertexBufferLayout

Comme son nom l'indique, cette classe représente l'objet VertexBufferLayout. Ainsi ici, on va juste expliquer comment on découpe notre vertexBuffer :

Figure 16 : Utilisation d'un VertexBufferLayout

```
//Définition de notre layout de notre VertexBuffer pour notre VertexArray
//Ici la numérotation des layouts est automatique (le premier est 0)
VertexBufferLayout layout;

//On utilise 3 floats par points (positions) => layout regroupe ce qui va ensemble
//layout location 0 (cf vertex shader)
layout.Push<float>(3);

// On utilise 2 points par UV coords
//layout location 1 (cf vertex shader)
layout.Push<float>(2);
```

VertexArray

Ensuite si vous avez bien compris, on regroupe tout dans notre VertexArray (ou Vertex Array Object) :

Figure 17 : construction d'un VertexArray

```
//On regroupe tout dans notre vertex Array (on donne du sens à nos données)
VertexArray VAO = VertexArray();
VAO.AddBuffer(vb, layout);
```

Ce dernier s'occupe de tout lier lorsque l'on le construit. Si on utilise plusieurs vertexArray, il faut le bind() avant le dessin et le unbind() après le dessin.

Shader

Il faut ensuite indiquer quel shader on souhaite utiliser :

Figure 18 : Création d'un objet Shader

```
//Création du shader
Shader shader = Shader("Ressources/Shaders/Basic.shader");
shader.Bind();
```

On définit ici les valeurs que l'on veut mettre pour les différents Uniforms :

Figure 19 : Utilisation d'un Uniform

```
//Uniform --> envoi de données du CPU vers les shaders pour être utilisé comme une variable dans le shader
//Utilisation uniform après bind avec le shader qui utilise l'uniform
//Uniform appelé à chaque dessin (draw call)
shader.SetUniform4f("u_Color", 0.0f, 0.0f, 1.0f, 1.0f);
```

IndexBuffer

Comme son nom l'indique, cette classe représente l'objet IndexBuffer. On lui donne juste le tableau (ici indices) avec nos indices que l'on souhaite utiliser et le nombre d'indice :

Figure 20 : Création d'un IndexBuffer

```
//On stocke et lie à l'index buffer les indices
IndexBuffer ib = IndexBuffer(indices, 36);
```

Renderer

C'est cette classe qui s'occupe de tout faire. En effet, on lui donne le VertexArray, l'IndexBuffer et le shader, et lui paramètre OpenGL pour que cela dessine ce que l'on veut :

Figure 21 : Utilisation du renderer

```
//Ici notre renderer ne stocke rien => on peut se permettre de le créer à chaque frame
Renderer renderer;

//en tant normal, un renderer prend un vertexArray + IndexBuffer + Materials (pas shader)
renderer.Draw(VAO, ib, shader);
```

Il faut que la méthode `renderer.Draw(...)` soit appelée dans la boucle de rendu (`while(true)`).

Framework de Test

Pour que vous puissiez créer des petites applications OpenGL rapidement, ce projet Visual Studio met à votre disposition un petit Framework de Test.

REMARQUE : Ce petit Framework de test n'est pas là pour faire de gros projet : il vise à vous aider dans la mise en œuvre de petites applications pour comprendre et illustrer le cours. Ainsi, il n'est par exemple pas adapté pour faire le rendu de nombreux objets différents (car il n'est ici pas automatisé pour un souci de clarté et d'explication).

Chaque mini application doit être une nouvelle classe qui hérite (syntaxe : *class NomTest : public Test*) de la classe Test et doit créer au moins 3 méthodes :

- Un constructeur vide *NomClasseDeTest()* qui s'occupera d'initialiser tout ce qui doit l'être dans votre application
- *OnRender()* ce que l'application doit faire dans la boucle de rendu
- *OnImGuiRender()* ce que doit faire l'interface de débogage ImGui dans votre mini application

La redéfinition des autres méthodes est facultative.

Figure 22 : Méthodes de l'interface de test

```
Test() {};  
virtual ~Test() {};  
  
//Fonction a redef par nos tests qui héritent de Test  
virtual void OnUpdate(float delta) {};  
virtual void OnRender() {};  
virtual void OnImGuiRender() {};
```

Ensuite une fois que tout est créé dans la mini-classe application, il suffit de l'ajouter dans le menu de test de notre main.cpp :

Figure 23 : Ajout d'un test au menu

```
//On ajoute les tests disponibles au menu des tests  
menuTest->AjouteTest<test::TestClearCouleur>("Couleur du Vide");  
menuTest->AjouteTest<test::TestTexture2D>("Texture 2D");  
menuTest->AjouteTest<test::Test3D>("3D");  
menuTest->AjouteTest<test::TestIB3D>("Cube IndexBuffer");  
menuTest->AjouteTest<test::TestLumiere>("Lumiere et Illuminations");  
  
menuTest->AjouteTest<test::NomDeVotretest>("Nom dans le Menu");
```

Si vous avez déjà regardé le code fourni, vous devez être un peu perdu par la façon de créer et d'utiliser ces tests. Nous allons désormais détailler pas à pas les étapes de la création d'un test.

Exemple d'utilisation du Framework :

- 1) Tout d'abord on crée la nouvelle classe héritant de Test qui va accueillir notre mini-application. Celle-ci se décompose en 2 fichiers (un .cpp et un .h) :

Figure 25 : Ajout d'un nouveau fichier

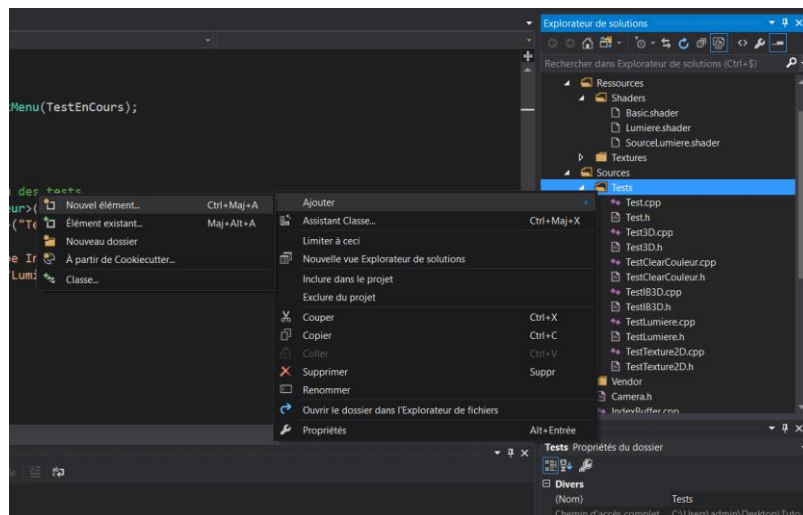
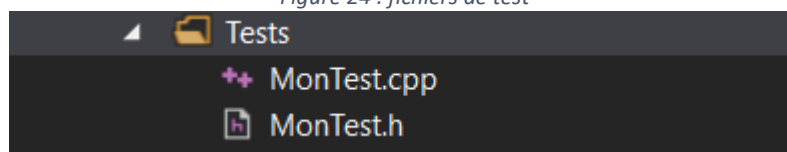


Figure 24 : fichiers de test



- 2) Ensuite, on crée la classe qui représentera notre mini-application :

Figure 27 : MonTest.h

```
1  #pragma once
2
3  #include "Test.h"
4
5  namespace test {
6
7      //Hérite de test
8      class MonTest : public Test {
9
10     public:
11         MonTest();
12         ~MonTest();
13
14         void OnUpdate(float delta) override;
15         void OnRender() override;
16         void OnImGuiRender() override;
17
18     private:
19
20
21     };
22
23
24 }
```

Figure 27 : MonTest.cpp

```
1  #include "MonTest.h"
2
3  test::MonTest::MonTest(){
4
5  }
6
7  test::MonTest::~MonTest(){
8
9  }
10
11 void test::MonTest::OnUpdate(float delta){
12
13 }
14
15 void test::MonTest::OnRender(){
16
17 }
18
19 void test::MonTest::OnImGuiRender(){
20
21 }
22
```


- 3) Dans le cadre de cet exemple, on va créer un Test très simple : il va juste dessiner un rectangle coloré. Pour que notre application ne gaspille pas la mémoire, nous allons faire en sorte qu'elle alloue et libère de la mémoire dynamiquement. Pour cela nous allons utiliser un type spécifique du C++ qui s'appelle « *unique_ptr* ». Pour faire simple, un *unique_ptr* est un pointeur (une variable qui pointe sur la case mémoire de nos données) qui lorsqu'il n'est plus accessible (hors de portée) libère sa mémoire. Ainsi, toutes nos données seront accessibles via ses pointeurs. Tout d'abord, on inclut ce dont on va avoir besoin :

Figure 28 : Include à faire

```
//Pour unique_ptr
#include <memory>

#include "Test.h"
#include "VertexBuffer.h"
#include "VertexBufferLayout.h"
```

Ainsi, nous déclarerons tout ce dont nous avons besoin ainsi :

Figure 29 : Attributs de la classe MonTest

```
24     private:
25
26         //pointeur unique sur notre VertexArray
27         std::unique_ptr<VertexArray> m_VAO;
28
29         //pointeur unique sur notre VertexBuffer
30         std::unique_ptr<VertexBuffer> m_VertexBuffer;
31
32         //pointeur unique sur notre IndexBuffer
33         std::unique_ptr<IndexBuffer> m_IndexBuffer;
34
35         //pointeur unique sur notre Shader
36         std::unique_ptr<Shader> m_Shader;
```

- 4) Ensuite, il faut initialiser toutes nos données (nos 4 attributs). On fait cela dans le fichier .cpp (dans le constructeur de notre classe (MonTest())). Tout d'abord on crée nos données brutes (les positions des points et les indices).

Figure 30 : Initialisation des données

```
test::MonTest::MonTest(){
    //Tableau contenant les positions des points (4 pour un rectangle)
    float positions[] = {
        100.0f, 100.0f, //0
        200.0f, 100.0f, //1
        200.0f, 200.0f, //2
        100.0f, 200.0f //3
    };

    //Pour éviter la redondance (créer 2x ou plus le même point) on crée un IndexBuffer (tab d'indice)
    //qui va stocker les indices des points à tracer
    unsigned int indices[] = {
        0,1,2,
        2,3,0
    };
};
```

- 5) Ensuite on initialise les différentes structures de données avec les données :

Figure 31 : Initialisation des structures OpenGL (MonTest.cpp)

```
//On crée le VertexArray
m_VAO = std::make_unique<VertexArray>();

//On stocke nos données (les positions)
//
//                                nb de pts * nombre de floats/pt * size(float)
m_VertexBuffer = std::make_unique<VertexBuffer>(positions, 4 * 2 * sizeof(float));

//Définition de notre layout pour notre VertexArray
VertexBufferLayout layout;
//On utilise 2 floats par points (positions) => layout regroupe ce qui va ensemble
//layout location 0 (cf vertex shader)
layout.Push<float>(2);

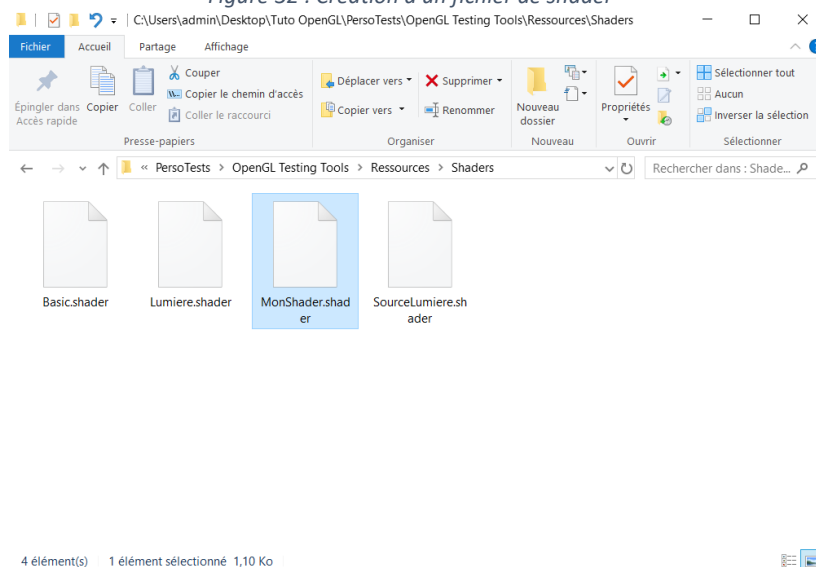
//On regroupe tout dans notre vertex Array (on donne du sens à nos données)
m_VAO->AddBuffer(*m_VertexBuffer, layout);

//On stocke et lie à l'index buffer les indices
m_IndexBuffer = std::make_unique<IndexBuffer>(indices, 6);
```

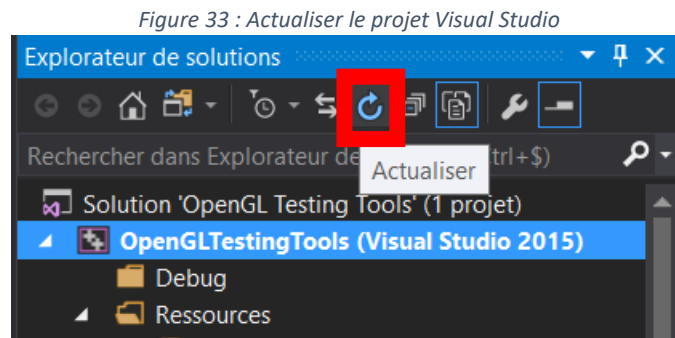
L'instruction `std::make_unique<...>` est juste la façon de créer un `unique_ptr`.

- 6) Ensuite on crée dans le dossier des shaders un fichier texte que l'on appellera MonShader.shader :

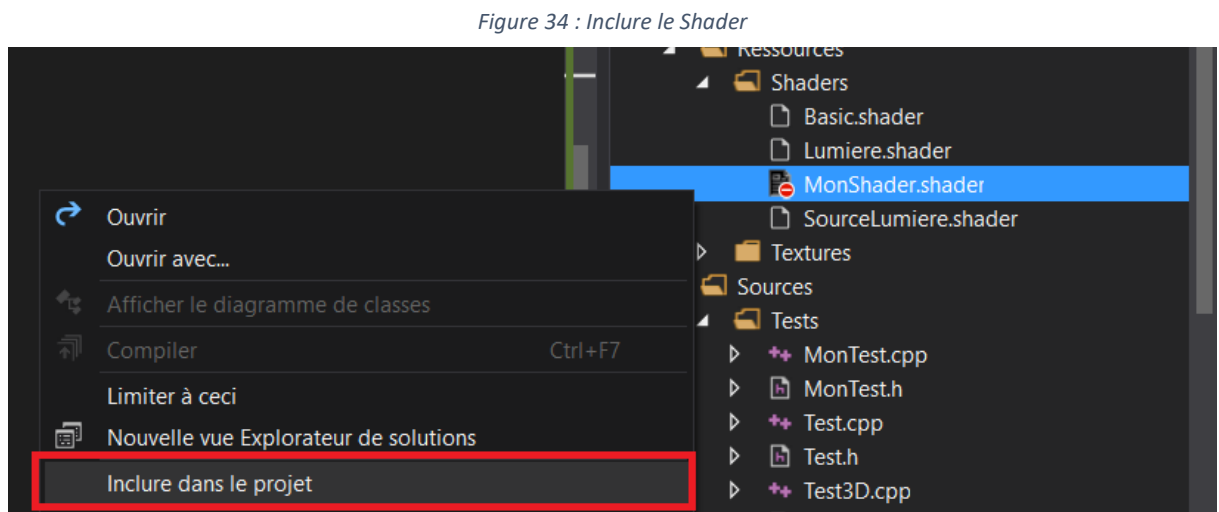
Figure 32 : Création d'un fichier de shader



Ensuite, on actualise le projet visual studio (pour qu'il trouve le fichier nouvellement créé) :



Et on inclut dans le projet le fichier qui vient d'apparaître :



- 7) Ensuite on écrit le corps du shader (ici il va juste récupérer les positions des points et dessiner les objets de la bonne couleur) :

Figure 35 : Corps du Shader (dans MonShader.shader)

```

1  #shader vertex
2  #version 330 core
3
4  //On écrit les shaders
5  //VertexShader ==> exécuté une fois par point
6  //FragmentShader ==> exécuté une fois par pixel
7
8  //on indique quel est l'id et le type d'entrée
9  layout(location = 0) in vec4 position;
10
11 uniform mat4 u_MVP; //ModelViewProjection Matrice => Matrice de projection
12
13 void main() {
14     // !\ L'ordre est important dans un produit Matriciel !\
15     gl_Position = u_MVP * position;
16 }
17
18
19
20
21 #shader fragment
22 #version 330 core
23
24 //Ce qui sort du fragment shader (ici que la couleur à afficher)
25 layout(location = 0) out vec4 color;
26
27 //Ce que l'on prend comme variable (sous forme d'uniform)
28 uniform vec4 u_Color;
29
30 void main() {
31     color = u_Color;
32 }
```

- 8) On va maintenant initialiser le shader et son uniform (sa variable) dans notre application (dans le constructeur de notre Test) :

Figure 36 : Initialisation du Shader (dans MonTest.cpp)

```

38
39 //Création du shader
40 m_Shader = std::make_unique<Shader>("Ressources/Shaders/MonShader.shader");
41
42 m_Shader->Bind();
43
44 //Uniform --> envoi de données du CPU vers les shaders pour être utilisé comme une variable dans le shader
45 //Utilisation uniform après bind avec le shader qui utilise l'uniform
46 //Uniform appelé à chaque dessin (draw call)
47 m_Shader->SetUniform4f("u_Color", 0.0f, 0.0f, 1.0f, 1.0f);
48
49

```

- 9) Il ne nous reste plus qu'à dessiner notre carré : cela se passe dans la fonction OnRender(). Ici tout ce que l'on fait c'est créer un renderer, lui passer en paramètre nos données pour qu'il dessine et initialiser la matrice MVP (Modèle Vue Projection) :

Figure 37 : Boucle de rendu de notre Test (MonTest.cpp)

```

void test::MonTest::OnRender(){

    GLCALL(glClearColor(0.0f,0.0f,0.0f, 1.0f));
    GLCALL(glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT));

    //Ici notre renderer ne stocke rien => on peut se permettre de le créer à chaque frame
    Renderer renderer;

    //Dessin 1
    {
        //Definition d'une matrice du modèle => Considère toutes les transformations
        //aka translation, rotation, scaling
        //à appliquer à notre modèle...
        //Ici on le déplace vers en haut à droite
        glm::mat4 modele = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f));

        //Matrice finale (que l'on va multiplier par chaque point pour qu'il s'affiche correctement)
        // !!\ ORDRE IMPORTANT ICI /\
        glm::mat4 mvp = m_Proj * m_Vue * modele;

        //Seul moyen de ne pas gérer les uniform à la main => materials
        //materials = shader + des données utiles au dessin
        m_Shader->Bind();

        //On envoit la matrice de projection au shader
        m_Shader->SetUniformMat4f("u_MVP", mvp);

        //en tant normal, un renderer prend un vertexArray + IndexBuffer + Materials (pas shader)
        renderer.Draw(*m_VAO, *m_IndexBuffer, *m_Shader);
    }
}

```

- 10) Voilà ! Notre Test est désormais complet ! Tout ce que l'on doit faire c'est l'ajouter au menu de test (dans main.cpp). Pour cela on inclut notre test :

Figure 38 : Inclusion de notre Test (main.cpp)

```

25 #include "Tests/MonTest.h"

```

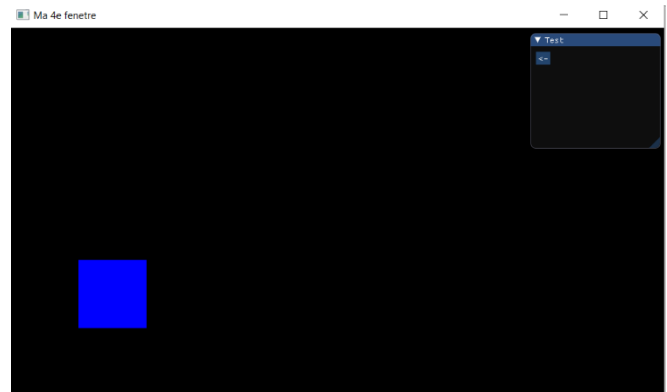
11) On ajoute au menu des tests notre test :

Figure 39 : Ajout du test au menu (main.cpp)

```
130 menuTest->AjouteTest<test::MonTest>("Mon Test");
```

Et voilà le résultat :

Figure 40 : Résultat



Sources

The Chernob

Source : https://www.youtube.com/watch?v=W3gAzLwfIP0&list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2

Description : Ce vidéaste australien (sur YouTube) développe des moteurs de jeu pour EA. Sa série sur les bases d'OpenGL est la source principale de ce tutoriel (jusque l'épisode 26).

learnOpenGL.com

Source : <https://learnopengl.com/>

Description : Site anglophone exhaustif sur OpenGL et sur un ensemble de techniques implémentables.

openGL-tutorial.com

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

Description : Site anglophone simple sur OpenGL.

openclassroom.com

Source : <https://openclassrooms.com/fr/courses/966823-developpez-vos-applications-3d-avec-opengl-3-3>

Description : Didacticiel francophone simple sur OpenGL.