TP N°5
Dessin 2D coloré et transformé

Les shaders :

Nous avons besoin de deux shaders. Un exécuté pour chaque sommet, le Vertex Shader et l'autre exécuté pour chaque fragment, le Fragment Shader.
Les shaders sont programmés dans le GL Shader Language, GLSL, qui est intégré à OpenGL.

Il faut inclure les fichiers contenus dans le dossier « shader » (fichiers fournis) dans votre projet. Le dossier contient le « vertex shader » et le « fragment shader » ainsi que le programme « shader.cpp » qui va les charger, les compiler et les combiner.

1- Le vertex shader :

On commence par la déclaration de la version GLSL, qui correspond à la version OpenGL (ici 3.3) :

```
#version 330 core
```

On déclare les données en entrée (in) qui représentent, dans notre cas, la position des vertex. On utilise le tampon « layout(location=0) » pour définir la localisation de la variable en entrée. L'instruction indique que le tampon a la même valeur que le premier paramètre de la fonction glVertexAttribPointer :

```
layout(location = 0) in vec3 VertexPosition ;
```

Dans la fonction principale, on donne une valeur à la variable prédéfinie gl_Position en utilisant ce qu'il y a dans le tampon :

```
void main()
{
	gl_Position.xyz = VertexPosition ;
	gl_Position.w = 1.0 ;

}
```

2- Le Fragment Shader :

La couleur de chaque fragment est définie à rouge :

```
#version 330 core
out vec3 color;
void main()
{
	color = vec3(1,0,0);
}
```

3- Compilation et appel des shaders :

Ajouter la directive suivante, tout en s'assurant de préciser le chemin dans les propriété du projet :

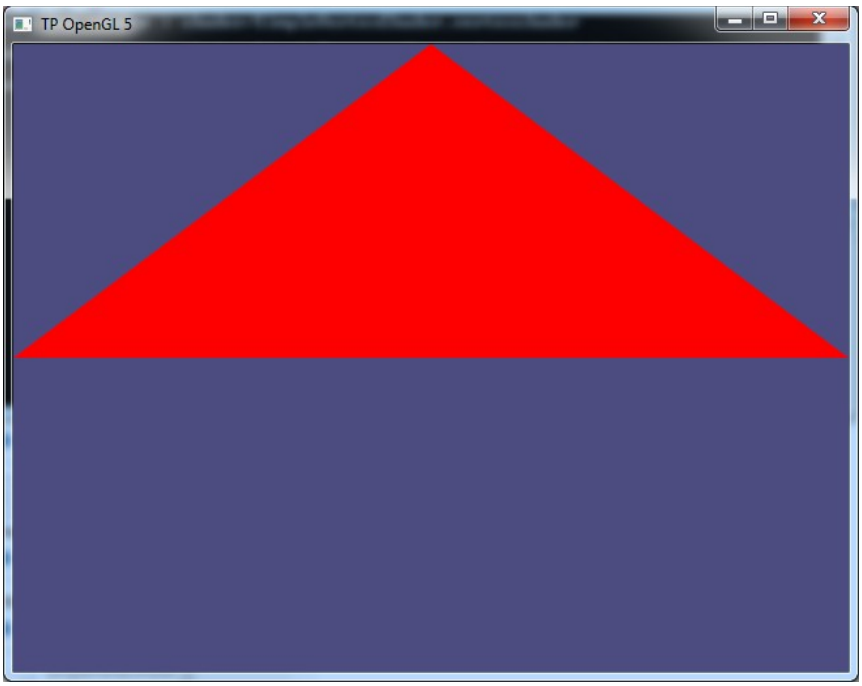
```
#include <shader.hpp>
```

Le programme « shader.cpp » va compiler les deux shader et afficher un message d'erreur le cas échéant. Il va ensuite les combiner pour créer un objet « Shader Program ». Pour cela, il faut appeler la fonction « LoadShader » dans votre main :

```
GLuint ShaderProgram;
ShaderProgram = LoadShaders( "shader/SimpleVertexShader.vertexshader",
"shader/SimpleFragmentShader.fragmentshader" );
```

Afin d'utiliser le Shader Program Object, on ajoute l'instruction suivante au début de la boucle d'affichage :
glUseProgram(ShaderProgram);

On obtient :



Les matrices de transformation

La bibliothèque glm nous permet de créer les matrices nécessaires pour la transformation, la visualisation et la projection.

- Télécharger glm, https://sourceforge.net/projects/ogl-math/
- Ajouter le chemin vers le sous répertoire glm dans Search directories > Compiler

1- La matrice Model:
Il existe dans glm une fonction pour calculer chaque transformation de base :

```
glm::mat4  R  = glm::rotate( glm::mat4(), angle, glm::vec3 () );
glm::mat4  T  = glm::translate(glm::mat4(), glm::vec3());
glm::mat4  S  = glm::scale(glm::mat4(), glm::vec3());
```

2- La matrice View :
La fonction glm:: lookAt permet de calculer la matrice de visualisation afin de positionner la caméra :

```
glm::mat4 ViewMat = glm::lookAt(glm::vec3(), glm::vec3(), glm::vec3());
```

Les paramètres sont dans l'ordre : la position de la caméra, le point de référence de visualisation et en dernier le vecteur view up.

3- La matrice Projection :
Il existe dans glm, deux fonctions glm :: perspective et glm::ortho pour générer les matrices de projection :

```
glm::mat4 ProjectionMat = glm::perspective(angle, ratio , 0.1f , 100.0f );
```

Exemple :

- Ajouter les includes :

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

using namespace glm;
```

- Dans le main de votre projet, avant la boucle d'affichage, réaliser une série de transformations de modélisation, la transformation de la caméra pour la visualisation et la transformation de projection :

```
mat4 Projection = perspective(radians(45.0f), 4.0f / 3.0f, 0.1f, 100.0f);
mat4 View       = lookAt(vec3(2,2,5),vec3(0,0,0),vec3(0,1,0) );
mat4 Model      = mat4(1.0f);
Model           = translate(Model,vec3(-1.0f, 0.0f, 0.0f));
Model           = scale(Model,vec3(2.5f, 1.5f ,1.0f));
Model           = rotate(Model,radians(45.0f),vec3(0.0f,0.0f,1.0f));
```

Ensuite on multiplie le tout :

```
mat4 MVP = Projection * View * Model;
```

On crée une variable uniforme MVP

```
GLuint MatrixID = glGetUniformLocation(ShaderProgram, "MVP");
```

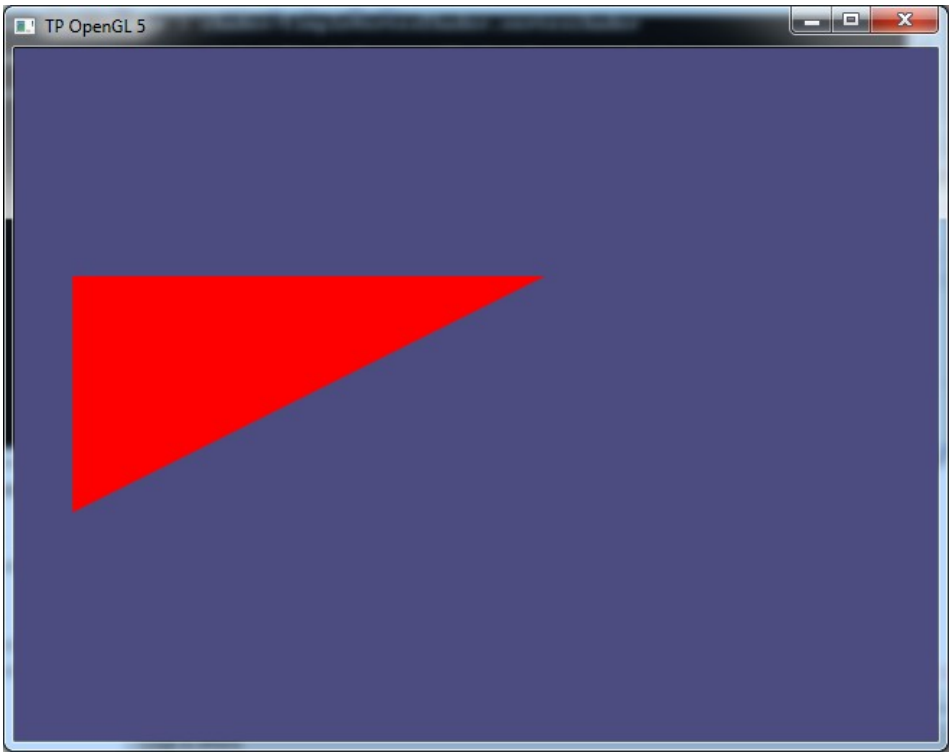
- Envoyer la ou les transformations au shader dans la variable uniforme MVP (pour chaque modèle affiché) dans la boucle d'affichage, avant glDrawArrays :

```
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

- Transformer les sommets dans le vertex shader :

```
uniform mat4 MVP;
gl_Position = MVP * vec4(VertexPosition, 1);
```

On obtient :

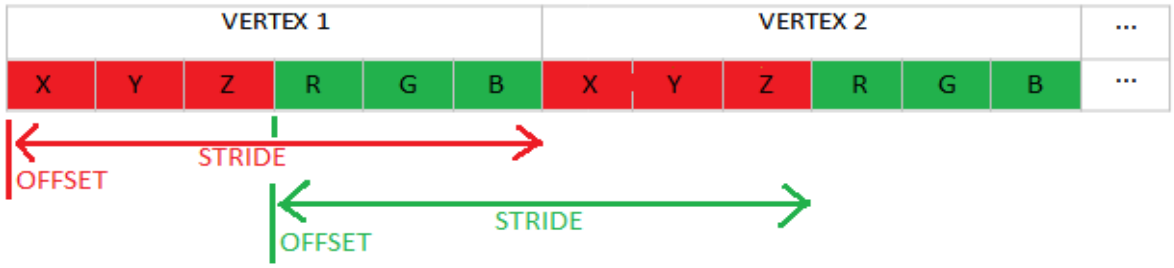


Exercices :

- Appliquer des transformations géométriques sur le triangle tout en positionnant la caméra et appliquant une projection perspectives.
- Nous allons ajouter l'attribut couleur à chaque sommet comme suit :

```
float vertices[] = {
                    1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
                    0.0f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
                    -1.0f,  0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
};
```

Le tableau a désormais la structure suivante :



- Faire la modification nécessaire dans la spécification de l'attribut « position » (5ème paramètre, le stride).
- Ajouter la spécification de l'attribut 1, qui va correspondre à la couleur du vertex (faire attention à l'offset).
- Récupérer l'attribut couleur dans le vertex shader et créer une variable sortie pour le passer au fragment shader.
- Récupérer la sortie du vertex shader comme entrée dans le fragment shader, puis remplacer la couleur rouge, utilisée précédemment, par les couleurs dans l'attribut 1.

On obtient :