TP OpenGL

Ressources

Un bon site pour trouver de la documentation sur les bibliothèques type OpenGL que nous allons utiliser est docs.gl.

Premiers pas

1. Créer un projet C++ dans QTCreator
2. Suivre les étapes ci-dessous pour créer le squelette de l'application.
   On commence par importer les bibliothèques nécessaires :

```
#include <iostream>

#include <GL/glew.h>
#include <GLFW/glfw3.h>
```

Puis ensuite on ajoute les codes suivants dans le main :

Initialisation de GLFW

```
  if(!glfwInit()){
      fprintf(stderr, "Failed to initialize GLFW\n");
      return -1;
  }

  glfwWindowHint(GLFW_SAMPLES, 4); //antialiasing
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); //version 3.3
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); //version
core
```

Création de la fenêtre

```
//définition de la taille de la fenêtre, avec une taille écrite en dur
  int width = 600;
  int height = 600;

  //Enfin on crée la fenêtre
  GLFWwindow* window = glfwCreateWindow(width,height,"OpenGL_API",NULL,NULL);
  glfwSwapInterval(1);

  //On vérifie que l'initialisation a bien marché
  if (window==NULL){
      fprintf(stderr, "Erreur lors de la création de la fénêtre\n");
      glfwTerminate();
      return -1;
  }
```

```
  //Enfin on définit la fenêtre créée comme la fenêtre sur laquelle on
va dessiner
  glfwMakeContextCurrent(window);
```

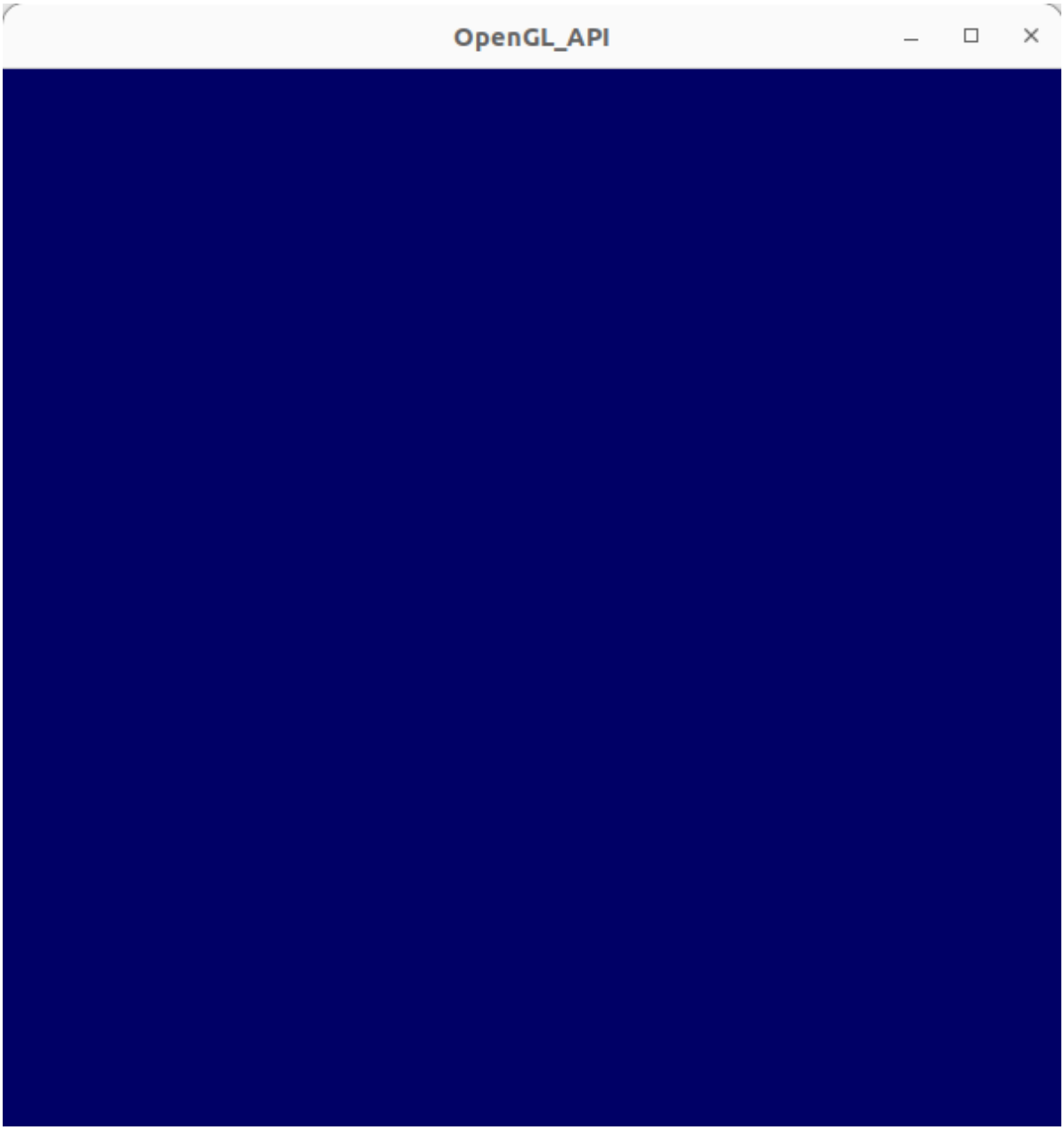
Initialisation de GLEW

```
  //Initialisation de GLEW
  glewExperimental=true;
  if (glewInit() != GLEW_OK){
      fprintf(stderr, "Erreur lors de l'initialisation de GLEW\n");
      return -1;
  }
```

Enfin, on crée la boucle de rendu

```
  // Assure que l'on peut capturer les touche de clavier
  glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

  //On indique la couleur de fond
  glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

  while(glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
 !glfwWindowShouldClose(window)){

      ////////////////On commence par vider les buffers///////////////
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      ////////////////Partie rafraichissement de l'image et des
évènements///////////////

      //Swap buffers : frame refresh
      glfwSwapBuffers(window);
      //get the events
      glfwPollEvents();
  }
  glfwTerminate();
```

Quand on exécute le code, on obtiens une fenêtre vide :



3. Essayez de changer la couleur de fond, puis faite la évoluer au cours du temps grâce à la boucle de rendu.

Bonus :

4. Avoir une fenêtre à taille fixe c'est bien, mais ce qu'on voudrait c'est pouvoir adapter la taille de la fenêtre à l'écran sur lequel on exécute notre application.
   Pour cela on va pouvoir utiliser la fonction de *glfw* `glfwGetMonitorWorkarea`.
   Mais avant d'utiliser cette fonction, il va falloir détecter l'écran en question. C'est ce que fait la fonction `glfwGetPrimaryMonitor`.
   Référez-vous à docs.gl pour voir comment utiliser ces 2 fonctions.

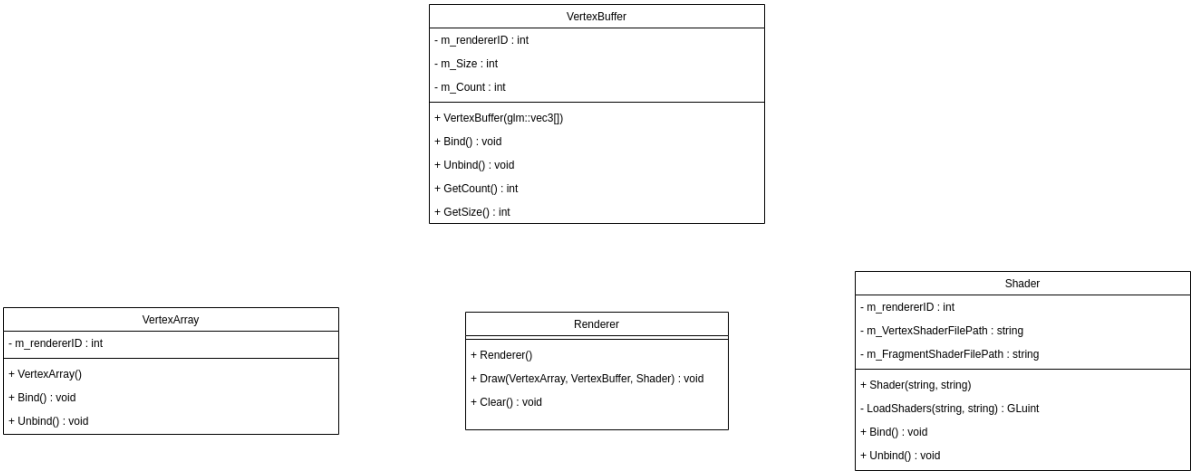
Modifiez le code de création de la fenêtre pour que la taille s'adapte à l'écran.

Dessiner des formes

Désormais on va commencer à afficher des choses dans notre fenêtre. OpenGL ne sait dessiner que des triangles, donc il va falloir découper nos objets en triangles pour les afficher dans notre fenêtre.

Mais chaque chose en son temps. Commençons par afficher un triangle.

Le diagramme ci-dessous présente l'architecture du code de l'application qu'on obtiendra à la fin de cette partie.



Le `VertexArray` est l'objet qui permet de faire le lien entre le CPU et le GPU. On le crée une fois, puis on ne s'en occupe plus.

Le `VertexBuffer` transmettra les informations géométriques au GPU.

Un `Shader` est le code qui tourne sur la carte graphique. Ici on crée une classe Shader qui compile ce code, puis le transmet au GPU.

Enfin le `Renderer` est l'objet que nous utiliserons pour faire les dessins dans la boucle de rendu.

Dans les 3 classes `VertexArray`, `VertexBuffer` et `Shader`, on retrouve les mêmes fonctions `Bind` et `Unbind`, et un attribut `m_rendererID`.

Du côté c++ de notre application, on ne manipule jamais les objets OpenGL directement, on crée nos objets et la fonction de création nous renvoie un entier : **l'identifiant** de notre objet. C'est cet identifiant qui est stocké dans la variable `m_rendererID`.

Une fois les objets OpenGL créés, on doit indiquer à la carte graphique quels objets elle doit utiliser (quel shader, quel vertexbuffer...). C'est ce qu'on appelle le **bind**.

Il se fait différemment selon type d'objet (buffer, shader...), mais le principe est toujours le même :

- On crée l'objet grâce à une fonction GLFW (ex : `glGenBuffers`, `glGenVertexArrays`...). **Cette fonction nous renvoie un entier** : il s'agit de l'index attribué à l'objet OpenGL créé.
- Ensuite on bind l'objet grâce à son index, avec la fonction de bind GLFW appropriée (ex : `glBindVertexArray(index)`).
- Et enfin si nécessaire, on fournit les données contenues par l'objet (le tableau de données pour les buffers par exemple).

Outil de debug

---

5. Commencez par créer une classe `Renderer` (Pour l'instant on ne mets pas les méthodes Draw et Clear).

Dans le fichier *renderer.h* ajoutez le code suivant, en dehors de la classe Renderer :

```
#define ASSERT(x) if (!(x)) raise(SIGTRAP);
#define GLCall(x) GLClearError();\
    x;\
    ASSERT(GLLogCall(#x, __FILE__, __LINE__));

void GLClearError();

bool GLLogCall(const char* function, const char* file, int line);
```

Dans le fichier *renderer.cpp* , ajoutez les fonctions suivantes :

```
void GLClearError()
{
    while (glGetError() != GL_NO_ERROR);
}

bool GLLogCall(const char* function, const char* file, int line){
    while (GLenum error = glGetError()){
        std::cout << "[OpenGL error] " <<error<<" : "<<function<<" "
<<file<<":"<<line<<std::endl;
        return false;
    }
    return true;
}
```

Ici pas besoin de comprendre ce code, il s'agit d'un outil de debug relativement pratique pour voir les erreurs OpenGL.

Il s'utilise sur une ligne de code, de la manière suivante : `GLCall([ligne_de_code]);`

Pour qu'il soit réellement efficace, il faut appeler `GLCall` sur chaque ligne où on fait appel à une commande OpenGL.

Vertex Array

---

6. On va maintenant pouvoir passer aux choses sérieuses, en créant la classe `VertexArray`.

- La fonction GLFW pour créer un VertexArray est `glGenVertexArrays(nombre_arrays, &id)`. Notez qu'on fournit un pointeur vers l'entier qui stockera l'identifiant de l'objet. Cela signifie qu'après avoir appelé la fonction, la variable `id` contiendra l'identifiant OpenGL du buffer créé.
- Pour bind un VertexArray, on utilise la fonction `glBindVertexArray(id)`, et pour l'unbind, on applique cette même fonction en mettant 0 comme valeur d'identifiant.
- Enfin, il ne faut pas oublier de faire un destructeur, qui détruit l'objet OpenGL. La fonction utilisée pour faire ça est glDeleteVertexArrays(nombre_arrays, &id).

Désormais, à vous de créer la classe VertexArray.

7. Une fois la classe implémentée, créez un objet de type VertexArray dans le main, avant la boucle de rendu, et bindez le.

---

8. Maintenant nous allons créer la classe `VertexBuffer`.

   Constructeur

   On commence par initialiser la valeur de `count` (nombre de points) et de `size` (taille du tableau en bits). La fonction `sizeof(object)` permet d'avoir la taille en bits d'un objet.

   Ensuite on initialise l'objet OpenGL :

   - Exactement comme pour un VertexArray, on utilise la fonction `glGenBuffers` pour générer un buffer, et pour le détruire, on utilise `glDeleteBuffers`.
   - La fonction `glBindBuffer(type_buffer, id)` permet de bind le buffer, mais il faut préciser le type de buffer dont il s'agit. Pour nous il s'agit d'un `GL_ARRAY_BUFFER`.
   - Cependant, cette fois ce n'est pas suffisant, il faut insérer les données dans le buffer une fois qu'il est créé. La structure de création est donc la suivante:
     1. Création avec `glGenBuffers`
     2. On bind le buffer avec `glBindBuffer`
     3. On utilise la fonction `glBufferData(type_buffer, taille_tableau , &data[0], GL_STATIS_DRAW)`. La valeur `&data[0]` donne l'adresse mémoire du premier élément du tableau de données, puis ensuite avec la connaissance de la taille du tableau, il lit le reste du tableau et l'ajoute à l'Objet OpenGL.

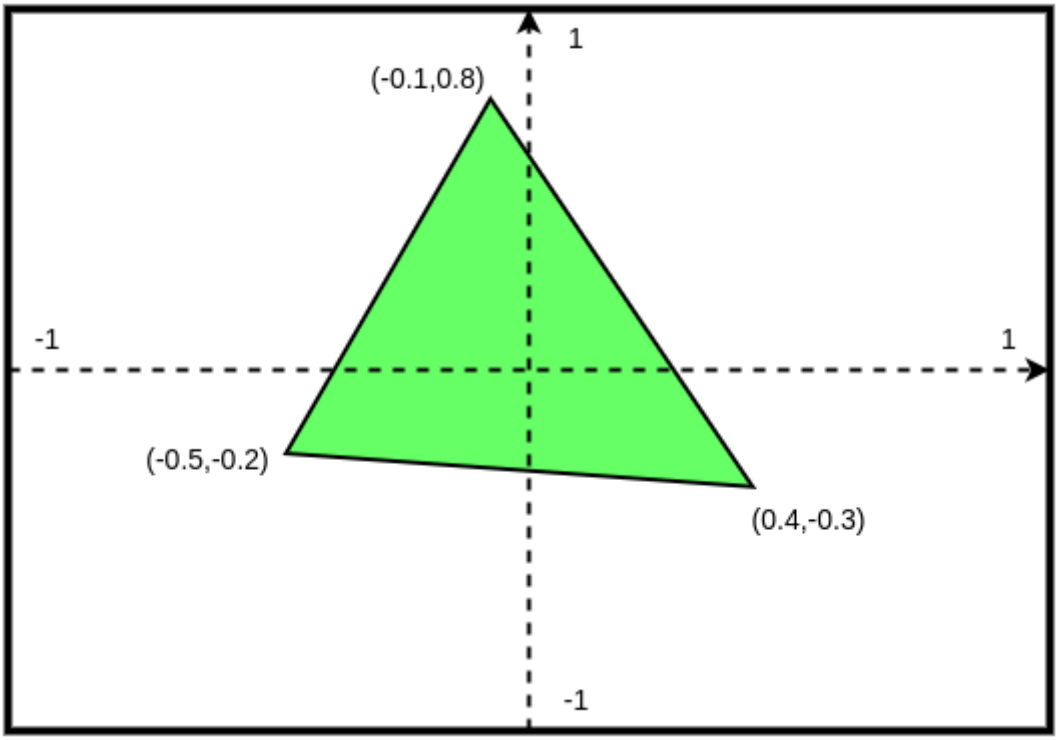
   Fonction Bind

   Pour Bind le VertexBuffer, c'est un peu plus compliqué qu'avec le VertexArray. Voici la marche à suivre :

   1. On appelle la fonction `glEnableVertexAttribArray(index)`, où index est un entier qui va nous permettre de retrouver notre variable côté GPU lorsqu'on créera nos shaders.
   2. Ensuite on appelle la fonction `glBindBuffer`.
   3. Enfin on appelle la fonction `glVertexAttribPointer(index, taille_point, type_valeur ,normalisation, 0, (void*)0)`. L'**index** est le même que précédemment, la **taille d'un point** est le nombre de float que contient un point, donc 3, le **type de valeur** est GL_FLOAT, car on utilise des floats comme coordonnées des points, et la **normalisation** est un booléen qui désigne si on veut normaliser les valeurs, donc GL_FALSE pour nous.

Implémentez la classe `VertexBuffer`.

9. Une fois la classe VertexBuffer implémentée, il va falloir l'utiliser. On va commencer par créer les coordonnées de notre premier triangle dans le main. Pour l'instant, on se place dans le repère de la fenêtre, dans lequel les coordonnées vont de -1 à 1.



Comme on doit donner des coordonnées 3D, on mettra 0 comme 3ème coordonnée pour l'instant.

Pour stocker les coordonnées d'un point, on utilisera la classe `vec3` de la bibliothèque GLM, et pour faire des tableaux, on utilisera la classe `vector`.

Créez un tableau contenant le 3 sommets de notre premier triangle. Une fois le tableau créé, instanciez un `VertexBuffer`.

Shader

---

10. On y est presque ! Mais il faut encore pouvoir dire à la carte graphique quoi faire avec les données qu'on lui envoi. Pour ça, on utilise des shaders. Il s'agit de programmes qui tournent sur la carte graphique. Un shader est écrit en GLSL, puis compilé côté c++ avant d'être envoyé à la carte graphique via un bind.

    Dans la classe Shader, on va donc mettre le code qui permet de compiler les shaders, et le code qui permet de les bind.

    1. Commencez par créer une classe Shader.
    2. Ensuite copiez les codes suivants dans les fichiers header et source de la classe. Il s'agit des codes de compilation, on ne va pas s'intéresser à eux durant ce cours.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <GL/glew.h>
#include <GLFW/glfw3.h
```

```
GLuint LoadShaders(const std::string vertex_file_path,const std::string
fragment_file_path);
```

```
GLuint Shader::LoadShaders(const std::string vertex_file_path,const
std::string fragment_file_path){

    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    // Read the Vertex Shader code from the file
    std::string VertexShaderCode;
    std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
    if(VertexShaderStream.is_open()){
        std::stringstream sstr;
        sstr << VertexShaderStream.rdbuf();
        VertexShaderCode = sstr.str();
        VertexShaderStream.close();
    }else{
        std::cout<<"Impossible to open "<< vertex_file_path <<std::endl;
        getchar();
        return 0;
    }

    // Read the Fragment Shader code from the file
    std::string FragmentShaderCode;
    std::ifstream FragmentShaderStream(fragment_file_path,
std::ios::in);
    if(FragmentShaderStream.is_open()){
        std::stringstream sstr;
        sstr << FragmentShaderStream.rdbuf();
        FragmentShaderCode = sstr.str();
        FragmentShaderStream.close();
    }

    GLint Result = GL_FALSE;
    int InfoLogLength;


    // Compile Vertex Shader
    char const * VertexSourcePointer = VertexShaderCode.c_str();
    glShaderSource(VertexShaderID, 1, &VertexSourcePointer , NULL);
    glCompileShader(VertexShaderID);

    // Check Vertex Shader
    glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
    if ( InfoLogLength > 0 ){
        std::vector<char> VertexShaderErrorMessage(InfoLogLength+1);
        glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL,
&VertexShaderErrorMessage[0]);
        printf("%s\n", &VertexShaderErrorMessage[0]);
    }
```

```
    // Compile Fragment Shader
    char const * FragmentSourcePointer = FragmentShaderCode.c_str();
    glShaderSource(FragmentShaderID, 1, &FragmentSourcePointer , NULL);
    glCompileShader(FragmentShaderID);

    // Check Fragment Shader
    glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(FragmentShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
    if ( InfoLogLength > 0 ){
        std::vector<char> FragmentShaderErrorMessage(InfoLogLength+1);
        glGetShaderInfoLog(FragmentShaderID, InfoLogLength, NULL,
&FragmentShaderErrorMessage[0]);
        printf("%s\n", &FragmentShaderErrorMessage[0]);
    }



    // Link the program
    printf("Linking program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);


    glValidateProgram(ProgramID);

    // Check the program
    glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &InfoLogLength);
    if ( InfoLogLength > 0 ){
        std::vector<char> ProgramErrorMessage(InfoLogLength+1);
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL,
&ProgramErrorMessage[0]);
        printf("%s\n", &ProgramErrorMessage[0]);
    }


    glDetachShader(ProgramID, VertexShaderID);
    glDetachShader(ProgramID, FragmentShaderID);

    glDeleteShader(VertexShaderID);
    glDeleteShader(FragmentShaderID);

    return ProgramID;
}
```

3. La méthode `LoadShaders` fait tout le travail de création de l'objet shader OpenGL, puis renvoie l'identifiant de cet objet.

   Pour ce qui est du bind, il se fait avec la commande `glUseProgram(index)`.

   Enfin, la desrtuction de l'objet OpenGL se fait avec la fonction `glDeleteProgram(index)`.

Implémentez le constructeur, le destructeur et les méthodes Bind et Unbind.

4. Il faut maintenant créer les fichiers qui vont contenir le code des shaders.

   Créez les fichiers `SimpleFragmentShader.fragmentshader` et `SimpleVertexShader.vertexshader`.

   Le vertex shader est exécuté pour chaque point du triangle, tandis que le fragment shader va s'exécuter pour chaque pixel, et renverra la couleur finale pour chacun d'entre eux. De plus, le vertex shader va pouvoir fournir des informations au fragment shader, tel que la couleur du point, sa normale, sa position...

   Copiez les codes ci-dessous respectivement dans le vertex shader et dans le fragment shader.

   Vertex Shader

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

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

   - La première ligne donne la verion d'OpenGL utilisée.
   - La seconde ligne signifie qu'on va chercher les coordonnées des points dans "l'espace mémoire" d'index 0 (cet index correspond à l'index donné question 8.3 lors du bind du vertex buffer). Cela signifie qu'il faudra mettre le buffer des sommets à l'index 0 lorsqu'on fera son bind.
   - Les deux lignes du main permettent d'obtenir la position des vertex côté GPU : on copie la position de la variable côté c++ (vertexPosition_modelspace), et on la met dans une variable côté GLSL (gl_position).

```
#version 330 core

out vec3 color;

void main(){
  color = vec3(1,1,1);
}
```

   - La seconde ligne définit la variable de sortie qui correspond à la couleur du pixel.
   - Dans le main, on donne la valeur "blanc" à cette variable. Donc on aura un triangle blanc si tout se passe bien.

11. Maintenant, créez une instance du la classe shader dans le main.

Renderer

---

12. Plus qu'à implémenter et utiliser les méthodes Draw et Clear de la classe Renderer.

    - La méthode Draw est assez courte, elle bind le shader, le vertex array puis le vertex buffer (dans cet ordre), puis elle fait le dessin de la frame avec la fonction `glDrawArrays(GL_TRIANGLES, first, nb_points)`.

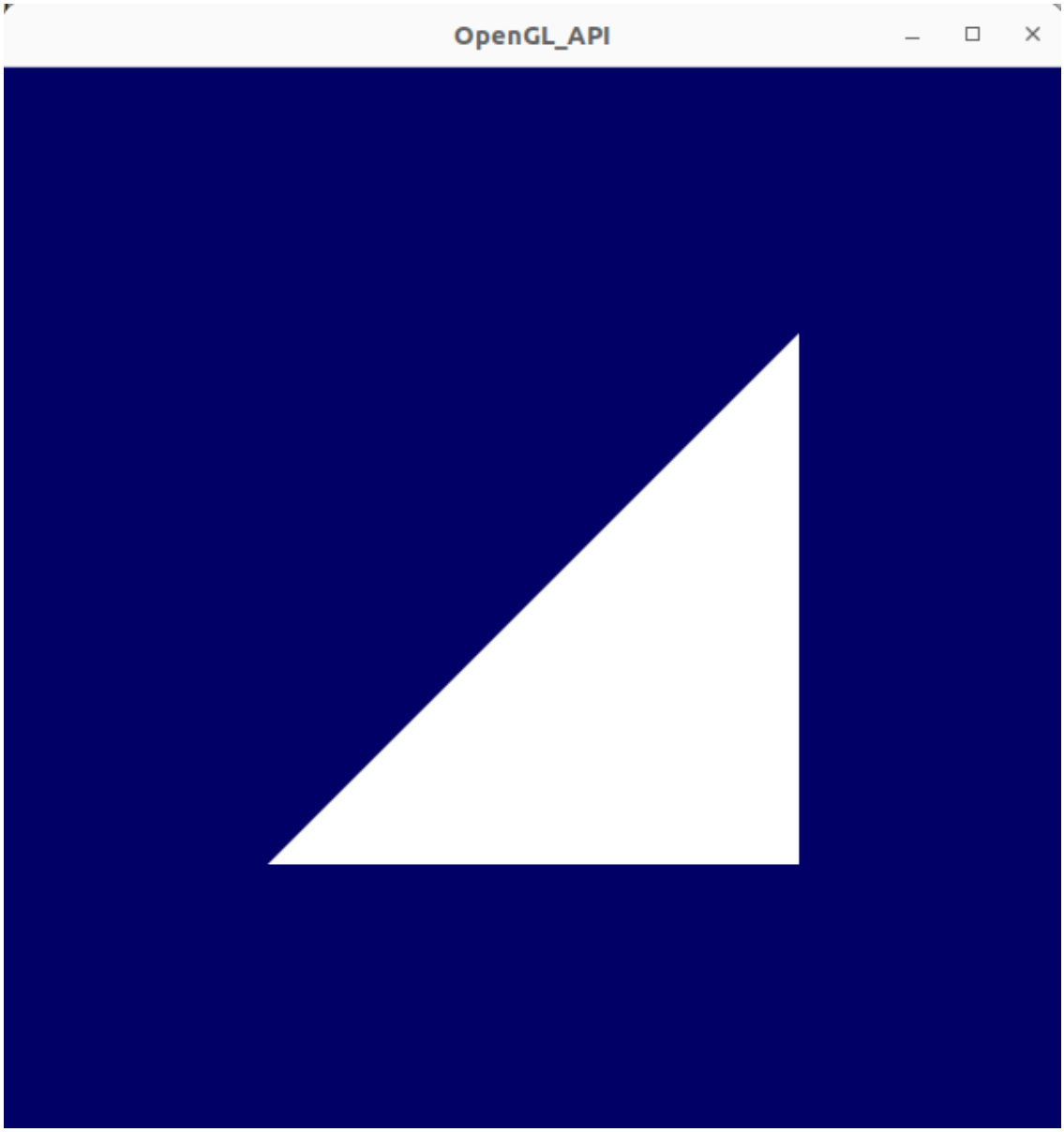
- La fonction Clear est encore plus courte : elle appelle la ligne `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` qui se trouve actuellement dans le main.

Implémentez ces 2 méthodes.

13. Une fois ces méthodes implémentées, il faut les appeler dans le boucle de rendu (d'abord le clear, ensuite le draw).

    Instanciez un objet renderer, puis faites les appels à clear et draw.

    Si tout se passe bien, vous devriez voir apparaître un triangle blanc :



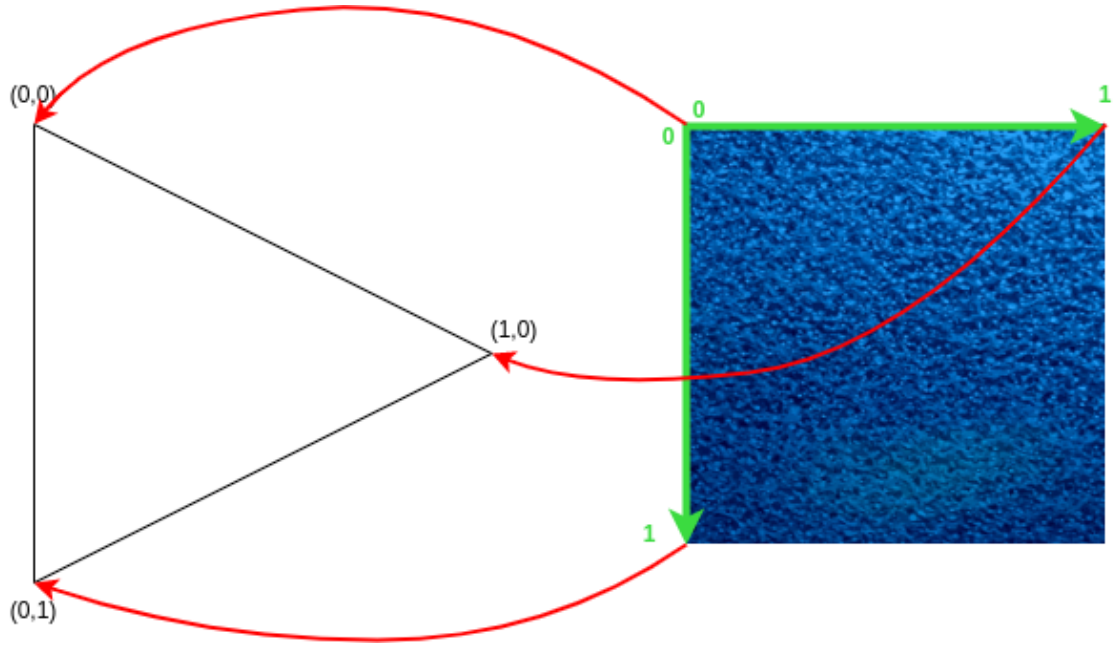
14. Maintenant changez la couleur du triangle.
15. Changez le code pour afficher un carré.

Bonus :

16. Faites tourner le carré.

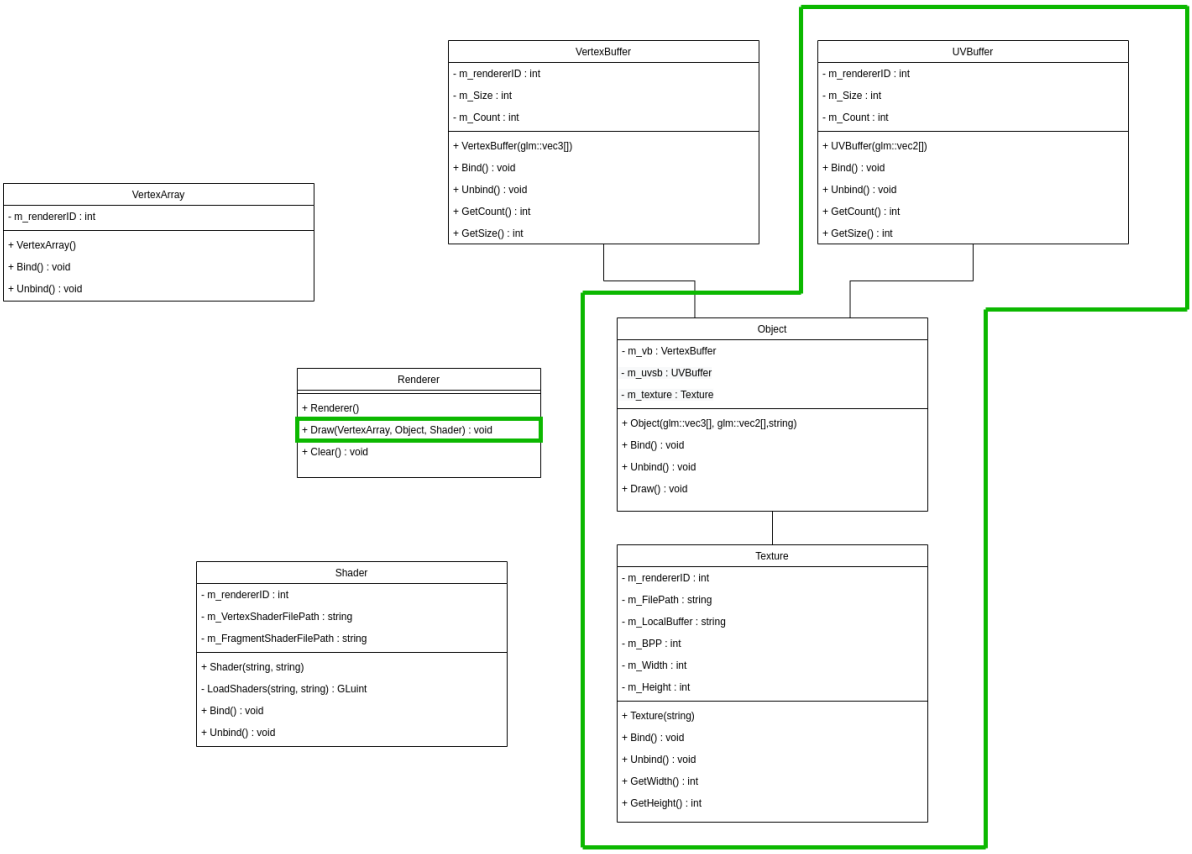
Ajouter une texture

Maintenant nous allons voire comment on applique des textures sur nos objets. Le principe est le suivant : au lieu de définir une couleur uniforme sur nos objets, on veut utiliser une image pour définir la couleur de nos objets. Pour cela, on a besoin de savoir comment appliquer cette image sur notre objet, et on fait cela avec un autre jeu de coordonnées, 2D cette fois, qui correspondent aux coordonnées de l'image qui sont envoyées sur les points de nos triangles (voir figure)



Dans cette partie, on va donc créer une classe texture capable de lire une image et d'appliquer cette image sur une forme, ajouter aux objets qu'on créé des coordonnées 2D pour savoir comment appliquer la texture, et enfin modifier les shaders pour qu'ils prennent en compte la texture.

Voici le diagramme de classe qu'on veut obtenir :



17. Avant de se lancer dans les textures, on va créer une classe `Object` qui regroupera toutes les informations concernant un objet à dessiner : sommets, texture...

- Les arguments du constructeur sont la liste des coordonnées des sommets, la liste des coordonnées 2D, et le chemin de l'image texture.
- La méthode `Bind` fera appel aux méthodes bind des objets VertexBuffer, UVbuffer, etc... qui sont contenus dans l'objet. (Idem pour `Unbind`)
- La méthode `Draw` fera les bind et l'appel à la fonction OpenGL de dessin.
- Il sera préférable d'utiliser des pointeurs pour les attributs de la classe.

En résumé, on regroupe les codes du main et du renderer qui concernent la création et le dessin de l'objet, et on les mets dans la classe `Objet`.

Créez la classe `Object` et implémentez là (Pour l'instant on ne s'occupe pas de `m_usvb` ni de `m_texture`).

18. Créez et implémentez la classe `UVBuffer`. (On pourra s'inspirer de la classe vertexBuffer)
19. Créez la classe Texture.

    Comme il n'est pas important de comprendre comment est créée la texture, on donne le code du constructeur :

```
Texture::Texture(const std::string
&path):m_RendererID(0),m_FilePath(path),m_LocalBuffer(nullptr),m_Width(0),m_H
eight(0), m_BPP(0)
{
    stbi_set_flip_vertically_on_load(1);
    m_LocalBuffer = stbi_load(path.c_str(), &m_Width, &m_Height, &m_BPP,4);

    GLCall(glGenTextures(1,&m_RendererID));
    GLCall(glBindTexture(GL_TEXTURE_2D,m_RendererID));

    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE));

    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, m_Width, m_Height,0,
GL_RGBA, GL_UNSIGNED_BYTE, m_LocalBuffer));
    GLCall(glBindTexture(GL_TEXTURE_2D,0));

    if (m_LocalBuffer){
        stbi_image_free(m_LocalBuffer);
    }
}
```

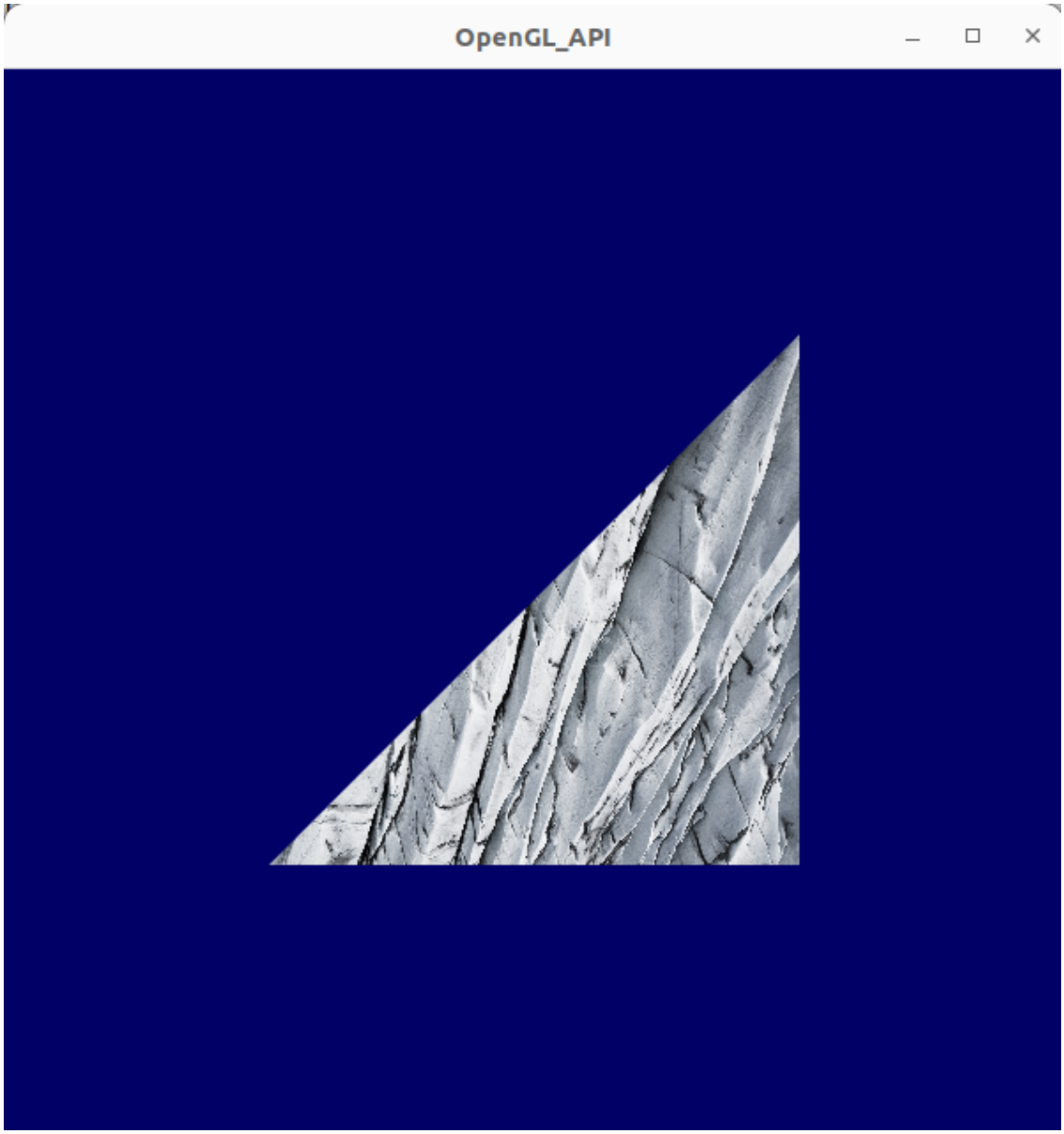
    Attention : le type de l'attribut `m_LocalBuffer` est `unsigned char*`.

    Ensuite il faut implémenter la méthode `Bind`. Pour ça on utilise les fonctions `glActiveTexture(GL_TEXTURE0+slot)` puis `glBindTexture(GL_TEXTURE_2D,id)`. L'argument `slot` correspond à l'emplacement où on mets la texture. C'est utilie quand on utilise plusieurs textures.

    Implémentez la classe `Texture`.

20. Complétez la classe object.

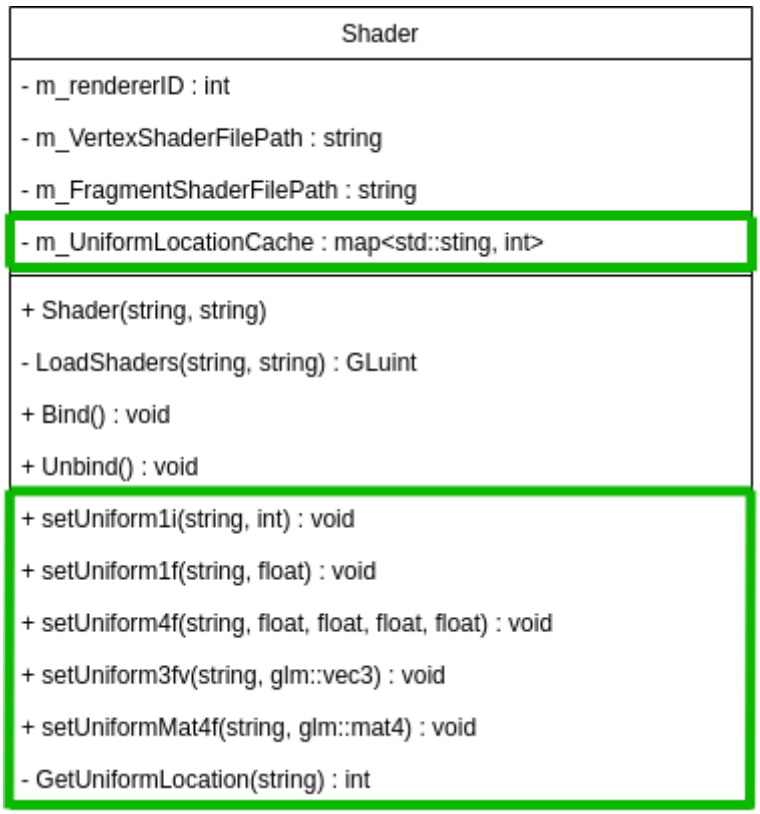
21. Modifiez la classe `Renderer`.
22. Affichez un triangle avec une texture.



Les uniforms

Ici on va apprendre à définir des variables globales côté GPU. Ce sera utile lorsqu'on voudra définir une rotation des objets, un déplacement d'une caméra... tout élément qui ne dépend pas du sommet ou du pixel qui est traité par le GPU.

Ici on ne va toucher qu'à la classe shader. Voici son nouveau diagramme de classes :



On remarque qu'on a créé un certain nombre de fonctions `setUniform...`. Ces fonctions ont pour rôle de générer la variable globale côté GPU, et de lui donner un nom (il s'agit du nom de variable qui sera utilisé ensuite dans le shader).

23. La fonction `GetUniformLocation` permet d'assurer que lorsqu'on appelle plusieurs fois la fonction de création d'un uniform avec un même nom, on ne recrée pas l'uniform à chaque fois, mais on change sa valeur si il existe déjà. Elle a besoin de l'attribut `m_UniformLocationCache` pour fonctionner.

    Comme les détails d'implémentation de cette fonction ne nous intéressent pas ici, on fournit le code directememnt :

    ```
    unsigned int Shader::GetUniformLocation(const std::string &name)
    {
        if (m_UniformLocationCache.find(name)!= m_UniformLocationCache.end()){
            return m_UniformLocationCache[name];
        }
        GLCall(int location = glGetUniformLocation(m_RendererID, name.c_str()));
        if (location == -1){
            std::cout<<"Warning: uniform '"<< name << "' doesn't exist!"
    <<std::endl;
        }
        m_UniformLocationCache[name] = location;

        return location;
    }
    ```

    Collez cette fonction dans votre `shader.cpp`. N'oubliez pas de créer également le header, et créez aussi l'attribut `m_UniformLocationCache`.

24. Pour un type `T`, la fonction OpenGL utilisée pour créer l'uniform de type T est `glUniformT(GetUniformLocation(nom),...,value)`.

Pour les type tableau, comme les vecteurs ou les matrices glm, on mettra `&value[0]` à la place de value.

Implémentez les fonctions `setUniform...`.

25. Quand on crée un uniform côté c++, il faut aussi le récupérer côté GLSL. Cela se fait avec le mot-clé uniform :

```
uniform type name;
void main(){
    type variable = name;
}
```

Maintenant vous avez tout en main pour utiliser des uniforms.

Appliquez un déplacement aléatoire au triangle à chaque frame en utilisant un uniform.

Aide : la fonction qui permet de créer un entier aléatoire est `rand()`. On peut ensuite borner cette valeur de la manière suivante :

```
int r = rand()%10; //entier entre 0 et 9
```

Puis on peut ensuite passer à des floats de la manière suivante :

```
int r = (rand()%11)/10.; //float entre 0 et 1
```

La matrice MVP

Maintenant qu'on sait afficher des objets, qu'on sait les texturer, on voudrait pouvoir les faire bouger, et aussi faire bouger la camera. Le moyen classique pour faire ça est d'utiliser le calcul matriciel, car toutes les transformations de coordonnées (translations, rotations et changement d'échelle) peuvent être exprimées comme une multiplication de matrices. Ici on ne va pas s'intéresser au détail algébrique de ces calculs matriciel, car on va utiliser la bibliothèque GMP pour calculer les matrices dont on a besoin.

Tout ce qu'on a besoin de savoir sur le calcul matriciel, c'est que l'ordre compte : les transformations s'appliquent de droite à gauche. Donc si j'ai une matrice de rotation R et une matrice de translation T que je veux appliquer à des coordonnées C, si je fait $R*T*C$, la translation sera appliquée avant la rotation, alors que si je fais $T*R*C$, la rotation s'applique avant la translation.

Les transformations qu'on va appliquer sont représentées par 3 matrices : la matrice **modèle (M)**, la matrice **vue (V)** et la matrice **projection (P)** .

La matrice Modèle

Il s'agit de la matrice qui nous permet de passer du système de coordonnées de l'objet au système de coordonnées global, qu'on peut appeler système de coordonnées monde.

Quand on donne les coordonnées de nos objets, ils sont centrés en *(0,0)* : ils sont dans le repère centré sur l'objet. On pourrait recalculer les coordonnées à la main lorsqu'on veut bouger l'objet, mais c'est exactement ce que fait la matrice modèle pour nous.

Il y a donc une matrice modèle par objet.

La matrice Vue

Une fois qu'on s'est placé dans le repère monde, on va vouloir déplacer la caméra. Pour ça on va encore changer le système de coordonnées, pour passer du repère monde au repère camera.

L'idée derrière la matrive vue est que appliquer un transformation à la camera est équivalent à appliquer la transformation opposée au monde. Par exemple, lorsqu'on déplace la caméra vers la droite, du point de vue de la camera, on va voir le monde entier se décaler vers la gauche. Idem pour les rotations.

Donc la matrice vue va être la multiplication de l'ensemble des transformations inverses à celles qui s'appliquent à la camera.

La matrice Projection

Il s'agit de la matrice qui permet de passer des coordonnées camera aux coordonnées écran (celles qu'on utilise depuis le début). Elle dépend de la taille de la fenêtre, et du type de camera qu'on veut simuler (vue ortho, camera avec perspective...).

Notre objectif dans cette partie va donc être de mettre en place le calcul et l'utilisation de le matrice *MVP*. Voici le diagramme de classe qu'on vide :

Position de l'objet

26. Modifiez la classe `Objet` pour y faire apparaître la position et l'orientation de l'objet.

27. Maintenant on va calculer la matrice modèle, et pour ça on va utiliser la bibliothèque `glm/gtx/transform.hpp`, qui contient les fonctions permettant d'appliquer des rotations et des translations à des matrices.

    Les commandes utiles sont les suivantes:

    ```
    glm::mat4(1); //Crée la matrice identité
    glm::translate(m, vec3); //renvoie la matrice correspondant à la translation de m par le vecteur vec3.
    glm::rotate(m, a, vec3); ////renvoie la matrice correspondant à la rotation de m d'un angle a autour du vecteur vec3.
    ```

    Avec ces 3 fonctions, codez la fonction `getModelMatrix`.

Création de la Camera

28. Créez la classe `Camera`, et implémentez les fonctions usuelles (laissez la fonction `Bind` vide pour l'instant).
29. Pour la caméra on procède différemment de l'objet : au lieu de recalculer les matrices vue et projection à chaque fois qu'on fait un get, on a une méthode `computeMatrices` qui calcule les matrices et stock le résultat dans les attributs de l'objet, et ensuite on a des getters qui permettent de récupérer ces valeurs.

    Le calcul des matrices de vue et de projection est un peu plus compliqué que celui de de la matrice modèle. On va commencer par la matrice de vue :

    - On utilise la fonction `glm::lookAt(positionCamera, directionVector, upVector)`. On connait déjà la position de la camera, il nous reste à calculer le vecteur donnant la direction dans laquelle regarde la camera et le vecteur "up", qui indique le haut de la camera.

      Le vecteur de direction est le suivant :

      $$direction = \begin{bmatrix} cos(verticalAngle) \times sin(horizontalAngle) \\ sin(verticalAngle) \\ cos(verticalAngle) \times cos(horizontalAngle) \end{bmatrix}$$

      Ensuite pour calculer le vecteur "up", on fait une hypothèse : on suppose qu'on ne va jamais pencher la camera à gauche ou à droite, qu'elle va rester bien droite. (Cette hypothèse se retrouve dans le fait que l'orientation n'est donnée que par 2 angles).

      Alors on définit le vecteur "droite" de la caméra par :

      $$right = \begin{bmatrix} sin(horizontalAngle - \frac{\pi}{2}) \\ 0 \\ cos(horizontalAngle - \frac{\pi}{2}) \end{bmatrix}$$

      Enfin, on peut calculer le vecteur up comme le produit scalaire de *right* et de *direction* (fonction `glm::cross` en c++) :

      $$up = right \cdot direction$$

    - Pour la matrice de projection, on ne va pas rentrer dans le détail. Voici la fonction à utiliser (on va créer une camera avec perspective) :

```
projetcionMatrix = glm::perspective(glm::radians(FoV), width/height,
0.1f, 100.0f);
```

où width et height correspondent aux dimensions de la fenêtre (à donner en argument de la fonction).

Codez la fonction `computeMatrices`.

Utilisation de la matrice MVP

30. Créez une instance de `Camera` dans le main, puis utilisez calculez la matrice MVP.
31. Une fois la matrice MVP créée, il faut l'envoyer aux shaders. Comme la matrice ne change pas lorsqu'on dessine un objet, on peut l'envoyer en tant qu'uniform. Une fois envoyée au shader, il suffit de multiplier les coordonnées d'entrée par la matrice pour obtenir les nouvelles coordonnées :

$$coord_{out} = MVP \times coord_{in}$$

Mettez à jour le shader (le type de MVP en GLSL est mat4), puis faites les modifications nécessaires dans le main.

Aide : Attention, comme *MVP* est une matrice 4*4, et que les coordonnées en sortie ont 4 composantes, il faut que ceux en entrée aient aussi 4 composantes, or ne donne que des points en 3 dimension, donc avec 3 composantes. On doit rajouter un 1 en 4ème composante dans le vertex shader, avec la commande suivante :

```
vec4(vertexPosition_modelspace,1)
```

32. Modifiez votre objet pour qu'il devienne un cube.

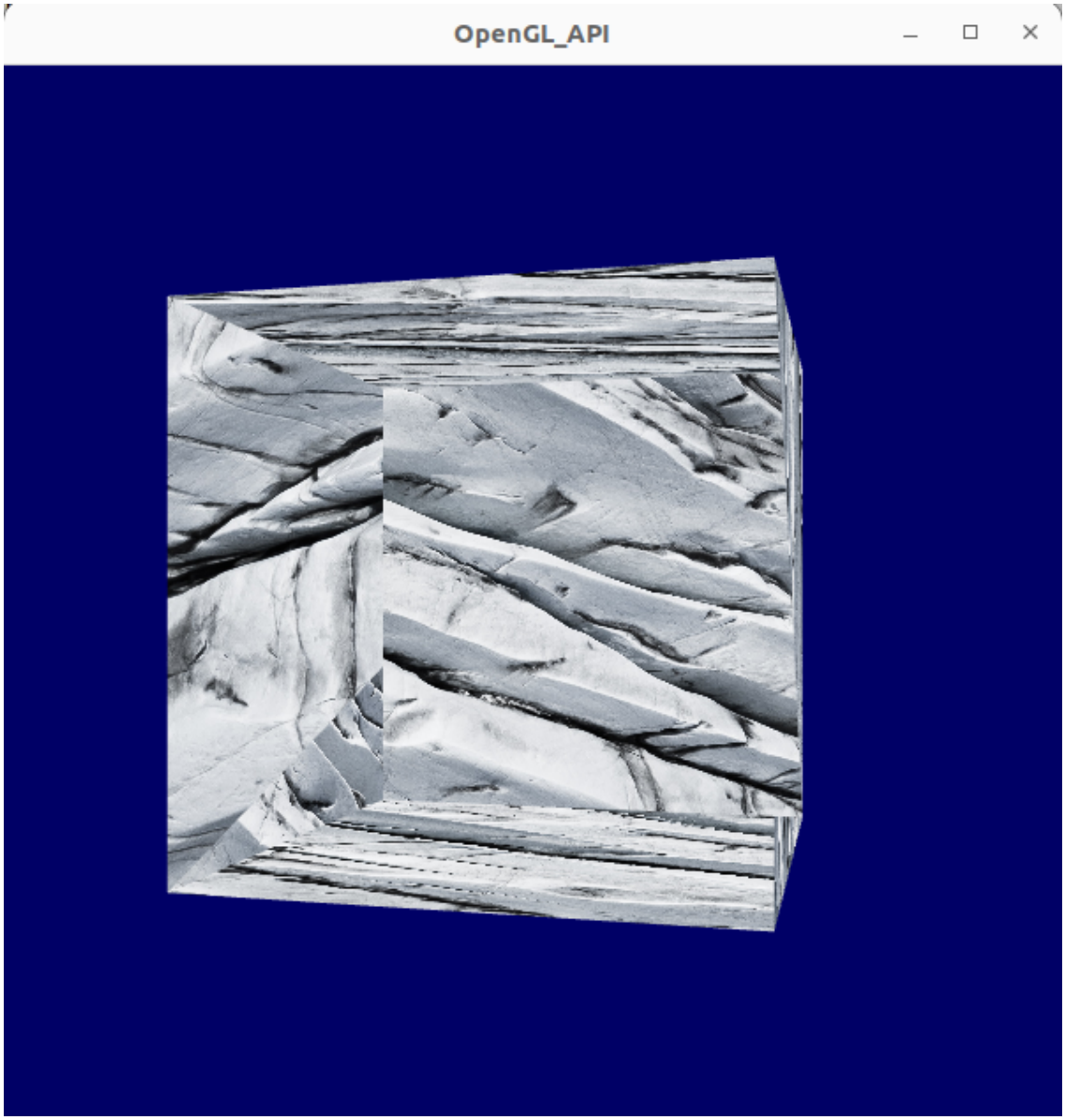
On donne ci-dessous la liste des coordonnées pour éviter les soucis de texture :

```
static const GLfloat g_vertex_buffer_data[] = {
    -1.0f,-1.0f,-1.0f, // triangle 1 : begin
    -1.0f,-1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f, // triangle 1 : end
    1.0f, 1.0f,-1.0f, // triangle 2 : begin
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f,-1.0f, // triangle 2 : end
    1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f,-1.0f,
    1.0f,-1.0f,-1.0f,
    1.0f, 1.0f,-1.0f,
    1.0f,-1.0f,-1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f,-1.0f,
    1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f,-1.0f, 1.0f,
    1.0f,-1.0f, 1.0f,
```

```
    1.0f, 1.0f, 1.0f,
    1.0f,-1.0f,-1.0f,
    1.0f, 1.0f,-1.0f,
    1.0f,-1.0f,-1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f,-1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, 1.0f,-1.0f,
    -1.0f, 1.0f,-1.0f,
    1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f,
    1.0f,-1.0f, 1.0f
};

static const GLfloat g_uv_buffer_data[] = {
    0.000059f, 1.0f-0.000004f,
    0.000103f, 1.0f-0.336048f,
    0.335973f, 1.0f-0.335903f,
    1.000023f, 1.0f-0.000013f,
    0.667979f, 1.0f-0.335851f,
    0.999958f, 1.0f-0.336064f,
    0.667979f, 1.0f-0.335851f,
    0.336024f, 1.0f-0.671877f,
    0.667969f, 1.0f-0.671889f,
    1.000023f, 1.0f-0.000013f,
    0.668104f, 1.0f-0.000013f,
    0.667979f, 1.0f-0.335851f,
    0.000059f, 1.0f-0.000004f,
    0.335973f, 1.0f-0.335903f,
    0.336098f, 1.0f-0.000071f,
    0.667979f, 1.0f-0.335851f,
    0.335973f, 1.0f-0.335903f,
    0.336024f, 1.0f-0.671877f,
    1.000004f, 1.0f-0.671847f,
    0.999958f, 1.0f-0.336064f,
    0.667979f, 1.0f-0.335851f,
    0.668104f, 1.0f-0.000013f,
    0.335973f, 1.0f-0.335903f,
    0.667979f, 1.0f-0.335851f,
    0.335973f, 1.0f-0.335903f,
    0.668104f, 1.0f-0.000013f,
    0.336098f, 1.0f-0.000071f,
    0.000103f, 1.0f-0.336048f,
    0.000004f, 1.0f-0.671870f,
    0.336024f, 1.0f-0.671877f,
    0.000103f, 1.0f-0.336048f,
    0.336024f, 1.0f-0.671877f,
    0.335973f, 1.0f-0.335903f,
    0.667969f, 1.0f-0.671889f,
    1.000004f, 1.0f-0.671847f,
    0.667979f, 1.0f-0.335851f
};
```

33. Faites tourner votre objet.

    A ce niveau vous devriez repérer un problème dans l'affichage de votre cube.

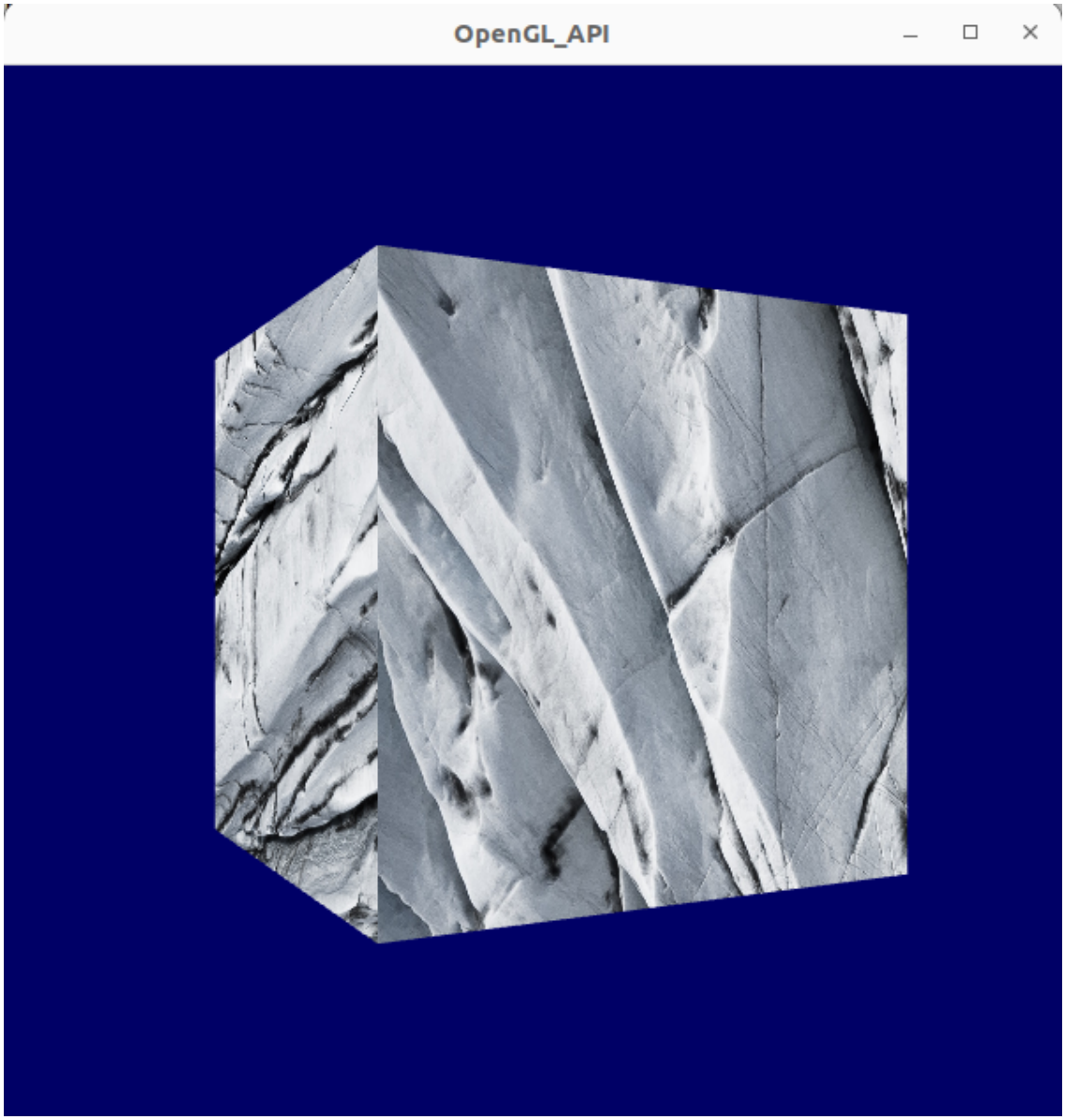


Le problème vient du fait qu'OpenGL ne repère pas quelle face est devant l'autre par défault, et il affiche toute les faces. Il faut donc dire à OpenGL de n'afficher que les surfaces les moins loins.

Pour cela, il faut autoriser les tests de profondeur avec les commandes suivantes :

```
//On autorise les tests de profondeur

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

Réessayez de lancer l'application. Vous devriez obtenir un meilleur résultat :



34. Faites bouger la camera autour de l'objet.

Bonus :

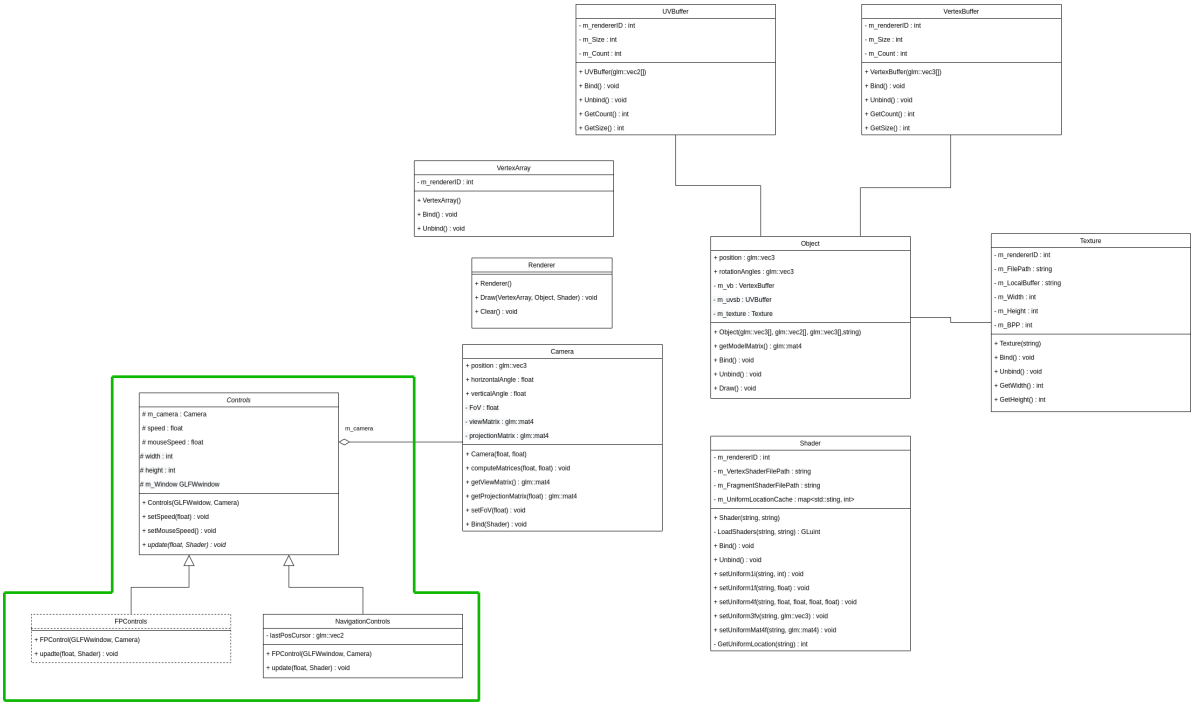
35. Utilisez la matrice MVP pour faire apparaître 2 cubes.

Les contrôles

Il est temps de permettre à l'utilisateur de se déplacer librement dans la scène. Pour ça on va devoir mettre en place une détection des inputs, et contrairement à ce qu'on peut voir généralement, ici on ne va pas passer par des event listeners.

La bibliothèque GLFW a ses propres moyens de détecter les inputs clavier et souris, donc pas besoin d'importer une autre bibliothèque pour ça. L'idée du code est de regarder à chaque passage dans la boucle de rendu quelle touche a été utilisée, et de bouger en conséquence.

Voici le diagramme de classes pour cette partie :



36. Créez et implémentez la classe `Controls`. **Attention**, la méthode `update` est abstraite.
37. Créez la classe `NavigationControls`.

    Le mode de contrôle Navigation est celui dont vous avez l'habitude avec google earth, Itowns, et la plupart des moteurs de navigation 3D géographiques : vous pouvez vous déplacer avec les flèches directionnelles (et avec z,s,q et d), vous pouvez monter avec espace et descendre avec la touche shift, et pour tourner, il faut cliquer sur l'écran et bouger la souris (et relâcher la souris).

    Vous pouvez voir dans le diagramme de classe un autre type de navigation (on l'a mis en pointillé car il est optionnel dans ce cours) : le mode FP (First Person). Dans ce mode de navigation, on cache le curseur. On va supposer que ce mode de contrôles est créée et peut être utilisée.

    Alors lorsqu'on passe du mode FP au mode Navigation, il faut faire réapparaître le curseur. On peut faire ça avec la fonction `glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL)`.

    Ensuite, il faut coder la méthode `update`, dont le rôle va être de regarder quelles touches et quel bouton de souris a été cliqué, et de modifier la position et l'orientation de la caméra en conséquence. Pour pouvoir bouger en avant, en arrière, à droite, en haut, ect... on a besoin de calculer le vecteur qui pointe vers le droite de la caméra, le vecteur qui pointe vers son haut et le vecteur qui pointe devant elle. Or, rappelez-vous, on a déjà vu comment calculer ça dans la partie précédente (cf question 29).

    En plus de ces 3 vecteurs, on va avoir besoin de 2 autres paramètres : `deltaTime`, qui correspond au temps écoulé entre 2 frames, et qui est donné en argument de la méthode, et `speed`, qui est un coefficient déterminant la vitesse de déplacement.

    Une fois qu'on a toutes ces informations, la modification de la position se fera toujours de la manière suivante :

$$position_{new} = position \pm vecteur_{directeur} \times \Delta_{time} \times speed$$

    Par exemple, pour descendre :

$$position_{new} = position - up \times \Delta_{time} \times speed$$

Le test pour tester qu'une touche a été pressée est (exemple pour la flèche vers le haut) :

```
glfwGetKey(m_Window, GLFW_KEY_UP ) == GLFW_PRESS;
```

Codez le déplacement de la caméra avec les touches du clavier dans la méthode `update`.

38. Pour l'utilisation de la souris, le principe global est le même : on test si le bouton gauche de la souris a été cliqué (`GLFW_MOUSE_BUTTON_LEFT`), puis on modifie l'orientation de la caméra si il est cliqué.

    Mais pour savoir comment la souris a bougé, il faut connaître sa position actuelle, et potentiellement sa position à la frame précédente (l'attribut `lastPosCursor` nous servira à stocker cette position). Pour obtenir la position du curseur, on utilise la fonction `glfwGetCursorPos(window, &xpos, &ypos)`, où `xpos` et `ypos` sont entiers que vous devez créer avant. Une fois la fonction appelée, `xpos` et `ypos` contiendront la valeur de la position du curseur sur l'écran.

    Ensuite, on peut mettre à jour l'orientation, mais seulement si la dernière position du curseur est bien définie. On utilise alors la formule suivante :

    $$angle_{new} = angle + speed_{mouse} \times \Delta_{time} \times (x - x_{last}) \text{ ou } (y - y_{last}) \text{ selon l'angle traité.}$$

    On applique cette formule à l'angle horizontal et à l'angle vertical de la caméra. **Et ensuite on mets bien à jour la valeur de `lastPosCursor` !**

    Toujours dans la méthode `update`, codez la partie rotation via la souris.

39. Instanciez un `NavigationControls` dans le main, et utilisez-le pour pouvoir vous déplacer dans le scène.

Bonus :

40. Créez et implémentez la classe `FPControls`.

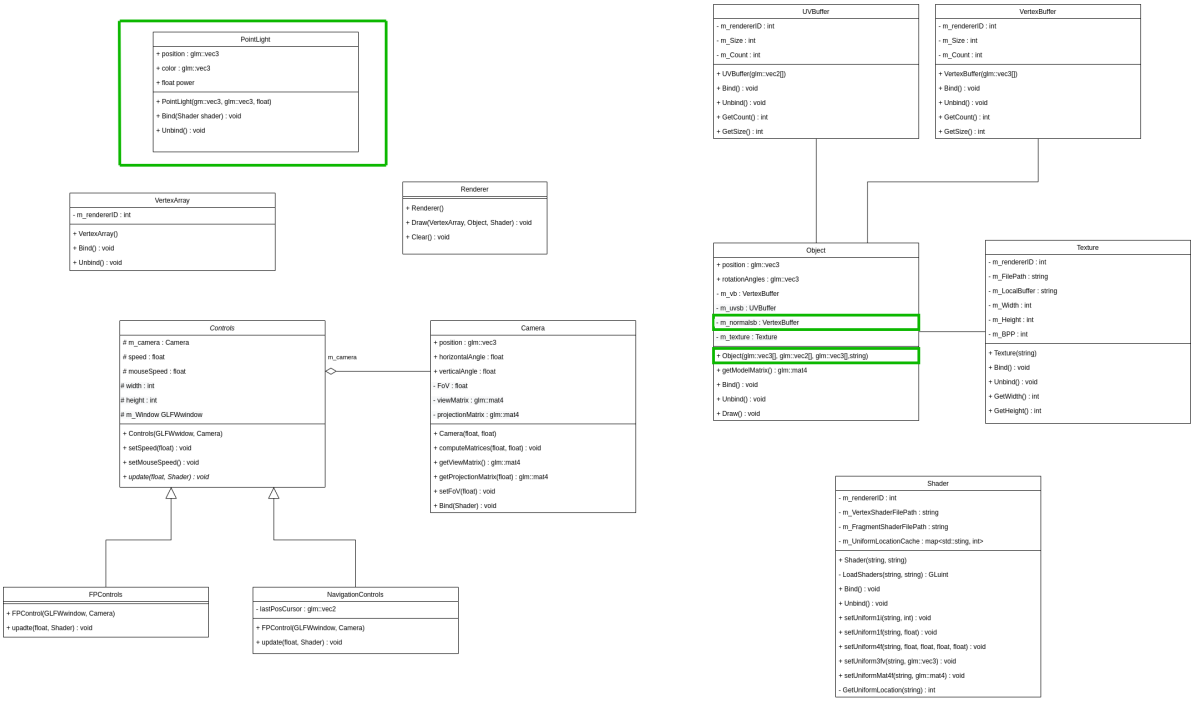
    Ce mode de contrôle marche comme dans un jeu vidéo, on garde les mêmes contrôles au clavier, mais dès qu'on fait bouger la souris, ça fait tourner la caméra. De plus le curseur est masqué, et pour ne pas qu'il sorte de la fenêtre, on doit systématiquement le remettre au milieu de l'écran.

    Pour masquer le curseur, on utilise l'input mode `GLFW_CURSOR_HIDDEN`, et pour positionner le curseur à un certain endroit, on utilise la fonction `glfwSetCursorPos(Window, x, y)`.

Ajouter de la lumière

Les scènes que nous avons rendues jusque là n'ont pas vraiment de lumière, ou plutôt on fait comme si il y avait une lumière ambiante qui éclairait parfaitement chaque zone de nos objets. Dans cette partie, nous allons ajouter la gestion de l'éclairage ambiant, puis la gestion de l'éclairage par une source ponctuelle.

Voici le diagramme de classe de cette partie :



Lumière ambiante

41. Pour créer une lumière ambiante, rien de plus simple : il suffit de créer un vecteur à trois dimensions dans le fragment shader, de lui mettre les valeurs rgb de la couleur de la lumière, puis de multiplier la couleur de l'objet par ce vecteur.

    Créez une lumière ambiante.

Bonus :

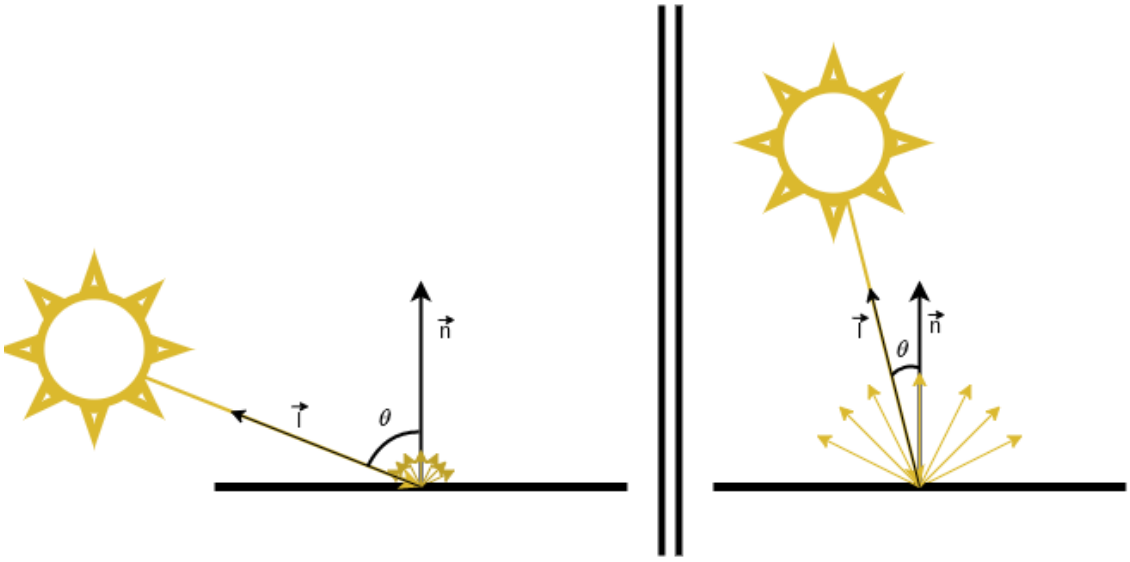
42. Passez cette lumière ambiante en uniform, et faites la varier au cours du temps.

Lumière ponctuelle

Attention : dans le suite, tout les vecteurs sont supposés unitaires (de longueur égale à 1). Cela signifie que dans le code, il faudra bien penser à normaliser tout les vecteurs avant de les utiliser.

Une manière simple de modéliser l'interaction d'un rayon de lumière avec un objet est de dire que lorsqu'un rayon frappe un objet, il est redistribué de différentes manières dans différentes directions.

- La composante diffuse : La majeure partie du rayon est redistribué dans toutes les directions. Plus l'angle avec la normale de l'objet est faible, et plus la composante diffuse de la lumière sera importante :

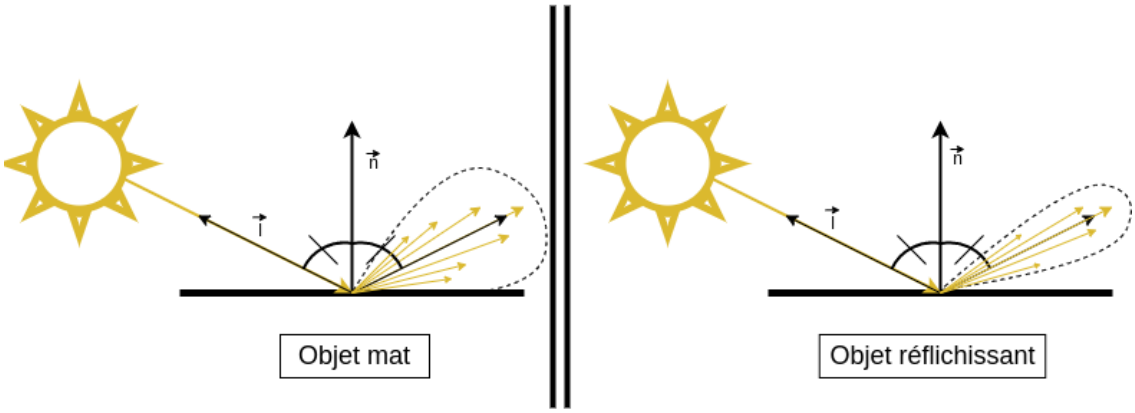


Le coefficient permettant de calculer le taux de lumière réfléchie est le cosinus de l'angle.

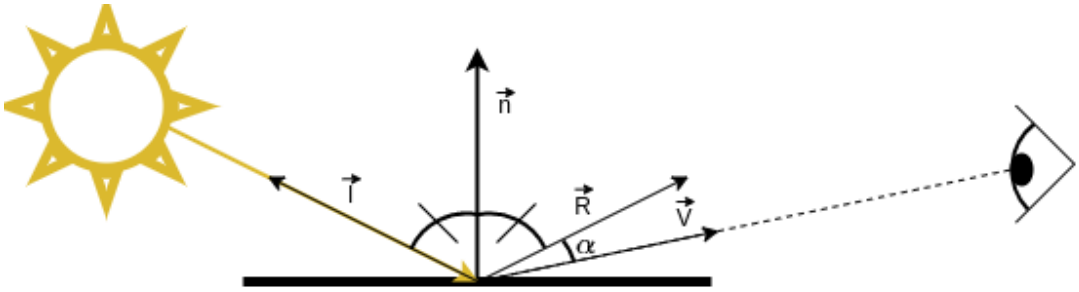
On calcule donc la lumière émise par diffusion de la manière suivante :

$$l_{diffuse} = l_{source} \times color_{object} \times \cos(\theta) = l_{source} \times color_{object} \times (\overrightarrow{l} \cdot \overrightarrow{n})$$

- La composante spéculaire : L'autre partie de la lumière, qui n'est pas renvoyée de manière diffuse, est renvoyée de manière symétrique par rapport à la normale de l'objet (il s'agit d'une réflexion), de manière plus où moins concentrée selon le caractère réfléchissant de l'objet.



Encore une fois, le facteur qui va nous indiquer à quel point la composante spéculaire est forte est un cosinus. Il s'agit du cosinus de l'angle entre le rayon réfléchi et la ligne de vue de la caméra (angle *alpha* sur le schéma), mais mis à une certaine puissance *n* :



Donc au final, la formule pour calculer la composante spéculaire sera :

$$l_{spéculaire} = l_{source} \times color_{object} \times \cos(\alpha)^n \text{ , avec } \cos(\alpha) = \overrightarrow{R} \cdot \overrightarrow{V}$$

Plus le nombre *n* sera fort, plus l'objet sera réfléchissant.

Dans les 2 cas, la lumière qui arrive sur l'objet est plus faible si la source de lumière est loin. On traduit ça en divisant la lumière renvoyée par le carré de la distance entre l'objet et la source :

$$l_{finale} = l_{ambiante} + \frac{l_{diffuse} + l_{spéculaire}}{distance^2}$$

Maintenant qu'on connaît la théorie, il est temps de passer à la pratique.

43. Commencez par créer et implémenter la classe `pointLight`.

    Les 3 attributs de `pointLight` sont constants lors du dessin d'une frame, on les passera donc aux shaders en tant qu'uniforms.

44. On va maintenant modifier les shaders pour ajouter la composante diffuse.

    Le vertex shader

    On donne la liste des variables en entrées, en sortie, et des uniformes utiles pour le vertex shader :

    - *in :* position, coordonnées uv, normales
    - *out :* coordonnées uv, normales, vecteur directeur *objet-lumière* (non normalisé)
    - *uniform :* matrice mvp, matrice m, position de la lumière

      On remarque que dans les uniformes, la matrice modèle est apparue. On va en avoir besoin car les positions et les normales sont dans le système de coordonnées objet, alors que la position de la caméra et la position de la source de lumière sont données dans le repère monde. On va donc utiliser la matrice modèle dans le vertex shader pour replacer les positions et les normales dans le repère monde.

    On va maintenant modifier le vertex shader. En plus de ce qu'il faisait déjà, on doit maintenant calculer les normales et le vecteur objet-lumière, dans le système de coordonnées monde.

    On sait déjà utiliser la multiplication matricielle pour modifier le système des sommets. Ce sera quasiment la même chose pour les normales, mais en mettant 0 au lieu de 1 en quatrième coordonnées (on doit faire cela car la normale est un vecteur, et pas un point). Pour rappel, le code ressemblera à ça :

    ```
    new_coord = matrice * vec4(old_coord, 1);
    new_normal = matrice * vec4(old_normal, 0);
    ```

    Le fragment shader

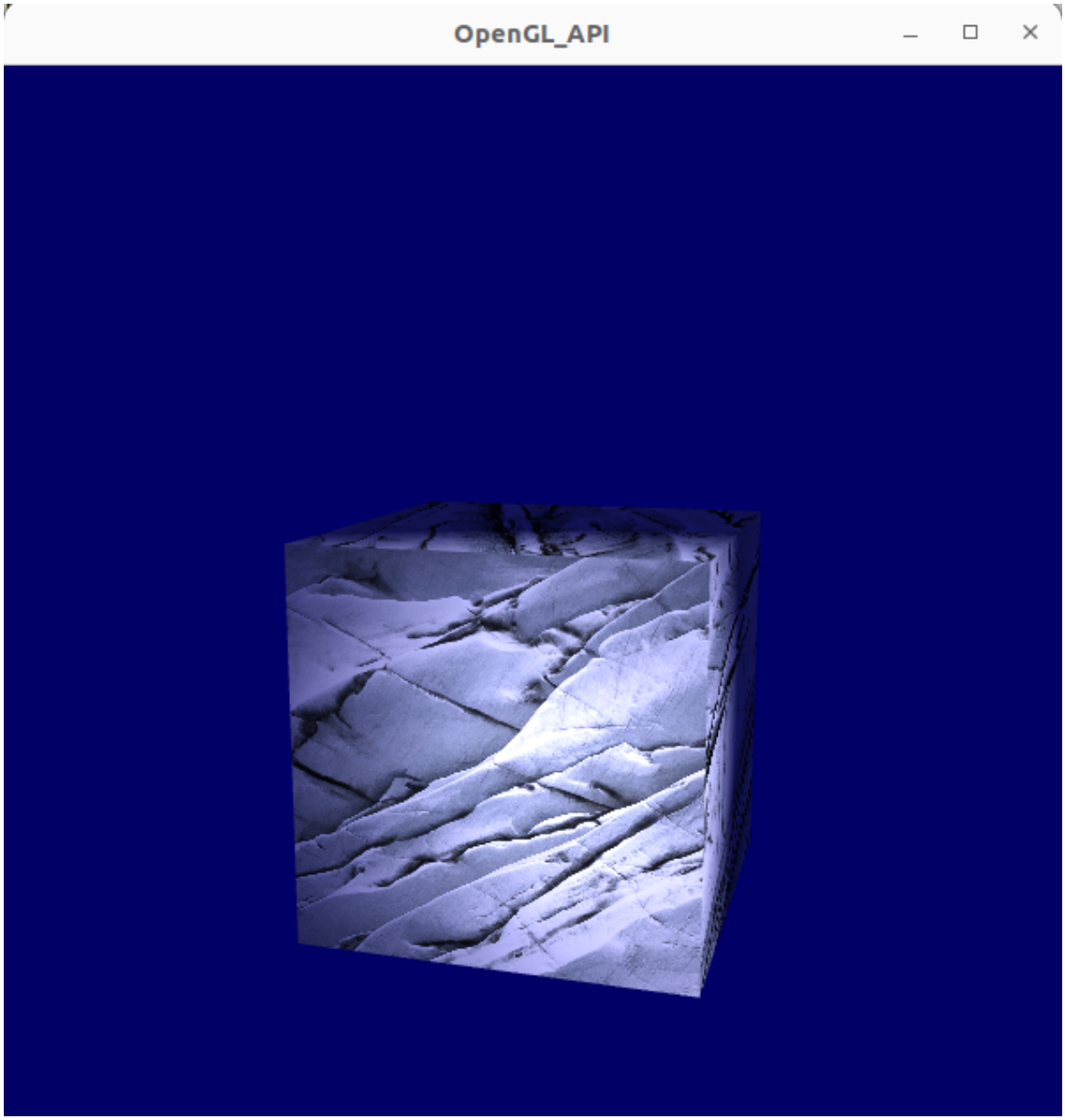
    On va appliquer les formules vues précédemment pour calculer la couleur finale des pixels. Les fonctions utiles sont les suivantes :

    ```
    vec3 vec_unitaire = normalize(vec); //renvoie le vecteur normalisé
    float longueur = length(vec); //renvoie la longueur du vecteur
    float produit_scalaire = dot(vec1,vec2); //renvoie le produit scalaire de
    vec1 et vec2
    float value2 = clamp(value, 0 ,1); //renvoie 0 si value < 0, 1 si value > 1,
    et value sinon
    ```

On utilisera la fonction clamp pour s'assurer que les cosinus calculés ne seront pas négatifs.

Implémentez le calcul de la composante diffuse dans les shaders.

Voici le résultat qu'on obtient :



45. On va faire pareil avec la composante spéculaire.

    Vertex shader

    On a un nouvel uniform : la position de la caméra, et une nouvelle valeur en sortie : le vecteur objet-caméra.

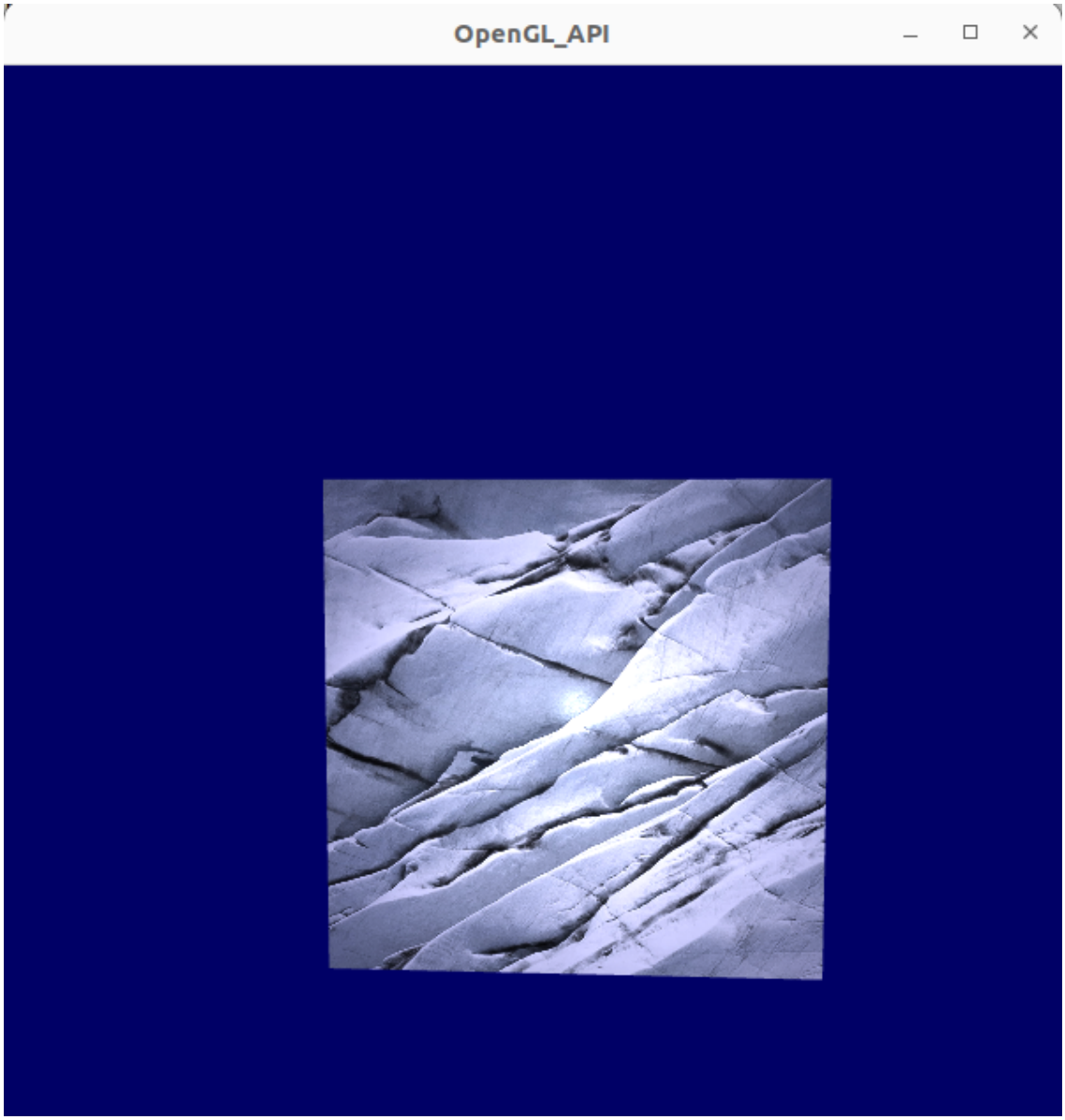
    Fragment shader

    On applique également la formule vue précédemment. On pourra utiliser les fonctions suivantes :

```
vec3 r = reflect(vec1,vec2); //renvoie le symétrique de vec1 par rapport à
vec2
float value2 = pow(value,n); //renvoie value^n
```

    Implémentez le calcul de la composante spéculaire dans les shaders. On pourra mettre une valeur de 5 pour la puissance du cosinus.

Voici le résultat qu'on obtient :



Bonus :

46. Faites en sorte de pouvoir modifier le caractère réfléchissant d'un objet.

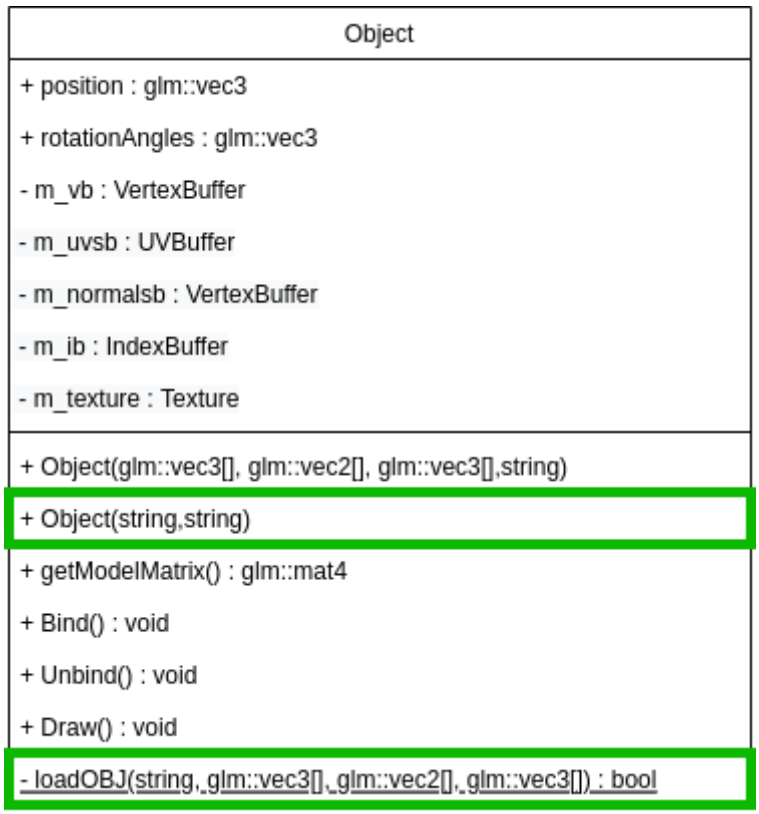
Lire un fichier .obj

Bon, pour dessine juste un cube, un cube joliment décoré et éclairé, mais un cube quand même, ce n'est pas très intéressant... Ce qu'on voudrait c'est être capable d'afficher de vrais objets, comme une voiture, une personne, une maison...

Mais on ne va pas écrire à la main les coordonnées, les normales et les coordonnées UV de tels objets dans le code, ce serait beaucoup trop gros et beaucoup trop long à faire. Il existe des logiciels qui permettent de faire ça et dessinant les objets, puis qui stockent le résultat dans des fichiers, et des personnes qui mettent en ligne leurs fichiers. On va donc en profiter et utiliser des fichiers déjà existant pour afficher des objets plus complexes.

Les fichiers qui stockent ces informations sont des fichiers `.obj`. On ne va pas détailler leur structure ici, tout ce qui nous intéresse c'est de savoir que ces fichiers existent et qu'on peut les intégrer dans notre code.

Voici les modifications du diagramme de classes de cette partie :



Au final, on va ajouter une fonction qui lit les fichiers obj et les convertit en tableaux, et un constructeur qui appellera cette fonction.

51. On fournit le code pour lire les fichiers .obj :

```
bool Object::loadOBJ(const char *path, std::vector<glm::vec3> &out_vertices,
std::vector<glm::vec2> &out_uvs, std::vector<glm::vec3> &out_normals)
{
    std::vector< unsigned int > vertexIndices, uvIndices, normalIndices;
    std::vector< glm::vec3 > temp_vertices;
    std::vector< glm::vec2 > temp_uvs;
    std::vector< glm::vec3 > temp_normals;

    FILE * file = fopen(path, "r");
    if( file == NULL ){
        printf("Impossible to open the file !\n");
        return false;
    }

    while( 1 ){

        char lineHeader[128];
        // read the first word of the line
        int res = fscanf(file, "%s", lineHeader);
        if (res == EOF)
            break; // EOF = End Of File. Quit the loop.

        // else : parse lineHeader
        if ( strcmp( lineHeader, "v" ) == 0 ){
            glm::vec3 vertex;
            fscanf(file, "%f %f %f\n", &vertex.x, &vertex.y, &vertex.z );
            temp_vertices.push_back(vertex);
        }
```

```
        else if ( strcmp( lineHeader, "vt" ) == 0 ){
            glm::vec2 uv;
            fscanf(file, "%f %f\n", &uv.x, &uv.y );
            temp_uvs.push_back(uv);
        }
        else if ( strcmp( lineHeader, "vn" ) == 0 ){
            glm::vec3 normal;
            fscanf(file, "%f %f %f\n", &normal.x, &normal.y, &normal.z );
            temp_normals.push_back(normal);
        }
        else if ( strcmp( lineHeader, "f" ) == 0 ){
            std::string vertex1, vertex2, vertex3;
            unsigned int vertexIndex[3], uvIndex[3], normalIndex[3];
            int matches = fscanf(file, "%d/%d/%d %d/%d/%d %d/%d/%d\n",
&vertexIndex[0], &uvIndex[0], &normalIndex[0], &vertexIndex[1], &uvIndex[1],
&normalIndex[1], &vertexIndex[2], &uvIndex[2], &normalIndex[2] );
            if (matches != 9){
                printf("File can't be read by our simple parser : ( Try
exporting with other options\n");
                return false;
            }
            vertexIndices.push_back(vertexIndex[0]);
            vertexIndices.push_back(vertexIndex[1]);
            vertexIndices.push_back(vertexIndex[2]);
            uvIndices.push_back(uvIndex[0]);
            uvIndices.push_back(uvIndex[1]);
            uvIndices.push_back(uvIndex[2]);
            normalIndices.push_back(normalIndex[0]);
            normalIndices.push_back(normalIndex[1]);
            normalIndices.push_back(normalIndex[2]);
        }
    }

    for( unsigned int i=0; i < vertexIndices.size(); i++ ){
        unsigned int vertexIndex = vertexIndices[i];
        glm::vec3 vertex = temp_vertices[ vertexIndex-1 ];
        out_vertices.push_back(vertex);
    }
    for( unsigned int i=0; i < uvIndices.size(); i++ ){
        unsigned int uvIndex = uvIndices[i];
        glm::vec2 uv = temp_uvs[ uvIndex-1 ];
        out_uvs.push_back(uv);
    }
    for( unsigned int i=0; i < normalIndices.size(); i++ ){
        unsigned int normalIndex = normalIndices[i];
        glm::vec3 normal = temp_normals[ normalIndex-1 ];
        out_normals.push_back(normal);
    }

    return true;
}
```

Ajoutez la fonction `loadOBJ`.

52. Créez le constructeur de la classe `Object` qui prend en argument un chemin vers un fichier obj au lieu de tableaux de données.
53. Avec notre code, on est tout de même limités à utiliser une texture par fichier obj, or en général on plusieurs textures pour un seul objet.

    Donc pour ce TP, on fournit des fichiers dans le dossier obj.

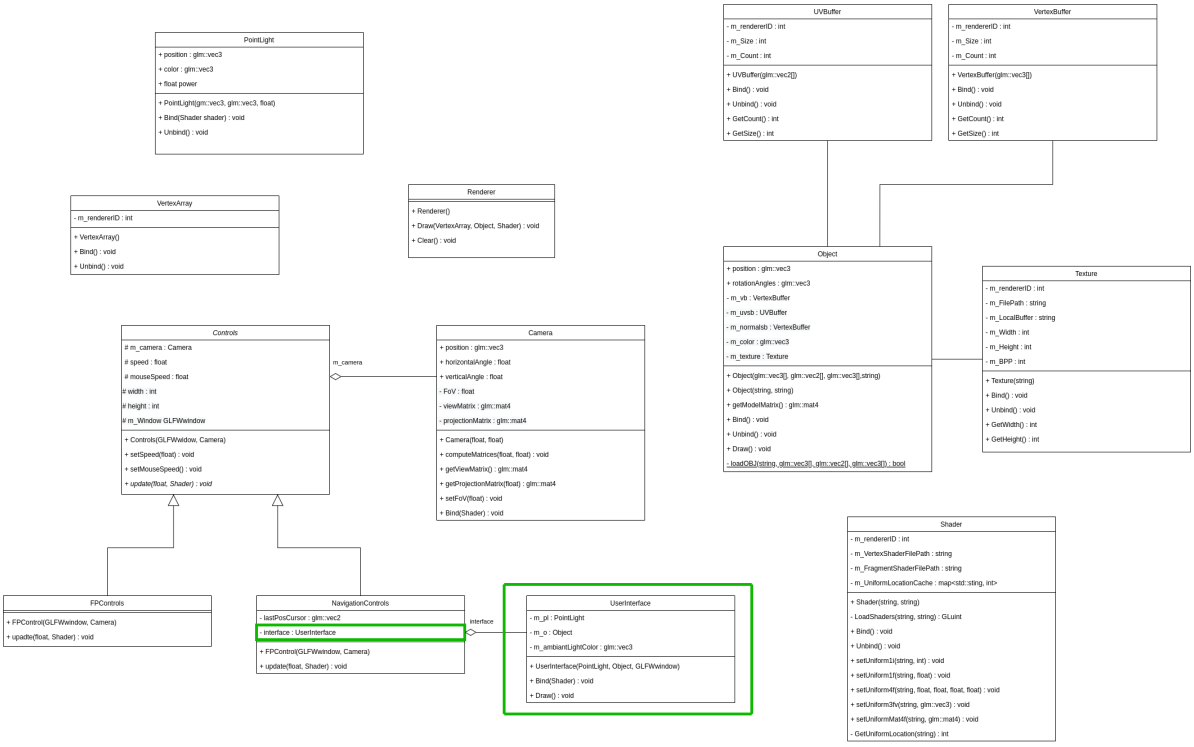
    Affichez des objets dans votre scène.

Ajouter une interface utilisateur

Dans cette partie on va ajouter une interface utilisateur lorsqu'on se trouve en mode navigation, interface qui permettra de modifier en live des valeurs telles que la couleur de la lumière, sa position, la rotation de l'objet... toutes les valeurs qu'on utilise dans notre code pour décrire la scène en fait.

Pour ça, on va utiliser une bibliothèque appelée ImGui.

Voici le diagramme de classes finale



54. Allez cloner le dépôt ImGui, puis supprimez les fichiers suivants :
55. Pour cette partie vous serez plus en autonomie, vous connaissez le résultat auquel on veut arriver, et la structure du code. Maintenant c'est à vous de coder tout ça, en vous aidant de la liste des fonctions de ImGui utiles que je mets ci-dessous :

```
ImGui::CreateContext;
ImGui_ImplGlfw_InitForOpenGL;
ImGui_ImplOpenGL3_Init;
ImGui::StyleColorsDark;
ImGui_ImplOpenGL3_Shutdown;
ImGui_ImplGlfw_Shutdown;
ImGui::DestroyContext;
ImGui_ImplOpenGL3_NewFrame;
```

```
ImGui_ImplGlfw_NewFrame;
ImGui::NewFrame;
ImGui::ColorEdit3;
ImGui::SliderFloat3;
ImGui::Text;
ImGui::Render;
ImGui_ImplOpenGL3_RenderDrawData;

//Et si besoin, pour détecter si la sourie est sur la fenêtre ImGui ou non :
ImGuiIO& io = ImGui::GetIO();
io.AddMouseButtonEvent(1, 0);

if (io.WantCaptureMouse){...}
```

Codez la classe `UserInterface`, puis faites les modifications nécessaires dans la classe `Naigation control` pour afficher l'interface utilisateur.

Attention : Lorsqu'on clique dans l'interface utilisateur pour modifier des valeurs, on ne veut pas que la caméra tourne avec elle...