Faculté des Sciences & Techniques — Université de Limoges

Projet de Moteur 3D

Compte Rendu

Élèves :
Mathurin Cartron

Professeur :
Maxime Maria

Table des matières

Liste des tableaux

Table des figures

Listings

1 Introduction

Le but de ce projet est de mettre en œuvre les différentes méthodes vues en cours et en TP, au travers d'une application de visualisation interactive de scènes 3D. L'objectif est d'implémenter des méthodes de rendu temps-réel pour réaliser des effets plus ou moins complexes. Parmi ces effets, on peut citer le light culling, le forward+, le calcul des ombres portées (shadow mapping, shadow volume), l'SSAO, l'FXAA et le bloom. Chacun de ces effets présente des défis spécifiques qui seront abordés dans le cadre de ce projet.

Une longue absence En raison d'une longue absence dont vous connaissez la raison, j'ai pris un lourd retard dans mon travail. Par conséquent, je me contenterai de rendre le travail des TPs malgré les efforts que j'ai pu fournir pour le rattraper. Bien que cela ne couvre pas l'ensemble de ce qui était prévu pour ce projet, je suis conscient de l'importance de rendre quelque chose de complet et de bonne qualité. Je ferai de mon mieux pour atteindre cet objectif en privilégiant une bonne compréhension de notion qu'un projet entier mais baclé.

Guacamole Malheureusement, j'ai également été confronté à un autre obstacle qui a encore aggravé mon retard : la panne de "guacamole" durant les vacances de Noël. Cette panne m'a empêché de travailler pendant les vacances et a considérablement affecté ma progression. Malgré tout, j'ai fait de mon mieux pour rattraper mon retard à la rentrer, même si je ne parviens pas à couvrir tous les aspects prévus pour ce projet.

De l'aide de mes camarades Heureusement, mes camarades m'ont apporté leur soutien et m'ont aidé à surmonter ces difficultés. Grâce à leur assistance, j'ai pu avancer dans mon travail et commencer à mettre en œuvre les différentes méthodes vues en cours et en TP. Cependant, je dois admettre que, malgré ma volonté de livrer un travail original et indépendant, il y a probablement plus de similarité entre le travail de mes camarades et le mien qu'il n'y en aurait eu sans leur intervention. En effet, ils ont dû déboguer mon code en se servant du leur, ce qui a inévitablement laissé des traces dans mon travail final. Je tiens à préciser que, malgré ces circonstances difficiles, j'ai fait de mon mieux pour assurer l'originalité de mon travail et éviter toute forme de plagiat. Si, malgré mes efforts, il y a des parties de mon travail qui présentent une forte similarité avec celui de mes camarades, je m'engage à le signaler explicitement dans les fichiers code et à préciser les contributions de chacun.

2 Contenu

2.1 Architecture des TP

Au début du TP1, nous avons recu un projet qui allait évoluer jusqu'à la fin (TP6). L'objectif était de ne disposer que d'un seul projet Visual Studio, en utilisant un système de lab_works pour gérer les différentes étapes du projet global.

Le lab_works était organisé de manière à séparer les différents éléments du projet en plusieurs répertoires, chacun ayant un rôle précis :

- Le répertoire "src" contenait le code source du projet, organisé en plusieurs sous-répertoires selon leur fonction (par exemple, "core", "graphics", "input", etc.).
- Le répertoire "lib" contenait les bibliothèques externes utilisées par le projet (par exemple, "glew", "glfw", etc.).
- Le répertoire "data" contenait les données utilisées par le projet (par exemple, les modèles 3D, les textures, etc.).

En utilisant cette architecture, nous avons pu travailler de manière organisée et efficace sur chaque TP, en ajoutant progressivement de nouvelles fonctionnalités au projet.

2.2 Le lab_works_manager

En outre, le dossier lab_works était divisé en sous-dossiers, un pour chaque TP. Ainsi, chaque TP disposait de son propre lab_works, qui contenait tous les fichiers nécessaires pour compiler et exécuter le projet (shaders et le code du workspace).

Pour faciliter la navigation entre les différents lab_works, nous avons mis en place un lab_works_manager. Ce dernier était accessible depuis le menu principal de l'application et permettait de passer d'un lab_works à l'autre sans avoir à recompiler le code. Cette fonctionnalité a été particulièrement aprécier pour pouvoir comparrer nos différents TP entre eux et entre nous quelque soit l'avancement de chacun.

Chargement d'un lab_works Pour illustrer comment le lab_works_manager fonctionnait, je vais maintenant vous montrer le morceau de code qui permettait de charger un lab_works. Dans la fonction `void LabWorkManager::drawMenu()` il suffisait de rajouter et de décliner pour chaque TP le code suivant :

```
1  void LabWorkManager::drawMenu()
2  {
3    // Here you can add other lab works to the menu.
4    if ( ImGui::MenuItem( "Lab work 1" ) )
5    {
6      if ( _type != TYPE::LAB_WORK_1 ) // Change only if needed.
7      {
8        // Keep window size.
9        const int w = _current->getWindowWidth();
10       const int h = _current->getWindowHeight();
11       delete _current;       // Delete old lab work.
12       _current = new LabWork1();   // Create new lab work.
13       _type  = TYPE::LAB_WORK_1; // Update type.
14       _current->resize( w, h );  // Update window size.
15       _current->init();      // Don't forget to call init().
```

Listing 1 – Appel d'un lab_works dans le lab_works_manager

2.3 Le lab_works

La classe LabWork1 est une classe dérivée de BaseLabWork et est utilisée pour implémenter le premier TP. Elle possède les méthodes suivantes :

init() : méthode appelée au lancement de l'application pour initialiser les données de la scène et les données OpenGL.

animate(float) : méthode appelée à chaque frame pour mettre à jour les données de la scène.

render() : méthode appelée à chaque frame pour dessiner la scène à l'écran.

handleEvents(SDL_Event) : méthode appelée pour gérer les événements envoyés par SDL (par exemple, les entrées clavier ou souris).

displayUI() : méthode appelée pour afficher l'interface utilisateur de la scène.

La classe LabWork1 possède également plusieurs membres privés qui stockent les données de la scène et les données OpenGL :

_triangle : un tableau de Vec2f qui contient les coordonnées du triangle à dessiner à l'écran.

_program : un identifiant OpenGL pour le programme de shaders utilisé pour dessiner la scène.

_vbo : un identifiant OpenGL pour le VBO (Vertex Buffer Object) utilisé pour stocker les données du triangle.

_vao : un identifiant OpenGL pour le VAO (Vertex Array Object) utilisé pour stocker les données de configuration de l'attribut de position du triangle.

_bgColor : une Vec4f qui contient la couleur de fond de l'application.

_shaderFolder : une chaîne de caractères qui contient le chemin vers le répertoire où se trouvent les shaders utilisés par cette classe.

2.3.1 La méthode init()

La méthode init() s'occupe d'initialiser toutes les données de la scène et de configurer OpenGL. Elle commence par appeler la méthode init() de la classe BaseLabWork, qui s'occupe de configurer SDL et OpenGL. Ensuite, elle charge et compile les shaders nécessaires pour dessiner la scène, crée le VAO et le VBO, et initialise les autres données de la scène.

2.3.2 La méthode animate(float)

La méthode animate(float) met à jour les données de la scène en fonction du temps écoulé depuis le dernier frame. Elle peut par exemple animer des objets en fonction de leur vitesse ou de leur accélération.

2.3.3 La méthode render()

La méthode render() s'occupe de dessiner la scène à l'écran. Elle commence par nettoyer le framebuffer avec la couleur de fond (_bgColor) et active le programme de shaders (_program). Ensuite, elle active le VAO (_vao) et appelle la fonction OpenGL glDrawArrays() pour dessiner le triangle stocké dans le VBO (_vbo).

2.3.4 La méthode handleEvents(SDL_Event)

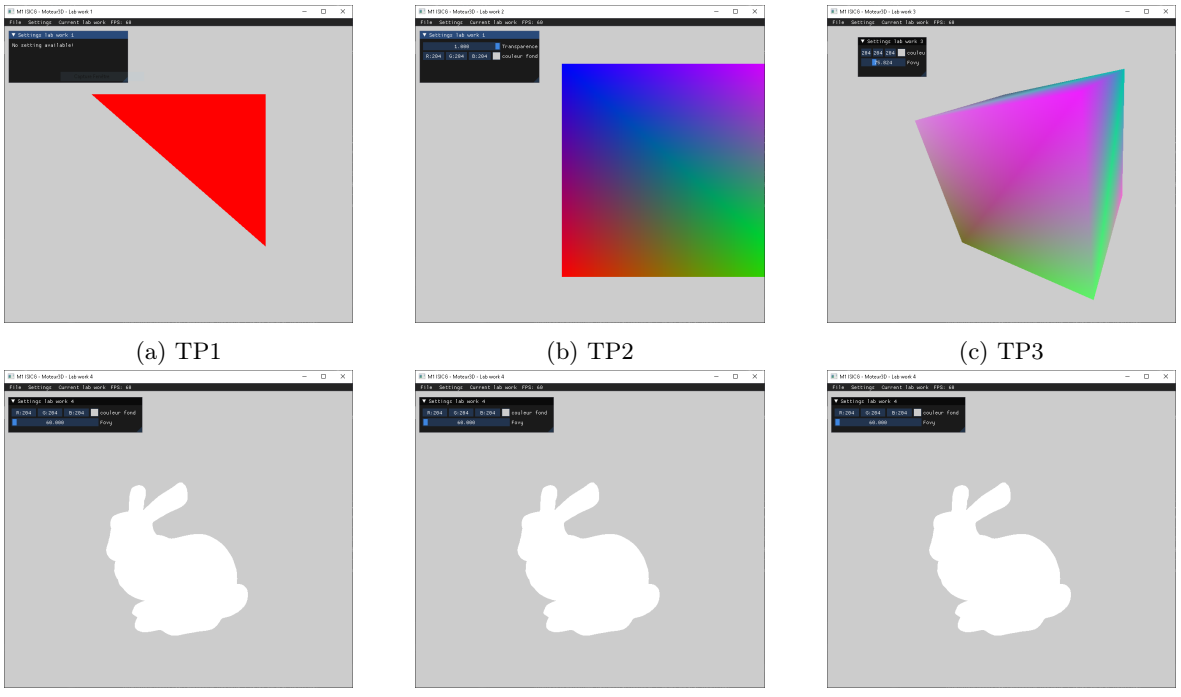
La méthode handleEvents(SDL_Event) est appelée à chaque frame pour gérer les événements envoyés par SDL. Elle peut par exemple gérer les entrées clavier ou souris pour permettre à l'utilisateur de naviguer dans la scène ou de changer les paramètres de l'application. Ce n'est pas utile dans le premier TP mais pas la suite je déplasserai la caméras avec.

2.3.5 La méthode displayUI()

La méthode displayUI() s'occupe d'afficher l'interface utilisateur de la scène. Elle peut utiliser la bibliothèque de l'interface utilisateur (UI) de votre choix (par exemple, ImGui) pour afficher des éléments tels que des boutons, des sliders, etc. qui permettent à l'utilisateur de contrôler l'application.

2.4 Présentation des 6 TPs

Le fonctionnement de lab_works et lab_works_manager restant quasiment le même jusqu'à la fin du TP6 (nous allons bien sur rajouter des chose dedans) Voici une description des 6 sujets de TP que nous avons abordés dans le cadre de ce projet. Ci dessous une image de synthèse de chacun d'entre eux. (*cf* Figure 1)



(a) TP1 (b) TP2 (c) TP3

(d) TP4 (e) TP5 (f) TP6

FIGURE 1 – Rendu de touts les TPs

— **TP1** : L'objectif de ce TP était simplement d'afficher un triangle en 2D sur le plan image. Pour ce faire, nous avons dû apprendre à utiliser les fonctionnalités de base de OpenGL, telles que la création d'un programme de shaders, la gestion de VAO et de VBO, et l'utilisation de glDrawArrays(). (*cf* Figure 1a)
— **TP2** : Dans le TP2, nous avons affiché un quad en utilisant un VBO de type "Indexed" et en utilisant des shaders de couleur. Nous avons également animé le quad en le faisant se déplacer de gauche à droite et en modifiant son opacité. (*cf* Figure 1b)

— **TP3** : Il a consisté à afficher un cube multicolore qui tournait sur lui-même en utilisant une caméra freefly. Nous avons appris à créer et à manipuler une matrice de vue pour contrôler la position et l'orientation de la caméra. (*cf* Figure 1c)
— **TP4** : Nous avons chargé un modèle 3D depuis un fichier (le bunny) et avons calculé l'éclairage local de se model. Nous avons aussi vue les éclairages plus avancer comme l'éclairage diffus et l'éclairage spéculaire du model de Phong. (*cf* Figure 1d)
— **TP5** : Il visait à manipuler des textures discrètes pour changer l'apparence de nos objets. Nous avons appris à charger et à appliquer des textures sur nos modèles 3D en utilisant des shaders de texture. (*cf* Figure 1e)
— **TP6** : Pour finir, le TP6 a consisté à mettre en place un pipeline de deferred rendering et à appliquer un post-traitement simple sur notre image. Nous avons appris à utiliser les techniques de deferred rendering pour séparer les différentes composantes de notre scène (couleur, position, normale, etc.) et à effectuer un rendu en plusieurs étapes pour obtenir une image finale de meilleure qualité. Nous avons également appris à utiliser les techniques de post-traitement pour améliorer l'apparence de notre image finale (par exemple, en utilisant l'effet de bloom). (*cf* Figure 1f)

2.5 TP1 : Manipulation des shaders

Dans le cadre de ce projet, nous avons étudié l'utilisation des shaders avec OpenGL. Les shaders sont des programmes qui s'exécutent sur la carte graphique et qui permettent de définir comment les données de géométrie et de texture doivent être transformées et rendues à l'écran. Nous avons mis en place différents shaders au cours de ce projet, en utilisant le langage GLSL (OpenGL Shading Language), pour ajouter des effets tels que l'éclairage, les textures et les couleurs à nos scènes 3D. Nous avons appris à créer et à utiliser des shaders de vertex, de fragment et de geometry, ainsi qu'à passer des données aux shaders via les variables uniformes et les VBO. Pour le premier TP rien d'impossible : afficher un triangle rouge.

2.5.1 Mise en place du fragment Shader et Vertex Shader

La variable aVertexPosition est une variable d'entrée de type vec3 envoyée à l'emplacement 0 et contenant les coordonnées du vertex. La variable varSortieVS est une variable de sortie du shader de vertex qui sera envoyée au shader de fragment. Dans la fonction principale, la position du vertex est affectée à gl_Position.

```
#version 450

layout( location = 0 ) in vec3 aVertexPosition;

out float varSortieVS;

void main() {
  gl_Position = vec4(aVertexPosition, 1.f);
}
```

Listing 2 – Vertex Shader du TP 1

Le code suivant utilise la directive #version 450 pour indiquer la version 4.5 du GLSL. La variable fragColor est déclarée comme une variable de sortie de type vec4 envoyée à l'emplacement de sortie 0. Dans la fonction principale, la valeur (1.0, 0.0, 0.0, 1.0) est affectée à fragColor, ce qui indique que chaque fragment sera affiché en rouge.

```
#version 450

layout ( location = 0 ) out vec4 fragColor;

void main() {
  fragColor = vec4(1.f,0.f,0.f,1.f);
}
```

Listing 3 – Fragment Shader du TP 1

2.5.2 Le VAO et VBO

Un Vertex Array Object (VAO) est un objet OpenGL qui stocke les informations de configuration des données de géométrie. Il permet de mémoriser les différentes configurations de VBO (Vertex Buffer Object, ou "tampon de sommets") et d'attributs de vertex, afin de pouvoir les rendre facilement. Par ailleur, un Vertex Buffer Object (VBO) est un tampon de mémoire qui stocke les données de géométrie (par exemple, les coordonnées des sommets, les couleurs, les normales, etc.). Il permet de transférer ces données de manière efficace vers le pipeline de rendu, sans avoir à les envoyer à chaque frame.

Le code ci-dessous crée un VAO et un VBO, puis configure l'attribut 0 du VAO pour utiliser les données du VBO. L'attribut 0 du VAO est activé et sa taille et son type sont définis. Le VBO et le VAO sont liés en indiquant la position de départ et l'espace entre chaque sommet dans le VBO. Enfin, l'attribut 0 du VAO est lié au descripteur d'attribut 0 du shader.

```
1    // Creation de la VBO ------------------------------------------------------------
2    glCreateBuffers( 1, &_vbo );
3    glNamedBufferData( _vbo, sizeof( Vec2f ) * _triangle.size(), _triangle.data(),
     GL_STATIC_DRAW );
```

Listing 4 – Création VBO TP1

```
1    // Creez le VAO ------------------------------------------------------------------
2    glCreateVertexArrays( 1, &_vao);
3
4    // Activez leattribut 0 du VAO ---------------------------------------------------
5    glEnableVertexArrayAttrib( _vao, 0 );
6
7    // Definissez le format de leattribut avec la fonction ---------------------------
8    glVertexArrayAttribFormat( _vao, 0, 2, GL_FLOAT, GL_FALSE, 0 );
9
10   // Liez le VBO et le VAO avec la fonction ----------------------------------------
11   glVertexArrayVertexBuffer( _vao, 0, _vbo, 0, sizeof(Vec2f));
12
13   // Enfin, connectez le VAO avec le Vertex shader ---------------------------------
14   glVertexArrayAttribBinding( _vao, 0, 0 );
15   glVertexArrayAttribBinding( _vao, 1, 1 );
```

Listing 5 – Création VAO TP 1

2.6 TP2 : On affiche un quad

2.6.1 L'EBO

Dans ce second TP j'ai créé ma première EBO. C'est un tampon de mémoire qui stocke les indices des sommets d'un maillage 3D. Il permet de réduire la quantité de données à envoyer au pipeline de rendu en ne transférant que les sommets nécessaires, au lieu de tous les sommets du maillage. Cela peut être particulièrement utile lorsque plusieurs objets partagent des sommets en commun, car cela permet d'éviter de les envoyer plusieurs fois. Pour utiliser un EBO, il suffit de le lier au VAO et de spécifier les indices des sommets à dessiner en utilisant une des fonctions de rendu d'OpenGL, comme glDrawElements().

```
1    _bufferVertices = { Vec2f( -.7, -.7 ), Vec2f( .7, -.7 ), Vec2f( -.7, .7 ), Vec2f( .7,
      .7 ) };
2    _bufferColor  = {
3         1.f, 0.f, 0.f, // rouge
4         0.f, 1.f, 0.f, // vert
5         0.f, 0.f, 1.f, // bleu
6         1.f, 0.f, 1.f, // magenta
7    };
8    _bufferTriangle = { 0, 1, 2, 1, 2, 3};
```

Listing 6 – Données qui serviront pour la création d'une EBO

2.6.2 Les varible uniforme

Les variables uniformes sont des variables de shader qui ont la même valeur pour tous les sommets ou pixels d'une image. Elles permettent de passer des données du programme OpenGL vers les shaders de manière efficace, car elles ne nécessitent pas de stockage en mémoire pour chaque sommet ou pixel.

```
    _translation = glGetUniformLocation( _program, "uTranslationX" );
    _transparenc = glGetUniformLocation( _program, "uAlphaCanal" );
```

Listing 7 – Création des variables Uniform dans le cpp du tP2

```
    ImGui::Begin( "Settings lab work 1" );
    if (ImGui::SliderFloat("Transparence", &_transparencValue, 0.f, 1.f)) {
      glProgramUniform1f( _program, _transparenc, _transparencValue );
    }
    if (ImGui::ColorEdit3("couleur fond", glm::value_ptr(_bgColor))) {
      glClearColor( _bgColor.x, _bgColor.y, _bgColor.z, _bgColor.w );
    }
    ImGui::End();
```

Listing 8 – MàJ des variables Uniform dans le cpp du tP2

```
layout( location = 0 ) in vec3 aVertexPosition;
layout( location = 1 ) in vec3 aVertexColor;
out vec3 color;
uniform float uTranslationX;

void main() {
  color = aVertexColor;
  gl_Position = vec4(aVertexPosition.x + uTranslationX, aVertexPosition.y, 0.f , 1.f);
}
```

Listing 9 – Utilisation des variables Uniform dans le Shader

2.7 TP3 Rotation du cube et caméra

Dans le TP3, vous avez rencontré un problème lors de l'utilisation de la caméra, qui se manifestait par un comportement inattendu ou des erreurs lors du rendu de la scène 3D. Après investigation, j'ai découvert que le problème était causé par le fait que je n'avais pas correctement instancié la caméra avant de l'utiliser dans votre code.

2.7.1 Création du Cube

La création d'un cube en 3D avec OpenGL peut être relativement simple, car il suffit de définir les positions de ses sommets et de les assembler en triangles pour former la géométrie de l'objet. Cependant, cette tâche peut être fastidieuse et prendre du temps, en particulier lorsque l'on utilise un tampon d'indices (EBO) pour spécifier les triangles.

```
  LabWork3::Mesh LabWork3::_createCube()
  {
    LabWork3::Mesh mesh;

    mesh._sommets_Position
      = { Vec3f( .5, .5, .5 ),  Vec3f( .5, .5, -.5 ),  Vec3f( .5, -.5, .5 ),  Vec3f( .5, -.5, -.5 ),
        Vec3f( -.5, .5, .5 ), Vec3f( -.5, .5, -.5 ), Vec3f( -.5, -.5, .5 ), Vec3f( -.5, -.5, -.5 ) };
    mesh._sommets_Couleur = { Vec3f( getRandomVec3f() ), Vec3f( getRandomVec3f() ), Vec3f( getRandomVec3f() ),
                  Vec3f( getRandomVec3f() ), Vec3f( getRandomVec3f() ), Vec3f( getRandomVec3f() ),
                  Vec3f( getRandomVec3f() ), Vec3f( getRandomVec3f() ) };
    mesh._sommets_Indices = { 0, 1, 2, 3, 1, 2, 1, 3, 5, 7, 3, 5, 5, 7, 4, 6, 7, 4,
                  4, 6, 0, 2, 6, 0, 5, 4, 1, 0, 4, 1, 6, 7, 2, 3, 7, 2 };
    mesh._objet_transformation = glm::scale( glm::mat4( 1.f ), Vec3f( .8 ) );
    return mesh;
  }
```

Listing 10 – Création d'un cube TP 3

2.7.2 La matrice MVP

La matrice MVP (Model View Projection) est une matrice de transformation. Elle est composée de trois matrices : la matrice de modèle, la matrice de vue et la matrice de projection. Ces matrices

permettent de définir la position, l'orientation et l'échelle de l'objet, ainsi que la position et l'orientation de la caméra et la manière de projeter l'objet sur l'écran. La matrice MVP est obtenue en multipliant ces trois matrices entre elles et est utilisée pour transformer les coordonnées des sommets de l'objet 3D dans le repère de l'écran.

```
void main() {
  color = aVertexColor;

  gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aVertexPosition,1.f
    );
}
```

Listing 11 – Utilisation de la matrice MVP TP 3

Dans mon code j'aurais trois foncitons qui mettrons à jours les inofrmations de ces matric pour les shaders en passant par des variable uniforme : `_updateViewMatrix()`, `_updateProjectionMatrix()` et `_updateModelMatrix()`

2.7.3 Ajout d'une caméra

Une caméra Free Fly en OpenGL permet de déplacer et de faire pivoter la caméra dans l'espace 3D de manière libre. Elle est utilisée pour naviguer dans l'espace 3D, comme dans les jeux vidéo ou les logiciels de modélisation 3D. La position de la caméra est mise à jour en utilisant des matrices (le V et P de la matrice MVP).

Il a fallu mettre à jour le code de la caméra pour qu'elle fonctionne. J'ai aussi eu un problème avec la caméra : après de longues heures de recherche, j'ai réalisé que le programme pouvait être compilé et exécuté sans jamais instancier la caméra (bien sûr, elle ne fonctionne pas). Cela m'a pris du temps avant de me rendre compte de l'erreur, car je pensais que mon code était correct alors qu'il était simplement incomplet.

2.8 TP4 Le bunny et la lumière

Pour afficher un modèle 3D dans ce TP, il était nécessaire de utiliser le code fourni dans le dossier "models" qui nous permettait de charger des maillages 3D. Nous avons dû éditer le fichier "triangle_mesh.cpp" pour qu'il puisse charger et configurer les composants nécessaires pour l'affichage, tels que VAO, VBO et EBO.

2.8.1 La fin du code de Triangle_mesh.cpp

Voici le code qui permet de configurer les VBO et VAO pour utiliser un maillage de triangles avec OpenGL. Le VBO contient les données des sommets, comme leur position, leur normale, leur coordonnée de texture et leur tangente et bitangente. Le VAO stocke les informations sur la façon de lire ces données dans le VBO. Un EBO est également utilisé pour stocker les indices des sommets à dessiner, ce qui permet de ne pas avoir à répéter les données des sommets. La configuration des VAO et VBO est effectuée en spécifiant la position, le format et l'offset des différentes données dans le VBO et en les connectant au VAO. Les données des indices sont chargées dans l'EBO et liées au VAO.

```
  void TriangleMesh::_setupGL()
  {
    // VBO
    glCreateBuffers( 1, &_vbo);
    glNamedBufferData( _vbo, _vertices.size() * sizeof( Vertex ), _vertices.data(),
    GL_STATIC_DRAW );

    // VAO
    unsigned int stride = sizeof( Vertex );
    glCreateVertexArrays( 1, &_vao );

    // POSITION
    glEnableVertexArrayAttrib( _vao, 0 );
    glVertexArrayAttribFormat( _vao, 0, 3, GL_FLOAT, GL_FALSE, 0 );
    glVertexArrayVertexBuffer( _vao, 0, _vbo, offsetof( Vertex, _position ), stride );
    glVertexArrayAttribBinding( _vao, 0, 0 );

    // NORMAL
```

```
18      glEnableVertexArrayAttrib( _vao, 1 );
19      glVertexArrayAttribFormat( _vao, 1, 3, GL_FLOAT, GL_FALSE, 0 );
20      glVertexArrayVertexBuffer( _vao, 1, _vbo, offsetof( Vertex, _normal ), stride );
21      glVertexArrayAttribBinding( _vao, 1, 1 );
22
23      // TEXTURE
24      glEnableVertexArrayAttrib( _vao, 2 );
25      glVertexArrayAttribFormat( _vao, 2, 2, GL_FLOAT, GL_FALSE, 0 );
26      glVertexArrayVertexBuffer( _vao, 2, _vbo, offsetof( Vertex, _texCoords ), stride );
27      glVertexArrayAttribBinding( _vao, 2, 2 );
28
29      // TANGENT
30      glEnableVertexArrayAttrib( _vao, 3 );
31      glVertexArrayAttribFormat( _vao, 3, 3, GL_FLOAT, GL_FALSE, 0 );
32      glVertexArrayVertexBuffer( _vao, 3, _vbo, offsetof( Vertex, _tangent ), stride );
33      glVertexArrayAttribBinding( _vao, 3, 4 );
34
35      // BITANGENT
36      glEnableVertexArrayAttrib( _vao, 4 );
37      glVertexArrayAttribFormat( _vao, 4, 3, GL_FLOAT, GL_FALSE, 0 );
38      glVertexArrayVertexBuffer( _vao, 4, _vbo, offsetof( Vertex, _bitangent ), stride );
39      glVertexArrayAttribBinding( _vao, 4, 4 );
40
41
42      // EBO
43      glCreateBuffers( 1, &_ebo );
44      glNamedBufferData( _ebo, _indices.size() * sizeof( unsigned int ), _indices.data(),
        GL_STATIC_DRAW );
45      glVertexArrayElementBuffer( _vao, _ebo );
46  }
```

Listing 12 – Fonction _setupGL TP4

Pour ce qui concerne ce TP, il se peut que mon code soit très similaire à celui de Timothée car il m'a beaucoup aidé à comprendre comment gérer l'affichage 3D (VAO, VBO, EBO). Même si cela peut affecter ma note, je préfère le mentionner afin d'éviter tout problème de plagiat.

2.8.2 Une seule matrice MVP

Pour ce TP, j'ai utilisé une seule matrice MVP pour transmettre les paramètres au shader, contrairement au TP précédent où j'en avais utilisé plusieurs. Pour mettre à jour la matrice MVP, j'ai multiplié la matrice de projection de la caméra, par la matrice de vue de la caméra, puis par la matrice de transformation de mon mesh. J'ai ensuite envoyé le résultat à mon programme de shader en utilisant la fonction `glProgramUniformMatrix4fv`.

```
1   void LabWork4::_updateMVPMatrix(){
2     _MVPMatrix = _camera.getProjectionMatrix() * _camera.getViewMatrix() * _mesh.
      _transformation;
3     glProgramUniformMatrix4fv( _program, _MVPMatrixUniform, 1, GL_FALSE, glm::value_ptr(
      _MVPMatrix ) );
4   }
```

Listing 13 – Matrice MVP TP4

2.8.3 Le shader

Pour ce TP, j'ai dû ajouter de nombreux paramètres au shader pour prendre en compte l'éclairage selon le modèle de Phong. Cela a nécessité un code plus complexe pour le shader. Dans ce code, nous calculons l'éclairage de Phong en utilisant les paramètres d'entrée du vertex shader. Nous calculons la direction de la lumière et de la vue, ainsi que la normale du vertex. Ensuite, nous calculons la composante diffuse en utilisant la direction de la lumière et la normale, ainsi que la composante spéculaire en utilisant la réflection de la lumière sur la surface et la direction de la vue. Enfin, nous combinons ces composantes avec la couleur ambiante pour obtenir la couleur finale de fragment.

```
1   void main()
2   {
3     vec3 vecLi = normalize(inLumierPosition - inVertexPosition);
4     vec3 vecLo = normalize(vec3(0,0,0)-inVertexPosition);
5
6     if (dot(normalize(vecLo),normalize(inVertexNormal)) < 0){
```

```
7       vecN = normalize(vec3(0,0,0)-inVertexNormal);
8     }
9     else {
10      vecN = normalize(inVertexNormal);
11    }
12
13    // DIFFUSE
14    theta = dot(normalize(vecLi),vecN);
15    vec3 DiffuseColor = uDiffuseColor * max(0,theta);
16
17    // SPECULAIRE
18    vec3 specularColor = uSpecularColor * pow( max (0, cos( dot( normalize( reflect(
        normalize(-vecLi),vecN)), normalize(vecLo)))),uShininess) ;
19
20    fragColor = vec4(uAmbientColor , 1.f ) + vec4(DiffuseColor , 1.f ) + vec4(specularColor
        , 1.f );
21  }
```

Listing 14 – Matrice MVP TP4

2.9 Le TP5 une première scène

Malheureusement, j'ai dû arrêter de travailler sur le TP5 avant d'avoir fini. Je n'ai pas eu le temps de rédiger l'intégralité du compte rendu, mais vous pouvez vous rendre compte de mon travail en examinant le code du programme et du Lab Work du TP5. Désolé pour cette inconvenience.

3 Conclusion

En conclusion, j'ai réalisé un grand nombre de tâches durant ce TP. J'ai commencé par afficher un triangle en 2D sur le plan image, puis j'ai affiché un quad multi-couleur animé avec une gestion de l'opacité. J'ai ensuite affiché un cube multicolore qui tournait sur lui-même avec une caméra Free Fly. J'ai chargé un modèle 3D depuis un fichier et j'ai calculé l'éclairage local selon le modèle de Phong. Malheureusement, je n'ai pas eu le temps de terminer les TPs et de rédiger un compte rendu complet, mais j'espère que vous avez pu découvrir les différentes tâches que j'ai réalisées grâce au lab Works.