Faculté des Sciences & Techniques — Université de Limoges

Projet d'Moteur 3D

Compte Rendu

Élèves :
Mathurin Cartron

Professeur :
Maxime Maria

Table des matières

Liste des tableaux

Table des figures

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

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)

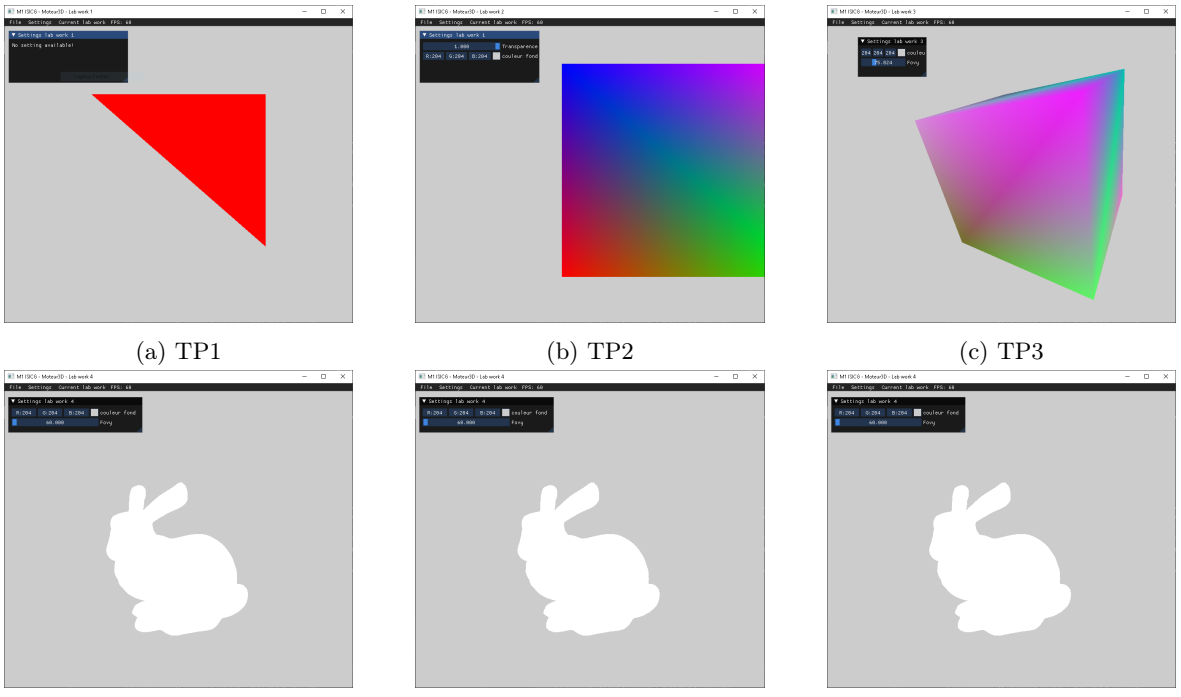
La méthode handleEvents(SDL_Event) est appelée à chaque frame pour gérer les événements envoyés par SDL. Elle peut par exemple gérer les entrées clavier ou souris pour permettre à l'utilisateur de naviguer dans la scène ou de changer les paramètres de l'application. Ce n'est pas utile dans le premier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

2.4.1 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)

2.4.2 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)

2.4.3 TP3

Le TP3 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)

2.4.4 TP4

Dans le 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)

2.4.5 TP5

Le TP5 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)

2.4.6 TP6

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)

3 Conclusion