Informatique Graphique

TP - Rapport

Mathis MARGOT
21606171

Université Paul sabatier
Master IGAI

Encadrant :
Mathias PAULIN

Introduction

Concepts abordés

Ce cours a pour but de donner une introduction à la représentation numérique d'objets géométriques 3D, du traitement de ces données ainsi que du rendu en temps réel de ces objets. Il nous a été présenté le Pipeline de Rendu exécuté par les cartes graphiques, et comment le manipuler via une API spécifique (OpenGL → utilisé ici, Vulkan, DirectX, Metal, etc.).

Objectifs

L'objectif de ce TP est d'implémenter un Moteur de Rendu minimaliste abordant les concepts étudiés en cours. Ainsi, se familiariser et se perfectionner sur la programmation graphique en temps réel et en comprendre les pratiques.

Technologies utilisées

Pour cela, nous utilisons C++ couplé d'OpenGL ainsi que le framework Qt permettant de gérer l'interface graphique autour de l'Engine.
Pour cela, il nous a été fournit une base logicielle implémentant un contexte OpenGL dans une fenêtre Qt présentant une démonstration d'utilisation d'OpenGL. Par ailleurs, la librairie mathématique GLM était fournie avec cette base.
J'ai ajouté la librairie Eigen, qui me permet d'utiliser des outils plus avancés comme les Bounding Box.

Fonctionnalités exigées

Le Moteur de Rendu doit permettre :

- D'afficher une sphère UV et une sphère Ico (obtenue après subdivision d'un icosaèdre régulier).
- D'afficher ces objets avec un éclairage diffus
- D'afficher une sphère avec une couleur variant selon l'erreur entre une sphère parfaite et la position effective des points du maillage de la sphère.
- En bonus : de pré-charger des fichier obj/mtl afin de lire une scène préparée et fournie avec l'Engine correspondant à un éclairage 3 points.

Fonctionnalités livrées

Fonctionnalités de base

Comme exigé par le sujet du TP, j'ai implémenté la construction d'un Mesh pour une Sphère UV et pour une Sphère Ico. En plus de ces deux objets, j'ai implémenter la construction d'une Sphère Cube (obtenu à partir d'un cube "subdivisé" dont les points (vertices) ont été normalisés).
Pour l'éclairage diffus, j'ai opté pour trois shaders différent, un Phong, un Blinn Phong et un PBR. Un shader pour la visualisation des erreurs des sphères ainsi qu'un autre qui attribue la normale à la surface à la couleur sont accessibles avec ceux offrant l'éclairage diffus.
Pour accompagner ces shader d'éclairage diffus, j'ai implémenter trois types de lumière, la lumière point, éclairant dans toute les directions depuis un point dans l'espace, la lumière directionnelle, éclairant depuis la même direction depuis tous les points de l'espace et la lumière spot, qui correspond à une lumière point soumise à une contrainte angulaire sur les directions d'émission possibles. Enfin, pour accompagner le shader PBR, j'ai implémenté un type Matériau permettant de définir l'aspect de la surface (rugosité, couleur albedo, …).
La lecture dans des fichiers obj/mtl n'a pas été implémentée, mais d'autres outils des autres fonctionnalités proposés pourront permettre de réaliser une scène en éclairage 3 points.

Fonctionnalités avancées

En plus des fonctionnalités exigée et des quelque suppléments parmis les fonctionnalités de base, j'ai implémenté le fait de pouvoir sélectionner un objet de la scène en cliquant dessus. Cela m'a permis de construire une interface autour de la selection. En effet, une bonne partie de l'interface graphique s'adapte à l'objet sélectionné. On peut ainsi modifier :

- Le matériau d'un objet (pas possible si c'est une lumière).
- Les transformations d'un objet :
    - La position (x, y, z) du centre de l'objet
    - L'angle (Euler) autour des axes (x, y, z) du repère local à l'objet (en degrés).
    - La mise à léchelle suivant les axes (x, y, z) du repère local à l'objet.
- Les paramètre spécifiques au type d'objet (e.g. Sphère UV : méridiens et parallèles).

En plus de ces paramètre accessibles depuis l'interface graphique, on trouve aussi des boutons pour créer des objets (modèles ou lumières), supprimer l'objet actuellement sélectionné et changer de shader.
Ces fonctionnalités permettent de gérer une scène 3D de manière minimaliste, mais j'ai ajouter une interaction plus avancée : le Gizmo. En effet, j'ai ajouté à la scène un Gizmo de translation et de mise à l'échelle (qu'on peut choisir via un bouton de l'interface). L'objet sélectionné sera modifié par le Gizmo, et ce dernier prendra la forme du repère local de l'objet sélectionné. De plus, le Mesh du Gizmo est rendu de manière "uniforme", ça veut dire que quelque soit la distance entre l'objet sélectionné et la caméra, le Gizmo aura toujours la même échelle.
Enfin, l'utilisateur peut choisir entre voir une scène type "rendue" ou type "édition".

Implémentation

Utilisation de la base logicielle

Après avoir parcouru les fichier de la base logicielle fournie, j'ai pris la décision de supprimer toute les démos dont je n'avais pas besoin, et de refaire une classe Engine reprenant les attributs de la classe OpenGLDemo fournie. De cette manière, je supprime l'héritage nécessaire depuis OpenGLDemo étant donné que je n'avais pas besoin de gérer plusieur types d'Engine, un seul suffit pour ce travail. J'ai donc simplement remplacé le pointeur vers un OpenGLDemo par un pointeur vers un Engine dans la classe MyOpenGLWidget fournie. D'autre part, j'ai modifié l'implémentation fournie de la classe MainWindow qui rendait MyOpenGLWidget « Central Widget ». Par conséquent, je ne pouvais pas ajouter d'autre éléments graphiques (comme des boutons) à ma fenêtre. J'ai donc supprimé l'appel de la fonction *makeCentralWidget* et le pointeur vers MyOpenGLWidget qui est devenu accessible depuis le membre **ui**. Après avoir adapté MainWindow à mes besoins, j'ai pu utiliser Qt Creator pour ajouter les éléments graphiques que je voulais.
Voici donc la liste des fichiers que j'ai gardé de la démo fournie :

- myopenglwidget.h
- myopenglwidget.cpp
- camera.h
- camera.cpp
- opengl_stuff.h
- main.cpp

Modèles et Lumières

Le sujet de TP était initialement de construire des maillages de Sphère (UV et Icosaèdre). Pour cela, j'ai tout d'abord créé la classe Mesh stockant la géométrie et la topologie. Pour cela, j'utilise deux simple listes. Pour la géométrie, j'ai d'abord créé un type struct Vertex contenant deux vecteurs 3D (position et normale) et un vecteur 2D (pour les coordonnées de texture, bien que je n'utilise pas ce vecteur dans mon code). Pour la topologie, c'est une liste d'entiers qui, pris trois par trois, correspondent aux indices des points dans la liste de Vertex. Enfin, trois entier permettant de stocker le VAO, le VBO et le EBO. Je me suis beaucoup inspiré de la page web fournie par le sujet pour réaliser cela.
Le Mesh étant fonctionnel, j'ai ensuite créé une classe Model contenant une liste de Mesh. De cette manière, je pouvais ajouter des informations à cette liste comme des transformations géométriques. De plus, j'ai pu hériter de la classe Model pour faire les classes UVSphere, IcoSphere, CubeSphere et Cube, pouvant ainsi ajouter des attributs propres à ces types de Model. J'ai pour cela ajouté des fonctions virtuelles au Model permettant de modifier ces attributs spécifiques en passant par un type union.
Pour les lumières, j'ai utilisé le même stratagème, j'ai d'abord créé une classe Light offrant des fonctions virtuelles de paramétrage selon un type union, et j'ai pu hériter de cette classe pour obtenir les classes PointLight, DirLight et SpotLight.

Construction de Mesh

Comme précisé plus haut, les classe UVSphere et IcoSphere, CubeSphere et Cube héritent de la classe Model. Ils offrent tous une fonction spécifique permettant de construire les Meshs correspondant à leur type.

Sphère UV

La sphère UV correspond au découpage de la terre en méridiens et parallèle. Donc l'idée est de trouver les coordonnées d'un point sphérique selon un indice allant de 0 au nombre de méridiens et un de 0 au nombre parallèles. J'ai utilisé cette page web pour me renseigner sur l'algorithme de création de cette sphère.

Sphère Icosaèdre

La sphère Icosaèdre correspond à un Icosaèdre régulier que l'on a subdivisé selon la demi-arête. Pour la construction de l'Icosaèdre, il s'agit de placer d'abord les sommets du haut et du bas, et d'alterner la disposition des points entre le sommet du haut et le sommet du bas tout en faisant le tour de ces derniers. Cette page web m'a aussi aidé pour cette partie ainsi que pour la demi-arête expliquée en suivant.
La demi-arête consiste à prendre chaque triangles, et de le subdiviser en 4 nouveaux triangles en créant 3 nouvelle arêtes entre les milieux des arête du triangle de base. J'ai donc utilisé l'algorithme expliqué sur la page. À noter que mon ordinateur doté d'une carte graphique NVidia GTX 1070 freeze complètement à la neuvième subdivision de l'icosaèdre. Le problème n'étant pas le nombre de triangles mais l'exécution très complexe de ma version de l'algorithme.

Sphère cube

La sphère cube est une sphère obtenue à partir d'un cube dont les faces ont plus ou moins de triangles et dont les points sont normalisé (ramenés à une distance au centre de 1).
Pour créer cette sphère, j'ai décidé de rajouter un Mesh par face du cube, de donner une paramètre de *résolution* permettant d'avoir plus ou moins de triangles sur chaque face, et, pour chaque face, normaliser ses points et attribuer la même valeur à sa normale.
Note : pour le cube, j'ai fait exactement la même chose sans normaliser ses points.

Shader

La classe Shader permet de contenir un *shader program* composé d'un *vertex shader* et d'un *fragment shader*. En effet, son constructeur prend en paramètre les chemins des fichier contenant les codes GLSL de ces shaders. C'est une simple abstraction d'OpenGL permettant d'avoir un accès moins pénible aux fonctions liées aux shaders (comme *glUniform*). De plus, les structures uniform plus complexe de mes shaders peuvent donc être paramétrés plus facilement avec cette classe.

Cliquer sur des objets

Comme décrit dans les fonctionnalités avancées, l'utilisateur peut choisir l'objet sélectionné en cliquant dessus. Pour rendre cela possible, j'ai utilisé une technique de Ray-Casting : le principe est de caster un rayon depuis la position de la souris, et de tester son intersection avec les Boîtes Englobantes Orientées (OBB) des objets de la scène. Si le rayon traverse une boîte, l'objet correspondant est alors sélectionné, sauf si un autre objet est plus proche, auquel cas l'objet le plus proche est toujours sélectionné.

Modifier les objets avec un Gizmo

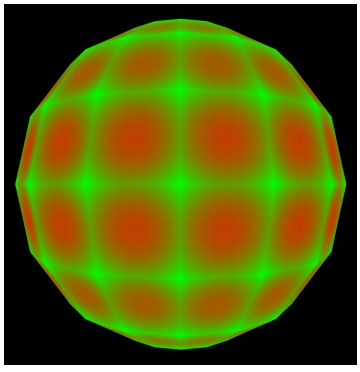
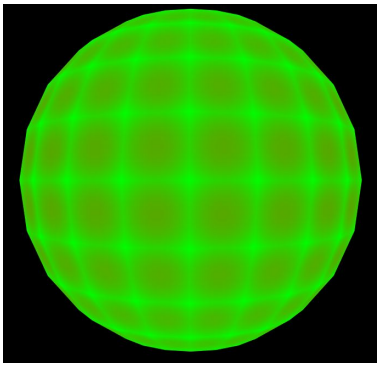
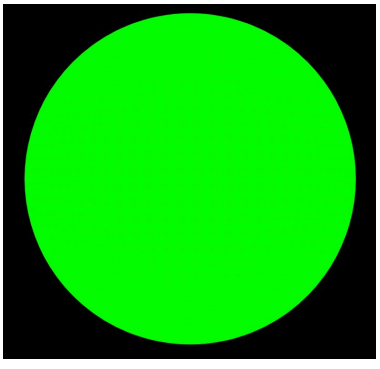
En plus des boutons permettant de changer la position, rotation, mise à l'échelle, des Gizmo prenant en compte les transformations de l'objet (translation + rotation). Il y a un gizmo pour la translation et un pour la mise à l'échelle. L'interaction du clic sur le gizmo utilise les OBB, et le mouvement du Gizmo est obtenu via le calcul de l'angle entre la direction prise par la souris et la direction du vecteur du gizmo selon le repère de l'écran.

Engine et ses composants

L'Engine est là pour rassembler objets, lumières, shaders ensemble et former une scène. L'engine possède donc une liste de Shader. L'un d'entre est sélectionné et est donc utilisé pour afficher les objets. Pour la scène en elle même, j'ai créé une classe ModelManager permettant de stocker tous les Model et toute les Light. Il permet aussi de gérer la sélection de ceux-ci, et donc d'afficher l'objet sélectionné avec une couleur particulière. Il permet aussi le passage du mode édition au mode rendu, de stocker les caméras, et de gérer les Gizmos.

Analyse

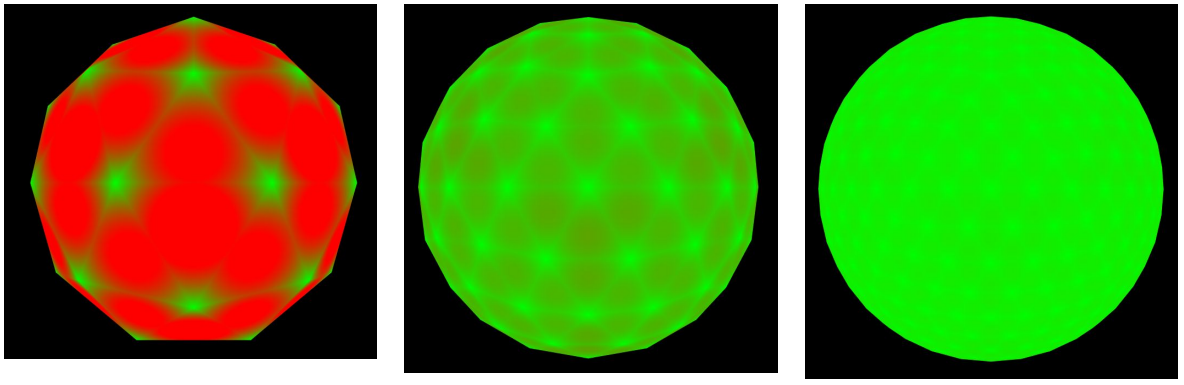
Erreur de la Sphère UV

224 triangles 528 triangles 9800 triangles

On remarque que la sphère UV « progresse » de manière très linéaire. En effet, si on double le nombre de triangle, la proportion d'erreur sur chaque triangle a l'air d'être divisée par deux.

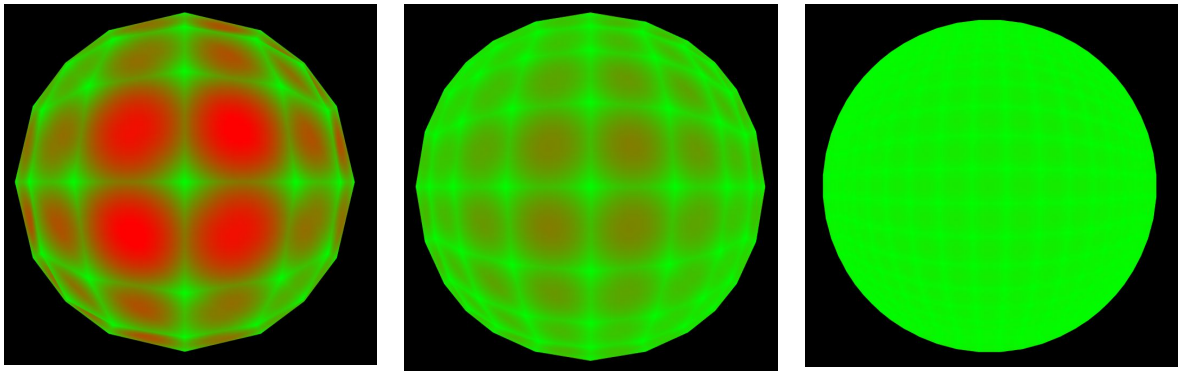
Erreur de la Sphère Icosaèdre



80 triangles 320 triangles 1280 triangles

On s'aperçoit ici que la sphère icosaèdre a un important taux d'erreur si peu de subdivision lui sont appliquées. Par contre, son erreur diminue de manière aussi rapide que le nombre de ses triangles.

Erreur de la Sphère Cube



192 432 2700

La cube sphère montre une progression plus particulière. En effet, les triangles concentrés au centre des face du cube de construction comporte plus d'erreur que les triangles concentrés sur les « bords ». Par conséquent, cela laisse supposer qu'il faut beaucoup de triangles, mais au contraire, il suffit de peu de triangle pour obtenir une sphère assez lisse.

Conclusion

Progression

Certains documents, personnes ou même difficultés rencontrées m'ont apporté de solides connaissance sur certaines facette de l'informatique graphique en temps réel. En effet, tout au long du TP j'ai été confronté à des problème à résoudres, tous autant instructifs que différents.

Motivation

Je pense réutiliser le travail réalisé lors de ce TP afin de continuer d'explorer des choses qui me sont encore inconnue. De plus, ça sera une bonne base logicielle pour les TP des années suivantes.

Commentaire

Un TP comme je les aimes : un sujet, un livrable, une note. Cela nous laisse toujours la possibilité d'aller très loin au-delà du sujet de TP, ce qui n'est pas toujours facile à faire quand on a une deadline plus rapide. De plus, cela permet aussi de continuer à travailler sur le TP après la fin de l'UE. Là encore c'est plus simple de continuer sur une lancée que de reprendre les premiers TP rendus en début de semestre.
C'est de loin le TP que j'ai préféré faire et sur lequel j'ai passé le plus de temps, merci pour cette UE.