Rapport M1 IIG3D

Robin Rouphael

Contents

1 Introduction

La compilation, le lancement, les commandes et les fonctionnalités de ce projet sont décrites dans les fichiers **README.md** et **MANUAL.md**. Il est fortement recommandé de les lire avant ce rapport afin d'avoir une vue d'ensemble des fonctionnalités implémentées.

2 Intention

Ce projet était pour moi l'occasion à la fois de m'initier à OpenGL et au rendu temps réel mais également de poser les bases d'un petit moteur de rendu et d'édition personnel sur lequel je pourrai continuer à me former.

J'ai donc voulu concevoir une implémentation la plus modulaire possible, facile à étendre et robuste.

3 Implémentation

3.1 Base fournie

J'ai commencé par nettoyer la base logicielle fournie en enlevant le principe des demos pour ne garder que la classe de base et la classe *Camera* à laquelle je n'ai pas touché.

Mon objectif était alors de permettre la creation et la manipulation de plusieurs objets dans la scène à travers une interface utilisateur

3.2 Meshes

Classe Mesh

Un Mesh est un ensemble de points et d'indices formants respectivement une géométrie et une topologie. Cette classe contient à la fois les données de CPU et les données de GPU sous la forme de buffers openGL. De plus, elle contient également un vecteur de textures pouvant contenir tout type de textures (speculaires, diffuses, etc...).

3.3 Modèles

Classe Model

Ma définition d'un modèle est celle d'un ensemble de meshes dont le point commun est leur appartenance à une même entité dans le moteur de rendu. Ils ont donc également pour point commun leur matrices de translation, de rotation et d'échelle. Cette définition rencontre toute son utilité dans le cas du chargement d'un fichier .obj car elle respecte sa structure. Une des amélioration future que je compte implémenter consiste à donner la possibilité à l'utilisateur de fusionner plusieurs modèles dans un seul afin de créer un "super modèle" regroupant encore plus de meshes.

3.4 Modèles spécialisés

Les modèles suivants peuvent être direction crées par l'utilisateur avec des paramètres par défaut qui peuvent ensuite être modifiés par l'utilisateur. Ils héritent donc de la classe *Model*

3.4.1 UVSphere

Classe Sphere

L'algorithme utilisé pour la création de la sphere UV vient directement du site `http://www.songho.ca/opengl/gl_sphere.html` et reste bien plus simple que celui de l'IcoSphere.

J'ai cependant modifié la representation des coordonnées en utilisant la convention rayon-colatitude-longitude avec θ la colatitude et φ la longitude et avec les formules suivantes :

$$\begin{aligned} x &= \rho \sin\theta \cos\varphi \\ y &= \rho \sin\theta \sin\varphi \\ z &= \rho \cos\varphi \end{aligned}$$

J'ai également rajouté des coordonnées de textures de type UV mapping ou projection de mercator.

3.4.2 IcoSphere

Classe IcoSphere

Pour l'Icosphere, j'ai également utilisé la même page web principalement pour la construction de l'icosahèdre.

L'algorithme de subdivision prends chaque triangle et le subdivise en 4 nouveau triangles en créant 3 nouvelles arrêtes au milieu de chaque arrêtes. L'inconvéniant majeur de l'algorithme de subdivision utilisé est sa complexité en temps qui a pour conséquence de limiter drastiquement le nombre de

subdivisions réalisables sous peine de ralentir massivement l'application à la création de la sphere.

3.5 Shaders

Classe Shader

La classe Shader fournit les fonctions nécessaires à l'ajout de tout uniform aux shaders glsl utilisés. De plus elle offre une gestion de l'ajout des lumières point et spot et de l'indexage des textures pour les sampler2D. Un objet Shader est surtout définit par son programme glsl construit à partir du vertex shader et du fragment shader.

3.6 Texture

Classe Texture

La classe Texture gère le chargement, la création et le binding des textures aux shaders. Elle est pour l'instant la base d'une hiérarchie comportant les classes *DiffuseMap*, *SpecularMap*, *NormalMap* et *HeightMap*. Pour l'instant cette hiérarchie n'est pas utile car toutes les textures ont le même comportement. Cependant je l'ai conservée en vue d'avancées futures.

3.7 Lumières

Classe Light

La classe Light sert de base à une hiérarchie comportant les classes *PointLight* et *SpotLight*. Cette hiérarchie a pour principale utilité de fournir un polymorphisme efficace lors de l'ajout des lumières aux shaders, chaque classe fille se chargeant d'appeller la bonne fonction dans la classe *Shader*.

3.8 QT

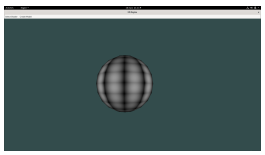
Un des problèmes assez vite rencontré lors de la conception de ce projet était de permettre à l'utilisateur d'acceder aux paramètres des modèles personnalisés à travers l'interface Qt. Une possibilité aurait été de charger la *MainWindow* avec une vingtaine de widgets différents et de gérer leur affichage ou non selon l'objet selectionné. Cette solution possédait deux inconvéniants majeurs : Le premier étant la surcharge de widgets et le peu de modularité offert et le second était la necessité de transferer les requètes de changement à la demo à travers des structures complexes de paramètres. J'ai à la place choisit de créer une hiérarchie ayant pour base la classe *ModelInterface* servant de widgets personnalisés pour chaque type de modèle. Pour celà, la demo contient dans le même vecteur que les modèles, des lambdas servant de constructeurs pour ces widgets. Lorsqu'un objet est selectionné, la demo renvoie à la *MainWindow* le bon widget qu'elle se charge ensuite de manager et de détruire lors d'un changement. Ce design permet notamment de fournir aux widgets un pointeur vers le modèle auquel ils sont affiliés et permettant ainsi l'appel direct des fonctions de modifications du modèle.

J'ai estimé que le coût de cette méthode était négligable face aux commodités de design qu'elle offrait. Globalement, la *MainWindow* et l'*OpenGLWidget* communiquent à travers des signaux Qt en nombre limité mais qui permettent la gestion de toutes les opérations utilisateur.

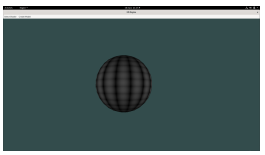
4 Calcul de l'erreur d'approximation

4.1 Sphere UV *vs.* IcoSphere

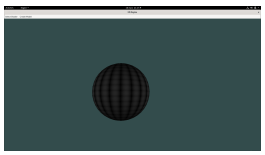
L'erreur d'approximation suit assez logiquement la complexité des deux algorithmes. On peut observer pour la sphère UV un rapport linéaire entre le nombre de stacks et de secteurs et l'erreur quand l'Icosphère semble faire diminuer l'erreur avec un rapport polynomial.



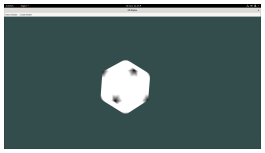
(a) UV with 10 stacks and sectors



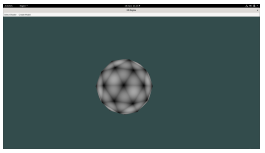
(b) UV with 15 stacks and sectors



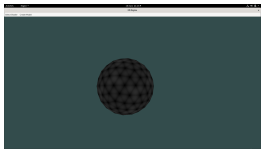
(c) UV with 20 stacks and sectors



(a) Ico with 0 subdivisions



(b) Ico with 1 subdivisions



(c) Ico with 2 subdivisions

5 Conclusion

J'ai beaucoup apprécié de travailler sur ce TP et il m'a permit de bien me former aux bases de l'OpenGL et du rendu temps réel. De plus, j'ai aimé pouvoir concevoir de A à Z un projet qui, bien que petit, avait pour ambition de resister à l'epreuve des modifications futures.

En conclusion, bien que j'y ai passé de nombreuses soirées, je ne regrette aucunement le temps et l'énergie que j'y ai consacré et j'ai énormement apprécié cette UE tout au long du semestre.