Projet moteur 3D

I- Expression du besoin

Les moteurs de rendu sont des outils essentiels pour contrôler la façon dont les éléments sont affichés sur un écran. Dans le cadre d'un jeu, la création d'un moteur de rendu offre une plus grande liberté pour déterminer ce qui y sera possible.
Bien que ce projet ne réponde pas directement à un besoin spécifique, il représente une opportunité de développer des compétences techniques utiles dans le domaine de la 3D.

II - Plan de Test

Tester une moteur de rendu peut s'avérer compliqué, car comme précisé avec plus de détails dans la section sur les difficultées rencontrées, on ne peut avoir accès qu'à un nombre limité d'informations provenant du GPU et un bonne partie des informations importantes sont visuelles, nous avons donc concentré nos test sur les calculs faits par le CPU qui représente tout de même un grosse partie de notre programme.
Nous avons décidé en premier lieu de tester les classes centrales de notre projet, c'est-à-dire les Modèles et les shaders car sans eux nous n'avons pas de rendu visuel. Pour les modèles nous avons vérifié que les informations obtenues via le chargement du modèle étaient correctes et bien organisées, nous avons aussi testé que la relation père-fils entre les modèles était respectée et que la configuration de ce dernier avec openGL ne causait aucune erreur. Pour ce qui est du shader, nous nous sommes assurés qu'il chargeait correctement et que sa configuration avec openGL était correcte, nous avons aussi pu tester un aspect très important des shaders qui sont la justesse des valeurs envoyées aux variables présentes dans le shader (appelées uniformes).
La caméra étant un aspect très important dans notre application, nous l'avons aussi testé en s'assurant qu'elle répondait correctement aux entrées utilisateur, la suite des test représente des aspects moins centraux de notre de notre projet que nous nous sommes tout de même assuré de tester afin d'assurer l'intégrité de l'application (lumière, skybox, textures)

III - Plan de conception

I - Architecture du programme

Notre programme est constitué au total de 16 classes dont les plus importantes sont:

Shader
Les shaders sont des programmes exécutés par le GPU lors du rendu et permettent de gérer la manière dont les éléments sont affichés à l'écran. Les deux types de shaders que nous utilisons sont le vertex shader et le fragment shader.
Le vertex shader va se charger de modifier la position des sommets (vertex) de nos modèles 3D.
Le fragment shader quant à lui va changer la couleur des pixels affichés à l'écran, on l'utilise notamment pour afficher des textures ou pour effectuer du post processing .
Cette classe se charge donc de charger ces shaders stockés que nous avons écrits dans des fichiers .vs et .fs

Caméra
La classe caméra se charge du déplacement et de la rotation dans l'environnement 3D, elle contient ce que l'on appelle une view matrix qui permet d'afficher les éléments à l'écran du point de vue de la caméra.

Mesh
La classe Mesh regroupe toutes les informations nécessaires permettant d'afficher des éléments à l'écran. Les informations principales sont les sommets, les normales et les coordonnées des textures (nous omettons certains aspects par raison de simplicité). Lors du rendu, ces informations sont envoyées au vertex shader qui sont envoyées par la suite au fragment shader.
Lors de l'appel à la fonction permettant de dessiner la mesh, les textures à utiliser dans le fragment shader modifiées et la matrice de transformation en world coordinates est créée.

Modèle
La classe modèle récupère toutes les informations obtenues via l'import d'un modèle 3D avec assimp et se charge de les mettre en forme, cela va du nombre de sommets aux textures à utiliser sur le modèle.
Un modèle 3D contient très souvent plusieurs mesh et il peut exister des relations

père-fils entre ces dernières, La classe modèle contient donc un tableau où sont stockés tous ses fils (de la même classe) et un tableau où sont stockés toutes ses mesh.

GUI

La classe GUI (Graphical User Interface) se charge de dessiner une interface utilisateur permettant d'afficher et de modifier un certains nombres d'informations relatives à la scène, on se sert pour cela d'une librairie nommée Dear ImGui.
Pour développer notre GUI, nous avons gardé les mêmes principes que ceux de DIG, qui sont de ne pas faire d'appel de fonctions inutiles, si un parent ne doit pas être dessiner, ses enfants ne le sont pas, c'est pour ça que chaque partie du GUI à une liste d'enfants et un booléen permettant de savoir si il est dessiné ou non.

Scène

La classe Scène est la classe centrale du projet, elle gère toutes les actions relatives à l'affichage et met en relations la plupart des classes du projet.
Elle contient ce que l'on appelle la render loop, c'est une boucle qui ne s'arrête qu'à la fermeture de l'application et qui va se charger de plusieurs choses:

- Configurer certaines options d'OpengL
- Dessiner les modèles et les lumières
- Afficher les GUI
- Choisir les shaders à utiliser
- Modifier les valeurs des uniformes dans le vertex shader suite aux mouvements de caméra afin de déplacer les objets.

II - Algorithmes

Les pseudo-codes des algorithmes sont fournis en annexe à la fin de ce projet pour des raisons de clarté, les titres des algorithmes sont cliquables pour vous y mener.

Render Loop:
Les algorithmes principaux de notre programme sont la Render Loop, et le calcul du post post processing. La render loop se charge de gérer les actions utilisateurs et de dessiner les éléments à l'écran tels que les modèles 3D (et leur version agrandie pour dessiner leurs contours), les lumières et le GUI.
Elle permet aussi de mettre à jour les matrices de transformation en fonction des déplacements de la caméra.

Bloom :
Le bloom est une technique de rendu qui permet de simuler un comportement plus naturel de la lumière, il existe plusieurs manières de créer un bloom, aucune d'entre-elles n'est la meilleure, elles se valent toutes.
Notre implémentation reste basique et en deux temps :
Le premier sera d'ajouter un flou à notre texture initiale, le second sera de calculer la valeur de sortie du pixel :

Le premier temps :
Le flou sera réalisé en suivant un algorithme simple :
"Parcourir tous les pixels, pour chaque pixels on fera une moyenne des couleurs des pixels alentours"

IV - Difficultés rencontrées

I - Lors de la mise en place du projet nous avons rencontré quelques difficultées, il existe une ressource assez populaire qui fournit des tutoriels sur openGL nommé learnopengl**,** les explications fournies sont d'une très bonne qualité et le site est complet, il ne peut cependant pas fournir toutes les informations dont nous avons besoins lors de la création d'un moteur 3D, et mise à part cette dernière, il est assez difficile de trouver des documentation claires sur les librairies que nous utilisons.
Si on prend l'exemple d'assimp (librairie pour importer des modèles 3D), il nous a été nécessaire de demander de l'aide sur des forums pour mieux cerner comment les données étaient organisées, il en va de même pour identifier certains bugs.

II - Il est aussi assez difficile de debugger avec openGL, lorsque nous avons un écran noir, il peut parfrois s'avérer difficile de localiser la source du problème,
effectuer une action qui peut paraître assez simple comme charger un modèle et le dessiner requiert un certain nombre d'étapes avec openGL ce qui multiplie le risque d'erreurs et les rends assez difficile à identifier.
Le plus difficile à débugger reste cependant les shader, comme ils sont exécutés sur le GPU, il n'y a pas de breakpoint et le nombre d'informations auquel nous avons accès est assez limité, par exemple, on ne peut pas connaître le résultat d'un calcul effectué dans un shader ou la valeur d'une variable autre que les uniformes (envoyés depuis le cpu), la seule manière d'obtenir ces informations sont de colorer les pixels en fonction des valeurs obtenues.
Il existe certains outils de debug comme RenderDoc que nous avons beaucoup utilisé lors de ce projet, il permet de connaître les valeurs des uniformes et des textures envoyées aux shaders et affiche toutes les étapes du rendu de notre scène.

V - Améliorations

I - Refactoring

Notre projet comporte 16 classes et environ 1700 lignes de code, il est donc assez difficile à organiser, même si nous avons été aidé par l'utilisation d'UML, il est difficile de tout prévoir. Ce dernier pourrait donc certainement bénéficier d'un refactoring et d'une organisation plus approfondie, notamment dans la classe Scene même si nous avons fait le maximum pour rendre le code clair.

II - Optimisation de rendu

Lors de la création d'un moteur 3D, un aspect extrêmement important est l'optimisation du code, le but étant de minimiser le coût des opérations effectuées par le GPU et le CPU dans le but d'afficher le plus d'éléments possible à l'écran (modèles 3D, post processing, …) tout en gardant un nombre d'images par seconde raisonnable.

Bien que l'objectif principal de notre moteur n'est pas l'application dans des cas d'utilisation réels (pour un jeu par exemple) mais plutôt pour de l'apprentissage, il peut bénéficier de techniques d'optimisation tel que le deferred shading ou d'autres aspects orientés conception.

III - Logger

L'utilisation d'un logger permet de généraliser la manière dont les informations sont transmises à l'utilisateur.
Dans notre projet, le logger a été défini parmi les premières classes, mais il pourrait être plus exploité, par exemple pour fournir des informations continues à l'utilisateur sur l'intégrité de la scène.

VI - Manuel de l’interface

Le gui est divisée en plusieurs parties :
La première permet de modifier la vitesse de rotation de la caméra (souris) et la vitesse de ses déplacements (clavier)
La seconde permet d'interagir avec la scène, on peut y voir les modèles et les lumières qui sont chargées par le programme, on peut également apercevoir une section dédiés aux effets post-processing.
Certaines sections permettent également de modifier des paramètres, par exemple :

En sélectionnant un modèle, on pourra changer certaines de ses options :

- Le fait de le dessiner ou non ,
- Sa position
- Sa taille (scale)

VII - Conclusion

Ce projet fut énormément instructif car travailler avec des API comme opengl requiert une bonne compréhension de comment les rendus 3D fonctionnent. Nous avons pu par le biais de ce moteur 3D, assimiler des connaissances qui nous seront utiles lors de notre futur parcours en informatique tout en nous familiarisant avec le C++, qui est un langage très utilisé lorsque pour les projets de ce type.

Pseudo code Draw scene:

```
Function drawScene(string shaderName):


    update the camera with mouse or keyoard input

    create the perspective matrix

    prevent writing to the stencil buffer

    draw the Cubemap

    update the default shader's uniforms with the view position,

    view matrix and projection matrix

    update the outline shader with the view and projection matrix

    activate writing to the stencil buffer

    for each model in the scene's models do
        if model is active
            draw the model with the default shader

    prevent writing to the stencil buffer

    for each model in the scene's model do
        if model should be outlined
            draw the model with the oultine shader

    update the light shader with the view and projection matrix

    for each light in the scene's lights
        if light is active
            draw the light with the light shader
        else
            reset the shader's uniforms for this light
```

Pseudo code bloom

```
taille kernel de 1 = [0.2, 0.4,  0.4]
                     [0.7, pixel 0.8]
                     [0.9, 0.2,  0.4]

taille kernel = nombre de pixels alentours à analyser

Definir (w,h) étant égal à la taille de la texture
blurColor sera notre texture flou
Pour i allant de -taille_kernel à taille_kernel :
    Pour j allant de -taille_kernel à taille_kernel :
        Definir les vecteurs (i, j) et (w,h)
        Les normaliser
        ajouter à blurColor texture(texture scène,textCoord + offset)


Ajouter ce qui a été calculé précèdement au pixel final

bloomColor = pixel + (blurColor divisé par (taille_kernel * taille_kernel + 1))

pixel final = bloomColor * l'intensitée à donner à la valeur finale du pixel
```