DÉPARTEMENT DU GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

INF8702
INFOGRAPHIE AVANCÉE

TP4 :
FBOS ET PROFONDEUR DE CHAMP
MAPPAGE D'OMBRES
MAPPAGE D'ENVIRONNEMENT ET IBL

OBJECTIFS DU LABORATOIRE

L'objectif de ce laboratoire final avant le projet est de se familiariser avec des techniques d'infographie plus avancées en incorporant principalement l'utilisation des *Frame Buffer Objects* (FBOs) pour plusieurs tâches ainsi que l'éclairage par mappage d'environnement (*Image-Based Lighting* – IBL). Ces techniques, bien que plus avancées que celles utilisées dans le TP2 sont néanmoins choses courantes dans l'industrie. On espère ici vous faire découvrir quelques utilités de ces outils et vous permettre d'apprécier l'étendue des effets possibles à réaliser en les appliquant.

Nous allons travailler avec le modèle 3D d'une des sculptures les plus connues au monde, soit la Vénus de Milo dont vous aurez deux versions ; l'une ayant un plus grand nombre de polygones pour plus de réalisme, et l'une plus rapide à charger en mémoire.

Figure 1 : Affichage initial de la Vénus de Milo.

MATÉRIEL

On met à votre disposition les choses suivantes :

- La solution Visual Studio intégrant un projet intitulé **TP4**
- Une nouvelle classe :
  - *CFBO.cpp*
- 6 paires de nuanceurs :
  - *modele3DSommets.glsl & modele3DFragments.glsl*
    - Responsable de l'éclairage des différents modèles 3D
  - *skyboxSommets.glsl & skyboxFragments.glsl*
    - Responsable de l'affichage du skybox
  - *gazonSommets.glsl & gazonFragments.glsl*
    - Responsable de l'affichage du skybox
  - *blurFXSommets.glsl & blurFXSommets.glsl*
    - Responsable de l'affichage d'un quad plein écran et de l'effet *depth of field* (DOF)

    - *shadowSommets.glsl & shadowFragments.glsl*
        - Responsable de la création de shadow maps
    - *debugSommets.glsl & debugFragments.glsl*
        - Permettent d'afficher les shadow maps si désiré.
- 4 nouveaux modèles 3D:
    - *venus.obj*
    - *venus-low.obj* (même chose que *venus.obj,* moins de polygones)
    - *sphere.obj*
    - *buddha.obj*
- Des nouvelles fonctionnalités clavier:
    - **F** : permuter l'activation du FBO principal
    - **B** : permuter l'activation de l'éclairage par IBL

DESCRIPTION DES TÂCHES

FBOS ET EFFET POST-RENDU (PROFONDEUR DE CHAMP)

Les Frame Buffer Objects (FBOs) sont des outils couramment utilisés dans toute application 3D voulant avoir un rendu intéressant. Le principe de base de l'utilisation des FBOs est très simple : on remplace la cible de rendu habituelle (écran) par un FBO ayant une ou plusieurs textures dans lesquelles on « sauvegarde » le rendu initial de la scène et possiblement plus d'informations pour chaque fragment (couleur, normale, réflectance profondeur, …). Ensuite on peut afficher un quad plein écran dans la cible de rendu par défaut et échantillonner certains attributs dans les textures fournies par le FBO afin de créer les effets désirés.

Dans notre cas, nous allons implémenter une version naïve et de la profondeur de champ, qui consiste à ajouter un flou gaussien aux fragments dont la profondeur est différente du fragment visé (au centre de l'écran), imitant le focus de nos yeux ou d'une caméra.

Nous aurons donc besoin de sauvegarder les composantes RGB et la profondeur de fragments dans deux textures du FBO afin de les échantillonner dans le nuanceur de fragments du quad plein écran.

1. Initialisation d'un FBO

La première étape consiste à créer un FBO ayant une texture pour les couleurs et une texture pour la profondeur. Vous devrez donc compléter les fonctions

- `CFBO::Init()`
- `initialisation()`

… afin de bien initialiser votre FBO pour avoir une texture pour les couleurs et une autre pour la profondeur. On vous invite à regarder le document FBOs.pdf fourni avec l'énoncé pour avoir plusieurs fonctions utiles. Plusieurs ressources en ligne, ou encore le *Red Book* contiennent des exemples de ce qui est attendu.

2. Rendu dans un FBO

Les différents nuanceurs de fragments pour les objets de la scène ont été modifiés pour vous afin d'écrire en sortie la couleur et la profondeur des fragments. Il vous reste donc compléter les fonctions

- `CFBO::CommencerCapture()`
- `CFBO::TerminerCapture()`

… afin de bien préparer OpenGL pour l'affichage dans le FBO. Ensuite, vous devrez modifier

- `blurFXSommets::main()`

... afin de passer au nuanceur de fragments du quad les bonnes coordonnées de textures, basées sur les positions des sommets du quad plein écran. Finalement, vous devrez modifier

- `DessinerScene()`

… afin de bien dessiner la scène dans le FBO, puis son contenu dans le quad plein écran.

Comportement observable :

Vous devriez, en appuyant sur « **F** » pour activer le FBO, avoir le même affichage qu'avant, à l'exception d'un point noir au milieu de l'écran :

Figure 2 : Affichage correcte avec FBO activé.

3. Profondeur de champ naïf :

Maintenant que vous affichez bien le contenu du FBO, on vous demande d'implémenter un effet de profondeur de champ naïf en appliquant du flou gaussien à deux échelles différentes sur les fragments dont la profondeur diffère de celle du fragment visé. Vous devez quantifier cette différence et appliquer le flou en conséquence. On vous offre deux les deux fonctions d'aide `LineariserProfondeur()` et `FiltreGaussien()` et il ne vous reste plus qu'à compléter

- `blurFXFragment::main()`

… afin d'utiliser ces fonctions et appliquer le filtre gaussien comme il se doit.

Comportement observable :

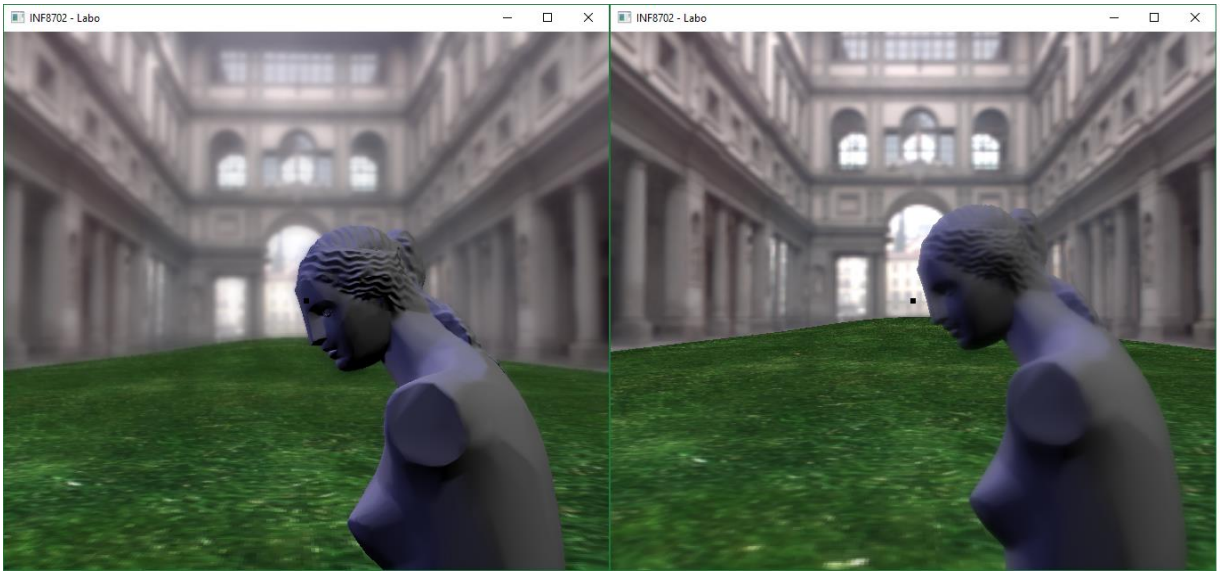

Figure 3 : Effet de profondeur de champ en regardant le modèle (gauche) et en regardant au loin (droite), avec le modèle haute résolution.

MAPPAGE D'OMBRES – *SHADOW MAPPING*

Le *shadow mapping* est une technique courante pour afficher des ombres cohérentes et dynamiques en temps réel, et qui fait usage des FBOs. Le concept pour une lumière peut s'expliquer selon les étapes suivantes (que vous aurez à compléter dans le code) :

1. Créer un FBO pour la lumière, ayant une texture pour stocker la profondeur.
2. Construire une matrice vue-projection relative à cette lumière (la partie « VP » de la matrice « MVP »).
3. Afficher les parties pertinentes de la scène selon le point de vue de la lumière (avec la bonne matrice modèle), et stocker la profondeur dans un FBO – le shadow map.
4. Lors de l'affichage de la scène, comparer la profondeur du fragment courant dans le référentiel de la lumière avec la profondeur dans le shadow map et affecter l'éclairage en conséquence.

1. Créer et initialiser les FBOs pour shadow maps

De la même manière qu'avec le premier FBO, on vous demande de compléter la fonction

- `initialisation ()`

… afin de bien initialiser les FBOs utilisés pour les shadow maps. La taille des textures des shadowMap n'est pas critique : on ne tient pas à avoir une résolution parfaite pour les ombres et réduire sa résolution peut améliorer les performances. On vous conseille ici d'utiliser `CCst::tailleShadowMap`.

2. Construire les matrices projectives de lumières

Afin de bien construire des shadow maps, il faudra dessiner la scène, ou certains modèles, du point de vue des chacune de nos 3 lumières. Pour y arriver vous aurez besoin de transformer des coordonnées du monde en coordonnées « clip » du point de vue des lumières. Vous aurez donc à peupler `LightVP[]` contenant les matrices Vue-Projection pour chaque lumière.

On vous demande donc de compléter la fonction :

- `construireMatricesProjectivesEclairage ()`

… afin de créer et sauvegarder ces matrices qui vous seront utiles à la prochaine étape. Des indices vous sont donnés en commentaires concernant les valeurs à utiliser dans les définitions des matrices Vue et Projection. Pour la lumière ponctuelle qui éclaire dans toutes les directions, comme nous n'avons qu'un seul modèle qui portera des ombres (la Vénus) on se contente ici des bien placer la lumière et de directement « regarder » vers le modèle.

3. Construire les shadow maps

Une fois les matrices projectives complétées, il vous faut maintenant peupler les shadow maps en dessinant la scène dans les bon FBOs. Vous devrez donc complétez la fonction :

- `construireCartesOmbrage ()`

… afin de fournir aux textures de profondeur des `ShadowMaps[]` les bonnes valeurs, relatives à chaque lumières. Vous devrez utiliser ici les bons nuanceurs (qui vous sont fournis).

Comportement observable :

Afin de déboguer cette partie, on met à votre disposition un moyen de visualiser le contenu des shadow maps sur un petit quad en bas à gauche de l'écran. Pour activer cette fonctionnalité, il vous faut fixer `afficherShadowMap` à `true` et choisir un `shadowMapAAfficher:`

Figure 4 : Les shadow maps pour lumière ponctuelle (gauche), spot (centre), et directionnelle (droite).

4. Ombrager pas fragment

Le nuanceur de gazon est le même que celui du TP2, dans lequel une fonction a été ajoutée afin de déterminer si le fragment courant est dans l'ombre. Cette fonction retourne toujours `1.0` pour l'instant et elle est utilisée afin de modifier l'éclairage diffus (l'ambiant reste intouché, et le gazon n'a pas d'éclairage spéculaire).

Il vous faudra donc compléter la fonction

- `gazonSommets.glsl :: main()`

… afin de donner au nuanceur de fragments les coordonnées en espace « clip » des fragments selon les points de vue des lumières. Ensuite, il ne vous restera plus qu'à compléter la fonction

- `gazonFragments.glsl :: Ombrage()`

… afin de calculer l'apport de l'ombrage sur le fragment.

Comportement observable :

Figure 5 : Ombre portée par la lumière directionnelle.

Figure 6 : Ombre portée par la lumière ponctuelle.

Figure 7 : Ombre portée par la lumière spot.

Une fois le calcul d'ombrage complété, il se peut que vous ayez des régions ombragées sans être obstruées par un modèle :

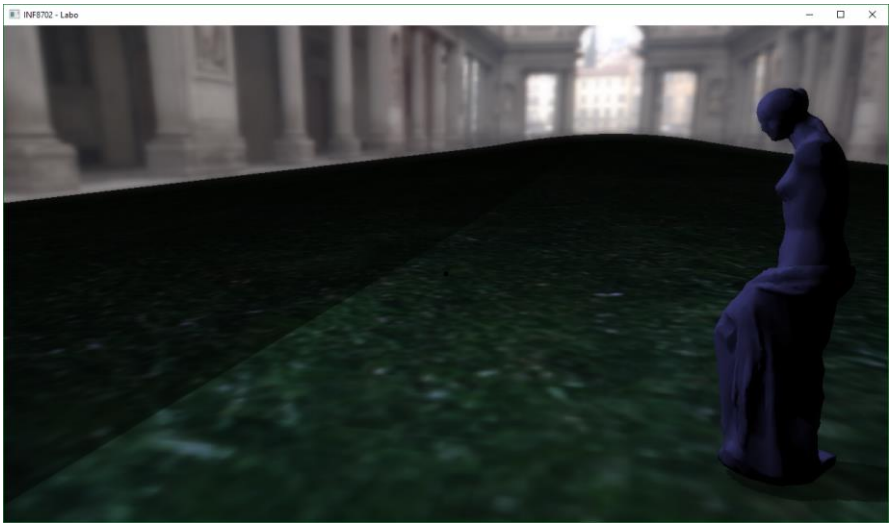

Figure 8 : Erreur d'ombrage avec la lumière ponctuelle.

Cela est dû au fait que certaines valeurs de profondeurs seront erronées si le fragment ne se trouvait plus loin que le plan « *far* » de la lumière ou en arrière de celle-ci. Dans ce dernier cas (qui arrive dans notre scène) la coordonnée homogène de la position en espace « clip » sera inférieure à 0.

Il vous suffit donc d'ajouter une condition à

- `gazonFragments.glsl :: Ombrage()`

… afin d'éliminer ces artéfacts.

Comportement observable :

Figure 9 : Affichage correct des ombres avec lumière ponctuelle.

MAPPAGE D'ENVIRONNEMENT ET IBL

L'illumination par image (*image-based lighting* ou IBL) est une technique d'illumination utilisant des *cubemaps* afin d'éclairer des modèles d'une scène. Cette technique peu remplacer les calculs d'éclairages utilisant les valeurs associées aux lumières OpenGL, tout en utilisant les propriétés des matériaux associées aux modèles. Souvent, on utilise des textures déjà en mémoires (comme celles du skybox courant) afin de colorer les modèles. Dans notre cas précis, la contribution spéculaire sera calculée avec l'apport de la texture du skybox, tandis que la contribution diffuse sera elle calculée avec l'apport d'une autre texture fournie pour cette tâche : « *uffizi_cross_LDR_diffuse.bmp* ». Ces textures sont liées aux nuanceurs des modèles 3D lors de l'affichage dans la fonction **`dessinerModele3D()`**.

Afin de mieux apprécier (et déboguer) votre implémentation du IBL, nous vous suggérons d'activer l'affichage de modèles supplémentaires comprenant une statue de Buddha, et une sphère avec peu de polygones en allant changer la valeur de **`afficherAutresModeles`** dans le fichier « `main.cpp` ».

La touche « **B** » vous permettra d'activer l'éclairage par IBL :

Figure 10 : Affichage des 3 modèles (sans IBL).

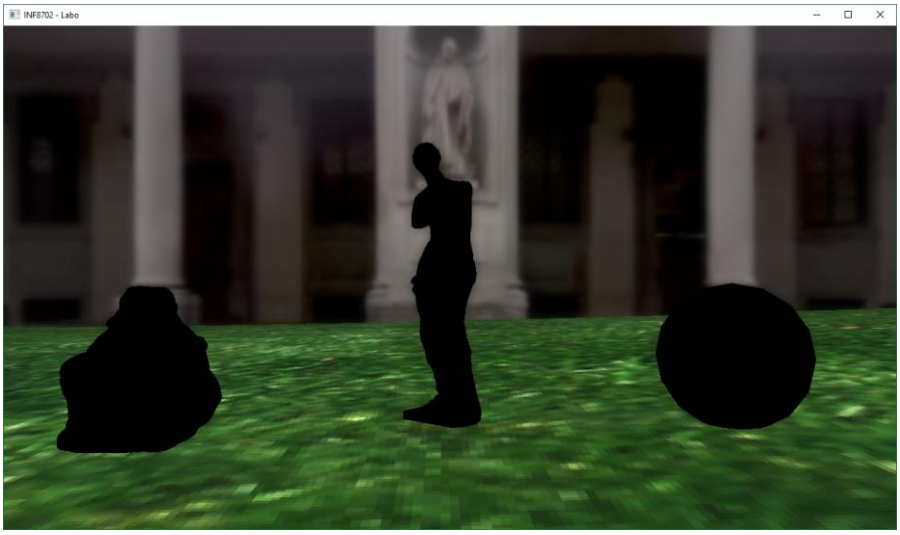

Figure 11 : Affichage avec IBL avant complétion des nuanceurs.

Il vous restera donc à compléter les fonctions

- `Modele3DSommets.glsl :: main ()`
- `Modele3DFragments.glsl :: lightingIBL ()`

… afin de passer les bonnes valeurs au nuanceur de fragment et calculer les contributions d'éclairage par IBL. Il est important ici d'utiliser dans ces calculs les coordonnées universelles (ou *world coordinates*) des modèles si l'on veut avoir un éclairage cohérent peu importe la position de la caméra et la position de l'objet. On fournit donc au nuanceur de fragment des modèles 3D la matrice Modèle afin d'exprimer les différents vecteurs dans ce référentiel.

Comportement observable :

Figure 12 : Affichage des 3 modèles avec IBL activé et fonctionnel.

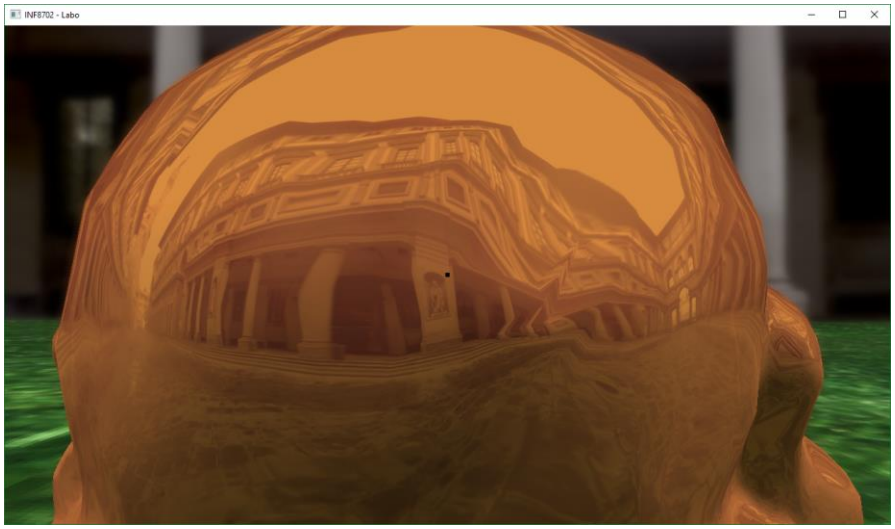

Figure 13 : Arrière de la tête du modèle de Buddha avec IBL.

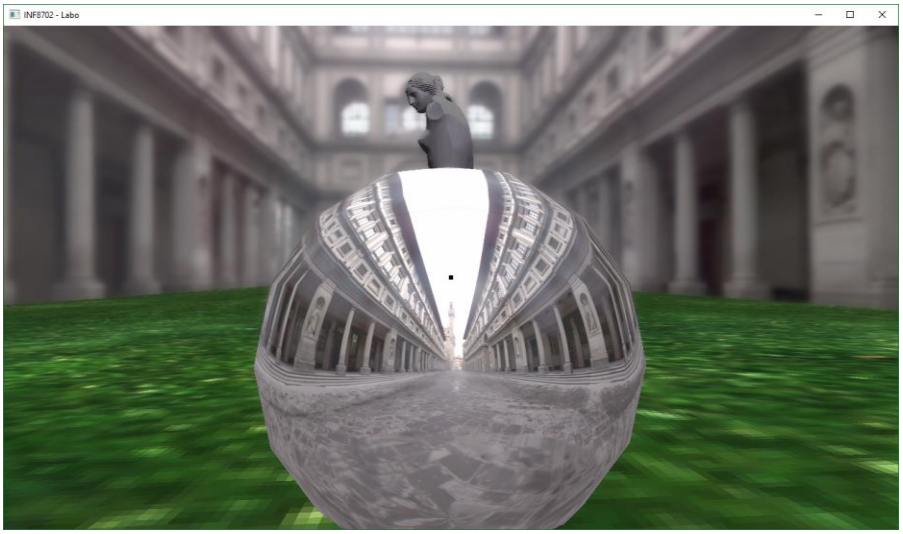

Figure 14 : Sphère éclairée par IBL en regardant vers le modèle de Vénus.

QUESTIONS

1. **On vous a fourni la fonction `lineariserProfondeur()`. Expliquez ce qu'accompli cette fonction.**
2. **On vous a aussi fourni la fonction `FiltreGaussien()`. Expliquez les détails de son fonctionnement.**
3. **Cette implémentation naïve du champ de profondeur effectue combien d'échantillonnage (*sampling*) de textures par fragment du quad plein écran ? Donner les détails sur une autre technique utilisée réduisant le nombre d'échantillonnage pour une même portée.**
4. **Avec notre implémentation présente du IBL, le gazon n'est pas visible dans les réflexions de l'environnement. Comment pourriez-vous régler ce problème en utilisant des FBOs ? Considérez ici un seul modèle 3D présent sur le gazon.**

BARÈME DE CORRECTION

Description des requis	**Points alloués**
Initialisation des FBOs	1.50
Utilisation/Affichage du FBO principal	1.50
Flou gaussien	1.00
Matrices projectives des lumières	1.00
Création des shadow maps	1.00
Ombres portées	1.00
IBL	2.00
Q1	1.00
Q2	1.00
Q3	1.00
Q4	2.00
Lisibilité du code	1.00
Total	**15.00**