Compte-Rendu TP GPU

Martin BLANCHARD, Anthony GULLIENT

8 Novembre 2019

1 Introduction

La programmation GPU est extrêmement utilisée dans notre monde actuel et nécessite plus que de savoir modéliser des objets. En effet, il faut savoir comment envoyer les informations correctement, les buffers à utiliser ainsi que les informations à spécifier pour un affichage optimal de notre scène car la finalité de la programmation GPU étant d'envoyer correctement ces informations à la carte graphique pour affichage. Le but des nombreux TPs de cette matière est de générer une scène 3D avec un personnage texturé en mouvement ainsi que de l'eau réaliste (en mouvement et avec des reflets).

Dans ce compte-rendu, nous allons décrire en détails les différentes étapes du processus d'affichage pour montrer la compréhension des différentes notions vues.

2 Résultats

2.1 Création des mesh, des textures

On crée un mesh qui va modéliser la surface de l'eau sous forme d'une grille. On texture ensuite cette grille avec la carte des normales de l'eau, une image spécialement faite pour la programmation à partir d'une vraie image d'eau. Cette texture va nous permettre d'obtenir une eau plutôt réaliste sans faire des calculs trop compliqués. Nous y reviendrons plus tard en évoquant le *normal mapping* dans la partie 2.5.

Le mesh de Frankie, notre célèbre écureuil, est déjà donné dans un fichier. Nous n'avons plus qu'à l'importer ainsi que sa texture grâce aux fonctions déjà prêtes pour ce faire.

Nous avons donc importé tout ce qu'il y avait à dessiner. Il est maintenant temps d'aller au coeur de la programmation GPU.

2.2 Création des shaders et des programmes

Nous allons maintenant créer des programmes pour gérer les différents éléments de notre scène.

Un programme permet d'associer des shaders à un élément de la scène pour pouvoir modifier sa géométrie, sa texture ou sa projection. Il est évident que pour N objets similaires dans une scène, 1 seul programme suffit si on souhaite qu'ils soient tous régis par les mêmes lois.

Dans notre scène actuelle, nous allons donc créer 2 programmes : un qui va gérer notre eau et l'autre qui va gérer Frankie. A chacun de ces programmes sont associés des shaders dont nous allons développer la fonction. Ils sont appelés dans l'ordre ci-dessous :

1. Le Vertex Shader : il est appelé une fois par sommet et permet de modifier sa projection ou ses coordonnées.
2. Le Geometry Shader : il permet de modifier la géométrie du mesh en profondeur. Un geometry shader peut par exemple prendre des points en entrée et donner des triangles en sortie, ou l'inverse. Nous n'utilisons pas ce shader dans ce cas-ci mais il reste très utile pour les tranformations géométriques.
3. Le Fragment Shader : il permet de modifier la couleur des sommets. Il est très utile pour l'éclairage et pour l'apoplication de texture sur un mesh.

Concernant le geometry shader, nous avons pu l'utiliser dans l'exemple suivant : une scène formée d'un objet (Frankie l'écureuil) auquel nous avons appliqué une texture, nous voulons alors avoir une vue éclatée de Frankie. Pour cela, nous utilisons le geometry shader qui va nous permettre d'agir sur les triangles formant l'objet et ainsi pouvoir les déplacer en fonction des normales mais aussi du temps. Le résultat est ensuite envoyé au fragment shader qui aura pour but d'appliquer la texture/couleur aux triangles.

On obtient les résultats suivants :

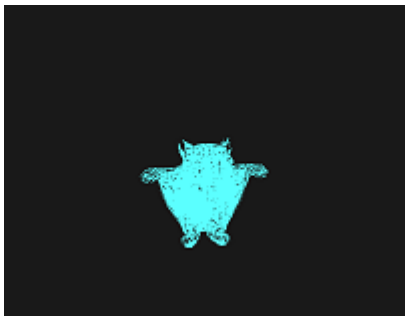

FIGURE 1 – Vue éclatée 1 de Frankie

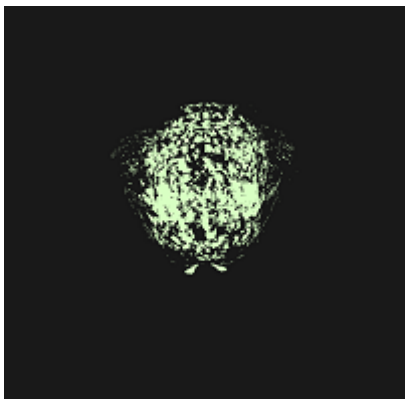

FIGURE 2 – Vue éclatée 2 de Frankie

Retour à notre scène actuelle.

Le vertex shader associé à l'eau projette donc notre grille de liquide dans notre plan. Le fragment shader va appliquer la texture de l'eau puis gérer les normal map (cf Partie 2.5) ainsi que l'éclairage de l'eau (on utilise ici l'éclairage

de Phong pour obtenir une texture plutôt réaliste). On va également donner l'illusion d'un mouvement d'eau en modifiant périodiquement les coordonnées de notre texture dans ce shader.

Le vertex shader de Frankie va projeter notre écureuil dans le plan de notre scène. Nous l'utilisons au début pour faire également bouger notre ami en fonction du temps mais nous allons rapidement adopter une autre alternative pour gérer son mouvement (cf Partie 2.6). Son fragment shader va appliquer la texture sur le mesh de notre animal.

On lie ces shaders aux bons programmes avant de passer à la création de buffers pour tranférer les informations à la carte graphique.

2.3 Buffers "basiques"

On va créer des VBOs afin de tranférer nos données au Frame Buffer.

Un Frame Buffer est un buffer qui va nous permettre d'afficher correctement la scène. Ce buffer est muni de 2 buffers : le color buffer et le Depth Buffer. C'est le Depth Buffer qui est appelé en premier. Il va décider des pixels à afficher ou non selon leur profondeur. Le Color Buffer est appelé ensuite pour gérer la couleur des pixels avant l'envoi à la carte graphique. Le Frame Buffer est présent par défaut sur OpenGL. C'est le seul buffer par lequel passer si on veut afficher quelque chose.

On crée 2 VBOs pour stocker notre Frankie ainsi que notre eau. On définit la taille de ces buffers grâce au type de données stockées (ici, des floats).

On utilise également dans notre scène l'Element Buffer Object (EBO). Un EBO est un buffer permettant d'associer des vertex à des indices, c'est-à dire que nous ne sommes plus contraints de devoir fournir plusieurs fois le même vertex lors de la création de forme geometrique. Par exemple 4 vertex suffisent pour un rectangle.

D'autres buffers sont également utilisés pour réaliser des fonctions plus complexes. Nous allons les évoquer dans la partie suivante.

2.4 Le FBO

Le FBO est un autre buffer très utilisé en programmation GPU. Il ne permet aucun affichage sur la scène mais gère un affichage en coulisses. Comme le Frame Buffer évoqué précédemment, il possède un Color Buffer et un Depth Buffer. Expliquons plus clairement : le FBO permet de créer une scène qui ne sera pas affichée à l'écran. Une telle description amène donc une question : à quoi sert ce buffer s'il ne permet aucun affichage ? L'explication est simple : ce buffer nous permet de stocker une vue de notre scène sous la forme d'une texture 2D... que l'on pourra appliquer dans notre scène visible. Montrons comment cela marche par un schéma.

Dans l'exemple ci-dessous, nous créons dans le Frame Buffer, une scène avec un tableau vide. Dans le FBO, nous allons créer une scène que l'on affichera sur le tableau ensuite, comme une texture.

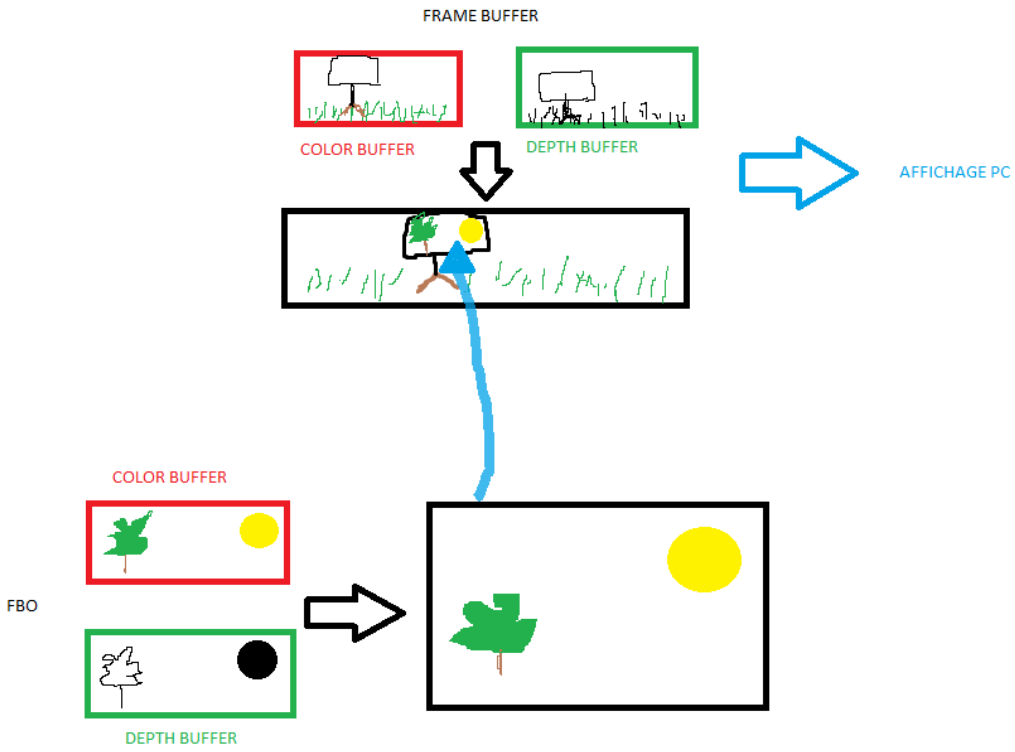

FIGURE 3 – Illustration FB et FBO

On pourrait utiliser le FBO pour produire une eau encore plus réaliste en produisant un reflet de la scène dessus. En utilisant 2 photos de la scène sous forme de FBO, on peut produire une texture de réflection et une de réfraction afin de donner un reflet très réaliste des différents éléments dans l'eau. En effet, on pourrait ensuite distordre ces 2 textures puis les fusionner pour créer le reflet en utilisant des Du/Dv map ainsi que l'effet Fresnel pour rendre réaliste le changement d'angle de caméra (l'eau aurait l'air plus réfractive quand la camera serait à sa surface et moins quand la caméra serait au dessus). Nous n'avons malheureusement pas eu le temps de traiter cet exemple dans notre scène.

Nous avons néanmoins trouvé un moyen de rendre l'eau beaucoup plus réaliste, que nous allons détailler dans la partie suivante.

2.5 Normal maps

Pour le moment, nous avons modélisé notre eau en appliquant la texture donnée sur un quad (ici une grille). Cette texture affiche les normales directement mais d'une façon peu réaliste (Figures 4 et 5).

FIGURE 4 – Normales avant modification

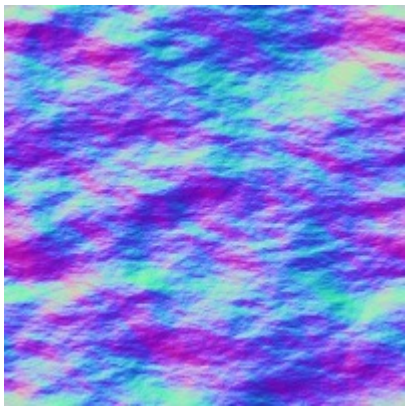

FIGURE 5 – Texture avant modification des normales

Nous voulons maintenant obtenir un rendu plus réaliste de notre scène. Pour cela, il nous faut ajouter de l'éclairage et nous allons donc utiliser la normal mapping.

Cette méthode nous permet de générer, à partir d'une texture de normales, un éclairage avec réflexion de la lumière sur la texture et donc d'obtenir un effet tridimensionnel. Cette texture dite «normal map» est nécessaire pour obtenir un éclairage réaliste puisque si nous voulions calculer les normales en tout point d'une texture basique, nous aurions obtenu des vecteurs de normales colinéaires pointant tous dans la même direction et donc un éclairage uniforme sur toute la texture.

Pour résoudre ce problème, nous allons associer les valeurs des couleurs de la normal map aux coordonnées des normales. C'est-à-dire qu'en tout point, la normale associée aura pour coordonnées les valeurs R, G, B en ce point. Nous obtenons ainsi des directions différentes pour chaque normale de la texture et donc un rendu plus réaliste qu'en Figure 4 :

FIGURE 6 – Normales après modification

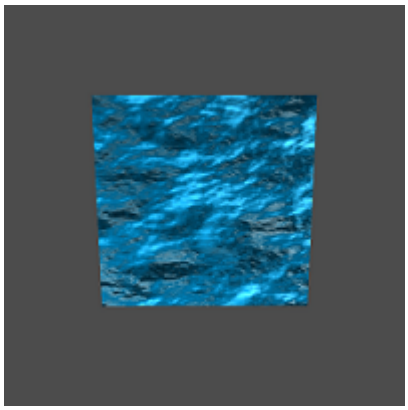

FIGURE 7 – Texture après modification des normales

Pourtant ces normales ont toutes le même sens puisque les valeurs R, G, B sont positives, nous les recentrons alors entre -1 et 1 pour obtenir un rendu plus réaliste.

Maintenant que nous disposons des normales, nous pouvons calculer la couleur obtenue avec l'éclairage. Dans notre cas, nous utilisons l'illumination de Phong générant 3 composantes : l'ambiant, le diffus et le spéculaire Le résultat correspond à la couleur à appliquer en chaque point.

Nous allons maintenant nous intéresser à la gestion du mouvement de notre écureuil, qui peut être améliorée avec le Transform Feedback.

2.6 Transform Feedback

Depuis le début du TP, nous gérons le mouvement de Frankie dans le vertex shader, ce qui est loin d'être optimal. En effet, nous ne pouvons récupérer les coordonnées de notre écureuil qu'à l'instant t et nous effectuons de plus la transformation dans son vertex shader en fonction du temps... Impossible, donc, de calculer correctement sa vitesse ou son accélération, ou plus généralement d'avoir des informations à l'instant t-1.

Le Transform feedback va justement nous permettre d'obtenir ces informations et de récupérer correctement le tout dans notre programme, au lieu de passer par le vertex shader. On le crée directement en lui associant le vertex shader qui réalise la transformation voulue. nous allons créer ensuite 2 VBOs : l'un pour le transfert des données au vertex shader, l'autre pour recueillir les données après transformation - afin de les envoyer à un autre programme qui sera celui associé à Frankie. En effet, le Transform Feedback nous permet d'effectuer des modifications sur les vertex en utilisant un vertex shader puis de récuperer et stocker le résultat dans un VBO dédié en sortie de ce vertex shader

et enfin de pouvoir utiliser ce VBOpour l'envoyé à un second vertex shader permettant la génération de la scène. Les données vont donc passer par un buffer supplémentaire mais permettront le stockage de l'étape n-1.

Cela permet de plus de libérer le vertex shader associé à Frankie. Le Tranform Feedback pourrait nous permettre de faire plus que simplement mettre le mouvement de Frankie dans un autre vertex shader. Nous pourrions par exemple calculer la vitesse de Frankie pour calculer une possible intersection avec un plan ou une sphère à la prochaine itération, grâce à cette technique. Nous n'avons pas eu le temps d'implémenter un calcul de vitesse mais nous avons néanmoins réalisé le reste.

Toutes les étapes précédentes nous permettent ainsi d'obtenir la scène suivante :

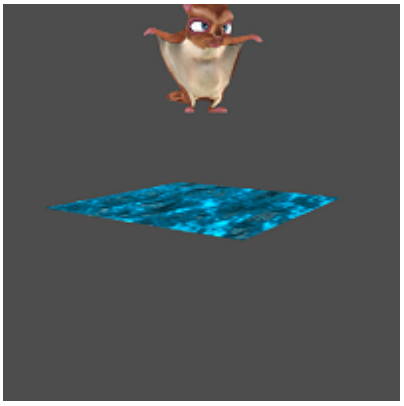

FIGURE 8 – Scène finale du TP

Nous avons donc pu utiliser tous les outils mis à notre disposition pour modéliser une scène 3D plutôt simple. D'autres existent, cependant comme les compute shaders, que nous allons évoquer sans les implémenter.

2.7 Compute shader

Dans des scènes bien plus avancées, les programmeurs utilisent ce qu'on appelle les Compute Shader. Différents des shaders évoqués en partie 2.2, ceux-ci permettent de réaliser beaucoup d'actions diverses et d'optimiser la modélisation. Cette structure est plutôt hors du pipeline graphique et n'a pas d'entrées/sorties fixes. On peut donc lui envoyer quasiment tout ce que l'on veut pour effectuer des opérations. Il fonctionne de plus en utilisant les sources de mémoire du GPU.

Son fonctionnement le plus novateur est également de pouvoir paralléliser les opérations en divisant les sources de mémoire en "groupes de travail" qui vont travailler sur différentes opérations ou des étapes parallèles d'une même action. Le but est d'alléger le GPU et d'optimiser au maximum l'espace mis à

notre disposition, de sorte à pouvoir effectuer des modélisations plus détaillées sans ralentir l'ordinateur.

Nous n'avons pas pu mettre en place ce compute shader, par choix de nous concentrer sur une modélisation poussée afin de bien connaître les outils de la programmation GPU avant de voir optimiser.

3 Conclusion

En conclusion, nous avons réussi à modéliser une scène plutôt détaillée et en utilisant la majorité des outils mis à notre disposition. Ceux qui n'ont pu être mis en place ont tout de même été compris et certains ont commencés à être implémentés.

Comme dit précédemment, la prochaine étape du projet serait de créer des reflets réalistes dans l'eau (avec les Du/Dv maps, les FBOs et l'effet Fresnel), puis de travailler sur l'optimisation de la scène grâce aux compute shaders pour obtenir une modélisation fluide.