INF2705 Infographie

Travail pratique 2
Transformation matricielle et textures

Département de génie informatique et génie logiciel
Polytechnique Montréal
Tristan Rioux, Automne 2023

Table des matières

1 Introduction

Ce document décrit les exigences du TP2 « *Transformation matricielle et textures* » (Automne 2023) du cours INF2705 Infographie.

1.1 But

Le but des travaux pratiques est de permettre à l'étudiant de directement appliquer les notions vues en classe.

1.2 Portée

Chaque travail pratique permet à l'étudiant d'aborder un sujet spécifique.

1.3 Remise

Faites la commande « `make remise` » ou exécutez/cliquez sur « `remise.bat` » afin de créer l'archive « **INF2705_remise_TPn.zip** » (ou .7z, .rar, .tar) que vous déposerez ensuite dans Moodle. (Moodle ajoute automatiquement vos matricules ou le numéro de votre groupe au nom du fichier remis.)

Ce fichier `zip` contient tout le code source du TP (`makefile`, `*.h`, `*.cpp`, `*.glsl`, `*.txt`).

2 Description globale

2.1 But

Le but de TP est de permettre à l'étudiant de mettre en pratique les concepts de transformation et d'utiliser des textures. Il sera en mesure d'effectuer diverses transformations afin d'afficher plusieurs objets à l'écran de façon simultanée dans différentes vues et perspectives. Il sera aussi capable d'utiliser des textures afin d'ajouter des détails aux objets. Le travail fera l'utilisation des fonctions d'OpenGL `glUniform3f` et `glUniformMatrix4fv` et les fonctions de glm `glm::translate`, `glm::rotate`, `glm::scale`, `glm::lookAt`, `glm::ortho` et `glm::perspective`.

2.2 Travail demandé

Le tp2 est une continuation du tp1, vous pouvez donc reprendre les mêmes fichiers de projet. Cependant, il va falloir retirer quelques bouts de code et rajouter certains fichiers au projet. Vous pouvez retirer les fonctions `changeRGB` et `changePos`, les formes de la partie 1 (on garde le cube qui tourne), le chargement des shaders basic et color (vous pouvez garder les .glsl), la selection de la forme avec la touche T. Vous devez aussi retirer toutes les variables qui ne sont plus utilisées (dans `vertices\_data.h` notamment). La main loop devrait que contenir l'utilisation du shader de transformation, les assignations des matrices et le dessin du cube.

Dans les fichiers à ajouter, il y a une mise à jour de la classe Window, une classe Model avec une fonction pour charger les .obj, une fonction pour le chargement des fichiers d'images, ainsi qu'un fichier de fonctions utilitaires. Ceux-ci utilise les librairies "OBJLoader" et "stb_image" qui sont du format "Header only". Il y a aussi des squelettes de classe de Camera, Model3d, et Texture2D que vous pouvez utilisé.

Pour être compatible avec les modèles .obj qui peuvent avoir beaucoup d'indexes, il faudra faire une modification à la classe `BasicShapeElements`. On changera les types `GLubyte` par des `GLuint`. N'oublier pas la méthode `draw` et de modifier le type des tableaux (celui du cube notamment). Il faudra aussi faire un constructeur par défaut qui ne fait que créer les vao, vbo et ebo. On ajoutera une méthode `setData` pour passer les données comme l'ancien constructeur du tp1.

Partie 1 : création d'une scène graphique

Maintenant que vous êtes à l'aise avec les bases d'OpenGL, nous allons commencer à dessiner une scène 3D. Une fonction vous est donnée pour faire le chargement des modèles 3D.

Avant de peupler la scène, il sera important de créer une caméra pour pouvoir facilement voir les objets. Une caméra en première personne et en troisième personne devront être implémentées. Utiliser le cube qui tourne du tp1 pour vous aider à débugger. Les touches W-A-S-D du clavier seront pour le déplacement, alors que le mouvement de la souris sera utilisé pour modifier l'orientation en x et en y.

Pour la caméra en première personne, vous devez faire les translations et rotation directement, alors que pour la caméra en troisième personne, vous devez utiliser `glm::lookAt`. Lorsqu'on change la caméra en troisième personne, vous devez dessiner le personnage (suzanne le singe). On utilisera le scroll de la souris pour alténer entre les deux.

Pour la caméra en première personne, il vous faudra faire des rotations selon l'orientation du personnage et une translation selon la position du personnage. Pour la caméra en troisième personne, l'utilisation de lookAt et des coordonnées sphériques sera utile. On utilisera un rayon de 6 pour la position de la caméra. On décalera la position de la caméra et du point observé par la position du joueur. La position et orientation du joueur sont passé par référence à la création.

Lorsque la classe Camera sera complétée, vous pouvez retirer le cube et faire le chargement des modèles 3d en complétant la classe Model. Vous pouvez charger les positions et indexes des modèles avec la méthode `loadObj`. Vous allez avoir besoin d'utiliser une instance de `BasicShapeElements` pour contenir les données et les dessiner. Vous allez avoir besoin d'un shader pour dessiner les modèles. Pour le moment, le seul attribut en entré du vertex shader sera la position. Celui-ci aura un uniform pour la matrice mvp. Pour le fragment shader, on aura un second uniform de type vec3 pour donner la couleur de sortie. Vous pouvez faire les modifications dans le shader de transformation du tp1 (que vous devriez probablement renommé comme bon vous semble, model dans mon cas).

Dans notre scène 3d, nous allons répartir des groupes d'objets sur un terrain. Un groupe sera composé d'un arbre, d'un roché et d'un champignon. Les groupes ont une position et une rotation en Y aléatoire. Pour le moment, les modèles seront seulement d'une couleur quelconque par un uniform pour pouvoir les différenciers (dans la partie 2, ils pourront être texturés).

Vous allez devoir :

- Créer un plane carré dans le plan X-Z centré en (0, 0, 0) de taille 60x60 pour faire le sol.
- Créer un plane rectangulaire sur un des côtés du sol de taille 60x20 pour faire un ruisseau.
- Répartir 25 groupes d'objets dans la scène.
- Les groupes possède une transformation de translation, rotation en Y aléatoire entre [0 ; 2π] et mise à l'échelle aléatoire entre [0.7 ; 1.3].
- Un groupe contient un arbre centré en (0, 0, 0) de la position du groupe. Il doit aussi avoir une mise à l'échelle d'une valeur aléatoire entre [0.7 ; 1.3].
- Un groupe contient un roché placé à un angle aléatoire entre [0 ; 2π] sur un cercle de rayon entre [1 ; 2] centré à la position du groupe.
- Un groupe contient un champignon toujours placé à la base de l'arbre à (0.3, 0, 0.3). La translation doit suivre la mise à l'échelle de l'arbre.
- Vous ne devriez pas avoir de trigonométrie dans le code des transformations des objets, utilisez les matrices de rotation efficacement.
- Toutes les multiplications de matrices devront être fait sur le cpu. Je vous recommande de calculer les matrices qui risque d'être réutiliser, par exemple la multiplication matriceProjection * matriceVue qui peut être fait une seule fois et être réutilisé pour les prochains calculs. C'est un peu l'équivalent de push/pop matrix vu en cours.

Pour la matrice de projection, nous allons continuer avec la matrice de projection du tp1 avec un fov de 70, le aspect ratio de la fenêtre, un near plane à 0.1, mais un far plane à 200 cette fois-ci. Il sera aussi possible de modifier cette matrice pour une matrice orthogonal afin de dessiner directement

dans l'écran. Pour le moment, on dessinera tous simplement un carrée de couleur en coin de l'écran.

De plus, les modèles 3d ont tendances à cacher certaines de leur faces. Ce phénomène est tout à fait normal, dû à la projection d'une forme 3d dans le plan de l'écran. Cependant, les fragments de ces primitives cachées seront encore calculer, ce qui est dégradant pour les performances. On peut facilement y remédier en activant le face culling sur les faces arrières.

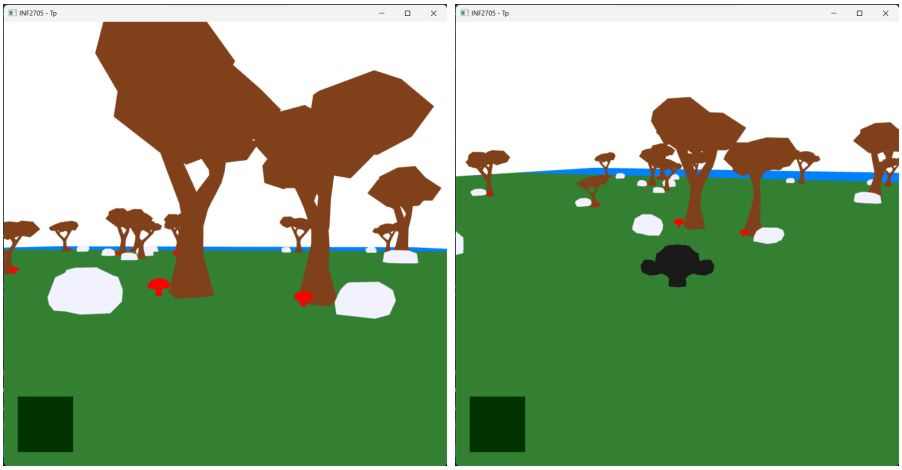

FIGURE 1 – La scène vue en fps et en tps.

Partie 2 : utilisation de texture

Malgré les nouveaux ajouts que nous avons fait à la scène, elle manque clairement de détail. Une technique couramment utilisée pour facilement rajouter du détail est d'utiliser des textures.

Pour se faire, il va falloir charger les textures de chaques modèles et les appliquer sur ceux-ci lorsqu'il sera temps de les dessiner. Une fonction de chargement des images vous est fournis. Vous devez complété la classe de Texture2D. Vous allez devoir modifier votre shader de la partie 1 pour accepter l'attribut des coordonnées de texture dans le vertex shader. Celui-ci devra être passé au fragment shader qui l'utilisera avec texture2d pour donner la couleur du fragment. N'oublier pas d'ajouter un uniform sampler2d dans ce dernier.

Pour les modèles 3D, l'attribut de coordonnée de texture est déjà défini, mais il vous faudra ajouter celui-ci pour le terrain et la rivière. Leurs textures sont des textures dites seamless. Elles sont idéales pour couvrir une très grande région tout en ayant des textures relativement petites en espace mémoire. Pour ce faire, il faut nécessairement charger cette texture dans le mode répétition pour pouvoir couvrir la totalité de la surface.

Pour le terrain, la texture couvrira le dixième d'une arête du plane (elle sera répété 10 fois sur le long d'une arète).

Pour la rivière, on créera un nouveau vertex shader pour faire le déplacement des coordonnées de texture en fonction du temps qui devrait être passé en uniform dans celui-ci. Le temps est obtenable à l'aide de la méthode getTime de la classe Window. Comme le terrain, la texture est répété et prend la totalité de la largeur, mais couvre seulement le dixième de la surface sur la longueur.

On peut aussi mettre une texture sur le fond pour remplacer la couleur uni qu'on a défini avec glClearColor. On utilisera ce qu'on appel un skybox. Un skybox est un cube qui englobe la totalité de la scène du point de vue de l'écran. On utilise une texture qui contient la projection d'un décor lointain et de nuage pour donner l'impression d'être dans un dôme. Il y a donc 6 textures à charger et des paramètres spécifiques pour pouvoir générer une seule texture de skybox. Les coordonnées de texture seront un vec3 pour pouvoir sélectionner la bonne face. Pour être capable de dessiner le skybox, une matrice spécial devra être passé au shader.

Partie 3 : LOD avec billboard et gazon

Maintenant qu'on a plusieurs éléments graphiques à l'écran, il serait pratique de ne pas utiliser toutes les ressources de l'ordinateur pour les éléments lointains, qui ne couvre déjà pas beaucoup d'espace à l'écran. Le niveau de détail n'a pas besoin d'être aussi élévé que pour les éléments près de l'observateur. Pour ce faire, une technique très commune est d'utiliser des LOD (level of details). Souvent, il est question de réduire la résolution de l'image, réduire le nombre de vertices dessinées, changer pour un shader moins coûteux ou encore en utilisant des billboards. Les billboards sont souvent utilisés pour réduire le détail des arbres. C'est ce que nous allons faire ici.

Lorsqu'un arbre aura une distance supérieur à 50 par rapport à la caméra, on ne dessinera pas le modèle 3d, mais plutôt un plane texturé d'une projection du modèle 3d. Le billboard sera toujours orienté vers la caméra.

Puisque la texture aura des texels transparent, il faudra ajouté un discard du fragment dans le fragment shader pour éléminer les fragments dont le alpha est en dessous de 0.3.

Pour finaliser ce tp, on ajoutera du gazon "3d" au sol. On utilisera pas de billboard ici, mais plutôt ce que j'aime bien appeler des "x shapes".

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

E1. TBD/À suivre.

Partie 2 :

E2. TBD/À suivre.

Partie 3 :

E3. TBD/À suivre.

3.2 Exigences non fonctionnelles

De façon générale, le code que vous ajouterez sera de bonne qualité. Évitez les énoncés superflus (qui montrent que vous ne comprenez pas bien ce que vous faites !), les commentaires erronés ou simplement absents, les mauvaises indentations, etc. [2 pts]

ANNEXES

A Liste des commandes

Touche	**Description**
`ESC`	Quitter l'application
`t`	Change la forme qui est dessinée