POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

INF2705 Infographie

Travail pratique 2

Transformation matricielle et textures

Département de génie informatique et génie logiciel
Polytechnique Montréal
Tristan Rioux, Hiver 2024

Table des matières

1 Introduction

Ce document décrit les exigences du TP2 « *Transformation matricielle et textures* » (Hiver 2024) 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 sélection 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, Model, Texture2D et TextureCubeMap que vous pouvez utiliser. Le fichier `utils.h` contient quelques indications supplémentaires sur la structure de votre main. **Attention : certains fichiers fournis doivent être "merge".**

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). Un constructeur par défaut qui ne fait que créer les vao, vbo et ebo devra être implémenté. 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.

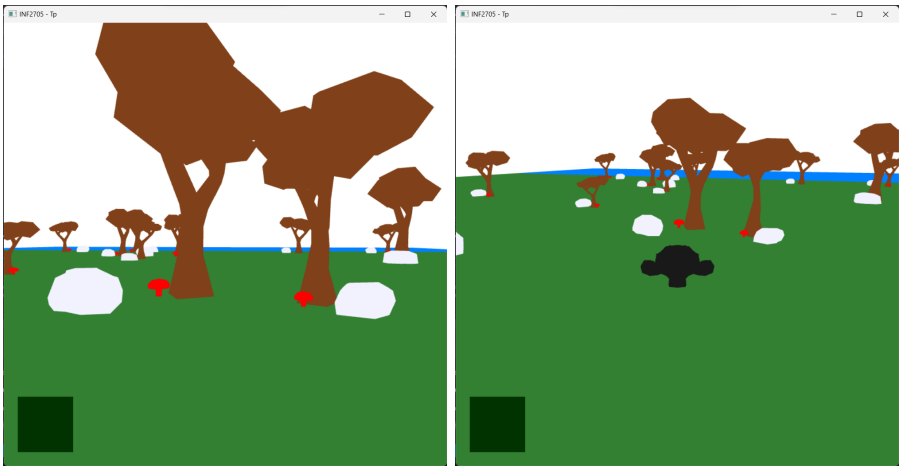

FIGURE 1 – La scène vue en FPS et en TPS.

Partie 1a : Caméra FPS et TPS

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. Utilisez 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. On utilisera le scroll de la souris pour alténer entre les deux modes de vue.

Penser à orienter le déplacement dans le référenciel du joueur (donc par rapport à son orientation en Y). Pour le changement de position et d'orientation, il va falloir appliquer des multiplicateurs pour réduire la vitesse un peu. Les valeurs sont libres à votre choix et confort (de mon côté, j'ai 0.05 et 0.01 respectivement).

La position et orientation du joueur sont passées par référence à la création de la caméra.

Pour la caméra en première personne, il faudra faire des rotations selon l'orientation du personnage et une translation selon la position du personnage (`glm::lookAt` ne sera pas accepté).

Pour la caméra en troisième personne, l'utilisation de `glm::lookAt` et des coordonnées sphériques sera utile. On utilisera un rayon de 6 pour la position de la caméra. Puisqu'on voudra initialement avoir la caméra derrière le personnage, un décalement de $\pi/2$ sera appliqué sur l'orientation. La position de la caméra et du point observé devront suivre la position du personnage.

Lorsqu'on utilise la caméra en troisième personne, vous devez dessiner le personnage (suzanne le singe). Le modèle doit être descendu à la hauteur du sol ($y = -1$), avoir une rotation de π en y et avoir une taille réduite de facteur 2 en plus des rotations et translation pour le déplacer selon l'orientation et position.

Partie 1b : Chargement des modèles

Lorsque la classe `Camera` sera complétée, vous pouvez retirer le cube et faire le chargement des modèles 3D en complétant la classe `Model`.

Une méthode vous est donnée pour faire le chargement des modèles 3D (`Model::loadObj`).

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. Attention, le nouveau constructeur par défaut implémenté devra être utilisé (n'appeler pas directement le constructeur).

Un nouveau shader devra être fait pour dessiner les modèles (`model.*.glsl`). Basez-vous sur le shader de transformation du tp1. Pour le moment, le seul attribut en entrée 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.

Partie 1c : Répartition des objets dans la scène

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 de votre choix par un uniform pour pouvoir les différenciers (dans la partie 2, ils seront texturés).

Vous allez devoir :

- Créer un plan carré dans le plan X-Z centré en $(0, -1, 0)$ de taille 60x60 pour faire le sol. Utiliser la classe `BasicShapeElements` comme au tp1 pour vous faire un modèle.
- Créer un plan rectangulaire sur un des côtés du sol de taille 60x20 pour faire un ruisseau. Utiliser la classe `BasicShapeElements` comme au tp1 pour vous faire un modèle.
- Répartir 49 groupes d'objets dans la scène (tableau de 7x7). Dans le code, les groupes ne sont représentés que par leur matrice. Vous devrez initialiser ces matrices dans l'initialisation du programme.
- Les groupes possèdent une transformation de translation qui est donnée par `getGroupRandomPos`, de rotation en Y aléatoire entre $[0; 2\pi]$ et mise à l'échelle aléatoire entre $[0.7; 1.3]$. Les groupes doivent être sur le sol.
- 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 $[0; 2\pi]$ sur un cercle de rayon entre $[1; 2]$ centré à la position du groupe. Vous allez possiblement vouloir le sortir un peu plus du sol avec une translation de 0.2 en Y. Une mise à l'échelle constante de 0.3 sera appliquée.
- Un groupe contient un champignon toujours placé à la base de l'arbre à $(0.3, 0, 0.3)$. Cependant, la translation doit suivre la mise à l'échelle de l'arbre. Une mise à l'échelle constante de 0.05 sera appliquée.
- **Vous ne devriez pas avoir de trigonométrie dans le code des transformations** des objets et des groupes, 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.
— Vous avez accès à la fonction `rand01` pour avoir une valeur aléatoire entre $[0; 1]$.

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.

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 calculés, ce qui dégrade les performances. On peut facilement y remédier en activant le face culling sur les faces arrières.

Partie 1d : HUD

Il sera aussi possible de modifier la matrice de projection pour une matrice orthogonale afin de dessiner directement dans l'écran. Pour le moment, on dessinera tout simplement un carré de couleur en coin de l'écran.

Il est libre à vous de définir la matrice orthogonale, tant que le carré garde les proportions au changement de la taille de la fenêtre (commenter `SDL_SetRelativeMouseMode(SDL_TRUE)` dans `window.cpp` pour avoir la souris visible).

Dans l'image de l'énoncé, il a une taille de 100 et un décalement de position du $3/4$ de sa taille en x et y dans le référenciel de l'écran. D'autres valeurs seront acceptées (puisque cela dépendent de vos définitions), tant que le résultat est semblable. Puisque le carré est dans l'écran, il faudra le dessiner en dernier en faisant toujours passé le test de profondeur pour ne pas qu'il soit impacté par les éléments 3D.

Partie 2 : Utilisation de textures

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.

Remarquer que les textures des modèles 3D ont de l'illumination de précalculer dedans, ce qui ajoute beaucoup de détails.

Essayer de dessiner les objets en changeant le moins souvent l'état d'OpenGL (les shaders et textures notamment).

FIGURE 2 – La scène après avoir été texturée.

Partie 2a : Chargement 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 avec la méthode `use`. Une classe `Texture2D` contenant du code pour le chargement des images vous est fournis. Vous devez complété son constructeur pour charger l'image en tant que texture, ainsi que les autres méthodes pour utiliser la texture.

Vous allez devoir charger les textures suivantes et les appliquer sur les objets respectifs :

"`suzanneTexture.png`", "`treeTexture.png`", "`rockTexture.png`",

"`mushroomTexture.png`", "`groundSeamless.jpg`", "`waterSeamless.jpg`", "`heart.png`".

On ne veut pas que les textures se répètent sur ces objets.

Pour les modèles 3D, l'attribut de coordonnée de texture est déjà défini dans le .obj. Il faudra charger les coordonnées de texture (en décommentant les lignes de la méthode de chargement) et activer l'attribut dans le `BasicShapeElements`.

Partie 2b : Utilisation de texture dans les shaders

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 la fonction texture pour donner la couleur du fragment. Il ne sera nécessaire que d'utiliser la couleur de la texture. Le alpha de la couleur de sortie sera à 1, puisque le blending n'est pas activé. N'oublier pas d'ajouter un `uniform sampler2d` dans le fragment shader. Vous pouvez retirer l'uniform de couleur, puisqu'il ne sera plus utilisé.

Partie 2c : Répétition de texture

Il faudra faire de même pour les autres géométries que vous avez défini en ajoutant les coordonnées de texture et activer l'attribut. Le terrain et la rivière possèdent 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. De plus, il faudra activer le mipmap pour ces textures spécifiquement afin de retirer les artéfacts de rendu.

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

Partie 2d : Animation de texture

Pour la rivière, on créera un nouveau shader program (`water.*.glsl`) pour faire le déplacement des coordonnées de texture en fonction du temps en seconde qui devrait être passé en uniform dans celui-ci. Le temps est obtenable à l'aide de la méthode `getTick` de la classe `Window`, qui retourne le temps en milliseconde depuis le début de l'exécution (pensez à faire une division par 1000 pour l'avoir en seconde). Comme le terrain, la texture est répétée et prend la moitié de la largeur et couvre seulement le cinquième de la surface sur la longueur.

Pour simuler l'écoulement de la rivière, on ajoutera un déplacement des coordonnées le long de la longueur. La vitesse devrait être d'un facteur $1/10$ du temps. On peut aussi rajouter un effet de vague en ajoutant un autre décalement dans le fragment shader. On utilisera l'équation 1 et 2 pour faire la fonction d'oscillation.

$$x = cos(time * timeScale.x + (texCoords.x + texCoords.y) * offsetScale.x) \quad (1)$$

$$y = sin(time * timeScale.y + (texCoords.x + texCoords.y) * offsetScale.y) \quad (2)$$

L'élément le plus important dans cette formule est le décalement du sinus/cosinus qui dépend de la coordonnée de texture sur le plan. On utilisera un timeScale de $x = 1$ et $y = 2$, un offsetScale de $x = y = 2$. Au final, on pourra multiplier le tout par l'amplitude de $x = 0.025$ et $y = 0.01$.

Partie 2e : Utilisation de cube map pour skybox

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 nuages pour donner l'impression d'être dans un très grand monde.

Il faudra complété la classe `TextureCubeMap`. Le mode de texture est plutôt `GL_TEXTURE_CUBE_MAP` au lieu de `GL_TEXTURE_2D`. Puisque les coordonnées de texture sont utilisées pour déterminer quelle texture sera utilisée, elles seront données en vec3. Cela implique qu'il faudra changer le paramètre de wrap pour la composante `r` du vecteur de coordonnée de texture.

Pour charger les 6 textures, la cible sera `GL_TEXTURE_CUBE_MAP_POSITIVE_X + i`, où i est l'index de la texture. Le tableau des chemins des textures vous ait donné, ainsi que les coordonnées de position.

Il faudra faire un nouveau shader programme (`skybox.*.glsl`). Pour le vertex shader, on a en entrée l'attribut vec3 de position, une sortie vec3 de coordonnée de texture et la matrice mvp en uniform. La sortie de coordonnée de texture est simplement la position. Pour le fragment shader, l'entrée est l'attribut de coordonnée de texture qui sera utilisé pour prendre la texture de type `samplerCube`.

Puisqu'on voudra seulement remplir le derrière de l'image (les endroits où rien n'a été dessiné), on dessinera le skybox à la toute fin (même après le carré dans l'écran) en prenant soin de changer la fonction du test de profondeur pour qu'il passe au endroit qui n'ont pas été dessiner (la valeur de ces endroits sera de 1 dans le depth buffer) (n'oublier pas de restaurer l'ancienne fonction après avoir dessiner le ciel). Il sera nécessaire de changer la valeur du Z dans la position finale du vertex shader pour permettre ce fonctionnement. De plus, on ne veut pas que le skybox bouge avec la caméra, il faudra donc retirer la translation de la matrice de vue (en la convertissant en matrice 3x3, puis 4x4 de nouveau par exemple).

Partie 2f : HUD

Finalement, il faut mettre une texture sur le quad du HUD. On lui mettra la texture du coeur pour imiter le HUD d'une barre de vie. Puisque la texture aura des texels transparents, il faudra ajouter un `discard` du fragment dans le fragment shader pour éliminer les fragments dont le alpha est en dessous de 0.3.

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

E1. La classe Camera est implémenté correctement. Il est possible de voir en première et troisième personne et de ce déplacer. [3 pts]

E2. La classe Model est implémentée correctement. Il est possible de dessiner les modèles 3D. [1 pt]

E3. Les modèles géométriques (ruisseau et sol) sont bien définis. [1 pt]

E4. Les transformations sont faites correctement sans trigonométrie et ont une bonne réutilisation des calculs matricielles. [5 pts]

E5. Le modèle du personnage est seulement dessiné en troisième personne. [1 pt]

E6. La projection orthogonal et le quad sont bien dessiné dans l'écran. Il ne traverse pas les objets 3D. [1 pt]

E7. Le face culling est activé. [1 pt]

Partie 2 :

E8. La classe Texture2D est implémenté correctement. [1 pt]

E9. La classe TextureCubeMap est implémenté correctement. [1 pt]

E10. Les textures sont chargées dans le bon mode. [1 pt]

E11. Les attributs de coordonnée de texture sont défini correctement. Ils sont correctement utilisés dans le shader pour voir les textures sur les objets. [2 pts]

E12. Les coordonnées de texture du terrain et de la rivière sont bien défini. Il active le mipmap et fait la bonne interpolation de minification. [2 pts]

E13. Le ruisseau est animé à l'aide du shader. On voit le courant et les vagues. [2 pts]

E14. Le skybox est visible et ne bouge qu'avec la rotation de la caméra. Il est dessiné en dernier avec le bon depth test. [4 pts]

E15. Il y a minimisation des changements d'états d'OpenGL. [1 pt]

E16. On discard les fragments avec un alpha trop petit. [1 pt]

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
`w`	Mouvement avant
`s`	Mouvement arrière
`a`	Mouvement gauche
`d`	Mouvement droit
`scroll`	Alterner entre fps/tps
`souris`	Changer l'orientation de la camera