INF2705 Infographie

Travail pratique 1
Introduction à OpenGL

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

Table des matières

1 Introduction

Ce document décrit les exigences du TP1 « *Introduction à OpenGL* » (Automne 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 notions de base d'OpenGL. Il sera en mesure d'initialiser un contexte OpenGL et un pipeline graphique basique pour dessiner des primitives simples de différentes façons. Les notions de bases comprennent entre autres : l'utilisation de vao, vbo, ebo, `glVertexAttribPointer`, `glBufferData`, `glBufferSubData`, `glMapBuffer`, `glDrawArrays`, `glDrawElements`.

Ce travail pratique lui permettra aussi de mettre en pratique les notions de transformations matricielles pour convertir les données dans les différents systèmes de coordonnées. Il sera aussi en mesure d'afficher des modèles 3D à l'écran et de les animés.

2.2 Travail demandé

Partie 1 : dessin de primitives simples

Comme tout début en infographie, il faut commencer par dessiner son tout premier triangle. Pour ce faire, un projet de base vous a été fourni. Celui-ci contient du code pour la gestion de la fenêtre, la gestion d'erreur et certaines fonctions qui ne nécessite pas particulièrement de notion en infographie afin que vous vous focussez sur la matière du cours. Il n'est pas requis d'analyser le code déjà écrit, mais il est tout de même intéressant de le comprendre.

Un système de gestion de scène est présent pour mieux séparé votre code entre les différents exercices.

Un squelette de classes avec des todos est disponible pour vous guider. Vous allez avoir besoin d'implémenter les classes d'objets OpenGL afin de pouvoir voir quoique ce soit à l'écran. Il est recommander d'y aller dans l'ordre suivant : `ShaderObjet`, `ShaderProgram`, `BufferObject`, `VertexArrayObject`, `DrawArraysCommand`. Il faudra remplir les tableaux de vertices dans `vertices_data.h` pour votre triangle. Pour votre premier triangle, utiliser la scène SceneTriangle.

Tout au long du développement, vous allez pouvoir utiliser la macro `CHECK_GL_ERROR` (dans le fichier utils.h) après un appel à une fonction OpenGL pour afficher les erreurs d'OpenGL et vous aidez à débugger.

Pour la partie 1, il y a 2 shaders simples à implémenter :

- Le shader de base `basic.vs.glsl` et `basic.fs.glsl`. L'entrée du vertex shader est un vec3 à la location 0 pour la position. L'implémentation du main du programme est simplement d'assigner la valeur de `gl_Position` à la valeur de l'entrée. Le fragment shader fait simplement l'assignation d'une variable out de type vec4 pour la couleur de votre choix avec un alpha de 1 (attention d'avoir une couleur différente du clearColor).
- Le shader de couleur `color.vs.glsl` et `color.fs.glsl`. On ajoute au vertex shader de base

l'attribut de couleur de type vec3 à la location 1. Une variable out vec3 de la couleur sera envoyée au fragment shader. Celui-ci possède en entrée la couleur transmise par le vertex shader et l'assigne à la couleur de sortie avec un alpha de 1.

L'instanciation de ces `ShaderProgram` est faite dans la classe `Resources` pour permettre la réutilisation des programmes au travers les scènes.

Il y a 7 formes à dessiner avec `GL_TRIANGLES` dont certaines seront identiques, mais la façon dont elles sont envoyées à la carte graphique sera différente :

- Votre premier triangle dans `SceneTriangle`. Il utilisera le shader de base et les vertices possèdent seulement l'attribut de position.
- Un carré dans `SceneSquare`. Il utilisera le shader de base et les vertices possèdent seulement l'attribut de position.
- Un triangle coloré dans `SceneColoredTriangle`. Il utilisera le shader de couleur et les vertices possèdent l'attribut de position et de couleur. On utilise un agencement des données entrelacées dans un seul vbo, ce qui permet dans beaucoup de cas une optimisation au niveau de la cache pour des données en lecture seules. Son `BufferObject` est dans la classe `Resource` pour être réutilisé.
- Un carré coloré dans `SceneColoredSquare`. Il utilisera le shader de couleur et les vertices possèdent l'attribut de position et de couleur. Remarquer le nombre de données dédoublées.
- Un autre triangle dans `SceneMultipleVbos` qui change de couleur et qui se déplace. Il utilisera le shader de couleur et les vertices possèdent l'attribut de position et de couleur. L'avantage d'avoir plusieurs vbo est pour la mise à jour des attributs lorsqu'il n'y en a qu'une partie à modifier. Pour le changement de couleur, on utilisera `glBufferSubData`, qui nécessite une copie des données au niveau de la mémoire principale, alors que le changement de position utilisera `glMapBuffer` et `glUnMapBuffer`, qui modifie les données directement sur la mémoire du processeur graphique. Pour la modification des données, il ne faut pas utilier `glBufferData` si la taille ne change pas, car cela génère une réallocation des données. Utiliser `changePos` et `changeRGB` pour changer les attributs.
- Un carré coloré dans `SceneDrawElements`. Il utilisera le shader de couleur et les vertices possèdent l'attribut de position et de couleur. On utilisera des indexes dans un ebo pour réutiliser les vertices du carré pour réduire la consommation de mémoire. On aura un `glDrawElements`. Ses `BufferObject` sont dans la classe `Resources` pour être réutilisés.
- Un triangle et carré coloré dans `SceneSharedVao`. On utilisera le même VAO pour dessiner les deux formes. Il faudra donc avant chaque dessin re-spécifier le `BufferObject` à être utilisé. Utiliser les `BufferObject` partagés dans la classe `Resources`.

Les triangles ont les points (-0.5 ; -0.5) (0.5 ; -0.5) (0.0 ; 0.5). Les carrés ont les points (-0.5 ; -0.5) (0.5 ; -0.5) (-0.5 ; 0.5) (0.5 ; 0.5). Pensez à l'ordre des points et au nombre de composantes qu'ils devraient avoir.

Pour l'initialisation des shaders programs dans la classe `Resources`, utiliser la fonction `readFile` pour lire les fichiers `.glsl` sur le disque.

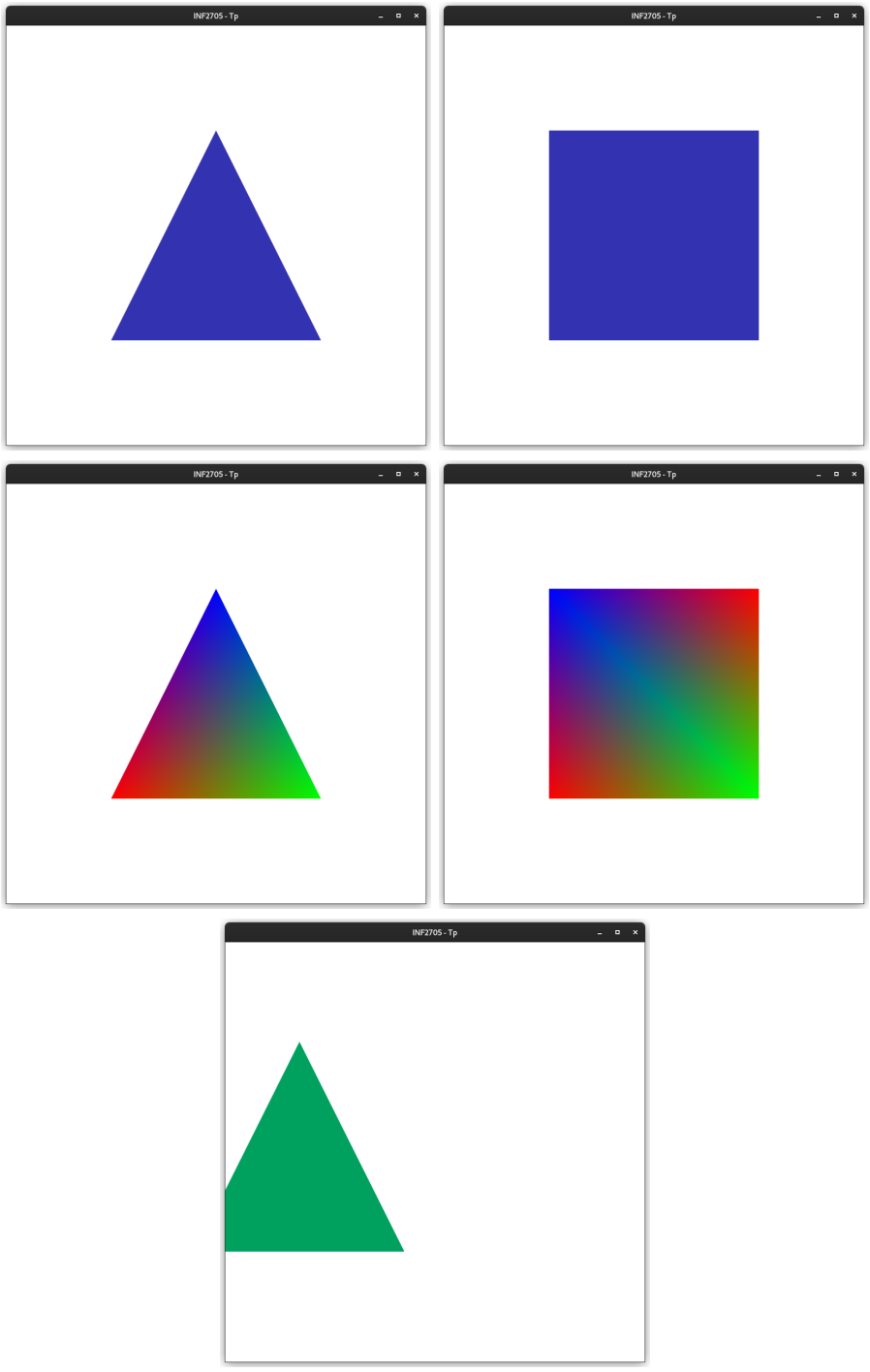

FIGURE 1 – Les 7 formes à implémenter dans l'ordre : triangle, carré, triangle coloré, carré coloré et triangle mis à jour

Partie 2 : transformations et système de coordonnées

Maintenant qu'on est à l'aise avec les bases du pipeline graphique, on peut commencer à faire de la 3D. Pour ce faire, il faut effectuer des transformations sur les formes qu'on dessine. Une variable de type uniforme dans le vertex shader va nous permettre d'envoyer facilement une nouvelle matrice pour chaque objet qu'on veut dessiner à un endroit différent.

Le shader de transformation est dans `transform.vs.glsl` et `transform.fs.glsl`. Il peut être fait à partir du shader `color.vs.glsl` et `color.fs.glsl`. On y ajoute une variable uniforme de type mat4 pour une matrice résultante modèle-vue-perspective (matrice mvp). On pourra multiplier la position par cette matrice. Le fragment shader est identique encore à celui du shader de couleur. Après l'initialisation du shader, il va falloir utiliser la méthode `getUniformLoc` pour pouvoir connaître la location du uniform de la matrice.

Utiliser le `ShaderProgram` `transformColorAttrib` dans la classe `Resources`.

Notre forme 3d est un cube coloré dans `SceneCube`. Il utilisera le shader de transformation et les vertices possèdent l'attribut de position et de couleur. Les vertices et indexes sont déjà définis dans `vertices_data.h`. Portez attention au dédoublement des vertices, qui est nécessaire pour l'attribut de couleur qui ne peut pas être partagé.

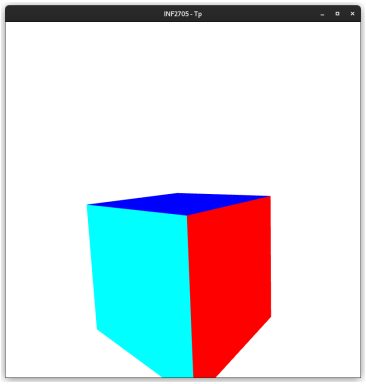

FIGURE 2 – Cube 3D

Si on essait de dessiner le cube sans envoyer de valeur à la matrice dans le shader, rien ne sera visible ; il faut construire la matrice mvp.

Pour la matrice de modèle, on demande à ce que le cube fasse une rotation dans l'axe (0.1 ; 1.0 ; 0.1) de l'angle `m_rotationAngleDegree` en degrée.

Pour la matrice de vue, il faut mettre la camera à la position (0 ; 0.5 ; 2). La camera sera déjà orienté vers le cube, donc aucune rotation ou matrice lookAt n'est nécessaire ici.

Pour la matrice de projection, il faut un fov de 70.0, la position du near plane à 0.1 et du far plane à 10.0. Le aspect ratio est calculable à partir des méthodes `getWidth` et `getHeight` de la classe `Window` (attention au division entière).

On peut envoyer la matrice mvp d'un coup avec `glUniformMatrix4fv`. Il est courant de voir la décom-

position de la matrice mvp dans les shaders (en 3 matrices séparées), mais cela peut devenir un problème de performance si trop de vertices sont traitées dans le vertex shader, puisque le programme effectura le calcul de la matrice résultante pour chaque vertice. Les calculs de matrices seront fait avec la librairie glm avec les fonctions `glm::rotate`, `glm::translate` et `glm::perspective`.

N'oubliez pas d'activer le depth testing (test de profondeur) et de clear le depth buffer (tampon de profondeur).

Partie 3 : Chargement et animation de modèles 3d

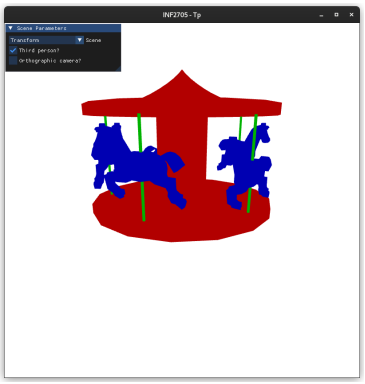

FIGURE 3 – Animation du carrousel

En temps normal, il est commun de faire les modèles 3d dans une autre application spécialisée pour la modélisation plutôt que remplir des tableaux de points directement dans le code. Dans votre cas, les modèles ont déjà été fait et une librairie de chargement de fichier `.obj` est disponible. Il faudra tout de même implémenter le chargement des données du modèle dans la carte graphique. Pour ce faire, vous devez remplir la classe `Model` en initialisant le VBO, EBO, drawCommand et spécifier l'attribut de position dans le VAO.

Pour les shaders `model.*.glsl`, on aura une version modifiée du shader de transformation qui ne possède que l'attribut de position en entrée dans le vertex shader. Dans le shader de fragment, on remplacera l'attribut en entrée par une variable uniform. Utiliser le `ShaderProgram transformSolidColor` dans la classe `Resources`.

Dans la scène `SceneTransform`, il faudra construire la matrice mvp pour les différents modèles :

- La matrice de projection orthographique projete l'origine au centre de l'écran et possède un frustum carré de taille `SCREEN_SIZE_ORTHO`. Le near plane est à 0.1 et le far plane à 100.
- La matrice de projection de perspective est identique à celle du cube, mais le far plane est à 100.
- La matrice de vue en première personne est obtenable à partir de la méthode `getCameraFirstPerson`. Il faudra faire des rotations selon l'orientation de la caméra et une translation selon la position de la caméra (`glm::lookAt` ne sera pas accepté).
- La matrice de vue en troisième personne est obtenable à partir de la méthode `getCameraThirdPerson`. 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 au niveau du sol, un décalement de $\pi/2$ sera appliqué sur l'orientation en x et y. La caméra fixe l'origine de la scène.
- La matrice modèle de la base du carrousel est une translation de -0.1 sur l'axe y.
- La matrice modèle du groupe de pôle et cheval est à une hauteur de 1 en y, un rayon de 1.7 en x et à un angle multiple de $2\pi/N\_HORSES$. **Vous ne devriez pas avoir de trigonométrie dans le code des transformations**.
- Une matrice d'animation de rotation pour le groupe de pôle et cheval pour faire tourner le carrousel. L'angle est `m_carouselAngleRad`.

— Une matrice d'animation de translation pour le cheval pour l'animé de haut en bas. Un cheval sur deux à une translation positive de `carouselHorseTranslation`, l'autre est négative.

Pensez à la réutilisation des matrices et essayer de minimiser le nombre de multiplication en mémorisant les matrices dans des variables.

Pour les couleurs, on a une intensité de 0.7. La base du carrousel est rouge, les pôles, verts et les chevaux, bleus.

L'attribut `m_isOrtho` permet d'alterner entre une matrice de projection orthographique et perspective.

L'attribut `m_isThirdPerson` permet d'alterner entre la matrice de vue première personne et troisième personne.

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

E1. La classe `ShaderObject` est implémentée correctement. [1 pt]

E2. La classe `ShaderProgram` est implémentée correctement. [1 pt]

E3. La classe `BufferObject` est implémentée correctement. [1 pt]

E4. La classe `VertexArrayObject` est implémentée correctement. [1 pt]

E5. Les classes `DrawArraysCommand` et `DrawElementsCommand` sont implémentées correctement. [2 pts]

E6. Les vertices et indexes sont définis comme spécifié dans l'énoncé et respectent le nombre de composants de l'entrée du vertex shader. Les attributs multiples sont entrelacés. [2 pts]

E7. Les shader programs de base et de couleur sont initialisées correctement dans `Resources`. [1 pt]

E8. Les scènes produisent le résultat attendu. [3 pts]

E9. Les données de couleur et de position sont mise à jour avec les bonnes fonctions. [1 pt]

E10. Les scènes sont dessinées avec le bon shader tel qu'énoncé. [1 pt]

E11. Les shaders utilisent les attributs correctement pour donner la couleur au fragment. [1 pt]

E12. La couleur de fond et le nettoyage sont fait correctement dans le main. [1 pt]

Partie 2 :

E13. Le `ShaderProgram` de transformation est initialisé correctement. [1 pt]

E14. Le vertex shader de transformation applique une seule matrice mvp en uniform. [2 pts]

E15. On utilise le depth testing . [1 pt]

E16. Les matrices de modèle, vue et projection ont les bons paramètres. La matrice résultante est calculée sur le cpu et est envoyé sur le gpu à partir du uniform. [3 pts]

Partie 3 :

E17. La classe `Model` est implémentée correctement. Il est possible de dessiner les modèles 3D. [1 pt]

E18. Le uniform de couleur est bien défini et les modèles sont de la bonne couleur [1 pt]

E19. La matrice première personne est implémentée correctement (sans lookAt). [2 pts]

E20. La matrice troisième personne est implémentée correctement (coordonnées sphériques). [2 pts]

E21. La projection orthogographique est défini correctement. [1 pt]

E22. Les transformations sont faites correctement sans trigonométrie. [5 pts]

E23. Les transformations ont une bonne réutilisation des calculs matricielles. [2 pts]

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. [3 pts]

ANNEXES

A Liste des commandes

Touche	**Description**
`ESC`	Quitter l'application
`t`	Change la scène affichée
`SPACE`	Afficher/Cacher la souris