POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

INF2705 Infographie

Travail pratique 1

Introduction à OpenGL

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

Table des matières

1 Introduction

Ce document décrit les exigences du TP1 « *Introduction à OpenGL* » (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 notions de base d'OpenGL. Il sera en mesure d'initialiser une contexte OpenGL et un pipeline graphique basique pour dessiner des primitives simples de différentes façons. Les notions de bases comprennent entre autre : 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.

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 focusez 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 squelette de classes avec des todos est disponible pour vous guidez. Vous allez avoir besoin d'implémenter les classes `BasicShapeArrays`, `BasicShapeMultipleArrays`, `BasicShapeElements` dans `shapes.h`, les classes `Shader`, `ShaderProgram` dans `shader_program.h`, de définir les positions, les couleurs et les indexes des formes dans `vertices_data.h`, d'écrire les shaders dans le répétoires shaders, et de faire l'intégration dans `main.cpp`.

Tous au long du développement, vous allez pouvoir utiliser la macro `GL_CHECK_ERROR` après un appel à une fonction OpenGL pour afficher les erreurs d'OpenGL et vous aidez à débugger.

Avant d'avoir un résultat, vous allez avoir besoin d'implémenter le code de plusieurs parties du tp. Entre autre, il faut les classes `Shader` et `ShaderProgram` pour exécuter les shaders sur la carte graphique, la classe `BasicShapeArrays` pour gérer les données des formes sur la carte graphique et permettre de les dessiner et bien entendu les données des formes dans `vertices_data.h`.

Pour la partie 1, il y a 2 shaders simples à implémenter (n'oublier pas de spécifier `#version 330 core` en première ligne de chaque fichier .glsl) :

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

Pour la partie 1, il y a 6 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, de type `BasicShapeArrays`. Il utilisera le shader de base et les vertices ont seulement l'attribut de position.
— Un carré, de type `BasicShapeArrays`. Il utilisera le shader de base et les vertices ont seulement l'attribut de position.
— Un triangle coloré, de type `BasicShapeArrays`. Il utilisera le shader de couleur et les vertices ont 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.
— Un carré coloré, de type `BasicShapeArrays`. Il utilisera le shader de couleur et les vertices ont l'attribut de position et de couleur. Remarquer le nombre de données dédoublées.
— Un triangle de type `BasicShapeMultipleArrays` qui change de couleur et qui se déplace. Il utilisera le shader de couleur et les vertices ont l'attribut de position et de couleur. L'avantage d'avoir plusieurs vbo est pour la mise à jour des attributs lorsqu'il 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 `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 de type `BasicShapeElements`. Il utilisera le shader de couleur et les vertices ont 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.

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.

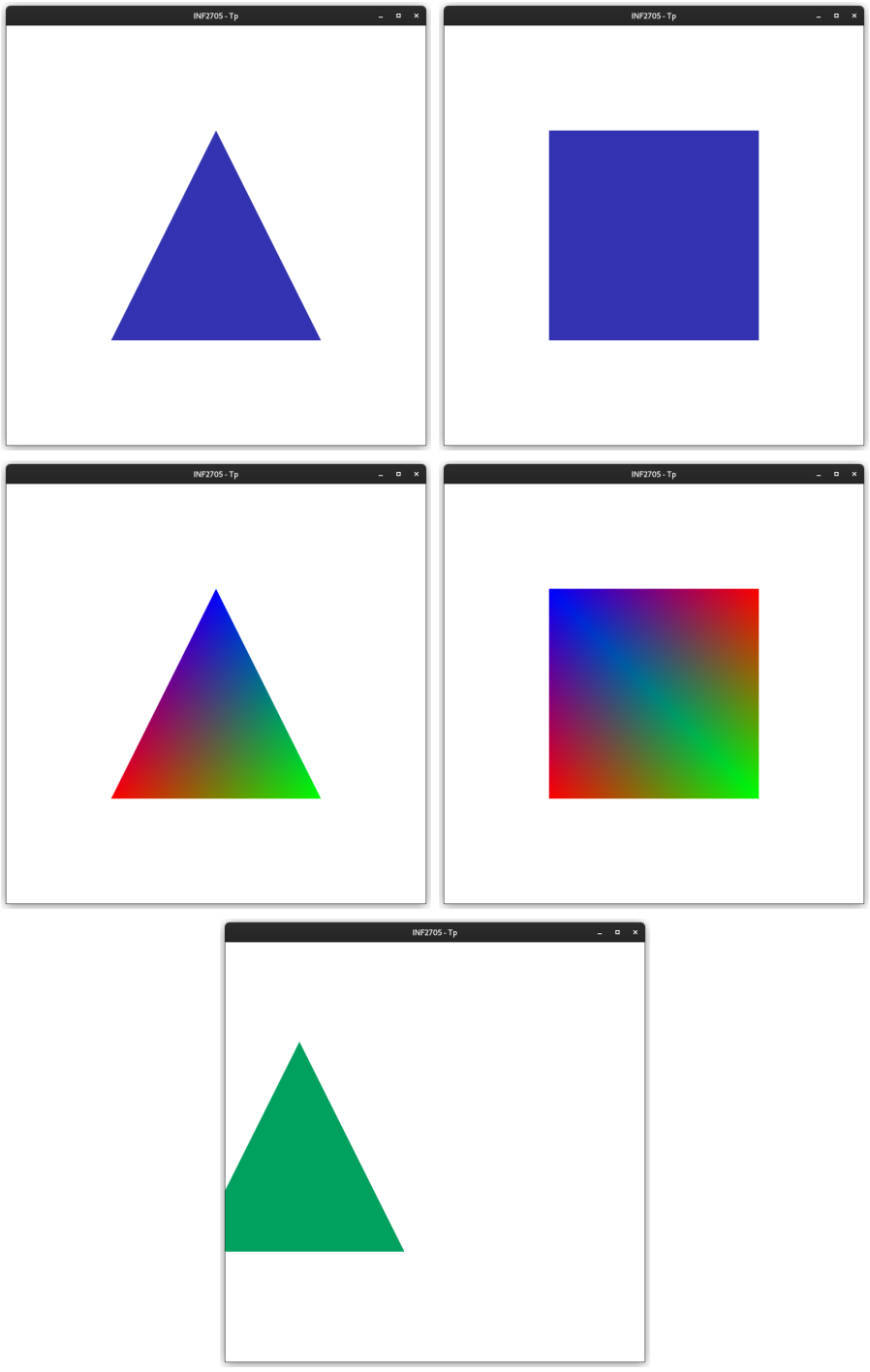

FIGURE 1 – Les 7 formes à implémenter dans l'ordre : triangle, carré, triangle coloré, carré coloré et triangle mis à jour

Partie 2 : transformations et système de coordonnées

Maintenant qu'on est à l'aise avec les bases du pipeline, 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 uniform 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`. On ajoute au vertex shader de couleur une variable uniform 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. Dans le main, il va falloir utiliser la méthode `getUniformLoc` pour pouvoir connaître la location du uniform de la matrice.

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

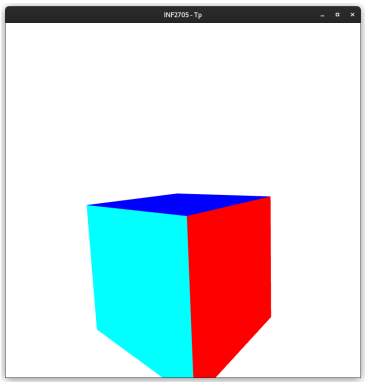

FIGURE 2 – Cube 3D

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 `angleDeg` en degrée. Pour la matrice de vue, il faut mettre la camera à la position (0 ; 0.5 ; 2) qui pointe vers l'origine de la scène. 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 10.0f. Le aspect ratio est calculable à partir des méthodes `getWidth` et `getHeight` de la classe `Window`.

On peut envoyer la matrice mvp d'un coup avec `glUniformMatrix4fv`. Il est courant de voir la décomposition de la matrice mvp dans les shaders (en 3 matrices séparées), mais cela peut devenir un problème de performance si trop de vertices sont traîtées dans le vertex shader, puisque le programme effectura le calcul de la matrice résultat pour chaque vertices. Les calculs de matrices seront fait avec la librairie glm avec les fonctions `glm::rotate`, `glm::translate` et `glm::perspective`.

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

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

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

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

E3. La classe `BasicShapeArrays` est implémentée correctement avec le bon mode d'usage. [1 pt]

E4. La classe `BasicShapeMultipleArrays` est implémentée correctement avec le bon mode d'usage et les attributs sont séparés dans des vbo différents. [1 pt]

E5. La classe `BasicShapeElements` est implémentée correctement avec le bon mode d'usage. On dessine correctement avec le ebo. [2 pts]

E6. Les vertices et indexes sont définis comme spécifié dans l'énoncé et respecte l'entré du vertex shader. Les attributs multiples sont entrelacés. [2 pts]

E7. Les shader programs de base et de couleur sont instanciés correctement dans le main. [2 pts]

E8. Les formes sont instanciées correctement avec les bons paramètres d'attributs. [2 pts]

E9. La couleur de fond et le nettoyage sont fait correctement. [1 pt]

E10. Les données de couleur et de position sont mise à jour avec les bonnes fonctions. [2 pts]

E11. Les formes sont dessinées adéquatement avec le bon shader tel qu'énoncé. [1 pt]

Partie 2 :

E12. Le shader program de transformation est instancié correctement dans le main. [1 pt]

E13. La location de la matrice est obtenu correctement. [1 pt]

E14. Le cube est instancié correctement. [1 pt]

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]

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