



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 2025

## Table des matières

<b>1 Description globale</b>	<b>2</b>
1.1 But . . . . .	2
1.2 Travail demandé . . . . .	2
<b>2 Exigences</b>	<b>7</b>
2.1 Exigences fonctionnelles . . . . .	7
2.2 Exigences non fonctionnelles . . . . .	8
2.3 Remise . . . . .	8

# 1 Description globale

## 1.1 But

Le but du 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 base comprennent entre autres : l'utilisation de vao, vbo, ebo, glVertexAttribPointer, glBufferData, glBufferSubData, 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 capable de créer une scène graphique et d'animer des modèles 3D.

## 1.2 Travail demandé

### Partie 0 : premier triangle

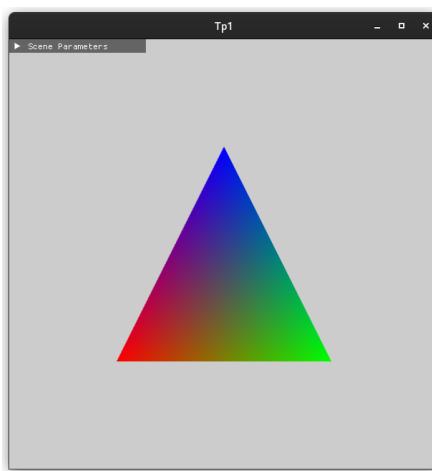


FIGURE 1 – Un triangle, rien de plus simple

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 nécessitent pas particulièrement de notions en infographie afin que vous vous focalisiez 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.

Une gestion de scène est présente pour mieux séparer votre code entre les différents exercices.

Le résultat final de cette partie n'est pas directement évalué, mais permettra de s'assurer de bien saisir les concepts de base reliés au dessin et sera plus facile à débugger.

Pour ce faire, vous devez remplir dans l'ordre les sections suivantes dans `main.cpp` :

- `init();`
- `drawFrame();`
- `loadShaderObject();`
- `loadShaderPrograms();`
- Le premier todo avant la classe `App`;
- Déclaration du type de `vertices_` et `elements_`;
- `initShapeData();`
- Le dessin du triangle dans `sceneShape()`. Vous pouvez faire un `glDrawArrays` pour commencer, puis le modifier par `glDrawElements`;
- `onClose();`

Vous allez aussi avoir besoin de remplir les fichiers `basic.vs.gls1` et `basic.fs.gls1` qui contiennent le code source des shaders.

Il est à noter que rien ne sera visible tant que toutes ces étapes ne seront effectuées.

Il est recommandé de mettre les sommets du triangle aux positions (-0.5 ; -0.5) (0.5 ; -0.5) (0.0 ; 0.5). Pensez à l'ordre des points et au nombre de composantes qu'ils devraient avoir.

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

Toujours face à un écran noir ? Essayez de vous assurer que les éléments suivants soient effectués :

- Les shaders compilent correctement (vérifiez les erreurs de compilation et de linking) ;
- Le programme de shader est en cours d'utilisation ;
- Les variables de sorties du vertex shader ont les mêmes noms que les variables en entrées du fragment shader ;
- Le nettoyage du tampon de couleur est effectué (l'ancienne image ne reste pas après avoir déplacer l'interface graphique) ;
- Les buffers (vbo et ebo) sont créés et remplis correctement ;
- Les bons attributs sont activés dans le vao et correspondent aux mêmes locations dans le vertex shader ;
- La fonction `glVertexAttribPointer` spécifie correctement le format des données : l'index correspond à la location dans le vertex shader, le nombre de composantes et le type de donnée réfletent le contenu du vbo, le saut (stride) représente correctement le nombre d'octets pour aller au prochain sommet, le décalage (offset, ou pointer dans la déclaration de `glVertexAttribPointer`) représente correctement le nombre d'octets à sauter avant de commencer à lire les données de l'attribut ;
- Le ebo (s'il y en a un) est attaché au vao ;
- Le vao est délié du contexte à la fin de la spécification du format pour éviter une modification de celui-ci ;
- Le bon vao est spécifié avant le dessin ;
- Le dessin avec `glDrawArrays` utilise le nombre de points à dessiner ;
- Le dessin avec `glDrawElements` utilise le nombre d'index à dessiner et le type correspond à celui utilisé dans le ebo.

## Partie 1 : dessin et mise à jours de primitives simples

Maintenant que vous êtes à l'aise de dessiner un triangle, on peut s'attaquer à des formes géométriques plus complexes : des polygones.

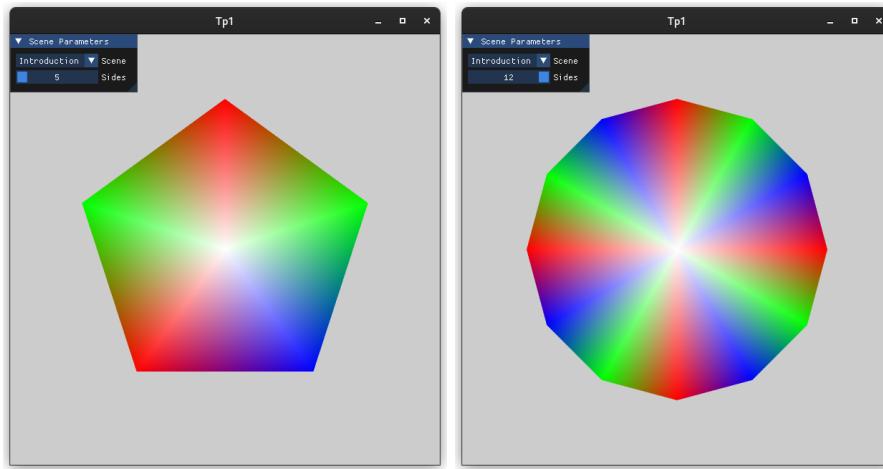


FIGURE 2 – Polygone à 5 et 12 côtés, un pentagone et un dodécagone

Si la partie 0 du TP a été réalisée sans problème, vous avez seulement besoin de remplir les sections suivantes dans `main.cpp` :

- Retirer le triangle de `initShapeData()` ;
- Remplir `generateNgon()` ;
- Adapter le dessin dans `sceneShape()` et effectuer la mise à jour des données.

## Partie 2 : transformations et animation de modèles 3D

Maintenant que vous êtes à l'aise avec les bases du pipeline graphique, on peut commencer à faire de la 3D. Pour être capable de visualiser les modèles 3D, 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.

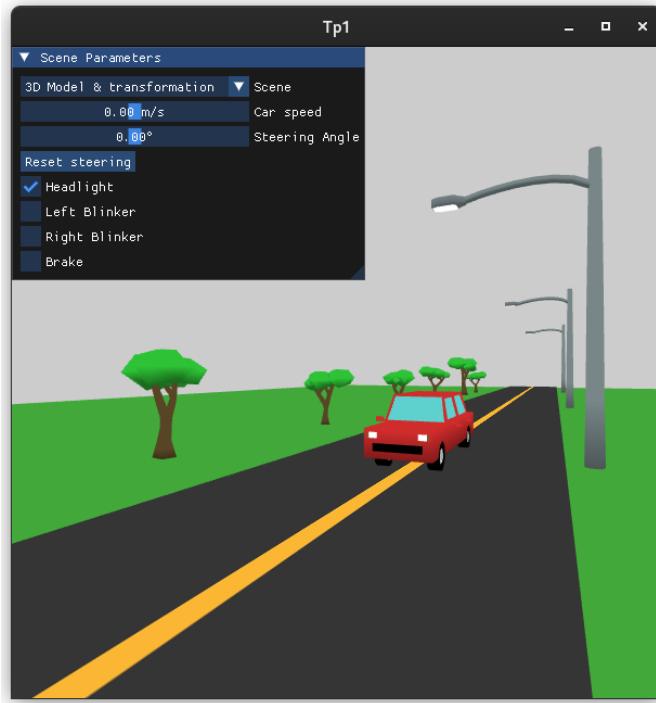


FIGURE 3 – La scène 3D : une route avec une automobile

Pour cette partie, on demande de positionner les différents modèles dans la scène pour représenter une route avec des arbres et lampadaires de chaque côté, puis une automobile. L'automobile est animée par l'interface graphique.

Cependant, pour permettre l'animation de celle-ci, le modèle 3D a été fragmenté pour permettre un meilleur contrôle sur les transformations. Il faudra alors la reconstituer. L'automobile est construite comme suit :

- 1 châssis ;
- 4 roues, dont celles du devant qui peuvent être orientées ;
- 4 phares ;
- Un phare est composé d'une lumière et d'un clignotant.

De plus, l'origine du modèle des roues n'est pas centrée. Il sera important de considérer cette information pour bien positionner les roues avant. Voir la figure suivante 4 :

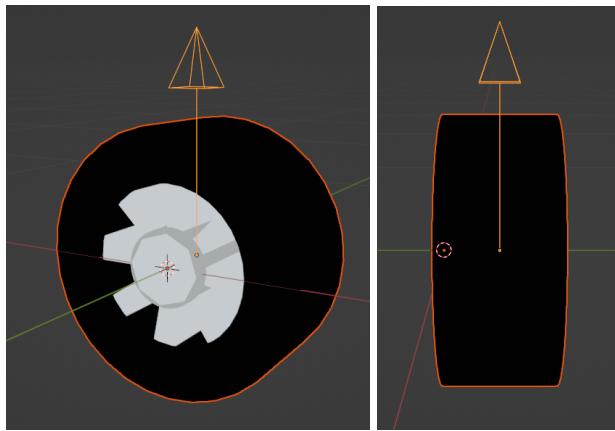


FIGURE 4 – Modèle de la roue : Le point orange est l'origine de la roue

Pour réaliser le travail, vous devez remplir dans l'ordre les sections suivantes dans `main.cpp`, `model.cpp`, `car.hpp` et `car.cpp` :

- `App::init();`
- `App::drawFrame();`
- `App::getViewMatrix();`
- `App::getPerspectiveProjectionMatrix();`
- `Model::load();`
- `Model::draw();`
- `App::drawGround();`
- `App::drawTrees();`
- `App::drawStreetlights();`
- La classe `Car`;
- `App::sceneModels();`

Le shader de transformation est dans `transform.vs.gls1` et `transform.fs.gls1`. Il peut être fait à partir du shader `basic.vs.gls1` et `basic.fs.gls1`.

Il est préférable d'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 traitées dans le vertex shader, puisque le programme effectuera le calcul de la matrice résultante pour chaque vertex. 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) dans `main.cpp`.

Pour l'automobile, il est recommandé de positionner tous les modèles correctement relatifs à l'origine avant de les animer avec les attributs de la classe.

## 2 Exigences

### 2.1 Exigences fonctionnelles

Partie 1 :

- E1. La couleur de fond est modifiée. [1 pt]
- E2. Les programmes de shader sont compilables avec l'application. [2 pts]
- E3. Les fichiers `basic.*.glsl` sont complétés pour dessiner le polygone avec différentes couleurs sur les sommets. [2 pts]
- E4. Le tampon de couleur est effacé à chaque frame. [1 pt]
- E5. La méthode `generateNgon` permet de générer les données appropriées pour dessiner un polygone à N côtés. [3 pts]
- E6. Le nombre de sommets est minimisé avec un ebo. [2 pts]
- E7. La méthode `initShapeData` permet de générer et d'allouer les vbo et ebo d'une taille suffisamment grande pour le problème. [1 pt]
- E8. Les vbo et ebo ont un bon mode d'usage dans `glBufferData`. [1 pt]
- E9. Le format de données est bien décrit dans le vao. [3 pts]
- E10. Le polygone est dessiné correctement. [1 pt]
- E11. Il est possible de modifier le nombre de côtés du polygone. [2 pts]
- E12. Les ressources graphiques sont libérées à la fermeture de la fenêtre. [1 pt]

Partie 2 :

- E13. On utilise le depth testing et le face culling. [1 pt]
- E14. Les matrices statiques sont initialisées qu'une seule fois au début du programme. [2 pts]
- E15. Le tampon de profondeur est effacé à chaque frame. [1 pt]
- E16. Le programme de shader de transformation utilise une seule matrice en variable uniform pour positionner les modèles à l'écran. [1 pt]
- E17. La variable uniform de couleur est bien utilisée pour teinter les lumières de l'automobile et elle n'est pas assignée inutilement. [1 pt]
- E18. La matrice de vue est définie en effectuant les opérations inverses pour placer l'observateur dans la scène. [3 pts]
- E19. La matrice de perspective est définie avec les bons paramètres. [1 pt]
- E20. La classe `Model` permet de charger les fichiers `.ply` dans la mémoire graphique. [5 pts]
- E21. Le destructeur de la classe `Model` libère les ressources graphiques. [1 pt]
- E22. Il est possible de dessiner un modèle 3D avec la classe `Model`. [1 pt]
- E23. Le sol est dessiné correctement. La route est par-dessus le gazon. [1 pt]
- E24. Les arbres sont positionnés correctement. Ils sont de taille et d'orientation aléatoire. [3 pts]
- E25. Les lampadaires sont positionnés correctement et sont orientés vers la route. [2 pts]

- E26. L'automobile est constituée d'un châssis, quatre roues et quatre phares. [1 pt]
- E27. Le châssis est bien placé. [1 pt]
- E28. Les roues sont bien placées, la jante vers l'extérieur. [1 pt]
- E29. Les phares sont bien placés, le clignotant pointant vers l'extérieur. [1 pt]
- E30. Les phares sont constitués d'une lumière et d'un clignotant. [1 pt]
- E31. La lumière et le clignotant sont bien positionnés. [1 pt]
- E32. L'automobile est animée correctement. [7 pts]
- E33. Il y a réduction des calculs matriciels en réutilisant autant que possible les matrices. [3 pts]
- E34. Tous les modèles sont dessinés dans la scène et sont au moins visibles. [1 pt]

## 2.2 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 (lorsque nécessaire), les mauvaises indentations, etc. Retirer les commentaires de TODOs dans la remise. [5 pts]

## 2.3 Remise

Créer une archive nommée « **INF2705\_remise\_TPn.zip** » 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 que vous avez modifié (`main.cpp`, `models.cpp`, `car.cpp`, `car.hpp`, `shaders/*.glsl`).