

# INF2705 Infographie

## Travail pratique 2

### *Détails et de réalisme des scènes graphiques*

## Table des matières

<b>1</b>	<b>Description globale</b>	<b>2</b>
1.1	But . . . . .	2
1.2	Travail demandé . . . . .	2
<b>2</b>	<b>Exigences</b>	<b>6</b>
2.1	Exigences fonctionnelles . . . . .	6
2.2	Exigences non fonctionnelles . . . . .	6
2.3	Remise . . . . .	7
<b>A</b>	<b>Formules utilisées</b>	<b>8</b>
A.1	Modèles de réflexion spéculaire de Phong et de Blinn . . . . .	8
A.2	Modèles de spot inspirés d'OpenGL . . . . .	9

# 1 Description globale

## 1.1 But

Le but de TP est de permettre à l'étudiant de mettre en pratique la matière reliée au texture, au test de stencil et à l'illumination. Il sera en mesure d'utiliser des textures afin d'ajouter des détails aux objets. Il sera aussi capable de configurer et d'utiliser le test de stencil. Pour finir, l'ajout de calculs d'illumination permettra d'ajouter plus de réalisme à l'ensemble de la scène. Le travail fera l'utilisation des fonctions d'OpenGL `glGenTextures`, `glBindTexture`, `glTexImage2D` et `glTexParameterf`, puis `glStencilOp`, `glStencilFunc`, `glStencilMask`.

## 1.2 Travail demandé

Le tp2 est une continuation du tp1. Il faudra reprendre à partir de votre ancien code ou utiliser la solution partielle offerte.

Un ensemble de modèles, textures et codes supplémentaires est disponible.

Il faut remplacer les modèles, qui contiennent maintenant les attributs de coordonnées de texture et de normales.

Pour le code, certains fichiers sont à ajouter à votre projet directement (textures, shaders, `shader_program`, `uniform_buffer`, `model_data`), alors que d'autres contiennent du code à ajouter dans certains fichiers actuelles (`car`, `main`, `model`).

Il faudra adapter légèrement le code pour être en mesure d'utiliser la classe nouvelle classe de `ShaderProgram`.

De plus, une implémentation de `UniformBuffer` est disponible pour faciliter l'envoi des informations de lumière au shader. Vous n'avez pas besoin de vous soucier de cette classe.

Vous pouvez retirer tout ce qui avait rapport au polygone de la partie 1 du tp1.

### Partie 1 : textures

Maintenant que la scène 3D est peuplée de modèles 3D, on remarque assez facilement les limitations de ce qu'on peut représenter avec seulement des couleurs sur chaque sommet. Pour corriger cet aspect, on peut ajouter des textures afin d'avoir un meilleur contrôle sur ce qui est dessiné sur la surface de nos triangles.

Pour se faire, il va falloir charger les textures de chaque modèle 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 fourni. Vous devez compléter son constructeur pour charger l'image

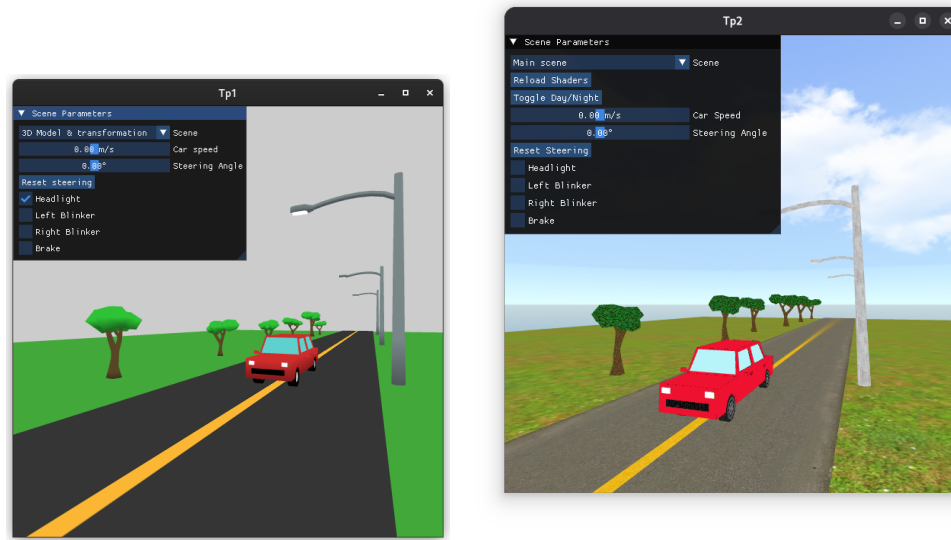


FIGURE 1 – Comparaison de avant et après l'ajout des textures

en tant que texture, ainsi que les autres méthodes pour utiliser la texture.

Utiliser les méthodes `setFiltering`, `setWrap` et `enableMipmap` à l'extérieur du constructeur (dans la méthode `init` dans le `main.cpp`).

On veut seulement que les textures se répètent sur les objets suivant : le sol, de la route, les arbres et les lampadaires.

On veut que les textures aient un fini lisse, à l'exception des arbres, des lumières de lampadaire et des fenêtres de la voiture, où on veut voir les pixels de façon plus définie.

Pour les modèles 3D, l'attribut de coordonnée de texture est déjà défini dans les modèles. Le chargement de celle-ci et la configuration de l'attribut a été fait pour vous dans la classe `Model`.

Pour ce faire, vous devez remplir dans l'ordre les sections suivantes :

- La classe `Texture2D`;
- Initialiser les textures dans `main.cpp`;
- Remplacer tous vos utilisation de shader par celui de `CelShading`;
- Modifier `CelShading` pour faire le dessin avec seulement des textures pour commencer;
- Dans `main.cpp`, ajouter les utilisations de texture avant chaque commande de dessin;

Ceci devrait être suffisant pour avoir les textures sur les objets.

Pour dessiner le ciel, le concept de cubemap est utilisé. Il est recommandé de consulter le site [LearnOpenGL](http://learnopengl.com) sur le sujet.

Pour les vitres de l'automobile, un effet de mélange de couleur sera fait. Utiliser le facteur classique : le facteur source sera l'alpha source ( $a\_src$ ) et le facteur destination sera un moins l'alpha source ( $1 - a\_src$ ). N'oublier pas d'utiliser la transparence dans la couleur finale du fragment (dans le shader).

## Partie 2 : effet de contour avec stencil

Maintenant que la scène contient plus de détail avec les textures, on voudrait la rendre plus stylée en lui donnant un aspect "cartoon". Ces dessins sont très distinct avec le tracé de contour noir, ainsi qu'un petit nombre de nuances de couleur pour les d'ombrages. Cela sera un bon moment de mettre en pratique les connaissances de test de stencil afin de réaliser l'effet de contour.



FIGURE 2 – Effet de contour sur les modèles 3D

Avant toute chose, il faut mentionné ici que l'effet n'est probablement pas le meilleur pour avoir un résultat de bonne qualité, mais celui-ci sera tout de même formateur.

Plusieurs méthodes sont faisables et autorisées dans notre cas, mais la façon la plus direct d'arriver au résultat souhaité est la suivante :

- Dessiner l'objet en couleur, mais aussi dans le tampon de stencil lorsque seulement le test de profondeur passe. On ne teste pas cette objet avec le stencil.
- On change le teste pour ne rien écrire dans le tampon de stencil.
- On change le teste pour réussir lorsque le tampon de stencil n'a pas eu d'écriture précédemment.
- On dessine une seconde fois le même objet, mais avec un autre shader, permettant de l'agrandir.
- On restaure l'état après avoir fini de dessiner tous les objets qui avaient besoin du test, et on désactive le test pour les objets qui n'en ont pas de besoin.

Il faudra faire attention à l'ordre de dessinage, spécifiquement pour les vitres.

### Partie 3 : illumination

Pour finaliser le tout, on voudrait ajouter plus de réalisme dans notre scène avec les calculs d'illumination.



FIGURE 3 – Illumination de la scène, de jour et de nuit

Une partie du code a déjà été réalisée pour vous économiser du temps. Celui-ci fait la gestion des uniform buffers, des matériels différents pour chaque objet, ainsi que des données de chaque lumière dans la scène.

Pour cette partie, vous allez faire les modifications principalement dans les shaders. On demande de faire le calcul d'illumination avec une source de lumière directionnel pour imiter un soleil, puis des sources de lumière de type spotlight avec facteur d'atténuation personnalisé. Vous devez implémenter le modèle d'illumination de Phong. Il est recommandé de se baser sur les exemples du cours.

Voici quelques directives supplémentaires pour vous aider :

- Un début de déclaration de shader a déjà été fait.
- On utilise un interface block pour permettre de mieux nommer les in et out des shaders. Seulement les noms de blocs ont besoin d'être identique entre les shaders, le nom de la variable peut changer.
- Les blocs d'uniforme sont définis pour contenir les données du matériel et d'illuminations. Vous ne devriez pas changer ces structures de données, au risque d'avoir des mauvais offsets de mémoire.
- Les calculs devront être fait dans le référentiel de la vue.
- Faites la normalisation des vecteurs après la rasterisation pour les attributs utilisés dans le fragment shader.
- N'oubliez pas d'appliquer la matrice de normale.
- Vous pouvez réaliser le calcul spéculaire de votre choix.
- Servez-vous de l'annexe pour quelques informations supplémentaires sur les calculs.

## 2 Exigences

### 2.1 Exigences fonctionnelles

Partie 1 :

- E1. Les textures appropriés sont affichées sur tous les modèles. [0.5 pt]
- E2. La classe Texture2D est implémenté correctement. [0.5 pt]
- E3. Les textures sont chargées dans le bon mode (mipmap, filtering, wrapping). [1 pt]
- E4. Les coordonnées de texture du sol et de la route sont bien défini. [1 pt]
- E5. Les vitres de la voiture sont transparentes. Elles teintes la vue et sont dessinées dans l'ordre. [1 pt]

Partie 2 :

- E6. L'effet de contour est réalisé avec le test de stencil. [3 pts]
- E7. Le buffer de stencil est modifié seulement lorsque nécessaire. [1 pt]
- E8. Aucun `glClear` supplémentaire n'est utilisé. [1 pt]
- E9. Les tests de stencil et blending sont activés seulement pour le dessin des fragments conservés. [1 pt]

Partie 3 :

- E10. Le modèle d'illumination de Phong est implémenté correctement. [4 pts]
- E11. Les spotlights sont implémentées correctement avec la version d'OpenGL. [1 pt]
- E12. Les calculs sont dans le référentiel de la vue. [0.5 pt]
- E13. Les normalisations sont effectués après la rasterisation plutôt qu'avant celle-ci. [0.5 pt]
- E14. Les propriétés du matériel sont utilisées correctement. [0.5 pt]
- E15. La texture diffuse est utilisée correctement. [0.5 pt]
- E16. La position des lumières suit correctement le modèle correspondant. [0.5 pt]
- E17. Les uniforms sont correctement envoyés au shader pour avoir des propriétés différentes par objet. [0.5 pt]

Bonus :

- E18. Il y a minimisation des changements d'états d'OpenGL (change le moins souvent possible les textures, les shaders, les modèles, etc.). [2 pts]

### 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. [2 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`, `model_data.hpp`, `shaders.cpp`, `shaders.hpp`, `car.cpp`, `car.hpp`, `textures.cpp`, `shaders/*.glsl`).

## ANNEXES

### A Formules utilisées

#### A.1 Modèles de réflexion spéculaire de Phong et de Blinn

Le calcul de la réflexion spéculaire fait intervenir un produit scalaire entre deux vecteurs. La différence entre les modèles de Phong et de Blinn réside dans le choix des deux vecteurs utilisés :

- Phong utilise :  $\vec{R} \cdot \vec{O} = \text{reflect}(-\vec{L}, \vec{N}) \cdot \vec{O}$

- Blinn utilise :  $\vec{B} \cdot \vec{N} = \text{bissectrice}(\vec{L} \text{ et } \vec{O}) \cdot \vec{N} = \text{normalise}(\vec{L} + \vec{O}) \cdot \vec{N}$

où :

$\vec{N}$  : normale à la surface

$\vec{L}$  : direction du point vers la source lumineuse

$\vec{R}$  : direction du rayon réfléchi  $= \text{reflect}(-\vec{L}, \vec{N})$ , avec  $\vec{L}$  et  $\vec{N}$  unitaires.

$\vec{O}$  : direction du point vers l'observateur

$\vec{B}$  : bissectrice entre les vecteurs  $\vec{L}$  et  $\vec{O}$   $= \text{normalise}(\vec{L} + \vec{O})$ , avec  $\vec{L}$  et  $\vec{O}$  unitaires.

Tous les calculs d'illumination se font dans le repère de la caméra en GLSL.

- Le calcul de la direction vers l'observateur ( $\vec{O}$ ) :

(un vecteur qui pointe vers le (0,0,0), c'est-à-dire vers la caméra)

```
observerPos = (-worldPos); // =(0-worldPos)
```

- La direction de la lumière ( $\vec{L}$ ) :

```
lightDir[i] = ( view * lights[i].position ).xyz - viewPos;
```

## A.2 Modèles de spot inspirés d'OpenGL

Un spot n'éclaire qu'à l'intérieur d'un cône, c'est-à-dire a une influence seulement si l'angle  $\gamma$  entre la direction du spot et la direction vers le point à éclairer est plus petit que l'angle d'ouverture  $\delta$  du spot. Lorsque c'est le cas, on a «  $\gamma < \delta$  » et mais on vérifiera plutôt si «  $\cos(\gamma) > \cos(\delta)$  » en évaluant des *produits scalaires* entre les vecteurs appropriés.

Pour déterminer la direction du spot dans le repère de la caméra, on peut calculer ce vecteur direction directement dans le nuanceur :

```
spotDir[i] = mat3(view) * -lights[i].spotDirection;
```

Pour avoir l'effet de bordure qui s'assombri vers l'extérieur, le modèle inspirés d'OpenGL utilise la formule suivante pour calculer le facteur qui multiplie l'intensité lumineuse du spot à l'intérieur du cône :

$$\text{fact} = (\cos(\gamma))^c$$

où :

$\cos(\gamma)$ :	est obtenue par	$(\vec{L} \cdot \vec{L}_n)$
$\cos(\delta)$ :	cosinus de l'angle d'ouverture	$= \cos(\text{LightSource.spotAngleOuverture})$
$\vec{L}_n$ :	direction du spot	$= \text{LightSource.spotDirection}[j]$
$c$ :	exposant du spot	$= \text{LightSource.spotExponent}$