



POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

INF2705 Infographie

Travail pratique 3

Stencil et illumination Phong, Gouraud et Flat

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

Table des matières

1	Introduction	2
1.1	But	2
1.2	Portée	2
1.3	Remise	2
2	Description globale	3
2.1	But	3
2.2	Travail demandé	3
3	Exigences	9
3.1	Exigences fonctionnelles	9
3.2	Exigences non fonctionnelles	10
A	Liste des commandes	11
B	Formules utilisées	12
B.1	Modèles de réflexion spéculaire de Phong et de Blinn	12
B.2	Modèles de spot inspirés d'OpenGL ou de Direct3D	13
B.3	Debugging avec RenderDoc	14

1 Introduction

Ce document décrit les exigences du TP3 « *Stencil et illumination Phong, Gouraud et Flat* » (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 stencil et d'illumination. Il sera en mesure d'effectuer des tests de stencil pour masquer certains éléments dans la scène. Il sera aussi capable de produire différents modèles d'illumination afin d'ajouter beaucoup plus de réalisme au scène 3D. Il sera en mesure d'implémenter différents types de source de lumière. Le travail fera l'utilisation des fonctions d'OpenGL `glStencilOp`, `glStencilFunc`, `glStencilMask` et des shaders de vertex, géometrie et fragment.

2.2 Travail demandé

Pour le tp3, un nouveau projet vous est fourni. Le projet contient quelques modifications aux classes existantes et un début de projet pour vous guider dans le tp3.

La classe `Camera` prend maintenant en argument une distance pour la camera en troisième personne qu'on peut modifier avec le déroulement de la molette. L'ajout de la classe `UniformBuffer` a été fait pour faciliter l'envois et le partage des données de lumières dans les nuanceurs. Dans la classe `Window`, on a ajouté une méthode pour cacher et montrer la souris et la gestion pour la librairie de `ImGui`.

Une restructuration du code a été faite pour gérer plusieurs scènes de rendu à la fois et partager les ressources entre les elles. Vous allez à partir de maintenant utiliser la classe `Resources` pour instancier les shader programs, les modèles et les textures. Ceux-ci seront accessible à partir de l'objet `m_res` dans la classe `Scene`. Au lieu de modifier la main loop dans le main, vous devez utiliser la méthode `Scene::render` pour générer votre affichage pour une scène en particulier.

Pour le déboguage, il peut être pratique d'utiliser l'application `RenderDoc` (voir l'annexe).

Partie 1 : Test de stencil

On demande d'implémenter un halo coloré qui nous permettra de distinguer qui sont les singes alliés et les singes ennemis dans la scène. Dans notre cas, nous avons 1 allié et 2 ennemis.

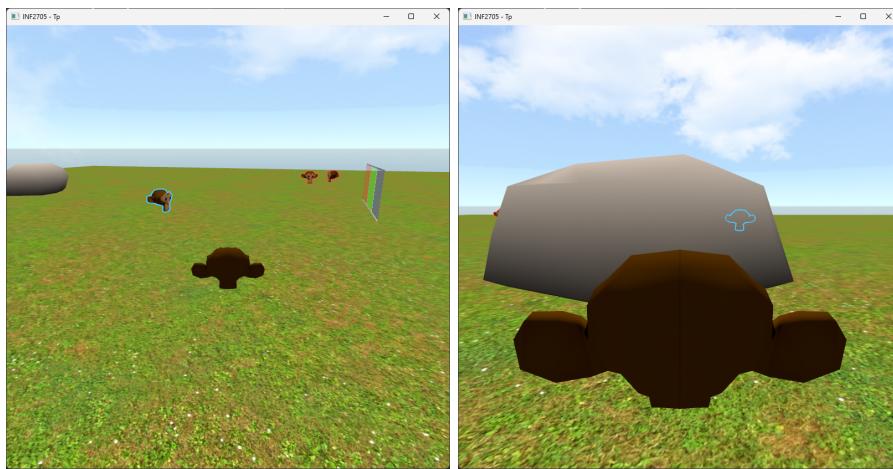


FIGURE 1 – Effet de halo. On voit le halo bleu au travers les autres objets.

L'utilisation de stencil est idéale pour produire cette effet visuel. On procéde généralement dans cette ordre :

- Dessiner l'objet avec le test de stencil pour écrire le tampon de stencil.
- Désactiver l'écriture du test de stencil.
- Modifier le test pour réussir seulement au endroit où l'objet n'est pas présent.
- Dessiner l'objet plus gros avec un shader simple d'une seule couleur.

Pour ce faire, vous devez activer le test de stencil lors du dessin des singes alliés et ennemis. Leur draw calls devront en tout temps écrire le stencil pour pouvoir définir leur contour. Pour chaque singe que vous dessinez, utilisez et modifiez un bit différent pour pouvoir les différencier (utiliser `glStencilMask` pour permettre l'édition que sur les bits spécifiés).

Les singes et leurs halos auront la même position. Il est recommandé de précalculer la matrice `mvp` une seule fois pour l'utiliser avec les modèles et les halos. Par la suite, on dessinera les halos avec un shader simple.

Le vertex shader (`simple.vs.gls1`) prendra en entrée la position et la normale de la vertex et n'a aucune sortie (autre que `gl_Position`). Il nous faudra l'attribut de normale des modèles 3D. Celui-ci est activable de la même façon qu'au `tp2` dans le chargement des données (déjà réalisé pour vous). Pour agrandir l'objet, une simple mise à l'échelle donnera un effet inconstant, tout autant plus que l'origine de l'objet n'est pas au centre du modèle. À la place, on déplacera les vertices dans la direction de leur normale d'un facteur constant. Ce facteur définira l'épaisseur de notre halo. Dans notre cas, une valeur de 0.1 est suffisante. N'oublier pas de normaliser votre normale.

Le fragment shader (`simple.fs.gls1`) prendra un uniform `vec3` de la couleur pour définir la couleur de sortie. Celle-ci définira la couleur du halo et devra être envoyé avant de dessiner l'objet.

Pour dessiner les halos, il faudra désactiver l'écriture du test de stencil. Le test doit passer lorsque le bit du singe traité est à 0 (où il n'est pas présent).

Pour le singe allié, on veut qu'il soit toujours visible au travers des autres objets. Vous allez devoir modifier la fonction du test de profondeur pour y parvenir.

Vous ne devez pas faire plusieurs appels à `glClear`. Activer le test de stencil seulement lorsque nécessaire.

De plus, on dessinera un mur de vitres colorées. On appliquera le test de blending pour pouvoir faire le mix de couleur avec la scène. Le facteur source sera l'alpha source (a_{src}) et le facteur destination sera un moins l'alpha source ($1 - a_{src}$). Le mur doit être visible des deux sens. N'oublier pas d'utiliser la transparence dans la couleur finale du fragment.

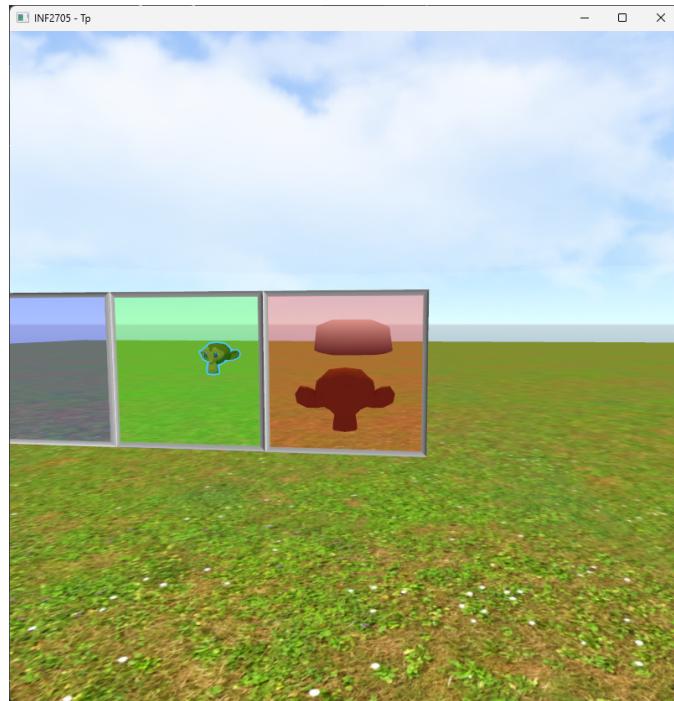


FIGURE 2 – Blending.

Partie 2 : Illumination

Même avec l'ajout des textures, nos scènes manquent de réalisme. Pour remédier à ce problème, l'ajout d'illumination va permettre de grandement améliorer le rendu.

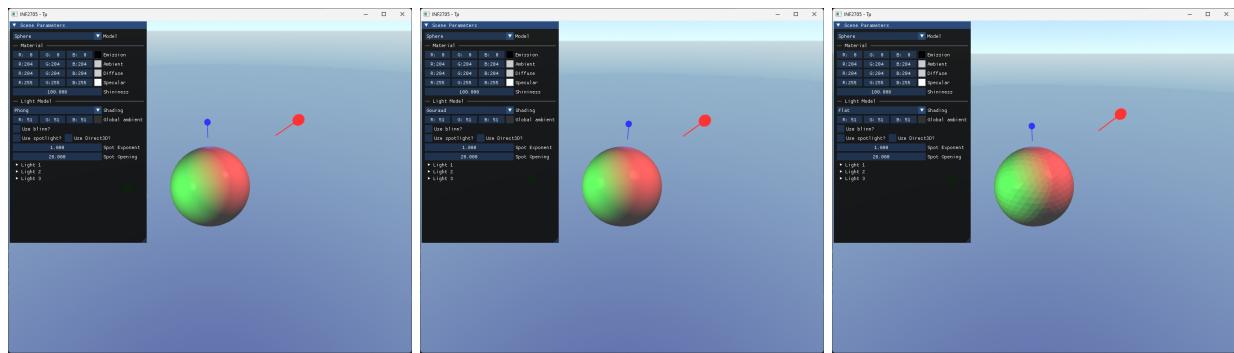


FIGURE 3 – Illumination de Phong, Gouraud et Flat.

Une scène de développement vous est fourni afin d'être en mesure de mieux observer le comportement de l'illumination. Celle-ci permet d'afficher un modèle qui sera éclairé par 3 lumières. Un affichage fait avec la librairie ImGui permet de facilement modifier les variables qui impactent le modèle d'illumination.

La majeure partie du code C++ a déjà été réalisée pour vous économisez du temps. Celui-ci fait la gestion du uniform buffer, gère le changement de shader, de modèle et le dessin des sources de lumière.

Pour cette partie, vous allez faire les modifications directement dans les shaders. On demande de faire le calcul d'illumination des sources de lumière positionnelles sans le facteur d'atténuation. Vous devez implémenter le modèle d'illumination de Phong, Gouraud et Flat. Il est recommandé de se baser sur les exemples vues en cours.

Voici quelques directives supplémentaires pour vous aidez :

- Un début de déclaration de shader a été fait dans chacun d'eux.
- 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.
- Le bloc d'uniforme `LightingBlock` est défini pour contenir les données d'illuminations. Vous ne devriez pas changer cette structure de données, au risque d'avoir des mauvais offset de mémoire.
- Le code de l'illumination est sensiblement le même dans les trois shaders. Vous allez être obligé de dédoubler le code dans les 3 shaders malheureusement.
- Les calculs devront être fait dans le référentielle de la vue.
- Faite la normalisation des vecteurs après la rasterisation pour les attributs utilisés dans le fragShader.
- N'oubliez pas d'appliquer la matrice de normale.
- Vous devez implémenter la réflexion spéculaire de Blinn et de Phong.

- Vous devez implémenter le calcul des spotlights avec le calcul d'OpenGL classique et le calcul de Direct3D.

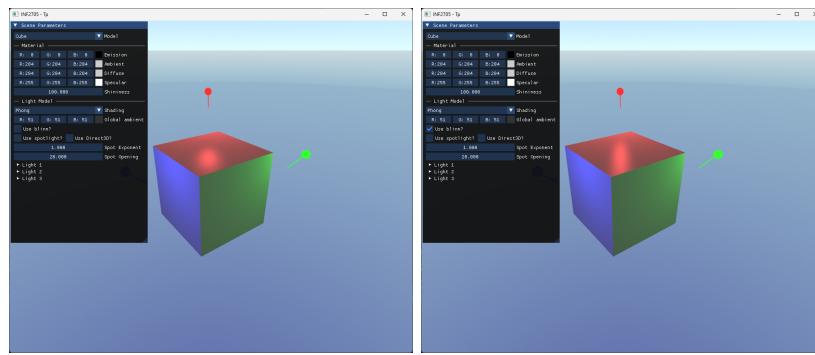


FIGURE 4 – Illumination de Phong, Gouraud et Flat.

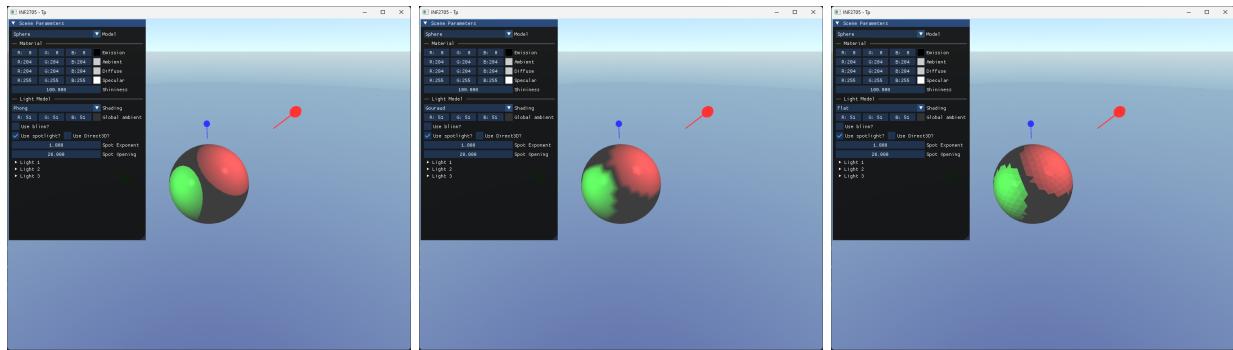


FIGURE 5 – Spotlight de Phong, Gouraud et Flat avec l'algorithme d'OpenGL classique.

Pour l'illumination Flat, on a besoin de calculer l'illumination par face. Le vertex shader calculera la position finale transformée par `mvp` et envoit les autres attributs tels quels. On aura un shader de géométrie pour faire le calcul d'illumination. Celui-ci calculera la normale par face de la façon suivante :

```
vec3 side1 = ( Vertex1 - Vertex0 );
vec3 side2 = ( Vertex2 - Vertex0 );
normal = cross( side1, side2 );
```

La position est le centre de la face du triangle. Vous devez calculer la position du centre. Les calculs d'illumination devrait être fait qu'une seule fois pour la face traitée.

Il est recommandé de commencer par l'illumination de Phong avec une seule lumière sans appliquer de texture. Par la suite, on peut facilement appliquer la même logique pour les autres lumières et les autres modèles d'illumination. Certaines formules sont expliquées davantage dans la section Formules utilisées et les notes de cours.

Dans l'illumination de Phong, on va pouvoir utiliser des textures pour paramétriser notre illumination. La couleur de la texture diffuse devrait seulement être prise en compte dans le facteur ambiant et diffus de l'illumination. Garder le alpha discard (si $\alpha < 0.3$, discard) pour la texture diffuse. La texture

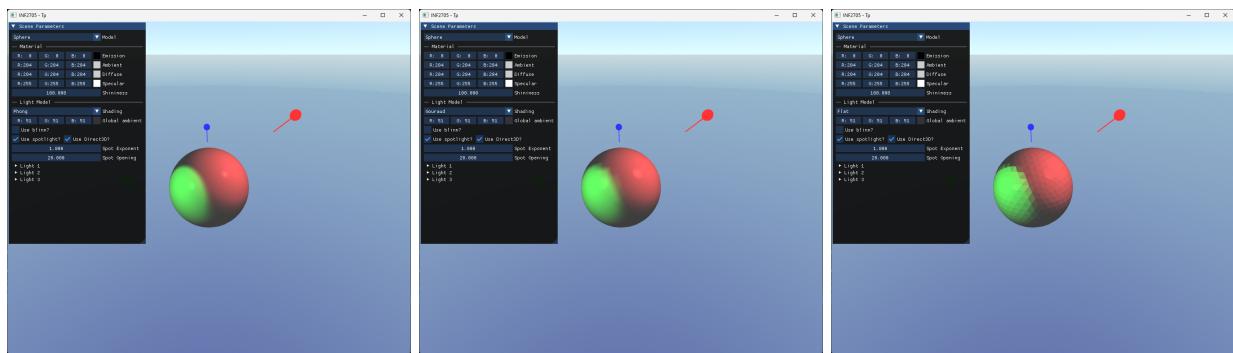


FIGURE 6 – Spotlight de Phong, Gouraud et Flat avec l'algorithme de Direct3D.

spéculaire sera appliquée sur le facteur spéculaire pour pouvoir mettre des reflets à des endroits spécifiques. Prenez note que c'est une texture à une seule composante.

Notez que la texture spéculaire est en teinte de gris (d'où la raison d'avoir une seule composante) ; principalement parce que la couleur spéculaire est dans la majorité des cas défini par la source de lumière plutôt que par le matériel. Seulement l'intensité nous intéresse ici.

Pour Gouraud, on envoie les quatres composantes (émission, ambiant, diffuse et spéculaire) séparées au fragment shader. Le fragment shader pourra les recomposer et appliquer les textures sur les bonnes composantes.

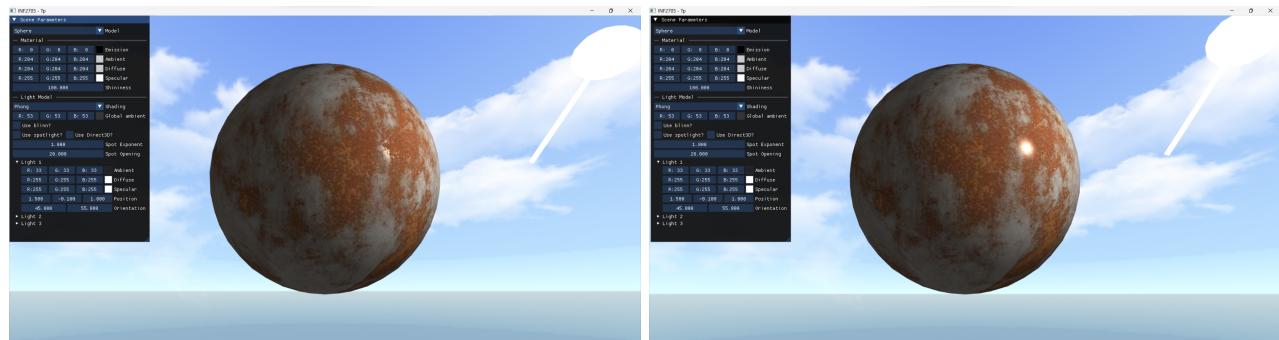


FIGURE 7 – Comparaison avec et sans lightmaps appliqués sur la sphère avec l'illumination de Phong.

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

- E1. Les matrices `mvp` sont précalculées pour les singes. [1 pt]
- E2. Le shader simple agrandie la géométrie et colorie celle-ci correctement. [1 pt]
- E3. Les paramètres de stencil sont correctes et permettent de modifier seulement le bit voulu. [1 pt]
- E4. Le stencil est modifié seulement lors du dessin des singes (les halos ne l'écrivent pas). [1 pt]
- E5. La halo est impacté seulement par le stencil écrit par le singe correspondant (le masque est correct). [1 pt]
- E6. Le halo des singes ennemis sont visibles que lorsqu'il n'y a aucun obstacle entre la vue et le singe. [1 pt]
- E7. Le halo du singe allié est visible en permanence d'au travers les autres objets. [2 pts]
- 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. [2 pts]
- E10. Les paramètres de blending sont correctes pour voir au travers de façon à teindre la vue. [2 pts]
- E11. Les halos sont visibles au travers de la vitre (et du contour de celle-ci, malheureusement). [1 pt]
- E12. Le mur est visible des deux côtés lorsqu'on le regarde. [1 pt]

Partie 2 :

- E13. Le modèle d'illumination de Phong est implémenté correctement. [5 pts]
- E14. Le modèle d'illumination de Gouraud est implémenté correctement. [5 pts]
- E15. Le modèle d'illumination de Flat est implémenté correctement avec un shader de géométrie. [5 pts]
- E16. La réflexion spéculaire de Blinn et Phong est implémenté correctement. [2 pts]
- E17. Les spotlights sont implémentées correctement avec la version d'OpenGL et Direct3D dans les 3 shaders. [5 pts]
- E18. Les calculs sont dans le référentiel de la vue. [1 pt]
- E19. Les normalisations sont effectués après la rasterisation plutôt qu'avant celle-ci. [1 pt]
- E20. Les propriétés du matériel sont utilisées correctement. [2 pts]
- E21. Les textures de lightmaps diffuse et spéculaire sont utilisées correctement pour les 3 shaders. [3 pts]
- E22. Les textures de lightmaps n'impactent pas les propriétés du matériel. [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. [5 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
espace	Activer/desactiver la souris
t	Changer de scène

B Formules utilisées

B.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)

```
attribIn.obsPos = (-pos); // =(0 - pos)
```

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

```
attribIn.lumiDir[i] = ( view * lights[i].position ).xyz - pos;
```

B.2 Modèles de spot inspirés d'OpenGL ou de Direct3D

Un spot n'éclaire qu'à l'intérieur d'un cône, c'est-à-dire a une influence seulement si l'angle γ entre la direction du spot et la direction vers le point à éclairer est plus petit que l'angle d'ouverture δ 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 :

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

La différence entre les modèles inspirés d'OpenGL et de Direct3D que nous utiliserons réside dans la formule pour calculer le facteur qui multiplie l'intensité lumineuse du spot à l'intérieur du cône :

- OpenGL utilise le facteur : $\text{fact} = (\cos(\gamma))^c$
- Direct3D utilise le facteur : $\text{fact} = \text{smoothstep}(\cos(\theta_{\text{outer}}), \cos(\theta_{\text{inner}}), \cos(\gamma))$

où :

$\cos(\delta)$:	cosinus de l'angle d'ouverture	$= \cos(\text{LightSource.spotAngleOuverture})$
\vec{L}_n :	direction du spot	$= \text{LightSource.spotDirection}[j]$
c :	exposant du spot	$= \text{LightSource.spotExponent}$
$\cos(\gamma)$:	est obtenue par	$(\vec{L} \cdot \vec{L}_n)$
$\cos(\theta_{\text{inner}})$:	est remplacé <i>dans ce TP</i> par	$\cos(\delta)$
$\cos(\theta_{\text{outer}})$:	est remplacé <i>dans ce TP</i> par	$(\cos(\delta))^{1.01+c/2}$

B.3 Debugging avec RenderDoc

RenderDoc est un outil de debugging open source pour les applications graphiques. Il est compatibles avec plusieurs API. Pour OpenGL, il faut avoir un contexte en mode core plus grand que 3.3 (ce qu'on utilise en ce moment). Il y a d'autres alternatives, comme Nvidia Nsight qui permet aussi de faire du debuggage, mais principalement sur les cartes nvidia.

Bref, si vous voulez l'utiliser, vous devez partir l'application directement dans RenderDoc. Mettre les path de l'executable et le working directory pour qu'il puisse charger les fichiers correctement. Une fois dans l'application, on peut cliquer sur F12 pour faire une capture de la frame.

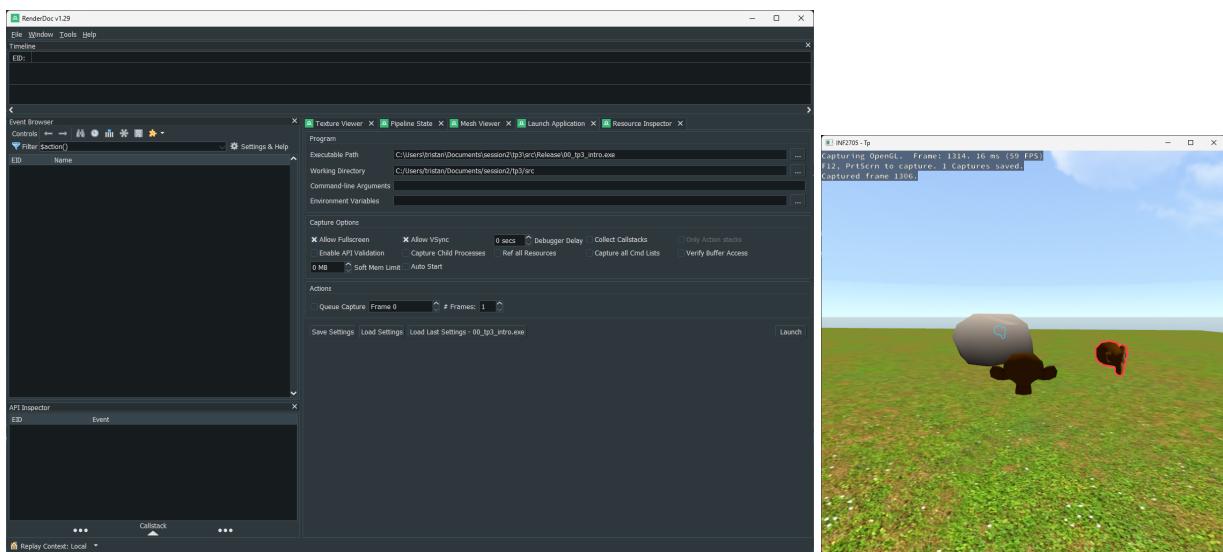


FIGURE 8 – Ouverture de l'application à profiler.

Un nouvel onglet devrait s'être ouvert avec les captures de l'application. Si double clic dessus, on a accès la liste des évènements graphiques. On peut les sélectionner et visualiser l'effet qu'ils ont eux sur les buffers dans l'onglet Texture Viewer.

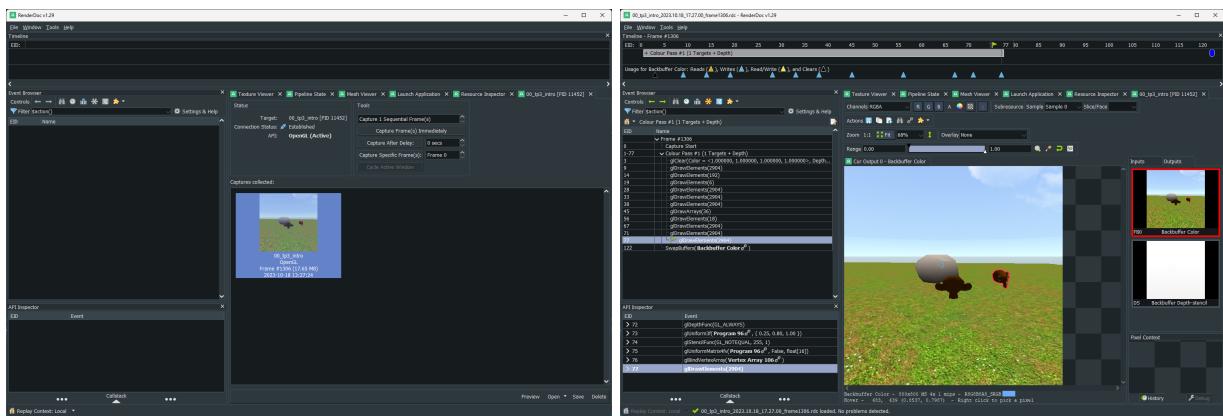


FIGURE 9 – Chargement de la capture et analyse du frame buffer.

Pour débugger le stencil, on peut le sélectionner à droite, puis cocher Stencil en haut. Les couleurs

seront mises en mode automatique. Avec l'icône de baguette, on peut ajuster le range automatiquement pour voir ce qui se passe sur le stencil. En mouse over ou clic droit, on peut aussi voir la valeur directement dans le buffer en hexadécimal (en bas).

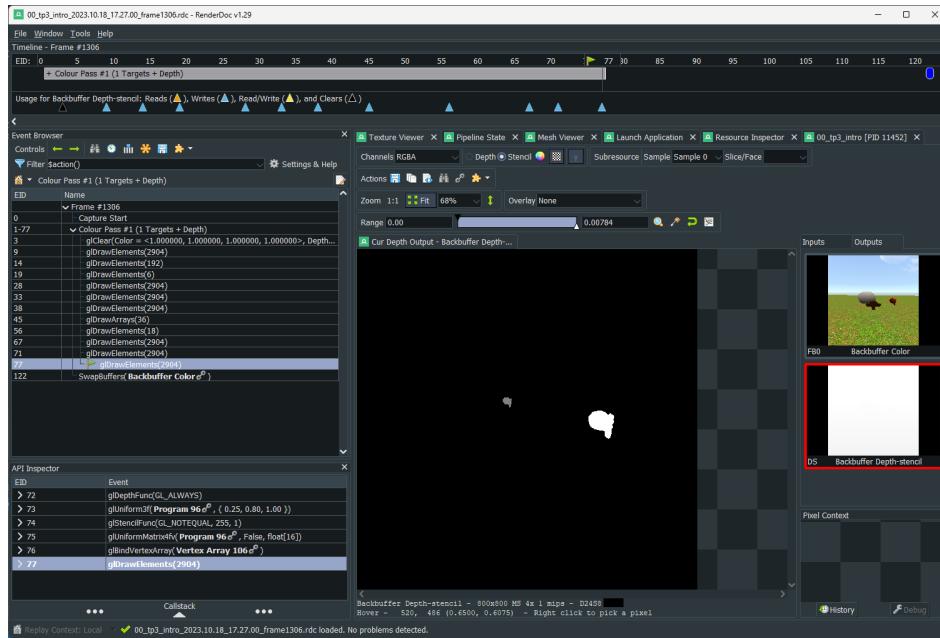


FIGURE 10 – Analyse du stencil buffer.

Tutoriel assez bref, n'hésitez pas à poser des questions sur l'outil. Il permet de faire bien plus que de débugger le stencil, je vous invite à l'explorer.