Programmation graphique 3D
M1 Informatique I3D - S1 Automne 2020–2021
Travaux Pratiques

TP 3 : Lighting et Shading (introduction)

Introduction

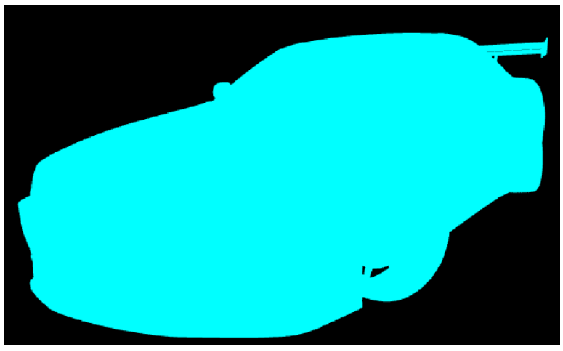
Dans ce TP, nous allons nous intéresser à l'éclairage d'un modèle représenté par un maillage de triangles. Nous allons voir en particluier le *shading* avec des BRDF diffuses et spéculaires (BRDF: Bidirectional Reflectance Distribution Function) qui caractérisent les propriétés d'un matériau lors des interactions avec un environnement lumineux. Le calcul d'éclairage que nous allons implémenter est une approximation de la réalité. Il s'agit d'un **modèle additif** : [Ambient + Diffus + Spéculaire].

Exercice 1 : Chargement d'un modèle

Vous manipulerez en entrée, un maillage avec des positions, des normales et des faces. Commencez par récupérer l'archive *nissan-gtr.zip* disponible sur moodle. Elle contient un ensemble de fichier texte au format *json* composant un modèle 3D de voiture découpé en plusieurs fichiers contenant chacun des informations de géométrie (vertices, indices) et d'apparence (matériau : Ka, Kd, Ks, Ns). Pour ce TP, il vous est fournis un code pour charger ce modèle 3D. Il n'y a pas de normales mais une fonction fournie vous les calculera automatiquement (parcours des triangles puis produit vectoriel de 2 de ses côtés, puis normalisation).

Il faut ensuite, créer les VBO (vertex buffer) associés (positions, normales et indices) et le VAO (vertex array) pour les contenir. Tout est déjà fait sauf pour les normales, à vous de jouer.

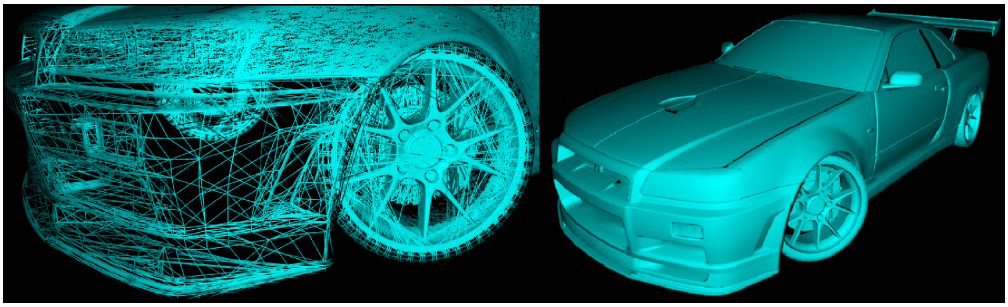
Afficher le maillage : sans lightging, avec une couleur uniforme, on ne peut pas voir la géométrie ...



Exercice 2 : Per-pixel shading

Ecrivez maintenant le vertex shader et le fragment shader dans lequel vous allez implémenter le lighting ("per-pixel").

Remarque : les calculs seront fait dans l'espace de la caméra ("View") où l'oeil est en zéro (c'est l'epace représenté par la matrice "View" qui définit les propriétés de la caméra : position, cible et vecteurs "up").



Vertex shader :

- **in** : positions [vec3], normales [vec3]
- **out** : positions [vec3], normales [vec3], dans le repère caméra (*View*).
- **uniform** :

  - matrices *modelMatrix*, *viewMatrix*, *projectionMatrix*" [mat4]
  - matrice *normalMatrix* [mat3] => les vecteurs ne se transforment pas comme les points ! On utilise la transposée de l'inverse de la matrice model-view pour transformer les normales. Voir les méthodes `inverse()` et `transpose()` de l'API easywgl (ou combinaison des 2 : `inverse3transpose()`) [NOTE: GLSL dispose également de ces méthodes dans les shaders mais ici ce n'est pas efficace car la matrice est constante pour tous les vertices]

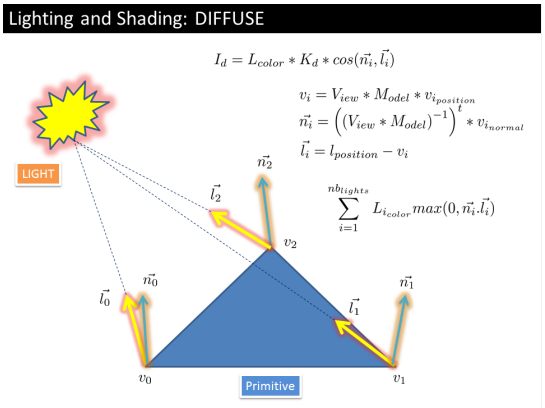
Fragment shader :

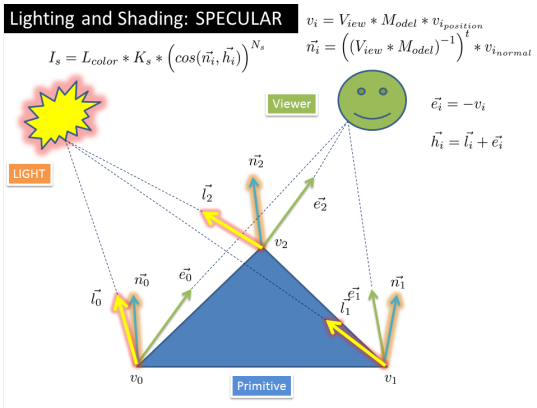
- **in** : positions [vec3], normales [vec3], dans le repère caméra (*View*). Valeurs interpolées par fragment.
- **out** : intensité lumineuse (ou couleur) [vec4] à envoyer au framebuffer OpenGL.
- **uniform** :

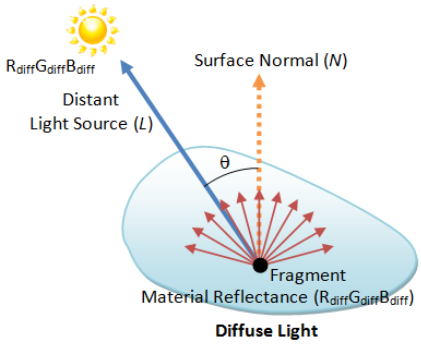
  - *LightIntensity* : intensité (et couleur) de la lumière [vec3].
  - *LightPosition* : pour un 1er test, on peut dire que la lumière ponctuelle est sur l'oeil, donc à la position (0,0,0) dans le repère caméra (pas besoin d'uniform). Mais on peut aussi envoyer une position [vec3].
  - Matériau : composante ambiante *Ka* [vec3] et diffuse du matériau *Kd* [vec3] de la partie du maillage courant. Si vous voulez gérer le spéculaire, il faut également ajouter *Ks* (couleur) [vec3] et *Ns* (taille de la tâche spéculaire) [float].

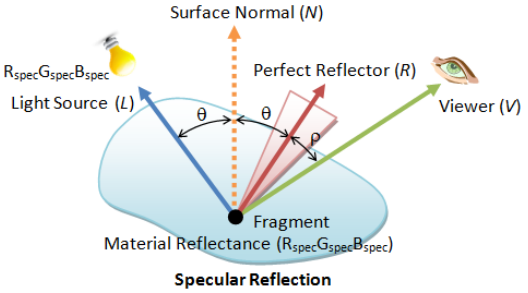
- **BUT** : calculer la couleur (ou intensité lumineuse) [vec3] en chaque fragment et l'envoyer au framebuffer d'OpenGL. Pour le diffus, on a besoin des vecteurs normalisés de la normale (n) et la direction de la lumière (lightDir). Pour le spéculaire, on a besoin des vecteurs normalisés de la normale (n) et de la direction de l'oeil (eyeDir).
L'intensité lumineuse (ou couleur) à calculer est la somme des 3 intensités ambiante, diffuse et spéculaire (modèle additif) : $I = Ia + Id + Is$ – avec :

  - AMBIANT (grossière approximation)
    Ia = lightIntensity * Ka
  - DIFFUS (ne dépend que de l'angle entre la normale et la direction de la source de lumière)
    Id = lightIntensity * Kd * cos(normal , lightDirection)
  - SPECULAIRE (ne dépend que de l'angle entre la normale, la direction de la source de lumière et la direction de l'observateur [la caméra])
    Is = lightIntensity * Ks * pow(cos(normal, lightDirection + viewDirection), Ns)



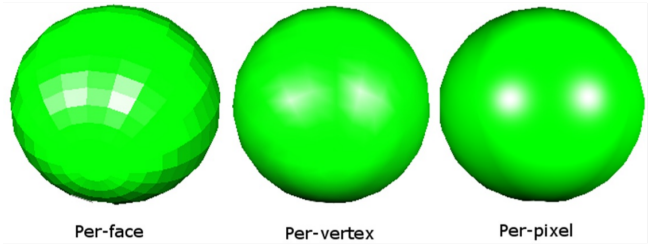






Améliorations : On peut améliorer le calcul de la BRDF diffuse en modulant l'intensité lumineuse par la distance de la lumière par rapport aux fragments :
Id = (lightIntensity / d) * Kd * cos(normal , lightDirection) – avec d = distance au carré de la lumière au fragment.
On peut aussi ajouter un coefficient pour chaque BRDF : I = (Ca * Ia) + (Cd * Id) + (Cs * Is)

Remarque : Ce calcul de shading aurait pû être fait dans le vertex shader (*per-vertex lighting*). C'était le cas dans l'ancienne partie fixe (cablée en hardware) du pipeline graphique d'OPenGL. Dans ce cas, on calcul l'éclairage en chaque vertex pour ensuite l'interpoler automatiquement à l'intérieur des primitives (pour les triangles) par l'étape de rastérization.
Cette approche est moins gourmande en ressources (plus rapide car potentiellement moins de vertices que de fragments [pixels]), mais offre de moins bon résultats :



Vous pouvez voir la différence dans la gestion de la réflectance sur les phares de la voiture. L'approche "per-vertex" ne permet pas de capturer les propriétés spéculaires du matériaux (=> le calcul n'est fait qu'aux vertices, et pas au milieu des triangles). L'approche "per-fragment" est plus coûteuse, mais permet de capturer cet effet.