Université de Limoges
Faculté des Sciences et Techniques
Année Universitaire 2016 - 2017

Master Informatique ISICG, semestre 1

Analyse et développement logiciel

Intégration du langage MDL au moteur GobLim

Rapport

version du 18 octobre 2017

Auteurs

Lemerle Joris
Nicot Fabien
Rouijel Mehdi
Le Gac Pierre

Responsable : M. Gilet Guillaume

Table des matières

Introduction

Objectif du projet

Ce projet s'inscrit dans une optique d'amélioration du moteur de rendu 3D GobLim développé au laboratoire Xlim ; que ce soit par le biais de projets donnés aux étudiants, ou par le travail des membres du département informatique du laboratoire, GobLim est en constante évolution.

Lorsqu'on parle de rendu 3D, et plus particulièrement de rendu réaliste, il devient rapidement apparent que la qualité du résultat obtenu est dépendante d'un certain nombre de facteurs ; réflexion, réfraction, transparence, absorption, émission, rugosité de la surface..., ces éléments sont en fait des résultantes de l'interaction entre la lumière et l'objet rendu à l'écran. Du fait de sa complexité et des limitations physiques des ordinateurs actuels, cette interaction est approximée sous la forme d'un modèle mathématique appelé fonction de distribution de réflectivité bidirectionnelle (ou BRDF, de l'anglais « Bidirectional Reflectance Distribution Function »).

Nous avons ainsi différentes BRDF avec différents paramètres pour différents types de surface. Il est alors évident que ce niveau de complexité devient d'autant plus difficile à gérer avec l'augmentation du nombre d'objets dans la scène dont on doit effectuer le rendu. L'approche usuelle pour aborder ce problème est de s'abstraire de cette complexité, au niveau de la construction de la scène, en définissant ce que l'on appelle des matériaux qui vont être associés aux objets. Un matériau est donc défini principalement par une BRDF, ainsi que plusieurs paramètres transmis à celle-ci (couleur de la surface, différents coefficients, etc.).

Le moteur GobLim contient sa propre implémentation pour définir un matériau. Cependant, celle-ci se réalise par modification directe du code source. Le travail que nous présentons ici se propose d'ajouter un niveau d'abstraction en limitant le besoin pour l'utilisateur d'interférer avec le code source. En améliorant ainsi sa facilité d'utilisation, le moteur GobLim se rapproche un peu plus de sa vocation à offrir une solution rapide et efficace de visualition utilisable par des personnes n'étant pas familières avec le langage C++ ou la programmation en général.

L'objectif final était donc de mettre en place une méthode aisée de définition de matériaux à intégrer au moteur. Nous avions initialement pour idée d'utiliser le langage MDL (« Material Definition Language » [1]) développé par NVIDIA. En plus des avantages qu'elle aurait apporté en termes de définition de matériaux, cette API aurait permis de faciliter les échanges de matériaux avec d'autres applications l'implémentant ; par exemple, il aurait été possible de créer un matériau à l'aide du logiciel Substance Designer [2], de la société allegorithmic, pour ensuite effectuer le rendu avec GobLim. Malheureusement, pour différentes raisons mentionnées plus loin dans ce rapport, cette idée a dû être abandonnée.

Le travail que nous avons réalisé offre moins de possibilités en cela que c'est une fonctionnalité interne spécifique au moteur GobLim, et que nous n'avons pas pu nous

permettre d'implémenter un niveau égal d'options — tant en termes de qualité que de quantité — que celui offert par le langage MDL et son kit de développement, sur lesquels NVIDIA travaille depuis plusieurs années. Néanmoins, l'objectif d'abstraction peut-être atteint sous une forme plus basique.

Abandon du langage MDL

La communication avec NVIDIA s'est avérée difficile. Notre projet et nos demandes d'accès au kit de développement pour nous permettre de commencer notre implémentation ont révélé un schisme au sein de l'entreprise. Certains membres, en particulier du côté de la branche française, se sont montrés intéressés par notre projet et prêts à nous fournir l'aide nécessaire.

Cependant, en remontant la hierarchie du projet MDL, l'idée n'a pas été aussi bien reçue. Le programme autour du langage MDL est relativement exclusif ; le rejoindre et obtenir l'accès au SDK développé par NVIDIA est soumis à approbation, et il n'est apparemment pas leur souhait d'intégrer des groupes universitaires.

Dès le premier semestre, nous avions émis une requête pour l'accès au kit de développement, en essayant de nous inscrire au programme DesignWorks[3] sur le site web de NVIDIA. Après plusieurs semaines sans réponse, un message envoyé au service de contact du programme nous a valu une réponse du chef de produit MDL nous indiquant qu'en l'absence d'un programme pour accorder des licences au niveau académique, notre demande ne pouvait pas être accordée.

Des e-mails échangés avec la branche française par la suite nous ont donné l'espoir d'une résolution du problème ; malheureusement NVIDIA n'a pas changé sa position. La raison qui nous a été donnée est que les équipes techniques en charge du langage craignaient « de ne pouvoir apporter le soutien supplémentaire aux différentes universités qui ont fait la requête d'y accéder » [1]. En fin de compte, l'accès au kit de développement nous à tout bonnement été refusé.

Des échanges supplémentaires nous ont appris l'existence d'un projet « open-source » d'analyse syntaxique (ou « parsing ») de fichier MDL, et un éventuel accès au dépôt Git de celui-ci a été évoqué mais cette idée est restée sans suite.

La seule option restante était alors d'implémenter notre propre « parser » de fichier MDL, en nous basant sur la spécification du langage disponible en ligne[4]. Cette solution sortait du cadre du projet, en cela qu'elle représentait une tâche relativement conséquente en elle-même et, surtout, qui ne serait pas en lien direct avec l'informatique graphique. Étant donné le retard déjà accumulé, l'ajout de cette étape à notre projet aurait représenté une quantité de travail trop importante pour que celui-ci soit réalisable dans le temps imparti. La décision finale a donc été d'abandonner complètement le langage MDL.

1. Extrait d'un mail de G. Polaillon, Sr. Alliance Manager à NVIDIA

Nous avons perdu beaucoup de temps sur l'ensemble du projet dans l'attente et la recherche d'un moyen de contourner ces obstacles. En définitive, tout notre travail préliminaire du premier semestre sur l'étude du fonctionnement du langage MDL et les possibilités en terme d'intégration au moteur GobLim aura été en vain.

Redéfinition du sujet

Suite au choix qui a été fait d'abandonner le langage MDL, nous avons dû avoir une réunion afin de faire une mise au point sur l'état du projet et prendre une décision quant à sa suite. Plutôt que d'abandonner complètement le sujet de départ, nous l'avons redéfini pour en rester au plus proche, tout en excluant l'intégration du langage MDL.

La décision à laquelle nous sommes arrivés a été de développer une nouvelle fonctionnalité pour le moteur GobLim reprenant les grandes idées du langage MDL, tout en nous abstrayant des considérations à avoir si nous avions implémenté un « parser » respectant la spécification relativement élaborée de celui-ci.

Le sujet de départ était axé sur le langage MDL, l'API autour de lui, et une intégration fonctionnelle au moteur GobLim avec l'implémentation de plusieurs BRDF d'exemple. Nous avons dû opérer un recentrage de notre objectif pour au final concentrer nos efforts sur le développement cette nouvelle fonctionnalité.

Plusieurs possiblités se présentaient à nous dès le départ quant à la forme qu'allait prendre la définition d'un matériau. Nous aurions pu définir notre propre format de fichier à analyser, mais nous avons jugé préférable d'opter pour une solution plus réaliste, étant donnés les délais dans lesquels nous devions travailler.

Dans cette optique, deux principaux candidats ont été considérés : XML[5] et JSON[6]. Notre choix s'est porté sur ce dernier, en particulier du fait de sa simplicité et de la découverte d'une API relativement simple d'utilisation pour l'intégrer à un projet en C++.

Pour la définition d'un matériau, nous utilisons donc des fichiers au format JSON. L'API en question est « JSON for Modern C++ »[7] par Niels Lohmann pour le « parsing ». Grâce à elle, nous avons pu éliminer la nécessité d'implémenter nous-mêmes l'analyse des fichiers pour nous concentrer sur l'intégration à GobLim ; l'API permet d'extraire aisément les données pertinentes pour les intégrer au code C++.

Nous utilisons les données extraites pour construire des shaders GLSL qui peuvent être appliqués aux objets de la scène rendue. L'implémentation est détaillée plus loin dans ce rapport.

Pour la conception générale du projet, nous nous sommes intéressés à une application reprenant la même idée générale : l'outil « BRDF Explorer »[8] développé par Walt Disney Animation Studio.

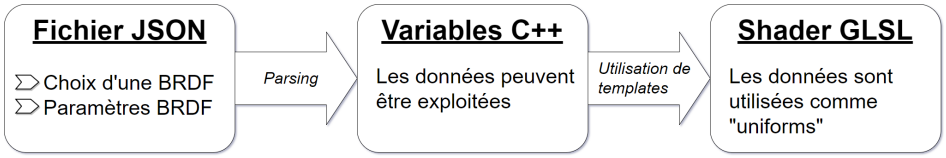


FIGURE 1 – Schéma basique de l'implémentation

Dans la première partie de ce rapport, nous revenons sur la définition d'une BRDF, sur ce que nous avons appris de l'application BRDF Explorer de Disney, ainsi que sur les différnts modèles de BRDF avant d'en voir quelques exemples. Dans une deuxième partie, nous regardons ensuite de plus près le travail réalisé, en détaillant l'intégration au moteur Goblim, puis en mentionnant les améliorations envisageables. Ensuite, nous nous penchons dans une troisième partie sur les résultats obtenus, avant d'aborder, dans une dernière partie, les difficultés rencontrées quant à l'implémentation.

1 BRDF

1.1 Qu'est-ce que le modèle de BRDF ?

Une BRDF ou Réflectivité Bidirectionnelle définit les propriétés de réflexion d'un matériau. Pour définir ces propriétés il faut définir la quantité de lumière reçue par le matériau mais il faut surtout définir la réflexion de la lumière.

$$L_o = \int_{2\pi} f_v(\Omega_i, \Omega_r) L_v * (\Omega_i) d\Omega_i$$

La fonction de transport de la lumière nous montre que pour avoir un rendu parfait de la réflexion de la lumière, il faut calculer la quantité de lumière entrante et la lumière réfléchie dans toutes les directions.

Le calcul scientifique du transport de la lumière étant trop complexe pour être résolu, il faut calculer la réflexion dans toutes les directions à chaque endroit du matériau car il n'est pas uniforme.

Il faut donc trouver des approximations comme définir les matériaux de manière homogène avec quelques variables.

1.1.1 Luminance entrante

La luminance entrante est la quantité de lumière qui touche l'objet à un certain endroit. La lumière peut être émise par une source lumineuse mais elle est aussi émise de manière indirecte par la réflexion des objets.

Dans notre cas nous ne prendrons en compte que la lumière provenant de sources lumineuses. Nous admettrons aussi que la quantité de lumière arrivant sur un objet dépend de l'angle entre la surface de l'objet et le rayon de lumière. Elle est déterminée par le produit scalaire entre la surface et la direction de la lumière par rapport à la surface.

$$I(\theta) = I(0)cos(\theta)$$

Cette quantité de lumière doit être pondérée par l'intensité de la lumière émettrice. L'angle θ est l'angle entre la normale à la surface et le vecteur direction du point de la surface regardé jusqu'à la lumière 0 et I(0) l'intensité de la lumière 0.

1.1.2 Luminance sortante

La luminance sortante est la quantité de lumière réfléchie par un objet. Cette luminance dépend de nombreux facteurs. Cela dépend surtout du matériau, mais aussi de l'angle de la lumière.

Il existe deux types de réflectances : la première étant la BRDF qui est une fonction permettant de déterminer la lumière réfléchie sur un point d'une surface. Cette méthode est apparue vers 1965, découverte par Fred Nicodemus.

La deuxième est la BSSRDF, qui permet de décrire les matériaux translucides, par exemple le lait, l'eau ou encore la peau humaine.

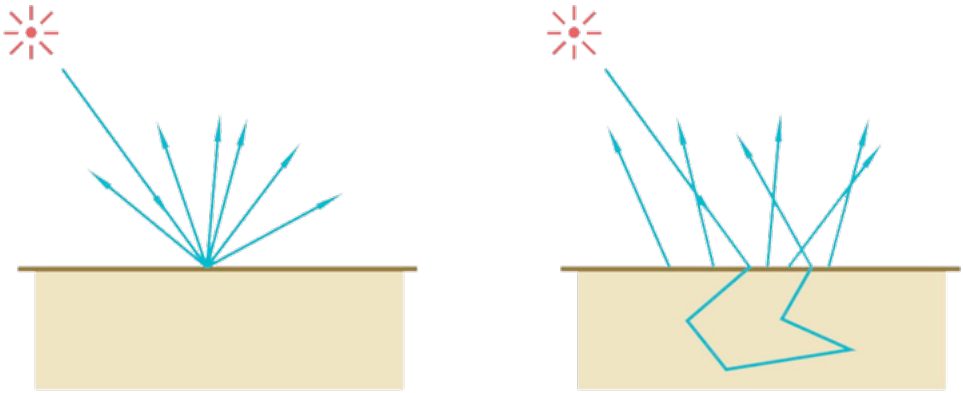


FIGURE 2 – Reflexion BRDF à gauche et BSSRDF à droite

Afin de faciliter les calculs nous n'allons implémenter que des méthodes de rendu de BRDF.

Une BRDF peut se faire de plusieurs manières :
- De manière empirique comme le modèle de Phong.
- En évaluant des fonctions mathématiques et physiques provenant de modèles analytiques comme l'équation de Fresnel qui est utilisée dans le modèle de Cook-Torrance.
- En utilisant les propriétés réelles d'un objet que l'on a mesuré au préalable dans toutes les directions de la lumière et en fonction des longueurs d'ondes.

Pour mesurer la réflexion d'un objet on peut utiliser un goniomètre qui permet, à l'aide d'une source lumineuse mobile ainsi que d'un capteur, de mesurer la réflexion de la lumière avec une lumière à différentes positions.

Les fonctions des réflectances utilisent quelques paramètres de base permettant de la définir.
- Un vecteur Ω_i qui correspond à la direction de la lumière sur le point (lumière entrante).
- Un vecteur Ω_r correspondant à la réflexion de la lumière en partant du point (lumière sortante).
- Un vecteur n correspondant à la normale du point à la surface.

— Ainsi que les deux tangentes à la surface formant un repère orthogonal avec la normale.

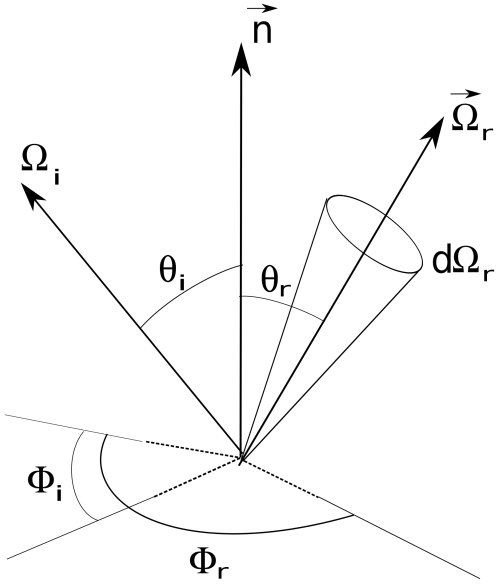


FIGURE 3 – Différents vecteurs nécessaires pour calculer la réflectance

Donc si on couple la lumière entrante en un point et la lumière sortante, on obtient la couleur d'un point. Cela se traduit par la formule suivante :

$$L_o = BRDF(\Omega_i, \Omega_o) L_i * cos(\theta_i)$$

Avec comme variable la BRDF qui est la fonction de réflectance décrivant la luminance sortante d'un objet. Cette fonction de réflectance doit être multipliée par la luminance entrante afin d'obtenir la couleur du point pour la lumière i. Cependant si l'on souhaite en gérer plusieurs, il faut répéter cette action pour toutes les lumières. On obtient donc cette formule :

$$L_o = \int_{j=1}^{m} BRDF(\Omega_i^j, \Omega_o) L_i^j * cos(\theta_i^j)$$

1.2 BRDF Explorer de Disney

Le studio Disney a commencé à s'intéresser au rendu physiquement réaliste (PBR Physically Based Rendering) suite au film Raiponce, sortie en 2010, dans lequel le rendu des cheveux de la princesse a été réalisé grâce à une approche basée sur la physique. Suite aux bons résultats du rendu obtenu, Disney a décidé de se concentrer un peu plus dans ce domaine.

Rapidement les développeurs de Disney se sont rendu compte que certains modèles de rendu fonctionnaient plus en fonction des matériaux qu'ils souhaitaient obtenir.

Ils ont donc décidé de réaliser un logiciel permettant de visualiser des BRDFs de matériaux mesurées au préalable. Le logiciel peut lire des fichier binaires fournis par exemple par la base de données de BRDFs MERL.

Le logiciel permet aussi de charger différents modèles de BRDF empiriques avec de nombreux paramètres afin de régler, le plus précisément possible, les modèles empiriques.

Il permet aussi de comparer une BRDF mesurée avec des modèles empiriques. Le logiciel dispose aussi de nombreuses vues permettant de comparer les modèles sous différents angles.

- La représentation 3D permet de visualiser la réflexion dans toutes les directions.
- La courbe Theta H montre le pic spéculaire.
- La courbe Theta D montre la réponse de Fresnel.

Le logiciel permet donc de se rapprocher le plus possible du rendu réel des matériaux en utilisant de modèles de rendu empiriques, en modifiant les variables de contrôle de ces rendus.

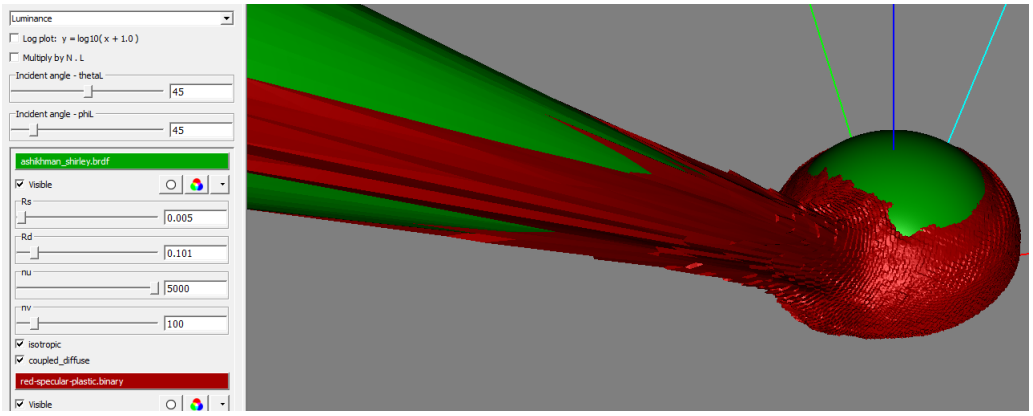


FIGURE 4 – Exemple de la BRDF Explorer de Disney

Dans cet exemple on peut voir la comparaison entre la BRDF du plastique rouge spéculaire ainsi que la BRDF du modèle d'Ashikhman-Shirley avec les diverses variables choisies afin que les deux BRDFs soient les plus ressemblantes possibles.

1.3 Les modèles de BRDF

Les modèles de BRDFs sont divisés en 3 catégories : les modèles diffus, les modèles spéculaires mais aussi les modèles combinant diffus et spéculaire dans la même fonction de réflectance.

La diffusion de la lumière sur un matériau se produit quand l'objet est rugueux et que la caméra ne se trouve pas proche de la direction du vecteur de réflexion de la surface. La lumière entrante est renvoyée dans toutes les directions.

Le reflet spéculaire de la lumière sur un matériau apparaît lorsque la direction du vecteur de réflexion est proche de la position de la caméra. La lumière entrante est majoritairement renvoyée vers le vecteur de réflexion.

La plupart des matériaux combinent les deux types de modèles, cependant un matériau uniquement spéculaire peut être considéré comme un miroir parfait, et un modèle purement diffus reflète la lumière sans prendre en compte la réflexion de la lumière.

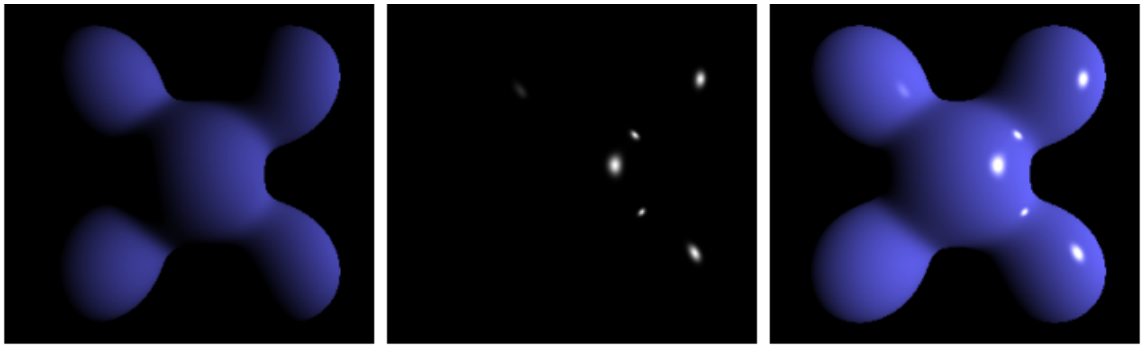


FIGURE 5 – Modèle diffus à gauche, spéculaire au milieu et les deux à droite

Toutes les BRDF que nous avons implémentées dans le moteur Goblim sont tirées du logiciel BRDF Explorer de Disney que nous avons présenté plus tôt.

1.3.1 Modèles diffus

Il y a 3 modèles diffus que nous avons implémentés dans le moteur Goblim.

Il y a d'abord la BRDF de Lambert, qui indique que la lumière est reflétée uniformément quel que soit la direction de la réflexion. Elle n'utilise qu'une variable qui définit la quantité de lumière reflétée. Ce modèle de rendu tire son nom de la loi de Lambert qui permet de calculer la quantité de lumière entrante sur un objet.

Il y a aussi la BRDF de Minnaert qui définit la quantité de lumière réfléchie de manière plus homogène lorsque la quantité de lumière est faible. On peut contrôler cette BRDF à l'aide d'une variable k permettant de contrôler cet effet. Ce modèle a été réalisé pour les objets dont le reflet diffus diminue peu sur les bords, par exemple la lune dont on a l'impression que son reflet est uniforme.

Enfin il y a la BRDF de Oren et Nayar qui sera présentée plus en détail dans la prochaine partie.

1.3.2 Modèles spéculaires

Il y a 10 modèles spéculaires que nous avons implémentés dans le moteur Goblim.

Tout d'abord nous avons implémenté Phong qui est une BRDF utilisant le produit scalaire du vecteur de réflexion avec le vecteur caméra afin de détecter si la caméra est proche du vecteur de réflexion. Ce produit scalaire est mis en puissance par une variable qui permet de définir la réflectivité du matériau. Phong permet de réaliser des reflets spéculaires de petite taille qui s'atténuent de façon régulière quand l'angle entre la réflexion et la caméra augmente.

Stretched Phong, quant à lui, est une version améliorée de Phong. Il permet de rajouter une variable qui permet de contrôler le reflet spéculaire lorsque la lumière est rasante sur le matériau.

Blinn Phong fonctionne sur le même principe que Phong sauf qu'il n'utilise pas l'angle entre le vecteur de réflexion et le vecteur de la caméra mais l'angle entre le vecteur H, dit « halfway » et la normale. Le vecteur halfway désigne le vecteur à mi chemin entre la lumière et la caméra. On utilise une variable n pour contrôler la puissance comme dans Phong. L'utilisation du vecteur halfway rend le pic spéculaire plus large par rapport à Phong.

Nishino est une BRDF utilisant aussi le produit scalaire entre la normale et le vecteur halfway. Elle utilise deux variables, la variable n qui permet de contrôler la largeur du pic spéculaire et la variable k contrôle la puissance du pic spéculaire. Nishino permet de réaliser de petits reflets spéculaires, mais on peut contrôler la vitesse d'atténuation du reflet spéculaire.

Le modèle de Beckmann utilise une fonction de distribution des micro facettes pouvant être contrôlée par la variable m qui définit la rugosité du matériau. Ce modèle produit des reflets spéculaires très puissants avec très peu d'atténuation. Il peut être utilisé pour des matériaux très réfléchissants.

Le modèle Exponential utilise une fonction exponentielle inverse contrôlée par la variable c et l'angle entre la normale à la surface ainsi que le vecteur halfway. Ce modèle est utilisé pour des matériaux très peu spéculaires car l'atténuation du reflet et très faible quand l'angle augmente.

GGX est un autre modèle utilisant une distribution réalisée grâce au cosinus carré et la tangente carré de l'angle entre la normale et le vecteur halfway. Cette distribution peut être contrôlée grâce à la variable α. Elle permet de produire des reflets spéculaires très puissants mais qui gardent quand même une atténuation visible quand le pic spéculaire est fin.

Trowbridge-Reitz est une BRDF que l'on peut contrôler à l'aide de deux variables uniquement, la première est la variable c et elle permet de contrôler la largeur du pic spéculaire. La deuxième variable permet de contrôler si l'on veut modifier la taille du pic spéculaire. On peut utiliser ce modèle de la même manière que le modèle Exponential, à part que son pic spéculaire est plus doux.

La BRDF de Blinn est adaptée du modèle de Torrance Sparrow. Il utilise une fonction de distribution des micro facettes qui peut être contrôlée par la variable n, et une fonction G qui détermine le nombre de micro facettes cachées par d'autres micro facettes. Il se sert aussi de la fonction de Fresnel pour déterminer la réflexion contrôlée par la variable *ior*. L'ajout de la fonction de Fresnel impacte le reflet du matériau lorsque la réflexion rase le matériau. Cette BRDF est utile pour définir des matériaux très peu spéculaires.

Enfin il y a la BRDF de Cook et Torrance qui sera présentée en détail dans la prochaine partie.

1.3.3 Modèles diffus et Spéculaire

La BRDF Ashikhman et Shirley possède six variables pour contrôler la réflectance. Rs etRd sont les coefficients de spécularité et de diffusion, nu est la taille du pic spéculaire, nv contrôle l'orientation du pic spéculaire si l'objet est anisotrope. La dernière valeur permet de déterminer si le spéculaire et le diffus sont couplés.

Ward permet de bien contrôler les reflets spéculaires sur les objets anisotropes grâce à deux de ces variables $alphaX$ et $alphaY$ déterminant la largeur du pic spéculaire dans l'axe X et dans l'axe Y. Il permet aussi de définir séparément la couleur des composantes diffuse et spéculaire.

Walter utilise la distribution de Fresnel ainsi que GGX. La BRDF est contrôlée par plusieurs variables. Ks et Kd sont les coefficients de spécularité et de diffusion, la variable *ior* permet de contrôler la fonction de Fresnel, et $alphaG$ permet de contrôler la fonction G de GGX.

1.4 Exemple de BRDF

1.4.1 Oren et Nayar

La BRDF d'Oren et Nayar permet de déterminer la rugosité d'un objet en utilisant une distribution des micro facettes grâce au modèle définit par Torrance et Sparrow. Elle permet de déterminer la quantité de facettes orientées vers la caméra mais aussi les facettes cachées.

FIGURE 6 – Comparaison du modèle Oren et Nayar avec Lambert

Du fait de la distribution aléatoire des micro facettes, il y a toujours une partie de la surface dont l'orientation permet la réflexion de la lumière vers la caméra, même lorsque la lumière rase l'objet.

Concernant les variables utilisées par le modèle nous avons *rho* qui est le coefficient de diffusion de l'objet et nous avons la variable *sigma* qui permet de contrôler l'intensité de la réflexion lorsque la lumière rase l'objet.

1.4.2 Cook et Torrance

La BRDF de Cook et Torrance utilise une distribution de micro facettes afin de déterminer la composante spéculaire en fonction de la rugosité de la surface. La distribution de Beckmann est utilisée dans cette BRDF pour calculer la réflexion de manière physiquement réaliste.

Nous ajoutons aussi la contribution du facteur de Fresnel pour calculer la réflexion. Nous utilisons pour cela l'approximation de Schlick. La BRDF utilise aussi une fonction d'atténuation géométrique appelé G, qui permet de prendre en compte les micro facettes qui sont dans l'ombre d'autres micro facettes. Cette fonction G influe lorsque le reflet spéculaire est rasant sur l'objet.

Il y quatre variables qui peuvent être contrôlées pour modifier le comportement de la BRDF. Tout d'abord il y a la variable m qui permet de modifier la rugosité du matériau contrôlant la fonction de distribution de Beckmann. Nous utilisons aussi la variable $f0$ permettant de contrôler le terme de Fresnel. Il y a aussi deux variables $includeF$ et $inculdeG$ permettant l'utilisation ou non du terme de Fresnel pour le premier ainsi que la fonction d'atténuation géométrique G pour la seconde variable.

2 Implémentation

2.1 Fonctionnement général

2.1.1 Import de matériau depuis un fichier

Comme expliqué précédemment, notre projet a dû être repensé suite à l'échec de l'obtention du SDK de Nvidia qui nous aurait servi pour lire et utiliser les fichiers MDL afin de stocker nos définitions de matériaux. Il nous a donc fallu, en premier lieu, trouver un moyen alternatif à ces fichiers. Nous avons donc choisi de stocker nos matériaux dans des fichiers au format JSON (JavaScript Object Notation), qui est un format textuel très simpliste quant à sa syntaxe et à son utilisation de données. Cependant, la lecture / écriture de ce fichier n'étant pas supporté nativement par le C++, langage avec lequel est développé GobLim, nous avons donc recherché une API permettant de réaliser cette tâche sans avoir à coder un moteur de parsing nous-même. Au terme de cette recherche nous avons retenu trois API :

— JsonCpp : `https://github.com/open-source-parsers/jsoncpp`
— Json de Nlohmann : `https://github.com/nlohmann/json`
— RapidJSON : `http://rapidjson.org/index.html`

Après comparaison des points forts et des faiblesses de chacune, nous avons conclu que le parser Json de Nlohmann serait plus adapté à nos besoins : sa syntaxe est simple et facile à relire, sans forcément avoir besoin de la documentation sous la main, car elle se rapproche énormément de celle du Javascript natif (utilisation des opérateurs [] pour accéder à un élément du document JSON par exemple). Il n'y a également pas de typage fort, ce qui assouplie encore plus son utilisation : une valeur numéraire stockée dans un fichier pourra être ainsi récupérée en tant que nombre flottant à simple ou double précision ou en entier par exemple. En comparaison, RapidJSON semblait être une bibliothèque plus complète quand à ses possibilités d'utilisation, mais ayant déjà accumulé du retard sur le projet et n'ayant que peu de besoins et contraintes sur l'import des matériaux, nous avons préféré nous tourner vers l'API de Nlohmann.

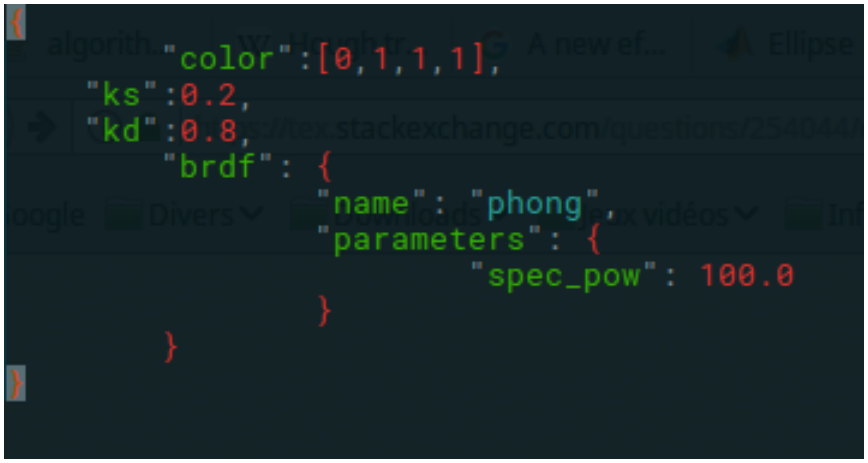


FIGURE 7 – Exemple de matériau au format de fichier JSON

Concrètement, un matériau sera décrit par plusieurs paramètres dans le fichier : sa couleur, les modèles d'éclairage utilisés avec leurs paramètres secondaires associés, etc...

```
//Chargement du fichier via flux dans un objet json
ifstream input_stream(filename);
json json_file;
input_stream >> json_file;

//Premier paramètre : couleur du matérial
glm::vec4 color;

//si paramètre trouvé
if (json_file.find("color") != json_file.end()) {
        color = glm::vec4(json_file["color"][0].get<float>(),
                json_file["color"][1].get<float>(),
                json_file["color"][2].get<float>(),
                json_file["color"][3].get<float>()
        );
}
```

FIGURE 8 – Import du fichier JSON et lecture des paramètres

2.1.2 Passage des variables aux shaders

La deuxième étape dans la phase de développement a été de concevoir et créer la classe BRDFMaterial, une classe héritant de MaterialGL, qui permet dans GobLim d'appliquer des matériaux aux objets et différents mesh durant le rendu en temps réel. Cette classe, en comparaison de celles que nous avions déjà pu réaliser durant le module de moteur 3D, se devait d'être généraliste, c'est à dire que d'après les paramètres extraits depuis un fichier JSON, son fonctionnement puisse s'adapter. En effet, chaque matériau peut avoir un modèle d'éclairage différent, et ces modèles n'ont pas forcément les mêmes paramètres. Deux choix s'offraient pour résoudre cette question : le premier était d'écrire en « dur » la

valeur des paramètres du matériau directement au sein du shader GLSL. Cette solution de facilité n'était cependant pas la plus optimale et pertinente. En effet, comme il sera expliqué plus loin, il avait été convenu de pouvoir gérer le comportement des matériaux à l'aide de l'interface de GobLim pendant la phase d'exécution, de pouvoir modifier la valeur des paramètres, etc... Nous avons donc opté pour le second choix d'implémentation, qui était de définir les paramètres en tant que variables uniforms dans le shader, et que les valeurs de ces variables seraient passées directement depuis le code host du moteur. BRDFMaterial a donc, pour chaque type de variable uniform, une fonction pour lire / modifier la valeur de la variable dans le code host, et de choisir dans quel shader cette variable est utilisée (vertex, fragment, etc...).

FIGURE 9 – Exemple des fonctions de création / lecture pour les uniforms de type float

Ici nous pouvons voir le code pour pouvoir créer / modifier des variables de type float en GLSL au sein de l'instance de BRDFMaterial. L'utilisation de ces dernières est ensuite très simple :

```
BRDFMaterial* resultMaterial =  new BRDFMaterial(filename + "Material", "BRDFMaterial");

//création des uniforms avec les paramètres
for (auto it = vec4_parameters.begin(); it != vec4_parameters.end(); ++it)
    resultMaterial->setUniformVec4(GL_FRAGMENT_SHADER, it->first, it->second);
for (auto it = float_parameters.begin(); it != float_parameters.end(); ++it)
    resultMaterial->setUniformFloat(GL_FRAGMENT_SHADER, it->first, it->second);
for (auto it = vec3_parameters.begin(); it != vec3_parameters.end(); ++it)
    resultMaterial->setUniformVec3(GL_VERTEX_SHADER, it->first, it->second);
for (auto it = bool_parameters.begin(); it != bool_parameters.end(); ++it)
    resultMaterial->setUniformBool(GL_FRAGMENT_SHADER, it->first, it->second);
```

FIGURE 10 – image exemple d'appel de fonction de création d'uniform

Cet exemple fait suite à celui présenté précédemment (cf. Figure 7), dans lequel le matériau possède un paramètre de type float, « specular_pow » avec une valeur de $20f$ qui sera utilisée dans le fragment shader. Pour pouvoir la créer au sein de l'instance de BRDFMaterial, il suffit donc d'appeler la fonction 'setUniformFloat()', avec comme paramètres le nom de la variable uniform, sa valeur ainsi que l'enumType du shader (définis dans OpenGL) :

- GL_VERTEX_SHADER
- GL_TESSEL_CONTROL_SHADER
- GL_TESS_EVALUATION_SHADER
- GL_GEOMETRY_SHADER
- GL_FRAGMENT_SHADER

De la même manière, si l'on souhaite récupérer la valeur de cette variable uniform, il suffira d'appeler la fonction getUniformFloat(). Ces fonctions vont interagir avec le GLUniformManager qui s'occupe de créer les emplacements mémoires dans le pipeline OpenGL et de les attribuer aux bons GLPrograms.

2.1.3 Construction des shaders

La classe BRDFMaterial, pour pouvoir fonctionner, a besoin d'avoir des shaders qui puissent s'adapter en fonction des paramètres fournis par le fichier JSON. On peut alors s'imaginer avoir un shader qui gèrerait lui même tous les cas possibles de paramètres, mais cela ne serait pas rapide en terme d'exécution et rendrait très complexe son support à long terme pour quelqu'un qui n'aurait pas travaillé à sa conception. Nous avons donc décidé de construire « à la volée » les shaders lors de l'import du matériau. Un problème s'est cependant posé car, en l'état actuel de GobLim, il nous est impossible de pouvoir envoyer directement le code source des shaders depuis le host au pipeline d'OpenGL sans devoir modifier bon nombre de classes au coeur du moteur. Nous avons donc décidé que suite à la construction des shaders, au lieu d'essayer de les envoyer au pipeline, nous les écrivons dans les fichiers Main-*.glsl de la classe BRDFMaterial. Ainsi, à l'appel du constructeur, les shaders seront gérés comme si ils avaient toujours été définis et le moteur n'aura aucun problème pour les charger en mémoire et les utiliser dans le pipeline durant le rendu. Cette tâche est donc gérée par deux classes : MaterialLoader et TemplateShader.

Tout d'abord, nous avons créé des fichiers « template » de shaders (situés à l'emplacement `/Materials/Common/BRDF/templates/`), dans lesquels nous avons écrit le code source des shaders qui ne changerait jamais, peut importe le matériau utilisé :

```
1   #version 430
2
3   #extension GL_ARB_shading_language_include : enable
4   #include "/Materials/Common/Lighting/Lighting"
5
6   ***INCLUDES***
7
8   layout(std140) uniform CPU
9   {
10      ***UNIFORMS***
11  };
12
13
14  in vec3 V;
15  in vec3 N;
16  in vec3 pointPosition;
17  in vec3 pos;
18  ***IN-OUT***
19
20  layout (location = 0) out vec4 Color;
21  ***LAYOUT***
22
23
24  void main()
25  {
26      ***MAIN-CODE***
27  }
```

FIGURE 11 – Template du fragment shader

Le but est de devoir créer et ajouter uniquement les « snippets » ou morceaux de code source GLSL qui sont susceptibles de changer en fonction du matériau. Ainsi, les emplacements des morceaux de shader manquants à compléter seront marqués par des « flags »,au nombre de cinq pour l'instant :

- ***INCLUDES*** : On ajoutera ici les #includes nécéssaires pour appeler du code externe au shader
- ***UNIFORMS*** : Pour ajouter le nom et le type des variables uniforms qui seront utilisées (ces noms seront les mêmes que ceux passés avec les fonctions setUniform et getUniform de la classe BRDFMaterial)
- ***IN-OUT*** : A la même utilité que pour le flag ***UNIFORMS***, mais ici c'est pour les variables avec le préfixe in / out qui servent à communiquer entre les différents shaders du pipeline d'OpenGL.
- ***LAYOUT*** : Ce flag correspond au snippet de code chargé d'appeler les variables interpolées par OpenGL (la position du vertex, sa normale, coordonnées de textures, etc...)
- ***MAIN-CODE*** : Ce dernier snippet correspond au code dans la fonction main d'un shader, c'est le coeur de celui ci.

Pour éviter d'avoir des conflits avec la classe ProgramSourceManager de GobLim, qui importe tous les shaders ayant l'extension « .glsl » dans l'arborescence du projet, nous avons renommé les fichiers des templates avec l'extension « .tglsl », afin qu'ils soient ignorés car dans tous les cas ils ne peuvent pas être compilés par OpenGL et être utilisés directement.

La classe TemplateShader sert d'interface pour cette étape, et va permettre d'importer un template de shader, remplacer les flags de ce dernier par les snippets GLSL et enfin va aller écrire le shader complété dans un fichier à l'endroit que l'on souhaite (dans notre cas, à la place des fichiers Main-*.glsl de BRDFMaterial).

```glsl
#version 430

#extension GL_ARB_shading_language_include : enable
#include "/Materials/Common/Lighting/Lighting"

#include "/Materials/Common/BRDF/diffuse/OrenNayar"
#include "/Materials/Common/BRDF/reflectance/Trowbridge"


layout(std140) uniform CPU
{
    vec4 color;
    float kd;
    float ks;
    float rho;
    float sigma;
    float c;
    bool normalized;
};


in vec3 V;
in vec3 N;
in vec3 pointPosition;
in vec3 pos;


layout (location = 0) out vec4 Color;



void main()
{
    Color = vec4(0.0, 0.0, 0.0, 1.0);
 for(int i = 0 ; i < nbLights.x; ++i){
    vec3 L = normalize(Lights[i].pos.xyz - pointPosition);
    float diffuse_value = BRDFOrenNayar(N,V,L,rho,sigma);
    vec4 diffuse_composante = color * max(0, dot(N,L)) * diffuse_value * kd;
    float specular_value = BRDFTrowbridge(L, V, N, c, normalized);
    vec4 specular_composante = color * specular_value * ks;
    Color += specular_composante + diffuse_composante;
    }

}
```

FIGURE 12 – Exemple du fragment shader généré après remplacement des flags

La classe MaterialLoader va être la classe principale qui orchestre l'import d'un fichier JSON, la création et l'écriture des shaders à la volée et la construction de l'instance BRDFMaterial associée, avec la création des variables uniforms passées depuis le code host :

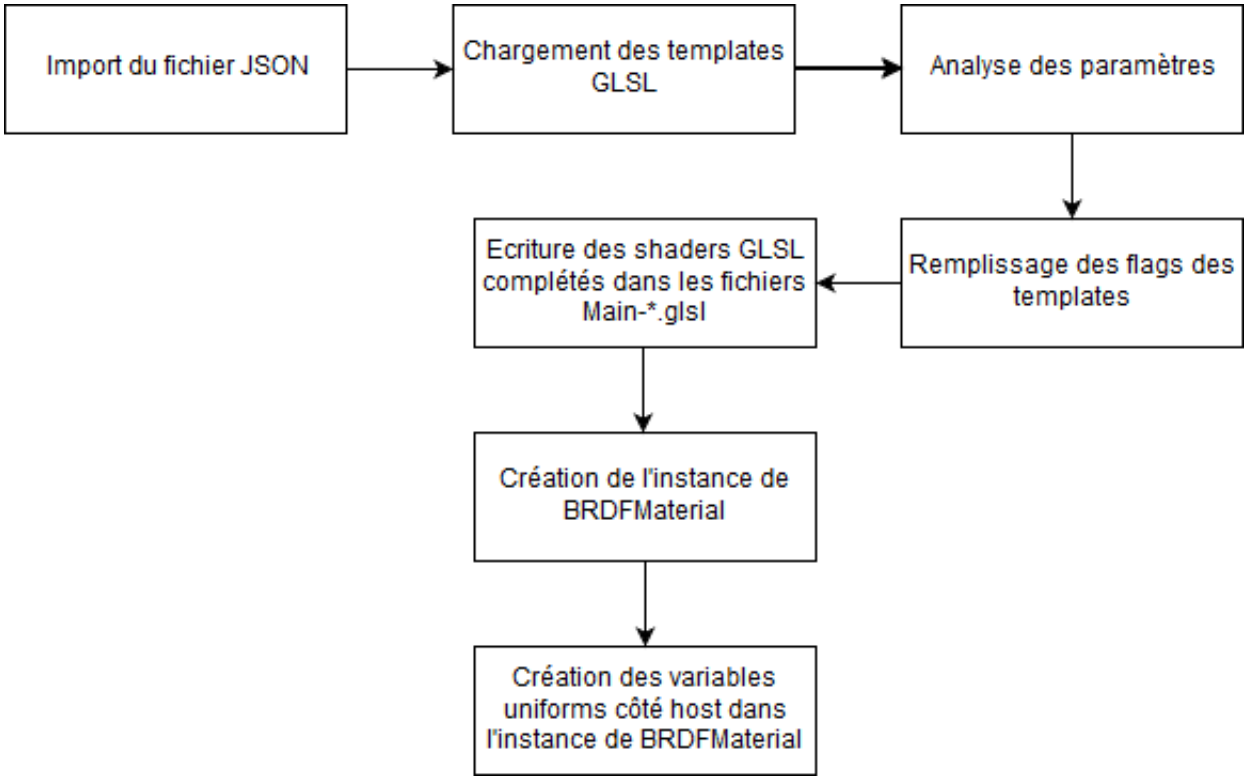


FIGURE 13 – Schéma du déroulement au sein du MaterialLoader

Nous avons, par la suite, repris le code des modèles diffus et spéculaires depuis le BRDF Explorer dans des fichiers GLSL (à l'emplacement `/Materials/Common/BRDF/reflectance/`, `Materials/Common/BRDF/diffuse/` et `Materials/Common/BRDF/other/`). Ainsi, selon le modèle d'éclairage choisi dans les paramètres du matériau, ce dernier sera importé via un include dans le shader et la fonction sera appelée dans le code de la fonction main.

```
1   // [type] [name] [min val] [max val] [default val]
2   //  float n; // 1  1000  100
3   //  float k; // 0.1  10  1
4   float BRDFNishino( vec3 L, vec3 V, vec3 N, float n, float k)
5   {
6       vec3 H = normalize(L+V);
7       float NdotH = max(0,dot(N,H));
8       float epd = 1-exp(-k * pow(NdotH,n));
9
10      return epd;
11  }
```

FIGURE 14 – Snippet en GLSL pour le modèle de Nishino

```
//Spéculaire de Nishino
else if (specular["name"] == "nishino") {

    fs_snippets["includes"] += "#include \"/Materials/Common/BRDF/reflectance/Nishino\"\n";

    ifstream input_stream(ressourceCoreMaterialPath + "Common/BRDF/reflectance/Nishino_default.json");
    input_stream >> default_config;
```

FIGURE 15 – Exemple d'utilisation de Nishino dans le MaterialLoader

Également avec les différents modèles, nous avons créé des fichiers de configuration par défaut pour chacun d'entre eux. Dans le cas où un matériau importé au format JSON utilise un modèle d'éclairage, mais oublie de définir un des paramètres nécéssaires à son fonctionnement, la classe MaterialLoader va aller chercher dans ce fichier une valeur donnée par défaut qui viendra se substituer à cet oubli. Cela permet d'une part, que la définition d'un matériau dans un fichier soit plus souple pour l'utilisateur, qu'il n'ait pas à devoir énumérer tous les paramètres si il ne souhaite qu'utiliser des paramètres par défaut. D'autre part, qu'il soit possible de modifier la valeur des paramètres par défaut sans avoir à retoucher au code source, car ces valeurs seront stockées en JSON et seront chargées à la volée quand le MaterialLoader en aura besoin, mais aussi pour faciliter leur modification si nécessaire. Ces fichiers de paramétrage sont stockés aux mêmes endroits dans GobLim que les modèles d'éclairage auxquels ils sont associés.

```
{
    "n" : 100,
    "k" : 1
}
```

FIGURE 16 – Paramètres par défaut pour le modèle de Nishino

```
ifstream input_stream(ressourceCoreMaterialPath + "Common/BRDF/reflectance/Nishino_default.json");
input_stream >> default_config;

//Récupération des paramètres de Nishino
float k;
float n;
if (specular.find("parameters") != specular.end()) {
    spec_param = specular["parameters"];
    if (spec_param.find("k") != spec_param.end())
        k = spec_param["k"].get<float>();
    else
        k = default_config["k"].get<float>();
```

FIGURE 17 – Gestion d'un paramètre par défaut pour Nishino

Les classes que nous avons élaborées et implémentées pour cette fonctionnalité de GobLim se présentent ainsi :

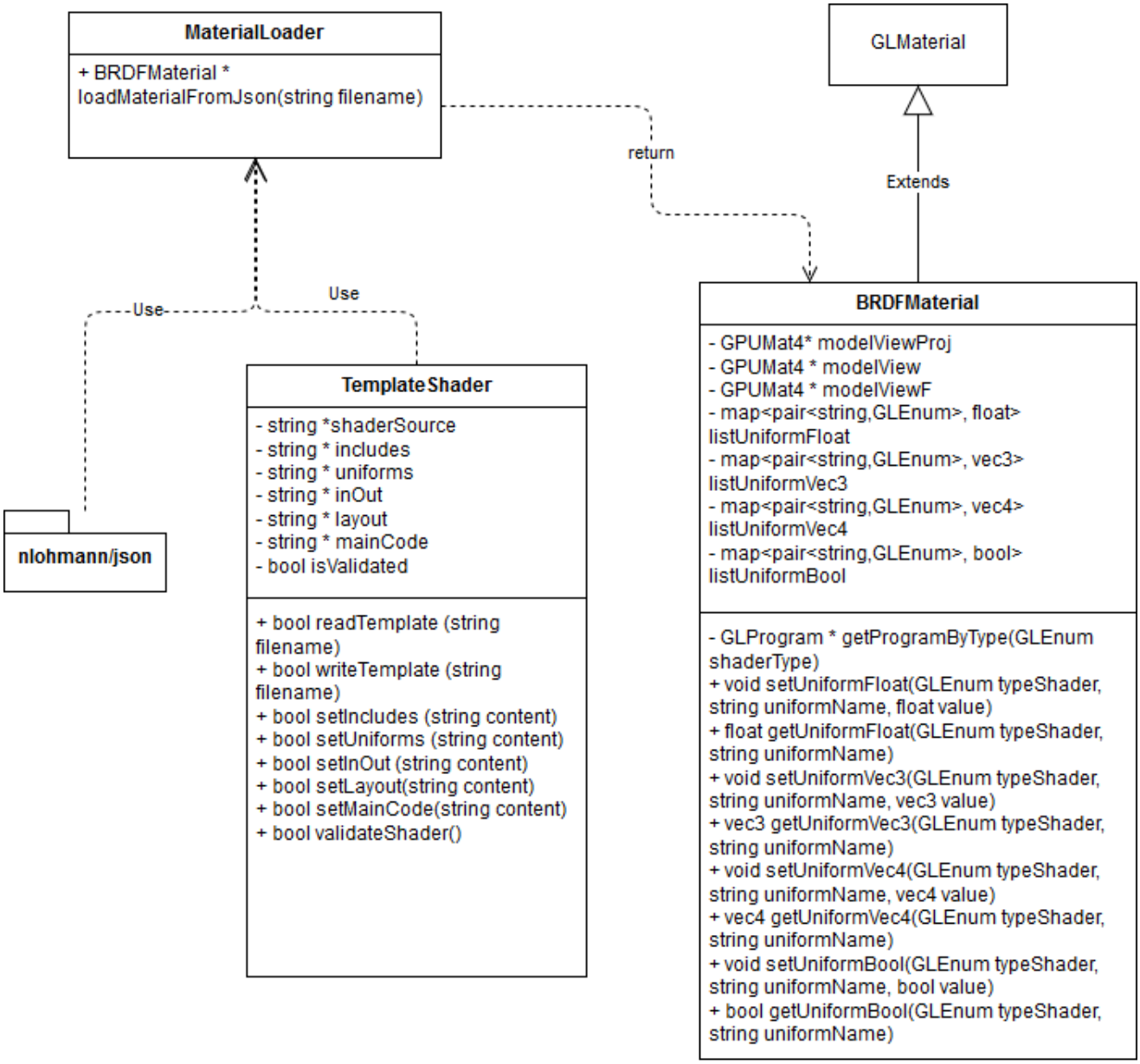


FIGURE 18 – Diagramme de classes

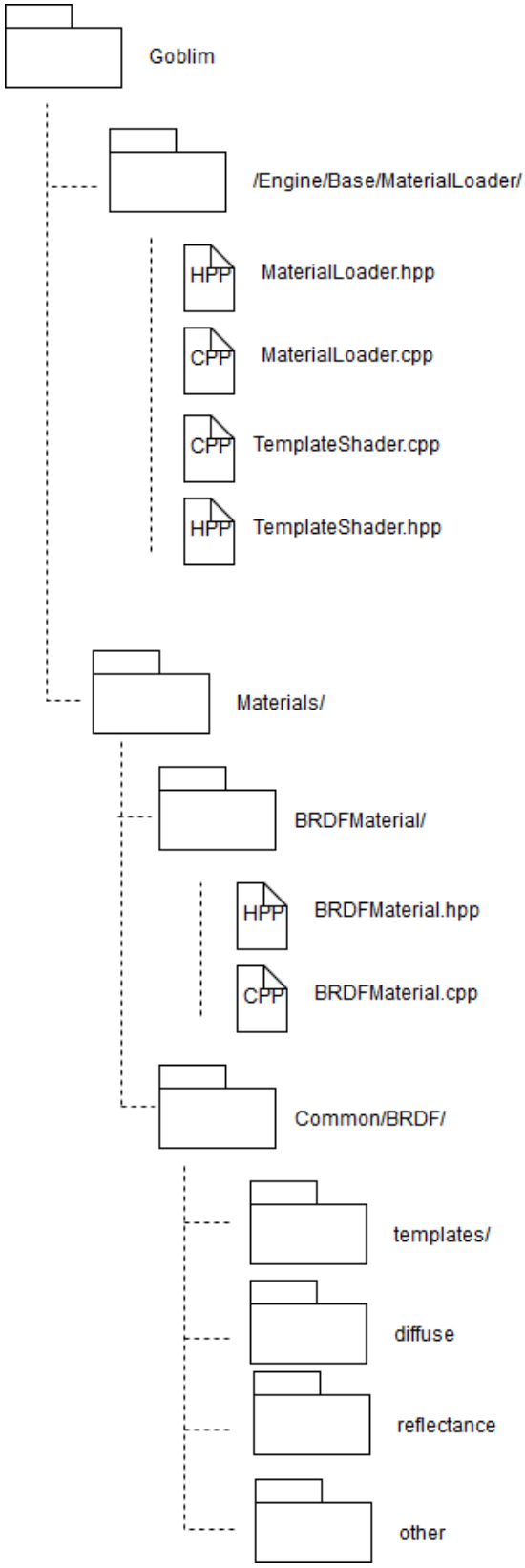


FIGURE 19 – Arborescence des fichiers du projet

2.1.4 Utilisation de Lighting.glsl

Afin de permettre une facilité d'utilisation pour les utilisateurs, nous avons également ajouté l'inclusion du fichier Lighting.glsl dans les shaders générés par la classe MaterialLoader. De base ce fichier comporte plusieurs fonctions et variables utiles qui sont utilisées dans les matériaux déjà présents dans GobLim. Parmi les variables présentes se trouvent deux variables qui nous intéressent. La première contient la liste des différentes lumières de la scène tandis que la deuxième correspond à la position de la caméra dans la scène.

Ainsi, en incluant, ce fichier nous disposons des positions des lumières et de la caméra dont nous avons besoin pour la totalité des matériaux qui seront générés à l'aide du MaterialLoader. De plus cela nous permet d'accéder directement à ces paramètres sans avoir à l'indiquer dans la classe BRDFMaterial, puisque le moteur le fait déjà à partir de la classe LightingModel.

2.2 Interface

L'interface du moteur est générée à l'aide d'ImGui. C'est une bibliothèque qui est disponible dans plusieurs langages (C++, Python, D, Lua, etc ...) et qui est compatible avec différents frameworks et API comme OpenGL, Vulkan ou encore DirectX.

Afin de faciliter les tests pour trouver les bonnes valeurs aux paramètres des matériaux, le moteur permettait à partir d'une interface de pouvoir modifier ces valeurs sans avoir à relancer le moteur. Pour cela il fallait spécifier, dans le code de la fonction displayInterface de la classe d'un matériau, les paramètres que l'on voulait pouvoir modifier.

Cependant, notre classe de matériau étant différente des autres classes, nous avons du nous y prendre différemment. En effet dans les classes déjà présentes, l'utilisateur indiquait dans le code du matériau le ou les paramètres qu'il voulait modifier. Or, dans la classe BRDFMaterial nous ne pouvions faire cela puisque les paramètres dépendaient du matériau que nous chargions.

Nous avons donc réfléchi et nous avons ajouté des variables et des méthodes au sein de BRDFMaterial pour permettre la modification de ces variables depuis l'interface. Ainsi, lorsque nous affectons une nouvelle variable uniform depuis les méthodes setUniformT (T étant le type de la variable), nous ajoutons également le nom, le type du shader, et la valeur dans une liste. Pour chaque type de variable, nous avons une liste qui stocke ces variables.

Grâce à ces listes nous pouvons ensuite générer les champs nécessaires dans l'interface afin de pouvoir modifier les valeurs depuis la fonction displayInterface. Toutefois, les modifications qui sont faites depuis l'interface sont réalisées sur la variable présente dans la liste, donc la variable n'est pas mise à jour dans le code exécuté par la carte graphique. Pour résoudre cela il a fallu ajouter, dans la fonction update de BRDFMaterial, une boucle pour chaque liste afin de faire la mise à jour des variables uniforms sur la carte graphique.

Pour ce projet nous avons surtout utilisé les fonctions d'ImGui permettant d'afficher des champs afin de modifier la valeur des variables. Par exemple en utilisant la méthode *InputFloat(string name, float *value)*, nous pouvons créer un champ dans lequel nous modifions la valeur de la variable *value* passée en paramètre.

Voici la liste des fonctions qui ont été utilisées :

- *InputFloat(string name, float *value)*
- *InputFloat3(string name, float *value)* : créer un champ permettant de modifier un tableau de flottants de taille 3
- *InputFloat4(string name, float *value)* : créer un champ permettant de modifier un tableau de flottants de taille 4
- *Checkbox(string name, bool *value)* : créer une checkbox permettant de modifier la valeur d'un booléen

Une fois les champs créés à l'aide d'ImGui, nous devons mettre à jour les valeurs à l'intérieur des listes. Pour cela il suffit juste d'utiliser la variable renseignée lors de l'appel à la fonction de création du champ.

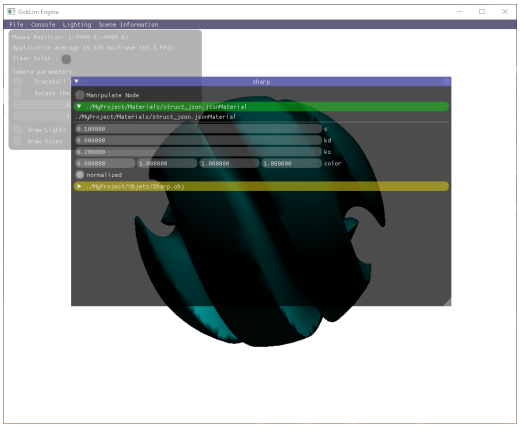


FIGURE 20 – Exemple avec les modèles de Lambert et Trowbridge

2.3 Améliorations possibles

La manière dont le parser de matériaux a été conçu permet de modifier assez facilement le code afin de pouvoir réaliser certaines améliorations. Pour cela, il suffit simplement d'ajouter le code nécessaire au sein du MaterialLoader.

La première amélioration à laquelle on peut penser est la prise en charge de nouvelles méthodes de rendu. Pour cela, il suffit simplement de rajouter un fichier .glsl et d'ajouter une condition dans le MaterialLoader pour traiter ce nouveau matériau.
Le fichier .glsl devra contenir le code correspondant au matériau en mettant en place la méthode qui est appelée pour n'importe quel matériau conçu à partir du MaterialLoader, afin de connaître l'éclairage d'un point d'un objet.
Au sein du MaterialLoader il faudra, comme dit précédemment, ajouter une condition pour le matériau dans la méthode de construction. Dans cette condition, il faudra préciser les paramètres que l'on souhaite pouvoir récupérer dans le fichier .json qui contiendra les valeurs des paramètres pour ce matériau.

Une autre des améliorations du MaterialLoader à prévoir est le support de textures depuis un fichier. Actuellement le parser de matériaux ne traite aucun type de texture, que ce soit des textures qui doivent être simplement affichées ou des textures qui réalisent des opérations comme des déformations sur un objet, ou encore des textures qui apportent des informations complémentaires sur un objet comme le fait le normal mapping par exemple.
Ainsi, en développant cette nouvelle fonctionnalité, le MaterialLoader pourra servir à réaliser des éclairages sur des objets ayant une texture, mais également pour d'autres méthodes de rendu comme le multi-texturing qui permet de gérer plusieurs effets comme

les éclairages ou des ombres en utilisant seulement des textures.
Pour réaliser cette amélioration il suffirait de mettre à l'intérieur du fichier .json le chemin vers la texture que l'on souhaite utiliser. Pour chaque type de texture, on aurait un champ différent (Texture, NormalMap, etc ...) afin de pouvoir les différencier.
Dans le MaterialLoader il suffirait simplement de vérifier si, dans le fichier .json que l'on utilise, les différents champs correspondants aux différents types de textures sont présents avec une valeur associée. Si c'est le cas il suffit d'ajouter l'image à la liste des paramètres du matériau.

En modifiant le MaterialLoader et en ajoutant des variables dans le fichier .json décrivant un matériau, on pourrait aussi permettre l'utilisation de fichiers de statistiques pour le rendu de matériaux plus complexes, comme l'utilisent certaines méthodes d'éclairage de micro-facettes ou encore certains modèles analytiques comme Cook-Torrance.

Actuellement, lorsque l'on crée un nouveau matériau en utilisant MaterialLoader, les fichiers .glsl qui sont créés pour ce nouveau matériau suppriment les fichiers du précédent matériau généré à l'aide de MaterialLoader. Cette suppression est due au fait que les fichiers créés possèdent tous le même nom.
Une amélioration qui pourrait être intéressante à réaliser serait de pouvoir garder les fichiers de certains matériaux générés. Cette amélioration permettrait ainsi de garder une trace des matériaux, mais également de ne pas avoir à générer à chaque fois les shaders d'un matériau si ils sont déjà créés et si ils ne diffèrent pas des fichiers précédemment conçus.
Pour cela on pourrait définir une variable dans le fichier .json contenant le nom qui servira de base aux fichiers .glsl générés. Cette amélioration serait assez rapide et facile à réaliser, puisque pour une même méthode d'éclairage les shaders générés diffèrent rarement étant donné que seules les valeurs des variables changent. Et puisque les valeurs de ces variables sont passées par le programme à la carte graphique en tant que variables et non pas en tant que données brutes écrites dans le fichier du shader nous n'avons pas besoin de les gérer dans cette adaptation.

Actuellement, lorsque l'on regarde les paramètres que l'on peut modifier pour un matériau généré à l'aide du MaterialLoader, on trouve tous les paramètres qui sont passés en tant que variables uniforms. Il pourrait être intéressant pour l'utilisateur de pouvoir choisir quelles variables peuvent être modifiées. Pour indiquer cela on pourrait avoir un champ dans le fichier .json. Ce champ serait une liste qui contiendrait le nom des paramètres que l'on souhaite pouvoir modifier depuis l'interface.

2.4 Travaux parallèles

Durant l'implémentation, nous avons été confronté à plusieurs problèmes, notamment la façon dont étaient gérés l'import et la création des shaders par GobLim au sein du pipeline OpenGL. Pour pouvoir comprendre comme cela fonctionnait, il nous a fallu aller regarder

le code source responsable de cette fonctionnalité, car aucune documentation n'est disponible ou n'a été réalisée pour le moteur, mais également comprendre le fonctionnement bas niveau des shaders dans OpenGL à l'aide de divers supports et documents fournis par Khronos, qui a rédigé les spécifications de cette API. C'est ainsi que nous avons, pendant cette période, décidé de rédiger une partie de la documentation pour le moteur en plus de celle des classes que nous avons implémentées pour le projet. A l'heure actuelle, quelques classes ont été complètement documentées, ainsi que leur code source qui a été refactorisé afin d'améliorer sa lisibilité :

— GLProgram
— GLProgramSource
— MaterialGL

Il est à noter qu'un autre étudiant, Simon Courtin, faisant parti d'un groupe travaillant également sur le moteur, a aussi rédigé la documentation de quelques classes dans GobLim.

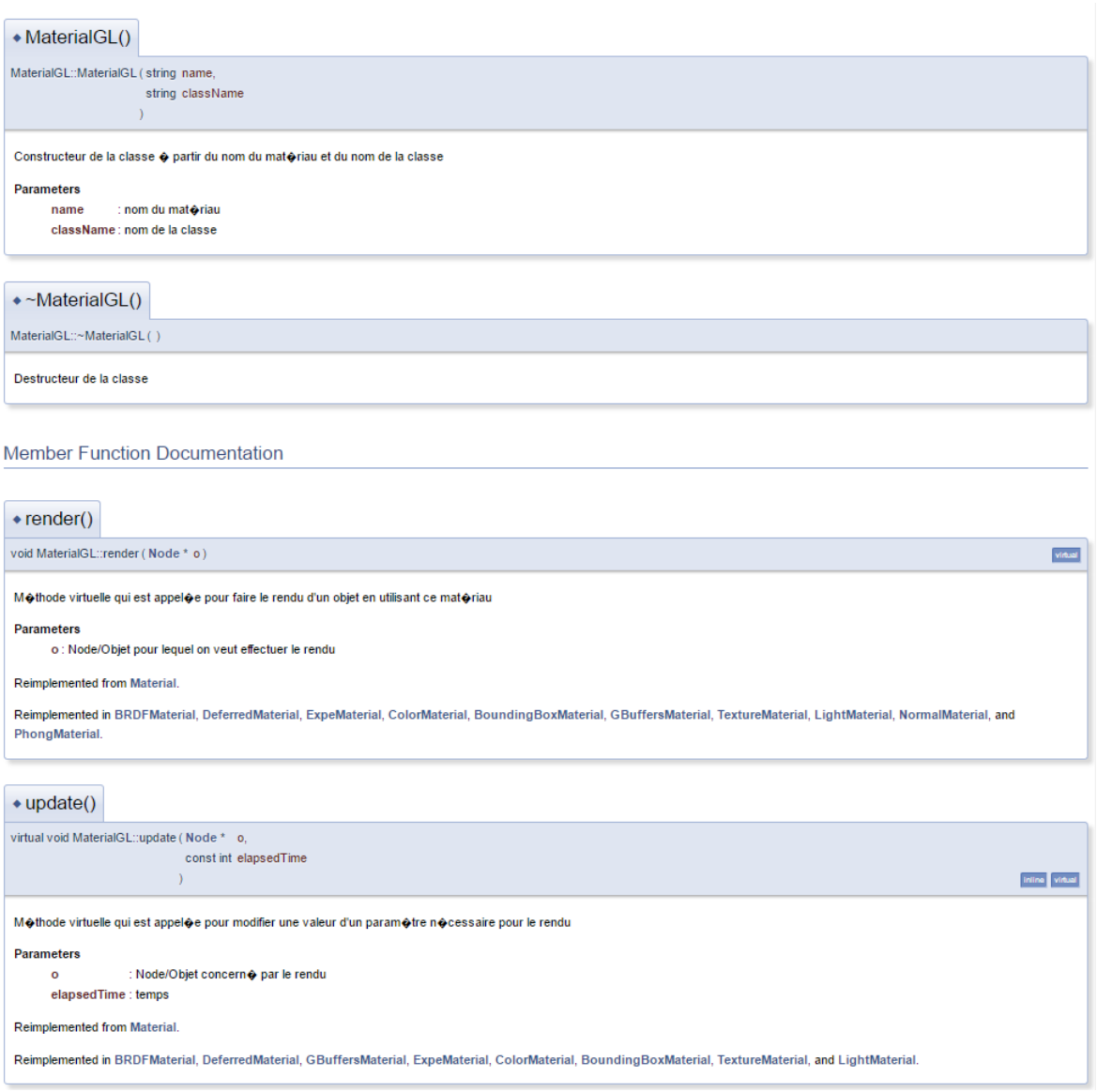
Le code générant la documentation depuis un fichier source se présente de cette manière :

MaterialGL.h (D:\Uni\Project\GobLim\Core\Engine\OpenGL) - GVIM

```
class MaterialGL : public Material
{
    public :

        //Attributs

        GLProgramPipeline* m_ProgramPipeline;

        //Constructeur-Destructeur

        /**
            Constructeur de la classe à partir du nom du matériau et du nom de la classe
            @param name : nom du matériau
            @param className : nom de la classe
        */
        MaterialGL(string name,string className);

        /**
            Destructeur de la classe
        */
        ~MaterialGL();

        //Méthodes

        /**
            Méthode virtuelle qui est appelée pour faire le rendu d'un objet en utilisant ce matériau
            @param o : Node/Objet pour lequel on veut effectuer le rendu
        */
        virtual void render(Node *o);

        /**
            Méthode virtuelle qui est appelée pour modifier une valeur d'un paramètre nécessaire pour le rendu
            @param o : Node/Objet concerné par le rendu
            @param elapsedTime : temps
        */
        virtual void update(Node *o, const int elapsedTime){}


    protected :

        //Attributs

        // Vertex / Geometry / Tessel Control / Tessel Eval / Fragment
        GLProgram *vp, *gp, *tcp, *tep, *fp;
};
```

FIGURE 21 – Code source documenté (dans la classe MaterialGL)

Cette syntaxe n'est pas celle du code documenté C++, mais est plus proche de la javadoc. On peut par exemple, dans la documentation d'une fonction, décrire les attributs au moyen de l'entête de ligne « @param », ou pour le résultat renvoyé avec l'entête « @return ». Nous nous servons ensuite de l'outil Doxygen, qui avec un fichier de configuration que nous avons ajouté au projet, va aller lire tous les fichiers sources du moteur et regarder dans chacun si du code, tel que celui présenté ci-dessus, est présent. Il génère ainsi une documentation au format HTML mais également en Latex.



◆ MaterialGL()

MaterialGL::MaterialGL (string name,
string className
)

Constructeur de la classe à partir du nom du matériau et du nom de la classe

Parameters
name : nom du matériau
className : nom de la classe

◆ ~MaterialGL()

MaterialGL::~MaterialGL ( )

Destructeur de la classe

Member Function Documentation

◆ render()

void MaterialGL::render (Node * o) virtual

Méthode virtuelle qui est appelée pour faire le rendu d'un objet en utilisant ce matériau

Parameters
o : Node/Objet pour lequel on veut effectuer le rendu

Reimplemented from Material.

Reimplemented in BRDFMaterial, DeferredMaterial, ExpeMaterial, ColorMaterial, BoundingBoxMaterial, GBuffersMaterial, TextureMaterial, LightMaterial, NormalMaterial, and PhongMaterial.

◆ update()

virtual void MaterialGL::update (Node * o,
const int elapsedTime
) inline virtual

Méthode virtuelle qui est appelée pour modifier une valeur d'un paramètre nécessaire pour le rendu

Parameters
o : Node/Objet concerné par le rendu
elapsedTime : temps

Reimplemented from Material.

Reimplemented in BRDFMaterial, DeferredMaterial, GBuffersMaterial, ExpeMaterial, ColorMaterial, BoundingBoxMaterial, TextureMaterial, and LightMaterial.

FIGURE 22 – Documentation générée (depuis la classe MaterialGL)

3 Résultats et rendus

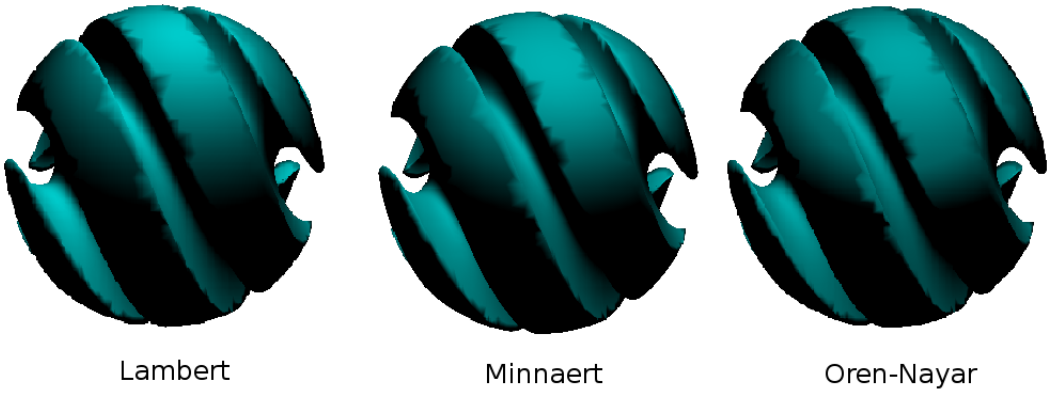


FIGURE 23 – Rendu des méthodes diffuses

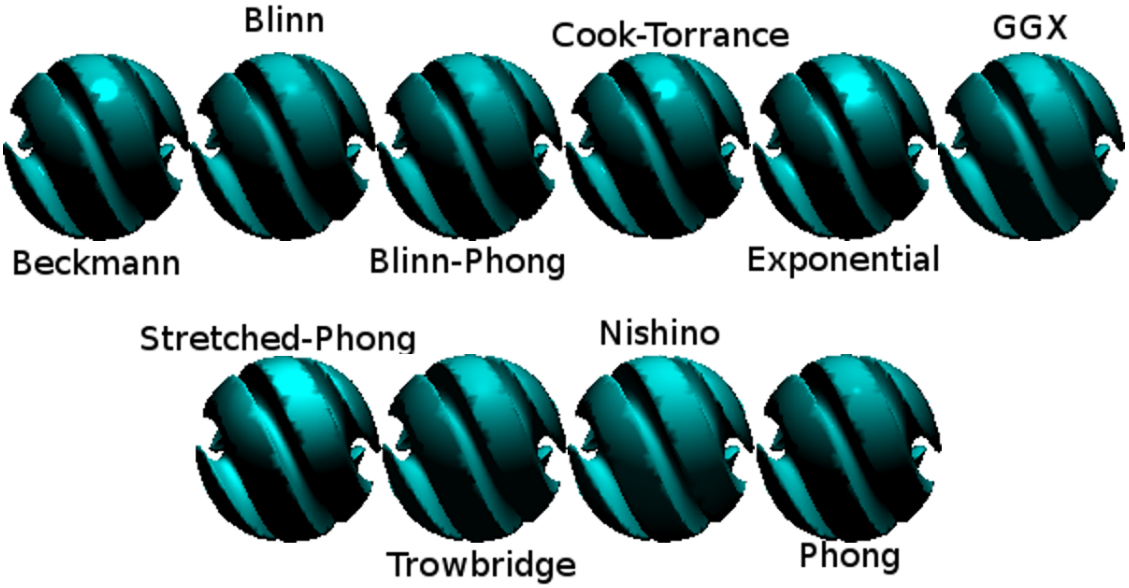


FIGURE 24 – Rendu des méthodes spéculaires avec en diffus un Lambert

FIGURE 25 – Rendu de méthodes diffuses et spéculaires

4 Problèmes et difficultés rencontrés

Durant ce projet, nous avons été confrontés à certaines difficultés et à certains problèmes. Cette partie est là pour présenter les différents problèmes et difficultés rencontrés sur le projet, en présentant également la manière dont nous avons réagi pour les résoudre.

Le principal problème que nous avons eu pour ce problème est lié à l'utilisation de la MDL, problème dont nous avons parlé dans une partie en début de ce rapport. Ce problème nous a fait perdre un peu de temps sur la partie développement du projet puisque nous avons mis du temps avant de changer le sujet du projet, du fait des problèmes rencontrés avec l'API MDL mentionnés plus tôt.

Notre projet étant de permettre la construction d'un matériau en se servant des indications d'un fichier, nous devions créer des variables contenant les codes .glsl du matériau souhaité afin de pouvoir ensuite les assembler et les traiter différemment.
Nous avons donc regarder si il était possible, sur Goblim, de fournir directement du code .glsl sans avoir à passer par un fichier déjà existant. Les matériaux déjà présents sur Goblim utilisaient un code déjà existant dans un fichier et le moteur chargeait ce code puis le traiter. Nous n'avions donc pas d'exemple de matériaux fonctionnant sans fichier contenant de code .glsl et nous avons donc dû regarder comment fonctionnait le passage des shaders à OpenGL dans Goblim.

Nous avons donc commencé par regarder comment se faisait la construction d'un matériau dans Goblim ainsi que le chargement de shaders en utilisant les fonctions d'OpenGL. Pour la construction d'un matériau on devait passer par une classe qui héritait de MaterialGL, qui elle même héritait d'une autre classe : Material. À la création du matériau, le programme crée une instance de GLProgramPipeline, une classe qui contient des instances de GLProgram, GLProgram qui représente un code .glsl. Cette classe permet de savoir de quel type de code il s'agit (FragmentShader, VertexShader, GeometryShader, etc ...) et a également une instance de GLProgramSource. Cette dernière classe, quant à elle, contient le code en .glsl dans une variable. Ainsi une instance de GLProgramSource permettait de gérer les différents types de shaders d'un matériau.

À ce stade là, il semblait plutôt facile de créer un shader sans utiliser de fichiers. Il nous fallait juste créer des GLProgramSource que nous pourrions remplir à l'aide des shaders formés. Cependant, en approfondissant un peu plus notre étude du code de Goblim, nous nous sommes aperçu que les shaders étaient envoyés à OpenGL dès la création du matériau en utilisant les fichiers .glsl du matériau. Donc, une fois le matériau créé, on ne pouvait plus modifier le code des shaders, puisque même en modifiant les codes ces derniers ne pouvaient pas être mis à jour sur la carte graphique.

La solution que nous avons trouvé pour résoudre ce problème a été de créer les fichiers .glsl, une fois les codes reconstitués, et de faire passer ces fichiers au programme comme s'il s'agissait d'un matériau normal. En faisant cela, nous n'avions pas besoin de toucher au

code déjà présent dans Goblim et nous avons pu ainsi éviter de provoquer des problèmes de compatibilité au sein du moteur.
Actuellement cette méthode fonctionne ; cependant, dès qu'un matériau est créé, les fichiers qui avaient déjà été générés par la précédente instance d'un matériau utilisant notre parser sont supprimés.

Cette première difficulté nous avait permis de regarder le code de Goblim plus en détails, ainsi que de commenter le code qui nous semblait utile afin de mieux le comprendre pour notre projet. Nous avons également dû faire quelques recherches sur la documentation d'OpenGL afin de mieux comprendre les fonctions OpenGL qui étaient utilisées dans le code du moteur. Ces différentes fonctions OpenGL concernaient surtout le passage de shader pour des matériaux.

Une autre difficulté que nous avons rencontré est liée à la manière dont nous devions récupérer certains paramètres au sein des shaders, comme la position d'un point sur un objet, ou encore sa normale. Puisque nous avions utilisé Goblim au premier semestre nous avions commencé par récupérer ces paramètres de la même manière qu'au premier semestre : en récupérant directement les valeurs transmises par le moteur à la carte graphique et que l'on pouvait récupérer facilement grâce aux variables fourni par OpenGL. Cependant, la version de Goblim utilisée au premier semestre et la version pour ce projet étant différentes, nous avons remarqué rapidement que quelque chose n'allait pas avec notre code.

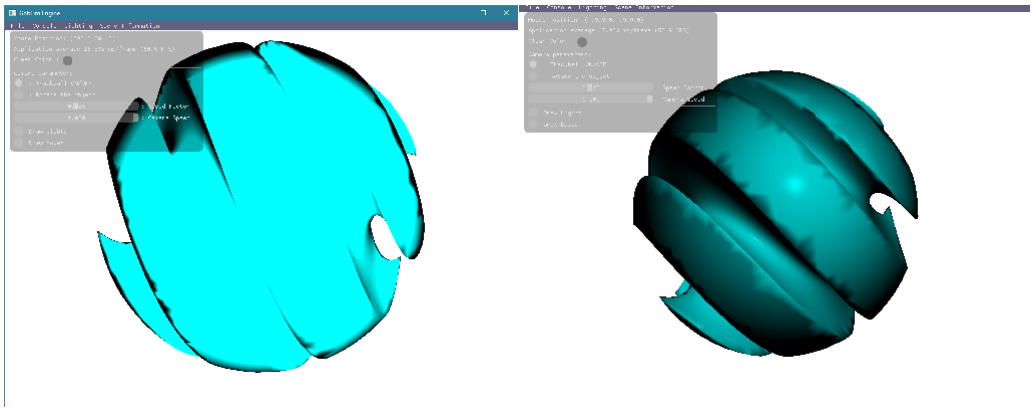


FIGURE 26 – Résultat inattendu d'un modèle de Phong

En regardant dans les shaders des matériaux déjà présents, nous avons donc pu voir ce qui n'allait pas et nous avons revu notre code.

5 Conclusion

Nos objectifs avec ce projet consistaient en trois points. Premièrement, nous voulions nous familiariser avec le langage MDL qui, du fait de son utilité apparente et des ressources de NVIDIA, pourrait s'imposer comme un format standard de définition de matériau. Deuxièmement, c'était l'occasion pour nous d'étudier différents modèles de BRDF, leur raison d'être et leur implémentation. Enfin, ce projet nous aurait permis de participer à l'amélioration du moteur GobLim.

Pour les raisons évoquées dans l'introduction, le premier objectif n'a pas pu être atteint. Nous nous sommes familiarisés avec le langage MDL, dans une certaine mesure, avec le travail du premier semestre, mais nous n'en avons qu'une idée générale qui n'aurait pu être poussée plus loin qu'en travaillant sur son implémentation. Les deux autres objectifs, cependant, ont pu être atteints.

Tout au long du projet, nous nous sommes rendus compte qu'il existait un grand nombre de BRDF différentes. Chacune a ses points forts et ses points faibles, mais toutes trouvent une utilité en cela qu'elles sont adaptées à différents types de matériaux ; le modèle de Blinn-Phong offre un contrôle sur le reflet spéculaire qui le rend pratique à utiliser pour des surfaces plastiques, alors que le modèle d'Oren-Nayar introduit un paramètre de « roughness » qui en fait un modèle adéquat pour une surface non lisse.

En ce qui concerne l'amélioration du moteur GobLim, le résultat obtenu se rapproche de ce qu'il était prévu de réaliser avec le langage MDL, dans une forme simplifiée. En effet, un fichier JSON contenant les informations du matériau, comme la couleur ou la BRDF utilisée et les paramètres de celle-ci, est chargé par notre classe MaterialLoader et automatiquement traduit en Material utilisable par GobLim.

Du point de vue de l'utilisateur, connaître le fonctionnement interne du moteur n'est ainsi plus nécessaire pour essayer différentes BRDF avec différents paramètres.

Du point de vue de l'implémenteur, l'ajout de nouvelles BRDF va passer par l'ajout d'un « snippet » GLSL l'implémentant dans le dossier `Core/Materials/Common/BRDF`, ainsi qu'un bloc « if » au fichier `Core/Engine/Base/MaterialLoader/MaterialLoader.cpp` — selon le modèle des BRDF déjà implémentées — pour définir comment utiliser les paramètres définis dans le fichier JSON.

En parallèle, nous avons travaillé sur la documentation, non seulement de notre code source, mais aussi sur quelques classes déjà présentes dans le moteur que nous avons été amenés à regarder de plus près lors de la phase de développement de ce projet, à savoir les classes `MaterialGL`, `GLProgramSource` et `GLProgram`.

Comme nous l'avons vu dans l'introduction, du fait de l'impossibilité d'utiliser le langage MDL, notre sujet de départ a été bousculé. Nous n'avons donc pas pu respecter notre diagramme de Gantt prévisionnel suivant :

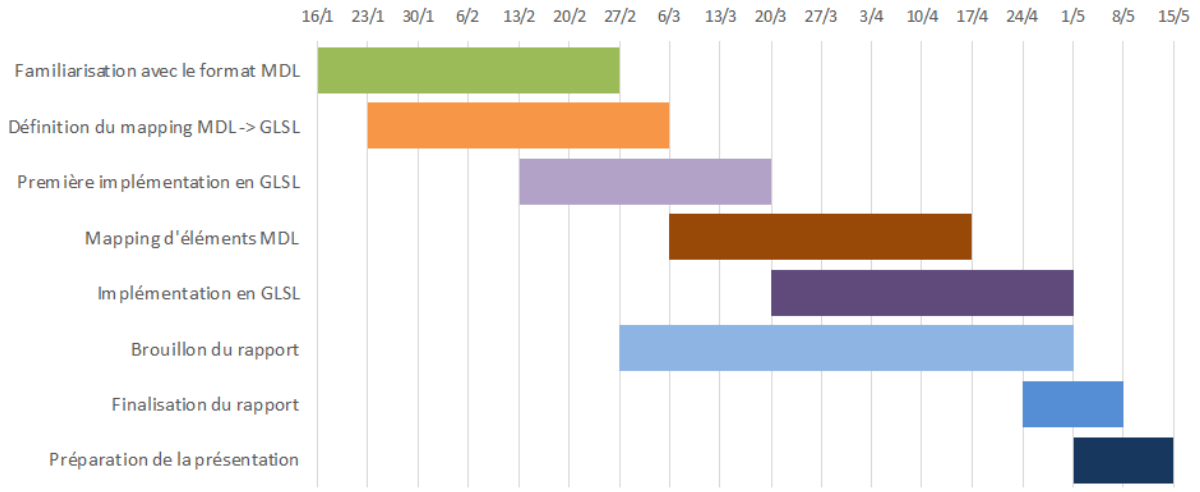


FIGURE 27 – Calendrier prévisionnel initial

Au final, notre travail a été organisé selon le diagramme suivant :

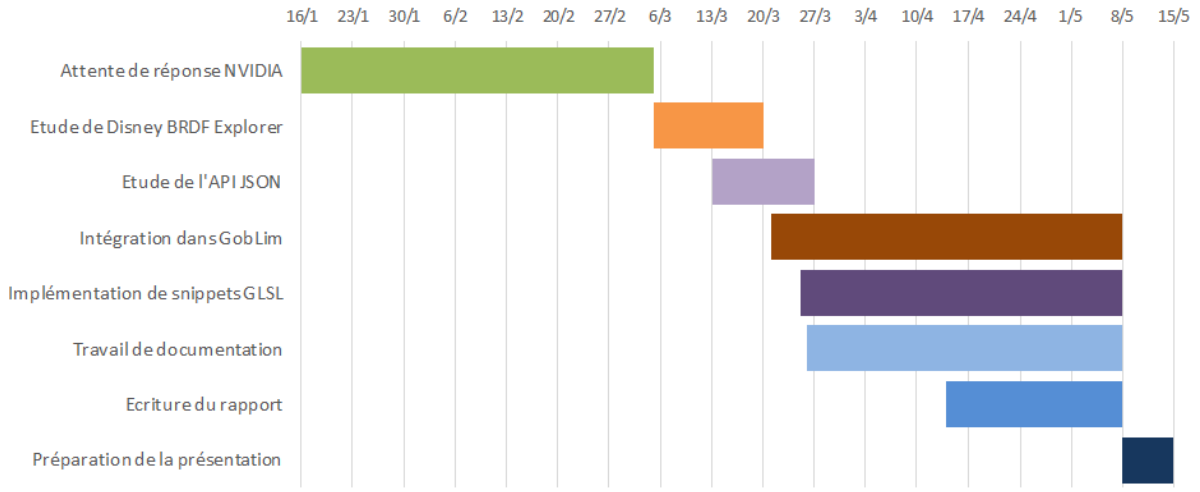


FIGURE 28 – Organisation effective

Références

[1] NVIDIA Material Definition Language, 2017. Page web de présentation.

[2] Substance Designer - Material Authoring Tool, 2017. Page web de présentation.

[3] Join the NVIDIA Developer Program, 2017. Page web de présentation de NVIDIA DesignWorks.

[4] NVIDIA Material Definition Language 1.3, 2016. Spécification du langage MDL.

[5] Extensible Markup Language (XML) 1.0 (Fifth Edition), 2013. Site web de présentation.

[6] Introducing JSON, 2017. Site web de présentation.

[7] Niels Lohmann. JSON for Modern C++, 2017. Documentation de l'API.

[8] BRDF Explorer, 2016. Site web de présentation.