Projet de fin d'études : Rapport final

BRAUD Gaelle - TRICOIRE Simon
TSIAPKOLIS Panagiotis - VESCHAMBRE Adrien
M2 IS

March 23, 2020

Contents

Introduction

Dans le cadre de la synthèse d'image, pour le cinéma d'animation par exemple, il peut-être intéressant de caractériser les comportements de la lumière de façon réaliste. Définir la réflexion d'une surface requiert une "*Bidirectional Reflectance Distribution Function*" (BRDF), qui définit comment la lumière est reflétée sur une surface opaque par le biais de quatre paramètres. Ces fonctions étant complexes à appréhender et manipuler, des solutions ont été développées afin de pouvoir visualiser et anticiper au mieux l'impact du choix d'une BRDF par rapport à une autre lors du rendu. On peut citer notamment BRDFLab ou encore BRDF Explorer développé initialement par Disney[7].

BRDF Explorer permet lors du rendu d'avoir recours à un "Image-based Ligthing" (IBL), qui consiste à éclairer un objet 3D à l'aide d'une carte d'environnement. Bien entendu, ce calcul d'illumination, bien trop coûteux à réaliser, est approximé à l'aide d'une intégration, au sens de Monte-Carlo, de la radiance émise par la carte d'environnement et reçue en chaque point de l'objet, le tout pondéré par la BRDF. La vitesse de convergence de ce calcul étant primordiale, elle est le plus souvent accélérée à l'aide de l'échantillonnage préférentiel ("*Importance Sampling*", abrégé IS), qui permet d'accélérer le temps de rendu tout en conservant une qualité de rendu équivalente.

Ainsi, il nous a été demandé par M.GUENNEBAUD et M.PACANOWSKI, dans le cadre de notre projet de fin d'étude, d'apporter des modifications au logiciel BRDF Explorer. Les modifications souhaitées consistent principalement à implémenter de nouvelles méthodes d'échantillonnage préférentiel, plus précisément:

- L'échantillonnage des cartes d'environnement par transport optimal à l'aide d'une stratégie de précalcul.
- L'échantillonnage des cartes d'environnement par transport optimal à l'aide d'une stratégie d'adaptation à la volée.
- L'échantillonnage de BRDF mesurée à l'aide de la méthode standard par CDF.

D'autres améliorations du logiciel de base sont prévues, comme le développement d'outils d'évaluation objective de la vitesse de convergence ainsi que des outils de visualisation du processus d'échantillonnage lui même.

Les termes du sujet

1.1 Définition BRDF

Une BRDF (*Bidirectional Reflectance Distribution Function*) décrit les propriétés de réflectance d'une surface en spécifiant la quantité de rayonnement incident d'une direction qui est réfléchie dans une autre direction, par rapport à la normale de la surface.

Si les rayons incidents et réfléchis sont symétriques, on dit que la BRDF respecte la réciprocité de Helmholtz. De même, elle respecte le principe de conservation de l'énergie si la puissance réfléchie totale est inférieure ou égale à l'énergie de la lumière incidente pour une direction donnée du rayon incident. Une BRDF respectant la réciprocité de Helmholtz et le principe de conservation de l'énergie est qualifiée de réaliste.

1.2 BRDF mesurées et base de données MERL

Un modèle de BRDF dit mesuré est obtenu par le biais d'un réflectogoniomètre[9] (Figure 1.1). La base de données MERL (Mitsubishi Electric Research Laboratories) élaborée en 2006 regroupe les fonctions de réflectance de 100 matériaux différents libres d'accès pour tout usage non lucratif[8].

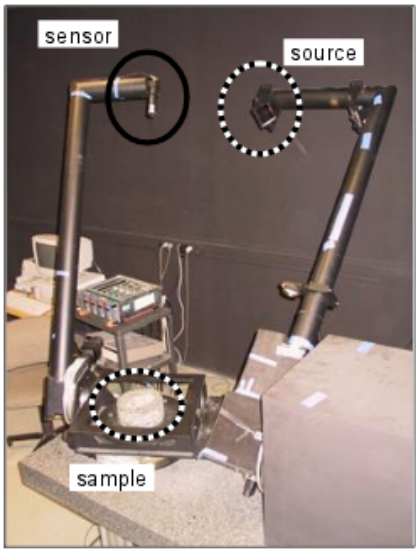

Figure 1.1: Un réflectogoniomètre à 3 angles[9]

1.3 Rendu 3D

Comme synthétisé dans le chapitre 20 du livre GPU Gems 3 publié par NVidia et accessible gratuitement en ligne[10], le rendu 3D consiste en l'évaluation de la lumière reflétée par une surface vers la caméra virtuelle pour chacun de ses pixels, en prenant en compte l'environnement 3D de la surface en question. Dans ce procédé, le rôle d'une **BRDF** est de convertir la lumière en provenance d'une direction donnée ($L_i(\mathbf{u})$) en lumière réfléchie dans la direction de la caméra. De façon pratique, pour calculer la réflexion totale de la lumière pour un pixel donné ($L_o(\mathbf{v})$), il faut faire la somme des contributions pour chaque direction incidente **u** dans l'hémisphère H. Ceci revient à en faire l'intégrale :

$$L_o(v) = \int_H L_i(u) f(u,v) cos\theta_u du$$

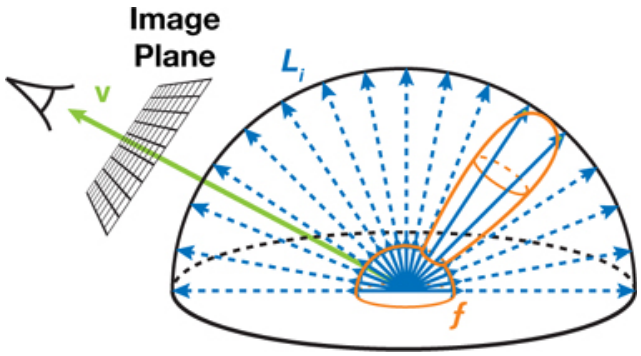

Figure 1.2: Illustration de la réflexion totale[10]

En *"Image Base Lighting"*, la lumière incidente L_i est approximée par une carte d'environnement. Cependant, même avec une approximation, le calcul de l'intégrale modélisant la réflexion totale reste trop coûteux en temps réel, puisque pour chaque pixel donné il faut accéder et sommer des milliers de texels de la carte d'environnement multipliés au préalable par la BRDF.

Du fait de cette limitation, il faudrait donc approximer cette intégrale. Pour ce faire, nous avons recours à l'intégration de Monte Carlo.

1.4 Intégration de Monte Carlo

L'origine de l'emploi du terme Monte Carlo est explicité dans le livre Advanced Global Illumination de Dutré, Bala et Bekaert comme suit: "Le terme "Monte Carlo" est apparu dans les années 1940 [...] pour décrire des techniques mathématiques qui utilisent l'échantillonnage statistique pour simuler des phénomènes ou évaluer des valeurs de fonctions."[12]. Il s'agit d'une référence au casino de Monte Carlo de Monaco, du fait des procédés aléatoires mit en jeu dans les méthodes de Monte Carlo[5]. Monte Carlo sert à approximer un calcul par un procédé aléatoire, comme par exemple calculer l'aire du lac sur la Figure 1.3: si sur 500 tirs on en compte 100 dans le lac, on peut approximer la superficie de celui-ci à 20% de la superficie totale.

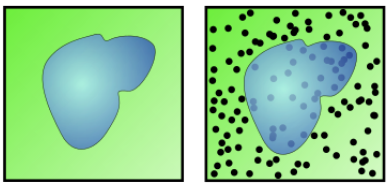

Figure 1.3: Principe de Monte Carlo illustré[5]

En utilisant Monte Carlo dans le cadre du rendu 3D, au lieu d'évaluer la réflexion totale et donc de faire le calcul de rendu pour toutes les directions incidentes, on va en choisir quelques unes aléatoirement pour lesquelles on va évaluer la fonction $L_i(u)f(u,v)cos\theta_u$. La moyenne de ces échantillons nous donne alors une approximation de l'intégrale vue précédemment en section 1.3. Lors d'une telle estimation, si l'échantillonnage se faisait pour une infinité de directions, la moyenne de ceux-ci correspondrait à la valeur exacte de l'intégrale. Cependant, le nombre fini d'échantillons utilisés fait différer la valeur estimée de la valeur exacte de l'intégrale, ce qui se traduit par du bruit sur les images rendues. Pour réduire ce bruit, les échantillons les plus pertinents doivent être priorisés pour approcher au mieux la valeur de l'intégrale.

Pour cela nous avons recourt à l'*"Importance Sampling"*.

1.5 Accélération du processus avec l'Importance Sampling

Générer des échantillons dans des directions aléatoires n'est pas pertinent pour approximer l'intégrale de l'équation du rendu dans notre cas, comme dans n'importe quelle situation où l'ont connait le comportement de la fonction à intégrer. De façon pratique dans notre cas, il est plus intéressant de chercher à échantillonner des directions autour de la réflexion spéculaire d'un matériau brillant puisque c'est dans cette zone que la plupart des rayons de lumière réfléchis trouvent leur origine. Afin de représenter ces directions optimales, nous avons recourt à une *Probability Density Function*, dorénavant abrégée **PDF**. La PDF est une fonction normalisée, où l'intégrale sur tout le domaine vaut 1 et où le maximum de *PDF(x)* représente la région la plus pertinente à échantillonner.

Du fait de la variation de la PDF, tous les échantillons tirés ne doivent pas avoir le même poids. Ainsi, un échantillon dans une région où la PDF est basse sera globalement représentatif de la valeur de beaucoup d'échantillons situés aux alentours. En revanche, un échantillon dans une région où la PDF est haute ne sera similaire qu'à peu d'échantillons proches. Pour compenser cette propriété lorsque l'on échantillonne la PDF de manière non uniforme, on pondère chaque échantillon par l'inverse de la PDF.

Ceci nous donne l'estimateur Monte Carlo suivant pour l'intégrale :

$$L_o(v) = \frac{1}{N}\sum_{k=1}^{N}\frac{L_i(u_k)f(u_k,v)cos\theta_{u_k}}{p(u_k,v)}$$

Etat de l'art

2.1 BRDF Explorer

BRDF Explorer est un logiciel initialement développé par Disney, qui permet d'afficher, d'analyser et de développer des fonctions BRDF. Les sources de BRDF Explorer sont *open source*, certaines améliorations ont d'ailleurs été apportées par l'INRIA et ce logiciel a également été l'objet d'un Projet de Fin d'Etudes en 2019.

La fonction principale du logiciel est de gérer le rendu sur un objet 3D (sphère ou Utah teapot) des fonctions BRDF analytiques écrites en GLSL (langage des shaders d'OpenGL), des données de BRDF mesurées de la base de données MERL et des données mesurées anisotropes de MIT CSAIL. Il propose aussi de visualiser la modification de différents paramètres de rendu sous forme de graphiques actualisés en temps réel.

Comme précisé dans l'introduction, le logiciel permet nativement d'avoir recours à "*l'Image-based Lightning*", avec la possibilité de l'activer ou non pour la carte d'environnement.

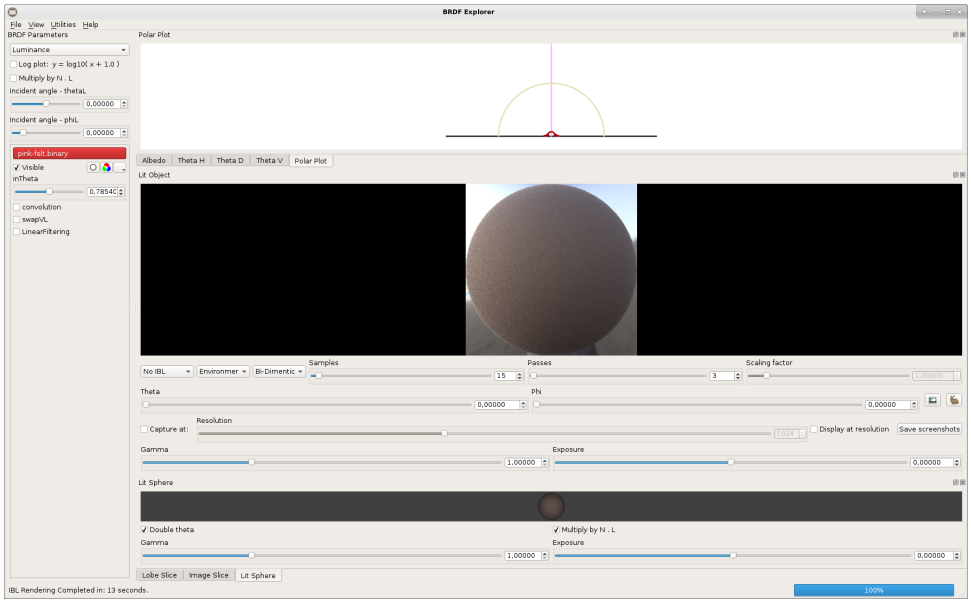

Figure 2.1: Interface BRDF Explorer

Lors du PFE de l'année dernière, plusieurs améliorations ont été apportées au logiciel, permettant la gestion de trois nouveaux modes de rendu:

- Le rendu par convolution qui permet d'obtenir un rendu le plus qualitatif possible, sans bruit. Il prend en compte toutes les contributions possibles au lieu d'échantillonner la carte d'environnement.
- L'échantillonnage préférentiel des BRDF analytiques qui permet de choisir les échantillons contribuant le plus à la réflexion totale lorsque la fonction d'échantillonnage est spécifiée pour la BRDF en question.
- Le rendu par Echantillonnage Préférentiel Multiple (abrégé MIS)(Figure 2.2) qui utilise conjointement l'échantillonnage de la BRDF et de la carte d'environnement afin de converger plus rapidement lorsque les deux échantillonnages sont très différents.

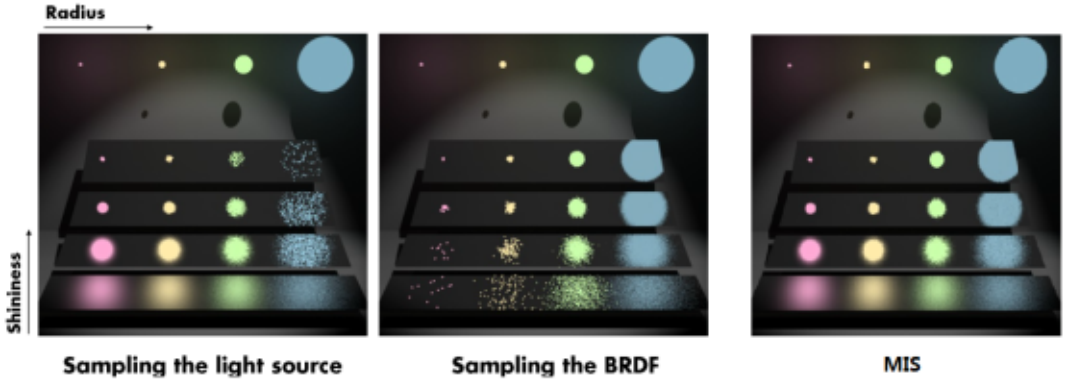

Figure 2.2: Mutltiple Importance Sampling[2]

2.2 OTMAP

Otmap est une bibliothèque développée en C++ par Georges Nader et Gael Guennebaud en 2016, proposant "une nouvelle manière extrêmement rapide de calculer des cartes de transport optimal entre des densités de probabilités 2D discrétisées sur des grilles uniformes"[11].
Le problème de transport optimal remonte à 1781, où il a été formulé par Monge[6]. Il consiste à déterminer la manière la plus économique, en temps ou en distance par exemple, pour transporter des objets depuis un ensemble de points de départ jusqu'à un ensemble de points d'arrivée. Le problème a ensuite était formalisé de manière plus précise par Kantorovich en 1942, comme consistant à "trouver la carte de transport T entre deux mesures de probabilité u et v minimisant un certain coût c"[11]. Le transport optimal est notamment utilisé aujourd'hui pour faire du transfert de couleur en traitement d'image.

Dans le cadre de notre projet, cette notion est intéressante puisqu'elle peut nous permettre de calculer différemment nos PDF et CDF nécessaires à l'échantillonnage préférentiel, le tout de manière efficace en temps.

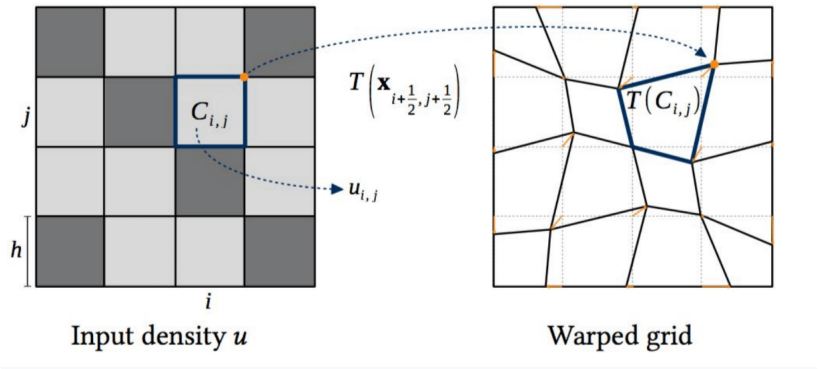

Figure 2.3: Exemple d'utilisation OtSolver : Déplacement des sommets de la grille régulière afin que la zone de chaque cellule corresponde à la densité de la cible

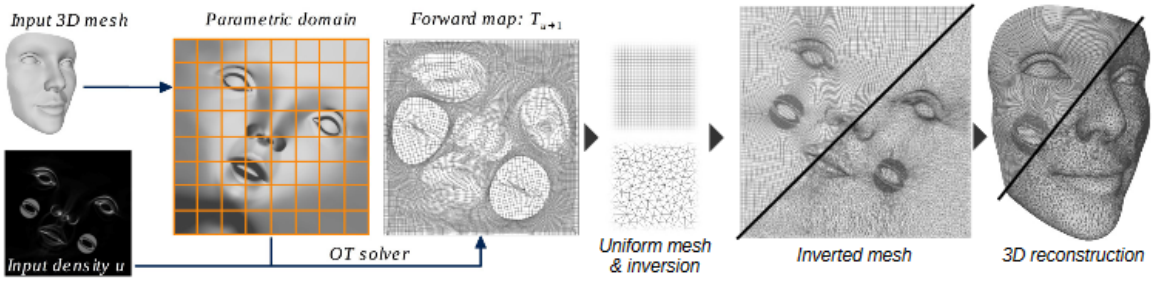

Figure 2.4: Pipeline d'Otsolver pour un exemple concret de remaillage

User Stories

Implémentée avec succès : ✔
En cours d'implémentation : ≈
Non implémentée : ✗

3.1 Échantillonnage des BRDF

Le client souhaite optimiser l'échantillonnage des BRDF grâce à *"l'Importance Sampling"* via un calcul par CDF inverse. Cette méthode existe déjà pour les BRDF analytiques, il faut donc la rajouter pour les BRDF mesurées au format MERL, UTIAH et ALTA.

3.1.1 Optimisation avec la CDF inverse

- Le client souhaite calculer une PDF à partir des données d'entrée. ✔
- Le client souhaite calculer une CDF inverse à partir de la PDF. ✔
- Le client souhaite envoyer la PDF et la CDF inverse calculées au *shader*. ✔
- Le client souhaite calculer la direction de la lumière dans le *shader*, à partir de deux variables aléatoires et des PDF et CDF. ≈

3.2 Échantillonnage des cartes d'environnements

Le client souhaite optimiser l'échantillonnage des cartes d'environnement grâce à *"l'Importance Sampling"* par transport optimal.

Le client souhaite pouvoir le faire selon deux stratégies différentes : l'adaptation à la volée et le pré-calcul d'une séquence avec décalage.

3.2.1 Optimisation avec l'adaptation à la volée

- Le client souhaite pouvoir envoyer une PDF et un *mapping* (entre la PDF et une PDF uniforme) au *shader*. ✔
- Le client souhaite pouvoir trouver en tout point l'équivalent du *mapping* sur la PDF dans le *shader*, pour calculer la couleur de chaque fragment. ✗

3.2.2 Optimisation avec le précalcul d'une séquence avec décalage

- Le client souhaite envoyer au *shader* une texture contenant un *mapping* inverse entre les coordonnées de la PDF et une PDF uniforme. ✔
- Le client souhaite retrouver, par un appel à la texture, la PDF associée à chaque fragment dans le *shader*, afin de calculer la couleur finale de celui-ci. ≈

3.3 Outils d'évaluation qualitative

3.3.1 Evaluation de la vitesse de convergence

Le client souhaite pouvoir évaluer et comparer le temps de convergence des différentes méthodes. ✗

3.3.2 Visualisation du processus d'échantillonnage

Le client souhaite visualiser les échantillons qui ont été généré par *"l'Importance Sampling"* sous forme de points.

- Le client souhaite pouvoir activer ou non cette option. ✗
- Le client souhaite disposer de cette option pour les cartes d'environnements ainsi que pour les BRDF. ✗

Implementation

4.1 Échantillonnage préférentiel des BRDF mesurées

De la même manière que pour les BRDF analytiques déjà implémentées, nous souhaitons ajouter l'échantillonnage préférentiel pour des BRDF mesurées.

Pour ce faire, nous avons besoin d'une **PDF**, nommée $pdf(x)$, telle que $x \in [0, 1]$ et $pdf(x) \in [0, 1]$, et d'un **générateur de nombres aléatoires uniforme**, générant un nombre aléatoire entre 0 et 1 nommé ξ.

L'idée est d'utiliser le nombre aléatoire généré uniformément et passer de cette probabilité uniforme à la PDF.
Dans cette optique, nous avons recours à une nouvelle fonction, la *Cumulative Distribution Function* abrégée **CDF**, notée $cdf(x)$, définie sur le même domaine que $pdf(x)$ par l'intégrale de $pdf(x)$.

$$cdf(x) = \int_0^1 pdf(x)dx$$

Nous souhaitons désormais calculer l'inverse de cette fonction, $cdf^{-1}(x)$. L'idée finale est d'obtenir une correspondance du nombre aléatoire uniforme vers la PDF voulue en passant ε à la CDF inverse. Une représentation graphique intuitive est donnée Figure 4.1.

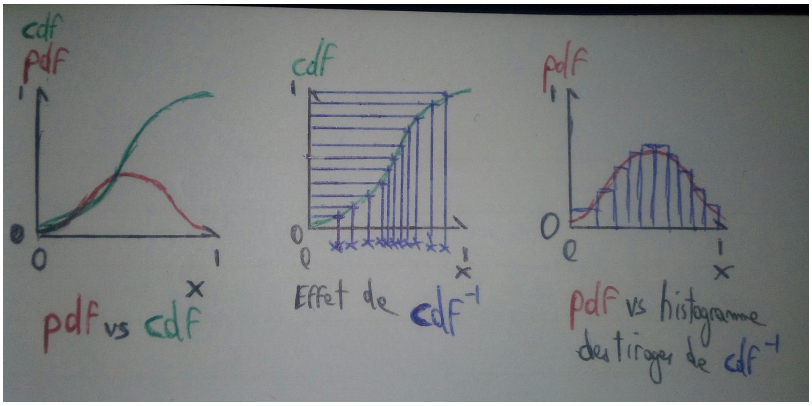

Figure 4.1: Schéma des liens entre PDF, CDF et CDF inverse

L'enjeu est aussi de pouvoir calculer la direction la plus pertinente à échantillonner $\overrightarrow{l}$ à l'aide de cette CDF inverse. Cela correspond à la zone autour de la réflexion spéculaire. Dans les faits, nous avons besoin de calculer la PDF associée puis de l'envoyer au *shader* en même temps que la CDF inverse, qui les utilisera pour calculer la direction de la lumière.

De manière plus précise, voilà comment nous avons procédé:
Dans le cas de la paramétrisation de BRDF de Rusinkiewicz, il y a 4 paramètres à savoir $(\theta_h, \phi_h, \theta_d, \phi_d)$. Dans un premier temps, on considère les matériaux isotropes ce qui permet d'éliminer une dimension, et donc le paramètre ϕ_h.

Les données dont nous disposons en entrée correspondent à trois données pour chaque valeur possible des angles θ_h, θ_d et ϕ_d correspondant aux trois composantes rouge, verte et bleue.
A partir de celles-ci, il faut calculer une valeur de BRDF pour chaque angle θ_h possible, ϕ_h ne variant pas dans le cas d'un matériau isotrope. Pour chaque valeur de θ_h, on calcule la moyenne des données mesurées pour tous les θ_d et ϕ_d:

$$\forall \theta_h, \phi_h, BRDF(\theta_h) = \frac{1}{90 * 180} \sum_{\theta_d}^{90} \sum_{\phi_d}^{180} brdf(\theta_h, \phi_h, \theta_d, \phi_d)$$

On obtient ainsi un tableau de valeurs de BRDF avec les trois composantes de couleurs R, G et B.
Une fois ces valeurs récupérées, on obtient la luminance en moyennant les trois canaux.

$$luminance(\theta_h) = \frac{1}{3}(R + G + B)$$

La PDF est alors obtenue en normalisant chaque valeur de luminance.

Pour obtenir la CDF inverse, il faut d'abord calculer la CDF. Il s'agit simplement de sommer pour chaque case la valeur de la PDF actuelle et les valeurs précédentes :

$$cdf_d(0) = 0$$

$$cdf_d(i+1) = cdf_d(i) + pdf(i')$$

Enfin, la CDF inverse est calculée en faisant un échantillonnage uniforme sur les valeurs précédemment trouvées, puis en faisant pour chaque valeur une recherche dichotomique et une interpolation sur la CDF, ce qui permet de retrouver la valeur correspondante en abscisse (correspondant à la valeur de CDF inverse, voir Figure 4.4).

$$\exists a, b | a' <= \xi <= b'$$

$$cdf^{-1}(\xi) = cdf_d^{-1}(a) + (cdf_d^{-1}(b) - cdf_d^{-1}(a)) * (\frac{\xi - a'}{b' - a'})$$

Ici, nous avons choisi de faire un échantillonnage resserré pour obtenir une CDF inverse précise (deux fois plus de valeurs dans la CDF inverse que dans les PDF et

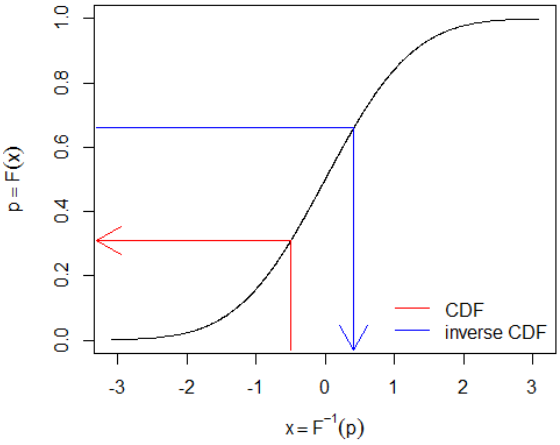

Figure 4.2: Correspondance entre CDF et CDF inverse[3]

CDF), ce qui permet de ne pas recalculer la PDF ensuite, car la CDF inverse sera fidèle à la PDF déjà calculée.

Une fois les PDF et CDF inverse obtenues, il faut les envoyer au *shader* pour en déduire la direction de la lumière. Pour cela, on utilise des `sampler1D`, qui permettent de bénéficier de l'interpolation linéaire.

Dans le *shader*, comme pour les BRDF analytiques, une nouvelle fonction est ajoutée pour calculer la direction de la lumière en chaque point, ainsi que la PDF correspondante (cf SampleBrdf).
A partir des deux variables tirées aléatoirement u et v, les valeurs θ_h et ϕ_h sont calculées :

$$\theta_h = \pi * cdf^{-1}(u)$$

$$\phi_h = 2\pi * v$$

Les deux valeurs d'angles calculées, on obtient ainsi le vecteur $\overrightarrow{h}$, *halfway vector* entre $\overrightarrow{v}$ et $\overrightarrow{l}$. Sachant $\overrightarrow{v}$, on peut donc calculer $\overrightarrow{l}$ comme suit (en suivant la formule du rayon réfléchi) :

$$\overrightarrow{l} = 2 * dot(\overrightarrow{v}, \overrightarrow{h}) * \overrightarrow{h} - \overrightarrow{v}$$

Pour calculer la PDF correspondante, on fait un appel texture sur la PDF à l'indice θ_h précédemment trouvé, que l'on converti de radians à degrés, ce qui nous permet d'obtenir la PDF_{thetaH}.
Ils nous faut aussi la PDF_{phiH} qui du fait de son uniformité vaut $(\frac{1.0}{2*PI})$.

On obtient alors la PDF finale grâce à la formule :

$$pdf = \frac{PDF_{thetaH} * PDF_{phiH}}{(4.0 * |dot(h_{local}, wo_{local})| * sin_{thetaH})}$$

Enfin, la direction de la lumière et la PDF associée sont envoyées au shader pour le calcul de la couleur du fragment.

4.1.1 Résultats:

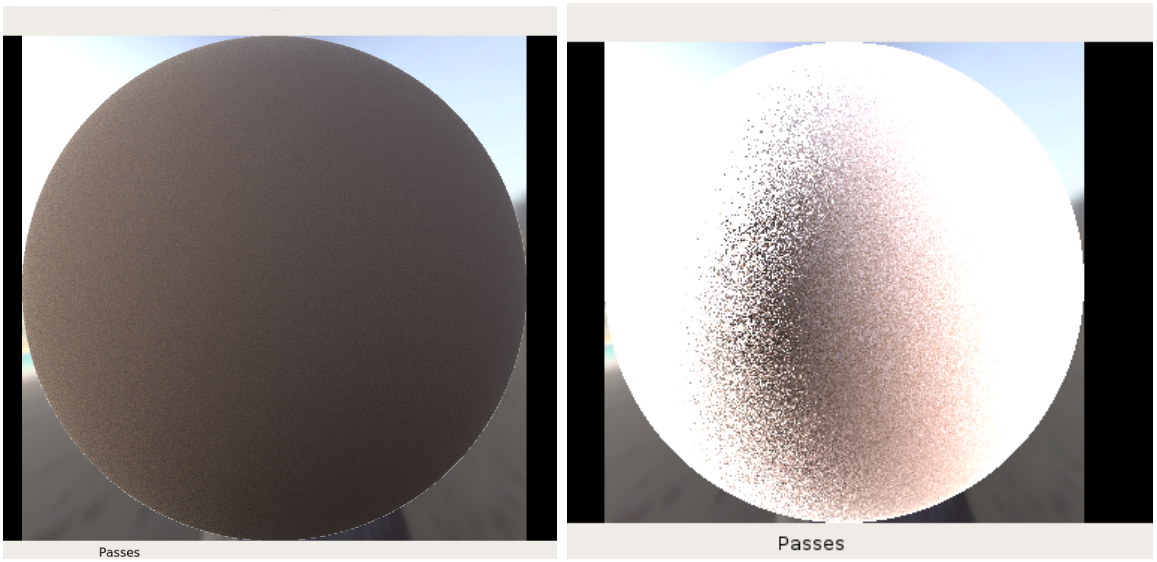

Figure 4.3: Résultat attendu et obtenu sur beige-fabric.binary avec calcul de la pdf

On constate que ce résultat ne correspond pas à l'image de référence. D'après nos clients et nos tests, il s'agirait d'une erreur dans le calcul de la formule finale de notre PDF mais malgré nos tentatives multiples nous n'avons pas réussi à la corriger.

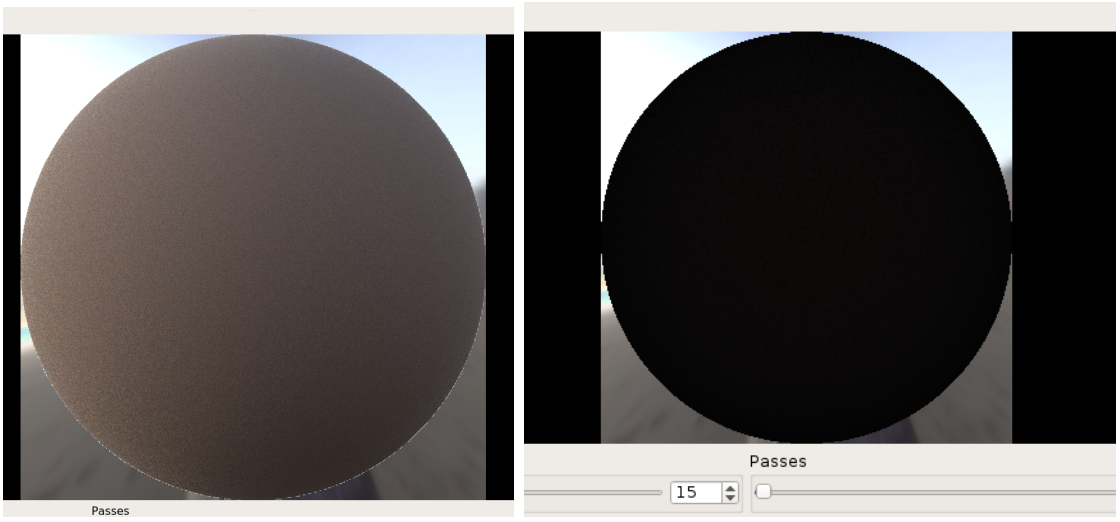

Figure 4.4: Résultat attendu et obtenu sur beige-fabric.binary avec calcul de la pdf et calcule de la direction de sortie

Là encore, le résultat ne correspond pas à l'image de référence. L'erreur viendrait de la manière dont on calcule notre direction de sortie dans notre shader mais nous n'avons pas réussi à trouver une solution à ce problème.

Ces deux aspects étant fortement liés aux calculs de la PDF, CDF et CDF inverse, nous avons pris soin de coder plusieurs tests pour nous assurer de leur bon fonctionnement respectif et ainsi pouvoir les exclure de la potentielle source d'erreur.

4.2 Amélioration de l'échantillonnage des cartes d'environnement par transport optimal

L'objectif de cette partie est d'améliorer l'efficacité en temps du calcul du rendu, en utilisant une carte de transport plutôt qu'une CDF pour calculer la couleur de chaque fragment.
Ainsi, à l'aide d'un échantillonnage uniforme et du mapping (Figure 4.5), on peut reproduire la PDF et naviguer de manière bi-dimensionnelle afin d'obtenir des images de qualité en peu de temps.

Pour cela, deux stratégies sont possibles :

- Précalculer la carte de transport inverse et l'envoyer au shader, qui fera un simple appel texture pour retrouver les coordonnées nécessaires au calcul de la couleur.
- Faire une adaptation à la volée, où l'on cherche directement dans la carte de transport côté shader, avec l'aide d'une structure accélératrice.

Dans les deux cas, nous passerons par l'API de OTMap et transmettrons les informations aux shaders par le biais de textures.

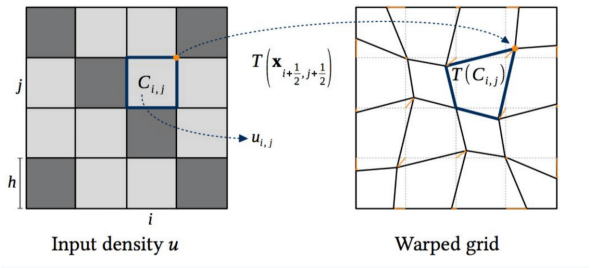

Figure 4.5: Déplacement des sommets de la grille régulière afin que la zone de chaque cellule corresponde à la densité de la cible

4.2.1 Accélération avec le précalcul de la carte de transport inverse

Mise en oeuvre

Tout d'abord, on souhaite envoyer au shader la carte de transport inverse.
Pour cela, on utilise la bibliothèque `otmap` qui fournit une méthode effectuant un *mapping* à partir de la carte d'environnement normalisée représentant la PDF de celle-ci vers une PDF uniforme.
On obtient une carte de transport et sa carte inverse associée. C'est cette dernière qui nous intéresse.
La carte de transport étant sous forme de maillage, il faut la transformer en texture, en stockant simplement les coordonnées de chaque point.
Enfin, la texture est envoyée au shader.

Dans le shader, on tire uniformément deux coordonnées. Puisque l'on a déjà la carte de transport inverse, il suffit de faire un appel texture sur celle-ci au point tiré, et l'interpolation linéaire d'OpenGL nous donne les coordonnées uv que l'on cherche pour calculer la couleur du fragment.

Cette méthode fournit un résultat plus rapide mais moins précis. C'est pourquoi il peut être intéressant de considérer la deuxième méthode d'adaptation à la volée.

Résultats

Cette partie a été implémentée dans la branche otmap-precalcul.
Voici ci-dessous les résultats obtenus :

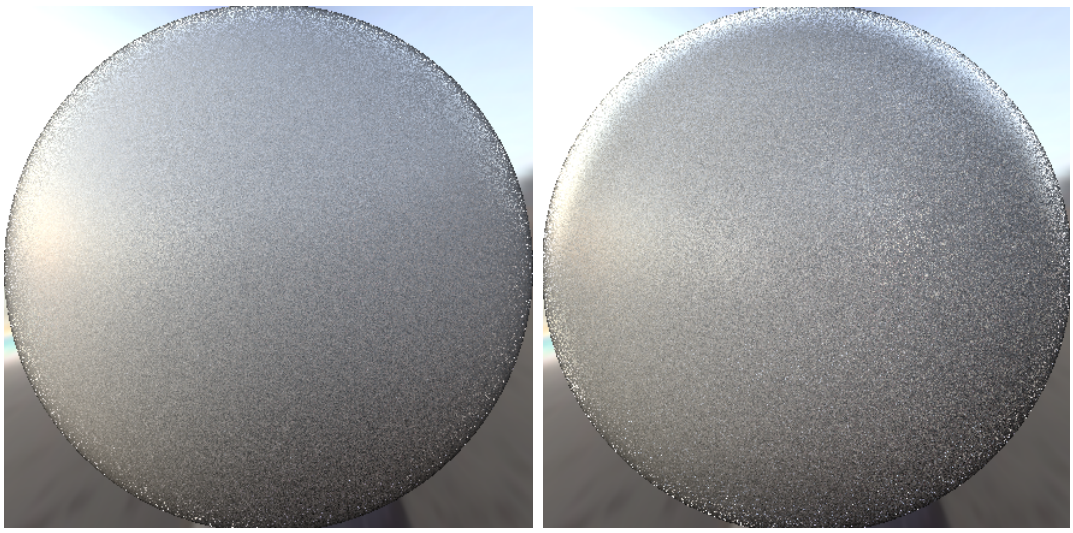

Figure 4.6: Résultats attendu avec CDF (gauche) et obtenu avec OTMAP (droite) sur ashikhman_shirley.brdf

On peut constater que pour le même nombre de samples et de passes, le résultat obtenu est plus bruité et contient des différences de couleur et d'éclairage.
Cela pourrait être lié à une mauvaise transformation du *mapping* en texture, ou à un mauvais calcul de la PDF en chaque fragment.
De plus, le temps de convergence n'est pas significativement meilleur avec cette nouvelle méthode, qui devait permettre une amélioration importante du temps de convergence. Il serait donc nécessaire de revoir l'implémentation. Le fait de redimensionner la carte d'environnement pour obtenir une image carrée, ou la transformation du *mapping* en texture pourraient être à l'origine du temps de convergence trop important.

De plus, en utilisant à la place de de la PDF calculée à partir de la carte d'environnement une PDF constante, nous constatons le résultat suivant :

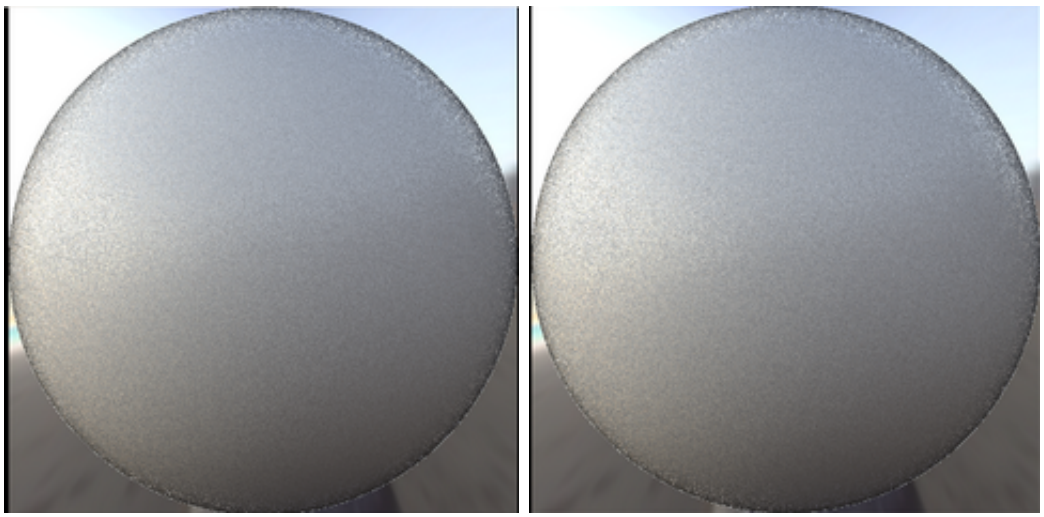

Figure 4.7: Résultats avec CDF (gauche) et avec OTMAP (droite) sur ashikhman_shirley.brdf

Nous pouvons donc penser qu'il y a un problème avec cette PDF calculée, mais nous n'avons pas pu trouver le problème malgré de multiples modifications.

4.2.2 Adaptation à la volée

Mise en oeuvre

Si on choisit l'adaptation à la volée, le même *mapping* est effectué. Le calcul du *mapping* inverse sera fait sur carte graphique, pour une précision accrue.

Le shader tire deux nombres aléatoires pour chaque rayon qu'il veut échantillonner, afin de trouver sa direction. Pour ce faire, il va comparer le point tiré avec les quads du mapping afin de trouver le quad le contenant, calculer les coordonnées barycentriques, puis le transposer dans le mapping et obtenir le résultat.

Ces données sont stockées dans une texture de $(n+1)*(n+1)$, avec n le nombre de quads sur une ligne. Cette structure utilise la topologie en grille du mapping pour limiter la place en mémoire. Chaque pixel correspond à un sommet, ses coordonnées sont stockées dans les canaux rouges et verts. Il est possible de référencer un quad avec juste un sommet d'index (i, j), et en dériver les 3 autres de coordonnées $(i+1, j)$, $(i+1, j+1)$ et $(i, j+1)$.

Cependant, comparer tous les quads prendrait beaucoup de temps de calcul. Une structure supplémentaire est nécessaire pour accélérer le calcul. Pour ce faire, une texture b de taille arbitraire est crée, et transposée sur le mapping. Pour chaque pixel de cette texture, la liste des quads q le chevauchant est calculée. Cette liste de quads est ajoutée ensuite à un tableau l, représenté dans le shader par une texture unidimensionnelle, contenant l'index du sommet définissant le quad dans ses canaux rouges et verts. L'index de début et de fin de q dans l est ensuite stocké dans le pixel correspondant dans les canaux rouges et verts de b.

Le shader ensuite récupère le pixel de b à la coordonnée tirée aléatoirement, récupère les bords de q dans l et effectue la comparaison uniquement sur ces quads, réduisant grandement le nombre de comparaisons à effectuer.

Résultats

Cette fonctionnalité n'étant pas développée jusqu'au but, nous n'avons pas de résultats dessus.

Architecture

5.1 Architecture de classes

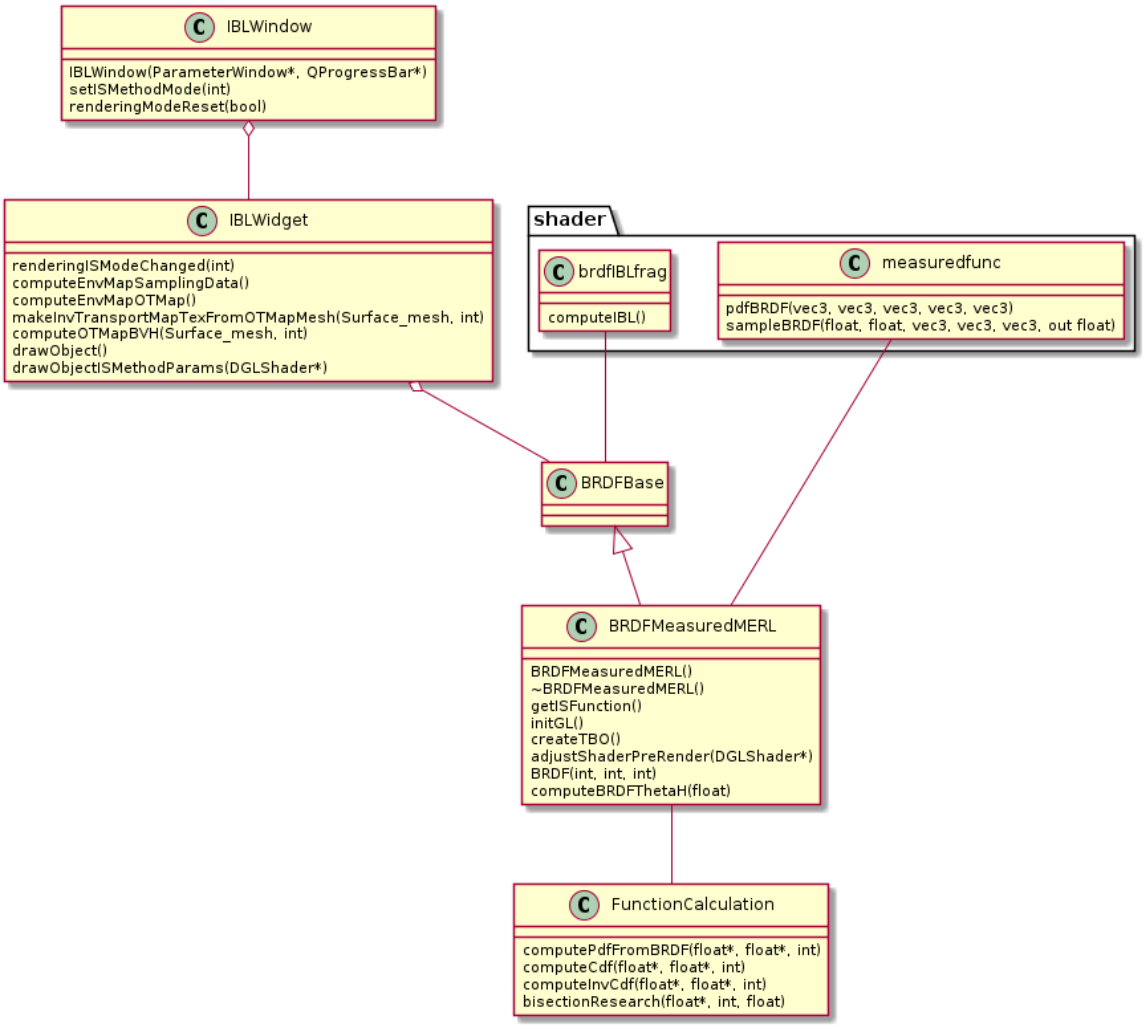

Figure 5.1: Diagramme UML des modifications apportées

5.2 Refactoring de IBLWidget

Une des premières choses faites a été de lire et nettoyer IBLWindow, afin d'améliorer la lisibilité du code, réduire les bugs et permettre une meilleure compréhension du style de programmation de BRDFExplorer.

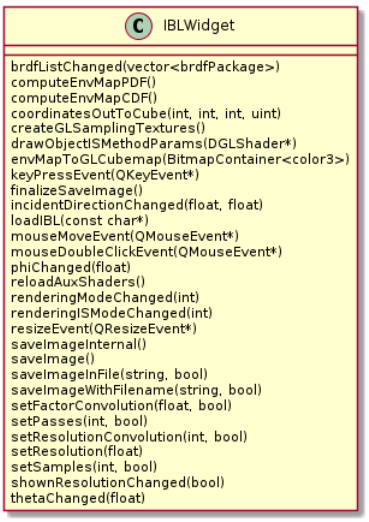

Figure 5.2: Liste des fonctions modifiées

5.2.1 Renommage de fonctions privées

Certaines fonctions avaient des noms confus ne faisant pas partie de l'interface publique de la fonction, ils ont été changés pour plus de lisibilité.

- calculateProbs → calculateCDF : Plus exact par rapport à l'utilité de la fonction et l'ajout de nouvelles méthodes
- saveImage(string, bool) → saveImageInFile(string, bool) : Renommée pour éviter la confusion avec saveImage() qui ne fait pas le même travail.

5.2.2 redrawAll()

La fonction redrawAll appelle resetComps() et updateGL(). Cette séquence existe telle quelle dans beaucoup de fonctions, et pour permettre une lisibilité accrue, redrawAll a été utilisée à la place. Voici la liste des fonctions en question :

- reloadAuxShaders()
- saveImageInternal()
- finalizeSaveImage()
- thetaChanged(float)

- phiChanged(float)
- setResolutionConvolution(int, bool)
- setFactorConvolution(float, bool)
- setSamples(int, bool)
- setPasses(int, bool)
- incidentDirectionChanged(float, float)
- brdfListChanged(vector¡brdfPackage¿)
- renderingModeChanged(int)
- renderingISModeChanged(int)
- shownResolutionChanged(bool)
- setResolution(float)
- resizeEvent(QResizeEvent*)
- mouseMoveEvent(QMouseEvent*)
- mouseDoubleClickEvent(QMouseEvent*)

5.2.3 Extraction de code

Plusieurs parties du code ont été extraites et mises dans des fonctions annexes afin d'augmenter la lisibilité et réduire les duplications de code.

- drawObjectISMethodParams(DGLShader*) : Extraite de drawObject pour pouvoir changer les paramètres envoyés en fonction de la méthode
- computeEnvMapPDF() et computeEnvMapCDF() : Extraits de computeEnvMapSamplingData() pour clarifier et permettre un ajout plus simple de la méthode OTMap
- saveImageInternal() : Fusion et extraction du code de keyPressEvent(QKeyEvent*), saveImage() et saveImageWithFilename(string, bool)
- loadIBL(const char*) : Extraction du code de fin de chargement de fichier qui était commun à deux sous-fonctions.

5.2.4 Factorisation et réduction de code

Certains morceaux de code ont été factorisés pour éviter la duplication à petite échelle.

- coordinatesOutToCube(int, int, int, uint) : Changements des ifs par un switch
- envMapToGLCubemap(BitmapContainer<color3>) : Remplacé par beaucoup de lignes identiques à l'exception d'une variable par une boucle.
- createGLSamplingTextures() : Remplacé par des lignes identiques à l'exception de deux variables par une boucle.

5.2.5 Captures d'écran

Le système de capture d'écran avait beaucoup de code dupliqué et est assez confus. Les fonctions affectées sont :

- saveImageInFile(string, bool) : Renommée depuis saveImage pour éviter la confusion.
- finalizeSaveImage() : Non modifiée. Fait la sauvegarde sur le disque en utilisant saveImageInFile.
- saveImageInternal() : Fusion et extraction du code de keyPressEvent(QKeyEvent*), saveImage() et saveImageWithFilename(string, bool)
- saveImage() : Utilise désormais saveImageInternal()
- saveImageWithFilename(string, bool) : Utilise désormais saveImageInternal()
- keyPressEvent(QKeyEvent*) : Apelle désormais saveImageWithFilename au lieu d'avoir son code entier dans la fonction.

5.3 Modifications pour OTMap

L'implémentation de OTMap étant faite avec la bibliothèque associée, les changements sont contenus dans peu de fonctions. Il a été nécessaire d'ajouter une boite de sélection, et d'effectuer les calculs permettant un rendu semblable à l'original obtenu plus rapidement.

5.3.1 IBLWindow

Les changements effectués dans IBLWindow permettent de choisir si on utilise OTMap ou non.

- *IBLWindow(ParameterWindow*, QProgressBar*)* : Ajout de la nouvelle boite de sélection et connexion de son signal a son slot.
- *setISMethodMode(int)* : Slot de la boite de sélection, apelle IBLWidget pour lui dire de changer la méthode utilisée.

- *renderingModeReset(bool)* : Permet de gérer le changement du nombre d'options disponibles en fonction du calcul à faire.

5.3.2 IBLWidget

IBLWidget s'occupe du côté C++ du rendu. Nous avons trouvé les parties importantes et ajouté les modifications nécessaires.

- *renderingISModeChanged(int)* : Permet de sélectionner la méthode à utiliser
- *computeEnvMapSamplingData()* : Ajout d'appel à *computeEnvMapOTMap()*
- *computeEnvMapOTMap()* : Calcule une carte de transport optimale (OTMap) et son inverse à partir de l'Environment Map
- *makeInvTransportMapTexFromOTMapMesh(Surface mesh, int)* : Transforme la carte de transport inverse en texture, pour la méthode avec précalcul
- *makeQuadList(Surface mesh)* : Crée et remplit une liste de quads à partir de la carte de transport optimale
- *drawObject()* : Ajout de paramètres dans le shader pour le choix de l'algorithme
- *drawObjectISMethodParams(DGLShader*)* : Ajout des textures à envoyer au shader

5.3.3 brdfIBL.frag

brdfIBL.frag s'occupe de la partie GPU du rendu. Nous avons ajouté les nouvelles méthodes qui emploient les structures faites plus tôt.

- Variables : Ajout de booléens pour le choix de l'algorithme et des textures nécessaires.
- *computeIBL()* : Ajout des méthodes d'OTMap avec précalcul

5.4 Modifications pour BRDF-MERL

5.4.1 BRDFMeasuredMERL

- *BRDFMeasuredMERL()* : Ajout de `isFunc` dans les shaders, fonction permettant l'*importance sampling*
- *˜BRDFMeasuredMERL()* : Libération des nouvelles ressources
- *getISFunction()* : Nouvelle fonction permettant de récupérer `isFunc` la fonction d'*importance sampling*

- *initGL()* : Modification lourde pour implémenter la fonctionnalité : appel aux différentes fonctions de création des PDF et CDF inverse et transformation de celles-ci en textures
- *createTBO()* : Simplifie la création de Texture Buffer Object
- *adjustShaderPreRender(DGLShader*)* : Envoi des nouvelles textures
- *BRDF(int, int, int)* : Structure la BRDF selon ses trois composantes de couleurs
- *computeBRDFThetaH(float)* : Calcule chaque valeur de la BRDF en fonction de l'angle θ_h

5.4.2 FunctionCalculation

Nouveau fichier d'aide.

- *computePdfFromBRDF(float*, float*, int)* : Nouvelle fonction calculant la PDF à partir de BRDF θ_h
- *computeCdf(float*, float*, int)* : Nouvelle fonction calculant la CDF à partir de la PDF
- *computeInvCdf(float*, float*, int)* : Nouvelle fonction calculant la CDF inverse à partir de la CDF
- *bisectionResearch(float*, int, float)* : Effectue la recherche dichotomique, utile pour le calcul de la CDF inverse

5.4.3 measured.func

- Variables : Nouvelles textures
- isFunc : Ajout des marqueurs
- *pdfBRDF(vec3, vec3, vec3, vec3, vec3)* : Nouvelle fonction utilisée lors du choix MIS dans l'interface
- *sampleBRDF(float, float, vec3, vec3, vec3, out float)* : Modifications permettant le calcul de la PDF d'un fragment et de la direction de la lumière

Tests

6.1 Tests existants

Les tests exitants sont regroupés dans un dossier "tests". Ils sont implémentés en Python 3 et utilisent la bibliothèque libre d'utilisation "ImageMagick"[4] qui permet la manipulation d'images en plus de 200 formats.

Il y a tout d'abord un fichier "test_all.py" qui permet de lancer l'ensemble des tests présents dans le dossier, il suffit de joindre en paramètre l'executable de BRDF-Explorer, comme spécifié dans le fichier readme.md : "python3 test_all /path/to/BRDFExplorer". Dès lors, le logiciel va être instancié 31 fois consécutivement de telle sorte que les 11 fichiers tests soient exécutés. Les fonctions du fichier "test_utils.py" sont appelées par chaque test, notamment "do_test(...)" qui génère les images de références et les images à comparer aux références.

6.2 Métrique de test

La métrique utilisée pour les tests existants est la Mean Squared Error, ou erreur quadratique moyenne, abrégée MSE. La MSE permet d'avoir une idée de la différence entre les deux images au niveau du pixel, c'est une façon synthétique d'évaluer la proximité absolue entre les pixels des deux images. Plus la MSE est basse, plus les deux images comparées sont proches.
Il est important de mettre en place une telle métrique dans la mesure où elle permet de distinguer des différences entre des images qui à l'oeil semblent identiques. Calculer la MSE dans notre cas de figure est particulièrement pertinent car la caméra de BRDF Explorer est fixe. On aura donc toujours la sphère au même endroit, il suffit de ne pas changer de fond et de ne pas modifier les paramètres de la scène pour recréer deux images avec des méthodes de rendu puis en calculer l'écart absolu par la MSE.
Le test de validation passera si la MSE ne dépasse pas un certain seuil défini auparavant, sinon le test échoue.

6.3 Tests Ajoutés

Pour compléter la base de tests existants, à l'aide notamment de tests unitaires, nous avons choisi d'avoir recours au framework Gtest proposée par Google pour automatiser des tests en C++[1].

Les premiers tests que nous avons mis en oeuvre visaient à nous assurer du bon fonctionnement de nos fonctions calculant la PDF, la CDF ainsi que la CDF Inverse. En effet, ces fonctions étant à la base de nos implémentations, il était primordiale pour nous de les tester au plus vite et de la manière la plus complète possible.
Sur la base des tests existants en Python, nous aurions dû ajouter trois tests de validation pour tester les principaux scénarios de nos *user stories*. Cette méthodologie s'y prêtait très bien dans la mesure où les nouvelles méthodes de rendu devaient donner un résultat similaire aux méthodes existantes. Malheureusement, ces derniers n'ont pas pu être menés à bien.

Conclusion

A la fin de ce projet, nous n'avons pas atteint la majorité de nos objectifs.
Toutefois ce manquement peut être expliqué et nuancé. En effet, ce sujet a nécessité une grande compréhension des notions déjà abordées en cours, ainsi qu'un approfondissement poussé de celles-ci dans trois domaines distincts : les mathématiques, l'optique ainsi que l'informatique.

Ce dernier a demandé une attention particulière dans les premières semaines du projet étant donné que celui-ci reposait déjà sur un existant dense. La nécessité de s'immerger dans le code pour le comprendre, bien qu'étant un exercice intéressant, ne nous a pas permis d'avancer aussi vite que nous l'avions prévu de prime abord. Nous avons en effet dû comprendre et schématiser les dépendances qu'il pouvait y avoir entre les parties de codes en C++ propre au CPU et celles côtés GPU qui reposaient sur des shaders, avant de pouvoir envisager d'implémenter quoique ce soit.

Notre vision sur le projet est beaucoup plus claire à présent et c'est pourquoi, même si nous n'avons pas réussi à implémenter les fonctionnalités demandées, nous espérons à travers ce rapport avoir suffisamment documenté et explicité toute nos démarches afin que notre code puisse être facilement complété et corrigé. Nous estimons en outre que les corrections à apporter à notre code pour que celui-ci soit fonctionnel sont sommes toutes mineures et qu'elles relèvent d'erreurs de formules mathématiques plus que de problèmes d'implémentations, ce qui a été fortement suggéré par nos clients et corroboré par nos différents tests.

Pour conclure nous souhaiterions remercier M.GUENNEBAUD et M.PACANOWSKI de nous avoir accordé beaucoup de leurs temps et de nous avoir donné de précieux conseils durant nos entrevues hebdomadaires et qui nous ont permis d'entrevoir le niveau élevé qu'il pouvait y avoir dans les domaines qui sont les leurs, dans le cadre de la recherche universitaire.

Bibliography

[1] Bibliothèque gtest de google - github. https://github.com/google/googletest.

[2] A graphics guy's note - monte carlo integral with multiple importance sampling. https://agraphicsguy.wordpress.com/2015/08/11/monte-carlo-integral-with-multiple-importance-sampling/.

[3] Help me understand the quantile (inverse cdf) function - thread on stackexchange. https://stats.stackexchange.com/questions/212813/help-me-understand-the-quantile-inverse-cdf-function.

[4] Imagemagick - convert, edit or compose bitmap images. https://imagemagick.org/.

[5] Page wikipédia de "méthode de monte-carlo". https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo.

[6] Page wikipédia de "théorie du transport". https://fr.wikipedia.org/wiki/Th\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\accent19e\egroup\spacefactor\accent@spacefactororie_du_transport.

[7] Brdf explorer. https://www.disneyanimation.com/technology/brdf.html, 1998.

[8] Merl brdf database. https://www.merl.com/brdf/, 2006.

[9] Karine Joulan Eric Dumont and Vincent Ledoux. Tabulation and completion of measures brdf data for lightning computations. 2011.

[10] Jaroslav Krivánek Mark Colbert. Nvidia developer - gpu-based importance sampling.

[11] Georges Nader and Gael Guennebaud. Instant Transport Maps on 2D Grids. *ACM Transactions on Graphics*, 37(6):13, November 2018.

[12] Philippe Bekaert Philip Dutré, Kavita Bala. *Advanced Global Illumination*. 2006.

Annexe

Project Installation

1. Clone the repo using

   ```
   git clone https://gitlab.inria.fr/guenneba/BRDFExplorer.git
   ```

2. Add and link otmap to the project :

   ```
   git submodule init
   git submodule update
   ```

3. Compile the project :

   ```
   cd BRDFExplorer
   mkdir build && cd build
   cmake ../src/
   make
   cd ../src/
   ln --symbolic ../build/BRDFExplorer BRDFExplorer
   ```

4. Run the program **from the src folder**:

   ```
   ./BRDFExplorer
   ```

BRDF Explorer Installation

Here's the list of commands / things to do to make it work :

1. Clone [the repo](https://github.com/wdas/brdf) using

   ```
   git clone https://github.com/wdas/brdf.git
   ```

2. ```
   cd brdf
   mkdir build
   ```
   ```

3. Add the line

   ```
   prefix = build
   ```

   to the top of this file : src/brdf/brdf.pro

4. ```
   cd build
   qmake -qt=qt5 ..
   make -j8
   cd ../src
   ln --symbolic ../build/src/brdf/brdf brdfexplorer
   ```

5. Now run the program by using ‘./brdfexplorer‘

A few notes on that :

- prefix uses pf-makevar and qmake won't work if you don't have it, that's why we had to add the prefix line
- If coding at the CREMI, you'll need to specify qt5 since it will default to qt4 and say it doesn't know QOpenGLContext
- The executable needs to be run in the src folder, we added a symbolic link for convenience

OTMap Installation

1. Clone [the repo](https://github.com/ggael/otmap) using
   ‘git clone https://github.com/ggael/otmap.git‘
2. Change the CMakeLists.txt by adding ‘-pthread‘ to every executable like this :

   ```
   add_executable(otmap apps/otmap)
   target_link_libraries(otmap otapputils otlib ${ALLLIBS} -pthread)

   add_executable(stippling apps/stippling.cpp)
   target_link_libraries(stippling otapputils otlib ${ALLLIBS} -pthread)

   add_executable(barycenters apps/barycenters.cpp)
   target_link_libraries(barycenters otapputils otlib ${ALLLIBS} -pthread)
   ```

3. ```
   mkdir build
   cd build
   cmake ..
   make -j 8
   ```

Link Otmap to BRDFExplorer

Here's the list of commands / things to do to make it work :

1. In BRDFExlorer/src/CMakeLists.txt add :

   ```
   include_directories(${PROJECT_SOURCE_DIR}/brdf/otmap/extern/)

   include_directories(${PROJECT_SOURCE_DIR}/brdf/otmap/otlib/)
   ```

2. Copy the whole otmap folder in BRDFExlorer/src/brdf/otmap

3. Then, to test it, just add in BRDFExlorer/src/BRDFMeasuredMERL (for example)...

   ```
   #include"./otmap/otlib/transport_map.h"
   ```

4. ... and in a called function (for example BRDFMeasuredMERL::loadMERLData)

   ```
   std::vector<otmap::TransportMap> tmaps;
   ```

WARNING, don't forget the "otmap::"
WARNING, this steps, only for work for the compilation with Cmake (not with QT)