Recette

Global Illumination with Radiance Regression Functions

Etudiants :
Yannick BERNARD
Pierre GUERINEAU
Kevin MENIEL
Romain MOUTRILLE
Matthias ROVES

Encadrant :
Mathias PAULIN

28 février 2017

Version	Notes	Date
0.1	Première version	23/02/2017
0.5	Ajout tests et améliorations	25/02/2017
1.0	Version finale	27/02/2017

Table des matières

1 Introduction

L'éclairage indirect est un des problèmes à résoudre lors du rendu physiquement réaliste d'une scène en temps réel. Un effet d'éclairage global détaillé s'obtient grâce à de nombreux rebonds effectués par la lumière, ceci engendrant un temps de calcul coûteux. Pour contourner ce problème, nous utiliserons la « Fonction de Régression de la Radiance » *(RRF : Radiance Regression Function)* nous permettant ainsi de calculer l'éclairage indirect pour chaque point des surfaces en connaissant la direction de la caméra et les conditions lumineuses.

Le projet a pour objectif de pouvoir effectuer le rendu physiquement réaliste d'une scène en temps réel avec calcul de l'éclairage global par une fonction de régression de radiance. Cette fonction sera modélisée par un réseau de neurones qui permettra donc un calcul de l'éclairage indirect en temps réel lors du rendu.

2 Présentation du logiciel

2.1 Module d'extraction

Le module d'extraction permet d'extraire les données d'apprentissage d'une scène PBRT v3.

— Dans un premier temps, les positions de la caméra et de la lumière sont échantillonnées par l'algorithme hypercube latin. Pour ce faire, le programme d'extraction prend en entrée un fichier de scène PBRT, un nombre N de positions de caméras dans l'espace, un nombre M de positions de lumières ponctuelles pour chaque position de la caméra, ainsi qu'une AABB (Axis Aligned Bounding Box) de la scène materialisée par ses deux points d'extremités. Le programme génère en sortie $(N * M)$ fichiers scènes PBRT correspondants à chacune des configurations caméra/lumière échantillonnées.

— Chaque fichier ainsi créé est envoyé à PBRT v3 (via un script) pour un rendu de l'éclairage indirect. Les rendus sont effectués avec une caméra sphérique utilisant l'échantillonnage de Fibonacci, permettant de générer des échantillons uniformément répartis autour d'une sphère. Le rendu est paramétré par une taille d'image et un nombre d'échantillons à calculer pour la caméra.
Pour chaque fichier scène envoyé à PBRT v3, un fichier du même nom avec l'extension *.data* est créé. Celui-ci contient les différentes données qui seront envoyées en entrée du module d'apprentissage (position du point sur la surface, vecteur vue, position de la source, normale à la surface, éclairage indirect)

2.2 Module d'apprentissage

Le module d'apprentissage que nous avons développé permet de modéliser une fonction de régression à l'aide d'un réseau de neurones pour le calcul de l'éclairage indirect.

Il prend en paramètre d'entrée le chemin vers le dossier contenant les fichiers générés par le module d'extraction.
Il prend également en entrée un fichier *config.xml*, qui permet de définir les paramètres du réseau de neurones et des jeux de données *(exemple : architecture du réseau, fonctions d'activation, algorithme d'apprentissage...)*.

Le réseau de neurones est créé selon le fichier de configuration, puis le programme itère sur les fichiers *.data* contenant les données d'apprentissage. Les données sont extraites du fichier, sont formattées selon le fichier de configuration, puis sont utilisées pour l'apprentissage du réseau. On récupère en sortie d'apprentissage le réseau appris sous la forme d'un fichier *.xml*.

Pour parer d'éventuelles interruptions pendant l'apprentissage, on enregistre alternativement, à chaque itération, le réseau de neurones dans un fichier *neuralnetworksave1.xml* et *neuralnetworksave2.xml*. On aura ainsi en mémoire, en cas d'arrêt du programme pendant l'apprentissage, une sauvegarde du réseau de neurones.
Un fichier *training.log* est créé en début d'apprentissage. Il permet de garder en mémoire les fichiers *.data* dont l'apprentissage a déjà été réalisé. On peut donc reprendre un apprentissage arrêté au fichier *.data* qui était en cours et les suivants.

2.3 Module de rendu

Le module de rendu prend en entrée un fichier scène PBRT v3 (le fichier d'entrée du module d'extraction) et un réseau de neurones, dont l'apprentissage a été effectué dans le module d'apprentissage pour cette même scène. Un moteur 3D en OpenGL va calculer l'éclairage direct en utilisant le modèle de Cook-Torrance, nous avons également implémenté les ombres portées pour des sources ponctuelles (point shadows) utilisant le shadowmapping. Le calcul de l'éclairage indirect est réalisé en évaluant le réseau de neurones à chaque fragment à l'aide d'un compute shader GLSL. L'éclairage final est calculé en faisant la somme des éclairages directs et indirects dans un shader.

3 Description détaillée des fonctionnalités

3.1 Module d'extraction

Le module d'extraction est divisé en deux parties :

1. l'échantillonnage de la scène par hypercube latin
2. l'échantillonnage de la caméra par Fibonacci

Ces deux parties peuvent être appelées séparément *(par exemple pour paralléliser le travail sur plusieurs ordinateurs, on va extraire dans un premier temps les fichiers scènes, que l'on va envoyer en plusieurs parties sur les différents ordinateurs, puis exécuter les rendus séparément)*, mais on peut également exécuter les 2 parties simultanément à l'aide d'un script *bash*.

Ce script va dans un premier temps générer les fichiers scènes à partir des paramètres spécifiés, puis lancer les rendus avec PBRT pour chaque fichier. Il permet également de spécifier le traitement d'une partie des fichiers avec un indice de début et de fin de fichiers sur lesquels itérer.

3.1.1 Hypercube latin

Ce programme permet d'échantillonner l'espace d'une scène, donnée en entrée sous la forme d'un fichier *.pbrt*, par l'échantillonnage de l'hypercube latin. Il prend en entrée un fichier scène, un nombre d'échantillons de caméras n_c, un nombre d'échantillons de lumières n_l et va donc générer n_c positions de caméras dans la scène et pour chaque position, n_l positions de lumières : nous aurons donc $n_c * n_l$ fichiers.
Les positions de caméras et de lumières sont définies de base dans le format des fichiers *.pbrt*, chaque fichier créé aura donc ses positions de caméra et de lumière modifiées.
L'espace d'échantillonnage est déterminé par la bounding box de la scène *(sous la forme d'une AABB)*, qui sera calculée à partir du fichier *.pbrt* grâce à un loader prévu à cet effet.

3.1.2 Modifications de PBRT

Pour réaliser les rendus, des modifications ont été réalisées sur PBRT. Nous avons créé un intégrateur `"CO_path"` basé sur l'intégrateur de pathtracing, permettant de calculer uniquement l'éclairage indirect d'une scène 3D. Une camera `"Fibonacci"` a été crée. C'est une caméra "sphérique" qui se base sur l'échantillonage de Fibonacci pour générer des échantillons autour de la sphère de manière uniforme, sans obtenir des anomalies au niveau des pôles.

Nous avons également ajouté une fonction d'écriture de fichiers de sortie de PBRT, afin d'écrire un fichier contenant les données d'apprentissage au format binaire. Ces données sont stockées dans un fichier d'extension *.data*.

3.2 Module d'apprentissage

3.2.1 Fichier de configuration des réseaux de neurones

Afin de permettre à l'utilisateur de spécifier les paramètres de configuration du réseau de neurones et de l'apprentissage, nous avons décidé d'utiliser un fichier *config.xml* qui va permettre cela. Celui-ci décrit :

- Des informations concernant le **réseau de neurones** :
  - nombre de perceptrons par couche
  - fonction d'activation pour chaque couche

    - différentes méthodes des scalings layers et unscaling layers
- Des informations concernant le **jeu de données** :
    - Infos pour le splitting, ou découpage des données (méthodes de découpage des données, ratio données-selection, ratio données-test, ratio données-apprentissage pour la crossvalidation)
    - Infos pour l'apprentissage (algorithme d'apprentissage, erreur à atteindre)

Un parser de ce type de fichier a été réalisé, permettant la récupération de ces informations pour les transmettre au module d'apprentissage. Des structures de données ont spécialement été créées pour stocker les paramètres et faciliter leur accès lors du paramétrage du réseau et du jeu de données.

3.2.2 Bouclage sur les données pour apprentissage

Le programme d'apprentissage vérifie la présence du fichier *config.xml* et boucle sur tous les fichiers ayant *.data* comme extension *(fichiers de sortie du module d'extraction contenant les données de rendu de l'éclairage indirect)*, pour extraire les données d'apprentissage et les faire apprendre au réseau de neurones.
A la fin de chaque itération d'apprentissage, le réseau de neurones est sauvegardé tantôt dans *neuralnetwork-save1.xml*, tantôt dans *neuralnetworksave2.xml (il choisira le plus ancien)*, permettant d'avoir l'état cohérent le plus récent du réseau en cas d'interruption. Le nom du fichier d'extension *.data* traité est également inscrit sur une ligne dans un fichier *training.log*, ce qui permet de savoir quels fichiers restent à traiter. Avant chaque itération d'apprentissage, on vérifie que le fichier à traiter n'est pas déjà présent dans le fichier log, est donc n'a pas été traité. Ce fichier log est particulièrement utile dans le cas où l'apprentissage est arrêté avant la fin. On peut alors le reprendre là où il s'était arrêté (en initialisant le réseau de neurone avec le dernier xml de sortie avant la reprise de l'apprentissage).

3.2.3 Modification d'OpenNN

Des modifications ont été effectuées dans la bibliothèque OpenNN, afin de rendre celle-ci compatible avec les anciennes versions des compilateurs présentes sur les machines de la salle U3-112 (GCC 4.7), mais elle reste utilisable pour des versions plus récentes. Il nous a fallu également comprendre comment OpenNN initialisait ses DataSets. En effet, la méthode conventionnelle est de charger un fichier au format définir par OpenNN, or notre format est légèrement différent. Nous aurions pu exporter au format OpenNN, au prix d'une place mémoire plus importante. Nous avons donc choisi d'extraire nous-mêmes les données et de les rentrer dans une matrice. Nous avons dû corriger une fonction de la bibliothèque, qui permettait de créer un DataSet à partir d'une matrice, mais engendrait un bug. Enfin, les données d'Apprentissage présentaient des redondances, que nous avons éliminées.

3.3 Module de rendu

3.3.1 Modifications du moteur

- Afin d'avoir une utilisation des données cohérente, il a fallu ajouter au moteur le support de chargement de scènes PBRT v3. L'actuelle implémentation du parser de tels fichiers ne gère actuellement que les pointlights *(Car c'est le seul type de lumières que le moteur est actuellement capable de gérer)*. Il est important de noter que la caméra est automatiquement placée par le moteur au centre de la scène. Il n'est donc pas nécessaire de spécifier une caméra dans le fichier *.pbrt*. Il n'est pas non plus nécessaire de spécifier des types de données telles que l'intégrateur, le spectrum ou le film, etc, le moteur ne fonctionnant pas de la même manière que PBRT v3. Le chargement de scène se contente donc de charger toute la géométrie et les différents matériaux avec une couleur diffuse seulement.

- Le calcul de l'éclairage direct du moteur initial ne prenait pas en compte les ombres portées, nous avons donc ajouté le support des *"omnidirectional shadowmaps"*. Celles-ci permettent le calcul d'ombres portées avec des pointlights. Pour ce faire, on génère une depth-cubemap depuis le point de vue de la point light. On réalise donc 6 rendus de profondeur de la scène pour chacune des 6 faces du cube et on vient ensuite, pour chaque fragment de l'image finale vérifier dans la depth-cubemap si celui-ci correspond à une zone d'ombre. Le moteur a également été modifié afin de pouvoir supporter les geometry shaders, qui ont été nécessaires pour l'implémentation des shadowmaps.

— Le moteur utilise le rendu différé. Néanmoins, il a fallu permettre la possibilité de connecter les éléments du pipeline de manière à envoyer les textures aux bons pipelines. (Fig. 1) Un pipeline classique va commencer par une *geometry pass*, qui va envoyer des textures au *lighting pass*, etc. Dans notre cas, les données du *geometry pass* sont envoyées à la fois à l'élément *(ou Renderprocess)* d'éclairage direct et indirect. Nous avons donc ajouté la possibilité de définir pour chaque process quelle texture de sortie sera connectée à quelle texture d'entrée d'un autre process. Une fois les connexions définies, on construit le pipeline avec le dernier élément *(Un pipeline peut avoir plusieurs entrée, par exemple dans notre cas la geometry pass et la construction de la shadowmap, mais qu'une sortie)* et une fonction se chargera de déterminer l'ordre correct des processes dans une liste, afin que les processes nécessitant une texture dans un autre process seront exécutés après ceux-ci. Notre pipeline de rendu final ressemble donc à ceci :

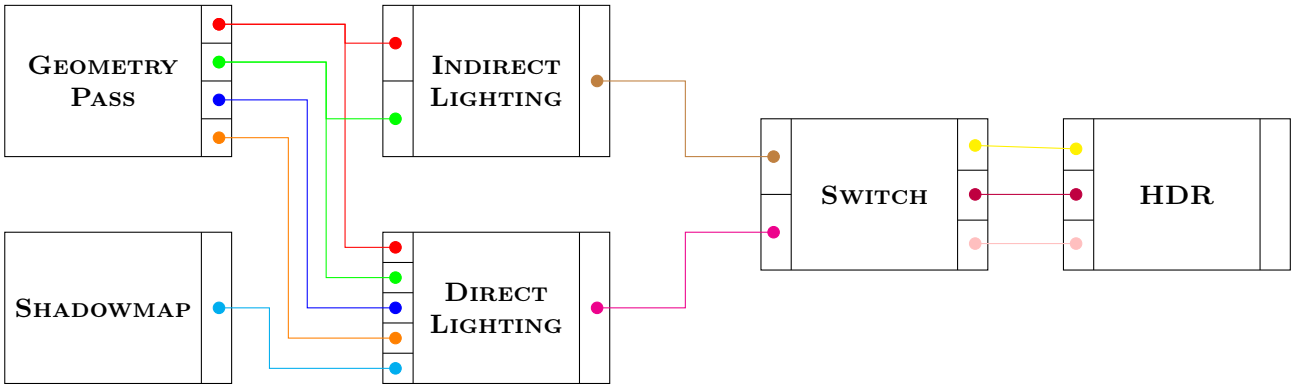


Figure 1 – Schéma du pipeline des *RenderProcesses* de notre moteur

- RGBA - position + profondeur
- RGB - normal
- RGBA - albedo + spéculaire
- RGB - matériau *(roughness + refraction + metalness)*
- DEPTH CUBEMAP - carte des profondeurs
- RGBA - éclairage indirect
- RGB - éclairage direct
- RGB - mix des éclairages direct et indirect
- RGB - résultat seuillé par la luminosité
- RED - luminosité

— **Sortie** : RGB - résultat avec HDR appliqué

3.3.2 Éclairage indirect

Le calcul de l'éclairage indirect est effectué par l'évaluation du réseau de neurones. Pour ce faire, nous utilisons un **compute shader** permettant la parallélisation des calculs de l'éclairage indirect pour chaque pixel. Nous détaillerons ici les entrées-sorties du compute shader ainsi que son fonctionnement.

Entrées :

— `frag_position` - *texture RGBA récupérée en sortie du geometry process*

— `camera_position` - *vec3 extrait depuis la scène*

— `point_light_position` - *vec3 extrait depuis la scène*

— `frag_normal` - *texture RGB récupérée en sortie du geometry process*

— `neural_network` - *Shader Storage Buffer Object*

Sortie :

— `indirect_lighting` - *Texture RGBA*

Le compute shader récupère un réseau de neurones à l'aide d'un SSBO (Shader Storage Buffer Object). Un SSBO est très utile ici car, il permet d'envoyer au shader une structure définie par le développeur, ici la structure du réseau de neurones. Nous avons donc accès à tous les élément du réseau de neurones à l'aide de structures à l'intérieur même du compute shader. L'évaluation du réseau de neurones sera identique au résultat de OpenNN, car nous avons récupéré les structures et fonctions d'évaluation de OpenNN que nous avons adaptées à notre utilisation. A noter que nous réduit les fonctions à leur strict minimum, c'est à dire les éléments utiles à notre Chef d'Oeuvre, afin d'optimiser au maximum le rendu en enlevant les élements "inutiles".
Nous avons également modifié la structure des fonctions *(de manière à obtenir un résultat identique)*, encore une fois dans un but d'optimisation.
Le choix des compute shaders s'est basé sur ces éléments :

Avantages :

— Calcul parallélisé sur GPU
— Utilisation de SSBO et donc de structures (lisibilité du code)
— Peu de configuration pour la mise en place (pas de link, compilation... intégré à OpenGL)

Inconvénient :

— Tableau dynamique à l'intérieur du compute shader impossible
— Taille des données limitées
— Optimisation difficile, rendant le code difficilement lisible
— Impossibilité d'exécution sur macOS

4 Tests

4.1 Extraction

4.1.1 Hypercube latin

Pour vérifier le fonctionnement du programme d'échantillonnage par hypercube latin, nous avons généré 100 échantillons dans la scène 3D, que nous avons projetés sur un plan en x, y et z. Nous avons pu vérifier visuellement, avec une figure Matlab que tous les échantillons étaient situés à l'intérieur de la bounding box et qu'ils respectaient l'échantillonnage par hypercube latin *(généralisation des 4 reines)*.

Résultat : Aucun point ne se trouve à l'extérieur de la bounding box. L'espace est bien échantillonné. Aucun point ne se trouve sur la même ligne qu'un autre, quelle que soit la coordonnée *(également vérifié à l'aide d'un parcours des points)*. Ce n'est pas tout à fait vrai pour les points situés sur les extrémités, en effet la génération des points donnait des positions légèrement en dehors de la bounding box, ce qui aurait engendré des résultats érronés lors du rendu *(on aurait eu du noir)*. Nous avons donc ramené ces points sur les bords de la bounding box, la contrainte de l'hypercube latin n'est pas respectée pour ces points, mais au moins ils n'engendreront pas d'erreur. (Fig. 2)

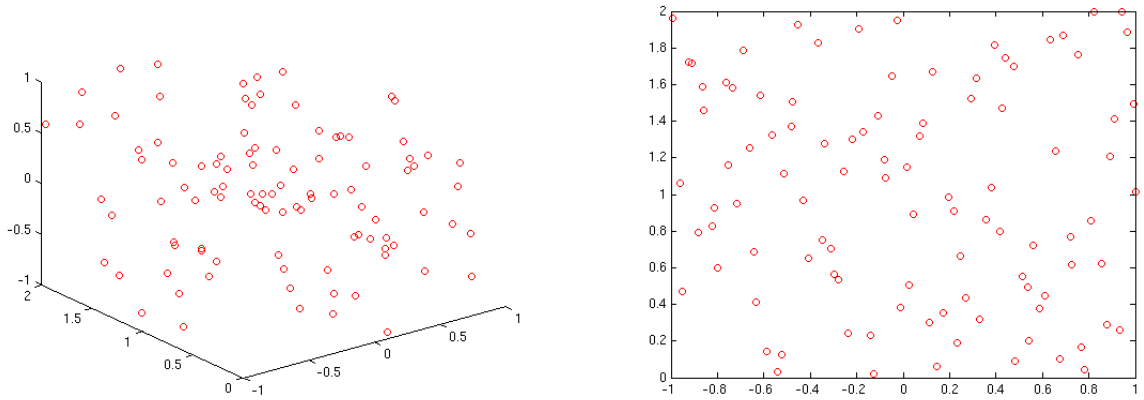
4.1.2 Fibonacci

La vérification de l'échantillonnage de Fibonacci s'est déroulée de manière similaire, nous avons généré des échantillons sur une sphère de rayon 1 et nous avons vérifié visuellement que le résultat était cohérent. A la page suivante, nous pouvons voir le résultat de l'échantillonage (Fig. 3), nous voyons bien que les échantillons sont répartis uniformément sur la sphère. La forme elliptique est due à la projection de la représentation.

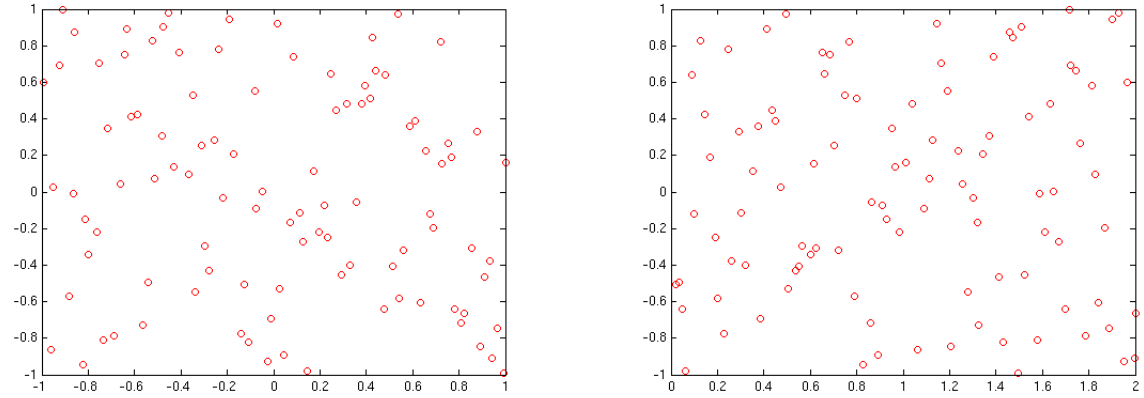
Résultat : Visuellement, l'échantillonnage est cohérent avec les résultats présentés dans le papier que nous avons implémenté. Il est difficile de vérifier les résultats autrement que visuellement. Néanmoins, nous pouvons observer que la sphère est échantillonnée uniformément, ce que nous voulions obtenir.

4.1.3 Script

Nous avons utilisé le script avec différentes valeurs en essayant de couvrir un maximum de configurations. Les fichiers sont correctements créés.

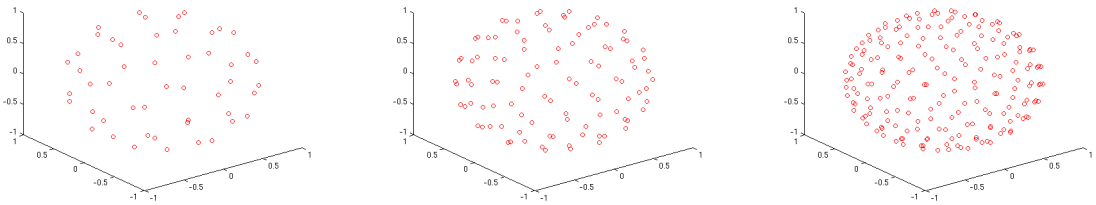


3D projection x



projection y projection z

FIGURE 2 – Affichage du résultat de notre algorithme d'échantillonnage par hypercube latin avec 100 échantillons



50 échantillons 100 échantillons 200 échantillons

FIGURE 3 – Affichage du résultat de notre algorithme d'échantillonnage de Fibonacci

4.1.4 Intégrateur

Afin de vérifier la cohérence de notre intégrateur, nous avons décidé de prendre pour référence l'image obtenue en ne prenant que l'éclairage indirect avec un intégrateur de type 'path'. Nous pouvons ainsi voir que l'image obtenue grâce à notre intégrateur (Fig. 5) est proche de celle de référence (Fig. 4). La seule différence se trouvant dans des effets de caustiques visibles à plusieurs endroits (Fig. 5). Ces derniers sont dûs à l'échantillonnage de la lumière qui se fait en un seul point plutôt qu'en une intégration de la surface couverte par le pixel.

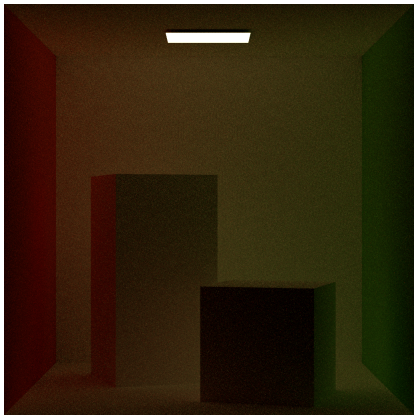


FIGURE 4 – Image de référence avec un intégrateur 'path'

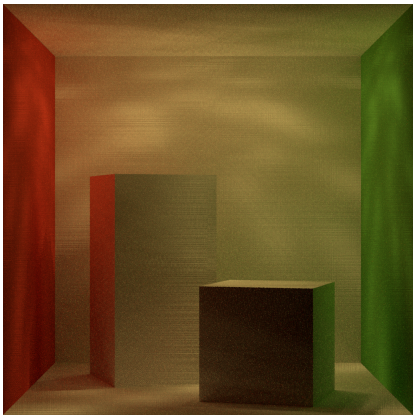


FIGURE 5 – Image obtenue avec notre intégrateur

4.2 Apprentissage

4.2.1 Fichier de configuration

Nous avons créé et parsé le fichier de configuration avec toutes les valeurs possibles et vérifié que les valeurs parsées étaient cohérentes. Nous avons également tenté de mettre des valeurs érronées, ce qui a engendré une erreur lors du parsing.

4.2.2 Apprentissage

L'apprentissage s'effectue via la bibliothèque OpenNN. Cette bibliothèque est très utilisée, nous partons donc du principe qu'elle est vérifiée et fonctionne correctement.

4.3 Rendu

4.3.1 Chargement de fichier PBRT

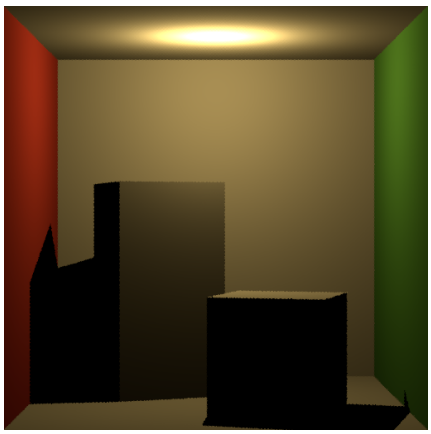


FIGURE 6 – Scène de référence avec PBRT v3

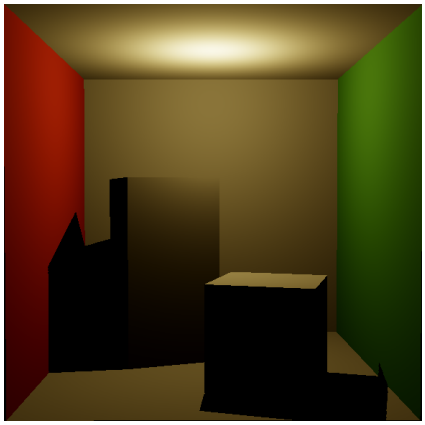


FIGURE 7 – Scène chargée dans notre moteur 3D

Résultat : Visuellement, la géométrie de la scène et les différents matériaux sont chargés. (Fig. 6) La point light est aussi bien présente. Il s'agit ici de voir la chargement des informations géométriques et des matériaux. Nous avons désactivé l'éclairage indirect de PBRT pour la comparaison. (Fig. 7)

4.3.2 Evaluation du réseau de neurones

Pour vérifier le résultat de l'évalutation du réseau de neurones dans le moteur, nous avons envoyé des input au réseau dans OpenNN et noté le résultat de l'évaluation. Ensuite, nous avons envoyé les mêmes inputs au réseau dans notre moteur et affiché la différence des outputs avec les résultats OpenNN *(1 si égal, 0 sinon).*

X : valeurs de sorties calculées par OpenNN
Y : valeurs de sorties calculées par le compute shader

vec4 pixel = vec4(abs(X.r - Y.r) < 0.0001, abs(X.g - Y.g) < 0.0001, abs(X.b - Y.b) < 0.0001, 1.0);

5 Matrice de couverture des exigences a posteriori

EXIGENCES	COUVERTURE	PRIORITÉ
Extraction		
La fonction de radiance va être apprise par morceaux, les données seront générées pour des sous-régions de la scène.	🔴	🔴
Une version modifiée de PBRT calculant uniquement la composante indirecte devra être réalisée.	🟢	🔴
Des formats de stockage interne et externe devront être définis.	🟢	🔴
Apprentissage		
Le réseau de neurone doit être créé grâce à la bibliothèque OpenNN.	🟢	🟡
Le réseau de neurone devra calculer l'éclairage indirect d'un point à partir de la position de ce point, de sa normale, de la direction de la lumière et de la direction de la caméra.	🟢	🟠
Le réseau de neurone devra calculer l'éclairage indirect d'un point à partir des paramètres de la BRDF.	🔴	🟢
Le réseau de neurone doit pouvoir calculer l'éclairage indirect d'un point avec une erreur quadratique inférieure à un seuil défini par l'utilisateur.	🟢	🔴
Rendu		
Le rendu de l'éclairage indirect doit se faire en temps réel grâce l'évaluation des données par un réseau de neurones.	🟠	🔴
Le rendu de l'éclairage indirect de la scène doit être similaire à celui d'un moteur de rendu hors ligne (PBRT, Mitsuba).	🟠	🟡
Le rendu de l'éclairage direct doit être fait en temps réel.	🟢	🔴
Le rendu de l'éclairage direct doit utiliser le même type de BRDF que l'éclairage indirect.	🟢	🔴
Le rendu de l'éclairage direct doit utiliser le même type de source que l'éclairage indirect.	🟢	🟡

Niveau de couverture : 🟢 Couverte 🟠 Partiellement couverte 🔴 Non couverte
Niveau de priorité : 🟢 Faible 🟡 Moyenne 🟠 Forte 🔴 Indispensable

6 Améliorations possibles

6.1 Module d'extraction

Il est possible améliorer la camera de Fibonacci en mettant en place la création de rayon différentiel. Ceci permettrait d'obtenir un meilleur rendu, et donc de meilleurs échantillons, dans les cas ou le filtrage de texture est nécessaire.

Une autre amélioration du système d'extraction des données aurait été un premier tri de celle-ci. Une séparation des données à l'aide d'un arbre en groupe de variance fixe pourrait permettre un gain de temps considérable lors de l'étape d'apprentissage.

Un outil de compression/décompression utilisant la bibliothèque zlib a été développé. Il permet d'écrire des données dans un fichier sous forme compressée au format gzip. Cet outil a été développé dans l'idée de réduire la taille des données extraites sur le disque. Nous avons décidé de ne pas l'utiliser dans le cadre de nos tests, la taille des données ne justifiant pas de les compresser. Dans le cas d'utilisation d'une quantité de données beaucoup plus importante, il pourra être intéressant d'implémenter cet outil lors de l'extraction des données.

6.2 Module d'apprentissage

Le papier que nous devions implémenter utilisait des Kd-Trees pour découper la scène et utiliser plusieurs réseaux de neurones, lorsque la scène était complexe. Par manque de temps, nous n'avons pas pu développer cette fonctionnalité.
La bibliothèque OpenNN permet d'utiliser CUDA pour un rendu utilisant la méthode Quasi Newton. Nous n'avons pas réussi à la linker, il serait intéressant de l'utiliser afin d'augmenter la vitesse d'apprentissage.

6.3 Module de rendu

Il est possible d'améliorer le chargement d'une scène PBRT.
Actuellement, il ne prend en compte que les attributs de point lights et les formes géométriques avec leurs indices, leurs points, leurs normales et leurs uv. Ajouter la gestion d'autres types de topologie pour le mesh, nous ne considérons que des triangles actuellement. Il faudrait ajouter la possibilité de charger d'autres types d'éclairage (Directionnal lights, Area lights). Il serait avisé aussi de rajouter le chargement d'autres paramètres tels que le roughness, le metalness ou l'indice de réfraction. Et, bien sûr, de charger tous les paramètres possibles contenus dans la documentation d'un input file de PBRT v3.

Pour le calcul de l'éclairage indirect avec le réseau de neurones, il serait judicieux d'optimiser le code au maximum mais cela rendrait le code illisible. Il est possible d'améliorer le calcul par compute shader en découpant en plusieurs compute shaders si possible afin de réduire l'utilisation de la mémoire utilisée par le compute shader actuel. Cette utilisation de la mémoire peut être trop importante et non supporté par certaines cartes graphiques qui n'ont pas assez de mémoire disponible.

De plus, il est à noter que les compute shaders ne fonctionnent pas sur macOS. Il serait bien d'implémenter un autre moyen de calcul pour cet OS.

7 Conclusion

Nous avons réalisé et testé la plupart des tâches du projet, il reste cependant quelques éléments à finaliser afin d'avoir un résultat final équivalent à celui du papier *Global Illumination with Radiance Regression Functions*. Ce projet est un projet demandant beaucoup de travail et nous avons manqué de temps pour le mener à bien dans son intégralité, celà dit, nous sommes satisfait du travail que nous avons réalisé et des modules qui sont fonctionnels. Ce chef d'oeuvre a été l'occasion pour nous de travailler en équipe, chose que nous avons rarement l'occasion de faire dans le cadre de notre formation.
Le sujet était motivant, malgré sa complexité et nous a permis de nous confronter à des technologies que nous n'avions jamais utilisées. Pour conclure, nous nous contenterons de dire :

Mundi placet et spiritus minima

Ca n'a aucun sens mais on pourrait très bien imaginer une traduction du type : "Le roseau plie, mais ne cède... qu'en cas de pépin" ce qui ne veut rien dire non plus.