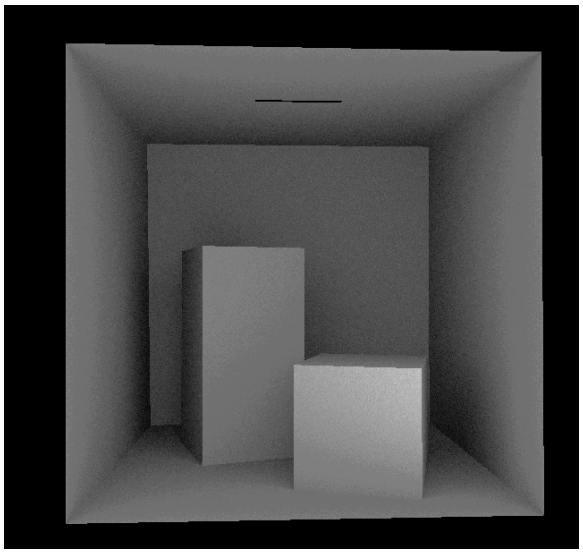
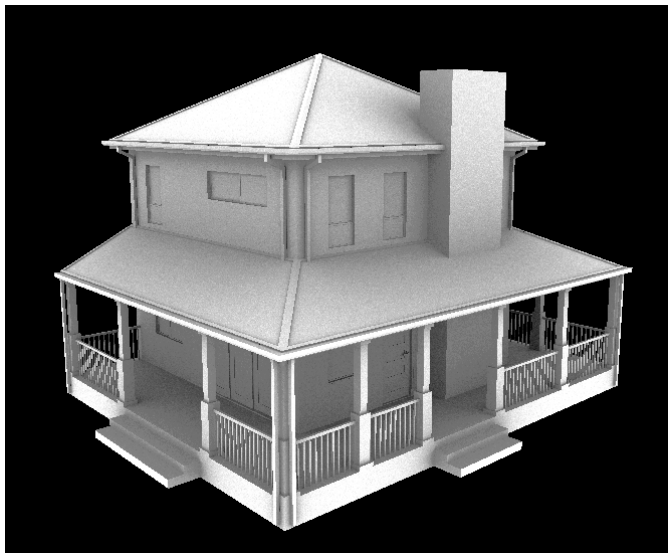
Rapport du TP1 du module de Programmation sur Cartes Graphiques

Lancer de rayons et (compute) shaders

Guillaume DURET (p2021346)



(a)



(b)

31 Janvier 2020

1 Introduction

Depuis la version 4.3 d'OpenGL il est possible de pouvoir programmer sur des cartes graphiques sans forcément passer par le pipeline graphique d'OpenGL. C'est en effet dans cette version que sont instauré les compute shaders qui permettent de réaliser de la programmation GPGPU. Ce TP a donc pour but d'utiliser ces compute shader pour réaliser du lancer de rayons sur la carte graphique qui permet de se rapprocher d'un rendu en temps réel.

Le code réalisé peut être retrouver sur le github : https://github.com/Guillaume0477/Carte$_g raphiquesurle fichier$tutos/M

2 Compute shaders

La première étape a été de se familiariser avec les compute shader et de l'utiliser pour un problème de lancer de rayon.

2.1 Pipeline

Le code d'origine a été le fichier $tuto_r aytrace_f ragment.cpp$ et $raytrace.glsl$ qui se trouve être un problème de lancer de rayon mais qui utilise des uniforms buffers. En effet un problème de lancer de rayon possède obligatoirement en données les triangles de la scène ainsi que les informations de la caméra intégré dans les matrices de changement de repère. Ainsi ce programme envoie tout le triangle de la scène dans un tableau d'uniform, ceux ont pour avantage de ne pas avoir de limite de mémoire (à part la limite de la mémoire vidéo) contrairement au uniform classique qui eux sont limité à 16 ou 32 Octets. Cette limite des uniform classique est loin d'être suffisant pour tout un ensemble de triangles, d'autant plus que la scène peut comporter un nombre sans limite de triangle dans le sens ou on peut avoir des scènes aussi grandes que voulu avec des détails aussi précis que voulu.

Ce programme utilise donc pour ce programme le pipeline graphique d'OpenGL et réalise les calculs d'intersection rayons/triangles dans le fragment shader. Au contraire l'idée est d'utiliser un compute shader pour pouvoir se séparer du pipeline graphique mais aussi pour avoir plus de contrôle sur la gestion de ce shader sur la carte graphique.

2.2 Compute shaders

En effet un compute shader a la possibilité de choisir combien de groupe de thread est exécuté. Les threads exécutés par le compute shader sont représentés en 3D avec chaque thread qui possède des indices 3D. Sachant que notre but est de réaliser une image 2D à partir de la génération de rayon, l'idée est de choisir une organisation de thread en 2D pour que chaque thread tire des rayons et se place dans une région précise de l'image résultante. Ainsi il est possible à partir de l'indice du thread obtenue avec la commande $gl\_GlobalInvocationID$ de connaître la position du pixel sur lequel on veut exécuter le thread.

Contrairement aux shader du pipeline graphique, il est possible d'échanger des données entre les threads. De plus du côté carte graphique il est possible de forcer la synchronisation localement à l'aide de $barrier()$ ou $memoryBarrier$ qui permet d'attendre que tous les thread du groupe ait terminé avant d'aller plus loin, donc de synchroniser les thread. Du fait du partage de donnés il est possible d'avoir des données lues et écrie par plusieurs threads. Si plusieurs threads écrivent en même temps il peut y avoir des conflits et le résultat peut se retrouver faux au final. Il existe alors des opérations atomiques permettant de gérer ses conflits mais ces opérations ont un effet de sequentialisation. Ces shaders ont accès aussi à une mémoire partagée qui est très rapide d'accès mais limité en taille. Afin de réaliser une synchronisation externe il faut alors utiliser les fonctions $glMemoryBarrier$

La sortie du compute shader est une image 2D qui se trouve être l'image final de la scène. Le fait d'avoir utilisé des threads en 2D proportionnellement à l'image de sortie a notamment l'avantage de ne pas avoir de conflit d'écriture en mémoire. En effet du fait de la numérotation des thread chaque thread va s'occuper d'écrire un seul et unique pixel qui sera différant des pixels des autres threads.

De plus afin de récupérer les résultats sou la forme d'une image on utilise une storage texture qu'il faut préalablement configurer.

La communication entre le CPU et le compute shader se fait à l'aide de storage buffers qui sont très proche des

Il faut bien faire attention à l'alignement des données, en effet les données envoyées au GPU avec des storage buffer doit être un multiple de 16Octets. Dans le cas contraire il risque d'y avoir des décalages sur les données récupérées sur le GPU.

Pour la suite du TP tout le code découle du fichier *tuto_raytrace_compute.cpp* et *raytrace_compute.cpp* qui sont les fichiers de correction de Mr IEHL pour le passage au compute shader.

(a)

Figure 2: Affichage de la cornell box avec l'utilisation d'un compute shader.

3 Éclairage Ambiant et Accumulation des rayons

3.1 Accumulation

Une fois le point d'intersection rayon/triangle obtenu, il est possible de générer des directions aléatoires afin d'estimer l'éclairage ambiant d'après Mont Carlo comme on a pu le voir dans le TP2. Les nombres aléatoires sont utilisé du coté carte graphique et doit être différent pour chaque pixel de l'image. L'idée est donc d'initialiser une texture buffers afin d'y stocker des nombres aléatoires pour chaque pixel que l'on pourra accéder à l'aide de $gl_GlobalInvocationID$.

Dans le principe de Mont Carlo plus on génère de direction de rayon plus l'éclairage ambiant sera uniforme et réaliste. Cependant pour pouvoir garder une exécution en temps réel il n'est pas possible d'augmenter le nombre de rayons de façon significative. L'idée est donc de cumuler les directions dans plusieurs exécutions du shader afin de converger vers un résultat cohérent.

Afin de réaliser cela il faut ainsi lire l'image résultat de l'exécution précédente et ajouter l'éclairage ambiant obtenu avec la nouvelle exécution du shader. Il est alors possible de pouvoir déterminer le résultat de l'éclairage en connaissant combien d'exécution sont cumulé. Ce nombre d'exécution cumuler ce fait à l'aide d'un uniform qui incrémente à chaque itération. De plus il faut bien réinitialiser l'accumulation si la caméra est bougée du fait qu'on lit le résultat de l'exécution précédente. Dans le cas contraire si les pixels bougent entre temps, l'image final serait un cumule de plusieurs cornell box dans des positions différentes.

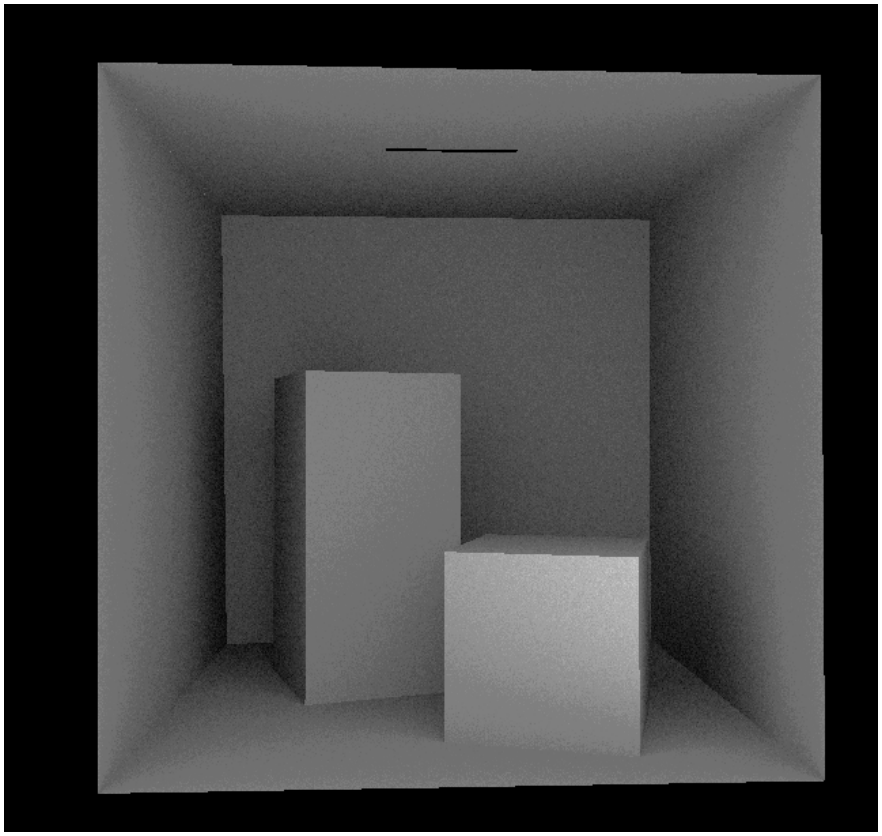
3.2 Aléatoire

Le fait d'ajouter une accumulation influe aussi sur la gestion des nombres aléatoire. En effet les nombres aléatoires sont initialisé une fois dans la texture et sont toujours identique selon l'exécution du shader. Cela a pour conséquence que l'accumulation se fait avec des directions identiques et perd tout son intérêt. L'idée est donc de générer aléatoirement un nouveau nombre aléatoire en fonction du premier pour qu'il soit diffèrent pour l'exécution suivante.

La génération du nouveau nombre aléatoire se fait à partie d'une fonction nommé "RNG" qui est un générateur linéaire qui suit une formule (1) pour obtenir une nouveau nombre aléatoire à partir du précédant.

$$X_{n+1} = (aX_n + b)\%m) \tag{1}$$

Les X_0 sont donc générer aléatoirement pour chaque pixel avec la texture de seeds initialiser avec l'outil *random_device* de c++



(a)

Figure 3: Affichage de la cornell box avec un ecalirage ambiant après quelques secondes d'accumulation.

3.3 Discussion et améliorations

La solution mis en place a été de réaliser toutes les étapes sur un même shader ce qui peut notamment créer des problèmes de cohérence, plus il y a d'étape de if/else dans le shader moins il y a de chance que l'exécution soit cohérente. De plus tout ressembler sur un seul shader va naturellement utiliser une plus grande quantité de mémoire et va réduire le nombre de groupe de thread qui peut être exécuté.

La solution opposée aurait été de réaliser chaque calcul sur un compute shader différent : génération rayon primaire, intersection rayon, génération rayons indirects, intersection rayons indirects et accumulation du résultat dans l'image. Au contraire cette solution a un bon potentiel de cohérence car chaque étape pourra utiliser un nombre approprié de thread. Cependant cette méthode est lourde car elle ferait beaucoup d'aller-retour entre le GPU et le CPU.

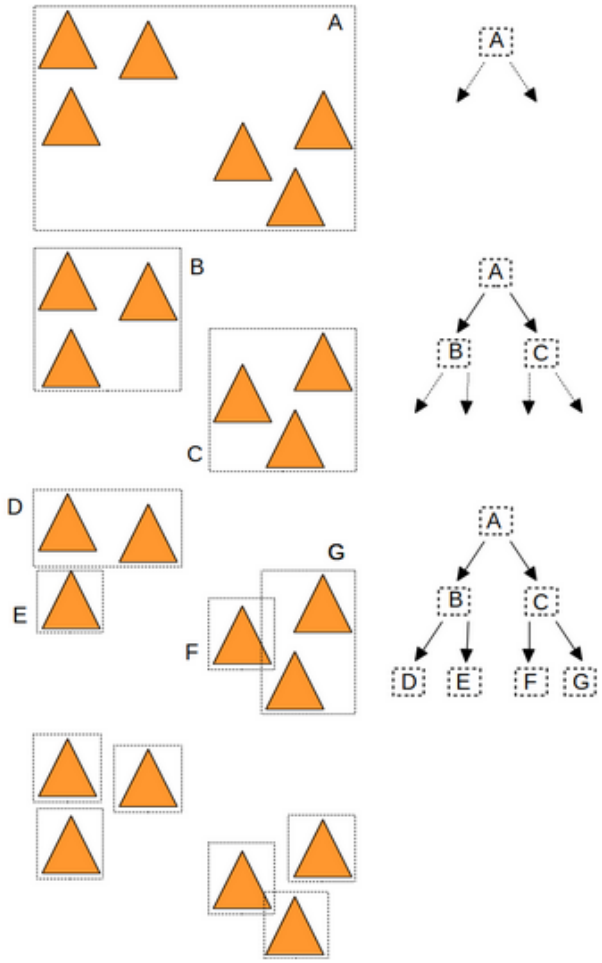
Une solution intermédiaire serait de réaliser l'intersection rayon triangle pour obtenir les points d'intersection, ensuite dans un second shader on réaliserait la génération des directions et le calcul de l'éclairage ambiant. Cette méthode pourrait permettre éviter de réserver des threads pour des pixels qui n'intercepte aucun point. Il faudrait alors utiliser autant de groupe de thread que de point intersecté en ajoutant éventuellement une composante en z (3D pour réaliser plusieurs calculs de rayons en parallèle), Il faudra alors bien faire attention aux conflits d'écriture en mémoire pour assembler le résultat de chaque rayon.

De plus cette méthode serait d'autant plus efficace avec l'accumulation, en effet avec un seul shader, le gpu calcul à chaque frame les points d'intersection rayon/triangle. Avec la deuxième solution il serait alors possible de garder le résultat des points d'intersection entre les rayons et la scène dans le cas où la caméra ne bouge pas et que l'accumulation se fait.

4 Structure accélératrice (bvh)

4.1 Principe

Pour l'instant que ce soit pour l'intersection rayon/triangle de la scène ou l'intersection rayon/triangle des rayon pour l'éclairage ambiant, le code actuel parcourt itérativement tout les triangles et test l'intersection. Dans le cas d'une scène avec des milliers de triangle il alors impossible d'avoir un affichage en temps réel. L'idée est donc d'instaurer une structure accélératrice qui consiste à créer un arbre de bounding autour des triangles de la scène. En effet une fois lancé, un rayon va rencontrer des bounding box qui englobe toute la scène et va progressivement intersecter avec des bounding box plus précise jusqu'à atteindre un triangle. La force du bvh est donc que si une bounding box n'est pas intersectée alors tous les triangles présents à l'intérieur ne seront jamais testés ce qui est un gain de temps et de calcul considérable.



(a)

Figure 4: Schéma de la création des bounding box d'un bvh.

Ainsi un bvh sera informativement une liste de nœud qui représente la bounding box mais aussi les indice des nœud suivant à parcourir dans le cas d'une intersection (fils droit et gauche dans l'arbre).

De plus une fois qu'on arrive à une feuille, on ne veut plus tester les nœuds/bounding box mais bien les

triangles directement. Afin de pas modifier la structure on choisit de mettre dans left l'indice du triangle à tester et dans right l'indice du triangle de fin à tester. Il reste cependant nécessaire de pouvoir différentier un nœud et une feuille, pour cela il est choisi de placer les indices des triangles dans les feuilles en valeur négative (tous les indices des nœuds sont logiquement positifs).

4.2 Construction

La construction de cet arbre se fait récursivement, en effet on crée le Noeud racine à partir de tous les triangles et les nœuds enfant sont créé avec une partition des triangles. Les Noeud sont créé récursivement en prenant les indices de début et de fin des triangles qui sont inclue dans la bounding box du noeud. Avec ces deux indices, le but est de créer deux nœuds enfant soit deux bounding box et 2 indices pour chaque nœud. Pour obtenir ces paramètres il est nécessaire de réaliser une partition des triangles à l'intérieur du noeud et de choisir quel triangle appartient à quelle noeud enfant. Le choix de l'appartenance d'un triangle au fil droit ou au fil gauche se fait en prenant en compte la position du centre du triangle par rapport à une séparation de la bounding box d'origine. Cette séparation de bounding box se fait suivant la moitié de l'axe de plus grand. Enfin il suffit alors de placer les triangles du premier fil au début de l'ensemble de triangles et les triangles du second fil à la fin de la liste. En connaissant l'indice qui permet de séparer les deux fils, il est alors possible d'évaluer les bounding box correspondantes aux de deux et d'obtenir les deux noeud enfant. La récursion se fait ensuite avec les indices de début et de fin de chacun des fils. La fin de la récursion arrive quand il reste moins de 2 triangles, ils sont alors stockés directement en négatif pour bien avoir une feuille.

4.3 Recursif

Ce parcourt de cet arbre de fait donc en profondeur en partant de la racine qui représente la bounding box de la scène totale. Le parcourt d'un arbre est parfaitement adapté à la récursivité car par exemple un enfant d'un noeud est lui-même un noeud. Ainsi pour chaque noeud il faut vérifier l'intersection du rayon avec la bounding box correspondante et s'il y a intersection il appelle récursivement la fonction avec les indices de nœud des fils droit et gauche. Dans le cas où le noeud tester est une feuille il faut alors tester l'interception rayon triangle mais seulement avec les triangles contenus dans la bounding box précédente.

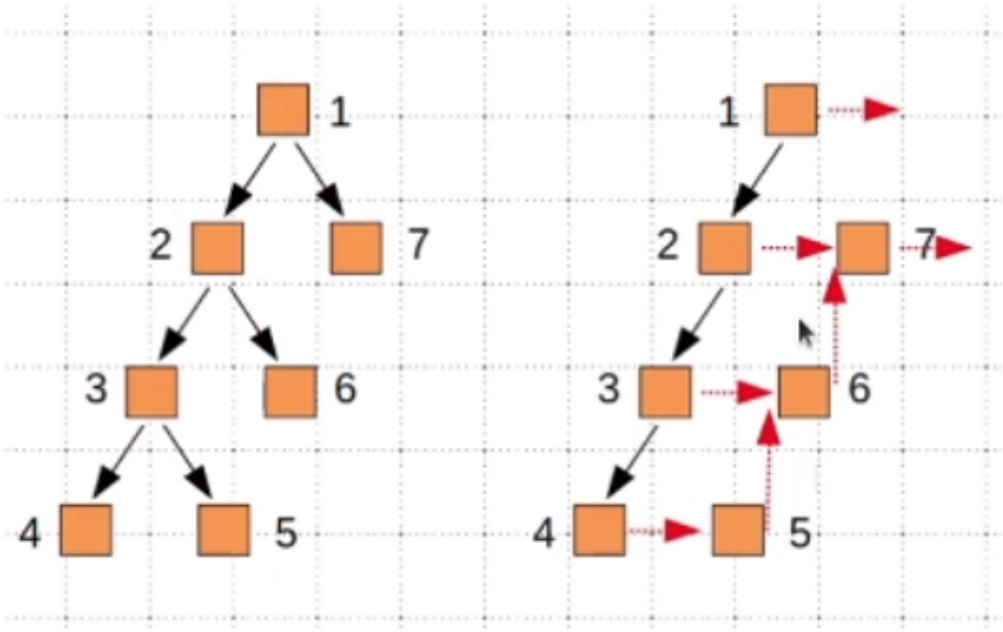
4.4 Pile

Il ne faut cependant pas oublier que le but est de réaliser l'intersection du rayon avec le bvh sur le compute shader et donc sur carte graphique. Un problème se pose donc car le shader n'a pas de pile pour pouvoir réaliser des fonctions récursives. Il faut donc adapter le parcourt pour qu'il soit réalisable sur GPU.

Pour le bon fonctionnement du parcourt du bvh su GPU il est donc nécessaire d'envoyer à la carte graphique la listes des triangles et la liste des noeud avec des storages buffers. Il est aussi nécessaire d'envoyer la racine que j'ai choisi d'envoyer par uniform. Toutes les structures respectent l'alignement des données pour envoyer seulement les informations nécessaires.

La première méthode est d'utiliser une pile pour pouvoir stocker les noeud qu'il y a à visiter. En effet cette méthode remplace l'appel récursif par des appel itératif sur une pile qui décrit l'avancement dans l'arbre du bvh. Plus précisément le principe est le même mais la différence est qu'à chaque itération le noeud parcouru et le dernier noeud inséré dans la pile. S'il y a intersection entre le rayon et le noeud au lieu d'appeler récursivement la fonction on ajoute à la pile les deux indices des fil droit et gauche pour pouvoir les parcourir ensuite. Cette itération se termine quand la pile est vide.

4.5 Cousu



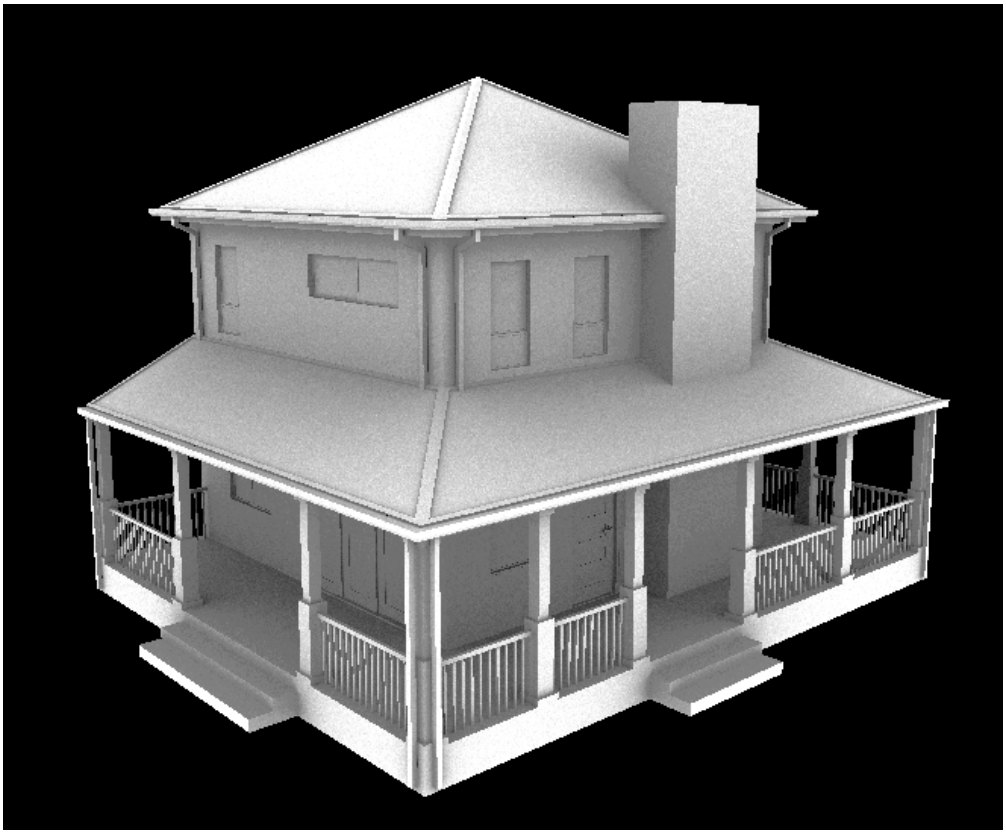
(a)

Figure 5: Affichage d'une maison avec un éclairage ambiant après quelques secondes d'accumulation.

Une autre façon de faire est de réaliser un arbre cousu. Cela a pour principe de ne plus stocker les fils droit et gauche mais seulement un fil et de stocker un noeud "skip" pour sélectionner le prochain noeud à parcourir. Le principe de parcourt de cet arbre est de tester un noeud et si le rayon intercepté ce noeud alors le prochain noeud est next, si il n'est pas intersecté alors le prochain noeud est skip comme c'est illustré sur la figure 5

Pour l'utilisation de cet arbre il est cependant nécessaire de modifier l'arbre originale et d'y ajouter les noeud next et skip. Une méthode pour transformer cet arbre est d'utiliser une méthode récursive. En effet en parcourant l'arbre en profondeur il est possible d'obtenir next qui est directement le fil gauche. Pour ce qui est de skip il y a deux cas, si le noeud est un noeud gauche alors le noeud skip est le noeud droite du parent, si c'est un noeud droit le noeud skip est le noeud skip du parent déjà obtenu par récursion.

Cette partie n'a pas été implémenté...



(a)

Figure 6: Affichage d'une maison avec un éclairage ambiant après quelques secondes d'accumulation.

4.6 Discussion et améliorations

Le tableau présentant les différent temps d'exécution su GPU pour les différentes manière est le tableau 1

Scene \ Méthodes	sans bvh	bvh pile(stack 64)	bvh pile (stack 16)	cousu
Cornell Box (32 traingles)	980ms	142ms	62ms	X
Maison (5456 triangles)	9ms	29ms	12ms	X

Table 1: Comparaison des temps d'exécutions sur GPU en fonction de la méthode choisie et le choix de la scène pour 4 rayons générés pour l'ambiant

Il est d'ores et déjà possible d'observer que, pour la scène de la maison qui possède plus de 5000 triangles, le BVH permet de gagner un temps considérable (figure 6). En effet sans bvh il n'y a même plus un affichage en temps réel.

Une autre observation est l'amélioration de temps d'exécution si la taille de la pile est réduite dans le bvh en utilisant une.

Un inconvénient de cette méthode est de nécessiter de la mémoire allouée pour la pile, cela a pour conséquence d'augmenter significativement les registres utilisés. Ainsi il est nécessaire d'avoir un nombre de sous thread plus élevé pour pouvoir continuer à masquer les accès mémoires des variables locales et garder le parallélisme. Or comme plus le nombre de sous thread nécessaire est élevé moins on peut appeler de groupe

de thread et donc moins on peut paralléliser le problème. Pour une bonne performance il faut aussi garder un nombre de groupe de thread suffisant pour masquer les accès mémoires vers la mémoire vidéo.

Dans notre cas, l'explication est que des threads sont occupé pour manipuler les accès mémoire de cette pile, c'est aussi pourquoi il y a un gain de performance si la pile est réduite. Cette perte de performance dû à la pile fait que pour une scène tel que la cornell box avec 32 triangles il est plus efficace de ne pas utiliser un bvh.

Il cependant possible d'améliorer les accès mémoire car on se rend compte que l'algorithme empile les deux fils pour dépiler à l'itération suivante. L'idée est donc de tout de suite sélectionner un des fils et d'empiler seulement le fil restant. Cette méthode permet de réduire de moitié la taille de la pile. Cette méthode utilisant une pile peut encore être améliorer en testant l'intersection des deux fils et d'empiler seulement si les deux fils sont intersectés.

Il faut cependant avoir conscience que les arbres ne sont pas idéals sur une exécution sur carte graphique du fait de l'ordonnancement des données. En effet, sur un groupe de thread, chaque thread s'exécute en même temps. Si les threads réalisent la même action tout s'exécute en même temps et tout est cohérent. Cependant si on prend l'exemple d'un if/else dont la condition n'est pas la même il y aura une exécution des threads qui vérifie la condition du if puis l'exécution des threads qui vérifie la condition du else. Pendant ces deux exécutions les threads qui ne vérifient pas les condition du if ou du else attendent ce qui représente une perte de performance.

Dans le cas des arbres les rayons qui sont générer sur chaque thread ne parcourent pas identiquement la scène, par exemple si on imagine le premier rayon qui arrive à une feuille, il est statistiquement probable que la plupart des autres rayons sont encore dans des noeuds, ainsi l'exécution du calcul d'intersection de la feuille ferait attendre tous les threads qui sont sur des noeuds.

Pour améliorer cela il est alors possible avec les fonctions *GL_ARB* par exemple de forcer d'exécuter toujours les noeuds et de faire attendre les feuilles jusqu'à attendre nombre de feuilles conséquents. Cette méthode permet de d'éviter les cas où une majorité des threads est en pause.

5 Conclusion

Ce TP a nous permettre de se familiariser avec les compute shader. Ceux-ci sont un moyen de réaliser du GPGPU et ainsi pouvoir utiliser la puissance de calcul du GPU. On arrive grâce à avoir quelques résultats de scènes conséquente avec du lancer de rayon pour l'éclairage ambiant quasiment en temps réel.