Lancer de rayons temps réel sur carte graphique

1st Mikel-ange Barros
computer science department
University Claude Bernard Lyon 1
Lyon, France
mikel-ange.barros@etu.univ-lyon1.fr

3nd Thomas Scouflaire
computer science department
University Claude Bernard Lyon 1
Lyon, France
thomas.scouflaire@etu.univ-lyon1.fr

***Abstract*—Dans ce papier, nous proposons une méthode de lancer de rayons basée sur des computes shader. Cette approche se base sur un modèle d'éclairage direct puis indirect, le tout soutenu par un bvh afin de garder de bonnes performances y compris sur de grosses scènes.**

***Index Terms*—Rendu, Lancer de rayons, bvh**

I. Introduction

Si le lancer de rayons s'est souvent retrouvé cantonné à une utilisation hors ligne, depuis opengl 4.3 et l'arrivée des computes shader il est possible de le faire en temps presque réel sur carte graphique. Cependant, les temps de calculs restaient trop lourds pour une véritable utilisation temps réel et cet article auraient été bien moins pertinent il y a quelques années. Mais, depuis la sortie de la génération 2000 des cartes graphiques nvidia, la technologie de ray tracing temps réel est devenu une réalité. Et c'est dans ce contexte que ce papier intervient.

A. Librairies utilisées

gkit2light, glew, sdl2

B. L'application

L'application finale est une scène opengl permettant l'affichage d'un maillage quelconque en calculant les interactions de lumières sur la scène à l'aide d'une méthode de lancer de rayons (figure 1).

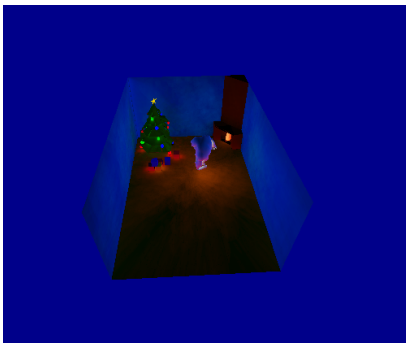


Fig. 1. Présentation de l'application

C. Lien vers le code

lancer de rayons sur carte graphique

II. Compute shader

Avant de parler de la manière dont nous avons mis en place cette méthode de lancé de rayon, il nous semble important de revenir sur l'outil qui nous a permis de le faire : le compute shader. Le compute shader est arrivé comme je l'ai indiqué plus haut avec la version opengl 4.3 et a amené de grands changements dans la manière de voir le calcul sur carte graphique. En effet, ce type de shader n'est dédié qu'aux calculs, il n'affiche rien et n'appparaît pas dans le pipeline graphique. On pourrait alors se demander, qu'est ce qui fait la force du compute shader dans les calculs par rapport à un algorithme sur processeur par exemple. La réponse à cette question est simple : la parallélisation. En effet, une carte graphique est faite pour pouvoir exécuter des millions de threads en même temps et donc d'effectuer les calculs en parrallèles bien plus rapidement qu'un processeur classique. Et c'est ce sur quoi se base notre lancé de rayon. En effet, cet algorithme est facilement parallélisable, chaque pixel de l'image de sortie étant calculé séparément et ne dépendant pas de l'image précédente ou d'un de ces voisins.

III. Première implémentation

La première approche consistait à reprendre l'implémentation utilisée sur Cpu et de l'adapter sur carte graphique. Nous procédions donc en deux étapes :

- on divise nos sources de lumière en N lumières et nous calculons l'éclairage direct pour chaque source de lumières nouvellement créée
- On calcul m directions sur une hemisphère et on calcul l'éclairage indirect pour chaque direction.

Cette approche, bien que simple à mettre en place, s'est vite avérée inutilisable pour du ray tracing temps réel. En effet, tout ces calculs étaient bien trop lourds et ralentissaient grandement l'exécution de l'algorithme et ce même sur de petites scènes. Nous avons donc rapidement abandonné cette méthode.

IV. Deuxième implémentation

Suite à l'échec de notre première implémentation, nous avons dû réfléchir à un autre moyen d'obtenir le même résultat. Et pour cela, nous nous sommes basé sur la perception humaine. En effet, l'oeil humain ne voit qu'un nombre d'images limité par seconde, et si nous pouvions créer l'image avant que l'oeil humain n'en remarque les défauts alors nous

aurions gagnés. C'est ainsi que nous est venu l'idée de calculer l'éclairage de manière incrémentale. Plutôt que de calculer l'image finale en une seule itération, à chaque itération nous allons calculer une image intermédiaire et l'accumuler avec les images précédentes. Pour cela, comme dans la méthode précédente, nous allons calculer un éclairage direct et un éclairage indirect, mais cette fois pour un seul rayon seulement. Avant de nous attarder sur les résultats, revenons rapidement sur ce que sont l'éclairage direct et l'éclairage indirect. Ainsi que sur la manière dont nous avons implémenté l'accumulation d'images.

A. Eclairage direct

L'éclairage direct consiste à calculer pour chaque point de notre objet si il peut voir la source ou non. Cela va nous servir à calculer les différentes ombres et à déterminer quels sont les endroits les plus lumineux de la scène. Le calcul pour le faire est :

$$L_{direct}(p,o) = \int L_e(s,p)V(p,s)f_r(s,p,o)\frac{\cos\theta\ \cos\theta_s}{d^2(p,s)}ds$$

.

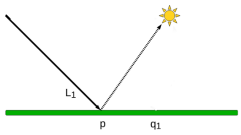


Fig. 2. éclairage direct

B. Eclairage indirect

L'éclairage indirect quant à lui consiste à calculer les rebonds.Ainsi on va considérer un rayon qui touche un point et on va regarder si en partant dans une autre direction (en faisant rebondir le rayon) on touche un autre objet. Le calcul pour le faire est :

$$L_{indirect}(p,o) = \int L_{direct}(q,p)f_r(q,p,o)\cos\theta dv \text{ avec } q = hit(p,\vec{v})$$

.

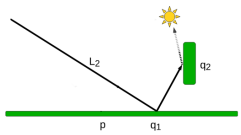


Fig. 3. éclairage indirect

C. L'accumulation d'image

Afin de pouvoir accumuler les images correctement, il nous a fallu trouver une formule permettant cette accumulation, de base, nous avions pensé à juste faire :

$$I = (Iprec + Icur)/(2);$$

.

Mais cela ne fonctionnait pas. En effet, l'influence de la deuxième image était trop importante par rapport à la première et donc nous n'obtenions pas un résultat cohérent. Nous avons donc pensé à prendre en compte l'ensemble des images précédentes et en accumulant en fonction du nombre d'images. Cela donne donc la formule :

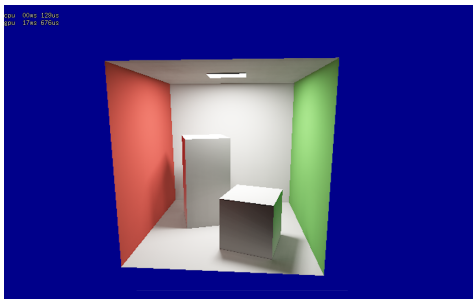
$$I = (Iprec * Acc + Icur)/(Acc + 1);$$

.

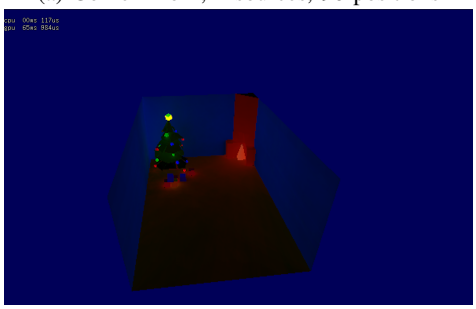
Avec I l'image en sortie, Iprec l'image précédente,Acc le nombre d'images déjà accumulée et Icur l'image à accumuler.

D. Exemples de résultats

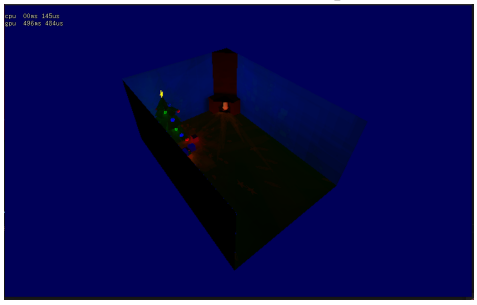
L'ensemble des tests ont été faits sur une amd vega 10 (le chipset intégré de l'un de nos processeurs).



(a) Cornell Box , 2 sources, 96 positions



(b) noel, 228 sources, 3000 positions



(c) noel, 5199 sources, 22000 positions

Fig. 4. Resultats L1+L2

Comme on peut le voir sur la figure 4, les résultats sont bons, mais le temps de calculs évolue de manière linéaire avec le nombre de positions, il est donc difficile de garder du temps réel avec cette méthode pour de grosses scènes.

Et c'est pourquoi la mise en place d'un bvh nous a semblé indispensables.

V. BVH

A. *Qu'est ce qu'un bvh?*

Un bvh est une structure de données servant à séparer les objets d'une scène en fonction de leur position dans l'espace. Cette structure de données se montre comme étant un arbre contenant des bouding box entourant les objets. sur chaque branche de l'arbre, on divise la bouding box en deux jusqu'à ce que le nombre d'objets qu'il nous reste dans chaque bouding box soit suffisamment simple à traiter.

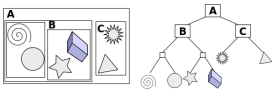


Fig. 5. bvh [WKP20]

B. *Particularité sur carte graphique*

Sur carte graphique, créer un arbre linéairement avec ces deux fils est impossible. En effet, il n'existe pas de pile récursive sur carte graphique et cela rends le parcours du bvh classique impossible. Il existe cependant une alternative à ce problème, le bvh cousu. Ce dernier, plutôt que de relier un noeud a ses deux fils, va le relier à son fils gauche et à son parent le plus proche (frère ou oncle). Cette méthode va permettre un parcours itératif de l'arbre et on va donc pouvoir l'utiliser sur carte graphique.

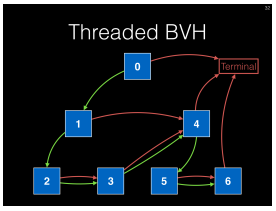
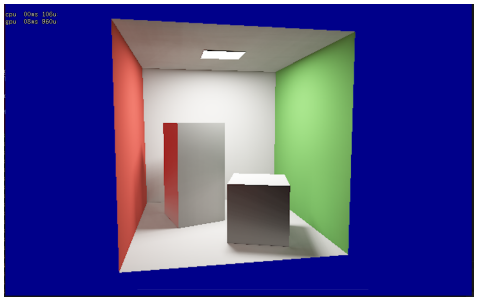


Fig. 6. bvh cousu [STO19]]

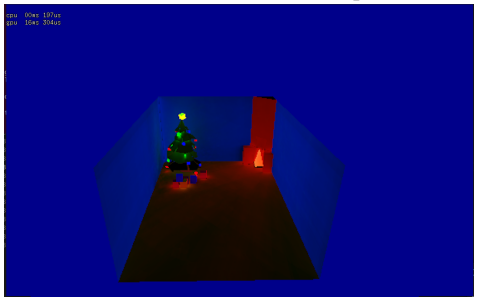
C. *Exemples de résultats*

L'ensemble des tests ont été faits sur une amd vega 10 (le chipset intégré de l'un de nos processeur).

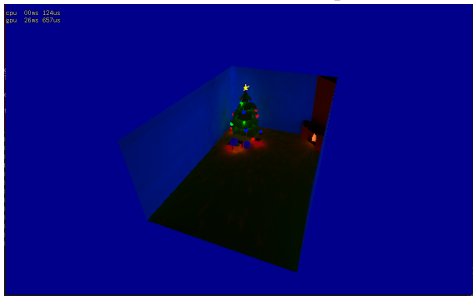
Comme on peut le remarquer, l'utilisation d'un bvh réduit drastiquement le temps de calcul nécessaire, et ce surtout sur les grosses scènes. Cela permet d'obtenir un résultat bien plus efficace et réaliste dans un temps proche du temps réel.



(a) Cornell Box , 2 sources, 96 positions



(b) noel, 228 sources, 3000 positions



(c) noel, 5199 sources, 22000 positions

Fig. 7. Resultats L1+L2

VI. Conclusion

Le raytracing temps réel est donc enfin possible et donnes des résultats bien plus réalistes que les méthodes d'affichage temps réels classiques. Cependant, il reste gourmand et sans les fonctions accélératrices de nvidia, impossible à mettre en place en condition réelle.

Reference

[WKP20] bouding volume hierarchy - wikipedia
[G2L20] gkit2light
[IEHL20] jciehl cours 2020
[STO19] Traversal of Bounding Volume Hierachy in Shaders - stackoverflow