Lancer de rayons dans un octree.

Régis Portalez, Florent Duguet

Lancer de rayon dans un octree
Utilisation de la connectivité via les coordonnées de Plücker

Portalez Régis
Altimesh

Duguet, Florent
Altimesh

2 mars 2016

Résumé

L'essentiel du temps de calcul des systèmes de tracé de rayons est passé dans le parcours de la structure accélératrice. Les facteurs limitants de ce parcours sont la latence mémoire et le branchement. Nous proposons ici un algorithme de traversée d'octree alliant les bénéfices de l'arbre et ceux de la grille régulière. Celui-ci est basé sur l'utilisation de la connectivité entre les feuilles de l'arbre. Via les coordonnées de Plücker, il est possible de l'implémenter d'une façon qui maintienne un certaine cohérence des données, tout en limitant drastiquement le nombre de branchements.

Most of the computation time in ray-tracing algorithms is spent traversing the accelerating structure. Usual bottlenecks are memory latency and branching. We present here an octree traversal algorithm which offers the best of both octrees and regular grid. This algorithm mostly uses the connectivity of the tree's leafs. Using the Plücker coordinates, it's possible to implement it in a way which keeps memory accesses coherent, while drastically limiting branching operations.

Introduction

La majorité du temps de calcul des systèmes de tracé de rayons est passé dans la recherche d'intersections de rayons lumineux avec les objets de la scène. La littérature sur le thème de l'optimisation des calculs d'intersection est riche de nombreux types de structures d'accélération, comme en témoignent les travaux de Frédéric Cazals [CDP95] ou Ingo Wald [WMG*09] sur le sujet. Nous proposons ici un algorithme de traversée d'arbre (octree) alliant les bénéfices de l'arbre, grandes cellules pour les régions vides de la scène, et ceux de la grille régulière, temps constant pour passer d'une cellule à la voisine. De plus, nous limitons fortement le nombre de branchements.

Les approches conventionnelles de traversée d'octree sont depth first ou breadth first. Nous proposons ici de ne pas utiliser la hiérarchie de l'arbre pour le traverser, mais d'utiliser des liens de face. Nous contraignons l'octree à n'avoir au maximum qu'un niveau de profondeur d'écart entre deux cellules adjacentes, de sorte que les faces n'exposent que zéro, un ou quatre voisins. La traversée de

l'octree se fait donc en recherchant la face de sortie d'une cellule, avec éventuellement une sous-face dans le cas d'un changement de niveau de subdivision.

La principale contribution de cet article est un algorithme efficace de calcul de la face de sortie. A la manière de J.Mahovsky dans [MW04], nous nous basons sur les coordonnées de Plücker. Cet algorithme permet d'identifier la face de sortie, et par extension la prochaine cellule traversée par un rayon, le tout sans division ni branchement. Nous proposons également une implémentation CUDA de ces algorithmes, dont nous comparons les performances obtenues avec celles réalisées par le code de [AL09].

1 État de l'art

L'utilisation des coordonnées de Plücker a déjà été utilisée pour optimiser le test d'intersection rayon-boîte. Nous pensons aux travaux de Mahovsky [MW04]. Cependant, nous n'avons pas trouvé de référence où cette méthode était utilisée pour traverser un octree. A notre connaissance, ce document en est la première occurrence.

La traversée d'un octree en marchant de voisins en voisins a déjà été présentée, notamment par Hana Samet [Sam89]. Cependant, dans ces travaux antérieurs, la recherche du voisin suivant utilisait la structure hiérarchique de l'arbre plutôt que la connectivité elle-même. Ceci conduit à un surcoût calculatoire et à une complexité algorithmique variable que nous évitons. En effet, cette approche serait dommageable pour la distribution du calcul sur plate-formes massivement parallèles.

A l'inverse l'utilisation à plein de la hiérarchie de l'arbre conduit aux méthodes *top-down* telles que présentées dans Revelles [RUL00]. Cette méthode a l'avantage de ne nécessiter que très peu de mémoire pour stocker l'octree, mais génère un surcoût de calculs d'intersection rayon-boîte lors du parcours de la structure.

D'autres structures comme les *Hiérarchies de Grilles Uniformes* ont été étudiées, notamment par Cazals [CDP95]. Il s'agit de grouper les objets par taille et par localité spatiale (phase de *clustering*), avant de subdiviser ces zones en grilles uniformes.

Les *kd*-tree sont également discutés dans [WMG*09]. La structure consiste essentiellement en un arbre binaire. L'espace est récursivement subdivisé par des plans parallèles aux axes. Cette structure est largement considérée comme étant une des meilleures. Toutefois, nous n'avons pas trouvé d'implémentation de référence qui nous permette de nous y comparer.

La grille régulière a également été largement étudiée, notamment par [WIK*06] ou [GIK*07]. Un de ses avantages est un temps de construction très faible, ce qui la rend très adaptée aux applications interactives. On trouvera notamment dans ces deux références un exemple ou la grille est entièrement recalculée à chaque image.

Enfin, la littérature semble aujourd'hui se concentrer sur les hiérarchies de boîtes englobantes ou *BVH*. Celles-ci offrent un temps de construction raisonnable, ce qui est intéressant pour les applications interactives. De plus, des progrès considérables ont été effectués dans les algorithmes de traversée de telles structures. Elles semblent offrir aujourd'hui les meilleures performances, comme en témoignent les travaux de Stich [SFD09], Aila [KA13] ou Gunther [GPSS07].

Notre implémentation sera testée contre cette méthode.

2 Eléments géométriques

Nous présentons ici quelques éléments d'implémentation de l'algorithme.

2.1 Rayons et coordonnées de Plücker

Une description détaillée de l'espace et des coordonnées de Plücker est disponible dans [Ave09]. Rappelons toutefois les propriétés essentielles qui nous seront utiles.

Un *rayon* sera défini par un point d'origine et un vecteur directeur. A partir de ces éléments nous calculerons l'expression de la droite orientée définie par ce rayon dans l'espace de Plücker. L'espace de Plücker est un espace projectif à 6 dimensions dont les éléments sont des 2-variétés.

Une droite orientée peut être définie par un couple de ses points (P, Q) (modulo translation et dilatation positive). Les coordonnées de Plücker de la droite sont le 6-uple :

$$\begin{aligned}
\pi_0 &= Q_x P_y - P_x Q_y \\
\pi_1 &= Q_x P_z - P_x Q_z \\
\pi_2 &= Q_x - P_x \\
\pi_3 &= Q_y P_z - P_y Q_z \\
\pi_4 &= Q_z - P_z \\
\pi_5 &= P_y - Q_y
\end{aligned} \tag{1}$$

L'opérateur *orientation* qui définit une orientation relative de deux droites : si le signe est positif, alors une droite tourne positivement autour de l'autre, au sens de l'orientation canonique de l'espace Euclidien. Si l'évaluation de l'opérateur est nulle, alors les droites sont soit en intersection, soit parallèles. La figure 1 illustre le sens de rotation d'un droite R, π par rapport à une droite D, π' :

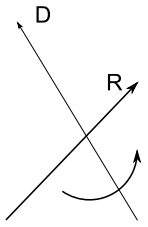


FIGURE 1 – Illustration d'une orientation positive de deux droites.

L'opérateur est évalué selon l'équation 1.

$$\begin{aligned}
\Pi(R, D) = \pi_0\pi'_4 + \pi_1\pi'_5 + \pi_2\pi'_3 \\
+ \pi_3\pi'_2 + \pi_4\pi'_0 + \pi_5\pi'_1
\end{aligned} \tag{2}$$

Ce dernier est commutatif, et est positif dans le cas illustré figure 1.

2.2 Subdivision de l'espace

Un *octree* est un arbre 8-aire. Chaque nœud représente une zone cubique de l'espace, qui est découpée en huit parties égales représentées par les enfants du nœud. La racine est la boîte englobante de la scène, dont les axes sont parallèles aux axes de coordonnées. Les feuilles de l'octree contiennent une liste d'objets (ce sera des triangles dans cet article) intersectant la boîte associée. Un nœud sera subdivisé s'il est suffisamment peu profond et s'il contient assez d'objets géométriques. Aussi, nous imposons à l'octree une contrainte de *régularité* : la différence de profondeur entre deux voisins ne peut excéder 1.

Chaque feuille a des voisins, dans les six directions de l'espace. Dans une direction, une boîte peut avoir plusieurs voisins s'ils sont à un niveau de subdivision supérieur. La contrainte de régularité sur le niveau de subdivision garantit un nombre de voisins de zéro, un ou quatre.

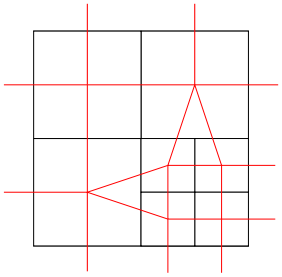


FIGURE 2 – L'octree (ici en coupe 2D) et la connectivité de ses feuilles (en rouge).

Les feuilles du bord n'ont pas de voisin dans au moins une direction de l'espace, ce sera le test de terminaison lors de la traversée d'octree.

A partir du maillage, nous construisons alors un octree et le graphe de ses feuilles.

3 Sérialisation

Étant donné un maillage, nous stockons 4 tableaux. Un pour l'octree, un pour les triangles, un pour les boîtes pointées par les cellules de l'octree et un dernier contenant les adresses des cellules au bord. Ce dernier sera utilisé dans une "cube map texture" nous permettant d'accéder à la première cellule traversée :

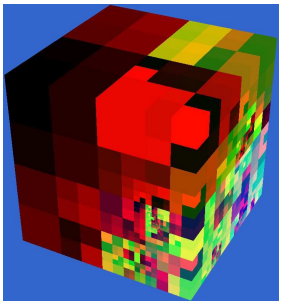


FIGURE 3 – Les adresses des cellules du bord

3.1 Octree

Une feuille de l'octree est stockée de la manière suivante :

$$\text{tCount}|\text{tOffset}|\text{i}|\text{nCount}|N_0^0|N_0^1|\dots|N_i^0|N_i^1|\dots \tag{3}$$

où :

- *tCount* est le nombre de triangles pointés par cette feuille.
- *tOffset* est l'adresse mémoire du premier triangle dans le fichier de triangles.
- i est l'index de la feuille et sera utilisé pour récupérer la boîte qu'elle représente.
- *nCount* contient le nombre de voisins dans chaque direction. Etant donnée la contrainte que nous imposons à l'octree, il ne peut pas y avoir plus de 4 voisins par direction, ce qui nous permet d'encoder ces 6 nombres sur les 24 premiers bits de *nCount*.
- N_i^j l'adresse mémoire du j^{ieme} voisin dans direction direction i.

4 Traversée de l'octree

Par construction, les droites pour lesquelles nous évaluons l'opérateur orientation avec les rayons sont toutes parallèles aux axes. La valeur de $\Pi(R, D)$ est également invariante par translation et dilatation du couple (P, Q) représentant D. Nous pouvons donc toujours nous ramener au cas où une coordonnée de P est nulle et où Q lui est distant de 1.

Par exemple, si $D = (P, Q)$ est un axe parallèle à Ox et orienté positivement, on peut se ramener à $P = (0, P_y, P_z)$ et $Q = (1, P_y, P_z)$. Les équations 1 et 2 donnent alors l'expression suivante pour évaluer l'orientation :

$$\Pi_x^P(R) = \pi_4 P_y + \pi_5 P_z + \pi_3 \tag{4}$$

où l'on définit $\Pi_x^P(R)$ comme une notation pour $\Pi(R, D)$.

De même, on obtient les équations suivantes pour les axes Oy et Oz :

$$\begin{aligned} \Pi_y^P(R) &= \pi_2 P_z - \pi_1 - \pi_4 P_x \\ \Pi_z^P(R) &= \pi_0 - \pi_2 P_y - \pi_5 P_x \end{aligned} \tag{5}$$

4.1 Calcul de la cellule d'entrée

Afin de retrouver la première cellule traversée par un rayon, nous calculons le point d'intersection du rayon avec la boîte pointée par la racine de l'octree. Ces coordonnées sont utilisées pour retrouver l'adresse de la cellule correspondante de l'octree à partir de la *cube map texture* visible figure 3.

4.2 Marche dans le graphe

Une fois que nous avons trouvé le point d'entrée du rayon dans l'octree, l'algorithme consiste à passer d'une cellule à sa voisine. Cela suppose de pouvoir calculer la direction de sortie du rayon, ainsi que l'index du voisin, s'il y en a plusieurs. Nous avons choisi la convention suivante pour la numérotation des faces et des sommets des cellules, ainsi que pour les orientations des axes de coordonnées :

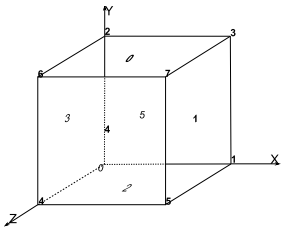
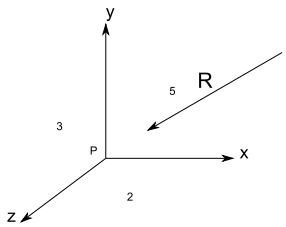


FIGURE 4 – Convention geométriques

4.2.1 Calcul de la direction de sortie

Les faces possibles de sortie d'un rayon sont restreintes par son orientation. Il n'y en a que 3 dans le cas général, voire moins pour les configurations dégénérées. Les signes de la direction du rayon selon les axes de coordonnées indiquent ces trois faces (voir [MW04] pour plus de détails).

Considérons le cas primitif suivant, qui correspond aux conventions présentées figure 4 :



Puisque le rayon entre dans la boîte délimitée par les axes (x, y, z), il nous suffit de tester les produits de Plücker avec chacun de ses axes pour savoir si le rayon sort par la face 2, 3 ou 5. Par exemple, si R passe au dessus de x il ne peut pas intersecter la face 2. Si de plus il passe à droite de y, il sort nécessairement par la face 5.

En appliquant cela à toutes les possibilités, on en déduit la table suivante :

$\Pi_x^P(R)$	$\Pi_y^P(R)$	$\Pi_z^P(R)$	résultat
0	0	0	N/A
1	0	0	2
0	1	0	5
1	1	0	2
0	0	1	3
1	0	1	3
0	1	1	5
1	1	1	N/A

On peut alors encoder les trois valeurs des produits Π ci dessus sur un seul entier, utilisé comme index dans une table d'accès. Cette table dépend de l'orientation du rayon, avec six configurations générales ; les configurations dégénérées peuvent réutiliser les mêmes tables :

$$\Pi^P(R) = \Pi_x^P(R)|\Pi_y^P(R)|\Pi_z^P(R) \tag{6}$$

$$\text{outFaceLookup} = [7, 2, 5, 2, 3, 3, 5, 7] \tag{7}$$

7 étant une valeur par défaut pour les cas inaccessibles.

Si l'on calcule la table 4.2.1 pour tous les sommets de sortie, on peut les regrouper dans une table à deux dimensions outFacesLookup :

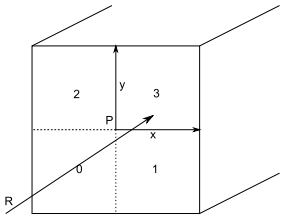
$$\begin{matrix} 7,2,5,2,3,3,5,7 \\ 5,7,1,1,5,2,7,2 \\ 3,3,7,5,0,7,0,5 \\ 0,5,0,7,7,5,1,1 \\ 2,4,2,7,7,4,3,3 \\ 1,1,7,4,2,7,2,4 \\ 4,7,3,3,4,0,7,0 \\ 7,0,4,0,1,1,4,7 \end{matrix} \tag{8}$$

Le calcul de la direction de sortie est donc un calcul qui consomme 6 multiplications, 6 additions, 5 opérations binaires, 3 décalages, et un lookup dans une table stockée en mémoire constante.

4.2.2 Calcul de l'index du voisin

Cette partie concerne les cellules disposant de plus d'un voisin du fait d'une profondeur de subdivision supérieure. Le calcul du voisin parmi les quatrees possibles est fait grâce aux coordonnées de Plücker. Considérons le cas illustré sur la figure 4.2.2 :

Comme précédemment, nous utilisons les coordonnées de Plücker pour calculer le positionnement relatif du rayon par rapport aux axes médians afin de déterminer l'index du voisin. Si le rayon passe au-dessus de x, alors les voisins 0 et 1 sont exclus. S'il passe aussi à gauche de y, alors la seule possibilité restante est 2. Ceci nous donne la table suivante :



$\Pi_x^P(R)$	$\Pi_y^P(R)$	*resultat*
0	0	2
1	0	0
0	1	3
1	1	1

Étant donné que cette table ne dépend pas de $\Pi_z^P(R)$, on peut la calculer en fonction de $\Pi^P(R)$. Si l'on refait ce calcul sur toutes les faces de la cellule, on en déduit une table à deux dimensions faceToNeighborIndex :

$$\begin{array}{c} 1,3,1,3,0,2,0,2 \\ 2,2,0,0,3,3,1,1 \\ 2,0,2,0,3,1,3,1 \\ 1,1,3,3,0,0,2,2 \\ 2,0,3,1,2,0,3,1 \\ 1,3,0,2,1,3,0,2 \end{array} \tag{9}$$

et l'index du voisin sera donc :

$$\text{neighborIndex}(R, \text{outFace}) = \text{faceToNeighborIndex}[\text{outFace}][\Pi^P(R)] \tag{10}$$

où P est le point central de la face de sortie.

4.2.3 Traversée de l'octree

Disposant des fonctions précédentes, on peut écrire l'algorithme de traversée du graphe :

```
address ← getEntryCell(octree, ray)
while true do
  triCount ← octree[address++]
  triOffset ← octree[address++]
  if intersectTriangles(ray, triangles + triOffset, triCount) then
    return HIT
  end if
  nodeIndex ← octree[address++]
  box← boxes + nodeIndex
  outDirection ← getOutDirection(box, ray)
  address ← goToDirection(octree, outDirection)
  neighborCount ← countNeighbor(octree, outDirection, address)
  if neighborCount == 0 then
```

```
        return MISS
      end if
      neighborIndex ← getNeighborIndex(octree, outDirection, box)
      address ← octree[address + (count / 4) * neighborIndex]
    end while
```

dans lequel les fonctions *goToDirection* et *countNeighbor* se contentent de parcourir la cellule pour modifier l'adresse courante, ou pour compter le nombre de voisins dans la direction considérée. A noter qu'on peut traiter de la même façon le cas où il n'y a qu'un voisin et le cas où il y en a quatre, ce qui supprime un test.

5 Performances

Nous présentons ici des résultats de performance sur des scènes de type objet scanné, pour lesquelles l'intégralité de la scène est visible. Un tel rendu est visible figure 5. La consommation mémoire, tout comme le temps de traversée sont les deux critères de performance que nous avons analysés.

5.1 Consommation mémoire

Un nœud de l'octree nécessite 4 entiers (sur 32 bits), plus autant d'entiers que de voisins. Nous avons observé un nombre moyen de 7 voisins par cellule. Cela fait donc 44 bytes pour stocker une cellule (en moyenne).

Prenons le cas des Stanford Dragon, Bunny et Armadillo. Nous avons construit des octree dont les feuilles contiennent 6 triangles en moyenne. Si l'on considère la totalité de la mémoire consommée par les quatre fichiers du maillage, on obtient la table 4, où toutes les unités sont en Mo.

5.2 Temps de calcul

Nous nous sommes focalisés sur les rayons primaires. La scène est donc uniquement constituée du maillage et le modèle de shading est flat.

Nous avons implémenté l'algorithme ci-dessus en C++ et CUDA (7.0), et l'avons exécuté sur un processeur Intel *Core i7 3770 3.4GHz* et une carte graphique NVIDIA *GTX 980*. Le compilateur utilisé pour le code C++ est celui de Visual Studio 2012 et le système d'exploitation est Windows 8.1 update dans sa version 64 bits.

Nous avons construit un octree de profondeur 11 pour le dragon, 10 pour armadillo et 8 pour le bunny. Cet octree est stocké en mémoire texture, afin de bénéficier d'un cache. Les différentes tables de lookup sont stockées en mémoire constante, tandis que les triangles et les boîtes sont également en mémoire texture.

La métrique utilisée est le nombre de rayons intersectés par seconde. Un rayon intersecté correspond à l'ensemble des calculs depuis le calcul de ses coordonnées jusqu'à la première intersection avec un objet.

Nous avons écrit tout le code dans un seul kernel, exécuté sur le GPU avec un grille de 128×128 blocs de 8×8 threads. Ces choix sont le résultat d'une phase d'analyse de performances avec l'outil de profilage. Les performances dans

différentes configurations sont mesurées en millisecondes et millions de rayons par seconde.. L'image rendue est de résolution 1024×1024.

Nous comparons notre méthode au lanceur de rayon fourni par les auteurs de [AL09], dont le code est disponible sur code.google.com. Leur implémentation repose sur une hiérarchie de boîtes englobantes (BVH).

La scène utilisée est la même dans les deux cas (même maillage et même caméra).

Les performances obtenues dans différentes configurations sont résumées table 3.

5.3 Discussions

5.3.1 Contrainte de régularité

La contrainte de régularité imposée à l'octree augmente le nombre de feuilles vides de l'octree. La quantité de mémoire nécessaire en est augmentée d'autant, de même que le nombre de cellules traversées par un rayon. Nous n'avons pas implémenté notre algorithme en relaxant cette contrainte. Cependant, nous avons compté le nombre de feuilles de l'octree dans les deux cas. Ces chiffres sont rassemblés table 1.

Maillage	non contraint	contraint	ratio
Dragon	855k	1144k	**133%**
Armadillo	367k	454k	**123%**
Bunny	69k	83k	**120%**

TABLE 1 – Nombre de feuilles de l'octree, avec ou sans contrainte de régularité.

Supprimer la contrainte de régularité diminuerait dont d'un quart le nombre de feuilles traversées par rayon. Cela signifie autant de lecture mémoire et de calculs en moins. On serait en droit d'attendre un gain de performance du même ordre de grandeur. Nous choisi de ne pas l'implémenter, et cela reste donc à faire.

5.3.2 Branchements

Notre algorithme ne nécessite que très peu de branchement (if), ce qui le rend adapté au architectures massivement parallèles comme un GPU. En fait, suite à une passe d'optimisation, il n'en contient que quatre :

- Un test d'arrêt si le rayon n'intersecte pas la boîte englobante du maillage.
- Un test d'arrêt de la boucle sur les triangles.
- Un test d'intersection avec un triangle.
- Le test de terminaison : si le nœud courant n'a pas de voisin dans la direction voulue.

Cela est dû à plusieurs facteurs :

- Grâce à la *cube map*, nous pouvons calculer l'adresse de la cellule d'entrée directement à partir des coordonnées du point d'entrée, et donc sans test.
- Grâce à Plücker, nous connaissons la direction de sortie sans test.
- De même, s'il y a plusieurs voisins, nous pouvons déterminer l'index du voisin correct sans test.

— S'il n'y a qu'un voisin, nous pouvons nous ramener au cas précédent, mais en multipliant l'offset obtenu par le nombre de voisins divisé par quatre (en division entière), ce qui fait un dans le cas de quatre voisins, et zéro sinon. Cela se paie par un calcul, mais évite un test.

La suppression de la contrainte de régularité rendrait difficile d'implémenter certaines de ces optimisations.

6 Conclusions

Notre méthode tente de réconcilier la grille uniforme et l'octree. Le temps de passage d'une cellule à sa voisine est constant et les zones vides de l'espace ne sont pas subdivisées inutilement. Le nombre de table pré-calculées est faible et n'impacte donc pas significativement l'empreinte mémoire. Celles-ci sont petites (quelques octets), ce qui permet de les stocker en mémoire constante. Les performances obtenues sont du même ordre que celles obtenues avec une traversée de *BVH*, comme on peut le voir table 3.

Il reste toutefois des points d'amélioration qui pourraient conduire à des travaux ultérieurs :

— Nous nous sommes essentiellement focalisés sur le rayon primaire, dont la cohérence spatiale est assurée par construction. Il faudrait voir comment l'algorithme se comporte pour des rayons secondaires ou diffus.

— L'adressage mémoire des cellules de l'octree pourrait être optimisé en tenant compte de la géométrie. Cela permettrait une lecture plus efficace de la mémoire texture sur le GPU.

— Nous avons relevé plusieurs points limitant les performances de notre implémentation. Premièrement, il faudrait limiter le nombre de lectures mémoire. On pourrait par exemple supprimer le stockage de la boîte associée à chaque nœud, en en recalculant les coordonnées. Deuxièmement, nous allouons 39 registres, ce qui devrait être réduit.

— Il serait intéressant de chercher à supprimer la contrainte de régularité sur l'octree et de mesurer le gain de performance éventuel.

— Nous n'avons pas cherché une méthode optimale pour construire la structure accélératrice. Il serait intéressant de chercher à la construire sur GPU, pour adapter la méthode à un environnement destructible, ou pour l'intégrer à une application interactive.

— A la manière de [AL09], il serait utile de connaître les performances théoriques de la méthode, en l'exécutant dans un simulateur, à la latence nulle et bande passante infinie.

Références

[AL09] Aila T., Laine S. : Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009* (2009), ACM, pp. 145–149.

[Ave09] Aveneau L. : Les coordonnées de plücker revisitées. *Revue Electronique Francophone d'Informatique Graphique. Vol. 3*, Num. 2 (2009).

[CDP95] Cazals F., Drettakis G., Puech C. : Filtering, clustering and hierarchy construction : a new solution for ray-tracing complex scenes. In *Computer graphics forum* (1995), vol. 14, Wiley Online Library, pp. 371–382.

[GIK*07] Gribble C. P., Ize T., Kensler A., Wald I., Parker S. G. : A coherent grid traversal approach to visualizing particle-based simulation data. *Visualization and Computer Graphics, IEEE Transactions on. Vol. 13*, Num. 4 (2007), 758–768.

[GPSS07] Gunther J., Popov S., Seidel H.-P., Slusallek P. : Realtime ray tracing on gpu with bvh-based packet traversal. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on* (2007), IEEE, pp. 113–118.

[KA13] Karras T., Aila T. : Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), ACM, pp. 89–99.

[MW04] Mahovsky J., Wyvill B. : Fast ray-axis aligned bounding box overlap tests with plucker coordinates. *Journal of Graphics Tools. Vol. 9*, Num. 1 (2004), 35–46.

[RUL00] Revelles J., Urena C., Lastra M. : An efficient parametric algorithm for octree traversal. In *WSCG* (2000).

[Sam89] Samet H. : Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics. Vol. 13*, Num. 4 (1989), 445 – 460.

[SFD09] Stich M., Friedrich H., Dietrich A. : Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 7–13.

[WIK*06] Wald I., Ize T., Kensler A., Knoll A., Parker S. G. : Ray tracing animated scenes using coherent grid traversal. In *ACM Transactions on Graphics (TOG)* (2006), vol. 25, ACM, pp. 485–493.

[WMG*09] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P. : State of the art in ray tracing animated scenes. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 1691–1722.

	Bunny 	Armadillo 	Dragon 
triangles	69k	346k	871k
Notre méthode	**700/1.5**	**570/1.8**	**530/2.0**
fermi speculative while while	780/1.35	690/1.5	660/1.6
kepler dynamic fetch	**850/1.25**	**760/1.35**	**730/1.4**
persistent packet	560/1.9	440/2.4	310/3.3
persistent while while	560/1.9	530/2.0	460/2.3
ratio	**82%-125%**	**75%-130%**	**73%-171%**

«««< .mine

TABLE 2 – Performances sur NVIDIA *GTX 980*, exprimées en millions de rayons par secondes / millisecondes par image (résolution 1024x1024). Les autres méthodes sont celles de Aila [AL09]. La dernière ligne correspond aux ratio de performance entre notre méthode et la plus et la moins performante des méthodes.

=======

TABLE 3 – Performances sur NVIDIA *GTX 980*, exprimées en millions de rayons par secondes / millisecondes par image (résolution 1024x1024). Les autres méthodes sont celles de Aila [AL09]. La dernière ligne correspond aux ratio de performance entre Notre méthode et la plus et la moins performante des méthodes.

»»»> .r3036

Maillage	*cubemap*	octree	triangles	boîtes	total
Bunny	1.5	3.6	12.2	1.3	**18.6**
Armadillo	6	19	57	7	**89**
Dragon	24	50	156	18	**248**

TABLE 4 – Consommation mémoire des différentes structures, en Mo

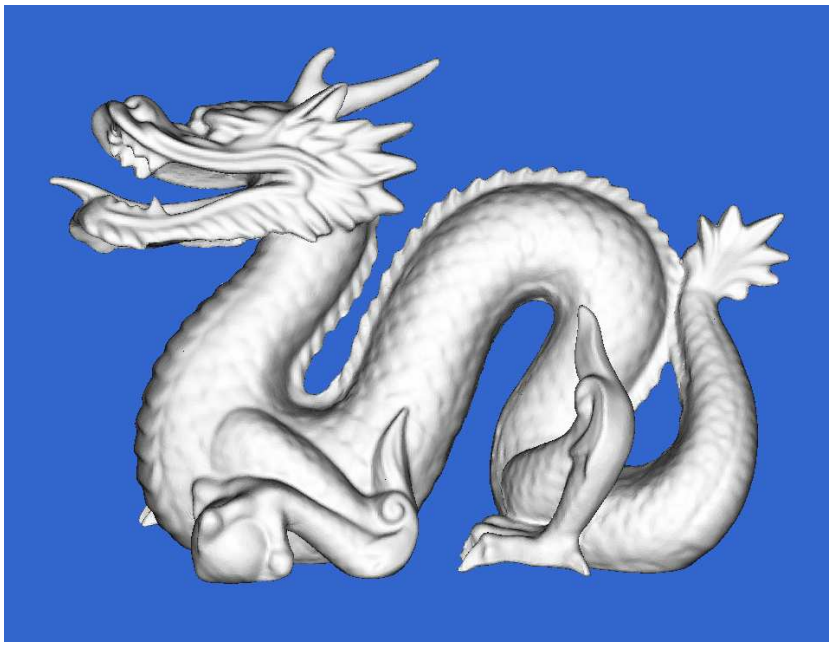


FIGURE 5 – Stanford dragon

FIGURE 6 – Stanford dragon. La couleur est donnée par le nombre de cellules traversées.