Le kd-Tree : une méthode de subdivision spatiale

Présentation de Master 2 Recherche Informatique - Module CTR
Université de Rennes 1 - INSA de Rennes

Cédric Fleury

19 Décembre 2007

1 Introduction

Le kd-Tree est une structure permettant d'organiser des données présentes dans un espace à k-dimensions selon leur répartition spatiale. Cette structure est très utile pour de nombreuses applications, comme par exemple, pour accélérer la recherche de données dans un espace multi-dimensions, la recherche d'intervalles, ou encore la recherche de plus proches voisins. Pour commencer, dans la première partie de cet exposé, nous définirons le kd-Tree.

Puis, comme dans l'article de Wald et Havran [6], nous nous intéresserons plus en détail à l'utilisation du kd-Tree dans le cas du lancer de rayon. Le lancer de rayon est une technique de rendu en synthèse d'image simulant le parcours inverse de la lumière, de l'œil de l'observateur vers la scène 3D. Il utilise une structuration spatiale de la scène à synthétiser afin de faciliter le calcul des intersections entre les rayons lancés et les objets de la scène. Parmi les différentes structures permettant de subdiviser l'espace de la scène, le kd-Tree semble être une des meilleures solutions au niveau du compromis entre la complexité de construction et la rapidité de parcours.

Nous verrons, dans une deuxième partie, différentes méthodes de construction d'un kd-Tree, ainsi que leur complexité. Puis, dans la troisième partie, nous traiterons l'intégration d'un kd-Tree dans un algorithme de lancer de rayon. Enfin, pour finir, nous ferons un rapide panorama des autres applications utilisant un kd-Tree.

2 Définition d'un Kd-Tree

Le kd-Tree, abréviation pour « *k-dimensional tree* », est un partitionnement spatial de l'espace à k-dimensions permettant de structurer les données en fonction de leur répartition dans l'espace.

Le kd-Tree est un cas particulier des « *Binary Space Partitionning (BSP) trees* ». Les BSP trees subdivisent l'espace à k-dimensions en coupant chaque volume englobant en deux sous-volumes par un plan de l'espace, et en re-itérant récursivement sur ces deux nouveaux volumes ainsi obtenus. Les BSP trees peuvent donc être représentés par un arbre binaire où les deux sous-volumes sont les deux fils du nœud correspondant au volume englobant de niveau supérieur (cf. figure n° 1). Les plans de coupe peuvent être choisis en fonction de la répartition des données, afin qu'il y ait des volumes englobants de grande taille, là où il n'y a pas une grande concentration de données et inversement.

Dans le cas du kd-Tree, ces plans séparateurs sont toujours choisis de telle façon que leur normal soit un des axes du système de coordonnées de l'espace (plans toujours perpendiculaires aux axes comme dans la figure n° 1). Cela permet de simplifier la construction, mais aussi le parcours de l'arbre.

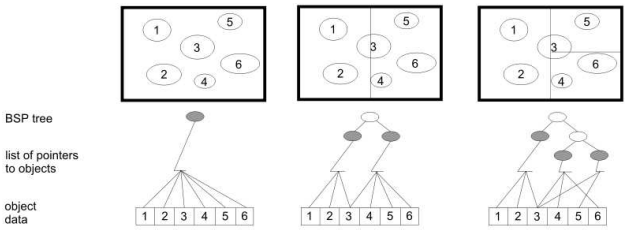

FIG. 1 – Exemple de BSP Tree dans un espace 2D extrait de la thèse d'Havran [2].

Le rôle du kd-Tree est double : il permet, d'une part, d'avoir une subdivision spatiale optimisée de l'espace permettant d'accélérer le traitement des données et, d'autre part, de stocker les données sous la forme d'un arbre binaire.

Dans le cas du lancer de rayon, la racine du kd-Tree représente le volume aligné sur les axes (AABB[1]) englobant toute la scène 3D à synthétiser. Puis, chaque nœud de l'arbre correspond à un volume AABB de l'espace (ou voxel) et contient les caractéristiques du plan séparateur qui permet de le diviser de façon optimale en deux sous-voxels, qui seront ces deux fils. Enfin, les feuilles de l'arbre contiennent la liste des objets de la scène présents dans le volume englobant qui leur correspond.

[1] Axis-aligned bounding box

3 Construction d'un Kd-Tree

Dans la littérature, il existe de nombreuses méthodes différentes pour construire un kd-Tree. Après avoir présenté l'algorithme général de la construction d'un kd-Tree, nous verrons quelles sont les variantes de ces différentes méthodes. Nous commencerons par voir dans un premier temps les méthodes les plus simples afin de bien comprendre le mécanisme de construction, ainsi que les critères importants pour l'optimisation. Puis dans un deuxième temps, nous étudierons la méthode proposée par Wald et Havran [6]. Enfin, nous finirons par comparer les complexités de ces différentes méthodes.

Pour la suite, on considérera une scène 3D à synthétiser, notée $\mathcal{S}$ et composée de N triangles (facettes des objets de la scène). Pour chaque nœud correspondant à un voxel V, le plan séparateur sera caractérisé par l'équation :

$$p_{n,\xi} : x_n = \xi \quad avec \quad n \in \{0, ..., k-1\},$$

où n est l'indice de l'axe normal au plan permettant de déterminer son orientation et ξ la valeur de la $n^{ième}$ coordonnée (x_n) de l'espace à k-dimensions permettant de déterminer la position du plan. Enfin, les feuilles contiendront la liste T des triangles appartenant à leur voxel V.

3.1 Algorithme général

Comme le présentent Wald et Havran [6], dans presque tous les cas, les algorithmes de constructions de kd-Tree suivent le même schéma récursif :

Algorithme 1 : Construction récursive d'un kd-Tree

fonction recConst(triangles T,voxel V) **retourne** Nœud
  Si (*terminer*(T,V))
    Alors retourner nouveau Feuille(T)

  p = *trouverPlan*(T,V) *trouver le bon plan de coupe*
  (V_G,V_D) = *couper* V *avec* p
  Répartir les triangles dans les sous-voxels
  $T_G = \{t \in T | (t \cap V_G) \neq \varnothing\}$
  $T_D = \{t \in T | (t \cap V_D) \neq \varnothing\}$

  retourner nouveau Nœud(p, *Plan*
    recConst(T_G,V_G),*fils G*
    recConst(T_D,V_D))*fils D*

fonction constKdTree(triangles T) **retourne** Racine
  V = *boîteEnglob*(T) *commencer avec tte la scène*
  retourner recConst(T,V)

Cependant, la structure de l'arbre construit (c'est-à-dire où se situe les plans séparateurs et quand les voxels sont créés) influence directement le nombre de voxels à traverser et le nombre d'intersections avec les triangles à calculer lors du lancer de rayon. Ainsi, la rapidité de parcours de l'arbre lors du lancer de rayon peut aller du simple au double, voire même plus, selon la façon dont l'arbre a été construit. Dans l'algorithme précédent, on peut distinguer deux paramètres différents qui influencent directement les performances du kd-Tree :

- le choix des plans séparateurs (fonction « *trouverPlan* »), c'est-à-dire à la fois le choix de leur direction (choix de l'axe normal au plan) et le choix de leur position dans le voxel (choix de la valeur de ξ). Ce choix détermine donc comment les voxels vont être coupés.
- le choix du critère de fin de l'algorithme (fonction « *terminer* »), qui détermine directement ou indirectement la profondeur de l'arbre et la taille des plus petits voxels.

3.2 Méthodes basiques

Pour la suite, on notera $D(V)$ la profondeur courante de la subdivision, qui correspond aussi à la profondeur courante dans l'arbre binaire.

3.2.1 Construction grâce à des plans médians

Tout d'abord, pour choisir l'orientation du plan séparateur, la solution la plus simple est de choisir comme axe normal au plan chacun des axes du système de l'espace à tour de rôle. Puis, la méthode plus basique pour choisir la position du plan séparateur est de choisir le plan médian qui divise le voxel courant en deux sous-voxels de taille identique (cf. figure n° 2). Ainsi, pour définir le plan séparateur $p_{n,\xi}$, on pourra prendre n et ξ tel que :

$$n = D(V) \; modulo \; k \quad et \quad \xi = \frac{1}{2}(V_{min,n} + V_{max,n}),$$

où $V_{min,n}$ et $V_{max,n}$ sont, respectivement, les valeurs minimum et maximum de la $n^{ième}$ coordonnées dans le voxel courant V.

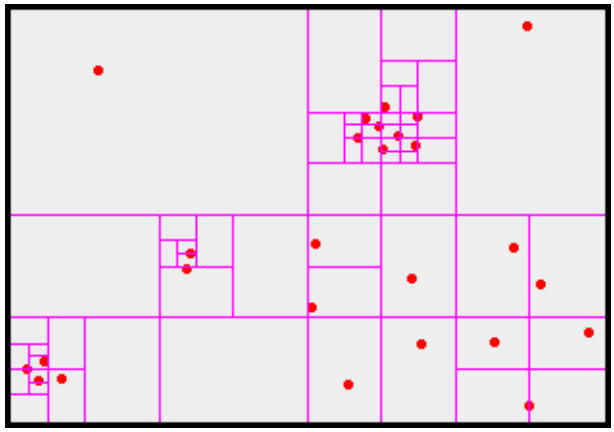

FIG. 2 – Exemple de kd-Tree construit en utilisant des plans médians de l'espace.

3.2.2 Construction grâce à des plans positionnés sur l'objet « médian »

Comme précédemment, l'axe normal au plan séparateur sera choisi à tour de rôle en fonction de la profondeur courante dans l'arbre. Par contre, au lieu de couper les voxels au milieu, cette méthode consiste à couper les voxels en fonction de la répartition des objets à l'intérieur de ces derniers. En effet, comme dans [7], on va choisir comme position du plan la position de l'objet « médian », c'est-à-dire l'objet se trouvant au centre de la liste des objets présents dans le voxel lorsque cette liste est triée en fonction de la $n^{ième}$ coordonnées, qui correspond à l'axe normal choisi (cf figure n° 3). Ce type de positionnement du plan médian assure donc qu'il y ait autant d'objets (plus ou moins un) dans chacun des deux sous-voxels ainsi constitués.

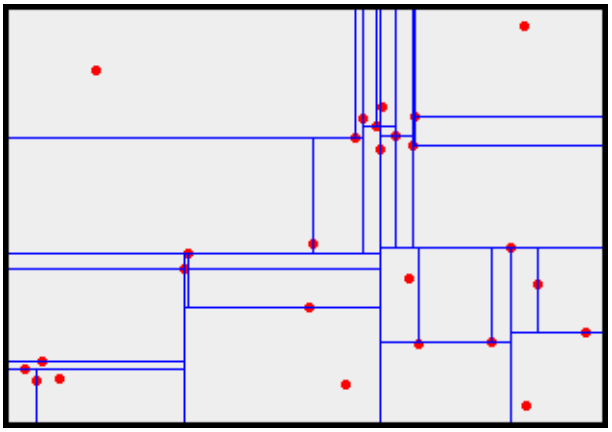

FIG. 3 – Exemple de kd-Tree construit en utilisant des plans positionnés sur l'objet « médian » de chaque voxel.

3.2.3 Fin de l'algorithme

Pour terminer simplement l'algorithme, il existe deux solutions qui sont généralement combinées. la subdivision s'arrête lorsque :

- le nombre de triangles présents dans le voxel est inférieur à un certain seuil prédéfini $\mathcal{K}_{minTri}$.
- le nombre de subdivision attend un certain seuil prédéfini $\mathcal{K}_{maxProf}$. Ce nombre correspond aussi au nombre d'itérations maximum de l'algorithme, ainsi qu'à la profondeur maximun de l'arbre binaire.

La fonction « *terminer* » peut donc s'écrire tout simplement :

$$terminer(T, V) = |T| \leqslant \mathcal{K}_{minTri} \vee D(V) \geqslant \mathcal{K}_{maxProf}.$$

3.3 Méthode utilisant la « Surface Area Heuristic » (SAH)

Les méthodes basiques précédentes ne sont pas optimales. En particulier, elles ne permettent pas de maximiser les espaces vides et de les placer le plus proche possible de la racine de l'arbre, ce qui permettraient d'avoir de meilleurs performances lors du parcours de l'arbre pour le lancer de rayon. Wald et Havran [6] propose d'utiliser la « Surface Area Heuristic » (SAH) pour trouver le plan séparateur, mais aussi pour terminer l'algorithme.

3.3.1 Principes de la SAH

En se basant sur certaines hypothèses, la SAH permet d'estimer à priori le coût de parcours de l'arbre lors de sa construction. Cela permet de choisir, pendant la construction, les plans séparateurs et les critères de fin qui minimiseront ce coût. Pour commencer, la SAH fait les hypothèses suivantes :

- Les rayons lancés lors de la synthèse d'image sont uniformément réparties et sont des lignes infinies.
- Les coûts de traversée d'un voxel $\mathcal{K}_t$ et de calcul d'une intersection avec un triangle $\mathcal{K}_i$ sont connus.
- Le coût du calcul de l'intersection d'un rayon avec N triangles est approximativement $N\mathcal{K}_i$, ce qui équivaut à dire qu'il est directement proportionnel au nombre de triangles.

Etant donné que la distribution des rayons lancés est uniforme, on peut définir la probabilité conditionnelle $\mathcal{P}$, qu'un rayon intersecte un sous-voxel $V_{sous} \in V$ sachant qu'il intersecte déjà le voxel V de niveau supérieur, de la manière suivante :

$$\mathcal{P}_{[V_{sous}|V]} = \frac{\mathcal{SA}(V_{sous})}{\mathcal{SA}(V)}$$

où $\mathcal{SA}(V)$ est l'aire du voxel V (d'où le nom de « Surface Area Heuristic »). Le coût de parcours estimé $\mathcal{C}_V(p)$ pour le voxel V et pour le plan séparateur p est égal à une fois le coût de traversée $\mathcal{K}_t$ plus les coûts estimés des deux fils du voxel :

$$\mathcal{C}_V(p) = \mathcal{K}_t + \mathcal{P}_{[V_g|V]}\mathcal{C}_{V_g} + \mathcal{P}_{[V_d|V]}\mathcal{C}_{V_d}$$

Cette équation permet donc de calculer le coût de parcours d'un kd-Tree en entier. En effet, il suffit de l'initialiser avec toute la scène $\mathcal{S}$, sachant que la probabilité qu'un rayon intersecte le volume englobant AABB de toute la scène ($V_{\mathcal{S}}$) est de un. Puis, on itère récursivement afin de calculer le coût de parcours des fils jusqu'à arriver aux feuilles de l'arbre, où le coût est égal à $|T|\mathcal{K}_i$ avec $|T|$ le nombre de triangles présents dans le voxel correspondant à la feuille.

3.3.2 Construction grâce à la SAH

Théoriquement, pour trouver le meilleur kd-Tree d'une scène donnée, il faudrait calculer tous les arbres possibles en estimant à chaque fois leur coût de parcours, grâce à l'équation précédente, afin de les comparer entre eux. Cependant, le nombre d'arbres possibles augmente très rapidement avec la taille de la scène et, de nos jours, trouver l'arbre globalement optimal est infaisable sauf pour les scènes triviales, comme le disent Wald et Havran [6]. Ils proposent donc d'utiliser une approximation locale en considérant à chaque pas de l'itération les deux fils comme des feuilles :

$$\begin{aligned} \mathcal{C}_V(p) &\approx \mathcal{K}_t + \mathcal{P}_{[V_g|V]}|T_g|\mathcal{K}_i + \mathcal{P}_{[V_d|V]}|T_d|\mathcal{K}_i \\ &= \mathcal{K}_t + \mathcal{K}_i \left(\frac{\mathcal{SA}(V_g)}{\mathcal{SA}(V)}|T_g| + \frac{\mathcal{SA}(V_d)}{\mathcal{SA}(V)}|T_d| \right). \end{aligned}$$

Cette approximation est une grosse simplification, qui surestime le coût correct. En réalité, les listes T_g et T_d risquent être subdivisées dans les itérations suivantes et donc le coût réel de parcours des deux fils risque d'être largement inférieur à celui calculé. Malgré cela, Wald et Havran [6] affirment qu'en pratique, cette approximation marche bien et qu'aucune approximation beaucoup plus performante ne peut être trouvée.

Pour trouver un bon plan de coupe du voxel V, il faudra donc chercher le plan p qui minimise le coût $\mathcal{C}_V(p)$ ainsi approximé (cf. figure n° 4).

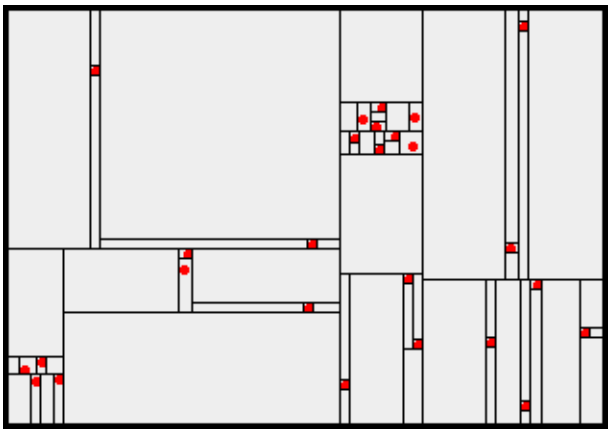

FIG. 4 – Exemple de kd-Tree construit en utilisant l'estimation du coût de parcours grâce à la SAH.

3.3.3 Fin automatique de l'algorithme

Par ailleurs, la SAH offre également une façon élégante et stable de déterminer quand arrêter l'algorithme. En effet, pour un voxel V contenant les triangles T, il suffit de comparer le coût d'une feuille $\mathcal{C}_{feuille} = |T|\mathcal{K}_i$ avec le coût $\mathcal{C}_V(p)$ obtenu si l'on divise le voxel V avec le meilleur plan séparateur p, afin de déterminer si c'est « rentable » de subdiviser le voxel ou non. La fonction « *terminer* » peut donc être exprimée ainsi :

$$terminer(T, V) = \begin{cases} vrai & ; \quad min_p \mathcal{C}_V(p) > |T|\mathcal{K}_i \\ faux & ; \quad sinon \end{cases}$$

3.3.4 Implémentations et Améliorations

En pratique, les hypothèses utilisées pour la SAH ne sont pas toujours exactes. Par exemple, la distribution des rayons lancés n'est pas souvent uniforme, les rayons ne passe généralement pas au travers des voxels contenant beaucoup de triangles, un coût de traversée d'un voxel constant n'est pas forcement très réaliste, etc. C'est pourquoi, certaines petites modifications permettent d'améliorer les performances du kd-Tree construit.

Avantager les voxels vides Comme nous l'avons vu précédemment, au regard des performances, il est préférable d'avoir des voxels vides (ne contenant pas de triangles) de taille maximale. C'est pourquoi, Wald et Havran [6] proposent de favoriser les plans qui séparent un voxel de telle manière que l'un des deux fils soit un voxel vide. Pour cela, ils utilisent un facteur de pondération qui permettra de biaiser la fonction du coût du voxel courant et ainsi de réduire le coût estimé lorsque le plan choisi coupe le voxel en deux sous-voxels dont l'un est vide (cf figure n° 5). Le coût $\mathcal{C}_V(p)$ sera donc multiplier par le coefficient suivant :

$$\lambda(p) = \begin{cases} 80\% & ; \quad |T_g| = 0 \vee |T_d| = 0 \\ 1 & ; \quad sinon \end{cases}$$

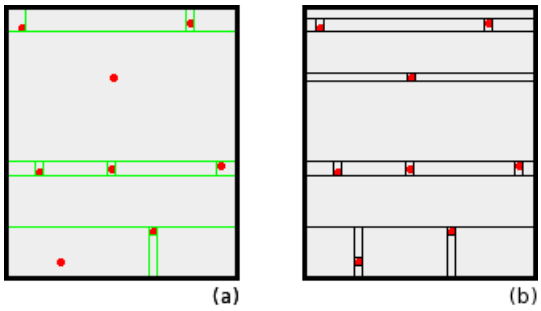

FIG. 5 – (a) kd-Tree construit sans avantager les voxels vides. (b) kd-Tree construit en pondérant de 80% le coût des voxels vides.

Terminer l'algorithme Le critère de fin tel que présenté précédemment est un peu radical et il se peut que l'algorithme se termine sur un minimum local. C'est pourquoi, il peut être intéressant de continuer à itérer quelques itérations supplémentaires même si le critère de fin indique qu'il faudrait arrêter. Par ailleurs, certaines implémentations utilisent à la fois le critère de fin basé sur le coût et celui basé sur une profondeur maximum de l'arbre, afin de réduire la mémoire nécessaire.

Rechercher le bon plan de coupe Nous avons vu que le meilleur plan de coupe p est celui qui minimise la fonction de coût $\mathcal{C}_V(p)$ pour un voxel V. Cependant, la recherche de ce plan séparateur n'est pas si simple que ça. En effet, il y a une infinité de plans qui peuvent être des plans de coupe potentiels. On ne peut pas calculer la fonction de coût pour chacun de ces plans et il faut donc une approche plus constructive.

Wald et Havran [6] proposent donc de commencer par trouver un nombre fini de plans « candidats », puis de calculer leur fonction de coût. Bien que la fonction de coût soit une fonction continue sur l'ensemble des plans du voxel à diviser (et cela pour chacun des axes de l'espace), Havran explique, dans sa thèse [2], qu'il est possible de trouver des points discrets pour simplifier la recherche. En effet, comme le montre la figure n° 6 pour les plans normaux à l'axe X, ce sont les positions des frontières des objets qui jouent un rôle clé pour trouver le plan séparateur avec un coût minimum. Il en va de même dans le cas des autres axes.

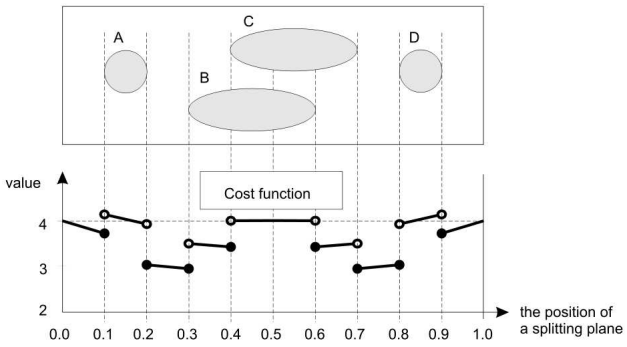

FIG. 6 – Valeur de la fonction de coût pour le voxel contenant les 4 objets selon la position du plan séparateur sur l'axe X.

La solution pour choisir les plans « candidats » est donc de choisir les six plans de la boite englobante AABB de chacun des triangles contenues dans le voxel à diviser. Il faut faire attention au cas où les triangles « dépassent » du voxel. Si le nombre de triangles est de $|T|$ dans le voxel, on trouve ainsi un ensemble d'environ $6.|T|$ plans « candidats » pour lesquels on calculera à chaque fois la fonction de coût, afin de trouver lequel d'entre eux sera le meilleur plan de coupe.

3.4 Complexité de construction

La méthode la plus simple de construction par plan médian a une complexité de $O(N\ log\ N)$ où N est le nombre total de triangles dans la scène $\mathcal{S}$. Cette complexité est, en particulier, due au fait que tous les triangles doivent être répartis dans les deux sous-voxels à chacune des itérations et que la complexité pour chaque itération est alors de $O(|T|)$. Cette phase de trie est nécessaire quelque soit la méthode de construction, c'est pourquoi, Wald et Havran [6] affirment que, théoriquement, la complexité minimale pour construire un kd-Tree est de $O(N\ log\ N)$.

Par ailleurs, les méthodes plus complexes comme celle utilisant la SAH peuvent entrainer un complexité plus importante si l'on ne fait pas attention à la manière dont on construit le kd-Tree. Par exemple, si on construit « naïvement »[2] un kd-Tree grâce à la SAH, la construction peut atteindre une complexité de $O(N^2)$. Un algorithme avec une telle complexité est difficilement exploitable. C'est pour cette raison que Wald et Havran [6] proposent un algorithme de construction utilisant la SAH dont la complexité descend à $O(NlogN)$. Cet algorithme utilise un trie astucieux des triangles qui évite d'utiliser la fonction de répartition des triangles à chaque fois que l'on veut calculer les nombres de triangles dans les sous-voxels.

4 Intégration d'un Kd-Tree dans un algorithme de lancer de rayon

Après un petit rappel sur le lancer de rayon, nous présenterons comment l'on peut utiliser un kd-Tree pour accélérer la synthèse d'image par lancer de rayon. Puis, nous verrons une méthode classiquement utilisée pour parcourir un kd-Tree.

4.1 Petit rappel sur le lancer de rayon

Le lancer de rayon est une technique de synthèse d'image qui consiste à effectuer le cheminement inverse de la lumière à partir de l'observateur vers la scène 3D, afin de déterminer la couleur de chacun des points de l'image à synthétiser. Pour lancer des rayons dans la scène 3D, on place devant l'observateur une image virtuelle (qui sera celle que l'on va synthétiser). Puis, on lance des rayons partant de l'observateur et passant par chacun des pixels de l'image virtuelle afin de déterminer leur correspondance dans la scène 3D (cf. figure n° 7).

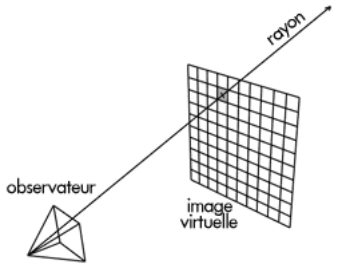

FIG. 7 – Lancer d'un rayon partant de l'observateur à travers un pixel de l'image virtuelle.

[2] « naïvement » : c'est-à-dire, si on utilise la fonction de répartition des triangles en $O(|T|)$ pour calculer le nombre de triangles par sous-voxels lorsqu'on calcule la fonction de coût de chacun des $6.|T|$ plans « candidats »

Une fois le premier point d'intersection du rayon avec la scène trouvé, il faut calculer la couleur de ce point afin de déterminer la couleur du pixel de l'image virtuelle correspondante. Pour cela, on va relancer différents rayons dans la scène afin de déterminer quelles sources de lumière éclairent le point, quels autres objets de la scène se réfléchissent ou se réfractent en ce point. Pour les nouveaux points d'intersection avec d'autres objets de la scène ainsi trouvés, on peut ré-itérer le même processus plusieurs fois selon la qualité du rendu souhaitée (cf figure n° 8).

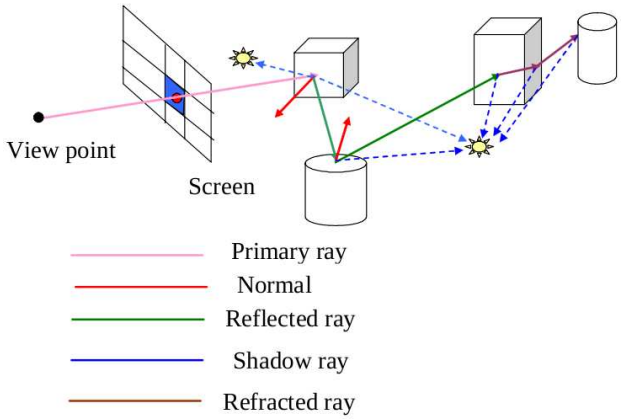

FIG. 8 – Schéma récapitulatif du lancer de rayon extrait de [1]

On pourra donc distinguer deux types de rayon :
- ceux dont l'origine est à l'extérieur du volume englobant la scène, comme les rayons partant de l'observateur.
- ceux dont l'origine est à l'intérieur du volume englobant la scène, comme les rayons de $2^{ème}$ ordre partant d'un objet intersecté.

4.2 Utilisation du kd-Tree dans le lancer de rayon

Une des tâches primordiales du lancer de rayons est donc de calculer les points d'intersection entre les rayons et la scène 3D. Un kd-Tree va permettre d'accélérer ce traitement. En effet, au lieu de tester les intersections entre un rayon et tous les objets (ou triangles) de la scène, on va pouvoir tester les intersections uniquement avec les objets contenus dans les voxels traversés par le rayon. Il va donc falloir parcourir le kd-Tree de voxel en voxel en testant si il y a intersection ou non avec les objets du voxel traversé s'il n'est pas vide. Plus le rayon traverse de voxels vides, moins il y a d'intersection avec des objets à calculer, ce qui permet de diminuer le temps nécessaire pour trouver le point d'intersection du rayon avec la scène. C'est pourquoi il était intéressant de maximiser la taille des voxels vides lors de la construction du kd-Tree.

4.3 Parcours d'un kd-Tree

Dans sa thèse [2], Havran présente plusieurs algorithmes de parcours de kd-Tree pour le lancer de rayon. Un algorithme de parcours d'un kd-Tree dans le cas du lancer de rayon prend en entrée le kd-Tree et le rayon, et rend en sortie le premier point d'intersection du rayon avec la scène.

L'algorithme récursif, que nous allons voir, est un de ceux qui sont les plus fréquemment utilisés. Le modèle du coût de parcours d'un kd-Tree utilisé pour calculer la fonction de coût lors de la construction par SAH suppose que l'on utilise ce type d'algorithme de parcours.

Comme l'explique Havran[2], cet algorithme récursif parcourt exactement une fois tous les nœuds et toutes les feuilles dont le voxel correspondant est traversé par le rayon . Quand le rayon entre dans un nœud du kd-Tree qui a deux fils, l'algorithme décide si les deux fils sont traversés et dans quel ordre. Il détermine, parmi les deux fils du nœud traversé, le nœud le plus porche (« *near* ») et le plus éloigné (« *far* »), en fonction de la position de l'origine du rayon par rapport au plan séparateur. Lorsque le rayon traverse uniquement l'un des deux fils, l'algorithme descend dans ce nouveau nœud et réitère. Lorsque le rayon traverse les deux fils, l'algorithme sauvegarde les informations du fils « *far* », descend dans le fils « *near* » et réitère. Si aucune intersection n'est trouvé à l'intérieur de ce fils « *near* », l'algorithme va alors dans le fils « *far* » et réitère.

Pour l'implémentation, on utilise les valeurs t_{min} et t_{max} qui correspondent aux distances d'entrée et de sortie du rayon dans le nœuds visité par rapport à l'origine du rayon. On calcule également, à chaque itération, la distance t_{split} qui correspond à la distance entre le point d'intersection du rayon avec le plan séparateur et l'origine du rayon (cf. figure n° 9).

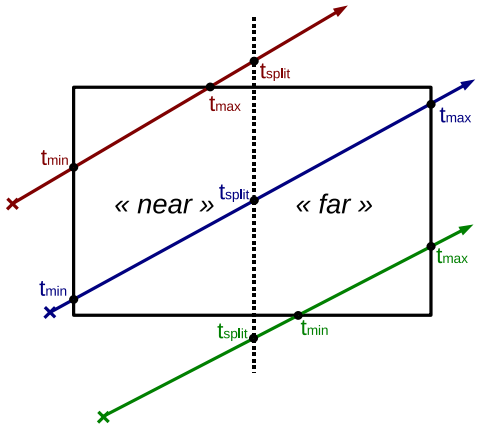

FIG. 9 – Différents cas d'intersection possible d'un rayon avec un voxel correspondant à un nœud interne du kd-Tree.

Ces trois valeurs permettent alors de déterminer quels

fils sont traversés par le rayon et dans quel ordre :

- si $t_{split} > t_{max}$, alors le rayon traverse seulement le fils « *near* ».
- si $t_{split} < t_{min}$, alors le rayon traverse seulement le fils « *far* ».
- si $t_{min} < t_{split} < t_{max}$, alors le rayon traverse en premier le fils « *near* », puis le fils « *far* ».

Enfin, la meilleure implémentation utilise une pile pour stocker le fils « *far* » dans le cas où les deux fils doivent être visités, afin d'éviter la récursivité. Cet algorithme de parcours d'un kd-Tree pourra alors s'écrire ainsi :

Algorithme 2 : Parcours d'un kd-Tree

```
Structure ElementPile
    Nœud noeud
    float t_min \\distance d'entrée du rayon dans le nœud
    float t_max \\distance à laquelle le rayon sort du nœud

fonction Parcours(Nœud racine, Rayon r)
                              retourne TriangleIntersecté

    Si(r intersecte la boîte englobante de scène)
      \\trouver le point d'entrée et de sortie du rayon
      Alors (t_min, t_max) = intersectRayBoite(racine,r)
      Sinon retourner {Pas de triangle intersecté}

    Pile.push(racine, t_min, t_max)
    Tant que(Pile n'est pas vide)
      (ndCourant, t_min, t_max) = Pile.pop()

      Tant que(ndCourant n'est pas une Feuille)
        \\trouver l'orientation du rayon par
        \\rapport au plan
        sens = orientRayPlan(ndCourant.plan,r)
        Si(sens == GAUCHEADROITE)
          Alors ndNear = ndCourant.filsGauche
                ndFar = ndCourant.filsDroit
          Sinon ndNear = ndCourant.filsDroit
                ndFar = ndCourant.filsGauche

        \\trouver le point d'intersection plan-rayon
        t_split = intersectRayPlan(ndCourant.plan,r)
        Si(t_split > t_max)
          Alors ndCourant = ndNear
          Sinon Si(t_split > t_max)
            Alors ndCourant = ndFar
            Sinon \\Si le rayon traverse les 2 fils
                  Pile.push(ndFar, t_split, t_max)
                  ndCourant = ndNear
                  t_max = t_split
      Fin Tantque

      \\Quand le ndCourant est une Feuille
      Pour tout tri ∈ ndCourant.Triangles
          Si(r intersecte t entre t_min et t_max )
            Alors ListeTriIntersect.ajouter(tri)
      Fin Pourtout

      Si(ListeTriIntersect n'est pas vide)
          Alors retourner{le triangle le plus proche de
                          l'origine du rayon ∈ ListeIntersect}
      Fin Tantque

      \\Pile est vide, pas intersection trouvée
      retourner {Pas de triangle intersecté}
Fin Parcours
```

Cette version simple de l'algorithme ne traite pas exactement tous les cas qu'on pourrait rencontrer et il peut subsister certains problèmes. Par exemple, le rayon peut être parallèle au plan séparateur ou l'origine du rayon peut se trouver à l'intérieur du nœud traité. Pour résoudre ces petites imperfections, Havran [2] propose une version un peu améliorée de l'algorithme dans laquelle les différentes positions et orientations du rayon par rapport au plan séparateur sont mieux répertoriés et donc mieux traités.

5 Autres applications

Dans le cas du lancer de rayon, nous avons vu qu'un kd-Tree permettait de structurer l'espace à trois dimensions afin d'accélérer le calcul des intersections entre les rayons et la scène à synthétiser. Si on reste dans le domaine de l'informatique graphique, le kd-Tree est généralement utilisé pour structurer un espace ne dépassant pas trois dimensions, comme nous verrons que c'est également le cas pour la détection de collisions. Mais si on s'éloigne un peu de l'informatique graphique pour aller vers les domaines du traitement d'image ou de la classification de données, les kd-Trees seront alors utilisées pour comparer, classer, structurer ou rechercher des données multi-critères dans des espaces, qui pourront alors dépasser largement les trois dimensions.

5.1 Détection de collisions

La façon la plus simple de réaliser la détection de collisions dans une scène 3D est de tester, pour chaque objet de la scène, s'il intersecte ou non un autre objet de cette scène. Comme pour le lancer de rayon, où il n'était pas envisageable de tester l'intersection d'un rayon avec tous les objets à cause d'un coût de calcul trop élevé, il faut trouver une structure accélératrice pour résoudre le problème. Le kd-Tree peut ici, aussi, être utilisé pour structurer l'espace 3D afin de permettre de calculer les intersections entre les objets en mouvement et les autres objets de la scène (cf. figure n° 10).

Cependant, l'utilisation d'un kd-Tree n'est pas forcément une solution générique pour la détection de collisions. Dans certains cas d'utilisation, par exemple, s'il y a plusieurs objets à avoir bougés dans la scène et qu'on veut également calculer les collisions entre ces objets en mouvement, il va être nécessaire de recalculer ou d'au moins mettre à jour le kd-Tree avec

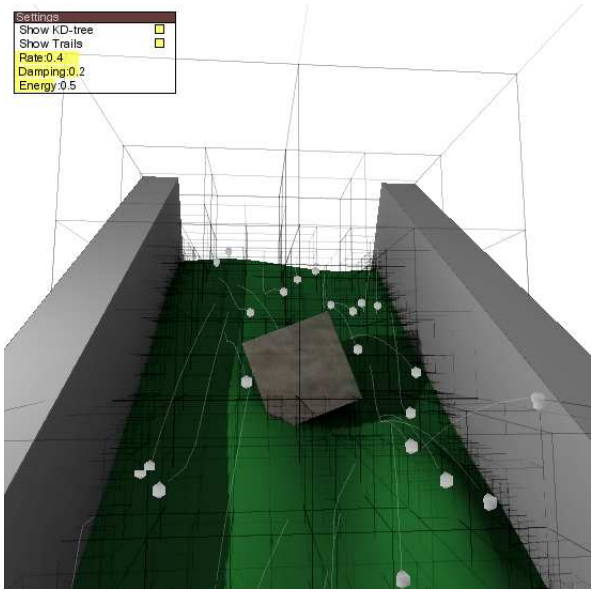

FIG. 10 – Un exemple d'application extraite de [3], où la détection de collisions est gérée grâce à un kd-Tree.

cette nouvelle disposition de la scène. Cela demande un temps de calcul assez important, ce qui ralentit l'algorithme de détection de collisions. Néanmoins, dans les cas relativement simples, comme l'application extraite de [3] où on cherche uniquement à détecter les collisions entre les « balles » blanches et la scène, qui est fixe (cf. figure n° 10), l'utilisation d'un kd-Tree offre de très bonnes performances.

5.2 Comparer des données

Les kd-Trees peuvent être utilisés pour comparer des données entre elles. En effet, après avoir contruit le kd-Tree correspondant à chacune des données, il est possible de les comparer entre elles, en comparant la structure de chacun de leur kd-Tree (Où sont situés les plans de coupe ? Où se trouvent les voxels de grandes tailles, ceux de petites tailles ? etc). Cette méthode permet d'avoir une approche plus structurelle des données décomposées. Pour l'instant, elle est plutôt utilisée dans le domaine du traitement d'image. Par exemple, Kubica et coll. [4] proposent d'indexer des images satellites d'astéroïdes en comparant leur transformation en kd-tree.

5.3 Organiser des données multi-dimensions

Les kd-Trees peuvent aussi être utilisés pour classer, structurer ou rechercher des données multi-dimensions, c'est-à-dire des données qui peuvent être caractérisées par un nombre plus ou moins important de paramètres. Généralement, l'ensemble des paramètres propres à un objet est représenté par un vecteur dans l'espace des paramètres. Ainsi, dans le cas du lancer de rayon, nos objets étaient des données qui pouvaient être caractérisées dans l'espace 3D par les trois paramètres X, Y et Z, représentés par un vecteur 3D : (X, Y, Z).

En particulier, les kd-Trees sont utilisés pour stocker des données multi-média en les structurant selon leurs caractéristiques propres. Il sera possible par la suite d'effectuer différentes recherches spécifiques grâce au kd-Tree en utilisant uniquement certains critères. Des données ayant des caractéristiques proches, auront une localisation proche dans l'espace et auront donc de forte chance d'être dans le même voxel ou dans un voxel voisin du kd-Tree. Cela facilitera leur traitement ainsi que leur recherche.

Par exemple, Reiss et coll. [5] proposent d'utiliser un kd-Tree pour structurer de la musique afin d'en extraire de l'information (« Music Information Retrieval »). La musique est décomposée en petits morceaux de trois secondes. Chacun des ces petits morceaux est à son tour décomposé en un vecteur de n caractéristiques grâce à des techniques de traitement du signal. Ces morceaux peuvent donc être classé dans un kd-Tree à n dimensions. Reiss et coll. [5] définissent alors le problème d'extraction d'information musicale comme un problème de « de recherche de trajectoire à n dimensions » pour lequel le kd-Tree leur semble être la structure la plus efficace.

6 Mise en œuvre

J'ai effectué une mise en œuvre en java des différentes méthodes de construction d'un kd-Tree. Pour cela, j'ai implémenté une classe abstraite qui met en œuvre l'algorithme général de construction d'un kd-Tree sans implémenter les fonctions « *terminer* » et « *trouverPlan* ». Puis, j'ai implémenté trois classes, correspondant aux trois méthodes de construction qui héritent de cette classe abstraite et qui implémentent les deux fonctions manquantes.

Ma mise en œuvre permet donc de tester pour le même ensemble d'objets, les trois méthodes de construction d'un kd-Tree, c'est-à-dire :

- la construction par plan médian.
- la construction par objet médian.
- la construction par SAH.

Pour cette dernière, j'ai utilisé les coûts de traversée et d'intersection proposés par Wald et Havran [6] ($\mathcal{K}_t = 15$ et $\mathcal{K}_i = 20$). Par ailleurs, il est aussi possible de paramétrer la pondération des voxels vides (cf. 3.3.4 paragraphe « *Avantager les voxels vides* »). C'est un paramètre intéressant à faire varier, étant donné qu'on obtient des kd-Trees très différents dés qu'on le modifie.

Enfin, j'ai réalisé une petite interface graphique dont la figure n° 11 donne un rapide aperçu.

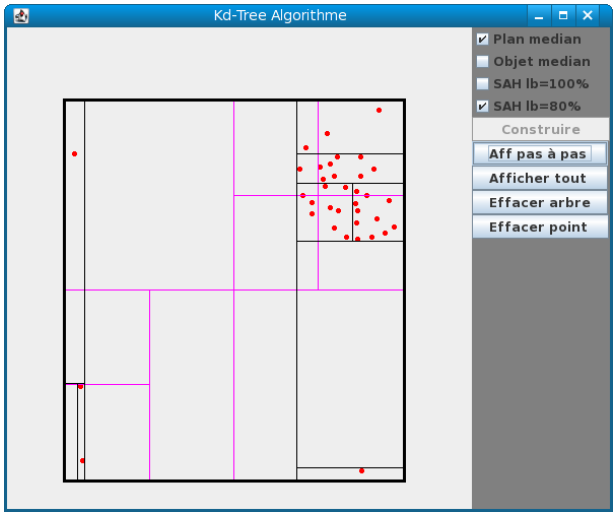

FIG. 11 – Un rapide aperçu de l'interface graphique de la mise en œuvre.

7 Conclusion

Le kd-Tree est à la fois une structuration spatiale de l'espace à k-dimensions permettant d'accélérer le traitement des données et un arbre binaire permettant de stocker ces données.

Nous avons vu qu'il existe plusieurs méthodes pour la construction d'un kd-Tree. Certaines, plus complexes, offrent de meilleures performances, mais imposent une mise en œuvre plus lourde, ainsi qu'une complexité de construction qui peut être plus importante. Ces méthodes plus complexes entraînent donc un temps de calcul plus long et une occupation de la mémoire plus importante pour construire et stocker l'arbre. Il faudra donc choisir la bonne méthode de construction en fonction des besoins de l'application dans laquelle on veut utiliser un kd-Tree.

Le kd-Tree est classiquement très utilisé dans les algorithmes de lancer de rayon pour accélérer le calcul des intersections entre les rayons et les objets de la scène 3D. Il offre de très bonnes performances dans ce domaine, mais il peut aussi être utilisé dans certains cas pour d'autres applications d'informatique graphique, comme la détection de collision. Par ailleurs, il commence à être aussi utilisé pour le stockage et la recherche de données multi-dimensions et en particulier, pour les données multi-média. Il est intéressant dans ce domaine, car il permet, de manière performante, une classification et une recherche multi-critères de ces données en fonction de leurs différents paramètres ou de leurs caractéristiques extraites.

Références

[1] K. Bouatouch. Cours : Ray tracing. `http://www.irisa.fr/prive/kadi/Cours_LR2V/RayTracing.pdf`.

[2] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Nov 2000.

[3] M. Kozak. Collision, Oct 2007. `http://www.screamyguy.net/collision/`.

[4] J. Kubica, A. Moore, A. Connolly, and R. Jedicke. A multiple tree algorithm for the efficient association of asteroid observations. In *The Eleventh ACM SIGKDD : International Conference on Knowledge Discovery and Data Mining*, pages 138–146. ACM Press, August 2005.

[5] J. Reiss, J.-J. Aucouturier, and M. Sandler. Efficient multidimensional searching routines for music information retrieval. In J. S. Downie and D. Bainbridge, editors, *the Second Annual ISMIR : International Symposium on Music Information Retrieval*, pages 163–171, 2001.

[6] I. Wald and V. Havran. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[7] Wikipédia, Nov 2007. `http://en.wikipedia.org/wiki/Kd-tree`.