# Volumes d'ombre en rendu temps réel : Complexité géométrique et stratégie de partitionnement

François Deves

**UNIVERSITÉ DE LIMOGES**

ÉCOLE DOCTORALE Sciences et Ingénierie pour l'Information

FACULTÉ DES SCIENCES ET TECHNIQUES



Année : 2019 Thèse N°X


# Thèse

pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITÉ DE LIMOGES**

**Discipline : Informatique Graphique**


**François Deves**


# Volumes d'ombre en rendu temps réel : complexité géométrique et stratégie de partitionnement

**Thèse dirigée par Frédéric Mora et Djamchid Ghazanfarpour**

**JURY :**

| | | |
|---|---|---|
| **Prénom NOM** | Professeur, Université de xxxxxxxx | Président |
| **Kadi BOUATOUCH** | Professeur, Université de Rennes | Rapporteur |
| **Mateu SBERT** | Professeur, Université de Gérone | Rapporteur |
| **Djamchid GHAZANFARPOUR** | Professeur | Examinateur |
| **Frédéric MORA** | Maitre de conférence | Examinateur |

# Table des matières

# Table des figures

# Liste des tableaux

# Introduction générale

L'informatique graphique est un domaine relativement récent mais qui a connu un essor très rapide, stimulé par l'évolution des capacités de calcul des ordinateurs dont la puissance toujours plus grande génère sans cesse de nouvelles possibilités et de nouveaux défis. À la fin des années 60 et au début des années 70 apparaissent les premiers algorithmes permettant l'affichage d'un modèle en trois dimensions sur un écran en deux dimensions. C'était logiquement le premier problème de "rendu" à résoudre. À ce moment là, il n'est question que d'affichage, mais déjà la question des ombres se posait car elles sont nécessaires au cerveau humain pour inférer une image 3D à partir de sa projection 2D. Ainsi les premiers algorithmes calculant des ombres sont publiés quasiment simultanément, du milieu à la fin des années 70. Dès cette époque, deux familles de méthodes apparaissent : Les méthodes qui opèrent dans l'espace image et celles qui opèrent dans l'espace objet. Chaque famille possède son algorithme fondateur : les cartes d'ombre ou *shadow maps* dans le premier cas, les volumes d'ombre ou *shadow volumes* dans le second. Toutes les techniques d'ombrage actuelles en temps réel héritent de ces deux approches.

Les travaux présentés dans cette thèse s'inscrivent dans la seconde catégorie, celle des volumes d'ombre. Les algorithmes de cette catégorie sont en général moins rapides que ceux basés sur les cartes d'ombres. Mais ils sont exacts quand les cartes d'ombres posent de nombreux problèmes et artefacts. De ce fait, les volumes d'ombre ont toujours conservé l'intérêt de la communauté scientifique. Ces dernières années, plusieurs travaux ont montrer que des algorithmes basés sur des volumes d'ombres pouvaient être portés à un très haut niveau de performance. Toutefois, ces performances se dégradent rapidement lorsque la complexité géométrique augmente. Car par nature, utiliser des volumes d'ombre induit une dépendance linéaire à la complexité géométrique.

## Contribution

Cette thèse comprend deux contributions principales appliquées au rendu d'ombres dures en temps réel. La première contribution apporte une solution au manque de robustesse des méthodes basées sur les volumes d'ombre face à l'augmentation de la complexité géométrique. La seconde contribution est plus exploratoire et porte sur la stratégie de partitionnement utilisée pour représenter et organiser les volumes d'ombre dans une structure de données. Plus précisément, les contributions de cette thèse peuvent être résumées comme suit :

- Tout d'abord, nous proposons un algorithme basé sur les volumes d'ombre qui conserve des performances temps-réelles grâce à l'utilisation de *clusters* y compris lorsque la complexité géométrique augmente fortement. Nous étudions et évaluons plusieurs options et optimisations, notamment pour le pré-calcul des clusters et son incidence sur le caractère conservatif des volumes d'ombre générés. De plus nous proposons une nouvelle structure basée sur deux niveau de clusters dont la construction peut être réalisée plus efficacement sur GPU. Nous proposons une comparaison de notre approche à deux méthodes issues de l'état de l'art. Nous les confrontons à des scènes allant de 1.8 à plus de 73 millions de triangles.

- Les algorithmes récents basés sur les volumes d'ombre reposent le plus souvent sur des structures accélératrice qui sont le produit d'un partitionnement de l'espace ou des objets par des plans. L'algorithme proposé dans notre première contribution

ne fait pas exception. Dans un second temps, nous explorons donc une stratégie de partitionnement différente reposant sur les arbres métriques. L'originalité de cette approche est qu'à notre connaissance, les arbres métriques n'ont jamais été utilisés en rendu. Nous montrons donc comment les exploiter pour partitionner autrement les volumes d'ombre de la scène. Nous montrons qu'il est ainsi possible d'obtenir une méthode compétitive dont la nature différente permet d'envisager des perspectives nouvelles.

## Présentation du mémoire

Dans le premier chapitre, nous revenons sur les différentes méthodes qui permettent de calculer des ombres dures en temps-réel. Nous distinguons deux familles de méthodes qui se différencient selon le repère dans lequel sont effectués les calculs. Les *shadow maps* sont appliquées dans le repère image, ce qui leur permet d'obtenir de très bonnes performances mais induit de nombreux problèmes d'aliassage difficiles à corriger entièrement. Les volumes d'ombres sont eux appliqués dans le repère objet. Ils garantissent un résultat exacte par pixel mais leur performances sont impactées par une dépendance linéaire à la complexité géométrique de la scène. Nous présentons ensuite les méthodes plus récentes héritées de ces deux familles. Nous insistons plus particulièrement sur les méthodes qui héritent des volumes d'ombre puisque c'est également dans ce cadre que s'inscrivent les travaux présentés dans ce mémoire.

Dans le second chapitre, nous présentons un algorithme basé volumes d'ombre qui est robuste à la complexité géométrique. Nous partons de la méthode proposée par Gerhards *et al.* [39] qui construit une structure organisant les volumes d'ombres entre eux, suivi d'un parcours de cette structure pour calculer la visibilité de la lumière en chaque point de l'image. Nous examinons les complexités algorithmiques de ces étapes pour expliquer que le temps de calcul finit toujours par être dominé par la construction à cause de sa complexité en $O(n\,log(n))$ pour $n$ triangles quand son parcours est lui logarithmique. Ainsi, nous proposons de construire la structure non pas sur des triangles mais sur des groupes (*clusters*) de triangles préalablement formés pour contenir l'augmentation du temps de construction. Nous étudions la meilleure approche pour former des clusters adaptés à notre contexte. Nous détaillons ensuite comment construire et parcourir la structure de données sur les clusters. Nous proposons également une seconde structure doublement hiérarchique plus adaptée au GPU. Enfin, nous présentons plusieurs séries de résultats pour valider les différentes options étudiées. Nous évaluons ensuite les performances de notre approche en les comparant à d'autres méthodes de l'état de l'art dédiées au calcul exact d'ombres dures en temps réel.

Dans le 3$^{\text{ème}}$ chapitre, nous étudions une alternative à la stratégie de partitionnement utilisée dans le chapitre 2. Nous nous intéressons alors à une approche pour laquelle nous n'avons pas connaissance d'utilisation en rendu : Les arbres métriques. Les arbres métriques permettent de partitionner les éléments en fonction de leur distance les uns par rapport aux autres. Nous commençons par rappeler les principales définitions pour cette structure. Puis nous l'étendons pour supporter des objets volumétriques en général et plus particulièrement dans notre cas, pour supporter des volumes d'ombre. Ensuite nous expliquons comment définir une distance appropriée entre deux volumes d'ombres.

Nous présentons alors les algorithmes de construction et de parcours qui peuvent s'appliquer aussi bien sur des triangles individuels que sur des clusters de triangles. Enfin nous évaluons l'algorithme obtenu en le comparant à la méthode présentée dans le chapitre 2. Nous incluons également une comparaison des ces méthodes à un rendu des ombres par lancer de rayons.

# Chapitre 1 :

# État de l'Art

## 1.1 Ombre : Définition et enjeux

Une ombre est une zone sombre de l'espace résultant de l'obstruction (partielle ou complète) par un occulteur de la lumière émise par une source lumineuse. Les ombres sont des indices primordiaux pour le système visuel humain. Elles permettent d'appréhender le positionnement relatif des objets dans l'espace, donnent des informations sur leur forme, sur l'emplacement des sources de lumière, etc... Ainsi les ombres sont indispensables pour permettre la bonne interprétation d'une image par le cerveau. C'est pourquoi elles sont nécessaires au réalisme des images de synthèse.

Le rendu d'ombre est très vite devenu une problématique de premier plan pour la recherche en Informatique Graphique. À la fin des années 60, les premiers algorithmes [6, 13] pour le rendu de scènes tridimensionnelles font leur apparition. À ce moment là, l'objectif est essentiellement de projeter un modèle en 3 dimensions sur un écran en 2 dimensions en fonction d'un point d'observation donné. Mais pour que ce rendu soit intelligible, les chercheurs ont pris conscience du caractère indispensable des ombres pour discerner correctement le positionnement relatif des objets. Aussi les premiers algorithmes permettant l'affichage d'une scène été accompagnés des premiers algorithmes permettant de l'ombrer.

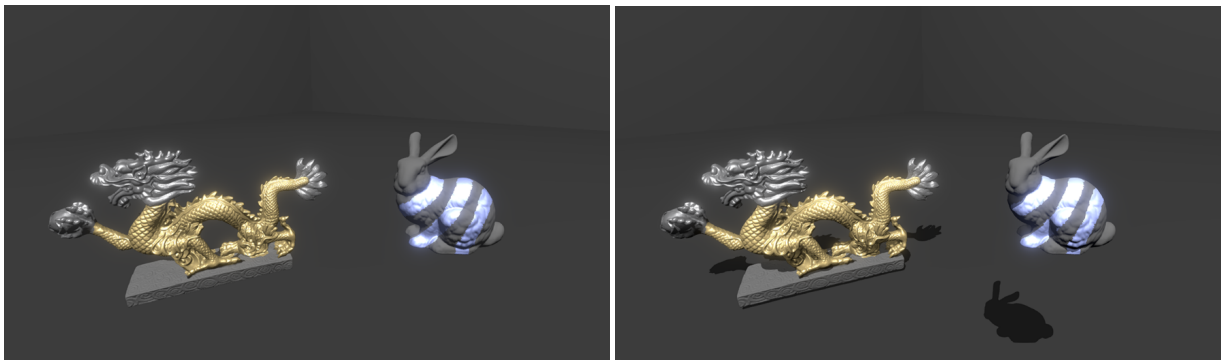

FIGURE 1.1 – Illustration de l'importance des ombres en synthèse d'images. À gauche, les deux objets semblent être situés au même niveau par rapport au sol. À droite, la présence d'ombres nous révèle la position exacte des objets.

### Le calcul d'ombre dure en synthèse d'images et en temps-réel

Les ombres dures sont engendrées par des sources de lumières ponctuelles. En pratique, ce type de source n'existe pas, puisqu'un objet qui émet de la lumière possède toujours un volume ou une surface propre. Toutefois, une source de lumière de petite taille, comme une simple ampoule, ou suffisamment éloignée, comme le soleil, peut être assimilée à une source ponctuelle (un point de l'espace). Dans ce cas, le problème du calcul d'ombre est simplifié puisqu'il se réduit à déterminer si en chaque point visible d'une scène, une source ponctuelle est visible ou non.

La propagation de la lumière dans l'air étant considérée comme rectiligne, cela induit que deux points de l'espace $p$ et $q$ sont visibles si et seulement si aucun obstacle de la scène n'intersecte le segment $[pq]$. Un environnement virtuel étant constitué de géométrie en générale et plus particulièrement de triangles, la visibilité d'une source ponctuelle se

résume à un problème de recherche d'intersection entre des segments et la géométrie de la scène.

C'est un problème qui n'est pas si difficile et que l'on sait résoudre avec précision. Ce qui est plus compliqué en revanche, c'est de le résoudre en temps-réel. Compte tenu des contraintes propres à ce domaine, même une version simplifiée du problème s'avère délicate à résoudre de manière exacte dans le budget de temps imparti. La grande quantité de travaux qui se sont et qui continuent de s'y intéresser peuvent en témoigner. C'est ce dont nous discuterons ici.

## 1.2 Méthodes de rendu d'ombre dure en temps-réel

Les méthodes de rendu d'ombres en temps-réel, dont nous présentons ici les différentes familles, se distinguent principalement par l'espace dans lequel sont effectués les calculs : l'espace monde ou l'espace image. L'espace monde (ou espace objet) correspond à l'espace 3D dans lequel sont plongés les objets de la scène. Les méthodes dites *basées géométrie* définies dans cet espace utilisent la géométrie de la scène pour déterminer la visibilité d'un point par rapport à la lumière. Les méthodes *basées image* effectuent quand à elles le rendu de l'ombre à partir d'une (ou plusieurs) image(s) calculée(s) du point de vue de la lumière par projection des objets de la scène. La géométrie n'est donc pas utilisée dans son espace de définition, mais projetée préalablement sur un plan image. Chaque espace de calcul possède des avantages et des inconvénients qui influencent le comportement des méthodes qu'il supporte. Générer une image s'avère trivial et particulièrement rapide sur les cartes graphiques modernes, mais les problèmes d'aliassage bien connus [25] inhérents à la nature discrète des images se répercutent sur la qualité des ombres obtenues. À l'inverse, les méthodes basées géométrie fournissent le plus souvent un résultat visuel exact, mais leurs performances moins élevées sont plus sensibles à la complexité géométrique de la scène.

### 1.2.1 Le calcul d'ombre en espace image, ou Shadow Mapping

Le shadow mapping est de loin la technique la plus répandue pour le rendu d'ombre en temps-réel. Introduite par Williams en 1978 [95], la méthode a progressivement été intégrée aux cartes graphiques dans les années 90 jusqu'à bénéficier d'un support matériel complet, disponible sur les cartes graphiques grand public dès le début des années 2000. Dans sa définition initiale, le shadow mapping présente une complexité linéaire. La puissance et le parallélisme des GPU ont toutefois permis de porter la méthode à un très haut niveau de performances, avec des temps de calculs très stables et une très faible sensibilité vis à vis de la complexité géométrique. Ceci explique sa large adoption dans l'industrie et notamment dans les applications temps réel, où il est primordial de respecter les budgets de temps alloués pour chaque étapes du rendu d'une image. Par sa nature basée image, le shadow mapping souffre cependant de nombreux artefacts visuels qui rendent difficile, voire impossible, l'obtention de résultats exacts en toutes circonstances.

#### Principe de fonctionnement

Le shadow mapping repose sur l'observation suivante : un point visible depuis la caméra est éclairé uniquement s'il est visible depuis la source de lumière. Il s'agit donc de calculer

la visibilité depuis la caméra d'une part, depuis la lumière d'autre part, puis de déterminer les éléments communs à ses deux vues. Le calcul des deux vues est effectué en deux passes d'algorithme du Z-buffer, produisant deux buffers de profondeur, celui correspondant à la vue depuis la lumière constituant la shadow map. Une shadow map est donc une image calculée depuis le point de vue de la lumière qui contient dans chacun de ses pixels la profondeur de l'objet visible le plus proche (voir Figure 1.2). Lors du rendu de l'ombre, chaque point visible depuis la caméra est projeté dans l'espace image de la shadow map, et un test de comparaison entre la profondeur stockée dans le pixel correspondant et celle du point permet de déterminer sa visibilité. Le point est dans l'ombre s'il possède une profondeur supérieure à celle stockée dans le pixel, sinon il est éclairé.

FIGURE 1.2 – Visualisation du tampon de profondeur (à droite) associé à un point de vue (à gauche).

Le calcul d'une shadow map repose donc sur l'algorithme de rastérisation qu'implémentent les cartes graphiques. Celui-ci consiste à projeter chaque primitive géométrique de la scène (de nos jours principalement des triangles) sur le plan écran de la caméra afin de déterminer les pixels qu'elle recouvre. Seule la primitive projetée la plus proche contribue à la couleur d'un pixel puisque les primitives plus éloignées seront masquées (en considérant uniquement des objets opaques). La rastérisation doit pour cela employer un mécanisme qui permet de déterminer pour chaque pixel la visibilité des primitives qui s'y projettent. Ce problème est résolu par l'emploi d'un tampon de profondeur (Z-buffer) [18], qui stocke pour chaque pixel la profondeur de la primitive projetée la plus proche. Une shadow map est ainsi générée très rapidement par simple rastérisation de la scène depuis la lumière.

### Artefacts visuels dus à l'aliassage

La shadow map est une discrétisation de la géométrie de la scène vue depuis la source de lumière. Comme pour tout échantillonnage d'un signal, des problèmes d'aliassage surviennent si le signal est échantillonné à une fréquence insuffisante. Cela se traduit dans le cas des ombres par la présence d'artefacts visuels lorsque la résolution des shadow maps est insuffisante par rapport à la résolution de la caméra. La portion de scène couverte par un pixel d'une shadow map contient de la géométrie qui peut varier spatialement, or une seule valeur de profondeur est stockée pour chaque pixel, approchant cette géométrie par

un plan orthogonal au rayon de lumière. Pour un groupe de points appartenant à une surface échantillonnée par un seul pixel d'une shadow map, chaque test de comparaison pour déterminer l'ombre se fait avec la même valeur de profondeur. La figure 1.3 décrit les cas où cela s'avère problématique. Une sphère et un cube éclairés par le dessus projettent une ombre sur un plan. La shadow map correspondante est représentée, avec en rouges les points visibles depuis le centre de chaque pixel. Les points trouvés à l'ombre par le test de profondeur sont affichés en noir, ceux éclairés en jaune.

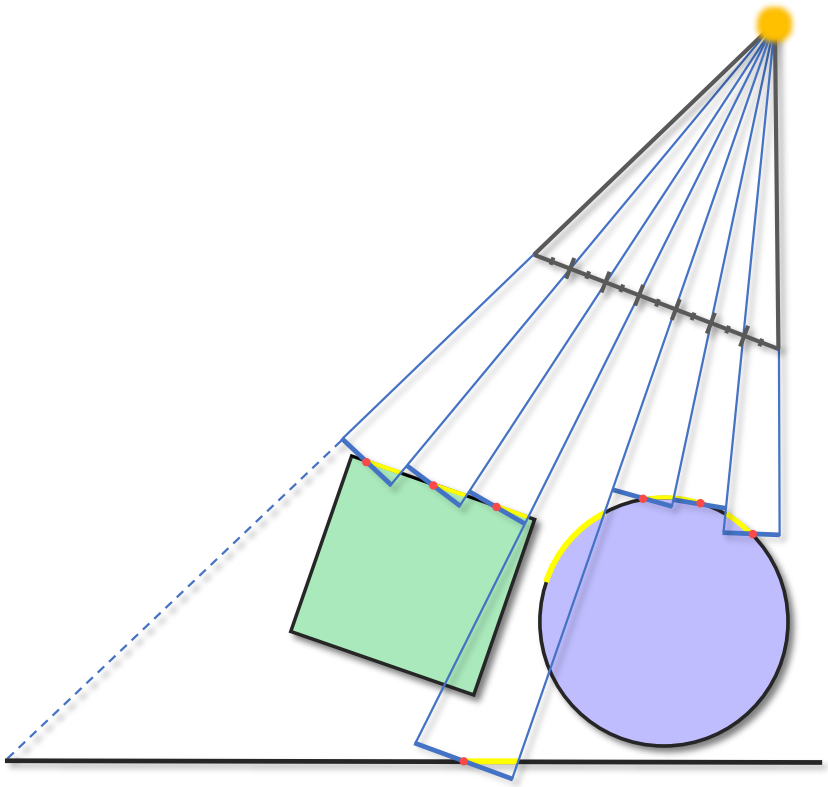

FIGURE 1.3 – Vue de profile d'une scène où une sphère et un cube projettent de l'ombre sur un plan. La profondeur du point visible depuis le centre de chaque pixel de la shadow map (en rouge) est stockée et définie un plan parallèle au plan de projection de la lumière. Les points déterminés à l'ombre par la shadow map sont affichés en noir, ceux éclairés en jaune.

La sphère et le cube souffrent d'un problème d'auto-ombrage sur des parties normalement exposées à la lumière car certains points se trouvent à une profondeur supérieure à celle stockée dans la shadow map. Ce *phénomène d'acné* est particulièrement visible (figure 1.4) mais peut généralement être évité en introduisant un biais sur la profondeur utilisée lors de la comparaison. Le biais est calculé en fonction de l'orientation de la surface et de la direction de la lumière pour garantir l'absence d'auto-ombrage. Ce décalage de l'ombre peu toutefois provoquer l'effet *Peter Pan* (un objet au sol semble être en suspension) lorsqu'un point dans l'ombre est trouvé éclairé car situé à une profondeur supérieure à la valeur stockée dans la shadow map, mais inférieure à cette valeur augmentée du biais (figure 1.5).

Toujours sur la Figure 1.3, on peut constater un autre problème induit par la nature discrète des shadow maps. Sur la partie éclairée du sol (en jaune), la sphère ombre une

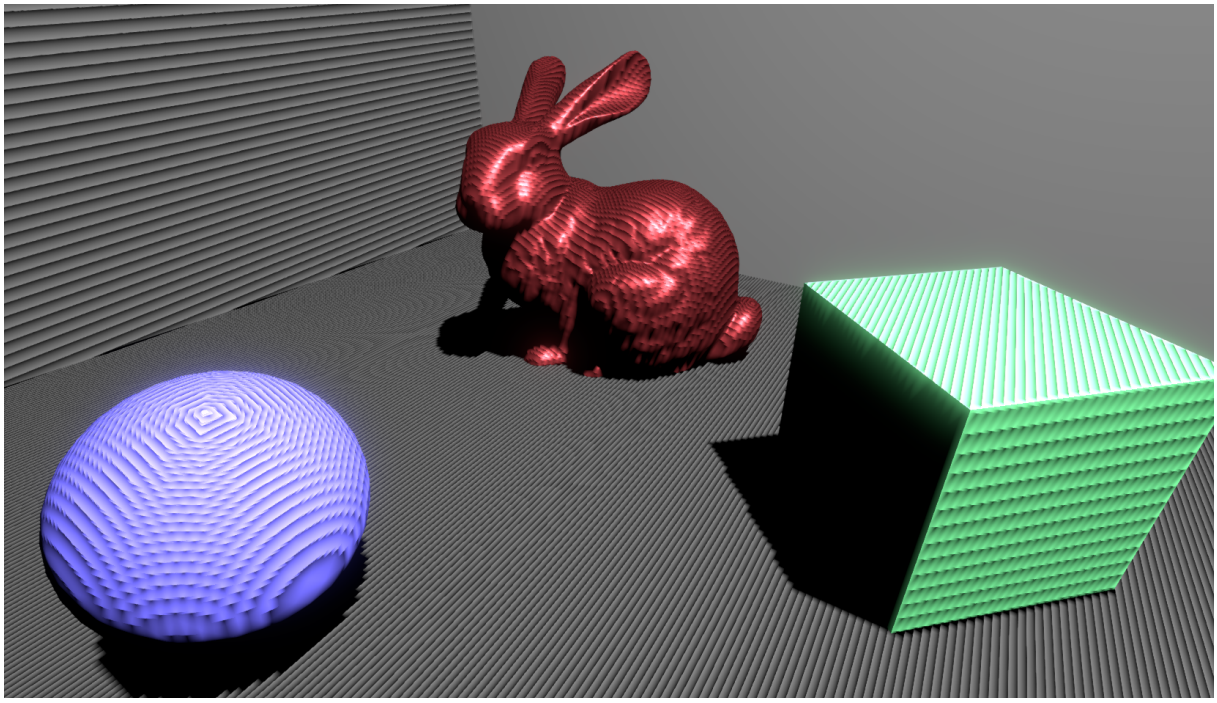

FIGURE 1.4 – Contours d'ombre pixelisés + Acné

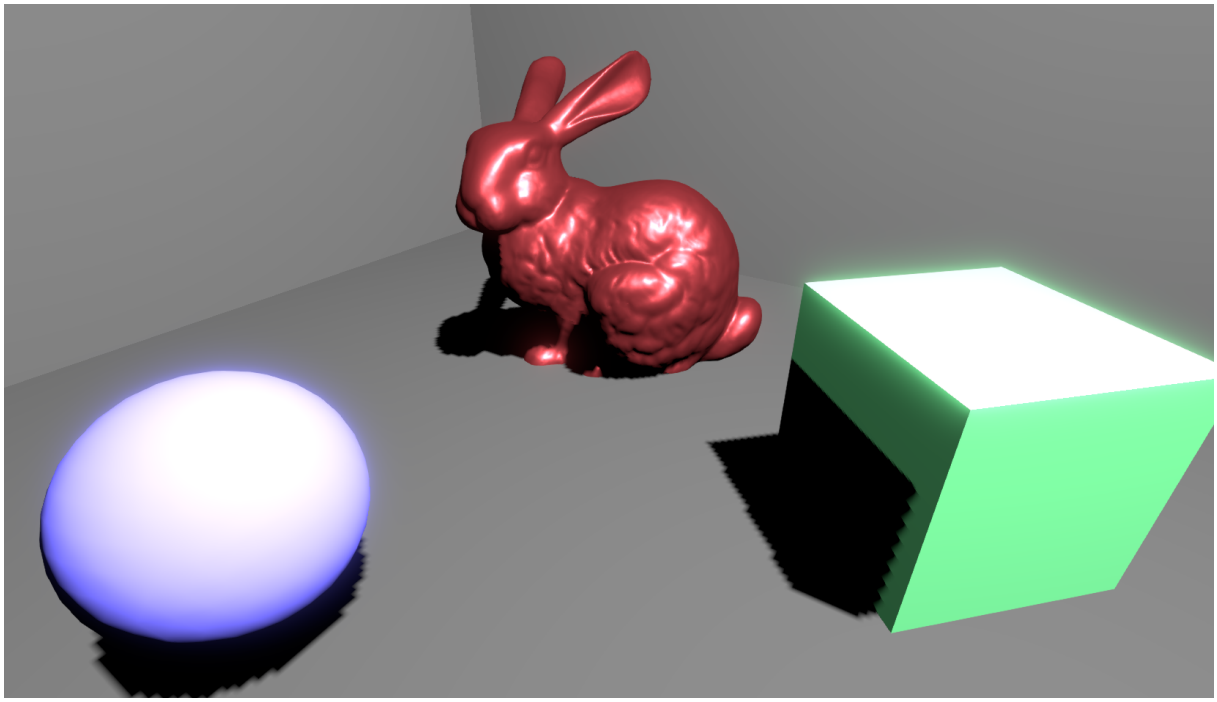

FIGURE 1.5 – Peter Panning

partie de la surface, qui est toutefois considérée éclairée puisque l'échantillon de la shadow map est situé sur le plan. L'inverse du problème est visible sur l'extrémité gauche du sol, où une partie de la zone trouvée dans l'ombre est en fait éclairée. C'est la raison pour laquelle les shadow maps peuvent produire des contours d'ombre pixelisés (figure 1.5).

Pour minimiser ces problèmes d'aliassage, il faut que la résolution de la shadow map

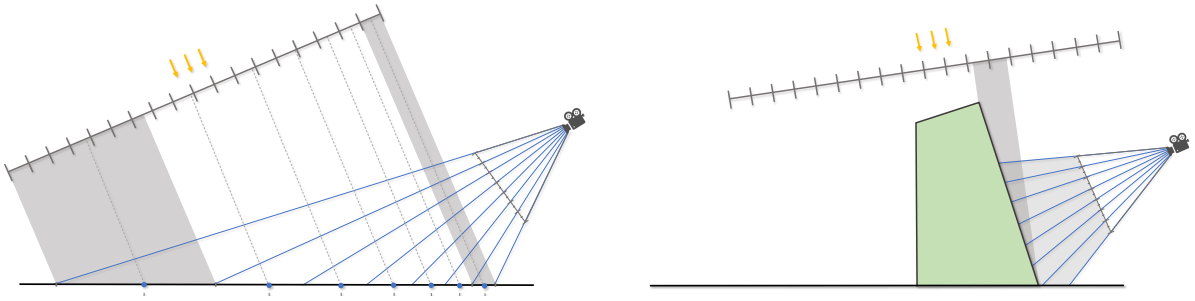

FIGURE 1.6 – À gauche, une illustration du problème d'aliassage de perspective. La partie du plan proche de la caméra requiert un échantillonnage de la shadow map plus dense que la partie éloignée. Or la partie proche reçoit moins d'échantillons de la part de la shadow map. À droite, le problème d'aliassage de projection survient lorsqu'une surface vue par la lumière depuis un angle rasant (et qui reçoit donc peu d'échantillons de la shadow map) est examinée de près par la caméra.

soit suffisante en regard de la distribution des points visibles depuis la caméra. Mais cet enjeu n'est pas forcément évident, puisque la shadow map constitue un échantillonnage uniforme, tandis que la distribution des points caméra vus depuis la lumière ne l'est pas. Stamminger et Drettakis [82] distinguent deux types d'aliassage induits par un l'échantillonnage de la scène au travers d'une grille uniforme :

- L'aliassage *de perspective* : la fréquence d'échantillonnage est plus importante pour les régions proches de la caméra, or ces régions reçoivent moins d'échantillons de la part de la shadow map (voir Figure 1.6 à gauche).
- L'aliassage *de projection* : une même surface peut se projeter sur une petite région de la shadow map tout en se projetant sur une grande région de l'image. C'est le cas lorsqu'une surface tend à être parallèle à la direction de projection de la shadow map mais perpendiculaire à celle de l'image (voir Figure 1.6 à droite). Dans ce cas, la shadow map sous-échantillonne la surface.

À ces problèmes s'ajoute l'imprécision du z-buffer : la précision de la shadow map se dégrade au fur et à mesure que l'on s'éloigne de la lumière, quand bien même une zone éloignée peut être vue de près par la caméra. Cette baisse de précision est due à la manière dont la valeur de profondeur est stockée dans le z-buffer, en $1/z$.

D'apparence triviale à première vue, la mise en pratique du shadow mapping s'avère délicate lorsqu'il s'agit d'éviter les problèmes décris ci-dessus. Au fil des années, de nombreux travaux ont cherché à réduire la présence d'artefacts, que ce soit en amont de l'application du shadow mapping, par l'emploi de méthodes visant à mieux répartir les échantillons de la shadow map, ou en aval, par filtrage du résultat obtenu. Dans le cadre du rendu d'ombres dures, nous nous intéressons ici aux différentes solutions mises en place pour améliorer la répartition des échantillons de la shadow map vis à vis de la distribution des points visibles depuis la caméra.

### Calibration des Shadow Maps

Si la portion de scène visible depuis la caméra se projette sur une petite partie de la shadow map, cela signifie que la majeure partie de ses échantillons sont inutiles. Le risque

de sous-échantillonnage est donc augmenté de même que le risque d'apparition d'artefacts visuels. Brabec et al. [15] adressent le problème en ajustant le *frustum* (la pyramide de vue utilisée pour la rastérisation) de la shadow map à la portion de scène vue par la caméra. Le volume définit par l'enveloppe convexe du frustum de vue et de la source de la lumière est pour cela calculé (dans le cas d'une source directionnelle, ce volume s'étend à l'infini dans la direction de la source), puis intersecté avec la scène et le frustum de la shadow map (figure 1.7). Le calibrage de la shadow map pour chaque image permet de contenir les artefacts dus à l'aliassage mais ne les supprime pas entièrement. En outre, un calibrage différent d'une image sur l'autre peut se traduire par des sauts et des clignotements de l'ombre obtenue.

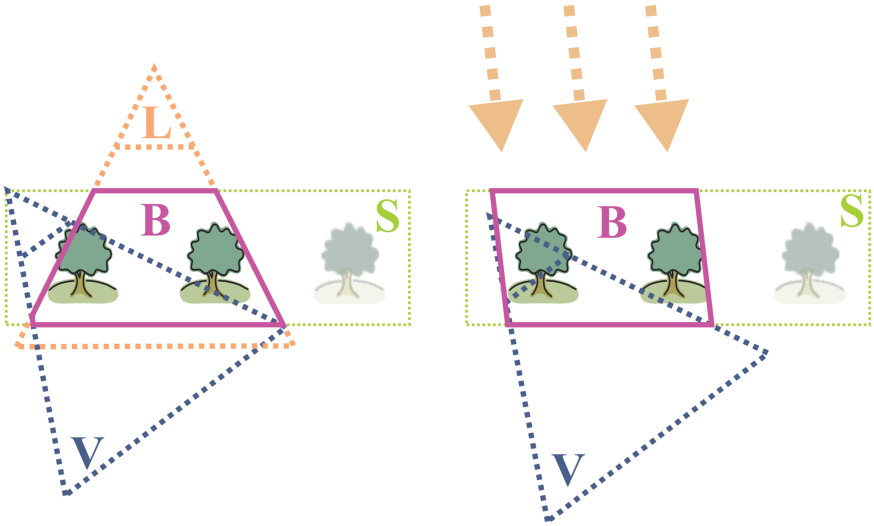


FIGURE 1.7 – À gauche : En bleu, le frustum de vue de la caméra, en orange le frustum de vue de la shadow map, et en vert la boîte englobante de la scène. L'intersection de ces trois volumes (en violet) permet de placer la shadow map au plus proche des objets de la scène et d'éviter le gaspillage des échantillons. À droite, idem avec une source de lumière directionnelle (le frustum de vue de la shadow map s'étend à l'infini). *Source [31]*

### Reparamétrisation des Shadow Maps

Lutter contre l'aliassage de perspective nécessite d'allouer plus d'échantillons de la shadow map sur les zones proches de l'observateur (figure 1.6). Comme nous le verrons, un échantillonnage irrégulier de la shadow map, bien qu'efficace pour supprimer les problèmes d'aliassage, est moins adapté à la rastérisation matérielle des cartes graphiques et apporte son propre lot de problèmes (section 1.2.1). Afin de conserver un échantillonnage régulier, Stamminger et Drettakis [82] proposent plutôt d'appliquer une transformation à la scène pour élargir les objets proches de la caméra et rétrécir les objets éloignés : un échantillonnage uniforme de l'espace résultant par une shadow map favorisera ainsi les zones proches de la caméra (figure 1.8). Les auteurs utilisent pour cela l'espace issu de la projection perspective (*post-perspective space*), où le frustum de vue est transformé en un cube (figure 1.9). La transformation perspective affecte cependant la position et la nature des sources de lumières (une source ponctuelle peut se projeter à l'infini et devenir une source directionnelle, etc...) et rend délicate l'implémentation de la méthode, qui

nécessite de prendre en compte de nombreux cas particuliers et présente des singularités. Wimmer et al. [96] évitent les problèmes de robustesse en appliquant la transformation dans l'espace de la lumière. Malgré une nette diminution des artefacts lorsque la lumière surplombe l'observateur, la méthode ne permet malheureusement pas toujours d'améliorer la répartition des échantillons de la shadow map, notamment lorsque lumière se trouve derrière la caméra.

FIGURE 1.8 – À gauche, utilisation d'une shadow map classique. À droite, le calcul de la shadow map *après* la transformation perspective de la caméra donne une plus grande importance aux objets proches de la caméra. Par conséquent, les artefacts sont nettement diminués pour les ombres proches de la caméra. *Source [82]*

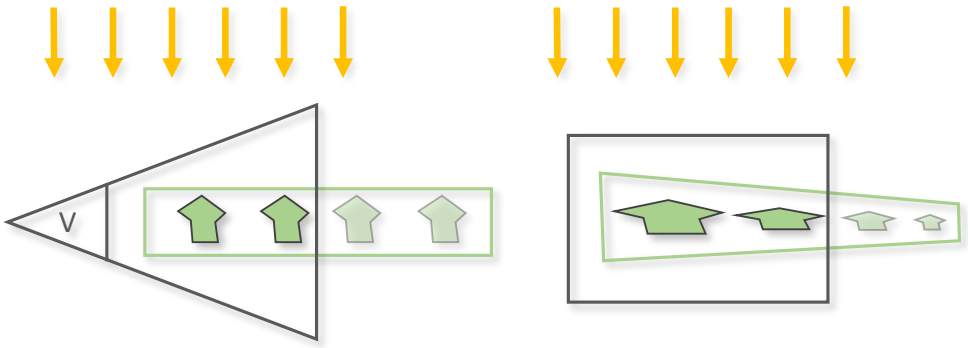


FIGURE 1.9 – À gauche : une scène avant la projection perspective dans le frustum de vue (en gris). À gauche : Suite à la transformation de l'espace par la projection perspective, le frustum de vue est déformé en un cube et les objets proches de l'observateur s'élargissent.

### Cascades de Shadow Maps

Une solution plus efficace contre les problèmes d'aliassage de perspective, les cascades de shadow maps (*Cascaded Shadow Maps* (CSM) [32], *Parallel-Split Shadow Maps* (PSSM) [102], ou encore *z-partitioning* [63]), consiste à partitionner le frustum de vue en $n$ sous-frustums, chacun associé à une shadow map dédiée. La résolution des shadow maps reste identique quelle que soit la partition du frustum de vue auxquelles elles sont attribuées, mais les partitions proches de l'utilisateur reçoivent un échantillonnage plus dense puisque la surface qu'elle recouvre est plus faible que pour les partitions éloignées (figure 1.10), cela à condition bien sûr que les frustums des shadow maps soient calibrés pour les portions de scène associées. L'attribution d'une shadow map distincte pour les différentes zones du frustum de vue permet de redistribuer l'erreur sur l'écran et rend les

artefacts moins perceptibles. L'efficacité des cascades dépend toutefois de la distribution des partitions le long du frustum de vue, et il n'existe pas de stratégie de partitionnement fixe qui convienne à tous les points de vue (qu'elle soit uniforme, logarithmique ou une combinaison des deux). Dans le domaine du jeu vidéo, la tâche a souvent été déléguée aux artistes [56] qui devaient régler manuellement les paramètres pour différents points de vue. Même dans ce cas, tous les points de vues ne peuvent pas être pris en compte, et le partitionnement fini toujours par devenir sous-optimal. Lauritzen et al. [56] proposent donc d'automatiser le processus de partitionnement. Ils utilisent le tampon de profondeur afin d'obtenir la distribution des échantillons le long du frustum de la vue, ce qui leur permet de placer efficacement les shadow maps.

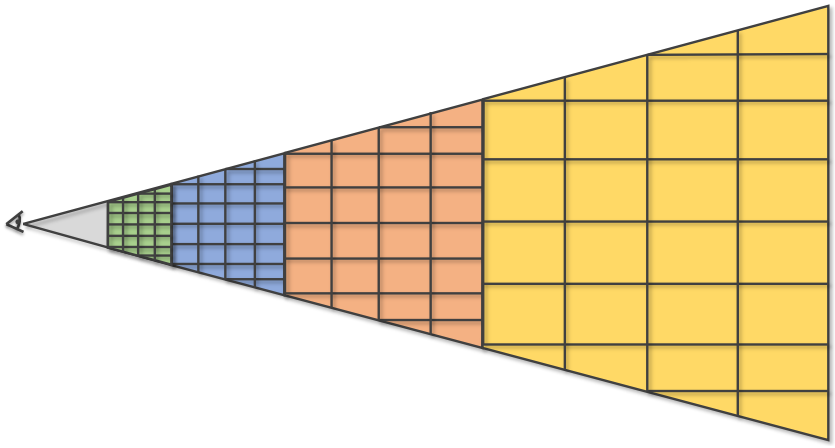

FIGURE 1.10 – Les cascades de shadow maps permettent une meilleure corrélation entre la distribution des points à ombrer et la définition des shadow maps. La densité des points images tend à diminuer lorsqu'on s'éloigne de la caméra en raison de la perspective. En subdivisant le frustum de vue en plusieurs régions, on se rapporte à des distributions de points plus homogènes, ce qui permet de mieux calibrer une shadow map pour chaque région. Une cascade de 3 ou 4 shadow maps est un nombre relativement commun en pratique.

### Shadow Maps adaptatives

La calibration, la reparamétrisation et la mise en cascade des shadow maps adressent les problèmes d'aliassage de manière globale : en considérant la scène dans son ensemble pour essayer de trouver un placement optimal de la/des shadow map(s). Ces méthodes sont donc inefficaces contre l'aliassage de projection car c'est un phénomène local, qui demande une forte concentration d'échantillons de la shadow map lorsqu'il se produit. Lutter contre ce phénomène avec un échantillonnage régulier passe obligatoirement par l'augmentation de la résolution de la shadow map. Cependant, la résolution nécessaire peut très rapidement dépasser la quantité de mémoire disponible (ou simplement la taille maximale de texture allouable, actuellement $32\,768^2$), sans parler du coût lié à la création d'un z-buffer d'une telle taille. Une solution plus raisonnable consiste à augmenter la résolution effective de la shadow map uniquement dans les régions concernées, de manière adaptative. Pour cela, une shadow map d'une résolution donnée est progressivement subdivisée en $2 \times 2$ sous-blocs. Une shadow map de même résolution est créée pour chaque

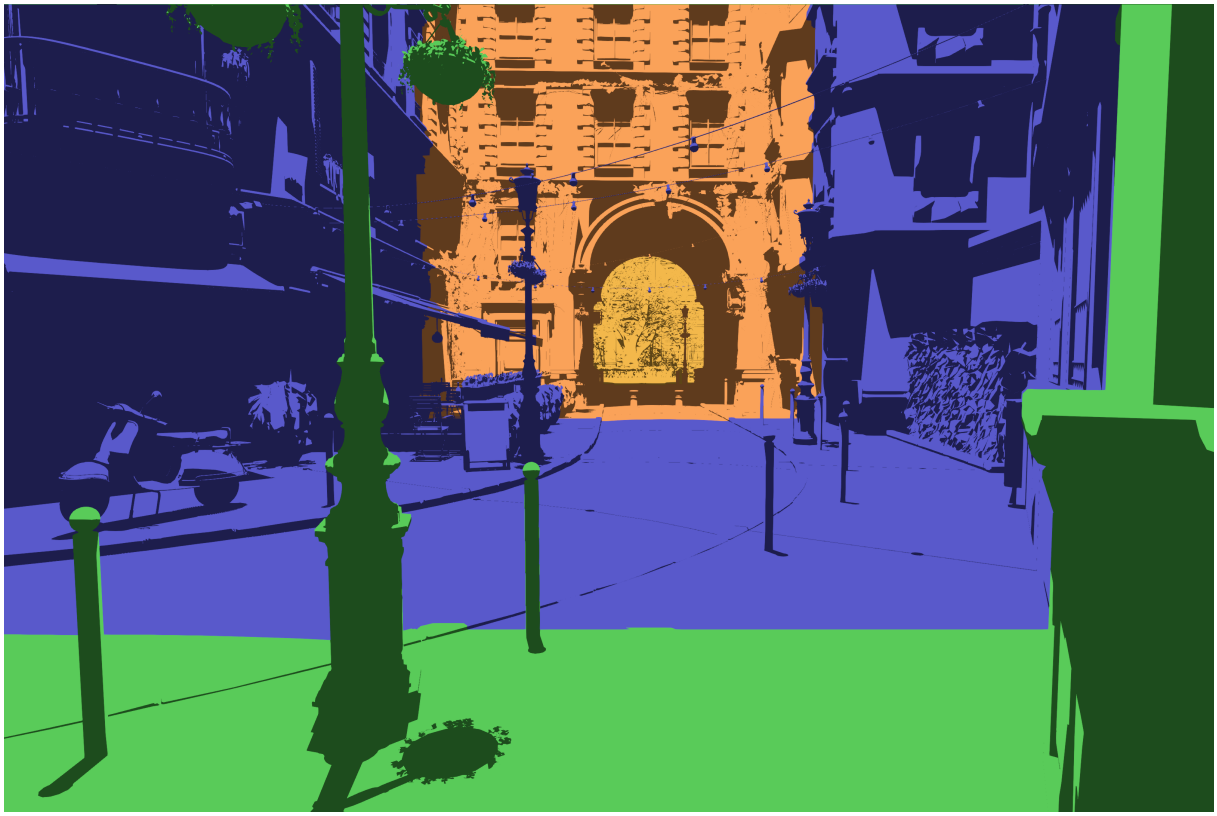

FIGURE 1.11 – Visualisation des shadow maps utilisées pour le rendu d'ombre. Chaque couleur correspond à une shadow map différente de la cascade.

nouveau bloc, et le processus de raffinement s'arrête lorsque la résolution des nouvelles shadow maps est suffisante pour ombrer la scène. Giegl et Wimmer [41] proposent une méthode simple et adaptée aux GPUs. La première shadow map sert à ombrer la scène pour produire un *buffer* d'ombre. Les $2 \times 2$ nouvelles shadow maps sont ensuite créées et utilisées pour mettre à jour le buffer d'ombre. Si le nombre de pixels modifiés est inférieur à un seuil donné, la résolution de la nouvelle shadow map est suffisante et la subdivision peut s'arrêter, sinon elle continue. Le comptage du nombre de pixels modifiés est fait pendant la mise à jour du buffer d'ombre : un pixel émet une valeur uniquement s'il modifie le buffer d'ombre, et une requête d'occlusion compte le nombre de pixels émis. Fernando et al. [34] déterminent si une shadow map doit être affinée en détectant si elle contient des pixels appartenant au contour d'ombre. Si c'est le cas, la taille de l'empreinte du bloc correspondant est utilisée pour déterminer la résolution des nouvelles shadow maps. Cette phase d'analyse est à l'origine conduite sur CPU, mais Lefohn et al. [58] l'adaptent sur GPU en utilisant un algorithme de construction de *quadtree* dédié [59].

La nature itérative de ces méthodes les rend coûteuses car un rendu de la scène est nécessaire pour chaque shadow map. Seules les shadow maps des derniers niveaux de la hiérarchie sont cependant nécessaires pour le rendu final de l'ombre. Dans une approche similaire, Giegl et Wimmer [40] ainsi que Lefohn et al. [60] essayent de résoudre ce problème en déterminant à l'avance la hiérarchie de shadow maps pour en construire uniquement les derniers niveaux. La scène est rendue depuis l'observateur pour créer un buffer contenant des informations sur la résolution de shadow map nécessaire pour ombrer chaque pixel. Pour un échantillon de vue donné, les informations du pixel correspondant indiquent ses

coordonnées $(s, t)$ dans la shadow map, sa profondeur $d$ dans l'espace lumière ainsi qu'une information de résolution $r$ (un gradient de texture qui correspond généralement à un niveau de mip-map). Le buffer est alors utilisé pour créer la hiérarchie de blocs de shadow maps (stockée sous la forme d'une table paginée mip-mappée ou d'une texture virtuelle). Les shadow maps des derniers niveaux de la hiérarchie sont finalement créées et utilisées pour calculer l'ombre finale.

### Shadow Maps : premier bilan

La technique du *shadow mapping* bénéficie d'un avantage non négligeable pour le rendu d'ombre temps-réel : l'exploitation de l'espace image permet de bénéficier de la puissance des cartes graphiques et offre ainsi d'excellentes performances. Opérant en espace image, la technique est de fait peu sensible à la complexité géométrique et visuelle. Elle reste stable et garantie de respecter un budget de temps alloué. Mais la force de la méthode fait aussi sa faiblesse : Comme le note Williams dans sa conclusion lorsqu'il introduit le concept en 1978 [95], la discrétisation de la scène opérée par les shadow maps entraîne de nombreux artefacts qu'il n'est pas simple de prévenir ou de corriger :

> *"The challenge of this approach to computer graphics is to cope successfully with the problems posed by the discrete nature of image space scene representations."*

> *"Le défi posé par cette approche à l'informatique graphique est de bien gérer les problèmes liés à la nature discrète de l'espace image pour représenter la scène."*

Ce défi a en effet occupé de nombreux chercheurs et développeurs, comme nous avons pu le voir, sans pour autant le résoudre totalement. Dans la pratique, les cascades de shadow maps demeurent la méthode la plus robuste et la plus simple pour réduire convenablement les artefacts. Il est cependant commun de devoir recourir à des résolutions de shadow maps quatre fois supérieures, ou plus, à la résolution de l'écran afin d'obtenir des résultats satisfaisants. Avec la tendance à l'augmentation des résolutions d'écran (4k et plus, casques de réalité virtuelle), plus la multiplication des points de vue pour la gestion des sources omnidirectionnelles, cela commence à poser le problème d'une consommation mémoire excessive [78].

Il reste néanmoins une dernière famille de méthodes basées sur le *shadow mapping* à prendre en compte. Comme nous l'avons évoqué, un échantillonnage irrégulier de la shadow map, avec une correspondance "*un à un*" entre les échantillons de vue et ceux de la shadow map, permettrait de s'affranchir totalement des problèmes d'artefacts. C'est la piste qui est explorée par les travaux que nous présentons maintenant.

### Shadow Maps irrégulières

Ces travaux s'intéressent à la source des problèmes d'aliassage qui affectent la technique du *shadow mapping* : la distribution uniforme des pixels de la shadow map induite par la rastérisation matérielle ne correspond pas à la distribution des échantillons de vue une fois projetés (figure 1.12). Des efforts ont été faits pour permettre un échantillonnage

irrégulier de la shadow map, soit en remplaçant la passe de rastérisation sur carte graphique, soit en l'adaptant. Dans tous les cas, le principe général reste le même et consiste à inverser les étapes du *shadow mapping* traditionnel :

— Le rendu de la scène depuis l'observateur permet d'obtenir la liste des échantillons de vue,
— Les échantillons de vue sont projetés dans l'espace lumière, et une structure accélératrice (la shadow map irrégulière) est construite sur ces échantillons,
— La scène est rastérisée depuis la lumière, et les échantillons couverts par chaque triangle sont localisés grâce à la structure d'accélération afin de mettre à jour leur information de visibilité.

Dans un premier prototype proposé par Aila et al. [3], les échantillons de la shadow map irrégulière sont insérés dans un *kd-tree* (un arbre BSP 2D aligné sur les axes et construit ici dans le plan de la shadow map) entièrement calculé sur CPU. Cette approche n'est donc pas applicable en temps-réel mais valide le concept de shadow map irrégulière. En 2005, Johnson et al. [49] proposent d'utiliser un *tampon de profondeur irrégulier* (ou *Irregular Z-Buffer*, IZB). Les échantillons de vue sont projetés sur une grille régulière, et les échantillons appartenant à une même cellule de la grille sont stockés dans une liste. Puisque les échantillons peuvent se projeter n'importe où à la surface d'un pixel de l'IZB, la scène doit être rastérisée de façon conservative : les triangles sont élargis de manière à ce qu'un pixel soit considéré comme couvert dès qu'un triangle intersecte le pixel correspondant de l'IZB (et pas seulement son centre). Pour chaque pixel couvert par un triangle, la liste des échantillons de vue est parcourue afin de déterminer leur visibilité. Le travail de Johnson cible la micro-architecture *Larrabee*, qui devait intégrer des mécanismes de parcours de liste au niveau matériel. Cette architecture n'a finalement pas vue le jour, mais le principe de cet algorithme a néanmoins inspiré d'autres travaux par la suite.

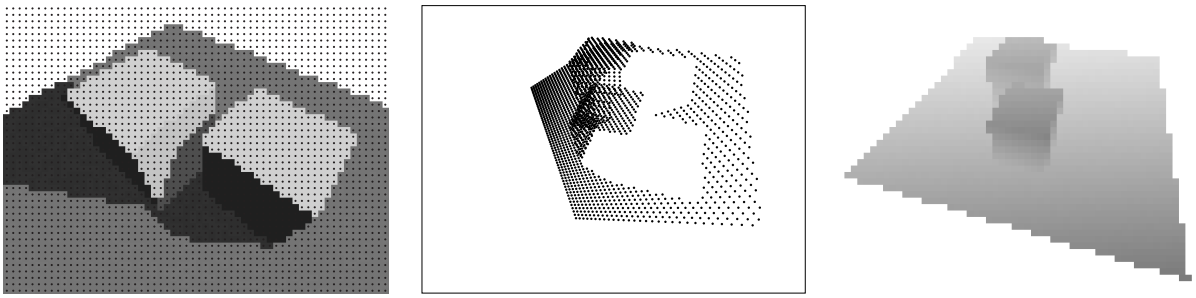

FIGURE 1.12 – À gauche : une scène vue par un observateur, les points noirs correspondent au centre des pixels. Contrairement aux pixels d'une shadow map classique (à droite), la distribution des points image projetés dans la shadow map (et visualisés au milieu) n'est pas uniforme. *Source [3]*

Une implémentation sur carte graphique des tampons de profondeur irréguliers est proposée en 2008 par Sintorn et al. [79] dans le cadre des ombres douces. La méthode souffre de problèmes de stabilité, avec des performances qui varient considérablement d'un point de vue à l'autre. La distribution des échantillons de vue dans l'IZB (qui dépend du point de vue de la caméra) détermine la taille des listes de l'IZB. Or, pour certains points de vue, la longueur de ces listes peut varier considérablement d'un pixel à l'autre et ralentir

la passe de rastérisation sur l'IZB. Cela est dû de l'architecture matérielle des GPUs, conçues pour favoriser l'exécution d'un grand nombre de threads en parallèle. À un instant donné, chaque processeur de la carte graphique exécute un groupe de 32 ou 64 *threads* appelés *warps* ou *wavefronts*. Les threads d'un warp exécutent les mêmes instructions, et ce de manière synchronisée. Il est donc important que la charge de travail dans le warp soit homogène afin d'éviter qu'une majorité de threads restent inactifs en attendant qu'un ou plusieurs retardataires aient terminé leur exécution. Lors de la rastérisation de la scène sur l'IZB, le temps d'exécution d'un warp (dont les threads sont attribués à un bloc de pixels contigus) est déterminé par le temps d'exécution du thread qui parcours la liste d'échantillons la plus longue. D'où l'enjeu d'assurer autant que possible une répartition globalement uniforme des échantillons sur l'IZB. Si cela était faisable, une shadow map classique et de bonne résolution suffirait. D'une certaine façon, on en revient à la difficulté inhérente au shadow mapping identifiée dès le départ par Williams. Mais au moins cela n'impacte les IZBs qu'en terme de performance, les problèmes d'artefacts étant ici résolus.

Cette propension des IZBs à répartir la charge de travail de manière inégale sur la carte graphique les a conduits à être mis de côté. La technique a toutefois été reprise en 2015 par Wyman [98], qui remarque que les problèmes de variation de taille des listes surviennent aux mêmes endroits que les problèmes d'aliassage pour les shadow maps traditionnelles. Il en déduit que si les stratégies mises en œuvre pour lutter contre l'aliassage des shadow maps s'appliquent à l'IZB et permettent d'homogénéiser la charge de travail, leur performances sur GPU devraient être améliorées. En combinant des cascades de shadow maps irrégulières à la manière de Lauritzen [56] avec une réduction des primitives à rendre en tirant parti des spécificités de l'implémentation matérielle du tampon de profondeur (*early z-cull*), il arrive à stabiliser les performances de l'IZB et à obtenir des résultats suffisamment rapides pour être utilisés en production dans certains jeux-vidéos [70]. Mais malgré une nette amélioration comparé à une version naïve, les variations de taille des listes persistent suffisamment pour faire varier les performances du simple au double. Aussi, contrairement aux shadow maps classiques, pour lesquelles la passe de rastérisation matérielle permet de gommer en partie la dépendance linéaire de l'algorithme à la géométrie de la scène, les IZBs restent assez sensibles à cette augmentation, avec des performances qui se dégradent pour des scènes au delà des 10 millions de triangles.

### 1.2.2 Les volumes d'ombre ou Shadow Volumes

La méthode des volumes d'ombre (ou *Shadow Volumes*) est une méthode analytique qui, contrairement au shadow mapping, utilise la géométrie de la scène pour déterminer si un point se trouve dans l'ombre. Les volumes d'ombre opèrent donc dans l'espace objet et sont à ce titre opposés aux shadow maps qui opèrent dans l'espace image. Introduite en 1977 par Crow [26], cette approche permet d'obtenir des résultats exacts en toute circonstances, au prix toutefois de temps de calculs plus importants et moins stables en raison de leur dépendance à la complexité géométrique et visuelle de la scène.

#### Principe de fonctionnement

Géométriquement, l'ombre projetée par un objet donne lieu à un volume. Pour un triangle, ce volume est une pyramide tronquée formée par les trois plans qui passent par

la lumière et chaque arête du triangle, avec comme base le triangle lui-même (figure 1.13). Un point de la scène est ombré s'il se trouve à l'intérieur d'au moins un volume d'ombre. À l'inverse, s'il n'est contenu dans aucun volume d'ombre, il est éclairé. Le but des méthodes dites *de volumes d'ombre* est donc de représenter la géométrie qui compose ces volumes afin de déterminer si les échantillons de vue sont contenus dedans ou non. Une approche naïve consisterait à tester pour chaque pixel l'ensemble des volumes d'ombre jusqu'à trouver une inclusion du point dans l'un d'eux. La complexité de ces requêtes en $O(n)$ pour $n$ triangles n'est évidemment pas viable en pratique. Bien que la notion de volumes d'ombre fût introduite en 1977, il faudra attendre 1991 [46] pour voir émerger la première implémentation adaptée aux cartes graphiques, et 2003 [16] pour qu'elle soit appliquée de manière à obtenir des performances temps-réelles sur des scènes suffisamment complexes.

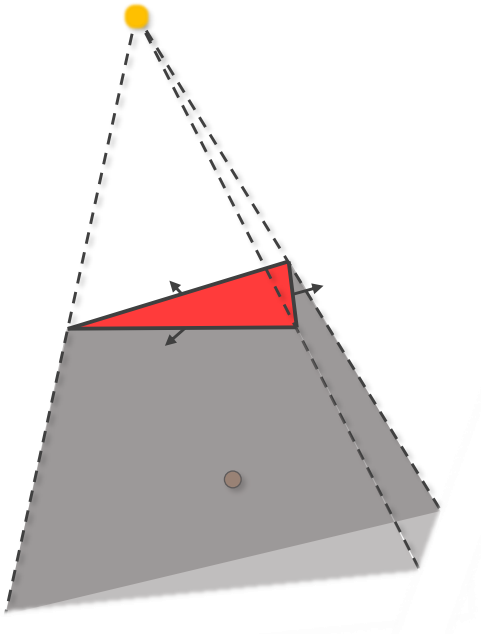

FIGURE 1.13 – Volume d'ombre généré par un triangle. Le volume est caractérisable analytiquement par 4 plans : 3 plans définis par la lumière et une des arêtes du triangle, plus le plan support au triangle.

### Le Z-Pass, ou volumes d'ombre rastérisés

Exploitant l'apparition des *stencil buffers* sur carte graphique, Heidmann [46] propose en 1991 une implémentation des volumes d'ombre basée sur l'observation suivante : déterminer si un point se trouve dans l'ombre revient à compter combien de fois le rayon partant de la caméra en direction du point entre et sort d'un volume d'ombre. Si le rayon est sorti autant de fois qu'il est entré, alors le point à son extrémité se trouve éclairé (figure 1.14). Un compteur initialisé à zéro est incrémenté/décrémenté à chaque fois que le rayon entre/sort d'un volume d'ombre. Le point se trouve dans l'ombre si la valeur finale du compteur est différente de zéro.

Heidmann remarque que les rayons entrent dans un volume d'ombre par les plans qui font face à la caméra, pendant qu'ils sortent par les plans qui sont de dos. Il propose de rastériser depuis la caméra les plans infinis (sous la forme de 3 quadrilatères plus le tri-

angle) qui constituent les volumes d'ombre et d'utiliser le *stencil buffer* comme compteur pour chaque pixel. Le stencil buffer est une grille dont chaque cellule possède un compteur. Dans le cas présent la résolution du stencil buffer est identique à celle de l'image. Lorsqu'un plan (en pratique un quadrilatère ou triangle) est projeté sur l'écran, les pixels qu'il recouvre voient leur valeur dans le *stencil buffer* incrémentée ou décrémentée selon l'orientation du plan (figure 1.14). L'optimisation du tampon de profondeur doit être désactivée afin de prendre en compte tous les volumes d'ombre le long d'un rayon de vue.

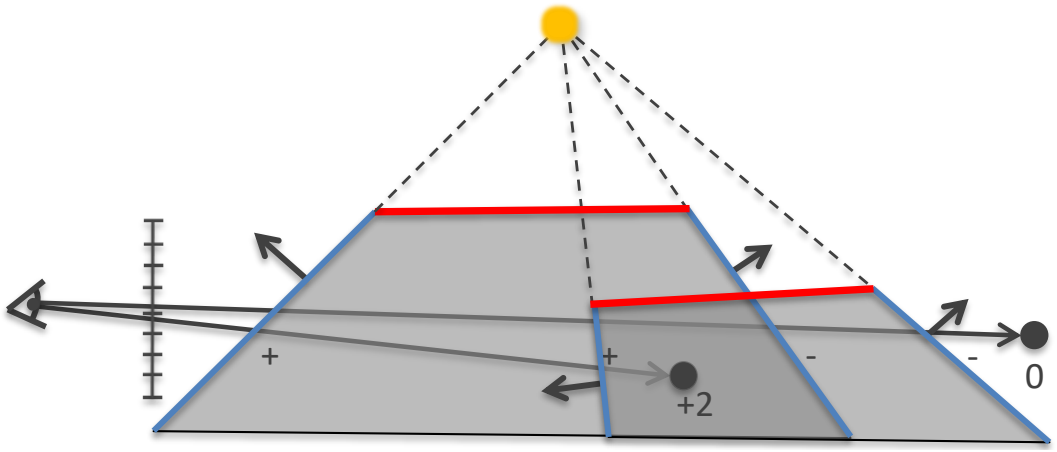


FIGURE 1.14 – Z-Pass : Un compteur est incrémenté/décrémenté à chaque fois que le rayon qui relie un point image entre/sort d'un volume d'ombre. Si la valeur finale du compteur est nulle, le point est éclairé, sinon il est ombré. *Source [31]*

Comme pour les shadow maps, la rastérisation des volumes d'ombre permet de tirer parti de l'accélération matérielle de la carte graphique. Elle sert également à réduire la complexité algorithmique de la méthode : pour chaque pixel, seuls les plans des volumes qui s'y projettent sont considérés. Toutefois, les plans générés par la géométrie peuvent couvrir une grande partie de l'écran et donner lieu à un recouvrement important le long de la direction de vue (figure 1.15). Cela peut conduire à saturer la bande-passante mémoire de la carte graphique, celle-ci possédant une limite (*fillrate*) quant au nombre de pixels qu'elle peut mettre à jour par seconde. Lorsque cela se produit, les performances s'effondrent. Pour la même raison, la méthode peut manquer de stabilité, les performances évoluant en fonction du nombre et de la taille des volumes d'ombres à traiter, ce qui peut varier rapidement d'un point de vue à un autre.

### Artefacts liés à la position de la caméra

La méthode de Heidmann échoue lorsque la caméra (l'œil) se trouve à l'intérieur de l'ombre (figure 1.16). Dans ce cas, chaque pixel du *stencil buffer* devrait être initialisé avec le nombre de volumes d'ombre contenant l'œil. De plus, le plan de projection de la caméra peut intersecter certains volumes d'ombre, ce qui peut modifier la valeur de chaque pixel. L'algorithme du Z-Pass utilise la caméra comme point de référence pour compter le nombre de volumes d'ombre traversés. N'importe quel point peut toutefois être utilisé à condition qu'il soit hors de tout volume d'ombre. Utiliser la source de lumière comme point de référence semble à première vue être un choix judicieux, mais cela implique d'effectuer le rendu de la scène depuis la lumière et revient à basculer sur l'algorithme

FIGURE 1.15 – Un recouvrement important des volumes d'ombre (dont les contours sont affichés en blanc), surtout lorsqu'ils couvrent la majeure partie de l'écran, comme c'est le cas pour la caméra située à gauche de la scène, peut sérieusement ralentir la méthode. *Source [80]*

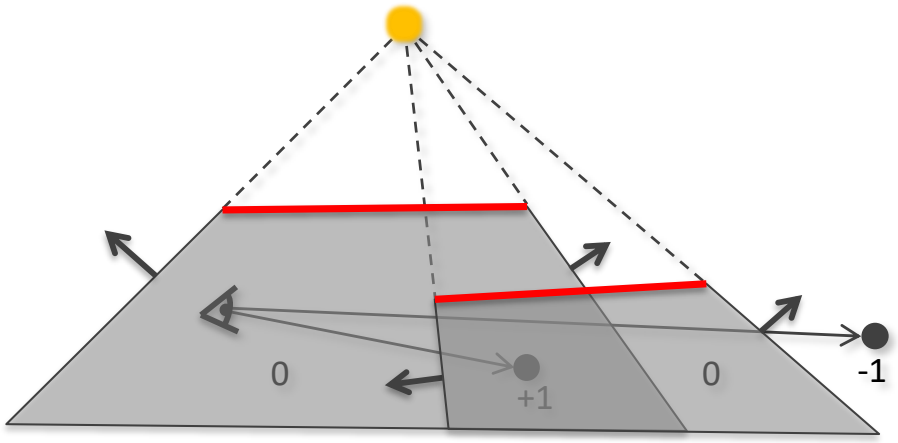


FIGURE 1.16 – L'algorithme Z-Pass échoue lorsque l'œil se trouve à l'intérieur de l'ombre. *Source [31]*

du *shadow mapping*, introduisant ainsi les problèmes d'aliassage inhérents à la technique (voir section 1.2.1). Carmack [17] note que si les volumes d'ombre sont limités à la boîte englobante de la scène, n'importe quel point à une distance infinie dans la direction de vue se trouve nécessairement hors de l'ombre. Il propose donc d'inverser le sens de comptage de l'algorithme Z-Pass, ainsi renommé Z-Fail. En pratique cela revient simplement à inverser le test du tampon de profondeur et la condition d'incrémentation/décrémentation du *stencil buffer*. Cette inversion du sens de comptage tend toutefois à augmenter le nombre de volumes d'ombre à rendre et offre généralement des performances moindres comparé au Z-Pass.

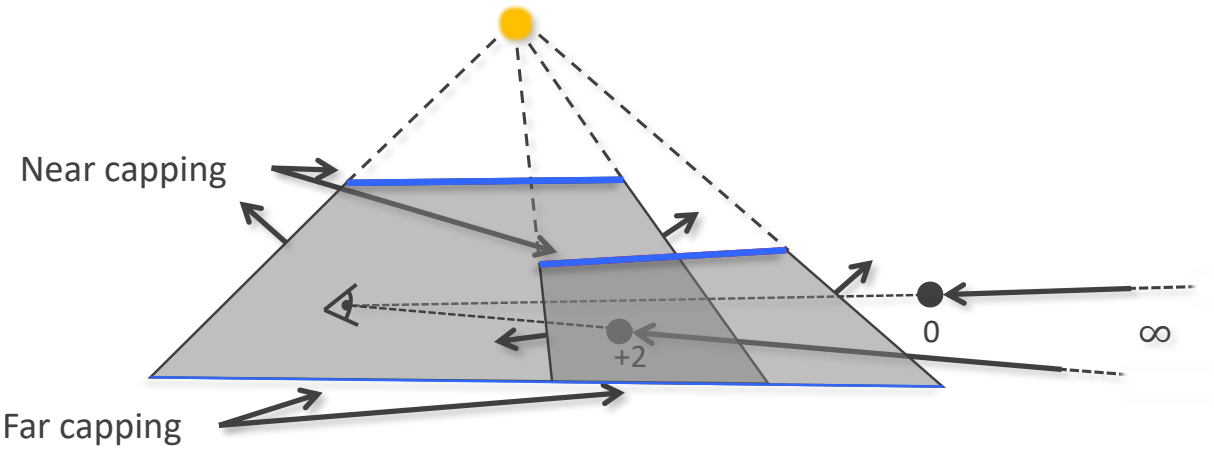


FIGURE 1.17 – Z-Fail : le sens de comptage, qui est inversé par rapport à l'algorithme Z-Pass, se fait à partir d'un point situé "à l'infini" dans la direction de vue. Un plan de section doit ainsi être ajouté pour fermer chaque volume d'ombre. *Source [31]*

Une autre solution est proposée par Eisemann et al. [31]. Ils notent que le nombre de volumes d'ombre contenant l'œil est équivalent au nombre de triangles intersectés par le segment lumière-œil. Cela se calcule efficacement sur GPU en effectuant un rendu de la scène depuis la lumière dans la direction de l'œil. Un *stencil buffer* de résolution 1x1 initialisé à zéro est incrémenté à chaque fois qu'un triangle est rencontré. À la fin, celui-ci contient le nombre de volumes d'ombre contenant l'œil, et cette valeur sert à initialiser le *stencil buffer* de l'algorithme *Z-Pass*. Cette solution ne prend pas en compte l'intersection potentielle du plan de projection de la caméra avec les volumes d'ombre, qui produirait des valeurs initiales différentes selon les pixels.

### Amélioration des performances des volumes d'ombre

Bien que moins nombreux que la littérature sur les shadow maps, plusieurs travaux visent à améliorer les performances des volumes d'ombres en utilisant les deux leviers d'action disponibles : limiter le nombre de volumes d'ombre à rendre, et réduire leur taille au stricte nécessaire pour minimiser leur empreinte sur l'écran. Dans les deux cas, l'objectif est de prévenir la saturation de la bande passante mémoire du matériel graphique.

Dès 1977, Crow remarque que les plans générés par les arêtes silhouettes des modèles vus depuis la lumière sont suffisants pour le rendu des volumes d'ombre [26]. Brabec et Seidel sont les premiers à proposer en 2003 une solution pour détecter les arêtes silhouettes "à la volée" sur la carte graphique [16]. La réduction conséquente du nombre de plans à rastériser a ainsi permis de gérer pour la première fois des scènes modérément complexes tout en offrant d'excellentes performances, au prix toutefois du calcul et du stockage des informations d'adjacence entre les triangles.

Le nombre de volumes d'ombres à rastériser peut également être réduit en éliminant les objets qui ne peuvent affecter l'ombrage pour un point de vue donné, à savoir :

— Les objets entièrement situés dans l'ombre,

— Les objets dont l'ombre n'affecte aucun autre objet ou seulement des objets qui ne

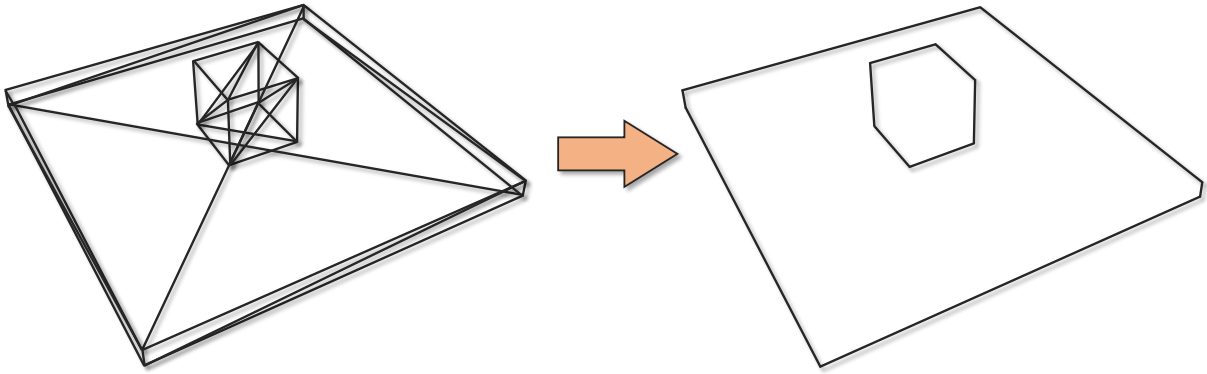

FIGURE 1.18 – L'élimination des arêtes non silhouettes permet de réduire considérablement le nombre de plans de volumes d'ombres à rastériser.

sont pas visibles depuis la caméra.
Lloyd et al. [64] proposent une version améliorée de l'algorithme de Govindaraju [43] pour détecter ces objets. Tout d'abord, l'ensemble des projeteurs (respectivement receveurs) d'ombre potentiels —c'est-à-dire les objets au moins partiellement visibles depuis la lumière (resp. depuis la caméra)— sont calculés. Outre le point de vue qui change selon l'ensemble considéré, l'opération reste la même : Un z-buffer est d'abord calculé depuis la lumière (resp. depuis la caméra). Ensuite, après avoir désactivé l'écriture dans le z-buffer, les boîtes orientées des objets sont rendues en utilisant des *requêtes d'occlusion (occlusion query)*. Ces requêtes sont utilisées pour indiquer le nombre de pixels qui ont passé avec succès le test de profondeur pour un objet donné. Si aucun pixel ne passe le test, alors l'objet n'est pas visible. Ainsi, cette étape élimine d'office les objets entièrement situés dans l'ombre de l'ensemble des projeteurs potentiels. Il suffit de compléter l'étape de détection des projeteurs potentiels pour supprimer ceux dont l'ombre n'affecte aucun receveur visible par l'observateur : Une fois le z-buffer calculé, les receveurs d'ombre potentiels sont rastérisés (toujours depuis la lumière) dans un *stencil buffer*. Les pixels du stencil buffer (initialement à 0) sont mis à 1 lorsqu'une primitive échoue au test de profondeur, ce qui permet de marquer les zones de l'écran qui se trouvent dans l'ombre. Lors de l'étape du rendu des boîtes orientées des objets, les requêtes d'occlusion indiquent si au moins un pixel a réussi le test de profondeur *avec* une valeur nulle du stencil buffer, auquel cas l'objet ombre au moins un receveur visible depuis l'observateur. Afin d'affiner les résultats des requêtes d'occlusion, Stich et al [85] effectuent le rendu d'une hiérarchie de boîtes orientées.

Dans un registre différent, Cool [66], ainsi que Chan et Durand [20] proposent une méthode hybride qui combine shadow mapping et volumes d'ombre afin de bénéficier des avantages de chaque technique. Une première passe de shadow mapping sert à calculer l'ombre et à marquer les pixels qui se trouvent en limite d'ombre. Le rendu des volumes d'ombre est ensuite appliqué sur les pixels marqués afin d'obtenir un contour d'ombre précis. La méthode n'est cependant pas exacte puisque des artefacts liés au sous-échantillonnage de la shadow map peuvent apparaître. De manière similaire, Aila et Akenine-Möller [2] appliquent une première passe de volume d'ombres à basse résolution afin de détecter les blocs de pixels en limite d'ombre. Dans une seconde passe à pleine résolution, seuls les pixels appartenant à un bloc en limite d'ombre sont affectés par le rendu des volumes d'ombre.

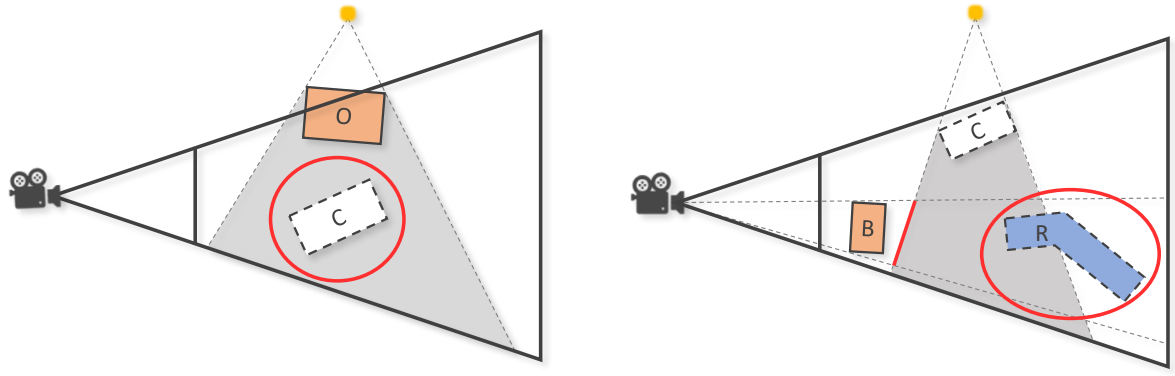


FIGURE 1.19 – Suppression des projeteurs d'ombre non utiles. À gauche : un objet qui projette une ombre peut être ignoré s'il est lui-même inclut dans un volume d'ombre. À droite : Un objet qui projette une ombre qui n'influence aucun objet visible par la caméra peut être ignoré. *Source [30]*

### Volumes d'ombre : Bilan

L'utilisation des données dans l'espace objet permet aux volumes d'ombre de garantir un résultat exact par pixel et de supporter naturellement les sources omnidirectionnelles. Toutefois, leur implémentation ne se prêtre pas aussi bien au matériel des cartes graphiques que le *shadow mapping*. Le coût de la rastérisation des volumes d'ombres varie nettement selon les points de vue et peut devenir prohibitif lorsque d'importants recouvrements se produisent. Qu'elles permettent de réduire le nombre ou la taille des volumes d'ombre à rendre, les améliorations apportées à l'algorithme d'Heidmann ont permis de repousser ses limites mais sans régler le fond du problème. Dans un contexte temps-réel, les baisses de performance sont généralement moins tolérables que des artefacts visuels, et c'est la raison qui a value aux shadow maps d'être privilégiées par rapport aux volumes d'ombre.

### Volumes d'ombre par triangle (*Per-Triangle Shadow Volumes*)

Les volumes d'ombre gardent l'attention des chercheurs car ils offrent un rendu exact par pixel. En 2011, Sintorn et al. [81] cible la cause des problèmes de performance des shadow volumes en proposant une méthode sensiblement différente. Puisque la rastérisation des volumes d'ombre tend à saturer la bande passante GPU, l'idée générale est de les rastériser hiérarchiquement afin d'amoindrir et de lisser le coût. Pour ce, Sintorn et al. commencent par structurer les points image de la caméra en une hiérarchie de frustums de vue. L'écran est pour cela divisé en blocs de pixels de dimension $m \times n$. Les sous-frustums de vue qui englobent les $m \times n$ points image de chaque bloc sont calculés puis regroupés hiérarchiquement. Cette hiérarchie, efficacement calculée sur GPU, est ensuite traversée par chaque volume d'ombre afin de déterminer avec une complexité logarithmique les points images ombrés par le volume d'ombre d'un triangle. Les performances de la méthode sont toutefois altérées lorsqu'il existe une trop grande discontinué de profondeur entre les points image de certains blocs de pixel (figure 1.21). Les frustums qui englobent ces blocs sont allongés dans la direction de vue, ce qui augmente nettement leur probabilité d'être intersectés par un nombre important de volumes d'ombres.

Pour résoudre le problème, Sintorn et al. [80] étend la structure à la 3D en partitionnant

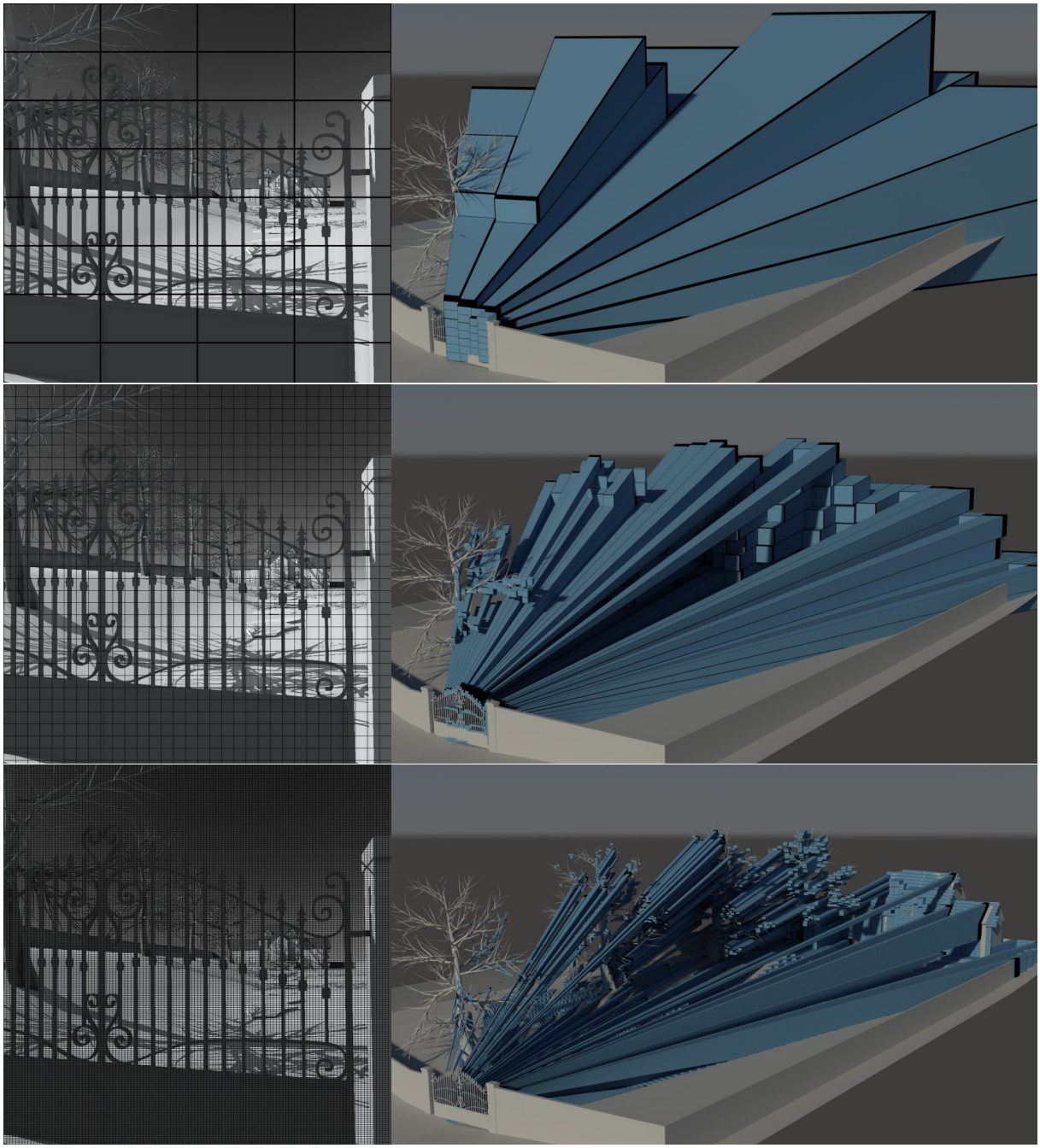

FIGURE 1.20 – De bas en haut, les frustums qui englobent les points images des blocs de pixels sur l'image de gauche sont regroupés hiérarchiquement. *Source [80]*

les frustums le long de la direction de vue (figure 1.21). La méthode s'avère plus rapide et plus stable que les shadow volumes traditionnels. De plus elle ne repose pas sur la silhouette des objets et ne nécessite donc pas de disposer des informations d'adjacence entre les triangles de la scène. La méthode n'est pas naturellement compatible avec le pipeline graphique habituelle. Elle profite néanmoins de la puissance des GPU grâce à une implémentation CUDA. Pour autant il n'en demeure pas moins que tous les volumes d'ombre doivent traverser la hiérarchie construite sur les points image. Par conséquent les performances tendent à baisser avec l'augmentation de la complexité géométrique.

Tout comme les travaux sur la rastérisation irrégulière se sont attaqués à la source des problèmes propres aux shadow maps, les travaux réalisés par Sintorn et al. ont ciblé la source des problèmes propres aux shadow volumes. Dans les deux cas, on peut voir ces méthodes comme les évolutions les plus abouties des principes posés très tôt par Williams et Crow dans la jeune histoire de l'informatique graphique. Si ces deux courants ont largement mobilisé la communauté scientifique ces 40 dernières années, il existe néanmoins une alternative qui n'a certes pas bénéficié d'autant d'intérêt, mais qui se démarque par son originalité.

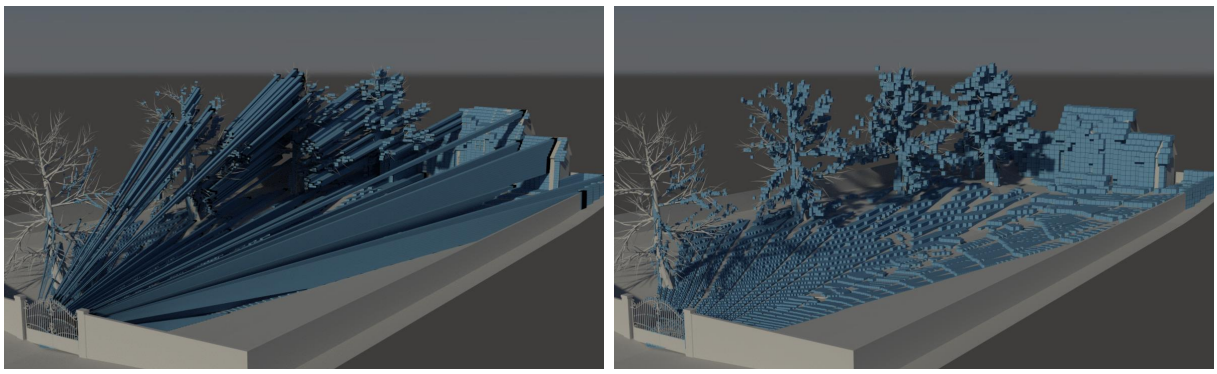

FIGURE 1.21 – Passage à la 3D. *Source [80]*

### 1.2.3 Arbre BSP de volume d'ombre

En 1989, Chin et Feiner [21] proposent eux aussi une méthode qui utilise les volumes d'ombres générés par les triangles de la scène. À la différence des *volumes d'ombre* traditionnels, la localisation des points image est traitée non pas du point de vue de la caméra mais depuis la lumière.

#### Principe de fonctionnement

Inspirés des travaux de Naylor et al. [69], Chin et Feiner proposent de découper l'espace en cellules convexes ombrées ou éclairées en construisant un arbre BSP (ou SVBSP, pour *Shadow Volume Binary Space Partition*) à partir des plans délimitant les volumes d'ombre. Le volume d'ombre d'un triangle est vue comme un polyèdre dont chaque plan partitionne l'espace en deux parties : l'une positive du côté de la normale, l'autre négative du côté opposé. Les plans du volume d'ombre correspondent chacun à un nœud de l'arbre BSP et sont liés entre eux par le côté négatif (figure 1.22). L'union des volumes d'ombres en un seul arbre permet le découpage des zones d'ombre et de lumière : une feuille négative de l'arbre referme une cellule à l'ombre, tandis qu'une feuille positive pointe vers une zone éclairée (voir Figure 1.23).

La structure est construite de manière incrémentale : les triangles sont insérés dans l'arbre puis ajoutés par union de leur volume d'ombre lorsqu'ils atteignent une feuille visible. Lors de l'étape d'insertion, la position d'un triangle est testée par rapport au plan de partitionnement contenu dans chaque nœud de l'arbre. De fait il peut être inclus dans le demi-espace positif, négatif, ou bien intersecté. Dans ce dernier cas, le triangle est subdivisé en fragments (polygones) qui sont insérés dans les sous-arbres correspondants

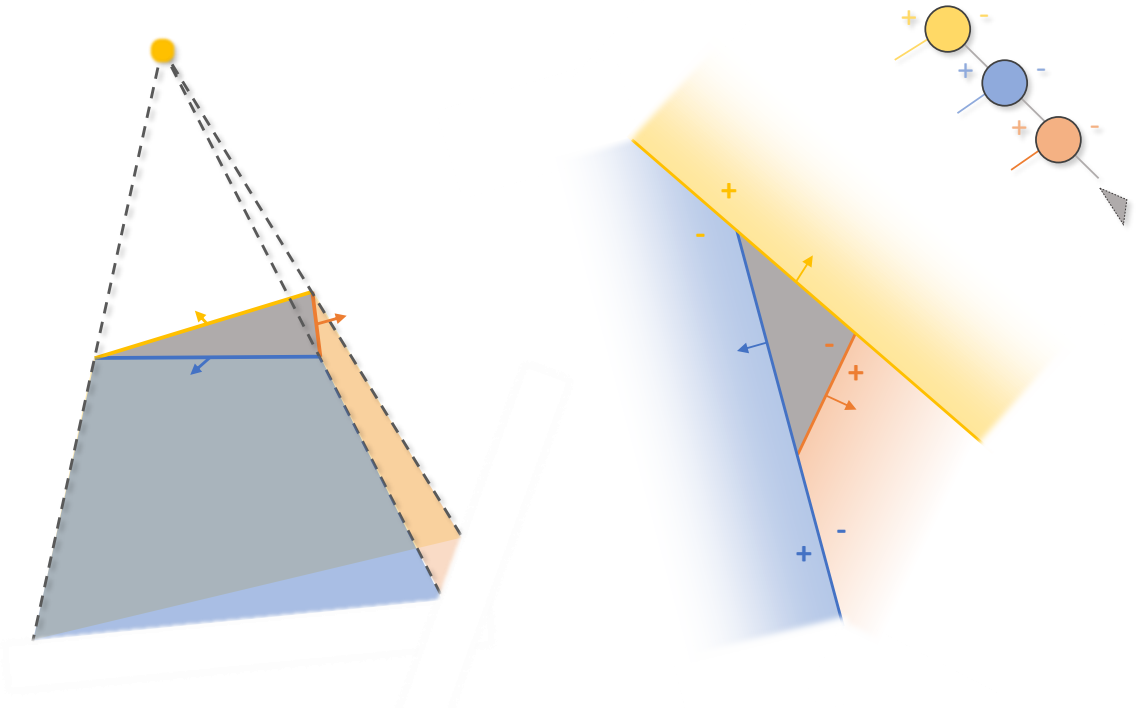

FIGURE 1.22 – À droite : illustration du partitionnement de l'espace par un volume d'ombre vu de la lumière (le plan support du triangle n'est pas représenté mais partitionne également l'espace en deux parties devant/derrière). L'intérieur du volume d'ombre, représenté en gris, correspond à l'union des côtés négatifs des plans du volume, eux-mêmes correspondants à des nœuds consécutifs de l'arbre liés par leur fils négatif.

(voir Figure 1.24). Seuls les fragments visibles sont ajoutés dans l'arbre, et ce en utilisant les plans du volume d'ombre original afin de garantir un nombre de nœuds constant par volume d'ombre. Une fois la structure construite, celle-ci est traversée par les points image afin de déterminer leur visibilité. Le parcours d'un point dans la structure est simple puisque la position du point par rapport à chaque plan de partitionnement indique dans quel sous-arbre poursuivre le parcours. Si le point atteint une feuille négative, il est à l'ombre, sinon il est éclairé (figure 1.23). Sous réserve d'avoir construit un SVBSP relativement équilibré, la méthode possède une complexité logarithmique pour déterminer la visibilité d'un point depuis la source, ce qui est intéressant. La construction du SVBSPs s'avère en revanche bien plus sensible en termes de complexité.

### Une méthode peu robuste

La nécessité de découper les triangles en fragments lorsqu'ils intersectent un plan de partitionnement rend la méthode particulièrement fragile et peu robuste. D'une part, les découpages successifs des fragments par des plans entraînent progressivement une perte de précision numérique due à la représentation discrète des nombre flottants en machine. D'autre part, avec des implications plus contraignantes, la subdivision des triangles en fragments répartis dans des sous-arbres distincts induit un coût mémoire variable et potentiellement exponentiel. Dans le pire des cas, $2^n$ fragments sont générés pour $n$ tri-

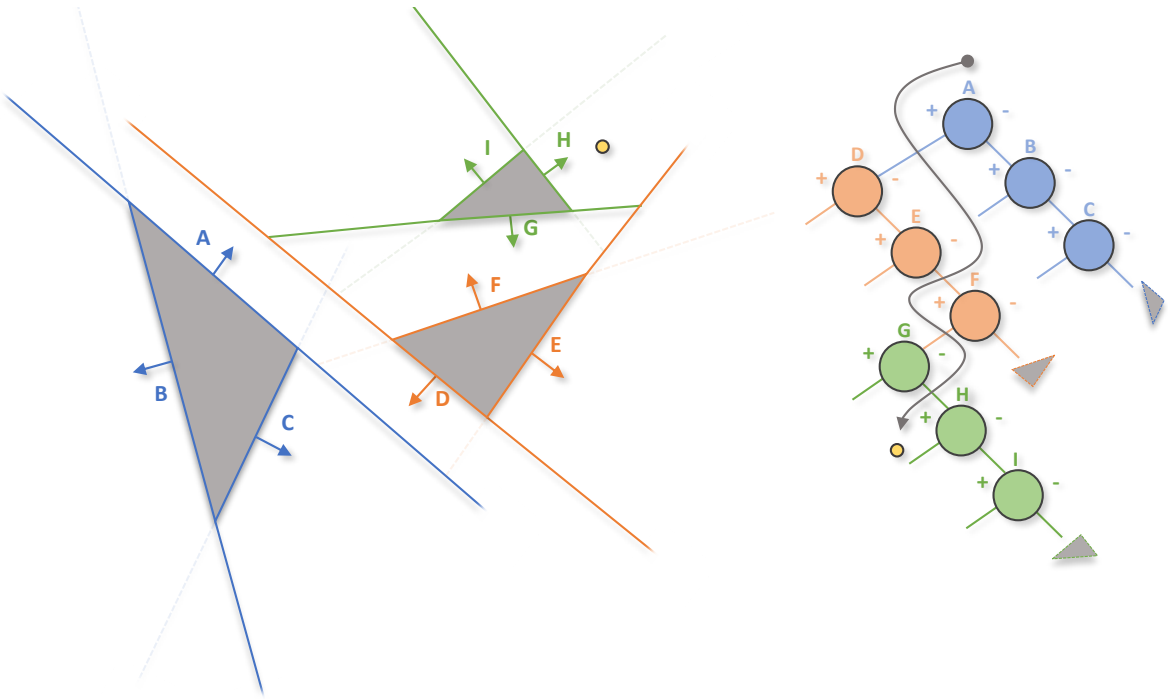


FIGURE 1.23 – À gauche, visualisation de 3 volumes d'ombre dans leur représentation polyédrique vue de la lumière. À droite, les nœuds qui forment les volumes d'ombre sont arrangés dans un arbre BSP (le plan support du triangle, qui referme le volume d'ombre, n'est pas représenté). Le chemin qui part de la racine de l'arbre indique le parcours effectué pour localiser l'échantillon (en jaune).

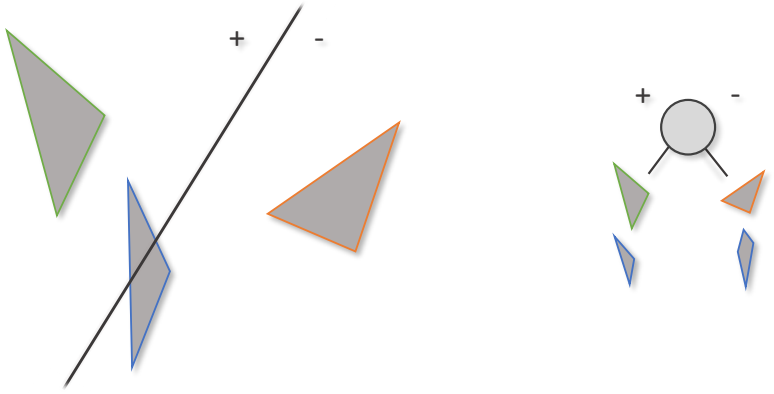


FIGURE 1.24 – Un triangle intersecté par un plan de partitionnement doit être découpé en fragments qui seront insérés dans leur branche respective.

angles, et bien qu'un tel scénario soit rare en pratique, même des scènes de taille modeste peuvent rapidement engendrer un coût mémoire trop important. En plus de la consommation mémoire difficilement prédictible et non contrôlable, la duplication des volumes d'ombre d'un même triangle augmente la taille de l'arbre et impacte les performances des requêtes lors du parcours. Malgré des efforts pour améliorer la méthode, en permettant une construction dynamique de l'arbre BSP [22], ou même en proposant une implémentation

sur cartes graphiques [8], ces différents problèmes ont conduit les SVBSPs à être mis de côté.

### Le partitionnement ternaire, une approche plus robuste

Gerhards et al. [39] revisitent en 2015 la méthode initiée par Chin et Feiner. Conservant le principe général, le partitionnement binaire de l'espace est abandonné au profit d'un partitionnement ternaire de la géométrie : Plutôt que de découper un triangle lors d'une intersection avec un plan de partitionnement et de répartir chaque fragment dans les sous-arbres du nœud correspondant, Gerhards propose d'insérer directement le triangle dans un troisième sous-arbre "intersection" (voir Figure 1.25). La structure d'arbre ternaire ainsi construite (un TOP-Tree, pour *Ternary Object Partitioning*) présente l'avantage de ne nécessiter aucun calcul d'intersection, ce qui permet de résoudre les problèmes qui en découlaient, à savoir l'instabilité numérique et un coût mémoire incontrôlable et potentiellement excessif. Le changement apporté par Gerhards permet au contraire une construction robuste de la structure car celle-ci repose uniquement sur des tests sommet/plan. Il offre également un coût mémoire fixe en $O(n)$ pour $n$ triangles puisqu'aucune géométrie n'est subdivisée ou dupliquée. On peut noter qu'Uhlmann [88] a été le premier à décrire une structure ternaire à partir d'un kd-tree pour partitionner des boites englobantes alignées aux axes.

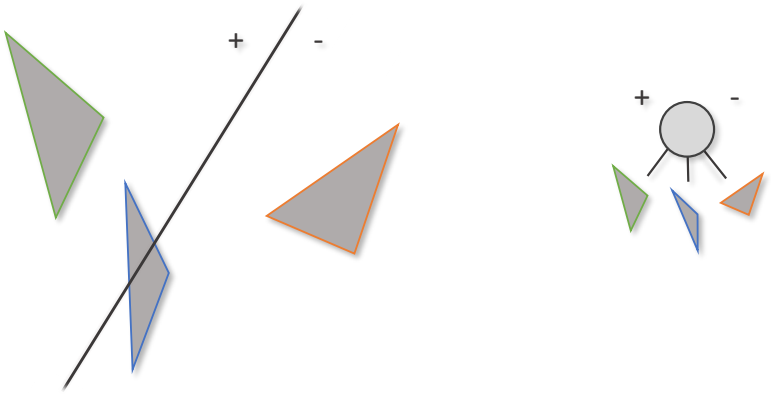


FIGURE 1.25 – Dans un TOP-Tree, un triangle qui intersecte un plan de partitionnement est inséré dans un sous-arbre dédié.

Un TOP-Tree ne doit cependant pas être vue comme un arbre BSP "à trois branches". Comme mentionné précédemment, le premier partitionne les objets quand le second partitionne l'espace. Aussi si chaque plan de partitionnement définit 3 ensembles disjoints de géométrie, des recouvrements existent entre les 3 régions de l'espace qui les contiennent. La géométrie appartenant à un sous-arbre "intersection" se situe par définition à la fois dans les demi-espaces positif et négatif du plan de partitionnement. De ce fait, lorsque la structure est parcourue par un point image, le sous-arbre intersection (s'il existe) doit systématiquement être visité *en plus* du sous-arbre (positif *ou* négatif) auquel le point appartient. Cette visite supplémentaire systématique s'avère coûteuse, mais il est possible de réduire son impact en bornant l'espace couvert par le sous-arbre "intersection" afin d'y limiter l'accès. Pour cela, chaque nœud stocke la distance angulaire

maximale des triangles qui intersectent son plan de partitionnement (figure 1.26), ce qui revient à délimiter la région de l'espace contenant la géométrie intersectée. Ainsi, un point doit visiter un sous-arbre intersection uniquement s'il est inclus dans cette région. Grâce à ce test (conservatif), le parcours d'un TOP-Tree par les points image conserve le comportement logarithmique hérité des SVBSPs.
Le parcours est effectué sur GPU dans un *Fragment Shader* à l'aide d'une pile, et des travaux ultérieurs [68] se penchent sur l'impact de la taille de cette pile sur les performances et proposent une version du parcours sans pile ainsi qu'une version hybride.

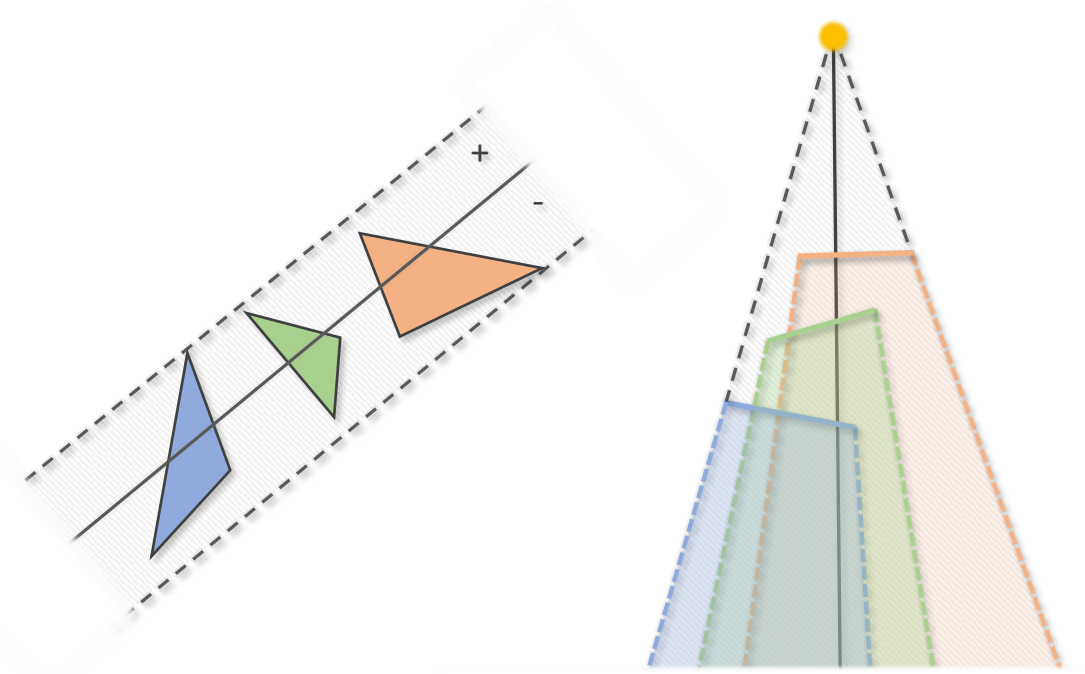


FIGURE 1.26 – Pour chaque nœud d'un TOP-tree, l'angle qui borne la géométrie intersectant son plan de partitionnement est stocké.

### Construction d'un TOP-Tree sur GPU

Comme pour les SVBSPs, un TOP-tree est dépendant de la position des objets et de la lumière (mais pas de la position de la caméra). Il est donc nécessaire dans un environnement dynamique de le mettre à jour à chaque image. Le référencement unique des primitives dans un TOP-Tree permet de le construire efficacement et en place sur GPU. Les auteurs proposent une construction parallèle incrémentale : chaque thread insère un des triangles de la scène jusqu'à épuisement. Le triangle est d'abord localisé dans une feuille où, si c'est une feuille visible, les nœuds représentant son volume d'ombre sont ajoutés. Ce processus peut donner lieu à des accès concurrents lorsque plusieurs threads accèdent simultanément à une même feuille de l'arbre. Ces problèmes sont gérés à l'aide d'opérations atomiques. D'un point de vue algorithmique, cette construction s'apparente à l'algorithme de tri *quicksort*, dont la complexité en temps est au pire en $O(n^2)$ pour $n$ éléments. En pratique, le traitement des éléments dans un ordre aléatoire permet d'éviter cet écueil et d'obtenir une complexité moyenne en $O(nlog(n))$. La construction d'un TOP-

tree suit la même logique. Pour une scène composée de $n$ triangles, la complexité moyenne pour l'insertion d'un triangle dans un TOP-Tree est en $O(log(n))$, donnant ainsi une complexité en $O(nlog(n))$ pour la construction d'un arbre complet. Cela suppose que l'arbre soit relativement équilibré. À l'instar du quicksort, cet équilibre est assuré en insérant les triangles dans un ordre aléatoire. Cela permet en outre d'éviter des scénarios pathologiques (voir Figure 1.27 à gauche) qui pourraient induire une complexité en $O(n^2)$. Il existe néanmoins des configurations géométriques très spécifiques qui conduiront toujours à une complexité en $O(n^2)$ quel que soit l'ordre d'insertion des triangles (voir Figure 1.27 à droite). Leur probabilité d'apparition demeure cependant extrêmement faible. Dans tous les cas, la complexité mémoire demeure en $O(n)$.
En pratique, pour générer une insertion aléatoire de la géométrie, les auteurs ont recours à une permutation des triangles. Cette permutation peut se faire au préalable, par mélange des triangles avant leur transfert sur la carte graphique, ou "à la volée", en travaillant avec des indices de triangles permutés à l'aide d'une fonction dédiée. Par ailleurs, on peut noter que le mécanisme d'ordonnancement des *warps* opéré par la carte graphique, couplé à une insertion concurrente des triangles, introduit en soit une seconde forme d'aléa.

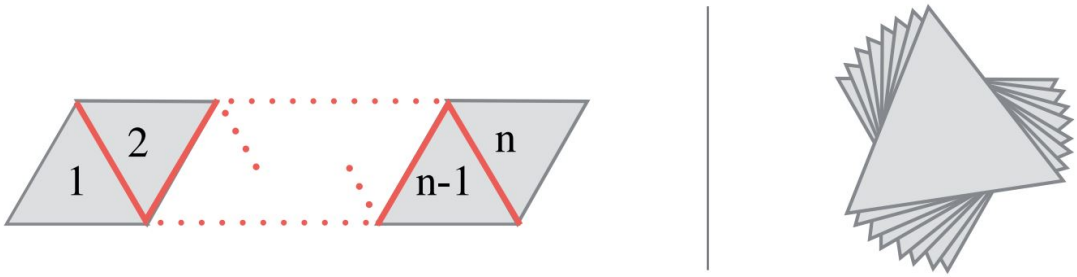


FIGURE 1.27 – Les figures sont vues depuis la lumière. Gauche : Un *strip* de triangles indicés de 1 à $n$. Chaque triangle d'indice $i$ se trouve dans l'espace positif du triangle $i-1$. Si les triangles sont insérés dans l'ordre de leur indice, un arbre de profondeur $n$ sera construit. Le temps de construction serait alors en $O(n^2)$. Le traitement aléatoire des triangles préserve une complexité en $O(n\,log(n))$ en assurant l'équilibre de la structure. Droite : Dans ce scénario, chaque volume d'ombre intersecte tous des triangles. Ici quel que soit l'ordre d'insertion, l'arbre construit sera toujours de profondeur $n$ et aucune mesure ne peut empêcher une complexité en temps en $O(n^2)$. Toutefois, là où un SVBSP donnerait lieu à une consommation mémoire en $O(2^n)$, celle du TOP-Tree reste fixe, en $O(n)$. *Source [39]*

### TOP-Tree : Bilan

Le TOP-Tree offre une alternative intéressante aux volumes d'ombres traditionnels. L'ajout d'un nœud intersection permet à la fois d'éliminer les problèmes de robustesse des SVBSPs, de proposer une construction simple et efficace de la structure sur GPU pour la gestion des scènes dynamiques ; et ce tout en conservant une complexité logarithmique avantageuse lors du parcours. Les performances ainsi obtenues sont généralement supérieures à celles des volumes d'ombre traditionnels (Z-Pass, Z-Fail), et bien que la méthode de Sintorn (voir section 1.2.2) soit plus efficace sur des scènes de petite et moyenne taille, les TOP-Tree se démarquent sur des scènes de taille plus conséquente

(de l'ordre du million de triangles). Malgré cela, la construction de la structure conserve une dépendance linéaire au nombre de triangles dans la scène. Ce qui peut devenir prohibitif et ne permet pas d'envisager le support de larges scènes dynamiques.

### 1.2.4 Lancer de Rayons

Le propos ici n'est pas de faire un état de l'art exhaustif et technique sur le lancer de rayons. Il s'agit de retracer ses grandes évolutions, dont les plus récentes font qu'aujourd'hui le lancer de rayons en temps-réel — dans une certaine mesure — commence à être une option envisageable pour des cas pratiques encore limités mais dont les ombres dures peuvent faire partie. Ces développements récents ont compté dans la dernière partie de cette thèse.

Les algorithmes basés sur le lancer de rayons sont coûteux, mais incontournables pour produire des images physiquement réalistes. À l'origine, la méthode proposée par Whitted [94] en 1980 n'est pas physiquement réaliste, mais elle permet déjà de traiter des phénomènes de réflection/réfraction ainsi que les éclairages directs. En intégrant plusieurs rayons depuis un même point, la version distribuée de Cook [24] permet d'ajouter des effets tels que les ombres douces, les réflections floues, la profondeur de champ ou encore le flou de mouvement. Finalement, avec l'équation intégrale du rendu [50], Kajiya généralise la notion de transport de la lumière pour reproduire l'ensemble des chemins parcourus en tenant compte des propriétés physiques des matériaux avec lesquels les rayons lumineux interagissent. Résoudre l'équation de rendu permet d'obtenir un photo-réalisme jusqu'ici jamais atteint. Cette résolution se fait principalement de manière stochastique (méthodes de Monte-Carlo). Pour cela, le *path tracing* [54] (et ses variantes) est probablement l'un des algorithmes le plus connus et utilisés en pratique parmi l'ensemble des algorithmes dits d'illumination globale.

Jusqu'à la fin des années 90, le coût très important de ces méthodes les a confinées à une utilisation *offline*. Mais au début des années 2000, la puissance des CPUs atteint un niveau tel qu'il devient envisageable de visualiser des scènes statiques de manière interactive. Il s'agit alors de visualisation directe avec un modèle d'illumination basique et simplifié. Les jeux d'instruction *SIMD* (*Single Instruction Multiple Data*) des CPUs, auparavant réservés aux nombres entiers, s'étendent aux nombres flottants [75]. Il devient alors possible d'effectuer des calculs sur quatre rayons pour le prix d'un seul. En formant des paquets de 4 rayons cohérents, la bande passante peut être divisée par 4 est les performances du parcours sont au moins doublées [10, 91, 92]. Afin de lever la limite imposée par le matériel quant au nombre de rayons tracés simultanément, Reshetov [76] et Benthin [9] proposent de former des paquets de taille quelconque et de tracer leur frustum englobant dans la structure. Bien que le tracé de paquets offre des performances interactives (pour des scènes statiques) et qu'il soit possible de l'adapter au lancer de rayon de Cook et Whitted [14] pour les premiers rebonds, il reste inefficace pour les rayons secondaires et incohérents indispensables aux algorithmes d'illumination globale. Or à la fin des années 2000, l'industrie du jeu vidéo s'oriente résolument vers plus de réalisme. Avec la transition des modèles d'éclairage empiriques vers des modèles basés physique [47], il devient clair que la rastérisation et le cortège de techniques ad hoc nécessaires pour pallier ses limites ne pourront atteindre le niveau de photoréalisme souhaité. Seul le lancer de rayon semble capable d'offrir un tel réalisme. Mais pour cela, le parcours des rayons

secondaires incohérents doit être performant, et aux alentours de 2010 la problématique change pour se focaliser sur l'optimisation du parcours d'un seul rayon [11, 27, 33, 93]. Dans le même temps, le lancer de rayons sur GPU est en plein essor. Les performances du parcours commencent à rattraper [4, 5] celles des algorithmes CPU, puis finissent par les dépasser [12, 61, 72, 100]. Le GPU est également utilisé pour augmenter la vitesse de construction des structures d'accélération, notamment les BVHs (*Bounding Volume Hierarchy*), préférés aux *kd-trees* [73, 74] pour leur plus grande flexibilité malgré des performances équivalentes [90]. Le LBVH (*Linear BVH*) de Lauterbach et al. [57] sert de base à ces différentes avancées. L'arbre est construit très rapidement en hiérarchisant les triangles selon leur code de Morton, mais il est de moins bonne qualité que les meilleurs BVHs [84] construits avec l'heuristique SAH (*Surface Area Heuristic*) [45]. Le HLBVH (*Hierarchical LBVH*) [37, 71] est une reformulation du LBVH. Il permet une construction plus rapide de l'arbre, avec une empreinte mémoire réduite et la possibilité d'incorporer le SAH durant la construction des premiers étages. Karras et al. [29, 51] proposent quant à eux un algorithme pour optimiser un arbre déjà existant. En partant d'un HLBVH, des groupes de nœuds contenus dans un voisinage donné (des *treelets*) sont réarrangés selon l'heuristique du SAH. Le TRBVH résultant est de très bonne qualité, équivalent aux meilleurs BVHs construits sur CPU, et son temps de construction rapide (bien que plus lent que celui d'un HLBVH et moins adapté pour une utilisation temps-réelle) fournit le meilleur ratio entre le temps de construction et le temps de parcours.

Toutes ces améliorations ont été concrétisées fin 2018 par l'intégration du lancer de rayons dans les APIs graphiques 3D [83, 87] aux côtés du pipeline de rastérisation. Au moment où ces lignes sont écrites, un seul constructeur (NVIDIA) propose des pilotes qui intègrent un support pour ces APIs de lancer de rayons GPU. Grâce à cela, les développeurs peuvent envisager de lancer quelques rayons par pixel. Mais c'est surtout avec les dernières générations qui intègrent en plus des cœurs de calcul dédiés pour un support matériel du lancer de rayons. Le budget (nombre de rayons par pixel et par image) reste limité et il s'agit surtout de trouver comment combiner lancer de rayons et rastérisation afin de tirer le meilleur parti des deux. On peut noter que dans l'immédiat, les autres constructeurs (dont AMD, premier concurrent de NVIDIA) ne supportent pas les API de lancer de rayons. Ni au niveau logiciel, ni au niveau matériel. Il semble toutefois inéluctable que ce support se généralise dans les années à venir.

## 1.3 Conclusion

Les ombres sont nécessaires au réalisme des images de synthèse, si bien que les premiers algorithmes pour le rendu de scènes tridimensionnelles adressaient déjà cette problématique. Deux concepts introduits à la fin des années 70 ont nourri ce domaine de recherche durant les décennies suivantes. D'un côté, les *shadow maps* souffrent de nombreux artefacts visuels dus à la projection de la géométrie sur un écran, mais elles offrent d'excellentes performances grâce au support matériel des cartes graphiques. De l'autre côté, les *volumes d'ombre* produisent un résultat visuel exact par pixel en exploitant la géométrie de la scène dans son espace original (l'espace objet). Mais ils sont ainsi plus coûteux et moins stables que les shadow maps. Dans un contexte temps-réel, les shadow maps ont généralement été privilégiées pour leur vitesse et pour la garantie qu'elles offrent de respecter un budget de temps et de mémoire alloué. De nombreux travaux ont

donc tenté de corriger les artefacts liés à leur utilisation, sans y parvenir totalement. La précision passe nécessairement par l'utilisation de l'espace objet, comme c'est le cas pour la méthode de Sintorn (PTSV), celle de Gerhards (PSV), mais aussi celle de Wyman (IZB) — qui se rapproche au final plus des PTSV de Sintorn que des shadow maps classiques (une structure d'accélération construite sur les points images est parcourue par les triangles de la scène pour effectuer un test géométrique rayon-triangle/frustum-point). Ces différentes méthodes ont amené les techniques d'ombre dures exactes à un niveau de performance jusque-là inégalé. Elles conservent malgré tout une dépendance linéaire à la complexité géométrique qui les rends progressivement inefficaces pour des scènes de grande taille. L'enjeu de notre travail est de dépasser cette limitation.

Dans la suite de ce mémoire, nous repartons des travaux de Gerhards pour concevoir un algorithme de rendu d'ombre dure temps-réel capable de supporter une forte complexité géométrique. Face aux shadow maps et aux shadow volumes "traditionnels", qui ont été explorés de manière extensive au cours des 40 dernières années, les SVBSPs et leur variante ternaire les TOP-tree restent un terrain relativement inexploré. La complexité algorithmique du parcours d'un TOP-tree (en $O(log(n))$ pour $n$ triangles) est déjà apte à supporter une montée en complexité géométrique. Cela contraste avec les PTSV de Sintorn : Les requêtes de la méthode, bien que logarithmiques (en $O(log(p))$ pour $p$ pixels), sont en quantité proportionnelle au nombre de triangles présents dans la scène. De ce point de vue, les TOP-tree offrent un meilleur point de départ pour gérer la montée en complexité géométrique. C'est donc en particulier à la dépendance linéaire de la construction d'un TOP-tree que s'intéresse le chapitre 2.

Le troisième chapitre se veut davantage exploratoire. Tout en reprenant les travaux développés dans le chapitre 2 pour supporter une complexité géométrique élevée, nous y étudions une nouvelle stratégie de partitionnement basée sur les arbres métriques. Ce type de structure n'a jamais été utilisée en informatique graphique. Notre motivation est de créer une méthode qui intègre naturellement la notion de distance à la géométrie. Cela permettrait d'ouvrir ensuite différentes perspectives, par exemple pour traiter l'aliasing des ombres dues à de petits objets. C'est en général une faiblesse commune à toutes les techniques présentées dans cet état de l'art. L'hypothèse est toujours faite que l'ombre déterminée via le centre d'un pixel vaut pour tout point observé à travers ce même pixel. Cette hypothèse trouve rapidement ses limites, en particulier lorsqu'il est question de petits objets dont l'ombre ne couvre que partiellement la surface d'un pixel.

# Chapitre 2 :

# Un algorithme basé sur les volumes d'ombre et robuste à la complexité géométrique

## 2.1 Introduction

### Problématique

Dans le précédent chapitre, l'état de l'art sur les ombres dures en temps réel permet d'observer que :

— Les algorithmes opérant en espace image ne sont jamais exacts par pixel en raison de nombreux problèmes d'aliassage

— Pour être exact par pixel, il est nécessaire d'opérer dans l'espace objet, auquel cas quelles que soient les méthodes existantes, il existe toujours une dépendance linéaire au nombre de triangles à traiter. Par conséquent il y a nécessairement un point de bascule où les performances se dégradent en deçà des exigences du temps réel.

Ce dernier point est inhérent aux algorithmes basés sur la géométrie. Leur utilisation apparaît donc incompatible avec des scènes contenant plusieurs dizaines de millions de triangles. C'est ce problème que nous nous proposons d'étudier dans ce chapitre : Comment proposer un algorithme d'ombrage exact par pixel qui demeure temps réel sur des scènes comportant un nombre très important de triangles ?

### Approche proposée

Pour répondre à ce problème, l'approche proposée par Gerhards est apparue comme le point de départ à privilégier. Purement géométrique, l'algorithme est exact par pixel et présente de bonnes performances. Surtout, nous nous intéressons à la complexité en temps de calcul des deux étapes de l'algorithme. À chaque image et pour $n$ triangles :

— Première étape : $n\,log(n)$ pour la construction de la structure (un TOP-tree)

— Seconde étape : parcours de la structure en $log(n)$ pour chaque point de l'image

En soit, la seconde étape ne pose aucun problème si le nombre de triangles augmente. C'est la construction de la structure qui finira toujours par excéder les contraintes de temps nécessaires au rendu temps-réel.

Ce que l'on propose dans un premier temps, c'est de baser la construction non pas sur des triangles mais sur des *clusters* (groupes) de triangles. C'est une stratégie relativement courante face à un très grand volume de données. Si la construction coûte $O(n\,log(n))$ pour $n$ triangles, un arbre construit avec des clusters de $p$ triangles sera calculé en $O(\frac{n}{p}\,log(\frac{n}{p}))$. La complexité intrinsèque n'est pas changée ($p$ est une constante), mais on peut espérer réduire le temps de calcul de la structure d'un facteur directement dépendant de la taille $p$ des clusters.

### Limites et enjeux

L'utilisation de clusters n'est pas sans conséquence et introduit d'emblée une limite : Les algorithmes de clustering sont coûteux, et il n'est pas envisageable de les calculer à chaque image, tout particulièrement sur des dizaines de millions de triangles. Ainsi, cette approche ne pourra pas traiter des environnements 100% dynamiques. Elle ne sera pas pour autant limitée à des environnements statiques et supportera toutes les transformations euclidiennes.

Comme nous le détaillerons pas la suite, une autre conséquence porte sur le parcours de la structure. Au lieu de déterminer si un triangle masque la source depuis un point, nous

allons trouver si un cluster masque *potentiellement* la source. Si c'est le cas, il faudra alors tester les triangles du cluster. On perçoit ainsi que l'utilisation des clusters va induire un surcoût sur le parcours de la structure. Du point de vue des performances, l'enjeu sera de gagner davantage de temps à la construction de la structure que l'on acceptera d'en perdre lors de son parcours.

## 2.2 Approche proposée

### 2.2.1 Principales étapes de l'algorithme

Nous commençons par décrire brièvement les grandes étapes de notre approche. Chacune sera détaillée ultérieurement.

#### Clustering et volumes englobants de clusters

Le clustering est une étape précalculée une seule fois pour chaque scène. Regrouper la géométrie en un nombre réduit de groupes ou clusters est une solution pour gérer une complexité géométrique élevée. Nous appliquons ce principe en regroupant les triangles en clusters contenant jusqu'à 32 triangles. Ensuite, nous calculons pour chaque cluster un volume englobant. Il est important que les volumes englobants ne soient pas trop conservatifs par rapport à la géométrie qu'ils contiennent. Dans le même temps, il est souhaitable d'utiliser des formes relativement simples, plus compactes en mémoire et plus efficaces à manipuler, notamment pour des tests d'occultation. Ainsi nous utilisons des boites orientées. Nous calculons également un cône de normales pour chaque cluster, ce qui permet d'étendre le principe de "suppression des faces avants ou arrières" (*front/back face culling*) à l'ensemble d'un cluster.

#### Du volume d'ombre d'un triangle au volume d'ombre d'un cluster

L'intérêt d'une boite orientée est aussi que le volume d'ombre qu'elle génère demeure un volume convexe délimité par des plans formés par la position de la source lumineuse et les arêtes silhouettes de la boite vue depuis cette source (plus les plans de la boite faisant face à la lumière). Cette description reste donc compatible avec la structure de TOP-tree proposée par Gerhards qui, rappelons-le, est une évolution ternaire d'un arbre BSP. En revanche, selon le point de vue, le nombre d'arêtes silhouettes peut varier et avec lui, le nombre de plans nécessaires à décrire le volume d'ombre engendré. Aussi nous proposons une représentation simplifiée du volume d'ombre d'une boite orientée. Au prix d'un léger sur-conservatisme cela permet de toujours utiliser le même nombre de plans et de préserver une empreinte mémoire fixe pour la structure de TOP-tree.

#### Construction d'un TOP-tree avec des clusters

À chaque image, une structure de TOP-tree est construite. Cette construction ne diffère pas fondamentalement de l'algorithme proposé par Gerhards. Toutefois les données en entrée sont les boites orientées englobantes des clusters et non plus des triangles. Or les boites comme les triangles sont des convexes. Par conséquent, tester leur position par rapport au plan d'un nœud consiste toujours à tester la position des sommets du

convexe par rapport à ce plan. Lorsqu'une boite atteint une feuille, celle-ci est remplacée par le sous-arbre représentant le volume d'ombre de la boite. Cependant, une feuille correspondant à l'intérieur du volume d'ombre d'une boite ne représente pas une région de l'espace totalement à l'ombre. La boite étant conservative, son volume d'ombre l'est aussi. Ces feuilles contiennent donc un index vers l'ensemble des triangles du cluster associé afin de pouvoir tester exactement la géométrie lors du parcours.

#### Parcours d'un TOP-tree de clusters

À chaque image, un parcours de l'arbre est effectué pour chaque point image afin de déterminer si celui-ci se trouve à l'ombre ou s'il est éclairé. Chaque nœud de l'arbre contient un plan qui participe à délimiter le volume d'ombre d'une boîte orientée. Lorsqu'un point rencontre un nœud, l'algorithme teste sa position par rapport au plan qu'il contient. Lorsqu'une feuille "négative" est atteinte, le point se trouve à l'intérieur du volume d'ombre d'une boite orientée. L'algorithme accède alors aux triangles du cluster et un test rayon/triangles est effectué avec chacun. Si le point est à l'extérieur du volume englobant, les triangles du cluster ne sont pas testés et le parcours du point se poursuit dans l'un des fils du nœud courant. Le processus se termine lorsqu'une intersection est trouvée (le point est à l'ombre) ou bien lorsqu'une feuille "positive" est atteinte (le point est éclairé).

#### Double hiérarchie de clusters

Enfin nous présentons une évolution de notre approche vers une structure qui s'appuie sur deux niveaux de clusters. Nous montrons que cette solution permet d'obtenir des meilleures performances notamment du fait d'une construction plus efficace.

Nous détaillons à présent chacune des étapes précédentes.

### 2.2.2 Clustering et volumes englobants de clusters

Notre méthode demeure valide quel que soit l'algorithme de *clustering* utilisé. Toutefois, la qualité (taille, régularité) des clusters produits pourra avoir une incidence sur les performances globales car le conservatisme des boites englobantes n'est pas indépendant de la "forme" des clusters calculés.

Dans cette section, nous présentons les deux principales méthodes que nous avons testées pour produire des clusters de triangles. La première, un *clustering* hiérarchique en $k$-moyenne, s'appuie sur l'état de l'art de travaux développés dans un contexte se rapprochant du notre. La seconde est davantage exploratoire et propose de "détourner" un algorithme d'optimisation de la cohérence des sommets des triangles en mémoire pour produire des clusters. Enfin, nous proposons une optimisation basée sur un tri de Morton pour produire rapidement un premier niveau de clusters permettant d'accélérer l'une ou l'autre des méthodes présentées.

#### Formation de clusters par application hiérarchique des $k$-moyennes

Partitionner un ensemble de données est une problématique classique que l'on retrouve dans de nombreux domaines d'applications comme l'analyse statistique de données, le trai-

tement d'images, etc... Dans notre cas, les données que l'on cherche à regrouper sont des triangles. Différents algorithmes permettent d'obtenir des clusters de triangles. Les algorithmes d'approximation de surface variationnelle cherchent à approcher des pans entiers d'un maillage par un minimum de formes géométriques [23]. Par exemple avec des plans, mais aussi avec des sphères ou des cylindres qui permettent d'approcher plus efficacement des surfaces courbes [97]. L'enjeu est alors d'identifier des groupes de triangles formant une surface approchant celle d'une forme simple (plan, sphère ou cylindre). En cherchant à minimiser le nombre de formes simples, l'algorithme tend aussi à maximiser le nombre de triangles par cluster. Dans notre cas, l'objectif est de former de petits clusters de triangles tout en minimisant leur volume englobant, il ne s'agit pas d'approcher la surface qu'ils représentent. Notre problématique est davantage comparable à celle rencontrée dans un contexte de lancer de rayons. Garanzha [36] propose de construire un BVH sur des clusters de triangles. Pour cela, les triangles sont regroupés dans des sphères en fonction d'une heuristique impliquant leur taille et leur densité. Cette approche requiert cependant d'obtenir les relations d'adjacence entre les triangles. Meister *et al.* [67] utilisent un partitionnement hiérarchique en $k$-moyennes ($k$-*mean*) pour former des clusters dans le but d'accélérer la construction d'un BVH sur GPU. Le partitionnement en $k$-moyennes, tout comme celui de Garanzha, tend à produire des clusters réguliers et évite les formes en "L" ou plus généralement concaves qui tendent à laisser des espaces vides dans les volumes englobants, accentuant leur conservatisme. Puisque nous travaillons déjà sous l'hypothèse de modèles non déformables, il deviendrait trop contraignant d'ajouter en plus la connaissance des relations d'adjacence entre triangles nécessaire à la méthode de Garanzha. [36]. Aussi nous avons privilégier le partitionnement hiérarchique en $k$-moyennes tel que proposé par Meister *et al.*.

Nous commençons par rappeler les principales étapes du partitionnement en $k$-moyennes dans sa version non hiérarchique : Étant donné un ensemble d'éléments à partitionner et un nombre entier $k$, la méthode des $k$-moyennes cherche à former $k$ groupes (ou *clusters*) d'éléments de telle sorte que la distance des éléments au centre de leur cluster soit minimale. En d'autres termes, la variance des distances des éléments au centre du cluster, c'est-à-dire la somme des *distances élément-centre au carré*, doit être minimale pour chaque cluster.

Ce problème est connu pour être NP-difficile. De fait, on utilise en pratique des algorithmes qui permettent d'approcher efficacement la solution exacte. L'un des plus utilisés est l'algorithme itératif de Lloyd [65] (voir l'algorithme 1). Tout d'abord, $k$ centres sont choisis parmi les $n$ éléments. Chaque élément est alors assigné au centre dont il est le plus proche, puis la position des centres est mise à jour pour devenir le barycentre des éléments contenus. L'algorithme converge en plusieurs itérations vers une solution où chaque centre est le barycentre d'un cluster qui correspond à une cellule de Voronoï. Une meilleure distribution des centres initiaux permet de réduire le nombre d'itérations nécessaires et d'améliorer la convergence de l'algorithme. Ainsi, la variante $k$-mean$_{++}$ de Arthur et al. [7] sélectionne les centres initiaux de manière à maximiser les distances inter-centres. De cette façon, les auteurs montrent que l'algorithme de Lloyd converge mieux et plus rapidement qu'en choisissant les centres initiaux de manière aléatoire.

Malgré tout, la complexité en temps d'une itération de l'algorithme de Lloyd, en $O(n * k)$ pour $n$ éléments et $k$ clusters reste problématique pour des valeurs de $k$ importantes. Nous sommes dans ce cas puisque nous cherchons à former de petits clusters (donc nombreux) à partir d'un nombre de triangles élevé. Si le nombre de clusters $k$ tend vers

le nombre d'éléments $n$, une itération tend vers une complexité en $O(n^2)$. Par exemple, le partitionnement d'un modèle constitué de 8 millions de triangles en clusters de 16 triangles nécessiterait une valeur de $k = 500\,000$.

S'agissant d'un pré-calcul pour notre approche, le temps consacré à créer les clusters n'est pas critique en soi. On ne souhaite pas pour autant que cette étape se mesure en jours ou en heures, particulièrement dans le cas de modèles de très grande taille. Afin d'éviter une complexité qui tendrait à devenir polynomiale et de garantir des temps de calcul raisonnables, une solution consiste à appliquer l'algorithme de Lloyd de manière hiérarchique. La procédure est d'abord appliquée avec une valeur de $k$ raisonnable ($e.g$ $k = 4$). Elle est ensuite reconduite dans chacun des clusters obtenus avec $k = min(k, \frac{tailleDuCluster}{tailleCible})$, et ce tant que la taille du cluster est supérieure à l'objectif fixé. Au final, les clusters qui seront utilisés sont les feuilles de la hiérarchie de clusters ainsi construite.

**Algorithme 2** Algorithme itératif de Lloyd pour partitionner un ensemble d'éléments selon la méthode des $k$-moyennes.

```
1:  procedure KMoyennesIter(Liste<Element> elements, Liste<Centre> centres,
    Entier k, Entier maxIterations)
2:
3:      centres ← ChoisirCentres(elements, k)        ▷ Choisit aléatoirement k centres
4:                                                   ▷ parmi les n éléments
5:      Entier nbIterations ← 0
6:
7:      tant que nbIterations < maxIterations faire
8:
9:          pour tout Centre c_i ∈ centres faire
10:             SupprimerElements(c_i.elements)
11:         fin pour
12:
13:         pour tout Element e_i ∈ elements faire
14:             Centre centre ← PlusProcheCentre(e_i, centres)
15:             AjouterElement(centre, e_i)
16:         fin pour
17:
18:         pour tout Centre c_i ∈ centres faire
19:             c_i.pos ← Barycentre(c_i.elements)
20:         fin pour
21:
22:         nbIteration ← nbIteration + 1
23:     fin tant que
24:
25:     ▷ Chaque centre contient maintenant une liste d'éléments qui forment un cluster
26:
27: fin procedure
```

En pratique, nous appliquons l'algorithme sur la boîte englobante des triangles comme proposé par Meister *et al.* [67]. Nous utilisons également la variante $k$-mean$_{++}$ pour améliorer la distribution des centres initiaux. La distance entre les boîtes englobantes

$\mathbf{b}_1$ et $\mathbf{b}_2$ de deux clusters correspond à la somme du carré des distances Euclidiennes entre les sommets aux extremums de chaque boîte :

$$d(\mathbf{b}_1, \mathbf{b}_2) = \left\|\mathbf{b}_1^{min} - \mathbf{b}_2^{min}\right\|^2 + \left\|\mathbf{b}_1^{max} - \mathbf{b}_2^{max}\right\|^2$$

À la fin de chaque itération, un cluster $C_i$ contenant $|C_i|$ éléments devient la boîte englobante moyenne des éléments qu'il contient :

$$C_i = \frac{1}{|C_i|}\left(\sum_{\mathbf{b}_j \in C_i} \mathbf{b}_j^{min}, \sum_{\mathbf{b}_j \in C_i} \mathbf{b}_j^{max}\right)$$

La figure 2.1 montre un exemple de clusters obtenus en appliquant la version hiérarchique de l'algorithme des $k$-moyennes (dans sa variante $k$-mean$_{++}$).

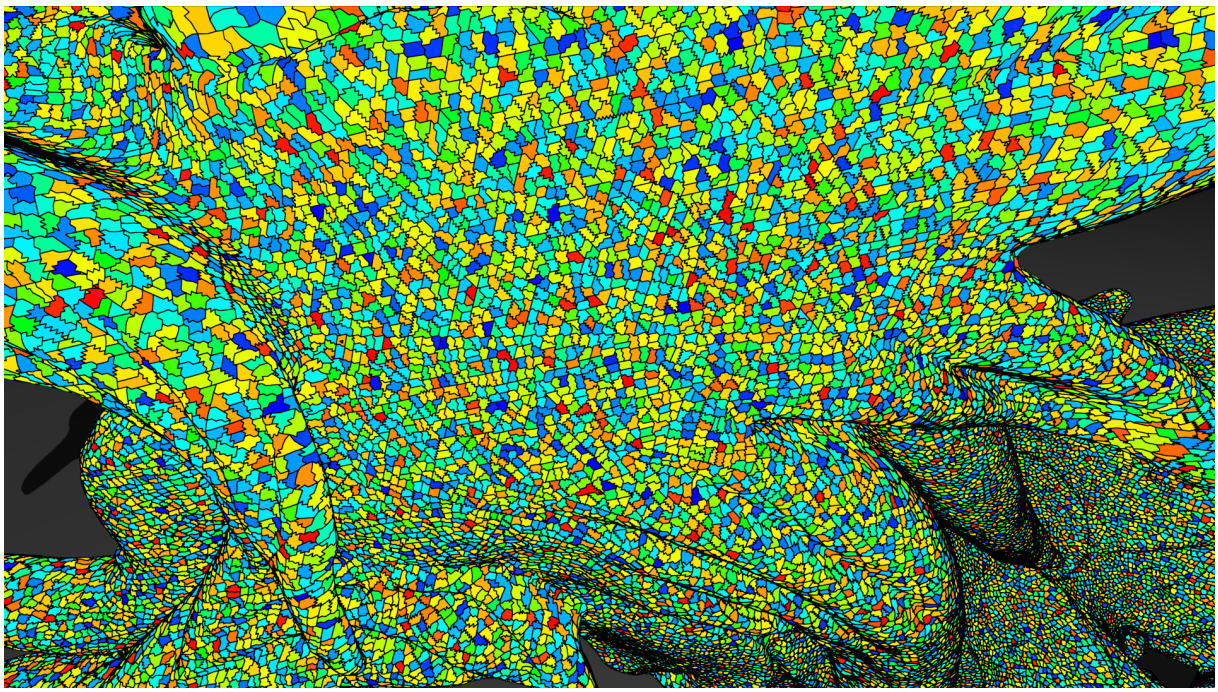

FIGURE 2.1 – Clusters issus d'une application hiérarchique de l'algorithme des $k$-moyennes.

### Clustering par découpage du *buffer* d'indices de sommets

Nous présentons à présent notre seconde méthode qui exploite la cohérence des données en mémoire pour produire des clusters.

Une manière simple de créer des clusters de $n$ triangles est de découper le *buffer* d'indices tous les $n*3$ indices (chaque triplet d'indices pointe vers les sommets d'un triangle). On peut toutefois fortement douter de la cohérence spatiale des triangles présents dans de tels clusters. À moins de faire en sorte que des indices consécutifs de sommets pointent vers des triangles voisins (figure 2.2). Les moteurs $3D$ et les pipelines de créations de contenu temps-réel ont justement recours à des algorithmes d'optimisation du maillage qui augmentent la cohérence spatiale des sommets. Ces algorithmes cherchent à accélérer

l'étape de traitement des sommets (*vertex processing*) du pipeline de rastérisation : Durant cette étape, les sommets des triangles sont récupérés puis transformés dans un espace donné. Un cache de sommets (*post-transform cache*) est utilisé pour éviter de transformer plusieurs fois les sommets redondants [48]. Les algorithmes d'optimisation de maillages ont pour but de réordonner les indices des sommets de manière à augmenter la cohérence des données dans le cache [35, 62, 77], ce qui a pour effet d'augmenter la cohérence spatiale des sommets du maillage. On peut noter que des travaux récents ont montré que la notion de cache de sommets "global" (utilisé pour l'ensemble des sommets traités) qui servait jusque-là d'abstraction au fonctionnement spécifique des différentes cartes graphiques n'est plus d'actualité sur les GPUs modernes [52]. Dans ces environnements hautement parallèles, le traitement des sommets est effectué *par lots* sur les différentes unités de calcul avec un cache indépendant. Les algorithmes existants restent toutefois très efficaces. De plus, ce qui nous intéresse ici est la haute cohérence spatiale des données apportée par ces algorithmes, indépendamment de leur impact sur l'optimisation du cache. L'algorithme de Forsyth [35] (*Linear Vertex Cache Optimization*) est l'un des plus utilisé dans l'industrie de par ses bons résultats et sa complexité linéaire face au nombre de triangles. C'est l'algorithme que nous utilisons pour augmenter la cohérence spatiale des données dans notre *buffer* de sommets.

Une fois l'algorithme d'optimisation de cache appliqué, il ne reste plus qu'à subdiviser régulièrement le *buffer* pour obtenir des *clusters* de la taille souhaitée. La figure 2.2 illustre le procédé. Ce découpage du buffer de triangles en tranches régulières peut toutefois produire des clusters avec des triangles disjoints (lorsqu'il y a un saut dans le buffer d'un groupe de triangles contigus à un autre, sans qu'ils soient tous deux voisins). Nous traitons ce problème en redécoupant chaque tranche en groupes de triangles contigus de manière à ce que chaque triangle possède au moins un sommet en commun avec les autres triangles du groupe. Pour cela, les triplets d'indices des sommets de la tranche sont parcourus et ajoutés à une liste de *patchs* à condition qu'un des sommets du triplet soit déjà présent dans cette liste. Si aucun des sommets n'est présent dans une des listes existantes, une nouvelle liste est créée et les trois indices y sont insérés. À la fin, les listes de *patchs* forment de nouveaux clusters qui sont garantis de contenir des triangles contigus. La figure 2.3 montre un exemple de clusters obtenus en utilisant cette méthode.

### Optimisation de l'étape de clustering à l'aide d'un tri de Morton

Un autre moyen d'augmenter la cohérence spatiale des sommets avant de découper le buffer d'indices en tranches et de trier les indices des triangles selon la courbe de Morton (figure 2.4). Cette méthode s'avère toutefois problématique pour nous car la courbe de Morton couvre un volume et non la surface de l'objet. Ainsi, des triangles voisins sur le maillage peuvent être éloignées le long de la courbe : Après avoir intersecté un triangle en surface, la courbe peut "retourner" vers l'intérieur du volume, trouver un ou plusieurs autres triangles sur une surface opposée, puis revenir intersecter un triangle voisin du premier. Les discontinuités que cela induit dans le buffer d'indices s'accentuent lorsque la résolution du maillage augmente. Dans ce cas, le nombre de triangles voisins consécutifs devient trop faible pour former des tranches de taille suffisante. Ce phénomène est visible sur la figure 2.5, où l'on peut voir que de petits clusters s'intercalent entre les plus gros clusters de taille souhaitée. Augmenter la résolution de la grille utilisée pour calculer le code de Morton a peu d'impact sur le problème : Nous avons comparé une grille de

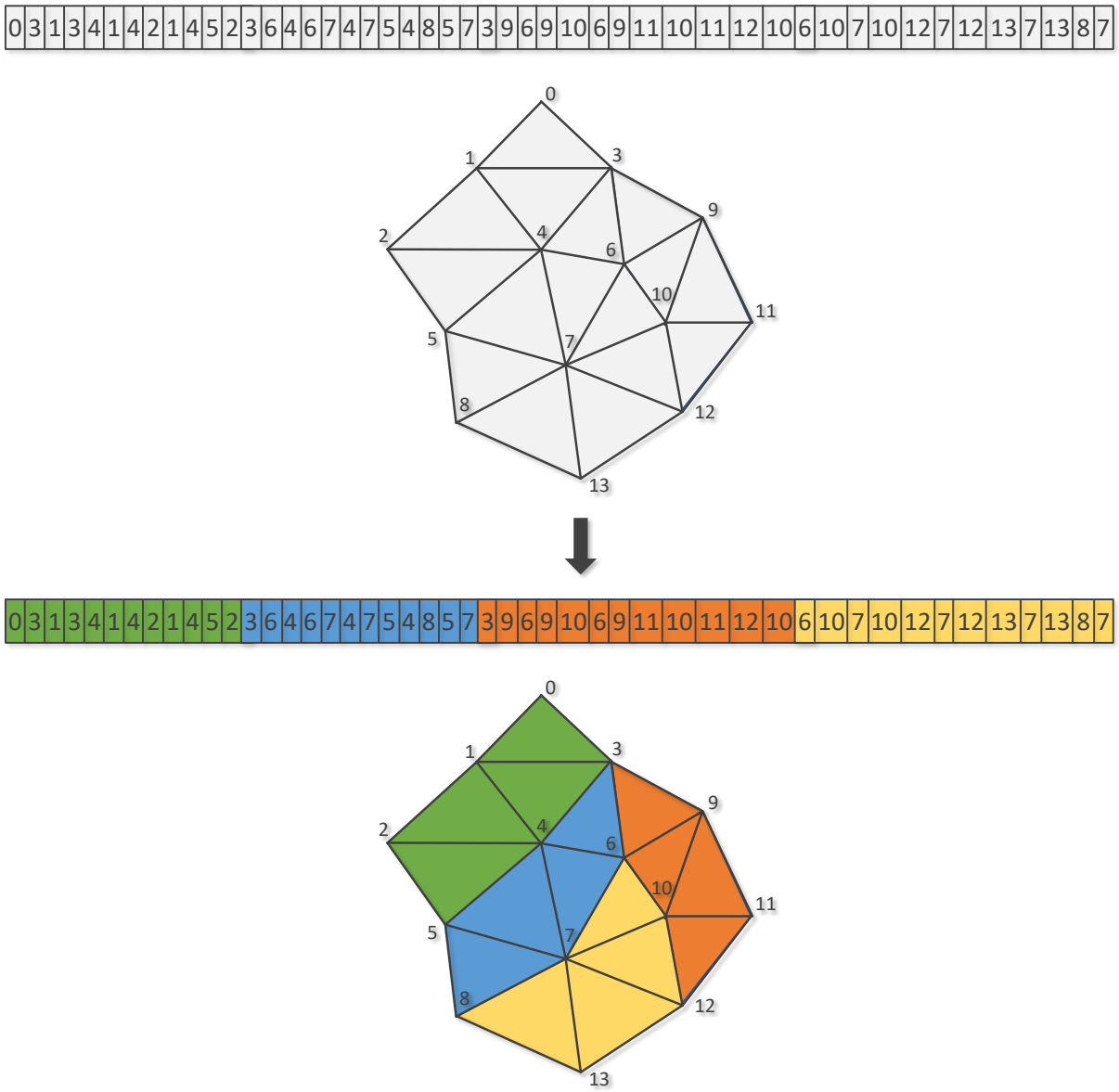


FIGURE 2.2 – En haut : Le buffer d'indices a été réordonné pour renforcer la cohérence spatiale des sommets formant le maillage d'un modèle. En bas : La subdivision régulière du buffer d'indices correspond de manière implicite à des groupes/*clusters* de triangles spatialement cohérents.

résolution $2^{10} \times 2^{10} \times 2^{10}$ (code de Morton sur 30 bits) avec une grille de résolution $2^{21} \times 2^{21} \times 2^{21}$ (code de Morton sur 63 bits) sans noter de réelle amélioration sur la taille des groupes de triangles voisins consécutifs dans le buffer d'indices.

Malgré tout, le tri des indices de triangles selon la courbe de Morton est une opération peu coûteuse qui peut être effectuée sur GPU. De plus, les problèmes de discontinuité sont moindres lorsque le buffer d'indices est découpé en larges tranches de triangles (par exemple 512 ou plus). Pour améliorer les temps de précalcule, nous appliquons l'algorithme des k-moyennes hiérarchique ou d'optimisation du cache de sommets sur des sections grossières du modèle obtenues grâce au tri de Morton (figure 2.6).

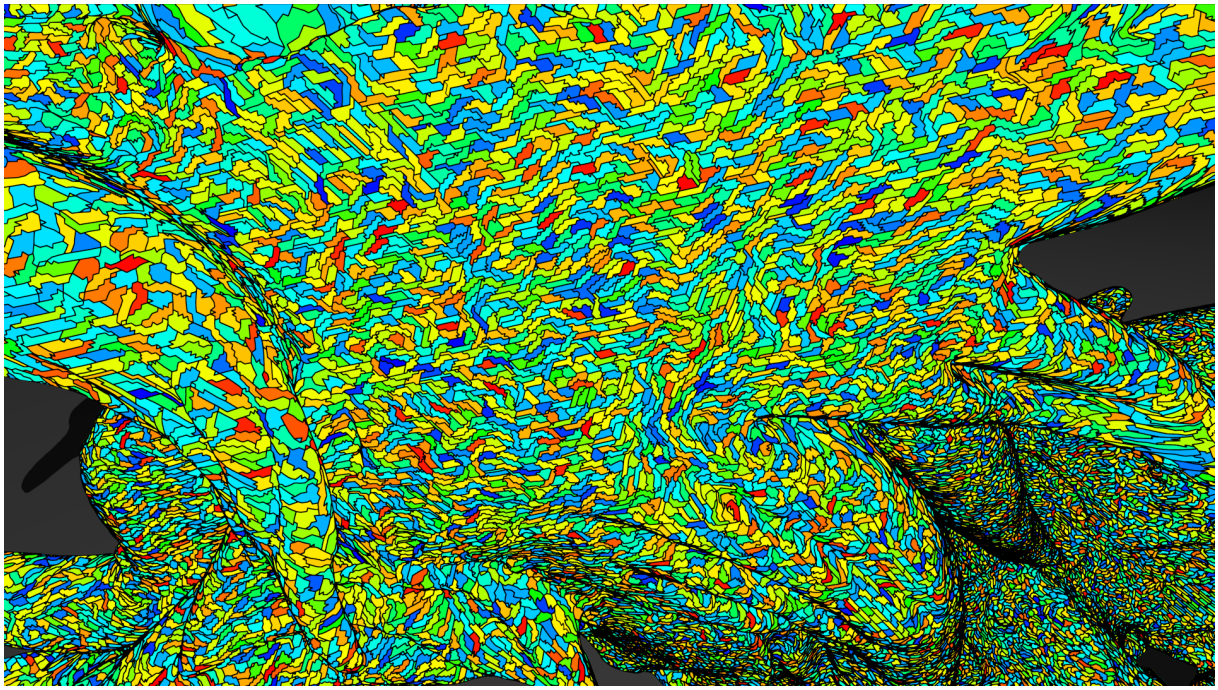

FIGURE 2.3 – Clusters issus du découpage du buffer d'indices après application d'un algorithme d'optimisation du cache de sommets sur le maillage. La cohérence spatiale des triangles au sein de chaque cluster est respectée. Comparée à l'algorithme de partitionnement hiérarchique par la méthode des $k$-moyennes (voir figure 2.1, les clusters produits sont sensiblement plus allongés, la cohérence spatiale des sommets tend en effet à former des bandes/*strips* de triangles.

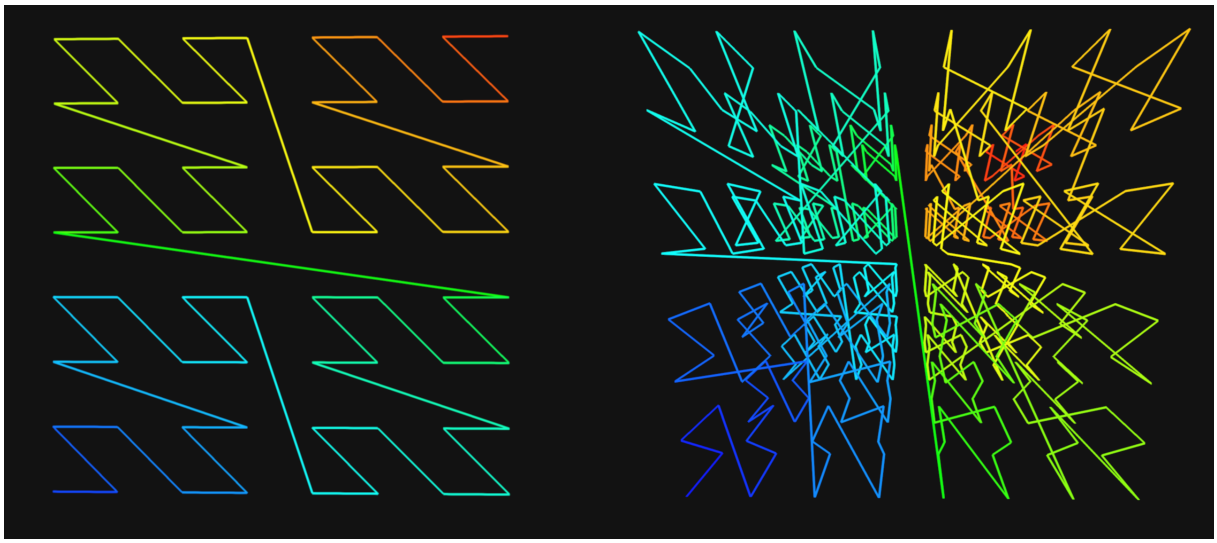

FIGURE 2.4 – Illustration de la courbe de Morton. En pratique, le tri de Morton est appliqué au barycentre des boites orientées (alignées aux axes) des triangles du modèle.

### 2.2.3 Volume englobant d'un cluster

Une fois l'étape de clustering terminée, il reste à calculer un volume englobant pour chaque cluster. Idéalement, ce volume doit être le moins conservatif possible tout en

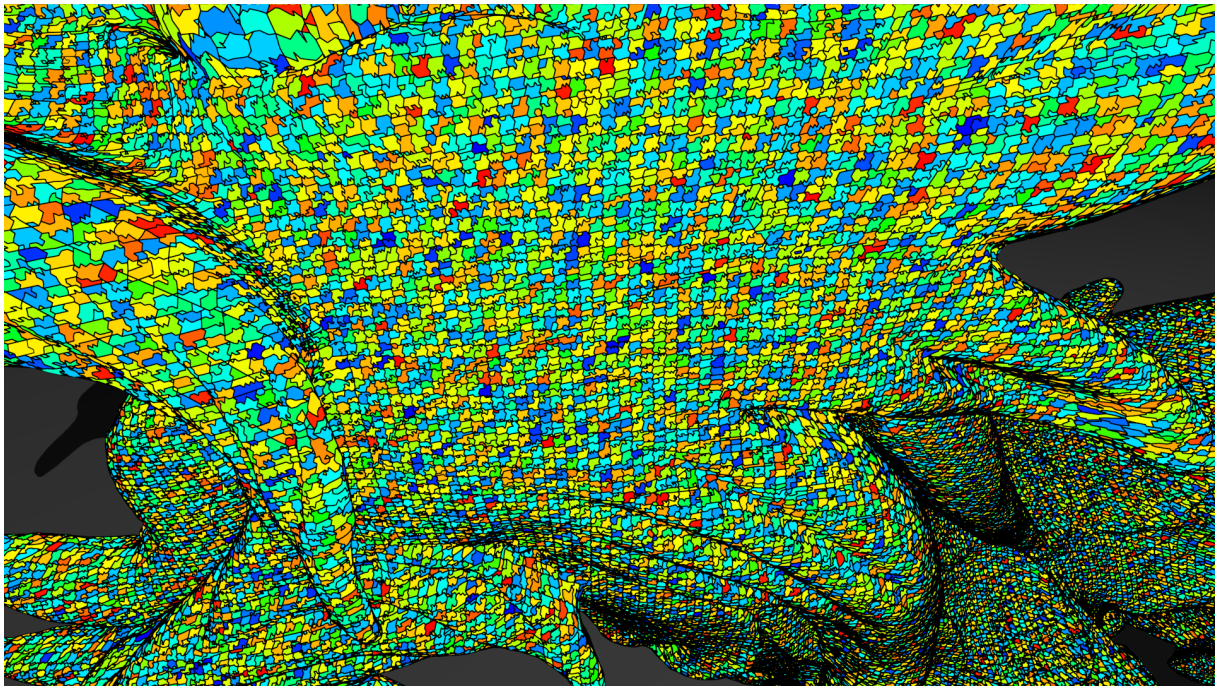

FIGURE 2.5 – Clusters issus du découpage du buffer d'indices après application d'un tri de Morton.

permettant d'extraire une représentation par plan de son volume d'ombre. La boîte englobante orientée semble être le meilleur choix de ce point de vue. Elle s'adapte au mieux à la géométrie du cluster et permet de calculer les plans de son volume d'ombre grâce aux arêtes silhouettes (les arêtes qui longent le contour de la boîte lorsqu'elle est vue depuis la lumière). La boîte englobante orientée est calculée en utilisant la moyenne $\mu$ et la matrice de covariance $C$ des sommets des triangles du cluster, de manière analogue à Gottschalk et al. [42]. Si $p^i$, $q^i$ et $r^i$ sont les 3 sommets du $i^{\text{ème}}$ triangle du cluster, alors $\mu$ et $C$ sont définis par :

$$\mu = \frac{1}{3n}\sum_{i=0}^{n}(p^i + q^i + r^i)$$

$$C_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(p'^i_j p'^i_k + q'^i_j q'^i_k + r'^i_j r'^i_k)$$

Avec $n$ le nombre de triangles dans le cluster, $p'^i = p^i - \mu$, $q'^i = q^i - \mu$, $r'^i = r^i - \mu$ et $C_{jk}$ les valeurs de la matrice $3 \times 3$ de covariance $C$. Les vecteurs propres de $C$ normalisés forment un repère pour la boite englobante orientée et les valeurs maximales et minimales des projetées des sommets sur ces 3 axes définissent les bords de la boite.

Lorsque des modèles fermés sont utilisés, il est toujours intéressant de retirer les triangles qui font face à la lumière (ou l'inverse) puisque l'ombre projetée par les deux faces est la même. Pour le calcul d'ombres, les triangles qui font face à la lumière sont retirés pour éviter les problèmes d'auto-ombrage. Environ 50% de la géométrie peut généralement être supprimée. Dans le cas des PSV, il suffit de tester si la lumière se trouve du côté de la

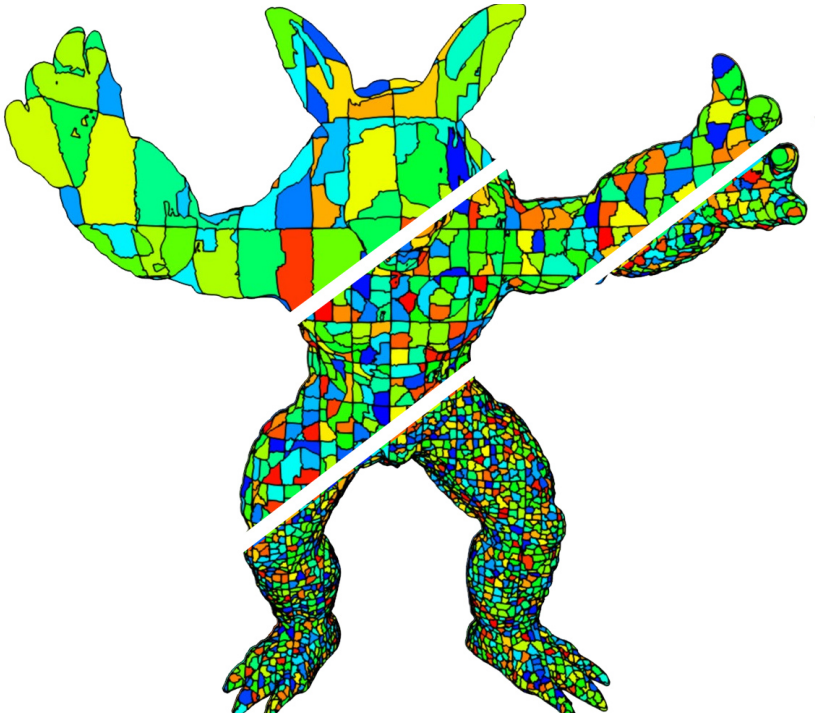

FIGURE 2.6 – En haut : Premier niveau de la hiérarchie de clusters obtenus en triant les triangles le long de la courbe de Morton. On obtient ainsi rapidement une première subdivision du modèle en grands ensembles. Milieu : Le second niveau de la hiérarchie est obtenu en appliquant un partitionnement hiérarchique en k-moyennes sur chaque ensemble de premier niveau. L'algorithme procède récursivement jusqu'à atteindre un nombre minimal de triangles par cluster. En bas : Dernier niveau de la hiérarchie dont les clusters vont être utilisés pour construire l'arbre métrique.

normale d'un triangle pour le mettre de côté. En présence de clusters de triangles, il serait trop coûteux de tester si la lumière se trouve du bon côté de *chaque* triangle. Toutefois, l'espace décrit par l'union des côtés positifs des triangles (le côté de la normale) peut être approché par un cône dit *cône de normales* [19]. Le calcul de ce cône est illustré dans la figure 2.8. La lumière fait face à tous les triangles du cluster si elle est à l'intérieur de son cône de normales, auquel cas le cluster peut-être écarté.

### 2.2.4 Volume d'ombre d'un cluster

Le volume d'ombre projeté par une boîte orientée est l'union des plans de "contour" qui passent par les arêtes silhouettes de la boîte et du plan "support" (pour reprendre la terminologie utilisée pour le volume d'ombre d'un triangle) qui referme la base du volume. Selon le point de vue, le nombre d'arêtes silhouettes d'une boite orientée varie de 4 à 6 quand il vaut toujours 3 pour un triangle. De même, quand le plan support d'un triangle délimite à lui seul la partie supérieure du volume d'ombre, 1 à 3 plans peuvent être nécessaires dans le cas d'une boite orientée. La conséquence est que la représentation du volume d'ombre d'une boite orientée peut demander de 5 à 9 nœuds d'un TOP-tree. D'une part cela entraîne un coup mémoire plus élevé par volume d'ombre. D'autre part, l'empreinte mémoire globale reste bornée mais n'est plus tout à fait fixe puisque le nombre

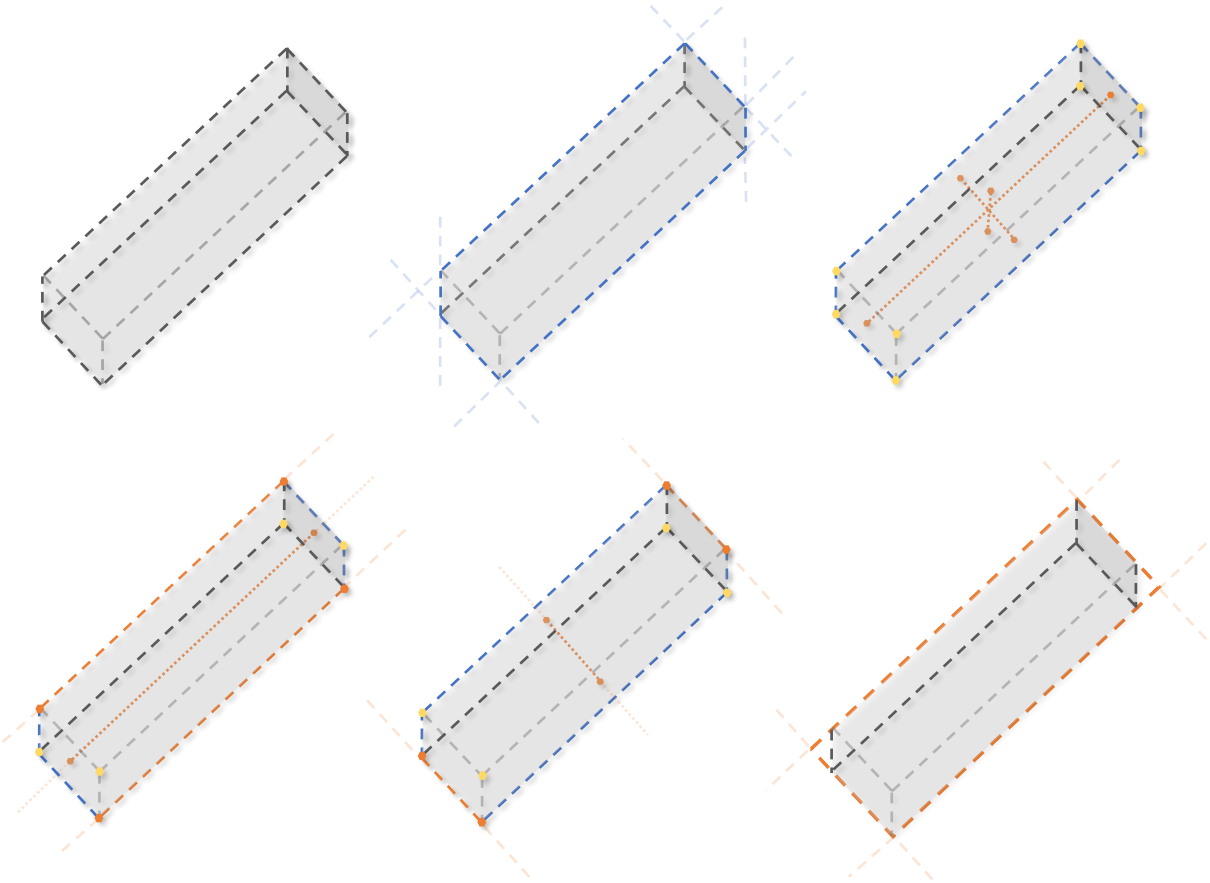

FIGURE 2.7 – En haut. À gauche : Une boîte orientée (OBB) vue depuis la lumière. Au milieu : les plans du volume d'ombre de l'OBB sont ceux qui passent par ses arêtes silhouettes (en bleu). À droite : les segments qui relient le barycentre des faces sur chaque axe sont calculés (en orange) à partir des sommets de l'OBB. Les plans qui passent par les deux plus grands segments (visuellement) sont calculés. De chaque côté de ces plans, la paire de points la plus éloignée définit un nouveau plan (en bas, à gauche et au milieu). Les 4 plans ainsi formés approchent le contour de l'OBB (en bas à droite).

de nœuds pourra varier entre 5 et 9 fois le nombre de clusters. Enfin, avoir des volumes d'ombres représentés par des sous-arbres de tailles variables est de nature à accentuer la divergence entre les threads lors du parcours de la structure. Pour toutes ces raisons, nous proposons une représentation simplifiée du volume d'ombre d'une boite orientée. Au prix d'un léger sur-conservatisme, cela permet d'utiliser exactement 5 nœuds de TOP-tree pour chaque volume. Nous limitons le nombre de plans de contour à 4, comme indiqué sur la figure 2.7. Nous sélectionnons le plan support du volume d'ombre parmi les faces de la boite orientée. La face orientée vers la lumière qui est la plus parallèle au plan orthogonal à la droite qui passe par la lumière et par le centre de la boite est sélectionnée (voir figure 2.9).

### Suppression du plan support du volume d'ombre

Nous proposons une optimisation pour obtenir une version plus compacte limitée à 4 nœuds, comme pour le volume d'ombre d'un triangle dans la version de Gerhards. La

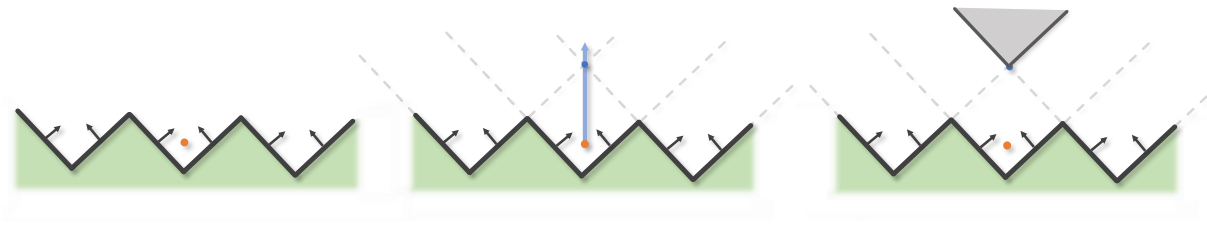

FIGURE 2.8 – Calcul du cône de normales d'un cluster de triangles. À gauche, le barycentre du cluster de triangles est calculé (en orange). Au milieu, un rayon est tracé depuis ce point dans une direction $\vec{N}$ qui est la moyenne des normales des triangles. En calculant l'intersection la plus éloignée avec les plans support des triangles, on trouve le premier point ($O$, en bleu) qui se trouve du côté positif de *tous* les triangles. À droite, les plans des triangles sont utilisés pour calculer l'angle du cône circonscrit de centre $O$ et d'axe $\vec{N}$. Les points qui se trouvent à l'intérieur de ce cône (en gris) ont la garantie de faire face à la normale de chaque triangle du cluster. *Source [19]*

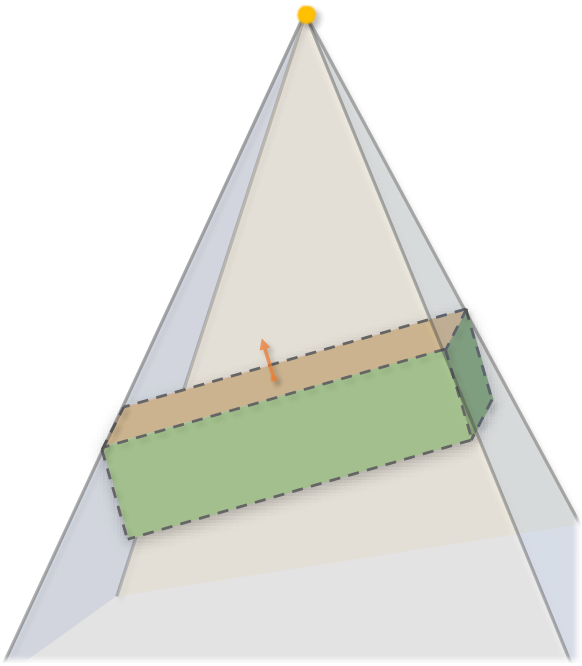

FIGURE 2.9 – Illustration du volume d'ombre d'une boite englobante orientée. Le plan issu de la face orange de la boite referme le volume d'ombre.

distance minimum de la boîte orientée à la lumière peut être utilisée pour remplacer le nœud support du volume d'ombre. Dans le cas des PSV de Gerhards, le plan support du triangle sert de test final pour déterminer si un point est occulté ou non. Lorsque le TOP-tree est appliqué aux boîtes englobantes orientées des clusters, le plan support sert uniquement à éviter de tester les triangles du cluster si le point est plus proche de la lumière que tous les triangles du cluster. Comme nous l'avons vu, le plan support que nous utilisons est issu d'une des faces de la boite englobante orientée du cluster. La distance minimum de la boite à la lumière décrit une sphère centrée autour de la lumière. C'est une approximation de la surface des clusters au même titre qu'une des faces de la boite orientée. Bien que la sphère soit légèrement plus conservative que le plan support

(figure 2.10), cela ne pose pas de problème en pratique puisque les clusters de triangles sont relativement petits : vue depuis la lumière, l'angle solide d'un cluster est très faible, et la portion de sphère intersectée par le volume d'ombre s'apparente à un plan. Nous remplaçons donc le plan support de la boite orientée par sa distance minimum à la lumière pour savoir si l'on se trouve à l'intérieur du cluster lors du parcours. Se faisant, un volume d'ombre ne nécessite plus que 4 nœuds d'un TOP-tree, la distance minimale au cluster depuis la lumière étant stockée au niveau du $4^{ème}$ nœud.

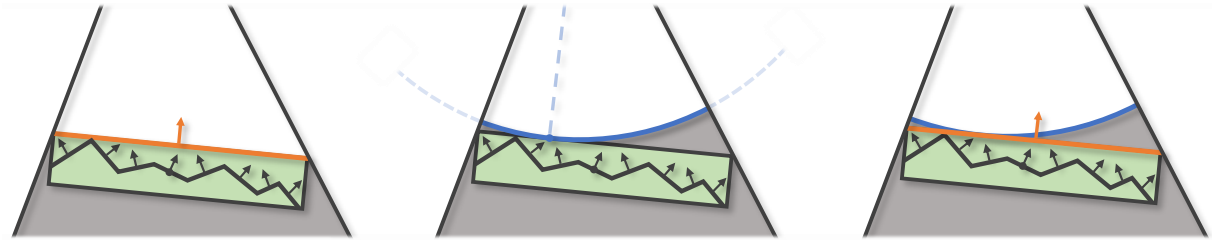

FIGURE 2.10 – À gauche : Le volume d'ombre de la boite orientée est refermé par un plan issue d'une des faces de la boite. Au centre : La distance minimale de la lumière à la boite orientée définie une sphère centrée autour de la lumière qui peut également être utilisée pour fermer le volume d'ombre. À droite : Bien que la sphère puisse être légèrement plus conservative qu'un plan, cela ne se ressent pas en pratique puisque les clusters sont de petite taille.

### 2.2.5 Construction d'un TOP-tree de clusters

La construction de l'arbre TOP-tree diffère peu de l'algorithme proposé par Gerhards si ce n'est qu'il faut l'adapter au fait de ne plus avoir des triangles mais des boites englobantes orientées. Le principe général de l'algorithme reste le même. Nous conservons pour chaque nœud une valeur angulaire qui permet de borner la géométrie (les boites) contenues dans son sous-arbre intersection. Lors du parcours, cette valeur servira à restreindre la visite des sous-arbres intersection aux seuls points contenus dans la zone décrite. La principale différence entre notre TOP-tree et la version de Gerhards est que les feuilles négatives représentent les volumes d'ombre de boites englobantes, ce qui est une représentation conservative de l'ombre générée par la géométrie des clusters associés. Dans notre structure, les feuilles négative pointent donc vers les triangles du clusters afin de permettre un test linéaire et exact lors du parcours.

L'algorithme 5 détaille la construction incrémentale de la structure. Chaque thread récupère un cluster dans une liste et l'insère dans l'arbre. Tout d'abord, le cône de normales est utilisé pour vérifier si le cluster est à rejeter (ligne 15). Ensuite, les 5 plans (ou 4 si le plan support est remplacé par la distance du cluster à la lumière) qui composent le volume d'ombre de la boîte englobante orientée sont calculés (ligne 20) et servent à initialiser les nœuds correspondants. Le fils négatif du premier nœud de contour ($A$) pointe vers le second nœud ($B$), dont le fils négatif pointe vers le troisième nœud ($C$), et ainsi de suite jusqu'au nœud "*support*". Finalement, la boîte englobante orientée est localisée dans l'arbre pour déterminer où placer les nœuds de son volume d'ombre (lignes 25-39). Lorsqu'un nœud $n$ est visité, les 8 sommets de la boîte déterminent de

quel côté du plan elle se trouve : si le signe des distances de chaque sommet au plan est identique, le sous-arbre positif (ou négatif, selon le signe) est à visiter. Sinon, la boîte intersecte le plan de partitionnement et le volume doit être inséré dans le sous-arbre intersection. Dans ce cas, l'angle qui borne la géométrie intersectant le plan est mis à jour. Si le sous-arbre à visiter n'existe pas, le nœud $n$ est modifié pour pointer vers le premier nœud ($A$) du volume en cours d'insertion (ligne 41). La figure 2.11 illustre un exemple de structure obtenue après l'insertion de trois volumes d'ombre.

En pratique, les nœuds sont stockés dans un tableau dédié, avec les nœuds d'un même volume d'ombre placés de manière contiguë. Les liens vers les sous-arbres positif, négatif et intersection sont des indices qui pointent sur le tableau. L'indice 0 sert de valeur sentinelle et indique un lien vide (une feuille). De plus, des opérations atomiques sont utilisées pour les instructions qui pourraient être exécutées simultanément par plusieurs threads sur les mêmes données :

— **Initialisation de la racine** (ligne 23) : L'indice de la racine est initialisé à zéro avant l'étape de construction. Pour insérer un volume dans l'arbre, un thread doit récupérer cet indice. Il vérifie pour cela si la valeur stockée dans la racine vaut zéro. Si c'est le cas, la valeur de la racine est remplacée par l'indice du premier nœud ($A$) du volume, qui devient ainsi la racine de l'arbre. Sinon, l'indice couramment stocké est récupéré et utilisé pour commencer l'insertion. En GLSL, l'instruction *atomicCompSwap* (comparaison et échange atomique) permet de réaliser cet échange de manière atomique.

— **Insertion du volume dans l'arbre** (ligne 41) : La mise à jour du lien (négatif, positif ou intersection) du nœud $n$ pour insérer le volume d'ombre est faite de manière analogue à l'initialisation de la racine, avec l'instruction *atomicCompSwap*.

— **Mise à jour des bornes du nœud intersection** (ligne 34) : L'angle stocké dans chaque nœud sert à borner la géométrie contenue dans son sous-arbre intersection. Il s'agit donc de borner la région contenant les boites intersectées par le plan du nœud. Étant donnée une boite englobante intersectée et son sommet dont la distance au plan est maximale, nous calculons l'angle au plan formé avec la droite passant par ce sommet et la lumière. Puis l'angle maximal conservé dans le nœud est mis à jour avec l'opération *atomicMax* avant que la boite intersectée ne poursuive son parcours dans le sous-arbre intersection. En OpenGL standard, les opérations atomiques peuvent être effectuées uniquement sur des nombres entiers. Puisque la valeur de la borne est toujours positive, celle-ci peut être stockée dans un nombre entier non signé (en *glsl* avec l'opération *floatBitsToUint*, qui effectue une copie bit à bit de la valeur, sans conversion de type). Effectuer l'opération *atomicMax* sur la valeur entière reste correct car un nombre flottant supérieur à un autre le sera également en interprétant leur représentation binaire comme un entier [1].

### 2.2.6 Parcours d'un TOP-tree de clusters

La structure est construite à chaque image. Pour chaque point visible depuis la caméra, l'arbre est parcouru depuis la racine afin de déterminer la visibilité depuis la lumière. Vu projectivement depuis la source, nous cherchons si un point est inclus dans le volume d'ombre d'une boîte orientée afin de tester si la géométrie qu'elle contient occulte le

**Algorithme 3** Construction du TOP-tree sur les boîtes englobantes orientées des clusters.

```
1: nœud {
2:     Plan plan,                                              ▷ Équation du plan
3:     nœud positif, intersection, negatif,                    ▷ Lien vers les fils
4:     Réel borne          ▷ Angle qui borne le sous-arbre intersection autour du plan
5: }
6:
7: ConeDeNormales {                  ▷ Cône qui englobe les normales du cluster
8:     Vecteur3D centre,
9:     Vecteur3D normal,
10:    Réel angle
11: }
12:
13: procedure AjoutVolume(OBB boite, ConeDeNormales cone, nœud racine,
    Lumière L)
14:
15:    si ContenuDansCone(L, cone) alors
16:        Retourner()          ▷ Quitter si tous les triangles font face à la lumière
17:    fin si
18:
19:    nœud A, B, C, D, support                    ▷ A, B, C, D : plans de contour
20:    Initialisernœuds(boite)          ▷ Calcul des plans qui composent le volume
21:                    ▷ Le fils négatif de A pointe sur B, lui-même pointe sur C, etc...
22:
23:    nœud n ← racine
24:
25:    tant que n n'est pas une feuille faire
26:        Réel pos ← Position(n.plan, boite)
27:
28:        si pos > 0 alors
29:            n ← n.positif
30:        sinon si pos < 0 alors
31:            n ← n.negatif
32:        sinon                                          ▷ La boîte intersecte le plan
33:            si n.plan est issue d'une arête alors
34:                MiseAJourBorneIntersection(n, boite)
35:            fin si
36:            n ← n.intersection
37:        fin si
38:
39:    fin tant que
40:
41:    RemplacerFeuilleParVolumeOmbreOBB(n, boite, L)
42:
43: fin procedure
```

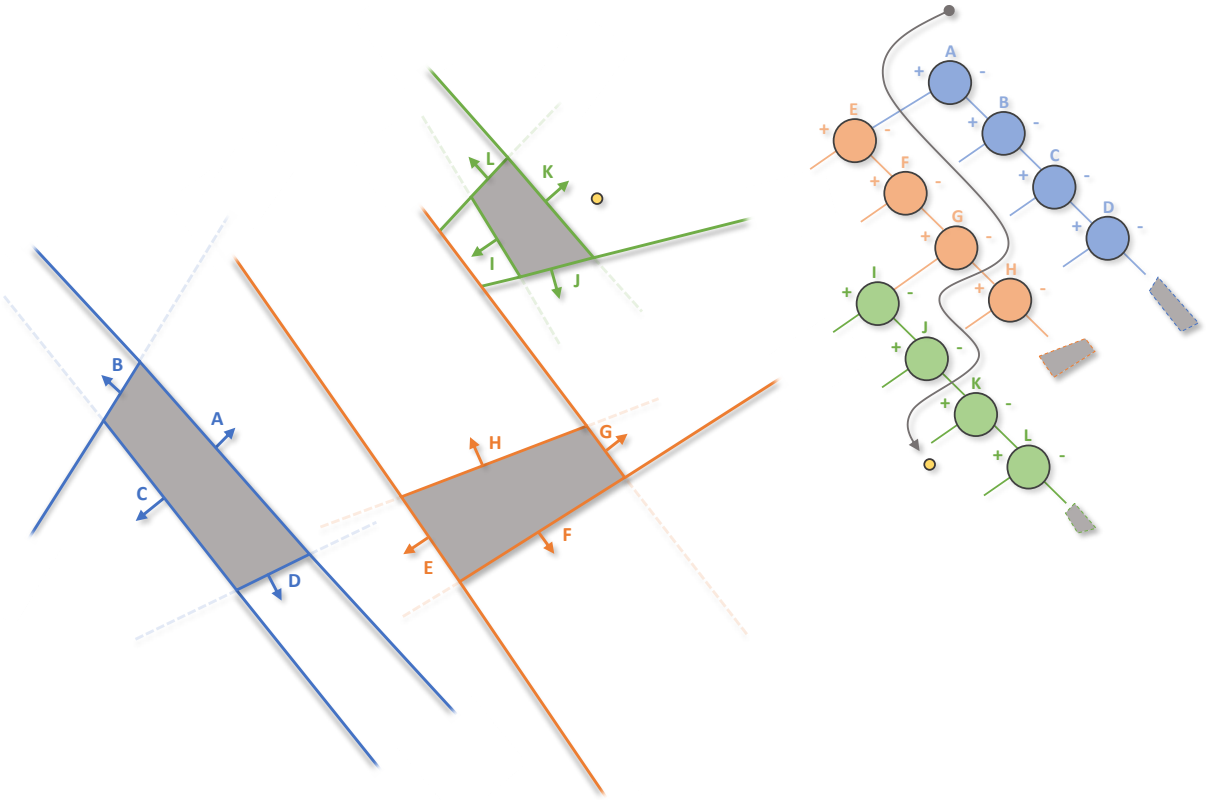


FIGURE 2.11 – Exemple de structure obtenue en construisant un TOP-tree sur des boites orientées de clusters.

point. La base de l'algorithme de parcours utilise une pile de nœuds (voire l'algorithme 3.3.3) et consiste à appliquer les étapes suivantes à chaque nœud visité :

1. Si le point se situe dans le demi-espace négatif (resp. positif) du plan, le parcours se poursuit dans le sous-arbre négatif (resp. positif).
2. Si le point est contenu dans la région du sous-arbre intersection, le fils intersection est empilé pour être visité ultérieurement.
3. Si le point atteint une feuille positive, le parcours se poursuit dans le dernier nœud empilé. Si la pile est vide, le parcours est terminé sans avoir trouvé d'occultation et le point est donc visible depuis la lumière.
4. Si le point atteint une feuille négative, les triangles du cluster sont testés individuellement pour vérifier si une occultation existe réellement. Dans le cas de la version à 4 nœuds par volume d'ombre, on vérifie au préalable que la distance du point à la lumière est supérieure à celle du cluster à la lumière. Si une occultation est trouvée, le point est à l'ombre et le parcours s'interrompt.

Ce parcours peut être optimisé en transposant une amélioration issue de la méthode de Gerhards : Il est inutile de visiter un sous-arbre qui contiendrait uniquement de la géométrie plus éloignée de la lumière que ne l'est le point testé [68]. Aussi, chaque nœud mémorise la plus petite distance (Euclidienne) de la source à tous les clusters contenus dans le sous-arbre enraciné en ce nœud. Si la distance du point testé à la lumière est inférieure à la distance stockée, le nœud n'a pas besoin d'être visité. En pratique nous

**Algorithme 4** Parcours de l'arbre pour localiser un point $p$.

```
1:  function ParcoursCPSV(nœud racine, Point p)
2:
3:      PileDenœud pile
4:      Ajouter(pile, racine)
5:
6:      tant que pile non vide faire
7:          nœud n ← Depiler(pile)
8:
9:          si n n'est pas une feuille alors
10:             Entier pos ← Signe(n.plan, p)
11:
12:             si n.plan est un plan contour alors
13:                 Réel angle ← CalculerAngle(n.plan, p)
14:                 si angle < n.borne alors     ▷ Faut-il visiter le sous-arbre intersection ?
15:                     Empiler(pile, n.intersection)
16:                 fin si
17:                 Empiler(pile, location > 0 ? n.positif : n.negatif)
18:             sinon                                ▷ Plan "support" qui referme l'OBB
19:                 si pos < 0 alors
20:                     si IntersectionTriangles() alors
21:                         Retourner(0)   ▷ Le point est ombré par un triangle du cluster
22:                     fin si
23:                 sinon
24:                     Empiler(pile, n.intersection)
25:                     Empiler(pile, n.positif)
26:                 fin si
27:             fin si
28:         fin si
29:
30:     fin tant que
31:
32:     Retourner(1)                          ▷ La pile est vide, le point est éclairé
33:
34: fin function
```

utilisons la plus petite distance de la lumière à la boîte englobante orientée du cluster. Se référer à la géométrie contenue dans les clusters serait moins conservatif. Mais il faudrait parcourir les triangles des clusters avant leur insertion dans la structure, ce qui serait prohibitif. Chaque fois qu'un élément passe par un nœud, la distance minimale stockée est mise à jour si la distance de l'élément à la lumière est plus petite. Tout comme les autres instructions pouvant générer des accès concurrents entre les threads, elle est faite au moyen d'une opération atomique (*atomicMin*).

Ce parcours reste relativement analogue à celui utilisé par Gerhards. Il tient cependant compte du fait qu'un point atteignant une feuille négative n'est pas nécessairement à l'ombre. Il faut tester les triangles des clusters individuellement, ce qui représente un surcoût par rapport à la version de Gerhards. Par ailleurs, le test des triangles étant linéaires, cela incite à ne pas utiliser des clusters de trop grandes tailles. Le parcours pourrait devenir trop coûteux si le nombre de triangles à tester par cluster est trop important. De manière générale, nos tests ont montré que 32 triangles par cluster était une valeur au-delà de laquelle l'impact sur le temps de parcours devenait significatif. Le gain attendu sur le temps de construction ne doit pas être contrebalancé outre mesure par le surcoût attendu sur le temps de parcours.

### 2.2.7 Double hiérarchie de TOP-tree

L'idée est ici d'exploiter non plus un, mais deux niveaux de clusters : Un premier niveau de clusters de grande taille (1024 à 4096 triangles) et un second niveau de clusters de 8 à 32 éléments, c'est-à-dire celui utilisé dans l'approche présentée précédemment. La figure 2.12 donne un exemple des deux niveaux utilisés pour un modèle. En se basant sur ces deux niveaux de clusters, nous construisons ensuite une double hiérarchie de TOP-tree.

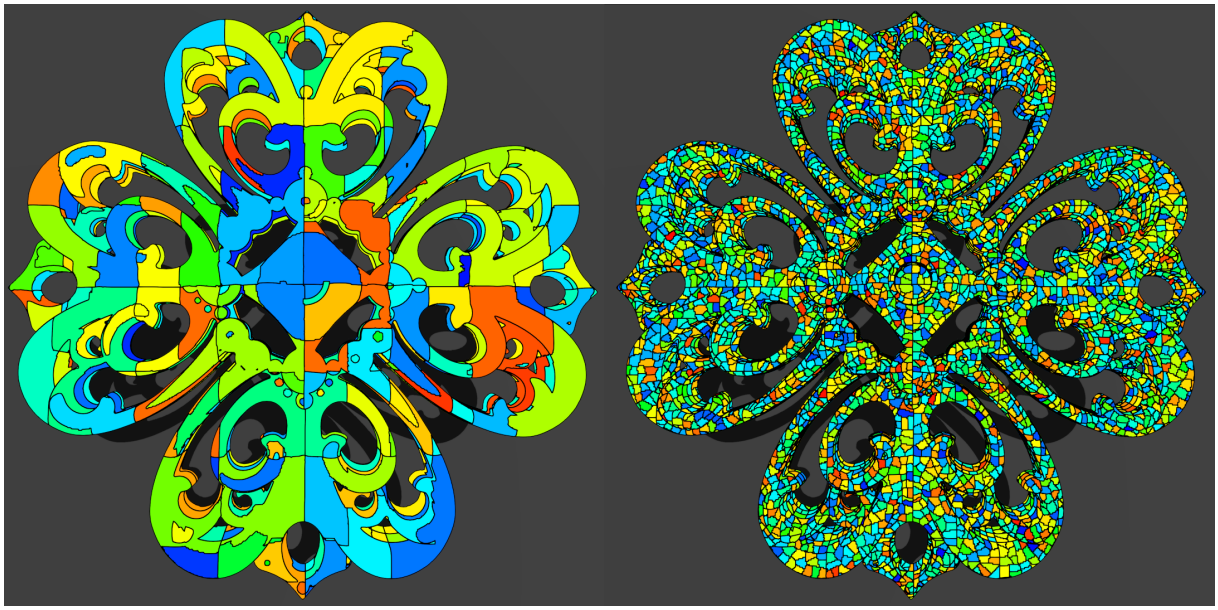

FIGURE 2.12 – À gauche, les clusters de haut niveau. À droite, ceux de bas niveau. Ces deux niveaux de clusters sont utilisés pour construire une double hiérarchie de TOP-tree.

Nous construisons un premier TOP-tree en utilisant les clusters de haut niveau. Cette construction est analogue à celle décrite dans la section 2.2.5 sauf qu'une feuille négative ne pointe pas vers les triangles d'un cluster, mais vers les clusters de bas niveaux qu'elle contient et avec lesquels nous construisons un TOP-tree à part entière (la feuille négative pointe vers la racine de ce sous-arbre). Nous obtenons de la sorte une double hiérarchie de TOP-tree.

Cette construction permet une meilleure distribution du travail sur le GPU. Dans l'approche présentée précédemment, nous construisions un unique TOP-tree avec tous les clusters de bas niveau. Ici, étant donné $NB$ clusters de haut niveau contenant chacun en moyenne $nb$ clusters de bas niveau, nous pouvons construire en parallèle $NB$ TOP-trees avec $nb$ clusters chacun (plus le TOP-tree sur les $NB$ clusters de haut niveau). Ce faisant, les accès concurrents sont minimisés puisque les threads travaillent sur $NB$ structures plutôt que sur une seule, ce qui peut ralentir sa construction comme l'analyse la figure 2.13.

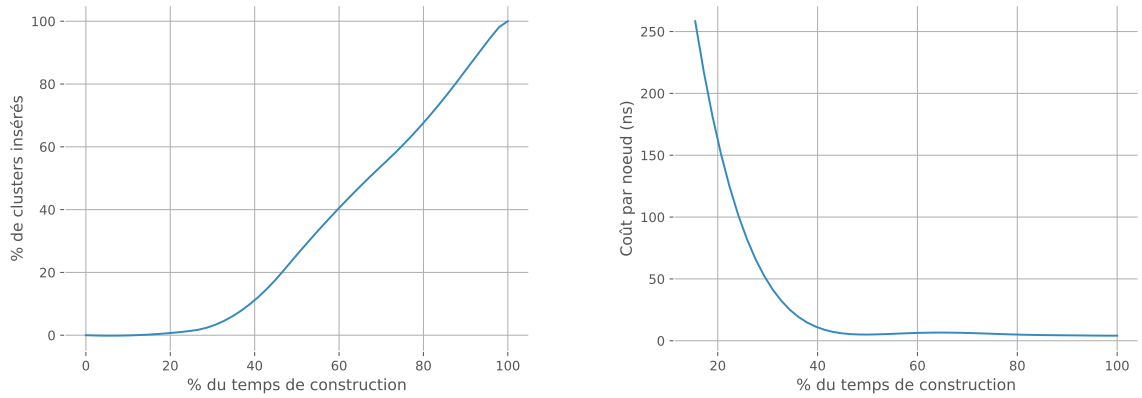


FIGURE 2.13 – Les deux graphes ci-dessus illustrent à quelle vitesse les nœuds (ou clusters) sont placés dans un TOP-tree au cours de sa construction avec tous les clusters. Ces mesures ont été réalisées sur le modèle xyzrgb_dragon. À gauche : Le graphe montre le pourcentage de clusters/nœuds insérés dans l'arbre en fonction du temps de construction écoulé (en pourcentage du temps total). On constate que pendant les 40 premiers pourcents, à peine 7% des clusters sont placés dans l'arbre. Ensuite, le nombre de clusters placés augmente linéairement. À droite : Le même phénomène est ici visible sous la forme d'un graphe montrant le temps nécessaire pour placer un cluster/nœud dans l'arbre au fur et à mesure de la construction. Les premiers nœuds sont très coûteux au début (250 ns), puis le coût décroît jusqu'à se stabiliser en dessous de 4 ns. Ces mesures illustrent la densité des accès concurrents dans les premiers niveaux de l'arborescence où tous les threads essaient de lire et surtout d'écrire aux mêmes endroits. Cela crée un "goulot d'étranglement" qui ralentit fortement le processus de construction. Une fois les premiers niveaux construits, le travail des threads est mieux distribué dans les différentes branches du TOP-tree et les accès concurrents sont diminués.

En passant sur une double hiérarchie de TOP-tree, on change de fait la nature de la structure en ajoutant un premier TOP-tree de "haut niveau" qu'il faudra par conséquent parcourir en plus. Toutefois le risque d'un surcoût significatif au parcours est limité car sa complexité demeure logarithmique et que le nombre de clusters de haut niveau est faible

en comparaison du nombre de clusters de bas niveau (32 à 128 fois moins).

On peut noter que les deux niveaux de clusters nécessaires peuvent être obtenus avec les deux méthodes de clustering que nous avons présentées dans la section 2.2.2. C'est relativement implicite dans le cas de l'algorithme hiérarchique en $k$-moyennes. Il suffit de sélectionner et conserver deux des niveaux produits par l'algorithme. Dans le cas de la subdivision du buffer d'indices de sommet, il suffit d'appliquer une première subdivision en grande section pour obtenir le plus haut niveau, puis de subdiviser à nouveau chacune en plus petite section pour obtenir le plus bas niveau. En pratique, lorsque nous utilisons le tri selon la courbe de Morton pour former plus rapidement une première subdivision de la géométrie, celle-ci est conservée en tant que clusters de haut niveau.

## 2.3 Résultats

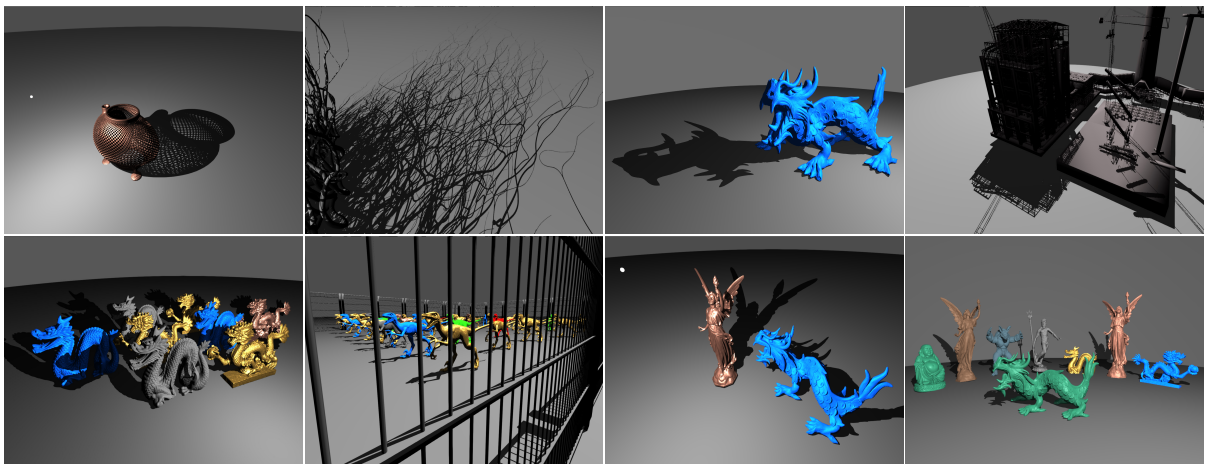

FIGURE 2.14 – En haut : *Birdfeeder* (1.8M de triangles) présente des ombres complexes avec un motif régulier. *Tentacles* (3.8M de triangles) projète des ombres complexes et irrégulières. *xyzrgb_dragon* est un modèle finement triangulé de 7.2M triangles. *Powerplant* (12.7M triangles) est un modèle très compliqué pour des algorithmes basés géométrie. L'intérieur du bâtiment est essentiellement un enchevêtrement de tubes, chacun étant formé de fins triangles allongés. En bas : *Dragons* (18.9M de triangles) comporte 8 dragons finement triangulés. *RaptorPark* (30M de triangles) regroupe 30 raptors finement triangulés derrière des grilles formées de longs triangles. *Lucy&Dragon*, avec 35M de triangles, est un modèle de très grande taille dans un contexte temps réel. *ManyModels* possède 73.8M de triangles et permet d'éprouver les limites de notre méthode.

Pour l'ensemble des tests, la machine utilisée possède un CPU Intel i7700k et une NVIDIA GTX 1080 comme carte graphique. Toutes les images sont produites à une résolution de 1920x1080. Rappelons que notre objectif est de montrer que notre méthode, basée géométrie, reste performante sur des modèles de très grande taille. Aussi nous utilisons des scènes contenant entre 1.8 et 73.8 million de triangles. La Figure 2.14 présente les différents modèles et précise la taille de chacun. Notre algorithme est implémenté avec OpenGL 4.3 en utilisant un rendu différé (*deferred rendering*). La construction de l'arbre ternaire se fait via un Compute Shader. Chacune de ses instances (un thread donc) est exécutée selon

un mode persistent : une instance prend un cluster, l'insère dans l'arbre et recommence jusqu'à ce que tous les clusters soient traités. Une passe de rastérisation depuis la caméra permet d'obtenir dans un GBuffer toutes les positions à tester. Le parcours de l'arbre est lui implémenté dans un Fragment Shader appliqué à ce GBuffer. Pour chaque pixel, une instance du shader fait traverser l'arbre à la position correspondante pour déterminer si elle est éclairée ou non. Toutes les données relatives aux clusters (pré-calculés) et aux TOP-tree sont stockées sur la carte graphique dans des Shader Storage Buffer Objects.

La géométrie qui ne peut produire d'ombre sur la partie visible de la scène est toujours éliminée (*light-view frustum culling*). Pour cela, nous calculons la pyramide englobant le frustum de la caméra et ayant pour sommet la lumière. Tout ce qui est en dehors de cette pyramide peut être ignoré. Ensuite, les triangles (où les clusters dans notre cas) faisant face à la lumière sont toujours éliminés. Toutes les comparaisons sont issues de parcours libres dans les scènes (les mêmes pour chaque scène et les méthodes comparées).

Nous commençons par tester les différentes options proposées précédemment. Tout d'abord nous comparons le conservatisme des volumes englobants des clusters en fonction de la méthode de clustering utilisée. Nous comparons ensuite la version utilisant 5 nœuds par cluster à celle n'en utilisant que 4. Enfin, nous testons l'apport d'une double hiérarchie de clusters. Nous retenons à chaque fois le meilleur choix pour comparer la méthode résultante à celle des PSV de Gerhards *et al.* [68] ainsi qu'à l'approche proposée par Wyman *et al.* [98].

### 2.3.1 Comparaison et sélection des variantes proposées

#### Comparaison du conservatisme des boites englobantes orientées en fonction de la méthode de clustering

Pour rappel, nous avons testé deux méthodes pour précalculer les clusters. D'une part une méthode issue de l'état de l'art, le partitionnement hiérarchique en $k$-moyenne ; d'autre part une méthode qui "détourne" un algorithme d'optimisation de cache pour ensuite former des clusters par simple subdivision du buffer d'indices des sommets. Dans les deux cas, nous calculons de la même façon les boites englobantes orientées des clusters. Mais leur "qualité" (forme, connectivité, régularité...) impacte nécessairement le conservatisme de leur boite englobante. Pour l'évaluer, nous avons rendu les ombres sur différents modèles en utilisant un TOP-tree construit avec des clusters produits avec chacune des deux méthodes. Dans les deux cas de figure, nous avons compté lors du parcours des TOP-tree combien de fois l'échantillon de vue se trouvait dans le volume d'ombre d'une boite orientée sans pour autant trouver d'intersections avec les triangles contenus. Dans tous les tests, il apparaît que les clusters calculés par partitionnement hiérarchique en $k$-moyenne donnent des volumes englobants moins conservatifs, même si la différence est parfois assez faible, notamment sur des modèles avec un maillage dense. La figure 2.16 illustre sur deux scènes les écarts mesurés. En soit, ce résultat est conforme à ce que l'on pouvait attendre d'une comparaison entre un algorithme classique et éprouvé de l'état de l'art avec une méthode détournée de l'optimisation de cache. Ce qui est finalement le plus surprenant est que la différence, même si elle existe toujours, n'est pas systématiquement très marquée.
Pour autant, l'impact sur le temps de parcours des TOP-tree est réel. La figure 2.16 montre les performances au parcours sur deux modèles en fonction du type de cluster

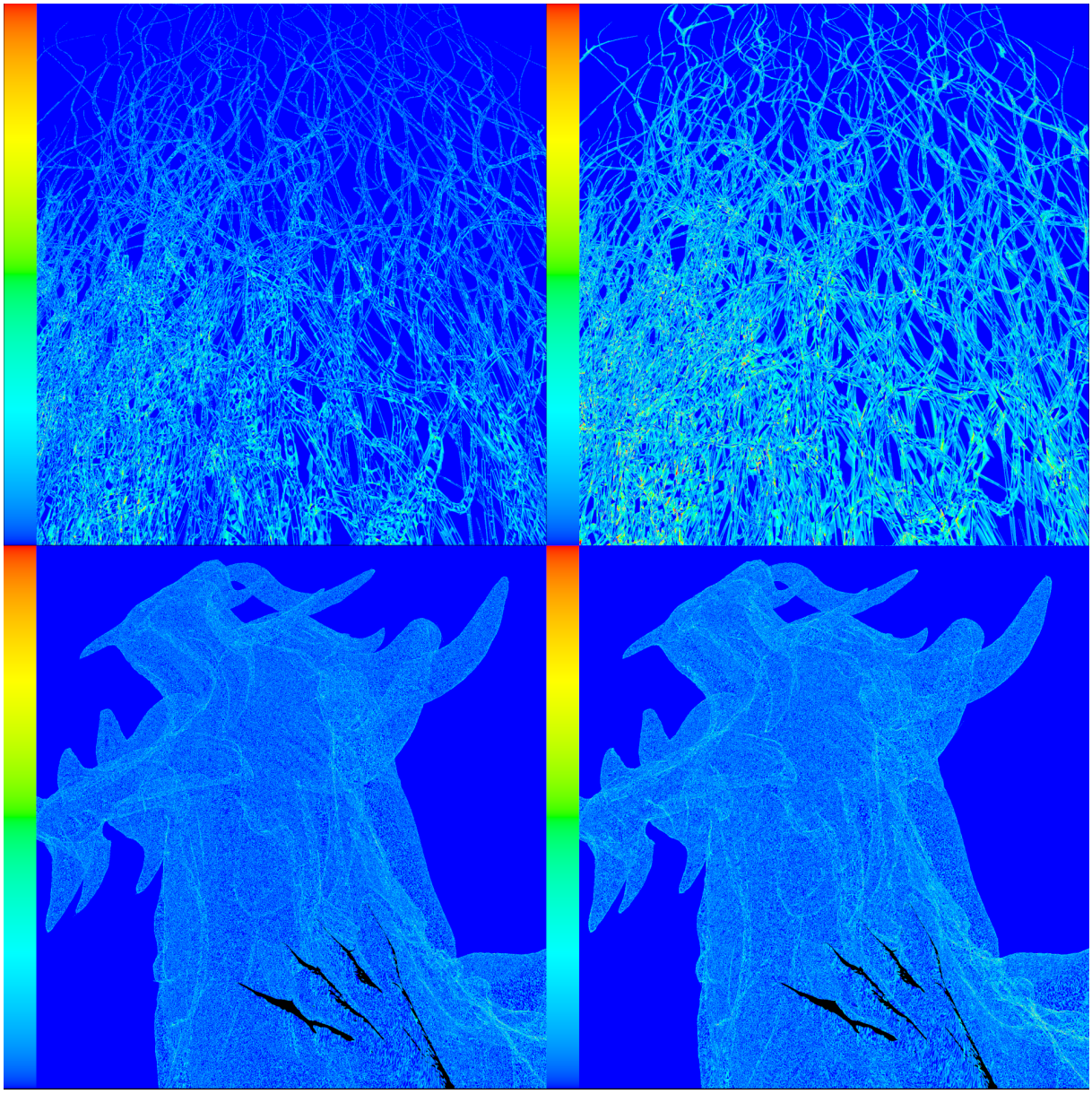

FIGURE 2.15 – Les cartes de chaleur ci-dessus illustrent le conservatisme des volumes englobants des clusters en fonction de la méthode de clustering utilisée, c'est-à-dire le $k$-mean (à gauche) ou bien la subdivision du buffer d'indices après application d'un algorithme d'optimisation du cache de sommets (à droite). Sur deux exemples, Tentacles en haut et xyzrgb_dragon en bas, on compte lors du parcours le nombre de fois que l'on teste les triangles d'un cluster sans pour autant y trouver une intersection. L'échelle utilisée va de 0 test en bleu foncé à plus de 12 tests inutiles en rouge. On constate que le $k$-mean produit des volumes englobants moins conservatifs, ce qui est davantage notable sur Tentacles car le phénomène est renforcé au niveau des contours, nombreux sur ce modèle. La différence est moindre sur xyzrgb_dragon mais néanmoins visible.

utilisé pour la construction du TOP-tree. Plus les boites englobantes sont conservatives, plus on devra tester (linéairement) les triangles du cluster correspondant. Au sein d'un même warp, le fait que des threads testent les triangles d'un cluster pendant que d'autres

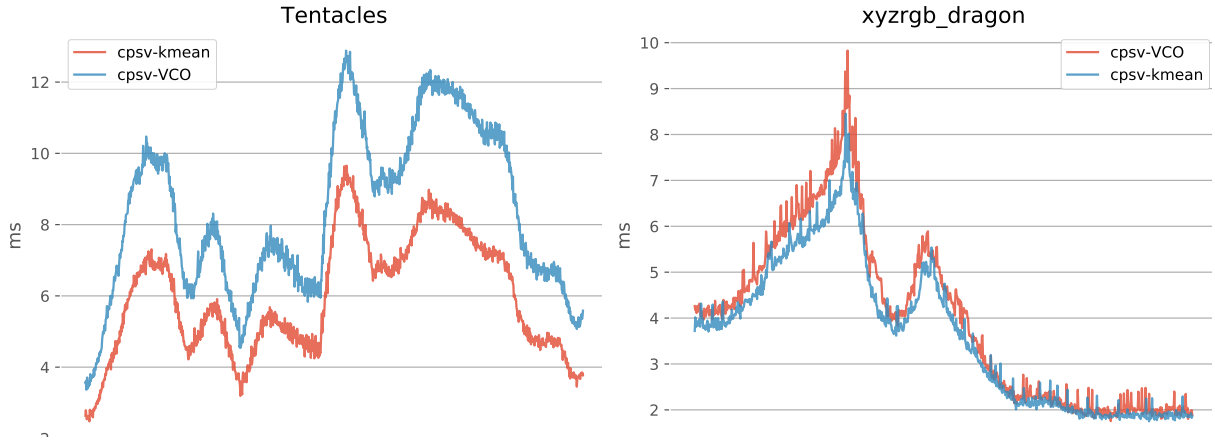


FIGURE 2.16 – Comparaison des performances au parcours de la structure en fonction du type de cluster utilisé pour construire le TOP-tree. Les mesures sont faites à chaque image au cours d'une promenade libre dans le modèle Tentacles (gauche) et xyzrgb_dragon (droite). Les courbes annotées "kmean" ont été produites avec des clusters calculés par partitionnement hiérarchique selon la méthode des $k$-moyenne. Les courbes annotées "VCO" (Vertex Cache Optimisation) ont été produites avec des clusters formés par subdivision du buffer d'indices de sommets après optimisation de cache. Les résultats sont consistants avec les cartes de chaleurs de la figure 2.16 avec un avantage plus marqué sur Tentacles que sur xyzrgb_dragon. Dans tous les cas, le partitionnement hiérarchique induit les meilleures performances.

poursuivent le parcours de la structure entraîne une divergence des instructions à exécuter tout comme une divergence des accès mémoires à réaliser. Sur GPU et dans les deux cas, cela est à éviter autant que possible pour bénéficier pleinement du parallélisme.
Sur un modèle comme *xyzrgb_dragon*, il est tout de même intéressant de souligner que les clusters formés par subdivision du buffer d'indices procurent des performances très proches de celles obtenues avec le partitionnement hiérarchique en $k$-moyenne. Cela montre que cette approche possède un réel potentiel qui présenterait un intérêt certain dans un contexte où le temps de calcul des clusters serait critique.
Toutefois, s'agissant pour nous d'un pré-calcul, le temps passé à former les clusters n'est pas un critère. Aussi par la suite, nous utilisons toujours des clusters issus d'un partitionnement hiérarchique selon la méthode des $k$-moyennes.

### Comparaison entre la représentation à 4 ou 5 plans des volumes d'ombre des boites englobantes orientées

Dans la section 2.2.4, nous avons proposé deux options pour représenter le volume d'ombre d'une boite englobante orientée. La première utilise 5 plans et donne donc lieu à 5 nœuds de TOP-tree par cluster. La seconde n'en utilise que 4 et utilise un test de profondeur additionnel à la place d'un $5^{\text{ème}}$ plan. Nous avons comparé les performances globales de ces deux versions. De manière générale, la représentation optimisée à 4 nœuds procure en moyenne une amélioration de 10% par rapport à la version à 5 nœuds. La figure 2.17 illustre ce comportement sur deux modèles (Birdfeeder et xyzrgb_dragon). La version optimisée produit des arbres ternaires avec moins de nœuds, ce qui réduit sa

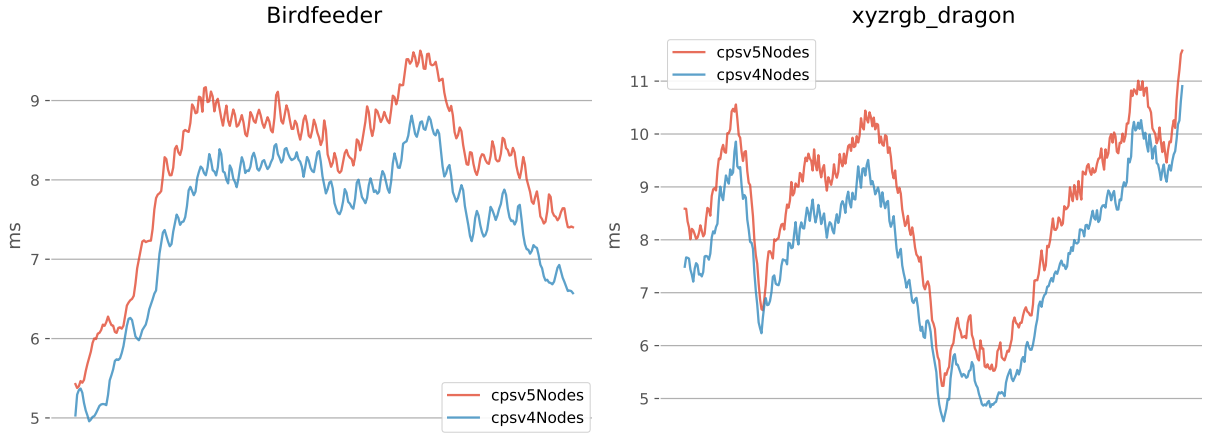


FIGURE 2.17 – Comparaison des performances globales (temps de construction plus temps de parcours) entre la version à 5 nœuds et la version optimisée à 4 nœuds des volumes d'ombre des boites englobantes des clusters. Les mesures sont faites le long de parcours libres dans deux scènes, Birdfeeder à gauche et xyzrgb_dragon à droite. Les courbes rouges sont obtenues avec la version à 5 nœuds, les bleues avec la version optimisée à 4 nœuds.

consommation en terme de bande passante mémoire. Bien souvent, les accès mémoires sur GPU coûtent davantage que les calculs, ce qui permet à la version optimisée d'avoir des performances sensiblement meilleures. Par définition, la version à 4 nœuds est plus conservative que la version à 5 nœuds, puisque l'on remplace un élément de plan par un élément de sphère. L'hypothèse faite dans la section 2.2.4 était que s'agissant d'éléments de très faibles dimension, les deux représentations seraient localement très proches et l'on pouvait espérer que cela n'impacte pas les résultats de manière significative. Les mesures réalisées le confirment.

Par la suite, nous utilisons toujours la version optimisée à 4 nœuds dans l'ensemble des résultats présentés.

### Comparaison entre simple et double hiérarchie de TOP-tree

Dans la section 2.2.7, nous avons expliqué comment la densité des accès concurrents dans les premiers niveaux de la structure de TOP-tree pouvait ralentir sa construction. Pour éviter ce problème, nous avons proposé d'utiliser une double hiérarchie de TOP-tree basée sur deux niveaux de clusters dont la construction permet de mieux répartir le travail sur GPU. Nous comparons ici les performances obtenues entre de simples TOP-trees de clusters et des doubles hiérarchies de TOP-trees de clusters. La figure 2.18 donne un aperçu des résultats obtenus et la table 2.1 regroupent les temps moyens et les tailles de cluster utilisées pour les deux versions. La double hiérarchie permet d'améliorer significativement les performances globales de la méthode. Si l'on regarde dans le détail, ce gain est obtenu sur le temps de construction. À taille de clusters équivalente (ceux de bas niveaux dans la version doublement hiérarchique), les temps de parcours sont les mêmes. Avec un simple TOP-tree, la taille des clusters est une valeur qui laisse peu de marge de manœuvre. Des clusters trop gros dégradent rapidement les performances lors du parcours car tester les triangles d'un cluster devient très coûteux. À l'inverse, des clusters trop petits seront plus nombreux et finiront pas dégrader le temps de construction. Ainsi dans nos tests, selon

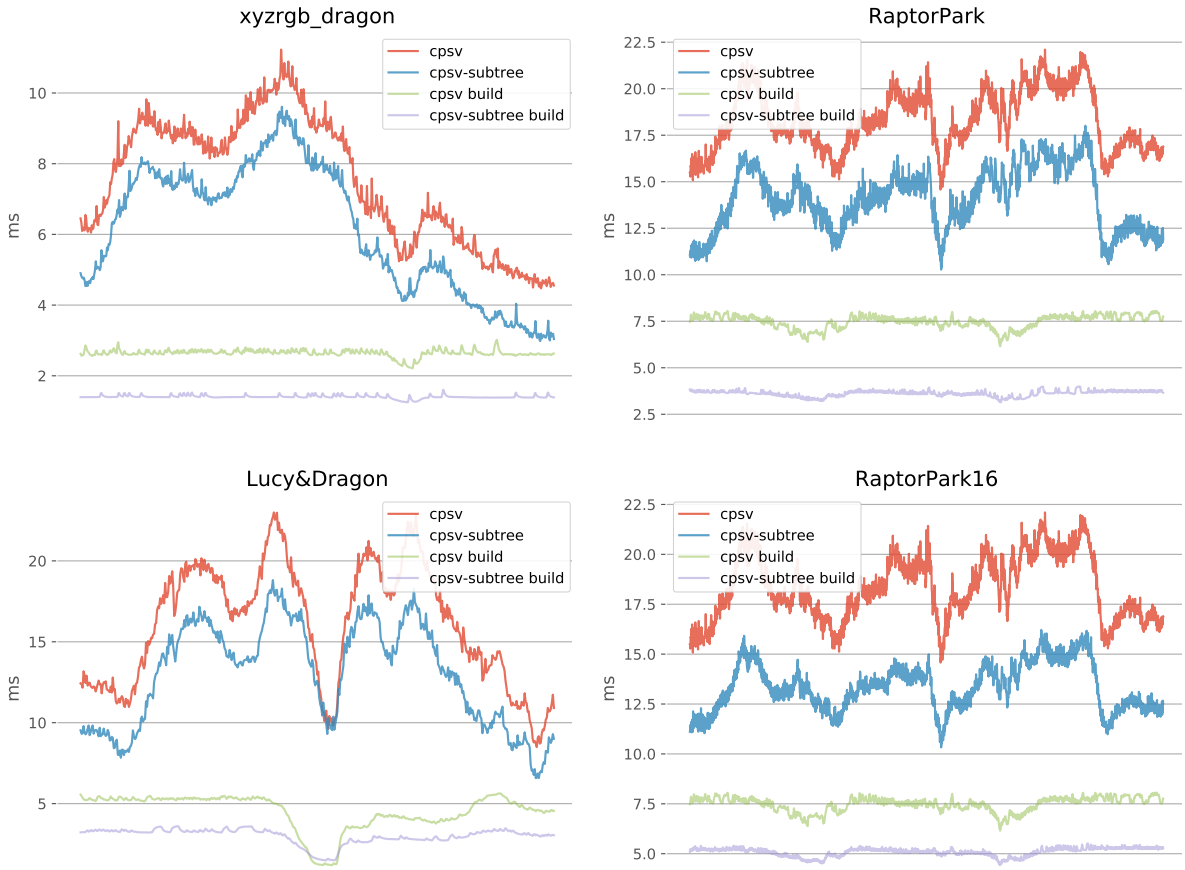


FIGURE 2.18 – Comparaison entre simple et double hiérarchie de TOP-tree. Les courbes rouges (cpsv) et vertes (cpsv-build) sont respectivement le temps total et le temps de construction pour la version simple TOP-tree. Les courbes bleues (cpsv-subtree) et violettes sont respectivement le temps total et le temps de construction pour la version doublement hiérarchique. Les valeurs moyennes et les tailles de cluster utilisées sont indiquées dans la table 2.1.

la taille des modèles, 24 ou 32 triangles par cluster sont des valeurs "d'équilibre" dont on ne peut guère s'écarter avec un simple TOP-tree. Il est intéressant de noter que dans la version doublement hiérarchique, le gain sur le temps de construction autorise plus de souplesse sur la taille des clusters (de bas niveau). Celle-ci peut être plus facilement abaissée pour obtenir un meilleur temps de parcours sans pour autant perdre tout le gain du temps de construction. On peut ainsi optimiser le temps total de la méthode. La figure 2.18 donne un exemple sur la scène RaptorPark où 16 triangles par clusters de bas niveau dégrade le temps de construction (en bas à droite, comparé à une version avec des clusters de taille 24). Mais cette perte est plus que compensée par un parcours plus rapide, délivrant in fine de meilleurs performances globales.

Par la suite, nous utilisons toujours la version doublement hiérarchique pour tous nos résultats et nous nous référerons à cette version par CPSV.

| | cpsv (version simple) | | | | cpsv-subtree (version double hiérarchie) | | | |
|---|---|---|---|---|---|---|---|---|
| | Taille Clust. | Construction | Parcours | Total | Taille Clust. BN | Construction | Parcours | Total |
| **xyrzrgb_dragon** | 8 | 2.64<br>2.18 / 3.65 | 4.86<br>1.87 / 8.25 | 7.5<br>4.74 / 11.23 | 8 | 1.4<br>1.24 / 1.83 | 4.72<br>1.60 / 8.21 | 6.12<br>2.98 / 9.61 |
| **LucyDragon** | 32 | 4.5<br>1.19 / 5.91 | 11.56<br>3.98 / 18.46 | 16.06<br>8.49 / 22.99 | 24 | 3.05<br>1.6 / 4.0 | 10.04<br>3.5 / 16.33 | 13.09<br>6.5 / 19.56 |
| **RaptorPark** | 24 | 7.5<br>5.99 / 8.39 | 10.8<br>7.43 / 14.62 | 18.3<br>14.58 / 22.1 | 24 | 3.61<br>3.1 / 4.15 | 10.4<br>6.86 / 14.33 | 14.01<br>10.27 / 18.0 |
| **RaptorPark16** | 24 | 7.5<br>5.99 / 8.39 | 10.8<br>7.43 / 14.62 | 18.3<br>14.58 / 22.1 | 16 | 5.09<br>4.319 / 5.939 | 8.2<br>5.559 / 10.89 | 13.34<br>10.33 / 16.214 |

Tableau 2.1 – Ce tableau donne les temps moyens correspondant aux graphes de la figure 2.18 pour des promenades virtuelles au sein de différents modèles. En rouge, le détail de la version avec un TOP-tree simple (cpsv). En bleu, le détail de la version doublement hiérarchique (cpsv-subtree). Taille Clust. et Taille Clust. BN donnent respectivement le nombre de triangles par cluster pour la version simple, et le nombre de triangles par cluster de bas niveau dans la version doublement hiérarchique (la taille des clusters de haut niveau est toujours de 2048). Construction est le temps de construction moyen par image du ou des TOP-tree. Parcours est le temps de parcours moyen par image du ou des TOP-tree. Total est la somme des temps de construction et de parcours. Pour chaque valeur moyenne sont également indiquées les valeurs minimales et maximales relevées lors des mesures.

### 2.3.2 Comparaison avec des méthodes de l'état de l'art

Nous comparons bien sûr notre approche à l'algorithme des PSV de Gerhards *et al.* [68] en utilisant l'implémentation disponible en ligne. Nous nous comparons également à la méthode de Wyman *et al.* [98] avec un échantillon par pixel. L'implémentation utilisée est celle disponible dans la librairie ShadowLib de NVIDIA [86]. Nous utiliserons "ShadowLib" pour faire référence à cette méthode. ShadowLib n'est pas une méthode purement géométrique puisqu'elle combine rastérisation et tests géométriques. Mais elle est exacte par pixel et est une des méthodes dédiées au calcul des ombres dures les plus performantes au moment de la réalisation de ce travail.

La Figure 2.19 montre pour chaque image les temps de calcul de chaque méthode sur l'ensemble des scènes testées. Et le Tableau 2.2 synthétise ces mesures en présentant le comportement moyen de chaque méthode.

#### Analyse des performances de notre méthode (CPSV)

La première chose à souligner est que notre méthode, bien que purement géométrique, résiste bien à l'augmentation du nombre de triangles. C'était l'objectif et cela conforte donc la stratégie mise en œuvre. Les PSV ou ShadowLib voient leurs performances se dégrader davantage. En construisant une double hiérarchie de TOP-tree sur des clusters plutôt que sur des triangles (comme c'est le cas pour les PSV), cela permet de contenir son temps de construction même lorsque le nombre de triangles dans la scène devient très important. On note que le temps de construction est toujours inférieur au temps de parcours, quelle que soit les modèles. Le temps de parcours est ainsi 1.7 à 4 fois plus important que le temps de construction, y compris sur *ManyModels* et ses 78.3M de

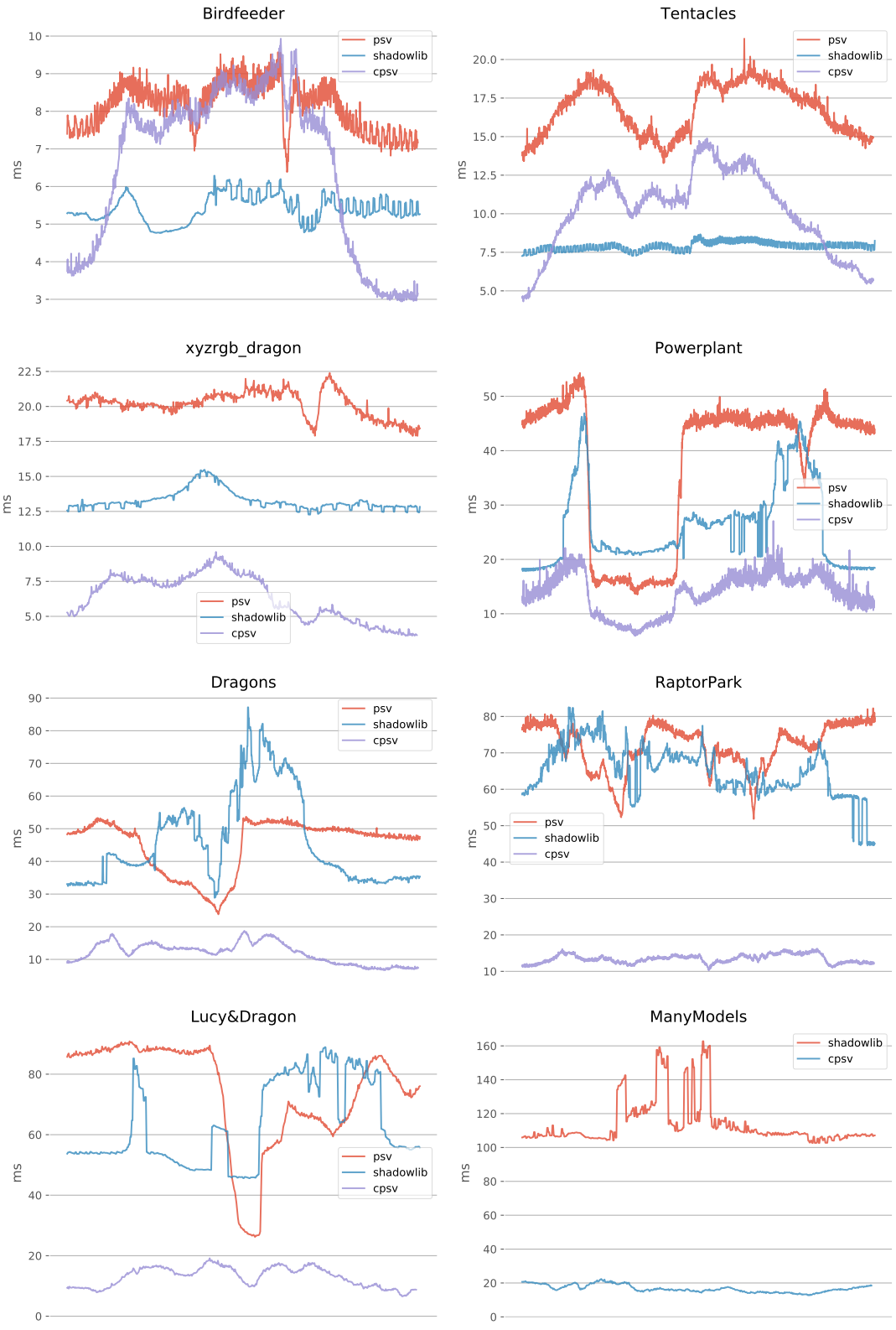


FIGURE 2.19 – Comparaison de notre méthode (CPSV) avec les PSV et la ShadowLib le long de parcours libres sur nos différentes scènes de test.

triangles, qui est faite pour éprouver les limites de notre approche.

| | CPSV | | | | PSV | | | ShadowLib |
|---|---|---|---|---|---|---|---|---|
| | Taille Clust. HN/BN | Construction | Parcours | Total | Construction | Parcours | Total | Total |
| **Birdfeeder (1.8M)** | 2048 / 8 | 0.72 | 5.81 | 6.54 | 5.01 | 3.12 | 8.13 | 5.40 |
| 2.6k + 337k clusters | | 0.52 / 1.13 | 2.20 / 9.22 | 2.92 / 9.93 | 3.11 / 5.66 | 1.84 / 4.7 | 6.38 / 9.57 | 4.72 / 6.28 |
| **Tentacles (3.8M)** | 2048 / 8 | 1.60 | 8.56 | 10.16 | 11.43 | 5.35 | 16.78 | 7.90 |
| 21.7k + 698k clusters | | 1.25 / 2.10 | 2.69 / 13.25 | 4.31 / 14.89 | 8.20 / 14.41 | 1.88 / 7.82 | 13.27 / 21.35 | 7.26 / 8.66 |
| **xyzrgb_dragon (7.2M)** | 2048 / 8 | 2.10 | 4.23 | 6.34 | 16.64 | 3.54 | 20.18 | 13.26 |
| 8k + 1.24M clusters | | 1.867 / 2.55 | 1.542 / 7.203 | 3.607 / 9.608 | 13.65 / 17.87 | 1.54 / 5.36 | 17.88 / 22.39 | 12.26 / 15.47 |
| **Powerplant (12.7M)** | 2048 / 12 | 4.92 | 8.85 | 12.68 | 28.83 | 9.36 | 38.20 | 25.80 |
| 362k + 1.86M clusters | | 1.72 / 7.10 | 4.10 / 21.37 | 5.98 / 23.23 | 9.17 / 38.02 | 4.08 / 18.88 | 13.51 / 54.30 | 17.84 / 46.87 |
| **Dragons (18.9M)** | 2048 / 16 | 2.79 | 9.18 | 11.97 | 39.83 | 5.27 | 45.11 | 45.82 |
| 23.7k + 1.68M clusters | | 1.90 / 3.39 | 3.76 / 15.84 | 6.69 / 18.74 | 19.17 / 47.14 | 2.15 / 8.73 | 23.78 / 53.69 | 28.86 / 87.1 |
| **RaptorPark (30M)** | 2048 / 16 | 5.15 | 8.26 | 13.41 | 66.64 | 5.33 | 71.97 | 65.06 |
| 444.29k + 2.65M clusters | | 4.41 / 5.70 | 5.51 / 10.89 | 10.31 / 16.21 | 46.24 / 75.879 | 3.22 / 7.64 | 51.83 / 82.26 | 44.57 / 82.51 |
| **Lucy&Dragon (35.2M)** | 2048 / 24 | 3.09 | 9.90 | 13.00 | 69.12 | 4.91 | 74.04 | 63.30 |
| 38.9k + 2.09M clusters | | 1.61 / 3.8 | 3.49 / 15.58 | 6.54 / 19.16 | 23.52 / 83.43 | 2.37 / 8.61 | 26.18 / 90.88 | 45.60 / 88.92 |
| **ManyModels (73.8M)** | 2048 / 24 | 6.61 | 10.05 | 16.6 | - | - | - | 113.85 |
| 82.2k + 4.39M clusters | | 5.41 / 7.42 | 6.06 / 15.76 | 12.66 / 22.39 | - / - | - / - | - / - | 102.65 / 162.75 |

Tableau 2.2 – Temps moyens par image le long d'un parcours libre dans les différentes scènes. Les valeurs minimales / maximales relevées sur les parcours sont également mentionnées. CPSV (notre méthode) : "Taille Clust. HN/BN" donne le nombre de clusters de haut et bas niveau. "Construction" est le temps pour construire la double hiérarchie de TOP-tree. "Parcours" est le temps pour la traverser pour ombrer chaque point. "Total" est la somme des deux précédentes. PSV (Gerhards) : "Construction" est le temps pour construire le TOP-tree de triangles. "Parcours" est le temps pour le traverser et ombrer chaque point. "Totale" est la somme des deux précédentes colonnes. ShadowLib : "Total" est le temps moyen par image utilisé par la méthode de Wyman *et al.* [98] pour ombrer chaque point.

Si notre approche résiste bien à l'augmentation du nombre de triangles, le fait qu'elle soit basée sur la géométrie la rend également sensible à la complexité visuelle. S'agissant des ombres, la complexité visuelle est à considérer depuis la source de lumière et se traduit pas la complexité des ombres projetées. Par exemple, notre algorithme est quasiment aussi rapide sur *xyzrgb_dragon* (7.2M de triangles) que sur *Birdfeeder* (1.8M de triangles). Et il est même plus lent sur *Tentacles*, qui possède pourtant moins de triangles (3.8M). Mais *Birdfeeder* et *Tentacles* présentent des ombres visuellement plus complexes que celles produites sur *xyzrgb_dragon*. Cela se traduit par un surcoût pour le parcours de l'arbre métrique ternaire, celui-ci étant une représentation de la vue depuis la lumière. La même analyse peut être faite sur *Powerplant* comparé à *RaptorPark* ou *Lucy&Dragon*.

### Comparaison avec les PSV

Quel que soit le modèle, notre approche demeure plus rapide que les PSV. Conçue pour des modèles de grande taille, on peut néanmoins logiquement penser que sur des modèles de taille inférieure (moins de 1M de triangles), la différence de performance serait faible et il demeurerait plus intéressant d'utiliser les PSV qui ne font aucune restriction sur les transformations géométriques applicables aux modèles. Le constat est différent pour les scènes plus importantes. Le temps de calcul des PSV augmente rapidement avec le nombre de triangles. Au-delà de 10M de triangles, les PSV peinent à rester temps

réel. Même sur les plus grand modèle, le parcours des PSV reste performant de par sa complexité logarithmique. Mais le comportement linéaire de sa construction domine assez vite le coût total jusqu'à dégrader les performances bien au-delà des exigences du temps réel. Les PSV sont tout simplement impraticables sur *ManyModels* par exemple.
En se basant sur des clusters, notre approche conserve une construction rapide puisque le nombre de clusters de la double hiérarchie de TOP-tree est nécessairement inférieur au nombre de triangles (5 à 22 fois moins selon les scènes). Par contre, le coût du parcours de notre structure et supérieur à celui des PSV (car lorsqu'un point est localisé dans un cluster, il faut tester linéairement l'intersection avec tous les triangles qu'il contient). C'est un point que nous avons souligné en introduction de ce chapitre car il est essentiel que ce surcoût prévisible n'excède pas le gain obtenu sur le temps de construction. Ainsi, les résultats montrent que le parcours de notre structure est de 1.1 à 2 fois plus coûteux que celui des PSV. Mais le temps de construction est en revanche de 7 à 23 fois plus rapide que celui des PSV. Le bénéfice demeure donc largement favorable à notre approche.

#### Comparaison avec la ShadowLib (NVIDIA)

Pour cette comparaison avec la méthode de Wyman *et al.*, la résolution du Z-Buffer irrégulier est fixée à $2048^2$ ou $4096^2$ en fonction de la scène pour obtenir des performances optimales. L'optimisation dite de "reprojection dynamique" (qui permet d'éviter des écarts de longueur trop importants entre les listes d'échantillon) améliore nettement les performances et est donc toujours activée, sauf sur *Birdfeeder* où elle n'apporte rien. La ShadowLib se basant sur la rastérisation de la géométrie depuis la source, elle est très peu sensible à la complexité visuelle des ombres projetées, bénéficiant pleinement du support matériel de la carte graphique. C'est un avantage sur les méthodes géométriques comme la notre ou les PSV. Par exemple, ShadowLib est sensiblement plus rapide sur *Birdfeeder* ou bien *Tentacles*, les deux plus petites scènes de nos tests, mais qui présentent les ombres les plus complexes. Pour autant, sur les scènes de plus grande taille, le nombre de triangles rastérisés (conservativement) sur le Z-Buffer irrégulier augmente mécaniquement. Et avec le nombre de test d'intersection avec les listes d'échantillons. C'est pourquoi les temps de calcul augmentent avec la complexité géométrique. Sur le modèle *xyzrgb_dragon* (7.2M de triangles) et *Powerplant* (12.7M de triangles), notre approche est en moyenne deux fois plus rapide que ShadowLib. Sur les autres modèles (entre 18.9M et 73.8M triangles), il varie de 5 à 6.8 en notre faveur.
Il faut toutefois souligner que ce sont des valeurs moyennes. Les écarts restent à notre avantage mais peuvent fluctuer comme l'indique les minimums et maximums relevés le long des parcours dans la table 2.2. Pour une observation plus fine, on peut se reporter aux graphes de la figure 2.19 qui permettent d'apprécier les variations des différentes méthodes le long de chaque parcours.

### 2.3.3 Discussion

#### Environnements dynamiques

Le clustering est un problème NP-difficile. Il n'est pas envisageable de l'appliquer dynamiquement à chaque image, sauf sur des modèles de faible taille. Mais dans ce cas, se baser directement sur les triangles ne pose pas de problèmes de performance. Au-delà du million de triangles, cela ne peut être qu'un pré-calcul. Il en résulte que notre

méthode ne peut pas supporter de la géométrie librement déformable. Elle est restreinte à des transformations rigides. C'est une limitation qu'il faut souligner car cela restreint le champ d'application de notre approche. Toutefois, puisque nous ciblons de très grandes scènes, cette restriction ne nous apparaît pas trop forte dans la mesure où les modèles librement déformables et composés de plusieurs dizaines de million de triangles ne sont pas les plus courants. En outre, cette limite pourrait être un peu repoussée : Tant que les triangles des clusters sont transformés sans pour autant "sortir" de leur volume englobant, notre méthode peut être appliquée. Il pourrait donc être possible de supporter sous cette contrainte des transformations non-rigides. Cela nécessiterait cependant de recalculer à chaque image le cône de normales des clusters.

**Complexité géométrique**

Comme indiqué dans la section 2.2.2, la construction d'un TOP-tree est en $O(n\,log(n))$, mais dans notre approche, $n$ fait référence au nombre de clusters et non au nombre de triangles. Comme le montre les résultats, cela permet d'atteindre notre objectif, à savoir résister à un fort accroissement de la complexité géométrique, car le nombre de clusters reste dans un intervalle que notre méthode sait traiter efficacement. Pour autant, la complexité intrinsèque de la construction de notre structure reste à dominante linéaire en nombre de clusters. Si celui-ci devenait à son tour très important, le coût de la construction deviendrait également problématique. On peut par exemple imaginer des modèles avec plusieurs milliards de triangles qui deviendraient des dizaines de millions de clusters. Pour diminuer le nombre de clusters, il est toujours possible d'augmenter leur taille. Mais cela engendrera un surcoût sur le parcours puisqu'il y aura plus de triangles à tester lorsqu'un cluster occultera un point. Entre 10M et 30M de triangles, nos tests (voir la Table 2.2) montrent que 12 à 16 triangles par clusters de bas niveau donne le meilleur équilibre avec le temps de parcours. Au-delà de 30M, 24 triangles est plus approprié. Il existe nécessairement un seuil au-delà duquel l'équilibre ne pourra plus se faire tout en maintenant de bonnes performances : Limiter le nombre de clusters nécessiterait trop de triangles par clusters et donc un parcours trop coûteux. Dans le même temps, diminuer le nombre de triangles par cluster engendrerait trop de clusters pour avoir une construction efficace. Cependant les résultats montrent que notre méthode demeure efficace sur de très grands modèles, en particulier lorsqu'ils sont finement triangulés, ce qui favorise un bon clustering. Sur ces tests nous n'avons pas atteint de seuil où l'équilibre entre construction et parcours ne peut plus se faire dans un cadre temps réel.

## 2.4 Conclusion et perspectives

L'objectif premier de ce travail pour le rendu des ombres dures en temps réel, était de créer un algorithme purement géométrique pour garantir un résultat exact par pixel tout en étant capable de résister à une forte augmentation de la complexité géométrique, point faible des autres méthodes s'inscrivant dans la même catégorie. De ce point de vue l'algorithme présenté répond aux attentes. En utilisant des clusters plutôt que des triangles, notre approche contient la complexité géométrique et maintien des performances compatibles avec une application temps réel. Cette stratégie est relativement commune et fait encore ici ses preuves.
Nous avons proposé et testé différentes options pour parfaire notre approche. Ainsi nous

avons cherché à optimiser le conservatisme des boites englobantes placées sur les clusters en fonction de l'algorithme de clustering utilisé. Nous avons optimisé la représentation par un TOP-tree du volume d'ombre généré par une boite orientée. Nous avons adapté la construction et le parcours d'un TOP-tree à des clusters plutôt que des triangles. Et enfin nous avons proposé une structure différente, une double hiérarchie de TOP-tree qui peut être construite plus efficacement sur GPU et qui demeure tout aussi rapide à parcourir. La double hiérarchie de TOP-tree autorise en outre plus de variation sur la taille des clusters pour trouver le meilleur compromis entre temps de construction et temps de parcours afin d'optimiser les performances globales de l'algorithme.
Au final, nos tests ont montré que sur des scènes de grandes tailles, notre approche surclassait sans difficulté des méthodes telles que les PSV ou la ShadowLib. Les scènes dynamiques et librement déformables ne sont toutefois pas supportées par notre approche, seules les transformations rigides le sont. Comme discuté précédemment, cette contrainte peut toutefois être relativisée sur de très grands modèles.

Les méthodes purement géométriques gardent nécessairement une dépendance à la complexité géométrique. Pour la contenir, il faut parvenir à diminuer cette complexité et donc diminuer le nombre de primitives traitées. Dans ce chapitre, nous avons pour cela exploiter une stratégie basée sur le clustering pour nous ramener à un niveau de performances temps réelles. Une autre voie théoriquement possible est d'exploiter le fait que sur de très grands modèles, la résolution géométrique du modèle excède souvent de loin la résolution écran. C'est à dire que le nombre de triangles qui contribuent à l'image peut être bien plus faible que le nombre de triangles dans la scène. Un principe renforcé ces dernières années par la forte augmentation de la résolution des écrans. Au moment de choisir la stratégie à suivre, nous avons aussi (un peu) exploré cette piste sans que nos tests soient aussi prometteurs que la piste du clustering. Pour autant c'est une approche qu'il serait intéressant d'approfondir dans la mesure où elle ne serait pas soumise à la contrainte d'utiliser des modèles non librement déformables, contrainte liée au clustering.

# Chapitre 3 :

# Arbres métriques : Une stratégie de partitionnement alternative

Une version des travaux expliqués dans ce troisième chapitre a été présentée en 2018 à Karlsruhe en Allemagne, lors de la conférence *Eurographics Symposium on Rendering*, dans la catégorie *Expérimental Ideas & Implementations* [28].

## 3.1 Introduction et motivation

Dans le précédent chapitre, le propos était de rendre robuste à la complexité géométrique un algorithme pour le rendu des ombres dures en temps réel. Nous nous sommes basés sur la technique des PSV proposée par Gerhards *et al.*, dont les complexités intrinsèques en temps de construction et en parcours étaient adaptées à notre approche. En construisant un TOP-tree sur les boites orientées de clusters pré-calculés au lieu de le construire avec tous les triangles de la scène, nous avons montré que le coût de construction de la structure pouvait être contenu, offrant un gain de temps qui compense de loin le sur-coût induit sur son parcours. En procédant ainsi, il devient possible de traiter de gros modèles que les PSV originaux ne peuvent traiter en temps réel. De manière générale, cette approche, dans son principe, est indépendante de la structure accélératrice utilisée. Sans que l'on puisse pré-juger des performances que l'on obtiendrait, il est toujours possible de construire n'importe quel type de structure accélératrice sur des volumes englobants plutôt que sur la géométrie qu'ils contiennent. Et pour cause, des volumes englobants demeurent des primitives géométriques.
S'agissant de géométrie, une structure accélératrice est en premier lieu le produit d'une méthode de partitionnement. Par exemple, les arbres BSP (*Binary Space Partitioning*) ou les *kd*-tree produisent des partitions de l'espace par des plans. Les TOP-tree correspondent aussi à une partition par des plans, mais de l'espace objet cette fois. Citons encore les hiérarchies de volumes englobantes qui sont une autre approche bien connue pour partitionner l'espace objet.

Ce troisième chapitre est davantage exploratoire et se propose d'expérimenter une stratégie de partitionnement qui, à notre connaissance, n'a jamais été utilisée en synthèse d'image : Les arbres métriques. La stratégie de partitionnement sous-jacente à un arbre métrique repose sur la distance des éléments les uns aux autres. Cela impose donc qu'une distance soit calculable. Mais c'est aussi une formulation générale qui peut être utilisée pour des éléments non vectoriels du moment que l'on dispose d'une fonction de distance qui leur soit applicable. Pour construire un arbre métrique, la stratégie de partitionnement consiste à choisir un élément, dit pivot, une distance de partitionnement $\delta$, puis à subdiviser tous les autres éléments en deux ensembles : ceux dont la distance au pivot est inférieure ou égale à $\delta$, et ceux dont la distance au pivot est supérieure à $\delta$. La structure d'arbre métrique s'obtient en appliquant récursivement le procédé sur les deux sous-ensembles ainsi formés tant qu'ils sont non vides.

Nous allons étudier comment passer d'une structure de TOP-tree à une structure d'arbre métrique, ce qui veut dire changer de stratégie de partitionnement. Au delà de l'intérêt que suscite l'étude d'une stratégie encore non utilisée dans notre contexte, disposer d'une structure qui repose par définition sur la distance aux éléments est intéressante. En effet dans notre cas, connaître la distance à la géométrie ou aux volumes d'ombres qu'elle génère est une information toujours utile. C'est par ailleurs la base

d'algorithme de rendu classique comme le tracé de sphères (*sphere tracing* [44]) souvent utilisé pour le rendu de modèles procéduraux : Si en une position donnée le plus proche obstacle est à une distance $d$, il est possible d'avancer de $d$ sans qu'il soit utile de rechercher de nouvelles intersections. La distance au plus proche obstacle peut ensuite être utilisée pour des techniques d'anti-aliassage ou des pseudo-effets ombres douces. De manière générale, cela permet d'étendre une information de visibilité calculée en un point ou entre deux points à un voisinage dont la limite est fixée par la distance au plus proche obstacle.

On peut toutefois souligner que le parcours d'un TOP-tree (ou plus généralement d'un arbre BSP) se fait aussi selon une distance signée : Pour chaque nœud rencontré, la position d'un point par rapport au plan de subdivision contenu est testée. C'est un produit scalaire dont la valeur absolue correspond à la distance du point au plan (celui-ci étant normé et passant par l'origine). Il serait alors possible de conserver la plus petite distance rencontrée le long du parcours, définissant un voisinage à l'intérieur duquel tous les points suivront un parcours identique. Par conséquent leur visibilité depuis la source serait la même. Mais dans ce cas, il s'agit de la plus petite distance aux plans du TOP-tree, pas nécessairement à la géométrie qu'ils partitionnent. Nous avons néanmoins tenté d'exploiter cette distance pour faire du sur-échantillonage (*upsampling*), mais la divergence entre la géométrie "réelle" et celle induite par la structure était trop importante pour que l'application soit convaincante.

En premier lieu, nous commençons par rappeler la définition formelle des arbres métriques. Puis nous étendons cette définition aux arbres métriques ternaires pour prendre en compte des objets volumétriques. Nous définissons ensuite plus précisément quels sont nos objets et la distance utilisée. Nous présentons alors notre approche nous permettant de construire un arbre métrique ternaire sur de simples triangles ou bien sur des clusters de triangles que nous aurons pré-calculés comme détaillé dans le chapitre précédent. Enfin, nous évaluons dans la partie résultat le potentiel des arbres métriques ternaires en les comparant aux TOP-trees. Pour les deux approches, nous présentons également des comparaisons avec des ombres calculées par lancer de rayons sur GPU, dont les performances ont grandement progressées durant la dernière année de cette thèse, notamment grâce à un support matériel qui n'existait pas auparavant.

## 3.2 Les arbres métriques

### 3.2.1 Définition

Les arbres métriques [89, 101] partitionnent un ensemble de données dans un espace métrique en se basant uniquement sur une distance. Définis en toutes dimensions, les arbres métriques sont souvent utilisés pour des recherches de plus proche voisinage ou de similitude. À notre connaissance, ils n'ont pas encore été utilisés en synthèse d'images.
Soit $\mathcal{O}$ un ensemble d'éléments à partitionner, $d$ une fonction permettant de mesurer une distance entre les éléments de $\mathcal{O}$. Alors $\mathcal{M}(\mathcal{O}, d)$ est un espace métrique si les conditions suivantes sur $d$ sont vérifiées :

- $\forall\, x,\, y \in \mathcal{O},\ d(x,\, y) \geq 0$ (non négative)
- $\forall\, x,\, y \in \mathcal{O},\ d(x,\, y) = d(y,\, x)$ (symétrie)
- $\forall\, x,\, y,\, z \in \mathcal{O},\ d(x,\, z) \leq d(x,\, y) + d(y,\, z)$ (inégalité triangulaire)

Un ensemble d'éléments dans un espace métrique $\mathcal{M}$ peut être organisé dans un arbre métrique comme suit : Chaque nœud de l'arbre contient un élément $p$ associé à une distance $\delta$. $p$ sert de pivot pour subdiviser les autres éléments en deux sous-ensembles disjoints $S_n$ et $S_f$ tels que :

$$S_n = \{x \in \mathcal{O} \mid d(p,\, x) \leq \delta\}$$
$$S_f = \{x \in \mathcal{O} \mid d(p,\, x) > \delta\}$$

$S_n$ contient tous les éléments dont la distance à $p$ est inférieure ou égale à $\delta$. $S_f$ contient tous les éléments dont la distance à $p$ est supérieure à $\delta$.

### 3.2.2 Arbre métrique ternaire

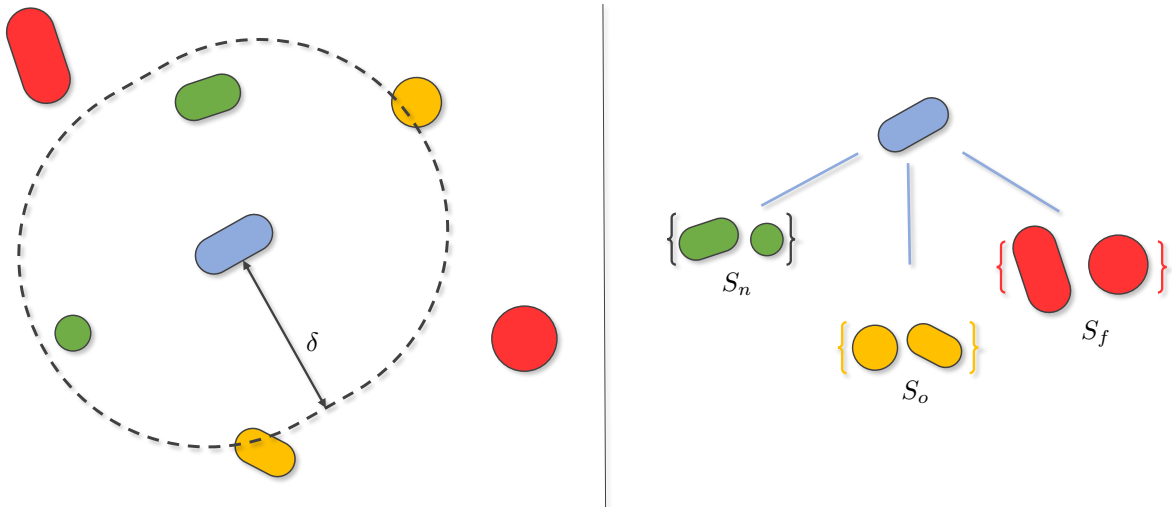


FIGURE 3.1 – Illustration 2D d'une partition ternaire des objets en fonction de leur distance (euclidienne pour cet exemple) à un élément pivot. À gauche : La capsule bleue est l'élément pivot avec une distance de partitionnement $\delta$. À droite : L'arbre métrique ternaire correspondant à la configuration de gauche. Les autres objets sont subdivisés en 3 ensembles en fonction de leur distance au pivot comparée à $\delta$.

La définition précédente vaut pour des éléments qui sont des points dans leur espace de définition. Mais dans notre cas nous aurons à traiter des éléments volumétriques. Par conséquent certains pourraient se trouver en partie dans les deux sous-ensembles $S_n$ et $S_f$. Ce type de cas peut être traité en passant d'une structure binaire à une structure ternaire, similairement à la modification apportée par Gerhards *et al.* au BSP de Chin et Feiner. Aussi nous définissons un troisième fils pour chaque nœud et passons d'un arbre métrique à un arbre métrique ternaire pour tenir compte des objets volumétriques. La distance $\delta$ associée au pivot $p$ d'un nœud partitionne les éléments restant en 3 sous-ensembles $S_n$, $S_f$ and $S_o$ tels que :

$$S_n = \{o \in \mathcal{O} \mid d^{far}(p,\, o) \leq \delta\}$$
$$S_f = \{o \in \mathcal{O} \mid d^{near}(p,\, o) > \delta\}$$
$$S_o = \{o \in \mathcal{O} \mid d^{near}(p,\, o) < \delta < d^{far}(p,\, o)\}$$

Avec $d^{near}(p, o)$ (resp. $d^{far}(p, o)$) la plus proche (resp. grande) distance de $p$ à n'importe quel point appartenant à $o$. La Figure 3.1 montre les 3 ensembles ainsi formés.

### 3.2.3 Nature des éléments et distance utilisée

Les arbres métriques (ternaires ou non) ont ceci d'intéressant qu'ils peuvent être construits pour n'importe quel type d'élément tant que l'on est capable de définir une distance entre ces éléments. Les éléments avec lesquels nous voulons construire un arbre métrique ternaire sont des volumes d'ombres. Il nous faut donc préciser ce qu'est la distance entre des volumes d'ombres. Ici la distance euclidienne ne fait pas sens puisque notre problématique se pose dans un espace projectif dont l'origine est la lumière. La distance adaptée est la distance angulaire qui permet de caractériser la distance apparente entre des objets depuis un point d'observation donné (pour nous, la source de lumière).

Si l'on considère deux triangles et leur volume d'ombre dont la lumière $O$ est le sommet commun, leur distance angulaire correspond au plus petit angle de sommet $O$ mesurable entre les deux pyramides. Nos premiers prototypes fonctionnels se basaient sur des volumes d'ombre de triangles et utilisaient cette distance. Toutefois, calculer l'angle minimale entre deux pyramides est assez coûteux et limitait les performances. Aussi nous avons décidé de ne pas nous baser sur les triangles, mais sur un volume englobant. Cela présentait de plus l'avantage de pouvoir traiter des triangles ou bien des clusters de triangles. Mais contrairement au chapitre précédent, il n'est pas raisonnable d'utiliser des boites orientées. En effet, le calcul de la distance angulaire entre les volumes d'ombre de deux boites orientées est encore plus coûteux qu'entre les volumes d'ombre de deux triangles. Pour

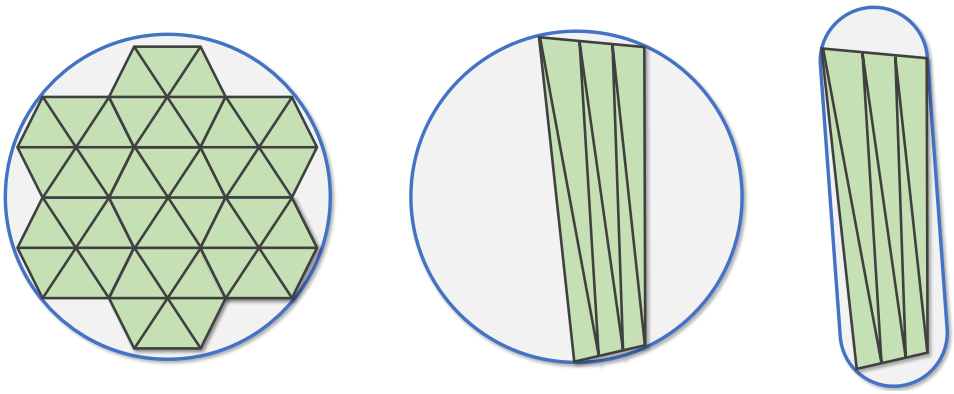

FIGURE 3.2 – Volumes englobants des clusters. À gauche : Nous utilisons des sphères pour englober la géométrie (simple triangle ou cluster de triangles). Au milieu : Néanmoins, les sphères sont très conservatives en présence de triangles allongés. À droite : Dans ce cas, une capsule est plus appropriée.

cette raison, nous avons écartés les boîtes englobantes au profit de sphères englobantes. L'intérêt des sphères est que quelle que soit la position de la source, le volume d'ombre qu'elles projettent correspond à un cône, une forme simple et compacte en mémoire. En revanche, une sphère peut être très conservative par rapport à la géométrie qu'elle contient. C'est en particulier le cas lorsque le triangle ou le cluster est allongé, comme l'illustre

la Figure 3.2. En plus des sphères, nous utilisons des capsules (une sphère extrudée le long d'un segment) puisqu'elles sont mieux adaptées à ces configurations. Ce principe est analogue à ce que propose Larsen [55] dans un contexte de détection de collision. Nous appelons cône-capsule le volume d'ombre engendré par une capsule depuis la lumière. Nos

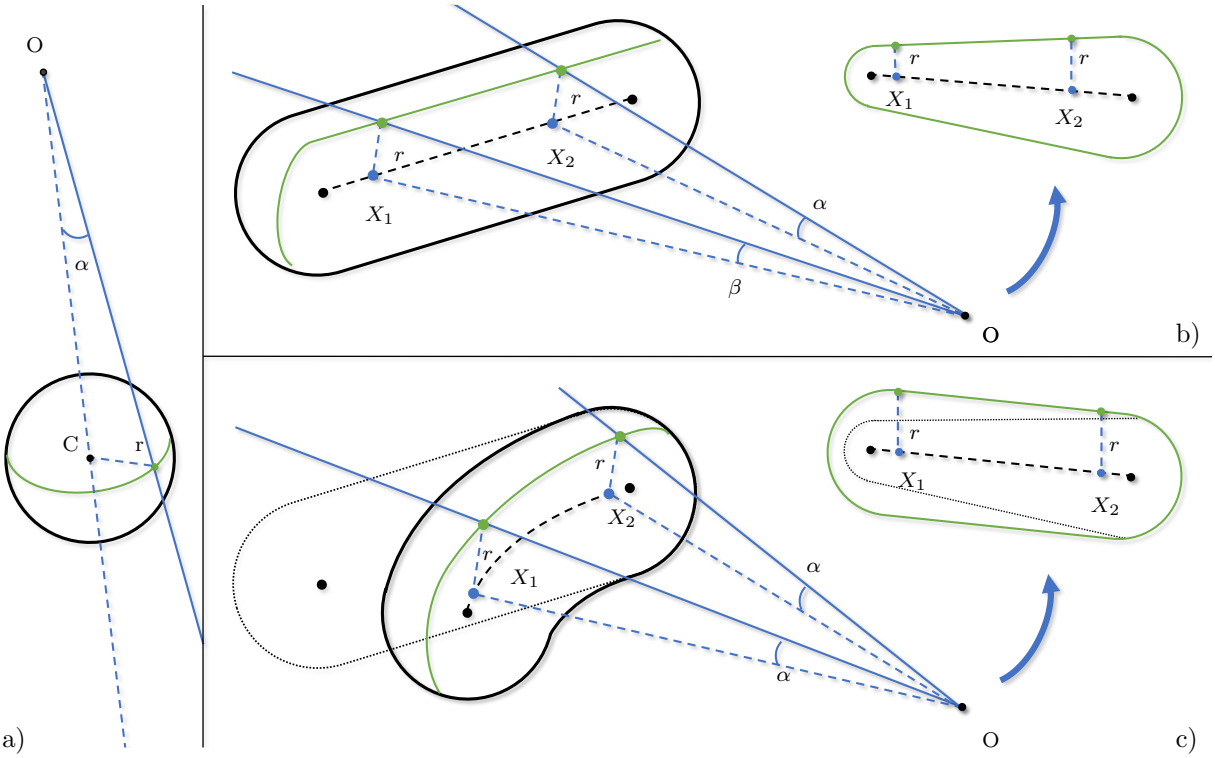


FIGURE 3.3 – Représentation d'un cône et d'un cône-capsule. $O$ est la position de la lumière. La silhouette de la sphère ou de la capsule vue depuis la lumière apparaît en rouge. (a) Représentation d'un cône : Les points contenus dans un cône ont une distance angulaire au centre de la sphère $C$ inférieure ou égale à $\alpha = \arcsin(\frac{r}{OC})$. (b) Cas plus particulier d'un cône-capsule : Contrairement à un cône, une valeur angulaire fixe ne permet pas de caractériser les points sur la surface de ce volume. En raison de la perspective, la distance apparente du bord de la capsule à son segment central varie le long de ce segment. Par exemple, $\alpha > \beta$ car $X_2$ est plus proche de $O$ que $X_1$ ($X_1$ et $X_2$ étant deux points quelconques du segment $AB$). Vue depuis la source (en haut à droite), la capsule apparaît plus large au niveau de $X_2$ que de $X_1$. (c) Représentation conservative d'un cône-capsule : Nous considérons la plus grande distance angulaire de la capsule, où la distance euclidienne du segment support à la lumière est la plus courte. Intuitivement, cela revient à "courber" le segment support de la capsule en un arc de cercle pour compenser la variation de taille due à la perspective. (en haut à droite).

éléments correspondent donc à des cônes et des cône-capsules. Nous définissons à présent plus précisément leur distance angulaire. Considérant une source omnidirectionnelle $O$, la distance angulaire entre 2 points $A$ et $B$ est l'angle formé par les droites $(OA)$ et $(OB)$ :

$$d(A, B) = \widehat{AOB}$$

Étant donnée une sphère de centre $C$ et de rayon $r$, le cône correspondant dans l'espace

de la lumière peut être défini comme l'ensemble des points dont la distance angulaire est inférieure ou égale à $\alpha = \arcsin(\frac{r}{OC})$ (voir la Figure 3.3.a). Le cas d'un cône-capsule est un peu plus compliqué car la taille apparente d'une capsule vue depuis la source varie le long de son axe principal en raison de la perspective. Si l'on voit une capsule comme une sphère extrudée le long d'un segment, alors projectivement la taille de la sphère va varier selon si elle se rapproche ou s'éloigne du point de vue. Si l'on considère une capsule de segment support $AB$ et de rayon $r$, un point $P$ est à l'intérieur du cône-capsule correspondant si la plus petite distance angulaire de $P$ à $X$ (un point du segment $AB$) est inférieure ou égale à $\alpha = \arcsin(\frac{r}{OX})$ où (voir la Figure 3.3.b).

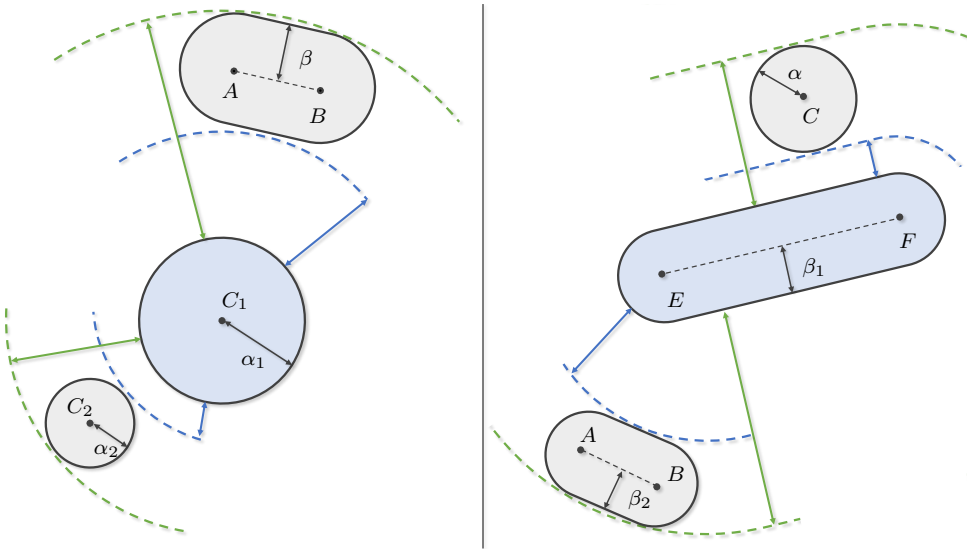


FIGURE 3.4 – La figure est vue depuis la source et intègre la compensation de la perspective pour les cônes-capsules. Les distances angulaires en bleu sont les distances les plus proches ($d^{near}$) et les distances angulaires en vert sont les plus grandes ($d^{far}$). À gauche, les distances angulaires illustrées relativement à un cône. À droite, les distances angulaires illustrées relativement à un cône-capsule.

Pour limiter et simplifier les calculs, nous faisons à la place une approximation conservative en utilisant la plus grande distance angulaire entre le bord de la capsule et son segment support. Nous déterminons $X$ sur le segment $AB$ tel que $\arcsin(\frac{r}{OX})$ soit maximal. On peut remarquer que la taille apparente de la capsule est plus importante là où la capsule est la plus proche du point d'observation (ce qui est proche apparaît plus grand). Aussi cela revient à minimiser la distance (euclidienne cette fois) $OX$ (voir la Figure 3.3.c). Cette approximation implique que l'occultation d'une capsule est surestimée. Toutefois en pratique l'impact est faible car si la taille apparente d'une capsule varie, cette variation est généralement moindre. Il faut imaginer des configurations peu réalistes où une extrémité de la capsule est très proche de la lumière et l'autre très éloignée pour créer d'importantes variations.
En outre, cette approximation permet de représenter les cônes comme les cônes-capsules par un élément central (point ou segment) et une valeur angulaire fixe. Nous pouvons

alors calculer $d^{near}$ et $d^{far}$ comme suit :

$$d^{near}(V_1, V_2) = d(C_1, C_2) - (\alpha_1 + \alpha_2)$$
$$d^{far}\ (V_1, V_2) = d(C_1, C_2) - (\alpha_1 - \alpha_2)$$

Où $V_i$ est un cône ou un cône-capsule d'élément central $C_i$ (un point ou un segment donc) et de valeur angulaire $\alpha_i$ (voir la Figure 3.4). En utilisant ces deux distances, nous pouvons à présent construire une arbre métrique ternaire sur l'ensemble des cônes et des cônes-capsules, qu'ils renferment un triangle ou un cluster de triangles.

Nous avons défini tout ce dont nous avons besoin pour construire un arbre métrique ternaire partitionnant les volumes d'ombres de sphères et de capsules en fonction de la distance angulaire qui les séparent. Nous détaillons à présent les différentes étapes de notre approche pour construire et parcourir cette structure afin de calculer des ombres dures en temps réel.

## 3.3 Approche proposée

Nous commençons par décrire brièvement les grandes étapes de notre approche. Chacune sera détaillée ultérieurement.

### 3.3.1 Principales étapes de l'algorithme

#### Sphères ou capsules englobantes des triangles ou des clusters

L'étape de clustering est optionnelle et identique à celle que nous avons décrite dans le précédent chapitre. Comme nous l'avons vu, elle est surtout nécessaire lorsque la taille du modèle devient importante. Pour des modèles dont la taille n'excède pas 1 à 2 millions de triangles, la géométrie peut être traitée directement, autorisant dans ce cas le support de scènes dynamiques et librement déformables. En revanche si des clusters sont précalculés, la restriction à des transformations rigides s'applique. Qu'il s'agisse de triangles individuels ou de clusters, une sphère ou une capsule englobante est calculée pour chacun en fonction de sa "forme" afin de ne pas produire des volumes trop conservatifs.

#### Construction de l'arbre métrique ternaire

La construction de l'arbre métrique prend en entrée un ensemble de sphères et de capsules. Nous commençons par calculer les valeurs angulaires permettant de définir depuis la source les cônes et les cônes-capsules correspondants. La construction de la structure peut débuter : un cône (ou cône-capsule) est choisi en tant que pivot, initialisant la racine de l'arbre ternaire. Une distance angulaire de partitionnement est choisie et, selon la distance angulaire des autres éléments au pivot, ceux-ci sont insérés dans l'un des trois sous-arbres correspondant aux trois ensembles possibles. Et ainsi de suite. Une différence notable est qu'ici un seul nœud d'un arbre métrique correspond à un volume d'ombre alors que dans le précédent chapitre, il fallait quatre nœud d'un TOP-tree pour représenter un volume d'ombre.

### Parcours de l'arbre métrique

À chaque image, un parcours de l'arbre métrique est effectué pour chaque point image afin de déterminer si celui-ci se trouve à l'ombre ou bien s'il est éclairé. Chaque nœud de l'arbre métrique contient un cône ou un cône-capsule. Lorsqu'un point rencontre un nœud de l'arbre, l'algorithme teste sa position par rapport au volume englobant qu'il contient. Si le point est à l'intérieur du volume, on effectue un test d'intersection avec le triangle qu'il contient, ou bien les triangles contenus s'il s'agit de clusters. Si le point est à l'extérieur du volume englobant, le ou les triangles ne sont pas testés et le parcours du point se poursuit dans l'un des fils du nœud courant. Le processus se termine lorsqu'une intersection est trouvée (le point est à l'ombre) ou bien lorsqu'une feuille est atteinte (le point est éclairé) et qu'il ne reste plus de nœud à visiter.

Nous détaillons à présent chacune des étapes précédentes.

## 3.3.2 Sphères/capsules englobantes des triangles ou des clusters

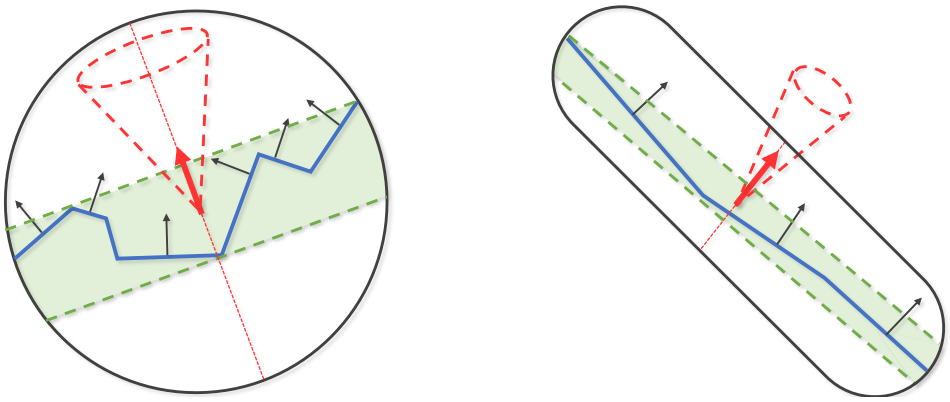

FIGURE 3.5 – Un cône de normales est calculé pour chaque sphère ou capsule englobante pour permettre la suppression des clusters ne contenant que des triangles faisant face à la lumière. Selon la configuration et selon le point de vue (c'est-à-dire la position de la lumière), une sphère ou une capsule peut être très conservative. Aussi nous ajoutons deux plans (appelés "slabs") orthogonaux (en pointillés verts sur le schéma) à l'axe du cône de normales et bornant la géométrie le long de cet axe .

Dans le cas où des clusters sont utilisés, nous calculons tout d'abord sa boite englobante orientée comme détaillé dans le précédent chapitre. Puis selon les caractéristiques de cette boite, nous choisissons d'utiliser une sphère englobante ou une capsule englobante. Leur calcul repose également sur la boite orientée.

Plus précisément, les axes de la boîte sont utilisés pour déterminer si nous utilisons une capsule ou une sphère : Si la plus grande dimension de la boite englobante est deux fois plus grande que les deux autres, la géométrie du cluster tend à être répartie le long d'un même axe et nous utilisons une capsule. Sinon nous utilisons une sphère. Pour calculer une sphère englobante, nous calculons la plus petite sphère englobante des sommets des triangles du cluster [38]. Si une capsule est choisie, les sommets des triangles du cluster sont projetés sur le plan orthogonal à l'axe principal de la boite orientée. Ensuite nous calculons le plus petit cercle englobant la projection des sommets (version

2D de la sphère englobante). Ce cercle détermine la section de la capsule de même que son axe, la perpendiculaire au centre du cercle. Enfin la longueur de la capsule est égale à la plus grande dimension de la boite englobante. Dans tous les cas, un cône de normales est également calculé pour permettre la suppression des volumes qui ne contiendraient que de la géométrie faisant face à la lumière.
Sphères et capsules peuvent tout de même rester des volumes très conservatifs selon les configurations géométriques. Lorsque cela se produit, Larsen *et al.* utilisent aussi des volumes (*rectangular swept spheres*) résultant de la convolution d'un rectangle et d'une sphère. Dans notre contexte, c'est une forme trop complexe et trop coûteuse à manipuler que ce soit en mémoire ou en temps de calcul.
Néanmoins, pour éviter des situations trop conservatives, nous calculons deux plans additionnels, nommés "slabs" pour restreindre le volume englobant à la géométrie contenue. Ces plans sont orthogonaux à l'axe du cône de normales comme illustré par la figure 3.5. Dans le cas d'une capsule, l'ajout de ces deux plans permet d'approcher le volume d'une boite orientée tout en étant plus économe en mémoire (48 octets au lieu de 84 dans le cas d'une boite orientée).

Les définitions de sphère ou de capsule englobante d'un cluster peuvent aussi être appliquées à un unique triangle que l'on peut voir comme un "cluster unitaire". Il est toutefois possible de procéder plus efficacement dans ce cas, en calculant le cône ou le cône-capsule englobant du triangle à la volée (donc de supporter les déformations libres) sur les sommets du triangle dans l'espace projectif de la lumière. Cela permet d'adapter au mieux le volume d'ombre englobant d'un triangle tel qu'il est vu depuis la lumière. Pour des cônes, il suffit simplement de calculer le cercle englobant minimal de ses sommets. Pour des capsules, le procédé est illustré dans la Figure 3.6. Le choix du volume englobant est fait comme lors du pré-calcul des clusters : si la plus grande arête apparente du triangle (c'est-à-dire en considérant l'angle formé par les sommets de l'arête et la lumière) est $x$ fois supérieure à la plus petite arête alors une capsule est utilisée, sinon un cône. Le choix du paramètre $x$ permet de contrôler la proportion de capsules utilisées. En théorie, il est intéressant d'utiliser des capsules dès que possible pour réduire le conservatisme des volumes englobants. Vue depuis la lumière, de nombreux triangles allongés apparaissent sur les contours du modèle. Cependant, comme nous le verrons dans la section résultats, le surcoût entraîné par le calcul de distance aux capsules peut devenir prohibitif si elles sont en trop grand nombre. Ainsi il est plus raisonnable de limiter leur usage aux triangles particulièrement allongés. D'après nos tests, une valeur de $x$ comprise entre 8 et 12 semble être un bon compromis entre la réduction du conservatisme et la vitesse de calcul.

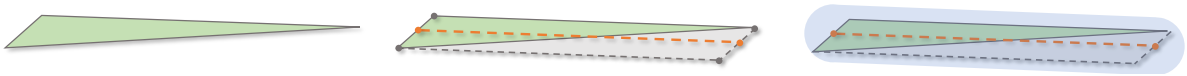

FIGURE 3.6 – À gauche : un triangle vue depuis la lumière dont on souhaite calculer un cône-capsule englobant. Au milieu, le parallélogramme issu du triangle est construit. Le segment qui passe par les points centraux des deux plus petites arêtes du parallélogramme sert de segment support au cône-capsule. La distance angulaire maximale des sommets du parallélogramme à ce segment support nous donne l'angle du cône-capsule (en bleu à droite).

### 3.3.3 Construction de l'arbre métrique

Les algorithmes proposés dans la littérature pour construire un arbre métrique sont similaires au tri rapide (*quicksort*). La racine de l'arbre est initialisée avec un pivot choisi parmi tous les éléments et une distance de partitionnement $\delta$ lui est attribuée. Ensuite, les autres éléments sont réparties dans les nœuds fils en fonction de leur distance au pivot comparée à $\delta$. Le processus se poursuit ensuite récursivement dans chacun des fils jusqu'à ce que tous les éléments soient placés dans un nœud de l'arbre. Les deux points clés de la construction sont : Comment choisir un pivot et comment choisir $\delta$. Ces deux questions sont au centre des précédents travaux sur la construction des arbres métriques.
Le plus souvent, la sélection du pivot est faite aléatoirement. C'est une façon de garantir en moyenne l'équilibre de l'arbre produit tout en restant peu sensible à la nature des données traitées. Yianilos [99] propose de choisir pour pivot celui qui maximise sa distance aux autres éléments. Il montre que cela peut sensiblement améliorer la construction. Le procédé est en revanche bien plus coûteux. Concernant le choix de $\delta$ associé à un pivot, Uhlmann [89] propose de la fixer à la distance médiane du pivot aux autres éléments. Cela permet d'assurer une subdivision des éléments en deux sous-ensembles de taille équivalente et donc de favoriser l'équilibre de la structure. Mais le coût du calcul de la distance médiane est proportionnel au nombre d'éléments présents dans un nœud. Une telle approche, bien que possible sur GPU, nécessiterait une synchronisation de la construction à chaque niveau de l'arbre. Un surcoût qu'on ne peut pas se permettre dans un contexte temps réel.
Aussi notre implémentation demeure similaire à la parallélisation proposée par Gerhards *et al.* qui a l'avantage d'être particulière simple à mettre en œuvre sur GPU et que nous avons déjà réemployée dans le précédent chapitre. Toutefois, nous allons voir que ce choix présente aussi des inconvénients. Notre construction GPU insère indépendamment les éléments (cônes ou cônes-capsules) dans l'arbre métrique ternaire. Les threads consomment l'ensemble d'éléments un à un jusqu'à ce qu'il soit vide.
Plus précisément, chaque thread accède au volume englobant du cluster. En testant son cône de normales par rapport à la position de la lumière, le cluster est inséré ou non. Si oui, le thread calcule le cône ou le cône-capsule (selon la nature du volume englobant du cluster) et l'insère à la racine de l'arbre métrique ternaire. Ses distances angulaires $d^{near}$ et $d^{far}$ au pivot du nœud sont calculées et comparées à la distance de partitionnement $\delta$ du pivot. En fonction, le processus se poursuit dans le fils approprié du nœud. Lorsqu'une feuille est atteinte, elle est remplacée par un nouveau nœud avec le cône ou le cône-capsule en pivot. Ces accès en écriture à la structure peuvent générer des accès concurrents entre les threads actifs. Ils sont gérés par des opérations atomiques.
L'aléatoire (ou pseudo-aléatoire) dans la sélection et l'insertion des éléments dans l'arbre est produit par l'ordonnancement des threads par le GPU et le jeu de opérations atomiques lors des accès concurrents. C'est une façon implicite de produire de l'aléatoire sur le GPU.

Par contre, en insérant les éléments indépendamment les uns des autres, il devient compliqué de calculer la distance de partitionnement $\delta$ associée à un pivot. Par construction, le pivot d'un nœud ayant remplacé une feuille est le premier élément à avoir atteint cette feuille. Au moment où la valeur de $\delta$ doit être fixée, l'algorithme n'a aucune connaissance des autres éléments qui atteindront ce nouveau nœud. Par exemple il n'est pas possible

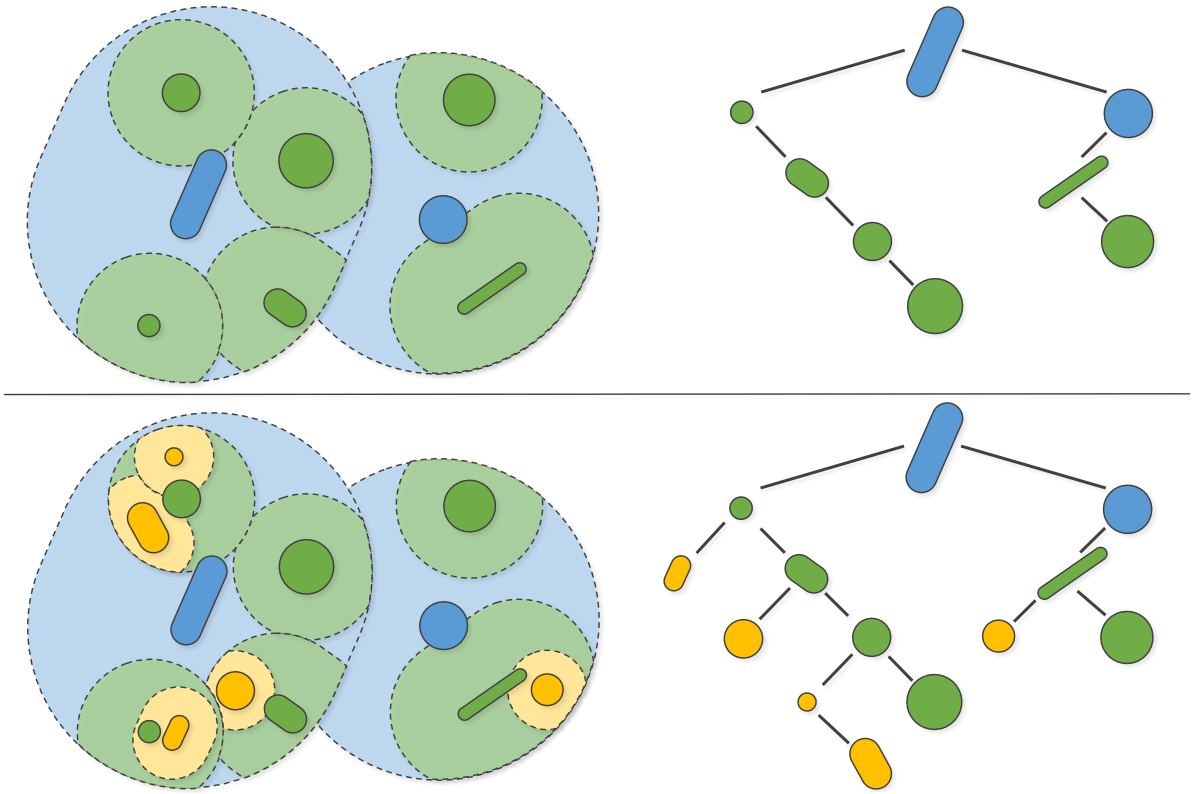

FIGURE 3.7 – Pour que cela reste lisible, les nœuds intersections ne sont pas représentés. À gauche : la partition construite sur un ensemble de cône ou cône-capsule vus depuis la lumière. À droite : l'arbre métrique ternaire correspondant. Les pivots sont représentés par des couleurs saturées et leur région proche est de la même couleur mais plus claire. Le fils gauche (resp. droit) est le fils proche (resp. lointain). Chaque fois qu'un élément passe par un fils gauche/proche, la distance de partitionnement $\delta$ associée à son élément est divisée par 2. En haut : Initialement, tous les éléments à insérer ont une valeur de $\delta$ égale à l'angle d'ouverture du cône englobant de la scène depuis la lumière. Les éléments de couleur bleue ne passent jamais par un fils gauche/proche donc leur $\delta$ n'est pas modifié. Les éléments verts passent une fois par un fils gauche/proche, donc leur $\delta$ est divisé par 2. En bas : Les éléments de couleur grise passent deux fois par un fils gauche/proche donc leur distance de partitionnement initiale est divisée par 4.

de calculer la distance médiane au pivot à cet instant. Au moment où il faut déterminer $\delta$, l'algorithme n'a qu'une connaissance très faible du domaine à subdiviser. Aussi nous utilisons l'heuristique suivante : Avant leur insertion, chaque cône ou cône-capsule est associé à une distance de partitionnement $\delta$ égale à l'angle d'ouverture du cône englobant de la scène. Cette valeur initiale est ensuite modifiée le long du parcours d'insertion de l'élément dans l'arbre métrique ternaire selon les règles suivantes :

1. $\delta$ est divisé par 2 chaque fois que l'élément est inséré dans un fils proche.
2. $\delta$ n'est pas modifié si l'élément est inséré dans un fils intersection ou un fils lointain.
3. Lorsqu'un élément atteint une feuille, il devient le pivot d'un nouveau nœud qui remplace cette feuille et sa distance angulaire de partitionnement $\delta$ conserve la valeur déterminée par l'application des deux premières règles le long du parcours d'insertion.

Plusieurs points ont motivé l'utilisation de cette heuristique. S'il n'est pas possible de formuler la moindre hypothèse sur le nombre et la distribution des éléments qui vont atteindre une feuille, nous avons néanmoins quelques indications sur la zone de l'espace qui les contiendra. Chaque nœud induit 3 régions : la région proche est délimitée par le pivot lui-même (un cône ou un cône-capsule). La région lointaine est le complément de la région proche par rapport à la région du nœud courant. Les limites de la région intersection ne seront pas connues avant la fin de la construction mais l'on sait qu'elle se situera au voisinage du bord de la région proche. Par construction, la région de l'espace contenant les éléments atteignant un nœud est égale à l'intersection de toutes les régions des nœuds rencontrés depuis la racine. Ce serait bien sûr très compliqué à calculer correctement, même sans la contrainte temps-réel. Par conséquent il n'est pas possible non plus d'utiliser dans notre heuristique l'angle solide sous tendu par la région d'un nœud. La seule information disponible porte sur les régions proches : Tous leurs éléments sont nécessairement contenus dans le cône ou le cône-capsule d'angle $\delta$ du nœud (sans que l'on puisse dire quoique ce soit sur leur distribution). Par conséquent, l'heuristique restreint la distance de partitionnement chaque fois que l'on descend dans un fils proche. Dans les deux autres cas, on ne dispose pas d'assez d'information pour modifier la distance de partitionnement de manière pertinente. Aussi $\delta$ est inchangé plutôt que de le modifier à mauvais escient. La Figure 3.7 montrent comment l'heuristique modifie progressivement la distance de partitionnement. L'avantage de cet heuristique dans un contexte temps-réel est qu'il est très simple et donc très peu coûteux. Nous avons également testé plusieurs autres stratégies mais aucune ne permettait de gagner plus que ce qu'elles coûtaient à calculer. Le pseudo-code notre construction est donné par l'algorithme 5. Nous verrons dans la section 3.4 que l'heuristique décrite ci-dessus est un bon compromis entre la vitesse de construction et la qualité de l'arbre métrique ternaire produit.

Nous venons de décrire comment construire un arbre métrique ternaire partitionnant les volumes d'ombres générés par des sphères ou capsules englobant de la géométrie, qu'il s'agisse d'un ou plusieurs triangles dans le cas de clusters. De manière analogue au précédent chapitre, il est aussi possible de construire une double hiérarchie d'arbres métriques en s'appuyant sur 2 niveaux de clusters. Le procédé que nous avons décrit avec des TOP-tree est facilement transposable. Un premier arbre métrique est calculé sur des sphères et capsules englobant des clusters de haut niveau (2048 triangles) dont les nœuds pointent chacun sur un arbre métrique construit avec une subdivision en plus petits clusters de la même géométrie. Nous avons implémenté et testé une version doublement hiérarchique d'arbres métriques, mais cela s'est avéré nettement moins concluant qu'avec des TOP-tree.
En effet, le caractère conservatif des volumes englobants (sphère ou capsule) est d'autant plus marqué que la taille des clusters est importante. Aussi, bien que le procédé permette d'améliorer les temps de construction de la structure (comme pour les TOP-tree), ce gain était perdu et même plus lors du parcours (contrairement au TOP-tree).
Pour cette raison, nous avons choisi de conserver un unique arbre métrique ternaire construit sur tous les volumes englobants d'un seul niveau de clusters.

## Parcours de l'arbre métrique ternaire

La structure est construite à chaque image. Pour chaque point visible de la caméra, l'arbre est parcouru depuis la racine afin de déterminer sa visibilité depuis la lumière. Il

**Algorithme 5** Construction de l'arbre métrique ternaire.

```
1: nœud {
2:     Vecteur3D d0,        ▷ Direction vers le 1^er sommet de la capsule / centre de la sphère
3:     Réel delta,                                          ▷ Distance de partitionnement
4:     Vecteur3D d1,                  ▷ Direction vers le 2ème sommet de la capsule
5:     Réel angle,                                          ▷ Angle du cône/capsule-cône
6:     Entier positif, négatif, intersection,
7:     Réel borne,
8:     Vecteur4D info_cluster        ▷ Données permettant d'accéder au tableau de triangles
9: }
10: ConeDeNormales {                                       ▷ Cône qui englobe les normales du cluster
11:     Vecteur3D centre,
12:     Vecteur3D normal,
13:     Réel angle
14: }
15:
16: procedure AjoutVolume(Capsule capsule, ConeDeNormales cone, nœud racine,
    Lumière L, Réel delta_initial, Vecteur4D info_cluster)
17:
18:     si ContenuDansCone(L, cone) alors
19:         Retourner()                ▷ Quitter si tous les triangles font face à la lumière
20:     fin si
21:
22:     nœud nœud
23:     nœud.delta ← delta_initial
24:     nœud.info_cluster ← info_cluster
25:     Initialisernœud(capsule)                          ▷ Calcul du cône / capsule-cône
26:
27:     nœud n ← racine
28:
29:     tant que n n'est pas une feuille faire
30:         Réel distance_min ← DistanceMin(n, capsule)
31:         Réel distance_max ← DistanceMax(n, capsule)
32:
33:         si distance_max < n.delta alors
34:             n ← n.negatif
35:             nœud.delta ← n.delta × 0.5
36:         sinon si distance_min > n.delta alors
37:             n ← n.positif
38:         sinon                                          ▷ Chevauchement
39:             MiseAJourBorneIntersection(n, capsule)
40:             n ← n.intersection
41:         fin si
42:     fin tant que
43:
44:     RemplacerFeuilleParVolumeOmbreCasule(n, capsule, L)
45:
46: fin procedure
```

faut souligner ici que nous ne traçons pas de rayons dans la structure. Vue projectivement depuis la source, nous cherchons si un point est inclus dans un disque ou une capsule afin de tester si la géométrie qu'il contient occulte le point. La base de l'algorithme de parcours est simple et consiste à appliquer les étapes suivantes à chaque nœud visité :

1. La distance angulaire $\alpha$ entre le pivot et le point à tester est calculée.
2. Si $\alpha$ est plus petit que la valeur angulaire du pivot, le point est contenu dans le pivot. Dans ce cas, le ou les triangles associés sont testés. Si l'un d'eux intersecte le segment défini par le point et la lumière, alors le point est dans l'ombre et le parcours est terminé.
3. Sinon le parcours continue dans le fils proche ou lointain en fonction de la comparaison entre $\alpha$ et la distance de partitionnement $\delta$. De plus, le fils intersection doit aussi être visité puisqu'il correspond à des éléments situés à la fois dans la région proche et dans la région lointaine.

Si le parcours se termine sans qu'aucune occultation ait été trouvée, alors le point est visible depuis la source et donc éclairé. L'algorithme 3.3.3 donne le pseudo-code du parcours.

**Algorithme 6** Parcours de l'arbre métrique ternaire pour localiser un point $p$.

```
1:  function ParcoursCPSV(nœud racine, Point p)
2:
3:      PileDenœud pile
4:      Ajouter(pile, racine)
5:
6:      tant que pile non vide faire
7:          nœud n ← Depiler(pile)
8:
9:          si n n'est pas une feuille alors
10:             Réel d ← Distance(n, p)          ▷ Distance angulaire au centre du nœud
11:                                              ▷ (centre du cône ou arête support de la capsule)
12:
13:             si d < n.angle alors
14:                 si IntersectionTriangles() alors
15:                     Retourner(0)             ▷ Le point est ombré par un triangle du cluster
16:                 fin si
17:             fin si
18:
19:             si abs(d − delta) < n.borne alors          ▷ Faut-il visiter le sous-arbre
20:                 Empiler(pile, n.intersection)          ▷ intersection ?
21:             fin si
22:
23:             Empiler(pile, d > delta ? n.positif : n.negatif)
24:         fin si
25:     fin tant que
26:
27:     Retourner(1)                             ▷ La pile est vide, le point est éclairé
28:
29: fin function
```

Plusieurs optimisations issues de l'état de l'art peuvent être transposées ou adaptées à notre cas :
**Étape 1 :** Il est inutile de visiter un sous-arbre qui ne contiendrait que de la géométrie plus éloignée de la lumière que ne l'est le point testé [68]. Aussi chaque nœud mémorise la plus petite distance (euclidienne) de la source à toute la géométrie contenue dans le sous-arbre enraciné en ce nœud. Si la distance du point à la lumière est inférieure à la distance stockée, le nœud n'a pas besoin d'être visité. En pratique nous utilisons la plus petite distance de la lumière à la sphère ou à la capsule, ce qui est légèrement plus conservatif, mais évite d'avoir à parcourir les triangles contenus, ce qui serait prohibitif. Chaque fois qu'un élément passe par un nœud, la distance minimale stockée est mise à jour si la distance de l'élément à la lumière est plus petite. Cette modification pouvant générer des accès concurrents entre threads, elle est faite au moyen d'une opération atomique.
**Étape 2 :** Lorsqu'un point se trouve dans un volume d'ombre (cône ou cône-capsule), une recherche d'intersection (point-lumière) est effectuée avec tous ses triangles. Aussi nous utilisons les deux plans calculés en même temps que les sphères et capsules (voir la section 3.3.2) pour faire d'abord un test d'intersection avec le volume englobant du cluster similaire à [53]. Cela reste conservatif mais permet néanmoins d'être plus précis et donc d'éviter potentiellement de tester inutilement les triangles un à un.
**Étape 3 :** La visite systématique du fils intersection nuit beaucoup à l'efficacité du parcours, un problème déjà souligné par Uhlmann [39,88]. Or un fils intersection correspond à une région située au voisinage de la distance de partitionnement. Un point en dehors de cette zone ne peut être occulté par la géométrie contenue dans le fils intersection. Aussi chaque nœud stocke un intervalle $[min, max]$ qui sont les distances angulaire minimales et maximales des éléments contenus dans le sous arbre intersection. Sa visite peut être évitée si $\alpha$ n'est pas contenu dans l'intervalle $[min, max]$.

## 3.4 Résultats

Comme dans le Chapitre 2, la machine utilisée pour les tests possède un CPU Intel i7700k et une carte graphique NVIDIA GTX 1080. Toutes les images sont produites à une résolution de 1920x1080. Notre algorithme est implémenté avec OpenGL 4.3 en utilisant un rendu différé (*deferred rendering*). La construction de l'arbre ternaire se fait via un Compute Shader, là encore dans un mode persistent. Le parcours de l'arbre est également implémenté dans un Fragment Shader appliqué sur le GBuffer. Toutes les données relatives aux clusters (précalculés) et aux arbres métriques ternaires sont stockées sur la carte graphique dans des Shader Storage Buffer Objects. La géométrie qui ne peut produire d'ombre sur la partie visible de la scène est toujours éliminée (*light-view frustum culling*), et les triangles (où les clusters dans la version clusterisée) faisant face à la lumière sont toujours éliminés. Cette implémentation suit le même schéma que dans le précédent chapitre. Seulement la structure de données n'est plus la même et par conséquent la construction (le Compute Shader) et le parcours (le Fragment Shader) changent aussi.

### Qualité de l'arbre métrique ternaire

Avant de les comparer avec d'autres méthodes, nous évaluons la qualité des arbres métriques ternaires calculés selon la méthode proposée dans ce chapitre (MTSV). Pour

construire ces arbres, nous avons privilégié une approche qui se prête à la programmation sur GPU et économe en calcul. Cela permet d'avoir un algorithme de construction efficace. Mais il y a tout de même un prix à cela : Nous sommes contraints de nous en remettre à une heuristique pour déterminer les distances angulaires de partitionnement $\delta$ parce que notre algorithme fait que l'on a trop peu d'informations pour les calculer autrement et plus précisément (voir la section 3.3.3). Cela pose donc question de la qualité de la structure obtenue avec notre heuristique. Contrairement à un arbre binaire dont la qualité peut se mesurer à son équilibre, c'est un peu différent dans le cas d'un arbre ternaire. Un "bon" arbre ternaire doit tendre vers un arbre binaire, c'est-à-dire qu'en chaque nœud, les sous-arbres proche et lointain doivent être équilibrés et dans le même temps, le sous-arbre intersection minimisé. Pour évaluer notre heuristique, nous avons pris le temps de construire sur CPU un arbre ternaire métrique de référence pour chaque scène en nous appuyant sur l'état de l'art. La sélection des pivots est faite selon les travaux de Yianilos [99] : Les pivots sélectionnés sont ceux qui maximisent leurs distances à tous les autres éléments de l'ensemble. Les distances angulaires de partitionnement sont calculées en prenant la valeur médiane des éléments. Uhlmann [89] montre que cette stratégie permet d'obtenir une structure plus efficace même si un arbre ternaire diffère d'un arbre binaire. Pour chaque scène, nous avons copié les arbres métriques CPU sur la carte graphique et les avons utilisés pour calculer l'ombre le long d'un parcours libre. Les scènes testées sont les mêmes que dans le précédent chapitre (voire également la figure 3.13). Mais nous avons utilisé une position fixe de la lumière ainsi qu'un environnement statique puisque l'arbre CPU ne peut être calculé à chaque image. Au final nous avons comparé les temps de parcours obtenus en utilisant les structures CPU avec ceux obtenus avec la construction GPU. Les performances de notre arbre métrique ternaire construit avec heuristique sur GPU s'avèrent au pire inférieures de 10 à 17.8% à celui de référence construit sur GPU. Dans un contexte temps réel, il est fréquent de "sacrifier" la qualité d'une structure accélératrice pour permettre une construction plus rapide. Nous sommes dans ce cas de figure. Mais les tests montrent que la construction GPU basée sur notre heuristique est un bon compromis entre vitesse et qualité.

## Comparaison avec les PSV

Nous comparons ici les arbres métriques ternaires (MTSV) appliqués sur des triangles (sans pré-calcul de clusters) avec la méthode des PSV de Gerhards. Les scènes utilisées sont présentées par la figure 3.10. Les temps mesurés pour chaque méthode sont regroupés dans le tableau 3.1.

Au niveau de la construction, les MTSV sont plus rapides (entre 7% et 32%) sur l'ensemble des scènes, excepté sur Fairy Forest où les temps sont identiques. Dans ce dernier cas, le nombre de triangles ($174k$) est encore trop faible pour réveler une différence significative. Les MTSV ont l'avantage sur les autres modèles, plus grands. Pourtant avec les MTSV, les distances angulaires entre les cônes ou/et les cônes-capsules sont plus coûteuses à évaluer que la position des triangles par rapport à des plans dans les PSV. En revanche, les MTSV sont plus économes en bande passante mémoire que les PSV : le volume d'ombre d'un triangle produit 4 nœuds composés de 2 "$vec4$" pour les PSV, contre un seul nœud de 2 "$vec4$" pour les MTSV. Le nombre de lectures/écritures des nœuds est ainsi divisé par 4 pour les MTSV. Il apparaît que cela compense amplement le surplus de calcul nécessaire.

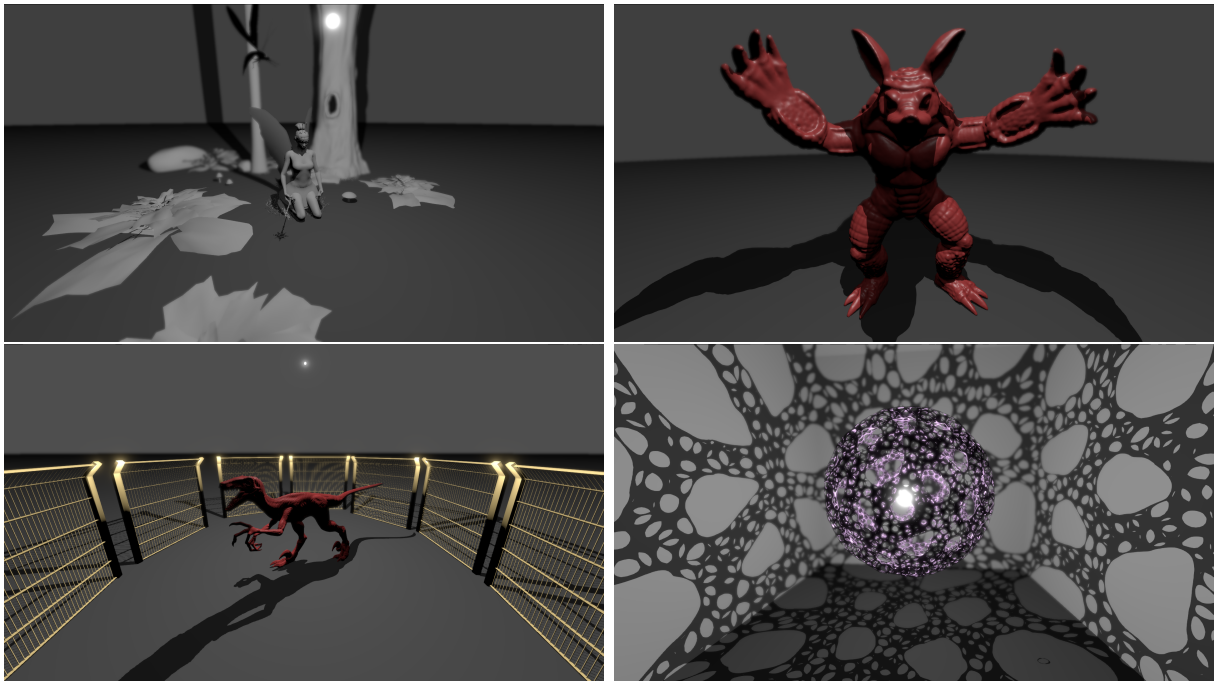

FIGURE 3.8 – Illustration des scènes utilisées pour la comparaison entre l'arbre métrique ternaire (MTSV) appliqué sur des triangles simples avec les PSV de Gerhards. En haut : À gauche, FairyForest est une scène composée de 174k triangles. À droite, Armadillo est un modèles régulier composé de 345k triangles. En bas : À gauche, un Raptor finement triangulé de 1M de triangles est entouré de barrières formées de 4 000 triangles très allongés. À droite, Voronoi est un modèle composé de 1M de triangles et projette des ombres complexes. La lumière est placée en son centre pour montrer que les méthodes supportent naturellement les sources omnidirectionnelles.

| | PSV | | | MTSV | | |
|---|---|---|---|---|---|---|
| | Construction | Parcours | Total | Construction | Parcours | Total |
| **Fairy Forest (174k triangles)** | 0.37<br>0.33 / 0.64 | 1.37<br>1.09 / 1.89 | 1.75<br>1.46 / 2.25 | 0.38<br>0.35 / 0.51 | 2.50<br>1.61 / 4.11 | 2.89<br>1.98 / 4.50 |
| **Armadillo (345k triangles)** | 0.85<br>0.81 / 1.02 | 1.10<br>0.76 / 1.46 | 1.96<br>1.62 / 2.29 | 0.66<br>0.62 / 0.84 | 1.23<br>0.82 / 1.61 | 1.89<br>1.46 / 2.35 |
| **Raptor (1M triangles)** | 2.43<br>2.34 / 2.51 | 2.72<br>2.46 / 3.15 | 5.16<br>4.85 / 5.62 | 2.26<br>2.21 / 2.29 | 3.58<br>3.39 / 4.13 | 5.84<br>5.64 / 6.37 |
| **Voronoi (1M triangles)** | 2.50<br>2.47 / 2.63 | 2.83<br>2.72 / 3.27 | 5.34<br>5.22 / 5.77 | 1.89<br>1.86 / 1.93 | 4.00<br>3.81 / 4.56 | 5.89<br>5.67 / 6.44 |

Tableau 3.1 – Ce tableau donne les temps moyens correspondant aux graphes de la figure 3.10 pour des promenades virtuelles au sein de différents modèles. Construction est le temps de construction moyen par image du TOP-tree. Parcours est le temps de parcours moyen par image du TOP-tree. Total est la somme des temps de construction et de parcours. Pour chaque valeur moyenne sont également indiquées les valeurs minimales et maximales relevées lors des mesures.


FIGURE 3.9 – Comparaison entre les MTSV (sur triangles individuels) et PSV de Gerhards. Les mesures sont faites le long de promenades virtuelles et détaillent les temps de construction, de parcours et totaux pour les deux méthodes. Les 4 modèles utilisés sont ceux présentés par la figure 3.10 et les valeurs moyennes des mesures sont celles résumées dans le tableau 3.1

Concernant le parcours, celui des MTSV est en moyenne plus lent que celui des PSV : +33% sur Fairy Forest, +11% sur Armadillo, +31% sur Raptor et +41% sur Voronoi. Ce surcoût est lié au fait que les MTSV testent d'abord le volume englobant d'un nœud et, si ce test est positif, testent ensuite l'intersection avec le triangle contenu. Ainsi pour chaque nœud, un test conservatif précède toujours un test exact. Le tout représente plus de calculs et d'accès mémoires qu'avec les PSV. Dans ce cas, les tests d'intersection sont implicites au parcours : si le point est à l'intérieur des 4 plans du volume d'ombre, alors il est assurément occulté.

Au final, le surcoût au parcours des MTSV est le plus souvent en partie compensé par sa construction plus rapide : −3% du temps total des PSV sur Armadillo, +13% sur Raptor, +10% sur Voronoi. En revanche sur Fairy Forest, PSV et MTSV ont des temps de construction identiques. Le surcoût au parcours des MTSV n'est donc pas compensé (+65% par rapport au PSV). En dehors de ce cas, probablement trop petit pour tirer partie des MTSV, ceux-ci affichent des performances très proches des PSV. Pour un premier test sur des triangles simples, les arbres métriques ternaires obtiennent ici des résultats honorables.

## Comparaison avec les CPSV

| | CPSV | | | MTSV | | |
|---|---|---|---|---|---|---|
| | Construction | Parcours | Total | Construction | Parcours | Total |
| **Birdfeeder (1.8M)** | 1.18<br>0.75 / 1.93 | 5.28<br>2.42 / 8.26 | 6.46<br>3.63 / 9.44 | 1.67<br>1.07 / 2.66 | 5.60<br>2.50 / 8.33 | 7.27<br>4.22 / 9.94 |
| **xyzrgb_dragon (7.2M)** | 3.34<br>2.78 / 3.87 | 4.73<br>1.75 / 8.07 | 8.08<br>5.05 / 11.42 | 3.43<br>2.82 / 3.98 | 4.53<br>1.20 / 8.67 | 7.97<br>4.58 / 12.01 |
| **RaptorPark (30M)** | 7.55<br>6.10 / 8.65 | 11.07<br>7.45 / 14.56 | 18.62<br>14.83 / 22.48 | 7.55<br>5.79 / 10.10 | 11.75<br>7.88 / 16.45 | 19.30<br>14.93 / 22.58 |
| **Lucy&Dragon (35.2M)** | 4.55<br>1.19 / 5.85 | 11.65<br>4.10 / 18.38 | 16.29<br>8.74 / 55.81 | 4.77<br>1.21 / 6.44 | 11.73<br>5.23 / 18.45 | 16.54<br>9.54 / 25.14 |

Tableau 3.2 – Ce tableau donne les temps moyens correspondant aux graphes de la figure 3.10 pour des promenades virtuelles au sein de différents modèles. CPSV correspond à un TOP-tree de clusters de triangles. C'est l'approche proposée dans le précédent chapitre dans sa version non hiérarchique. MTSV correspond à un arbre métrique ternaire construit avec des sphères ou capsules englobantes de clusters. Construction et Parcours sont les temps de construction moyen et de parcours moyen par image pour chaque méthode. Total est la somme des temps de construction et de parcours pour chaque méthode. Pour chaque valeur moyenne sont également indiquées les valeurs minimales et maximales relevées lors des mesures.

Nous comparons maintenant les arbres métriques ternaires (MTSV) appliqués sur des clusters de triangles avec la méthode des CPSV présentée au chapitre précédent. Pour rappel, les CPSV dans leur version double hiérarchie de TOP-trees permet d'accélérer son temps de construction et par la même de réduire le nombre de triangles stockés par cluster, allégeant ainsi le temps de parcours. Mais nous avons également expliqué dans ce chapitre qu'une double hiérarchie était moins intéressante pour les arbres métriques car les sphères/capsules englobantes de très larges clusters sont trop conservatives et entraînent un surcoût non négligeable lors du parcours. Pour permettre une comparaison pertinente entre les deux méthodes, nous appliquons ici le CPSV sur un seul niveau de clusters, comme pour les MTSV (nous ferons quand même dans la prochaine section une comparaison avec les CPSV dans leur version doublement hiérarchique). Nous utilisons toujours les mêmes scènes, celles de la figure 3.10. Les temps moyens issus de parcours libres dans les scènes sont illustrés dans la table 3.2. Le détail des mesures correspond aux graphes de la figure 3.10.

De manière générale, les performances obtenues par les deux méthodes sont très similaires. Au niveau de la construction, les performances sont identiques sur les trois plus grosses scènes : +2% pour les MTSV sur xyzrgb_dragon, +0% sur RaptorPark et +4% sur Lucy&Dragon. Sur Birdfeeder, la construction est toutefois 40% plus lente pour les MTSV. Cela s'explique par la présence d'un plus grand nombre de capsules dans la scène (12%). Le calcul des distances angulaires pour les cônes-capsules est en effet plus coûteux qu'avec des cônes. Bien que RaptorPark contienne elle aussi des capsules au niveau des barrières, elles sont en proportion plus faible pour avoir un réel impact sur le temps de construction. On pourrait s'étonner du très faible surcoût à la construction sur les autres scènes puisque

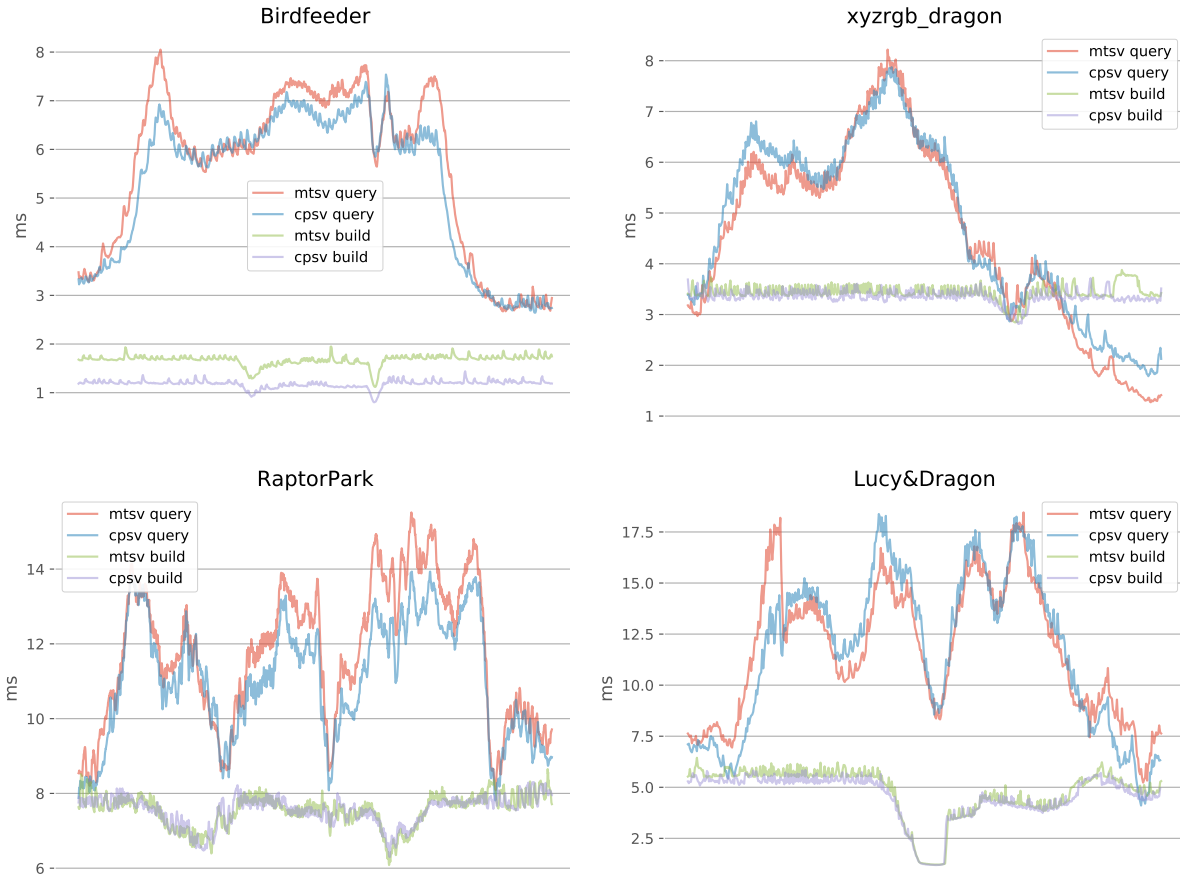


FIGURE 3.10 – Comparaison entre les MTSV (sur clusters de triangles) et les CPSV du chapitre 2 (dans leur version non hiérarchique). Les mesures sont faites le long de promenades virtuelles et détaillent les temps de construction et totaux pour les deux méthodes. Ici les temps de parcours sont volontairement omis pour ne pas nuire à la lisibilité, ces courbes étant quasiment confondues. Les 4 modèles utilisés sont ceux présentés par la figure 3.10 et les valeurs moyennes des mesures sont celles résumées dans le tableau 3.1

le calcul de distance cône/cône est plus coûteux que le simple produit scalaire opéré par les CPSV. Mais là encore, ce surcoût est contrebalancé par la consommation réduite des MTSV en bande passante mémoire. Comme expliqué dans la précédente comparaison, les MTSV consomment deux fois moins de mémoire que les CPSV.

Au niveau du parcours, les écarts sont également très faibles : +0.06% pour les MTSV sur Lucy&Dragon, et −4% sur xyzrgb_dragon. Sur Birdfeeder et RaptorPark, l'écart est de +6% pour les MTSV. En plus du surcoût lié au calcul de la distance angulaire point/cône-capsule, cet écart peut être expliqué en comparant le caractère conservatif des volumes englobants pour chaque méthode. La figure 3.11 illustre ces différences. En utilisant des boites orientées, les CPSV sont les moins conservatifs (à gauche). Lorsque seuls les capsules et les cônes sont utilisés lors du parcours (à droite), ceux-ci sont nettement plus conservatifs que les boites orientées, notamment sur les contours et pour les cônes. Toutefois, l'ajout des slabs permet une nette amélioration (au milieu). C'est particulièrement visible pour un modèle finement triangulé comme xyzrgb_dragon. Sur Tentacles ou Birdfeeder, des modèles visuellement complexes avec de nombreuses arêtes silhouettes, le caractère

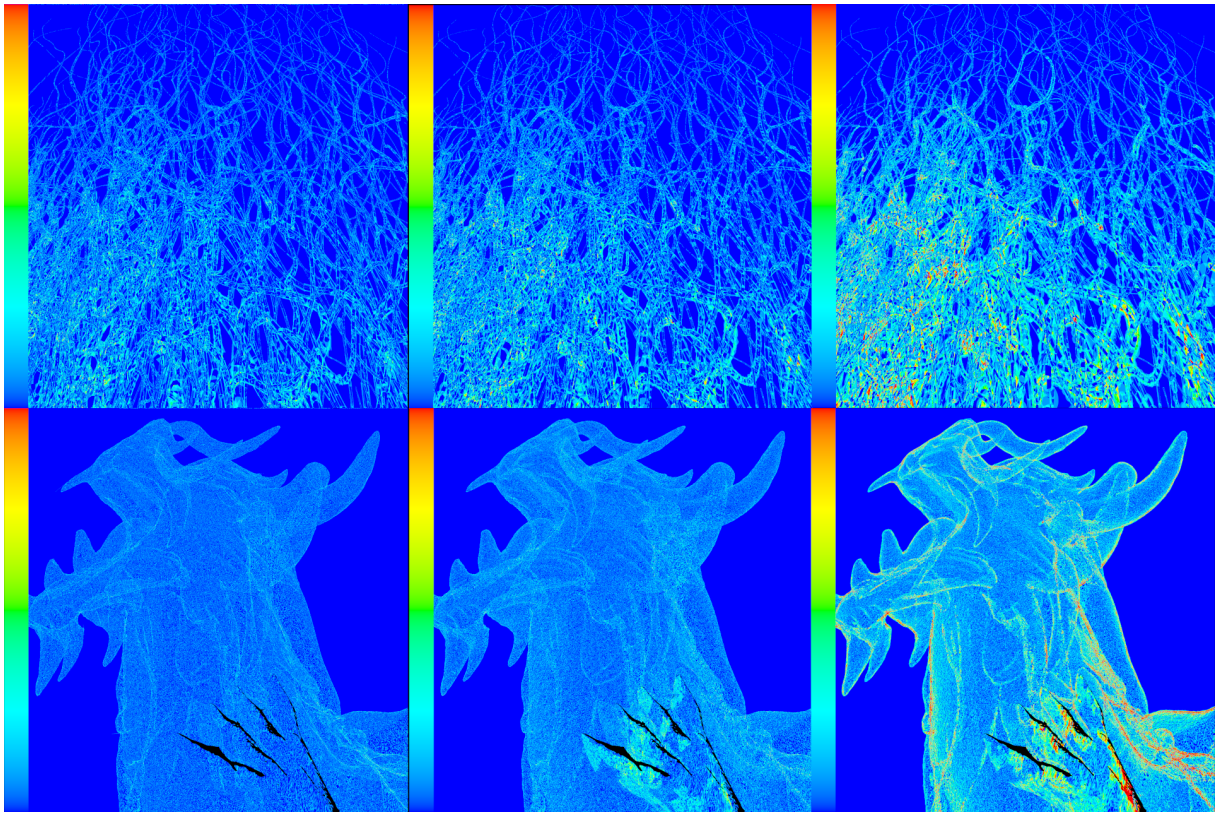

FIGURE 3.11 – Les cartes de chaleur ci-dessus illustrent le caractère conservatif des volumes englobants en fonction de la méthode utilisée. Sur deux exemples, Tentacles (même caractéristiques que Birdfeeder) en haut et xyzrgb_dragon en bas, le nombre de fois que l'on teste les triangles d'un cluster lors du parcours sans pour autant y trouver une intersection est indiqué. L'échelle utilisée va de 0 test en bleu foncé à plus de 12 tests inutiles en rouge. À gauche, la méthode des CPSV qui utilise des boites orientées englobantes de clusters. Au milieu, les MTSV sont appliqués à des sphères et capsules englobantes de clusters. L'utilisation des slabs est activée lors du parcours pour réduire le conservatisme des volumes. À droite, même chose qu'au centre mais cette fois les slabs sont ignorées.

sur-conservatif des cônes et cônes-capsules persiste par rapport aux boites orientées, ce qui explique le léger surcoût que l'on peut constater au parcours.

Concernant les MTSV, on peut souligner l'importance d'utiliser des capsules en plus de sphères comme volumes englobants de clusters. Même si les capsules tendent à augmenter un peu le temps de construction, elles sont indispensables pour traiter efficacement les clusters de forme très allongée. La figure 3.12 donne un exemple qui montre bien à quel point des sphères seules, même avec l'ajout de slabs, produisent des volumes sur-conservatifs dans ces cas. Le surcoût au parcours serait prohibitif.

## Comparaison avec OptiX et OptiX-RTX

Dans cette partie, nous comparons les MTSV ainsi que les CPSV (dans leur version la plus rapide, doublement hiérarchique) à un lancer de rayons. Pour celui-ci, nous utilisons OptiX, le moteur haute performance de lancer de rayons par NVIDIA. Cette comparai-

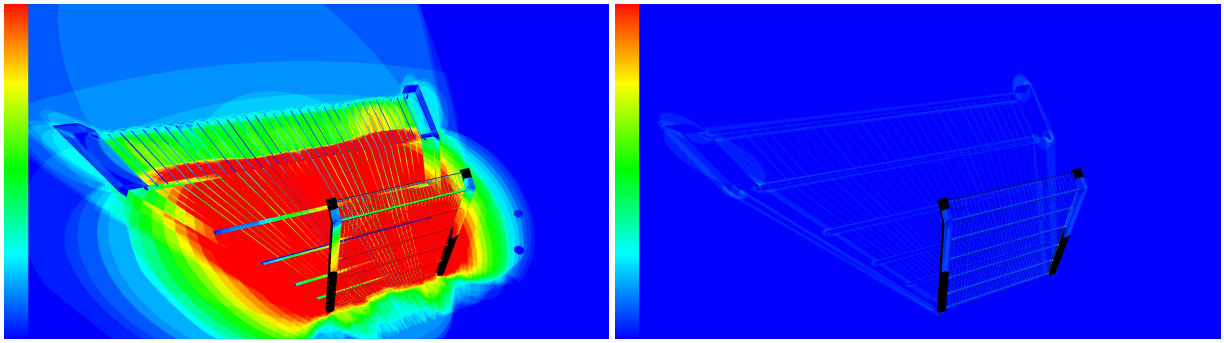

FIGURE 3.12 – Utiliser des capsules en plus des sphères comme volumes englobants des clusters permet des tests d'intersection moins conservatifs et améliore les performances. C'est particulièrement visible avec des clusters de forme étirée. Ici le modèle est une grille issue de la scène *RaptorPark*. Elle est formée de cylindres qui génèrent de nombreux clusters tout en longueur. Les deux *heat maps* illustrent le nombre de nœud dont les triangles ont été tous testés durant le parcours sans trouver d'intersection. À gauche, les volumes englobants des clusters ne sont que des sphères. À droite, des capsules ont aussi été utilisées.

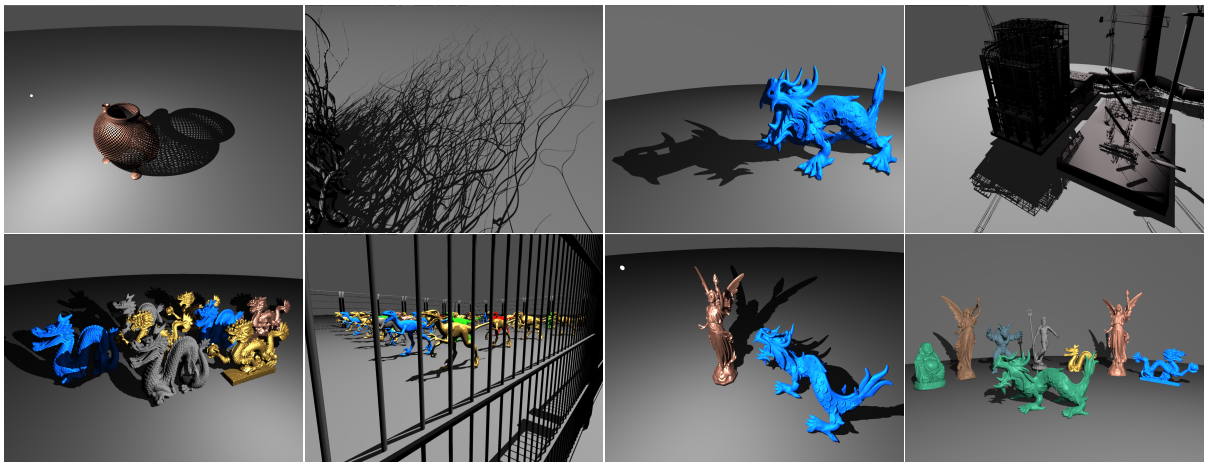

FIGURE 3.13 – En haut : *Birdfeeder* (1.8M de triangles) présente des ombres complexes avec un motif régulier. *Tentacles* (3.8M de triangles) projette des ombres complexes et irrégulières. *xyzrgb_dragon* est un modèle finement triangulé de 7.2M triangles. *Powerplant* (12.7M triangles) est un modèle très compliqué pour des algorithmes basés géométrie. L'intérieur du bâtiment est essentiellement un enchevêtrement de tubes, chacun étant formé de fins triangles allongés. En bas : *Dragons* (18.9M de triangles) comporte 8 dragons finement triangulés. *RaptorPark* (30M de triangles) regroupe 30 raptors finement triangulés derrière des grilles formées de longs triangles. *Lucy&Dragon*, avec 35M de triangles, est un modèle de très grande taille dans un contexte temps réel. *ManyModels* possède 73.8M de triangles et permet d'éprouver les limites de notre méthode.

son est justifiée pour plusieurs raisons. D'abord, la lancer de rayon est bien une méthode géométrique, comme celles que nous avons étudiées dans ce mémoire. Ensuite, au cours de cette thèse, le lancer de rayons n'a pas connu sur le fond de révolution, mais plusieurs optimisations et évolutions, notamment matériel, qui lui ont fait franchir un palier en terme

| | MTSV-1080 | CPSV-1080 | OptiX | MTSV-2080 | CPSV-2080 | OptiX-RTX |
|---|---|---|---|---|---|---|
| **Birdfeeder (1.8M)** | 7.21<br>4.11 / 10.06 | 6.54<br>2.92 / 9.93 | 9.73<br>4.35 / 15.75 | 4.39<br>2.59 / 6.84 | 3.92<br>1.84 / 6.68 | 1.39<br>0.01 / 4.16 |
| **Tentacles (3.8M)** | 12.88<br>7.32 / 17.50 | 10.16<br>4.31 / 14.89 | 11.29<br>3.18 / 20.14 | 7.65<br>4.35 / 11.89 | 5.72<br>2.4 / 9.95 | 1.50<br>0.03 / 3.63 |
| **xyrzrgb_dragon (7.2M)** | 7.55<br>3.96 / 12.0 | 6.34<br>3.60 / 9.60 | 8.18<br>2.93 / 12.12 | 4.94<br>2.78 / 8.79 | 4.24<br>2.52 / 7.05 | 1.44<br>0.01 / 3.41 |
| **Powerplant (12.7M)** | 22.33<br>7.76 / 34.75 | 12.68<br>5.98 / 23.23 | 6.12<br>2.75 / 9.09 | 13.50<br>4.86 / 22.07 | 7.98<br>3.78 / 15.97 | 0.9<br>0.00 / 4.46 |
| **Dragons (18.9M)** | 14.91<br>9.14 / 24.44 | 11.97<br>6.69 / 18.74 | 11.63<br>3.84 / 18.98 | 9.46<br>5.83 / 16.01 | 7.52<br>4.24 / 12.39 | 1.43<br>0.00 / 4.55 |
| **RaptorPark (30M)** | 18.85<br>13.84 / 23.96 | 13.36<br>10.33 / 16.21 | 9.01<br>0.57 / 17.11 | 13.53<br>9.89 / 18.37 | 9.30<br>6.97 / 12.45 | 1.37<br>0.00 / 3.78 |
| **Lucy&Dragon (35.2M)** | 16.54<br>9.54 / 25.14 | 13.00<br>6.54 / 19.16 | 11.62<br>6.19 / 21.0 | 10.53<br>5.43 / 17.97 | 8.30<br>4.43 / 12.59 | 1.59<br>0.10 / 10.84 |
| **ManyModels (73.8M)** | 22.38<br>17.17 / 27.75 | 16.49<br>12.23 / 22.89 | 10.08<br>5.31 / 17.94 | 16.61<br>12.90 / 21.79 | 11.44<br>8.49 / 16.23 | 1.43<br>0.03 / 3.93 |

Tableau 3.3 – Comparaisons sur deux cartes graphiques différentes : une GTX 1080 (les 3 premières colonnes) et une RTX 2080 (les 3 dernières colonnes) qui offre un support matériel du lancer de rayon dont bénéficie OptiX, le moteur haute performance de NVIDIA.

de performance, rendant possible son utilisation pour des applications temps-réelles (voir Chapitre 1 section 1.2.4). Ainsi, un "budget" de l'ordre de 1 à 4 rayons par pixel et par image est envisageable tant que la structure accélératrice n'a pas besoin d'être reconstruite à chaque fois. Or pour les CPSV ou MTSV, les modèles librement déformables ne peuvent être supportés, seules les transformations rigides le sont de par l'utilisation de clusters. La comparaison peut donc être faite avec un lancer de rayons puisque sous les mêmes hypothèses, la structure accélératrice n'a pas besoin d'être complètement reconstruite à chaque image. Dans ce cas, une structure locale est construite pour chaque objet et une méta-structure est construite sur les boites englobantes de ces objets. Seule la méta-structure doit être reconstruite à chaque image, ce qui représente un coût négligeable, le nombre de boîte englobante étant généralement faible comparé au nombre de triangles. Dans les mêmes conditions, le budget de 1 à 4 rayons disponible est suffisant pour permettre le calcul d'ombres dures avec des rayons d'ombre.

Nous utilisons les mêmes scènes que dans le précédent chapitre. Pour rappel, nous incluons à nouveau leur description dans la figure 3.13. Nous comparons d'abord les CPSV et MTSV avec OptiX sur une carte graphique NVIDIA GTX 1080 qui ne possède pas de support matériel pour le lancer de rayon. Nous comparons ensuite nos deux méthodes sur une carte graphique NVIDIA RTX 2080 avec OptiX dans sa version "RTX", c'est-à-dire avec une implémentation matérielle cette fois. Le tableau 3.3 regroupe l'ensemble des performances mesurées pour ces méthodes sur les 2 cartes graphiques.

### **Comparaison sur la GTX** 1080

Concernant les MTSV et les CPSV dans leur version doublement hiérarchique, l'avantage tourne aux CPSV sans surprise, puisque comme nous venons de le voir, les MTSV ont un niveau de performance équivalent aux CPSV mais dans leur version simple. Ici nous mesurons à nouveau l'apport de la double hiérarchie sur le temps de construction mais aussi sur le parcours puisque l'on peut se permettre d'abaisser le nombre de triangles par cluster pour obtenir un meilleur temps de parcours.

Par rapport à OptiX, les CPSV sont plus rapides sur les 3 première scènes : −48% sur Birdfeeder, −11% sur Tentacles et −8% sur xyzrgb_dragon. La complexité visuelle des ombres sur *Birdfeeder* et *Tentacles* nuit à la cohérence des rayons d'ombre, ce qui impacte les temps du lancer de rayons. OptiX est en revanche particulièrement efficace sur *Powerplant* (2.07 fois plus rapide en moyenne). L'intérieur du bâtiment, qui comprend l'essentiel de la géométrie du modèle, est ici totalement évitée par les rayons qui ne "rentrent" jamais à l'intérieur. Sur Dragons, les performances des deux méthodes sont identiques (+2% en moyenne pour les CPSV). Sur les plus grandes scènes, OptiX affiche une certaine stabilité (205M de rayons par seconde en moyenne) imputable au coût logarithmique de la traversée de sa structure accélératrice par un rayon. Les CPSV restent également relativement stables, notamment sur Lucy&Dragon (+11%). Sur RaptorPark et ManyModels les écarts sont respectivement de 48% et 63%. La stabilité moindre des CPSV s'explique par le fait que même si la complexité de la méthode au parcours est logarithmique, le coût du test d'occultation par un cluster augmente avec le nombre de triangles stockés dans les clusters.

**Comparaison sur la RTX** 2080

Les MTSV et CPSV bénéficient ici d'une amélioration qui n'est due qu'à l'utilisation d'une carte graphique plus performante. Selon les modèles, les deux méthodes présentent un gain de performance qui va de 28 à 40%.

Quant à OptiX, il bénéficie bien sûr de la plus grande puissance de la RTX 2080 mais surtout de ses cœurs RTX qui assurent le support matériel du lancer de rayons. Ainsi le gain pour OptiX se situe entre 85 et 88% selon les modèles, atteignant environ 1.5G de rayons par seconde en moyenne et un coût quasiment stable autour de 1.5ms par image. Il ne nous a pas été possible d'évaluer OptiX en désactivant le support RTX sur la RTX 2080 pour dissocier la part du gain imputable au support matériel. En effet, OptiX exploite automatiquement les cœurs RTX lorsqu'il les détecte. Quoi qu'il en soit, à ce niveau de performance, Optix est de loin le plus rapide sur tous les modèles.

## Coût mémoire

| | MPSV | CPSV | Optix |
|---|---|---|---|
| Birdfeeder (1.8M) | 43.23 Mb | 86.46 Mb | 171 Mb |
| Tentacles (3.8M) | 89.38 Mb | 178.76 Mb | 359 Mb |
| xyzrgb_dragon (7.2M) | 79.5 Mb | 159.0 Mb | 676 Mb |
| Powerplant (12.7M) | 104.5 Mb | 209.0 Mb | 1.21 Gb |
| Dragons (18.9M) | 121.17 Mb | 242.34 Mb | 444 Mb |
| RaptorPark (30M) | 114.98 Mb | 229.96 Mb | 93 Mb |
| Lucy&Dragon (35.2M) | 197.1 Mb | 394.2 Mb | 3.32 Gb |
| ManyModels (73.8M) | 230.5Mb | 461.4Mb | 4.31 Gb |

Tableau 3.4 – Consommation mémoire des différentes méthodes. La première colonne indique les modèles en rappelant le nombre de triangles qu'elles contiennent. Les colonnnes suivantes sont respectivement les coûts en mémoire des MPSV, CPSV et OptiX

Nous comparons à présent la mémoire nécessaire aux MPSV, CPSV et OptiX. Le tableau 3.4 regroupe ces coût pour chaque scène. Comme expliqué, les MPSV nécessitent deux fois moins de mémoire que les CPSV. Cela tient compte du tableau stockant les

volumes englobants des clusters, leur cône de normales et des slabs dans le cas des MPSV. À ceci s'ajoute le tableau de nœuds des TOP-tree ou des arbres métriques ternaires. Les deux méthodes demeurent dans des intervalles de valeurs très raisonnables, y compris pour des cartes graphiques d'entrée de gamme qui possèdent au moins 1Gb de mémoire à l'heure de la rédaction de ce mémoire. En comparaison OptiX occupe 4 à 18 fois plus de mémoire que les MPSV, et donc 2 à 9 fois plus que les CPSV. Sur ce point, les MPSV et CPSV ont l'avantage sans ambiguïté. Et clairement, il ne faut pas posséder n'importe quelle carte graphique puisque 3 modèles (Powerplant, LucyDragon et ManyModels) excèdent le Giga de données en mémoire. Toutefois il faut aussi penser qu'OptiX stocke une structure indépendante de la position de la source. La même structure peut donc être utilisée pour plusieurs sources alors qu'il faudrait en calculer une par source s'agissant des MPSV ou CPSV. Certes les sources pourraient être traitées l'une après l'autre, par conséquent les coûts par structure ne se cumuleraient pas. Sur le seul plan de la consommation mémoire, MPSV et CPSV conserveraient leur avantage. Mais les deux verraient leur temps de calculs multipliés par le nombre de sources traitées.

## Discussion

Dans ce chapitre, l'objectif premier était d'expérimenter une stratégie de partitionnement originale dans la mesure où elle n'avait jamais été utilisée pour du rendu. Sur ce point, nous avons montré que les arbres métriques ternaires, qui partitionnent les objets selon leurs distances relatives, permettent d'obtenir des performances comparables aux TOP-tree reposant sur une subdivision de l'espace objet par des plans. Ce constat était difficilement prévisible. En effet, les arbres métriques sont, par nature, plus coûteux à construire et plus difficile à équilibrer. Nous avons néanmoins montré que le coût des calculs pouvait être compensé par une consommation mémoire moindre et qu'en utilisant une heuristique il était possible d'obtenir une structure de bonne qualité. Cette expérimentation montre donc que les arbres métriques possèdent bien un potentiel exploitable pour des applications en rendu. La question est ensuite de savoir pourquoi utiliser des arbres métriques si on n'obtient pas mieux qu'avec des TOP-tree, voire un peu moins dans leur version doublement hiérarchique. Comme expliqué en introduction à ce chapitre, disposer d'une structure qui repose sur la distance aux objets (ou de manière conservative, à leur volume englobant) ouvre des perspectives différentes, y compris en dehors du calcul des ombres dures en temps réelle. Nous reviendrons sur ce point en conclusion.

A présent, du point de vue strictement applicatif, les ombres dures donc, l'utilisation de clusters fait que le lancer de rayons devient une alternative possible. Et, compte tenu des progrès importants réalisés en la matière ces deux dernières années, le lancer de rayons avec support matériel offre les meilleures performances. Sa consommation mémoire importante peut néanmoins poser un problème le cas échéant. Actuellement, il faut aussi préciser que seul NVIDIA vend des cartes graphiques avec support matériel pour le lancer de rayons. On peut toutefois être confiant sur le fait que tous les constructeurs offrent à terme à support comparable. Mais à l'heure actuelle, ce n'est pas encore une option disponible pour le plus grand nombre. Le lancer de rayons deviendra-t-il pour autant la première solution pour les ombres dures en temps réel ? Probablement pas dès demain. D'une part il demeure encore trop coûteux sur des scènes librement déformables où la structure accélératrice doit être reconstruite à chaque image. D'autre part, le "budget" de rayons offert par pixel et par image sera prioritairement utilisé pour des effets d'illumination

indirects, particulièrement compliqués ou impossibles à produire par rastérisation sur GPU. Mais à long terme, si le budget de rayons par pixel augmente significativement, y compris pour des scènes librement déformables, en consommer quelques-uns pour traiter quelques sources ponctuelles ne posera probablement pas de problème.

## 3.5 Conclusion et perspectives

L'objectif premier de ce travail était d'explorer les possibilités offertes par une nouvelle stratégie de partitionnement : Les arbres métriques. Nous n'avons pas trouvé d'utilisation de cette structure de donnée en informatique graphique jusqu'à présent. Dans la continuité de nos travaux, nous avons appliqué cette stratégie de partitionnement au rendu d'ombres dures en temps réel. L'avantage des arbres métriques est de pouvoir s'appliquer à n'importe quelles formes tant que l'on est en mesure de calculer une distance entre elles. C'est grâce à cela que nous avons pu utiliser des volumes englobants différents (sphères ou capsules) pour les triangles ou clusters de triangles tout en mélangeant dans un même arbre métrique les cônes et cônes-capsules engendrés. En adoptant une stratégie de partitionnement basée sur la distance, nous avons montré que l'on pouvait atteindre des performances comparables aux TOP-trees, basés eux sur un partitionnement par plan. Ce chapitre montre donc le potentiel de cette structure pour toutes les applications où l'information de distance aux objets est importante.

Sur le strict plan des performances, nous avons constaté l'écart qui demeure entre une approche "expérimentale" et la concrétisation matériel de près de 40 années de recherche sur le lancer de rayons. Nous pensons néanmoins que les travaux présentés dans ce chapitre, mais aussi le précédent, ouvrent des perspectives différentes, au-delà de la seule application aux calculs des ombres dures. On peut tout d'abord penser à des problématiques qui peuvent être traitées par lancer de rayons, mais nécessitent trop de rayons par pixel et par image pour demeurer temps réel. Par exemple, les ombres générées par de petits objets créent des problèmes d'aliasing que le lancer de rayons ne peut traiter que par sur-échantillonnage, ce qui dégrade fortement ses performances. Les ombres douces sont une autre application qui nécessite trop de rayons pour rester temps-réel, même combinée à un algorithme de débruitage en post-traitement. Dans ces deux cas de figure, exploiter la distance aux objets sur laquelle repose les arbres métriques constitue une piste intéressante, par exemple pour détecter les petits objets. Enfin nos travaux présentent aussi un intérêt pour d'autres problématiques en informatique graphique, problématiques pour lesquelles le lancer de rayons ne peut rien. On peut par exemple penser à de la détection de collisions, un classique de l'informatique graphique pour lequel l'information de distance entre objet fait bien entendu sens. Dans le même esprit, nous pouvons aussi penser à toutes les applications à base de particules pour lesquelles calculer ce qui se trouve dans le voisinage de chaque particule conditionne les performances globales. Pour les travaux présentés dans ce chapitre, toutes ces problématiques sont autant de perspectives.

# Bibliographie

[1] Ieee standard for floating-point arithmetic. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4610935, 2008.

[2] Timo Aila and Tomas Akenine-Möller. A Hierarchical Shadow Volume Algorithm. In Tomas Akenine-Moeller and Michael McCool, editors, *Graphics Hardware*. The Eurographics Association, 2004.

[3] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 161–166, 2004.

[4] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149. ACM, 2009.

[5] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[6] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[7] David Arthur and Sergei Vassilvitskii. K-means++ : the advantages of careful seeding. In *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[8] Harlen Costa Batagelo and Ilaim Costa Junior. Real-time shadow generation using bsp trees and stencil buffers. In *In Proc. SIBGRAPI 99*, pages 93–102, 1999.

[9] Carsten Benthin and Ingo Wald. Efficient ray traced soft shadows using multi-frusta tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 135–144. ACM, 2009.

[10] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A scalable approach to interactive global illumination. In *Computer Graphics Forum*, volume 22, pages 621–630. Wiley Online Library, 2003.

[11] Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R Mark. Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(9) :1438–1448, 2011.

[12] Nikolaus Binder and Alexander Keller. Efficient stackless hierarchy traversal on gpus with backtracking in constant time. In *Proceedings of High Performance Graphics*, HPG '16, pages 41–50, Goslar Germany, Germany, 2016. Eurographics Association.

[13] W Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, 13(9) :527–536, 1970.

[14] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007*, pages 177–184. ACM, 2007.

[15] Stefan Brabec, Thomas Annen, and Hans peter Seidel. Practical shadow mapping. *Journal of Graphics Tools*, 7 :9–18, 2000.

[16] Stefan Brabec and Hans-Peter Seidel. Shadow Volumes on Programmable Graphics Hardware. *Computer Graphics Forum*, 2003.

[17] John Carmack. Z-fail shadow volumes. 2000.

[18] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, 1974. AAI7504786.

[19] Matthaeus Chajdas. Geometryfx 1.2 – cluster culling. http://gpuopen.com/geometryfx-1-2-cluster-culling/, 2016.

[20] Eric Chan and Frédo Durand. An efficient hybrid shadow rendering algorithm, 2004.

[21] Norman Chin and Steven Feiner. Near real-time shadow generation using bsp trees. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 99–106. ACM, 1989.

[22] Yiorgos Chrysanthou and Mel Slater. Shadow volume bsp trees for computation of shadows in dynamic scenes. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 45–50. ACM, 1995.

[23] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 905–914, New York, NY, USA, 2004. ACM.

[24] R. L. Cook, T. Porter, and L. Carpenter. Tutorial : Computer graphics ; image synthesis. chapter Distributed Ray Tracing, pages 139–147. Computer Science Press, Inc., New York, NY, USA, 1988.

[25] Franklin C. Crow. *The Aliasing Problem in Computer-synthesized Shaded Images.* PhD thesis, 1976. AAI7612386.

[26] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11(2) :242–248, July 1977.

[27] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Computer Graphics Forum*, volume 27, pages 1225–1233. Wiley Online Library, 2008.

[28] François Deves, Frédéric Mora, Lilian Aveneau, and Djamchid Ghazanfarpour. Scalable real-time shadows using clustering and metric trees. In Wenzel Jakob and Toshiya Hachisuka, editors, *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*. The Eurographics Association, 2018.

[29] Leonardo R Domingues and Helio Pedrini. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics*, pages 13–20. ACM, 2015.

[30] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH, 2006.

[31] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. *Real-time shadows.* CRC Press, 2011.

[32] Wolfgang Engel. Cascaded shadow maps. *ShaderX5 : Advanced Rendering Techniques*, 1, 2007.

[33] Manfred Ernst and Gunther Greiner. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 35–40. IEEE, 2008.

[34] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P Greenberg. Adaptive shadow maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 387–390. ACM, 2001.

[35] Tom Forsyth. Linear-speed vertex cache optimisation. http://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html, 2006.

[36] Kirill Garanzha. The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. In *Proceedings of the Eurographics Symposium on Rendering*, pages 1199–1206, 2009.

[37] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 59–64. ACM, 2011.

[38] Bernd Gärtner. Fast and robust smallest enclosing balls. In *Proceedings of the 7th Annual European Symposium on Algorithms*, ESA '99, pages 325–338, 1999.

[39] Julien Gerhards, Frédéric Mora, Lilian Aveneau, and Djamchid Ghazanfarpour. Partitioned Shadow Volumes. *Computer Graphics Forum, Proceedings of Eurographics 2015*, 2015.

[40] Markus Giegl and Michael Wimmer. Fitted virtual shadow maps. In *Proceedings of Graphics Interface 2007*, pages 159–168. ACM, 2007.

[41] Markus Giegl and Michael Wimmer. Queried virtual shadow maps. In *Symposium on Interactive 3 D Graphics : Proceedings of the 2007 symposium on Interactive 3 D graphics and games*, volume 30, pages 65–72, 2007.

[42] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Obbtree : A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM, 1996.

[43] Naga K Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 501–510. ACM, 2003.

[44] John C Hart. Sphere tracing : A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10) :527–545, 1996.

[45] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[46] Tim Heidmann. Real shadows real-time. 1991.

[47] Stephen Hill, Stephen McAuley, Alejandro Conty, Michał Drobot, Eric Heitz, Christophe Hery, Christopher Kulla, Jon Lanz, Junyi Ling, Nathan Walster, et al. Physically based shading in theory and practice. In *ACM SIGGRAPH 2017 Courses*, page 7. ACM, 2017.

[48] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276, 1999.

[49] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer : Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4) :1462–1482, October 2005.

[50] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[51] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[52] Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. Revisiting the vertex cache : Understanding and optimizing vertex processing on the modern gpu. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(2), August 2018.

[53] Linus Käellberg and Thomas Larsson. Ray tracing using hierarchies of slab cut balls. In *Eurographics 2010 - Short Papers*, 2010.

[54] Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, 1996.

[55] Eric Larsen, Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Fast proximity queries with swept sphere volumes. Technical report, 1999.

[56] Andrew Lauritzen, Marco Salvi, and Aaron Lefohn. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 97–102, New York, NY, USA, 2011. ACM.

[57] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384, 2009.

[58] Aaron E Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D Owens. Dynamic adaptive shadow maps on graphics hardware. In *SIGGRAPH Sketches*, page 13, 2005.

[59] Aaron E Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D Owens. Glift : Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics (TOG)*, 25(1) :60–99, 2006.

[60] Aaron E Lefohn, Shubhabrata Sengupta, and John D Owens. Resolution-matched shadow maps. *ACM Transactions on Graphics (TOG)*, 26(4) :20, 2007.

[61] Daqi Lin, Konstantin Shkurko, Ian Mallett, and Cem Yuksel. Dual-split trees. In *Symposium on Interactive 3D Graphics and Games (I3D 2019)*, New York, NY, USA, 2019. ACM Press. to appear.

[62] Gang Lin and TP-Y Yu. An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4) :640–648, 2006.

[63] D. Brandon Lloyd, David Tuft, Sung-eui Yoon, and Dinesh Manocha. Warping and Partitioning for Low Error Shadow Maps. In Tomas Akenine-Moeller and Wolfgang Heidrich, editors, *Symposium on Rendering*. The Eurographics Association, 2006.

[64] D. Brandon Lloyd, Jeremy Wendt, Naga K. Govindaraju, and Dinesh Manocha. Cc shadow volumes. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 197–205, 2004.

[65] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28 :129–137, 1982.

[66] Michael D McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics (TOG)*, 19(1) :1–26, 2000.

[67] Daniel Meister and Jiří Bittner. Parallel bvh construction using k-means clustering. *Vis. Comput.*, 32(6-8) :977–987, June 2016.

[68] Frederic Mora, Julien Gerhards, Lilian Aveneau, and Djamchid Ghazanfarpour. Deep Partitioned Shadow Volumes Using Stackless and Hybrid Traversals. In *Eurographics Symposium on Rendering*, 2016.

[69] Bruce Naylor, John Amanatides, and William Thibault. Merging bsp trees yields polyhedral set operations. *SIGGRAPH Comput. Graph.*, 24(4) :115–124, September 1990.

[70] NVIDIA. Shadowworks.

[71] Jacopo Pantaleoni and David Luebke. Hlbvh : hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.

[72] Arsène Pérard-Gayot, Javor Kalojanov, and Philipp Slusallek. Gpu ray tracing using irregular grids. *Comput. Graph. Forum*, 36(2) :477–486, May 2017.

[73] Stefan Popov, Johannes Gunther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of sah kd-trees. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94. IEEE, 2006.

[74] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424, 2007.

[75] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE micro*, 20(4) :47–57, 2000.

[76] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1176–1185. ACM, 2005.

[77] Pedro V Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (TOG)*, 26(3) :89, 2007.

[78] Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. Compressed multiresolution hierarchies for high-quality precomputed shadows. *Comput. Graph. Forum*, 35(2) :331–340, May 2016.

[79] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample based visibility for soft shadows using alias-free shadow maps. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1285–1292, 2008.

[80] Erik Sintorn, Viktor Kämpe, Ola Olsson, and Ulf Assarsson. Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 111–118. ACM, 2014.

[81] Erik Sintorn, Ola Olsson, and Ulf Assarsson. An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 153 :1–153 :10. ACM, 2011.

[82] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Trans. Graph.*, 21(3) :557–562, July 2002.

[83] Martin Stich. Introduction to nvidia rtx and directx ray tracing. http://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing.

[84] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*, 2009.

[85] Martin Stich, Carsten Wächter, and Alexander Keller. Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. *GPU Gems*, 3 :239–256, 2007.

[86] Jon Story. Advanced geometrically correct shadows for modern game engines, 03 2016.

[87] Nuno Subtil. Introduction to real-time ray tracing with vulkan. http://devblogs.nvidia.com/vulkan-raytracing.

[88] Jeffrey K. Uhlmann. Adaptive partitioning strategies for ternary tree structures. *Pattern Recognition Letters*, 12(9) :537 – 541, 1991.

[89] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4) :175 – 179, 1991.

[90] Marek Vinkler, Vlastimil Havran, and Jiří Bittner. Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. In *Computer Graphics Forum*, volume 35, pages 68–79, 2016.

[91] Ingo Wald and Carsten Benthin. Openrt - a flexible and scalable rendering engine for interactive 3d graphics. Technical report, 2002.

[92] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer graphics forum*, volume 20, pages 153–165. Wiley Online Library, 2001.

[93] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree : a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4) :143, 2014.

[94] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6) :343–349, June 1980.

[95] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3) :270–274, August 1978.

[96] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light Space Perspective Shadow Maps. In Alexander Keller and Henrik Wann Jensen, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 2004.

[97] Jianhua Wu Leif Kobbelt. Structure recovery via hybrid variational surface approximation. In *Computer Graphics Forum*, volume 24, pages 277–284. Wiley Online Library, 2005.

[98] Chris Wyman, Rama Hoetzlein, and Aaron Lefohn. Frustum-traced raster shadows : Revisiting irregular z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, pages 15–23. ACM, 2015.

[99] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, 1993.

[100] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics*, page 4. ACM, 2017.

[101] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search : The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[102] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*, VRCIA '06, pages 311–318, New York, NY, USA, 2006. ACM.

# Conclusion

Le calcul d'ombre dures en temps-réel est un problème de longue date en informatique graphique. Pour chacun des points visibles depuis l'observateur, il s'agit de trouver si la lumière est occultée ou non par un élément géométrique qui compose la scène. Il s'agit d'un problème de visibilité. Bien qu'il soit simple d'un point de vue conceptuel, il est plus délicat à résoudre avec des contraintes temps-réelles.

Pour satisfaire le niveau de performance requis, la technique du *Shadow Mapping* est depuis longtemps un choix privilégié dans l'industrie, et cela perdure encore de nos jours. C'est le choix de la vitesse grâce au support matériel complet de la rastérisation sur les cartes graphiques. Il faut cependant s'accommoder des nombreux artefacts visuels qu'il est difficile voire impossible d'éviter ou de corriger totalement. Pour cette raison, les méthodes géométrique basées sur les volumes d'ombre ont continuées de susciter l'intérêt pour la qualité visuelle des ombres qu'elles permettent d'obtenir en toute circonstance. C'est dans ce contexte que nous avons situé nos travaux.

Nous avons tout d'abord traité d'un problème inhérent aux méthodes géométriques telles que les volumes d'ombres : leur dépendance intrinsèque à la complexité géométrique. Sous cette angle, cela les condamnes à des performances inversement proportionnelles à l'augmentation du nombre de triangles dans les modèles. Aussi nous avons proposé une approche permettant de contenir cet écueil en utilisant des clusters de triangles afin de se ramener à un nombre d'éléments à traiter bien inférieur au nombre de triangles. L'évaluation de cette approche s'est avérée concluante, maintenant de bonnes performances y compris sur des modèles dépassant les 70 millions de triangles. En comparaison, les méthodes de référence auxquelles nous nous sommes comparés voient leur efficacité se dégrader avec l'augmentation de taille des modèles, jusqu'à ne plus respecter la contrainte temps réel. Toutefois, il existe une limite à notre approche : Le calcul de clusters est un problème trop complexe pour envisager de le faire en temps-réel sur des modèles de très grande taille. Aussi nous sommes obligés de les précalculer, ce qui interdit de fait d'utiliser des scènes librement déformables. Nous avons cependant souligné que cela ne constituait pas nécessairement une contrainte forte quant aux domaines d'applications. Les scènes librement déformables de très grande taille ne sont pas les plus communes, et notre approche reste valable pour des scènes dynamiques comprenant des transformations géométriques rigides.

Dans un second temps, nous avons exploré une autre stratégie de partitionnement pour organiser les volumes d'ombres dans un arbre métrique ternaire, une approche peu commune en rendu. Nous avons montré comment adapter à des éléments géométriques les arbres ternaires pour aboutir à une méthode dont les performances se sont avérées plutôt convaincantes dans le cadre d'une première expérimentation. On peut noter que cette approche originale dépasse le seul cadre applicatif du calcul des ombres. Cela permet de montrer le potentiel des arbres métriques comme stratégie de partitionnement pour des éléments géométriques. Et de ce point de vue, d'autres applications seraient envisageables. La distance aux objets sur laquelle repose par construction les arbres métriques est en effet une information intéressante à exploiter pour de nombreuses problématiques. Principalement, on pense naturellement à la détection de collisions ou bien aux simulations à base de particules. Sans sortir du cadre du rendu, les arbres métriques pourraient être utilisés pour détecter les petits objets et mieux traiter les problèmes d'aliassage qu'ils engendrent.

Enfin on peut également penser aux méthodes d'occultation ambiante volumétrique qui nécessitent de connaître la géométrie présente au voisinage d'un point. Là encore, c'est une problématique pour laquelle les arbres métriques sont applicables.

**Volumes d'ombre en rendu temps réel : complexité géométrique et stratégie de partitionnement**

**LABORATOIRE XLIM - UMR CNRS n° 7252**
123, avenue Albert Thomas - 87060 LIMOGES