Troisième partie

Modèle hiérarchique à base d'images pour le rendu interactif de forêts avec illumination et ombrages [MNP01]

CHAPITRE 5

Modèle à base d'images avec illumination et ombrage

Le rendu des nombreux arbres d'un paysage est particulièrement coûteux. Le nombre important de détails visibles, la complexité de l'illumination (micro-ombrage d'une branche sur une autre, illumination du ciel, etc.) et de la situation en terme de visibilité (les bloqueurs sont très nombreux et de petites tailles) font que bien souvent le modèle simpliste de *billboard* est utilisé pour des applications temps réel (*cf.* chapitre 2 section 2.3.1). Malheureusement ce modèle ne permet pas un réalisme géométrique et photométrique poussé.

Les deux aspects entrant en compte dans le calcul de la couleur d'un objet sont l'illumination ("réaction" de l'objet à la lumière), et l'ombrage (quantité de lumière arrivant sur l'objet sans être bloquée) qui correspond à un problème de visibilité. La nouvelle représentation que nous introduisons dans ce chapitre est inspirée par l'idée des *billboards* à laquelle nous associons une *fonction bidirectionnelle de texture* (*cf.* chapitre 2 section 1.1.7) afin de gérer l'illumination et d'améliorer l'effet 3D. Pour les problèmes de visibilité liés à l'ombrage, nous introduisons une structure de données inspirée des cartes d'horizon (*cf.* chapitre 1 section 2.2.3) et des cellules de visibilité (*cf.* chapitre 1 section 2.2.1). Notre but est d'obtenir une impression visuelle plus réaliste, avec une gestion de l'illumination et de l'ombrage dans des temps de calcul permettant l'interactivité, y compris pour la prise en compte du changement d'éclairage.

Ce chapitre est découpé en quatre parties : en section 1, nous traiterons de notre représentation à base de *billboards* et de *BTF* , nous détaillerons à la section 2 notre structure de visibilité pour l'ombrage, puis nous proposerons en 3 une technique pour traiter les ombres au sol et nous finirons en 4 par les résultats.

1 Billboard et fonction bidirectionnelle de texture

La nouvelle représentation que nous introduisons dans cette section est dérivée de l'association d'une fonction bidirectionnelle de texture (*Bidirectional Texture Function*, ou *BTF*) avec une représentation basée sur l'image inspirée des *billboards*. Nous voulons améliorer le concept de *billboard* en remédiant à l'absence de relief et de parallaxe qui le caractérise et en introduisant la prise en compte de l'illumination.

Nous présentons en 1.1 la technique de construction d'une *BTF*, puis nous verrons en 1.2 comment utiliser ces données au moment du rendu.

1.1 Construction d'une BTF

1.1.1 Principe

Le principe d'une fonction bidirectionnelle de texture (*BTF*) est d'associer une image de l'objet à chaque couple direction de vue plus direction de lumière, tout comme une *BRDF* associe une couleur à un couple de directions de vue et de lumière (*cf.* chapitre 2 section 1.1.6).

Pour le calcul, nous considérons n directions de vue, et pour chacune de ces directions de vue nous considérons m directions de lumière (*cf.* figure 5.1). À chacun de ces couples (directions de vue, directions de lumière), nous associons une image de l'objet, composée de la couleur (*RGB*) et de l'opacité (*alpha*).

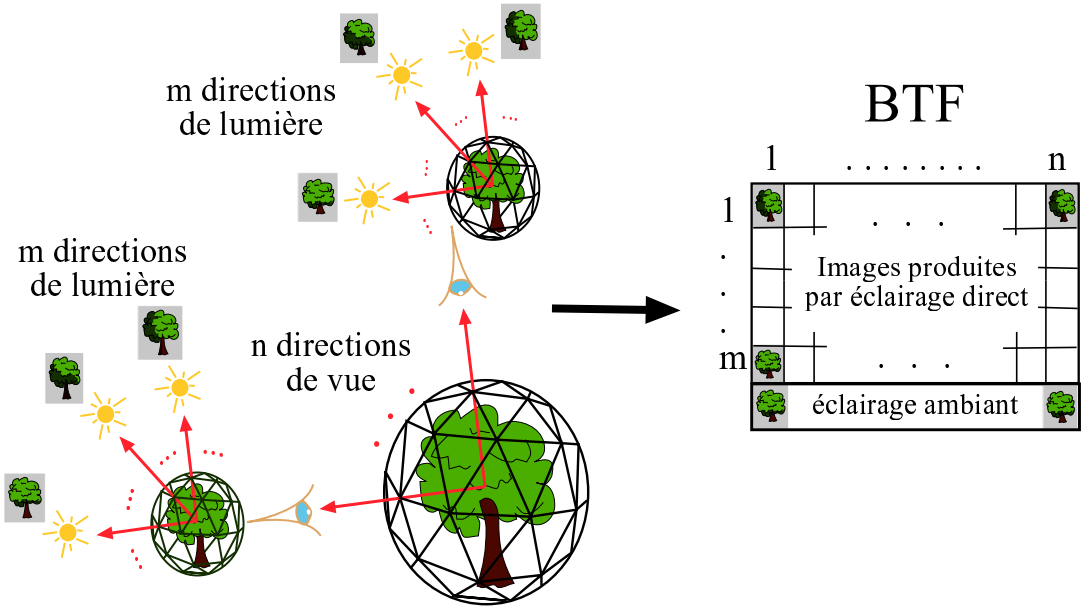

FIG. 5.1 – Pour construire une fonction bidirectionnelle de texture (*BTF*), on considère n directions de vue, autour de l'objet. Pour chacune de ces directions de vue on considère m directions de lumière. On calcule ainsi $n \times m$ images contenant l'objet éclairé par une lumière directe (*i.e.* éclairage diffuse et spéculaire) et n images contenant l'objet éclairé par une lumière ambiante.

Si nous nous limitons à des objets dont les reflets ne varient pas brusquement, un petit nombre de directions de lumière suffit. Nous destinons ces images à remplacer la géométrie 3D d'un objet apparaissant petit à l'écran, ce qui nous autorise à stocker des images de petites résolutions (*e.g.* 32×32 ou 64×64), et ainsi limiter l'occupation mémoire (*cf.* figure 5.2). Il y a un compromis

à choisir entre qualité et coût mémoire : plus la résolution de nos images est grande (donnant des objets précis) et moins on dispose de mémoire pour calculer des directions de vue différentes.

Afin de disposer d'un véritable espace vectoriel permettant de représenter des conditions d'éclairage variées, il faut séparer les fonctions de bases. Nous choisissons de distinguer une base pour l'éclairage direct et une pour l'éclairage ambiant, conformément à l'usage en synthèse d'images. On spécifie l'éclairage par un ensemble de sources directes et une source ambiante, l'intensité et la couleur de chacune étant librement choisies. Ceci nous conduit à calculer deux séries d'images : une série de n images contenant l'objet éclairé par une lumière ambiante (*i.e.* non directionnelle) et une série de $n \times m$ images contenant l'objet éclairé par une lumière directe (*i.e.* diffuse et spéculaire) (*cf.* figure 5.1). La séparation de l'ambiant augmente légèrement la place mémoire occupée (quelques Ko), mais elle permet d'effectuer des rendus où l'ambiant joue un rôle important. Par exemple si la majorité de l'éclairage provient du ciel car le soleil est opacifié par les nuages ou est déjà couché (*e.g.* effet de ciel orangé)), ou encore des rendus de scènes comportant plusieurs sources de lumière. À noter que cette séparation de l'ambiant et de la lumière directe est optionnelle dans notre implémentation : l'utilisateur choisit si ce léger surcoût mémoire est nécessaire ou non à son application.

FIG. 5.2 – Un exemple de *BTF* : un pin avec 18 directions de vue et 6 directions de lumière. Les images sont de taille 64×64 ce qui donne une occupation mémoire de $(64 \times 64 \times 4) \times 18 \times 6 = 1.7\text{Mo}$, compatible avec la mémoire texture d'une station (32 Mo ou 64 Mo sur une *GeForce*).

Grâce à la *BTF*, l'illumination et l'auto-ombrage sont encodés directement dans l'image : aucun autre calcul est nécessaire lors de la phase de rendu, ce qui permet d'obtenir l'interactivité comme nous le verrons plus loin. La construction d'une *BTF* en pré-calcul est effectuée par un moteur de rendu donnant les meilleurs résultats visuels possibles en terme de réalisme (*i.e.* illumination et ombrage), ce qui augmente d'autant l'aspect réaliste du rendu final.

1.1.2 Discrétisation de la sphère

Nous choisissons les directions de vue et de lumière autour de la sphère de manière régulière. Pour cela nous utilisons un schéma de discrétisation récursif basé sur une subdivision des triangles équilatéraux. Nous partons d'une sphère discrétisée régulièrement par six points formant huit triangles équilatéraux (*cf.* figure 5.3). Ensuite, nous divisons chacun de ces triangles en quatre nouveaux triangles en prenant le milieu de chaque segment, que nous reprojetons sur la sphère. Nous obtenons la discrétisation suivante en nombre de sommets et donc en nombre de directions de vue ou de lumière : 6, 18, 66, 257, etc. Ce schéma à l'avantage d'être simple et régulier.

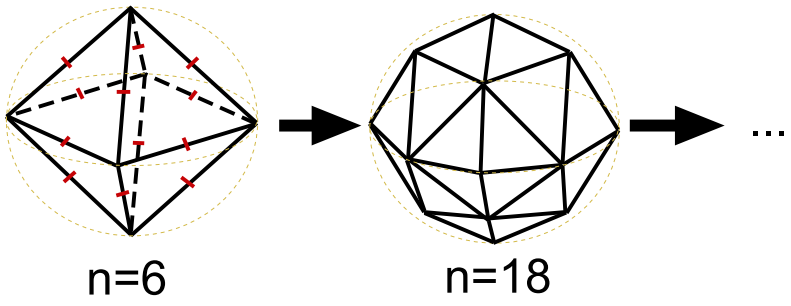

FIG. 5.3 – La discrétisation de la sphère que nous utilisons pour choisir les directions de vue et de lumière lors de la construction d'une *BTF*. La sphère initiale comporte six points formant huit triangles équilatéraux. La subdivision consiste à former, à partir de chaque triangle, quatre nouveaux triangles équilatéraux en prenant le milieu de chaque segment, que nous reprojetons sur la sphère.

Pour la construction des directions de vue et de lumière de nos *BTF* il est possible d'utiliser n'importe quels autres schémas de discrétisation. En particulier nous pourrions regrouper l'échantillonnage des directions de vue et des directions de lumière et ainsi effectuer l'échantillonnage dans un espace 4D (en représentant les directions par leurs coordonnées polaires, nous aurions 2D pour les directions de vue et 2D pour les directions de lumière). Cette technique est classiquement utilisée pour la construction de *BRDF*.

1.2 Rendu

Nous venons de décrire la structure et la construction d'une *BTF*, nous présentons maintenant la méthode de rendu de notre représentation, ainsi que les améliorations apportées. Notre objectif étant le temps réel pour le rendu d'un grand nombre d'objets, l'idée est d'étendre la technique des *billboards* en se servant au maximum des informations contenues dans une *BTF*.

Nous décrirons en 1.2.1 l'idée de base de notre rendu, suivie en 1.2.2 par les deux schémas de composition d'images possibles pour former un *billboard*, et nous finirons en 1.2.3 par proposer une technique simple augmentant les performances de rendu.

1.2.1 Principe

Lors du rendu, la direction de l'œil et la direction de la source de lumière nous permettent d'extraire les images les plus proches stockées dans la *BTF*. Notre représentation est fondée sur la combinaison de ces images pré-calculées dans le but de former un *billboard*.

Le premier problème est de trouver les directions échantillonnées stockées dans la *BTF* les plus proches des directions de vue et de lumière. Nous tabulons dans des cartes pré-calculées (*cf.* figure 5.5) le numéro des trois échantillons les plus proches d'une direction donnée. Dans ces tables, nous pré-calculons aussi les pondérations associées à ces trois directions. Nous extrayons les trois directions de vue les plus proches stockées dans la *BTF* et, de même, les trois directions de lumière[1]. Ce qui nous donne un total de 12 images (*cf.* figure 5.4), chaque direction de vue donne quatre images : une image représentant l'objet éclairé à la lumière ambiante et trois images représentant l'objet éclairé par la lumière directe depuis les trois directions de lumière. Si le traitement séparé de l'ambiant n'est pas activé nous obtenons neuf images.

[1]Dans le cas où, dans la *BTF*, le nombre de directions de vue est différent du nombre de directions de lumière nous pré-calculons deux tables.

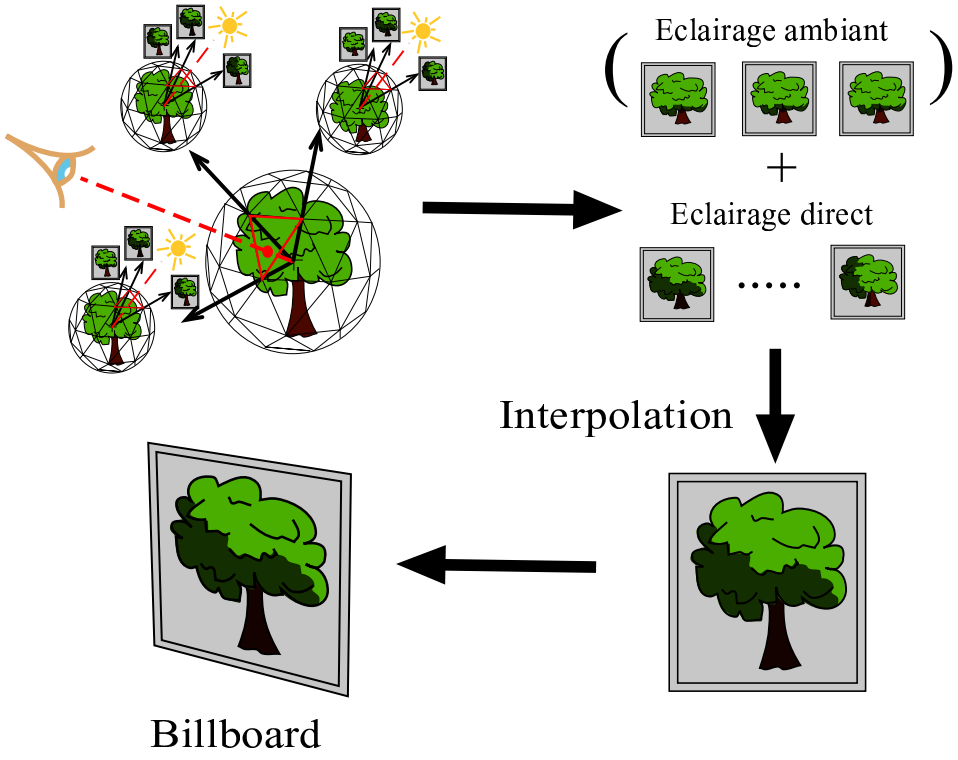

FIG. 5.4 – A partir de la direction de vue et de la direction de lumière on extrait les images de la *BTF*. Ces images sont interpolées de manière pondérée afin d'obtenir une représentation proche du *billboard* avec *OpenGL*.

En nous plaçant dans un espace 4D pour l'échantillonnage des directions de vue et de lumière, comme nous l'avons suggéré à la section précédente, nous n'aurions plus neuf images pour les directions de vue et de lumière mais cinq images, plus les trois images pour l'éclairage ambiant (si il est activée), soit un total de huit images.

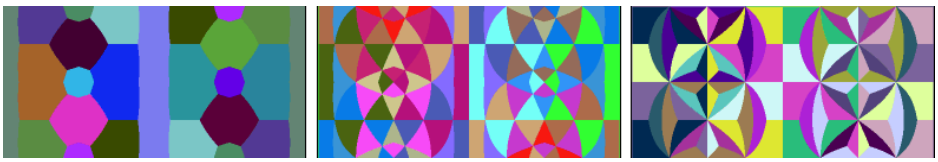

FIG. 5.5 – Nous utilisons une table pré-calculée pour déterminer les trois directions les plus proches. Les deux axes correspondent aux coordonnées polaires de la direction et les trois composantes *RGB* aux trois numéros des directions les plus proches (le coefficient de pondération n'est pas visible sur ces images).

1.2.2 Deux techniques pour les billboards

Nous disposons d'images de l'objet et de leur pondération. Il nous reste à composer ces images le plus adéquatement possible. L'idée générale est d'interpoler ces images pour les utiliser comme *billboard*, mais deux questions se posent alors :

– Faut-il plaquer ces images dans le plan perpendiculaire à la direction de l'œil, ou faut-il utiliser leur orientation d'origine, *i.e.* les trois plans perpendiculaires aux directions de vue (*cf.* figure 5.6) ? On peut rapprocher ces deux modes de rendu aux deux techniques

de *billboard* existantes : les *billboards* plats, toujours orientés vers l'œil, et les *billboards* croisés formés de trois plans perpendiculaires (*cf.* chapitre 2 section 2.3.1).

– Comment moyenner de manière pondérée plusieurs images en utilisant le matériel graphique ? Ceci n'est pas une simple composition des transparences (*alpha blending*) comme on pourrait le penser au premier abord. Il faut donc trouver un moyen d'utiliser astucieusement le matériel pour résoudre ce problème (*cf.* annexe A), d'autant plus qu'au moment de ces travaux le *multi-texturing* n'existait pas et le matériel *SGI* utilisé ne permettait pas de traiter plusieurs images simultanément.

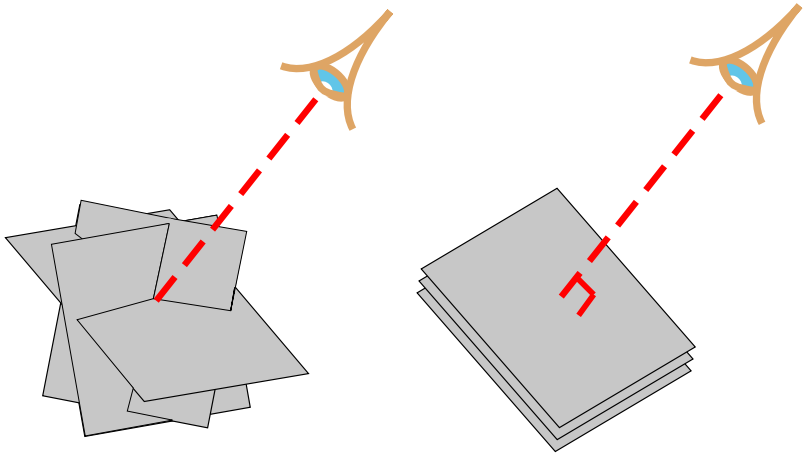

FIG. 5.6 – Pour nos *billboards* deux possibilités de rendu sont possibles. *À gauche* : les images sont perpendiculaires à la direction de vue encodée dans la *BTF*. *À droite* : les images sont toutes dans le plan perpendiculaire à la direction de l'œil.

Nous avons testé ces deux techniques, *i.e.* avec un plan unique ou avec trois plans. Notre critère étant l'apparence visuelle, nos tests ont montré que le choix de l'une ou l'autre technique dépend du nombre de directions de vue pré-calculées dans la *BTF*. Lorsque ce nombre de vues est très grand (257 ou plus), la technique utilisant trois plans tend vers la technique utilisant le plan perpendiculaire à l'œil. En effet, plus on a de directions encodées dans la *BTF*, plus ces directions sont proches les unes des autres, et donc, plus les plans perpendiculaires correspondants sont proches. Dans ce cas là, les deux techniques engendrent des résultats similaires. Lorsque le nombre de directions de vue encodées dans la *BTF* est de six, il vaut mieux utiliser trois plans perpendiculaires et ainsi obtenir un *billboard* croisé qui offre un meilleur effet de perspective et de volume qu'un *billboard* plat (*cf.* chapitre 2 figure 2.10). Pour des nombres de directions de vues intermédiaires (18 ou 66), il vaut mieux utiliser un plan unique dont la continuité est plus douce.

Que nous utilisions l'une ou l'autre de ces techniques, nous devons effectuer, lors du rendu, une combinaison pondérée des images : $\sum_i \mathtt{Poids}_i \times \mathtt{Image}_i$. Pour obtenir un temps de rendu proche du temps réel il est impossible d'effectuer ces calculs par logiciel et la composition des transparences effectuée par le matériel graphique ne donne pas les résultats attendus. Nous proposons en annexe A une solution utilisant le matériel graphique, adaptée de la composition des transparences. Pour une compréhension du coût de rendu nous dirons succinctement que cette technique fonctionne en séparant le traitement de la composante *alpha*, des composantes de couleur *RGB*. Pour la mise en place de la composante *alpha* nous effectuons trois passes (une pour chaque direction de vue) sans mettre à jour la couleur, puis nous réalisons les passes pour les composantes *RGB* sans mettre à jour la composante *alpha*. Cette technique ajoute donc trois passes de rendu, ce qui entraîne un total de 15 passes par objet (12 passes quand la séparation de l'ambiant

n'est pas activée). Pour le cas d'un échantillonnage dans un espace 4D la mise à jour de l'*alpha* coûterait cinq passes, ce qui donnerait un rendu en 13 passes (10 passes quand la séparation de l'ambiant n'est pas activée).

Aux vues des performances actuelles des cartes graphiques, 15 passes par objet permet un rendu temps réel de bon nombre d'objets. Si ce nombre de passes peut paraître élevé, il faut se rappeler que ces textures servent à représenter un arbre ou une branche qui sont des milliers de fois plus complexes en nombre de polygones. De plus, ces passes peuvent être factorisées en utilisant les capacités de multi-textures dont disposent les nouvelles générations de cartes graphiques. Avec le *multi-texturing*, les passes préliminaires de mise à jour de la composante *alpha* ne sont plus utiles : avec quatre textures simultanées[2], on peut factoriser les images de l'objet sous les différentes directions de lumière en une passe, et obtenir ainsi un rendu en trois passes au total (soit des performances cinq fois supérieures à ce que nous obtenons actuellement sans cela).

1.2.3 Optimisation

Le rendu d'un *billboard* demande plusieurs passes de rendu par objet (entre 8 et 15). Nous avons vu que, lors du rendu, chaque image est pondérée. L'optimisation proposée ici consiste à ne tenir compte que des images dont le poids est supérieur à un seuil paramétrable par l'utilisateur. Si une image n'est pas affichée, les poids des autres images sont normalisés. Nous verrons dans la partie résultat que cette simple technique permet de diminuer de moitié en moyenne le nombre de passes, tout en gardant un aspect visuel acceptable. En revanche, un seuil trop élevé introduit un effet de clignotement (*poping*) immédiatement visible par un observateur averti.

2 Cube de visibilité (VCM)

Nous venons de voir que l'utilisation d'une *BTF* associée à une adaptation des *billboards* permet d'afficher un objet, par exemple un arbre, en temps réel avec une illumination et un auto-ombrage correct, même avec une source de lumière en mouvement. À ce stade, il nous manque encore l'information concernant l'ombrage d'un objet sur un autre.

Le matériel graphique n'est pas prévu pour calculer les ombres, et une scène de forêt est trop complexe pour pouvoir évaluer celles-ci à la volée par lancer de rayons. Une carte d'ombrage globale (*cf.* chapitre 1 section 2.2.2) demanderait une résolution trop importante, et ne pourrait gérer les objets partiellement transparents qu'autorise notre représentation. La solution que nous proposons ici passe par un pré-calcul de l'information de visibilité entre les objets et la lumière.

Bien que nous pré-calculions l'information, nous aimerions pouvoir changer la direction de la lumière interactivement. Cependant, il parait difficile de traiter cet ombrage au niveau de chaque pixel du *billboard*, puisque celui-ci est traité comme un polygone par la carte graphique, laquelle ne permet pas de calcul très élaboré au niveau du pixel[3].

Nous introduirons à la section 2.1 le principe des cubes de visibilité, puis nous expliquerons à la section 2.2 comment utiliser un tel cube de visibilité avec les *billboards* que nous venons de voir, et nous finirons à la section 4 par un court aperçu des possibilités et des problèmes de ces techniques.

[2] Ce qui est supporté par une *GeForce3*.

[3] La fonctionnalité permettant un calcul par pixel des couleurs n'est apparue que très récemment parmi les cartes graphiques, et ne permet pas encore d'effectuer n'importe quel traitement. Sous *OpenGL* 1.1, que l'on trouve notamment sur les *SGI* utilisées pendant ma thèse, l'éclairage est évalué aux sommets des faces, et la couleur résultante est interpolée à l'intérieur.

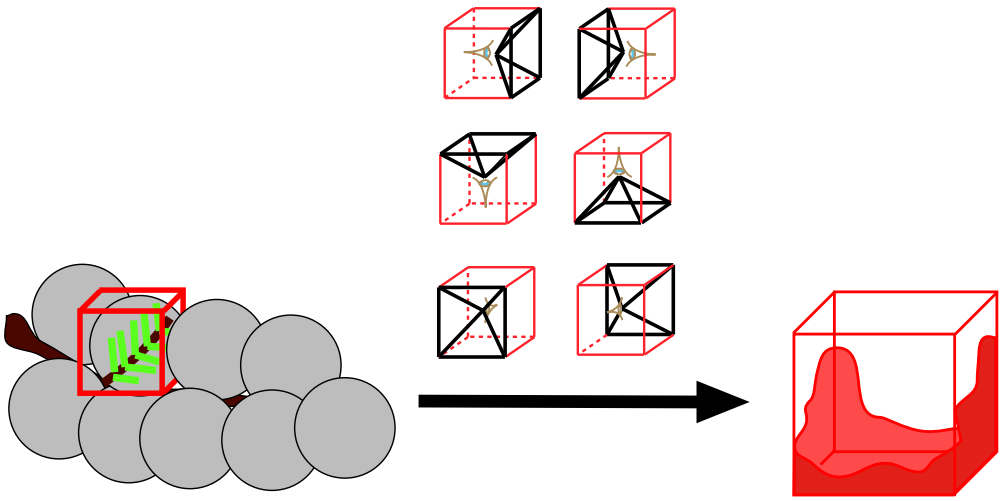

FIG. 5.7 – Pour construire un cube de visibilité (*VCM*), nous effectuons le rendu de toute la scène sur ses six faces en ne considérant que l'information d'opacité (*alpha*). Ces six images contiennent alors la visibilité depuis ce point dans toutes les directions. Ce pré-calcul permettra de savoir instantanément si la lumière d'une source d'une direction donnée peut atteindre le point central.

2.1 Principe et construction

Le but des cubes de visibilité (*Visibility Cube Maps, ou VCM*) est d'offrir une structure de données pré-calculée permettant de traiter l'ombrage d'une scène et d'augmenter ainsi le réalisme de nos *billboards*. Le principe est de stocker la visibilité d'un point donné dans toutes les directions en utilisant six cartes de visibilité formant un cube. L'idée est inspirée des cartes d'horizons[4] (*cf.* chapitre 1 section 2.2.3) et des cellules de visibilité (*cf.* chapitre 1 section 2.2.1).

Pour construire un *VCM*, nous effectuons le rendu de la scène complète sur chacune des six faces du cube, la caméra étant au centre du cube, et en ne considérant que les valeurs d'opacité (*cf.* figure 5.7). Ce rendu est effectué par le matériel graphique avec l'algorithme de rendu vu à la section précédente puis, par logiciel, on extrait la composante alpha de l'image. Un *VCM* peut être vu comme le dual d'une carte d'ombre (*cf.* chapitre 1 section 2.2.2), qui encode la visibilité de toute la scène depuis un point unique : la source de lumière. Pour chaque *VCM* on calcule la valeur d'opacité moyenne, permettant de pondérer l'illumination ambiante (*i.e.* éclairage dû au ciel). Elle est obtenue en moyennant toutes les valeurs d'opacité des faces.

Grâce aux six cartes de visibilité, nous disposons pour chaque direction de l'accessibilité de la lumière vers ce point. L'accessibilité (équivalente à une transparence) de la lumière dans une direction considérée vaut 1 moins la valeur d'occlusion. Chaque pixel des six cartes de visibilité contient une valeur d'occlusion comprise entre 0 et 1 : 0 quand la source de lumière est complètement occultée dans cette direction, 1 quand elle est complètement visible.

[4] Excepté que Max [Max88] utilise des données 1D alors que nous avons besoin de donnée 2D.

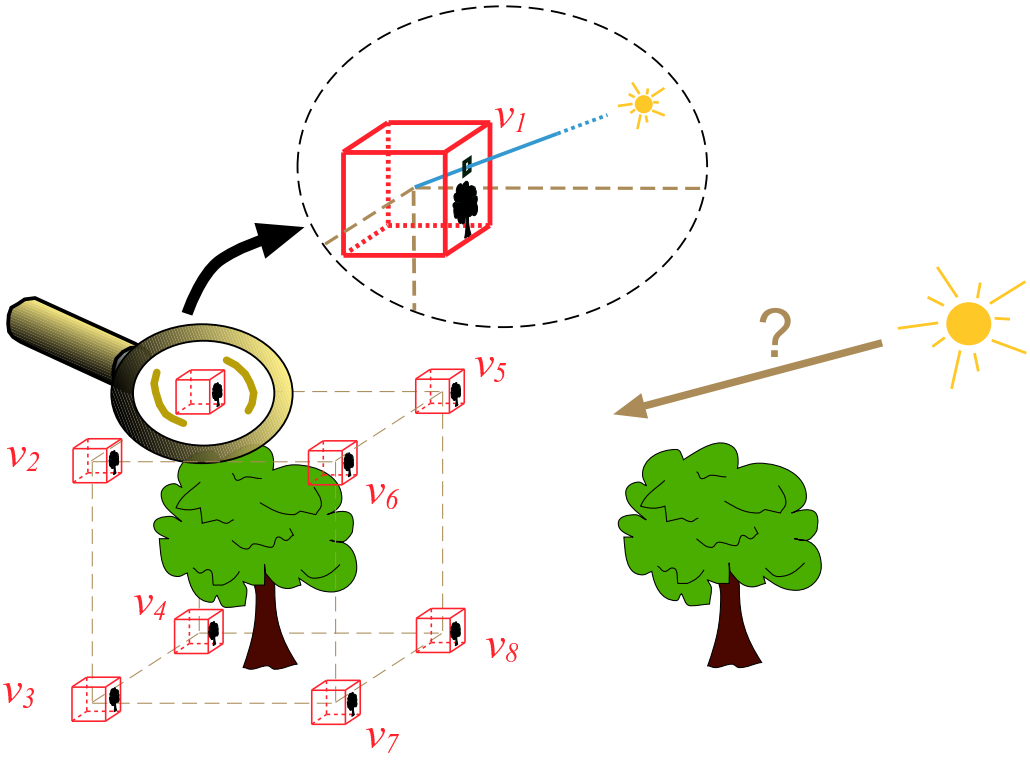

FIG. 5.8 – On place n cubes de visibilité autour de l'objet. Dans notre implémentation, nous en plaçons en général huit aux sommets de la boîte englobante.

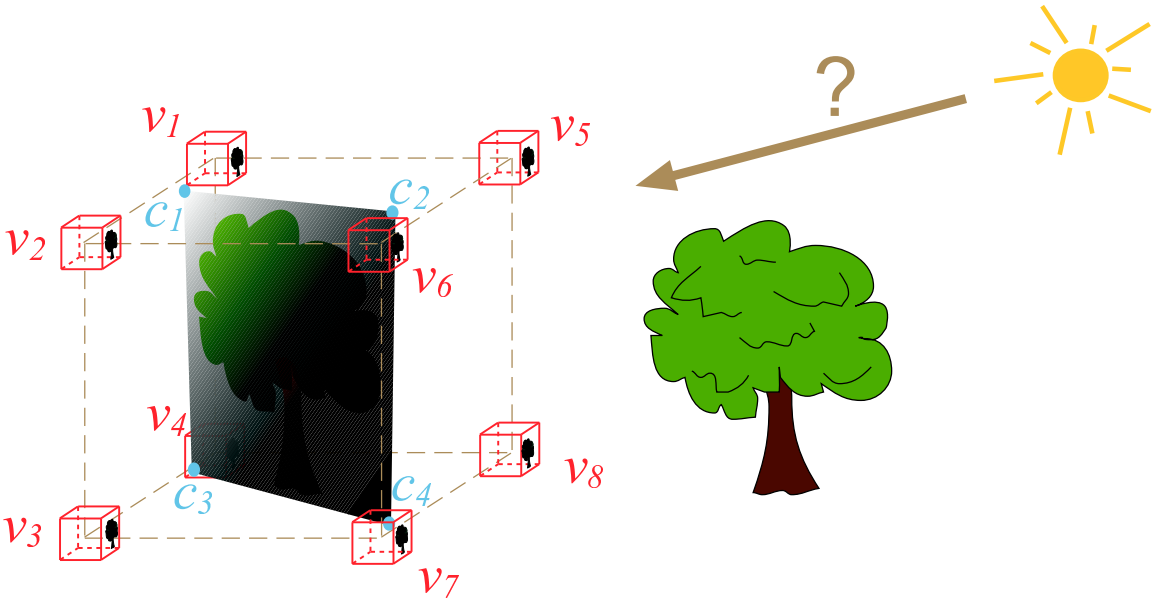

FIG. 5.9 – Soit une direction de lumière. Avec les huit valeurs de visibilité données par les cubes de visibilité qui se trouvent autour de l'objet, nous calculons par interpolation les quatre coefficients d'assombrissement aux sommets du *billboard*. Ces quatre coefficients sont utilisés par *OpenGL* comme couleurs aux sommets du polygone, le matériel graphique se chargeant de l'interpolation lors du remplissage du polygone. Cette couleur multipliera en chaque pixel le dessin du *billboard*.

2.2 Cube de visibilité et rendu de billboard

Un *VCM* fournit l'information de visibilité d'un point dans toutes les directions. Cependant, celle-ci varie d'un point à un autre. Pour représenter la visibilité d'un objet complet, nous échantillonnons le "champs de *VCM*" dans le volume de l'objet[5]. Au moment du rendu, la visibilité en un point précis du volume de l'objet est évaluée par interpolation entre les valeurs données par les *VCM* aux noeuds voisins. Comme l'objet est représenté par un *billboard*, nous calculons une valeur d'ombrage aux quatre sommets de celui-ci (par logiciel), puis nous confions au matériel graphique le soin de les interpoler lors de la *rasterisation.* Le matériel multiplie, en chaque pixel, ces valeurs d'ombres avec la couleur de la texture pour obtenir (assombrir) la couleur finale du polygone (*cf.* figure 5.9). Par la suite, lorsque nous ferons référence à une valeur de visibilité associée à une direction, il s'agira de la valeur calculée par interpolation entre les valeurs de visibilité des *VCM* aux nœuds comme expliqué dans cette section.

3 Ombres au sol

Pour obtenir l'information de visibilité d'un terrain, Max [Max88] et Stewart [Ste98] échantillonnent la surface et y place des cartes d'horizons (*cf.* chapitre 1 section 2.2.3). Avec le même esprit, nous pourrions placer des *VCM* sur la surface échantillonnée, mais cette solution serait très coûteuse et peu précise, d'autant plus qu'au sol, on s'attend à bien discerner le contour des ombres portées. En effet, pour échantillonner suffisamment finement une surface et ainsi obtenir des ombres détaillées, il faudrait des millions de *VCM* , avec ce que cela implique en temps de précalcul et en mémoire. La solution que nous proposons est une adaptation des techniques classiques de cartes d'ombre (*cf.* chapitre 1 section 2.2.2).

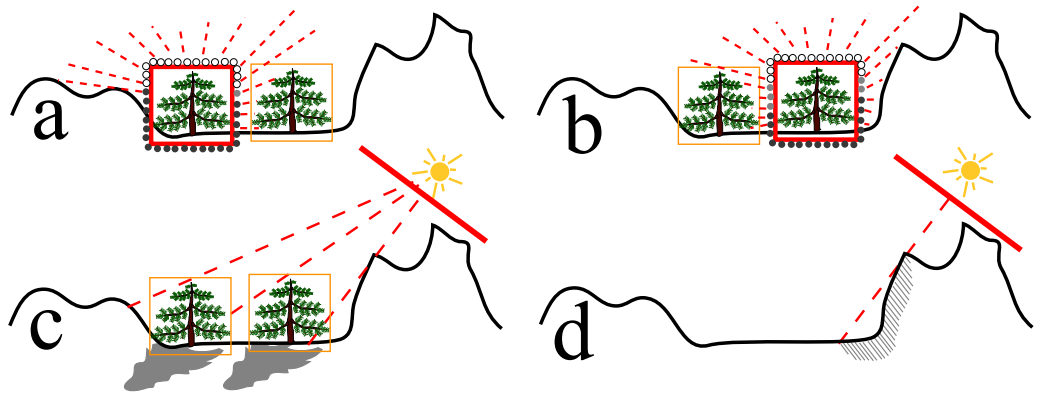

FIG. 5.10 – Les quatre cas de figure d'interaction entre les arbres et le terrain. *a,b* : Les cubes de visibilité de chaque arbre prennent en compte la présence de la montagne et des autres arbres (l'auto-ombrage étant inclus dans les *BTF* et par les *VCM* pour les niveaux inférieurs de la hiérarchie). *c* : L'*alpha shadow map* nous donne les ombres douces des arbres sur le sol. *d* : La *Z shadow map* (ou *depth map*) nous donne l'auto-ombrage de la montagne.

Nous utilisons deux sortes de *shadow map* combinées, dont les avantages et les inconvénients sont complémentaires (*cf.* chapitre 1 section 2.2.2) : une *alpha shadow map* qui ne gère pas l'auto-ombrage, est utilisée pour les objets semi-transparents se projetant sur le terrain (*e.g.* les arbres),

[5]Dans notre implémentation nous plaçons généralement huit cubes de visibilité aux huit sommets de la boîte englobante de l'objet (*cf.* figure 5.8). Si ce choix offre un bon compromis entre coût mémoire et qualité, il n'est pas une contrainte du modèle.

et une *Z shadow map* qui ne gère pas les objets semi-transparents, pour l'auto-ombrage du terrain (*cf.* figure 5.10 en bas). Ces *shadow maps* sont construites en effectuant un rendu (simplifié) de la scène depuis le point de vue de la source de lumière[6].

Pour l'*alpha shadow map*, seuls les arbres sont dessinés, en ne considérant que les valeurs de transparence (*alpha*) des textures, utilisées comme niveaux de gris (*cf.* figure 5.11). Le terrain est dessiné "en invisible" *i.e.* en blanc : il ne doit pas produire d'ombre, mais il doit quand même cacher les arbres qui sont de l'autre côté. Pour la *Z shadow map* qui prend en compte les auto-ombrages du terrain (*e.g.* l'ombre d'une montagne sur une vallée), une carte de profondeur est utilisée (*depth map*[7]). À noter qu'à la place d'une *Z shadow map* n'importe quelle technique de calcul des ombres de terrains peut être utilisée (*cf.* chapitre 1 section 2.2.3).

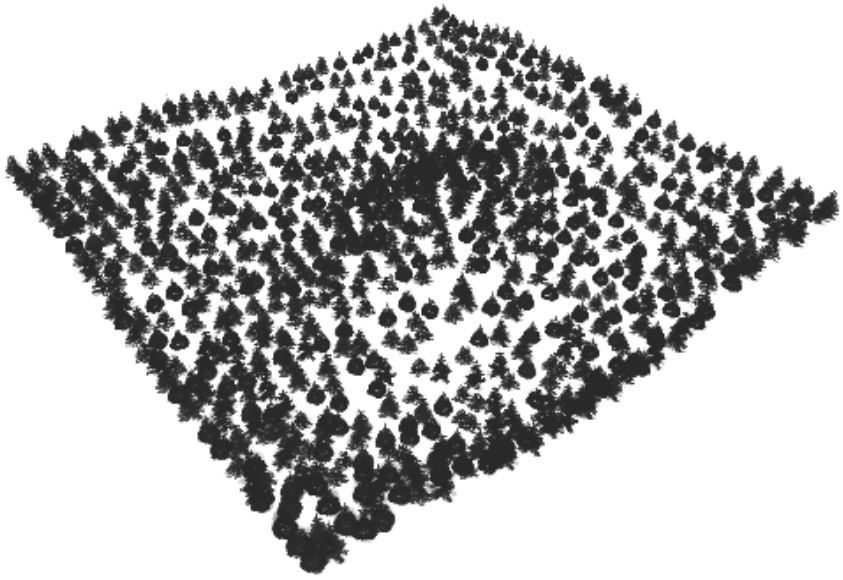

FIG. 5.11 – Un exemple d'*alpha shadow map* prise depuis la source de lumière où seule l'opacité des arbres est représentée. Le sol est tracé en blanc pour cacher les objets qui sont éventuellement derrière lui et pour ne pas générer d'ombre. Cette texture est ensuite utilisée comme texture d'ombre plaquée sur le terrain, et est recalculée à chaque changement de position de la lumière.

4 Résultats et conclusions

Cette extension des *billboards* par l'utilisation de *BTF* et des cubes de visibilité nous permet d'afficher en temps interactif (5 à 20 images par seconde sur une Onyx^2 Infinite Reality, en utilisant un algorithme de suppression des objets hors du champ de la caméra) des scènes complexes comportant des centaines d'arbres (*cf.* figure 5.13) avec illumination, auto-ombrage (inclus dans les images) et ombrage (grâce aux cubes de visibilité). Cette structure de données autorise des objets semi-transparents et, pour cette occasion, l'antialiassage (*i.e.* de petits détails dans un pixel

[6]Malheureusement la construction dynamique de ces deux *shadow map* prend un temps non négligeable, même en utilisant le matériel. Quand la direction de la lumière change nous devons mettre à jour ces cartes et le taux de rafraîchissement s'en trouve diminué.

[7]L'ombrage par *depth map* doit être supporté par le matériel graphique, ce qui est le cas de l'*Infinite Reality* de *SGI* que nous avons utilisée, mais aussi de la carte *Geforce 3* de *Nvidia*.

correspondent à une fraction d'opacité).

Cependant, la faible résolution des images utilisées (en général 64×64) et l'interpolation d'images introduisent un effet de flou qui diminue le réalisme en masquant les détails pour les objets au premier plan (*cf.* figure 5.13 à droite). Un arbre suffisamment éloigné de la caméra ne souffre pas de ces artefacts. La représentation de l'objet doit avoir une taille à l'écran inférieure à la résolution des images de la *BTF* : si un pixel de la texture (de la *BTF*) recouvre plusieurs pixels de l'image on observera un manque de finesse dans les détails. Dans le cas contraire, l'image paraîtra détaillée (*cf.* chapitre 3 section 3.2). Comment adapter cette représentation pour traiter un arbre proche de la caméra ? Augmenter la résolution des images n'est pas raisonnable, car le coût en mémoire deviendrait vite rédhibitoire. Cette représentation fonctionne bien pour des objets apparaissant petits à l'écran.

Notre solution consiste à décomposer un arbre en sous-parties : une branche peut être représentée par un de nos *billboard* . Un arbre entier et réaliste, même pour des vues proches, peut être affiché par un ensemble de branches de ce type. Nous introduisons ainsi des niveaux de détails, mais ceci revient à augmenter l'ordre de grandeur du nombre de primitives présentes. Calculer et stocker un cube de visibilité différent pour chaque instance d'objet de la scène devient alors excessivement coûteux, étant donné le grand nombre de branches composant une forêt. Nous avons donc mis au point une structure de données hiérarchique basée sur l'instanciation qui fait l'objet du chapitre suivant.

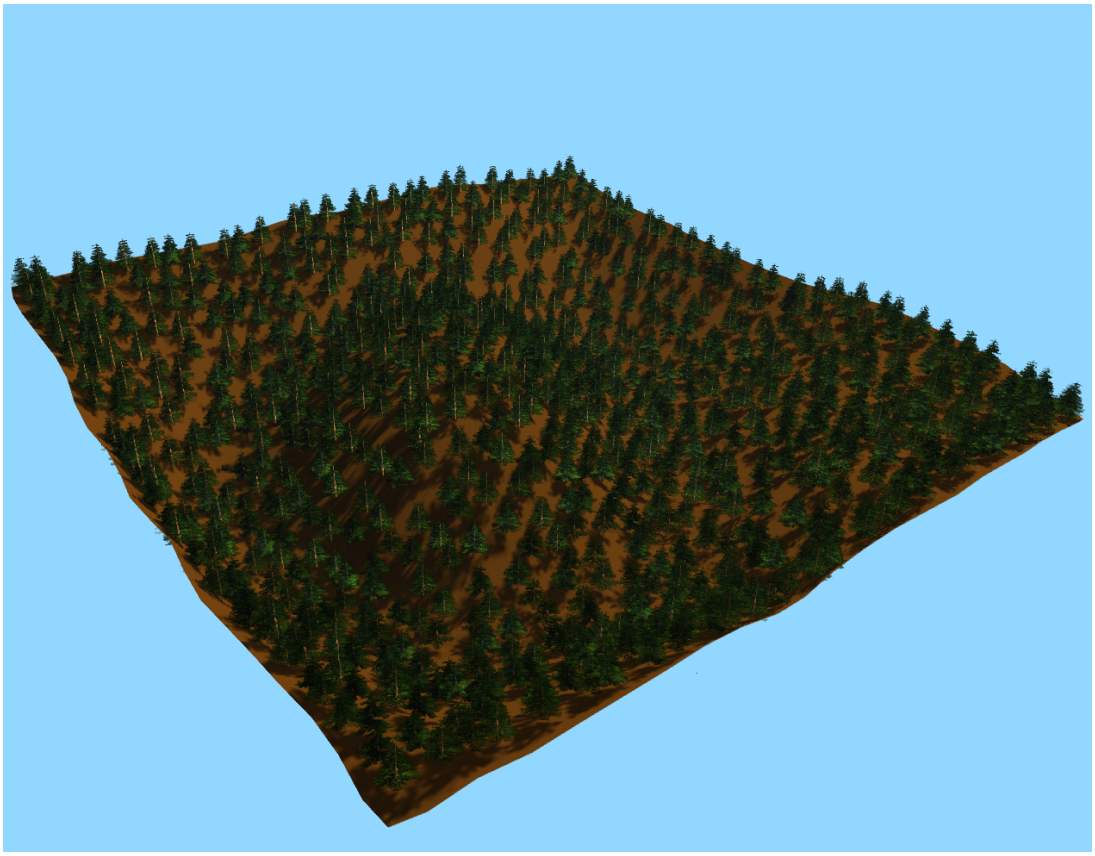

FIG. 5.12 – Une vue globale de la scène.

FIG. 5.13 – une scène de 1000 arbres rendue et ombrée en temps interactif grâce à nos extensions des *billboards* et des cubes de visibilité. *À droite* : notre extension des *billboards* mélange plusieurs vues d'objet ce qui donne un effet de flou quand l'objet est rapproché.

CHAPITRE 6

Hiérarchie de BTF

Au chapitre précédent nous avons exposé une nouvelle représentation permettant de traiter efficacement des objets (*e.g.* un arbre ou une branche) ayant une taille réduite à l'écran (moins de 64×64 pixels) avec illumination et ombrage. En exploitant les propriétés pertinentes des arbres, nous proposons dans ce chapitre de construire une mouture hiérarchie de ces représentations. La complexité de l'apparence des arbres tient essentiellement à la redondance de leurs éléments et à la structure naturellement récursive de leur construction. Les arbres sont composés d'un tronc, de branches principales, elles-mêmes composées de branches secondaires. Les rameaux sont constitués de feuilles ou d'aiguilles. D'autre part, les feuilles, les aiguilles, les branches et les arbres se ressemblent entre eux (*cf.* Étude de cas). Cette propriété permet des simplifications de modélisation : les éléments d'une famille peuvent être représentés par des instances d'un modèle générique, et un élément complexe par un ensemble d'objets similaires, plus simples (*cf.* figure 6.3 colonne de gauche).

Notre but est d'obtenir l'interactivité dans des scènes de forêts, même avec des arbres proches, tout en gérant l'illumination, l'auto-ombrage et l'ombrage. Pour atteindre ce niveau de qualité, nous introduisons une hiérarchie de textures bidirectionnelles : un arbre est encodé en utilisant plusieurs niveaux de détails calqués sur la hiérarchie naturelle évoquée précédemment. Ceci permet un rendu adapté à la taille de l'objet à l'écran, et donc au nombre de détails explicitement visibles. Nous utilisons la même organisation hiérarchique pour le calcul de la visibilité vis à vis de la source de lumière. En combinant les occlusions de cet élément à tous les niveaux de la hiérarchie nous pouvons recalculer en temps interactif les ombrages au fur et à mesure que la lumière bouge.

Nous décrirons à la section 1 le principe de notre hiérarchie, nous ferons à la section 2 une synthèse de la construction des données, nous verrons à la section 3 comment s'effectue le rendu, et nous finirons par les résultats et la conclusion en 4 et 5.

1 Hiérarchie

Nous profitons de la structure fortement hiérarchique des arbres pour construire les niveaux de détails, composés par des *billboards* associés à une fonction bidirectionnelle de texture (*BTF*), par des *VCM* et, éventuellement, par de la géométrie (*e.g.* pour les troncs). La ressemblance entre deux éléments d'arbre nous permet d'utiliser l'instanciation et ainsi de gagner en mémoire. Nous verrons en 1.1 la hiérarchie de *BTF*, en 1.2 la hiérarchie de *VCM* associée et nous finirons en 1.3 par une synthèse expliquant nos niveaux de détails.

1.1 Hiérachie de BTF

Notre construction d'arbre est hiérarchique, nous considérons qu'un arbre est constitué de branches principales, puis qu'une branche principale est constituée de branches secondaires (*cf.* figure 6.3 colonne de gauche). Les branches principales sont construites à partir d'instances de la branche secondaire plus le tronc. Sur le même principe, les arbres sont élaborés à partir d'instances des branches principales (et éventuellement d'instances de branches secondaires lorsque celles-ci ne font pas partie d'une branche principale) et du tronc. Nous profitons de des possibilités d'instanciation que nous offrent les arbres pour ne stocker qu'un nombre limité de formes de branches secondaires et de formes de branches principales.

Cette hiérarchie est conçue pour y calquer les *BTF* et profiter des instanciations pour ne pas surcharger la mémoire. Nous avons trois niveaux de *BTF* entièrement calqués sur cette hiérarchie : un niveau de *BTF* représentant la branche secondaire, un niveau représentant les branches principales et un niveau représentant l'arbre (*cf.* figure 6.1).

Dans notre implémentation, nous utilisons des *L-systems* (*cf.* chapitre 1 section 1.2.1) pour la construction des arbres. Nous avons fait ce choix à cause de la simplicité de l'implémentation, mais n'importe quelle technique fournissant l'information de hiérarchie est utilisable.

Pour l'ombrage il nous faut maintenant associer des *VCM* à cette hiérarchie, ce que nous détaillerons à la section suivante.

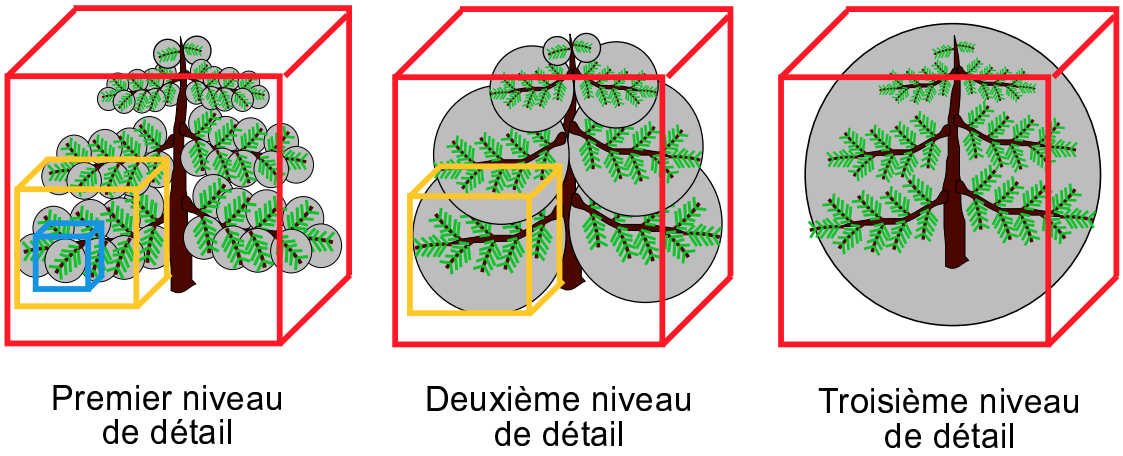

FIG. 6.1 – Nos trois niveaux de détails sont calqués sur la hiérarchie naturelle des arbres. Les objets entourés d'un cercle représentent une instance de *BTF*. Les cubes représentent la hiérarchie de *VCM* : en rouge le *VCM* de l'arbre, en jaune le *VCM* d'une branche principale et en bleu le *VCM* d'une branche secondaire.

1.2 Hiérarchie de VCM

Pour traiter les ombres, chaque instance de branches doit être munie à priori d'une structure de visibilité. Cependant, associer un cube de visibilité à chacune serait extrêmement coûteux : une forêt de 1000 arbres formés de 10 branches principales, chacune constituée de 15 branches secondaires demanderait $1000 \times 10 \times 15 = 150000$ *VCM*, ce qui n'est acceptable ni pour le temps de pré-calcul, ni pour le stockage. Nous introduisons une hiérarchie de *VCM* s'appuyant sur celle des arbres que nous avons vue à la section précédente. Grâce à cette possibilité d'instanciation, le concept de *VCM* devient utilisable à une échelle plus grande.

Pour un niveau de hiérarchie donnée, les *VCM* ne vont contenir que l'information de visibilité vis à vis des autres objets du même niveau. Au niveau le plus haut (l'arbre pour notre implémentation) un *VCM* (en rouge sur la figure 6.1) est stocké pour chaque instance d'objet et contient l'information de visibilité entre l'arbre et le reste de la scène (*i.e.* les autres arbres et le terrain). Pour les niveaux inférieurs, un *VCM* contient l'information de visibilité vis à vis des autres objets de son niveau (et non par rapport à toute la scène). Prenons l'exemple d'une branche principale : elle est composée de branches secondaires dont les *VCM* (en bleu sur la figure 6.1) stockent l'information de visibilité entre elles. Autrement dit le *VCM* d'une branche secondaire ne contient que l'information de visibilité relative à ses congénères, qui se trouvent dans la même branche principale qu'elle.

Avec ce principe, un objet est entièrement "autonome" vis à vis de son auto-ombrage : les *VCM* des sous-objets dont il est composé fournissent l'information d'ombrage interne. Ceci permet d'instancier entièrement cet objet : ses *BTF* tout comme ses *VCM*. L'utilisation massive de l'instanciation est alors possible, donc l'occupation mémoire reste raisonnable : quelques dizaines de Mo pour une scène de 1000 arbres (*cf.* section 4). Si nécessaire, il serait possible d'ajouter des niveaux supplémentaires à la hiérarchie, correspondant par exemple à un groupe d'arbres, ce qui diminuerait d'autant le coût mémoire (mais contraindrait la disposition et l'orientation des arbres).

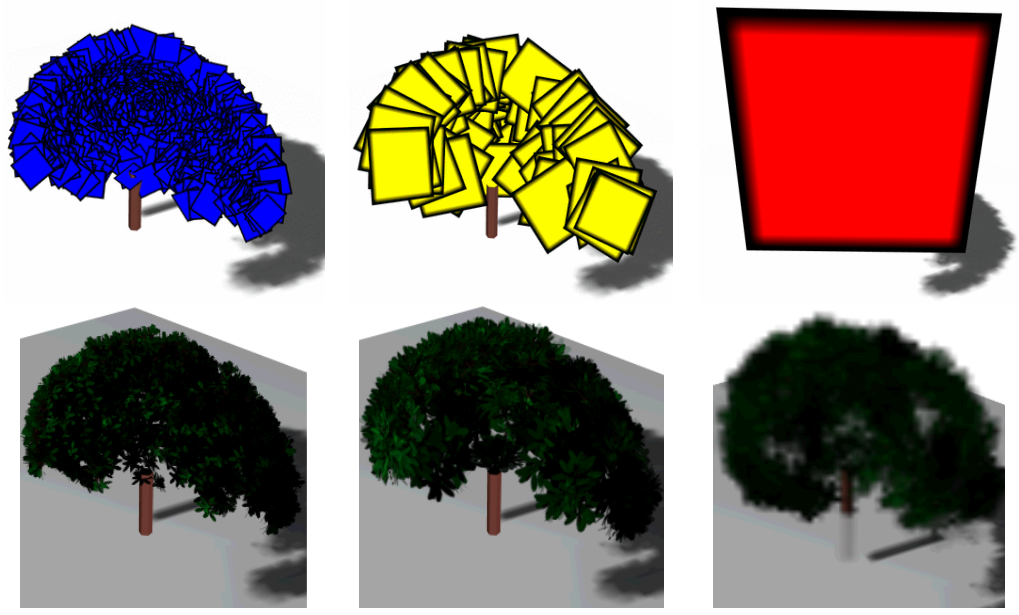

FIG. 6.2 – Les trois niveaux de détails. *À gauche* : les *billboards* (en bleu) représentent les branches secondaires. *Au milieu* : les *billboards* (en jaune) représentent les branches principales. *À droite* : le *billboard* (en rouge) représente tout l'arbre.

1.3 Niveaux de détails

Les trois niveaux de notre hiérarchie constituent aussi les niveaux de détails pour le rendu (*cf.* figures 6.1 et 6.2). Pour ce qui suit, on suppose que nous avons un unique type d'arbre, un unique type de branche principale et un unique type de branche secondaire.
Le premier niveau de détails (le plus précis) est composé :

- des instances de la *BTF* de la branche secondaire,
- des instances des *VCM* donnant la visibilité entre les branches secondaires d'une même branche principale,
- des instances des *VCM* donnant la visibilité entre les branches principales de l'arbre,
- du *VCM* donnant la visibilité entre l'arbre et le reste de la scène (*i.e.* les autres arbres et le terrain).

Le deuxième niveau de détails (intermédiaire) est composé :

- des instances de la *BTF* de la branche principale,
- des instances des *VCM* donnant la visibilité entre les branches principales d'un même arbre,
- du *VCM* donnant la visibilité entre l'arbre et le reste de la scène.

Le troisième niveau de détails (le plus grossier) est composé :

- d'une instance de la *BTF* d'un arbre complet,
- du *VCM* donnant la visibilité entre l'arbre et le reste de la scène.

Nous verrons à la section 3.2 comment les valeurs de visibilité des différents niveaux de hiérarchie sont combinées lors du rendu. L'ordre et la manière de construire les données sont très importants pour la compréhension de cette hiérarchie, nous allons donc les détailler dans la section suivante.

2 Constructions des données

Nous détaillons maintenant le parcourt à suivre lors de la construction des données d'une scène complète. Nos scènes sont principalement composées du terrain, et de nombreuses instances d'un ou plusieurs modèles d'arbres. Pour fabriquer les données, nous avons besoin d'encoder dans notre représentation un modèle d'arbre existant, de manière aussi transparente que possible pour l'utilisateur. Deux structures hiérarchiques imbriquées sont à construire : les *BTF* encodant l'apparence de chaque arbre et branche, et les *VCM* encodant la visibilité d'un objet dans toutes les directions (*i.e.* visibilité avec la lumière). Cette construction s'effectue récursivement et est illustrée par la figure 6.3.

Nous partons d'une représentation géométrique d'une branche secondaire, dont nous calculons la *BTF* par lancer de rayons[1]. Le lancer de rayons offre de nombreuses fonctionnalités comme l'auto-ombrage, l'anti-aliassage (en utilisant par exemple le lancer de cônes déjà utilisé au chapitre 3), la transparence, et l'utilisation de modèles d'illumination évolués de manière à obtenir des images les plus réalistes possibles. Les éléments de la hiérarchie sont composés de *BTF* et d'objets géométriques quelconques (*e.g.* des polygones pour le tronc), la seule contrainte étant que leur rendu soit rapide pour ne pas pénaliser le rendu de toute la scène.

Les *BTF* des branches secondaires sont instanciées pour construire une branche principale, puis nous calculons leurs *VCM* associés pour l'ombrage. Avec ceci, les *BTF* des branches prin-

[1] Les rendus pour les autres niveaux et le rendu final se font avec notre moteur de rendu accéléré par le matériel graphique.

cipales sont calculées par notre moteur de rendu[2]. Pour le calcul de la *BTF* de l'arbre complet, on instancie les *BTF* des branches principales et des branches secondaires auxquelles on associe des *VCM*. Cette *BTF* d'un arbre complet est ensuite instanciée pour former la scène, pour finir le *VCM* associé à chaque arbre donnant la visibilité entre l'arbre et le reste de la scène est calculé. À chaque étape de la construction, nos objets disposent de l'illumination, l'auto-ombrage et l'ombrage, notre rendu est donc complet et permet de construire des *BTF* réalistes.

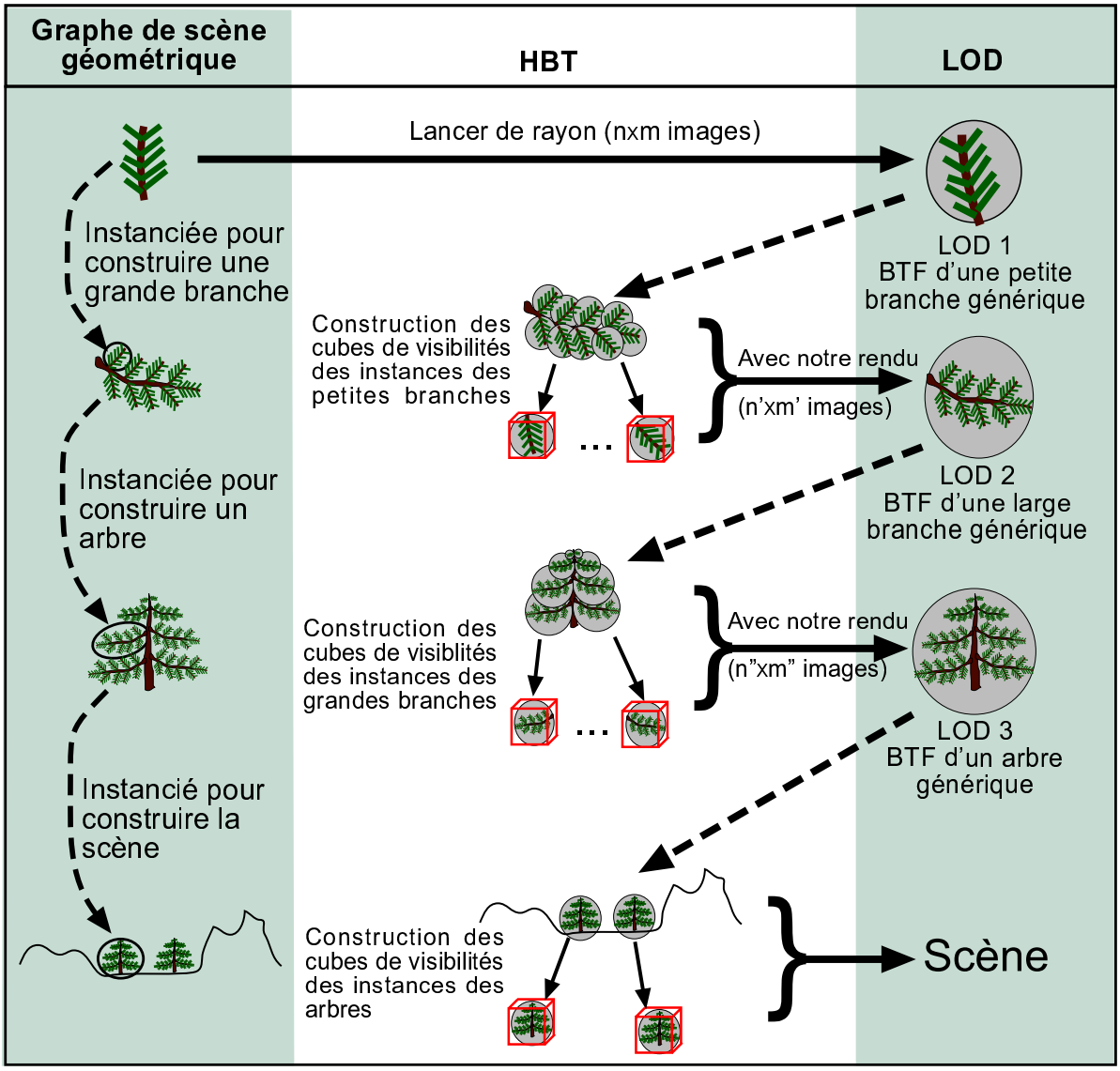

FIG. 6.3 – Schéma global de construction de la hiérarchie.

[2]Notre implémentation utilise *OpenGL* et l'accélération matérielle des cartes graphiques ($Onyx^2$ Infinite Reality dans notre cas).

3 Rendu

Nous venons de décrire la structure de notre représentation hiérarchique ainsi que sa construction. Cette partie est consacrée à la description de la phase de rendu. Nous traiterons en 3.1 la question du choix du niveau de détails, puis nous présenterons en 3.2 le calcul de l'ombrage d'un objet de la hiérarchie, et nous finirons en 3.3 par notre algorithme de rendu en indiquant les parties accélérées par le matériel graphique.

3.1 Choix du niveau de détails

Lors du rendu il faut déterminer le "bon" niveau de détails, celui qui sera le moins coûteux, tout en réduisant l'aliassage au maximum, et en limitant l'effet de clignotement (*poping*) lors des transitions.

En utilisant des textures, le critère que nous avons présenté à la section 3.2 du chapitre 3 se traduit par : la résolution de l'image utilisée comme texture doit être, une fois projetée, plus fine que la résolution de l'écran. Un polygone de taille l, se trouvant à distance z de la caméra aura une taille de $\frac{l}{z}N.V$ dans le repère écran, avec N la normale du polygone et V le vecteur allant de l'œil vers l'objet. Soit r la résolution de la texture se trouvant plaquée sur le polygone[3], la taille d'un pixel de texture $t_{texture}$ dans le repère écran vaut $t_{texture} = \frac{l}{z.r}N.V$. Les polygones de nos *billboards* sont quasiment toujours parallèles à l'écran, nous supposons donc que $N.V \approx 1$. Notre critère indique que la taille d'un pixel de texture doit être inférieure à la taille d'un pixel écran, ce qui se traduit par $t_{texture} = \frac{l}{z.r} < 1$.

Pour déterminer si un niveau de détails donné est valide il faut considérer les valeurs de l et r de ses *billboards* dont le rapport $G = l/r$ est le plus grand, car ceux sont eux qui sont susceptibles de ne pas respecter le critère. Si $\frac{G_{max}}{z} < 1$ est vérifié alors, ce niveau de détails est valide pour la distance z.
Remarque : cette formule nous donne les plages de distances pour lesquelles le niveau de détails est valide.

3.2 Calcul de l'ombrage dans la hiérarchie

Nous avons vu à la section 2.2 du chapitre 5, que pour habiller d'ombres un *billboard*, nous calculions quatre coefficients d'assombrissement pour les quatre sommets du polygone sous-jacent. Nous obtenons chacune de ces quatre valeurs par interpolation tri-linéaire en fonction des valeurs de visibilité données par les *VCM* placés autour de l'objet. Avec la version hiérarchique de notre représentation il nous faut tenir compte aussi des *VCM* placés dans tous les niveaux supérieurs de la hiérarchie. Avec les trois niveaux de détails que nous utilisons pour les arbres, l'ombrage se calcule ainsi (*cf.* figure 6.4) :

- L'ombrage d'un arbre rendu avec le premier niveau de détails (*i.e.* le *billboard* représente l'arbre complet) est similaire à ce que nous avons vu à la section 2.2 du chapitre 5. L'ombrage d de l'arbre par rapport à la scène est calculé par logiciel aux quatre sommets du polygone, et sera multiplié avec l'auto-ombrage a inclus dans l'image de la *BTF* par le matériel graphique au moment de la rasterisation.
- L'ombrage d'une branche principale d'un arbre rendu avec le deuxième niveau de détails est calculé par logiciel aux quatre sommets du polygone. Chacune de ces quatre valeurs est calculée par $c \times d$, où c est la valeur de visibilité du sommet par rapport aux autres branches

[3] Dans notre implémentation la résolution des images des *BTF* est 32×32 ou 64×64.

principales de l'arbre (donnée par les *VCM* jaunes), et d est la valeur de visibilité du sommet par rapport à la scène (donnée par les *VCM* rouges).

– L'ombrage d'une branche secondaire d'un arbre rendu avec le troisième niveau de détails est calculé par logiciel aux quatre sommets du polygone. Chacune de ces valeurs est calculée par $b \times c \times d$, où b est la valeur de visibilité du sommet par rapport aux autres branches secondaires de sa branche principale (donnée par les *VCM* bleus), c est la valeur de visibilité du sommet par rapport aux branches principales autres que celle à qui elle appartient (donnée par les *VCM* jaunes), et d est la valeur de visibilité du sommet par rapport à la scène (donnée par les *VCM* rouges).

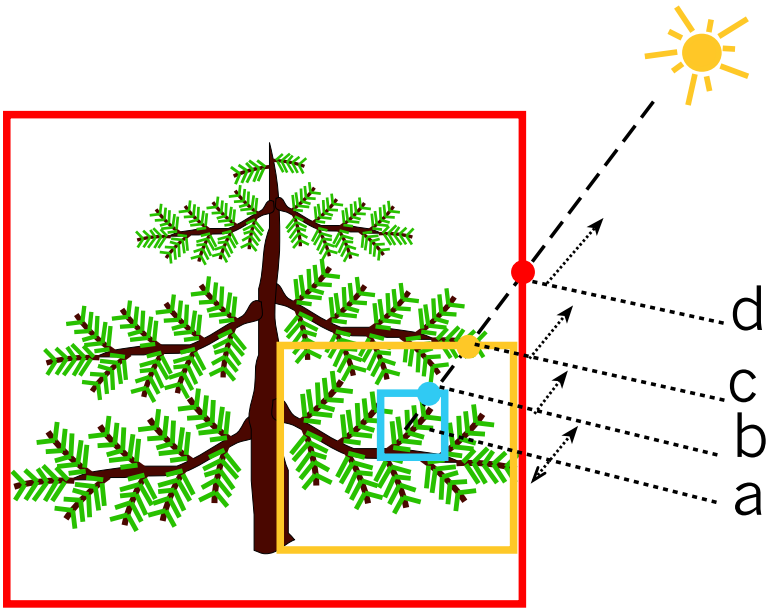

FIG. 6.4 – L'ombrage dans la hiérarchie de visibilité est donné par la multiplication de : a l'auto-ombrage inclus dans les images de la *BTF*, b l'ombrage des branches secondaires entre elles, c l'ombrage des branches principales entre elles, d l'ombrage de l'arbre par rapport à la scène.

En résumer, le statut des ombres suit le schéma suivant :

– l'auto-ombrage a est encodé dans les images de la *BTF* (*cf.* figure 6.4).
– les *VCM* d'un objet contiennent l'information de blocage de la lumière avec les autres objets du même niveau de hiérarchie.
– les cubes de visibilité au plus haut niveau de la hiérarchie encodent l'information de blocage entre leur objet associé (*i.e.* un arbre) et le reste de la scène (*i.e.* les autres arbres et le terrain).

3.3 Algorithme de rendu

Notre algorithme de rendu est le suivant (*cf.* figure 6.1) : nous n'affichons que les arbres qui se trouvent dans la pyramide de vue, nous les trions de l'arrière vers l'avant (pour gérer correctement la transparence), puis nous choisissons le niveau de détails de chaque arbre en fonction de sa distance à l'œil. Nous traitons tous les éléments géométriques éventuellement existants (*e.g.* tronc) puis, toutes les *BTF* (en les représentant par des *billboards*), en déterminant les coefficients d'ombrage à l'aide de la hiérarchie de visibilité.

Algorithme 6.1 Algorithme de rendu

Afficher_terrain(*depthshadowmap, alphashadowmap*)
Affiche le terrain avec ses ombres (matériel).

Arbres_pyramide ← ***Selectionne_Arbres_Visibles***(Arbres)
Sélectionne les arbres se trouvant dans la pyramide de vue (logiciel).

Arbres_triés ← ***Trie_Arbres***(Arbres_pyramide)
Trie les arbres visible de l'arrière vers l'avant (logiciel).

pour *A* ∈ *Arbres_triés* **faire**

Alod ← ***Determine_LOD***(A)

Affiche_ElementGéométrique(*Alod*)
*Affiche les éventuelles éléments géométrique (*e.g. *tronc) du niveau de détails,*
i.e. *ceux qui ne sont pas des BTF (matériel).*

pour *BTF* ∈ *Alod* **faire**

VCM ← ***Donne_VCM***(*BTF, Alod, A*)
Détermine les VCM de la hiérarchie nécessaire au calcul des ombres qui suit.

Coeff_Ombre ← ***Calcul_Ombre***(*BTF, VCM*)
Calcule les coefficients d'ombre du billboard en multipliant
les différentes valeurs d'ombrage données par la hiérarchie de VCM (logiciel).

pour *Image* ∈ *BTF* **faire**

Pimage ← ***Donne_Pondération***(*Image, Table*)
Détermine la pondération de l'image grâce à une table pré-calculée (logiciel).

Affiche_Billboard(*Image, Pimage, Coeff_Ombre*)
OpenGL *traite le billboard (matériel).*

fin pour

fin pour

fin pour

FIG. 6.5 – *De gauche à droite et de haut en bas* : un pin avec ombres, sans ombres, valeur de l'ombrage. Le pin avec seulement l'ambiant, coefficient de visibilité ambiante (*i.e.* près du tronc moins de lumière arrive du ciel), les *billboards* utilisés au niveau de détail le plus fin.

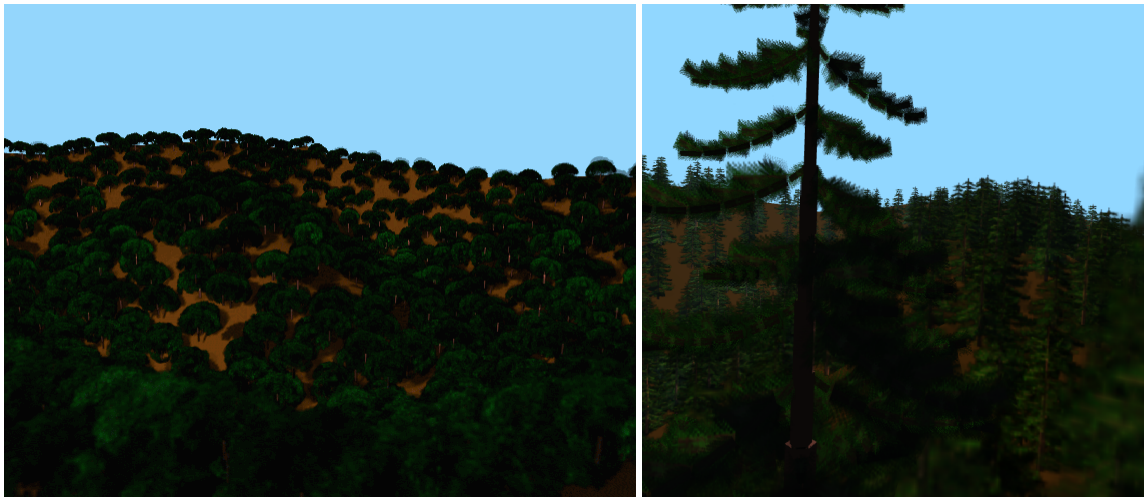

FIG. 6.6 – *cf.* figure 6.7

4 Résultats

Dans notre implémentation, nous utilisons des images de tailles 32×32 ou 64×64 en *RGBA*. Pour l'observateur et pour la lumière, notre discrétisation de la sphère est de 6 ou 18 échantillons, ce qui offre un bon compromis entre coût mémoire et qualité visuelle. Ces *BTF* coûtent en mémoire texture lors du rendu $32 \times 32 \times 4 \times 6 \times 6$ (144 Kb), $32 \times 32 \times 4 \times 18 \times 6$ (432 Kb), ou $32 \times 32 \times 4 \times 18 \times 18$ (1.3 Mb), plus la partie ambiante $32 \times 32 \times 4 \times 6$ (24 Kb) ou $32 \times 32 \times 4 \times 18$ (72 Kb).

Nos modèles d'arbres ont été générés par *L-system*. Nous avons utilisé deux sortes d'épineux et une sorte de feuillus pour nos tests, chacun représenté par trois niveaux de la hiérarchie décrite à la section 1 : les branches secondaires comportent des feuilles ou des aiguilles, les branches principales sont constituées à partir d'instances de branches secondaires, et les arbres sont constitués d'instances de branches principales et de branches secondaires. Les sapins comptent environ 30 branches principales, 300 branches secondaires et 40000 aiguilles. Un arbre suit le même graphe de scène que l'arbre géométrique (ce qui ne pose aucun problème grâce au *L-system*).

Pour les *VCM*, nous avons calculé une résolution de 32×32 soit 6 Ko par *VCM*. La scène de test (*cf.* figure 6.7) contient environ 1000 arbres, et grâce à l'instanciation $8 \times (1000 + 30 + 10)$ *VCM* seulement : 1000 pour les arbres, 30 pour les branches principales formant l'arbre de référence et 10 pour les branches secondaires formant la branche principale de référence. La structure de visibilité coûte donc $8 \times 6 \times 1000$ Ko = 48 Mo pour une scène comportant 1000 instances d'un arbre (elle est stockée en mémoire centrale sans jamais être chargée sur la carte graphique). Le temps de pré-calcul de toutes les données de la scène est d'environ 75 minutes en utilisant une Onyx^2 Infinite Reality, dont 2/3 pour la visibilité et 1/3 pour la hiérarchie de *BTF*.

Durant le rendu, nous tenons compte du soleil comme source directionnelle de lumière, et de l'illumination du ciel comme source ambiante (*i.e.* une source de lumière plus l'ambiant). Dans le pire des cas, notre algorithme nécessite l'affichage de quinze polygones texturés par objet : trois pour traiter la composante *alpha*, neuf pour l'illumination directe et trois pour l'illumination ambiante (*i.e.* le ciel). L'optimisation que nous avons énoncée à la section 1.2.3 du chapitre 5 pour afficher le *billboard* permet de ne considérer que cinq images en moyenne au lieu de neuf, ce qui multiplie quasiment par deux le taux de rafraîchissement.

La carte graphique *Infinite Reality* que nous avons utilisée ne dispose pas de la fonctionnalité de *multitexturing*[4], alors que les cartes graphiques récentes permettent plusieurs textures par polygones : ce qui diviserait d'autant le coût de rendu.

Nous avons produit des animations de 640×480 (*cf.* figure 6.7), calculées sur Onyx^2 Infinite Reality, montrant un survol d'une petite forêt de 1000 arbres avec un taux de rafraîchissement de 7 à 20 images par seconde (nous avons un gain de 20% par rapport à ces chiffres en utilisant le même programme avec la carte *Nvidia GeForce 2* de *Nvidia*, *i.e.* sans profiter du *multitexturing*).

Un sapin géométrique classique est représenté par à peu près 120000 polygones. Avec le niveau de détails le plus précis nous avons un taux de rafraîchissement huit fois supérieur au rendu avec tous les polygones, avec le niveau intermédiaire un gain de 18 et avec le niveau le plus grossier nous avons un gain de 30.

[4] Le *multitexturing* permet d'appliquer plusieurs textures à un polygone en une seule passe de rendu.

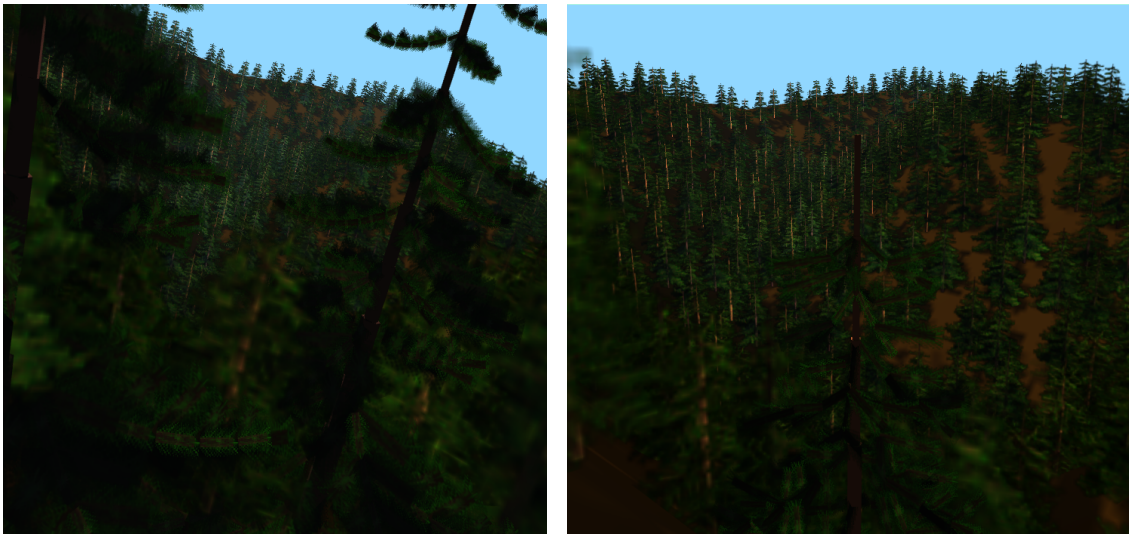

FIG. 6.7 – Quatre vues d'une forêts (1000 arbres sur un terrain avec éclairage et ombres). Le rendu s'effectue entre 7 et 20 images par seconde sur notre machine de test ($Onyx^2$ Infinite Reality). En utilisant les fonctionnalités des nouvelles cartes graphiques, le nombre d'images par seconde peut largement être amélioré. Remarquez les arbres détaillés au premier plan. Une animation est disponible à l'adresse : http ://www-imagis.imag.fr/ Alexandre.Meyer/research/MNP01/index.html

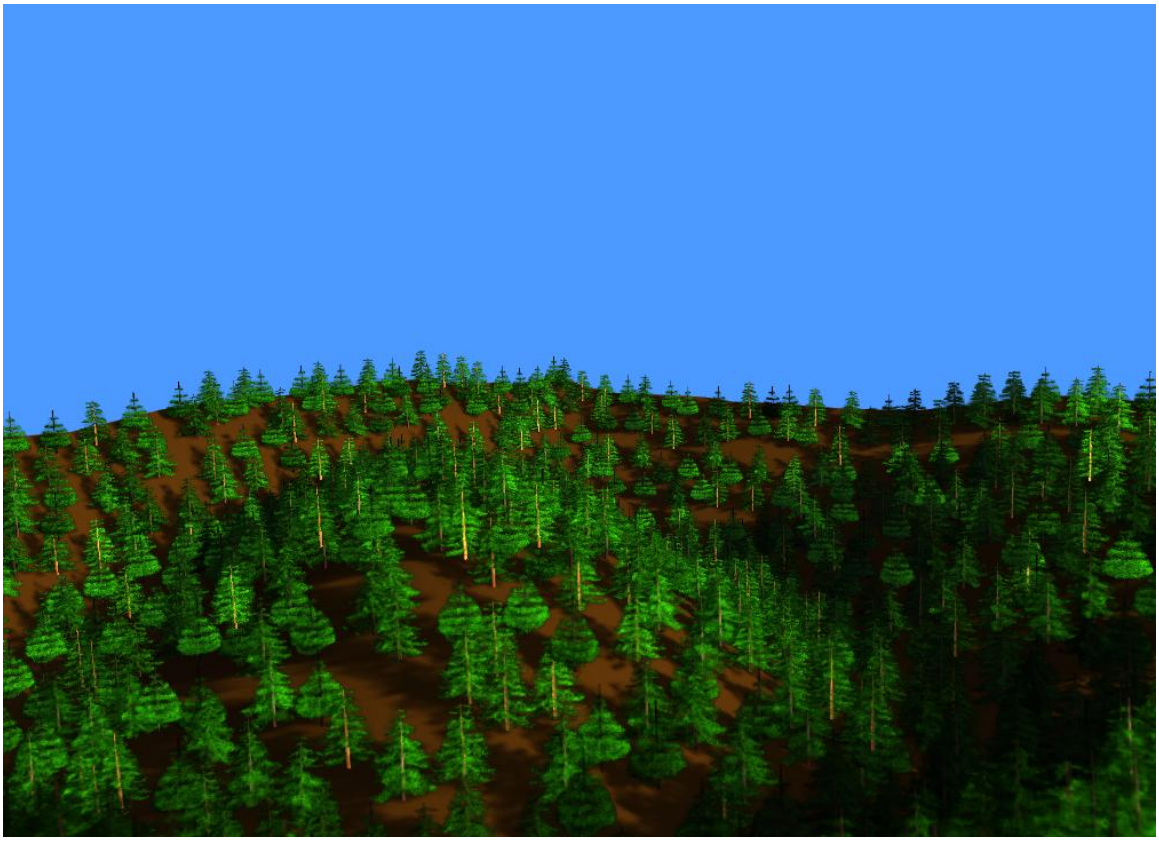

FIG. 6.8 – *"La forêt du Père Noël"* (Paul, 3 ans)

5 Conclusion et perspectives

Nous avons introduit une nouvelle représentation à base d'images, associée à une structure de visibilité, destinée au rendu d'arbres. Celle-ci donne des images de qualité avec des effets complexes comme l'illumination, l'auto-ombrage et l'ombrage, l'illumination du ciel, en tenant compte du mouvement du soleil. Notre implémentation, naïve, permet un taux de rafraîchissement de 7 à 20 images par seconde sur une Onyx^2 Infinite Reality avec une scène de 1000 arbres. L'usage des nouvelles générations de cartes devrait permettre de gagner encore un ordre de grandeur.

Notre hiérarchie est constituée de *BTF* associées à des *billboards* sur laquelle se calque une hiérarchie de *VCM* pour l'ombrage. Malgré les six degrés de liberté d'une *BTF*, la mémoire consommée pour la scène complète est seulement de quelques dizaines de mégaoctets. Notre représentation est efficace en terme de mémoire si les données se prêtent à une hiérarchisation et à l'instanciation, ce qui est le cas des arbres. Elle pourrait être appliqué sur d'autres types de données (*e.g.* ville).

Spécularités, *BRDF* et transparence sont gérées par notre méthode, mais la précision en est limitée par la densité de l'échantillonnage des directions de vue et de lumière. L'échantillonnage des directions et des positions des *VCM* ont les mêmes défauts, et l'interpolation tri-linéaire ne permet pas une reconstruction parfaite des données. Ce problème disparaît quand l'échantillonnage est très fin, au prix du coût mémoire. Il y a donc un équilibre à trouver entre qualité visuelle et quantité de mémoire utilisée. Néanmoins, les résultats montrent qu'un survol de qualité au-dessus d'un terrain peut être produit avec un coût mémoire raisonnable, et avec plusieurs types d'arbres différents.

Des améliorations à notre algorithme de rendu peuvent être apportées, comme l'introduction d'une grille pour limiter le nombre d'objets à rendre ce qui permettrait d'augmenter la taille de la scène calculée. Différents autres choix d'implémentation (par exemple avec une sphère 4D) peuvent être envisagés pour notre structure de *BTF* et de *VCM* pour la construction et le rendu.

Afin d'améliorer le réalisme, il peut être envisager d'utiliser comme point de départ de notre hiérarchie des photos réelles de branches secondaires (comme toute la hiérarchie est basée sur la première *BTF*, en améliorant le réalisme de celle-ci, on améliore le réalisme global). Des tests sur des données réelles d'arbres sont aussi à envisager : tester la réaction de notre hiérarchie avec des arbres dont la structure est plus proche du réel que nos arbres générés par *L-system*.

Il serait aussi intéressant de voir comment les nouvelles et futures fonctionnalités des cartes graphiques permettraient d'améliorer cette technique. Le *multitexturing* permettrait certainement de diminuer rapidement le nombre de passes et donc d'augmenter le nombre d'images par seconde. Le calcul d'illumination par pixel pourrait être appliqué à l'ombrage des *billboards*, en vue de le rendre plus précis. Les texture 3D font leur apparition sur les cartes de type *Geforce*, on peut imaginer des textures 4D, qui permettraient de traiter les *BTF* directement par le matériel graphique.