An Efficient Visibility Model for Real-time Camera Control

Murat Fersatoglu

An Efficient Visibility Model for Real-time Camera Control

Murat Fersatoglu

5 juin 2012

Résumé

Beaucoup d'avancées ont été réalisées sur les systèmes de synthèses d'images, aussi bien sur les procédés basés sur le lancer de rayons que ceux basés sur la rasterisation. Cependant l'image finale dépend du point de vue dans lequel on se place et donc de la caméra virtuelle et sa configuration. Pour les applications temps réelles telles que les jeux vidéos, la gestion de cette caméra est primordiale pour ne pas frustrer l'utilisateur. Or, pour bien planifier ses mouvements, le système doit être capable d'estimer la visibilité de la cible et pouvoir la suivre. C'est ce que nous cherchons à faire. C'est à dire trouver un modèle pour la visibilité permettant de suivre une cible dans un environnement virtuel. Nous réalisons également une étude comparative des différents modèles déjà existants.

Mots clefs : visibilité, contrôle de caméra, réalité virtuelle, infographie
Keywords : visibility , camera planning, virtual reality, computer graphics

Table des matières

1 Introduction

Aujourd'hui, les environnements virtuels ont atteint un haut niveau de réalisme. Aussi bien du point de vue de la qualité du rendu des images que de la simulation physique et de l'animation. Cependant, l'image générée et rendue dépend du point de vue dans lequel on se place. Ainsi, la caméra virtuelle et sa configuration induisent le résultat escompté.

La littérature scientifique est peu fournie concernant la gestion de la caméra virtuelle et particulièrement pour l'estimation de la visibilité. Les buts de ce stage sont doubles : faire une étude comparative des différentes méthodes qui ont été présentées jusqu'à maintenant et en déduire ensuite un nouveau modèle.

Dans un premier temps il serait intéressant de définir le calcul de visibilité et le contrôle de caméra.

Contrôle de caméra : Gestion des mouvements, du positionnement, de l'orientation et de la configuration[1] de la caméra virtuelle dans son environnement. Elle se fait avec le calcul de la visibilité du ou des sujets, la composition voulue dans le rendu final (positionnement des objets dans l'image) et la recherche de chemins à suivre pour l'animation. Dans certains cas de figure, comme le jeu vidéo par exemple, les contraintes esthétiques inspiré de la cinématographie sont également à prendre en compte. Ce contrôle peut être effectué soit automatiquement par l'application soit par l'utilisateur via des métaphores d'interactions, c'est le cas des jeux vidéos type FPS[2] par exemple. Dans notre cas, nous nous intéresserons au contrôle automatique.

Calcul de visibilité : Estimer la visibilité d'une cible par rapport à une référence. La pertinence ainsi que la précision de l'estimation dépendent du domaine dans lequel on se place. Pour la suppression des objets cachés en synthèse d'image elle est surestimée et binaire. En effet un objet majoritairement occulté sera considéré par le système de traitement comme étant totalement visible pour ne pas altérer la qualité de l'image rendue et ne pas avoir de pertes d'informations. Pour le contrôle de caméra automatique et temps réel par contre, on a besoin d'une estimation plus précise pour déterminer le meilleur point de vue dans le contexte voulu.

Les contributions sont tout d'abord d'apporter une étude comparative sur les deux principaux types de calculs réalisés pour l'estimation de la visibilité dans le

[1] ouverture (focale), profondeur de champs, aspect ratio (16/9, 4/3, etc.)

[2] Jeux de tirs à la première personne

contrôle de caméra. C'est à dire le raycast ainsi que les techniques basées sur le rendu. Ensuite nous proposons un modèle pour que le comportement de la caméra soit restreint par des trajectoires prédéfinies pour le suivi d'objet mobile dans un environnement dynamique. Ce modèle est expliqué plus en détails dans la section 4 du présent rapport.

Tout d'abord nous allons présenter un état de l'art du calcul de visibilité en infographie et de la gestion de caméra. Ensuite nous allons procéder à une étude comparative des différents modèles présentés. Pour finir nous présenterons un nouveau modèle.

2 État de l'art

Afin de bien comprendre le contexte du stage, nous allons établir quelques définitions. Ensuite nous allons présenter ce qu'est le contrôle de caméra en décrivant les différentes techniques proposées dans la littérature ainsi que la classification associée. Pour finir nous expliquerons plus en détails l'estimation de la visibilité en général.

2.1 Définitions

Fragment : le fragment est la terminologie utilisée par OpenGL pour définir la projection d'un point d'une face sur la fenêtre de rendu.

Stencil buffer : C'est une mémoire tampon qui permet de faire un rendu à travers un pochoir, *stencil* se traduit par pochoir en français. Elle est notamment utilisée pour rendre le reflet de la scène sur un miroir.

Occlusion query : C'est une extension qui a été ajoutée par Nvidia et qui depuis avril 2003 fait partie du standard ARB OpenGL. Elle permet de compter le nombre pixels visible d'un objet rendu.

2.2 Classification des techniques de contrôle de caméras

Dans cette section nous allons présenter une classification des différents systèmes de contrôle de caméra temps réel en nous basant sur les travaux de M. Christie et coll. [9].

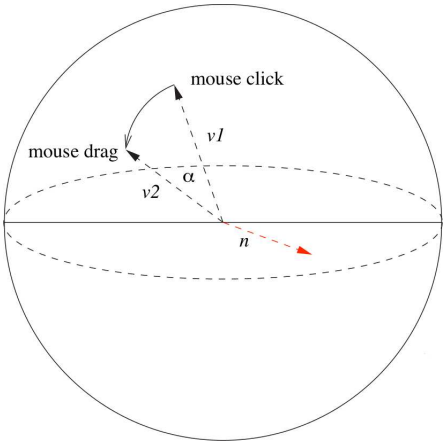


FIG. 1 – l'arcball de Shoemake, l'angle et l'axe de rotation sont calculés à partir de deux position de souris et le centre d'une boule. Schéma repris d'une publication de Christie et coll.[9]

Les systèmes algébriques

Le modèle est un système algébrique avec la configuration de caméra souhaitée comme résultat. Un exemple possible est le système proposé par Blinn[4] dans lequel on détermine la configuration de la caméra pour voir deux sujets abstraits par un point. Ces procédés ont l'inconvénient de ne pas prendre en considération les occultations éventuelles.

Les systèmes de contrôle interactifs

Utilisation de règles de mappage entre les périphériques d'entrées comme la souris ou le clavier et les mouvements de la caméra associés dans l'environnement virtuel. Ceci implique que la caméra est sous le contrôle de l'utilisateur. L'*arcball* proposé par K. Shoemake[18] est un exemple dans lequel l'orientation de la caméra est calculée en fonction des mouvements de la souris déplacée par l'utilisateur, comme montré dans la figure 1.

Les systèmes temps réels réactifs

Procédés utilisant des algorithmes provenant de la robotique ou dédiés pour suivre une cible unique dans un environnement virtuelle dynamique.

2.3 Calcul de visibilité dans la synthèse d'images

Nous nous basons sur la thèse de M. Durand[10] qui range les différentes techniques en quatre catégories :

Viewpoint-space

Dans cette catégorie la visibilité d'un objet est déterminée par l'ensemble des points de vue possibles dudit objet. Cette représentation est structurée sous la forme d'un graphe où les arcs décrivent les transitions entre chaque point de vue.

Object-space

Ici les propriétés propres à l'objet considéré sont utilisées pour obtenir la visibilité de ce dernier.

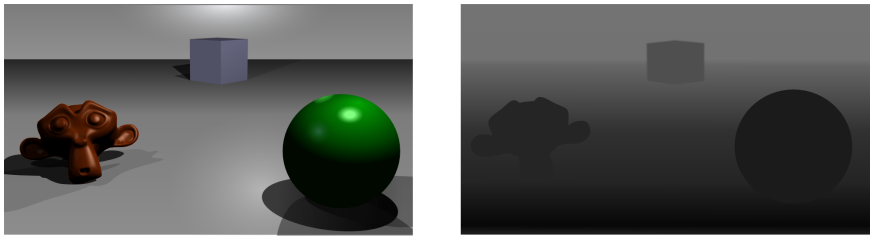
Un exemple simple est le *backface culling* qui détermine si une face d'un objet (polygone) est visible ou non et donc si elle peut être supprimé de la chaine de rendu. Cette méthode consiste à calculer la normale du polygone et la comparer à la configuration de la caméra. Si l'angle entre le point focal, la face considérée et la normale à cette face est supérieur à 90 degrés, alors elle n'est pas visible. On dit qu'elle ne *regarde* pas la caméra.

Image-space

Dans ce domaine, on estime la visibilité via une projection de la scène dans un espace de dimension inférieure (image 2D pour le cas d'un environnement 3D). Ce type de méthode est beaucoup utilisé en synthèse d'images.

Le Z-buffer (carte de profondeur) en est un exemple et implémenté matériellement dans les cartes accélératrices 3D actuelles. Le fonctionnement est simple et suit les étapes suivantes :

1. Étalonnage du tampon de profondeur à partir des deux plans du frustum de vue
2. Initialisation de la carte de profondeur avec la profondeur maximale pour chaque pixel.



Une simple scène 3D | La carte de profondeur associée

FIG. 2 – Représentation d'un rendu (à droite) et de la carte de profondeur correspondante (à gauche)

3. Pendant la rasterisation le fragment est calculé s'il est plus près du point de vue que le fragment précédent. La profondeur du fragment est également mis à jour

Ainsi lors de la rastérisation la visibilité d'un fragment est déterminée en fonction de sa profondeur.

Un autre procédé *Image-space* est l'*occlusion query* qui compte le nombre de pixels visibles d'un objet en comptant le nombre de fois que le tampon de profondeur est mis à jour. La carte de profondeur doit bien sûr être initialisée. Bittner et coll.[3] proposent un modèle utilisant les *occlusion queries* pour enlever les objets non visibles, aucun pixel visible, de la chaine de rendu à des fins d'optimisation.

Line-space

Dans ce domaine, la visibilité est déterminée entre des points. Si un segment entre deux points n'est pas coupé, les deux points en question se voient mutuellement.

Par exemple le *2D visibility complex* représente la visibilité entre les objets dans un espace à deux dimensions par l'ensemble des segments de taille maximal ne coupant aucun objet. Vous pouvez voir un schéma représentant ce modèle dans la figure 3.

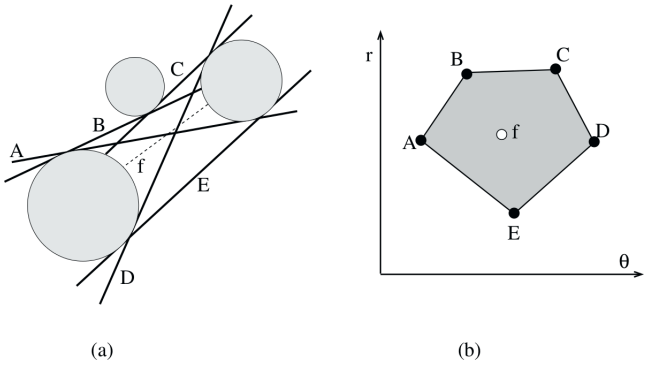


FIG. 3 – a) Représentation du *2D visibility complex* dans l'espace. b) Même chose en coordonnées polaires. Schéma repris de la thèse de Durand[10]

2.4 Calcul de visibilité dans la gestion de caméras virtuelles

Maintenant que nous avons vu quelques méthodes générales d'infographie pour le calcul de visibilité, nous allons nous intéresser aux modèles proposés pour le contrôle de caméra virtuelle. Nous allons distinguer deux grandes familles de méthodes :

Déterminer la visibilité d'une cible à partir d'une configuration de caméra

Dans ce cas de figure, la visibilité est obtenue à partir d'une ou plusieurs configurations de caméra. Ainsi, le système peut réagir en conséquence. Un exemple de méthode utilisant cette approche est celle proposé par Li et coll.[13] qui est basée sur la PRM[3][5].

L'algorithme génère et met à jour dynamiquement via un graphe l'ensemble des mouvements de caméra possibles autour d'une cible. Pour cela, il calcule tout d'abord un nuage de points, correspondant aux points de vue, autour de la cible. Ensuite tout les points se *voyant* sont connectés entre eux. Le graphe ainsi généré se met à jour au fur et à mesure que le sujet se déplace, ajoutant des nœuds si nécessaire. La meilleur position de caméra est obtenue à partir d'une fonction de coût qui prend en considération la hauteur de la caméra, sont angle par rapport à

[3]Probabilistic Roadmap Method

la ligne d'intérêt[4], la distance par rapport à la cible et enfin la visibilité de cette dernière. Ce coût est minimisé au fur et à mesure du parcours dans le graphe.

Les procédés présentés plus tard dans les sections 3.4 et 3.3 sont également à mettre dans cette catégorie.

Déterminer l'ensemble des configurations de caméra qui *voient* la cible, abstrait par un point

Ici on calcule l'ensemble des positions possibles pour que la caméra puisse voir la cible. Cette approche contrairement à la précédente a l'avantage de calculer un ensemble des configurations de caméra possibles. Cependant nous n'obtenons que la visibilité d'un point. En effet, si nous voulons une approximation plus précise du sujet, la méthode doit être réitérée sur plusieurs points de la cible.

Le PVR[5][11], proposé par Halper et coll, est un modèle de cette famille. La méthode est illustrée par la figure 4, les rendus se font de la cible vers la position focale courante.

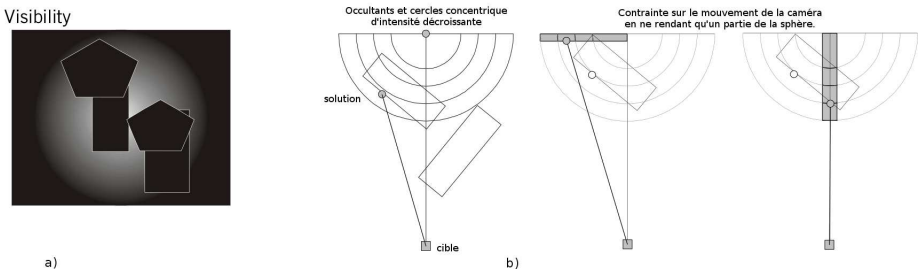


FIG. 4 – a) Rendu de la visibility map (occulants en noir et position de caméra en blanc. b) Schéma explicatif

La position de la caméra est obtenue en récupérant le pixel le plus lumineux du rendu final. La luminosité du point considéré donne la profondeur du point focal calculé par rapport à la cible.

L'inconvénient majeur de ce modèle est qu'il nécessite plusieurs rendus, le nombre étant dépendant de la complexité de la scène. De plus cette méthode

[4]Ligne suivant laquelle
[5]*Potential Visibility Region*

ne prend pas en considération les chemin suivis ou possibles de la caméra et l'implication de la visibilité de la cible sur cette dernière.

3 Étude comparative des techniques actuelles

Maintenant que nous avons présenté quelques modèles pour le calcul de visibilité dans le contrôle de caméra, nous allons en faire une étude comparative. Pour cela, nous allons différencier les techniques basées sur le raycast et celles basées sur le rendu. Dans le cas présent nous considérerons la visibilité de notre cible comme un nombre compris entre 0 et 1, 0 correspondant à une occultation totale et 1 à une visibilité totale. Ce comparatif s'intéresse principalement à la visibilité de la cible en fonction d'une configuration de caméra, comme présenté dans la section 2.4. Dans un premier temps précisons les critères de comparaison utilisés dans cette étude. Ensuite présentons les deux procédés les plus utilisés sur lesquels la majorité des modèles de contrôle de caméra sont basés.

3.1 Critère de comparaison

Nous nous baserons sur les critères suivants pour notre étude :

- La précision de l'estimation de la visibilité en fonction du niveau d'abstraction de notre modèle.
- L'impact du modèle utilisé sur le temps de calcul.

3.2 Protocole

Nous chercherons à estimer le temps de calcul nécessaire en fonction de la précision de notre modèle. On récupère également la visibilité calculée de notre cible.

Le modèle pour représenter notre cible diffère entre le raycast et le procédé basé sur le rendu. En effet, nous approchons notre modèle par des boîtes englobantes alignées sur les axes pour le raycast alors que nous utilisons plusieurs niveaux de détails dans l'autre cas.

La scène étudiée est statique et composée de notre cible, le personnage Sintel du court métrage du même nom de la fondation blender, et d'une boîte bleue

occultant son bras gauche. Vous pouvez voire un rendu de la scène à la figure 5. Nous calculerons la visibilité à partir d'un unique point de vue.

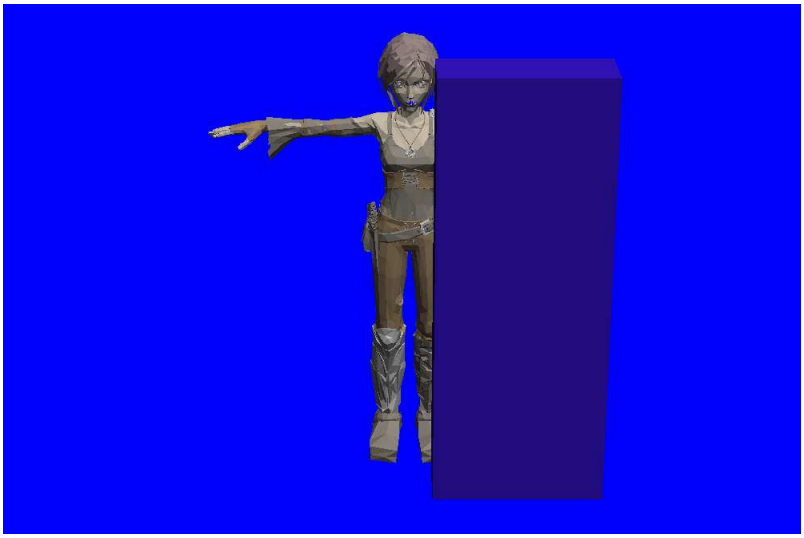


FIG. 5 – Scène utilisée pour les expériences.

Les expériences sont réalisées sur un ordinateur équipé d'une carte graphique Nvidia Quattor FX 1800, d'un microprocesseur intel Xeon W3550[6], et 5,8 Gio de mémoire RAM. Il tourne sous le système d'exploitation Fedora linux 16. Les bibliothèques utilisées pour l'implémentation sont *Ogre3D*[2] pour le rendu et *Bullet*[1] pour le raycast.

3.3 Le raycast

Il est simple à implémenter et est utilisé dans le domaine du jeu vidéo. C'est un procédé *Line-space* qui consiste a lancer un rayon du point de vue considéré vers la cible, abstrait par un point dans l'espace. Si ce rayon est intercepté avant d'avoir atteint son objectif, alors la cible est occultée. Plus la visibilité nécessite d'être précise, plus il faudra lancer de rayons. On remarquera également que la complexité de calcul s'intensifie avec le nombre de cibles.

Ce procédé pâtit d'un temps de calcul dû au nombre de tests d'intersection qu'il faut réaliser pour obtenir le polygone le plus proche. Dans une implémentation naïve, l'ensemble des polygones de la scène serait testé et le plus proche

[6]Processeur quadri-cœurs cadencés à 3,07 GHz

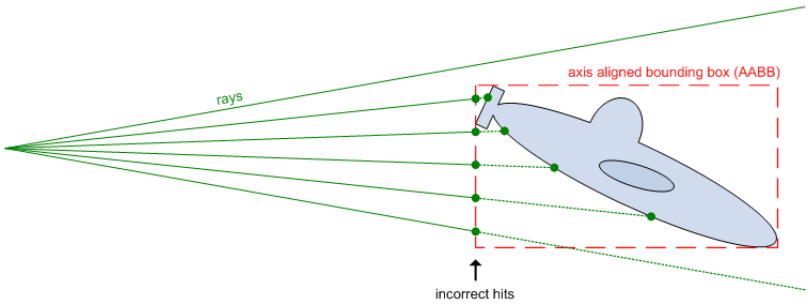


FIG. 6 – Lancer de rayons sur une boîte englobante alignée sur les axes. On peux remarquer l'apparition de rayons coupés qui ne devraient pas l'être. Image prise sur le site d'Ogre3D[2]

retourné comme résultat. La complexité serait donc en $O(n \times m)$, n et m étant respectivement le nombre de polygones et le nombre de rayons lancés.

Afin de réduire ce temps de calcul, il existe des structures de données *accélératrices* qui permettent de réduire le nombre de tests d'intersection à effectuer. On arrive ainsi à atteindre avec une structure en forme d'arbre, une complexité moyenne en $O(m \times \log n)$. En effet, l'environnement étant divisé en sous-volumes, il suffit juste de calculer les intersections avec les polygones contenus dans les sous-volumes. Le temps d'exécution peut encore être réduit en simplifiant au maximum la scène via des boîtes englobantes au prix de la précision comme indiqué sur la figure 3.3. Ainsi la complexité de dépend plus du nombre de polygones présents dans la scène.

le raycast est hautement parallélisable et peut donc profiter de la puissance de calcul des cartes accélératrices 3D avec des technologies comme OpenCL ou CUDA de Nvidia.

Expériences

Nous avons abstrait la cible par une boîte englobante alignée sur les axes. Ensuite nous avons déterminé la visibilité de cette boîte en lançant plusieurs rayons sur les faces visibles de la cible. Nous sommes à 0.16 ms par rayon. Donc si nous nous limitons à 4 ms pour faire le calcule nous ne pouvons pas lancer plus de 25 rayons.

3.4 Procédés basés sur le rendu

Les techniques basées sur le rendu sont plus complexe à mettre en œuvre mais induisent des résultats plus précis pour un temps de calcul acceptable. Le procédé que l'on va étudier ici est basé sur les *occlusions queries* et est utilisé dans le modèle présenté à la section 4.

Tout d'abord on initialise la fenêtre de rendu et la configuration de la caméra associé en fonction de la boîte englobante de notre cible. Ainsi si la caméra ne cadre pas la cible au complet, la visibilité ne sera pas considéré comme complète. Ensuite, on initialise la carte de profondeur en rendant uniquement le sujet, qui peut être un ensemble d'objets. A partir de là, on lance la requête pour obtenir le nombre de pixels visibles sans occultant. L'initialisation de la carte de profondeur est indispensable puisque l'*occlusion query* compte le nombre de pixels visibles en comptant le nombre de fois que le Z-buffer est mis à jour. On évite ainsi de compter plusieurs fois le même pixel comme visible ou un pixel occulté. Ensuite, on rends la scène avec tout les occultants potentiels. Pour finir, on relance la requête pour obtenir le nombre de pixels visibles avec les occultants. On obtient ainsi le rapport de visibilité de notre sujet, un nombre entre 0 et 1.

A des fins d'optimisations, on désactive toutes les mémoires tampons autre que le Z-buffer, qui est le seul utiliser dans ce cas. Afin d'éviter, la famine du GPU[7] pendant que le processeur est occupé, on récupère le résultat de la requête au dernier moment, c'est à dire juste avant la requête suivante. En effet cela implique un dialogue entre le GPU et le CPU qui est couteux en temps et cela laisse le temps au GPU de faire l'ensemble des calculs nécessaires.

Expériences

Sept niveaux de détails différents on été établis pour la cible, allant de 10 000 à 1 000 000 de faces. Nous considérons également la taille de la fenêtre de rendu, nous l'avons fait varié d'une granularité de 8x8 à 2048x2048.

Nous remarquerons que la précision de l'estimation de la visibilité dépend peu de la taille de la fenêtre de rendu ainsi que du nombre de polygones de notre modèle comme le montre les courbes des figures 10 et 9. En effet elle varie de 3% au maximum dans l'ensemble des tests que nous avons réalisés. Ainsi nous pouvons réduire l'impact qu'apporte la taille de la fenêtre de rendu et le nombre de polygones (cf. figures 7 et 8) sur le temps de calcul en les minimisant.

[7]Graphical Processor Units : Unités de calcul de la carte accélératrice 3D

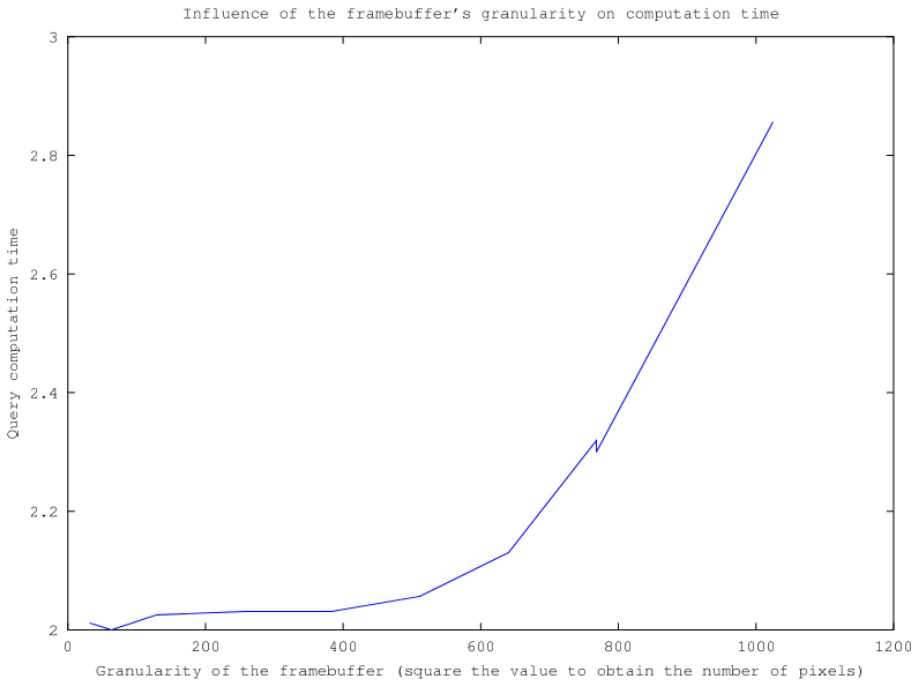


FIG. 7 – Temps de calcul en fonction de la granularité de la fenêtre de rendu. Les valeurs en abscisse doivent être élevées au carré pour obtenir le nombre de pixels de la fenêtre de rendu.

3.5 Interprétation

Le raycast est suffisant dans les cas où il suffit de lancer un nombre limité de rayon, obtenir la visibilité d'un unique point ou d'un objet petit dans le rendu finale par exemple. Cependant, lorsqu'il est nécessaire de lancer un nombre conséquent de rayons, le temps de calcul devient pénalisant pour une application temps réelle. En effet, obtenir la visibilité de la cible abstrait par une boîte englobante alignée sur les axes avec 20 rayons est équivalents en temps à faire un rendu sur une fenêtre de 200 par 200 pixels pour une précision plus faible. De plus, si on veut augmenter la précision, on doit augmenter le nombre de boîtes et les mettres à jour en fonction de l'animation de la cible. Il faut également noté que le raycast est insensible au problème d'échelle contrairement aux procédés basés sur le rendu qui peut être soumit aux erreurs de précisions dû à l'étalonnage du Z-buffer et peut donc provoquer un Z-fight[8] si la cible est très loin du point de vue que l'on considère.

[8]Erreur de précision du à un sous-échantillonnage du Z-buffer pour deux plans de profondeur très proche.

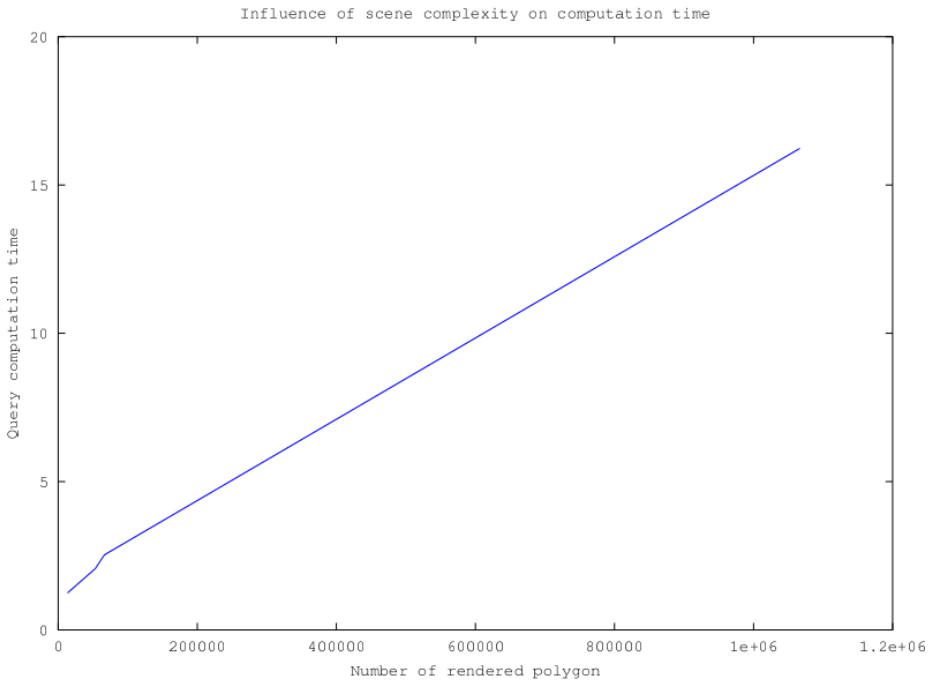


FIG. 8 – Temps de calcul en fonction du nombre de polygone présent dans la scène.

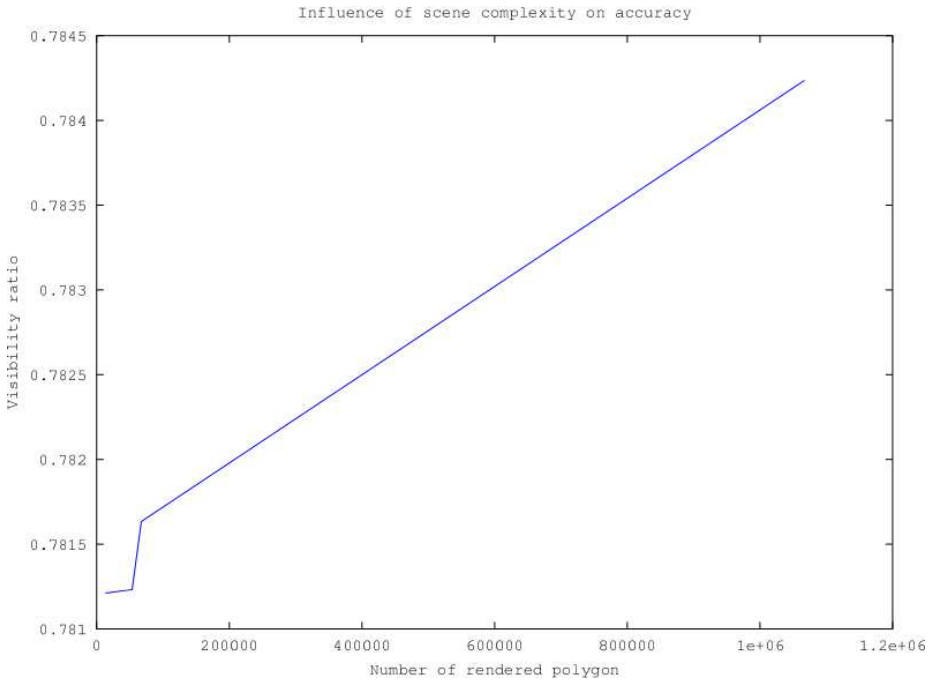


FIG. 9 – Visibilité obtenue en fonction du niveau de détails de la cible.

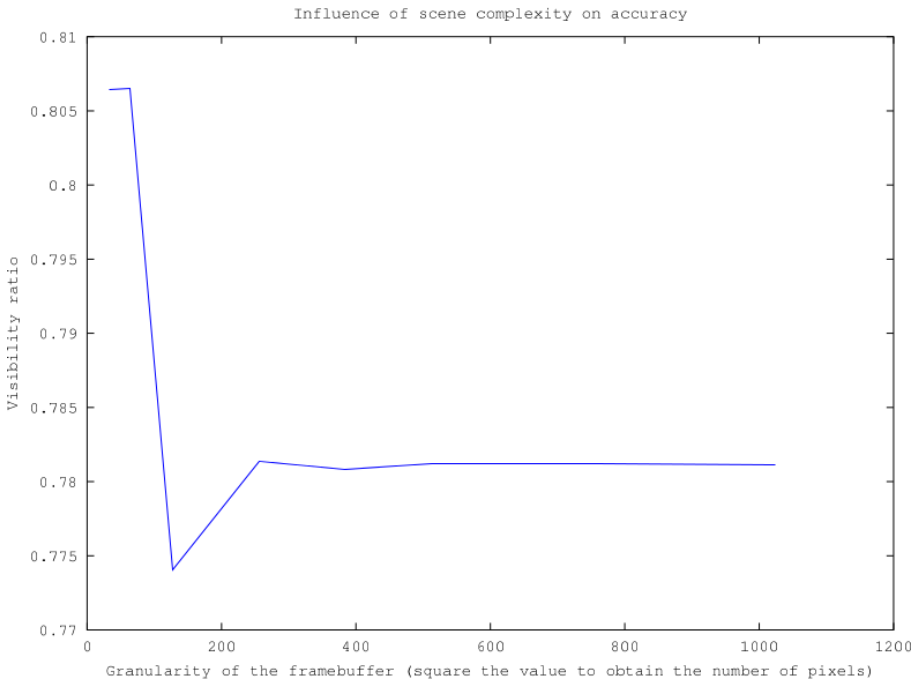


FIG. 10 – Visibilité obtenue en fonction de la granularité de la fenêtre de rendu. Les valeurs en abscisse doivent être élevées au carré pour obtenir le nombre de pixels de la fenêtre de rendu.

4 Notre contribution : le CubeFrustum

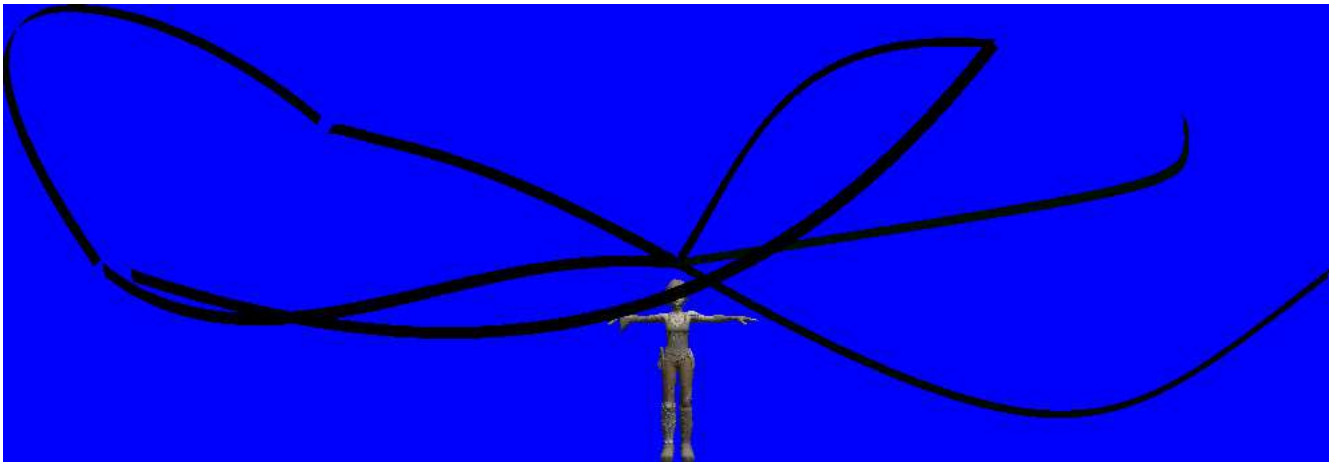


FIG. 11 – Représentation du graphe autour de la cible en noir.

Le but de ce modèle est de restreindre les mouvements possibles de la caméra

à des chemins prédéfinis pour une application de suivi d'objet mobile. Pendant l'exécution si le sujet est occulté, il suffira de choisir un des chemins prédéfinis offrant la meilleure visibilité de la cible tout au longs du trajet.

Dans un premier temps nous allons proposer un premier modèle pour vérifier la faisabilité de cette approche. Cette méthode offre cependant trop de limitation et donc une ouverture sur les possibilités d'amélioration vous sera présentées.

4.1 Le Modèle

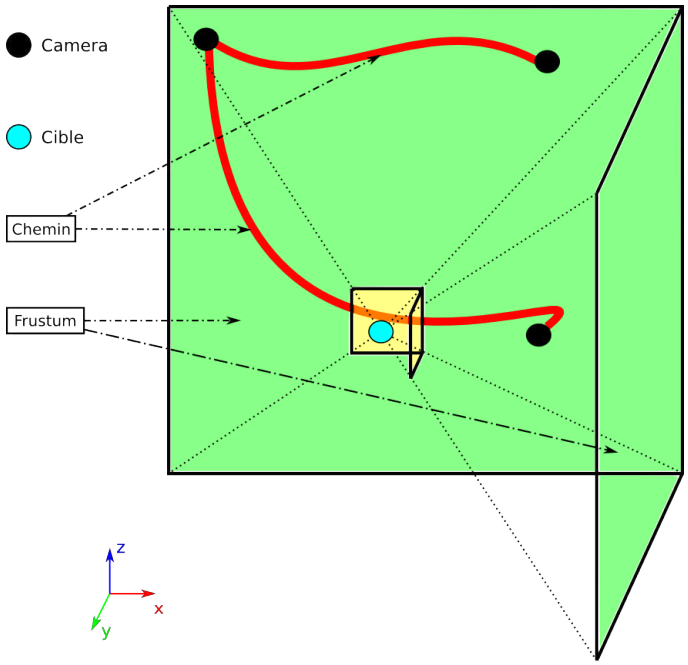


FIG. 12 – Représentation du CubeFrustum. Les deux rectangles jaunes sont les deux plans proches de la caméra alors que les deux rectangles vert sont les deux plan éloignés de la caméra. Ces deux plan permettent d'étalonner le Z-buffer.

L'ensemble des configurations de cameras autour d'une cible, ainsi que les chemins les reliant est attaché au nœud de la cible dans le graphe de scène (cf. figure 13). Ainsi, tout les points de vues resteront cohérents par rapport au sujet puisqu'ils sont positionnés et orientés par rapport à ce dernier.

Afin de déterminer la visibilité de tout les chemins du graphe, on met en place quatre caméras positionnées sur le point dont on veut obtenir la visibilité. Chacune

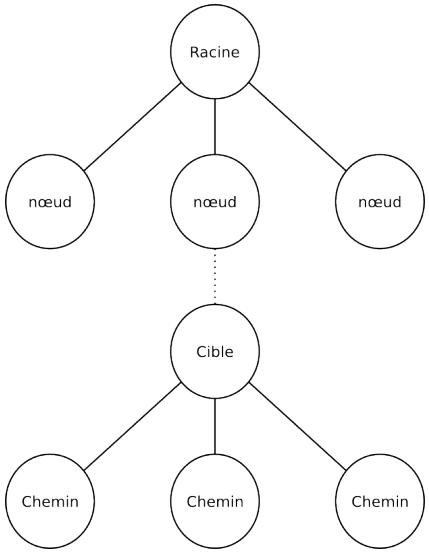


FIG. 13 – Graphe de scène associé au CubeFrustum. Les chemins sont les fils du noœud associé à notre cible.

d'elle est dirigée vers un des *quatre points cardinaux*[9] comme indiqué sur la figure 12. Ainsi nous avons une vue sur tout les chemins possibles autour de la cible. Dans le scénario où une caméra passe *au dessus* de la cible on peut ajouté une caméra regardant *vers le haut*. Un procédé de rendu similaire à celui indiqué dans la section 3.4 est utilisé pour obtenir les visibilités.

4.2 L'algorithme

Nous proposons ici une première approche naïve qui a certes beaucoup de limites mais qui introduit bien le fonctionnement du modèle vue dans la sous-section précédente.

Dans un premier temps l'algorithme initialise la structure de données. Les chemins sont représentés par des bandes qui seront attachées au nœud de scène de la cible, comme montré sur la figure 13. À Chaque chemin est associé une couleur RGBA[10]. Elle est codé sur un entier 32 bits avec un seul bit à 1. Ainsi on pourra s'en servir comme d'un masque et cela nous limite à 32 chemins possibles et donc 32 configurations de caméra maximum. On utilise également le *stencil buffer* pour

[9]Dans le cas d'un repère avec l'axe z vers le haut, l'est correspond à l'axe x, le nord à l'axe y, et vous pouvez en déduire l'ouest et le sud

[10]Rouge, vert, bleu et canal alpha

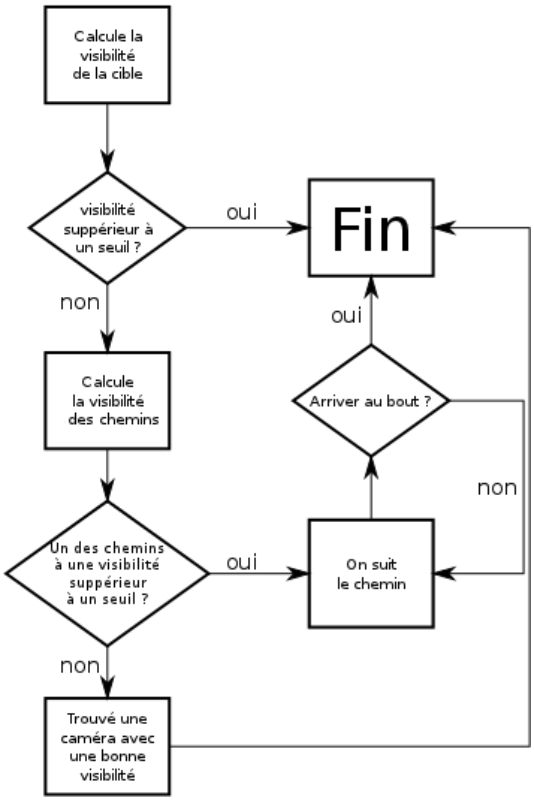


Fig. 14 – Organigramme de programmation du CubeFrustum

gérer la profondeur entre des chemins qui se superposent sur le même plan z. Pour cela on utilise le numéro du bit du chemin le plus loin. On met aussi en place quatre fenêtres de rendu regardant vers les *quatre points cardinaux* comme indiqué dans la figure 12 et avec une ouverture horizontale de 45 degrés. Ainsi on englobe l'ensemble de l'environnement de la cible. Si nécessaire, on peut ajouter une fenêtre de rendu qui regardera *vers le haut* pour le cas ou la caméra passe par dessus le sujet.

L'algorithme procède de la façon qui suit. Dans un premier tant on calcul la visibilité de la cible dans la configuration courante. Si elle est au dessus d'un certain seuil on change rien. Sinon, on détermine la visibilité de toutes les chemins partant de la caméra courante. Le chemin ayant la meilleur visibilité et au dessus d'un certain seuil est choisit et suivi. Sinon aucun des chemin n'est probant, l'algorithme calcule la visibilité de toutes les caméras disponible et dès que l'on obtient un résultat satisfaisant, on *saute* sur la configuration correspondante. Vous trouverez un organigramme de programmation récapitulatif dans la figure 14.

Le rendu pour obtenir la visibilité de chaque chemin se fait comme suit :

1. Tout d'abord on réalise un rendu de tous les chemins sans les occultants.
2. Ensuite on compte le nombre de pixels de chaque chemin pour chaque fenêtre de rendu.
3. Ensuite on rend tout les occultants en blanc (tout les bits à 1), couleur qui leur est réservé.
4. On réitère 2
5. On calcule de rapport de visibilité pour chaque chemin.

l'implémentation n'est actuellement pas encore fonctionnel.

4.3 Améliorations possibles

Nous allons maintenant proposer quelques solutions pour répondre aux limites qui nous sont imposées par la méthode que nous venons de voir. En effet, nous ne pouvons prédéterminer que 32 chemins différents, il ne peut pas y avoir plus de deux chemins passant par le même plan $z = \alpha$[11] et si aucun des chemins partant de la caméra courante ne peut être enprûnté, nous devons calculer la visibilité de la cible à travers chaque caméra et dans le pire des cas on les fait toutes.

Les quatre rendus effectués par le CubeFrustum peuvent être utilisés pour calculé la visibilité de la cible par rapport à une configuration de caméra. En effet, le fait de la considérer comme un *carré* regardant la cible permet d'en obtenir une visibilité. Et donc nous appliquons le même procédé que pour les chemins. Cependant, sachant que dans l'état actuelle de l'implémentation, il n'est possible de déterminer la visibilité que de 32 chemin, l'ajout de cette fonctionnalité ne permettra de prendre en considération que 16 chemins et 16 caméra. Ceci limite encore plus les configurations possibles. Cependant nous pouvons nous défaire de cette contrainte par la l'amélioration que nous allons présenter ensuite.

Le premier rendu fait dans le CubeFrustum permet d'obtenir la visibilité des chemins sans occultations pour pouvoir calculer le rendu ensuite. Ce rendu sera toujours le même à chaque requête d'où l'idée de le faire en précalcul et le mettre en mémoire. Cependant n'oublions pas que nous avons besoins de l'image générée et de la carte de profondeur associé. Sinon le Z-test ne peut être réalisé et un occultant qui ne le serait pas peut être considéré comme tel. Il est impératif donc

[11] Pour z axe vers le haut

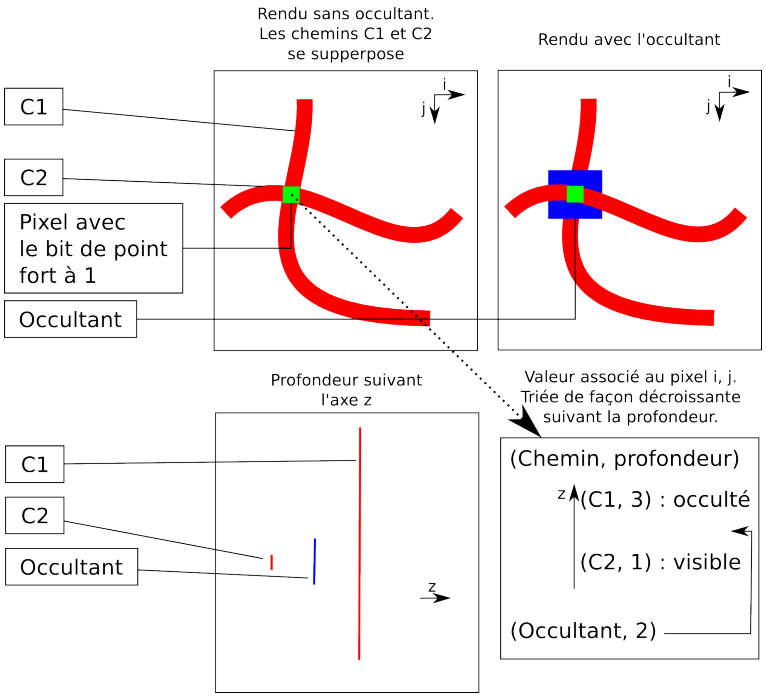


FIG. 15 – Codage de la profondeur pour les chemins se superposant.

d'associé la carte de profondeur avec l'image. Ainsi, à chaque début de requête, il suffit d'injecter l'image et la carte de profondeur dans le FBO[12] associé à notre fenêtre de rendu. Notons qu'il est également possible de mettre en mémoire la première requête pour obtenir le nombre de pixels visibles. En effet il n'est pas susceptible d'être modifier également vu qu'il n'y pas d'occulants.

Une autre modification envisageable serait de coder dans l'image finale les caméras et les chemins différemment. Plutôt que d'utiliser un seul bit sur les 32 disponibles pour coder une entité[13], il est possible de déterminer une couleur sur 31 bits pour chaque chemin, le bit de poids fort étant forcé à 0. Ainsi 2^{31} chemins ou caméra peuvent être pris en considération. On s'affranchit ainsi de la limite de 32 entités. De plus, dans les cas où plusieurs chemins passerait par le même plan $z = \alpha$, on peut utilisé un autre codage où le bit de poids fort est forcé à 1. La valeur ainsi codé serait mis en corrélation avec un tableau associatif reliant l'indice du pixel avec tout les chemins, ainsi que leur profondeur. Ainsi les chemins occultés pourraient être obtenue avec la carte de profondeur comme montré dans

[12]Frame Buffer Object : objet associé à la carte accélératrice 3D permettant de manipuler une fenêtre de rendu. Il est constitué d'une mémoire pour les couleurs, une pour la carte de profondeur et une pour le *stencil buffer*

[13]Ici on considérera une entité comme étant soit un chemin, soit une caméra

la figure 15. Il faudra noté cependant que la valeur hexadécimale 0xFFFFFFFF est interdite puisqu'elle est réservée à détecter l'occultation. Il faut également que la profondeur soit réinitialisée à la valeur maximale pour tout les pixels codant la profondeur. Ainsi, on peut récupérer la profondeur de l'occultant s'il est entre plusieurs entités. Cette amélioration n'est envisageable que si l'on utilise également celle présentée dans le paragraphe précédent puisque qu'elle implique de détecter les chemins s'entrecroisant et donc un plus long temps de calcul.

5 Conclusion

Nous avons proposé un nouveau modèle qui tire pleinement profit de la capacité de calcul des cartes accélératrices 3D en se basant principalement sur les mécanismes de rendu qu'elles implémentent matériellement. Cependant nous n'avons pas encore pris en considération la prédiction temporelle et ce serait une piste prometteuse à suivre.

Il serait également intéressant d'utiliser le CubeFrustum avec le procédé de Li et coll.[13] expliqué dans la section 2.4 pour déterminer le chemin à suivre au lieu de le faire via un parcours de graphe.

Références

[1] *Bullet*. `http://bulletphysics.org/wordpress/`.

[2] *Ogre3D*. `http://www.ogre3d.org`.

[3] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling : Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3), 2004.

[4] Jim Blinn. Where am i ? what am i looking at ? *IEEE Computer Graphics and Applications*, July 1988.

[5] R. Bohlin and L. E. Kavraki. A lazy probabilistic roadmap planner for single query path planning. *IEEE Int. Conf. on Robotics*, 2000.

[6] Paolo Burelli and Arnav Jhala. Dynamic artificial potential fields for autonomous camera control. *Association for the Advancement of Artificial Intelligence*, 2009.

[7] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, December 1974.

[8] Marc Christie, Jean-Marie Normand, and Patrick Olivier. Occlusion-free camera control for multiple targets. *ACM Transactions on Graphics*.

[9] Marc Christie, Patrick Olivier, and Jean-Marie Normand. Camera control in computer graphics. *Computer Graphics Forum*, 2008.

[10] Frédo Durand. *A Multidisciplinary Survey of Visibility*. PhD thesis, Université Joseph Fourier, 2000.

[11] Nicolas Halper, Ralph Helbing, and Thomas Strothotte. A camera engine for computer games : Managing the trade-off between constraint satisfaction and frame coherence. *EUROGRAPHICS*, 2001.

[12] F. Lamarche. Topoplan : a topological path planner for real time human navigation under floor and ceiling constraints. *Eurographics*, 2009.

[13] Tsai-Yen Li and Chung-Chiang Cheng. Real-time camera planning for avigation in virtual environements. *International Symposium on Smart Graphics*, 2008.

[14] C. Lino, M. Christie, F. Lamarche, G. Schofield, and P. Olivier. A real-time cinematography system for interactive 3d environments. *Eurographics*, 2010.

[15] Thomas Oskam, Robert W. Summer, Nils Thuerey, and Markus Gross. Visibility transition planning for dynamic camera control. *EUROGRAPHICS*, 2009.

[16] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*,

chapter 6. Addison Wesley, 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html.

[17] Peter Shirley, James Arvo, and Henrik Wann Jensen. *Monte Carlo Ray Tracing*. Siggraph, 2003. course 44.

[18] K. Shoemake. Arcball : a user interface for specifying three-dimentional orientation using a mouse. In *Proceedings of Graphics Interface '92*, pages 151–156, May 1992.

[19] Dave Shreiner and The Kronos OpenGL ARB Working Group. *OpenGL Programming Guide : The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. 7 edition, July 2009.

[20] Roy Thompson and Christopher J. Bowen. *Grammar of the Shot*. Focal Press, March 2009.

[21] Andrew Woo, Pierre Poulin, , and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graph- ics and Applications*, pages 13–32, 1990.