# Modélisation Tri-dimensionnelle Temps-réel et Distribuée

Clément Ménier[1,2] Jean-Sébastien Franco[1] Edmond Boyer[1] Bruno Raffin[2]
[1]GRAVIR - INRIA Rhône-Alpes, France [2]ID - INRIA Rhône-Alpes, France

## Résumé

*Cet article traite du problème de la modélisation 3D en temps-réel à partir d'images issues de plusieurs caméras. Les environnements comportant plusieurs caméras et PC deviennent de plus en plus courants, principalement grâce aux nouvelles technologies des caméras et aux très hautes performances des PC modernes. Cependant la plupart des applications de vision par ordinateur n'utilisent qu'un seul ou un petit nombre de PC et passent mal à l'échelle. La motivation de cet article est donc de proposer un cadre distribué permettant d'obtenir des modèles 3D précis en temps-réel avec un nombre variable de caméras, ceci via une utilisation optimisée de l'ensemble des machines disponibles. Nous nous intéressons particulièrement dans cet article aux méthodes calculant l'enveloppe visuelle à partir des silhouettes et la manière de distribuer leurs calculs sur un ensemble de PC. Notre contribution consiste en une stratégie de distribution s'appliquant à différentes méthodes de ce domaine et permettant de concevoir des applications temps-réel. Cette stratégie repose sur différents niveaux de parallélisation, de la répartition de tâches indépendantes à l'exécution concurrente permettant un contrôle précis à la fois sur la latence et le débit du système de modélisation. Nous détaillons aussi l'implémentation d'une telle stratégie pour des applications de modélisation d'enveloppes visuelles. En particulier, nous montrons que des modèles surfaciques précis peuvent être calculés en temps-réel avec uniquement du matériel standard. Nous donnons aussi des résultats sur des données de synthèse et en conditions réelles.*

## Mots Clés

Reconstruction 3D, Enveloppe Visuelle, Parallélisme, Distribution.

## Abstract

*This paper addresses the problem of real time 3D modeling from images with multiple cameras. Environments where multiple cameras and PCs are present are becoming usual, mainly due to new camera technologies and high computing power of modern PCs. However most applications in computer vision are based on a single, or few PCs for computations and do not scale. Our motivation in this paper is therefore to propose a distributed framework which allows to compute precise 3D models in real time with a variable number of cameras, this through an optimal use of the several PCs which are generally present. We focus in this paper on silhouette based modeling approaches and investigate how to efficiently partition the associated tasks over a set of PCs. Our contribution is a distribution scheme that applies to the different types of approaches in this field and allows for real time applications. Such a scheme relies on different accessible levels of parallelization, from individual task partitions to concurrent executions, yielding in turn controls on both latency and frame rate of the modeling system. We report on the application of the presented framework to visual hull modeling applications. In particular, we show that precise surface models can be computed in real time with standard components. Results with synthetic data and preliminary results in real contexts are presented.*

## 1 Introduction

Les évolutions technologiques récentes des caméras ont permis une large diffusion de l'acquisition temps-réel d'images numériques. De nos jours, n'importe quel ordinateur peut acquérir de telles images à des débits vidéo standard. Ceci permet de concevoir des systèmes complets d'acquisition en connectant simplement un ensemble de caméras et de PC sans avoir recours à du matériel spécifique. Cet intérêt grandit dans plusieurs domaines d'application utilisant plusieurs caméras et où un traitement temps-réel des informations est requis à des fins d'interactions ou de contrôle. Ces domaines incluent, par exemple, la *virtualisation* de scènes, la vidéo-surveillance ou encore l'interaction homme-machine. Cependant, alors que de nombreux travaux s'intéressent aux algorithmes dans le cas où un petit nombre de caméras sont connectées à un seul ou quelques PC, très peu d'efforts ont été investis dans les situations où un nombre plus grand de caméras et PC sont impliqués, environnements qui deviennent pourtant de plus en plus courants. De plus, la plupart des applications de vision utilisant plusieurs caméras connectées à plusieurs PC n'utilisent pas pleinement la puissance de calcul disponible et reposent généralement sur un calcul séquentiel sur une seule machine. Dans ce papier, nous traitons ces problèmes de passage à l'échelle et d'utilisation optimale des ressources en considérant des stratégies de parallélisation

pour la modélisation 3D. L'objectif est de proposer des solutions pratiques et scalables pour produire des modèles 3D précis en temps-réel en utilisant plusieurs caméras.

Un petit nombre seulement de systèmes distribués de modélisation 3D ont déjà été élaborés. L'institut de robotique CMU introduisit un dôme 3D avec environ $50$ caméras pour la virtualisation [17] ; le modèle 3D de la scène est obtenu via une approche stéréoscopique. D'autres systèmes ont aussi été proposés au CMU avec moins de caméras et basés sur une approche voxelique [7] ou un mélange des deux approches [6]. Cependant, ces systèmes effectuent les calculs soit *off-line*, soit en temps-réel mais avec uniquement un nombre restreint de caméras étant donné qu'aucune parallélisation n'est considérée pour effectuer les calculs de modélisation. Une autre catégorie d'approches temps-réel mais non parallèles utilise les cartes graphiques pour générer directement des images issues de nouveaux points de vue [15]. L'utilisation des cartes graphiques permet de grandement accélérer le rendu mais de tels systèmes reposent encore sur un seul PC pour les calculs et ne fournissent pas de modèles 3D explicites souvent requis par la plupart des applications. Davis *et al.* [4] présentent un système distribué pour l'enregistrement, l'accès et le traitement de flux vidéo sur une grappe de PC. Mais ces travaux mettent l'accent sur l'aspect base de données et ne considèrent pas des applications temps-réel. Des systèmes de modélisation parallèles et temps-réel ont été proposés pour gérer la modélisation voxelique [13, 2]. Les méthodes voxeliques produisent des modèles 3D discrets grâce à un découpage régulier de l'espace. Les cellules parallélipipédiques – les voxels – sont *sculptées* en fonction des informations issues des images [20, 9]. De telles méthodes peuvent être facilement parallélisées étant donné qu'une part importante des calculs est effectuée, de manière indépendante, par voxel. Les approches mentionnées utilisent cette propriété et obtiennent de bonnes performances. Cependant les méthodes voxeliques sont imprécises et coûteuses en temps de calcul. De plus les principes développés dans ces travaux ne se généralisent pas facilement à des méthodes de modélisation plus avancées. Dans cet article, nous essayons de développer des concepts de haut niveau pour les rendre applicables à la plupart des méthodes de modélisation. Plusieurs travaux intéressants concernent cette problématique de parallélisation pour la vision par ordinateur. Medioni *et al.* [10] proposent une stratégie multithreads pour améliorer la latence et le débit des traitements vidéo tels que la segmentation ou le tracking. Le projet Skipper [19] fournit un environnement de programmation parallèle pour le traitement d'images. Cependant ces travaux ne fournissent principalement qu'un environnement adapté de conception parallèle mais ne considèrent pas les aspects algorithmiques pourtant vitaux pour les applications de modélisation visées.

Dans cet article, nous présentons une architecture parallèle et temps-réel pour les algorithmes multi-caméras et son application à des applications de modélisation 3D, en particulier à des approches basées sur les silhouettes. Notre but est de fournir des solutions scalables en utilisant une stratégie de parallélisation. Cette stratégie est conçue pour être suffisamment générale pour s'appliquer à différents contextes tout en limitant l'effort de parallélisation. Notre contribution par rapport aux travaux présentés est double : d'une part nous étendons aux algorithmes multi-caméras les concepts de parallélisation déjà proposés, d'autre part nous montrons l'application de ces concepts à des approches basées sur les silhouettes et proposons des solutions pratiques scalables et surpassant les méthodes voxeliques.

L'organisation de l'article est la suivante. La section 2 introduit notre stratégie de distribution pour la parallélisation des traitements multi-caméras. Son application au calcul de l'enveloppe visuelle à partir d'images est présentée en section 3. La section 4 valide les principes proposés dans un contexte réel et fournit des preuves numériques à partir de données synthétiques.

# 2 Stratégie de distribution

Nous présentons dans cette section notre stratégie de parallélisation pour les algorithmes multi-vues. Celle-ci repose sur des techniques classiques de parallélisation. La simplicité de ces techniques permet de limiter l'effort nécessaire pour paralléliser un algorithme existant. Les résultats expérimentaux montrent que cette stratégie offre cependant de bonnes performances.

Soit $n$ le nombre de caméras et $m$ le nombre de *nœuds* (PCs). Nous supposons que chaque caméra est connectée à un nœud dédié à l'acquisition et au traitement local de l'image. Nous considérons que toutes les caméras génèrent des images à la même fréquence (*débit*). Nous appellerons une *trame* l'ensemble des $n$ images prises à un temps $t$. Nous supposons que $m > n$ ; les $p = m - n$ nœuds restants sont dédiés aux calculs. Les PC sont inter-connectés par un réseau standard. Accéder à une donnée située sur un nœud distant est beaucoup plus lent qu'accéder à une donnée locale. Nous n'utilisons aucun outil masquant cette disparité tel que de la mémoire virtuellement partagée. Ceux-ci mènent bien souvent à des performances moindres que des outils, tels que MPI [12], permettant à l'utilisateur de tenir compte de cette disparité et d'optimiser en conséquence les transferts de données. Comme nous le verrons par la suite, le traitement spécifique des transferts de données induit un effort relativement faible pour un gain significatif.

Nous mesurons, en tant que critère de performance, l'*accélération*, obtenu en divisant le temps d'exécution séquentielle par le temps d'exécution parallèle. L'efficacité de la parallélisation est d'autant plus grande que le facteur d'accélération est proche du nombre de processeurs. Pour les contraintes de temps-réel, nous mesurons le débit et la *latence*, c.-à-d. le temps pour traiter une unique trame.

La stratégie que nous proposons est basée sur deux niveaux distincts de parallélisation : une parallélisation *par flux* et une parallélisation *par trame*.

## 2.1 Parallélisation par flux

Les applications multi-caméras traitent une séquence de trames, appelée un *flux*. Une méthode classique pour accélérer des applications traitant des flux est d'utiliser un *pipeline*. L'application est scindée en une séquence d'étapes (cf. fig. 1(b)), chaque étape étant exécutée sur différents nœuds. Ceci permet à un nœud de traiter la trame $t$ pendant que le nœud de l'étape suivante du pipeline traite la trame $t-1$. Les différentes étapes sont naturellement déduites de la structure du programme. Chercher à reconcevoir l'algorithme en un pipeline plus long est assez coûteux et augmente la latence à cause des communications supplémentaires.

Une étape est *temps-indépendante* si elle n'utilise pas de cohérence temporelle, c.-à-d. que le traitement de la trame $t$ ne dépend pas des résultats liés aux trames précédentes. Ceci permet de dupliquer un processus appliqué de manière identique à toutes les trames sur différent nœuds, appelés *unités de calcul* (cf. fig. 1(c)). Une nouvelle trame $t$ peut être traitée dès qu'une des unités de calcul est disponible. Le nombre d'unités de calcul devrait être suffisamment grand pour éviter toute attente de traitement d'une trame. Adapter cette technique à une étape temps-dépendante peut s'avérer réalisable mais nécessite des techniques d'ordonnancement avancées et des communications supplémentaires.

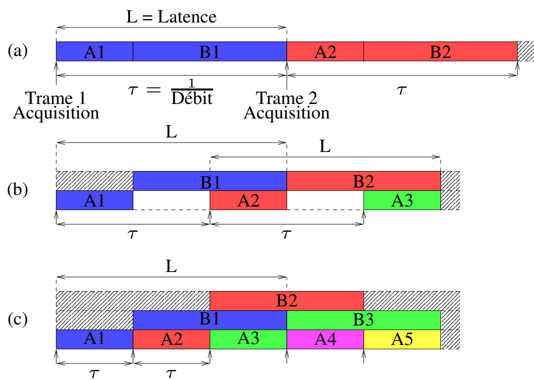


FIG. 1 – Différents niveaux de parallélisation proposés par notre stratégie. A et B sont les deux étapes de calcul de notre programme. A$t$ et B$t$ correspondent au traitement de la trame $t$. Chaque ligne correspond à un processeur. Les blocs colorés correspondent à l'exécution d'une tâche et les blocs blancs à des périodes d'inactivité. Le schéma (a) représente une exécution séquentielle, le schéma (b) une exécution avec un pipeline à 2 étapes. Le schéma (c) ajoute une seconde unité de calcul pour la seconde étape B du pipeline.

Cette stratégie de parallélisation peut être appliquée à l'approche classique voxelique. Les trames traversent deux étapes : soustraction de fond et "sculpture" des voxels. Nous obtenons ainsi un pipeline à 2 étapes. La première étape étant habituellement beaucoup plus rapide que la seconde, plusieurs unités de calcul peuvent être dédiées à cette seconde étape temps-indépendante.

## 2.2 Parallélisation par trame

Les techniques de distribution précédentes peuvent améliorer significativement le débit. Cependant, elles affectent la latence. Le pipeline introduit des communications supplémentaires qui augmentent la latence, réduisant ainsi la réactivité du système. Pour améliorer la latence, nous choisissons de réduire le temps de traitement d'une trame par une étape. Ceci peut être fait en parallélisant le travail effectué pour une unique trame sur plusieurs nœuds d'une unité de calcul (cf. fig. 2). Notre approche est basée sur le modèle classique *Bulk Synchronous Programming* [21] (BSP) qui propose un compromis entre performances et complexité d'implémentation. L'exécution est effectuée en une série de phases, chacune décomposée en un échange de données impliquant tous les nœuds suivi d'un calcul local sur chaque nœud séparément. Ce modèle simplifie la description d'algorithmes parallèles étant donné qu'il sépare les communications et les calculs. A partir de cette approche BSP, nous proposons un schéma en 3 phases pour paralléliser les opérations d'une unité de calcul :

- **Préparation des données** : la première phase (distribution des données d'entrée) consiste en l'envoi des données d'entrée aux nœuds les nécessitant. Chaque nœud effectue alors localement les opérations d'initialisation requis par la phase parallèle suivante.
- **Calcul parallèle** : en parallèle, chaque nœud exécute localement (pas de communications) une tâche différente qui lui a été assignée.
- **Calcul séquentiel** : les résultats partiels issus de la phase parallèle précédente sont réunis sur un nœud. Ce nœud effectue de manière séquentielle le reste des calculs qui n'ont pu être effectués lors de la phase parallèle. Selon les besoins de l'étape suivante dans le pipeline, ce calcul séquentiel peut être dupliqué sur plusieurs nœuds. Ceci permet de réduire le coût des communications pour envoyer les données à cette étape suivante.

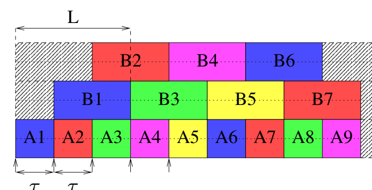


FIG. 2 – Parallélisation par trame (2 processeurs pour l'étape A et 4 processeurs pour l'étape B) : amélioration de la latence par rapport à la parallélisation par flux (cf. fig. 1.)

Bien que très simple ce modèle s'avère très général. Dans le pire des cas, tous les calculs sont effectués dans la dernière phase séquentielle. Cependant cela est évidemment inefficace. Nous montrons par la suite qu'il est possible d'obtenir une phase de calcul parallèle significativement

plus importante que les deux autres phases. Dans de telles situations, nous montrerons que cette technique permet d'atteindre des performances temps-réelles.

Nous pouvons illustrer l'application de cette technique à la parallélisation de la méthode voxelique décrite par Arita *et al.* [2] :

– Préparation des données : initialisation de l'espace des voxels.
– Calcul parallèle : pour chaque image, calcul du cône visuel dans l'espace voxelique.
– Calcul séquentiel : rassemblement des cônes visuels sur un nœud et calcul de leur intersection.

Les résultats présentés par Arita *et al.* montrent que cette technique donne de bonnes performances. Généralement, notre technique en 3 phases ne fournit pas une parallélisation optimale. De nombreuses optimisations peuvent encore être apportées. Sur cet exemple voxelique, Borovikov *et al.* ont décrit une optimisation algorithmique [4] pour le calcul de l'intersection des cônes visuels d'une manière plus complexe que notre modèle en 3 phases ne peut représenter. Cependant nous montrons dans ce papier que la stratégie présentée offre une méthode simple et efficace pour la parallélisation par flux et par trame de contraintes temps-réelles.

# 3 Modélisation à partir de silhouettes

Attardons-nous à présent sur les différentes techniques de reconstruction de l'enveloppe visuelle à partir de silhouettes et sur les stratégies de parallélisation possibles pour les rendre temps-réelles. Nous nous focalisons sur les techniques à base d'enveloppe visuelle, car celles-ci sont devenues très utilisées en raison de leur simplicité et rapidité. Rappelons à cette occasion que l'enveloppe visuelle est une approximation simple des objets de la scène, qui est définie en tant que la forme maximale cohérente avec les silhouettes de ces objets obtenues dans les images [14]. Un certain nombre d'algorithmes ont été proposés pour calculer cette enveloppe visuelle. Certaines approches utilisent un partitionnement discret de l'espace en voxels. De telles approches, dites *volumiques*, sont simples à implémenter et peuvent s'avérer très rapides. Nous avons étudié la parallélisation de telles méthodes dans la section précédente. Dans cette section, nous nous pencherons donc sur les méthodes plus récentes et proposerons une application possible des différentes stratégies de parallélisation possibles sur celles-ci.

En l'occurrence, une catégorie récente de méthodes de reconstruction de l'enveloppe visuelle, dites *surfaciques*, sont dédiées au recouvrement de la surface de l'enveloppe visuelle, en tant que polyèdre [3, 16]. Certaines donnent des garanties topologiques supplémentaires et des calculs plus simples, comparées aux approches volumiques. Une approche hybride intéressante existe aussi, qui combine les avantages des approches volumiques et surfaciques [5]. Bien que ces méthodes produisent des modèles précis en un temps restreint, elles restent encore trop lentes pour des calculs temps réel dans une configuration lourde (10 caméras ou plus). Cela en fait des candidats intéressants pour une parallélisation : nous montrerons dans ce contexte que le parallélisme est un outil qui rend accessible pour des algorithmes relativement rapides la possibilité du temps réel dur, à 30 images par seconde ou plus.

## 3.1 Résumé des méthodes de reconstruction

Pour offrir une vision large de la parallélisation d'approche basée sur des silhouettes, nous nous focaliserons sur deux des approches les plus récentes, l'approche *hybride* [5], qui offre un compromis robuste entre approches volumique et surfacique, et une méthode de reconstruction surfacique, l'*approche surfacique exacte* [11]. La figure 3 donne une vision globale des deux méthodes. Rappelons le contexte de telles méthodes : $n$ caméras calibrées sont utilisées pour l'acquisition de $n$ vues d'un objet ; une méthode standard de soustraction de fond permet l'extraction de silhouettes binaires pour chaque image. Les contours de telles silhouettes sont vectorisés afin d'obtenir une représentation de chaque silhouette sous la forme d'un polygone 2D. Cette représentation discrète des silhouettes induit une représentation discrète polyédrique de l'enveloppe visuelle. La méthode hybride fournit une très bonne approximation de ce polyèdre. La méthode surfacique exacte le calcule exactement.

Dans les deux cas la reconstruction a lieu en trois étapes, comme illustré dans la figure 3. La première étape, qui est commune aux deux méthodes, calcule un sous-ensemble initial de la géométrie de l'enveloppe visuelle, les *segments de vue*. Ce sont des segments situés sur les lignes de vue de chaque point 2D du polygone de la silhouette de l'objet (les détails suivront en section 3.2). Le but commun de la deuxième étape est de calculer une représentation intermédiaire qui contient implicitement la surface de l'enveloppe visuelle. La méthode hybride partitionne l'espace en cellules convexes, qui peuvent facilement être éliminées si elles se projettent en dehors d'une silhouette. La méthode exacte, en revanche, calcule toutes les intersections de cônes définissant l'enveloppe visuelle. Au final, la troisième étape se charge d'identifier l'information surfacique sous-jacente à la représentation intermédiaire. Nous allons préciser le fonctionnement de ces méthodes dans les sections suivantes.

Notons qu'une première possibilité de parallélisation s'offre à nous une fois ces trois étapes identifiées : en effet chacune de ses étapes conceptuelles peut être identifiée à un étage d'un pipeline, dans le cadre d'un environnement multi-processeur. Ceci permet notamment d'augmenter le débit de traitement des images et donc la fréquence à laquelle on peut générer des modèles géométriques. De plus nous pouvons utiliser plusieurs unités de calcul, chaque étape étant temps-indépendante. Nous présentons les détails ci-dessous.

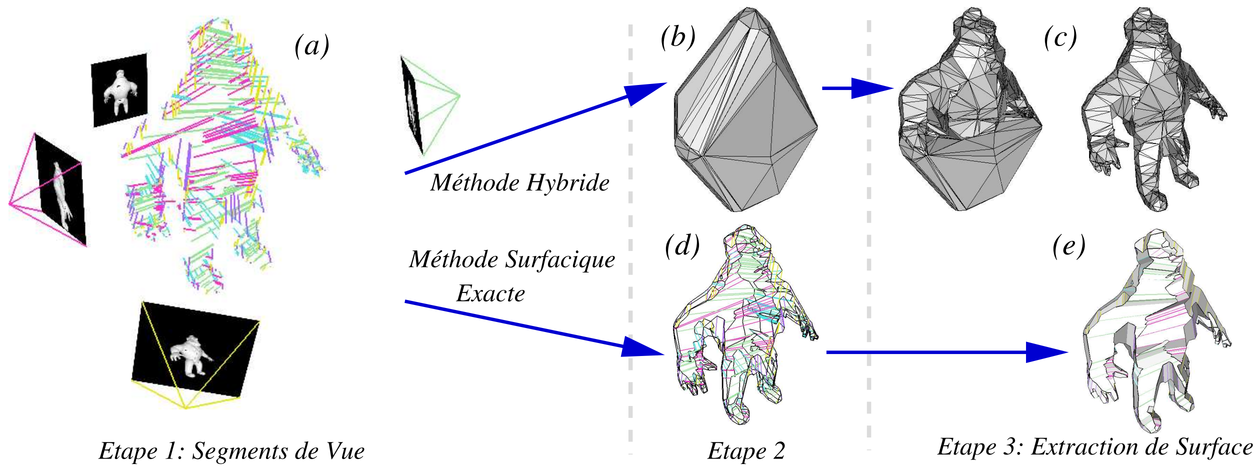


FIG. 3 – Schéma illustrant le déroulement des différentes techniques de reconstruction de l'enveloppe visuelle que nous parallélisons. (a) Des vues de l'objet sont acquises, la silhouette de l'objet identifiée, son contour vectorisé, et les segments de vue calculés à partir des différents points de cette discrétisation. (b) La méthode hybride calcule une triangulation de Delaunay de l'espace à partir des points extrêmes des segments de vue. (c) Chaque tétraèdre est éliminé s'il se projette en dehors d'une silhouette, ce qui permet d'obtenir un modèle géométrique de l'enveloppe visuelle. (d) La méthode surfacique exacte calcule la géométrie des intersections de cônes faisant partie de l'enveloppe visuelle, afin de finir le calcul du maillage de l'enveloppe visuelle. (e) Les facettes sont extraites de cette dernière représentation ce qui permet de finaliser le modèle de l'enveloppe visuelle.

## 3.2 Calcul des segments de vue

Nous décrivons maintenant le calcul des segments de vue en chaque point 2D du polygone de la silhouette, qui est l'étape initiale commune des méthodes présentées (voir figure 3).
Les *segments de vue* sont définis par des intervalles le long des lignes de vue. Ils correspondent à la contribution d'une ligne de vue à la surface de l'enveloppe visuelle et sont donc associés à des points 2D situés sur le contour d'une silhouette. Ils sont obtenus en calculant l'ensemble des intervalles le long d'une ligne de vue dont les points se projettent à l'intérieur de toutes les silhouettes (voir figure 4).

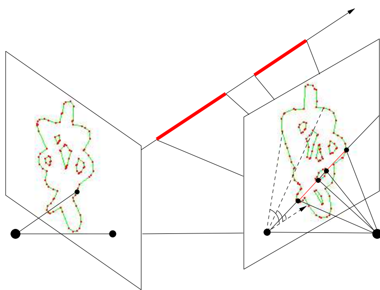

FIG. 4 – Segments de vue (en gras) le long d'une ligne de vue. Leur calcul peut être accéléré en triant les lignes par rapport à l'angle directeur des lignes épipolaires correspondantes dans les images.

Cet algorithme donne une grande liberté pour le parallélisme, car il consiste à calculer de nombreux résultats partiels et indépendants. En effet chaque ligne de vue peut être traitée indépendamment des autres, avec pour seul prérequis la disponibilité de toute l'information silhouette des images sur chaque noeud du calcul. Une stratégie efficace de parallélisation par trame peut donc être obtenue en partitionnant toutes les lignes de vues de toutes les images en $p$ ensembles durant la phase de préparation des données, puis en distribuant chaque ensemble à l'un des $p$ neouds prévus pour le calcul pendant la phase de calcul parallèle. Il est important de bien équilibrer la charge entre les noeuds, pour éviter au maximum le temps passé à attendre le noeud le plus lent. Construire des ensembles de lignes de vue de cardinalité identique est une heuristique qui s'avère efficace. Notons que ce traitement impose la diffusion à tous les noeuds de toute l'information des contours des $n$ silhouettes. La finalisation de la tâche consiste simplement à réunir l'ensemble des résultats partiels sur chaque noeud qui en aura besoin à l'étape suivante, pendant la phase de calcul séquentiel.
Grâce à cette stratégie de parallélisation nous atteignons des gains en vitesse d'un facteur 8 avec 10 noeuds pour cette étape de l'algorithme. C'est un très bon résultat, qui n'a nécessité qu'une adaptation succincte du code séquentiel initial. Des accélérations encore meilleures peuvent être réalisées, mais au prix d'une complexité de mise en œuvre substantielle.

## 3.3 Approche hybride distribuée

Nous décrivons ici la parallélisation de l'approche hybride [5]. Après avoir calculé les segments de vue, la méthode hybride utilise la triangulation de Delaunay des points extrêmes de ces segments, ce qui en constitue la deuxième étape (voir figure 3). L'union de ces tétraèdres définit l'enveloppe convexe de cet ensemble de points : à ce titre il faut sculpter cette représentation et donc éliminer des tétraèdres pour obtenir une bonne approximation de l'enveloppe visuelle. A la suite de la triangulation l'espace se trouve discrétisé en un ensemble de cellules convexes d'une forme plus générale que de simple voxels, mais qui peuvent être éliminés avec le même type de vé-

rifications de cohérence avec les silhouettes que pour les méthodes voxeliques ordinaires, telles celles proposées par Cheung et Kanade [7]. De telles vérifications sont utilisées par la troisième étape de l'algorithme pour déterminer quel est l'ensemble de tétraèdres appartenant à l'enveloppe visuelle, et ceux appartenant à son complémentaire. Les polygones de la surface du polyèdre sont alors extraits de ce modèle en identifiant simplement les triangles situés à la frontière entre les deux régions de l'espace ainsi définies.

Malgré la simplicité relative de cet algorithme, la construction d'une version parallèle de celui-ci est non triviale. Le principal obstacle est la triangulation de Delaunay, qui génère de nombreux résultats certes partiels mais soumis à des contraintes globales. Certaines possibilités pour paralléliser cet algorithme ont été explorées [8], autour de l'idée de diviser l'espace en sous-régions où l'on peut calculer des sous-résultats cohérents. Cette idée très générale peut être appliquée à beaucoup d'algorithmes de vision. Mais la difficulté dans ce cadre se concentre en général dans l'identification des dépendances entre régions et la fusion des résultats partiels. Dans le cas de la triangulation de Delaunay, un programmeur passerait beaucoup de temps à réimplémenter et adapter l'algorithme initial. La complexité de mise en oeuvre se trouve alors décuplée. Dans de telles conditions, il n'est pas toujours possible ou rentable de passer le temps nécessaire à réaliser une version parallèle. La parallélisation par flux peut néanmoins être utilisée pour améliorer le débit de traitement global du système. Mais nous verrons au chapitre suivant un cas où il est bénéfique de s'attaquer au problème des dépendances de calculs entre régions.

Nous avons tout de même une autre possibilité de parallélisation pour la troisième étape. Celle-ci consiste en effet en l'élimination des tétraèdres incohérents, une tâche qui peut être effectuée de manière indépendante pour chaque tétraèdre. Le seul prérequis est la disponibilité de la totalité des silhouettes sur tous les noeuds du calcul. La phase d'exécution séquentielle de cette troisième étape consistera simplement à réunir toute l'information des tétraèdres éliminés sur un seul noeud, pour finalement extraire les triangles de la surface de l'enveloppe visuelle comme précédemment expliqué. Il est possible de réaliser des accélérations par 9,5 pour 10 noeuds pour cette étape.

## 3.4 Approche surfacique exacte distribuée

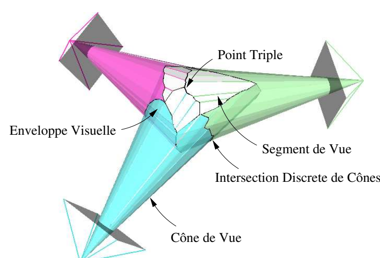


FIG. 5 – Enveloppe visuelle d'une sphère obtenue avec 3 vues.

Nous décrirons brièvement dans cette partie l'application de notre stratégie de parallélisation à l'approche surfacique exacte [11] (dont le déroulement général est visible en figure 3). La figure 5 fournit une représentation de toutes les entités géométriques utiles.

Comme vu précédemment, les segments de vue fournissent un sous-ensemble initial de la géométrie de l'enveloppe visuelle. Cependant, l'enveloppe visuelle est encore incomplète : à la suite de l'étape 1 tous les segments de l'enveloppe visuelle n'ont pas encore été calculés. L'enveloppe visuelle est l'intersection de *cônes de vue*, volumes rétro-projetés à partir des silhouettes obtenues pour chaque vue. Pour obtenir une enveloppe visuelle complète, il faut aussi calculer les courbes d'intersection de cônes, qui font également partie de la géométrie de l'enveloppe visuelle. On peut observer que de telles courbes, qui sont linéaires par morceaux (et donc constituées de segments, dans notre cas discret), courent sur la surface de l'enveloppe visuelle, en connectant les points des segments de vue existants à des points d'un nouveau type, appelés *points triples*. Les points triples sont le lieu d'intersection des surfaces de trois cônes de vue. Comme tout autre point du maillage du polyèdre, un point triple est connecté à trois voisins sur la surface. En l'occurence, un point triple est le point de rencontre de trois courbes d'intersection de cône sur la surface.

En profitant de ces observations, l'approche surfacique exacte cherche à suivre et reconstituer ces courbes d'intersection de cônes de proche en proche. Pour chaque point $P$ déjà créé sur la surface, si un voisin sur le maillage parmi les trois est déjà connu, il est possible de calculer chacun des deux voisins manquants, en utilisant les informations qui ont permis le calcul de $P$. Il s'agit tout d'abord de calculer la direction des segments reliant $P$ à ses deux voisins manquants. Puis, il faut trouver où se trouvent les voisins eux-mêmes, c'est à dire de manière équivalente la longueur de ces segments. Il est possible de retrouver cette information en utilisant une nouvelle fois une contrainte inhérente à l'enveloppe visuelle : les segments en questions n'appartiennent à l'enveloppe visuelle que s'il se projettent entièrement à l'intérieur des silhouettes dans les images. Calculer les coordonnées des voisins de $P$ manquants consiste alors à trouver quels points permettent de satisfaire ces contraintes. Il suffit alors de recommencer le calcul avec les nouveaux points trouvés : il est ainsi possible de calculer récursivement la géométrie de toutes les courbes d'intersection de cônes, à partir des points initiaux des segments de vue. Une fois le maillage entier reconstruit de proche en proche, il ne reste plus qu'à identifier les facettes du polyèdre sur la surface orientée. Les contours de telles facettes peuvent être retrouvés dans le maillage en tournant à gauche à chaque point rencontré. Pour plus de détails, consulter [11].

Tout effort de parallélisation de cet algorithme sera probablement confronté aux problèmes, inhérents aux maillages, liés aux fortes dépendances spatiales. Pour permettre une exécution concurrente, nous choisissons de manière classique de partitionner l'espace en $p$ régions distinctes en

utilisant $p - 1$ plans parallèles, divisant ainsi l'espace en $p$ "tranches". La largeur des tranches est ajustée en attribuant un même nombre d'extrémités de segments de vue à chaque tranche pour équilibrer la charge de calcul. Étant donné que nous ne manipulons que des points et des segments, le coût de partition de ces primitives durant la phase de préparation des données est relativement faible. Ainsi un nœud dédié à cette seconde étape de la méthode surfacique exacte peut être programmé pour suivre les courbes d'intersection au sein de sa région désignée $\mathcal{R}_i$, jusqu'à ce que celles-ci traversent les frontières vers une autre région $\mathcal{R}_j$. Ce nœud s'arrête alors de traiter cette courbe, déléguant le calcul du reste de la courbe au nœud en charge de la région $\mathcal{R}_j$. Remarquons au passage que les dépendances entre régions sont faciles à identifier étant donné qu'elles n'apparaissent que pour des segments qui traversent un plan de partition.
Il est assez aisé d'identifier les trois phases de la parallélisation par trame. La préparation des données consiste à partitionner l'espace en régions, et à distribuer les primitives géométriques concernées à chaque région ; la phase de calcul parallèle consiste en le calcul de portions du maillage au sein de chaque région ; le calcul séquentiel réunit et fusionne les maillages partiels. Cette parallélisation s'avère efficace étant donné que nous obtenons un facteur d'accélération de 6 pour 10 noeuds avec notre implémentation, un résultat très bon compte tenu du problème des dépendances et du faible surcoût lié à cette parallélisation. Les mesures globales fournies dans la section suivante confirmeront l'intérêt de cette parallélisation, pour tout nombre de caméras.
Nous avons aussi parallélisé l'étape d'extraction de la surface : le maillage complet est diffusé à $p$ nœuds durant la phase de préparation des données, ces $p$ nœuds calculent ensuite un sous-ensemble de la surface (chaque facette du polyèdre est indépendante), et la phase séquentielle ne fait que collecter les différentes facettes calculées. Les accélérations obtenues sont de l'ordre de 7 pour 10 noeuds.

# 4 Implémentation et résultats expérimentaux

Dans cette section, nous détaillons des résultats expérimentaux obtenus avec l'implémentation des deux algorithmes précédents distribués grâce à notre stratégie de parallélisation. Nous obtenons des performances temps-réelles pour la modélisation 3D de haute qualité ; rappelons que dans le cadre de la seconde méthode le polyèdre calculé est exact par rapport aux silhouettes d'entrée. Des tests sur des données de synthèse démontrent que même avec un grand nombre de caméras de très bonnes performances peuvent être obtenues.
Notre grappe de 16 processeurs est composée de 8 PC bi-Xeon (2.66 GHz) connectés via un réseau Gigabit Ethernet. La latence est mesurée à partir du début du calcul des segments de vue. Notre implémentation utilise une librairie, FlowVR [1], dédiée aux applications interactives distribuées. La triangulation de Delaunay est calculée grâce à la librairie séquentielle Qhull [18]. Les résultats présentés ont été obtenus à partir de séries de 10 expériences.

## 4.1 Conditions temps-réelles

Notre environnement expérimental est composé de 4 caméras IEEE 1394 chacune connectée à un PC en charge de l'acquisition, soustraction de fond et vectorisation des silhouettes. Les images (640x480) sont acquises à 30 images par seconde. La scène est composée d'une personne (see fig. 6), un objet de complexité standard (150 sommets de contour par image).

FIG. 6 – Reconstruction temps-réelle d'une personne avec 4 caméras et l'approche surfacique exacte.

En utilisant un tel système pour la méthode hybride, nous arrivons à modéliser la scène en temps-réel à 30 trames par seconde en utilisant les 16 processeurs. Une unité de calcul parallélisée sur 4 nœuds est dédiée à la première étape de l'algorithme. 10 unités de calcul sont dédiées au calcul séquentiel de la triangulation de Delaunay. L'extraction de la surface est réalisée sur une seule unité de calcul parallélisée sur 2 processeurs. La latence mesurée se situe aux alentours de 400 ms, mais est limitée principalement par le temps d'exécution séquentielle de la triangulation qui peut atteindre 300 ms.
La méthode surfacique exacte s'avère plus efficace étant donné qu'une exécution temps-réelle (30 trames par seconde) est obtenue avec seulement 12 processeurs. Chaque étape a une seule unité de calcul, chacune étant parallélisée sur 4 processeurs. La latence mesurée est d'environ 100 ms. Cette faible latence et le débit temps-réel permettent d'utiliser cet algorithme pour des applications interactives. Des vidéos sont disponibles à `http://www.inrialpes.fr/movi/people/Franco/ORASIS05`

## 4.2 Validation pour un grand nombre de points de vue

Nous avons choisi de tester la scalabilité de nos algorithmes distribués sur des images de plusieurs points de vue générées à partir d'un modèle synthétique. Nous avons ainsi pu tester nos implémentations avec un nombre très importants de caméras. Nous nous intéressons ici au pro-

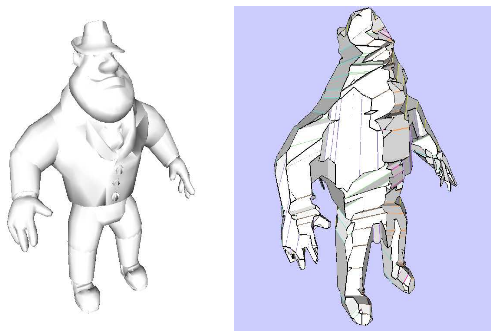

FIG. 7 – (gauche) Modèle d'origine. (droite) Reconstruction du modèle avec 12 points de vue.

blème de latence uniquement. Nous ne considérons que la méthode surfacique exacte étant donné que la méthode hybride est limitée en latence par la triangulation de Delaunay. Le problème du débit temps-réel n'est pas discuté : il peut en effet être résolu facilement en multipliant le nombre de nœuds assignés à la parallélisation flux-orientée.
Le modèle considéré est un personnage de synthèse présentant une complexité similaire à celle d'un personnage réel (environ 130 sommets de contour par image). La figure 8 présente les latences obtenues avec différents nombres de processeurs utilisés pour 16, 25 et 64 points de vue. La parallélisation de l'algorithme permet de réduire la latence de manière significative (par presque un ordre de grandeur). Avec 25 points de vue et 16 processeurs, la latence est inférieure à 200 ms, une latence acceptable pour l'interactivité du système.

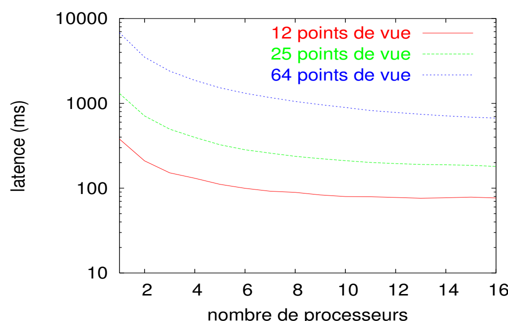


FIG. 8 – Graphe logarithmique des latences pour le personnage de synthèse.

La figure 9 présente les accélérations associées. Jusqu'à 9 processeurs pour 12 points de vue, 14 processeurs pour 25 points de vue, et plus de 16 processeurs pour 64 points de vue, le facteur d'accélération est supérieur à la moitié du nombre de processeurs utilisés. Au-delà, les facteurs d'accélération tendent à se stabiliser étant donné que la quantité de travail dans la phase de calcul parallèle diminue comparée à celle requise par les phases de préparation de données

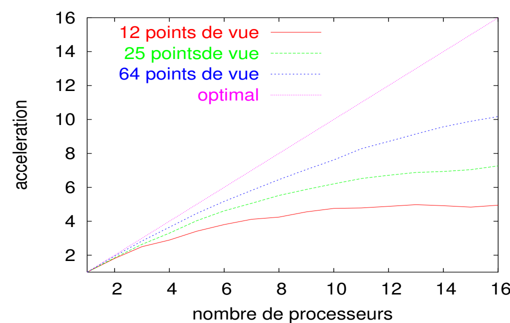


FIG. 9 – Accélérations pour le personnage de synthèse.

et de calcul séquentiel.

# 5 Conclusion

Nous avons présenté un système de modélisation 3D qui utilise le parallélisme pour atteindre une exécution temps-réelle avec un nombre flexible de caméras et de PC. Un tel système repose sur un environnement de distribution que nous avons formalisé. Cette stratégie de parallélisation vise les applications de vision manipulant plusieurs points de vue. Nous avons démontré son efficacité dans le cadre de la modélisation 3D à partir de silhouettes. Les enveloppes visuelles de haute qualité générées par ces algorithmes distribués peuvent être utilisées dans diverses applications, telles que la réalité virtuelle (cf fig. 10). Notre contribution principale par rapport aux travaux existant dans ce domaine consiste en la présentation de nouvelles implémentations parallélisées d'algorithmes de modélisation 3D ainsi que la formulation d'une stratégie de parallélisation pour les applications multi-vues. Des résultats sur des données réelles et de synthèse montrent que notre approche permet de rendre les systèmes de modélisation scalables et étend ainsi le potentiel de tels systèmes. Nous nous attachons actuellement à tester notre système dans une plateforme de réalité virtuelle comportant une douzaine de caméras. De plus nous cherchons à étudier l'utilisation de tels systèmes de modélisation à des fins d'interactions. Des vidéos préliminaires sont disponibles à : `http://www.inrialpes.fr/movi/people/Franco/ORASIS05`.

FIG. 10 – Deux nouvelles vues du modèle (enveloppe visuelle) de la figure 6 avec un texturage vue-dépendante.

# Références

[1] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR : a Middleware for Large Scale Virtual Reality Applications. In *Proceedings of Europar 2004*, Pisa, Italia, August 2004.

[2] D. Arita and R.-I. Taniguchi. RPV-II : A Stream-Based Real-Time Parallel Vision System and Its Application to Real-Time Volume Reconstruction. In *Computer Vision Systems, Second International Workshop, ICVS, Vancouver (Canada)*, 2001.

[3] B.G. Baumgart. A polyhedron representation for computer vision. In *AFIPS National Computer Conference*, 1975.

[4] E. Borovikov, A. Sussman, and L. Davis. A High Performance Multi-Perspective Vision Studio. In *17th Annual ACM International Conference on Supercomputing, San Francisco (USA)*, 2003.

[5] E. Boyer and J.-S. Franco. A Hybrid Approach for Computing Visual Hulls of Complex Objects. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Madison, (USA)*, volume I, pages 695–701, 2003.

[6] G. Cheung, S. Baker, and T. Kanade. Visual Hull Alignment and Refinement Across Time : A 3D Reconstruction Algorithm Combining Shape-From-Silhouette with Stereo. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Madison, (USA)*, 2003.

[7] G. Cheung, T. Kanade, J.-Y. Bouguet, and M. Holler. A real time system for robust 3d voxel reconstruction of human motions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Hilton Head Island, (USA)*, volume 2, pages 714 – 720, June 2000.

[8] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay Triangulation. *Computer Graphics Forum*, 12(3) :129–142, 1993.

[9] C.R. Dyer. Volumetric Scene Reconstruction from Multiple Views. In L.S. Davis, editor, *Foundations of Image Understanding*, pages 469–489. Kluwer, Boston, 2001.

[10] A. François and G. Médioni. A Modular Software Architecture for Real Time Video Processing. In *Computer Vision Systems, Second International Workshop, ICVS, Vancouver (Canada)*, pages 35–49, 2001.

[11] J.S. Franco and E. Boyer. Exact Polyhedral Visual Hulls. In *Proceedings of the 14th British Machine Vision Conference, Norwich, (UK)*, 2003.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engeneering Computation Series. The MIT Press, 1994.

[13] Y. Kameda, T. Taoda, and M. Minoh. High Speed 3D Reconstruction by Spatio Temporal Division of Video Image Processing. *IEICE Transactions on Informations and Systems*, (7) :1422–1428, 2000.

[14] A. Laurentini. The Visual Hull Concept for Silhouette-Based Image Understanding. *IEEE Transactions on PAMI*, 16(2) :150–162, February 1994.

[15] M. Li, M. Magnor, and H.-P. Seidel. Improved hardware-accelerated visual hull rendering. In *Vision, Modeling and Visualization Workshop, Munich, (Germany)*, 2003.

[16] W. Matusik, C. Buehler, and L. McMillan. Polyhedral Visual Hulls for Real-Time Rendering. In *Eurographics Workshop on Rendering*, 2001.

[17] P.J. Narayanan, P.W. Rander, and T. Kanade. Constructing Virtual Wolrds Using Dense Stereo. In *Proceedings of the 6th International Conference on Computer Vision, Bombay, (India)*, pages 3–10, 1998.

[18] Qhull, convex hull and delaunay triangulation library. http ://www.geom.uiuc.edu/software/qhull/.

[19] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the skipper project. *Parallel Computing*, 28(12) :1685–1708, 2002.

[20] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafe. A Survey of Methods for Volumetric Scene Reconstruction from Photographs. In *International Workshop on Volume Graphics*, 2001.

[21] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8) :103–111, August 1990.