INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Communications"

préparée au laboratoire INFORMATIQUE ET DISTRIBUTION
dans le cadre de ***l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Jérémie ALLARD

le 25 novembre 2005

FlowVR : calculs interactifs et visualisation sur grappe

Directeur de thèse : Mme Brigitte Plateau

JURY

MME MARIE-PAULE CANI	, Présidente
M. ALAN CHALMERS	, Rapporteur
M. THIERRY PRIOL	, Rapporteur
MME BRIGITTE PLATEAU	, Directeur de thèse
M. BRUNO RAFFIN	, Co-encadrant

Remerciements

Je tiens à remercier Bruno Raffin qui m'a encadré pour ces travaux et dont les efforts ont permis de monter la plateforme expérimentale GrImage, sans laquelle une grande partie de ce travail n'aurait pas été possible.

J'ai travaillé avec de nombreuses personnes au cours de cette thèse, et je remercie plus particulièrement Clément Ménier qui a toujours été présent qu'en cela était nécessaire. J'ai eu de nombreux échanges trés bénéfiques avec François Faure autour des problèmes liés à l'animation et aux interactions et je lui en suis très reconnaissant. Je remercie aussi Nicolas Turro et Florian Geffray pour leurs contributions sur les aspects techniques des expérimentations.

Merci aux membres de mon jury, Marie-Paule Cani, Brigitte Plateau, Alan Chalmers et Thierry Priol pour avoir accepté d'en faire partie. Leurs remarques avisées m'ont permis de prendre conscience de nouveaux points de vue sur mon travail.

Je tiens également à remercier l'équipe du laboratoire ID pour la bonne ambiance de travail et les discussions fort intéressantes.

Enfin je ne serais pas la sans ma famille, qui m'a soutenu moralement pendant ces trois ans. Je remercie mes quatre grand-parents, qui ont eut la patience de relire tout ce document plusieurs fois. J'ai une pensée toute particulière pour mes deux neveux, Nathaël et Fabian, qui sont nés pendant cette thèse, et à qui je la dédie.

Table des matières

Introduction 1

Contexte

Depuis les années 1970 avec l'arrivée des premiers casques de visualisation et les projets précurseurs de simulateurs de vol, la réalité virtuelle n'a cessé d'évoluer vers des environnements de plus en plus complexes, nécessitant le développement de plateformes matérielles et logicielles performantes. Cette évolution est stimulée par des applications industrielles comme la conception de futurs produits (aéronautique, automobile) ou l'exploration de gros volumes de données (recherches pétrolières, mesures ou simulations scientifiques). Ces applications demandent des ressources en calculs importantes pour piloter les différents périphériques, faire face aux traitements nécessaires ainsi qu'aux contraintes liées à l'interactivité. Une tendance actuelle des plateformes de réalité virtuelle est d'exploiter la puissance apportée par des grappes de machines, afin de répondre à moindre coût à ces besoins. Cependant ceci requiert un travail supplémentaire lors du développement des outils et des applications lié à leur nécessaire parallélisation.

Dans le domaine du parallélisme les grappes sont désormais bien maîtrisées. Plusieurs paradigmes d'exploitation subsistent (passage de messages, mémoire virtuellement partagée, parallélisation semi-automatique avec annotation du code), chacun adapté à certains types d'applications. Pour obtenir plus de puissance de calcul les grappes sont désormais agrégées pour former des *grilles de calcul*, réparties à l'échelle régionale ou nationale. L'exploitation de ces plateformes implique de gérer les multiples réseaux (intra et inter grappes), ainsi que l'hétérogénéité des architectures. Ceci entraîne un rapprochement avec les outils d'applications distribuées, basés sur des concepts de couplage de codes et de déploiement d'applications. Ceux-ci doivent toutefois être adaptés pour supporter la parallélisation des éléments de l'application, ainsi que le volume de données échangées.

Problématique

L'application des méthodes du parallélisme aux applications de réalité virtuelle n'est pas immédiate. En effet, ces applications présentent des contraintes supplémentaires primordiales liées aux multiples périphériques utilisés ainsi qu'au besoin d'interactivité. Ainsi, les travaux de parallélisme se concentrent généralement sur le temps global nécessaire à l'exécution d'une application, alors qu'une application interactive nécessite une fréquence minimum de mise à jour de la boucle de calculs, ainsi qu'une bonne réactivité, c'est-à-dire une latence minimale entre une action de l'utilisateur et l'affichage du résultat. Par exemple, une parallélisation basée sur un pipeline permet d'augmenter la fréquence des calculs mais pas ce paramètre de latence.

Disposer de plus de ressources de calcul pour une application de réalité virtuelle permet d'augmenter la complexité du monde virtuel, par exemple en intégrant de plus en plus d'objets, ou en gérant des interactions de plus en plus complexes avec l'utilisateur ou entre objets. Cette évolution introduit une difficulté supplémentaire au niveau du design de l'application, car il est nécessaire d'intégrer un nombre croissant de composants, algorithmes, codes ou librairies. Cette intégration n'est pas triviale car chaque partie peut avoir des contraintes très différentes tant au niveau pratique (langage de programmation, dépendances, formats de données), que théorique (fréquence de fonctionnement, méthode de parallélisation).

Du fait de la complexité de telles applications, ainsi que la présence de nombreux outils ou codes existants, un besoin important concerne la réutilisation des éléments de l'application. Ainsi, la construction de l'application doit permettre l'intégration de composants préexistants, ainsi que la réutilisation des composants nouvellement développés dans les applications futures. Ceci implique d'utiliser des méthodes génériques de couplage de codes, permettant d'adapter aussi facilement que possible les différents composants aux conditions spécifiques liées à la plateforme d'exécution, aux périphériques utilisés, ainsi qu'aux demandes de l'utilisateur.

Un système permettant de résoudre les problèmes précédents permet d'envisager des applications nouvelles. Des programmes de simulation à l'origine *off-line* peuvent être couplés à des dispositifs d'interactions et ainsi permettre de manipuler directement les objets simulés. Plusieurs de ces simulations peuvent ensuite être intégrées dans un même monde virtuel pour gérer différents types d'objets et offrir un plus grand réalisme. Il faut alors trouver des techniques nouvelles pour prendre en compte les nouveaux types d'interactions entre ces objets. Ce genre de système peut être exploité par exemple pour simuler en temps réel une partie du corps humain, qui contient a la fois des objets rigides (os), déformables (muscles, veines), et liquides (sang), et ainsi permettre de simuler une opération chirurgicale.

Contributions

Cette thèse présente un ensemble de modèles et d'outils pour la construction d'applications distribuées interactives. Ces réalisations concernent différents niveaux de l'application. Une série d'applications expérimentales permet d'évaluer leurs utilisations sur des cas concrets.

FlowVR

Notre premier travail repose sur la spécification de *FlowVR*, un nouveau modèle servant de base pour la construction d'applications interactives modulaires, inspiré des travaux en parallélisme et systèmes distribués mais adaptés pour la réalité virtuelle.

Les composants ont très peu d'informations sur le reste de l'application, afin d'en être le plus indépendant possible. Ils peuvent être parallélisés en interne, toutefois les communications entre composants distincts et les politiques de synchronisation et de couplage sont gérées de manière externe. L'accent est porté sur l'utilisation d'éléments simples et relativement indépendants pour gérer chaque fonctionnalité nécessaire. Ainsi il nous est possible d'explorer facilement différentes politiques de couplage, et de développer des extensions adaptées aux besoins de certaines parties de l'application.

L'objectif étant de construire des applications de plus en plus complexes, l'accent est mis sur le passage à l'échelle des mécanismes de développement de ces applications. En particulier, cette construction repose sur l'élaboration de graphes de flux de données générés par un système de scripts permettant un *dépliage* paramétrable des composants sur l'architecture cible. Cette approche permet d'obtenir une représentation visuelle de l'application très utile pour suivre sa conception et apporter des modifications.

FlowVR Render

Sur cette base nous avons travaillé sur la modularisation et l'optimisation de la partie visualisation de l'application, grâce à une description de la scène 3D en primitives indépendantes utilisant des *shaders* pour en contrôler le rendu. Cette représentation permet une grande souplesse sur la conception et le placement des composants de visualisation tout en garantissant une implantation efficace des traitements nécessaires aux techniques de rendu distribué. Grâce à une conception particulière du protocole de communication notre approche évite la plupart des surcoûts des systèmes classiques de rendu distribué lors des opérations de découpage et fusion des différents flux graphiques, liés en particulier aux problèmes de gestion des interdépendances entre les primitives classiques.

La description de l'apparence des objets utilise des *shaders*, permettant d'exploiter pleinement les fonctionnalités avancées des nouvelles générations de cartes graphiques. Grâce à cela une partie des calculs peut être déportée sur les machines de rendu, ce qui modifie le volume de données à transmettre sur le réseau.

Applications

Plusieurs applications ont pu être développées grâce à FlowVR. Nous avons ainsi intégré dans une application de réalité virtuelle des calculs complexes permet de calculer en temps réel une représentation 3D de l'utilisateur à partir d'un réseau de caméras. Cette application est ensuite devenue de plus en plus complexe par l'ajout de nombreux composants d'interactions et de simulations parallèles.

L'architecture de rendu distribuée *FlowVR Render* a abouti à des applications permettant un rendu volumique haute-résolution sur mur d'image, des algorithmes de texturage du modèle 3D de l'utilisateur par reprojection des flux vidéos des caméras, ou encore la visualisation de l'état d'une grille par un ensemble de composants gérant chacun un type d'information.

Enfin une application expérimentale très ambitieuse, combinant dans un même environnement un fluide 3D parallélisé, un moteur de collisions rigides, un filet déformable et la reconstruction 3D multi-caméras de l'utilisateur, constitue l'application la plus complexe développée avec FlowVR. Elle ouvre de nouvelles perspectives quand aux évolutions des approches de construction d'applications permettant de faciliter le développement d'environnement comportant de nombreuses interactions multi-physiques.

Organisation

L'état de l'art présente les principaux travaux en réalité virtuelle (chapitre 2), en parallélisme et systèmes distribués (chapitre 3), ainsi que les outils existants de conception d'applications de réalité virtuelle distribuées (chapitre 4).

La partie II s'attache tout d'abord aux travaux préliminaires liés à Net Juggler et aux premiers prototypes de couplages parallèles simulation/visualisation (chapitre 5), posant ainsi la problématique importante pour la suite, dédiée à la description du modèle de construction d'applications de réalité virtuelle distribuées (chapitre 6), son implantation (chapitre 7) ainsi que son évaluation au travers d'une étude de cas des applications développées (chapitre 8).

Une nouvelle approche de rendu distribué est présentée dans la partie III. Basé sur un modèle de description de la scène (chapitre 11) et des schémas de transmission de cette description (chapitre 12), ce travail permet de construire des applications de visualisation avancées et de les exécuter de manière efficace sur mur d'image (chapitre 13).

Enfin la partie IV présente une première architecture de couplage de haut niveau entre plusieurs simulations interdépendantes, permettant la construction d'applications ambitieuses gérant des environnements virtuels complexes et très interactifs.

I

Etat de l'art

Notre travail se rapporte principalement à la conception et l'utilisation d'outils liés au parallélisme et au couplage de codes pour la réalité virtuelle. Pour bien cerner les contraintes spécifiques à ce domaine, le chapitre 2 présente un panel d'outils utilisés pour la réalisation d'applications de réalité virtuelle. Ensuite dans le chapitre 3 nous présentons les travaux en parallélisme et systèmes distribués liés aux applications plus classiques de calculs haute performance et de couplage de codes parallèles dont les principes pourront être adaptés pour la réalité virtuelle. Les travaux existants de conception d'applications de réalité virtuelle distribuées sont présentés dans le chapitre 4.

Logiciels pour la réalité virtuelle 2

"REALITY IS MERELY AN ILLUSION,
ALBEIT A VERY PERSISTENT ONE."
Albert Einstein

La réalité virtuelle peut être définie comme un rapprochement entre le monde réel et un monde synthétique, c'est-à-dire calculé par ordinateur. Ce rapprochement a pour but d'immerger l'utilisateur dans un environnement virtuel, via 3 étapes principales :

- Entrée : capture des actions (mouvements) de l'utilisateur ;
- Calculs : Mise à jour de l'environnement virtuel ;
- Sorties : présentation de la représentation (visuelle, auditive, tactile, etc) de cet environnement à l'utilisateur.

Le coeur d'une application de réalité virtuelle réside dans la *boucle d'interaction* (figure 2.1). Cette boucle contient les 3 étapes décrites précédemment qui sont répétées continuellement. D'autres boucles peuvent se retrouver dans des simulations (calculs de l'état de certains objets en fonction de leur état précédant et de lois d'évolution et d'interactions).

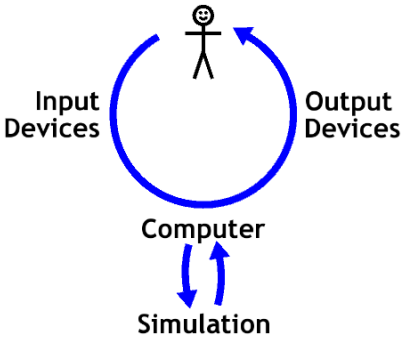


Figure 2.1 *Boucle principale d'une application interactive.*

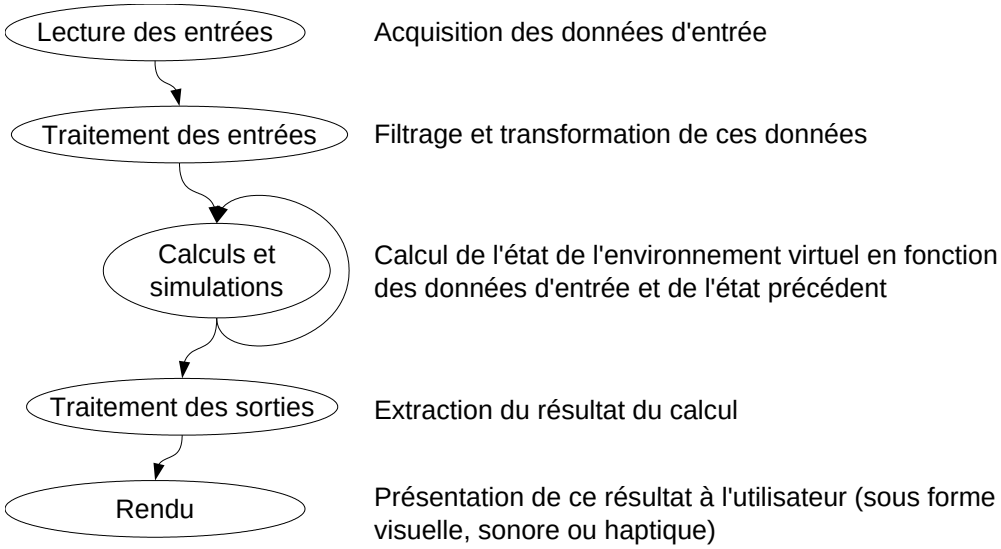


Figure 2.2 *Structure d'une application de réalité virtuelle.*

Cette modélisation des applications par une boucle de traitement a été utilisée par Robertson et al. [149] qui proposent une architecture d'application basée sur plusieurs agents modélisant l'utilisateur et les éléments de l'environnement virtuel. Ces agents sont mis à jour à chaque itération de la boucle d'interaction. Par la suite, Shaw et al. [160] introduisirent le *Decoupled Simulation Model* qui sépare la boucle de simulation de celle d'interaction, pour ainsi permettre l'intégration de calculs plus complexes tout en gardant une bonne interactivité.

De nombreuses techniques permettant de réaliser chacune de ces étapes sont développées depuis maintenant plus de 40 ans [171]. Récemment la recherche s'est concentrée sur l'utilisation de plusieurs périphériques et ressources de calculs en parallèle pour supporter des applications de plus en plus évoluées. Ceci contribue à un accroissement significatif de la complexité de ces applications. Pour faire face à cette complexité il est nécessaire d'utiliser un certain nombre d'outils permettant de mettre en oeuvre chacune des tâches nécessaires.

Ces outils sont devenus indispensables pour la réalisation d'applications de réalité virtuelle. Il est donc primordial de les connaître avant de travailler sur les environnements de plus haut niveaux dédiés à la construction de ces applications, ce qui constitue le thème principal des prochains chapitres.

Ce chapitre présente un tour d'horizon des travaux récents, regroupés en fonction de la tâche qu'ils accomplissent, en utilisant le découpage présenté sur la figure 2.2.

(a) Gant (b) Traqueurs magnétiques (c) Souris 3D

Figure 2.3 *Périphériques classiques de capture de mouvement.*

2.1 Lecture des entrées

La première étape est la récupération des données d'entrée utilisées par le reste de l'application. Cette tâche est extrêmement dépendante de la provenance des données, et en particulier du matériel utilisé.

Pour capturer les actions de l'utilisateur et les intégrer dans le monde virtuel, il est souvent nécessaire d'obtenir le déplacement dans l'espace d'au moins certaines parties de son corps (tête, mains). De nombreux périphériques permettent d'obtenir cette information (figure 2.3). Ils sont très spécialisés et assez complexes. De plus, ils imposent souvent des limitations en terme de temps de préparation, précision, ou nombre de points traqués. Des périphériques différents sont donc utilisés en fonction de l'application ou de l'environnement immersif.

Des librairies logicielles telles que Trackd [180] ou VRPN [88] permettent de s'abstraire du matériel utilisé, et supportent l'accès à des périphériques distants, ce qui est utile dans le cas où le driver fourni requiert une plateforme particulière.

Une information plus riche peut être obtenue en combinant les données de plusieurs capteurs, tels qu'un ensemble de caméras, mais nécessite des traitements coûteux pour extraire de ces données les informations requises. C'est le rôle de la prochaine étape.

2.2 Traitement des entrées

Cette tâche rassemble tout type de traitements qu'on peut effectuer sur les données permettant d'en extraire les informations utiles au reste de l'application. Ceci peut consister en un simple filtrage des données pour corriger les imprécisions des capteurs, mais peut parfois nécessiter des traitements plus complexes (calibration, détection d'objets, de gestes, de mots)

Ainsi des capteurs basés sur des caméras nécessitent de nombreux traitements pour extraire les informations requises. Il peut par exemple être nécessaire d'extraire la sil-

houette de l'utilisateur en séparant les pixels du fond de l'image de ceux correspondant à l'utilisateur (opération de *soustraction de fond* [174]). En utilisant plusieurs caméras il est possible de combiner ces silhouettes pour obtenir une information 3D [21, 103]. Les avantages de cette méthode sont principalement le fait que l'utilisateur n'a pas besoin d'être équipé de capteurs particuliers, et que l'information obtenue correspond à l'ensemble du corps et non un petit nombre de points.

Comme les traitements nécessaires sont très dépendants du type de données recueillies et des informations à extraire, il n'existe pas d'outil générique. Lorsqu'il s'agit de filtrer un signal d'entrée il existe de nombreuses classes de filtres [79] permettant de réduire le bruit, changer la fréquence d'échantillonnage, ou prédire les prochaines valeurs pour diminuer la latence. OpenTracker [147] fourni pour les traqueurs des éléments de filtrages sous forme d'un graphe de flux de données. Pour le cas du traitement de flux vidéos un ensemble d'outils de base est rassemblé dans la librairie OpenCV [90].

2.3 Calculs et simulations

La plupart des applications de réalité virtuelle possèdent un "état" qui évolue au cours du temps, correspondant à l'environnement virtuel. Certaines applications, telles que les simulations physiques, sont entièrement centrées autour de cet état et l'essentiel du travail consiste en sa mise à jour. Pour d'autres cas, comme la visualisation de données, l'état de la scène dépend peu voir pas de l'état précédent.

Contrairement aux étapes précédentes, le rôle principal de cette tâche n'est pas de produire des données, mais plutôt d'implanter le comportement de l'application. Par exemple, une simulation de fluide génère en sortie une matrice de densité du fluide, mais surtout calcule en interne les déplacements de ce fluide correspondant aux lois physiques souhaitées (forces appliquées, incompressibilité, conservation de l'énergie). De même, un moteur de gestion de collisions entre objets solides cherche à garantir que ces objets ne s'interpénètrent pas.

On peut distinguer plusieurs types de calculs à l'intérieur d'une application de réalité virtuelle :

Les systèmes multi-agents : ensembles d'objets indépendants échangeant de faibles quantités de données (évènements, positions, vitesses) utilisés pour décrire le comportement des entités de l'environnement (véhicules, avatars).

Les simulations interactives : calculs de comportements complexes s'inspirant fréquemment de la physique (collisions, écoulements). Ces simulations requièrent l'utilisation de calculs coûteux mais mettent l'accent sur l'interactivité et non la précision.

Les simulations externes : programmes de calcul indépendants reliés à l'application de réalité virtuelle permettant d'explorer les résultats. La liaison est soit différée, l'application charge alors le résultat de la simulation depuis un fichier, soit en

temps-réel. Il est alors possible de contrôler en retour certains paramètres ou forces appliquées. On parle alors de *computational steering*.

Nous allons détailler chacun de ces types de calculs dans les prochaines sections.

2.3.1 Systèmes multi-agents

Un système multi-agents dans le cadre d'un environnement virtuel peut être réalisé différemment en fonction de la taille de l'environnement et de la complexité des comportements simulés. Les points importants reposent sur l'intégration des différents agents ainsi que le mécanisme de communication et synchronisation contrôlant leur exécution.

OpenMASK [109] repose ainsi sur un arbre rassemblant tous les objets de la scène. Ces objets sont ensuite instanciés dans plusieurs threads et éventuellement sur plusieurs machines. Ils sont exécutés de manière synchrone à l'intérieur d'une boucle principale gérant des fréquences d'activation fixes pour chaque objet. OpenMASK communique des valeurs et des évènements entre ces objets et offre un mécanisme de ré-échantillonnage pour gérer le multi-fréquences.

A l'opposé, *DIS* [89, 86] (Distributed Interactive Simulation) et son successeur *HLA* [87] (High Level Architecture), des standards IEEE issus de l'armée américaine, permettent des simulations complexes avec plusieurs dizaines de milliers d'objets distribués sur de nombreux sites distants. Ces systèmes sont basés sur un mécanisme d'exécution complètement asynchrone, où chaque objet se met à jour localement et reçoit occasionnellement l'état des autres objets. Des techniques de multicast par publication/souscription permettent de gérer les communications de manière efficace.

De nombreux systèmes se concentrent sur les environnements collaboratifs distribués [31, 73, 78, 140]. Ils reposent sur des principes similaires, mais gèrent en plus l'aspect interface multi-utilisateurs.

Les approches de type multi-agents sont très bien adaptées à la simulation d'environnements vastes (villes, champs de batailles), mais difficilement extensibles aux autres types d'applications qui ne comportent généralement que quelques dizaines d'objets distincts, ou dont les interactions nécessitent un couplage plus resserré. Ainsi les algorithmes inspirés de la physique, et qui sont utilisés dans les sections suivantes, utilisent pour la plupart une résolution de systèmes d'équations combinant tous les objets de l'environnement.

2.3.2 Simulations interactives

Simuler des interactions complexes et réalistes dans un environnement virtuel requière l'utilisation d'algorithmes de calcul évolués.

Le premier problème concerne la détection de collision [38]. Les approches standards utilisent des hiérarchies découpant l'espace ou regroupant les volumes englobant des objets [70, 101]. Des implantations de ces algorithmes sont disponibles [176, 173].

Il est ensuite nécessaire de gérer la réponse à ces collisions pour animer des objets rigides [77]. Les outils les plus utilisés sont NovodeX [127] (Commercial) et Open Dymanics Engine [163].

Pour les autres types d'objets il n'existe pas d'outils similaires. Cependant de nombreux travaux présentent les algorithmes requis pour les fluides [165, 54], les tissus [30], et les objets déformables [47].

2.3.3 Simulations externes

L'intégration de simulations de grande taille pose deux problèmes importants : la parallélisation des calculs et la transmission des données vers la visualisation. Le premier est très classique et de nombreux travaux s'y attachent, comme les librairies ScaLAPACK [27], PETSc [20] et Global Arrays [126]. Le second est un problème de couplage de code auquel les outils génériques (section 3.2 page 21) peuvent répondre. Toutefois des travaux spécialisés dans le couplage entre une simulation et une visualisation (*computational steering* [121]) offrent une réponse plus adaptée. On trouve par exemple CUMULVS [65] qui permet de coupler une simulation parallèle à une ou plusieurs visualisations séquentielles, et ESPN [56] qui gère le couplage entre une simulation parallèle et une visualisation parallèle en utilisant la librairie de communication RedGRID [55]. De manière un peu différente, COVISE [145] intègre dans un même environnement la construction de la simulation à partir du couplage de différents composants, et la visualisation collaborative des résultats.

2.4 Traitement des sorties

Bien souvent les données produites dans l'étape précédente ne sont pas exploitables directement, il est donc nécessaire de les traiter pour en extraire les informations recherchées. Ces informations doivent être dans un format utilisable par l'étape suivante de rendu proprement dit. Ce format dépend du type de rendu. Par exemple, pour un affichage visuel il faut décrire la scène sous forme de triangles, mais d'autres données sont utilisées pour un rendu sonore ou tactile.

De nombreux travaux s'articulent autour de la visualisation scientifique et s'intègrent à cette problématique. Parmi les outils les plus utilisés on trouve AVS [177], OpenDX [1], SCIRun [139, 158] et VTK [157]. Ces outils sont structurés autour d'une gestion de volumes de données importants, souvent même trop importants pour tenir en mémoire vive. Des opérations sont appliquées sur ces données selon une approche modulaire à base de graphe de flux de données ou chaque noeud effectue une opération de base (extraction de surface, découpage, coloration). Certains outils comme SCIRun permettent de construire ce graphe visuellement (figure 2.4).

Les opérations sont souvent ordonnancées selon un mécanisme très lié au problème

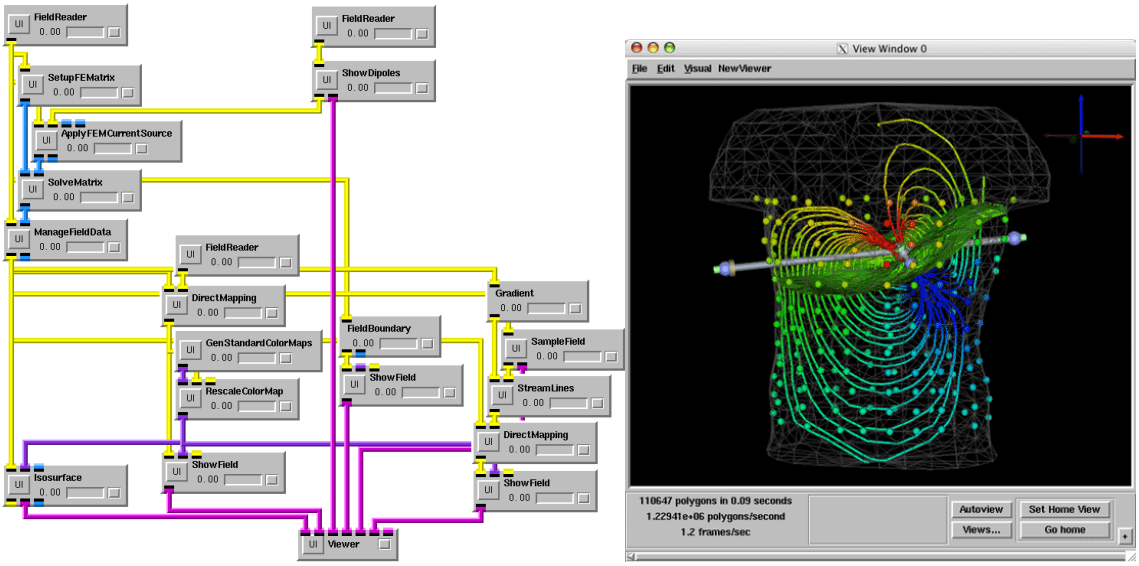


Figure 2.4 *Application utilisant l'environnement de visualisation interactive SCIRun [139].*

de visualisation. Leur activation se fait en fonction de leur utilité pour produire l'image demandée (*demand-driven scheduling*). Si les données ne tiennent pas en mémoire elles sont traitées morceau par morceau.

2.5 Rendu

La dernière étape est l'envoi aux périphériques de sortie des données produites par l'application. Pour le cas d'un rendu graphique cela consiste par exemple en l'envoi des objets de la scène sous forme de triangles et de textures. Ces données sont ensuite utilisées par le driver pour produire l'image finale. L'interface de programmation (*Application Programming Interface – API*) la plus répandue est OpenGL [159], qui repose sur une suite de commandes spécifiant les triangles composant la scène ainsi que leurs paramètres (transformations, lumières, matériaux, textures). Le fonctionnement d'OpenGL sera détaillé dans la section 10.1 (page 96).

Des outils de plus haut niveau permettent à l'application de manipuler une représentation de la scène plus évoluée. Celle-ci est généralement structurée sous forme de graphe de scène, organisant les différents objets sous forme d'une arborescence de noeuds pouvant contenir des objets 3D ou des attributs comme des transformations ou des matériaux. Ceci facilite le parcours et la manipulation des différents objets, et permet de fournir des fonctionnalités réutilisables entre les applications (chargement d'objets à partir de fichiers, édition des paramètres des objets, détermination des branches visibles pour le rendu, niveau de détail dynamique). Ce principe est utilisé par des outils tels que Inventor [169], Performer [152], et plus récemment OpenSG [146].

(a) HMD (b) Workbench (c) CAVE

Figure 2.5 *Systèmes classiques d'affichage.*

(a) (b)

Figure 2.6 *Mur d'images combinant plusieurs vidéo-projecteurs. (a) Paul Rajlich devant le mur d'image de la NCSA comportant* 6 *projecteurs pour ensuite passer à* 40 *projecteurs (b).*

Afin d'obtenir un environnement immersif, les systèmes de réalité virtuelle utilisent fréquemment plusieurs affichages destinés à envoyer une image différente sur chaque oeil (vision stéréoscopique) ou à couvrir un champ de vision le plus large possible. Les dispositifs classiques sont présentés sur la figure 2.5.

Récemment l'accent a été mis sur la résolution et la luminosité de l'affichage, permettant de visualiser de grandes quantités de données ou d'utiliser plus facilement des caméras vidéos. Pour cela on utilise un ensemble de vidéo-projecteurs pour former un mur d'images (figure 2.6). Les premiers prototypes furent développés à partir de 1995 [175, 40] en utilisant des technologies haut de gamme (stations SGI, projecteurs CRT). Une deuxième génération de murs d'images a émergé, basée sur des composants standards [81, 105] (grappes de PC, projecteurs LCD ou DLP, calibration logicielle [36]) et permet de construire de très grands murs d'images.

2.6 Bilan

Ce chapitre a présenté un découpage des applications de réalité virtuelle en tâches successives, ainsi qu'un panel d'outils permettant d'implanter chacune de ces tâches

Pour construire une application de réalité virtuelle, il est nécessaire de combiner plusieurs de ces outils. On est alors confronté à des problèmes de compatibilité (format des données, langage de programmation et plateformes supportées) et de conflits (contrôle de la boucle principale, temps CPU nécessaires). Un début de solution consiste à découper l'application en modules aussi indépendants que possibles, qu'il faut alors coupler pour obtenir le résultat voulu.

Ces problèmes sont encore plus apparents pour les plateformes distribuées comme les grappes. Il faut alors gérer la distribution des périphériques, les communications entre les machines, et la cohérence de l'ensemble tout en maintenant des performances permettant l'interaction.

Deux approches possibles peuvent y apporter une réponse : utiliser les travaux existants développés pour les applications "classiques" (c'est-à-dire non interactives), ou bien concevoir de nouveaux cadres d'applications spécialement dédiés à la réalité virtuelle. Ces deux approches sont l'objet des deux prochains chapitres.

Communications et couplage de codes 3

Construire une application complexe ne se limite pas aux outils permettant d'implanter chacune des fonctionnalités nécessaires, faut-il encore pouvoir les rassembler en un ensemble cohérent. Deux grands domaines y contribuent : les méthodes issues du calcul haute performance, dédiées aux communications sur supercalculateur parallèle, grappe ou grille, et les modèles de couplage de codes à base d'objets ou de composants, utilisés pour les applications distribuées liées à Internet, au travail collaboratif et aux réseaux d'entreprises. Chacun de ces domaines utilise un vocabulaire spécifique. Pour uniformiser le discours, nous utiliserons dans la suite le terme *module* pour désigner l'unité de distribution (processus de calcul, objet distant, composant, etc) et *connexion* le mécanisme de liaison entre ces modules (canal de communication, appel de procédure distante, publication/souscription d'évènements).

En sus des deux domaines d'applications (parallèles / distribuées), les différentes approches peuvent être classifiées selon les critères importants suivants :

Granularité : Niveau de découpage de l'application. Ainsi un découpage à grains fins désigne de petits modules, à l'échelle d'un appel de procédure par exemple, alors qu'un découpage à gros grains utilise un seul module par machine.

Dynamicité : Est-il possible d'ajouter des modules ou des machines en cours d'exécution ? Un module peut-il migrer sur une autre machine ? La suppression intentionnelle ou non d'une ressource est-elle supportée ?

Distribution et hétérogénéité : Les différents modules sont placés sur des machines identiques appartenant à une même grappe ou au contraire il est possible d'utiliser

des machines d'architectures différentes dans plusieurs lieux distants.

Environnement de développement et d'exécution : Le lancement des modules et leurs connexions sont-ils à la charge des modules eux-mêmes, d'un module particulier, ou alors l'outil utilisé l'implante-t-il de façon transparente ?

Dans la suite de ce chapitre, les approches relativement bas-niveau prenant en charge les communications seront d'abord présentées, suivis par les modèles de couplage de plus haut niveau, avant de conclure par un bilan récapitulatif et une discussion sur les réponses que ces outils apportent concernant les applications interactives.

3.1 API de communication

Le plus bas niveau utilisé pour les communications est la couche socket [167] TCP ou UDP, qui est standardisée sur toutes les plateformes et permet d'implanter tout type de schéma de communication. Ce niveau très bas ne permet pas d'offrir l'abstraction suffisante pour la plupart des applications, mais sert très souvent de base aux outils de plus haut niveau. TCP permet d'implanter des canaux de communications point à point surs, alors qu'UDP est utilisé pour les envois à faible latence ou pour plusieurs destinataires simultanés (*broadcast* ou *multicast*). Pour des réseaux plus spécialisés comme SCI, Myrinet [29] ou Infiniband d'autres API sont utilisées permettant de plus hautes performances, en particulier au niveau latence, comme BIP [67] ou GAMMA [37].

Toutes ces API de communication sont utilisées pour découper l'application en un seul module par machine. La granularité de la parallélisation des calculs peut cependant être beaucoup plus fine du fait que des communications fréquentes peuvent être utilisées. Toutefois ce découpage est très vite limité par les performances du réseau (en particulier la latence des communications). Les API les plus générales comme TCP permettent une grande dynamicité et supportent des applications fortement distribuées, en revanche la plupart des API plus spécialisées sont limitées à des communications locales sur un réseau précis souvent dédié. Tout l'environnement de développement et d'exécution de l'application est à la charge du programmeur.

3.1.1 Passage de messages

Les applications de calcul parallèle reposent essentiellement sur des librairies de passage de messages telles PVM [64] et MPI [113].

PVM fut l'une des premières librairies portables de passage de messages pour le calcul parallèle. Des machines peuvent être ajoutées et supprimées de l'application en cours d'exécution. Cette API fut progressivement remplacée par le standard MPI qui a l'avantage d'être supporté par tous les vendeurs de systèmes parallèles.

MPI permet la parallélisation d'un calcul en lançant un même programme sur un ensemble de machines et en fournissant les primitives de communications point à point ou

collectives entre ces machines. Il est bien adapté aux applications de calcul haute performance en fournissant de nombreux schémas de communications : diffusion (*Broadcast*), répartition (*Scatter*), rassemblement (*Gather*), communications $N \times N$ (*Alltoall*, *Allgather*). Les messages transmis sont construits à partir de tableaux de données d'un type de base (entiers, flottants, etc) ou de données structurées contenant plusieurs types. MPI 1 ne supporte aucune dynamicité et la plupart des implantations supposent des machines uniformes. Un nouveau standard, MPI 2 [114], lève ces limitations mais aucune implantation complète n'est pour l'instant disponible. Les implantations de MPI les plus courantes sont LAM/MPI [32, 164] et MPICH [74, 75]. Elles sont toutes deux basées sur TCP/IP, mais MPICH est porté sur d'autres API pour les grilles [95] ou les réseaux hautes performances [18].

MPI offre un outil de lancement de l'application (*mpirun*), ainsi que des outils de traces et d'analyse de performances [132]. Le code du programme est responsable à la fois des calculs et des communications. Ce mélange rend difficile l'agrégation de plusieurs codes existant dans une même application MPI. MPI est donc très utilisé pour l'implantation de calculs parallèles homogènes (*SPMD, Single Program Multiple Data*), mais ne convient pas facilement au couplage de plusieurs de ces calculs. Toutefois des travaux en ce sens en tant que surcouche au dessus de MPI ont étudié ce type de couplage pour des applications de simulation [2].

3.1.2 Appel de procédure à distance

Plutôt que laisser à l'application le soin de construire et recevoir les messages envoyés sur le réseau, une approche d'un peu plus haut niveau consiste à s'appuyer sur un élément très courant de la programmation : l'appel de procédure. Ainsi un module peut utiliser les fonctionnalités d'un autre module en utilisant un appel de procédure à distance [136] (RPC – *Remote Procedure Call*). Ce modèle permet d'utiliser les paramètres de la procédure pour transmettre les données, ce qui supprime la nécessité de construire explicitement les messages. De plus, la boucle de réception et de répartition des messages est aussi implémentée par l'API et l'utilisateur doit uniquement implémenter les calculs à l'intérieur de la procédure appelée.

Cette technologie fut ensuite associée à la programmation orientée objet pour gérer des objets distribués sur le réseau, à la base de beaucoup d'applications de services d'entreprise. L'approche objet ajoute l'utilisation d'une abstraction des fonctionnalités de l'objet sous forme d'interfaces virtuelles. L'un des standards d'objets distribués les plus utilisés est CORBA [128] qui est multi-plateformes et supporte la plupart des langages de programmation. De nombreuses implantations libres existent [112], telles OmniORB [137] ou ORBit [68]. D'autres standards existent pour des langages ou des plateformes spécifiques, comme RMI [76] (*Remote Method Invocation*) en Java ou DCOM [115] pour Microsoft Windows.

Ce modèle est très bien adapté pour les applications à base de *services*, mais introduit

une latence importante du fait des aller-retours que nécessitent chaque appel de procédure et de l'encodage et le décodage des paramètres d'appel. CORBA utilise un système de compilateur spécialisé utilisant une description de ces paramètres (IDL – *Interface Definition Language*), afin de générer automatiquement le code de codage/décodage. Ceci permet de diminuer le surcoût de ces opérations mais introduit une complexité additionnelle au système. D'autres systèmes de composants utilisent une description au moment de l'exécution, comme XML-RPC [182] ou SOAP [183] qui reposent sur du XML. Cette approche allège le système mais augmente le surcoût d'encodage.

Pour la parallélisation de codes de calcul le modèle RPC peut être utilisé à condition d'optimiser les surcoûts d'encodage des données et rendre les appels asynchrones pour recouvrir les calculs et les communications. C'est le principe des *messages actifs* [179], qui associent à l'intérieur de chaque message l'identifiant de la procédure distante à appeler ainsi que les données à utiliser. Ce principe est repris par des outils tels que Madeleine [17] utilisé par PM2 [124], Nexus [59] utilisé par Globus [58], ou Inuktitut [104] utilisé par Athapascan (section 3.1.3).

Du fait du faible surcoût des appels de procédures entre objets d'une même machine, les outils de RPC peuvent être utilisés pour découper l'application en tout petits modules, effectuant chacun une tâche particulière et déférant certains traitements à d'autres modules. Le système de nommage et recherche des objets permet une grande dynamicité au cours de l'exécution de l'application. Du fait de l'utilisation d'un protocole d'interopérabilité IIOP [128] entre implantations de CORBA, de nombreuses architectures peuvent être utilisées dans une même application. En revanche, le lancement des différents objets et la construction des connexions entre ceux-ci sont toujours à la charge du programmeur.

3.1.3 Athapascan : graphe de flux de données par appels de procédure

Du fait de l'omniprésence d'appels de procédure dans tout code, en rendant ces appels distants il est possible d'introduire du parallélisme de manière assez transparente. C'est le principe de base utilisé par Athapascan [63, 150]. Les appels de procédure d'un programme sont transformés en un ensemble de tâches, les paramètres transmis formant un graphe de flux de données à respecter lors de l'exécution de ces tâches. Ce découpage en tâches peut être effectué de manière dynamique : en fonction du degré de parallélisme voulu chaque appel de fonction peut générer une nouvelle tâche ou simplement être exécuté localement comme un appel classique.

Des algorithmes d'ordonnancement permettent ensuite de choisir l'ordre d'exécution des différentes tâches. Si un processeur n'a plus de tâche à exécuter il peut déporter des tâches distantes par un mécanisme de *vol de tâche*. De ce fait, la répartition des calculs est faite de manière dynamique lors de l'exécution de l'application. En cas de panne ou de suppression d'une machine le calcul peut être redémarré par une technique de *checkpoints* [91].

A la différence des systèmes RPC classiques, Athapascan prend en charge tout l'aspect lancement de l'application, répartition des modules sur les différentes machines, et établissement des communications. Tout ceci est géré de manière dynamique par l'algorithme d'ordonnancement.

3.1.4 PadicoTM : Communications multi-protocoles

Parmi les exécutifs de communication, PadicoTM [49] est un outil particulier puisqu'il a pour objectif d'intégrer les différents paradigmes de communication en un ensemble cohérent, permettant aux applications d'utiliser le meilleur outil pour chaque cas. En effet, l'utilisation simultanée de plusieurs outils de communication peut poser des problèmes d'accès concurrents au driver réseau, ce qui peut engendrer des ralentissements ou même des plantages. PadicoTM recherche une consommation des ressources réseau et CPU *"coopérative plutôt que compétitive"*.

PadicoTM utilise un système de micro-noyau qui charge dynamiquement les applications et les outils de communication nécessaires. Des modules de bas niveaux contrôlent l'accès au réseau et gèrent les threads utilisés. Plusieurs API réseau sont supportées, en particulier TCP/IP et Madeleine/PM2 [17]. Les threads sont gérés par Marcel/PM2 [43], qui en collaboration avec Madeleine garantit une bonne réactivité par rapport aux évènements réseaux.

Des implantations de CORBA, MPI et JAVA ont été portées au dessus de PadicoTM. Ceci permet aux applications d'utiliser ces outils de concert sans rencontrer de problèmes de concurrence. De plus, du fait de l'accent mis sur les performances par les couches de bas niveau et le support de réseaux hautes performances, PadicoTM permet d'obtenir de meilleures performances [49] que l'implantation initiale de CORBA.

3.2 Couplage de codes

Le couplage de codes est un problème très vaste. Après une brève présentation de travaux généraux en ce domaine, nous nous attacherons au couplage de codes parallèles, et plus particulièrement au couplage pour les applications interactives.

Différents modèles peuvent être utilisés pour l'encapsulation des codes permettant leur couplage. La plupart se basent sur une approche d'objets distribués de type CORBA en ajoutant des informations sur leur manipulation (dépendances, lancement, connexions) permettant de relier les objets de façon plus générique. Ceci correspond à un modèle de type *composants* [172]. Le but principal est de libérer le programmeur des tâches liées au lancement et la connexion des modules de l'application.

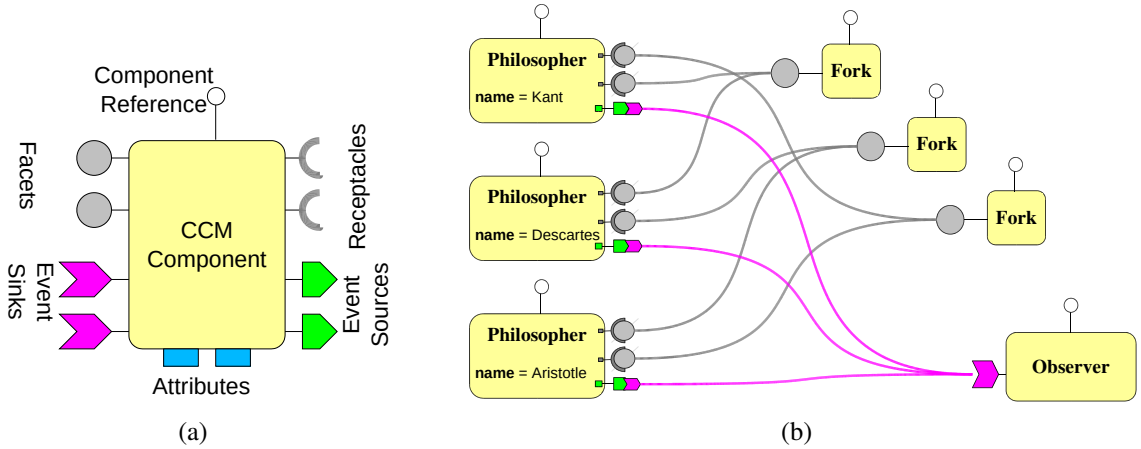


Figure 3.1 *(a) Modèle d'un composant CCM. Les ports de gauche constituent les ports de réceptions d'appels ou d'évènements, alors qu'à droite se trouvent les ports complémentaires d'envoi. (b) Exemple d'application CCM pour le problème du dîner des philosophes [135].*

3.2.1 Composants logiciels

Les outils de gestion d'objets distribués offrent à chaque objet des services d'enregistrement et de recherche des autres objets présents (*naming service*) permettant de s'y connecter pour accéder à leurs fonctionnalités. Cette approche a pour conséquence un mélange du code *fonctionnel* (les fonctionnalités implémentées par chaque objet) avec les parties non-fonctionnelles (lancement, recherche, connexion). Dans un système de type *composant* il y a une séparation entre les composants, qui sont des objets auxquels sont ajoutées les informations nécessaires à leur exécution (dépendances, paramètres, fonctionnalités), et les conteneurs qui sont responsables de la gestion de la vie des composants (déploiement, lancement, destruction) ainsi que leur liaison.

La norme CORBA 3.0 [130] définie le *CORBA Component Model* (CCM) [131]. Chaque composant est stocké dans une archive contenant le binaire ainsi que la description XML de ses fonctionnalités ainsi que ses caractéristiques d'exécution. Une fois lancé, un composant dispose de plusieurs types de ports pour communiquer avec le reste de l'application (figure 3.1(a)) :

Component Reference : port de contrôle du composant utilisé par son conteneur.

Facets : interfaces supportées par le composant. Ceci correspond aux méthodes implantées par le composant, exactement comme pour un objet CORBA classique.

Receptacles : interfaces utilisées par le composant pour remplir ses fonctions. Elles doivent être reliées à des *facets* d'autres composants (figure 3.1(b)). A la différence des modèles objets, le composant n'a pas le contrôle sur la création de cette liaison.

Event sources : port d'envoi d'évènements asynchrones. Cela correspond à un mode de communication par messages, qui peuvent être destinés à un ou plusieurs composants.

Event sinks : port de réception d'évènements.

Attributes : paramètres ou états du composant.

Grâce à cette description de haut niveau de chaque composant il est possible de construire l'application visuellement (figure 3.1(b)). L'application peut ensuite être déployée automatiquement sur l'architecture cible. L'implantation d'un composant peut utiliser de nombreux langages de programmation (JAVA, C++, etc).

D'autres modèles similaires existent tels que Entreprise Java Beans [170] pour JAVA ou COM&.NET [107] de Microsoft. Le modèle CCM est toutefois plus souple et indépendant de la plateforme utilisée (langage, système d'exploitation). Intégrant des fonctionnalités avancées, comme la sécurité, la gestion de la persistance des objets, ou la réflexivité, le standard CCM est assez complexe à maîtriser et implanter complètement.

3.2.2 Couplage de codes parallèles

De nombreux travaux se sont attachés au couplage de plusieurs codes parallèles, permettant de combiner plusieurs simulations ou de relier une simulation avec une visualisation. Par rapport au couplage de code classique, la difficulté ici se trouve dans l'aspect parallélisé de chaque module (objet ou composant), ainsi que dans la quantité importante de données à transmettre. Plusieurs approches sont possibles, ainsi on peut distinguer les solutions basées sur des standards existants modifiés pour gérer ces contraintes, ou au contraire sur la définition de nouveaux modèles de découpage de l'application.

3.2.2.1 Adaptation de modèles existants

Plusieurs travaux se basent sur CORBA (section 3.1.2 page 19). PARDIS [97, 98] et PaCO [148, 93] étendent le langage de description IDL des objets pour spécifier la parallélisation des données lors des appels de procédure. Le compilateur doit ensuite être modifié pour utiliser ces informations dans la génération du code associé. L'inconvénient majeur de cette approche est la nécessité de modifier le standard et donc son implantation à l'intérieur de l'ORB utilisé, ce qui en limite la portabilité. L'organisation responsable de la norme CORBA, l'OMG, a proposé une nouvelle spécification [129] intégrant le concept d'objet parallèle, mais celle-ci requiert aussi un support particulier pour l'implantation de l'ORB. Pour contourner ce problème il est possible d'introduire une couche intermédiaire entre le code de l'application et l'ORB CORBA. C'est ce que fait PaCO++ [48, 50] qui utilise un objet *manager* servant de proxy vers l'objet parallèle réel. Les clients CORBA classiques y accèdent de manière transparente, alors que les clients parallèles utilisent des méthodes additionnelles pour découvrir la parallélisation de l'objet et ainsi communiquer directement les données entre les deux objets parallèles. Ce mécanisme peut être

caché à l'application en utilisant un compilateur pour générer le code nécessaire à cette négociation.

Les technologies à base d'objets comme CORBA ne répondent que partiellement au problème de couplage de codes. En effet, il n'y a pas de mécanisme générique permettant de déployer l'application, et surtout les communications entre objets doivent être explicitement demandées par l'un des objets concernés. Ces tâches sont rendues génériques par l'utilisation de modèles à base de composants (section 3.2.1 page 22). Le modèle de composant CCM au dessus de CORBA peut aussi être adapté pour les simulations parallèles, comme le fait GridCCM [143] qui reprend des concepts similaires à PaCO++.

3.2.2.2 Conception de nouveaux modèles

D'autres projets se basent cette fois sur des modèles de composants développés spécifiquement. C'est le cas de PAWS [22] qui définit un mode d'interaction entre composants basé sur des transmissions de données uniquement (approche de type *data-flow*). Ligature [96] étend ce modèle en ajoutant des interactions basées sur des appels de procédures.

En s'inspirant de plusieurs projets antérieurs, un effort regroupant plusieurs grandes universités et centres de calculs américains essaye de définir un modèle de composants communs [15] (CCA – *Common Component Architecture*). L'objectif est de définir une norme permettant le développement de composants réutilisables pour le calcul scientifique haute performance. Contrairement au modèle de composants de CORBA 3 (section 3.2.1), chaque composant ne définit qu'un ensemble de *ports Provides/Uses* (équivalent aux *Facets* et *Receptacles* de CCM) et pas de ports d'évènements. Ainsi toutes les communications sont basées sur des appels de procédures. Pour émuler un fonctionnement de type envoi d'évènements où un nombre indéterminé d'autres composants peuvent recevoir les informations, un port *Uses* peut être connecté à plusieurs ports *Provides* dans le cas ou aucune procédure de l'interface ne renvoi de valeur de retour. Dans le cas contraire l'association est obligatoirement point à point. Pour supporter les applications de calculs parallèles le modèle supporte le concept de *ports collectifs* qui sont utilisés par les différents threads ou processus d'un composant parallèle. Le système doit ensuite gérer les différents cas de redistribution des données en fonction des connexions. Cette redistribution inclut des cas relativement simples (*scatter* $1 \times N$, *gather* $N \times 1$, connexion entre composants avec la même répartition des données $N \times N$), mais nécessite parfois une redistribution plus générique entre deux composants parallèles arbitraires ($M \times N$).

Bien que des travaux soient en cours pour supporter les redistributions parallèles de données [25, 24, 42], c'est un problème difficile. Cette fonctionnalité n'est pas encore disponible dans la plupart des implantations actuelles. CCAFEINE [4], l'implantation la plus utilisée, supporte des liaisons entre composants parallèles mais uniquement si ces liaisons sont locales à une machine donnée. XCAT [102] supporte les connexions entre

composants distribués mais non parallèles.

De nombreux travaux gravitent autour de CCA. SCIRun2 [186] est une évolution de l'environnement de visualisation interactive SCIRun [139] (section 2.4 page 12) intégrant le support de composants CCA. Uintah [45] permet de relier des composants CCA à SCIRun. Babel [41] permet une interopérabilité entre différentes implantations de CCA.

3.3 Bilan

Ce chapitre a montré différentes approches liées à la construction d'applications parallèles et/ou distribuées, en partant d'outils de communication bas niveau, jusqu'aux environnements évolués de couplage de codes parallèles basés sur des approches composants. Plusieurs tendances importantes peuvent être dégagées de cet état des lieux.

On peut constater un fort rapprochement entre les outils liés au parallélisme, traditionnellement concentrés sur les aspects performances au détriment de la dynamicité et de l'hétérogénéité, et les outils axés sur les applications distribuées, basés sur une forte dynamicité et hétérogénéité ainsi qu'un environnement de développement et d'exécution plus évolué. De nombreux projets font la liaison entre les deux domaines. Ceci est très certainement lié à l'émergence du *metacomputing* [162] et des grilles de calculs, ainsi qu'au développement d'applications de plus en plus complexes, nécessitant le couplage entre une partie calculs haute performance et une partie visualisation interactive des données calculées.

Une autre conclusion qu'on peut dégager est qu'il n'existe pas encore de modèle générique utilisé à grande échelle par la communauté de calculs haute performance. En revanche certaines approches semblent s'imposer, en particulier l'approche à base de composants utilisant des ports de communication. C'est dans les fonctionnalités de ces ports que la plupart des différences se situent. En effet, certains modèles se basent sur une approche *data-flow* n'utilisant que des transferts de données unidirectionnels, alors que d'autres utilisent plus des appels de procédures permettant plus de souplesse et un couplage plus resserré des composants, mais diminuant d'autant la généricité des interfaces entre composants.

Pour gérer la parallélisation, la plupart des approches se basent sur un ensemble de modules SPMD, et définissent un mécanisme de spécification de la répartition des données entres les processus ou fils de calculs de chacun de ces modules. Cette spécification peut être donnée de manière statique à l'intérieur du langage de spécification de l'interface du module, ou bien indiquée par le code du module au moment de l'exécution. La redistribution efficace des données dans le cas général entre deux modules parallélisés de façon différente reste un problème d'actualité.

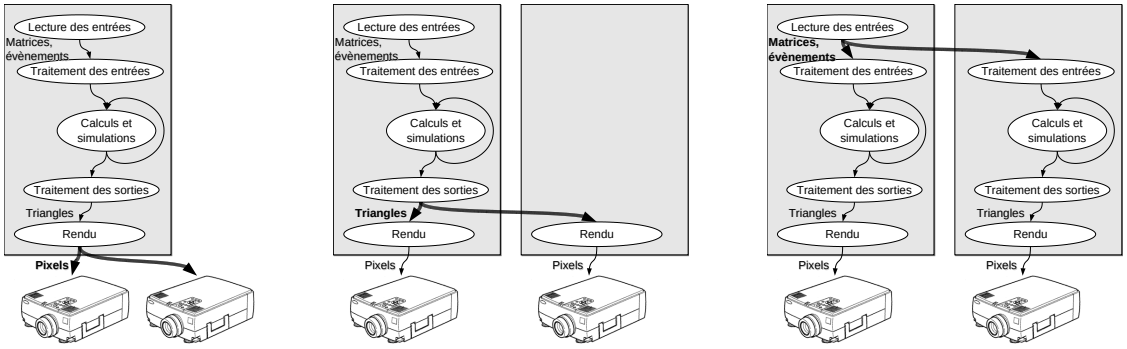
La construction d'applications interactives nécessite un support des différentes fréquences de fonctionnement des modules, liées aux périphériques externes et aux besoins

de l'interactivité (si une simulation lente est présente, la boucle d'interaction doit rester la plus réactive possible). Ce support n'est pas présent dans les travaux présentés dans ce chapitre. Il peut toutefois être ajouté dans le cas de fréquences fixes, en intercalant des modules d'interpolation et extrapolation des données. Il est aussi possible de concevoir les connexions entre modules pour obtenir un couplage asynchrone ou la vitesse de chaque module est indépendante. Par exemple dans le cas d'un module de visualisation relié à une simulation, plutôt qu'utiliser un appel de procédure demandant à la simulation d'effectuer une nouvelle itération et d'en communiquer le résultat, il est possible de demander simplement à la simulation d'envoyer le dernier résultat disponible, sans attendre la fin de l'itération en cours. Cependant, ces modifications doivent être effectuées à l'intérieur des modules eux-mêmes, ce qui limite leur réutilisabilité. Un même module ne sera peut-être pas utilisable à la fois dans une application interactive et dans une application de calcul classique.

Parallélisation pour la réalité virtuelle 4

Notre travail se focalise sur les applications de réalité virtuelle ambitieuses nécessitant de nombreuses ressources. Ces applications sont similaires aux applications de calcul haute performance “classique”, par le fait qu’elles manipulent de grandes quantités de données, et parfois requièrent des calculs complexes pour simuler le comportement des objets virtuels. En revanche des différences importantes apparaissent, de par l’utilisation de nombreux périphériques physiques, et l’existence d’une boucle d’interaction avec l’utilisateur imposant de fortes contraintes en terme de fréquence de mise à jour et de latence. Pour répondre à ces besoins et piloter les environnements multi-projecteurs (CAVE, murs d’images), les grappes de PC remplacent de plus en plus les machines dédiées de type SGI Onyx, en raison de leur faible coût et de l’évolution rapide des cartes graphiques (stimulée par le marché des jeux vidéos). Ce type d’architecture offrant en général des capacités réseaux plus limitées, une attention particulière doit être apportée pour les gérer au mieux.

En conséquence on assiste à un rapprochement des domaines du parallélisme et de la réalité virtuel, accentué par les couplages de plus en plus utilisés entre une partie visualisation sur un équipement de réalité virtuelle, et une partie simulation à grande échelle. Ces nouvelles applications ont pour objectif de mieux appréhender le résultat des applications de calculs complexes, et de pouvoir le faire sans attendre la fin de l’exécution du calcul, ce qui peut économiser de nombreuses heures de calculs dans le cas ou les paramètres initiaux étaient incorrects. D’autres applications sont aussi concernées par ce rapprochement. Les environnements virtuels tendent à être de plus en plus complexes, de par leur taille, leur aspect visuel, mais aussi le comportement et les interactions entre les



(a) Sort-last (b) Sort-first (c) Duplication

Figure 4.1 *Différents niveaux de parallélisation d'une application de réalité virtuelle.*

objets présents. Ceci nécessite l'intégration de tâches de calculs lourds. Cette intégration est cette fois beaucoup plus forte, du fait des contraintes d'interactivité de l'application.

Concevoir une application de réalité virtuelle destinée à une plateforme de type grappe nécessite de combiner les fonctionnalités de base (chapitre 2 page 7) tout en gérant leur répartition sur les différentes machines. Ce chapitre étudie tout d'abord les différentes approches de parallélisation utilisées, puis présente les cadres applicatifs (*application framework*) de réalité virtuelle les plus répandus.

4.1 Rendu multi-projecteurs

Le premier objectif de l'utilisation des grappes de PC pour la réalité virtuelle fut le pilotage des environnements multi-projecteurs pour obtenir un affichage cohérent. De nombreux travaux s'y attachent, le cours SIGGRAPH [16, 7] correspondant à l'utilisation des grappes de PC pour la réalité virtuelle présente un large panel. Le problème principal est la gestion du placement des différents périphériques d'entrée et de sortie, qui impose une redistribution des données pour que les informations nécessaires à chaque sortie soient disponibles. Les différentes approches peuvent se distinguer en fonction du niveau utilisé pour implanter cette redistribution [118] dans la chaîne de traitements de l'application (figure 4.1).

4.1.1 Sort-last

Toute application graphique produit au final des pixels. C'est donc à cette dernière étape du pipeline que peut s'effectuer la première approche de parallélisation, appelée *sort-last*. Il est possible de transférer ces pixels sur le réseau pour distribuer aux différentes machines reliées aux vidéo-projecteurs l'image calculée par un serveur conte-

nant l'application (figure 4.1(a)), ou recomposer les images calculées par différentes machines [108, 168, 119].

Cette approche est très indépendante de la structure de l'application générant les images. Elle est applicable pour la majorité des applications. Toutefois elle souffre de problèmes de performances liés à la bande passante nécessaire pour relire les pixels calculés par la carte graphique et les envoyer sur le réseau. De ce fait son utilisation se limite fréquemment à certaines catégories comme le rendu volumique de gros volumes de données.

4.1.2 Sort-first

Une deuxième approche consiste à paralléliser la description des primitives 3D formant la scène utilisée pour le rendu. Une ou plusieurs machines exécutent l'application et produisent les polygones et les textures constituant la scène. Ensuite cette description est envoyée aux machines de rendu, en fonction de la partie d'image visible sur leur vidéo-projecteur respectif (figure 4.1(b)). Cette approche, appelée *sort-first*, est notamment implantée par Chromium [85].

Du point de vue de l'application, Chromium remplace le driver OpenGL de la carte graphique. L'utilisation de cette API permet une compatibilité binaire avec toutes les applications OpenGL. De ce fait cette approche est aussi transparente pour le reste de l'application. Chromium fourni toutefois des extensions n'appartenant pas au standard OpenGL pour supporter la spécification parallèle de la scène à visualiser. Ces extensions permettent à chaque machine de spécifier les contraintes d'ordre entre les différentes parties de la scène, via un mécanisme de barrières et sémaphores. L'utilisation de ces fonctionnalités requiert l'ajout de codes au niveau de l'application, mais cette modification est assez simple.

Pour communiquer les commandes OpenGL depuis la ou les machines exécutant l'application vers les machines effectuant le rendu, un réseau de filtrage de type *data-flow* est utilisé pour appliquer les traitements requis (*frustum culling*, fusion de plusieurs flux en respectant les contraintes d'ordre). Cette partie sera détaillée dans la section 10.2 (page 97).

4.1.3 Graphes de scènes distribués

Les approches précédentes permettent de supporter un grand nombre d'applications mais demandent un réseau rapide. En effet, le volume des communications est proportionnel au nombre de pixels affichés pour l'approche *sort-last*, ou à la complexité de la scène 3D pour le *sort-first*. Pour éviter ce surcoût, d'autres approches travaillent à un niveau plus élevé. Si l'application repose sur un graphe de scène il est possible de le dupliquer partiellement sur les machines de rendu grâce à un protocole de synchronisation des modifications sur les noeuds du graphe [82, 123, 154, 156]. Cela permet de diminuer le surcoût

réseau mais nécessite que l'application utilise l'API du graphe de scène distribué.

4.1.4 Duplication

Une dernière approche consiste à dupliquer entièrement l'application sur chaque machine en diffusant toutes les données modifiant l'état de l'application (périphériques d'entrée, horloge, générateur aléatoire) afin de garder toutes les copies identiques (figure 4.1(c)).

Dans le cas où l'application accède à ces données via une librairie de programmation telle VR Juggler [92, 26] (section 4.2.2), cette duplication peut être implantée de manière transparente pour l'application. C'est ce qui est fait par Net Juggler [10, 11] (section 5.1 page 37), et repris pour VR Juggler 2.0 par Cluster Juggler [134]. Du fait que ces données d'entrée sont assez petites, cette approche a un surcoût en terme de communication faible, ce qui permet de bien passer à l'échelle pour le nombre de machines de rendu. En revanche, du fait que toutes les données et les calculs de l'application sont dupliqués sur chaque machine, cette approche ne permet pas d'accélérer l'exécution des applications trop exigeantes.

4.2 Cadres applicatifs de réalité virtuelle

Faire collaborer tous les outils de bas niveaux nécessaires à une application de réalité virtuelle, tout en supportant une ou plusieurs stratégies de parallélisation n'est pas une chose facile. Comme ce besoin se retrouve dans toute application, des cadres applicatifs génériques sont utilisés afin de factoriser ce travail. Du fait de l'évolution constante des plateformes de réalité virtuelle, il n'existe pas d'API standardisée permettant d'accéder aux fonctionnalités de ces outils. Ainsi, une application est développée pour un cadre particulier. Ils ont chacun une conception et des fonctionnalités propres. Nous allons présenter dans cette section quelques environnements logiciels types.

4.2.1 CAVELib

La librairie commerciale CAVELib [138] fut développée pour piloter les premières CAVE [39]. L'application enregistre ses fonctions d'affichages et d'initialisation qui sont ensuite appelées pour chaque projecteur. A l'origine conçu pour les stations à mémoire partagée ONYX, le rendu est séparé en plusieurs processus pilotant chacun un projecteur. Toutefois, du fait que les ONYX de l'époque étaient limitées à 3 sorties graphiques, CAVELib supporte aussi l'exécution distribuée sur plusieurs stations en dupliquant l'application.

Etant principalement dédiée aux plateformes à mémoire partagée, CAVELib découpe l'application en un ensemble de processus liés à chaque tâche (calculs, entrées, rendu).

L'application a la charge de transférer les données entre ces processus en passant par la mémoire partagée, ou le réseau dans le cas distribué. Ceci permet à chaque tâche de fonctionner à sa propre fréquence, mais alourdi le développement de l'application.

4.2.2 VR Juggler

VR Juggler [92, 26] est un environnement logiciel open-source conçu à l'université de l'IOWA pour développer des applications de réalité virtuelle supportant de nombreuses plateformes (stations SGI, PC sous Linux ou Windows, Mac). Carolina Cruz-Neira, qui a participé au développement de CAVELib pour les premières applications du CAVE [39], a initié le projet VR Juggler pour fournir un cadre d'application plus mature et surtout open-source, permettant le partage des ressources développées dans la communauté de réalité virtuelle. VR Juggler étant utilisé dans plusieurs applications présentées par la suite, nous allons détailler son fonctionnement.

VR Juggler permet à l'application de faire abstraction des périphériques matériels utilisés, et donc de développer plus facilement des applications portables et compatibles avec un grand nombre de périphériques. Il offre une grande liberté quand à l'architecture de l'application et la méthode d'affichage, tout en fournissant les fonctionnalités transversales comme le calcul du point de vue pour la stéréo traquée, ou la spécification du placement des différentes surfaces d'affichages.

De manière interne VR Juggler est organisé sous forme d'un micro-noyau contrôlant la boucle d'exécution de l'application et faisant appel à un ensemble de *managers* pour gérer chaque fonctionnalité. Ces managers sont :

Config Manager : Gère toute la configuration de l'application, ainsi que les possibles re-configurations en cours d'exécution.

Performance Manager : Permet de tracer l'exécution de l'application pour obtenir des statistiques de performances.

Input Manager : Contrôle les périphériques d'entrées utilisés. Les différents drivers sont chargés par un mécanisme de plugins.

Display Manager : Fourni les fonctionnalités de gestion de l'affichage indépendantes de l'API utilisée et de la plateforme (particulièrement le calcul des matrices de projections).

Draw Manager : Implante le rendu en fonction de la plateforme et de l'API utilisée par l'application. Différentes versions de ce manager sont chargées en fonction de l'application.

Les applications sont développées en étendant une classe particulière en fonction de l'API graphique utilisée. Actuellement, les API supportées sont OpenGL, Performer, OpenSG et OpenSceneGraph. la plupart des périphériques d'entrées sont supportés, soit par des drivers intégrés (parfois développés par d'autres équipes), soit par des liaisons avec Trackd ou VRPN (section 2.1 page 9).

De plus, VR Juggler fourni des outils graphiques permettant d'éditer la configuration de l'application et de visualiser les mesures de performances, ainsi qu'un mode simulé permettant de tester des applications conçues pour des environnements immersifs sur une machine de bureau.

L'équipe de développement de VR Juggler ainsi que la communauté des utilisateurs sont très actives. De nombreuses évolutions sont apparues au cours des dernières années. Ainsi, VR Juggler 1.0 ne supporte pas l'exécution d'une application sur une grappe. VR Juggler 2.0, dont la version stable est disponible depuis juillet 2005, ajoute ce support, soit de manière interne par *duplication* (section 4.1.4), soit en utilisant les implantations distribuées d'OpenGL (section 4.1.2) ou d'OpenSG (section 4.1.3).

4.2.3 Syzygy

Syzygy [156] est un cadre d'application développé spécialement pour les plateformes à base de grappes. Il est basé sur une couche d'abstraction *Phleet* gérant les machines participant à l'application. Ces machines peuvent être hétérogènes et distantes. Les fonctionnalités d'entrées/sorties et de rendu distribué sont implantées au dessus de cette abstraction. Ainsi, Syzygy supporte un rendu parallélisé via un graphe de scène propre ou par duplication de l'application. L'application choisi le mode qui lui convient.

D'une diffusion moins large que VR Juggler, Syzygy est tout de même utilisé dans plusieurs centres de recherches et musées, pour des applications principalement de collaboration artistique distante.

4.2.4 OpenMASK

OpenMASK [109], un outil principalement dédié aux simulations multi-agents (section 2.3.1 page 11), comporte aussi des fonctionnalités de cadre d'application en intégrant le support du rendu via Performer [152] ou OpenSG [146]. Contrairement aux travaux précédents, OpenMASK considère l'application non pas comme un seul programme monolithique, mais comme un ensemble d'objets communiquants. Ces objets correspondent aux entités de l'environnement virtuel et peuvent interagir en s'échangeant des valeurs ou des évènements. Le moteur d'exécution d'OpenMASK calcule un ordonnancement des calculs en fonction de la fréquence d'activation des différents objets. Plusieurs implantations sont proposées, supportant ou non la répartition des calculs sur une grappe.

Pour permettre la visualisation de l'environnement, OpenMASK utilise un graphe de scène mis à jour par l'état des objets. La parallélisation du rendu peut être faite soit en communiquant cet état à plusieurs machines, soit en utilisant la parallélisation du graphe de scène OpenSG [154].

4.3 Bilan

Ce chapitre a présenté les principes et les outils permettant de construire des applications de réalité virtuelle parallèles. De part le rapport coût/performances des grappes et l'aspect open-source de la plupart des réalisations, ces éléments permettent un accès plus facile aux plateformes de réalité virtuelle. Toutefois la complexité de mise en oeuvre des applications reste importante. Par exemple, la gestion de la distribution est souvent soit à la charge du programmeur, soit réalisée de manière transparente au détriment des performances (communications sur le réseau de données bas niveaux).

Bien que les cadres applicatifs permettent de réutiliser les outils de bas niveaux et les techniques de parallélisation, le reste de l'application est souvent spécifique, en particulier les calculs liés à l'environnement virtuel et aux fonctionnalités de haut niveau. A part dans le cas d'OpenMASK, l'application est construite de façon monolithique. Dans tous les cas sa structure est déterminée par le cadre applicatif utilisé, ce qui rend l'intégration de codes existants difficile. Cependant, l'utilisation de ces cadres pour construire une application de réalité virtuelle est tout de même très bénéfique, car les fonctionnalités de bas niveau requises sont nombreuses pour gérer tous les périphériques utilisés. En effet, il est non seulement nécessaire de recevoir ou d'envoyer les données à chacun des périphériques, mais il faut aussi gérer l'asynchronisme entre leurs différentes fréquences de fonctionnement.

L'utilisation des travaux de couplage de codes (section 3.2 page 21) permettrait le développement d'applications de réalité virtuelle à plus grande échelle. En effet la plupart des applications actuelles utilisent un environnement virtuel simple, parfois complètement statique, ou alors dont seulement quelques objets sont interactifs. Le support d'environnements plus riches et dynamiques nécessite un important effort de développement, qui n'est pas justifiable s'il ne peut être réutilisé par la suite. Toutefois, comme cela a été discuté dans le bilan du chapitre 3, l'intégration des outils de couplage de codes n'est pas immédiate, du fait des besoins particuliers de la réalité virtuelle, en particulier au niveau des contraintes d'interactivité et de l'asynchronisme entre les différents éléments.

II

FlowVR

Après un chapitre d'expérimentations préliminaires, cette partie présente la contribution principale de cette thèse : un modèle de couplage de composants parallèles ainsi que l'implantation associée pour construire des applications interactives. L'originalité repose sur la possibilité d'exprimer des motifs de couplages avancées, permettant de désynchroniser les différentes parties de l'application tout en respectant les contraintes de cohérence exigées par l'utilisateur. Les connexions entre composants ainsi que ces contraintes de cohérence sont définies par un graphe de flux de données, comportant des objets particuliers implantant les opérations de filtrage et de synchronisations nécessaires. Ces considérations étant extérieures aux code des composants, ce modèle permet une bonne réutilisabilité des composants ainsi qu'une construction très modulaire des applications.

Ce travail a été publié lors de la conférence Euro-Par 2004 [9]. Les applications et expérimentations développées ont fait l'objet de publications à IEEE VR 2002 [10], Euro-Par 2003 [14] et IPT/EGVE 2002 [11], 2003 [8], 2004 [5] et 2005 [6].

Travaux préliminaires 5

Dans le cadre de mon stage de maîtrise au laboratoire *LIFO* de l'Université d'Orléans, des premiers travaux ont été effectués pour pouvoir exécuter des applications de réalité virtuelle en environnement multi-projecteurs piloté par grappe de PC utilisant des composants standards. Ceci a abouti à un ensemble d'outils offrant une solution à ce problème. Ce fut l'une des premières disponibles. Plusieurs publications [7, 8, 10, 11, 14] ont contribué à la diffusion de ce travail.

Le résultat de ce travail a servi de base pour le début de cette thèse, avec des premières expérimentations testant le couplage d'une simulation parallèle et interactive à de telles applications.

5.1 Net Juggler : duplication transparente d'applications VR Juggler

Dans le cadre de la réalité virtuelle, une grappe de PC sert tout d'abord à piloter les multiples vidéo-projecteurs nécessaires à la surface d'affichage. Pour cela, chaque PC est relié à un ou deux projecteurs et il faut donc qu'il calcule l'image correspondante. Plusieurs techniques sont possibles (section 4.1 page 28), la plus simple à implanter et la plus performante est la *duplication* de l'application sur chaque machine de rendu. C'est cette méthode que nous avons exploré dans le cadre de *Net Juggler* [10].

5.1.1 Exécution sur grappe

VR Juggler 1.0 (section 4.2.2 page 31), qui était la version disponible au moment de ce travail, ne supportait pas l'exécution d'une application sur une grappe. Net Juggler ajoute donc ce support en dupliquant VR Juggler et l'application sur chaque machine et en utilisant l'abstraction des périphériques d'entrée fournie par VR Juggler pour les synchroniser entre toutes les machines de manière transparente (*datalock*). Les communications ainsi que le lancement de l'application utilisent MPI (section 3.1.1 page 18).

Le système de configuration de VR Juggler est modifié pour ajouter une information de placement de chaque périphérique, tels que les périphériques d'entrée ou les projecteurs. Ensuite chaque machine calcule l'image correspondant au projecteur local. Juste avant d'afficher chaque nouvelle image, une barrière est utilisée afin de synchroniser les différents projecteurs (*swaplock*).

Du fait de l'organisation interne de VR Juggler sous forme de micro-noyau faisant appel à un ensemble de *managers* pour gérer chaque fonctionnalité, les modifications requises par Net Juggler furent assez simple à implanter. En revanche cette approche impose deux limitations importantes :

- Toutes les informations ayant une influence sur l'état de l'application doivent être diffusées à l'ensemble des machines. Les données d'entrée sont gérées par Net Juggler, mais d'autres informations comme l'horloge ou le générateur pseudo-aléatoire peuvent introduire des différences entre machines. Pour les corriger il est possible de les transformer en périphérique d'entrée. Ainsi Net Juggler fourni un périphérique d'horloge globale.
- Du fait de la barrière de *swaplock*, la vitesse de l'application dépend de la machine la plus lente. Si toutes les machines sont identiques et comme les mêmes calculs sont dupliqués sur chaque machine (mise à part la modification de point de vue lié au projecteur), alors les performances sur la grappe seront proches des performances obtenues sur une seule machine (affichant sur un seul projecteur), moins le surcoût des communications réseau (en général très faible).

5.1.2 Résultats

Du fait de la faible quantité d'informations à transmettre entre les machines (généralement quelques matrices et évènements d'appuis sur des boutons), le surcoût réseau est très faible, ce qui fait que les performances obtenues sont proches de celles observées sur une seule machine. L'avantage est que les performances sur la grappe sont faciles à anticiper en testant sur une seule machine, en revanche cela ne permet pas d'accélérer une application trop exigeante, mais uniquement d'augmenter la résolution (le nombre de projecteurs).

La figure 5.1 présente l'application CAVE Quake III Arena [144] exécutée sur une grappe de PC grâce à Net Juggler. Les performances obtenues sur une machine sont de

Figure 5.1 *CAVE Quake III Arena exécuté sur une grappe grâce à Net Juggler et SoftGenLock [8].*

38 images par secondes. Sur 4 machines avec un réseau Myrinet Net Juggler obtient 37 images par secondes. Le surcoût lié à l'exécution sur la grappe est donc assez faible.

Pour le cas des applications coûteuses en calculs, du fait de l'utilisation de MPI par Net Juggler il est relativement facile de l'utiliser aussi pour répartir les calculs de l'application. Ceci permet de développer des applications intégrant des simulations interactives. Une application exemple est décrite dans la section 5.2.1.

5.2 Simulations interactives sous Net Juggler

Intégrer une simulation parallèle, généralement d'un phénomène physique, dans une application de réalité virtuelle peut être utile pour deux objectifs différents. Cela permet d'explorer les résultats de la simulation et ainsi mieux les comprendre, avec parfois la possibilité d'influer sur la simulation de manière interactive. Mais la simulation peut aussi être intégrée à un environnement virtuel de manière à le rendre plus dynamique et plus riche, donc plus immersif. Dans les deux cas les problèmes sont assez similaires, liés au couplage entre la partie simulation et la partie visualisation. Mais l'accent est mis soit sur la précision de la simulation soit sur ses performances et sur la latence de la boucle d'interaction.

5.2.1 Simulation de fluide 2D interactive

Une première approche pour intégrer une simulation parallèle à une application Net Juggler est l'utilisation de l'environnement MPI déjà intégré. C'est l'approche retenue pour l'une des premières applications développées, intégrant une simulation de fluide

Figure 5.2 *Rendu de l'application de simulation de fluide 2D interactive.*

parallélisée interactive [11].

La simulation de fluide est basée sur l'algorithme de résolution des équations de Navier-Stokes proposé par Stam [165, 166]. L'espace est discrétisé par une grille de cellules. Chaque cellule possède un vecteur vitesse du fluide ainsi qu'une valeur de densité caractérisant le fluide présent dans la cellule. A chaque pas de simulation, l'algorithme met à jour ces valeurs. Ce calcul utilise des opérations matricielles simples ainsi qu'un gradient conjugué et une résolution des équations de Poisson.

L'implantation utilise PETSc [20], une librairie mathématique fournissant des opérations matricielles parallélisées par MPI. Toutes les matrices sont réparties sous forme de blocs sur les différentes machines, et les valeurs aux frontières sont communiquées après chaque calcul.

La plupart des implantations de MPI ne supportant pas le *multi-threading*, la simulation est placée dans le même thread que la visualisation. De ce fait, elles sont exécutées de manière synchrone, c'est-à-dire que leur fréquence de rafraîchissement est identique. Cette approche est surtout adaptée au cas de l'intégration d'une simulation de relativement faible échelle dans un environnement virtuel. Une simulation trop coûteuse serait un obstacle à l'interactivité de la visualisation.

Cette simulation de fluide, intégrée dans un environnement virtuel pour former un lac dans une vallée, est visible sur la figure 5.2. L'utilisateur interagit en déplaçant un pointeur qui applique des forces au fluide.

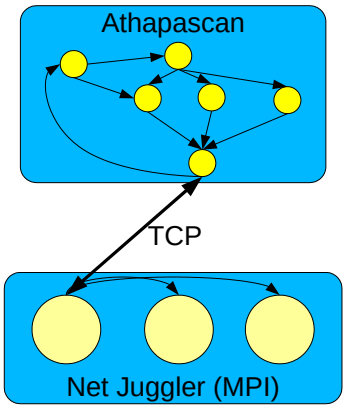


Figure 5.3 *Schéma de couplage entre Net Juggler et Athapascan.*

5.2.2 Simulation de tissus préexistante

Il existe souvent des codes de simulation déjà opérationnels pour des calculs non-interactifs et il semble très intéressant de les réutiliser dans un environnement interactif. Pour tester cette approche, nous avons intégré le travail de Florence Zara sur la simulation de tissus [184, 185] avec une visualisation distribuée par Net Juggler. Cette simulation utilise Athapascan (présenté dans la section 3.1.3 page 20) pour distribuer le calcul. Du fait de l'utilisation de deux paradigmes de parallélisation différents entre la visualisation (répartition statique avec passage de messages MPI) et la simulation (ordonnancement et répartition dynamiques gérés par Athapascan), ces deux parties sont implantées séparément.

Le tissus est discrétisé en un maillage de particules reliées par des ressorts. Pour paralléliser le calcul, ce maillage est partitionné en blocs qui sont ensuite répartis sur les processeurs de manière automatique par Athapascan. Les positions sont mises à jour à chaque pas de simulation par une intégration des équations de Newton.

Le couplage entre la simulation et la visualisation utilise un simple canal TCP, comme indiqué sur la figure 5.3. Ceci implique une centralisation des données vers le noeud qui possède le canal du coté de la simulation, et ensuite une diffusion vers tous les noeuds de visualisation des données reçues. En retour les mouvements de la souris sont envoyés à la simulation et servent à bouger un coin du drap de manière interactive.

Contrairement à l'exemple du fluide, il est possible d'introduire de l'asynchronisme entre la simulation et la visualisation, par exemple en prenant en compte à chaque image uniquement le dernier résultat disponible. Ceci permet de supporter des calculs plus complexes du côté de la simulation.

Cette approche est très simple mais a donné de bons résultats que nous avons présentés lors d'une démonstration interactive pour la conférence Euro-Par 2003 [14]. La figure 5.4 présente une capture d'écran de cette application.

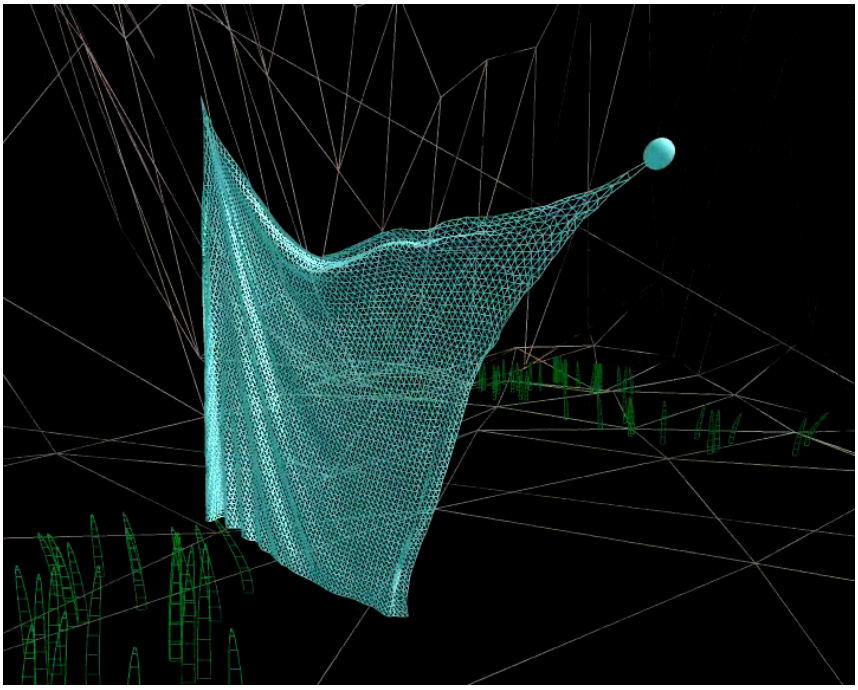


Figure 5.4 *Rendu de l'application de simulation de drap interactive.*

5.3 Bilan

Ce travail a abouti à plusieurs applications couplant une visualisation multi-projecteurs avec plusieurs simulations parallèles. Les performances obtenues sont suffisantes pour une bonne interactivité avec chaque simulation. Les deux applications présentées (simulation de fluide et de tissus) ont même abouti à une seule application intégrant les deux simulations dans le même environnement virtuel. Une vidéo démontrant ce résultat est disponible : `http://www-id.imag.fr/~allardj/these/valley2003.avi`.

Toutefois un certain nombre de problèmes sont apparus. Tout d'abord l'implantation du couplage est spécifique à chaque application. Il doit donc être réécrit à chaque fois qu'un nouvel élément est ajouté. De plus il est difficile d'augmenter considérablement la taille des simulations. En effet, pour le cas du fluide une simulation plus complexe ralentirait d'autant la visualisation. Pour le tissus la liaison vers la visualisation par un canal TCP introduit une centralisation des données qui limite le passage à l'échelle.

Enfin, un autre problème important apparaît lorsqu'on introduit un couplage asynchrone. En effet, le fait de ne plus mettre à jour tous les objets de l'environnement à la même fréquence introduit des décalages entre les objets qui devraient être associés, comme le pointeur de la souris et le coin du drap (figure 5.5). Ce problème est lié aux contraintes de cohérence attendues par l'utilisateur entre les différents objets de l'environ-

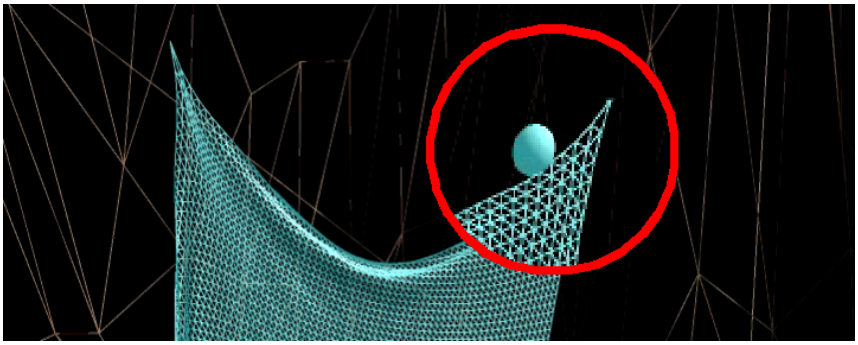


Figure 5.5 *Problèmes de cohérence de la scène.*

nement. Plusieurs approches peuvent être utilisées pour le résoudre, comme par exemple retarder la mise à jour du pointeur de la souris pour être synchronisé avec le pas de temps de la simulation du tissus (au détriment de la latence des mouvements de souris), ou encore afficher un lien entre le pointeur et le coin du drap pour matérialiser un ressort les reliant (et ainsi assouplir la contrainte de liaison entre ces objets). Néanmoins ces modifications sont spécifiques à l'application et ne répondent pas au problème fondamental des contraintes de cohérence dans les systèmes distribués interactifs.

Un modèle d'application distribuée interactive 6

"IF COMPUTERS GET TOO POWERFUL, WE CAN ORGANIZE THEM INTO A COMMITTEE – THAT WILL DO THEM IN."
Bradley's Bromide

6.1 Introduction

Les expériences présentées dans le chapitre 5 permettent de dégager la problématique de notre travail, à savoir le besoin d'un outil générique permettant le couplage de composants parallèles à l'intérieur d'une application distribuée et interactive. Pour permettre une recherche des solutions les plus adaptées et de par la diversité des applications et la complexité du problème, ce modèle doit être simple et facilement modifiable. Les performances à l'exécution sont primordiales, en particulier en ce qui concerne la latence des boucles d'interactions. Enfin, et c'est là une spécificité de ces applications, les contraintes de cohérence de la scène virtuelle doivent être intégrées et modifiables en fonction des souhaits de l'utilisateur afin d'adapter les compromis entre synchronisations et performances.

Ce chapitre présente le modèle adopté en discutant les choix effectués par rapport aux différentes options possibles, avant de le détailler plus formellement.

6.2 Le modèle choisi

La base de notre approche est un découpage de l'application en *modules* communiquant entre eux. En fonction de la granularité de ce découpage, chaque *tâche* de l'application (entrées, simulations, visualisation, ...) peut être constituée d'un ou plusieurs

modules. Une tâche peut être parallélisée et réutilise fréquemment un code ou un algorithme préexistant. Le modèle doit donc faciliter leur intégration au reste de l'application, en particulier en limitant les modifications nécessaires.

Le rôle principal du modèle doit être de gérer les flux de données entre les modules en respectant les contraintes de performance et de cohérence de l'application.

Pour respecter les contraintes de performance il faut pouvoir répartir et utiliser efficacement les ressources disponibles (réseau, calculs, mémoire). Pour supporter de données volumineuses et diminuer la latence il faut par exemple réduire au maximum les recopies des données. Pour imposer les contraintes de cohérence il faut introduire certains mécanismes de synchronisation entre les flux de données.

Les caractéristiques de performances (débit, latence) et de synchronisation (cohérence) de l'application sont très liées. Par exemple, en parallélisant une tâche via un découpage en pipeline le débit est amélioré mais la latence peut être augmentée étant donné le surcoût des communications entre chaque étape. De même, une contrainte forte de synchronisation entre plusieurs machines peut entraîner des mises en attentes et donc une perte de débit. Il n'est en général pas possible d'utiliser une même approche pour toutes les applications et notre modèle doit donc permettre de spécifier des techniques différentes en fonction de l'application, des ressources disponibles et des attentes de l'utilisateur.

Pour pouvoir tester les différentes approches et construire rapidement des applications expérimentales, il doit être possible de changer facilement les techniques employées. En particulier certaines tâches peuvent supporter plusieurs modes de parallélisation (répartition des données, répartition des itérations, pipeline). Le modèle doit permettre de changer ce mode pour une tâche de l'application sans que cela n'affecte les autres tâches.

6.2.1 Granularité du découpage

Le niveau auquel s'effectue le découpage de l'application en modules est primordial pour déterminer les fonctionnalités de l'application. On peut distinguer plusieurs niveaux et leurs conséquences sur l'expression d'une boucle de simulation distribuée :

Opération de base : le calcul est exprimé sous forme d'une suite d'opérations matricielles, traitements sur les données, etc. Ces opérations sont parallélisées et réparties sur les machines.

Boucle de calcul (*thread*) : le calcul est exprimé sous forme d'un ensemble de boucles de calcul, chacune affectée à une machine.

Tâche parallèle : la tâche complète est vue comme un seul composant occupant un groupe de machines.

Un découpage de très bas niveau introduit de très nombreux éléments à gérer, ce qui peut permettre d'exploiter pleinement le parallélisme mais entraîne un surcoût important et une contrainte forte pour l'implantation de l'application qui doit être faite en fonction de ce découpage. Des composants de plus haut niveau rendent l'implantation de chaque

composant responsable de sa parallélisation, ce qui nécessite d'utiliser d'autres outils et méthodes pour le faire, mais permet de choisir celui qui convient le mieux.

Du fait de la prédominance des boucles dans une application interactive, cela semble être une base de découpage intéressante. Chaque composant, que nous appellerons *module*, est un calcul itératif exécuté sur une machine donnée. Plusieurs modules peuvent former une tâche parallèle en utilisant des communications internes. Bien que ce regroupement en tâches parallèles est utile comme concept de haut niveau, du point de vue du modèle il est souhaitable de ne pas le faire apparaître, pour traiter de manière similaire un ensemble de modules, qu'il soit lié par des communications internes ou non.

6.2.2 Connaissance des modules sur le reste de l'application

Une fois l'application découpée en modules, ils doivent être combinés suivant les fonctionnalités requises. Cela se traduit par des échanges de données et/ou de commandes entre ces modules. Ces échanges peuvent être fait de manière totalement transparente aux modules, c'est-à-dire qu'ils n'ont aucune connaissance de la provenance ni la destination des données qu'ils utilisent. Il peuvent alternativement êtres conscients des modules avec lesquels ils sont reliés, ce qui peut leur permettre d'adapter au mieux leur comportement, mais introduit une dépendance plus forte entre module. Cette connaissance du reste de l'application peut être uniquement locale (i.e. uniquement les modules directement reliés) ou alors globale, ou chaque module a conscience de tout le reste de l'application et peut donc directement contacter n'importe quel autre module. Cette décision a un impact fort sur l'interdépendance des modules et les fonctionnalités qui leurs sont offertes pour interagir avec l'application.

Pour favoriser la modularité du système ainsi que la simplicité de sa programmation, il semble désirable que les modules n'aient aucune connaissance des autres modules de l'application.

6.2.3 Configuration de l'application

Pour mettre en place l'application il est nécessaire de spécifier les modules à instancier ainsi que les connexions qui les relient. Cette spécification peut être effectuée par différents intervenants :

- le récepteur : chaque module indique explicitement où il va récupérer les informations qu'il utilise ;
- l'émetteur : de manière opposée le module qui produit une donnée peut spécifier où elle doit être transmise ;
- un autre module : un module particulier peut spécifier les connexions entre les autres modules de l'application.

De plus, les connexions entre les modules peuvent être soit statiques, c'est-à-dire spécifiées lors du démarrage de l'application puis inchangées, soit dynamiques, c'est-à-dire modifiables à n'importe quel moment.

Etant donné que les modules de l'application n'ont pas connaissance des autres modules, ils ne peuvent établir directement les connexions. Un module particulier appelé *contrôleur* est donc utilisé pour créer l'application en lançant les autres modules. Du fait qu'il a connaissance de tous les modules qu'il a créé, il peut ensuite spécifier les connexions entre-eux.

6.2.4 Synchronisation et cohérence

Le niveau de synchronisation utilisé dans une application distribuée a d'importantes conséquences sur l'extensibilité de l'application ainsi que la cohérence du résultat obtenu. Par exemple, un modèle de type BSP [178] permet d'obtenir un résultat déterministe mais au prix d'opérations de synchronisations globales qui peuvent introduire un surcoût important. En particulier pour les applications interactives, il est important que les performances d'une partie de l'application n'affectent pas outre mesure les autres parties. Cela nécessite donc de découpler les différents modules en autorisant des fréquences d'activation différentes, tout en respectant les contraintes demandées par l'application. La première conséquence se situe au niveau des flux de données entre modules. Il est nécessaire de modifier ces flux, par exemple en sélectionnant une partie des données (échantillonnage) ou en interpolant les données. Cette opération peut être effectuée à plusieurs niveaux :

- le récepteur : chaque module peut spécifier la méthode de traitement de ses données, voir directement l'implanter s'il a accès à l'historique des messages ;
- un autre composant : de la même façon que les connexions, les méthodes de filtrage peuvent être spécifiées par un composant particulier ;
- déduite de contraintes : l'utilisateur peut spécifier ses exigences sous formes de contraintes de cohérence que le système aura la responsabilité d'implanter.

La fréquence d'activation de chaque module peut être soit implicite via filtrage des données, c'est-à-dire que chaque module est activé quand les filtres dont il dépend envoient des nouvelles données, soit explicite via un algorithme d'ordonnancement particulier, et dans ce cas le filtrage des données devra agir en fonction de cet algorithme.

Dans le but de réutiliser un même module dans une application interactive ou non, il faut que celui-ci soit indépendant des mécanismes de synchronisation avec les autres modules. De ce fait, nous introduisons de nouveaux objets appelés *filtres* et *synchroniseurs*, placés sur les connexions entre modules pour implanter ces mécanismes. L'ordonnancement des modules ainsi que les politiques de couplage associées sont entièrement spécifiés par les mécanismes de filtrage des données d'entrée de chaque module. Ainsi,

chaque module attend au début de chaque itération un nouveau message sur chacun de ses ports d'entrées. Cette attente est implicitement contrôlée par les filtres et synchroniseurs présents.

6.2.5 Placement et allocation des ressources

Une fois la structure de l'application en place, il est nécessaire de l'instancier en fonction des ressources disponibles. Ce placement peut être spécifié directement à la construction de l'application, ou bien déduit à partir du placement de certains modules (périphériques, données). Il peut être optimisé en se basant sur un modèle de la plateforme paramétré, par exemple, par des mesures de performance. Enfin, ce placement peut être statique ou dynamique, ce qui implique dans ce dernier cas de disposer d'opérations de migration des modules de l'application.

Bien que ces fonctionnalités soient intéressantes, elles ne sont pas primordiales en environnement homogène de type grappe. Dans un premier temps la spécification de l'application est donc entièrement statique. Il est impossible de modifier les connexions ou les modules après le démarrage de l'application. Une extension permettant plus de souplesse pourra être envisagée par la suite, par exemple en exploitant des environnements autorisant la migration transparente de processus de type Kerrighed [120].

6.3 Exemple d'application

Avant la présentation formelle du modèle FlowVR, nous allons l'illustrer en étudiant le cas de l'application de simulation de fluide parallèle interactive présentée à la section 5.2.1 (page 39).

La structure de cette application (figure 6.1(b)) comporte trois modules :

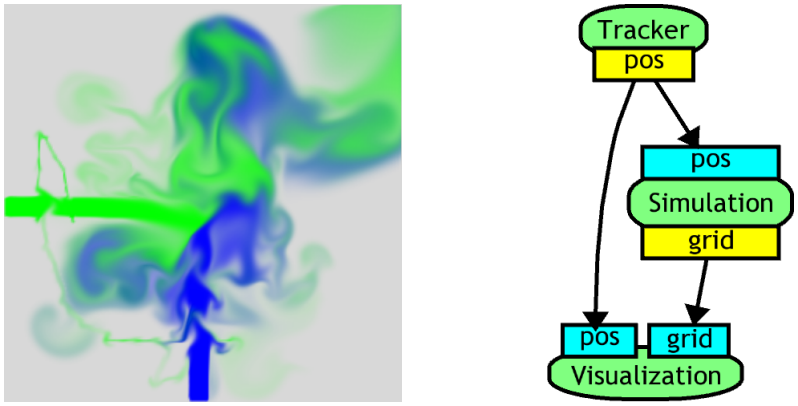
Tracker : récupère la position de la souris.

Simulation : mets à jour la densité du fluide en fonction des mouvements de la souris.

Visualization : affiche le curseur de la souris et le résultat de la simulation.

Pour représenter graphiquement les applications, nous utiliserons dans la suite les conventions suivantes :

- les modules sont représentés par des ovales de couleur verte ;
- les données d'entrée sont représentées par des rectangles bleus au dessus du module associé ;
- les données de sortie sont représentées par des rectangles jaunes au dessous du module ;
- les connexions sont représentées par des flèches noires.



(a) (b)

Figure 6.1 *Structure de l'application exemple.*

6.4 Des modules réutilisables

6.4.1 Définition d'un module

Un module est un programme (dans le sens processus ou thread) s'exécutant sur une machine donnée. Chaque module de l'application est différencié par un *identifiant unique*. Pour échanger des informations avec les autres modules, il déclare une liste de *ports d'entrée* et de *ports de sortie*. Chaque port est défini par un nom et optionnellement un type de donnée.

Une tâche parallèle donne lieu à plusieurs processus ou threads de calculs. Chacun d'entre-eux se traduit par un module différent dans l'application. Ces modules possèdent leurs propres ports. Dans la plupart des cas ces ports seront identiques sur tous les modules, mais cela peut ne pas être le cas. Par exemple, une donnée d'entrée peut n'être requise que sur le premier module, alors qu'un résultat peut être produit sur chaque module, mais de façon découpée. Dans le cas de l'exemple, la simulation de fluide est parallélisée avec MPI, chaque processus donnant lieu à un module avec un port d'entrée *pos* et un port de sortie *grid*. Le résultat calculé par chaque module correspond au bloc de la grille 2D. Si cette simulation est découpé en 4 processus, cela correspond du point de vue du reste de l'application à 4 modules (figure 6.2).

6.4.2 Données échangées par les modules

Les ports d'entrée et de sortie des modules communiquent des données sous forme de *messages*. Un message est un bloc de données associé à un certain nombre d'informations sémantiques appelées *estampilles*. Ces estampilles sont définies par un nom et un type de donnée (parmi *int, float, string, array*). Tous les messages contiennent au moins les

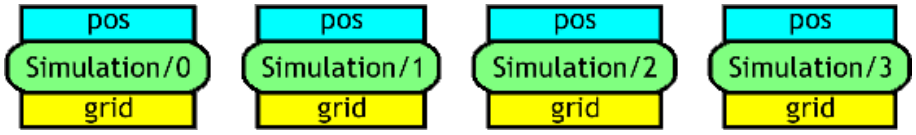


Figure 6.2 *Modules de simulation parallélisée sur 4 machines.*

estampilles suivantes :

source de type *string* : l'identifiant de la source du message (nom du module et du port ayant produit ce message).

it de type *int* : le numéro d'itération du module source lors de l'envoi du message.

num de type *int* : le numéro du message dans le flux de données.

Ces estampilles permettent d'identifier certaines propriétés du messages, sans avoir à l'analyser. Elles sont utilisées pour implanter les opérations de routage et de filtrage sur les messages. Par exemple, **source** permet de déterminer à qui envoyer le message en regardant dans le graphe de l'application les connexions partant de la source. **Num** permet de s'assurer que les messages sont reçus dans le bon ordre. **It** peut être utiliser pour retrouver les dépendances, c'est-à-dire les données utilisées pour le calcul d'un message donné. En effet, les modules consommant un message sur chacune de leur entrée à chaque itération, grâce au numéro d'itération it d'un message on peut retrouver les données utilisées en recherchant dans les flux de messages en entrée du module le message dont le numéro num correspond au it cherché. Ce type de calcul sera utile par la suite pour gérer la cohérence entre flux de données.

En plus de ces estampilles prédéfinies l'utilisateur peut ajouter d'autres informations utiles comme les dimensions des données si celle-ci sont sous forme de vecteurs ou matrices, ou encore la boite englobante d'un modèle 3D. Ces estampilles seront lisibles par tous les objets de l'application FlowVR et permettront d'effectuer des communications complexes adaptées à l'application (frustum culling par exemple). Cependant les données elles-mêmes seront toujours considérées comme des données binaires opaques, permettant ainsi aux modules d'utiliser le format le plus approprié sans affecter le système.

6.4.3 Opérations utilisées par les modules

Comme défini dans la section 6.2.2 (page 47), les modules n'ont qu'une vision locale de l'application. Ainsi les opérations qu'ils ont à leur disposition se limitent au minimum nécessaire pour remplir leur rôle. Cela permet d'avoir un modèle simple du fonctionnement des modules, ce qui est important car ce modèle fait la liaison entre l'étape de création des modules et celle de la construction de l'application les utilisant.

Les cinq premières opérations sont les plus importantes et sont indispensables. Les deux restantes permettent d'optimiser les performances du module.

6.4.3.1 *init()*

Avant de pouvoir travailler, un module doit s'initialiser via l'opération ***init()***. Son rôle est notamment de spécifier l'identifiant du module ainsi que la liste des ses ports d'entrée et ses ports de sortie. Cette opération n'est pas bloquante. En particulier elle n'attend pas que le reste de l'application soit initialisé. Il n'est donc pas possible de lire des informations sur les ports avant le premier appel à ***wait()***.

6.4.3.2 *wait()*

Les modules reposant sur un modèle itératif, l'opération ***wait()*** est utilisée pour démarrer une nouvelle itération. Sa sémantique est la suivante :

---

pour tout les ports d'entrée **faire**
  si ce port est connecté **alors**
    Attendre un nouveau message sur ce port

Cette opération bloque donc le module jusqu'à ce qu'il puisse effectuer l'itération suivante, ce qui est déterminé par les messages en entrée. La disponibilité de ces messages est liée au filtrage effectué en dehors du module. La nature des contraintes résolues par ce filtrage n'est pas visible pour le module. C'est cette séparation qui permet de rendre le module indépendant des méthodes de couplage choisies pour une application donnée. A titre d'exemple, ces contraintes peuvent être une fréquence d'activation ou une barrière entre plusieurs modules (section 6.6.1.1 page 60).

En cas d'erreur ou de fin de l'application, l'opération ***wait()*** renvoie une valeur nulle.

6.4.3.3 *get()*

L'opération ***get()*** s'effectue sur un port d'entrée et permet de lire le message courant reçu sur ce port. Elle ne peut être appelée qu'après le premier ***wait()***. C'est une opération non bloquante qui se contente de récupérer le message reçu lors du ***wait()*** précédant. En particulier, entre deux ***wait()*** le résultat de ***get()*** sur un port donné ne changera pas. Si le module n'appelle pas ***get()*** sur un port d'entrée l'opération ***wait()*** attendra tout de même un message sur ce port.

Il est possible qu'un port d'entrée ne soit pas connecté, dans ce cas ***get()*** renverra un message invalide. Le module doit savoir gérer ce cas. Il peut utiliser une donnée par défaut, ou produire une erreur et terminer son exécution.

6.4.3.4 *put()*

Pour transmettre ses résultats le module utilise l'opération ***put()*** en spécifiant le port de sortie ainsi que le message à envoyer. C'est une opération non bloquante. Lors d'une itération il n'est possible d'envoyer qu'un seul message sur chaque port de sortie. Si ***put()*** n'est pas appelé aucun message n'est envoyé. Cette contrainte d'un seul message par itération permet de gérer les contraintes sur la taille des buffers de communication lors de chaque ***wait()***.

Dans le cas ou le port de sortie n'est pas connecté, le message est silencieusement supprimé. Pour éviter de faire des calculs inutiles dans ce cas, il est possible de le détecter via l'opération ***isConnected()*** décrite dans la section 6.4.3.7.

6.4.3.5 *close()*

Symétriquement à ***init()***, l'opération ***close()*** permet de déclarer la fin de l'exécution du module. Aucune autre opération ne peut être utilisée après l'appel à ***close()***.

6.4.3.6 *alloc()*

Pour éviter les recopies inutiles de données, il est souhaitable d'utiliser un système de mémoire partagée qui permet, en particulier pour le cas ou plusieurs modules se trouvent sur une même machine, de ne transmettre que le pointeur sur ces données. L'inconvénient de cette approche est qu'il n'est pas possible d'utiliser l'allocateur de mémoire standard mais il faut fournir une opération permettant de faire ces allocations dans la mémoire partagée. C'est le rôle de l'opération ***alloc()***, qui alloue un bloc de taille donnée et renvoie le pointeur vers ce bloc.

Comme une même donnée peut être accédée par plusieurs modules simultanément, il est nécessaire d'utiliser une synchronisation de ces accès. Nous utilisons ici une politique très simple : le module qui alloue un bloc via ***alloc()*** peut le modifier librement jusqu'à ce qu'il exécute ***put()***. A partir de ce moment aucune modification n'est permise sur ce bloc. Les données reçues par ***get()*** ne peuvent être que lues. Il est par contre permit d'envoyer via ***put()*** des données en lecture seule, comme dans le cas d'un envoi d'une même donnée sur plusieurs ports ou de la retransmission d'une des données d'entrée.

6.4.3.7 *isConnected()*

Il arrive fréquemment qu'un calcul produise plusieurs résultats. Dans l'exemple de la simulation de fluide il y a une grille de densité du fluide ainsi qu'une grille des vecteurs vitesse. Pour rendre chacun de ces résultats disponibles au reste de l'application, le module déclare plusieurs ports de sortie. En fonction de l'application seuls certains ports seront réellement utilisés. Cela ne pose pas de problème étant donné que l'opération ***put()*** supprime les messages envoyés sur des ports non connectés, mais cela introduit

un certain nombre d'opérations inutiles pour construire ces messages, voir même pour en calculer les données. Pour économiser ces opérations le module peut utiliser l'opération ***isConnected()*** qui détermine si un port donné est connecté dans l'application. Cette information n'est disponible qu'après le premier appel à ***wait()***.

6.4.4 Exemple

Dans notre application d'exemple, les trois modules utilisent les opérations décrites précédemment suivant les algorithmes présentés ci-dessous.

Tracker

init("Tracker"***)***
tant que ***wait()*** **faire**
  lecture de la position de la souris
  put(pos***)***
close()

Simulation

MPI_Init()
init("Simulation/"+rank***)***
tant que ***wait()*** **faire**
  get(pos***)***
  mise à jour de l'état de la grille
  put(grid***)***
close()

Visualization

init("Visualization"***)***
tant que ***wait()*** **faire**
  get(pos***)***
  get(grid***)***
  affichage d'une nouvelle image
close()

6.5 Graphe de flux de données

Une fois que les modules de l'application sont définis, ils sont assemblés en connectant leurs ports d'entrée et de sortie. Ces connexions forment le graphe de flux de données de l'application.

6.5.1 Connections simples

Une connexion de base relie un port d'entrée à un port de sortie et transmet les messages en respectant l'ordre d'émission (mode *FIFO*). Bien qu'un port d'entrée ne peut être relié qu'à un seul port de sortie, à l'inverse un port de sortie peut être connecté à plusieurs port d'entrée, les données sont dans ce cas transmises à tous les ports connectés.

En reprenant notre exemple de simulation de fluide interactive (section 6.3 page 49), pour construire une première version simple non parallélisée il suffit de connecter le port de sortie *pos* du module *Tracker* aux port d'entrée correspondant des deux modules *Simulation* et *Visualization*, et de connecter le résultat de la simulation (port *grid* du module *Simulation*) au port *grid* du module *Visualization* (figure 6.1(b) page 50).

Le mode de communication *FIFO* permet de garantir une cohérence forte pour l'application. En effet les modules sont exécutés à la même fréquence et leurs données d'entrée correspondent à la même itération. Dans notre exemple, la position du traqueur reçue par la visualisation est la même que celle utilisée par la simulation. Toutefois l'inconvénient majeur de ce mode d'exécution est qu'il fixe les performances de toute l'application à la fréquence du module le plus lent.

6.5.2 Filtrage des données

Dans le cas de modules parallélisés des schémas de communications collectives sont nécessaires. Ces schémas reposent sur des constructions plus complexes que de simples assemblages de connexions point-à-point. Pour répondre à ces contraintes nous introduisons de nouveaux objets dans le graphe de flux de données.

La plupart des schémas de couplage de codes parallèles peuvent s'exprimer à l'aide d'opérations de filtrage sur les données transmises. Par exemple pour paralléliser un module il est parfois nécessaire de découper les données, ou bien de répartir chaque messages vers différentes machines (parallélisme inter-itérations). Ensuite les résultats doivent être recombinés en concaténant les données ou en réordonnant les messages.

Appliquer des filtres sur les données transmises peut être utile pour d'autres utilisations telles que convertir le type des données, compresser/décompresser, ou encore ne sélectionner qu'une partie des données, comm le *frustum culling* (section 12.3 page 113) par exemple.

Ces opérations dépendent des modules impliqués dans la communication. Comme la conception d'un module doit être indépendante des autres modules avec lesquels il

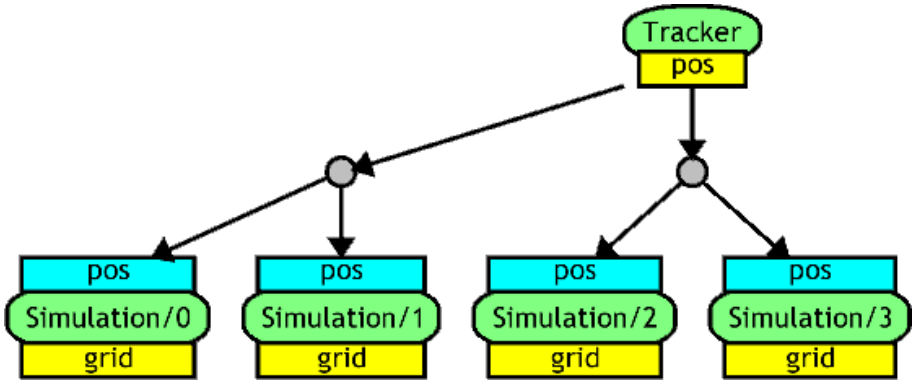


Figure 6.3 *Arbre binaire de broadcast utilisant des noeuds de routages (représentés par des ronds).*

communique ce filtrage doit être implanté par une autre entité. Nous définissons donc les *filtres*, qui sont des objets implantant chacun une opération de filtrage de base. Comme les modules, ils possèdent des ports d'entrée et de sortie. Toutefois ils ne sont pas basés sur un modèle itératif reposant sur son propre fil de calcul mais plutôt sur une structure de type évènements. Un filtre est activé à chaque fois qu'un nouveau message est disponible sur l'un de ses ports d'entrée. Il peut aussi utiliser d'autres évènements tels qu'une horloge périodique. Pour pouvoir implanter des opérations telles que réordonner des messages ou interpoler/combiner plusieurs données un filtre a accès à l'historique des messages reçus sur chacun de ses ports.

L'implantation d'un filtre dépend souvent du type de donnée transmise. Il est donc important que l'utilisateur puisse implanter ses propres filtres. Comme les filtres ne sont pas contenu dans leur propre programme comme le sont les modules, cette implantation est très dépendante de l'implantation du modèle lui même. Ceci est toutefois inévitable pour assurer un surcoût minimal. Chaque filtre implante une opération de base et une série de ces opérations peut parfois être nécessaire lors d'une communication. Leur surcoût doit donc être le plus faible possible.

6.5.3 Communications collectives

Comme expliqué ci-dessus, les filtres permettent d'implanter des schémas de communications entre modules parallèles. Dans cette section quelques uns des ces schémas classiques sont présentés à l'aide de l'exemple d'application de simulation de fluide. Tous ces schémas se retrouvent dans les librairies de passage de messages comme MPI. Nous présentons ici leur transposition au modèle *data-flow* de FlowVR.

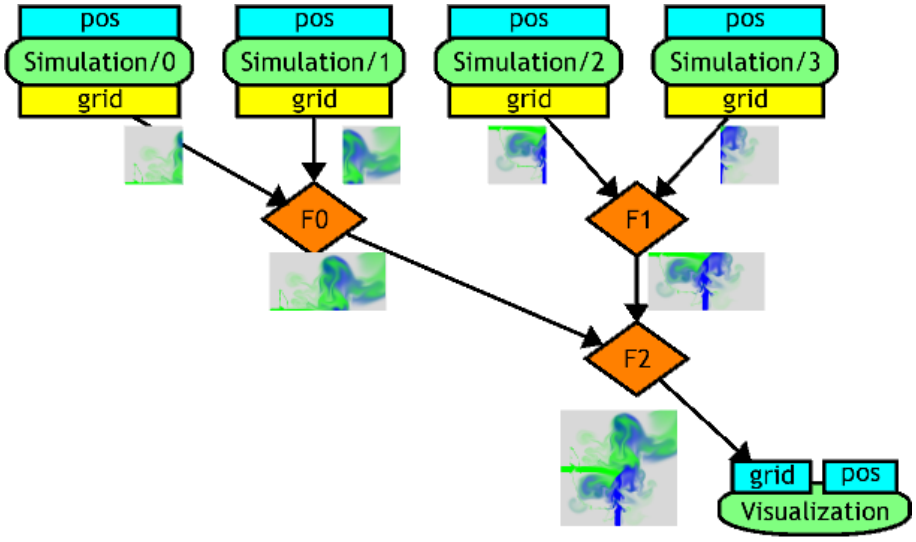


Figure 6.4 *Filtres de fusion du résultat de la simulation.*

6.5.3.1 Arbre de broadcast et découpage (*Scatter*)

Un broadcast *plat*, c'est-à-dire où la source des données les envoie directement et séquentiellement à chaque destinataire, a un coût linéaire en nombre de machines. Pour réduire ce coût il est possible d'utiliser des communications en arbre, ou la source envoie les données à quelques machines, qui les retransmettent à d'autres machines, récursivement jusqu'à ce que tous les destinataires obtiennent une copie des données. Cette méthode réduit le coût du broadcast en $O(\log n)$.

Pour implanter ce schéma dans le graphe de flux de données il faut pouvoir spécifier via quelles machines les communications sont routées. Pour cela on utilise un type de filtre particulier appelé *noeud de routage*. C'est un filtre qui ne modifie en rien les données mais permet de spécifier un chemin de routage dans le graphe. La figure 6.3 montre le schéma utilisant ces noeuds de routage pour faire un arbre de broadcast binaire entre le module *Tracker* et les modules *Simulation*.

Si tous les destinataires n'ont pas besoin de toutes les données mais seulement d'une partie on utilise une opération de découpage ou *Scatter* à la place d'un broadcast. Cela ce traduit par le même schéma de communication à la seule différence que les noeuds de routages sont remplacés par des filtres plus évolués qui découpent les données en 2 morceaux (ou n pour un arbre n-aire).

6.5.3.2 Fusion de données (*Gather*)

Quand on utilise un parallélisme basé sur les données, le calcul est réparti sur plusieurs machines qui produisent chacune une partie du résultat. Si un autre calcul nécessite le résultat complet il est nécessaire de fusionner les données partielles via l'opération appelée

Gather. Cette opération dépend du type de données transmises. Dans le cas d'un vecteur de données souvent il suffit de concaténer les sous-vecteurs. Si les données sont des forces à appliquer à une liste d'objets alors il faut additionner chaque valeur. Dans notre exemple les données sont des densités sur une grille 2D. En utilisant un filtre qui fusionne deux sous-grilles adjacentes en fonction de leur position relative on peut fusionner récursivement toutes les données (figure 6.4). Pour connaître la disposition relative des blocs à fusionner le filtre utilise les estampilles contenues dans chaque message (section 6.4.2 page 50).

6.5.3.3 Combinaison de schémas de base

Les communications collectives en $N \times M$, c'est-à-dire ou N modules produisent des données qui doivent être transmises à M autre modules, sont un problème récurrent en couplage de codes parallèles (section 3.2.2.2 page 24). Par le jeu des filtres et connexions, il est possible d'implanter des schémas très optimisés. Une approche simple consiste à combiner plusieurs schémas comme ceux présentés en les ajoutant dans le graphe de l'application. Un système de scripts (section 7.3 page 71) permet à l'utilisateur de spécifier le schéma adapté.

Si on reprend notre application d'exemple en considérant le cas ou le rendu est distribué sur plusieurs machines, alors une telle communication est nécessaire entre les modules de simulation et ceux de visualisation. Pour l'implanter on peut simplement ajouter un arbre de broadcast après l'opération de fusion. La figure 6.5 présente le résultat pour le cas ou on utilise 8 modules de simulations et 2 modules de rendu.

6.6 Asynchronisme contrôlé : ordonnancement par les données

Les réseaux de modules et filtres tels que présentés dans la section précédente permettent de spécifier des applications complexes impliquant plusieurs composants parallèles. Toutes les communications étant en mode *FIFO*, tous les composants de l'application s'exécutent de manière synchrone, ce qui garantit une très forte cohérence mais limite les performances en terme de fréquence de rafraîchissement, latence et passage à l'échelle de l'application. Dans cette section nous allons introduire un nouveau type d'objets pour spécifier d'autres politiques de synchronisation.

6.6.1 Synchronisation entre modules

L'application étant basée sur un graphe de flux de données, il est nécessaire de modifier ce flux pour modifier la fréquence d'activation des différents composants. Par exemple

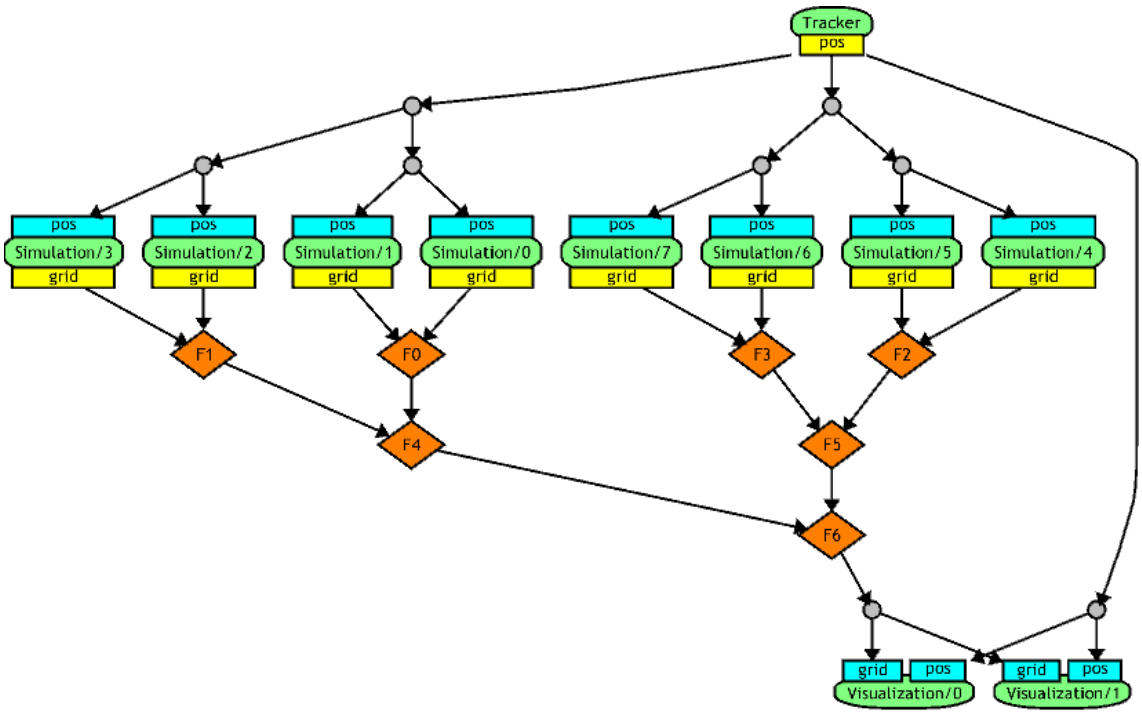


Figure 6.5 *Application complète de simulation de fluide parallèle et interactive.*

il est possible de filtrer les messages en en supprimant un sur deux et ainsi activer une partie de l'application deux fois moins fréquemment que le reste. Un schéma plus courant pour les applications interactives est l'ordonnancement de chaque partie à la fréquence maximale que le permettent les ressources qui lui sont allouées. Dans ce cas, les données communiquées entre les différentes parties ne doivent pas introduire d'attente comme dans le cas *FIFO*. Un mécanisme simple est d'utiliser un mode de communication de ré-échantillonnage *glouton* : les modules utilisent la dernière donnée disponible en jetant les autres données et potentiellement en réutilisant la même donnée qu'à l'itération précédente si aucune nouvelle donnée n'est parvenue. Ce type de filtrage peut introduire des incohérences dans l'application : les données utilisées par un module ne sont plus garanties comme étant calculées à la même itération. Cela veut dire par exemple que le module de rendu peut visualiser des parties de la scène correspondant à des temps de simulations différents. Dans certain cas cette incohérence n'est pas gênante pour l'utilisateur mais dans d'autres circonstances cela peut ne pas être acceptable et il faut alors implanter une politique de couplage différente.

Le choix de la politique de couplage dépend de nombreux facteurs extérieurs. Comme le filtrage des données, cette opération ne doit donc pas être implantée par les modules de l'application. C'est important pour qu'un même module (la simulation de fluide parallèle par exemple) puisse être exploité dans des types d'applications différents (interactions

temps-réel versus simulation classique en batch-processing). Contrairement aux filtres qui ont une influence localisée sur l'application (au niveau des communications reliant deux composants), les politiques de couplage ont des répercussions plus globales. De plus il y a un important compromis entre la centralisation des décisions liées à cette politique, la cohérence de l'application, et l'extensibilité du système. Une politique gérée par un composant centralisé peut garantir une forte cohérence entre toutes les communications mais pénalise le passage à l'échelle de l'application, alors qu'une politique décentralisée sera plus performante mais offrira moins de cohérence. De plus, les prises de décisions de ces politiques peuvent suivre un schéma de distribution différent du filtrage des données en résultant. Pour ces différentes raisons nous avons choisi d'utiliser un nouveau type d'objets appelé *synchroniseur*.

6.6.1.1 Les synchroniseurs

Un *synchroniseur* est un objet similaire aux *filtres* (section 6.5.2 page 55), à la différence qu'il est responsable de la prise des décisions liés à une certaine politique de couplage. Pour prendre ces décisions un synchroniseur reçoit les estampilles (section 6.4.2 page 50) indiquant les messages disponibles ainsi que l'état des différents modules. Le résultat de ses décisions est transmis sous forme d'ordres qui sont ensuite envoyés à des filtres implantant le filtrage des données en conséquence. Cette séparation entre la prise de décision et le filtrage des données est importante pour pouvoir spécifier un schéma de distribution différent pour ces deux opérations (i.e. avoir une politique de couplage centralisée sans pour autant avoir à centraliser les données elles-mêmes). Elle permet aussi de changer l'implantation d'une partie sans affecter l'autre. Par exemple une politique *glouton* implique la destruction de certain messages. Ceci peut ne pas être acceptable dans le cas ou les données du message sont exprimées de manière incrémentale, c'est-à-dire par rapport au message qui les précèdent (évènements, déplacement de souris, ...). Dans ce cas le filtre ne doit pas détruire les messages mais au contraire les fusionner (concaténation des évènements, sommation des déplacements). L'implantation du filtre doit alors être modifiée mais pas celle du synchroniseur. De même il est possible de modifier le synchroniseur pour implanter une politique légèrement différente (comme ne jamais réutiliser le même message et ainsi éviter des calculs inutiles si c'est la seule dépendance de donnée) sans affecter le filtrage des données lui-même.

6.6.1.2 Les ports d'activations

Les seules informations disponibles aux synchroniseurs sont les estampilles des messages envoyés par les modules (numéro d'itération, éventuellement dimensions, bounding box, ...). Pour être informé de l'état des modules, et en particulier être signalé quand le module a fini son itération et donc a besoin d'une nouvelle donnée pour l'itération suivante, les synchroniseurs peuvent utiliser les messages de ports particuliers appelés ports

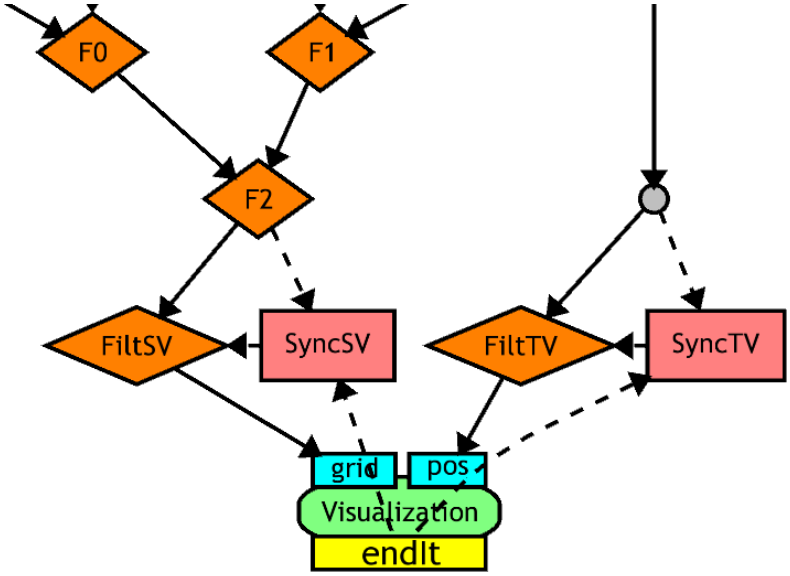


Figure 6.6 *Couples filtres (losanges oranges) + synchroniseurs (rectangles roses) découplant la visualisation des autres modules (couplage asynchrone). Les pointillés représentent les connexions ou transitent des estampilles uniquement.*

d'*activation*. Ces ports sont prédéfinis pour chaque module. Le port de sortie *endIt* signale la fin d'une itération, c'est-à-dire qu'un message est envoyé à chaque fois que le module termine une itération (i.e. appelle la méthode *wait*). Ce port permet donc de savoir quand une nouvelle donnée doit être envoyée. De manière symétrique le port d'entrée *beginIt* permet de faire attendre le module avant de commencer la prochaine itération. Ce port est particulièrement important pour pouvoir bloquer un module (par exemple pour imposer une fréquence maximum ou pour empêcher la saturation des buffers en aval) quand celui-ci n'a pas d'autre port d'entrée qui pourrait le permettre.

6.6.2 Exemple d'utilisation des filtres et synchroniseurs

Dans notre application exemple nous pouvons appliquer une politique glouton pour le module de visualisation de manière à rafraîchir l'image le plus souvent possible (et ainsi permettre de changer le point de vue de manière fluide par exemple) et à mettre à jour la position du curseur le plus rapidement possible pour diminuer la latence perçue par l'utilisateur et ainsi augmenter le confort d'utilisation. Dans ce cas nous pouvons utiliser les couples filtres + synchroniseurs présentés sur la figure 6.6.

Dans le cas d'une tâche distribuée comme la simulation il est souvent nécessaire de garder une cohérence forte entre les différents modules. Dans ce cas il est possible d'utiliser un synchroniseur centralisé qui utilise l'état de tous les modules pour décider de la donnée à utiliser (figure 6.7).

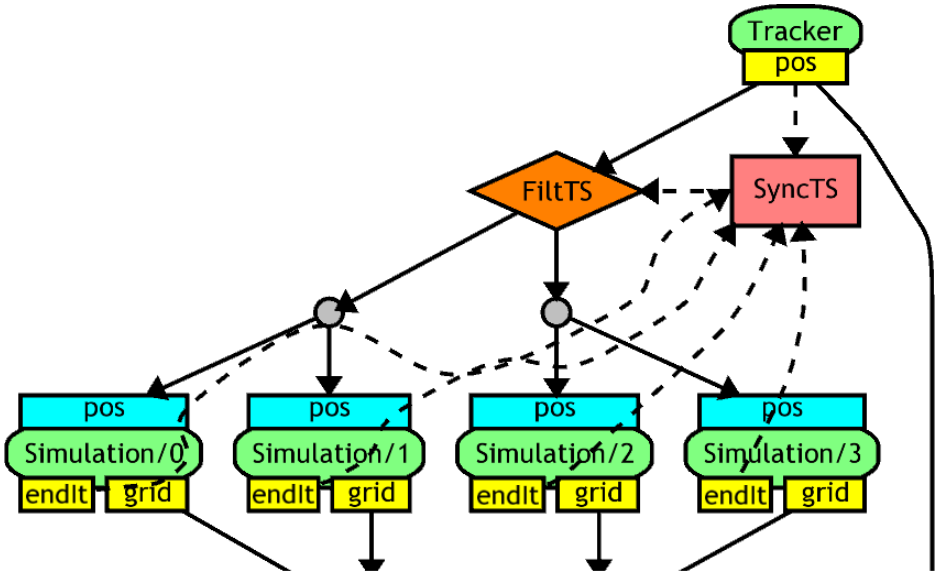


Figure 6.7 *Schéma de couplage asynchrone cohérent : un synchroniseur glouton est utilisé pour filtrer les données d'un ensemble de modules.*

En ajoutant ces schémas de synchronisation chaque partie de notre application peut maintenant être exécutée aussi vite que ses ressources le permettent. Dans le cas ou certaines de ces ressources sont partagées, comme le réseau par exemple, cela peut entraîner une consommation inutile des ressources. Par exemple, le traqueur peu envoyer des nouvelles données très fréquemment (souvent plusieurs milliers de fois par secondes), ce qui ne sert à rien si les autres modules ne dépassent jamais les 100 itérations par seconde. Il est possible de restreindre le module *Tracker* à une fréquence plus faible. Pour cela on ajoute au graphe de l'application un synchroniseur relié aux ports d'activation du module *Tracker* permettant de le mettre en attente s'il dépasse la fréquence fixée. Ce synchroniseur transmet les messages de fin d'itération vers le port de début de l'itération suivant en les mettant en attente s'ils sont trop rapprochés, implantant ainsi la contrainte de fréquence souhaitée. La figure 6.8 présente le schéma correspondant dans le graphe de l'application.

6.6.3 Exemples d'algorithmes de filtres et synchroniseurs

Afin de présenter de manière plus précise le fonctionnement des synchroniseurs et leur couplage avec les filtres, cette section contient les algorithmes utilisés pour le synchroniseur glouton ainsi que le filtre associé de sélection des messages. Les ordres transmis entre le synchroniseur et le filtre sont les numéros d'itération des messages à sélectionner. L'algorithme du synchroniseur de limitation de fréquence est aussi présenté.

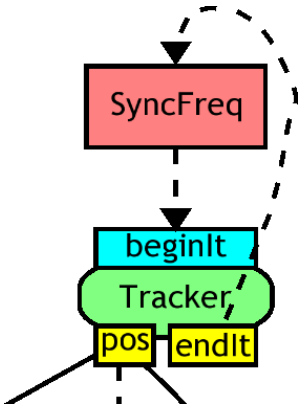


Figure 6.8 *Synchroniseur permettant de contraindre la fréquence du module Tracker.*

Synchroniseur de fréquence maximum

Ce synchroniseur comporte un paramètre *Freq* exprimé en Hertz contrôlant la fréquence d'activation maximale du module connecté. Il est utilisé dans le schéma de la figure 6.8.

Entrée : *endIt* : signaux de fin d'itération du module
Sortie : *beginIt* : signaux de début d'itération du module
Quand début **:**
    put*(beginIt)*
    démarrer le timer pour $1/Freq$ secondes
Quand nouveau message sur *endIt* **:**
    si le timer est expiré **alors**
        supprimer un message sur *endIt*
        put*(beginIt)*
        démarrer le timer pour $1/Freq$ secondes
Quand le timer expire **:**
    si il existe un message sur *endIt* **alors**
        supprimer un message sur *endIt*
        put*(beginIt)*
        démarrer le timer pour $1/Freq$ secondes

Synchroniseur glouton

Le synchroniseur glouton est extrêmement simple. Il se contente d'ordonner l'utilisation du message le plus récent à chaque fois qu'une nouvelle donnée est requise.

Entrée : *in* : estampilles des messages disponibles
Entrée : *endIt* : signaux de fin d'itération du module de destination
Sortie : *order* : numéros des messages à sélectionner
Quand nouveau message sur *endIt* **:**
    n = numéro du message le plus récent sur *in*
    put*(order,n)*
    supprimer tous les messages plus anciens que *n* dans *in*

Filtre de sélection

Pour appliquer les décisions d'un synchroniseur, le filtre de sélection gère la liste des messages reçus et applique les ordres du synchroniseur. Son fonctionnement est plus complexe que le synchroniseur glouton, du fait que l'application d'un ordre reçu peut être mise en attente si le message de donnée correspond n'a pas encore été reçu.

Entrée : *in* : messages disponibles
Entrée : *order* : numéros des messages à sélectionner
Sortie : *out* : messages sélectionnés
Quand nouveau message sur *order* **:**
    n = numéro contenu dans le premier message sur *order*
    supprimer tous les messages plus anciens que *n* dans *in*
    si le premier message sur *in* correspond au numéro *n* **alors**
        put*(out*,premier message sur *in)*
        supprimer le premier message sur *order*
Quand nouveau message sur *in* **:**
    n = numéro contenu dans le premier message sur *order*
    si le nouveau message sur *in* correspond au numéro *n* **alors**
        put*(out*,nouveau message sur *in)*
        supprimer le premier message sur *order*

Implantation 7

Pour faciliter une implantation étape par étape, FlowVR est structuré en un ensemble de couches empilées, permettant de fournir un niveau d'abstraction et de fonctionnalité variable en fonction des besoins. Un ensemble d'outils relativement simples et indépendants implante chaque niveau, ce qui permet d'expérimenter facilement certaines modifications sans affecter le reste du mécanisme. Ce chapitre présente ces outils en partant du niveau le plus bas, constituant les bases du système, jusqu'aux couches supérieures, beaucoup plus souples.

7.1 Environnement d'exécution

Pour favoriser la modularité du système nous avons choisi un environnement d'exécution où chaque module est exécuté dans son propre processus. Cela permet de compiler et lancer chaque module indépendamment et ainsi d'éviter les problèmes de conflits entre les dépendances de chaque module (librairies, outils de compilation, etc). Cela simplifie aussi l'intégration de codes existants, dont les scripts de compilation et de lancement ne doivent être modifiés que très légèrement.

7.1.1 Communications inter-processus

La séparation des modules implique l'utilisation de primitives de synchronisation et communication inter-processus, ou *IPC* (*Inter-Processus Communication*). Plusieurs mécanismes existent sous UNIX tels que les *pipes*, les *semaphores*, les *message queues*, les *sockets* ainsi que les *shared memory areas*. Dans le cadre de FlowVR il est nécessaire de

transmettre des flux de données importants avec le minimum de surcoût possible. Pour cela le plus adapté semble les segments de mémoires partagées (*shared memory areas*). Ceux-ci permettent d'envoyer des données à potentiellement plusieurs modules sans avoir à faire de recopie. Comme ces données sont partagées il est important de gérer explicitement la synchronisation entre les processus pour éviter les corruptions et signaler l'arrivée de nouvelles données. L'API *POSIX Thread*, ou *pthread*, permet de créer des *locks* et des signaux pour cela. Sous Linux depuis l'arrivée du noyau 2.6 une nouvelle implantation [52] de cette API permet d'utiliser ces primitives entre processus différents et avec de très bonnes performances (les *locks* par exemple n'ont recours à un appel système qu'en cas de contention). FlowVR repose donc sur un segment de mémoire partagée pour les communications inter-processus ainsi que les primitives *pthread* pour la synchronisation. Quand ces primitives ne sont pas disponibles une autre implantation peut utiliser des opérations atomiques et des boucles d'attentes actives pour pouvoir tout de même développer des applications FlowVR sur les plateformes ne fournissant pas les *locks* inter-processus (comme le noyau Linux 2.4), mais avec des performances plus faibles.

7.1.2 Démon

Une application FlowVR ne consiste pas uniquement en des communications de données entre processus d'une même machine, mais nécessite aussi des communications avec des modules distants, ainsi que des opérations de filtrage des données (section 6.5 page 55). Un processus particulier appelé *démon* est utilisé pour implanter ces opérations, comme indiqué sur la figure 7.1. Un démon est exécuté sur chaque machine. Il communique via la mémoire partagée avec les modules locaux et envoie via le réseau les messages destinés aux machines distantes.

Le démon implante aussi les filtres et les synchroniseurs utilisés dans le graphe de l'application. Comme ceux-ci dépendent de l'application et peuvent être modifiés en fonction des besoins, ils sont implantés via un mécanisme de plugins. Quand une application nécessite un certain filtre, le démon charge dynamiquement la librairie contenant son implantation. Cette implantation dérive d'une classe abstraite définissant les méthodes utilisées par le démon pour exécuter le filtre (initialisation, réception et envoi de messages). La plupart des autres éléments du démon (connexions réseaux, contrôle des modules) sont aussi implantés sous forme de plugins. Ils permettent ainsi de fournir facilement des implantations alternatives. Par exemple le réseau utilise actuellement le protocole TCP mais une autre implantation autour de MPI a aussi été effectuée.

Le coeur du démon repose sur deux structures de données principales :

– La *table des objets*, répertoriant tous les objets instanciés (tels que les filtres) en fonction de leur identifiant.
– La *table de routage*, indiquant quelles sont les *actions* à effectuer sur un message (i.e. les filtres à activer) en fonction de la provenance du message (l'estampille *source* du message).

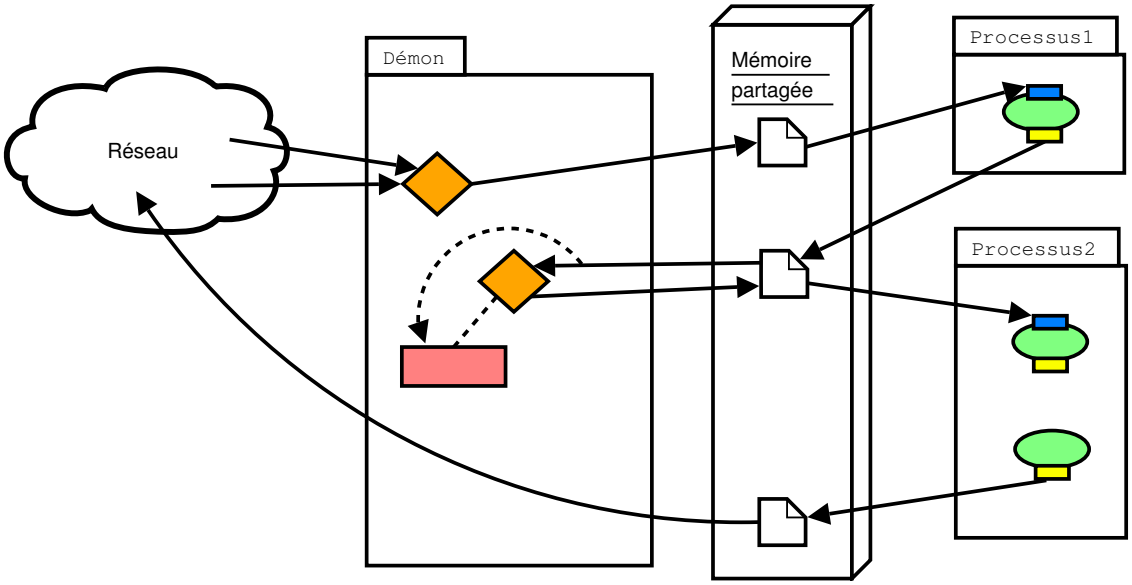


Figure 7.1 *Implantation d'une application FlowVR via un démon par machine.*

En interne le démon est architecturé autour d'un mécanisme de routage de messages via plusieurs threads liés aux modules locaux, connexions réseaux, et évènements périodiques. Chaque thread gère une queue de messages en cours de traitement. Ces messages sont traités un par un en utilisant les actions indiquées dans la *table de routage*. Ces actions peuvent :

– générer de nouveaux messages qui seront alors placés dans la queue de messages à traiter ;
– envoyer un signal à d'autres threads de traitement (modules, connexions réseaux).

En revanche une action ne doit pas effectuer un traitement long ou attendre longtemps un signal car cela ralentirait la boucle de traitement des messages du thread appelant. Pour les cas où c'est nécessaire l'action doit faire appel à un autre thread. Par exemple, quand un message doit être envoyé sur le réseau cet envoi n'est pas fait directement par l'action sur ce message mais il est plutôt transmis au thread spécifique d'envoi des données sur la connexion réseau correspondante.

La figure 7.2 montre le schéma de fonctionnement des threads du démon. La fonction principale *main* de l'exécutable crée un premier thread *Commander* qui gère les commandes de contrôle de l'application (décrite dans la section 7.1.3 page 69). Si le mode réseau est activé, un deuxième thread *NetTCP* est créé pour attendre les connexions TCP. Dès qu'un autre démon se connecte, un thread *RecvThread* est créé pour recevoir les messages transmis. Quand le *Commander* reçoit une commande indiquant un nouveau module, il crée alors un thread *Regulator* qui va gérer les messages produits par ce module. La plupart des filtres et synchroniseurs sont de simples objets qui sont ajoutés dans la table des objets, mais certains peuvent aussi créer leur propre thread, qui viennent dans

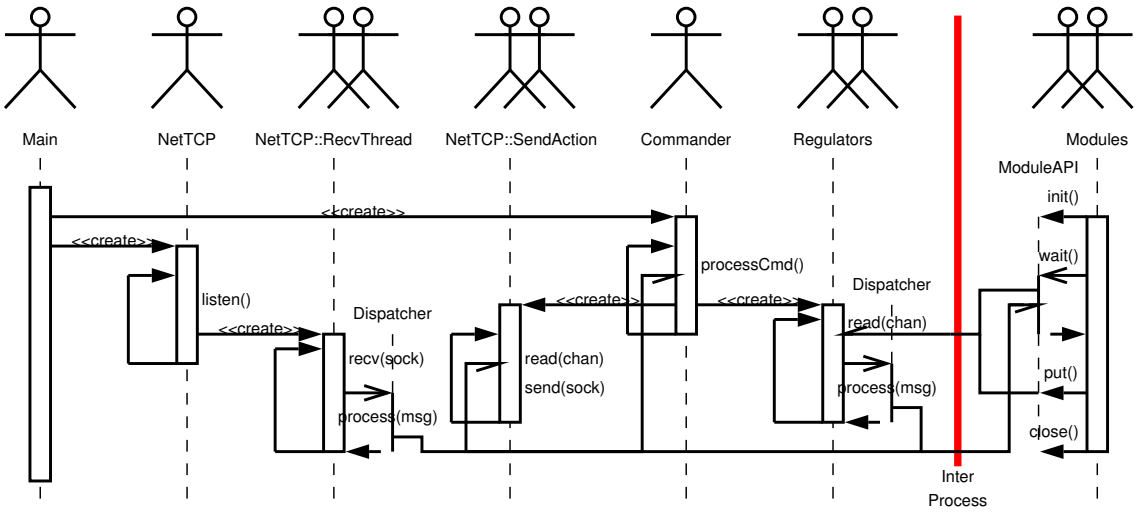


Figure 7.2 *Threads utilisés à l'intérieur du démon et des modules ainsi que les communications entre ces threads.*

ce cas s'ajouter à ce schéma.

Tous les threads qui peuvent produire des messages, comme par exemple les threads *Regulator* et *RecvThread*, utilisent une instance de la classe *Dispatcher* pour gérer la boucle de traitement des messages via la méthode *process(msg)*. C'est une classe abstraite dont l'implantation est faite par un plugin. L'implantation actuelle exécute toutes les actions dans le thread ayant produit le message, mais d'autres politiques pourront être étudiées par la suite. Ces actions peuvent au besoin réveiller le bon thread (module, envoi réseau, commander) en utilisant le signal associé.

Du fait de la séparation entre le processus du démon et les modules, à chaque itération un changement de contexte vers le thread *Regulator* est nécessaire pour implanter les filtres et les synchroniseurs associés à ce module. Dans le cas ou toutes les données sont disponibles localement, comme dans le cas de la figure 6.6 où un couple de filtre et synchroniseur greedy est utilisé sur chaque port d'entrée, les actions correspondantes sont exécutées dans la boucle de traitement des messages du thread *Regulator* et donc le module sera réveillé immédiatement, ce qui fait au total deux changements de contextes. Si en revanche un message doit être reçu par le réseau, l'action de réveil du module sera exécutée directement depuis le thread de réception des messages de la connexion réseau correspondante. Les messages envoyés par le module via la méthode *put* passent eux aussi par le thread *Regulator*, et vont ensuite réveiller un thread d'envoi réseau ou un autre module local.

7.1.3 Configuration de l'application : langage de commandes

Pour spécifier les objets à mettre en place ainsi que les routes (i.e. association source–action) de la table de routage il est nécessaire d'envoyer au démon des commandes les décrivant. Nous utilisons pour cela un protocole de commandes à base de XML. Les commandes suivantes sont disponibles :

addobject : ajoute un nouvel objet en spécifiant son identifiant, la classe à instancier, ainsi que d'éventuels paramètres.

delobject : supprime un objet en spécifiant son identifiant.

addroute : ajoute une route dans la table de routage en spécifiant son identifiant, la source du message à traiter, ainsi que l'action à effectuer. Cette action contient l'identifiant de l'objet destinataire ainsi qu'un paramètre (nom du port dans le cas d'un filtre, nom de la machine distante dans le cas d'une transmission réseau).

delroute : supprime une route en spécifiant son identifiant.

action : invoque une action spécifique d'un objet en spécifiant son identifiant ainsi que ses paramètres. Ceci permet de déclencher des actions telles que le démarrage et l'arrêt de l'application.

group : permet d'invoquer une commande sur un groupe d'objets. Utile pour mettre en pause et redémarrer tous les objets de l'application par exemple.

Toutes ces commandes utilisent un système d'identifiants pour spécifier les objets ou routes auxquelles elles s'appliquent. FlowVR utilise un système similaire aux systèmes de fichiers classiques. Les objets existants dans le démon indépendamment de l'application ont un identifiant *absolu*, c'est-à-dire commençant par `"/"`, tel que `"/NET"` pour le réseau. Les objets appartenant à une application sont exprimés *relativement* à l'identifiant de l'application, apparenté à un répertoire courant. Par exemple, une commande `addobject "Module1"` dans le cadre de l'application `"/A1"` créera un objet portant l'identifiant `"/A1/Module1"`. Cela permet d'exécuter plusieurs applications simultanément de manière indépendante, tout en autorisant le partage de certains objets en utilisant leurs identifiants absolus.

Un objet particulier à l'intérieur du démon appelé *Commander* est responsable de l'implantation de ces commandes. La façon dont ces commandes sont générées et envoyées aux démons est décrite dans les sections suivantes.

7.2 Mécanisme de contrôle d'applications

Démarrer et contrôler une application distribuée constituée de nombreux composants n'est pas trivial, surtout considérant que les modules FlowVR doivent pouvoir utiliser n'importe quelle librairie de communication, et donc souvent le mécanisme de lancement associé. Pour cela FlowVR repose sur un type de module particulier appelé *contrôleur*.

Un contrôleur est un module particulier qui est responsable du lancement et du contrôle du reste de l'application (i.e. démarrage, arrêt, éventuellement pause, monitoring). Pour cela il déclare un port de sortie lui permettant d'envoyer des commandes XML telles que décrites dans la section 7.1.3. Ces commandes sont transmises une par une et sont associées à une estampille *dest* indiquant à quelle machine elles sont destinées. Le démon local envoie alors chaque commande au *Commander* du démon correspondant et récupère la réponse qui est retransmise au contrôleur via un port d'entrée.

7.2.1 Lancement des modules

Les commandes XML envoyées aux démons permettent de contrôler le réseau FlowVR mais elles ne lancent pas les modules eux-mêmes étant donné que ce sont des programmes indépendants qui peuvent utiliser leur propre mécanisme de lancement. De plus, le lancement à distance de ces programmes pose d'importants problèmes de sécurité. Ainsi le contrôleur lance les modules en utilisant les outils appropriés, comme `mpirun` ou `ssh` par exemple. Pour faire la liaison avec l'application FlowVR les variables d'environnement suivantes sont utilisées :

FLOWVR_DAEMON : l'identifiant du démon à utiliser, c'est-à-dire le numéro du segment de mémoire partagée à ouvrir ;

FLOWVR_PARENT : l'identifiant du module contrôleur de l'application, qui sert de préfixe pour les identifiants du module ;

FLOWVR_MODNAME : l'identifiant du module ou groupe de modules à lancer.

Ces variables sont initialisées par le contrôleur avant d'exécuter chaque script lançant un groupe de module de l'application.

Dans l'implantation actuelle de FlowVR un programme assez simple appelé *flowvr-telnet* implante ce rôle de contrôleur de manière similaire au programme *telnet*. Il lit les commandes XML écrites sur son entrée standard et les fait suivre au démon. En plus des commandes décrites à la section précédente des informations supplémentaires sont nécessaires. Pour spécifier quelle machine est concernée par chaque commande un tag XML `dest` est utilisé. Pour spécifier les scripts de lancement des modules de l'application un tag `run` est utilisé. Un tag `wait` est employé pour demander d'attendre le résultat des commandes précédentes. Cela permet par exemple d'attendre que tous les objets soient créés avant de créer les connexions entre eux. Enfin les tags `pause` et `start` permettent de stopper et redémarrer l'application.

7.2.2 Construction du graphe de flux de données

Une application FlowVR peut nécessiter plusieurs milliers de commandes à transmettre. Spécifier ces commandes une par une directement serait difficile voir impossible pour les applications de grande taille. C'est pour cela qu'un ensemble d'outils permet

de le faire en utilisant des descriptions de plus haut niveau de l'application. Le premier d'entre eux est *flowvr-network*, un outil permettant de générer les commandes de lancement des éléments d'une application à partir d'une description XML de ces éléments. Cette description est appelée *graphe instancié* et décrit de manière complète l'application FlowVR telle qu'elle est instanciée sur des machines données. Elle contient la liste des objets (modules et filtres) de l'application, spécifiant leur identifiant, placement, ainsi que leurs ports. Elle contient aussi la liste des connexions entre ces objets, spécifiant les identifiants et ports de la source et la destination, ainsi que le type de connexion (envoi des messages complets ou juste de leurs estampilles).

Bien que le graphe instancié reste une description assez bas niveau elle contient plus d'informations que le langage de commandes. Par exemple, la description des ports des objets ainsi que des connexions permet de vérifier leur validité (i.e. les ports connectés existent bien, le type de connexion est compatible, ...). Changer certains paramètres tel que le placement des modules n'affectent que les données des modules concernés, alors qu'au niveau des commandes ces changements affectent aussi les connexions associées. Cependant d'autres changements tels que la modification du nombre de modules instanciés requièrent des changements plus profonds dans le graphe de l'application.

7.3 Passage à l'échelle : dépliage de graphes

Comme indiqué à la section précédente, le graphe de l'application dépend des paramètres des modules utilisés. Par exemple si on change le nombre de modules utilisés par la simulation de fluide (figure 6.5 page 59), il faut refaire le graphe en modifiant les filtres *Merge2D* qui permettent de rassembler les résultats de chaque processus. Cependant la structure du graphe reste la même : les données d'entrée utilisent un arbre de broadcast pour être envoyées aux n modules de la simulation, et les résultats sont fusionnés 2 à 2 par un arbre inversé de filtres *Merge2D*, quel que soit le nombre exact n de modules utilisés.

En utilisant une représentation de la structure du graphe ainsi que des outils permettant de paramétrer les modules et d'appliquer cette structure pour générer le graphe instancié alors l'utilisateur n'aura qu'à changer les paramètres pour adapter l'application à l'environnement qu'il souhaite et la lancer automatiquement. Nous avons envisagé deux types de représentations de la structure du graphe :

- une représentation *déclarative*, à base de règles élémentaires de transformation des données (i.e. comment transformer deux sous-grilles 2D en une seule grille 2D via le filtre *Merge2D*, ou comment utiliser un filtre pour convertir des données du format *float* en *double*) et d'un moteur de résolution qui cherche à connecter les ports des modules en combinant ces règles ;
- une représentation *procédurale*, à base de procédures spécifiant les structures classiques (broadcast, gather, ...) à partir des éléments de base (connexion FIFO, filtres, noeuds de routage) et permettant à l'utilisateur de définir ses propres procédures.

La première solution peut être très puissante mais difficile à maîtriser, surtout dans le cadre de la conception des premières applications FlowVR. Nous avons donc choisi la deuxième solution, plus simple à implanter. Après avoir acquis de l'expérience avec cette solution il sera toujours possible de tenter des alternatives via la première solution.

FlowVR fournit donc un module Perl permettant de modifier le graphe XML d'une application en utilisant des procédures prédéfinies ou implantées par l'utilisateur. Par exemple notre application de simulation de fluide en mode *FIFO* telle que présentée sur la figure 6.5 peut être générée par le script suivant :

```
addRecursiveBroadcast('Tracker','pos','Simulation','pos');
addRecursiveBroadcast('Tracker','pos','Visualization','pos');
addRoutingNode('R','pc1');
addRecursiveGather('Simulation','grid','R','',Merge2D);
addRecursiveBroadcast('R','','Visualization','grid');
```

La plupart des procédures utilisées prennent en paramètre le préfixe et le port des modules sources et destinations. Ils sont reliés en utilisant le schéma de communication demandé en fonction du nombre d'instances de ces modules. Le script déplie en quelque sorte le graphe de l'application, c'est-à-dire qu'il créé le graphe complet à partir d'une spécification repliée de la structure de celui-ci.

Pour générer la description des modules instanciés utilisée en entrée des scripts de dépliage ainsi que les commandes de lancement nous utilisons une description XML des modules disponibles. Chaque description de module contient la commande de lancement ainsi que la description des instances de modules créées (identifiant, placement et liste des ports), en fonction de la liste des machines que ce module doit utiliser ainsi qu'un certain nombre d'autres paramètres dépendant du module (par exemple nom du périphérique du tracker, taille de la grille à simuler, configuration des vidéo-projecteurs). Ensuite une application est spécifiée en listant les modules à utiliser ainsi que leurs paramètres. Un outil appelé *flowvr-module* prend ces informations en entrée et génère les commandes de lancements de tous les modules de l'application ainsi que la description des modules instanciés. Cette description est donnée en entrée des scripts de dépliage pour générer le graphe instancié de l'application, qui est ensuite transmis en entrée de *flowvr-network* pour générer les commandes XML utilisées par *flowvr-telnet* pour configurer l'application finale. Cette chaîne de traitement est résumée sur la figure 7.3.

En utilisant les structures conditionnelles et de bouclage du langage procédural il est possible de faire des scripts génériques qui sont réutilisables dans de nombreuses applications. Par exemple, dans la suite nous présenterons l'utilisation de FlowVR dans des applications impliquant des chaînes de traitement sur des flux vidéos pour de la reconstruction 3D temps réel et des interactions dans un monde virtuel. Ces applications demandent des centaines de modules et des milliers de connexions et filtres. Ils sont décrit via une dizaine

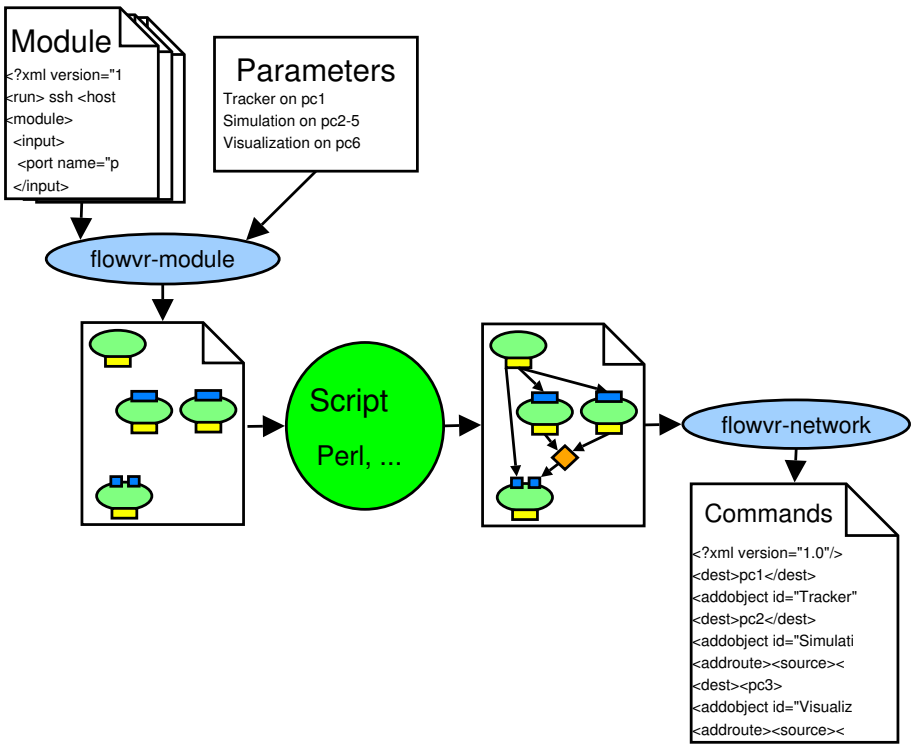


Figure 7.3 *Chaîne d'outils permettant de générer les commandes de lancement d'une application FlowVR en changeant les paramètres des modules instanciés.*

de scripts Perl d'environ 100 lignes chacun et utilisant des schémas différents en fonction des conditions. Par exemple on peut soit utiliser des flux vidéos *"live"* soit les lire à partir de fichiers pré-enregistrés, les schémas de synchronisation étant alors différents. La figure 7.4 présente le graphe instancié d'une telle application. Les objets et les connexions ajoutés par chacun des scripts sont représentés par des couleurs différentes. Un graphe d'une telle complexité ne peut pas être généré manuellement.

7.4 Performances

FlowVR est avant tout un environnement destiné à développer de grandes applications interactives. En ce sens, les résultats les plus importants concernent l'efficacité du modèle et de l'environnement de développement pour la construction et l'exécution de telles applications. Cependant la performance obtenue lors de l'exécution des applications est aussi un critère important, nous allons donc présenter quelques tests effectués sur des applications simples.

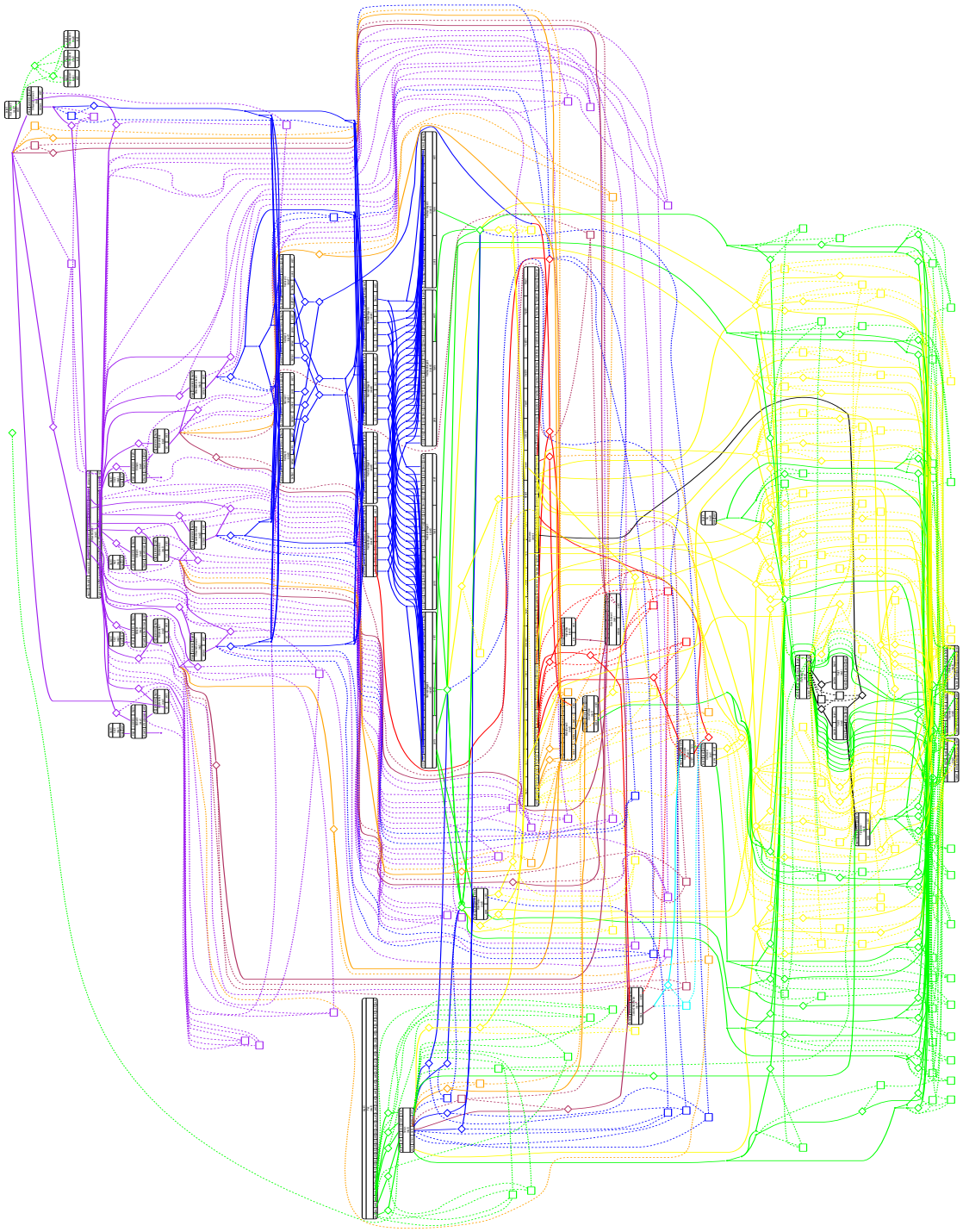


Figure 7.4 *Application FlowVR utilisant de nombreux modules et plusieurs scripts de dépliage, qui totalisent environ 1000 lignes de Perl.*

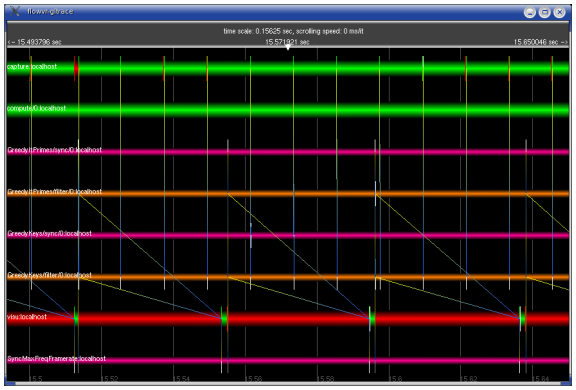


Figure 7.5 *Visualisation de la trace d'une application FlowVR distribuée.*

Les résultats présentés sont toutefois très partiels, du fait que cette première implantation de FlowVR était surtout destinée à tester les concepts du modèle et non à obtenir les performances optimales.

7.4.1 Mesure et analyse de trace

Mesurer les performances d'une application distribuée n'est pas toujours facile. La vitesse de chaque module peut facilement être calculée localement sur chaque machine. En revanche, obtenir des données de latence est plus compliqué si l'opération concernée est répartie sur plusieurs machines. Une latence peut être définie comme la différence de temps entre le moment où une donnée est produite (mouvements de l'utilisateur) et le moment où le résultat correspondant est disponible (l'image de l'environnement virtuel est affichée). Entre ces deux temps le calcul peut passer par un certain nombre de modules distribués sur plusieurs machines (la simulation de fluide par exemple). Mesurer cette latence nécessite donc de tracer les évènements le long du chemin de traitement et comparer les horloges des machines impliquées.

Pour effectuer ces mesures un système de traces a été ajouté à FlowVR, dans le cadre du stage de Julien Garand. Une machine sert d'horloge de référence. Avant et après l'exécution de l'application une série d'allers-retours réseau est effectuée entre cette machine et toutes les autres. Ces *ping* permettent de faire la correspondance entre les cycles CPU de chaque machine et l'horloge de référence. Lors de l'exécution un ensemble d'évènements est stocké, estampillé par l'horloge CPU de la machine locale. Ensuite, en utilisant la description du réseau de l'application et les correspondances des horloges il est possible d'analyser ces données, pour par exemple les visualiser comme sur la figure 7.5.

Plateforme	Fréquence maximale (Hz)	Temps de changement de contexte (microseconde)
Intel Xeon 2.66 GHz	69400	9.11
AMD Opteron 2.0 GHz	130500	4.52
Intel Pentium M 1.7 GHz	88800	5.63
Intel Itanium II 900 MHz	39680	13.93

TAB. 7.1 *Surcoût engendré par les changements de contexte.*

7.4.2 Signaux inter-processus

Les modules étant instanciés dans des processus distincts, toutes les communications entraînent des changements de contexte entre ces processus. Pour quantifier le surcoût associé, nous avons mesuré le nombre d'itérations maximal effectué par seconde pour un module n'effectuant aucun calcul. Ceci permet de déduire le temps nécessaire pour chaque changement de contexte entre les modules et le démon (table 7.1).

Le temps de changement de contexte est très variable en fonction de la plateforme utilisée. Toutefois il ne dépasse jamais les 15 microsecondes, ce qui est négligeable pour les modules fonctionnant aux fréquences classiques d'interaction (de l'ordre de 100 Hz). Ces performances permettent d'atteindre plus de 10000 Hz, ce qui est suffisant pour les modules nécessitant les fréquences les plus élevées, comme pour le contrôle d'un retour haptique.

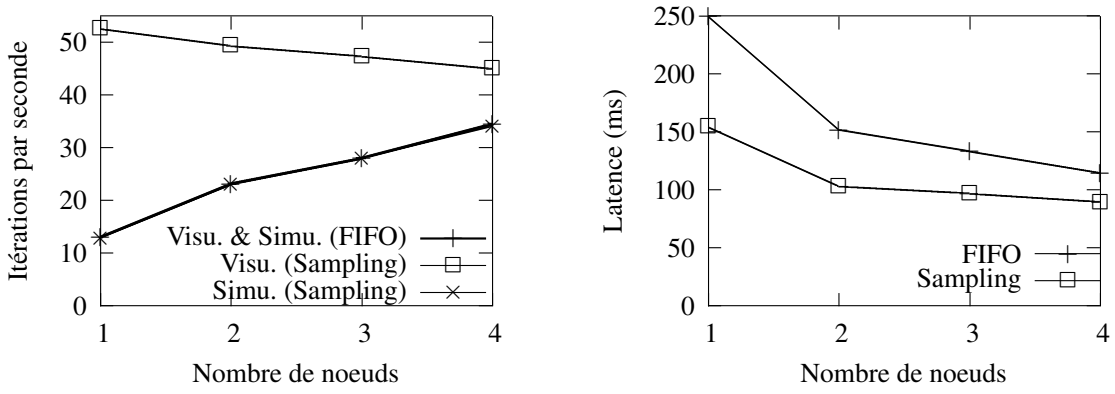
Comme indiqué dans la section 7.1.1 (page 65), le noyau Linux 2.4 ne supporte pas les primitives de synchronisation inter-processus. Dans ce cas, FlowVR utilise une implantation basée sur des attentes actives qui entraînent un surcoût important dès que plusieurs modules sont lancés sur une même machine. Ainsi, une application comportant deux modules tourne en moyenne 30 % moins vite sur un noyau 2.4 que sur un 2.6, et cette différence s'accroît considérablement pour plus de modules. Cette implantation permet donc de développer et tester de petites applications, mais une utilisation plus poussée de FlowVR sous Linux requiert le noyau 2.6.

7.4.3 Couplage parallèle Simulation / Visualisation

Une mesure importante concerne les performances du couplage entre une simulation et visualisation parallèles, comme notre application d'exemple (figure 6.1 page 50), en fonction de la politique de synchronisation utilisée. Deux politiques ont été testées :

FIFO : Les données de la simulation sont utilisées une par une, tous les modules s'exécutant donc à la même vitesse (figure 6.4).

Sampling : La visualisation utilise les dernières données disponibles et tourne aussi vite que possible (figure 6.6). Dans le cas où la visualisation est distribuée sur plusieurs



(a) Fréquence

(b) Latence

Figure 7.6 *Performance du couplage simulation / visualisation avec un réseau d'échantillonnage cohérent et un réseau FIFO.*

machines cet échantillonnage est effectué de manière cohérente (les mêmes données sont utilisées sur toutes les machines).

La figure 7.6 présente les résultats pour un nombre de machines affectées à la visualisation et à la simulation variant de 1 à 4. Les détails de cette expérimentation ont été publiés à Euro-Par 2004 [9].

On remarque que la politique *Sampling* permet d'exécuter la visualisation plus rapidement, ce qui était attendu. En ce qui concerne la latence, on observe que la politique *Sampling* est aussi avantageuse, du fait du choix de la dernière donnée disponible, alors que la politique *FIFO* utilise toujours la donnée bufferisée la plus ancienne. Dans les deux cas les performances augmentent avec le nombre de machine utilisée, mis à part pour la visualisation dans le cas *Sampling*, qui est ralentie au fur et à mesure que la simulation est plus rapide, du fait du coût d'envoi des données sur la carte graphique.

Expérimentations 8

Pour valider l'architecture proposée, cette section présente son utilisation au travers du développement de plusieurs applications.

8.1 La plateforme GrImage

La plupart des applications développées au cours de ma thèse l'ont été sur la plateforme GrImage (figure 8.1). C'est une plateforme regroupant plusieurs caméras firewire (jusqu'à une dizaine) et un mur de 16 vidéo-projecteurs, tout ceci piloté par 11 Bi-Xeon 2.66 GHz et 16 Bi-Opteron 2.0 GHz équipés de cartes graphiques Geforce 6800 Ultra. Ces machines sont reliées par un double réseau Ethernet Gigabit. Un lien 10 Gigabits relie cette grappe à deux autres grappes du laboratoire, l'une avec 100 Bi-Itanium 2 et l'autre regroupant 48 Bi-Xeon 2.4 GHz. Ces grappes font aussi parties du projet Grid 5000, dont l'objectif est de rassembler 5000 processeurs répartis sur plusieurs sites en France.

Le but de la plateforme GrImage est de fédérer les travaux des équipes de recherche en parallélisme, vision, graphisme et animation autour de projets communs.

8.2 Applications graphiques simples

Il arrive fréquemment qu'une application graphique existante relativement simple, souvent programmée en OpenGL, doive être adaptée pour fonctionner sur une plateforme parallèle comme un mur d'image. En considérant que cette application n'utilise pas beaucoup de données en entrée, ni des calculs trop complexes, seule la partie rendu doit être

Figure 8.1 *Matériels utilisés sur la plateforme GrImage.*

modifiée. Deux problèmes principaux se posent :

- le rendu parallèle, i.e. fournir les données de la scène à chaque carte graphique ;
- la configuration des vidéo-projecteurs (matrices de positionnement, optionnellement masques de recouvrement).

Plusieurs solutions sont possibles :

- exécuter l'application originale sur une machine et utiliser Chromium pour distribuer les commandes OpenGL ;
- porter l'application sous Net / VR Juggler qui la répliquera sur chaque machine ;
- ajouter des ports de communications FlowVR pour synchroniser les données internes.

Chacune de ces solutions a des avantages particuliers. Si Chromium est disponible sur la plateforme, alors la première solution est la plus rapide à implanter, mais risque d'avoir un surcoût important si la scène est très dynamique. Porter l'application sous Net Juggler permet de ne pas être dépendant de ce facteur mais demande parfois des modifications profondes dans le code de l'application. Enfin la dernière solution impose de réajuster pour chaque application le code nécessaire à la synchronisation des données et à l'ajustement des projecteurs. Cependant, ce code est assez simple et tient en quelques dizaines de lignes.

Plusieurs applications ont été adaptées au mur d'image en utilisant Net Juggler. Plus récemment l'utilisation de ports FlowVR a été employée par exemple pour adapter Ani-

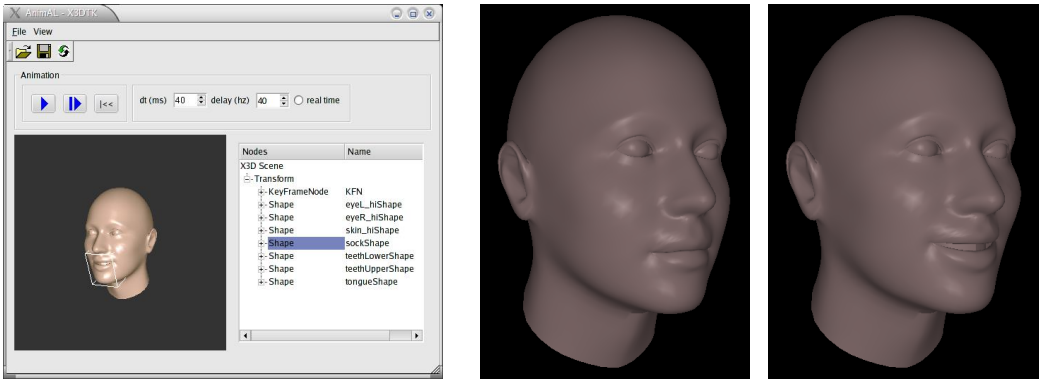


Figure 8.2 *Animation faciale simple utilisant AnimAL.*

mAL [57] (figure 8.2), le moteur d'animation développé dans le cadre du projet EVASION du laboratoire GRAVIR de l'INRIA Rhône-Alpes.

Quel que soit la méthode choisie, une fois l'application intégrée dans l'architecture FlowVR il est possible de l'étendre facilement en la couplant avec d'autres composants. En ajoutant quelques ports d'entrée l'application peut recevoir des données produites par les modules de reconstruction multi-caméras par exemple. Que l'application soit répliquée ou bien instanciée sur une seule machine via Chromium n'affecte alors que le réseau d'interconnexion FlowVR. Enfin, pour intégrer cette nouvelle application à une autre application plus conséquente, il est possible d'ajouter des ports de sortie communiquant les données calculées, qui seront alors intégrées à la visualisation du reste de l'application, court-circuitant l'ancien affichage OpenGL local. L'ancienne application intègre donc l'architecture en tant que nouveau module appartenant à la tâche *Calculs* dans ce cas (chapitre 2 page 7).

8.3 Reconstruction 3D temps réel

La première application développée exploitant toutes les ressources de la plateforme GrImage fut la parallélisation d'une méthode de reconstruction 3D. L'objectif est de calculer en temps réel l'enveloppe visuelle (volume) 3D d'un objet à partir des images obtenues par plusieurs caméras le filmant. Nous avons utilisé un algorithme de reconstruction surfacique basé sur une extraction de silhouettes (images noir et blanc séparant les pixels correspondant à l'objet à reconstruire du fond) développé par Edmond Boyer et Jean-Sébastien Franco [61]. Clément Ménier a implanté une parallélisation de la phase de reconstruction en utilisant MPI [62] que nous avons intégrée grâce à FlowVR aux étapes d'acquisition et de traitement des images ainsi qu'une visualisation utilisant Net Juggler. L'application complète est présentée sur la figure 8.3. Ce travail a été publié lors d'IPT 2004 [5].

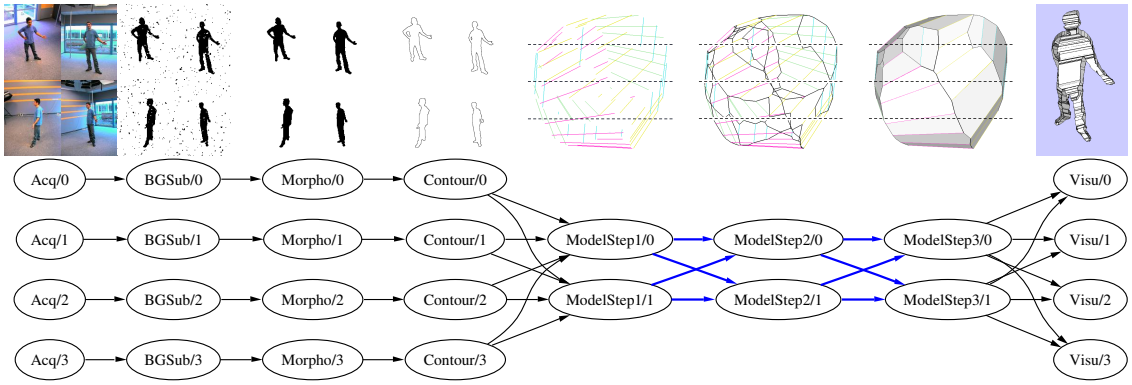


Figure 8.3 *Graphe de flux de données de la reconstruction parallèle. Les liaisons en bleu entre les modules Model sont implantées avec MPI, le reste utilise FlowVR.*

La première phase de cette application concerne le traitement des images des caméras. Ce traitement est effectué en local sur chaque machine contrôlant une caméra, ce qui évite de transférer les flux d'images sur le réseau. Les différentes étapes de ce traitement (acquisition, soustraction de fond, morpho-maths, vectorisation des contours) sont implantées par des modules FlowVR distincts. Malgré la taille importante des données échangées, le surcoût de ce découpage est faible (quelques microsecondes) du fait de l'utilisation d'une mémoire partagée. Cette modularité a permis de faire évoluer chacune des étapes de manière indépendante. Ainsi la plupart des modules ont été réimplantés ou optimisés par différentes personnes au cours de l'évolution de l'application. Par exemple, le module d'acquisition est implanté en utilisant la librairie *Blinky* [51] pour contrôler les caméras Firewire de GrImage. Une version alternative utilise l'API *Video4Linux* pour accéder à des caméras USB, de manière transparente pour les autres modules. De même, le module de vectorisation de contour a été étendu pour permettre de calculer une vectorisation approximative [46], et ainsi contrôler la précision de la reconstruction pour maîtriser la durée des calculs en fonction des besoins de l'application.

La deuxième phase concerne l'algorithme de reconstruction lui-même. Sa parallélisation est basée sur un pipeline en 3 étapes et un découpage en tranches de l'espace 3D. Les communications liées à cette parallélisation sont effectuées par MPI. Les différents processus MPI sont vus comme un ensemble de modules dans l'application FlowVR. Les contours issus des différentes caméras doivent être diffusés vers une partie de ces modules. Le résultat de cette phase, le modèle 3D reconstruit, est produit de manière découpée par les modules correspondant à la dernière étape. Ce modèle est ensuite diffusé vers les modules de visualisation, instanciés sur chaque machine contrôlant un vidéo-projecteur du mur d'image.

Avec 6 caméras capturant des images 640×480 à 30 images/seconde, en dédiant 8 processeurs à la phase de reconstruction celle-ci est effectuée en temps réel (entre 30 et 20 fois par secondes, en fonction de la complexité du modèle reconstruit) avec une latence

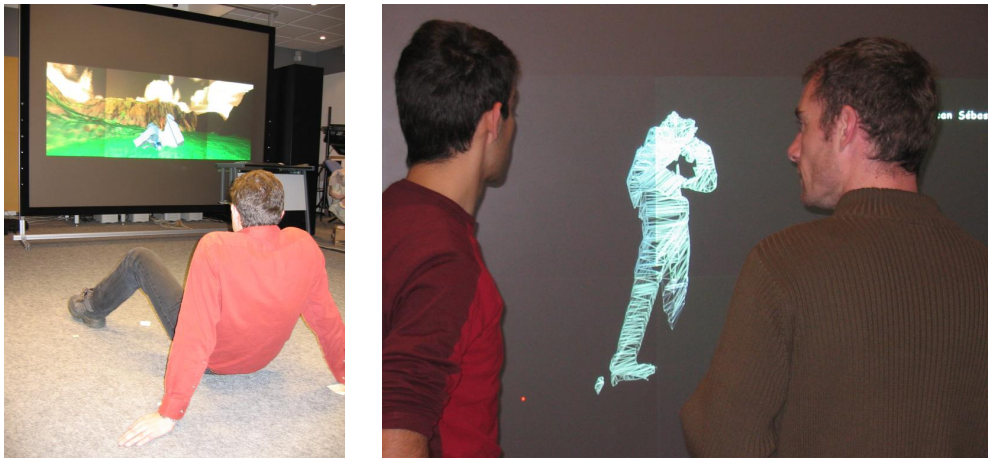


Figure 8.4 *Reconstruction 3D temps réel sur GrImage. Le mur d'image permet de visualiser les détails du modèle. La précision obtenue peut atteindre 0,5 cm.*

globale inférieure à 150 microsecondes. Ce modèle est ensuite visualisé sur le mur d'image en utilisant une application Net Juggler développée antérieurement (section 5.2 page 39), modifiée en ajoutant les ports nécessaires pour recevoir les nouvelles données. Le résultat est visible sur la figure 8.4.

La reconstruction étant réutilisée pour plusieurs applications, elle a été améliorée au cours du temps. En particulier, les communications MPI entre les phases de reconstruction ont été remplacées par des connexions FlowVR. Ceci permet d'utiliser les informations intermédiaires (comme les points générés par la phase 1) pour effectuer des calculs sans attendre la fin de la reconstruction complète (il est par exemple possible d'exprimer la boite englobante de l'objet reconstruit. Le code interne des différentes étapes a été allégé, du fait qu'il ne doit plus gérer les communicateurs MPI liés aux communications collectives. De plus, les performances ont été améliorées du fait du meilleur contrôle de la bufferisation des messages par FlowVR. Cette deuxième version a été présentée lors d'IPT 2005[6].

8.4 Interactions

Une fois les données 3D de l'utilisateur disponibles en temps réel, il est possible de concevoir de nouvelles interactions avec les objets virtuels.

Figure 8.5 *Module d'interaction Carving permettant de sculpter virtuellement un cube qui tourne à la manière d'un potier.*

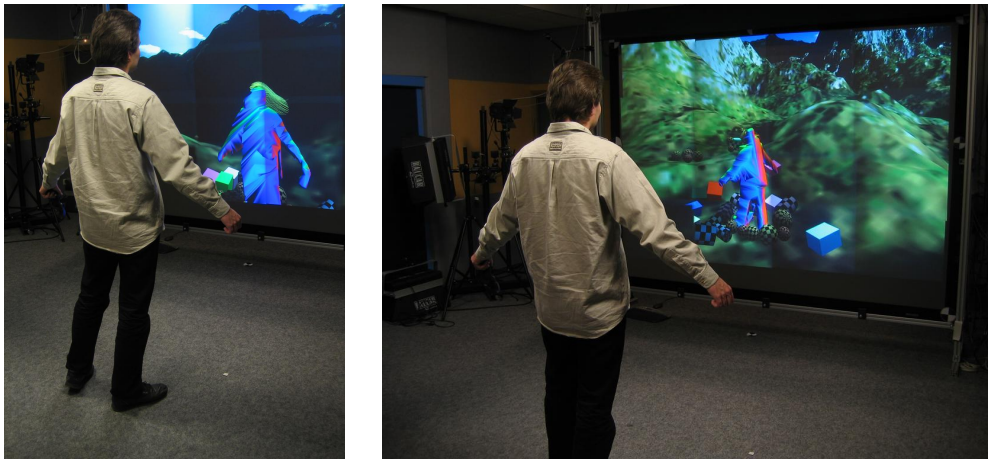
8.4.1 Sculpture

Le premier module développé permet de sculpter un octree virtuel (figure 8.5). Ce module génère un octree qui est modifié dès qu'il entre en contact avec une partie du corps de l'utilisateur. Selon le mode d'interaction il est possible de supprimer de la matière, d'en ajouter ou d'en changer la couleur. L'algorithme utilisé projette récursivement chaque cube dans les images des caméras. Le cube peut alors couvrir des zones pleines sur toutes les images, ou vide sur l'une des images, ou des cas intermédiaires. L'octree est alors modifié en fonction du mode d'interaction.

Cette application a démontré une utilisation nouvelle des informations obtenues sur tout le corps de l'utilisateur. Le fait de pouvoir utiliser tout son corps était très bénéfique. Par exemple modeler une forme de vase est très facile en utilisant le coude. Une vidéo présentant cet exemple est disponible : `http://www-id.imag.fr/~allardj/these/carve.avi`.
La configuration mur d'image est mal adaptée pour cette application du fait du décalage entre la zone d'interaction et l'affichage (environ 2 mètres) qui était donc peu immersif. Il est alors nécessaire de déplacer le point de vue virtuel pour mieux voir les actions effectuées, ce qui désoriente l'utilisateur.

8.4.2 Simulation de cheveux

Les autres interactions que nous avons testées concernent le couplage entre la reconstruction 3D et des simulations physiques. Le premier couplage, assez simple, fut de localiser et suivre la position de la tête et de la relier à une simulation de cheveux pour faire une sorte de perruque virtuelle (figure 8.6(a)). La simulation de cheveux était implantée au préalable dans le cadre de la thèse de Florence Bertails [23]. Le module de calcul de la position de la tête utilise une technique très simple basée sur l'utilisation du point le



(a) (b)

Figure 8.6 *Couplage de la reconstruction de l'utilisateur avec des simulations physiques : (a) cheveux virtuels ; (b) objets rigides.*

plus haut du modèle reconstruit. Pour ajouter de la précision une reconstruction voxelique autour de ce point est utilisée pour approximer un centre de masse beaucoup plus stable.

Ce travail a permis de mettre en lumière les problèmes de couplage multi-fréquences. En effet, la reconstruction, et donc le suivi, tourne approximativement à 30 Hz alors que la simulation nécessite 100 Hz. En utilisant un simple échantillonnage, la position en entrée de la simulation sera constante pendant quelques itérations puis va "sauter" à la position suivante. Ces sauts rendent la simulation de cheveux instable. Nous avons donc utilisé des filtres d'interpolation pour adoucir ces mouvements. Plusieurs solutions sont possibles, chacune offrant des caractéristiques de continuité et latence différentes. Une méthode qui est apparue comme le meilleur compromis dans ce cas est un filtrage par amortissement, calculant à chaque itération une position qui est une combinaison linéaire entre la dernière position calculée par le suivi et la position utilisée lors de l'itération précédente. Le coefficient utilisé pour combiner les deux valeurs permet d'adapter le filtre en fonction de la latence et de la stabilité voulue. Ce filtre peut aussi être utilisé en plus en sortie du module de suivi pour lisser les erreurs, dans le cas ou la reconstruction n'est pas stable (mauvaise soustraction de fond).

Cette application fut présentée dans le cadre de la Fête de la Science 2004 sur le campus de Grenoble. Environ 250 personnes ont pu vivre l'expérience d'être reconstruit dans un monde virtuel et jouer à la fausse blonde. Pour cet évènement nous avons utilisé une configuration impliquant deux sites distants. En effet, seule 7 machines étaient présentes sur le campus pour piloter 4 caméras et 2 vidéo-projecteurs. Tous les calculs de reconstruction et de simulation étaient effectués par 8 machines de la grappe GrImage, au travers d'un lien Gigabit de 13 kilomètres du projet *VTHD*. Du fait de sa rapidité ce lien n'introduisait pas une grande latence additionnelle. En une journée de

préparation et deux jours de démonstration cette application a tout de même utilisé environ deux Tera-octets de bande passante. Une vidéo de cet évènement est disponible : http://www-id.imag.fr/~allardj/these/sef2004.avi.

8.4.3 Collisions

Un deuxième couplage plus complexe concerne la gestion des collisions entre le modèle reconstruit et les objets rigides de l'environnement virtuel (figure 8.6(b)). Ce type d'interaction permet d'obtenir un sentiment de présence dans l'environnement virtuel, du fait que le corps de l'utilisateur ne passe plus au travers des objets mais a une influence réaliste sur eux. Nous utilisons un module calculant les collisions entre objets en testant les triangles de chaque objet avec des champs de distance signée des autres objets [77, 35]. Pour calculer ce champ de distance pour le modèle reconstruit, nous utilisons une reconstruction voxelique suivit d'un module de calcul de distance signée basé sur l'algorithme présenté dans [155].

Les collisions obtenues entre l'utilisateur et les objets virtuelles sont relativement réalistes, toutefois il manque l'information de vitesse de déplacement de l'utilisateur, ce qui fait qu'il n'est pas possible de donner une impulsion aux objets, comme taper dans un ballon. En revanche, quand ce sont les objets qui se déplacent la collision est réaliste. Une vidéo présentant cette application est disponible : http://www-id.imag.fr/~allardj/these/collision.avi.

Ce dernier couplage, ainsi que des couplages intégrant plusieurs simulations simultanément, sera développé dans la partie IV (page 133).

8.5 Contrôle et paramétrage

Contrôler tous les paramètres des applications les plus complexes devient vite problématique. Utiliser des raccourcis clavier ou les boutons d'un Pad n'est pas facile à maîtriser car ces actions changent en fonction de l'application. Pour résoudre ce problème nous avons développé une interface 2D assez simple offrant un certain nombre d'éléments de contrôle reliés à l'activation de certains modules, des paramètres des simulations physiques ou de filtrage des données (figure 8.7). Chaque élément est lié à un port de sortie qui est ensuite connecté vers le bon module ou filtre dans l'application. Un autre module permet de recevoir d'autres évènements liés aux boutons du Pad qui peuvent aussi être connectés à certains paramètres utilisés fréquemment. L'influence de ce contrôle sur l'exécution de l'application est visible sur la figure 8.8. Ce type de paramétrage pendant l'exécution est important pour tester l'utilité de chaque fonctionnalité séparément.

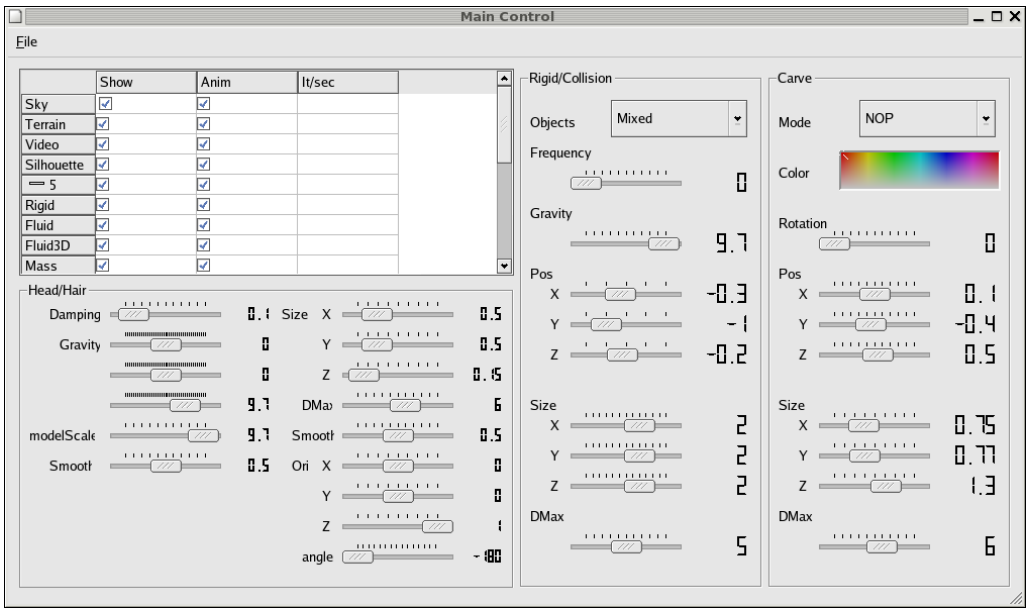


Figure 8.7 *Interface utilisateur 2D de contrôle de l'application.*

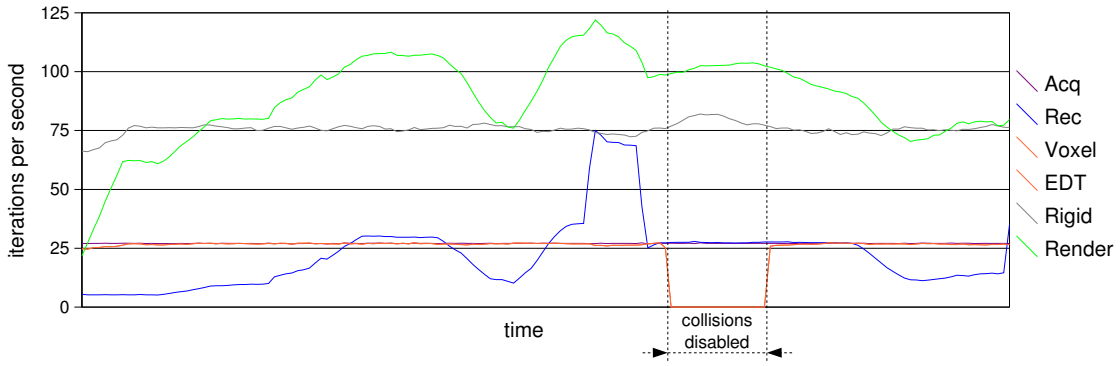


Figure 8.8 *Fréquence d'activation de chaque partie de l'application de collisions. En utilisant l'interface de contrôles certaines parties peuvent être désactivées.*

Bilan 9

Nous avons présenté FlowVR, un middleware dédié au couplage de codes parallèles interactifs. Il repose sur un modèle simple, représentant une application sous forme d'un graphe de flux de données reliant des *modules* et utilisant des *filtres* pour implanter des communications collectives et des opérations sur les données ; et des *synchroniseurs* pour appliquer des politiques de synchronisation entre les différentes parties de l'application.

La construction de l'application se fait en deux étapes distinctes. Les modules sont programmés le plus indépendemment possible du reste de l'application. Ils proviennent souvent de codes existants. Une API de programmation très simple leur permet de déclarer des *ports* de communication. Ils utilisent ensuite un modèle itératif. Entre chaque itération ils appellent la méthode *wait()*. C'est une opération bloquante dont la sémantique est de lire un nouveau message sur tous les ports d'entrée du module. Ensuite le module peut lire le message de chacun des ports d'entrée et utilise la méthode *put()* pour envoyer au plus un message sur chaque port de sortie de façon non bloquante.

Ces modules sont ensuite connectés entre-eux pour former l'application complète. Les connexions sont point à point mais peuvent passer par des noeuds de routages et des filtres pour exprimer les différents types de communications collectives et au besoin les conversions ou répartitions de données.

Un même module peut être utilisé pour un calcul hors-ligne en temps dilaté comme dans une application interactive. La différence se retrouve souvent dans les politiques de synchronisation entre les différentes parties de l'application. Dans un calcul hors-ligne les connexions sont souvent *FIFO*, chaque module travaillant de manière synchronisé sans perte de donnée. Dans une application interactive en revanche on cherche à obtenir le maximum de performance, même si une partie comme une simulation coûteuse ne peut pas suivre le même rythme. Dans ce cas on utilise des mécanismes d'échantillonnage des

données pour désynchroniser chaque partie. Ses mécanismes sont adaptés en fonction de la nature des données et des exigences de cohérence de l'utilisateur.

FlowVR utilise un système de spécification de ce graphe d'application utilisant des descriptions XML générées via un ensemble de scripts. Ces graphes peuvent être construits morceau par morceau et offrent une représentation visuelle de l'application.

En découpant les applications en tâches génériques nous avons construit des applications de plus en plus ambitieuses en réutilisant au maximum des composants d'anciennes applications ou d'autres codes existants. Par exemple la simulation de fluide 2D parallèle, qui fut développée en tant qu'exemple d'application Net Juggler il y a quatre ans, a été convertie en module FlowVR et utilisée depuis dans plusieurs applications sans retoucher au code. Ceci réponds bien à la problématique initiale, à savoir le développement d'un outil permettant le couplage de composants parallèles à l'intérieur d'une application interactive.

Grâce à des concepts repris des méthodes de couplage par composants et adaptés pour l'interaction, FlowVR permet de combiner de nombreux travaux existants issus des domaines comme le calcul parallèle et la réalité virtuelle. Ainsi les applications les plus récentes intègrent des modules de traitement d'images et de vision 3D ; des simulations parallèles de fluide, de tissus, de collisions, de cheveux utilisant MPI, Athapascan, AnimAL ; une extraction de données à visualiser parallélisée sous VTK (section 13.1 page 121) ; et enfin un moteur d'affichage OpenGL distribué par Net Juggler ou VR Juggler 2.

D'autres équipes ont développé des applications utilisant FlowVR. Dans le cadre du projet RNTL Geobench, l'équipe de Sabine Coquillart à couplé avec FlowVR un moteur de rendu haptique avec une visualisation sous Amira pour un workbench, utilisant une grappe de PC. Le LIFO en collaboration avec le BRGM travaille sur du rendu de grands terrains ou couches géologiques utilisant plusieurs machines pour extraire les données et les visualiser grâce à différents algorithmes de niveaux de détails dynamiques [71, 72].

Le modèle et l'implantation de FlowVR semble bien convenir à ces applications. Plusieurs améliorations sont toutefois apparues comme nécessaires. Les performances de l'implantation actuelle pourraient être améliorées en optimisant certaines parties. En particulier, l'implantation réseau utilise la couche TCP qui entraîne plusieurs recopies. Cette première implantation pourrait être améliorée en utilisant des travaux plus évolués comme PadicoTM [49] (section 3.1.4 page 21). Le système de spécification des applications par des fichiers XML et des scripts Perl est parfois difficile à maîtriser, surtout pour créer de petites applications. Plusieurs stages en cours essayent de définir une interface permettant de construire les applications de manière plus visuelle en cachant l'ensemble des outils utilisés.

Ensuite pour de applications de grande envergure l'allocation des ressources reste un problème difficile. Le placement des modules sur les machines et le routage des communications reste à la charge de l'utilisateur. Pour réduire cette charge il serait nécessaire

de modéliser les performances du système, peut-être en utilisant les mesures du mécanisme de traces, pour déterminer un placement et une politique d'activation des modules efficace. Dans le cas général ce problème est NP-Complet. Toutefois en ce restreignant à des cas simples (connections FIFO uniquement par exemple), il est possible d'utiliser des techniques comme l'*ordonnancement cyclique* [44], en les modifiant pour intégrer le critère de latence. Le stage de DEA de Christophe Sadoine co-encadré par Denis Tristram va dans ce sens.

Enfin FlowVR est un outil de couplage d'assez bas niveau. Des surcouches spécialisées pour certains types d'applications sont souvent très utiles pour en faciliter le développement. Dans les dernières versions de FlowVR des types de données sont fournis permettant de gérer facilement les vecteurs et les matrices de valeurs en gérant automatiquement les estampilles pour spécifier les dimensions et le type des données. Les couplages à l'intérieur des premières applications, telles que présentées dans ce chapitre, sont réalisés manuellement. Quand un nouveau composant doit être visualisé ou une nouvelle simulation doit interagir avec le reste de l'environnement il faut souvent modifier certaines parties existantes en ajoutant des ports pour communiquer les données concernées. Un système de visualisation et d'interaction plus générique permettrai de faciliter ces ajouts. Les prochaines parties présenteront nos travaux en ce sens.

III

FlowVR Render

La phase de rendu d'une application de réalité virtuelle nécessite l'intégration des informations concernant tous les objets de l'environnement virtuel. Même si le reste de l'application est construit de façon modulaire, l'ajout d'un nouveau type d'objet virtuel requiert fréquemment des modifications dans le code de rendu. Dans cette partie une architecture de rendu distribué est présentée, permettant de modulariser la phase de rendu pour minimiser ces modifications, tout en optimisant les performances dans le cadre d'une exécution sur grappe pilotant plusieurs projecteurs.

Notre approche est basée sur la définition d'un nouveau modèle de description de la scène 3D, exploitant les dernières avancées des cartes graphiques, les *shaders*, pour alléger les transferts réseau et permettre un implantation efficaces des opérations de fusion et découpage des données. Les performances obtenues sont comparées à un outil existant, et de nombreuses applications exploitant cette nouvelle architectures sont présentées.

Ce travail a été publié lors de la conférence IEEE Visualization 2005 [12].

Contexte et problématique 10

"ALL OUR KNOWLEDGE HAS ITS ORIGINS IN OUR PERCEPTIONS."
Leonardo da Vinci

Un système de rendu distribué est primordial pour pouvoir augmenter la surface et la résolution d'affichage, ainsi que la taille de l'environnement virtuel. Plusieurs machines sont mises à contribution pour calculer chacune une partie de l'image finale et/ou des objets dans l'environnement virtuel. Cette parallélisation peut intervenir à plusieurs niveaux dans le pipeline de l'application (chapitre 4). Nous nous intéressons ici au cas ou cette parallélisation intervient entre la génération de la scène 3D et le rendu lui même.

Traditionnellement, les systèmes de rendu distribué se basent sur une API graphique existante (OpenGL ou graphe de scène) et envoient les données correspondantes sur le réseau. L'avantage de cette approche est le support relativement aisé des applications existantes utilisant déjà cette API. En revanche l'algorithme de distribution est limité par la sémantique de l'API utilisée ainsi que le niveau d'information disponible.

Les approches basées sur les graphes de scènes [82, 123, 154] ont à leur disposition plus d'informations, mais doivent gérer les interdépendances entre les objets de la scène dues à la hiérarchie du graphe de scène. Une telle hiérarchie est utile pour gérer les différentes parties de la scène et faire des traitements en passant à l'échelle sur les grandes scènes (en construisant une hiérarchie de boites englobantes et ainsi rejeter facilement les parties non visibles). En revanche, cette hiérarchie est plus difficile à construire et à justifier pour les scènes très dynamiques, ou utilisant beaucoup les *shaders*. Les shaders sont de petits programmes exécutés par la carte graphique et contrôlant les transformations effectuées sur les sommets des objets ainsi que le calcul de la couleur des pixels, remplaçant ainsi l'étape traditionnelle de transformation des sommets par une pile de matrices. Avec un shader rien ne garantit qu'un objet va appliquer sa matrice de transformation tel quel, ni utiliser celle du noeud père dans le graphe de scène.

Les travaux liés aux API de rendu (section 2.5 page 13) et au rendu distribué (section 4.1 page 28) ont déjà été présentés, nous allons rappeler dans ce chapitre les concepts

utilisés par OpenGL, Chromium et les shaders afin de présenter le contexte et la problématique de notre travail.

10.1 OpenGL

OpenGL [159] est actuellement l'API de rendu prédominante, étant disponible pour la plupart des plateformes et des cartes graphiques. Elle utilise un modèle assez bas niveau basé sur une machine à état spécifiant comment les objets doivent être affichés (transformations, lumières, matériaux, transparence, textures, etc), ainsi qu'une description des objets eux mêmes selon un mode *immédiat*, c'est-à-dire ou chaque primitive est décrite séquentiellement pour chaque nouvelle image. Depuis l'introduction d'OpenGL en 1992, les machines et les cartes graphiques ont évolué de façon significative. Pour suivre cette évolution un mécanisme d'extensions est utilisé, permettant aux fabricants de cartes graphiques d'exposer des fonctionnalités additionnelles, correspondant aux nouvelles possibilités des cartes graphiques. Périodiquement une nouvelle version d'OpenGL est créée par l'*Architecture Review Board* (*ARB*), standardisant les extensions les plus utiles. La dernière version, OpenGL 2.0, est disponible depuis septembre 2004 et ajoute le support de fonctionnalités tel que les textures rectangulaires ou les langages de shaders de haut niveau.

L'utilisation du mode immédiat pur requiert une retransmission de toutes les primitives graphiques à chaque image, et ne peut donc pas bénéficier de la cohérence entre les images. Cela introduit un important surcoût au niveau du CPU, du GPU (*Graphics Processing Unit* – le processeur de la carte graphique), et du bus de communication entre les deux. Ce problème s'est intensifié avec l'évolution des performances des cartes graphiques. Pour réduire ce goulet d'étranglement, OpenGL a introduit plusieurs mécanismes optimisant les transferts de données vers la carte graphique. Il y a entre autres le mécanisme originel des *display-lists*, les *compiled vertex arrays*, et les récents *vertex/pixel buffer objects* d'OpenGL 2.0.

Chaque nouvelle version d'OpenGL est compatible avec toutes les versions précédentes. De ce fait, il existe de nombreuses fonctionnalités obsolètes ou redondantes, comme les multiples façons de spécifier les données des sommets, ou encore toute la machine à état liée au calcul de l'apparence des objets rendue obsolète par la programmabilité des shaders [19]. Ceci augmente significativement la complexité des implantations d'OpenGL. Un effort pour épurer les commandes et les états obsolètes du coeur d'OpenGL a été proposé pour OpenGL 2.0. Cette proposition n'a pas été acceptée, mais a abouti à une API dérivée OpenGL ES [28], dédiée aux systèmes embarqués. Cette API est utilisée dans de nombreux produits comme la future console Sony PlayStation 3.

10.2 Chromium

Chromium [84] est un outil permettant de découper et recombiner des flux de commandes OpenGL afin d'implanter un mécanisme de rendu distribué basé sur le principe du *sort-first* (section 4.1.2 page 29). Les primitives graphiques de la scène à visualiser sont produites par une application exécutée sur une ou plusieurs machines. Ces primitives sont ensuite triées et acheminées vers d'autres machines effectuant le rendu lui-même.

L'utilisation d'OpenGL permet le support de nombreuses applications sans nécessiter de modifications, Chromium remplaçant le driver OpenGL du point de vue de l'application. Ceci permet l'utilisation directe d'applications propriétaires sur un mur d'image. Chromium intercepte les commandes OpenGL émises par l'application et utilise un réseau de filtres implantant un système de caches avancés et de compression pour optimiser les communications.

Pour supporter des optimisations ou des schémas de rendu avancés, Chromium défini un certain nombre d'extensions spécifiques. Ainsi, l'utilisation parallèle de plusieurs applications clientes se répartissant le calcul de la scène est possible grâce à des extensions permettant de spécifier l'ordre de recombinaison des différents flux. Cette extension utilise un mécanisme de barrières et de sémaphores permettant d'indiquer les sections de commandes atomiques et l'ordre possible de recombinaison entre celles-ci. D'autres extensions permettent de réduire le surcoût lié à Chromium, comme la transmission des boites englobantes associées aux objets, permettant aux filtres de tri de calculer rapidement leur visibilité. Toutefois l'utilisation de ces extensions requiert une adaptation spécifique du code de l'application.

Utiliser OpenGL comme base de distribution rend difficiles les opérations de découpage et recombinaison de flux graphiques qui servent de base au rendu distribué. En particulier, les commandes OpenGL partageant un état commun, il faut faire attention de bien respecter les modifications faites à cet état. Le mode immédiat d'OpenGL impose de retransmettre beaucoup d'informations à chaque image. Ces retransmissions deviennent rapidement un goulot d'étranglement, même en rendu local. Plusieurs optimisations ont été développées et intégrées à OpenGL. Tous ces mécanismes doivent être supportés par le système de rendu distribué, ce qui augmente beaucoup la complexité de son implantation.

10.3 Shaders

Des shaders procéduraux sont très souvent utilisés pour le rendu logiciel hors ligne [80] pour spécifier l'apparence visuelle des objets. Les premières cartes graphiques étaient limitées à des calculs fixés, configurés au travers d'un ensemble de paramètres. Les nouvelles générations de carte graphique sont maintenant capable d'exécuter des shaders complètement programmables [133, 142]. Les modèles les plus récents [100] permettent

l'utilisation de langages de haut niveau [110, 153] incluant des appels de fonctions, des boucles, des structures conditionnelles, et des calculs et données utilisant une précision de 32 bits en virgule flottante.

Les shaders apporte une flexibilité et une précision additionnelle qui permet aux cartes graphiques d'exécuter des algorithmes jusqu'alors réservé à une exécution sur CPU. Ceci concerne par exemple les modèles d'éclairage de haute qualité (évaluation par pixel, ombres douces), la colorisation dessinée [69], ou le rendu volumique [151]. Ce déplacement de certains calculs du CPU vers le GPU permet parfois de réduire les transferts de données entre-eux. Par exemple en visualisation une *fonction de transfert* est souvent appliquée aux données brutes pour obtenir une représentation visuelle (couleur). Si ce calcul est effectué par le CPU, alors le résultat doit être retransmis à chaque fois que la fonction de transfert est modifiée. L'utilisation d'un shader pour appliquer la fonction de transfert permet de ne transmettre qu'une seule fois les données brutes. Un changement de la fonction n'entraîne alors qu'une modification des paramètres du shader.

10.4 Bilan

En utilisant les fonctions d'optimisation des transmissions de données comme les *buffers objects* d'OpenGL et les shaders matériels, les applications peuvent exploiter pleinement les dernières avancées des cartes graphiques. Cependant, la machine à état d'OpenGL et les nombreuses fonctions redondantes encore supportées augmentent considérablement la complexité et bride l'efficacité de toute implantation, en particulier les systèmes distribués.

L'utilisation de shaders permet de répartir les calculs entre les CPU et les GPU. En conséquence le niveau des informations envoyées à la carte graphique est modifié, celle-ci pouvant désormais traiter des données génériques en plus de simple données graphiques (couleurs, positions). D'autre part, le changement des paramètres des calculs implanté par des shaders ne nécessite plus la retransmission de toutes les autres données. Dans un contexte distribué ou les primitives graphiques sont produites sur des machines distinctes, et parfois distantes, des machines de rendu ceci peut dramatiquement changer les performances de la chaîne d'affichage.

Basé sur ces constatations, nous proposons un nouveau protocole de rendu distribué *sort-first*, reposant sur une utilisation omniprésente de shaders et dédié au rendu parallèle haute-performance.

Description d'environnements virtuels distribués 11

Dans ce chapitre, nous présentons une nouvelle approche de rendu distribué, basé sur un modèle de description de la scène spécialement conçu pour une construction la plus modulaire possible de la tâche de rendu, en utilisant les possibilités offertes par les shaders des dernières générations de cartes graphiques. Bien que les idées utilisées soient transposables à d'autres environnements, FlowVR est utilisé comme cadre pour la conception et l'implantation de ce système de rendu distribué, que nous appellerons *FlowVR Render*.

Après avoir défini les principes fondamentaux, nous nous attacherons aux éléments permettant de décrire la scène 3D et de transmettre cette description sur le réseau.

11.1 Principes fondamentaux

Nous définissons un modèle de description de la scène 3D. Cette description est générée par des modules appelés *viewers*, et envoyée aux modules appelés *renderers* effectuant le rendu sur les cartes graphiques.

Le modèle de FlowVR Render est basé sur les principes suivants :

– Chaque objet de la scène est décrit comme un ensemble de groupes de polygones (*batches*). Chaque groupe est indépendant des autres et son apparence visuelle est décrite à l'aide de shaders. Un groupe de polygones est appelé *primitive* dans le modèle.

– Les primitives ne sont pas ordonnées sauf dans certains cas particuliers (objets transparents, éléments d'interface utilisateur) spécifiés par le *viewer*. Ceci permet

aux *renderers* d'optimiser l'ordre local de rendu, réduisant par exemple les changements de textures et de shaders.
- Des informations de haut niveau comme la boite englobante ou les changements depuis la dernière image peuvent être spécifiés directement par les *viewers*, permettant de réduire les surcoûts de traitement et de transmission réseau. Pour la plupart des applications ces informations sont directement disponibles.
- Les shaders ne nécessitant qu'un petit nombre de paramètres, il n'est pas nécessaire de maintenir un état commun à toutes les primitives, ce qui simplifie beaucoup le protocole de transmission. Les opérations de découpage et recombinaison de plusieurs flux graphiques sont peu coûteuses du fait de l'indépendance des primitives.
- Les informations communiquées sont strictement unidirectionnelles, depuis les *viewers* jusqu'aux *renderers*. Ceci évite les allers-retours coûteux et permet de désynchroniser l'exécution de chaque module (par exemple envoyer les données vers un fichier pour les relire plus tard).
- Les shaders permettent d'utiliser facilement toute la puissance des dernières cartes graphiques. Certains calculs comme l'application de fonctions de transfert pour obtenir des informations de couleur à partir de données brutes peuvent être effectués par la carte graphique, libérant ainsi le CPU pour d'autres tâches.

11.2 Primitives graphiques

Comme indiqué dans la section précédente, notre modèle se base sur une scène 3D formée d'un ensemble de *primitives*. Chaque primitive décrit un groupe de polygones en spécifiant l'ensemble des *ressources* (coordonnées, textures, shaders) à utiliser. Les propriétés associées aux primitives et ressources sont présentées sur la figure 11.1.

Pour éviter la duplication des ressources utilisées par plusieurs primitives, ainsi que des ressources et primitives inchangées entre chaque image, il est nécessaire de pouvoir identifier les ressources et primitives envoyées et ainsi pouvoir les réutiliser. On associe donc à chaque ressource et primitive un identifiant unique *ID*. Chaque primitive spécifie les identifiants des ressources qu'elle utilise. Ceci introduit une dépendance unidirectionnelle entre les primitives et les ressources, mais pas d'interdépendance entre les primitives elles-mêmes.

Le système doit garantir que les identifiants utilisés soient globalement uniques pour toute l'application distribuée. Il existe plusieurs méthodes pour le faire, en particulier utiliser un serveur centralisé de nommage, ou alors inclure dans l'identifiant des informations locales à chaque machine garantissant sont unicité. La première méthode permet de générer des identifiants de petite taille, mais introduit un point de contention qui limite le passage à l'échelle du système. L'implantation de FlowVR Render utilise donc la seconde approche. Des identifiants sur 64 bits sont générés en combinant un compteur atomique local à chaque machine avec l'adresse IP de la machine.

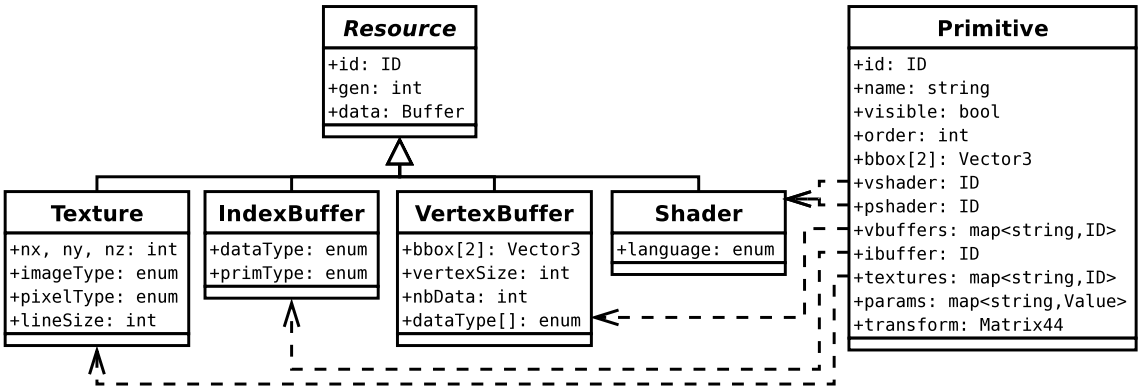


Figure 11.1 *Schéma UML des objets utilisés dans la description de la scène graphique.*

11.2.1 Ressource

En plus de son identifiant, chaque ressource possède un nombre *gen* spécifiant la génération actuelle. Ce nombre doit être incrémenté à chaque fois que la ressource est modifiée, et permet de savoir quelle donnée est la plus récente. Un numéro de génération égal à -1 signifie que la ressource ne sera a priori pas modifiée et que donc le module de rendu peut en optimiser son stockage (transférer les données directement dans la mémoire de la carte graphique par exemple). Ceci n'est toutefois qu'une indication, il est tout à fait possible de modifier la ressource plus tard, en réponse à un réglage de l'utilisateur par exemple. Le but de cette distinction est de fournir une heuristique simple pour déterminer ou stocker chaque ressource pour le rendu. Cela peut aussi être utile pour le filtrage des transferts réseaux. Une donnée statique peut être transmise qu'elle soit visible ou non, étant donné qu'elle va être utilisée pendant une longue période, alors qu'une donnée plus dynamique mérite d'être filtrée vu qu'elle est retransmise souvent et donc utilise une part importante de la bande passante.

Les données elles-mêmes de la ressource sont stockées dans un buffer FlowVR, qui est donc en mémoire partagée, ce qui évite de les dupliquer lorsque plusieurs modules ou filtres les utilisent. Le reste des propriétés des ressources dépendent du type de la ressource, qui peut être une *Texture*, un *Index Buffer*, un *Vertex Buffer*, ou un *Shader*.

11.2.1.1 Texture

Une texture stocke une table de valeurs 1D, 2D ou 3D, utilisée généralement pour plaquer une image sur les objets de la scène, mais peut aussi servir à stocker d'autres informations (normal maps, voxels, *Look Up Tables*) utilisées par le shader. Chaque texture possède une taille *nx*, *ny* et *nz* selon les axes X, Y et Z. Cette taille est égale à 0 si la texture est de dimension inférieure.

Les valeurs stockées à l'intérieur de chaque pixel de la texture sont spécifiées selon un

type de donnée spécifié dans *pixelType* en utilisant un type énuméré regroupant les types de base (*byte*, *int*, *float*, ...), certains types spéciaux (comme *null* spécifiant que toutes les données sont nulles), ainsi que de petits vecteurs ou matrices de taille fixe. Par exemple, une image couleur utilise généralement un vecteur de 3 octets pour chaque pixel.

Les données des pixels peuvent être interprétées de manière différente selon la convention utilisée, comme par exemple *RGB* spécifiant que la première valeur représente la composante rouge, la deuxième représente le vert, et la troisième le bleu, contrairement à la convention *BGR* qui spécifie l'ordre opposé. Cette convention est indiquée par *imageType*. Enfin les données sont stockées en mémoire de manière contiguë par ligne puis par plan. Il arrive souvent que le début de chaque ligne soit aligné sur un bloc de n octets. Pour prendre en compte cet alignement la variable *lineSize* indique le décalage entre chaque début de ligne.

11.2.1.2 Vertex Buffer

Les sommets qui composent les polygones à afficher sont spécifiés dans des ressources de type *vertex buffer*. Chaque sommet contient plusieurs données appelées attributs. Le nombre de ces attributs est spécifié par la variable *nbData*, et leur type est stocké dans le tableau *dataType*. Ce tableau utilise le même type énuméré que celui utilisé précédemment pour les pixels des textures. La taille des données de chaque sommet est spécifiée dans la variable *vertexSize*. Elle permet aussi de connaître le nombre de sommets contenus dans le buffer en divisant la taille totale des données par la taille de chaque sommet.

Un vertex buffer peut stocker des données indiquant la position 3D des sommets. Cette information à une incidence sur la visibilité des polygones et peut être utilisée pour filtrer les communications réseaux. Pour faciliter ce filtrage la boite englobante des sommets contenus dans le buffer est indiquée dans *bbox*. Dans le cas ou le buffer ne contient pas de donnée de positions mais uniquement d'autres attributs (couleurs, vecteurs normaux), alors cette boite est spécifiée comme vide.

11.2.1.3 Index Buffer

Une fois les attributs des sommets spécifiés, il est nécessaire de les relier pour former des points, lignes, ou polygones. Pour cela on utilise un *index buffer* qui contient les indices des sommets à utiliser. Ces indices sont stockés dans le buffer de donnée de la ressource en utilisant le format spécifié par *dataType* (généralement des entiers sur 8, 16 ou 32 bits en fonction du nombre de sommets). Dans certains cas les sommets sont utilisés exactement dans le même ordre qu'ils sont stockés dans les vertex buffers. Il n'est alors pas nécessaire de stocker les indices, et *dataType* contient alors *null*.

Le type de primitive formé par les sommets est spécifié dans la variable *primType* et correspond aux primitives classiques utilisées par OpenGL (figure 11.2).

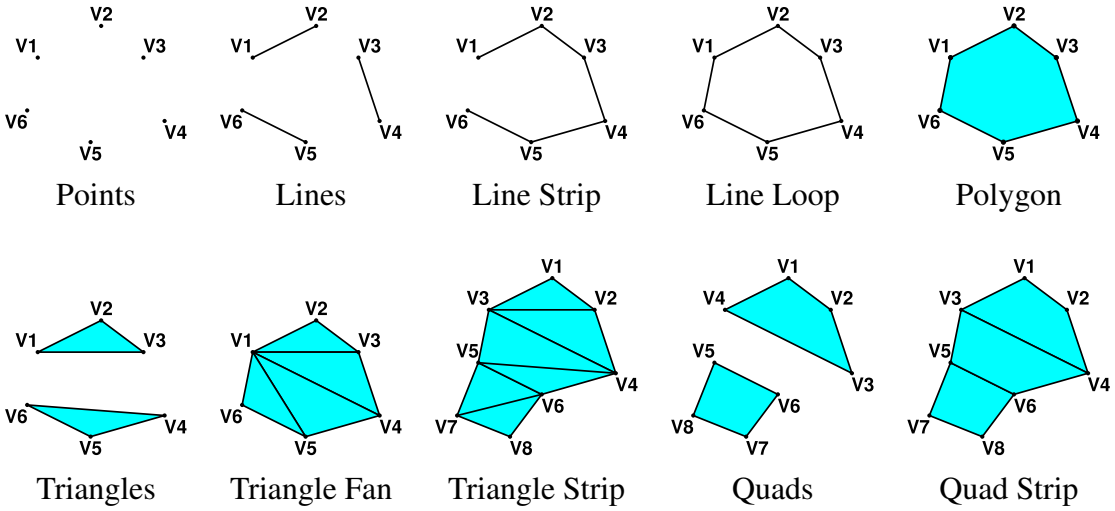


Figure 11.2 *Types de primitives (identique à OpenGL). Points, Lines et Triangles sont les plus utilisés. Les autres servent à optimiser le nombre d'indices à transmettre.*

11.2.1.4 Shader

Le dernier type de ressource concerne les shaders. Un shader est un petit programme qui utilise les données provenant des textures, des attributs des sommets, ainsi que d'autres paramètres spécifiques à l'objet à afficher (matrices de transformations, propriétés de la surface, ...) pour calculer son apparence visuelle finale. Ce calcul se fait en deux étapes :

vertex shader : utilise les attributs de chaque sommet pour en calculer sa projection sur l'écran ainsi que les valeurs à interpoler sur les polygones.

pixel shader : utilise les valeurs calculées par le vertex shader ainsi que les textures pour produire la couleur de l'objet à chaque pixel. Cette couleur peut contenir un coefficient de transparence, et sera dans ce cas combinée avec la couleur des autres objets de la scène pour obtenir la couleur du pixel final.

Il existe plusieurs langages permettant de spécifier ces shaders, notamment Cg [110] et GLSL [153], deux langages proches de C développés par NVIDIA et 3DLabs respectivement, ainsi que d'autres langages assembleurs plus bas niveau. Pour ne pas dépendre d'un seul langage, il est possible de spécifier quel est celui qu'on utilise via la variable *language*. En revanche l'implantation du moteur de rendu ou le driver de la carte graphique peut ne supporter que certains langages. Par exemple, l'implantation actuelle de FlowVR Render n'implante que les shaders Cg, principalement du au fait que des drivers supportant OpenGL 2.0 et GLSL ne sont disponibles que depuis très récemment sous Linux.

Les données d'un shader correspondent uniquement à son code source. Celui-ci est en général très cours (quelques lignes). Il ne semble donc pas très utile de le placer dans

une ressource externe aux primitives. Cependant, un shader implique un coût important de compilation sur la carte graphique, et il est aussi bénéfique de limiter les changements de shaders entre les différents objets de la scène. De ce fait il est important qu'un même shader puisse être explicitement partagé par plusieurs primitives.

11.2.2 Primitive

Une primitive rassemble toutes les données utilisées pour afficher un groupe de polygones. Ces données proviennent de ressources externes spécifiées par leurs identifiants, comme le vertex et pixel shader (*vshader* et *pshader*), l'index buffer (*ibuffer*), ainsi que les textures et les attributs des sommets (*textures* et *vbuffers*). Ces deux dernières données sont reliées à une variable utilisée par les shaders en spécifiant le nom de cette variable telle qu'elle apparaît dans le code source du shader. Des paramètres additionnels utilisés par les shaders ou par OpenGL pour régler par exemple la transparence, les tests de profondeurs ou de stencil, sont spécifiés dans la variable *params*.

Par défaut, les primitives de la scène sont considérées comme non ordonnées et le module de rendu est libre de les afficher dans n'importe quelle ordre. Toutefois pour certaines primitives, notamment utilisant de la transparence ou implantant un mécanisme particulier comme des ombres stencil, il est nécessaire d'imposer un ordre particulier. Pour cela on utilise la valeur de *order* comme une clé de tri. Les primitives sont triées par ordre croissant avant l'affichage. Les primitives utilisant la même valeur *order*, comme c'est le cas si la valeur par défaut 0 n'est pas modifiée, sont affichées dans un ordre non spécifié. Ce système permet de définir un ordre de précédance entre les primitives, sans toutefois connaître précisément les primitives envoyées par les différents viewers de l'application. Par exemple la plupart des applications actuelles utilisent par convention -10 pour le fond de la scène (*skybox*), 0 pour les objets courants, 10 pour les objets transparents, et 20 pour les éléments d'interface utilisateur (*overlays*).

Un paramètre particulier *transform* correspond à la matrice de transformation de l'objet dans la scène. Bien que cette information puisse être spécifiée comme un paramètre normal des shaders, il est souvent utile de le traiter de manière particulière. Dans le cas ou le shader applique cette matrice aux coordonnées des sommets, elle permet de calculer la boite englobante de la primitive dans la scène en appliquant la matrice *transform* à la boite englobante des attributs des sommets (section 11.2.1.2). Dans les autres cas, c'est-à-dire si le shader utilisé effectue d'autres transformations, alors le viewer doit spécifier la boite englobante de la primitive dans la variable *bbox*. Ce mécanisme permet de n'avoir à spécifier explicitement la boite englobante des primitives que rarement, même quand on change la matrice de transformation.

Enfin, une dernière information facultative *name* spécifie un nom textuel pour la primitive. Ce nom n'est pas obligatoirement unique et sert surtout pour aider l'utilisateur à reconnaître les primitives dans les messages d'erreurs ou visuellement dans un mode d'affichage de contrôle. Il peut aussi être utilisé par des filtres comme la capture de la

scène pour produire des fichiers X3D par exemple.

Un type de paramètres particuliers dans la scène concerne la gestion de la caméra. Pour les contrôler on utilise une primitive spéciale dont l'identifiant est *ID_CAMERA*. Cette primitive contient la matrice de transformation de la caméra, ainsi que d'autres paramètres tels que la focale. Si aucun viewer ne spécifie ces paramètres, les renderers utilisent les boites englobantes des primitives pour positionner la caméra de manière à voir toute la scène. L'utilisateur peut alors utiliser la souris ou le clavier pour se déplacer dans la scène. Ce contrôle par défaut de la caméra est désactivé dès qu'un viewer spécifie les paramètres de la caméra, ce qui permet d'implanter d'autres modes de déplacement (utilisant des traqueurs 3D, prenant en compte la topologie du terrain virtuel, etc).

11.3 Protocole de communication

La description des primitives graphiques de la scène doit être transmise depuis les viewers, responsables des différents objets de la scène, jusqu'aux renderers, qui vont chacun calculer une partie de l'image finale. Cette transmission doit utiliser un protocole efficace en bande passante, étant donné que les viewers et les renderers sont souvent placés sur des machines différentes, et doit en même temps être facile à traiter par les filtres intermédiaires qui vont être responsables du filtrage et de la combinaison de ces données. Etant donné qu'une partie souvent importante des données de la scène sont identiques d'une image à l'autre il semble bénéfique de décrire les changements dans la scène plutôt que toute la scène à chaque image. En relation avec les données décrites dans les sections précédentes, il y a plusieurs changements possibles :

Ajout d'une primitive, potentiellement en recopiant les données à partir d'une primitive existante.

Suppression d'une primitive.

Modification d'un paramètre d'une primitive, en indiquant quel paramètre, sa nouvelle valeur, ainsi que le nom associé (pour les valeurs de *vbuffers*, *textures*, et *params*).

Ajout d'une ressource, spécifiant le type de ressource, les paramètres associés ainsi que les données. Ceci peut correspondre à la création d'une nouvelle ressource ou à la mise à jour des données d'une ressource existante.

Suppression d'une ressource.

Modification d'une ressource, permettant de n'envoyer qu'une partie des données d'une ressource, dans le cas ou le reste est statique.

A chaque itération, les viewers envoient un message contenant l'ensemble des modifications à appliquer pour l'image suivante. Chacune de ces modifications est décrite dans un morceau (*chunk*) de ce message. Ces chunks sont mis bout à bout et suivent le format décrit dans la figure 11.3.

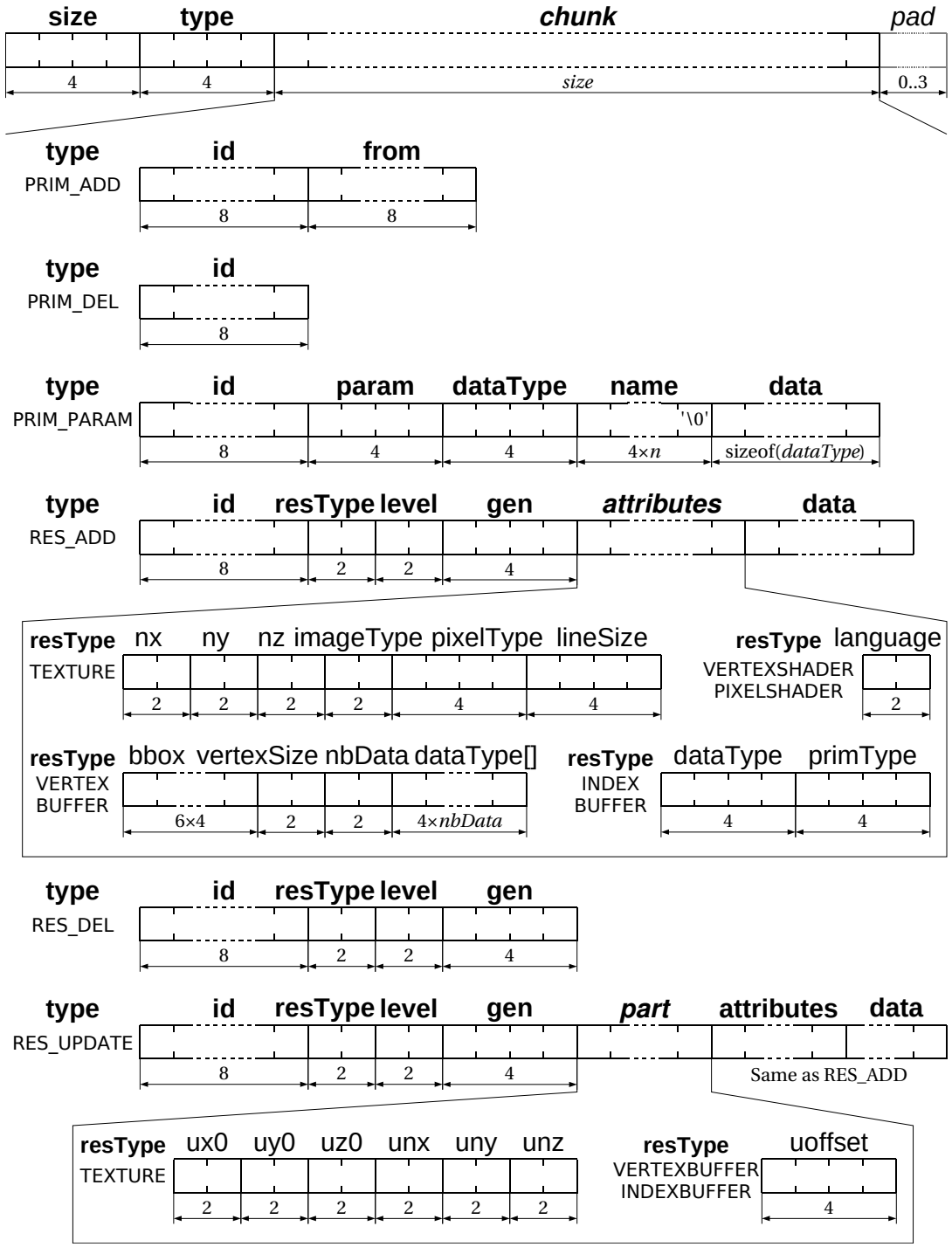
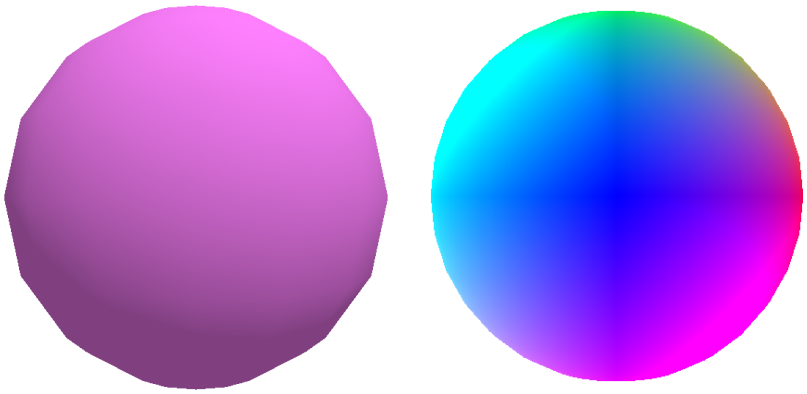


Figure 11.3 *Format des chunks contenant les mises à jours de la scène 3D. Les données correspondant aux attributs des ressources et des primitives (figure 11.1). Level est prévu pour gérer les niveaux de détails.*



(a) (b)

Figure 11.4 *Sphère affichée en utilisant (a) un shader de type Phong ou (b) un shader visualisant les composantes du vecteur normal.*

Comme il n'y a que 6 types de chunks différents les messages sont très faciles à décoder et traiter. De plus, il suffit de mettre bout à bout les différents messages pour combiner plusieurs flux. Enfin, laisser les primitives d'un viewer tel quel équivaut à envoyer un message vide, ce qui permet de désynchroniser la vitesse d'affichage de certains viewers plus lents. En effet il suffit de récupérer à chaque image les messages disponibles, et utiliser un message vide pour le cas ou il n'y a pas de nouveau message.

11.4 API

Un module peut tout à fait générer les messages de description de la scène en utilisant le format décrit dans la section précédente, mais cela l'oblige à travailler à un niveau très bas. Pour permettre de programmer les viewers plus facilement, une classe *ChunkWriter* est utilisée pour construire ces messages. Elle offre une API procédurale permettant de créer chacun des types de chunks et ensuite d'envoyer le message résultant sur un port FlowVR. Des fonctions de plus haut niveau très souvent utilisées permettent d'envoyer des modèles 3D de base (cube, sphère, cylindre), ou encore de charger à partir de fichiers les textures et les shaders.

11.5 Exemple

Voici un exemple d'un viewer simple envoyant une sphère et la faisant tournée sur elle-même. Le résultat est visible sur la figure 11.4. Cet exemple utilise *ChunkWriter* pour envoyer le modèle 3D de la sphère et un shader Cg pour implanter un calcul de lumière de type Phong (i.e. interpolation de vecteur normal et calcul de la lumière à chaque pixel). En

changeant le shader il est possible de changer complètement l'apparence visuelle, comme le montre la deuxième sphère ou des couleurs visualisent le vecteur normal de la surface.

Code du viewer

```
// Basic FlowVR Render example viewer module
// Simple sphere
#include <flowvr/module.h>
#include <flowvr/render/chunkwriter.h>
#include <stdlib.h>
using namespace flowvr::render;

int main(int argc, char** argv)
{
  // FlowVR Render output port
  SceneOutputPort pOut("scene");

  // Initialize FlowVR
  std::vector<flowvr::Port*> ports;
  ports.push_back(&pOut);
  flowvr::ModuleAPI* module = flowvr::initModule(ports);
  if (module == NULL) return 1;

  // Helper class to construct primitives update chunks
  ChunkWriter scene;

  // Get IDs for all our primitives and resources
  ID idPrim = module->generateID();
  ID idVB = module->generateID();
  ID idIB = module->generateID();
  ID idVS = module->generateID();
  ID idPS = module->generateID();

  // Create vertex+index buffers for a sphere
  scene.addMeshSphere(idVB, idIB);

  // Load custom shaders
  scene.loadVertexShader(idVS, "shaders/sphere_v.cg");
  scene.loadPixelShader(idPS, "shaders/sphere_p.cg");

  // Create a new primitive
  scene.addPrimitive(idPrim, "Sphere");
```

```
  // Set shaders
  scene.addParamID(idPrim, VSHADER,"",idVS);
  scene.addParamID(idPrim, PSHADER,"",idPS);

  // Set vertex buffers
  scene.addParamID(idPrim, VBUFFER_ID, "position", idVB);
  scene.addParamID(idPrim, VBUFFER_ID, "normal", idVB);
  scene.addParam(idPrim, VBUFFER_NUMDATA, "normal", int(1));

  // Set index buffer
  scene.addParamID(idPrim, IBUFFER_ID, "", idIB);

  // Set shaders parameters
  scene.addParam(idPrim, PARAMVSHADER, "color", Vec4f(1,0.5,1,1));
  Vec3f light(1,3,2); light.normalize();
  scene.addParam(idPrim, PARAMPSHADER, "lightdir", light);

  // Send initial scene
  scene.put(&pOut);

  // Main FlowVR loop. Contains the animations of the scene
  int it=0;
  while (module->wait())
  {
    // Update scene
    scene.rotatePrimitive(idPrim, it);
    // Send message
    scene.put(&pOut);
    ++it;
  }

  module->close();
  return 0;
}
```

Schémas de rendu parallèle 12

Ce chapitre présente la conception du graphe de flux de données reliant les modules *viewers* produisant la description de la scène 3D et les modules *renderers* en charge du rendu, au travers de plusieurs cas typiques de parallélisation de ces modules. La plupart de ces schémas existent déjà dans d'autres systèmes (section 4.1 page 28). Nous nous concentrerons alors sur les éléments nécessaire à FlowVR Render pour les implanter.

12.1 Exemple

Pour illustrer notre propos nous utiliserons l'exemple des applications présentées dans la section 8.4 d'interaction entre des simulations physiques et une reconstruction 3D de l'utilisateur dans un environnement virtuel utilisant la plateforme GrImage. Plusieurs viewers sont utilisés :

Sky : texture cubique contenant l'image du ciel et du paysage lointain.

Terrain : génère un terrain à partir d'une matrice de hauteur et d'une texture. Calcule aussi les mouvements de la caméra lorsqu'il est relié aux boutons d'un Pad.

Model : modèle 3D calculé par la reconstruction multi-caméras.

Octree : cubes constituant l'octree calculé par le module de sculpture.

Fluid : résultats de la simulation de fluide 2D.

Hair : résultats de la simulation de cheveux.

Rigid : ensemble d'objets rigides.

Video : images provenant d'une des caméras envoyées sous forme de texture.

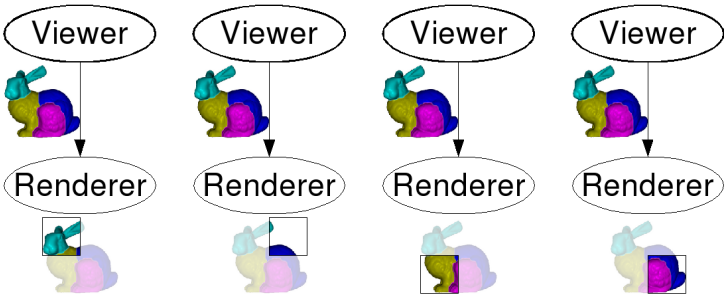


Figure 12.1 *Duplication d'un viewer sur chaque machine. Le rendu se fait localement en fonction de la position de chaque vidéo-projecteur.*

12.2 Duplication

Lorsqu'un module viewer est instancié sur chaque machine et produit la même description de scène il est alors dit *répliqué*. Cela correspond à une stratégie de parallélisation en amont dans le graphe de l'application (diffusion des données d'entrée par exemple). Dans ce cas il suffit de connecter localement chaque viewer au renderer de la machine, qui utilisera la configuration du projecteur local pour calculer une partie de l'image, comme présenté sur la figure 12.1. Cette approche est identique à celle utilisée par Net Juggler (chapitre 5).

Comme les données de la scène ne passent pas par le réseau ce schéma a une très bonne extensibilité en terme de dynamicité de la scène. En revanche la taille totale de la scène est limitée par les capacités de rendu et de mémoire de chaque machine.

Le modèle de FlowVR Render n'est ici pas indispensable, les viewers pourraient directement dialoguer avec le driver de la carte graphique. Toutefois la présence du modèle permet d'ajouter de manière transparente des éléments qui peuvent utiliser voir modifier les données de la scène. Il est par exemple facile d'ajouter un filtre récupérant des informations statistiques sur la scène (nombre de polygones, images par seconde, etc), ou modifiant certains aspects visuels comme l'utilisation de rendu proche du dessin [69, 116, 117], ou séparant les étages d'un bâtiment [125].

Dans les applications GrImage, ce schéma est utilisé pour le viewer **Sky** étant donné qu'il envoi de grandes textures (25 Mo). Comme il n'y a aucune animation cette duplication ne nécessite aucune synchronisation. Le viewer **Video** est aussi répliqué car il génère une quantité de données importante (une caméra en 640×480 à 30 Hz génère 27 Mo/s). Pour transmettre un maximum d'images un mécanisme de compression est utilisé et les viewers **Video** décodent localement les textures (section 13.2.1 page 127).

12.3 Répartition des données vers plusieurs noeuds de rendu (*sort-first*)

L'ensemble des approches présentées dans cette section sont classifiées dans les techniques de rendu parallèle *sort-first* [118], car la diffusion des données se fait *avant* la phase de rendu elle-même.

Ce deuxième cas classique de rendu parallèle concerne l'envoi via le réseau des primitives graphiques. Le module viewer est localisé sur une machine, et les données produites doivent être transmises à toutes les machines de rendu.

En fonction de la taille de ces données et du nombre de machines plusieurs approches sont possibles. Il est par exemple possible d'envoyer directement à chaque machine toutes les données (*flat broadcast*). La bande passante utilisée par la machine émettrice devient vite le goulot d'étranglement, étant donnée que son utilisation dépend du produit de la taille des données par le nombre de machines de rendu.

Une autre approche consiste à envoyer les données de manière récursive par un arbre de diffusion. Chaque machine retransmet les données à n autres machines jusqu'à ce que toutes les machines obtiennent les données (n*-ary broadcast tree*). Avec ce schéma la bande passante utilisée sur chaque machine ne dépend plus du nombre total de machines, en revanche la latence de la transmission augmente avec le logarithme du nombre de machines. Quand la plupart des objets de la scène sont statiques, ce schéma est souvent le plus efficace.

Une optimisation importante des approches précédentes consiste à n'envoyer à chaque machine que les données qui sont visibles dans la partie d'image affichée par la machine (*frustum culling*). Ce filtrage peut s'effectuer à plusieurs niveaux :

- au niveau des messages : un message de description de scène est transmis à une machine si au moins l'un des chunks qu'il contient a une influence sur l'image affichée par cette machine ;
- au niveau des primitives : on supprime des messages tous les chunks qui n'ont pas d'influence sur une machine donnée ;
- à l'intérieur de chaque primitive : on supprime les polygones non visibles, ou les zones non utilisées des textures (niveaux de mipmap détaillés pour les objets lointains par exemple).

Plus on utilise un filtrage fin, plus on diminuera les données transmises sur le réseau. En revanche, la complexité en terme d'exécution comme de développement augmente considérablement. Par exemple, si on reste au niveau des messages ou des chunks on peut se contenter de regarder les "meta-informations" (boite englobante, matrice de transformation) de chaque primitive, alors qu'un filtrage plus fin requiert de décoder toutes les données. De plus, le fait qu'une donnée ne sert pas pour l'image courante ne garantit pas qu'elle ne sera pas nécessaire dans le futur, si l'utilisateur bouge la caméra par exemple. Le filtre doit alors le détecter et envoyer les données, ce qui implique qu'il doit stocker la

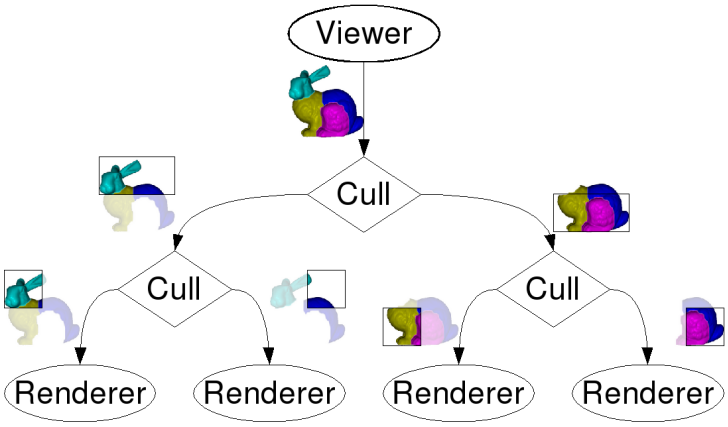


Figure 12.2 *Arbre de filtrage des données de la scène pour les diffuser sur les machines de rendu en fonction de la visibilité des différentes primitives sur chaque machine.*

dernière version de toutes les informations qu'il ne retransmet pas. Enfin, de la même façon que pour les schémas de diffusions, le filtrage peut être fait localement sur la machine émettrice, qui concentre donc tout le surcoût en terme de traitements et bande passante, ou récursivement en utilisant plusieurs niveaux de filtrage. Il est par exemple possible de filtrer les données en groupant les machines par lignes, puis de filtrer à l'intérieur de chacune de ces lignes. Ce schéma correspond à la figure 12.2. Sur cette figure, le filtrage se fait au niveau des primitives (les différentes parties du lapin).

Comme cela sera détaillé pour l'exemple de rendu volumique dans la section 13.1.3 (page 123), l'utilisation de shaders permet de diminuer le trafic nécessaire à la transmission des mises à jour de la scène. En effet, beaucoup d'animations (*keyframes*, *morphing*, *skinning*) ou d'extractions de données (conversions colorimétriques, fonctions de transfert, coordonnées de mapping) peuvent être calculées par les shaders appliqués aux sommets ou aux pixels. En conséquence il n'est souvent nécessaire de mettre à jour que les paramètres (matrices, coefficients, tables) utilisés par les shaders plutôt que tout le modèle 3D.

La plupart des viewers des applications GrImage utilisent un arbre de diffusion sans aucun filtrage des données. **Terrain** et **Rigid** ne mettent à jour que quelques matrices de transformations à chaque itération ; alors que **Model**, **Octree** et **Hair** mettent à jour le modèle 3D lui-même mais cela ne représente que quelques kilo-octets par secondes. Le dernier viewer, **Fluid**, utilise une seule texture. Un filtre de culling permettrait de gagner en performance seulement si le filtrage est effectué à l'intérieur des primitives et des ressources, ce qui n'est pas actuellement implanté de façon suffisamment générique.

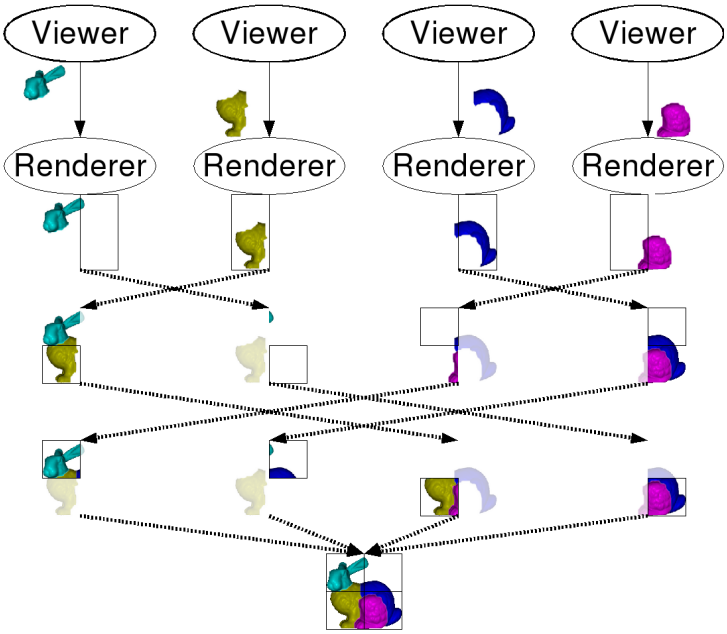


Figure 12.3 *Redistribution des pixels des images (flèches pointillées grasses).*

12.4 Rendu local et recomposition des pixels (*sort-last*)

Le dernier schéma classique de rendu parallèle concerne la recombinaison des pixels des images après le rendu. Cette méthode est surtout utilisée quand on souhaite accélérer le rendu d'une image de faible résolution, étant donné que le surcoût dépend directement de celle-ci. Elle est primordiale pour pouvoir visualiser les très grandes scènes, dont la taille dépasse la quantité de mémoire de chaque machine. Dans ce schéma plusieurs viewers, instanciés sur des machines distinctes, calculent chacun une partie de la scène, qui est ensuite rendue par un renderer local. Une fois ce rendu effectué, les images doivent être recombinées pour obtenir l'image de la scène complète. Plusieurs algorithmes existent, l'un des plus efficace est le *Binary Swap* [108]. C'est un algorithme en plusieurs étapes. A chaque étape les machines échangent deux-à-deux une partie de l'image, jusqu'à ce que chaque machine contienne une zone de l'image finale (figure 12.3). Ces zones peuvent ensuite être envoyées à une certaine machine ou affichées localement.

Cette approche n'est pas directement du ressort de FlowVR Render, étant donné qu'elle est basée sur un transport de pixels et non de primitives graphiques. En revanche, FlowVR Render permet de le faire de manière transparente pour les applications. Il est même possible d'utiliser un schéma de redistribution des primitives graphiques en amont des renderers, de manière similaire à la figure 12.2. Dans ce cas le filtrage des données

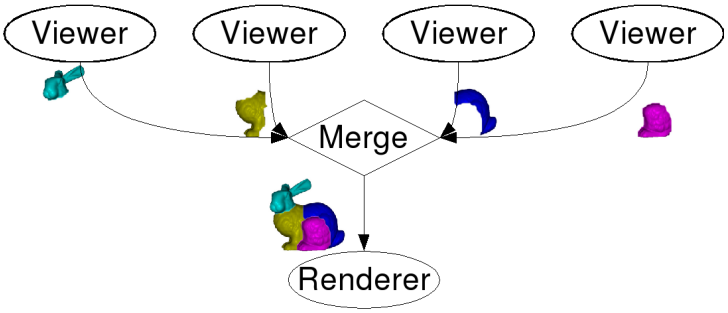


Figure 12.4 *Combiner plusieurs viewers revient à concaténer les messages produits par chacun d'eux.*

ne se fait pas par rapport à la zone visible sur chaque machine, mais plus pour un souci d'équilibrage de charge (i.e. envoyer approximativement le même nombre de polygones à toutes les machines).

12.5 Description parallèle utilisant plusieurs viewers

Combiner plusieurs modules viewers dans une même scène est une fonctionnalité très importante. Elle permet de distribuer les calculs de génération des primitives, par exemple dans le cas d'une extraction d'iso-surface, mais aussi de modulariser la description de la scène. Cette modularité est très utile pour une application basée sur du couplage de code, par exemple pour visualiser deux simulations qui ont chacune leur module de visualisation préexistant. C'est ce qui permet de combiner tous les viewers composant l'environnement virtuel sur GrImage.

Une autre utilisation importante de cette configuration concerne l'intégration de plusieurs schémas de rendu parallèle pour chaque partie de la scène. Par exemple, certains objets générant beaucoup de données peuvent être générés localement en dupliquant les viewers associés, tandis que d'autres viewers plus coûteux en calculs ou nécessitant des données d'entrée volumineuses peuvent être répartis sur d'autres machines avec diffusion par le réseau de leurs résultats. Ensuite, quelle que soit la provenance de chaque flux de données ils sont combinés avant d'être envoyés au renderer. Du fait de l'indépendance des primitives de la scène cette recombinaison est extrêmement simple : il suffit en effet de mettre bout-à-bout les messages de chacun des flux. C'est ce que fait le filtre *Merge* de FlowVR (figure 12.4).

12.6 Asynchronisme

L'utilisation de plusieurs viewers permet de coupler facilement des objets différents dans une même scène, avec toutefois la contrainte que tous les viewers doivent se mettre à jour simultanément. En utilisant un schéma d'échantillonnage avant le *Merge* on peut autoriser des fréquences de mises à jour différentes pour chaque viewer, et entre les viewers et les renderers. Cela permet une meilleure scalabilité et réactivité du rendu du fait du relâchement des synchronisations entre les différents modules. Cette désynchronisation est très simple à implanter du fait de l'aspect unidirectionnel des communications entre les viewers et les renderers. Il peut même être poussé à l'extrême pour exécuter les viewers et les renderers séparément en utilisant un fichier pour stocker les messages.

Quasiment toutes les applications utilisant FlowVR Render se servent de ce mécanisme de désynchronisation (voir par exemple la figure 8.8 indiquant les différentes fréquences de rafraîchissement dans l'application GrImage). Cela permet par exemple de bouger le point de vue de manière fluide, sans attendre la mise à jour du viewer, ceci même dans le cas ou un seul viewer est présent. En revanche certains problèmes d'incohérences apparaissent dans le cas multi-viewers. En effet certains groupes de viewers, comme par exemple ceux implantant une extraction d'iso-surface parallèle, doivent être mis à jour de manière synchrone pour ne pas faire apparaître les frontières.

Un problème similaire se pose au niveau des différents renderers. En effet, les différents projecteurs composant une seule image du point de vue de l'utilisateur doivent souvent se rafraîchir simultanément pour donner l'impression d'un seul grand affichage (*swaplock*). En effet, si une machine est plus rapide que les autres ou à une charge de calcul plus faible elle risque de mettre à jour son affichage constamment avant les autres ce qui risque de casser la cohérence globale. En revanche si deux affichages ne sont pas accolés il est souvent bénéfique de laisser leurs rafraîchissements se faire aussi vite que possible, afin de ne pas les ralentir inutilement. Cela permet d'avoir une console de contrôle distincte de l'environnement immersif sans affecter les performances de celui-ci.

Nous devons donc introduire des contraintes de rafraîchissement synchrone ou non pour les différents viewers et renderers. Heureusement ces politiques de couplages s'implantent facilement dans le graphe de l'application FlowVR en regroupant les synchroniseurs d'échantillonnage des données des différentes connexions concernées.

Ce mécanisme permet à FlowVR Render d'être en quelque sorte l'équivalent pour la 3D du protocole X pour les interfaces 2D. En effet, un serveur X Window permet à plusieurs clients distants de mettre à jour de manière asynchrone un ensemble de primitives (fenêtres) à l'intérieur d'une même surface d'affichage (2D dans ce cas). Le système est basé sur un protocole réseau gérant un certain nombre de fenêtres et de ressources (bitmaps, polices de caractères) en utilisant des identifiants uniques pour tous les clients.

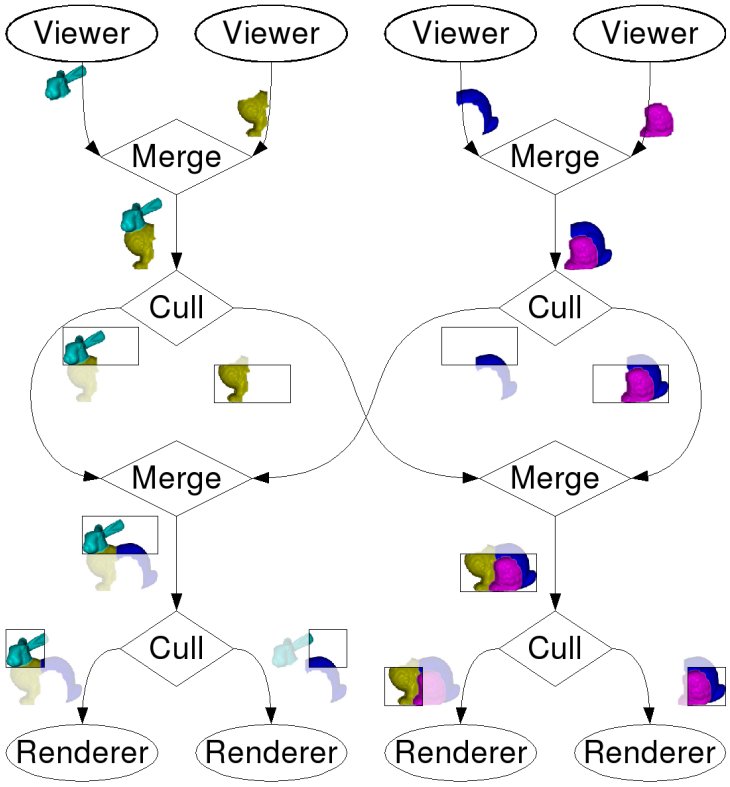


Figure 12.5 *Intercallage d'étapes de fusion et de découpage pour implanter efficacement l'envoi des données de plusieurs viewers vers plusieurs renderers.*

12.7 Passage à l'échelle

Pour une grande application avec de nombreux viewers et renderers, plusieurs schémas sont possibles pour redistribuer les données. En effet, il est possible d'effectuer une première phase de fusion (section 12.5), puis une phase de découpage (section 12.3), ou alors d'inverser l'ordre de ces deux phases. Cependant, une stratégie intermédiaire parfois plus efficace consiste à mélanger ces phases en une suite d'étapes de fusion et de découpages partiels (figure 12.5), similaire au principe du *Binary Swap* [108] utilisé en rendu *sort-last* (section 12.4). A chaque étape les primitives de chaque couple de machines de tri sont fusionnées puis redécoupées par zone. En utilisant N machines de tri, il faut $\log N$ étapes pour compléter l'algorithme. On peut noter que le cas $N = 1$ correspond à l'algorithme centralisé (fusion puis découpage). Ainsi la figure 12.5 utilise 2 machines de tri, et donc une seule étape de l'algorithme, alors qu'un cas plus complexe avec 4 machines de tri et 8 renderers est présenté sur la figure 12.6.

Pour comparer ces stratégies, il est possible d'estimer le volume des transmissions réseau en terme de messages et de bande passante. Soit V le nombre de viewers et R le nombre de renderers. La bande passante utilisée dépend de la finesse du découpage

des primitives. Ainsi des primitives très petites seront en moyenne visible sur un seul renderer, alors qu'au contraire de larges primitives devront être communiquées à tous. Ces deux extrêmes permettent de borner la bande passante utilisée.

En utilisant la stratégie de centralisation des données, à chaque image tous les viewers envoient un message à la machine de tri, qui retransmet ensuite un message à tous les renderers, soit $V + R$ messages au total. En ce qui concerne la bande passante utilisée, chaque primitive est envoyée une fois du viewer vers la machine de tri, puis elle est retransmise vers un seul renderer dans le meilleur cas, et vers tous dans le pire cas, soit un nombre d'envois compris entre 2 et $R + 1$.

La stratégie opposée de découpage puis fusion implique une connexion directe de chaque viewer vers chaque renderer, soit $V \times R$ messages. Les primitives sont envoyées directement vers les renderers concernés, sont entre 1 et R envois.

Enfin la stratégie intermédiaire nécessite $V + N \log N + R$ messages. Pour la bande passante, dans le meilleur cas une primitive a une chance sur deux d'être envoyée sur le réseau lors de chaque étape, soit un total d'envois de $2 + \frac{1}{2} \log N$ (en comptant les envois avant et après le tri), alors que dans le pire cas ce schéma est équivalent à une diffusion de chaque primitive, soit $R + N$ envois.

En conclusion, on remarque que la stratégie inspirée du *Binary Swap* permet d'éviter une centralisation des données ainsi qu'une explosion du nombre de messages envoyés sur le réseau (qui atteint le carré du nombre de modules si les viewers sont directement reliés aux renderers). En configurant le nombre de machines de tri, il est possible de contrôler l'équilibre entre la centralisation du réseau et le surcoût en terme de bande passante. En effet, ce surcoût est compris entre $\log N$ et N envois, ou N et le nombre de machines de tri.

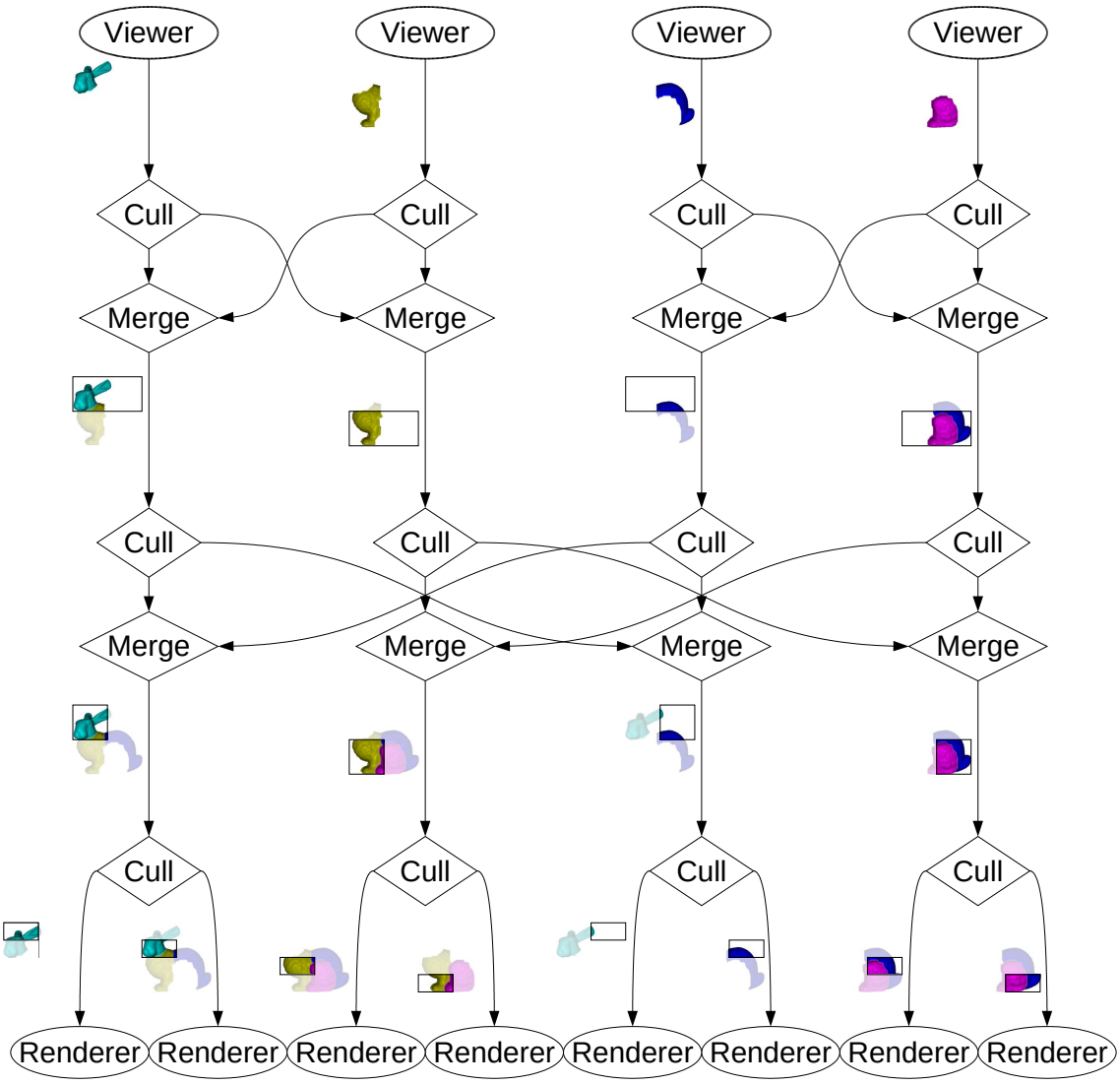


Figure 12.6 *Redistribution entre 4 viewers et 8 renderers utilisant 4 machines de tri et 2 étapes de découpage–fusion.*

Applications 13

13.1 Visualisation scientifique

L'un des premiers domaines d'application des techniques de rendu parallèle concerne la visualisation de gros volumes de données scientifiques, comme des résultats de simulations ou d'instruments de mesures. Nous avons donc expérimenté l'adéquation de FlowVR Render à ce type d'applications, en particulier en comparant avec l'approche existante utilisant Chromium [85]. Ces expérimentations sont aussi présentées dans le premier article publié sur FlowVR Render [12].

13.1.1 Couplage avec VTK

Les applications de visualisation scientifique reposent le plus souvent sur des librairies implantant un ensemble de filtres applicables sur les données pour analyser la partie désirée, ainsi qu'un environnement permettant de relier ces filtres entre-eux et générer l'affichage lui-même (section 2.4). Pour nos expérimentations nous avons choisi VTK [157] car c'est une librairie open-source avec une communauté d'utilisateurs active et offrant les fonctionnalités requises (en particulier le support de la parallélisation du filtrage des données [3]).

Nous avons développé une librairie qui remplace les classes de rendu de VTK de manière transparente. Le design de VTK fait qu'il y a peu de telles classes : une par type de données à afficher entre les images, les modèles 3D (composés de points, de lignes et de triangles) ou les volumes. L'implantation de ces classes fournie avec VTK utilise des commandes OpenGL en mode immédiat ou utilisant des *display lists*. Elle est relative-

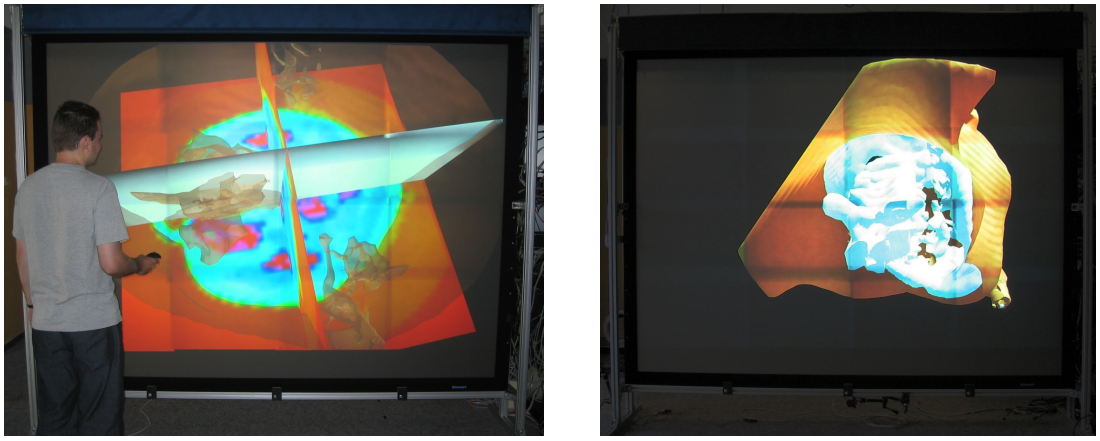


Figure 13.1 *Une des applications d'exemple de VTK affichée sur le mur d'image grâce à FlowVR Render.*

ment complexe pour gérer toutes les combinaisons possibles de données présentes sur les sommets, ainsi que pour supporter les anciens drivers OpenGL. La nouvelle implantation utilisant FlowVR Render est plus simple du fait que bien souvent il suffit d'encapsuler les données brutes dans des ressources FlowVR Render et sélectionner le bon shader. Des applications VTK existantes peuvent alors être affichées sur le mur d'image sans nécessiter de modification (figure 13.1).

13.1.2 Extraction de données

Pour comparer les performances de FlowVR Render par rapport à Chromium nous avons utilisé une application parallèle d'extraction d'iso-surface. Les données utilisées proviennent d'une simulation de fluide 3D de $132 \times 132 \times 66$ cellules pour 900 pas de temps (figure 13.2). L'application extrait et affiche de manière interactive une iso-surface (contenant approximativement 100000 triangles) pour chaque pas de temps. Cette extraction est parallélisée en découpant les données en blocs, chacun assigné à un viewer.

La figure 13.3 présente les mesures de performance de cette application utilisant soit Chromium soit FlowVR Render en fonction de la taille du mur d'image (nombre de renderers) ainsi que du nombre de viewers calculant l'iso-surface. FlowVR Render est plus performant que Chromium et offre un meilleur passage à l'échelle, à la fois en terme de nombre de renderers comme de viewers. FlowVR Render atteint 12 images par secondes avec 16 viewers et 16 renderers pour afficher le résultat sur le mur d'images 4×4 . Les performances de Chromium sont certainement affectées par le surcoût important lié aux opérations de culling et de fusion sur les flux d'opérations OpenGL.

L'utilisation des shaders pour calculer l'éclairage à chaque pixel permet d'augmenter significativement la qualité du rendu visuel. Par rapport au calcul OpenGL classique évalué uniquement aux sommets, les shaders permettent de cacher le découpage en triangle

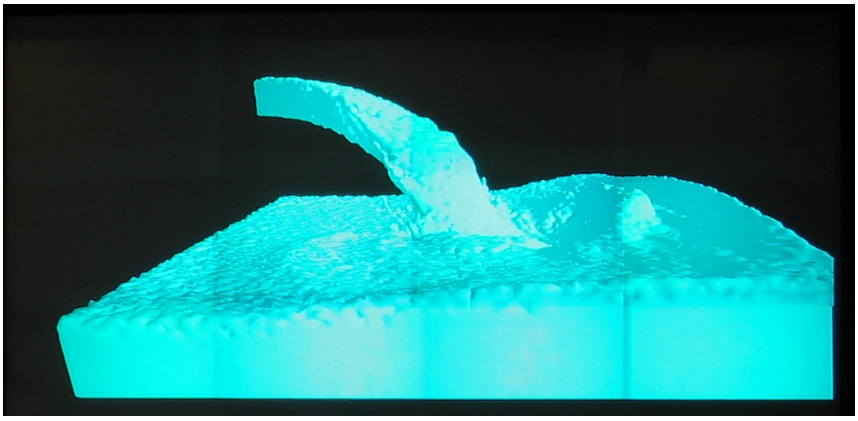


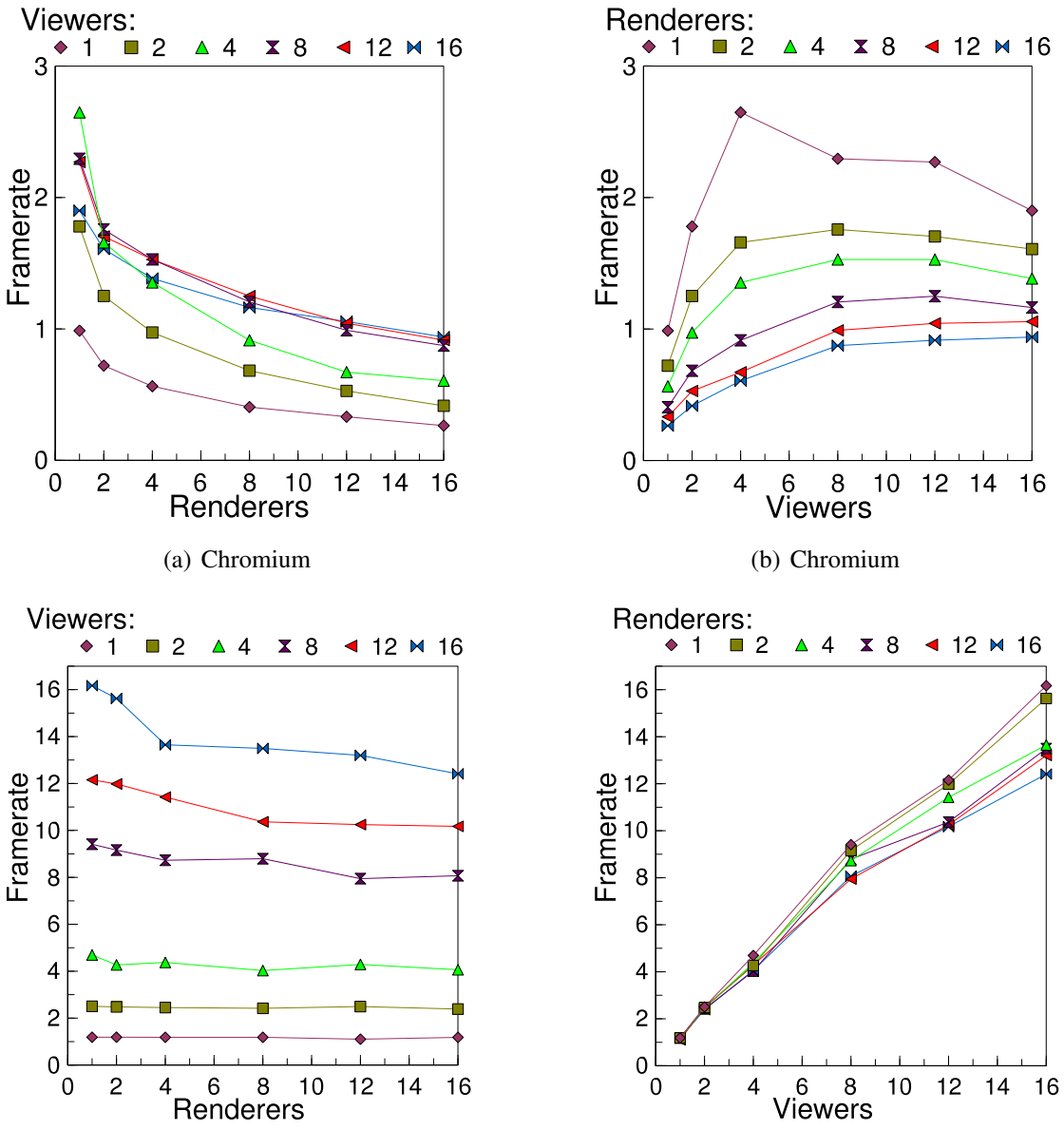
Figure 13.2 *Iso-surface extraite d'un pas de temps d'une simulation de fluide 3D.*

de la surface pour obtenir un rendu beaucoup plus lisse. Une vidéo présentant le résultat est disponible : `http://www-id.imag.fr/~allardj/these/vtk-fluid3d.avi`.

13.1.3 Rendu volumique

Le deuxième test utilise de manière plus avancée les shaders afin de démontrer leurs avantages dans le cadre du rendu parallèle. Nous étudions donc le cas d'une application de rendu volumique en rendu parallèle *sort-first*. Notez que pour le rendu volumique les méthodes *sort-last* se sont en général montrées plus efficaces [181]. Nous voulons ainsi démontrer que l'utilisation de shaders peut augmenter de manière significative les performances des algorithmes *sort-first* pour les raisons suivantes :

- Grâce à la nature massivement parallèle des GPU actuels, les shaders ont accès à beaucoup plus de ressources que les programmes classiques, à la fois en terme de bande passante mémoire et en puissance de calcul [100].
- Un shader peut appliquer les fonctions de transfert permettant de convertir les données volumiques brutes en couleur et opacité finale. Dans le cas ou seul ces fonctions sont modifiées, ceci permet de n'envoyer qu'une seul fois les données volumiques et ensuite uniquement mettre à jour la fonction de transfert lorsque nécessaire. Même dans le cas ou les données sont dynamiques au cours du temps cette approche est intéressante car ces données peuvent être jusqu'à quatre fois plus petites que les couleurs et opacités finales (une valeur par voxel contre quatre pour les couleurs).
- En utilisant des fonctions de transfert *pré-intégrées* [53] et un pas d'échantillonnage adaptatif [151], un shader peu créer une image de très haute qualité tout en nécessitant moins d'accès aux données volumiques, permettant ainsi d'utiliser de plus gros volumes de données.



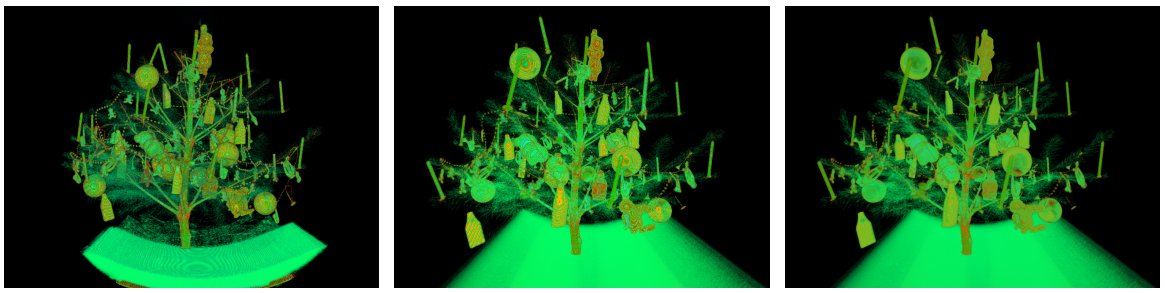
(a) Chromium

(b) Chromium

(c) FlowVR Render

(d) FlowVR Render

Figure 13.3 *Extraction parallèle d'iso-surface avec un rendu sort-first, utilisant Chromium (a)-(b) ou FlowVR Render (c)-(d). Les courbes de gauche présentent l'évolution en augmentant le nombre de renderers, alors qu'à droite sont présentées les courbes en fonction du nombre de viewers.*



(a) Plans texturés 2D VTK (b) Shader de raycasting (c) Raycasting de raycasting avec fonction de transfert pré-intégrée

Figure 13.4 *Rendu volumique d'un sapin de noël utilisant différentes méthodes de calcul (*512 *pas d'échantillonnage par pixel).*

Méthode	Pas d'échantillonage	1 écran 1024×768	Mur d'image 4×4 4096×3072	Mur d'image 4×4 2048×1536	Mur d'image 4×4 1024×768
Plans texturés 2D VTK	512	0.18			
Shader raycasting	512	1.16	2.25	5.40	8.21
Shader raycasting pré-intégré	512	1.10	2.04	4.97	7.70
Shader raycasting pré-intégré	200	2.79	5.14	12.44	19.11

TAB. 13.1 *Performances en nombre d'images par seconde du rendu volumique parallèle avec des données statiques en* $512 \times 512 \times 512$.

Par défaut VTK utilise une approche basée sur un ensemble de plans 2D texturés pour le rendu volumique matériel. Nous avons implanté une nouvelle classe utilisant des shaders. Un pixel-shader est utilisé pour parcourir un rayon au travers du volume (*raycasting*) en accumulant la couleur et l'opacité via un calcul de transparence par atténuation, par additivité ou par valeur maximale. Comme cette accumulation est effectuée dans des registres temporaires du shader et non via le buffer d'image, il garde une précision en 32-bits et économise la bande passante normalement nécessaire pour écrire et relire les valeurs du buffer d'image utilisée par les approches traditionnelles multi-passes.

Pour implanter une fonction de transfert pré-intégrée nous utilisons une texture 2D qui, étant donnée la valeur de densité précédente et courante sur le rayon, stocke la couleur et l'opacité obtenue en intégrant toutes les densités intermédiaires dans la fonction de transfert originale. Ce calcul augmente sensiblement la qualité visuelle du rendu, surtout pour le cas des fonctions de transfert utilisant des hautes fréquences, et permet aussi d'utiliser de plus grands pas d'échantillonnage pour les données volumiques qui ne comportent pas de variation abrupte.

Comme cette application n'est limitée que par le *fill-rate* de la carte graphique (nombre de pixels que la carte graphique peut traiter en une seconde), nous avons utilisé pour l'envoi des données un simple schéma de diffusion où tout est envoyé à tous les renderers.

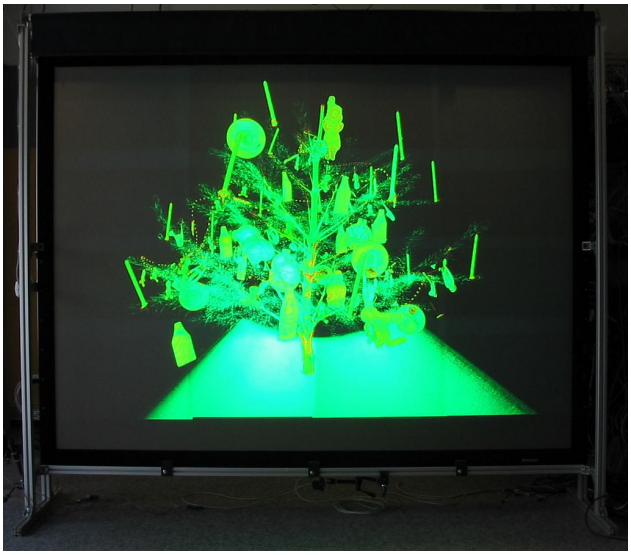


Figure 13.5 *Rendu volumique sur le mur d'image.*

La figure 13.4 présente les résultats utilisant la méthode originale de VTK ainsi que la nouvelle approche utilisant FlowVR Render et notre shader de tracé de rayon, optionnellement utilisant une fonction de transfert pré-intégrée. Les données sont un volume de $512 \times 512 \times 512$ contenant un sapin de noël [94]. La table 13.1 rassemble les mesures de performance de cette application. Le rendu parallèle sur le mur d'image n'introduit pas de surcoût significatif par rapport au rendu sur une seule machine comme les données transférées sont relativement petites (la position de la caméra ainsi que la fonction de transfert). Nous obtenons même de meilleures performances sur le mur d'image du fait d'une plus grande cohérence entre les pixels voisins qui favorise les caches texture à l'intérieur de la carte graphique.

On peut aussi noter qu'il est possible d'améliorer les performances pendant les interactions (pendant que la caméra se déplace par exemple), en réduisant la résolution de rendu en plus de la fréquence d'échantillonnage. Ceci permet d'obtenir des mouvements tout en gardant une qualité raisonnable.

13.2 Autres applications

L'application GrImage a été la raison initiale du développement de FlowVR Render. Les applications utilisant VTK décrites ci-dessus ont permis de le valider en comparant avec l'outil existant prédominant pour le rendu distribué. Depuis, des applications développées par d'autres personnes ont utilisé l'architecture de FlowVR Render. Cette section en présente quelques unes.

Figure 13.6 *Calcul de la texture du modèle reconstruit en utilisant un shader pour recombiner les images des caméras en calculant les coefficients appropriés (représentés par des couleurs sur le rendu en haut à gauche).*

13.2.1 Texturage du modèle reconstruit

Cette application est une collaboration avec Florian Geffray, Marc Lapierre et Clément Ménier.

La reconstruction 3D multi-caméras décrite dans la section 8.3 permet d'obtenir en temps réel un modèle géométrique de l'utilisateur. Toutefois il manque les informations de couleur et de texture pour obtenir une représentation réaliste. Il est possible de reprojeter les images des caméras sur le modèle 3D pour obtenir cette information. Plusieurs problèmes se posent :

- Les flux vidéos doivent être transmis aux machines de rendu. Comme la bande passante réseau et d'envoi vers la carte graphique n'est pas suffisante il faut utiliser des algorithmes de sélection d'une partie des caméras, et de compression des données.
- Certains objets cachent des objets plus loin sur certaines caméras (comme la main par rapport au corps), ce qui introduit des erreurs si on projette l'image de la main sur le corps.
- Une fois les flux à utiliser déterminés, il faut les recombiner en calculant des coefficients pour utiliser la meilleure image tout en cachant les transitions.

Nous avons implanté une première approche utilisant FlowVR Render. Le viewer existant **Video** a été adapté pour recevoir les données sous forme compressée. La compression utilisée est pour l'instant très simple et consiste à n'envoyer que les couleurs des pixels de l'objet et non du fond de l'image, ce qui supprime environ 75 % des données. Les flux d'images compressées sont ensuite diffusés à toutes les machines de rendu. Sur chacune de ces machines un viewer décompresse les images et les envoie sous forme de texture. Ensuite le viewer **Model** a été adapté pour utiliser un nouveau shader utilisant ces flux de textures ainsi que les matrices de calibration des caméras pour calculer à chaque pixel

la projection du point sur chaque image et les recombiner en fonction de l'angle entre le vecteur normal à la surface et la direction de la caméra. Ce coefficient permet de n'utiliser que les caméras qui font face à la surface. Le résultat est visible sur la figure 13.6.

Avec 6 caméras en 780×580 et le mur de 16 vidéo-projecteurs ceci permet d'envoyer quelques images par seconde. Pour augmenter ces performances et éviter de saturer le réseau pour le reste de l'application nous avons utilisé un second réseau gigabit pour toutes les communications des images compressées. Cela permet d'envoyer environ 10 images par seconde sans affecter les autres modules.

Cette première implantation a servi de prototype, plusieurs améliorations sont depuis en développement. Pour augmenter le nombre de caméras ainsi que la fréquence de rafraîchissement un module de sélection des flux les plus utiles en fonction de la position de la caméra va être ajouté. Bien que le résultat visuel est satisfaisant, les occlusions ne sont pour l'instant pas gérées. Pour le faire il faudrait utiliser un shader plus avancé, utilisant des textures de distances (*depth maps*) ou des volumes d'ombres calculés avec le *stencil buffer*.

13.2.2 Vidéo haute-définition sur mur d'image

Cette application est une collaboration avec Clément Ménier.

Plutôt qu'envoyer un ensemble de flux vidéos, un autre problème concerne l'affichage de vidéos haute-résolution. La haute résolution du mur d'image permet d'afficher des images très précises, ce qui demande beaucoup de puissance de calcul et de bande passante dans le cas d'une vidéo. Les étapes nécessaires sont :

- lecture du fichier ;
- décodage de la vidéo ;
- conversion des images du format YUV (luminance et chrominance) vers RGB ;
- affichage des images.

Clément a converti le logiciel open-source de lecture de vidéo MPlayer[1] en viewer FlowVR Render. Utiliser FlowVR Render pour l'affichage permet de bénéficier des corrections liées aux vidéo-projecteurs (matrices de calibration, masques de blending) et d'utiliser plusieurs schémas de distributions. Nous avons testé deux approches : un seul viewer décode la vidéo et le résultat est diffusé aux différents renderers (rendu *sort-first*), ou alors une *réplication* du viewer décodant la vidéo sur chacune des machines de rendu. La première stratégie est bien adaptée à la lecture de vidéos de taille similaire au format DVD (720×576), en revanche la bande passante du réseau n'est pas suffisante pour des vidéos plus haute résolution. La réplication des viewers permet de supprimer ce goulot d'étranglement. Le facteur limitant devient alors la puissance de calcul nécessaire. Pour décharger le processeur des machines nous avons utilisé un shader qui se charge du décodage des couleurs YUV vers RGB. Du fait de l'encodage de la plupart des vidéos en

[1] http://www.mplayerhq.hu/

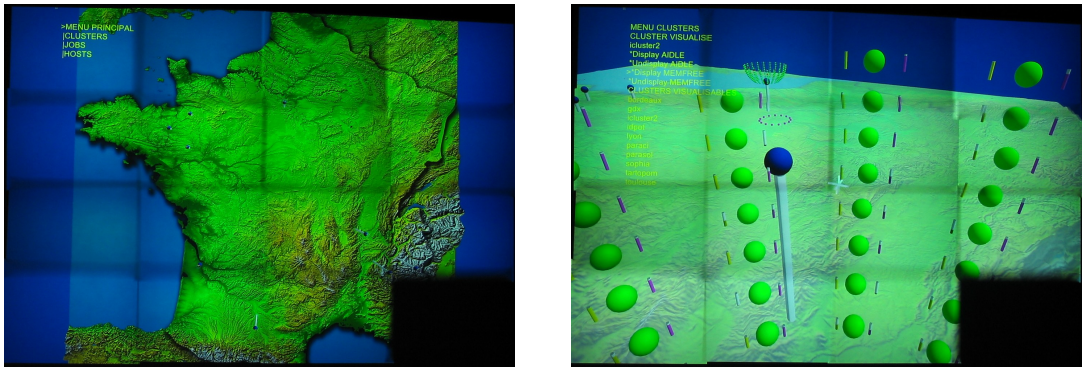


Figure 13.7 *Visualisation de l'état de la grille Grid 5000 sur le mur d'image.*

plans séparés avec un sous-échantillonnage des canaux de couleur, cela permet aussi de diminuer la taille des données à transmettre à la carte graphique (12 bits par pixels au lieu de 24). Cette architecture permet de visualiser sur le mur d'image des vidéos au format HDTV (1920×1080 à 60 Hz).

Cette application offre une solution simple à un problème concret mais reste assez limitée. En effet, toutes les données sont décodées sur chaque machine et envoyées sur chaque carte graphique. Pour atteindre la résolution du mur d'image (4000×3000) il faudrait paralléliser le décodage de la vidéo.

13.2.3 Visualisation interactive de l'état d'une grille

Cette application est développée dans le cadre du stage d'Oscar Ayoun et Julien Bouvier en collaboration avec Olivier Richard.

Le but est de visualiser en 3D sur un ordinateur portable ou un environnement immersif l'état de la grille Grid 5000 (statistiques d'utilisation des ressources, tâches lancées) pour démontrer le déploiement des applications en temps réel. Pour cela un ensemble de modules FlowVR est déployé sur la grille et recueille les informations nécessaires au niveau des machines, des switchs réseaux ou des logiciels de monitoring et d'allocation des tâches (*Ganglia*[2] et *OAR*[3]). Ensuite des modules viewers placés sur la grappe de visualisation les convertissent en représentations visuelles. D'autres viewers ajoutent des informations de contextes (carte de France). L'utilisateur peut alors sélectionner les données visibles dans la scène 3D en contrôlant par une GUI ou des boutons d'un Pad des petits filtres qui vont modifier ou bloquer les messages des viewers concernés.

[2] http://ganglia.sf.net/
[3] http://oar.imag.fr/

Bilan 14

Nous avons présenté un outil de rendu parallèle appelé FlowVR Render proposant un protocole de description basé sur des shaders et utilisant un ensemble de groupes de polygones comme primitives. Par rapport aux approches standards basées sur des commandes OpenGL, ce protocole offre les avantages suivant :

- Des shaders sont utilisés pour spécifier l'apparence visuelle des objets graphiques. Ils requièrent seulement quelques paramètres et non la complexité de toute la machine à état du pipeline classique OpenGL. Cela permet de définir un protocole plus simple qui n'a pas à gérer les changements d'états.
- Les shaders permettent d'utiliser toute les fonctionnalités des dernières générations de cartes graphiques.
- Les primitives ne sont pas ordonnées et sont indépendantes (du fait de l'absence d'état commun), ce qui rend les opérations de filtrages plus simples et efficaces et permet d'optimiser la boucle de rendu.
- Des informations de haut niveau comme des volumes englobants ou les changements entre chaque image permettent de réduire le coût des traitements et des communications.

Le fait d'utiliser une description de la scène particulière oblige à modifier les applications existantes. Toutefois cet inconvénient peut être facilement surmonté, par exemple dans le cas des applications de visualisation qui n'utilise que quelques primitives graphiques différentes. L'adaptation du code de rendu d'un outil comme VTK, l'un des outils de visualisation les plus utilisés, permet de supporter de manière transparente toutes les applications existantes l'utilisant.

Cet outil n'est pas limité aux applications de visualisation scientifique mais trouve des utilisations pour d'autres applications. La description modulaire de la scène permet

de construire de façon incrémentale les éléments d'un monde virtuel et de les distribuer de manière appropriée sur la grappe de visualisation. L'utilisation de shaders permet de répartir la charge de travail entre les CPU et les GPU. Enfin, l'adjonction de petits filtres entre les viewers et les renderers permet d'implanter facilement des contrôles permettant à l'utilisateur de sélectionner les données à afficher.

Parmi les améliorations intéressantes à développer par la suite il y a la conception d'un mécanisme standardisé d'interaction avec les différents éléments de la scène 3D. Un peu comme le *Window Manager* gère les fenêtres 2D des autres clients d'un serveur *X Window* (section 12.6), il manque un viewer particulier et un jeu de filtre permettant de répartir intuitivement l'espace virtuel et les actions de l'utilisateur entre les différents viewers.

Pour supporter de plus grands environnements virtuels, une autre amélioration importante consiste en l'utilisation de plusieurs niveaux de détails au niveau des primitives, permettant lors du rendu de répartir les ressources en fonction de la visibilité des objets. Ici il est possible d'étendre le protocole utilisé par FlowVR Render, peut être en intégrant des outils existants comme Magellan [111].

IV

Couplage de simulations distribuées interactives

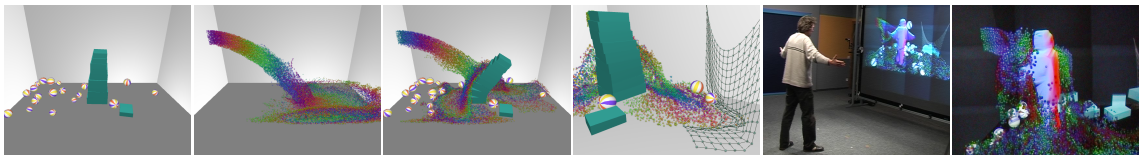
L'aboutissement des travaux sur le couplage de codes parallèles interactifs (partie II) et la modularisation de la visualisation (partie III) permettent d'envisager de nouvelles applications. C'est l'objet de cette dernière partie qui présente une expérimentation en ce sens. Bien que ce travail soit très exploratoire et ne soit pas complètement abouti, il permet de mieux appréhender les nouvelles perspectives offertes par les travaux précédents.

Cette partie s'attache donc à la conception d'une architecture de couplage de plusieurs simulations, en gérant les interactions entre chacune de ces simulations ainsi qu'avec l'utilisateur. Un prototype d'application combine ainsi des simulations d'objets rigides, de fluide 3D et d'objets déformables. Cette application est la plus ambitieuse construite avec FlowVR et FlowVR Render jusqu'à maintenant.

Ce travail a été publié lors de la conférence IEEE VR 2006 [13].

Introduction 15

"THE DIFFERENCE BETWEEN TRUTH AND FICTION :
FICTION HAS TO MAKE SENSE."
Mark Twain



(a) (b) (c) (d) (e) (f)

Figure 15.1 *Coupler plusieurs simulations comme des objets rigides (a) et un fluide (b) permet la construction d'environnements complexes (c)-(d). Plusieurs stratégies de distribution et de parallélisation peuvent ensuite être appliquées pour offrir des interactions en temps réel avec l'utilisateur (e)-(f).*

Plus un environnement virtuel parait complexe, plus l'immersion de l'utilisateur y sera de qualité. En effet, un environnement statique ou presque vide contraste trop avec la dynamicité et la richesse de la réalité. Cette complexité peut être obtenue au niveau du rendu visuel (textures, jeux de lumières) ainsi qu'au niveau du mouvement des objets et de leurs interactions (non pénétration, impression de gravité, réactions aux mouvements de l'utilisateur). L'objectif de cette expérimentation est de définir une architecture de haut niveau permettant de coupler de multiples simulations physiques à l'intérieur d'un tel environnement pour le rendre le plus complexe et interactif possible.

15.1 Simulations physiques

Notre approche se base sur des travaux antérieurs de simulations inspirées des lois physiques, et de distribution et parallélisation de simulations.

15.1.1 Simulation pour les applications graphiques interactives

De nombreux travaux existent autour des simulations pour l'animation graphique, à la fois pour le rendu hors ligne et les interactions en temps-réel. L'objectif est de produire des effets visuels convaincants tout en minimisant le coût des calculs, souvent en sacrifiant la précision physique. Plusieurs algorithmes existent pour simuler le comportement d'un type particulier d'objet. Des travaux récents se sont attachés aux simulations d'objets solides [77], de fluides [166, 60, 54], de tissus [30], et d'objets déformables [47].

Implanter les interactions entre des types d'objets différents, comme entre des fluides et des objets rigides par exemple, est un problème difficile. Dans certains cas cette difficulté peut être évitée en ne considérant qu'une interaction à sens unique, c'est-à-dire où un objet a une influence sur l'autre mais pas l'inverse. Un cas type de telles interactions concerne les objets contrôlés de manière externe (suivant une animation prédéterminée ou reliés aux actions de l'utilisateur). Ces interactions à sens unique permettent aussi de traiter le cas où une différence d'échelle importante existe entre les deux objets, au niveau du poids par exemple, permettant de négliger l'influence d'un des objets sur l'autre. D'autres approximations peuvent aussi être utilisées, comme le calcul des mouvements d'objets immergés dans un fluide en utilisant la vitesse de celui-ci au niveau du centre de masse de l'objet (feuilles à la surface de l'eau, algues suivant l'orientation des courants).

Les interactions à sens unique les plus couramment traitées sont certainement la collision d'un objet sur un obstacle rigide. Les simulations de fluide les expriment principalement par des conditions aux bords adéquates [54] mais peuvent aussi traiter des obstacles plus complexes [83]. Les simulations de tissus prennent en compte les collisions avec d'autres objets [30] (le corps en particulier).

Pour traiter des interactions dans les deux sens, combiner deux algorithmes simulant chacun un sens d'interaction n'est pas suffisant pour obtenir un résultat convainquant. La loi de conservation de l'énergie impose que la force de l'interaction soit égale et opposée sur les objets concernés, cette contrainte ne peut être résolue par deux méthodes de calculs indépendantes pour chaque sens. Des travaux récents permettent de traiter les interactions dans les deux sens entre les fluides et les objets déformables [66], ou entre les fluides et les objets rigides [34]. Un problème important concerne la différence de représentation de chaque type d'objet. Un objet rigide utilise une représentation *Lagrangienne* (la simulation considère les caractéristiques d'éléments mobiles), alors que la plupart des fluides sont représentés par un découpage fixe *Eulerien* de l'espace. Une façon de résoudre leurs interactions et d'utiliser un même modèle pour les deux objets. Cette approche est celle

adoptée par les travaux cités, utilisant soit une simulation de fluide Lagrangienne [122], soit insérant les objets rigides dans la grille du fluide en ajoutant aux cellules concernées une contrainte de rigidité [34].

15.1.2 Simulations parallèles et distribuées

Les simulations interactives ont déjà été traitées dans la section 2.3 (page 10). Cependant, notre objectif ici étant de coupler des simulations plus ambitieuses, nous allons nous intéresser plus en détail à deux classes de simulations proches : les simulations scientifiques à grande échelle, et les simulations distribuées sur des sites distants (environnements collaboratifs, jeux en ligne).

Les simulations scientifiques à grande échelle reposent principalement sur le calcul parallèle pour exploiter les systèmes hautes performances. Contrairement aux simulations pour les animations graphiques, l'objectif principal est la précision physique. Pour maîtriser cette précision la simulation est souvent structurée en codes monolithiques parfois utilisant un couplage très étroit entre plusieurs parties, comme par exemple la gestion des échanges entre l'océan et l'atmosphère via un échange synchrone de paramètres précis de chaque côté au niveau des bords des simulations. Ce type d'application requiert beaucoup de temps pour effectuer le calcul [141]. Peu de travaux réutilisent ce type d'approche pour paralléliser les simulations pour les animations graphiques et pouvoir traiter des données plus importantes ou réduire le temps de calcul [99].

Les environnements virtuels en réseau [161], comme les simulations de champs de bataille ou les jeux en ligne, doivent échanger les données partagées selon des critères de cohérence déterminant la bande passante et la latence des communications [187]. Les protocoles garantissant une forte cohérence entre les données répliquées sur les différents sites requièrent beaucoup d'échanges ainsi qu'une synchronisation forte entre les sites. Une réduction de la cohérence entre les copies permet d'augmenter les performances, en considérant en particulier que les interactions de l'utilisateur sont localisées. Par exemple, la position d'objets lointains peut être transmise moins fréquemment, l'utilisateur ne pouvant pas distinguer les petits mouvements. Des techniques d'interpolation et de partitionnement permettent de découpler les mises à jour des différentes machines, permettant à la simulation de ne pas être limitée par la machine la plus lente. La prédiction des mouvements par *dead-reckoning* est très étudiée et utilisée [106, 33].

Architecture de couplage 16

De la même façon que FlowVR Render (partie III page 93) est conçu pour modulariser et étendre la tâche de visualisation, cette expérimentation étudie un découpage des calculs dans l'environnement virtuel en couplant plusieurs simulations. Cependant, bien que la visualisation peut être modélisée comme une chaîne de traitement (du calcul de la géométrie à l'affichage des pixels), les simulations utilisent un schéma plus complexe. Ainsi, chaque simulation peut être vue comme une boucle de calcul, où le résultat de l'itération précédente est utilisé pour la suite. Entre les différentes simulations des interconnexions uni ou bi-directionnelles sont nécessaires en fonction du type d'interaction. Ce chapitre propose une architecture de découpage et de couplage de ces simulations, définissant le rôle des composants nécessaires, ainsi que le format des données échangées.

16.1 Conception générale

Pour gérer toutes les interactions, les simulations doivent s'insérer dans une vision commune de l'environnement virtuel. Ainsi nous définissons le monde comme étant composé d'*objets*. Chaque objet possède ses propres paramètres (type, position, volume englobant) et peut faire référence à certaines *resources* partagées (mesh 3D, champs de distance signée, ...). Cette représentation est très similaire à celle utilisée par FlowVR Render (chapitre 11 page 99), à la différence qu'elle contient des objets plus génériques en lieu des primitives graphiques. En fait, on pourrait définir le modèle de FlowVR Render comme un cas particulier de ce nouveau modèle, ou tous les objets sont de type primitive graphique.

Deux classes principales de modules se répartissent les calculs nécessaires. Les *animators* sont responsables des objets de la scène et calculent leurs mouvements en fonc-

tion des forces qui leurs sont appliquées. Ces forces sont calculées par les *interactors* qui traitent de l'interaction entre types d'objets spécifiques, en utilisant les informations sur ces objets transmissent par les animators.

Les calculs sont répartis en fonction du type d'objet. Ainsi chaque animator est en général dédié à un certain type d'objet, et un interactor gère uniquement certaines interactions sur les types qu'il sait manipuler (collisions entre solides par exemple). Une deuxième répartition peut être nécessaire pour paralléliser les calculs. Ainsi, plusieurs animators peuvent se répartir les objets d'un même type en traitant chacun un groupe. Ce découpage peut suivre les objets (i.e. un objet appartient toujours au même groupe quel que soit ses mouvements), ou bien être lié à un découpage spatial. Les communications entre les animators et les interactors doivent alors utiliser des filtres adaptés pour transmettre uniquement les données concernant chaque module.

16.2 Objets et animators

La scène virtuelle est composée d'*objets* répartis entre plusieurs modules *animators*. Chaque objet comporte un identifiant *id* unique et possède une liste de *paramètres*. Ces paramètres définissent toutes les informations requises par l'application. Ceci peut inclure des données graphiques, sonores, physiques, etc. Un objet appartient à un certain *type*. Tout objet d'un même type doit définir les paramètres correspondants. Par exemple, un objet rigide dans le cadre d'une simulation physique possède une position, un volume englobant, une masse, un tenseur inertiel, une forme géométrique pouvant utiliser un mesh 3D et un champ de distance signée, ainsi que pour le rendu des shaders, textures, et potentiellement un mesh 3D plus détaillé.

La description de chaque objet est auto-contenue. Ainsi la spécification d'un paramètre d'un certain objet ne dépend d'aucun autre objet. En particulier, la position est toujours définie dans un système de coordonnées unique, sans notion de hiérarchie de systèmes de coordonnées comme dans un graphe de scène. Cela rend les mises à jour d'un objet indépendantes des autres objets, et permet de distribuer les objets entre plusieurs animators de manière arbitraire. Les opérations de découpage des objets de la scène en sont aussi plus efficaces, permettant de travailler simplement avec des sous-ensembles des objets. Ces points sont fondamentaux pour obtenir de bonnes performances dans un contexte distribué.

Un module animator possède un port de sortie *object*, lui permettant de communiquer au reste de l'application les mises à jour des objets dont il a la charge. Le format des données transmises est décrit dans la section 16.4. Plusieurs animators sont utilisés dans l'application, permettant de gérer chaque type d'objet, et potentiellement de répartir les calculs nécessaires en découpant les objets en sous-groupes.

Le calcul des mouvements de chaque objet dépend théoriquement des interactions avec tous les autres objets présents. Ces interactions imposent au calcul lié à un certain

type d'objet de prendre en compte tous les autres types d'objets présents dans la scène. Afin de permettre l'ajout d'un nouveau type d'objet de manière modulaire et d'éviter de rendre chaque animator dépendant de tous les autres animators de la scène, ces interactions ne sont pas calculées directement par les animators. Elles sont implantées par d'autres modules, les *interactors*, en combinant les informations produites par plusieurs animators. La section 16.3 présente leur fonctionnement. Le résultat de ce calcul est envoyé sous forme d'*évènements* d'interaction aux animators concernés. En fonction du modèle utilisé par la simulation, ces évènements peuvent prendre la forme de forces appliquées, d'impulsions, de contraintes à respecter, etc. Les animators responsables d'objets interactifs, c'est-à-dire la plupart à l'exception des objets suivant une animation précalculée ou contrôlée par des périphériques d'entrée, ont donc un port d'entrée *event* leur permettant de recevoir ces informations.

16.3 Interactors et filtrage des données

Les modules d'interactions, aussi appelés *interactors*, sont responsables des interactions entre les objets de la scène. Ils reçoivent la description de ces objets par un port d'entrée *object*, détectent les interactions puis envoient les évènements associés sur un port de sortie *event*. Différents animators peuvent être conçus pour gérer chaque interaction possible. Par exemple, un animator peut implanter la détection des collisions entre objets rigides alors qu'un autre peut être dédié aux collisions faisant intervenir un objet masses-ressorts.

Conceptuellement, un interactor peut nécessiter les données produites par tous les animators présents, et en retour les évènements produits peuvent aussi tous les concerner. Ainsi, relier tous les animators avec tous les interactors à la fois pour les ports *object* et *event* garantirait que tous les modules obtiennent les informations requises. Ce schéma n'est en revanche pas efficace et écroulerait les performances au fur et à mesure de l'ajout de nouveaux modules. Ainsi les liaisons entre les modules font intervenir des filtres pour router de manière intelligente les communications afin de na pas transmettre d'information inutile. Une première optimisation consiste à supprimer de manière statique toutes les connexions qui sont sémantiquement inutiles. Ainsi, la plupart des animators produisent un type d'objet particulier, il est alors inutile de le relier aux interactors ne supportant pas ce type d'objet. Un deuxième filtrage plus dynamique peut se faire selon des critères de localité (pour les modules parallélisés par découpage spatial), ou de superposition de volumes englobants (il n'est pas nécessaire de transmettre les informations d'un objet dont la boite englobante n'intersecte aucune autre boite englobante des autres objets de la scène).

16.4 Protocole de communication

Les animators décrivent la scène en utilisant un protocole identique à FlowVR Render (section 11.3 page 105). Ainsi, les changements sont décrits par une liste de *chunks*. La différence principale étant que les éléments décrits par les animators sont des objets et non des primitives, et des évènements pour le cas des interactors. De même que les primitives, les objets font appel à des ressources partagées pour les données conséquentes.

16.4.1 Objects

Les chunks de spécification des objets sont similaires à ceux des primitives graphiques, à la différence importante qu'un *type* particulier est spécifié lors de la création d'une nouvelle primitive, et les paramètres utilisables sont différents en fonction de ce type. Toutefois, tous les objets définissent les paramètres de position, boite englobante et forme géométrique (qui peut être représentée par une UN mesh, un découpage régulier en voxels ou en octree). Actuellement les types d'objets suivants sont définis (figure 16.1) :

Rigid : objet rigide définissant une masse, un tenseur inertiel, une vitesse linéaire et angulaire, ainsi que la force et torsion totale appliquée.

Fluid : fluide 3D comportant un champ de vitesse et de densité. Il peut posséder une représentation géométrique sous forme de discrétisation volumique et de mesh de la surface.

MassSpring : ensemble masses / ressorts simulant un tissu ou un objet déformable. Ces paramètres sont la masse de chaque particule, la raideur des ressorts les reliant, ainsi que leur élongation minimale et maximale. La représentation géométrique d'un tel objet est formée des vertice correspondant aux masses et des segments correspondant aux ressorts.

D'autres objets peuvent être intégrés très simplement en ajoutant une nouvelle classe.

16.4.2 Evènements

Les interactions étant généralement ponctuelles, les données les concernant sont probablement différentes entre chaque itération. De ce fait, plutôt qu'envoyer des chunks de mise à jour, il est plus simple d'envoyer les évènements eux-mêmes. En plus du type de l'évènement, chaque chunk contient les identifiants des objets concernés (2 en général), ainsi qu'une durée d'application pour le cas ou l'évènement reste actif pendant un certain temps. Ensuite les données restantes dépendent du type, et correspondent aux paramètres présentés sur la figure 16.2.

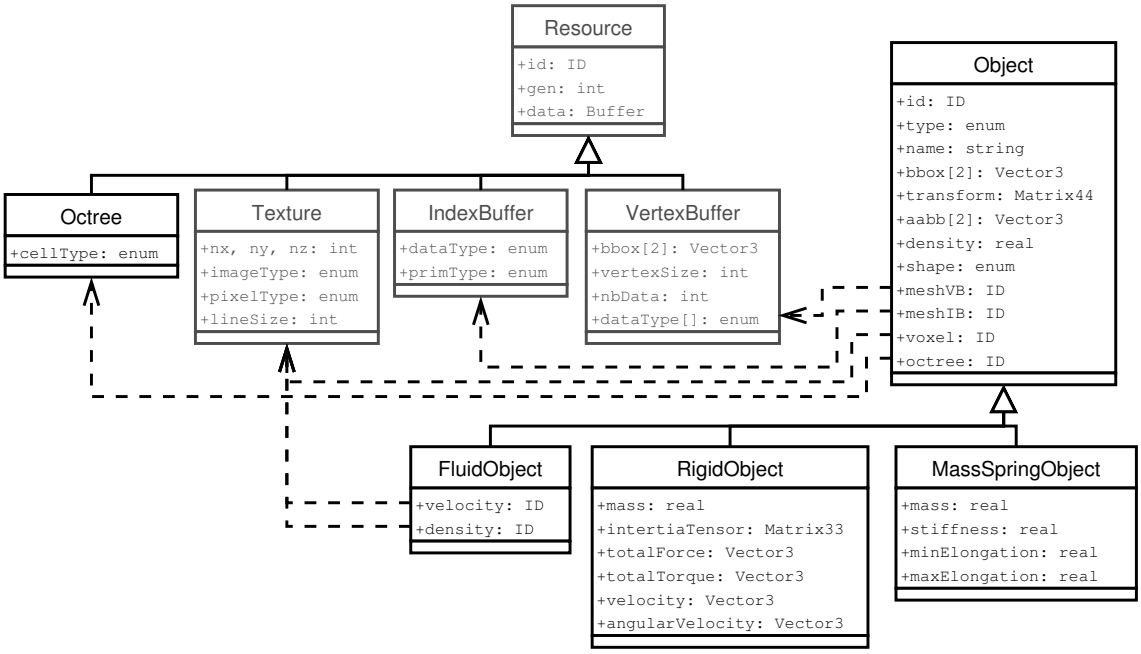


Figure 16.1 *Schéma UML des objets utilisés dans la description de la scène. Les classes grisées sont reprisent de FlowVR Render.*

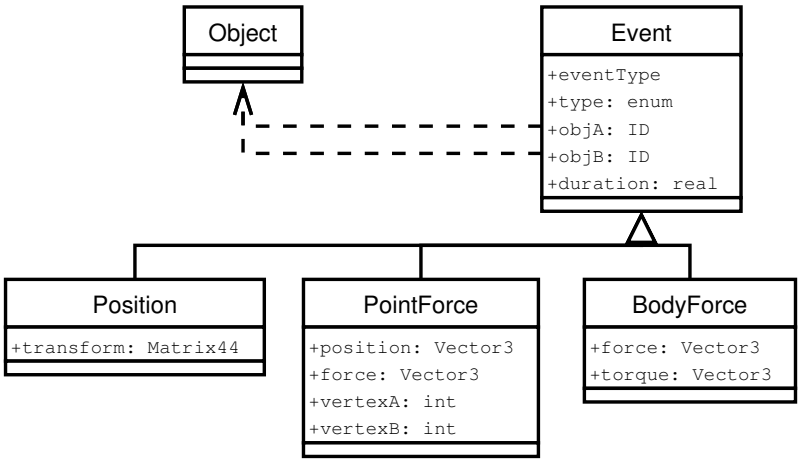


Figure 16.2 *Schéma UML des classes d'évènements.*

16.5 Construction de l'application

Les différents composants de notre architecture étant décrits dans les sections précédentes, nous allons maintenant présenter la construction de l'application complète en prenant un cas d'exemple.

16.5.1 Exemple

Pour clarifier la présentation de la construction d'application utilisant notre architecture nous allons utiliser le cas de notre application prototype : le couplage d'objets rigides, d'un fluide et d'un filet masses-ressorts. L'animation doit prendre en compte les contraintes liées aux contacts qui se présentent entre tout type d'objets.

L'implantation des calculs se base sur des algorithmes existants. Ainsi les collisions entre solides se basent sur Guendelman et al. [77] et les interactions avec le fluide utilisent la méthode *rigid fluid* [34]. Deux modifications doivent toutefois y être apportées :

- Du fait du découpage des différents objets, l'algorithme calcule les forces à appliquer aux solides immergés dans le fluide, plutôt qu'en mettant à jour directement leur vitesse.
- Le filet masses-ressorts est considéré comme un ensemble de petites particules. Cette simplification ne permet pas de gérer sa perméabilité par rapport au fluide.

Le mouvement du filet masses-ressorts utilise un algorithme très simple basé sur une intégration explicite d'Euler des forces et vitesses, ainsi que des ressorts avec une raideur constante et une élongation bornée.

16.5.2 Application séquentielle synchrone

Commençons par les objets rigides, contrôlés par un animateur et dont les interactions sont calculées par un interactor détectant les collisions (figure 16.3(a)). Un premier réseau peut être conçu en reliant deux-à-deux les ports object et les ports event. A chaque itération, l'animator met à jour l'état des objets en fonction des forces de réponse aux collisions calculées par l'interactor.

Les objets masses-ressorts nécessitent une structure différente du fait de l'existence de deux interactions internes : les forces résultant des ressorts, ainsi que les collisions entre deux objets masses-ressorts ou deux parties d'un même objet (*self-collisions*). Ceci se traduit par l'existence de deux interactors reliés à l'animator en charge du mouvement des masses (figure 16.3(b)).

Ces deux réseaux ne présente que la modularisation de simulations très classiques et assez simples. L'intérêt de notre architecture apparaît en combinant les deux réseaux via un nouvel interactor en charge des collisions entre objets rigides et objets masses-ressorts (figure 16.4). Le découpage des responsabilités permet d'effectuer ce couplage

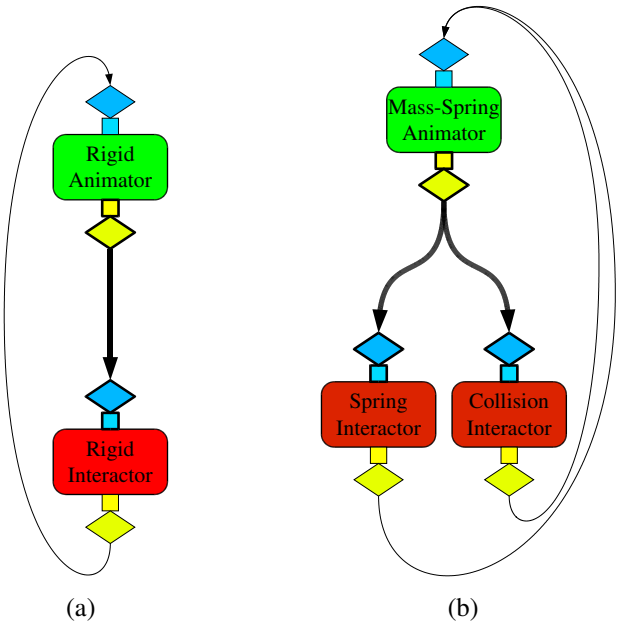


Figure 16.3 *Une simulation d'objet rigides (a) et une simulation de tissus (b). Les ports d'objets et leurs filtres associés sont en gras, les ports d'évènement en traits fins.*

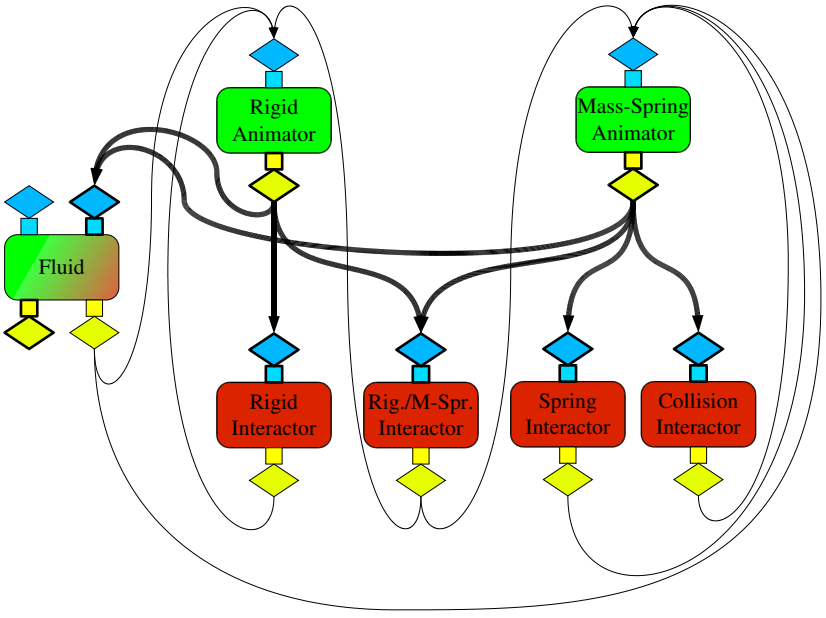


Figure 16.4 *Une application comportant des interactions bidirectionnelles entre des objets rigides, des tissus et un fluide. Le graphe de flux de données relie les composants interdépendants en supprimant les connexions inutiles (par exemple les forces issues des ressorts ne concernent que l'animator masses-ressorts).*

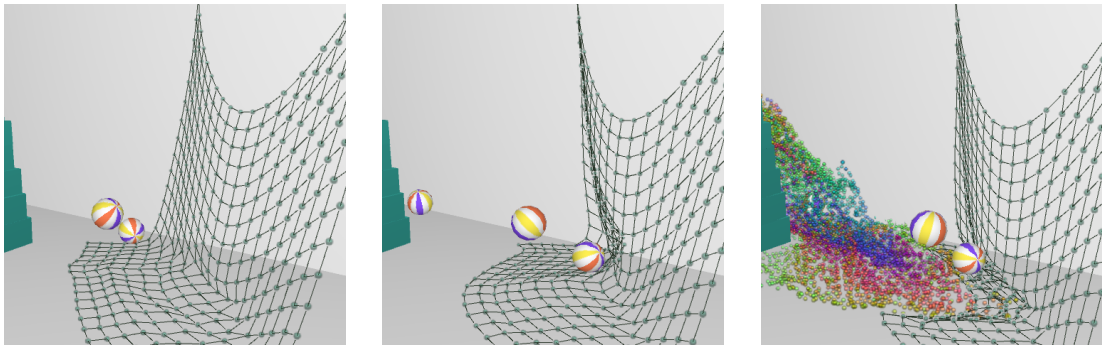


Figure 16.5 *Séquence présentant successivement les interactions d'un filet avec un objet rigide puis avec un fluide.*

sans modifier les autres modules de l'application. Il facilite aussi la répartition des calculs sur plusieurs machines.

Découper une simulation n'est pas toujours bénéfique. C'est par exemple le cas du fluide qui utilise en interne de nombreuses données (grilles de vitesses, particules marqueurs, *level-set* de la surface). Le fluide est donc implanté dans un module unique jouant à la fois le rôle d'animator et d'interactor. La dynamique du fluide est calculée en considérant les objets présents dans le fluide. La pression exercée par le fluide sur ces objets est communiquée en tant que force sur le port de sortie d'évènements.

Le réseau de l'application contient plusieurs cycles. Si les modules s'attendent mutuellement cela peut introduire des *deadlocks*. Pour les résoudre il faut que l'un des éléments de la boucle envoi un message avant la première itération. Par exemple, tous les animators peuvent commencer par envoyer l'état initial des objets.

Une fois la simulation construite, elle peut être reliée au reste de l'application en introduisant des ports supplémentaires pour relier les périphériques entrées (via par exemple un animator associant un objet aux mouvements de la souris, on un interactor permettant d'appliquer des forces particulières aux objets). Le résultat de la simulation peut être visualisé grâce à des modules *viewers* de FlowVR Render utilisant la description des objets produite par les animators en entrée. La figure 16.5 présente le résultat de notre simulation d'exemple, ou des interactions sont visibles entre tous les types d'objets.

16.5.3 Applications distribuées

Pour attendre des performances permettant l'interaction, il est nécessaire de distribuer les calculs sur plusieurs machines. Plusieurs approches sont disponibles pour le faire, et chaque composant de la simulation peut utiliser celle qui est la plus appropriée.

16.5.3.1 Filtrage des données en sous-ensembles

Un module animator ou interactor peut être placé sur plusieurs machines et un réseau de filtrage approprié peut répartir les données entre les différentes instances. Ainsi, plusieurs animators et interactors d'objets rigides peuvent traiter différents groupes d'objets. Le répartition du calcul des collisions doit toutefois prendre en compte les interactions entre les groupes en plus des interactions internes aux groupes.

16.5.3.2 Module parallèle

La répartition et le découpage externe des données peut ne pas être possible, par exemple pour le fluide qui forme un objet unique. Dans ce cas une parallélisation interne des calculs peut être considérée. Le module de fluide utilisant une grille 3D pour ses calculs, une technique classique consiste à découper cette grille en blocs. Des filtres sont ajoutés dans le graphe FlowVR pour filtrer la description des objets vers les blocs concernés, et récupérer les évènements de sortie en conséquence. Les objets possédant une information de boite englobante, ces opérations sont assez simples à réaliser.

16.5.4 Multi-fréquences

Les applications discutées jusqu'à maintenant sont synchrones, c'est à dire que tous les modules s'exécutent à la même fréquence. Cette fréquence est limitée par le module le plus lent. Ces performances peuvent être significativement augmentées en autorisant des fréquences d'exécution différentes entre les modules. Cependant, comme l'expérimentation sur le couplage entre une simulation de cheveux et la reconstruction multi-caméras l'avait démontré (section 8.4.2 page 84), ces différences de fréquences peuvent introduire des instabilités (crénelage, oscillations). Introduire des filtres d'interpolation ou d'amortissement des mouvements et des forces permet d'atténuer ces effets, souvent au détriment de la latence des boucles d'interactions.

Le filtrage des échanges entre modules de fréquences différentes est primordial. Pour simplifier ce filtrage, nous allons nous limiter ici à des différences de fréquences fixées au lancement de l'application. Ainsi par exemple certains modules peuvent être activés 10 fois plus souvent que le reste de l'application. Ce type de différence peut être utile pour les objets rigides, dont les calculs sont relativement peu coûteux mais dont les interactions sont presque instantanées et demandent par conséquent un pas de simulation très court. De même pour les modules utilisant des méthodes d'intégration explicites, rapides mais instables pour les pas de simulations trop longs, ils peuvent s'exécuter beaucoup plus fréquemment que les modules plus coûteux et complexes comme la simulation de fluide.

Dans une simulation où tous les modules sont synchronisés, leur pas de simulation se réfère à la même durée. Ce n'est plus le cas pour les applications multi-fréquences. Cela impose que les modules utilisent une unité de temps commune qui n'est pas en relation

avec leur pas de simulation. Si cela n'est pas le cas alors des filtres doivent être ajoutés pour convertir les données en conséquence.

Implantation et résultats 17

Le modèle de couplage décrit dans le chapitre précédent a été utilisé pour implanter le prototype intégrant trois types d'objets distincts, ainsi qu'une parallélisation des calculs et la gestion d'interactions avec une représentation 3D de l'utilisateur. Ce chapitre présente cette implantation ainsi que les résultats obtenus en terme de performance et de qualité de la simulation.

17.1 Implantation

17.1.1 Objets rigides

L'animation des objets rigides utilise *Jiggle* [35], une implantation effectuée par Danny Chapman de l'algorithme de Guendelman et al. [77]. Il se base sur un champ de distance signée discrétisé sur une grille régulière entourant chaque objet pour calculer les collisions.

17.1.2 Fluide

Le fluide 3D utilise une grille régulière contenant la vitesse du fluide ainsi que le champ de pression. Pour traquer la surface un ensemble de particules marqueurs est utilisé. Elles sont déplacées par le champ de vitesse à chaque itération et permettent de détecter les cases contenant du fluide. Les interactions avec les objets rigides sont calculées selon l'algorithme du *Rigid Fluid* [34], basé sur une discrétisation des objets immergés

dans la grille du fluide. Ces objets sont ensuite modélisés comme le fluide, mais avec une contrainte supplémentaire de rigidité.

Ces calculs étant très longs, l'implantation se base sur MPI pour paralléliser la simulation en découpant l'espace en blocs. Les calculs sont ensuite effectués sur plusieurs machines sur chacun de ces blocs, en prenant soins d'échanger les valeurs aux bords avec les blocs voisins. Le coût de ces communications est proportionnel à la surface des bordures, ce qui rend cette parallélisation efficace plus la taille des données est importante.

17.1.3 Filet masses-ressorts

Les objets déformables les plus faciles à implanter sont ceux à base de masses reliées par des ressorts. Notre objectif étant le couplage de simulations plutôt que les simulations elles-mêmes, c'est cette approche que nous avons utilisé. Ainsi l'animateur masses-ressorts initialise la liste des masses ainsi que les ressorts les liant. Il se contente ensuite de mettre à jour la position des masses en intégrant les forces appliquées par le schéma d'Euler explicite. Les ressorts sont ensuite calculés par un interactor en utilisant une élasticité constante ainsi que des bornes sur l'élongation. Les collisions sont gérées en associant une sphère à la position de chaque masse.

Les collisions avec les objets rigides se font grâce au champ de distance signée de ces objets. Chaque masse est testée dans ce champ, et si elle est à l'intérieur d'un objet une force les repoussant est calculée et appliquée à la masse ainsi qu'à l'objet. Les interactions avec le fluide sont en revanche plus compliquées. Bien qu'une force d'entraînement des masses est facilement calculable en considérant le champ de vitesse du fluide, il est plus difficile d'empêcher le fluide de traverser un tissu imperméable. Pour simplifier ce problème nous considérons l'objet déformable comme un filet, qui laisse donc passer le fluide entre les mailles. L'opposé de la force appliquée aux masses peut être appliquée en retour au fluide pour le ralentir au contact du filet.

17.1.4 Interactions avec l'utilisateur

Pour permettre d'intégrer les actions de l'utilisateur dans les interactions de la simulation, nous utilisons la reconstruction multi-caméras décrite dans la section 8.3 (page 81). Cette étape fournit un mesh 3D du corps de l'utilisateur ainsi que le champ de distance signée associé (section 8.4.3 page 86). Cet objet est alors considéré comme tout autre objet rigide, à la différence qu'il est généré par un animator spécial et qu'aucune force appliquée n'y a d'influence (il faudrait un périphérique de retour haptique pour appliquer ces forces).

17.1.5 Visualisation

Le résultat est visualisé en ajoutant des *viewers* FlowVR Render (chapitre 11 page 99) pour convertir les descriptions des objets en primitives graphiques, qui sont ensuite affichées sur un ou plusieurs projecteurs par un ensemble de *renderers*. L'objet le plus difficile à traiter est le fluide. Pour le visualiser nous utilisons des sprites représentant les particules internes à la simulation. Ces particules étant nombreuses (jusqu'à 200000 dans les scènes testées), des filtres de *culling* sont utilisés pour les envoyer uniquement aux projecteurs où elles sont visibles.

17.2 Résultats

17.2.1 Animation séquentielle hors-ligne

Considérons tout d'abord l'exécution sur une seule machine sans interaction utilisateur. Un graphe de flux de données FIFO force les modules à s'exécuter à la même vitesse. Nous avons testé une simulation d'un environnement comportant les objets suivant :

- 20 objets rigides ;
- un fluide 3D utilisant une grille de $32 \times 64 \times 32$ cellules ;
- un filet masses-ressorts, avec 20×20 noeuds.

Une vidéo présentant le résultat de cette simulation est disponible : `http://www-id.imag.fr/~allardj/these/distsimu.avi`.

L'application était exécutée sur un Bi-Opteron 1.6 GHz avec une carte graphique NVIDIA GeForce FX 5700. La vitesse atteinte fut de 6.5 itérations par seconde.

17.2.2 Parallélisation

Pour atteindre des performances permettant les interactions temps-réel, le module le plus lent, c'est-à-dire le fluide, est parallélisé sur plusieurs machines. La figure 17.1 présente les performances de cette parallélisation, dans le cas où le fluide est exécuté seul, ou lorsqu'il est couplé aux autres objets de la scène. On remarque que le couplage introduit un surcoût assez faible, qui est dû en partie au fait que le mouvement du fluide est modifié par les remouds additionnels autour des objets. Par contre la parallélisation n'est pas très performante, avec une accélération de 4.3 sur 8 machines. Utiliser de plus grandes grilles augmenterait son efficacité, mais ne permettrait plus d'atteindre des fréquences suffisantes.

17.2.3 Exécution interactive

En se basant sur les performances de la simulation une fois parallélisée, nous avons utilisé cette application sur la plateforme semi-immersive GrImage (section 8.1 page 79).

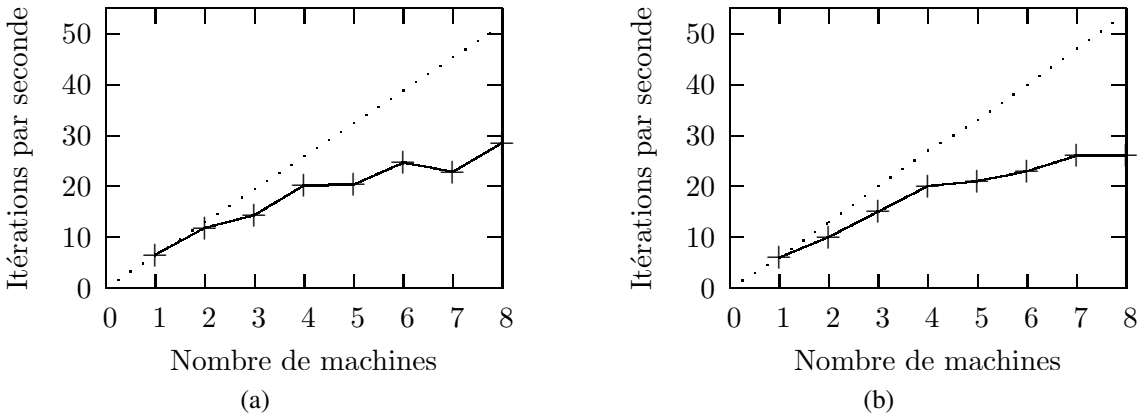


Figure 17.1 *Performance de la parallélisation du fluide en exécution synchrone non interactive avec une grille de* $32 \times 64 \times 32$. *(a) Simulation du fluide seule. (b) Fluide couplé avec les autres objets.*

Les modules de la simulation sont répartis sur les 16 noeuds Bi-Opterons de la grappe. La simulation de fluide utilise 8 noeuds, et le reste des modules sont répartis sur les 8 noeuds restants. La visualisation se fait sur 16 vidéo-projecteurs, pilotés par les mêmes 16 machines. Le reste des machines de la grappe (11 Bi-Xeons) sont utilisées pour piloter les 5 caméras et calculer la reconstruction 3D de l'utilisateur.

Le résultat de cette expérimentation est visible sur la figure 17.2, ainsi qu'à la fin de la vidéo suivante : `http://www-id.imag.fr/~allardj/these/distsimu.avi`. Toutes les interactions entre objets de chaque type sont bien présentes. Toutefois les interactions avec l'utilisateur sont de qualité assez médiocre. En effet, la reconstruction n'est pas très précise, et les objets rebondissent sur le corps reconstruit mais il réagissent mal quand c'est l'utilisateur qui rentre en contact (comme cela est visible au début de la séquence). Ceci est en parti dù au fait que l'algorithme de reconstruction ne calcule pas la vitesse des mouvements, mais uniquement une succession de mesh 3D. Ce manque d'information empêche de transférer une impulsion aux objets, et le corps doit pénétrer sensiblement dans un objet avant que celui-ci ne soit repoussé assez violemment.

Cette application atteint les 18 itérations par seconde, soit approximativement trois fois plus que l'exécution séquentielle, avec en plus un affichage sur 16 projeteurs et des interactions 3D avec l'utilisateur. Le réseau est un goulot d'étranglement important, particulièrement en ce qui concerne la visualisation du fluide. Si les particules sont diffusées à tous les noeuds de rendu la vitesse chute à moins d'une itération par seconde. L'intégration d'une extraction de surface parallélisée avec VTK par exemple (section 13.1.2 page 122) permettrait de diminuer ce problème et d'augmenter le réalisme visuel.

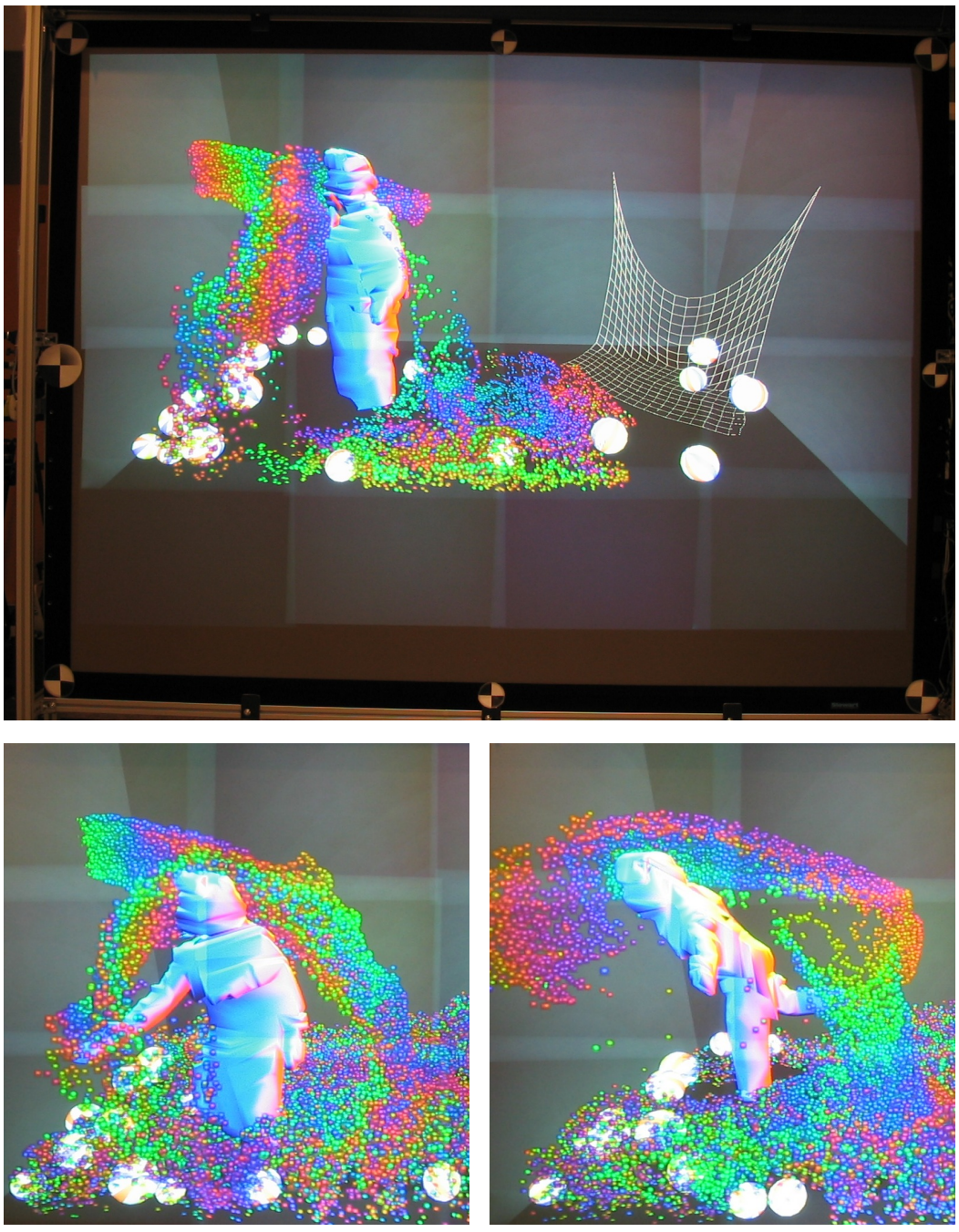


Figure 17.2 *Exécution interactive sur mur d'image avec une interaction utilisateur basée sur la reconstruction multi-caméras.*

Bilan 18

Nous avons présenté une architecture logicielle pour le couplage de simulations basées sur la physique dans le cadre d'applications de réalité virtuelle évoluées. Notre approche repose sur des objets distribués entre des *animators* responsables de la mise à jour de leurs états, et des *interactors* calculant les forces appliquées. Cette approche permet à la fois un développement modulaire de l'application et une exécution distribuée efficace. Les différentes forces appliquées à un objet sont calculées de façon concurrente, permettant de paralléliser le calcul des interactions.

Cette architecture a été validée par une application couplant une simulation de fluide, un moteur de collisions rigides, et une simulation de filet masses-ressorts. Des fréquences de rafraîchissements interactives ont été atteintes en distribuant les modules sur une grappe de PC et en utilisant une parallélisation classique par blocs du fluide. Des périphériques complexes, un réseau de caméras ainsi qu'un mur d'image, ont été rattachés à cette application pour permettre des interactions avec l'utilisateur. Ce système a démontré l'efficacité de l'approche modulaire ainsi que l'intérêt de l'exploitation d'une grappe de calcul pour les applications de réalité virtuelle fortement dynamiques.

Par rapport aux outils et applications développés précédemment, ce travail permet de valider l'adéquation du modèle de base de FlowVR pour développer de nouvelles architectures de couplage. Les outils de développement et d'exécution ont aussi pu être validés sur une application complexe. Ainsi, une centaine de modules sont utilisés dans la version parallèle et interactive de l'application, combinant de nombreux périphériques externes, des calculs parallélisés avec MPI, et de échanges fréquents de données volumineuses.

Les expérimentations effectuées ont montré la faisabilité d'environnements hautement interactifs contenant plusieurs objets de types différents. Des difficultés subsistent pour le paramétrage des différentes simulations (fréquences d'exécution, élasticité des contacts, masses relatives) afin d'obtenir des interactions réalistes.

Certains algorithmes de simulation comportant un couplage très étroit des étapes de calcul, comme la simulation de fluide, ne s'adaptent pas facilement au modèle de découpage animator/interactor. Ils nécessitent de fortes dépendances de données qui requièrent une parallélisation spécifique pour atteindre une exécution haute-performance. La conception modulaire de l'application permet de moduler progressivement ce découpage de façon à choisir le schéma le plus efficace, et intégrer de nouveaux algorithmes plus modulaires quand ils deviennent disponibles.

Les perspectives à plus long terme concernent les applications plus ambitieuses, comportant plus de simulations différentes (objets déformables volumiques, gestion des fractures, objets articulés) et supportant des environnements plus vastes (techniques de niveau de détail dynamiques, approches multi-grilles). Une autre piste de recherche intéressante concerne les interactions à distance, contrairement aux contacts localisés, comme les calculs d'éclairages. L'objectif pourrait par exemple être le calcul distribué des ombres projetées dans les scènes complexes et dynamiques.

V

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés au couplage de codes parallèles pour les applications interactives à grande échelle. Les points suivants ont été abordés :

– Nous avons fourni un travail préliminaire de couplage interactif d'une simulation parallèle avec une visualisation elle aussi parallèle. Deux scénarios ont été étudiés, l'un avec un couplage fort à l'intérieur d'une même boucle de calcul, l'autre mettant en relation deux programmes distincts par un couplage asynchrone.

– Nous avons proposé un nouveau modèle de couplage de code (*FlowVR*), alliant modularité, simplicité, efficacité et extensibilité. Basé sur des composants (*modules*) reliés par un graphe de flux de données, ce modèle sépare la conception des modules de la construction du réseau de l'application. Des objets de filtrages et de synchronisations placés sur ce réseau permettent d'exprimer des schémas de communication collective et des politiques de couplage avancées, respectant les contraintes de cohérences spécifiques aux applications interactives.

– Nous avons présenté une implantation de ce modèle permettant de valider son efficacité au travers de nombreuses applications innovantes et de grande taille, notamment une reconstruction 3D multi-caméras permettant des interactions en temps réel entre le corps de l'utilisateur et les objets et simulations présentes dans l'environnement. Cette application utilise jusqu'à 54 processeurs, 200 processus et 4000 connexions.

– Plusieurs extensions de haut niveau ont été étudiées pour faciliter et standardiser la construction de certaines parties de l'application. La première, *FlowVR Render*, concerne la phase de rendu. En ce basant sur les dernières avancées des cartes graphiques (les shaders) et sur une description modulaire de l'environnement virtuel, nous avons proposé un modèle de rendu parallèle permettant de filtrer et

recombiner de manière efficace les données de différents modules afin de fournir aux machines de rendu les informations nécessaires.

– L'efficacité de FlowVR Render a été testée en adaptant l'outil de visualisation scientifique VTK. En comparaison avec l'outil prédominant de rendu parallèle (*Chromium*), FlowVR Render s'est montré performant et permet de mieux passer à l'échelle, à la fois en terme de machines de rendu que de découpage de la description de la scène. FlowVR Render a aussi permis le développement de nouvelles applications exploitant les fonctionnalités avancées des cartes graphiques pour répartir la charge de calcul entre les CPU et les GPU tout en optimisant les transferts sur le réseau (rendu volumique haute qualité, texturage du modèle reconstruit en recombinant les flux vidéos des caméras, vidéos haute-définition sur mur d'image, visualisation interactive de l'état de la grille Grid5000).

– Enfin nous avons exploré la définition d'un modèle de haut niveau pour la boucle de simulation afin d'obtenir un couplage multi-physique gérant les interactions entre de nombreux objets de types différents. La simulation est découpée en modules gérant les mouvements des objets (*animators*) ou calculant les forces d'interactions entre-eux (*interactors*). Un prototype combinant des objets solides, fluides et déformables a été présenté. Alors que la plupart des travaux n'étudient que des environnements impliquant seulement deux types d'objets différents, notre architecture démontre la faisabilité d'interactions plus complexes. De plus, grâce à la parallélisation des calculs et la distribution des tâches, la simulation atteint le temps réel et permet des interactions entre tous les objets et l'utilisateur. L'environnement virtuel ainsi obtenu dépasse les simples collisions rigides classiques et permet d'envisager la construction d'environnements complexes et réalistes.

En ce concentrant sur les applications interactives et les environnements immersifs, les principes et les outils développés l'ont été dans un objectif d'efficacité et de simplicité. Ceci a permis d'éviter la complexité des modèles de couplage classiques (nécessitant de volumineuses spécifications et un important effort d'implantation). Toutefois, de nombreuses fonctionnalités non essentielles (trop complexes ou redondantes) sont par conséquence non supportées. Ainsi, FlowVR ne supporte aucune dynamicité ni résistance aux pannes. De même, FlowVR Render ne supporte pas de niveau de détail dynamique pour les primitives. Cependant, ces lacunes peuvent faire l'objet d'extensions futures intéressantes.

Les applications développées grâce à FlowVR ont démontré de nouvelles interactions complexes exploitant la puissance des plateformes de type grappe. Quelques expérimentations, notamment dans le cadre de la Fête de la Science 2004, ont relié des machines sur des sites distants. L'étape suivante serait le développement d'applications exploitant les nouvelles plateformes de type grille ou intégrant plusieurs environnements immersifs

distants de manière collaborative.

Un des problèmes principaux à ce jour lors de la construction d'applications FlowVR évoluées concerne l'allocation des ressources disponibles. Le concepteur du réseau est responsable du placement de chaque module sur les différentes machines. Un mauvais placement peut entraîner un déséquilibre de la charge de travail. Des modèles de performances pourraient être utilisés pour calculer le placement optimal sur des machines données. C'est un problème qui se complexifie rapidement avec la taille de l'application (NP-Complet), avec en plus l'originalité que la latence des calculs est un critère de performance important pour une application interactive. Quelques réponses préliminaires à ces questions ont été apportées lors du stage de DEA de Christophe Sadoine co-encadré avec Denis Tristram.

Ce projet de recherche se poursuit actuellement dans le cadre du projet MOAIS (INRIA, ID-IMAG) par l'intégration de FlowVR et de KAAPI, un outil de parallélisation dynamique basé sur un ordonnancement par vol de tâche et graphe de dépendance de données. L'idée serait d'utiliser KAAPI pour la parallélisation interne de certains modules FlowVR. Il faut alors étudier comment interfacer FlowVR et l'ordonnanceur de KAAPI pour gérer les tâches reliées aux ports de communications : FlowVR doit-il contraindre le placement de ces ports ou au contraire KAAPI peut décider dynamiquement de l'emplacement des données, peut-être en prenant en compte un facteur de coup fourni par FlowVR.

Un autre travail en cours dans le cadre du projet MOVI (INRIA, GRAVIR-IMAG) concerne le développement d'une couche d'abstraction concernant la partie vision des applications, afin de gérer facilement les différents formats d'images et de données extraites, ainsi que fournir une librairie de filtres et modules parallèles implantant chaque étape du traitement des données. Un des objectifs étant de pouvoir extraire différentes informations en fonction des besoins de l'application (reconstruction 3D, champs de distance, mouvements et suivi des membres, actions sémantiques, etc). Ces problématiques sont étudiées par Clément Ménier, co-encadré par Edmond Boyer.

L'objectif à long terme, en continuité avec les travaux de cette thèse, est à mon sens la construction d'environnements virtuels de plus en plus étendus et complexes, permettant de simuler de manière réaliste des lieux comme un bâtiment complet (objets tous dynamiques, éclairage photo-réaliste, robinets fonctionnels, piscine, etc), des phénomènes complexes comme le fonctionnement du corps humain (apprentissage des gestes chirurgicaux), ou tout simplement recréer les conditions pour que l'utilisateur se croie totalement immergé. A quand la grille capable de créer la matrice...

Bibliographie

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *6th IEEE Visualization Conference (VIS '95)*, 1995.

[2] R. Ahrem, M. G. Hackenberg, U. Karabek, P. Post, R. Redler, and J. Roggenbuck. Specification of MpCCI. Technical Report 12, GMD-SCAI, Sankt Augustin, Germany, 2001.

[3] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A parallel approach for efficiently visualizing extremely large. Technical Report LAUR-001630, Los Alamos National Laboratory, 2000.

[4] B. A. Allan, R. C. Armstrong, A. P. Wolfe, and J. Ray. The CCA core specification in a distributed memory SPMD framework. *Concurrency Computation*, 14 :1–23, 2002.

[5] J. Allard, E. Boyer, J.-S. Franco, C. Ménier, and B. Raffin. Marker-less real time 3D modeling for virtual reality. In *Proceedings of the Immersive Projection Technology Workshop*, Ames, Iowa, May 2004.

[6] J. Allard, E. Boyer, C. Ménier, and B. Raffin. Running large VR applications on a PC cluster : the FlowVR experience. In *Proceedings of the IPT / EGVE Workshop*, Aalborg, Denmark, October 2005.

[7] J. Allard, M. C. Cabral, C. Goudeseune, H. Kaczmarski, B. Raffin, B. Schaeffer, L. Soares, and M. K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *ACM SIGGRAPH 03 Course*, San Diego, California, July 2003.

[8] J. Allard, V. Gouranton, G. Lamarque, E. Melin, and B. Raffin. Softgenlock : Active Stereo and Genlock for PC Cluster. In *Proceedings of the Joint IPT/EGVE'03 Workshop*, Zurich, Switzerland, May 2003.

[9] 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 Euro-Par 2004*, Pisa, Italia, August 2004.

[10] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net Juggler : Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE VR*, pages 275–276, Orlando, USA, March 2002.

[11] J. Allard, V. Gouranton, E. Melin, and B. Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium*, Orlando, USA, March 2002.

[12] J. Allard and B. Raffin. A shader-based parallel rendering framework. In *IEEE Visualization Conference*, Minneapolis, USA, October 2005.

[13] J. Allard and B. Raffin. Distributed physical based simulation for large vr application. In *IEEE VR Conference*, Alexandria, USA, January 2006.

[14] J. Allard, B. Raffin, and F. Zara. Coupling parallel simulation and multi-display visualization on a PC cluster. In *Euro-Par 2003*, Klagenfurt, Austria, August 2003.

[15] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceeding of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[16] P. Augerat, C. Goudeseune, H. Kaczmarski, B. Raffin, B. Schaeffer, L. Soares, and M. K. Zuffo. Commodity Clusters for Immersive Projection Environments. ACM SIGGRAPH 02 Course, July 2002.

[17] O. Aumage, L. Bougé, A. Denis, J.-F. Méhaut, G. Mercier, R. Namyst, and L. Prylli. A portable and efficient communication library for high-performance cluster computing. In *IEEE International Conference on Cluster Computing (CLUSTER 2000)*, pages 78–87, Tecnhnische Universität Chemnitz, Saxony, Germany, November 2000.

[18] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine : a true multi-protocol MPI for high-performance networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Fransisco, April 2001.

[19] I. S. Bains and J. A. Doss. ShaderGen – fixed functionality shader generation tool. `http://developer.3dlabs.com/downloads/shadergen/`.

[20] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 Users Manual. Technical Report ANL-95/11 - Revision 2.0.29, Argonne National Laboratory, November 2000.

[21] B. Baumgart. A polyhedron representation for computer vision. In *AFIPS National Computer Conference*, pages 589–596, 1975.

[22] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*, July 1998.

[23] F. Bertails and M.-P. Cani. Animation de chevelures : du temps-réel à la recherche de réalisme. In *Groupe de travail français Animation et Simulation (GTAS'04)*, Reims, Juin 2004.

[24] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium : IPDPS 2005*, 2005.

[25] F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley. An approach to parallel MxN communication. In *Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium*, Santa Fe, NM, October 2003.

[26] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler : A Virtual Platform for Virtual Reality Application Development. In *IEEE VR 2001*, Yokohama, Japan, March 2001.

[27] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK : A portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of Supercomputing '96*, 1996.

[28] D. Blythe, editor. *OpenGL® ES Common/Common-Lite Profile Specification, Version 1.0*. The Kronos Group Inc., 2003.

[29] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet : A gigabit-per-second local area network. *IEEE Micro*, 15(1) :29–36, February 1995.

[30] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 594–603. ACM Press, 2002.

[31] K. W. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Walton, and J. D. Wood. Distributed and collaborative visualization. *Computer Graphics Forum*, 23(2), 2004.

[32] G. Burns, R. Daoud, and J. Vaigl. LAM : An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[33] W. Cai, F. B. S. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *PADS '99 : Proceedings of the 13th workshop on Parallel and distributed simulation*, pages 82–89, 1999.

[34] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid : animating the interplay between rigid bodies and fluid. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 04)*, 23(3) :377–384, 2004.

[35] D. Chapman. Jiggle – rigid body demo. http://www.rowlhouse.co.uk/jiggle/.

[36] Y. Chen, D. W. Clark, T. H. A. Finkelstein, and K. Li. Automatic Alignment Of High-Resolution Multi-Projector Displays Using An Un-Calibrated Camera. In *IEEE Visualization 2000*, Salt Lake City, USA, October 2000.

[37] G. Chiola and G. Ciaccio. Efficient Parallel Processing on Low-cost Clusters with GAMMA Active Ports. *Parallel Computing*, 26, 2000.

[38] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-collide : an interactive and exact collision detection system for large-scale environments. In *SI3D '95 : Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff. ACM Press, 1995.

[39] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM*, 35(6) :64–72, 1992.

[40] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. L. Dawe, and M. D. Brown. The immersadesk and infinity wall projection-based virtual reality displays. *SIGGRAPH Comput. Graph.*, 31(2) :46–49, 1997.

[41] Dahlgren, et al. Babel user's guide. Lawrence Livermore National Laboratory, January 2005.

[42] K. Damevski and S. Parker. Parallel remote method invocation and M-by-N data redistribution. In *Proceedings of the 4th Los Alamos Computer Science Institute Symposium*, October 2003.

[43] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, pages 1160–1167, Cancun, Mexico, May 2000. In conjunction with IPDPS 2000.

[44] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[45] J. D. de St. Germain, S. G. Parker, J. McCorquodale, and C. R. Johnson. Uintah : A massively parallel problem solving environment. In *HPDC'00 : Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–42, August 2000.

[46] I. Debled-Rennesson, S. Tabbone, and L. Wendling. Fast polygonal approximation of digital curves. In *Proceedings of International Conference on Pattern Recognition, Cambridge (UK)*, volume 1, pages 461–468, 2004.

[47] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr. Dynamic real-time deformations using space and time adaptive sampling. In *Proceedings of ACM SIGGRAPH 01*, Annual Conference Series. ACM Press / ACM SIGGRAPH, August 2001. Proceedings of ACM SIGGRAPH 01.

[48] A. Denis, C. Pérez, and T. Priol. Portable parallel CORBA objects : an approach to combine parallel and distributed programming for grid computing. In *Proceedings of the 7th International Euro-Par'01 Conference*, pages 835–844, Manchester, UK, 2001. Springer-Verlag.

[49] A. Denis, C. Pérez, and T. Priol. PadicoTM : An open integration framework for communication middleware and runtimes. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID2002)*, 2002.

[50] A. Denis, C. Pérez, and T. Priol. Achieving portable and efficient parallel CORBA objects. *Concurrency and Computation : Practice and Experience*, 15(10) :891–909, August 2003.

[51] F. Devernay and B. Holveck. Blinky, 2005. http://www.inria.fr/rapportsactivite/RA2004/movi/uid34.html.

[52] U. Drepper and I. Molnar. The native POSIX thread library for linux, 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[53] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM Press, 2001.

[54] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744. ACM Press, 2002.

[55] A. Esnard. Modèle pour la redistribution de données complexes. In *RENPAR'16*, Le Croisic, France, April 2005.

[56] A. Esnard, M. Dussere, and O. Coulaud. A time-coherent model for the steering of parallel simulations. In *Proceedings of Euro-Par 2004*, Pisa, Italia, August 2004.

[57] F. Faure and L. Heïgéas. Animal, 2004. http://www-evasion.imag.fr/Membres/Laure.Heigeas/AnimAL/.

[58] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2) :115–128, 1997.

[59] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37 :70–82, 1996.

[60] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30. ACM Press, 2001.

[61] J. Franco and E. Boyer. Exact Polyhedral Visual Hulls. In *Proceedings of BMVC2003*, 2003.

[62] J.-S. Franco, C. Ménier, E. Boyer, and B. Raffin. A distributed approach for real time 3d modeling. In *Proceedings of the IEEE Workshop on Real Time 3D Sensors and Their Use*, Washington, USA, July 2004.

[63] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *Pact'98*, Paris, France, oct 1998.

[64] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1994.

[65] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS : providing fault-tolerance, visualization and steering of parallel applications. *International Journal of High Performance Computing Applications*, 11(3) :224–236, August 1997.

[66] O. Génevaux, A. Habibi, and J.-M. Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pages 31–38. CIPS, Canadian Human-Computer Commnication Society, A K Peters, June 2003. ISBN 1-56881-207-8, ISSN 0713-5424.

[67] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP : High Performance Message Passing over a Cluster of Commodity SMPs. In *Proceedings of Super Computing 99*, Portland, USA, November 1999.

[68] GNOME Project. Orbit2. `http://www.gnome.org/projects/ORBit2/`.

[69] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of ACM SIGGRAPH 98*, pages 447–452. ACM Press, 1998.

[70] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree : A hierarchical structure for rapid interference detection. In *Proceedings of ACM SIGGRAPH 96*, pages 171–180. ACM Press, 1996.

[71] V. Gouranton, S. Limet, S. Madougou, and E. Melin. A scalable cluster-based parallel simplification framework for height fields. In *Eurographics Symposium on Parallel Graphics and Visualization, EGPGV'04*, June 2004.

[72] V. Gouranton, S. Madougou, E. Melin, and C. Nortet. Interactive rendering of massive terrains using PC cluster. In *EuroVis 2005 : Eurographics/IEEE-VGTC Symposium on Visualization*, June 2005.

[73] C. Greenhalgh and S. Benford. Massive : a collaborative virtual environment for teleconferencing. *ACM Trans. Comput.-Hum. Interact.*, 2(3) :239–261, 1995.

[74] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6) :789–828, September 1996.

[75] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[76] W. Grosso. *Java RMI*. O'Reilly & Associates, 2001.

[77] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 03)*, 22(3) :871–878, 2003.

[78] O. Hagsand. Interactive multiuser ves in the dive system. *IEEE MultiMedia*, 3(1) :30–39, 1996.

[79] R. W. Hamming. *Digital Filters*. Prentice-Hall, 3rd edition, 1989.

[80] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proceedings of ACM SIGGRAPH 90*, pages 289–298. ACM Press, 1990.

[81] M. Hereld, I. Judson, and R. Stevens. Introduction to Building Projection-based Tiled Display Systems. *IEEE Computer Graphics and Applications*, 20(4) :22–28, 2000.

[82] G. Hesina, D. Schmalsteig, A. Fuhrmann, and W. Purgathofer. Distributed Open Inventor : A practical Approach to Distributed 3D graphics. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 74–81, London, December 1999.

[83] B. Houston, C. Bond, and M. Wiebe. A unified approach for modeling complex occlusions in fluid simulations. In *Proceedings of the ACM SIGGRAPH 03 conference on Sketches & applications*, pages 1–1. ACM Press, 2003.

[84] G. Humphreys, M. Houston, and R. Ng. Chromium : A stream processing framework for interactive rendering on clusters. In *Computer Graphics Proceedings*, Annual Conference Series. ACM Press / ACM SIGGRAPH, Aug 2002. Proceedings of ACM SIGGRAPH 02.

[85] G. Humphreys, M. Houston, R. Ng, S. Ahern, R. Frank, P. Kirchner, and J. T. Klosowski. Chromium : A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *Proceedings of ACM SIGGRAPH 02*, pages 693–702, 2002.

[86] IEEE. Standard for Distributed Interactive Simulation – Application Protocols (IEEE Std 1278.1). IEEE Comp. Soc., 1995.

[87] IEEE. Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules (IEEE Std 1516). IEEE Comp. Soc., 2000.

[88] R. M. T. II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN : a device-independent, network-transparent vr peripheral system. In *ACM Symposium on Virtual Reality Software & Technology*, 2001.

[89] Institute for Simulation and Training. Communication architecture for distributed interactive simulation (CADIS). Final Draft IST-TR-93-20, University of Central Florida, Orlando, Florida, 1993.

[90] Intel. Open Source Computer Vision Library. http://www.intel.com/research/mrl/research/opencv/.

[91] S. Jafar, T. Gautier, A. W. Krings, and J.-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work- stealing. In L. Springer-Verlag, editor, *Euro-Par'2005*, Lisboa, Portogal, august 30 – september 2 2005.

[92] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira. VR Juggler : A Framework for Virtual Reality Development. In *2nd Immersive Projection Technology Workshop (IPT98)*, Ames, USA, May 1998.

[93] T. Kamachi, T. Priol, and C. René. Data distribution for parallel CORBA objects. In *Euro-Par'2000 : Parallel Processing*, pages 1239–1250, Munchen, Germany, August 2000. Springer-Verlag.

[94] A. Kanitsar, T. Theussi, L. Mroz, M. Sramek, A. V. Bartroli, B. Csebfalvi, J. Hladuvka, D. Fleischmann, M. Knapp, R. Wegenkittl, P. Felkel, S. Roettger, S. Guthe, W. Purgathofer, and M. E. Groller. Christmas tree case study : computed tomography as a tool for mastering complex real world objects with applications in computer graphics. In *Proceedings of IEEE Visualization'02*, pages 489–492. IEEE Computer Society, 2002.

[95] N. Karonis, B. Toonen, and I. Foster. MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5) :551–563, 2003.

[96] K. Keahey, P. Beckman, and J. Ahrens. Ligature : Component architecture for high-performance applications. In *1st NASA Workshop on Performance-Engineered Information Systems*, 1998. To appear in the International Journal of High-Performance and Scientific Applications.

[97] K. Keahey and D. Gannon. PARDIS : A parallel approach to CORBA. In *6th International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 31–39, Portland, Oregon, USA, August 1997. IEEE.

[98] K. Keahey and D. Gannon. Developing and evaluating abstractions for distributed supercomputing. *Cluster Computing*, 1(1) :69–79, 1998.

[99] M. Keckeisen and W. Blochinger. Parallel Implicit Integration for Cloth Animations on Distributed Memory Architectures. In *Proceedings of the Eurographics Parallel Graphics and Visualization Ssymposium*, Grenoble, France, June 2004.

[100] E. Kilgariff and R. Fernando. The GeForce 6 series GPU architecture. In M. Pharr and R. Fernando, editors, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 30, pages 471–491. Addison-Wesley, 2005.

[101] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1) :21–36, 1998.

[102] S. Krishnan and D. Gannon. XCAT3 : A framework for CCA components as OGSA services. In *Proceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[103] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Transactions on PAMI*, 16(2) :150–162, February 1994.

[104] N. A. Le Khac. *Définition et évaluation d'INUKTITUT : une interface pour l'environnement de programmation parallèle asynchrone Athapascan*. PhD thesis, ID / IMAG, Institut National Polytechnique de Grenoble, France, March 2005.

[105] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early Experiences and Challenges in Building and Using A Scalable Display Wall System. *IEEE Computer Graphics and Application*, 20(4) :671–680, 2000.

[106] K.-C. Lin. Dead reckoning and distributed interactive simulation. In *Proceedings of SPIE Conference (AeroSense'95)*, April 1995.

[107] J. Löwy. *COM & .NET Component Services*. O'Reilly & Associates, 2001.

[108] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14 :59–68, July 1994.

[109] D. Margery, B. Arnaldi, A. Chauffaut, S. Donikian, and T. Duval. Openmask : {Multi-Threaded | Modular} animation and simulation {Kernel | Kit } : a general introduction. In Simon Richir, Paul Richard, and Bernard Taravel, editors, *VRIC 2002 Proceedings*, pages 101–110. ISTIA Innovation, June 2002.

[110] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg : a system for programming graphics hardware in a c-like language. In *Proceedings of ACM SIGGRAPH 03*, pages 896–907. ACM Press, 2003.

[111] J. E. Marvie, J. Perret, and K. Bouatouch. Remote interactive walkthrough of city models. In *Proceedings of Pacific Graphics*, volume 2, pages 389–393. IEEE Computer Society, October 2003. Short Paper.

[112] C. Meerwald. Free JAVA & C++ ORBs. http://cmeerw.org/freeorbwatch/.

[113] Message Passing Interface Forum. *MPI : A Message–Passing Interface Standard*. University of Tennessee, Knoxville, Tennessee, 1994.

[114] Message Passing Interface Forum. *MPI-2 : Extensions to the Message–Passing Interface*. University of Tennessee, Knoxville, Tennessee, April 1997.

[115] Microsoft. Distributed component object model. http://www.microsoft.com/com/.

[116] A. Mohr and M. Gleicher. Non-invasive, interactive, stylized rendering. In *SI3D '01 : Proceedings of the 2001 Symposium on Interactive 3D graphics*, pages 175–178. ACM Press, 2001.

[117] A. Mohr and M. Gleicher. HijackGL : reconstructing from streams for stylized rendering. In *NPAR '02 : Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 13–ff. ACM Press, 2002.

[118] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4) :23–32, July 1994.

[119] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[120] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed : a single system image cluster operating system for high performance computing. In *Euro-Par 2003*, Klagenfurt, Austria, August 2003.

[121] J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1), 1999.

[122] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds (CAVW)*, 15 :159–171, 2004.

[123] M. Naef, E. Lamboray, O. Staadt, and M. Gross. The Blue-C distributed scene graph. In *Proceedings of IEEE Virtual Reality 2003*, pages 275–276, March 2003.

[124] R. Namyst and J.-F. Méhaut. PM2 : Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo'95)*, pages 279–285. Elsevier, 1995.

[125] C. Niederauer, M. Houston, M. Agrawala, and G. Humphreys. Non-invasive interactive visualization of dynamic architectural environments. In *SI3D '03 : Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 55–58. ACM Press, 2003.

[126] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays : A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2) :169–189, 1996.

[127] NovodeX AG. NovodeX. http://www.novodex.com.

[128] Object Management Group. The common object request broker : Architecture and specification, v. 2.0, July 1995. Document formal/97-02-25.

[129] Object Management Group. Data parallel CORBA, November 2001. Document ptc/01-11-09.

[130] Object Management Group. The common object request broker : Architecture and specification, v. 3.0, July 2002. Document formal/02-06-01.

[131] Object Management Group. CORBA components specification, v. 3.0, June 2002. Document formal/02-06-65.

[132] Ohio Supercomputer Center. Xmpi – a run/debug GUI for MPI. http://www.lam-mpi.org/software/xmpi/.

[133] M. Olano and A. Lastra. A shading language on graphics hardware : the pixelflow shading system. In *Proceedings of ACM SIGGRAPH 98*, pages 159–168. ACM Press, 1998.

[134] E. Olson. Cluster Juggler–PC cluster virtual reality. Master's thesis, Iowa State University, 2002.

[135] OMG CCM Implementers Group. CORBA component model tutorial, April 2002. Document ccm/2002-04-01.

[136] Open Software Foundation. OSF DCE application environment specification, 1992.

[137] ORL. omniorb. http://omniorb.sf.net.

[138] D. Pape, C. Cruz-Neira, and M. Czernuszenko. *CAVE User's Guide*. Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.

[139] S. Parker and C. Johnson. SCIRun : A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[140] S. Pettifer, J. Cook, J. Marsh, and A. West. Deva3 : architecture for a large-scale distributed virtual reality system. In *VRST '00 : Proceedings of the ACM symposium on Virtual reality software and technology*, pages 33–40. ACM Press, 2000.

[141] T. Pohl, N. Thuerey, F. Deserno, U. Ruede, P. Lammers, G. Wellein, and T. Zeiser. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *Proceedings of SC2004*, November 2004.

[142] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 01*, pages 159–170. ACM Press, 2001.

[143] C. Pérez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. In *Proceedings of the 3th International Workshop on Grid Computing*, pages 88–99, Baltimore, Maryland, November 2002. Springer-Verlag.

[144] P. Rajlich. Cave Quake 3 Arena. http://www.visbox.com/cq3a.

[145] D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Wössner. COVISE in the CUBE : An environnement for analyzing large and complex simulation data. In *2nd Immersive Projection Technology Workshop (IPT98)*, Ames, USA, May 1998.

[146] D. Reiners, G. Voss, and J. Behr. OpenSG basic concepts. In *OpenSG 2002 Symposium*, Darmstadt, 2002.

[147] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. In *ACM Symposium on Virtual reality software & Technology*, pages 47–54, 2001.

[148] C. René and T. Priol. MPI code encapsulating using parallel CORBA objects. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, Redondo Beach, California, USA, August 1999. IEEE.

[149] G. Robertson, S. K. Card, and J. D. Mackinlay. The cognitive coprocessor architecture for interactive user interfaces. In *UIST '89 : Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 10–18. ACM Press, 1989.

[150] J.-L. Roch and *et al.* Athapascan : API for Asynchronous Parallel Programming. Technical report, INRIA Rhône-Alpes, projet APACHE, feb 2003.

[151] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03 : Proceedings of the symposium on Data visualisation 2003*, pages 231–238. Eurographics Association, 2003.

[152] J. Rohlf and J. Helman. IRIS Performer : A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (ACM SIGGRAPH 94)*, pages 381–394. ACM Press, July 1994.

[153] R. J. Rost. *OpenGL® Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[154] M. Roth, G. Voss, and D. Reiners. Multi-threading and clustering for scene graph systems. *Computers & Graphics*, 28(1) :63–66, 2004.

[155] T. Saito and J. Toriwaki. New algorithms for euclidean distance transformations of an n-dimensional digitised picture with applications. *Pattern Recognition*, 27(11) :1551–1565, 1994.

[156] B. Schaeffer and C. Goudeseune. Syzygy : Native PC Cluster VR. In *IEEE VR Conference*, 2003.

[157] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition*. Kitware, Inc., 2003.

[158] Scientific Computing and Imaging Institute. SCIRun : A Scientific Computing Problem Solving Environment. http://software.sci.utah.edu/scirun.html.

[159] M. Segal and K. Akeley. *The OpenGL™ Graphics System : A Specification, Version 1.0*. Silicon Graphics, 1992.

[160] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the mr toolkit. *ACM Trans. Inf. Syst.*, 11(3) :287–317, 1993.

[161] S. Singhal and M. Zyda. *Networked Virtual Environments - Design and Implementation*. ACM SIGGRAPH Series. ACM Press Books, 2000.

[162] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6) :44–52, 1992.

[163] R. Smith et al. Open dynamics engine. http://www.ode.org.

[164] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[165] J. Stam. Stable Fluids. In *Proceedings of ACM SIGGRAPH 99*, pages 121–128, August 1999.

[166] J. Stam. Interacting with smoke and fire in real time. *Communications of the ACM*, 43(7) :76–83, 2000.

[167] W. R. Stevens. *UNIX Network Programming – Networking APIs : Sockets and XTI*, volume 1. Prentice-Hall, 2nd edition, 1998.

[168] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2 : A High-Performance Display Subsystem for PC Clusters. In *Proceedings of ACM SIGGRAPH 01*, 2001.

[169] P. S. Strauss. Iris inventor, a 3d graphics toolkit. In *OOPSLA '93 : Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 192–200. ACM Press, 1993.

[170] Sun Microsystems. Entreprise JavaBeans™ specification, version 2.0, August 2001. http://java.sun.com/products/ejb/docs.html.

[171] I. E. Sutherland. The ultimate display. In *Proceedings of the IFIPS Congress*, volume 2, pages 506–508, New York City, NY, USA, May 1965.

[172] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[173] P. Terdiman. OPCODE : Optimized collision detection. http://www.codercorner.com/Opcode.htm.

[174] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers. Wallflower : Principles and practice of background maintenance. In *Proceedings of the 7th International Conference on Computer Vision*, pages 255–261, Kerkyra, Greece, 1999.

[175] University of Minnesota. PowerWall. http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[176] University of North Carolina. Collision detection systems. http://www.cs.unc.edu/~geom/collide/.

[177] C. Upson, T. Faulhaber Jr., D. Kamins, D. H. Laidlaw, D. Schlegel, L. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System : A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4) :30–42, 1989.

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

[179] T. von Eicken, D. Culler, S. C. Goldstein, and K. E. Schauser. Active messages : a mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, May 1992.

[180] VRCO. Trackd. http://www.vrco.com/trackd/Overviewtrackd.html.

[181] C. Wang, J. Gao, and H.-W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Proceedings of the Eurographics Parallel Graphics and Visualization Ssymposium*, pages 23–30, Grenoble, France, June 2004.

[182] D. Winer. XML-RPC specification, 1999. http://www.xmlrpc.com/spec.

[183] World Wide Web Consortium. SOAP 1.2. http://www.w3.org/TR/soap/.

[184] F. Zara. Simulation physique de textiles sur grappe de processeurs. In *AFIG 2001*, Limoges, November 2001.

[185] F. Zara, F. Faure, and J.-M. Vincent. Physical cloth simulation on a PC cluster. In X. P. D. Bartz and E. Reinhard, editors, *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, September 2002.

[186] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker. SCIRun2 : A CCA framework for high performance computing. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 72–79, 2004.

[187] S. Zhou, W. Cai, B.-S. Lee, and S. J. Turner. Time-space consistency in large-scale distributed virtual environments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(1) :31–47, January 2004.

Figures couleur

Résumé

FlowVR : calculs interactifs et visualisation sur grappe

Cette thèse combine le calcul haute performance à la réalité virtuelle pour permettre la conception de méthodes de couplage de composants parallèles à l'intérieur d'applications distribuées et interactives.

Un nouveau modèle de couplage est présenté, conçu selon des critères de modularité, simplicité, efficacité et extensibilité. La construction des applications repose sur une séparation entre la programmation de modules parallèles réutilisables et la définition de l'application sous forme de graphe de flux de données contenant des mécanismes de filtrage et de synchronisations, permettant d'exprimer des schémas de communication collective et des politiques de couplage avancées.

Ce travail sur le couplage interactif est complété par une extension haut niveau concernant le rendu distribué. En exploitant une description modulaire de la scène 3D en primitives indépendantes basées sur l'utilisation de shaders, des réseaux de filtrage permettent de combiner plusieurs flux pour acheminer efficacement les informations aux machines de rendu. Ce système est très extensible et permet la création de nouvelles applications exploitant la puissance des cartes graphiques pour décharger certains calculs des processeurs et réduire les transferts réseau.

De nombreuses applications nouvelles sont ainsi développées, combinant des algorithmes de vision parallélisés immergeant l'utilisateur dans l'environnement virtuel, et des interactions avec des objets contrôlés par des simulations physiques distribuées (poterie, collisions, fluides).

Mots-clés Couplage de codes parallèles, simulations interactives, rendu distribué, réalité virtuelle, grappes de PC.

Abstract

FlowVR : interactive computations and cluster-based visualization

This thesis combines high performance computing and virtual reality to design parallel components coupling methods inside distributed interactive applications.

A new coupling model designed with modularity, simplicity, efficiency and extensibility criteria is presented. Applications are constructed following a two phases process : programming reusable parallel components, and defining the application as a data-flow graph with filtering and synchronisation mechanisms, expressing advanced collective communication schemes and coupling policies.

An extension of this work is presented for distributed rendering. Using a modular scene description based on independent shader-based primitives, filtering networks are designed to efficiently combine multiple graphics streams up to the rendering nodes. This framework scales well and allow the creation of new applications, exploiting the graphics cards' power to remove computations from CPUs which reducing network communications.

Several innovative applications are created using these tools. They includes parallel computer vision algorithms, immersing the user inside a virtual world, which can then interact with objects controlled by several distributed physical simulations (carving, collisions, fluids).

Keywords Parallel code coupling, interactive simulations, distributed rendering, virtual reality, PC clusters.