High Performance Interactive Computing

Bruno Raffin

HABILITATION À DIRIGER DES RECHERCHES

INPG

Spécialité : "Informatique"

Bruno Raffin

Calcul interactif haute performance

Le 3 mars 2009

JURY

MARIE-PAULE CANI	Grenoble INP	Présidente
CAROLINA CRUZ-NEIRA	University of Louisiana at Lafayette	Rapporteur
CHARLES HANSEN	University of Utah	Rapporteur
THIERRY PRIOL	INRIA Rennes - Bretagne Atlantique	Rapporteur
JEAN-MICHEL DISCHLER	Université Louis Pasteur, Strasbourg	Examinateur
PASCAL GUITTON	Université Bordeaux I	Examinateur

Contents

Chapter 1

Introduction

Images and interactivity are central to scientific visualization and virtual reality. Large data sets are processed to produce 3D images often visualized on advanced display devices like CAVEs or display walls. The user expects to interact smoothly with the application, relying on devices like 3D trackers or haptic systems for an improved feel of immersion. Parallel machines can be used to provide the required I/O, storing and computing resources.

Over the last 8 years my research work focused on harnessing the power of parallel machines, mainly PC clusters, for *high performance interactive computing*. New issues and opportunities appeared as processors have been switching to multi and many core architectures, and new generations of powerful co-processors, some of them derived from GPUs, have emerged. A synthesis of this research activity is presented in Chapter 2 (in French) as well as a detailed resume (Chapter 3 in French, Chapter 4) in English). Initially focused on powering immersive multi-projector environments using commodity PCs and graphics cards, the work extended to a more generic approach for distributed rendering (Chapter 5). The problem of handling the complexity of large interactive applications, developed by several users, over several years, coupling various heterogeneous codes, some of them parallelized, led to the development of the FlowVR middleware (Chapter 6). FlowVR is dedicated to interactive applications and relies on a hierarchical component model. It has been used for several applications, presented in this chapter as well. Amongst these applications, we developed two parallel multi-camera 3D modeling algorithms, one of them relying on work stealing to balance the load between CPUs, some of them using a GPU as co-processor. This work on the Grimage platfrom led to a complete real-time computer vision system that, coupled with a physics simulation engine, enabled full body interactions with virtual objects (Chapter 7). Chapter A includes a selection of articles. After a short summary, the conclusion presents on-going works and discusses some research directions that remain to be explored (Chapter 8).

Chapter 2

Synthèse de l'activité de Recherche

Ce document retrace l'essentiel de mes activités de recherche depuis 2000. Cette période correspond à une réorientation de mon activité de recherche par rapport à mes travaux antérieurs. Mon travail de thèse à l'Université d'Orléans portait sur l'étude et la conception de langages de programmation parallèle par une approche théorique basée sur des spécifications par sémantiques opérationnelles et dénotationnelles avec preuves d'équivalence. Le séjour postdoctoral à l'Iowa State University s'est focalisé plutôt sur de la mesure de performance de diverses machines parallèles (grappes de PC, IBM SP, Cray T3D et SGI Origin 2000). De retour en France à l'automne 1999 comme Maître de Conférences à l'Université d'Orléans, nous avons monté avec Valérie Gouranton et Emmanuel Melin, eux aussi Maîtres de Conférences à l'Université d'Orléans, une petite équipe pour étudier les problèmes de l'usage du parallélisme pour la réalité virtuelle. Cette réorientation fait suite à la découverte du domaine de la réalité virtuelle durant mon séjour postdoctoral. L'Iowa State University héberge en effet le VRAC (Virtual Reality Applications Center), dirigé à l'époque par Carolina Cruz-Neira, co-conceptrice du CAVE. La rencontre avec Carolina Cruz-Neira m'a permis de mesurer l'intérêt et les difficultés que pouvaient représenter l'usage des grappes de PC équipées de cartes graphiques 3D pour piloter des CAVE. A l'époque, le domaine des cartes graphiques 3D pour le jeux vidéo était en pleine expansion tout comme l'usage des grappes de PC pour le calcul intensif. Mais encore très peu de configurations à base de grappes pilotaient des environnements immersifs multi-projecteurs. Les CAVE étaient pilotés par des machines dédiées de type SGI Onyx.

Dans un premier temps, nous nous sommes concentrés sur le développement d'une solution pour piloter des environnements immersifs multi-projecteurs avec du matériel standard. Ces travaux ont donné lieu au développement de deux logiciels, Net Juggler, une extension pour grappes de PC d'une des suite logicielle libre la plus utilisée pour la réalité virtuelle, et SoftGenLock, un logiciel pour le genlock (synchronisation des signaux vidéos) de cartes graphiques standards, com-

posant essentiel pour permettre la visualisation stéréoscopique de type active. Ne disposant pas d'environnement immersif, Sabine Coquillart, alors à l'INRIA Rocquencourt, nous a permis de tester notre solution sur son Workbench.

Sur la base de cette première expérience, au caractère assez technologique, mais essentielle dans ce processus de reconversion thématique, nous avons développé une approche plus généraliste de l'usage des capacités offertes par les grappes de PC pour les applications de réalité virtuelle. Ce travail a conduit au développement de l'intergiciel FlowVR, dont le développement reste encore très actif aujourd'hui. A l'opposé des intergiciels habituels du domaine de la réalité virtuelle construits par agrégation d'outils plus spécialisés comme des graphes de scènes, des bibliothèques de pilotes de périphériques, etc., FlowVR a été conçu comme un environnement de couplage logiciel dépouillé, focalisé sur les aspects réseau, parallélisme et modularité des applications. L'intégration d'outils plus spécialisés se fait dans une seconde phase en fonction des objectifs applicatifs. FlowVR est un intergiciel basé sur des composants hiérarchiques. Il est destiné aux applications interactives de grande taille utilisant de multiples ressources, souvent hétérogènes, de type capteurs, entités de calcul et périphériques de sorties. L'accent est mis sur un environnement qui contraint dans une certaine mesure le développeur à construire une application très modulaires. Cette modularité à un double objectif. Elle favorise des pratiques saines de génie logiciel pour la maintenance, l'extension et la réutilisation de l'application. Elle force aussi l'utilisateur à clairement définir les échanges, c'est-à-dire les dépendances de données, entre les différents composants, facilitant la distribution de ces même composants sur une architecture parallèle. L'objectif est de permettre au développeur, qui n'est souvent pas un expert en parallélisme, de construire une application qui conduira à des exécutions parallèles relativement performantes. Un usage optimal des ressources demande dans tous les cas un effort plus significatif d'optimisation nécessitant une expertise avancée en parallélisme. On retrouve ici des motivations des approches orientées composants ou de la programmation parallèle par squelettes.

En Octobre 2001, alors que les réflexions autour de FlowVR n'en étaient qu'à leurs débuts, j'ai intégré l'équipe-projet APACHE (maintenant scindée entre MESCAL et MOAIS) de l'INRIA Rhône-Alpes en tant que Chargé de Recherche INRIA. Ce changement a eu une influence significative sur la direction qu'ont pris mes recherches. Sous l'impulsion de Brigitte Plateau, s'est élaboré le projet d'une plate-forme matérielle pour les applications interactives 3D associant une grappe de PC, un réseau de caméras et un mur d'images. Cette plate-forme, appelée Grimage, a fait émergé des collaborations durables et enrichissantes avec les équipe-projets PERCEPTION, en particulier Edmond Boyer, et EVASION par l'intermédiaire de François Faure. Le travail avec PERCEPTION s'est focalisé sur la parallélisation de l'algorithme de reconstruction 3D multi-caméras EPVH développé par Jean-Sébastien Franco et Edmond Boyer. Ce travail s'est avéré un excellent cadre pour valider FlowVR et nous a conduit à développer une première application interactive de taille conséquente. Le travail avec EVASION s'est lui plutôt focalisé sur la simulation physique temps-réel. Il s'est concrétisé par

l'intégration du logiciel SOFA dans les applications Grimage. Dans un second temps, nous avons travaillé au problème de la parallélisation de SOFA sur machine multi-processeurs à mémoire partagée, travail en cours qui n'a pas encore conduit à des publications. Nous avons par ailleurs mené des travaux sur le rendu en environnement multi-projecteurs piloté par grappe de PC. Les travaux principaux sur le domaine ont conduit à FlowVR Render, une extension de FlowVR qui implante un protocole de transport des données graphiques définissant des primitives indépendantes. Le protocole repose sur l'usage des shaders plutôt que la machine à états OpenGL qui se prête difficilement à la définition de primitives indépendantes. L'objectif est de pouvoir émettre des primitives graphiques de plusieurs sources sans qu'il y ait des conflits concernant l'ordre d'exécution des ces primitives au niveau du rendu. Pour les applications de grande taille, cette approche permet d'éviter de devoir centraliser la génération des données de rendu. Ces travaux ont été associés dans des applications interactives impliquant jusqu'à un cinquantaine de processeurs, une dizaine de caméras et 16 projecteurs. Ce travail de développement lourd, où les problèmes à la fois matériels et logiciels sont courants, est néanmoins fondamental. C'est une base de test pour valider les logiciels tels que FlowVR ou SOFA et guider les nouveaux développements. Disposer d'une implantation temps réel opérationnelle permet de tester in situ de nouveaux modes d'interaction et d'avancer beaucoup plus concrètement vers de nouvelles problématiques qu'à travers des exécutions temps-réel imaginées. Évidemment ce travail rallonge significativement le cycle des publications. Par contre, il ouvre la voie à du transfert technologique. Une première tentative avait été menée en 2004 avec Icatis, société co-fondée par Philippe Augerat, Pierre Neyron et moi-même. Plus récemment l'ensemble de la suite logicielle développée pour la reconstruction 3D multi-caméras a été transférée vers la société 4DView Solution, crée en septembre 2007 par Richard Broadbridge, Clément Ménier et Florian Geffray, ces deux derniers ayant activement participé à Grimage.

FlowVR permet d'aborder un premier niveau de parallélisation statique et à gros grain des applications. Certains composants de l'application nécessitent des parallélisations plus fines pour tirer efficacement parti des ressources disponibles. Une charge de travail variable peut en particulier motiver le recours à des techniques dynamiques d'équilibrage. Un travail initié avec Jean-Louis Roch à porté sur l'étude des techniques de vol de tâches pour les applications interactives. L'application visée est une modélisation 3D temps-réel par raffinement recursif d'un structure d'octree. C'est un algorithme classique de reconstruction 3D qui ne calcule pas un enveloppe visuelle exacte comme EPVH, mais produit une structure régulière et non pas un maillage. L'octree peut être intéressant dans certains cas, par exemple pour des calculs rapides de collisions. Étant dans un contexte interactif, nous avons rajouté une contrainte supplémentaire par rapport au vol de tâches classique: le calcul doit pouvoir être arrêté à tout moment en produisant un résultat acceptable. Cet algorithme est donc de type "anytime". L'objectif est d'imposer une durée maximum de calcul pour assurer une exécution interactive. L'algorithme obtenu donne à la fois de bons résultats expérimentaux et une

efficacité théorique asymptotiquement optimale. Cette première expérience s'est prolongée avec des objectifs plus ambitieux autour de la parallélisation de SOFA, travail en cour comme mentionné plus haut.

Outre les chercheurs avec qui j'ai collaboré sur ces différents travaux, il est important de mentionner la participation importante des stagiaires, ingénieurs et doctorants. Je m'excuse par avance de ne pouvoir mentionner explicitement toutes ces personnes. Je me dois cependant de citer Jérémie Allard, d'abord stagiaire de maîtrise à l'Université d'Orléans puis thésard dans l'équipe-projet MOAIS. Jérémie Allard est l'un des piliers du développement de Net Juggler, SoftgenLock, FlowVR et de la plate-forme Grimage. Il est maintenant Chargé de Recherche à l'INRIA Lille. Clément Ménier, co-encadré avec Edmond Boyer, a eu la difficile tâche de travailler à l'intersection du parallélisme et de la vision par ordinateur. Le travail qu'il a effectué avec Jérémie Allard autour de Grimage, FlowVR et de la modélisation 3D temps réel a été très créatif. Il participe aujourd'hui à l'aventure 4DView Solutions dont il est l'un des fondateur. Luciano Soares nous a rejoint pour un postdoc. Il a travaillé à la parallélisation de la modélisation par octree. Il est actuellement chercheur chez Petrobras au Brésil. Jesus Verduzsco a travaillé sur le rendu sur mur d'images des applications X. Il est maintenant Maître de Conférences à l'IUT de Colima, Mexique. Plus récemment, Benjamin Petit, co-encadré avec Edmond Boyer, Everton Hermann, co-encadré avec François Faure, Marc Tchiboukdjian, co-encadré avec Vincent Danjean et Jean-Louis Roch, ainsi que Jean-Denis Lesage se sont aussi lancés dans des thèses sur le calcul interactif 3D. Une difficulté commune qu'ils rencontrent est ce positionnement à mi-chemin entre plusieurs communautés, la vision par ordinateur, le parallélisme, la réalité virtuelle et l'informatique graphique. La publication des résultats obtenus demande un effort supplémentaire pour assimiler les habitudes, le style et le point de vue de chaque communauté (forme des papiers, travaux à mentionner, etc.).

Initialement centrées sur le pilotage par des grappes de PC des environnement immersif multi-projecteurs, mes activités de recherche se positionnement aujourd'hui sur une thématique que je qualifierais de *calcul interactif haute performance*. La puissance de calcul disponible poursuit sa progression exponentielle mais aujourd'hui en offrant plus de parallélisme, à travers le calcul nébuleux ("cloud computing"), les processeurs multi-coeurs, les cartes graphiques massivement parallèles et programmables pour le rendu 3D mais aussi le calcul intensif, les MPSoCs. Cette progression de la puissance disponible peut évidemment être mise à profit pour des simulations plus complexes, mais aussi pour rendre interactifs certains calculs. Alors que dans le calcul intensif traditionnel, l'humain est peu pris en compte, il devient ici un élément important. Comment interagir avec un programme ? Il y a évidement des critères simples tels que la fréquence et la latence, mais aussi des problèmes difficiles sur les interfaces d'interaction, c'est-à-dire comment et quelles informations extaire du réel, que restituer à l'utilisateur en utilisant le plus efficacement possible les capacités sensorielles et cognitives de l'humain. L'humain est aussi un expert qui, lorsqu'il développe une application, ne parvient pas à transférer tout son savoir-faire dans le programme. Le pilotage

interactif de l'application peut-il lui permettre d'améliorer ce transfert d'expertise ?

Chapter 3

Curriculum Vitae

Bruno RAFFIN
Né le 31 décembre 1970 en France (Vienne)
Marié, un enfant.

3.1 Fonctions

- **Depuis Octobre 2001** Chargé de Recherche INRIA Rhône-Alpes.
- **Septembre 1999 - Septembre 2001** Maître de Conférences au LIFO, Université d'Orléans.
- **Janvier 1998 - Août 1999.** Séjour postdoctoral à l'Iowa Sate University, USA. Une charge d'enseignement de 60 heures par semestre complète mon activité de recherche.
- **Septembre 1997 - Décembre 1997.** Demi poste d'ATER (Université d'Orléans).
- **Octobre 1996 - Août 1997.** Boursier MESR et vacataire (Université d'Orléans).
- **Décembre 1995 - Septembre 1996.** Service militaire.
- **Novembre 1993 - Novembre 1995.** Boursier MESR et vacataire (Université d'Orléans).

3.2 Cursus universitaire

- **1993-1997.** Thèse d'informatique au Laboratoire d'Informatique Fondamentale d'Orléans (LIFO).
  - *Titre* : "Un modèle structuré de communication et de synchronisation pour le parallélisme de tâches"
  - *Direction scientifique* : Bernard Virot et Robert Azencott
  - *Soutenance* : 16 décembre 1997
  - *Composition du jury* :

Président :	Guy-René Perrin	Université Louis Pasteur - Strasbourg
Rapporteurs:	Luc Bougé	École Normale Supérieure de Lyon
	Joaquim Gabarró	Université Polytechnique de Catalogne
Examinateurs :	Robert Azencott	École Normale Supérieure de Cachan
	Gaétan Hains	Université d'Orléans
	Henri Thuillier	Université d'Orléans
	Bernard Virot	Université d'Orléans

- **1993.** DEA d'Informatique Fondamentale de l'École Normale Supérieure de Lyon.
- **1992.** Maîtrise de Mathématiques Discrètes de l'Université Claude Bernard, Lyon.

3.3 Enseignement

- **Depuis 2003**
  - **Université d'Orléans** : Architectures parallèles et protocoles hautes performances : 8h de cours par an - DEA d'informatique.
  - **Université Joseph Fourier, Grenoble** : Introduction au parallélisme : 18h de cours par an - 2de année ISTG.
- **2001-2002**
  - **Université d'Orléans** : Architectures parallèles et protocoles hautes performances : 8h de cours - DEA d'informatique.
  - **Université de Laval** : Réalité virtuelle sur grappe de PC : 6h de cours - DESS INOREV

  - **Université Joseph Fourier, Grenoble** : Introduction au parallélisme : 18h de cours - 2de année ISTG.

- **1999-2001 : Université d'Orléans.**
  - Architecture des ordinateurs : 38h de cours et 27h de TD - Licence d'informatique.
  - Système d'exploitation : 81h de TD et projets - Licence d'informatique.
  - JAVA : 45h de TP et projets - DESS CCI (Compétences Complémentaires en Informatique).
  - Réseaux et technologie internet : 56h de TD et 28h de TP - DESS CCI
  - Cryptographie et sécurité des réseaux : 41h de TD et projets - DESS SIRAD (Sécurité, Réseaux et Aide à la Décision).
  - Architectures parallèles et protocoles hautes performances : 18h de cours - DEA d'informatique.
  - Encadrement de 12 stages en entreprise - Maîtrise informatique, DESS CCI, DESS SIRAD.

- **1998-1999 : Iowa Sate University.**
  - Analyse : 180 heures de cours-TD - Classes undergraduate math 165 et math 166 (principalement de futurs ingénieurs).
  - Parallélisme de tâches : 4h de cours - Classe de master Computer Science 525.

- **1994-1997 : Université d'Orléans.**
  - Algorithmique et Turbo Pascal : 132 heures de TD - Deug scientifique.
  - Mathématiques de l'informatique : 94 heures de TD et 40 heures de cours/TD - Deug scientifique.
  - Bureautique : 44 heures de TD - Deug AES.

3.4 Encadrement d'activités de recherche

3.4.1 Thèses

- **Depuis Octobre 2007 : Marc Tchiboukdjian (thèse BDI financée CNRS/CEA)**. Algorithmes parallèles et cache oblivious pour les structures de données 3D. Directeur de thèse: Denis Trystram. Co-encadrants: Vincent Danjean, Jean-Louis Roch et Bruno Raffin.

- **Depuis Octobre 2007 : Benjamin Petit (thèse financée par le contrat ANR DALIA)**. Interaction et environnements multi-caméras. Directeur de thèse: Edmond Boyer. Co-encadrant: Bruno Raffin.

- **Depuis Octobre 2006 : Everton Hermann (thèse CORDI financée par l'INRIA)**. Algorithmique parallèle pour le moteur physique SOFA. Directeur de thèse: Bruno Raffin. Co-encadrant: François Faure.

- **Depuis Septembre 2006 : Jean-Denis Lesage (thèse MESR)**. Algorithmes adaptatifs pour les applications interactives de grande taille. Directeur de thèse: Denis Trystram. Co-encadrant: Bruno Raffin.

- **Octobre 2005 - Janvier 2006 : Thomas Arcila (thèse CIFRE financée par BULL)**. Rendu "sort-first" haute performance. Directeur de thèse: Denis Trystram. Co-encadrant: Bruno Raffin. Arrêt après un an de thèse. Ingénieur chez Mercury Computer, Bordeaux.

- **Septembre 2003 - Août 2007 : Clément Ménier (Allocataire Normalien)**. Reconstruction 3D multi-caméras en temps-réel sur grappe de PC. Directeur de thèse: Radu Horaud. Co-encadrants: Bruno Raffin et Edmond Boyer. Co-fondateur et responsable du développement logiciel de la société 4D View Solutions crée en Septembre 2007.

- **Septembre 2002 - Novembre 2005 : Jérémie Allard (thèse MESR).** Intergiciels pour les applications de réalité virtuelle de grande taille. Directeur de thèse: Brigitte Plateau. Co-encadrant: Bruno Raffin. Chargé de Recherche INRIA, équipe-projet ALCOVE, Lille.

- **Novembre 2001 - Juin 2005 : Jesus Verduzco (thèse financée par le Mexique).** Environnement X Window pour mur d'images. Directeur de thèse: Brigitte Plateau. Co-encadrant: Bruno Raffin. Maître de Conférences à l'IUT de Colima, Mexique.

3.4.2 Postdoc

- **Décembre 2005- Décembre 2006 : Luciano Soares (financé par l'INRIA).** Octree parallèle adaptatif. Chercheur chez Petrobras, Brésil.

3.4.3 Ingénieurs

- **Depuis Octobre 2008 : Thomas Dupeux (financé par ADT GrimDev).** Support à la plateforme Grimage.

- **Depuis Mai 2008 : Antoine Vanel (financé par contrat ANR FVNANO).** Développement de FlowVR.

- **2003- 2004 : Loick Lecointre (financé par contrat RNTL GEOBENCH).** Développement de FlowVR. Ingénieur chez Amadeus, Paris.

3.4.4 Stages

- Encadrement ou co-encadrement de 28 stages depuis 2000 (niveau M1 et M2).

3.5 Responsabilités administratives

- Président de la commission de développement technologique (CDT) de l'INRIA Rhône-Alpes (depuis 2007). Gestion des appels pour les postes d'ingénieurs associés.

- Membre suppléant de la commission d'évaluation (CE) de l'INRIA (2005-2008).

- Membre du jury de recrutement CR2 de l'INRIA Bordeaux (2008) en tant que membre de la CE.

- Membre de la commission de spécialistes de l'Université Joseph Fourier (suppléant en 2004-2006, titulaire depuis 2007).

- Membre suppléant de la commission de spécialistes de l'Université d'Orléans (depuis 2008).

- Membre du conseil de perfectionnement du master IRAD, Université d'Orléans (depuis 2008).

- Gestion et animation de la collaboration des équipe-projets MESCAL et MOAIS avec les Universités du Rio Grande do Sul, Brésil. Financements obtenus:

  - PICS CNRS (2005-2007),
  - Capes/Cofecub (2006-2008),
  - Equipe associée INRIA Diode-A (2006-2008),
  - INRIA/Cnpq (2008-2010).

- Remplaçant du responsable du DESS SIRAD, Université d'Orléans, pendant le premier semestre de 2000-2001. Gestion des emplois du temps, des examens, de l'attribution des bourses, recherche d'intervenants extérieurs, etc.

3.6 Contrats

- **Participe au projet Européen Interact (2007-2008).** Objet : Interfaçage de la modélisation 3D et de la parole. Partenaires: les équipe-projets PERCEPION et MOAIS, Eptron, Holographika, Total Immersion et Vecsys SA.

- **Participe au projet ANR VULCAIN (2007-2010)**. Objet : fuiabilité des structures industrielles soumises à des contraintes dynamiques. Partenaires: les équipe-projets INRIA EVASION et MOAIS, le 3S-R, l'IPSC-ELSA, le CEG-DGA, le LEES, le LaM, l'INERIS, l'IRSN, le CEA, le SME Environnement, Phimeca et Bull.

- **Thèse BDI co-financée par le CNRS et le CEA/DIF (2007-2010)).**

- **Cordinateur national du projet ANR DALIA (2007-2009)**. Objet : Applications interactives 3D distribuées et hétérogènes. Partenaires: les équipe-projets INRIA PERCEPTION, MOAIS, I-parla et le LIFO de Université d'Orléans.

- **Responsable INRIA du projet ANR FVNANO (2008-2010)**. Objet : Applications interactives de manipulation de nano-structures. Partenaires: l'équipe-projets INRIA MOAIS, le LIFO de Université d'Orléans, le CEA/DIF et le Laboratoire de Biochimie Théorique de l'IBPC.

- **Participe au projet RNTL OCETRE (2004-2005)**. Objet : Capture de mouvement temps-réel multi-caméras. Partenaires: les équipe-projets INRIA MOVI et APACHE, les sociétés Thalès et Total Immersion.

- **Responsable INRIA du projet RNTL GEOBENCH (2003-2004)**. Objet : Visualisation immersive distribuée et interaction haptique appliquées aux données (géo)scientifiques. Partenaires : les équipe-projet INRIA APACHE et I3D, le LIFO de l'Université d'Orléans, le CEA, le BRGM et la société TGS.

- **Participe à l'ACI Masse de données CYBER II (2003-2005)**. Objet : Capture, reconstruction et incrustation temps réel d'un animateur dans un monde virtuel. Partenaires: équipes-projets INRIA ARTIS, MOVI et APACHE, le laboratoire LIRIS de Lyon.

- **Thèse CIFRE financée par Bull (2005).**

3.7 Transfert technologique

- **Transfer du code de reconstruction 3D temps réel à la société 4DView Solutions (2007).** La société 4DViews Solutions a été fondée en septembre 2007 et propose des environnements multi-caméras basés sur la technologie développée pour la plate-forme Grimage.

- **Co-fondateur de la société Icatis (2003)** au titre de l'article 25.2 de la loi sur l'innovation (conseiller scientifique), avec Philippe Augerat (25.1) et Pierre Neyron. Icatis a développé des solutions logicielles dans le domaine du calcul intensif sur grappe de PC, grappe graphique et intranet de calcul. Elle a

été lauréate du concours ANVAR émergence 2003 et du concours création d'entreprise 2004. L'entreprise a stoppé son activité en 2007.

3.8 Logiciels et plate-formes

- Membre du comité de pilotage de la plate-forme Grimage (depuis 2003), http://www.inrialpes.fr/grimage. Cette plate-forme expérimentale localisée à l'INRIA Rhône-Alpes, associe une grappe de PC, un réseau de caméras et un mur d'images multi-projecteurs. Cette plate-forme est issue d'une collaboration entre les équipe-projets MOAIS, PERCEPTION et EVASION. Les sources de financement sont variées (locales, nationales et européennes).

- Membre du comité de pilotage de la plate-forme Digitalis de l'INRIA Rhône-Alpes. Digitalis est une machine parallèle de 2048 coeurs pour le calcul en ligne et hors ligne. L'installation est prévue en deux tranches (2008 et 2009).

- Suite FlowVR (102 000 lignes de code, http://flowvr.sourceforge.net), première version Décembre 2003, plus de 800 téléchargements. Dépôt APP IDDN.FR.001.400021.000.S.P.2008.000.10000 (FlowVR), APP IDDN.FR.001.410004.000.S.P.2008.000.10000 (FlowVR Render) et IDDN.FR.001.390033.000.S.P.2008.000.10000 (VTK FlowVR). Licence GPL et LGPL. FlowVR est un intergiciel pour les applications parallèles interactives. Co-auteurs principaux : Jérémie Allard, Clément Ménier, Bruno Raffin, Jean-Denis Lesage, Emmanuel Melin, Valérie Gouranton, Sophie Robert et Sébastien Limet.

- Calibration (19 000 lignes de code), première version Juin 2004. Calibration automatique de la géométrie et de la luminosité d'un mur d'image. Tentative de commercialisation via la société Icatis en 2004. Dépôt APP IDDN.FR.001.450012.000.S.P.2004.000.21000. Co-auteurs : Sofia Zaidenberg, Frederic Devernay et Bruno Raffin.

- MVREALTIME (36 000 lignes de code), première version 2004. Modélisation 3D temps-réel. Transféré à la société 4D View Solutions. Dépôt APP IDDN.FR.001.190020.000.S.P.2007.000.10000. Co-auteurs principaux : Clément Ménier, Florian Geffray, Jérémie Allard, Bruno Raffin et Edmond Boyer.

- Net Juggler (82 000 lignes de code, http://netjuggler.sourceforge.net, première version Juin 2001, près de 1000 téléchargements. Net Juggler permet d'exécuter des applications VR Juggler sur grappe de PC. Licence LGPL. Co-auteurs principaux : Jérémie Allard, Bruno Raffin, Emmanuel Melin and Valérie Gouranton.

- SoftGenLock (6 000 lignes de code, http://netjuggler.sourceforge.net, première version Juin 2001, téléchargé près de 700 fois). Stéréoscopie active en environnement multi-projecteurs piloté par grappe de PC. Licence LGPL. Co-auteurs principaux : Jérémie Allard, Bruno Raffin, Emmanuel Melin et Valérie Gouranton.

3.9 Participation à la communauté académique

- Membre du comité de piloage de Eurographics Symposium on Parallel Rendering and Visualization depuis 2007.
- Co-éditeur du numéro spécial Parallel Graphics and Visualization, Parallel Computing, Volume 33, Issue 6, 2007.
- Co-éditeur du numéro spécial Parallel Graphics and Visualization, Parallel Computing, Volume 31, Issue 2, 2005.
- Co-responsable du Eurographics Symposium on Parallel Rendering and Visualization 2006, mai 2006, Braga, Portugal.
- Co-responsable et organisateur local du Eurographics Symposium on Parallel Rendering and Visualization 2004, 10 et 11 Juin 2004, Grenoble.
- Co-responsable et organisateur du Workshop on Commodity Clusters for Virtual Reality, IEEE VR, 22 Mars 2003, Los Angeles.
- Co-responsable des tutoriaux et membre du comité de programme de IEEE VR 2009, Lafayette,USA.
- Membre du comité de programme du CLCAR 2009 (Conferencia Latinamericana de Computatición de Alto Rendimiento), Venezuela.
- Membre du comité de programme du EGPGV 2009 (Eurographics Symposium on Parallel Graphics and Visualization), Munich, Germany.
- Membre du comité de programme du PAPP 2009 (Sixth International Workshop on aPpplications of declArative and object-oriented Parallel Programming), Baton Rouge, USA.
- Membre du comité de programme du 4th International Symposium on Visual Computing (ISVC08), Las Vegas, USA.
- Membre du comité de programme de ACM VRST 2008, Bordeaux, France.
- Membre du comité de programme de IEEE VR 2008 (Virtual Reality), Reno, USA.

- Membre du comité de programme de EGPGV 2008 (Eurographics Symposium on Parallel Graphics and Visualization), Crète, Grèce.
- Membre du comité de programme de SVR 2008 (Symposium on Virtual and Augmented Reality), João Pessoa, Brazil.
- Participe régulièrement à la relecture d'articles pour des revues et conférences: PEMCS'99, HLPP'01,02 and 03, EG'2003, RENPAR'03, SPAA'02, Presence 2003, EGPGV'04, EGVE'04, HPCSE'04, PCS'04, OPODIS'04, Europar'98, 02,05 and 06, IEEE VR'05, Parallel Computing Journal, TSI, IEEE Vis 2008, PAPP 07 and 08, relecteur de livre Elsevier.

3.10 Cours, tutoriaux and présentations invitées

[1] Jérémie ALlard, Clément Ménier, Bruno Raffin, and François Faure. Grimage: Markerless 3D Interactions. Game Developers Conference, Lyon, 2007.

[2] Bruno Raffin. High Performance Virtual Reality. Universidad da Coruna, España, 2007. Invited Seminar.

[3] Bruno Raffin. Adaptive Algorithms for new Parallel Supports. First International Summer School on Emerging Trends in Concurrency (TiC'06), Bertinoro, Italia, 2006.

[4] Bruno Raffin. Componentes estandard para muros de imagenes de alta resolución y gran tamaño. Jornadas Internacionales de Ciencias Computacionales, Colima, Mexico, 2006.

[5] Jérémie Allard, Marcio C. Cabral, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 03, Course 18*, California, July 2003.

[6] Pilippe Augerat, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 02, Course 47*, Texas, July 2002.

[7] C. Cruz-Neira, C. Just, K. Meinert, A. Bierbaum, P. Hartling, and B. Raffin. Open Source Virtual Reality. IEEE VR 2002 Tutorial, Florida, March 2002.

[8] M. Knorich-Zuffo, B. Schaeffer, C. Cruz-Neira, B. Raffin, and R. Blach. PC Clusters for Multiprojection Immersive Environments: Time to Go? Immersive Projection Technology (IPT) 2002, Florida, March 2002. Panel discussion.

[9] Bruno Raffin. Des grappes de PC pour la réalité virtuelle. Imagin@.02, Monaco, February 2002. Invited Speaker.

3.11 Publications

3.11.1 Calcul interactif haute performance

Journaux internationaux

[1] Jean-Denis Lesage and Bruno Raffin. A Hierarchical Component Model for Large Parallel Interactive Applications. *Journal of Supercomputing*, July 2008. Extended version of NPC 2007 article.

[2] Luciano P. Soares, Bruno Raffin, and Joaquim A. Jorge. PC Clusters for Virtual Reality. *The International Journal of Virtual Reality*, 7(1):67–80, March 2008. Extended Version of IEEE VR 20006 survey.

Conférences et ateliers internationaux

[3] *Grimage: 3D Modeling for Remote Collaboration and Telepresence*, Bordeaux, France, October 2008.

[4] Everton Hermann, François Faure, and Bruno Raffin. Ray-traced Collision Detection for Deformable Bodies. In *3rd International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 293–299, Madeira, Portugal, January 2008.

[5] Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. A Fast Cache Oblivious Mesh Layout with Theoretical Guarantees. In *1st International Workshop on Super Visualization (IWSV08)*, Kos, Grece, June 2008.

[6] Jean-Denis Lesage and Bruno Raffin. High Performance Interactive Computing with FlowVR. In *IEEE VR 2008 SEARIS workshop*, pages 13–16, Reno, USA, March 2008. Shaker Verlag.

[7] Jean-Denis Lesage and Bruno Raffin. A Hierarchical Programming Model for Large Parallel Interactive Applications. In *IFIP International Conference on Network and Parallel Computing*, volume 4672 of *Lecture Notes in Computer Science*, pages 516–525, Dalian, China, September 2007. Springer. Excellent Student Paper Award.

[8] Jérémie Allard, Clément Ménier, Bruno Raffin, Edmond Boyer, and François Faure. Grimage: Markerless 3D Interactions. In *Proceedings of ACM SIGGRAPH 07*, San Diego, USA, August 2007. Emerging Technologies.

[9] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch. Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling. In *Eurographics 2008 Symposium on Parallel Graphics and Visualization (EGPGV'08)*, pages 273–274, Lugano,Switzerland, May 2007.

[10] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis. Roch. Parallel Adaptive Octree Carving for Real-time 3D Modeling. In *IEEE Virtual Reality Conference*, Charlotte, USA, March 2007. Poster.

[11] Clément Ménier, Edmond Boyer, and Bruno Raffin. 3D Skeleton-Based Body Pose Recovery. In *International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'06)*, pages 389–396, 2006.

[12] Van Dat Cung, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive algorithms : theory and application. In *SIAM Parallel Processing*, San Francisco, February 2006.

[13] Van Dat Cung, Vincent Danjean, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving. In *Transgressive Computing*, Grenada, April 2006.

[14] Bruno Raffin and Luciano Soares. PC Clusters for Virtual Reality. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[15] Jérémie Allard and Bruno Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[16] Jérémie Allard, Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. The GrImage Platform: A Mixed Reality Environment for Interactions. In *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*, pages 46–52, New York, January 2006.

[17] Jérémie Allard, Clément Ménier, Edmond Boyer, and Bruno Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In *IPT & EGVE Workshop 2005*, Denmark, October 2005.

[18] Jérémie Allard and Bruno Raffin. A Shader-Based Parallel Rendering Framework. In *IEEE Visualization Conference*, pages 127–134, Minneapolis, USA, October 2005.

[19] Jérémie Allard, Edmond Boyer, Jean-Sébastien Franco, Clément Ménier, and Bruno Raffin. Marker-less Real Time 3D Modeling for Virtual Reality.

In *Immersive Projection Technology Symposium (IPT'04)*, Ames, Iowa, May 2004.

[20] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Sébastien Limet, Emmanuel Melin, Bruno Raffin, and Sophie Robert. FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 497–505, Pisa, Italia, August 2004.

[21] Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. A Distributed Approach for Real Time 3D Modeling. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPRW) 2004*, pages 31–38, Washington, USA, July 2004.

[22] Jérémie Allard, Bruno Raffin, and Florence Zara. Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster. In *Euro-par 2003*, Klagenfurt, Austria, August 2003.

[23] Jérémie Allard, Valérie Gouranton, Gilles Lamarque, Emmanuel Melin, and Bruno Raffin. Softgenlock: Active Stereo and Genlock for PC Cluster. In *IPT & EGVE Workshop 2003*, pages 255–260, Zurich, Switzerland, May 2003.

[24] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Emmanuel Melin, and Bruno Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference*, pages 275–276, Orlando, USA, March 2002.

[25] Jérémie Allard, Valérie Gouranton, Emmanuel Melin, and Bruno Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium (IPT)*, Orlando, USA, March 2002.

3.11.2 Mesure de performance

Journaux électroniques et atelier internationaux

[26] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a Linux and an NT Cluster of PCs, a SGI Origin 2000, an IBM SP and a Cray T3E-600. *The Journal of Performance Evaluation and Modelling for Computer Systems*, March 2000. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[27] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. The Performance of the MPI Collective Communication Routines for Large Messages on the Cray T3E-600, the Cray Origin 2000, and the IBM SP. *The Journal of*

Performance Evaluation and Modelling for Computer Systems, July 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[28] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a SGI Origin 2000, a cluster of Origin 2000's and a Cray T3E-1200 using SHMEM and MPI Routines. *The Journal of Performance Evaluation and Modelling for Computer Systems*, October 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[29] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600. In *Proceedings of the IEEE International Workshop on Cluster Computing (IWCC'99)*, pages 26–35, Melbourne, Australia, December 1999.

[30] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Scalability of the Cray T3E-600 and the Cray Origin 2000 Using SHMEM Routines. *The Journal of Performance Evaluation and Modelling for Computer Systems*, December 1998. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

3.11.3 Programmation parallèle

Journaux internationaux

[31] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50:3–27, 1998.

Conférences et ateliers internationaux

[32] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Cost Model For Asynchronous and Structured Massage Passing. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, volume 1685 of *LNCS*, pages 552–560. Springer-Verlag, 1999.

[33] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code. In *Third International Conference on Massively Parallel Computing Systems - MPCS'98*, pages 482–489, Colorado Springs, USA, April 1998. IEEE Computer Society Press.

[34] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, pages 189–196, London, United Kingdom, January 1997. IEEE Computer Society Press.

[35] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. SCLchan: An Asynchronous Data-Parallel Language for Irregular Algorithms. In *Second International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS'97 (in conjonction with 11th International Parallel Processing Symposium - IPPS'97)*, Geneva, Switzerland, April 1997. IEEE Computer Society Press.

[36] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Implementation on Cray T3D/T3E of SCLchan, a Programming Language Unifying Data and Task Parallelism. In *Third European CRAY-SGI MPP Workshop*, Paris, September 1997.

[37] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *LNCS*, pages 732–741. Springer-Verlag, 1996.

[38] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *Second International Conference on Massively Parallel Computing Systems - MPCS'96*, pages 482–489, Ischia, Italy, May 1996. IEEE Computer Society Press.

3.11.4 Réseaux de neurones

Journal international

[39] B. Raffin and M. B. Gordon. Minimerror, a temperature dependent learning rule. *Neural Computation*, 7(6):1206–1224, November 1995.

Atelier international et chapitre de livre Français

[40] Bruno Raffin and Bernard Virot. A Learning Rule Safe From Local Minima for a Generalized Perceptron. In P.G. Anderson and K. Warwick, editors, *Proceedings of the international ICSC Symposia IIA'96 and SOCO'96*, volume B, pages 223–229. ICSC Academic Press, March 1996.

[41] Bruno Raffin and Bernard Virot. Algorithmique neuronale. In G. Authié, J. Garcia, A. Ferriera, J. Rach, G. Villard, J. Roman, C. Roucairol, and B. Virot, editors, *Parallélisme et applications irrégulières*, pages 49–68. Hermes, Paris, France, 1995.

Chapter 4

Curriculum Vitae (English version)

Bruno RAFFIN
Born the 31th of December 1970, France
Married, one child

4.1 Employment History

- **Since October 2001.** INRIA Research Scientist at INRIA Rhône-Alpes Grenoble, France.
- **September 1999 - September 2001.** Assistant Professor at Université d'Orléans, France.
- **January 1998 - August 1999.** Postdoc fellow at Iowa Sate University, USA. Funded by a Cray-SGI grant. Teaching duty of 60 hours per semester.

4.2 Education

- **1993-1997.** Ph.D. in Computer Science, Université d'Orléans, France. *Title: A Structured Model of Communications and Synchronizations for Task Parallelism*. Advisers: Bernard Virot and Robert Azencott.
- **1993.** DEA d'Informatique Fondamentale (Diploma of Advanced Studies in computer sciences with specialization in parallelism). École Normale Supérieure de Lyon, France.

- **1991-1992** Master in Mathematics. Université Lyon I, France.

4.3 Teaching

- **2001-2008:** Parallel computing (Université d'Orléans, 8 hours per year, Ph.D. program), Introduction to parallel computing (Grenoble Université, 18 hours per year, Master program).
- **1999-2001: Université d'Orléans.** 230 hours per year. Computer architecture, operating systems, networking and internet technology, security, parallel computing. Master program.
- **1998-1999: Iowa Sate University.** 180 hours. Math class. Undergraduate program.
- **1994-1997: Université d'Orléans.** 96 hours per year. Algorithms, Turbo Pascal programming and mathematics for computer science. Undergraduate program.

4.4 Research Advising

4.4.1 Ph.D. Students

- **Marc Thiboukdjian (since 2007)**. Parallel and cache-oblivious algorithms for 3D mesh layouts. Co-advised with Vincent Danjean and Jean-Louis Roch.
- **Benjamin Petit (since 2007)**. Interaction with multi-camera environments. Co-advised with Edmond Boyer.
- **Everton Hermann (since 2006)**. Parallel algorithms for the SOFA physics simulation framework. Co-advised with François Faure.
- **Jean-Denis Lesage (since 2006)**. Adaptive algorithms for large interactive 3D applications.
- **Thomas Arcila (2005)**. High performance sort-first rendering. Stopped after one year. Engineer at Mercury Computer, Bordeaux, France.
- **Clément Ménier (2003-2007)**. Computer system for real-time 3D modeling. Co-advised with Edmond Boyer. Co-founder and chief software architect of the 4D Views company, Grenoble, France.
- **Jérémie Allard (2002-2005)**. Middleware for large virtual reality applications running on PC clusters. INRIA research scientist at Lille, France.
- **Jesus Verduzco (2001-2005)**. X Windows environment for display-wall. Associate Professor at the Universidad de Colima, Mexico.

4.4.2 Postdoc

- **Luciano Soares (2006)**. Postdoc. Adaptive parallel octree. Research scientist at Petrobras, Brazil.

4.4.3 Engineers

- **Thomas Dupeux (since 2008).** Grimage platform support.
- **Antoine Vanel (since 2008)**. FlowVR development.
- **Loick Lecointre (2003-2004)**. FlowVR development. Engineer at Amadeus, Paris.

4.4.4 Internships

- 28 internships since 2000 (graduate level).

4.5 Administrative Responsibilities

- President of the INRIA Rhône-Alpes Technological Development Committee (since 2007). In charge of evaluating proposals for the attribution of engineer grants.
- Member of the INRIA Evaluation Committee (2005-2008)
- Member of the CS (associate and assistant professor recruiting committee) for the Université Joseph Fourier (2004-2008).
- Member of the CS for the Université d'Orléans (2008).
- Grant manager for the long term collaboration with the Universities of Rio Grande do Sul, Brazil. Grants:
  - PICS CNRS (2005-2007).
  - Capes/Cofecub (2006-2008).
  - INRIA associate team Diode-A (2006-2008).
  - INRIA/Cnpq (2008-2010).

4.6 Grants

- **European Project Interact (2007-2008).** Partners: Eptron, Holographika, Total Immersion, Vecsys SA and INRIA Rhône-Alpes. 3D modeling and speech interfacing technologies.

- **ANR (French national research agency) grant Vulcain (2008-20010)**. Partners: Université Joseph Fourier, INERIS, Université de Bourges, SME, PHIMECA, CEA, BULL SAS, Université de Marne la Vallée, INRIA Rhône-Alpes.

- **Ph.D. Grant co-funded by CNRS and CEA (2007-2010)**.

- **ANR grant DALIA (2007-2009)**. National Coordinator. Partners: Université d'Orléans, Université Paris 7, CEA and INRIA Rhône-Alpes. Interactive grid, collaborative interaction and telepresence.

- **ANR grant FVNANO (2008-2010).** Coordinator for the INRIA Rhône-Alpes. Partners: Université d'Orléans, Université de Bordeaux and INRIA Rhône-Alpes. Interactive simulations of nano structures on PC cluster.

- **ANR grant OCETRE (2004-2005).** Partners: Thalès, Total Immersion and INRIA Rhône-Alpes. Real-time multi-camera motion capture.

- **ANR grant GEOBENCH (2003-2004).** Coordinator for the INRIA Rhône-Alpes. Partners: Université d'Orléans, Mercury Computer, CEA, BRGM, INRIA Rhône-Alpes. Virtual reality for scientific visualization.

- **ANR grant CYBER II (2003-2005)**. Partners: Université Lyon I, INRIA Rhône-Alpes. Real-time 3D modeling.

- **Ph.D. Grant funded by the Bull SAS (2005)**.

4.7 Technology Transfer

- Real-time 3D modeling code licensed to the 4DViews company co-founded by a former Ph.D. Student (2007).

- Co-founder of the Icatis Start-up (2003). Consultant for the company from 2003 to 2006.

4.8 Software and Platforms

- Member of the steering board of the Grimage platform (since 2003). Experimental platform located at INRIA Rhône-Alpes. It gathers a PC cluster, a camera network and a display wall. This platform is shared by 3 different INRIA teams (PERCEPTION, EVASION and MOAIS) and funded by various grants (locals, nationals and Europeans).

- Member of the steering board of the Digitalis project. Digitalis is a future cluster of 2048 computing cores that will be installed at INRIA Rhône-Alpes in 2009.

- FlowVR Suite (102 000 code lines, http://flowvr.sourceforge.net. First public release: 2003. More than 800 downloads. FlowVR is a middleware for parallel interactive applications.
- MVREALTIME (36 000 code lines). First version: 2004. Real-time 3D modeling. Code transfered to the 4DView Solutions company.
- Calibration (19 000 code lines). 2004. Software calibration of multi-projector display walls. Not publicly available.
- Net Juggler (82 000 code lines). 2001-2004. More than 1000 downloads. Cluster support for VR Juggler applications.
- SoftGenLock (6 000 code lines). 2001-2004. More than 700 downloads. Software genlocking of commodity graphics cards for active stereo rendering.

4.9 Participation to the Academic Community

- Member of the steering board of the Eurographics Symposium on Parallel Rendering and Visualization since 2007.
- Co-editor special issue "Parallel Graphics and Visualization", Parallel Computing, Volume 33, Issue 6, 2007.
- Co-editor special issue "Parallel Graphics and Visualization", Parallel Computing, Volume 31, Issue 2, 2005.
- Co-chair of Eurographics Symposium on Parallel Rendering and Visualization, May 2006, Braga, Portugal.
- Co-chair and local organizer of Eurographics Symposium on Parallel Rendering and Visualization, June 2004, Grenoble, France.
- Co-chair and organizer of the Workshop on Commodity Clusters for Virtual Reality, IEEE VR, 22 March 2003, Los Angeles, USA.
- Tutorial co-chair and programm committee of IEEE VR 2009, Lafayette, USA.
- Program committee of CLCAR 2009 (Conferencia Latinamericana de Computatición de Alto Rendimiento), Venezuela.
- Program committee of EGPGV 2009 (Eurographics Symposium on Parallel Graphics and Visualization), Munich, Germany.
- Program committee of PAPP 2009 (Sixth International Workshop on aPpplications of declArative and object-oriented Parallel Programming), Baton Rouge, USA.

- Program committee of ISVC08 (4th International Symposium on Visual Computing), Las Vegas, USA.
- Program committee of ACM VRST 2008, Bordeaux, France.
- Program committee of IEEE VR 2008 (Virtual Reality), Reno, USA.
- Program committee of EGPGV 2008 (Eurographics Symposium on Parallel Graphics and Visualization), Crete, Greece.
- Program committee of SVR 2008 (Symposium on Virtual and Augmented Reality), João Pessoa, Brazil.
- Reviewer for several conferences and journals: PEMCS'99, HLPP'01,02 and 03, EG'2003, RENPAR'03, SPAA'02, Presence 2003, EGPGV'04, EGVE'04, HPCSE'04, PCS'04, OPODIS'04, Europar'98, 02,05 and 06, IEEE VR'05, Parallel Computing Journal, TSI, IEEE Vis 2008, PAPP 07 and 08, book reviewer for Elsevier.

4.10 Courses, Tutorials and Invited Talks

[1] Jérémie ALlard, Clément Ménier, Bruno Raffin, and François Faure. Grimage: Markerless 3D Interactions. Game Developers Conference, Lyon, 2007.

[2] Bruno Raffin. High Performance Virtual Reality. Universidad da Coruna, España, 2007. Invited Seminar.

[3] Bruno Raffin. Adaptive Algorithms for new Parallel Supports. First International Summer School on Emerging Trends in Concurrency (TiC'06), Bertinoro, Italia, 2006.

[4] Bruno Raffin. Componentes estandard para muros de imagenes de alta resolución y gran tamaño. Jornadas Internacionales de Ciencias Computacionales, Colima, Mexico, 2006.

[5] Jérémie Allard, Marcio C. Cabral, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 03, Course 18*, California, July 2003.

[6] Pilippe Augerat, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 02, Course 47*, Texas, July 2002.

[7] C. Cruz-Neira, C. Just, K. Meinert, A. Bierbaum, P. Hartling, and B. Raffin. Open Source Virtual Reality. IEEE VR 2002 Tutorial, Florida, March 2002.

[8] M. Knorich-Zuffo, B. Schaeffer, C. Cruz-Neira, B. Raffin, and R. Blach. PC Clusters for Multiprojection Immersive Environments: Time to Go? Immersive Projection Technology (IPT) 2002, Florida, March 2002. Panel discussion.

[9] Bruno Raffin. Des grappes de PC pour la réalité virtuelle. Imagin@.02, Monaco, February 2002. Invited Speaker.

4.11 Publications

4.11.1 High Performance Interactive Computing

International Journals

[1] Jean-Denis Lesage and Bruno Raffin. A Hierarchical Component Model for Large Parallel Interactive Applications. *Journal of Supercomputing*, July 2008. Extended version of NPC 2007 article.

[2] Luciano P. Soares, Bruno Raffin, and Joaquim A. Jorge. PC Clusters for Virtual Reality. *The International Journal of Virtual Reality*, 7(1):67–80, March 2008. Extended Version of IEEE VR 20006 survey.

International Conferences and Workshops

[3] *Grimage: 3D Modeling for Remote Collaboration and Telepresence*, Bordeaux, France, October 2008.

[4] Everton Hermann, François Faure, and Bruno Raffin. Ray-traced Collision Detection for Deformable Bodies. In *3rd International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 293–299, Madeira, Portugal, January 2008.

[5] Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. A Fast Cache Oblivious Mesh Layout with Theoretical Guarantees. In *1st International Workshop on Super Visualization (IWSV08)*, Kos, Grece, June 2008.

[6] Jean-Denis Lesage and Bruno Raffin. High Performance Interactive Computing with FlowVR. In *IEEE VR 2008 SEARIS workshop*, pages 13–16, Reno, USA, March 2008. Shaker Verlag.

[7] Jean-Denis Lesage and Bruno Raffin. A Hierarchical Programming Model for Large Parallel Interactive Applications. In *IFIP International Conference on Network and Parallel Computing*, volume 4672 of *Lecture Notes in Computer Science*, pages 516–525, Dalian, China, September 2007. Springer. Excellent Student Paper Award.

[8] Jérémie Allard, Clément Ménier, Bruno Raffin, Edmond Boyer, and François Faure. Grimage: Markerless 3D Interactions. In *Proceedings of ACM SIGGRAPH 07*, San Diego, USA, August 2007. Emerging Technologies.

[9] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch. Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling. In *Eurographics 2008 Symposium on Parallel Graphics and Visualization (EGPGV'08)*, pages 273–274, Lugano,Switzerland, May 2007.

[10] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis. Roch. Parallel Adaptive Octree Carving for Real-time 3D Modeling. In *IEEE Virtual Reality Conference*, Charlotte, USA, March 2007. Poster.

[11] Clément Ménier, Edmond Boyer, and Bruno Raffin. 3D Skeleton-Based Body Pose Recovery. In *International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'06)*, pages 389–396, 2006.

[12] Van Dat Cung, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive algorithms : theory and application. In *SIAM Parallel Processing*, San Francisco, February 2006.

[13] Van Dat Cung, Vincent Danjean, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving. In *Transgressive Computing*, Grenada, April 2006.

[14] Bruno Raffin and Luciano Soares. PC Clusters for Virtual Reality. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[15] Jérémie Allard and Bruno Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[16] Jérémie Allard, Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. The GrImage Platform: A Mixed Reality Environment for Interactions. In *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*, pages 46–52, New York, January 2006.

[17] Jérémie Allard, Clément Ménier, Edmond Boyer, and Bruno Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In *IPT & EGVE Workshop 2005*, Denmark, October 2005.

[18] Jérémie Allard and Bruno Raffin. A Shader-Based Parallel Rendering Framework. In *IEEE Visualization Conference*, pages 127–134, Minneapolis, USA, October 2005.

[19] Jérémie Allard, Edmond Boyer, Jean-Sébastien Franco, Clément Ménier, and Bruno Raffin. Marker-less Real Time 3D Modeling for Virtual Reality. In *Immersive Projection Technology Symposium (IPT'04)*, Ames, Iowa, May 2004.

[20] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Sébastien Limet, Emmanuel Melin, Bruno Raffin, and Sophie Robert. FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 497–505, Pisa, Italia, August 2004.

[21] Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. A Distributed Approach for Real Time 3D Modeling. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPRW) 2004*, pages 31–38, Washington, USA, July 2004.

[22] Jérémie Allard, Bruno Raffin, and Florence Zara. Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster. In *Euro-par 2003*, Klagenfurt, Austria, August 2003.

[23] Jérémie Allard, Valérie Gouranton, Gilles Lamarque, Emmanuel Melin, and Bruno Raffin. Softgenlock: Active Stereo and Genlock for PC Cluster. In *IPT & EGVE Workshop 2003*, pages 255–260, Zurich, Switzerland, May 2003.

[24] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Emmanuel Melin, and Bruno Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference*, pages 275–276, Orlando, USA, March 2002.

[25] Jérémie Allard, Valérie Gouranton, Emmanuel Melin, and Bruno Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium (IPT)*, Orlando, USA, March 2002.

4.11.2 Performance Measure

International Electronic Journals and Workshop

[26] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a Linux and an NT Cluster of PCs, a SGI Origin 2000, an IBM SP and a Cray T3E-600. *The Journal of*

Performance Evaluation and Modelling for Computer Systems, March 2000. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[27] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. The Performance of the MPI Collective Communication Routines for Large Messages on the Cray T3E-600, the Cray Origin 2000, and the IBM SP. *The Journal of Performance Evaluation and Modelling for Computer Systems*, July 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[28] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a SGI Origin 2000, a cluster of Origin 2000's and a Cray T3E-1200 using SHMEM and MPI Routines. *The Journal of Performance Evaluation and Modelling for Computer Systems*, October 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[29] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600. In *Proceedings of the IEEE International Workshop on Cluster Computing (IWCC'99)*, pages 26–35, Melbourne, Australia, December 1999.

[30] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the Scalability of the Cray T3E-600 and the Cray Origin 2000 Using SHMEM Routines. *The Journal of Performance Evaluation and Modelling for Computer Systems*, December 1998. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

4.11.3 Parallel Programming

International Journals

[31] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50:3–27, 1998.

International Conferences and Workshops

[32] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Cost Model For Asynchronous and Structured Massage Passing. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, volume 1685 of *LNCS*, pages 552–560. Springer-Verlag, 1999.

[33] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code. In *Third International Conference on Massively Parallel Computing Systems - MPCS'98*, pages 482–489, Colorado Springs, USA, April 1998. IEEE Computer Society Press.

[34] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, pages 189–196, London, United Kingdom, January 1997. IEEE Computer Society Press.

[35] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. SCLchan: An Asynchronous Data-Parallel Language for Irregular Algorithms. In *Second International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS'97 (in conjonction with 11th International Parallel Processing Symposium - IPPS'97)*, Geneva, Switzerland, April 1997. IEEE Computer Society Press.

[36] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Implementation on Cray T3D/T3E of SCLchan, a Programming Language Unifying Data and Task Parallelism. In *Third European CRAY-SGI MPP Workshop*, Paris, September 1997.

[37] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *LNCS*, pages 732–741. Springer-Verlag, 1996.

[38] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *Second International Conference on Massively Parallel Computing Systems - MPCS'96*, pages 482–489, Ischia, Italy, May 1996. IEEE Computer Society Press.

4.11.4 Neural Networks

International Journal

[39] B. Raffin and M. B. Gordon. Minimerror, a temperature dependent learning rule. *Neural Computation*, 7(6):1206–1224, November 1995.

International Workshop and French Book Chapter

[40] Bruno Raffin and Bernard Virot. A Learning Rule Safe From Local Minima for a Generalized Perceptron. In P.G. Anderson and K. Warwick, editors, *Proceedings of the international ICSC Symposia IIA'96 and SOCO'96*, volume B, pages 223–229. ICSC Academic Press, March 1996.

[41] Bruno Raffin and Bernard Virot. Algorithmique neuronale. In G. Authié, J. Garcia, A. Ferriera, J. Rach, G. Villard, J. Roman, C. Roucairol, and B. Vi-

rot, editors, *Parallélisme et applications irrégulières*, pages 49–68. Hermes, Paris, France, 1995.

Chapter 5

High Performance Distributed Rendering

This chapter focuses on high performance rendering and multi-projector rendering in particular. This work led to three main publications [11, 9, 15], two courses at Siggraph 2002 and 2003 [20, 6], a tutorial at IEEE VR 2002 [33], and one survey initially published at IEEE VR [75] then extended to a journal version [85]. The work led to 3 main open source software developments: Net Juggler, SoftGenLock and FlowVR Render. Jérémie Allard's work, during his Master and next during his Ph.D. [4], and to some extent Jesus Verduzsco's Ph.D. work [89], were focused on this topic.

5.1 PC Clusters for Multi-projector Rendering

5.1.1 Context and Motivation

Multi-projector rendering was first motivated by the emergence of projector based immersive environments like the CAVE [34]. The CAVE is a cube shaped environment where one or several users can stand. Projectors back-project images on the different sides. The first CAVE used 4 projectors (front, left, right and floor sides), while today some 6-sided CAVEs use multiple projectors per side. The first CAVE was driven by a cluster of SGI machines, but the computing and graphics resources were classically provided by SGI Onyx machines [72]. It is only during the late 90's, early 00's, that commodity components, i.e PCs equipped with video game graphics cards, started to be considered as a possible alternative to these costly dedicated machines.

Clusters were required at first to overcome a hardware limitation: no PC could provide enough video outputs to drive a multi-projector environment. Clusters of workstations and later of PCs appeared in the mid 90's for high performance computing. They started to be used for virtual reality latter when the performance of commodity 3D graphics cards started to take-off. This was at the time of the

3DFX Voodoo cards, NVIDIA TNT and Geforce cards.

The classical configuration uses one graphics card per PC and one PC per projector. The issue is to distribute the computations on the different computing hosts and more importantly to keep coherent the images displayed by the different projectors, i.e. to obtain a seamless single image from the set of images displayed by the projectors. There are three levels of synchronization to ensure:

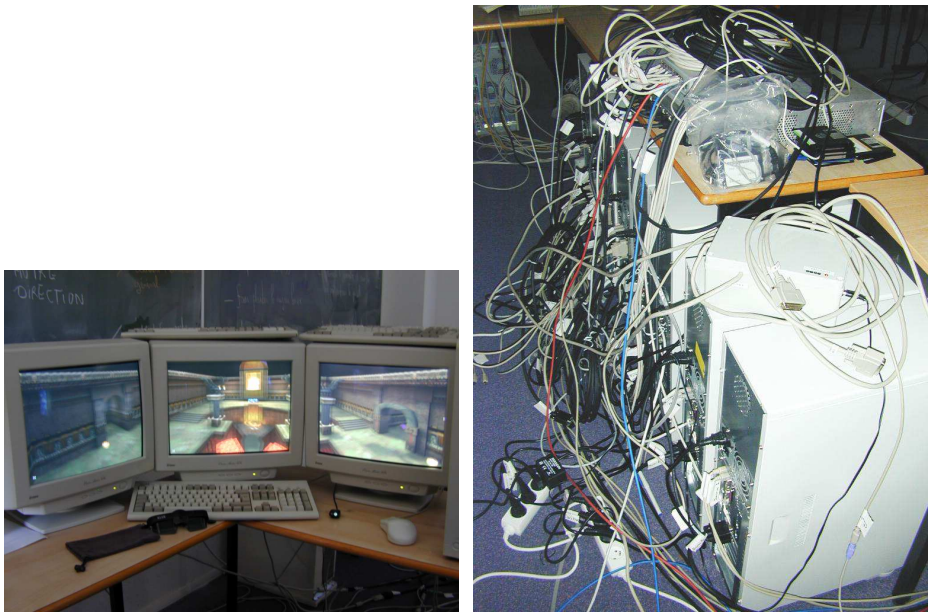
- Datalock: each host have access to coherent data to compute its local image
- Framelock: the hosts synchronize their frame buffer swap, i.e. the swap between the buffer read to generate the video signal and the buffer containing the new image to be displayed.
- Genlock: the hosts synchronize their video signals generation. Genlock is critical for active stereo rendering, when the video signal alternatively draw the right eye image and the left eye one: the user equipped with shutter glasses should only see a right eye image on each projector when the left eye shutter is on and vice-versa.

Some high-end graphics cards, like the 3Dlabs Wildcat, offered the framelock and genlock at a hardware level. This approach was for instance adopted to drive a CAVE at Urbana-champaign [78]. NVIDIA provided such feature latter (2003) with its high-end Quadro graphics cards. Our goal was to propose a solution that did not depend on the availability of such features. We favored a software approach for low-cost commodity based clusters.

The work we performed was centered on the Net Juggler and SoftGenLock software libraries. Net Juggler takes care of task distribution, datalock and framelock, while ensuring genlock requires a low level access to the graphics cards, a task SoftGenLock takes care of (Fig. 5.1).

5.1.2 Net Juggler Overview

Net Juggler is an extension of the VR Juggler library enabling to run any VR Juggler [25] application on a Cluster with minimal code modifications. Net Juggler ensures the task distribution, datalock and framelock. Task distribution is simple: all cluster hosts execute the same code but with a different and complementary viewing frustum. To ensure datalock, input events that can modify the data loaded at the starting time must be carefully broadcasted from the source devices to the various hosts. Usually the amount of data sent from these input devices is small, limiting the communication overhead. The main originality of Net Juggler is its decentralized architecture. Input event sources, including random number generators as well as timers, can be freely distributed on the various hosts. Each host stores in a continuous memory area the events to be considered for the next frame computation. Then, a communication layer takes care of broadcasting these massages to every other hosts, following a allgather collective communication for efficiency



(a) (b)

Figure 5.1: (a) Quake running on 3 screens, each one driven by a PC, using Net Juggler and SoftGenlock. (b) The Cluster with the SoftGenLock box on top of one PC.

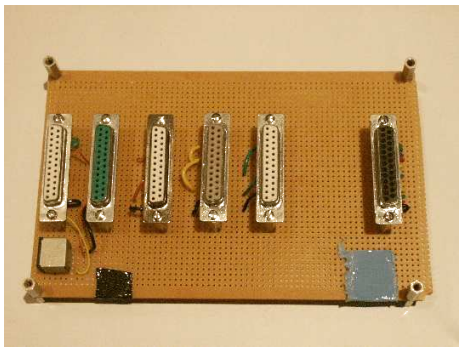


Figure 5.2: The simplest SoftGenLock board we developed. It broadcasts the sync signal from the master to the 5 slaves and the IR emitter of the shutter glasses

purpose. Framelock is a simple MPI synchronization barrier performed after flushing the OpenGL pipe-line and before the buffer swap. The user can not detect image discrepancies even with a simple 100 Mbits/s Ethernet network.

The speed-up this approach provides is high. Doubling the number of hosts enables to compute almost twice the number of pixels. Of course, the use of computing and memory resources is not optimal. Some computations and data are duplicated. But the very first performance issue for virtual reality is latency. This approach proved to induce a small extra latency the user usually does not perceive.

5.1.3 Net Juggler IEEE VR 2002 Article

For more details about Net Juggler, please refer to the article [11] included in section A.1 page 111.

5.1.4 SoftGenLock Overview

SoftGenLock proposes a software approach for video signal genlocking. It provides a synchronization at the v-sync level, i.e. it ensures video retrace starts on all graphics cards with only a few microseconds of delay. As the primary goal of SoftGenLock is to provide active stereo support, it also flips the memory page read (right/left eye image) at each retrace and send a sync signal to the shutter glasses to close the right shutter according to the image displayed.

One key component of SoftGenLock is to use the parallel port as a low latency broadcasting network (some microseconds). The parallel port is a very low level device, directly accessible from the OS kernel without requiring any driver. Having no driver reduces the overhead caused by handling interrupts and executing the driver code. It also allows direct access from a real time Linux kernel like RTLinux or RTAI, a feature used for the first version of SoftGenLock. An interesting side effect of using the parallel port is that it can directly power and control the shutter glasses IR emitter, as it produces the required TTL signal (Fig. 5.2).

We basically developed two versions of SoftGenLock. The first one was based on an active pooling performed by the slave hosts to detect when it receives the sync signal from the master. On Linux Kernel 2.4, the overhead was excessive. It impaired the proper execution of the application. To avoid this issue, we used a real-time Linux kernel to wake up the task a few microseconds before the expected synchronization signal. Once the signal received, SoftGenLock gets the video signal position reading the CRTC registers available on VGA compliant graphics cards. If video blanking occurs before or after the expected synchronization signal, small unnoticeable modifications of the video signals are applied by removing or adding hidden pixels.

To get the video card to retrace alternatively the right and left image, we build a quad-buffering system from the standard double buffering one. The X server is configured to have a display size twice as large as the actual display resolution. The application is requested to draw the left and right images side by side. At each video blanking SoftGenLock changes the starting address of the image to retrace.

The first version of SoftGenLock suffered from instabilities. From time to time, SoftGenLock crashed. It was due to some concurrency access to the CRTC registers. We then developed a second version to solve this issue. We relied on NVIDIA specific registers to control the pixel clock instead of writing the CRTC register to add or remove pixels. We also used interrupts, one generated when receiving the synchronization signal and one when the video blanking occurred, to get ride of pooling.

5.1.5 SoftGenLock IPT 2003 Article

The article [9], included in section A.2 page 120, presents in details SoftGenLock. It is based on SoftGenLock I but the concepts still apply to SoftGenLock II.

5.1.6 Discussion

This initial work enabled us to make our first steps into the computer graphics and virtual reality communities. To our knowledge we were the first to be able to drive an active stereo multi-projector environment with a commodity cluster. This work was rather technological, but answered to a real need at the time.

Net Juggler and/or SoftGenLock were used or tested by several labs (a few we know: NASA Johnson Space Center - USA, HRL Laboratories - USA, Argonne Futures Lab - USA, Iowa State University - USA, University of Kentucky - USA, VRlab - Sweden, V-lab - Italy, University of Western Sydney - Australia, Ars Electronica - Austria). Beside some citations in classical academic publications, these tools were also at the base of a Linux Journal article in 2002 [68].

We collaborated for a time with the VRAC, Iowa State University. The VRAC developed VR Juggler. We collaborated to provide cluster support directly in the

initial VR Juggler distribution. The team decided to take a different approach relying on a more classical client/server model [26]. One of the main reason was to get ride of the MPI dependency. Once this version was available, we quit supporting Net Juggler, as it became easier for users to rely on the integrated VR Juggler distribution.

We know two genlocking software libraries that adopted some of the concepts developed for SoftGenLock, a windows Version developed at ETZH, Switzerland [32], and a Linux version from HLRS, Germany [92]. This last version takes advantage of the more reactive scheduler of the Linux Kernel 2.6 to get ride of the real time kernel we had to use.

Today, driving multi-projector environments with a PC cluster is the norm. Several open-source and commercial solutions exist. The software architecture can vary but the approach is alway the same: intercept and distribute input events to the copies of the application running on each node and computing images for its own viewing frustum. Cluster support is transparent to the application developer. Datalock is properly ensured as long as the application developer does not intend to use event generators that can not be intercepted (this is a common error in particular for timers and random number generators).

The offer of genlock ready graphics cards is still very reduced (NVIDIA Quadro graphics cards). Software genlocking is thus still an attractive solution. But today immersive systems tend to turn away from active stereo, the main reason why genlocking is required. First, CRT projectors almost disappeared today. They are sometimes replaced by high-end DLP stereo-enabled projectors. But the majority of immersive systems tend to use passive stereo, based on polarization or interference filters (Infitec), where each projector is dedicated to one eye view only. This is usually a cheaper solution that can use commodity projectors. Passive stereo quality also improved significantly in particular with the Infitec solution.

For a large range of VR applications, the approach presented answers to the user needs. But some large applications may require to involve more computing hosts beside the ones attached to video projectors. This is typically the case when coupling a parallel code with a VR application for scientific visualization or to embed a complex simulation requiring a parallelization to be interactive. We explored such approach with Net Juggler by coupling a 2D fluid simulation written with MPI [12] and a cloth simulation parallelized with ATHAPASCAN [17].

This work highlighted the limitation of the approach in this situation. A VR application, like one we could write with VR Juggler, tends to be monolithic in the sense that it is built around a single loop that:

- updates the input states,
- updates the virtual scene,
- publishes the new state of the scene for various renderings (image, sound, force feedback).

This sequence imposes the different steps to be synchronous, i.e. to have the same refresh rate. Of course, it is possible to circumvent this loop. This is classically the case for instance when reading the state of input devices. Each input device is usually managed in an external thread or process. It asynchronously updates a buffer that the application reads when needed. A double buffering mechanism ensures a safe concurrent read/write access. The main loop stays anyway the central coordination point of the application. It impairs the ability to design and execute an application taking advantage of numerous distributed resources. We switched to a different approach, borrowed from scientific visualization tools like Iris Explorer [40]. The application is seen as a set of tasks, each one executing a local loop, synchronized through the flow of data. But, in opposite to scientific visualization tools, parallel execution support and synchronization policy control were top level requirements. This approach led to FlowVR (Chapter 6 page 51).

5.2 FlowVR Render

5.2.1 Motivation

As discussed before in section 5.1.6, moving to large VR applications requires to better rationalize the role of each processing step. This means separating the rendering steps from the other ones, possibly executing this task on a different machine or cluster. We saw before how multi-display rendering could be achieved. We now consider the complementary situation, where multiple distributed tasks concurrently provide data for rendering. The goal is to enable any task, be an input device or a physics simulation for instance, to directly provide 3D data to the tasks in charge of rendering the scene.

Gathering graphical data can be achieved at the pixel level. This sort-last approach [71] is commonly adopted for rendering large data sets in scientific visualization applications. The data set is partitioned between different hosts. Each hosts perform a local rendering on its own data. Partial images are them composited in a reduction phase [64] to obtain the final image. This approach requires each task to have rendering capabilities, which is not the situation we are targeting here.

Another approach for distributed rendering consists in intercepting the calls to a graphics library, typically OpenGL, to encapsulate the data into messages that are then broadcasted to distant hosts for rendering. This approach was first developed for rendering unmodified OpenGL applications on a display-wall. The Chromium library [53, 54] implements multiple optimizations to improve performance. For instance, it computes bounding boxes around objects to route them only towards the displays where they are to be rendered. But OpenGL makes it complex to merge data streams produced by multiple hosts. As OpenGL is based on a sequential state machine, commands must respect a strict ordering. Merging multiple streams together requires Chromium to track state changes and to use special commands defining the relative ordering of each stream.

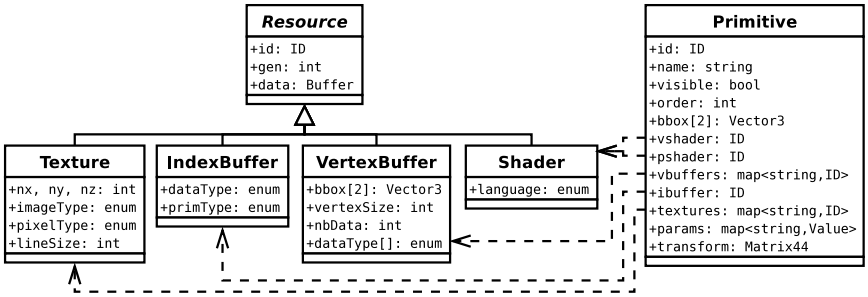


Figure 5.3: UML diagram of the objects used to describe a 3D scene with FlowVR Render.

5.2.2 Overview

FlowVR Render proposes to rely on recent OpenGL extensions (object buffers and shaders) to build a protocol that makes it simple and efficient to transport and merge graphics primitives.

A scene is described as a set of independent primitives. Each primitive contains the description of a batch of polygons and their rendering parameters (Fig. 5.3). To reduce the number of these parameters, as well as to take advantage of the advanced features of recent graphics cards, we use shaders to define each primitive's appearance. Large resources such as textures or vertex attributes are often used by several primitives in the scene. To avoid duplicating these resources in each primitive, we define *resources*. A resource can encapsulate a shader, texture, vertex buffer or index buffer. It is identified by an unique ID. Each primitive refers to the IDs of all the resources it uses. Notice that it introduces a one-way dependency between primitives and resources, but not between primitives.

Each primitive also stores its bounding box. This information is required to optimize communications using frustum culling. It is useful to specify this information in the primitive as it is costly to recover from the other parameters (especially when using shaders that can change vertex positions). Some rendering algorithms require primitives to be processed in a given order. FlowVR Render provides a mechanism to enforce this ordering by associating each primitive with an order value. This number defines a partial ordering between primitives. Primitives with the same value can be rendered in any order. Different values will mainly be used when rendering order can affect the final image like for transparent primitives or user-interface overlays. For a given order value, primitives can be re-ordered to improve performance. For instance, primitives can be sorted front-to-back to take advantage of fast Z-culling, primitives with similar parameters such as textures and shaders can be gathered to reduce state switching overheads. This approach enables to easily implement performance optimizations compared to the strict ordering defined by an immediate-mode command stream.

We call *viewers* the tasks generating primitives and *renderers* the ones in charge

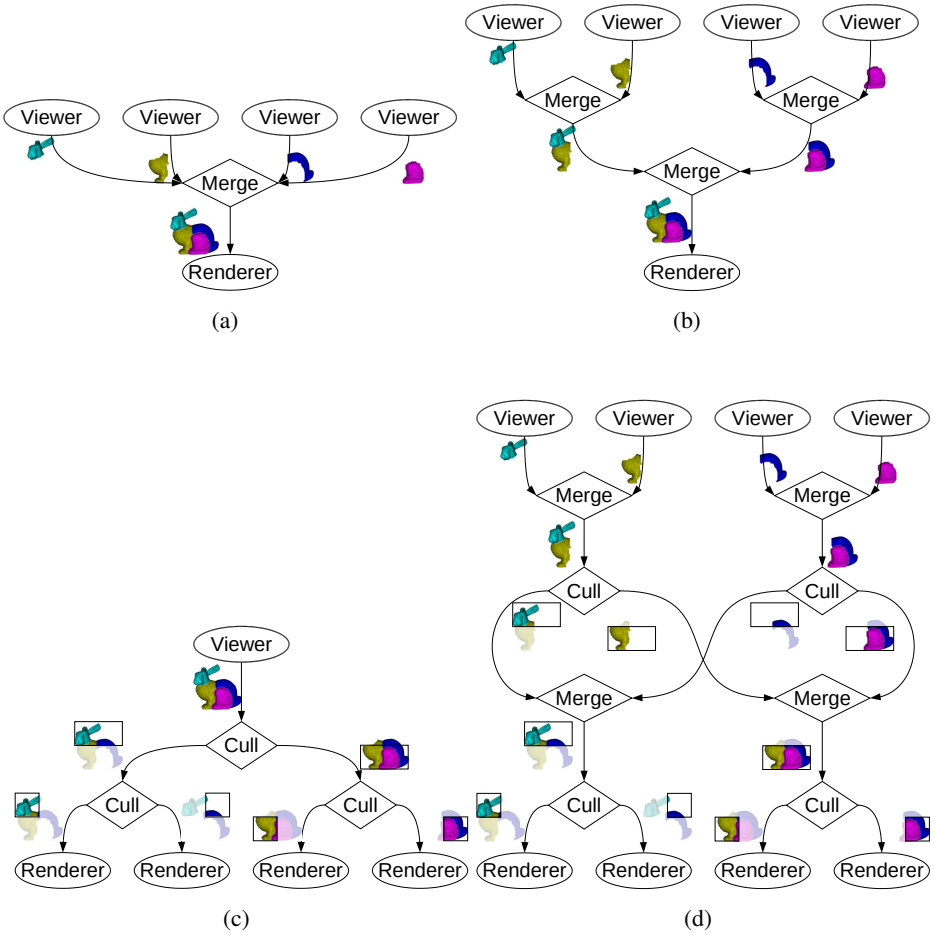


Figure 5.4: Different FlowVR Render rendering schemes

of rendering them. Performance tests show that FlowVR Render is significantly more scalable than Chromium regarding the number of viewers or renderers.

Combining viewers, renderers and simple message processing tasks, FlowVR Render offers a large range of rendering schemes. The classical one is to have several viewers sending primitives to one renderer (Fig. 5.4 (a)). Primitives are merged into a single message handled to the render on the same hosts. It enables to keep the renderer code simple and ignorant of the number of viewers. Rather than to merge at a central point, it can easily be done progressively using a binary tree structure (Fig. 5.4 (b)). One viewer can also provide data to multiple renders using a simple broadcast, or, using again a binary tree communication scheme and culling filters to route each primitive where it is required based on its bounding box (Fig. 5.4 (c)). These schemes can be combined when several viewers and

Figure 5.5: Rendering HDTV videos on a 16 projector display wall using Mplayer and FlowVR Render.

renderers are used (Fig. 5.4 (d)). On the Grimage platform[1] (16 projectors display-wall), we use a simple communication scheme where each viewer directly sends its primitives to each renderer. We did some preliminary tests with binary-tree based schemes, but performance appeared to be inferior. The benefits of the binary tree are lost in the overhead caused by the extra message copies that take place at each intermediate host. This scheme should become efficient for a larger number of renderers. Notice that the schemes presented here are the sort-first equivalent of the sort-last direct-send and binary swap algorithms [64].

5.2.3 FlowVR Render IEEE Vis 2005 Article

The article [15], included in section A.3 page 127, presents in details FlowVR Render.

5.2.4 Discussion

FlowVR Render is publicly available and distributed with the FlowVR Suite[2]. It is used for most of Grimage applications [62]. It shows a high versatility in a distributed context that makes it convenient to use. Beside its classical use presented before, it can also carry pixel data into dynamic textures. Following this approach we ported the Mplayer video player (Fig. 5.5). Mplayer becomes a viewer that decompresses a video stream, stores the result (YUV encoding) into textures sent over to the renderers. The renderers call a shader for converting the data to RGB and map the texture on any surface for rendering. Performing the conversion with a shader as several benefits:

- The viewer is unloaded from part of the decompression process handled to the renders.

[1] http://www.inrialpes.fr/grimage
[2] htt://flowvr.sf.net

- YUV to RGB conversion is a data parallel task adapted to the GPU.
- YUV encoding takes half the size of RGB encoding, reducing the network load.

Two videos are available at the FlowVR gallery[3] (one HDTV movie video and three HDTV movie video).

Using a similar approach, we developed an OpenGL wrapper. The openGL application is executed on one machine using the local GPU. Each rendered image is read back from the GPU memory and handled as a texture to build FlowVR render primitives that are next sent to renderers. This approach enables to render any OpenGL application in a FlowVR Render context. The resolution is obviously limited by the one of the initial rendering. The capabilities of the distant renderers are seldom used.

By getting ride of the OpenGL state machines FlowVR Render gains in versatility and performance, but, in opposite to Chromium, does not support legacy code. An OpenGL application needs to be rewritten using FlowVR Render, while Chromium enables its execution without even requiring a recompilation. Here are two partial answers to circumvent this weakness beside the OpenGL wrapper makeshift:

- Many applications are not written directly in OpenGL but using higher level libraries. Extending these libraries so they can support FlowVR Render enables to transparently support a large range of applications. We validated this approach with VTK [80] developing a VTK viewer. The low level OpenGL rendering layer of VTK was rewritten with FlowVR Render. It enables any VTK application to render using FlowVR Render [15].
- The Next version of OpenGL, OpenGL 3, abandons the state machine for a model that is close to the one adopted for FlowVR Render. Future work will focus on studying a new version of FlowVR Render supporting, at least partially, OpenGL 3.

As it was our goal, FlowVR Render isolates rendering in renderers. A renderer only has to deal with rendering issues. The renderer accepts some input events to control the rendering, like changing the viewing frustum, or switching to a wireframe rendering. The version for display-walls implements framelocking and binds camera controls so all renderers keep complementary viewing frustums. We also developed a library for the software calibration of the Grimage display-wall. Based on camera pictures, it computes transformation matrices and brightness correction masks to get a seamless global image (Fig. 5.6). Applying the corresponding corrections is directly handled by renderers.

[3] http://flowvr.sourceforge.net/FlowVRGallery.html

Figure 5.6: Display-wall software calibration.

Chapter 6

A Middleware for High Performance Interactive Computing

This chapter deals with the design of a middleware adapted to large interactive applications. This work led to FlowVR[1], an open source software still actively used and developed. FlowVR is the software backbone of the Grimage platform[2]. Its design was presented in two main articles [10, 61]. It is also at the center of four publications, which are more application oriented [13, 16, 14, 62]. Both Ph.D. works of Jérémie Allard [4] and Jean-Denis Lesage (on-going work) are focued on FlowVR design. Clément Ménier, an early and advanced FlowVR adept, intensively used it for his Ph.D. work [69].

6.1 Motivation

As seen in section 5.1.6 page 43, FlowVR development was motivated by the limitations experienced when intending to use Net Juggler for large parallel interactive applications.

Today, the number of such large interactive applications is still limited. The main difficulties to overcome include:

- Algorithmic issues to run correct simulations, to produce convincing and useful images or other non-visual renderings, to integrate and extract data from various captors like cameras.
- Software engineering issues where multiple pieces of codes (simulation codes, graphics rendering codes, device drivers, etc.), developed by different persons, during different periods of time, have to be integrated in the same framework to properly work together.

[1] http://flowvr.sourceforge.net/
[2] http://www.inrialpes.fr/grimage

- Hardware performance limitations bypassed by multiplying the units available (disks, CPUs, GPUs, cameras, video projectors, etc.), but introducing at the same time extra complexity. In particular it often requires to introduce parallel algorithms and data redistribution strategies that should be generic enough to minimize human intervention when the target execution platform changes.

Examples of such applications include the Hercules system that couples an earthquake simulation and an on-line visualization using 2000 processors to reach the frequency of 2Hz on a 1200 billions elements simulation [87]. Other initiatives intend to design cross-continental interactive applications relying on the performance of optical networking [83]. A number of virtual reality applications are relying on parallel machines to provide the required I/O and computing resources. Blue-C [49] and Grimage [14] are good examples of high performance immersive platforms relying on parallel machines to process in real time data acquired through a network of cameras.

FlowVR was designed with the goal of enforcing a modular programming that leverages software engineering issues while enabling high performance executions on distributed and parallel architectures. We did target a middleware tailored for high performance interactive applications rather than a generic purpose one. The objective was to take advantage of the specificity of these applications to achieve an improved modularity and performance. The base concepts FlowVR is built upon are:

- A static data-flow model. An application is seen as a static graph, where vertices are computing tasks distributed on computing resources and edges are FIFO data communication channels.

- A task, or component, executes an endless iterative loop ignoring networking issues. It just endlessly gets data from input ports, process the data and provide results on output ports.

- A set of tasks can actually be a parallel application, i.e. tasks that communicate between each-other by other means than the data channels defined by the graph vertices. The middleware should not be aware of these communications and should not impair them.

- Turning an existing code, parallel or not, into a component of the middleware should lead to minimal code modifications.

- The middleware should enable the design of advanced communication schemes, in particular collective communications, data re-sampling policies and complex synchronization patterns.

- The model should enable to define patterns, built from other patterns, easy to handle for further reuse.

- The specialization and parameterization of an application for a given target architecture should be as independent as possible from the core application design.

The data-flow model is classically used in several scientific visualization tools [27, 40, 1], but usually not in a distributed context or only in a limited way. The data-flow graph being static, it enables off-line error detections and optimizations. We decided to postpone the introduction of dynamic components in FlowVR, to gain experience on application development and better evaluate what could be required.

6.2 The FlowVR Model

6.2.1 Messages and Stamps

Each message sent on the FlowVR network is associated with a list of stamps. Stamps are lightweight data that identify the message. Some stamps are automatically set by FlowVR. The user can also define new stamps if required. A stamp can be a simple ordering number, the id of the source that generated the message or a more advanced data like a 3D bounding volume. To some extent, stamps enable to perform computations on messages without having to read the message content. The stamp list can be sent on the network without the message payload if the destination does not need it. It enables to improve performance by avoiding useless data transfers on the network.

6.2.2 Module

The base component of FlowVR is the *module*. A module has input and output ports. It executes an endless loop and its application programming interface (API) is built around three basic instructions:

- *wait*: lock the module as long as no new message is available on each of its input port.
- *get*: get a pointer on the new message received on a given port.
- *put*: publish a new message on a given output port.

If a port is not connected to any other FlowVR component it is inactive. The *wait* instruction ignore it. There is no imposed form for a module as long as its semantics is respected. It can be a process, a thread, a group of threads collaborating to implement a module.

By default each module has:

- an output port *endit* where a message is sent every time the module enters the *wait* instruction,

- an input port *beginit* that, when active, locks the module as long as no message is received on the port (the message is used as an event and its content is ignored).

6.2.3 Connection

A *connection* is a simple FIFO channel connecting an input port to an output port. Several connections can connect to one output port. In this case each message available on the output port is broadcasted on each connection. One input port can only have one incoming connection.

The simplest application a user can write involves two module connected through a connection. The size of the buffer associated to a connection is only limited by the amount of memory available. If the receiver is slower than the sender, an overflow will occur once the memory is saturated. We will see later how to avoid such situation.

FlowVR defines two types of connections. Full connections carry the full messages, i.e the stamp list and the payload, while stamp connections extract from the message the stamp list and carry only this data to the destination.

6.2.4 Routing Node

A *routing node* is a simple message routing task that has one incoming connection and possibly several output connections. It is used for data broadcasting or to reroute messages.

6.2.5 Filter

A *filter* has input and output ports. In opposite to modules that can only access the last message received on each input port, a filter has access to the full buffer of incoming messages stored locally. It can freely modify or discard any of these messages. A filter can for instance re-sample messages, by discarding all incoming messages except the last one that it forwards on its output port.

The API to develop modules is intentionally simple and constraining (see the *wait* semantic for instance). The goal is to keep module development simple. In opposite, filters offer more freedom in particular regarding buffer access. Usually an application developer does not have to develop new filters. Filters provided with FlowVR perform generic message handling tasks, making them easy to reuse in multiple applications. Combining filters and modules enable to implement complex behaviors as we will see in the following section.

6.2.6 Synchronizer

Synchronizers are a specific class of filters that are in charge of implementing synchronization policies. In opposite to general filters, they only receive message stamp lists and not the full messages.

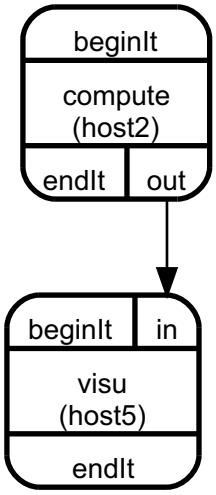


Figure 6.1: A simple FlowVR application with only one connection between two modules.

6.3 Simple Examples

We detail a few simple examples of some common patterns. Assembling such patterns for large applications can lead to complex data-flow graphs (Fig. 6.8 page 62).

The simplest application connects one data producer, the modules *compute*, and the *visu* consumer module (Fig. 6.3). The consumer frequency at most reaches the one of the producer. If the consumer is slower than the producer, the number of messages sent will grow because the consumer will not be able to process them. In the model, the buffers associated to FIFO connections are unlimited, but practically they are limited by the amount of memory available. So a memory overflow will occur.

A simple approach to avoid such memory overflow is to switch from a push paradigm to a pull one. Because FlowVR allows cycles in the data-flow graph, it is possible to have the *visu* module controlling the frequency of *compute*. Each time *visu* ends an iteration, it sends a request message on it *endIt* port. This message is forwarded by the *PreSignal* filter to the *beginIt* port of *compute*. This message will unlock *compute*, that will proceed to produce a new data.

To avoid deadlocks caused by the cycle, we use the *PreSignal* filter. This filter starts by sending a first message to *compute* and proceed by forwarding incoming messages: it puts an initial token into the cycle to unlock it.

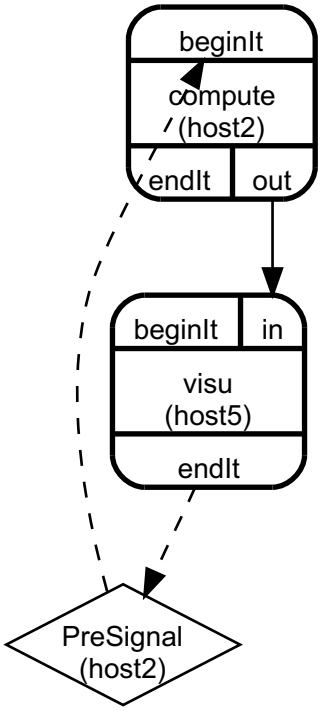


Figure 6.2: The *visu* module pull on demand messages from *compute*.

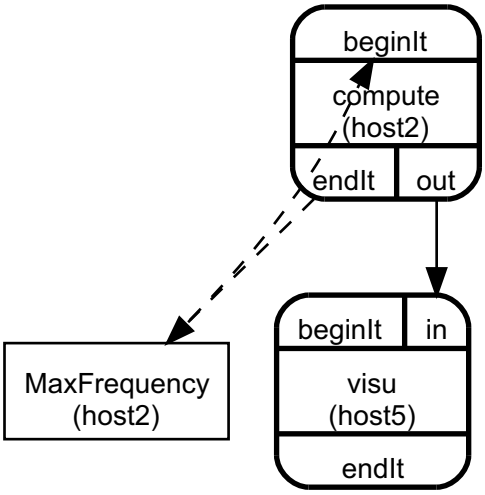


Figure 6.3: The max frequency filter bounds the frequency of the *compute* module.

Another approach to control the frequency of a module, is to use a *MaxFrequency* synchronizer (Fig. 6.3). Each time the module ends an iteration, it sends a request message to the synchronizer (*endIt* port). The synchronizer respond by sending a message to *compute* on it *benginIt* port, enabling a new iteration. The frequency of these responses matches the request frequency as long as it is lower than the maximum allowed frequency (a parameter of the synchronizer).

Here we have a cycle too. This is the *MaxFrequency* synchronizer that is in charge of sending the unlocking first message. As this synchronizer is always used in cycles, it is convenient to give it this responsibility.

We usually add an input port to the *MaxFrequency* synchronizer to modify online the maximum frequency parameter through an other module that implements a graphical user interface.

For large interactive applications, having all modules running at the frequency of the slowest one can severely affect the reactivity of computations. A possible approach is to consider that a data stream is a sampled signal that can be re-sampled. The consumer only gets the data it can process, discarding the other ones.

We present the greedy pattern, a basic re-sampling pattern favoring the reactivity by providing the consumer with the most recent data available (Fig. 6.4). Because filters have access to the full buffer of messages stored locally, it can easily implement a sampling policy. This pattern relies on 2 filters and one synchronizer.

Each time the module *visu* ends an iteration (*endIt*ăport), the synchronizer re-

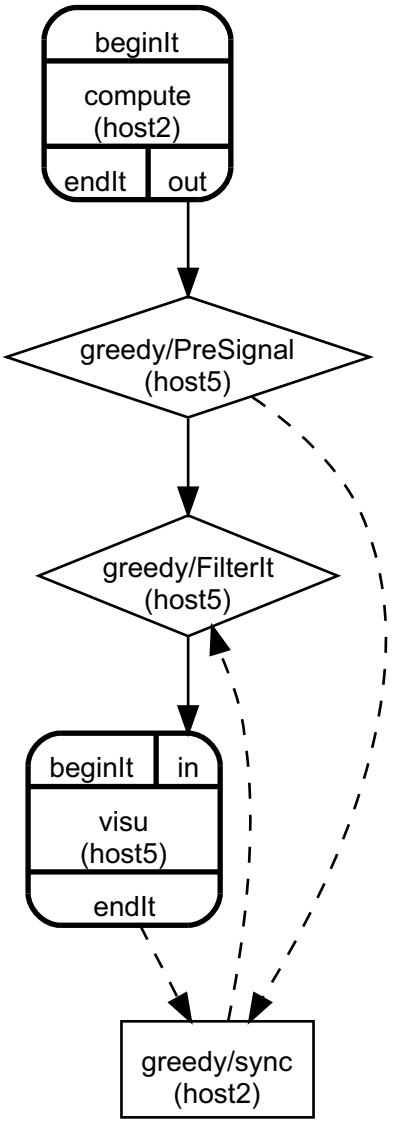


Figure 6.4: The greedy pattern enables *compute* and *visu* to run at independent frequencies

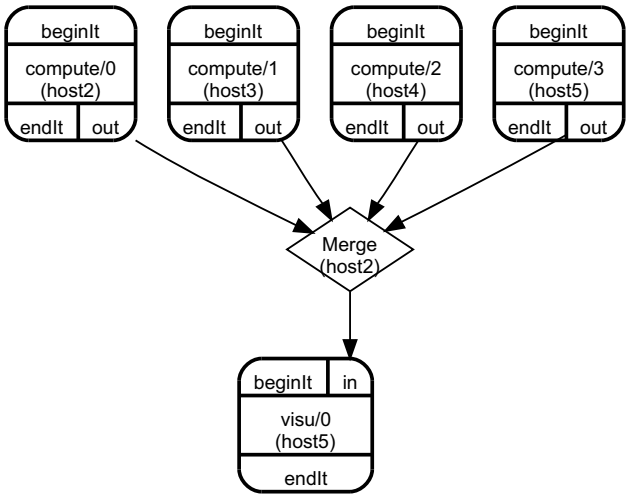


Figure 6.5: Merging messages with one filter.

ceives one message. The synchronizer also receives a stamp message for each message sent by the producer. This stamp message is actually provided by the *PreSignal* filter. Like for the previous example, the *PreSignal* is present to unlock the cycle created by the synchronizer. When the synchronizer receives a request from *visu*, it forwards to the *FilterIt* filter the stamp of the last stamp message received from *PreSignal*. The filter *FilterIt* waits to receive the full message having this same stamp, discards all older messages locally stored and forwards this message to *visu*. If no message is available when the synchronizer receives the request, it tells, with a special empty message, the module to reuse the last message already received.

Assume now that *compute* is a parallel application starting four modules *compute/O*,... *compute/3*. We could modify *visu* to have four input ports to receive the data part produce by each process. Though possible, this is an approach we usually avoid with FlowVR. It makes the *visu* module code dependent of its external environment. To enforce the modularity of the application, *visu* is not modified and the reduction of the partial results to one message is performed by an external filter (Fig. 6.5). Performance is affected if this involves data copies that would not perform a modified *visu* module. The reduction can also be implemented along a binary tree merging pattern for performance reasons (Fig. 6.6). Switching between these different patterns is external to the module code. It involves neither code modification nor module recompilation.

Notice that a host name, *host1*, *host2*, etc. is associated with each component. This is the host that executes the component. Module mapping is the responsibility

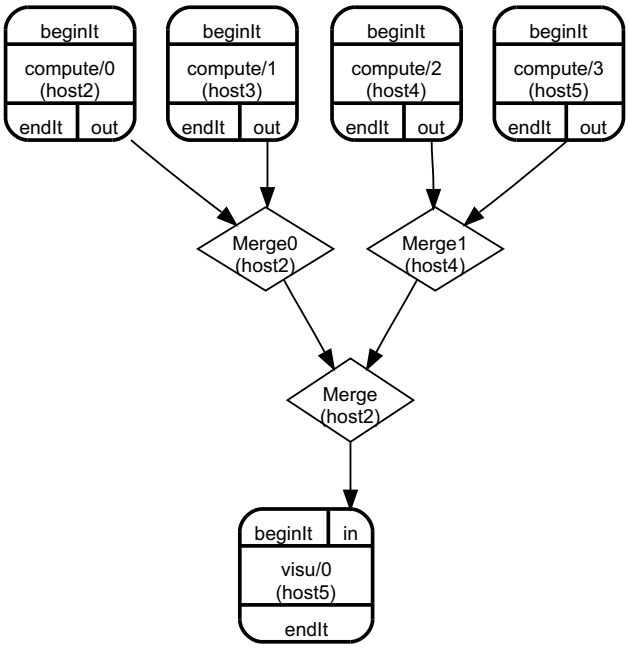


Figure 6.6: Merging messages along a binary tree of filters.

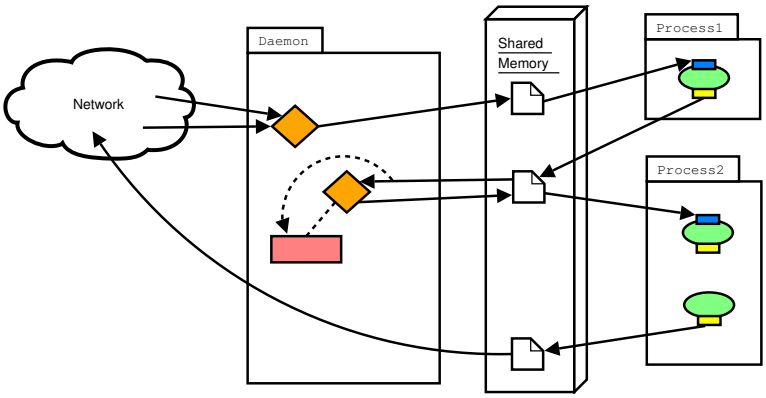


Figure 6.7: The FlowVR daemon acts as a broker between modules.

of the user. Filter mapping is usually automatically derived from the module hosts. This automatic mapping is based on our experience rather than some optimization process.

6.4 Run-time Environment

FlowVR run-time environment relies on a daemon that runs on each host of the target machine. This daemon is in charge of:

- registering each module running on its host,
- implementing all message exchanges transiting through its host,
- executing all local filters and routing nodes.

Modules are launched directly using their own command. FlowVR does not imposes any specific launching command to ease portability of existing codes. For instance a MPI code can use *mpirun*. Once launched, each module registers at the daemon of its host.

The daemon manages a shared memory segment that is used to store all messages it handles (Fig. 6.7). When a module requests a memory space to store a new message, the daemon provides the module a pointer to a free space in the shared memory segment. Having direct access to the shared memory segment saves data copies.

When a module executes a *put*, it tells the daemon that the message is ready to be forwarded to its destination. If the destination module runs on the same host, the demon provides a pointer to the module that directly reads the message from the shared memory. If the message has to be forwarded to a module running on a distant host, the daemon sends it to the daemon of the distant host. The target daemon retrieves the message, stores it in its shared memory segment and provides a pointer on the message to the destination module.

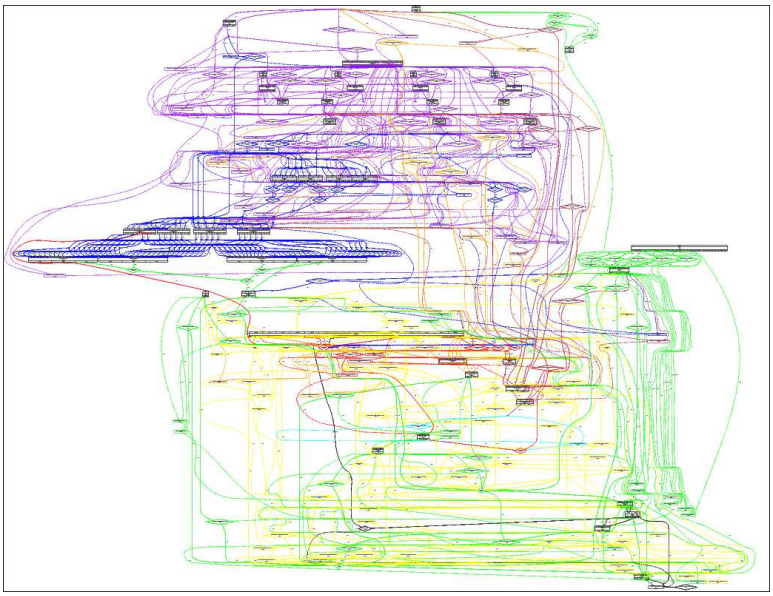


Figure 6.8: Example of the data-flow graph of an application running on the Grimage platform. Edges represent connections and vertices modules, filters or routing nodes.

Each daemon stores a table of actions to perform on messages according to their origin. An action can be a simple message routing, or the execution of a filter. Filters ares plugins loaded by the daemons when starting the application. Appear here an other important difference between a filter and module. A filter is necessarily a daemon plugin, while a module is a process or thread external to the daemon.

From the description of the application, we compute a list of commands that are sent to each daemon at starting time and that enables it to:

- instantiate a module controller for each module that will be registered,
- built its action table,
- load the required filter plugins.

The daemon and its plugins are multi-threaded, ensuring a better performance scalability on multi-core and multi CPU architectures.

6.5 Hierarchical Component Model

Because FlowVR is designed for large applications, the data-flow graph can be complex (Fig. 6.8). We have to provide the user tools to face this complexity and avoid him the burden of explicitly describing such a graph. We rely on the

composite design pattern [45] to support hierarchies of components. It enables to encapsulate in one component a complex pattern recursively built from simpler ones.

A component defines input and output ports. We distinguish two kinds of components:

Primitive components. A primitive component is a base component that cannot contain an other component. Primitive components are modules, filters, routing nodes and connections.

Composite components. A composite component contains other components (composite or primitive). Each port is visible from the outside and the inside of the component. Component encapsulation is strict. A component can not be directly contained into two parent components.

A link connects two component ports. It cannot directly cross a component membrane. A link between 2 ports is allowed only for the 2 following cases:

- A descendant link connects a port of a parent composite component to a port of one of its child component. Such links must always connect an input/input or output/output pair of ports.
- A sibling link connects two ports of two components having the same parent component. Such a link must always connect an input/output pair of ports.

A simple and common composite component is a *metamodule*. A metamodule handles modules that are logically related, in particular when they are all started from a single command. This is for instance the case for a MPI code that uses `mpirun` to start all its processes. Such metamodule takes as parameter the list of hosts where to start the program. From this list of hosts it defines the different modules, one per MPI process, and the exact syntax of the command to start the application.

In this model, a FlowVR application becomes one component. Each component is defined locally according to the other sibling components and its parent. It favors modularity. The developer does not need to control all components of the application as he would if he had to directly write the data-flow graph. For instance, all components need to have a unique id. With the hierarchical model, the user just needs to make sure the component he develops has an id different from the other sibbling components having the same parent component. He does not have to pay attention to other component ids. Then, the components of the data-flow graph are produced processing the hierarchical description. If we consider the examples of the component ids, the data-flow graph component ids are obtained by concatenating all ascendants ids. The unicity of ids between siblings components ensures the final ids are all unique.

The function to process a component is local to each component. It enables to adapt the function to the local context. For instance, a component could decide

to capitalize its id before to concatenate it. Such function is called a *controller*. Because components are developed using object inheritance, controllers are overwritten only when required, the inherited controller being used otherwise.

Controllers are called during an application *traverse*. Usually one traverse just calls one controller per component. During a traverse, parameters can be exchanged between controllers to propagate data. It enables for instance to get the concatenation of all ascendant names and a file where to append the newly computed component name.

Some controllers, called *configuration controllers*, modify the state of their component. In this case the execution order of various controllers is significant.

FlowVR application processing relies on 4 main controllers:

- an *execute* controller in charge of creating child components and ports components according to their environment (external parameters as well as other component states). For a MPI metamodule it creates the list of child component (modules) from the lists of hosts where to execute the application.

- a *mapping* controller in charge of defining the host each primitive component is executed on. For a MPI metamodule, it associates one host name to each child module.

- a *run* controller that extracts the launching command for each metamodule.

- an *XMLBuild* controller that builds the data-flow graph of the application (using the XML markup language).

Here is an example of component hierarchy (Fig. 6.9). The *Computes* metamodule is a MPI application spawning several modules. The *Visualization* metamodule only spawns two modules (interleaved threads), one dedicated to keyboard and mouse event capture, and the other to process and render incoming data. To interconnect both components, we use an extra composite component, called *Connect*. *Connect* is in charge of gathering the partial results from the various *Compute/i* processes to forward a single message containing a full simulation state to *Visualization*. *Connect* is built from the *NtoOne* component. This component encapsulates a generic tree pattern for data redistribution. *Connect* just sets the parameters of *NtoOne*: the arity of the tree (2) and the type of the component used for the tree nodes (*Merge*). The actual content of *NtoOne* is known once *Computes* is properly instantiated. Only at this point *NtoOne* knows how many pieces of data it has to gather to set the tree depth. The *execute* controller of *NtoOne* must be executed after the one of *Computes* that creates the modules *Compute/i*. If *Computes* spawns only one module *Compute/0*, *Connect* will produce one point-to-point connection between *Compute/0* and *Render*.

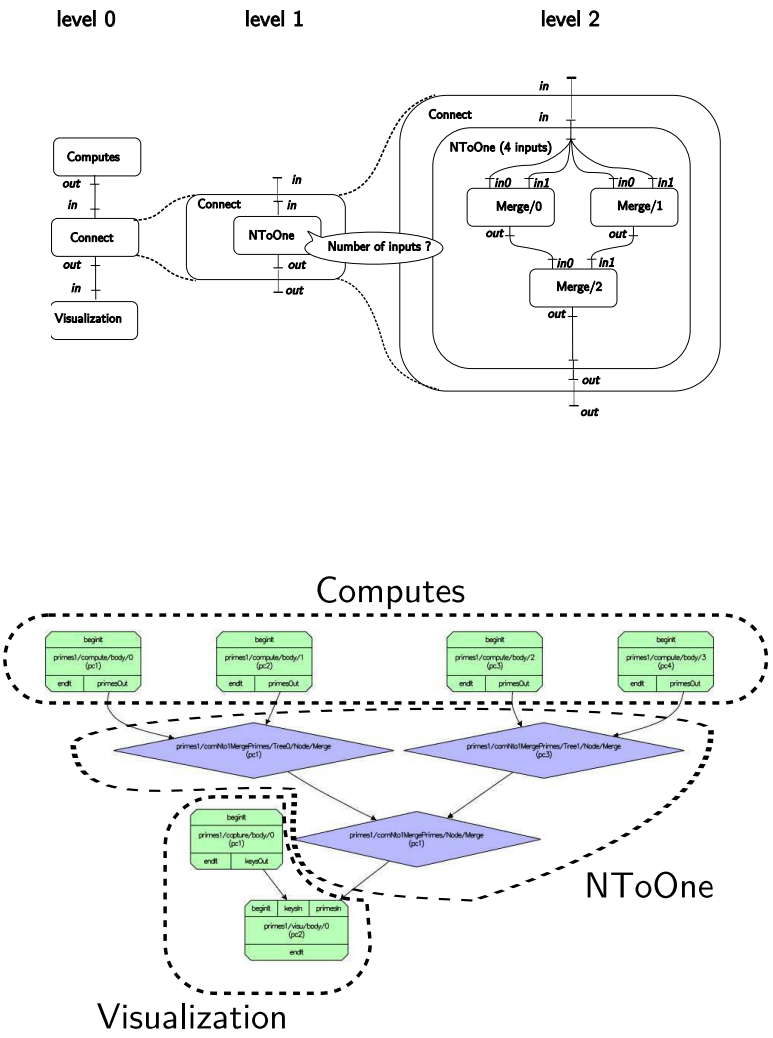


Figure 6.9: A component hierarchy (top) and the associated data-flow graph (bottom). Dashed sets show the composite components the graph elements are related to.

6.6 Application Processing

For configuration controllers that depend on the state of other linked (directly or not) components, the traverse must obey a certain execution order to respect data dependencies. For instance, a component may define its number of children according to the number of modules of the metamodule it is linked to. It thus requires that the metamodule has defined its child modules before the controller can be called on the current component.

We use a simple algorithm that guarantees to complete the traverse when possible or return the list of misprogrammed components if some data dependencies cannot be solved whatever the execution order is. The traverse algorithm is a greedy process. The algorithm manages a queue of non-executed components, initialized with the top-level components of the application. For each component in this queue, the algorithm tries to execute the associated controller. If the controller is successfully executed, then all of its children are pushed in the queue. Otherwise, the algorithm restores the component initial state and push it at the end of the queue. The traverse ends successfully when the queue is empty. If no controller can be called on the rest of the components in the list, then the algorithm stops in a fail state. The controller of the remaining components cannot be executed either because at least one of these components is misconfigured (a parameter is not instantiated for instance), or because a cycle of dependencies has been introduced when assembling the components.

The hierarchical component model only affects the front-end of FlowVR (Fig. 6.10). The run-time environment is not modified. Components are written in C++ and compiled into shared libraries. An application is also a composite component compiled into a shared library. It can thus be reused in other applications without being recompiled. The FlowVR front-end loads the application and applies a sequence of traverses to produce the list of commands to start the modules and the instructions to forward to the different daemons to implement the data-flow graph.

6.7 Europar 2004 and Supercomputing Journal 2008 Articles

The first article [10], included in section A.4 page 136, presents the FlowVR Model in details. Is is not based on the hierarchical component model but on the initial application description environment using Perl scripts and XML. The second article [61], included in section A.5 page 146, is focused on the hierarchical model. It presents the traverse algorithms, a complexity analysis and a convergence proof.

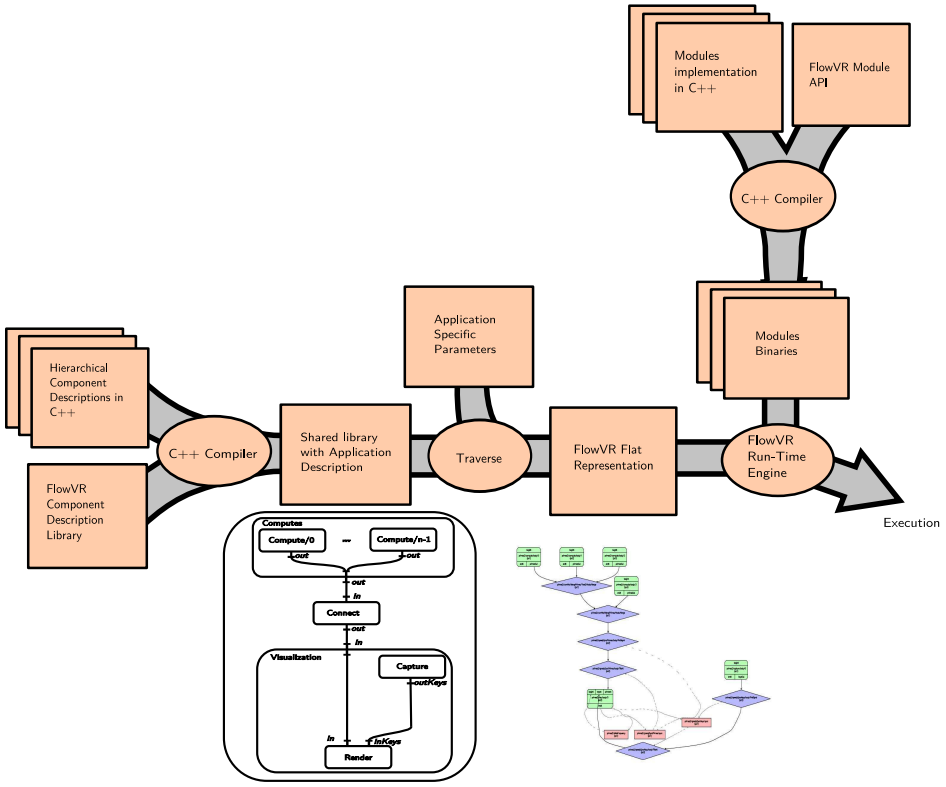


Figure 6.10: The FlowVR front-end. Components (left to right) are compiled, loaded and traversed to provide the module launching commands and the instruction sets for the daemons. Once compiled, modules (top to bottom) are started as requested by the application.

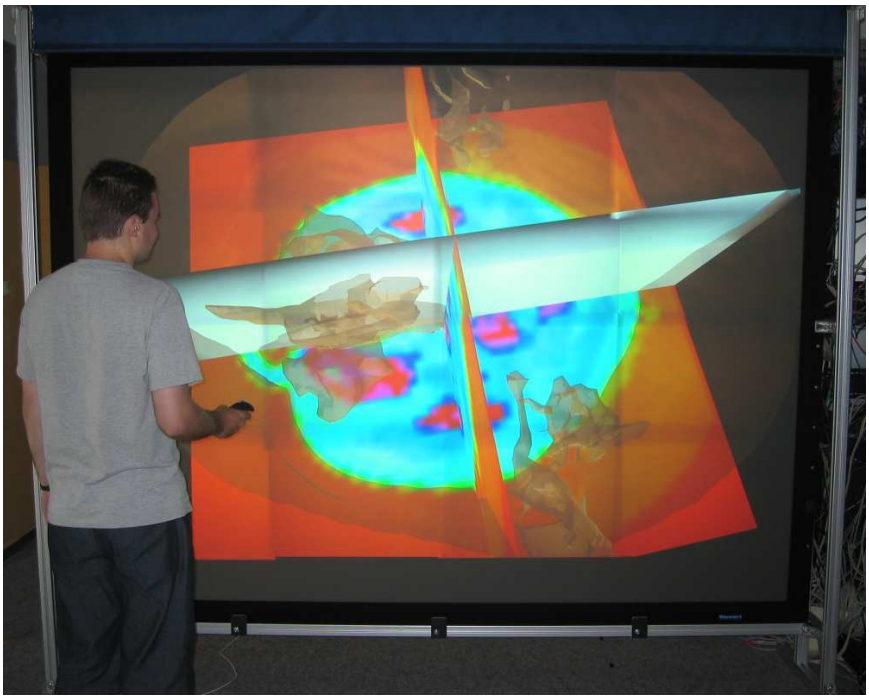


Figure 6.11: Iso-surface extraction performed with VTK and multi-projector rendering with FlowVR Render.

6.8 Positioning

The data-flow model is classical for scientific visualization tools. But usually distributed executions and parallel code coupling are not first class features and their support is limited [27].

Some tools, like VTK [2], VISIT [29] or AVS [88], support data parallelism relying on MPI. These tools adopt either a push (data-driven) or pull model (demand-driven). For instance VTK and VISIT are both pull based, OpenDX [1] and Iris Explorer [40] use a push one.

In a distributed context the pull model can impair performance, as it adds the latency caused by the request message. For this reason FlowVR adopts a push model. But because it allows cycles, it can also implement pull actions. It enables to mix between a push or pull model depending on the application requirements.

FlowVR, being a low level middleware, is not directly a scientific visualization tool. However it can encapsulate visualization tools to provide higher level functionalities. For instance, the VTK-FlowVR [15] extension enables to encapsulate VTK codes into FlowVR components (Fig. 6.11). Encapsulation can take place at an application level, where a full VTK application is one FlowVR component, or at a lower level, where a VTK pipeline is split in several FlowVR components.

Virtual reality frameworks are first focused on parallelism for multi-projector

rendering and distributed input devices [85]. For instance VR juggler [26] uses CORBA for distributing input devices. Parallel simulations coupling is supported in some cases relying on simple data redistribution mechanisms [90].

OpenMask [65] makes sampling one of its base concept. OpenMask is based on a data-flow model where each task run asynchronously. Each data stream is associated with an interpolation/extrapolation function to re-sample it according to the destination data needs. FlowVR filters can be used to implement similar re-sampling schemes.

Component oriented middleware libraries were first developed for distributed applications [73, 74], making flexibility their primary goal. It results in tools that are not well adapted for high performance computing, motivating the development of extensions and new models [58, 36, 37, 23, 18]. Notice in particular the on-going project SCIRun2, whose goal is to extend the SCIRun environment to support the CCA component model for high performance computing [93]. Limitations of classical component oriented middleware include:

- missing abstractions for parallel programming,
- performance issues,
- usually require substantial modification of existing applications.

The component models aforementioned, mainly derived from CCM, do not support component hierarchy. Fractal makes it one of its basic feature [38]. But like the CCM model, parallel code coupling is not directly supported. Pro-active, a grid middleware, implements part of the Fractal specification and supports both, component hierarchy and parallel code coupling [21]. However redistribution patterns are coded into the ports of the parallel components. A pattern cannot be modified without affecting the component, limiting the application modularity.

FlowVR adopts a component oriented approach based on a data-flow model rather than remote procedure calls. It takes advantage of its static data-flow model to enable component hierarchies, while enabling parallel component coupling. One important issue of parallel code coupling is data redistribution (classically called the $N \times M$ issue) [94, 39, 76]. The hierarchical FlowVR model enables to encapsulate redistribution patterns into components that are independent from the data producers and consumers. Though the model should support complex $N \times M$ redistribution patterns, we have not yet implemented such components.

Let us end this positioning section by mentioning works related to parallel skeletons. The skeleton model proposes a pattern language for parallel programming [31, 66]. A program is written from the composition of predefined parallel patterns. Various environments rely on this model like ASSIST [3] for grid computing or Skipper [81] for vision applications. Skeletons have a clear semantics, can be associated to a cost model and hide their implementation details to the application developer. Given the target architecture, the application is compiled down to a specialized parallel code. Hierarchies of skeletons are supported by some environments like Skipper-D.

FlowVR composite components can be considered as parametric skeletons. Jubertie *et al.* propose in [55] a cost model for the data-flow graph that could be extended to composite components. However, FlowVR components tend to be closer to design patterns than skeletons. Application developers can freely ignore all predefined patterns and develop their own patterns. There is no discontinuity between primitive components and high level ones that would make such development specifically difficult.

6.9 Applications

Interactive application development has always counted for a significant part of our activity. It enables us to:

- validate our approaches with realistic applications. FlowVR is designed to enforce application modularity for leveraging software engineering issues while enabling high performance executions on parallel architectures. It is thus essential to have access to realistic applications to evaluate both aspects aforementioned. It includes code scalability and reuse, maintenance ease, multi-user application development practicability, and portability on different architectures.

- detect important emerging issues that must be faced when the complexity of applications increases. Developing our own applications gives us hands-on experiences, which are crucial to foresee issues, understand them, and propose relevant solutions. A good example is FlowVR Render, developed after facing difficulties to merge and render 3D data provided by several distributed sources.

We already introduced some of the applications we developed with Net Juggler, FlowVR and its extensions (FlowVR Render and VTK FlowVR). In this section, we focus on the most significant ones.

6.9.1 Grimage

This work is closely related to the research activity on the Grimage platform (Fig. 6.12). The application was developed over several years (2004-2008), involving various developers (about 10 different persons). We provide some snapshots of different versions of this application (Fig. 6.13), but we strongly advice to refer to video materials to get a quick and precise understanding of the application and evaluate the performance as well as the interaction level (Videos available from the FlowVR gallery[3]). As mentioned in the introduction of this Chapter, four papers are related to applications [13, 16, 14, 62].

[3] http://flowvr.sourceforge.net/FlowVRGallery.html

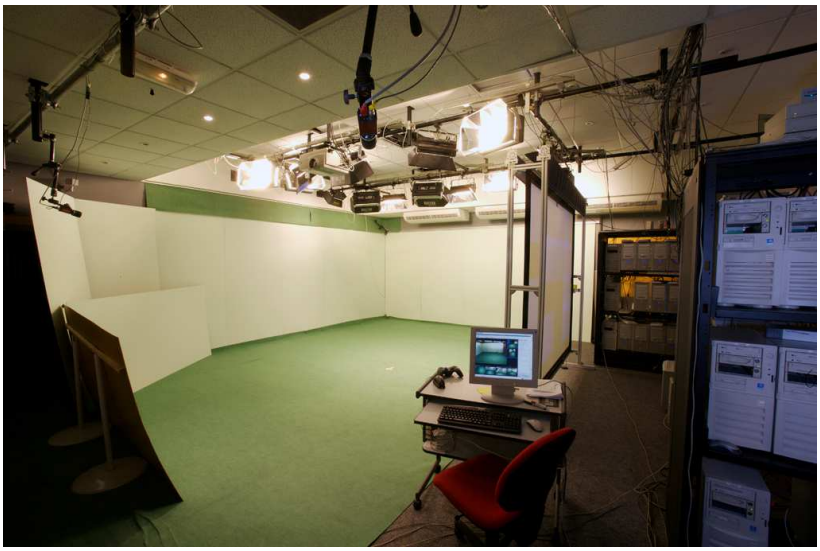


Figure 6.12: The Grimage platform.

6.9.2 Interactive 3D Modeling

The core element of this application is real-time 3D modeling used as an input device. 3D modeling computes a real-time model of the objects present into the acquisition space from a set of images shot from different videos cameras. We developed a parallel version of the Polyhedral Visual Hull algorithm [41] that computes the complete and exact visual hull polyhedron with regard to silhouette inputs (Fig. 6.14). The algorithm and its parallelization is presented in details in Chapter 7. The first parallelization attempt was based on MPI. It led to a monolithic code abusing of communicators to provide some modularity (at the time FlowVR was still in its early stages). We switched to FlowVR and experienced a significant improvement of the code modularity. It became easier to detect erroneous programming choices, or to test different approaches playing with filters. The application performance quickly outperformed the MPI one. This code was initially developed in 2004 and after 4 years of improvements is still actively used. Amongst the most significant changes, models are now dynamically textured using the photometric data extracted from the images. We also recently started to study collaborative interactions through 3D models by having two platforms (Grimage and its portable version) connected.

6.9.3 Distributed Physics Simulation

3D modeling is the interaction device for applications that involve two other important components: rendering and physics simulations. FlowVR Render provides distributed rendering support (Section 5.2 page 45). Different approaches have been evaluated for physics simulation. The first one consisted in encapsulating in

a FlowVR sequential component the Open Dynamics Engine (ODE). It enables to simulate the dynamics of rigid and articulated objects. We next developed a distributed physic simulation framework (Fig. 6.13 (a) and Video). The model relies on two main classes of modules, *animators* and *interactors* [16]. Animators own objects. They are responsible for updating their object states from the forces that apply to these objects. Objects are self-defined to ensure that adding or removing an object does not affect other objects. The forces applied to objects are computed by interactors, based on the object states they receive from animators. Each interactor is usually dedicated to one algorithm, for handling collisions between rigid objects for instance. It enables a modular coupling of existing algorithms to build simulations combining the capabilities of each of these algorithms.

The simulation developed couples a net simulation based on a mass-spring system, a rigid object simulation, and a fluid simulation enabling two-way interactions between rigid objects and fluids based on Carlson *et al.* algorithm [28]. Because the fluid simulation is computationally intensive, it was parallelized with MPI using a classical block partitioning. Simulation and rendering on a dual processor machine (1.6 GHz dual Opteron PC) with a scene consisting of:

- 20 rigid objects
- one fluid simulation on a $32 \times 64 \times 32$ grid.
- one mass-spring 2D net, with 20×20 nodes.

reached 6.5 frames per second.

For the interactive execution, we added 3D modeling as an input device. A specific animator is dedicated to the 3D model. It is managed as a classical rigid object with one-way interactions (no force can affect it). The application was executed on the Grimage platform that consisted in:

- sixteen 1.6 GHz dual Opteron PCs equipped with NVIDIA FX 5700 graphics cards
- eleven dual-Xeon 2.6 GHz PCs
- a gigabit Ethernet network
- five FireWire cameras
- a sixteen projector display-wall

Eight hosts were dedicated to the parallel fluid simulation. Fluid rendering used particles. To avoid collapsing the network when sending the particles from each MPI process to the 16 rendering nodes (up to 200000 particles generated), the output filter of MPI process was set to dynamically route the grid cell content according to each projector's frustum. The application reached 18 fps, about three times faster than on a single machine, but with 16 times more pixels to compute

and a dynamics 3D mesh to handle. The network is an important bottleneck. When all fluid particles are forwarded to all 16 rendering hosts, the frame rate is reduced 1 fps. Dynamic particle routing enabled to significantly reduce this bottleneck to reach 18 fps.

This application, one of the largest resource consumer we have developed so far, was built from a pool of 20 modules. The data-flow graph contained about 200 module instances, 4000 connections and 5000 filters.

6.9.4 SOFA Simulation Framework

We relied on the SOFA [7] simulation framework for the application developed for the Siggraph 2007 Emerging Technology Show [14]. SOFA enables to simulate complex scenes involving various types of objects (solid, articulated, soft, fluids). Because objects are tightly coupled, the parallelization effort of SOFA mainly focus on multi-processor and multi-core PCs that offer high bandwidth data exchanged rates. Jérémie Allard, at INRIA Lille, currently develops a fine-grain parallelization approach to transfer computations on the GPU, using the Cuda library. We are studying a coarser grain parallelization based on work-stealing with KAAPI [46] (Everton Hermann Ph.D. work). The application architecture fellows a similar organization than previous ones. SOFA is simply encapsulated in one FlowVR module (Fig. 6.15).

The demo run during 5 days at San Diego on a portable version of the Siggraph platform. About 3000 persons experienced the demo (Fig. 6.13 (c), Siggraph 2007 submission video and Siggraph 2007 Etech show video). This demo was well covered by medias[4].

The on-going work on collaborative interaction is a direct extension of this architecture where two 3D models are produced from different platforms and are gathered into a virtual world managed by SOFA (Fig. 6.13 (d), (e)).

6.9.5 Molecular Dynamics

We are today extending our application domain to molecular dynamics (Fig. 6.16). This on-going work is performed in collaboration with the Université d'Orléans, the CEA (Bruyère le Châtel) and the IBPC (Paris). The goal it to enable interactive handling of molecules [44]. The user interacts through a haptic device. Molecular simulation is performed by the Gromacs parallel code (MPI based parallelization). With the help of FlowVR, we expect to couple Gromacs, the haptic device, and a FlowVR Render rendering to execute interactive applications involving tens to hundreds of processors.

[4] http://www.inrialpes.fr/grimage/#press

6.9.6 Interactive Grid

An other on-going project involving the Université d'Orléans, the INRIA Bordeaux (project-team I-parla) focuses on grid oriented applications. The goal is to develop interactive applications that harness the resources of heterogeneous distributed resources, from clusters to mobile devices. Typically some clusters are in charge of providing georeferenced data [47] to build a virtual scene where users from different sites meet. Users may participate using ultra mobile PC with reduced capabilities as well as immersive and multi-camera environments powered by clusters. FlowVR is used as the backbone for coupling the components running on the different clusters. This work already led to some low-level improvements of FlowVR to control the application deployment on several sites.

6.10 Debugging

Parallel application debugging is a critical issue. It is not different for FlowVR applications, which tend to be heterogeneous and complex (Fig. 6.8). We developed several complementary approaches.

The first one, well adapted to the data-flow model, consists in visualizing the graph of the application. We developed a first simple tool based on the Graphviz library to produce images of the data-flow graph. However, it suffered from different issues. It produces a 2D vectorial image (pdf, svg) with a high quality, even when closely zooming on a specific elements. But navigating, zooming and even opening the image was slow. To circumvent these limitations we developed a graphical interface based on OpenGL for the graph drawing (layout is still computed by Graphviz). Relying on OpenGL enables to smoothly and quickly zoom on the graph, change the view point, color the graph, etc. (Fig. 6.17). It is associated to a QT user interface to control the element displayed, to color vertices according to their host or their type, etc. Experience shows it is a must-have tool appreciated by all developers. It provides an alternate view of the application that users quickly understand. Many bugs are easily detected, like wrong connections or incorrect host mapping.

This tool does not support yet hierarchical graph visualization. This is an ongoing work. The difficulty comes from the layout computation, drawing, and ergonomy of the hierarchy navigation.

Trace capture and visualization is also an other debugging tool we provide. Trace capture has been developed to minimize the performance overhead. It takes advantage of the FlowVR architecture. A specific filter, called a *logger* is loaded to capture events that it sends as messages on its output port. A specific module runs per host to save these events to a file. Using a different module, we could for instance directly compute and display some average monitoring values instead of storing them into files.

Some events are predefined (entering or leaving the *wait*) while others can be user-defined. Activating trace capture affects the data-flow graph but does not

require to recompile modules or filters. Because the target machine usually does not provide a global clock, we perform tests to evaluate the clock drifts just before and after activating the trace capture.

We developed a simple OpenGL based tool for trace visualization (Fig. 6.18). Like for the data-flow graph, OpenGL enables to quickly and smoothly navigate, zoom, slide on the time-line.

The third level of debugging we provide is even more classical. We take advantage of the different traverses of the application to detect possible errors, and correct them when possible. A typical error is to have several incoming connections to an input port, or to forget a connection along a path of links. These can easily be detected during traverses.

6.11 Discussion

6.11.1 Hierarchical Components

Up to version 1.5, FlowVR application description was based on a mix of XML files and Perl scripts the user had to provide. This environment was quickly developed to enable application programming, as our efforts were focused on the core FlowVR implementation. This approach proved limited for several reasons:

- Large Perl scripts quickly become difficult to understand and debug ;
- The user had to navigate between 3 languages, C++, XML and Perl, an extra difficulty especially for students without a good programming experience ;
- The modularity offered by Perl functions was limited.

Preliminary user feedbacks shows that the hierarchical component model overcomes these limitations. The C++ API is sometimes too verbose, making application description fastidious. In the future we will provide a binding of the API with scripting languages using Swig. An extra effort is still required on patterns (components). The goal is to develop relevant patterns, generic yet usable, while avoiding their proliferation. Our experience with realistic large applications is here crucial.

6.11.2 Static data-flow Graph

The data-flow graph is static. It has several benefits, one being the ability to display it for debugging purpose. It also eases optimizations during traverses, the full graph being available. Having a static graph does not prevent dynamic behaviors. A module can dynamically spawn processes or threads (FlowVR is thread-safe), as long as it takes care of scheduling and data sharing. A module or filter can also implement a classical client or server. This is the solution adopted by FlowVR VRPN. Our approach is to confine dynamicity where it is required not to lose the

benefits of a static graph. In particular, we are studying the type of sub-graph connection/reconnections that could be enabled and the associated modifications to apply to the FlowVR code.

6.11.3 Module Pool

In comparison to other VR middleware libraries, we offer few support for scene graphs or other higher-level libraries. Our goal is to focus on the middleware core design rather than providing a large range of peripheral tools. We will provide some modules. For instance we should shortly make available a SOFA module. But we mainly count on external contributors to help us fill a module pool.

6.11.4 Application Monitoring

Rudimentary tools have been developed to monitor and control a running FlowVR application. They demonstrated their usefulness as it enables the user to understand the behavior of his application and eventually change some parameters. We expect to generalize this approach by developing an integrated environment for steering FlowVR applications. The modular design of FlowVR enables to implement these extra tasks as FlowVR components.

6.11.5 Multi-CPU/GPU Support

How FlowVR behaves on multi-core, multi-CPU and multi-GPU machines ? The FlowVR daemon is multithreaded to take advantage of multi-core/CPU machines. Executed on 8 dual-core PC, we experienced no performance issue. But advanced performance tests are required. On the GPU side, we are starting a collaboration with the Universidade da Coruña to integrate into the FlowVR daemon, via filters, features to simplify and optimize GPU access for modules developed with Cuda.

6.11.6 Interoperability

Interoperability between languages and operating systems is a base feature of many component oriented environments. FlowVR currently provides only a C++ version of its module API. We expect to provide bindings for other languages using Swig, enabling to develop applications mixing modules programmed with different languages. The FlowVR daemon runs on Linux and Mac OS X platforms, but Windows support is not planed yet (seldom used for clusters but more common in VR labs).

6.11.7 Diffusion

FlowVR[5] was first publicly released in December 2003. It became the FlowVR suite with the addition of FlowVR Render and VTK FlowVR (Fig. 6.12). Recently

[5] http://flowvr.sourceforge.net/

FlowVR VRP [77], a FlowVR wrapper of the VRPN device library developed by the Université d'Orléans, joined the suite. On average, today the suite is downloaded 40 times per month. Beside the FlowVR users we are directly working with, we have few feedbacks on the possible uses of FlowVR. We suspect that a large amount of these downloads are for testing purpose. Adopting a middleware library like FlowVR to support heavy development efforts requires a long term support guarantee. We are committed to provide such support, but having a larger community of users and developers would make it even more convincing. Beside publications and a public web site, we intend to promote FlowVR through different other actions. Effort are made to make on-site and off-site demos (IEEE VR 2006, Vision 2006, Siggraph 2007, 40 ans INRIA 2007, VRST 2008, Fête de la Science au Grand Palais de Paris 2008). FlowVR is also the backbone software of the interactive solutions the 4D View Solutions[6] company proposes. Starting in September 2008, FlowVR will be used for a Master class at the Université d'Orléans.

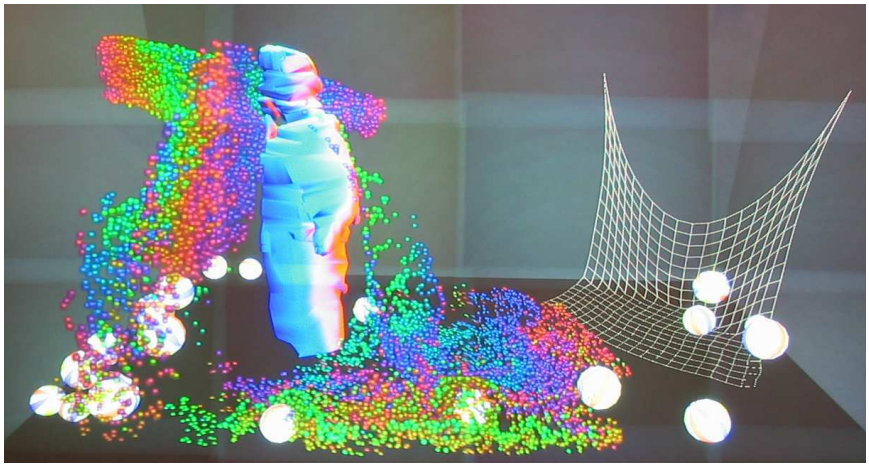
6.11.8 Long Term Perspectives

When developing large applications, users are facing an explosion of the number of design choices and parameters to tune. FlowVR intends to assist the user by offering an environment that separates module development, assembly, parameterization and instantiation for a given target architecture. Still, the user needs expertise to master these different aspects. Should a given module be parallelized ? What communication pattern to use between modules (FIFO mode or re-sampling) ? How to map the components on the available resources ? Decisions are made taking into account multiple criteria, often antagonists, like latency, frequency, level of details. High frequencies require to allocate more resources to components, usually obtained by distributing them on different hosts. But it leads to extra communication overheads that affect latency. Tuning the level of detail enables to control the amount of resources used, but obviously affect the quality of computations. We also have to consider variations during execution. For instance, when a second user enters the acquisition space, the frequency of 3D modeling decreases significantly. How the application could dynamically adapt to this changing context to improve the use of the available resources, taking into account the user will (higher priority on latency than on level of details for instance) ? In the context of multi-modal interactions, but also when considering approaches like frameless rendering, spatial and temporal coherency are important criteria to consider. The answer is not trivial as it depends on the application and the human capabilities of accepting some incoherences. For instance a haptic device could compute force feedbacks based on a collision position that is slightly different from the one the user visually experience. This may be caused by an inaccurate calibration, the latency on event transmission, rounding errors, etc. In molecular dynamics, each simulated time step represents a few femtoseconds. They are obviously computed at a much slower frequency, but

[6] http://www.4dviews.com/

will not be experienced as a discrepancy as long as it enables comfortable interactions. In a virtual football game, users will be much more sensible to the dynamics of the ball as they will directly compare their virtual experience to the real one.

All these issues are long term research goals we are considering from a system/middleware point of view. On-going work with Jean-Denis Lesage focuses on on-line adaptive frequency control. We intend to develop local adaptive algorithms that require sparse and local monitoring data about the current state of execution. Re-sampling schemes lead to useless data production, i.e. data that will be discarded. Are these data really all useless? Over producing data can improve latency to some extent. If we are able to suppress useless data, can we re-allocate resources to improve the performance? We are still in the early stages of understanding these behaviors and developing a model for future algorithms.



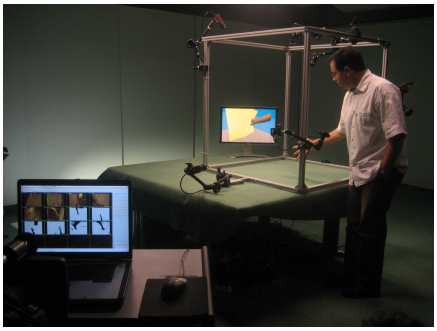


Figure 6.13: Various versions of 3D modeling based interactive applications (2005-2008). (a) Distributed simulation, 3D model not textured (2005). (b) 3D model textured (2006). (c) Siggraph 2007 Emerging Technologies demo with textured 3D models and the SOFA simulation engine. (d), (e) Collaborative application using two 3D modeling platforms (2008).

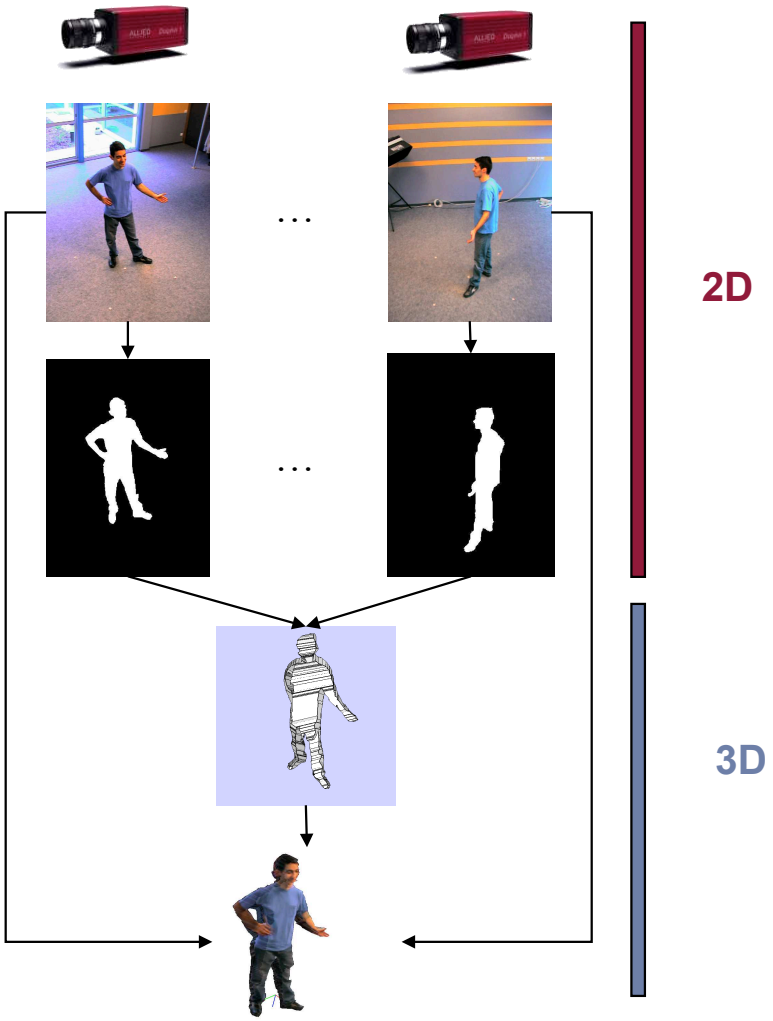


Figure 6.14: The 3D modeling pipe-line.

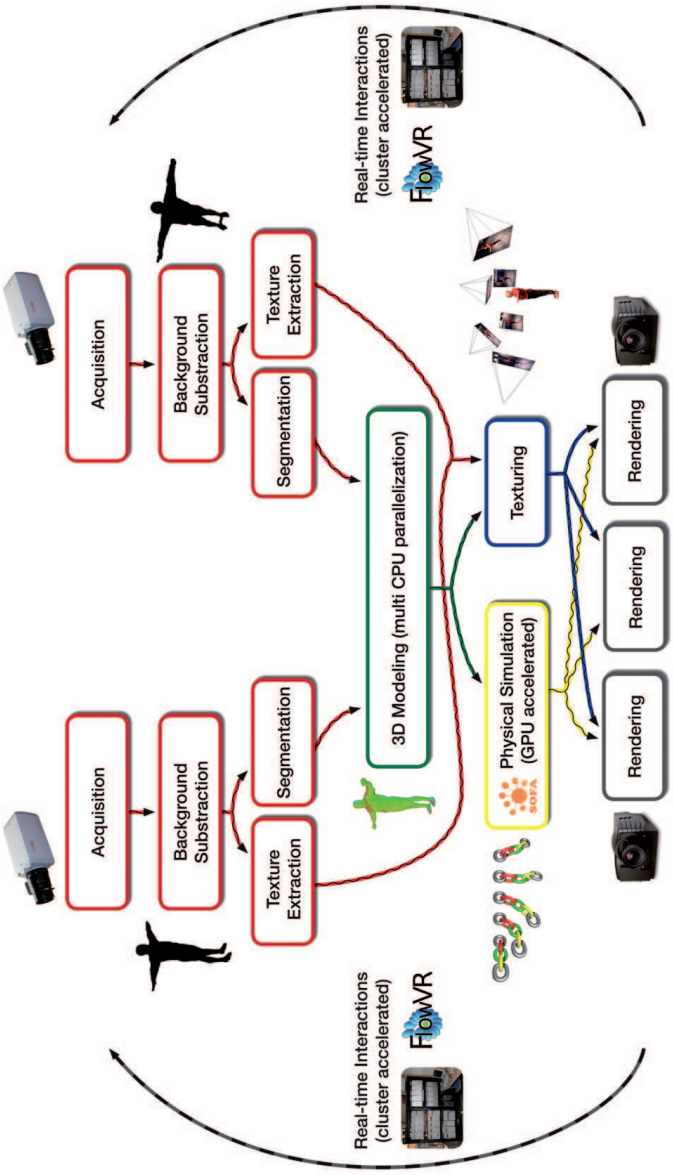


Figure 6.15: Overview of the application architecture presented at Siggraph Emerging Technologies 2007.

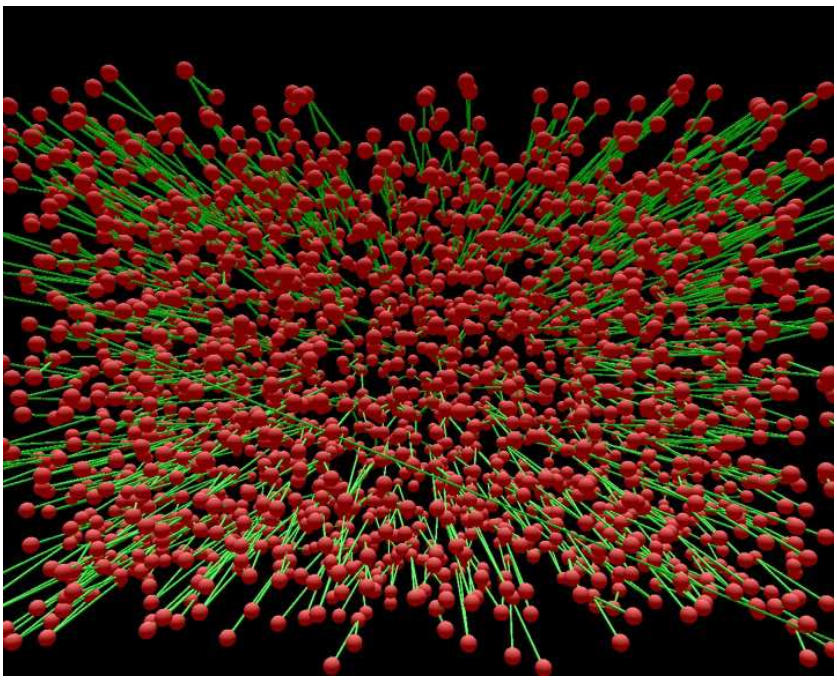


Figure 6.16: Screen shot of an 2500 atom interactive simulation. Gromacs simulation coupled to a FlowVR Render rendering.

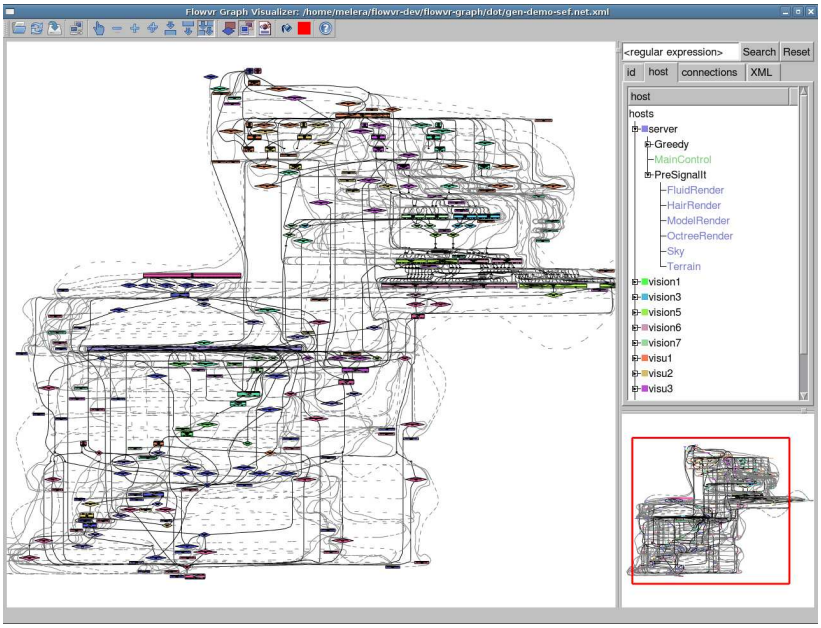


Figure 6.17: The OpenGL/QT graphical user interface for navigating application data-flow graphs.

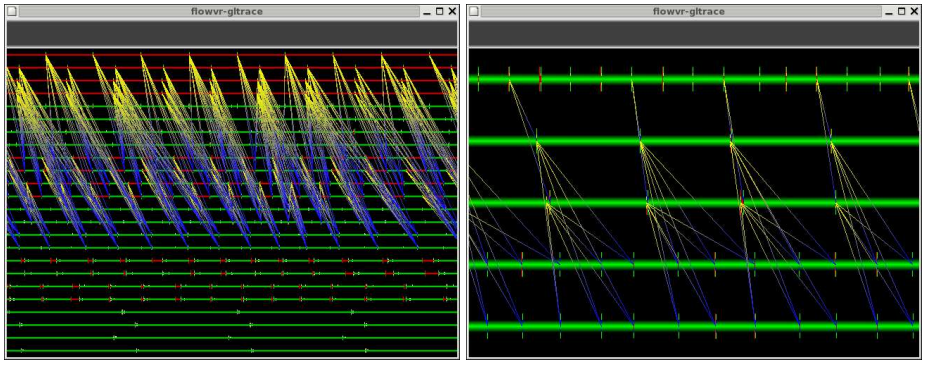


Figure 6.18: Trace visualization at 2 different zoom levels. Horizontal lines represent modules or filters when active (green) or waiting (red), diagonal arrows are communications.

Chapter 7

Real-Time 3D Modeling

This chapter presents the parallelization of two 3D modeling algorithms, EPVH [41, 43], and octree carving [86]. EPVH is the algorithm used on the Grimage platform. The code of the parallel version has been transfered to the 4D View Solutions[1] company. The parallelization is based on FlowVR. This work led to several publications [42, 5, 8], including a derivative computer vision work on pose recovery [70]. The second algorithm relies on an anytime work stealing approach. The proposed algorithm enforces a relaxed parallel width first octree carving that enables to stop computations at anytime while ensuring a balanced carving. It was published at EGPGV 2007 [84]. Clément Ménier's Ph.D. work was focused on this topic [69]. Benjamin Petit started his Ph.D. in 2007 to study interactions through 3D models. Both Ph.Ds are co-advised by Edmond Boyer. Luciano Soares worked on octree carving during its Postdoc stay at INRIA.

7.1 Motivation

One important VR device is the position tracker that provides in real-time the 3D position of a marker. Various approaches exist, relying on passive or active markers. Because each marker is attached to a known object or body part, it enables to track its position, compute its velocity and acceleration. But data density is limited, and trackers require to attach markers to the objects or users, restricting the flexibility of the system. In VR environments for instance, trackers classically provide data for less than ten points. Up to a few tens of markers are used for special effects and avatar animation in motion capture environments.

Markerless 3D modeling takes a different approach. The goal is to extract relevant data from a scene based on its observation, without having to equip objects with markers. We are considering here modeling the 3D shape of a dynamics scene using multiple video cameras.

Markerless interactions were pioneered by Krueger *et al.* [59] whos used one camera. Several multi-camera systems have been designed for 3D TV or free

[1] http://www.4dviews.com/

viewpoint video [56, 52, 63]. Because interactions are limited to changing the view point on the scene, 3D modeling can be performed off-line or with a high latency. Computing a full 3D model is not even required as only one or several 2D views on the scene are used for rendering. For telepresence, distant participants exchange their 3D and textured model. The closed interaction loops between users require low latencies [91, 49, 67]. One-way interactions with virtual objects were experimented by Hasenfratz *et al.* [50].

We can distinguish two main approaches to compute 3D models from images, the former using photoconsistancy [79], i.e. color consistency across images, and the latter based on silhouettes [82], i.e. image contours. We focus on the latter approach that retrieves the *visual hull* of the objects by reconstructing their shape from the silhouettes extracted from the video streams [22, 60]. Geometrically, the visual hull is the intersection of the *viewing cones*, the generalized cones whose apex are the cameras' projective centers and whose cross-sections coincide with the scene silhouettes. When considering piecewise-linear image contours for silhouettes, the visual hull becomes a regular polyhedron. A visual hull cannot model concavities.

We worked on two different algorithms. We first parallelized the EPVH algorithm developed by Franco and Boyer [41, 43], both of them involved in the Grimage platform. The EPVH algorithm has the particularity of retrieving an exact 3D model, i.e. the back projection of the 3D model into the images provides the exact same silhouettes than the ones used to produce the model. This is an important feature when the models need to be textured as it makes textures, extracted from silhouettes consistent with the 3D model.

The second algorithms is a classical one based on an "octree carving" [86]. This algorithm, not being exact, gives poor results when intending to texture the models produced (after computing a surface using a marching cube algorithm for instance). But this algorithm is interesting for uses that do not require rendering. It provides a regular volumetric data structure that can be efficiently accessed, for fast collision detection for example. Our goal when parallelizing this algorithm was first to test how work stealing approaches could be used in an anytime context.

7.2 Computer Vision System

3D modeling is one part of the computer vision system we built for Grimage. It also includes video streams processing, simulations and rendering tasks (Section 6.9). We give a brief overview of the computer vision part. Refer to [69] for more details.

Both algorithms work from silhouettes. First, cameras need to be calibrated and tuned, an off-line process that takes place before data acquisition. Cameras also need to be genlocked, i.e. to synchronize image shooting. Inappropriate synchronization can lead to poor 3D models with missing parts as the image set the algorithm works on does not show objects at the same position. Quality of synchronization depends on the speed of the objects considered. The 3D modeling pipe-

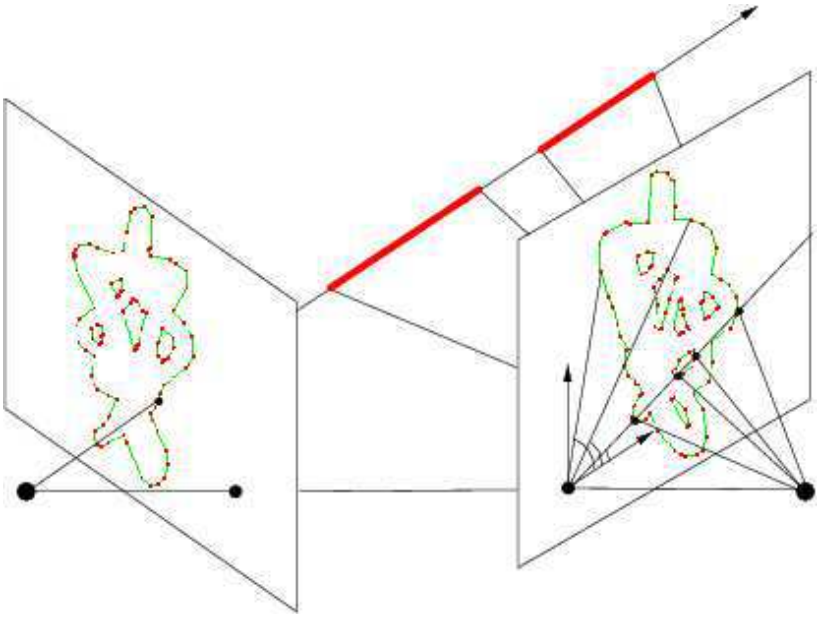


Figure 7.1: Viewing Edges.

line starts with computations that are local to each camera (Fig. 6.14 page 6.14). It consists in subtracting the background and computing the vectorized silhouettes. Background subtraction is performed based on the fixed background learned during the initialization phase [57]. Vectorization relies on a algorithm proposed by Debled *et al.* [35]. It enables to control the vectorization level of details, a feature we use to control the work load of the 3D modeling algorithm. Once vectorized we extract the silhouettes as well as the texture delimited by each silhouette. These steps, are performed locally by each host in charge of a camera (for performance's sake we usually have only one camera per host).

The 3D modeling algorithm takes as input the silhouettes as well as the calibration data for each camera. Texturing takes place when the 3D model needs to be rendered. On Grimage we use FlowVR Render and a shader that textures each face by mixing the three closest images.

7.3 Parallel EPVH

7.3.1 EPVH Overview

We first give an overview of the sequential algorithm. For more details refer to [43].

EPVH first computes *viewing edges* (Fig. 7.1). Viewing edges are the edges of the visual hull induced by viewing lines of contour vertices. There is one viewing line per silhouette edge. On each viewing line, EPVH projects the silhouette of each other image delimiting segments included in all silhouettes. Each segment,

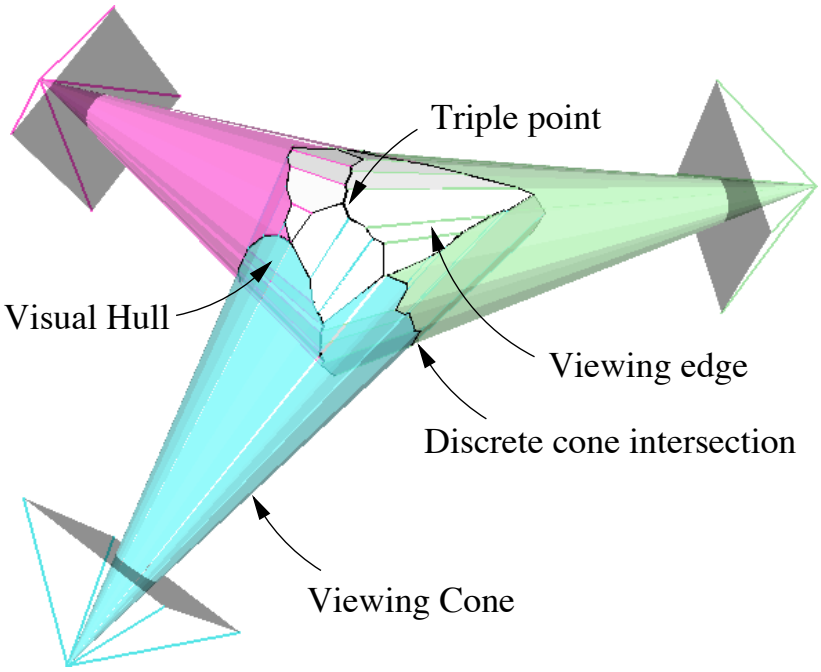


Figure 7.2: Polyhedral visual Hull build from the intersection of 3 image silhouettes (viewing cones). This case exhibits one triple point, the intersection point of 3 surfaces that project into 3 silhouette segments from 3 different images.

called a viewing edge, is an edge of the visual hull and each segment extremity a vertex. Each vertex is trivalent, i.e. the intersection point of 3 edges (higher valence is neglected because highly unlikely).

For each viewing edge, EPVH stores the two generator planes, i.e. the two planes defined by the projection of the two silhouette vertices incident to the edge the viewing line is originated from. It also stores for each viewing vertex the third generator plan defined by the projection of the silhouette edge that intersects the viewing line.

However the visual hull is incomplete (Fig. 7.2). Somes edges are missing to fulfill the mesh connectivity. It also misses somes vertices, called triple points. A triple point is a vertex of the visual hull generated from the intersection of 3 planes defined by silhouette segments from three different images.

Existing vertices are only connected by one edge, a viewing edge. This edge is delimited by the intersection of the two generator planes belonging to the same silhouette. EPVH intersects each of this generator plane with the third one, providing the missing edges. Let P be this third generator plane. The missing extremity of each of these edges is first delimited by an existing vertex, the one defined by the intersection with one of the generator plane incident to P in the same silhouette. Then, like for viewing edges, it checks if the segment is intersected by other silhouettes. If this occurs, we have a new vertex, a triple point.

This process provides the full mesh connectivity. A last step consists in traversing the mesh to identify the polyhedron faces. 3D face contours are extracted by walking through the complete oriented mesh while always taking left turns at each vertex. Orientation data is inferred from silhouette orientations (counter-clockwise oriented outer contours and clockwise oriented inner contours).

7.3.2 Parallel Algorithm

The parallel EPVH algorithm is a three stage pipe-line.

- **Stage 1: viewing edges**. Let P be the number of hosts in charge of computing the viewing edges. Each host in charge of a silhouette extraction broadcasts it to the P hosts. Each host computes locally the viewing edges for P/n viewing lines, where n is the total number of viewing lines.

- **Stage 2: mesh connection**. Let M be the number of hosts in charge of computing the mesh. The P hosts from the previous step broadcast the viewing edges to the M hosts. Each host is assigned a slice of the space (along the vertical axis as we are usually working with standing humans) where it computes the mesh. Slices are defined to have the same number of vertices. Each host complement the connectivity of it sub-mesh, creating triple points when required. The sub-mesh are then gathered on one host that merges the results, taking care of the connectivity on slice boundaries (removing duplicate edges or adding missing triple points).

- **stage 3: face identification**. The mesh is broadcasted to K processors in charge of face identification. Workload is balanced by evenly distributing the generator planes the processors extract faces from.

This parallel algorithm gives satisfactory results on the Grimage set-up. We are able to match the cameras refresh rates (tuned in between 20 and 30 frames per second) and ensure a latency below 100 ms (including video acquisition and 2D image processing) when one person is present in an acquisition space surrounded by 4 to 8 cameras. Processors involved in the 3D modeling vary from 4 to 8. Involving more processors did not proved to significantly increase the performance.

Using a large number of cameras raises several issues. The algorithm complexity is quadratic in the number of cameras, quickly leading to non acceptable latencies. Increasing the number of cameras can also affect the quality of the model due to error accumulation on input data (calibration errors for instance). Today our efforts focus on using higher resolution cameras (2M pixel cameras are in test) rather than significantly more cameras. The algorithm complexity is in $n\log(n)$, where n is the maximum number of segment per silhouette, making it more scalable on this parameter. Having more cameras makes sense for large acquisition spaces, where 3D models are computed per sub-sets of cameras.

7.4 Parallel EPVH ICVS 2006 Article

The ICVS 2006 article [8], included in section A.6 page 170, presents the Grimage computer vision system and the EPVH parallel algorithm.

7.5 Parallel Octree Carving

7.5.1 Octree Carving

Octree carving is a simple 3D modeling algorithm. A voxel, corresponding to a 3D volume of the acquisition space, is projected into all silhouettes. If it lies inside all silhouettes, it is part of the 3D model. If it is outside at least one silhouette, it is excluded from the 3D model. If it intersects at least one silhouette, the test is recursively applied on the 8 sub-voxels. The algorithm stops when all voxels up to a certain maximum depth level have been tested. Carving can also take place under a time constraint. Once the algorithm is requested to stop, it applies a final test to decide if each untested voxel is rather inside or outside the model. In this context, a width-first carving ensures an even level of detail on the full 3D model.

7.5.2 Work Stealing

Because the work-load is not evenly distributed, mainly concentrated on the model surface, we relied on work stealing for our parallelization. It is a classical approach for dynamic load balancing that has been used for various computations, including parallel graphics[51, 30]. It extends the Graham list scheduling principle [48] for programs that create tasks recursively. The principle is simple. When starting the execution, a first processor is assigned all source tasks (the initial ready tasks). At runtime, each processor maintains a local list where it stores the ready tasks it has locally created. A task becomes ready when all its predecessor tasks, i.e. the tasks it depends on, have been executed. The tasks are organized in the list according to a total sequential depth first order. When a processor P completes a task, it pops the first one t (according to depth first order) in its local ready list if non empty. If its local list is empty, P is idle and becomes a thief: it randomly selects another processor until finding one victim processor V that owns ready tasks. Then it picks-up the oldest ready task t in the ready list of V. In both cases, P starts the execution of t.

Work stealing achieves a provable performance with respect to the *work* and *depth* of the parallel algorithm. The work W_1 is the total number of elementary operations performed during the execution of the algorithm. An instruction may be a standard operation or a task creation. The depth W_∞ is the critical path in number of operations for an execution on an unbounded number of processors, i.e. the number of instructions along the longest dependency path. Let T_p be the execution time on p identical processors with execution speed Π (in number of instructions per time unit). An execution takes a time $T_1 = \frac{W_1}{p\Pi}$ on a single

Figure 7.3: Parallel octree carving with 16 cores (max depth of 8). Voxels are color-coded according to the core that produced them. It shows the good spatial locality of work stealing.

processor and a time $T_\infty = \frac{W_\infty}{\Pi}$ on an unbounded number of processors. On p processors, work stealing ensures that with a high probability [19]:

$$T_p \leq \frac{W_1}{p\Pi} + O\left(\frac{W_\infty}{\Pi}\right), \tag{7.1}$$

the number of steals being small, $O(W_\infty)$ per processor.

With slight modification in the work stealing strategy, Bender *et al.* [24] prove that this result also holds for p heterogeneous processors with average speed π per processor.

Thus, if the depth W_∞ is small compared to the total amount of work W_1 the parallel execution time is close to the lower bound $\frac{W_1}{p\Pi}$. This motivates the use of work stealing to schedule parallel programs having a small depth W_∞.

Octree carving shows properties making it very well adapted to work stealing. The computation associated with one voxel is considered as one task. The dependency graph corresponds to the octree graph, as a given voxel can be computed as soon as its parent has been treated. In the worst case where no pruning occurs, a tree of depth n leads to $W_1 = \frac{8^{n+1}-1}{7}$ tasks, while the critical path is

$W_\infty = O(n)$. Thus, having $W_1 \gg W_\infty$, work stealing should lead to optimal parallel executions. Even when pruning occurs, the ratio is usually very favorable to work stealing. For instance, our test data set, consisting of a full human body (Fig. 7.3), has $W_1 = 162799$ voxels when going up to depth level $W_\infty = 8$.

Work stealing efficiency also depends on the overhead to handle tasks. Stealing a task just consists in getting a voxel coordinate and size, a small amount of data. A task only depends on its parent voxel. It is ready to be executed as soon as created.

Work stealing efficiency can be impaired as parallelism is reduced when starting the algorithm. Moreover, the amount of computation to test a voxel against a silhouette is proportional to its size. To soften this effect, carving starts at a higher depth level. Experiments show that a good choice is to start at the smallest depth level providing at least one task per processor.

7.5.3 Parallel Algorithm

We assume the target parallel machine supports a global address space with a shared (or virtually shared) memory access.

Each voxel is represented as a quadruple of its coordinates and level (i, j, k, d). Voxels are organized in ready lists. We call a task, the computations required to test a voxel status.

The algorithm starts at depth level n with 8^n initial voxels. These voxels are split into p ready lists, where p is the number of processors. The goal is to avoid the performance bottleneck of the first depth levels that do not provide enough parallelism.

The ready lists are organized in a singly-circularly-linked list. There is one list of ready lists per depth level, called a level list. All level lists, except the ones from the starting depth level, are initially empty. Each processor is assigned a first ready list from the starting level.

One processor is the manager. It takes care of initializing the first ready lists and signal all other processors when computations must be stopped (because the timeout occurred).

All processors start to work on the tasks of the same depth. Each processor tests the voxels of its ready list. If a voxel needs to be split, its 8 child are inserted into the ready list of the next level. Once it ends its ready list, it cycle through the list, starting from a random position, for other freely available ready lists. If not available, it starts a second cycle to steal tasks from the ready lists of other processors. When encountering a victim with a non empty ready list, it grabs half of the remaining voxels, leaving a minimum of voxels (defined by a threshold). This aims at avoiding to steal a too small number of voxels, with a stealing overhead that would not be compensated by the new workload balance.

When a processor ends its second cycle through the current level list, it starts working on the next level, processing the voxels of its ready list if not empty.

To enforce a width first voxel processing without compromising the efficiency of work stealing, we propose a loosen synchronization mechanism. It enables a

processor to process its ready list at depth d only if all voxels from depth $d-2$ have been processed. The goal is to overlap synchronization overhead while limiting the octree unbalance. To each depth level d corresponds a shared counter $C[d]$ initialized to the number of processors p. When a processor completes all its ready tasks at level d and if $C[d-1] = 0$, it starts stealing. If it does not succeed to get voxels from other processors, then it decreases $C[d]$ by 1 and starts the computation of its local voxels at level $d+1$. Once completed, it waits until $C[d-1] = 0$ before starting new steal requests.

A time control routine is integrated in the algorithm to bound the execution time. The algorithm is requested to stop through a timeout event that all processors check every time they end a ready list. They flush the remaining voxels with a specific fast test to associate them a inside/outside status.

7.5.4 Provable Performance

We prove the performance of the parallel algorithm against the reference sequential algorithm.

The following theorem states that the algorithm, without time limit, is almost p times faster than the sequential one if the depth of the octree is small w.r.t. its number of voxels.

Theorem Let T_s be the time of the reference sequential algorithm to compute an octree with n nodes and depth d on a processor with speed Π. The adaptive algorithm running on p identical processors with speed Π computes the same octree in time:

$$T_p =_{n\to\infty} \frac{T_s}{p} + O\left(\frac{d \log n}{\Pi}\right) \tag{7.2}$$

However, due to real time interactive constraints, the depth of the octree is truncated at a given unknown time limit. The next theorem states that in a fixed time t, the parallel algorithm on p processors reaches almost the same level of details as the reference sequential algorithm in a time $p.t$.

Theorem Let n_p be the number of voxels computed by the adaptive algorithm in a time limit t on p identical processors. Let n_s be the number of voxels computed by the sequential reference algorithm in a time limit $p.t$ on 1 processor. Let d_p (resp. d_s) be the last fully completed level of the adaptive (resp. sequential) algorithm.

Then:

$$n_p = n_s - O(\log n_s) \text{ and } |d_p - d_s| \leq 1 \tag{7.3}$$

For the proofs of both theorems, refer to [84], included in section A.7 page 178.

7.5.5 Experimental Results

The algorithm was implemented with Posix threads. For a better performance, the use of mutex like semaphores was eliminated. Instead, assembly atomic operations like compare_and_swap "cmpxchg" and atomic_add_return "xadd" com-

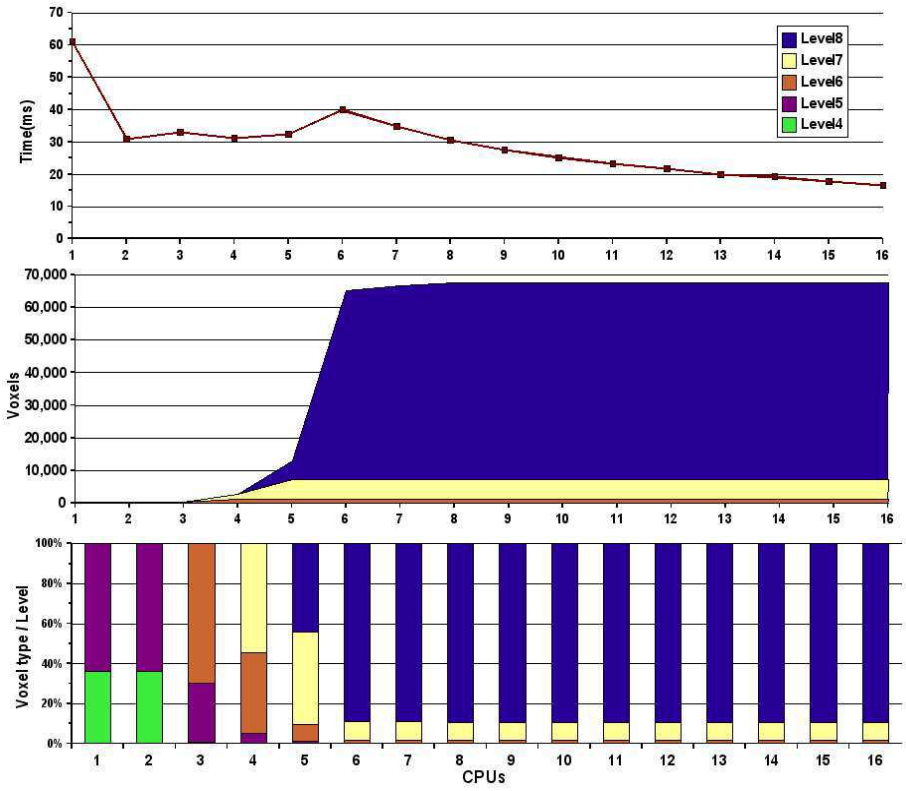


Figure 7.4: Performance result of octree carving with a 30 ms time limit. (Top) Execution time, (Middle) number of voxels computed, (Down) percentage of voxel per depth level (max depth level set to 8).

bined with the LOCK prefix were used. These atomic operations are supported by most modern CPUs. We noticed a performance increase of about 20% compared to a mutex based implementation. We run one thread per processor, locked with the "pthread_setaffinity_np" instruction.

For shake of conciseness, we only give an overview of the experimental results. Tests were executed on a PC equipped with 8 dual Core 2.2 GHz Opteron processor. Carving was performed against a pre-recorded 8 camera video sequence of a full body person (780x582 pixel images). The speed-up reaches 14 with 16 cores, no time limit and a max depth level of 7. As expected from the theoretical results the number of steals is small with an average of about 42 steals attempt, 25 successful, compared to the 162799 voxel computed (Fig. 7.3).

We tested the time control routine with a 30 ms deadline. With just one core it is not even possible to complete level 5, making the model unrecognizable. The execution time is significantly larger than 30 ms, because the core does not check the elapsed time before it completes the first ready list. Up to 8 cores, the time control is effective: the execution stops before all voxels of level 8 are computed. Notice

that the measured execution time is usually slightly higher than 30 ms because all cores apply a fast test algorithm to guess if each pending voxel is full or empty after the timeout occurs. With 8 cores, the 30 ms limit enables to reach the max depth level. Next, as the number of cores further increases, the extra computing resources available enable to decrease the execution time, ending below 20 ms (the number of voxels computed at level 8 stops to increase).

On-line experience on the Grimage platform is coherent with these off-line results (see the (Octree carving Video[2]).

7.6 Parallel Octree Carving EGPGV 2007 Article

The EGPGV 2007 article [84], included in section A.7 page 178, presents the parallel octree carving algorithm in details and comes with the theoretical proofs of performance.

7.7 Discussion

The parallel EPVH algorithm has been used for all applications involving 3D modeling developed on Grimage. It proved stable and reaches a performance that enables interactions with a low latency. From a parallelization point of view this algorithm could be significantly improved. The parallelization we proposed is strongly based on the sequential version to take advantage of its complex data structure and various optimizations. But highly optimized data structures adapted to sequential algorithms can impair independent task extraction if not carefully revisited. This is a work that has not been done yet as we mainly focused on developing an operational vision system, 3D modeling being only one part of this full system.

On Grimage, EPVH provides models for the physical simulation engines and for rendering. Being exact, EPVH leads to good quality textured models. Texturing increases drastically the ability of the observer to recognize the object or person being modeled. It gives the illusion of geometrical details missing on the 3D model (cloth folds or concavities for instance). On-going work focuses on efficient handling of texture streams that put a high pressure on network bandwidth. It becomes an important issue as we are increasing the number of cameras (about 15), their resolution (2M pixels), and testing telepresence between distant sites. Texture mapping is also a difficult research problem we have not been able to pursue beside the direct algorithm we are using on Grimage. Related issues include occlusion detection, multi-view texture blending, re-lighting.

Though not yet used for Grimage applications, octree carving provides an interesting data structure for physics. The work stealing mechanism achieves a good load balancing, leading to high speed-ups. Combined with a relaxed width first carving and the ability to stop the execution under the occurrence of an external

[2] http://vpod.tv/menier/151271

signal, it demonstrates its ability to take advantage of the cores available to compute faster or more voxels. Blender *et al.* [24] proved the performance of work stealing for heterogeneous processors with varying speeds. We extended our algorithm to enable a core to delegate part of voxel testing to one GPU. The goal was to validate if work stealing enabled us to easily take advantage of the cores as well as the GPUs available. A CPU is very efficient in testing a voxel as it probes the voxel status after each individual point projection test. In opposite, the SIMD nature of the GPU and the overhead of CPU/GPU data transfer impose that many points be tested before returning the result. It generally slows down the core/GPU couple, impairing performance in comparison to an execution without GPU. The GPU was programmed with OpenGL. Because Cuda offers more flexibility, we could expect some performance gains by switching to Cuda and using more recent generations of NVIDIA graphics cards.

3D modeling provides an instantaneous surface or volume of the observed scene as well as photometric data. It enables a textured rendering and to compute collisions. Because we have no velocity data, reaction forces due to collisions are incorrect. We do not identify the objects or some of their sub-parts, limiting higher level interactions. These are open issues. Using markers can solve some of them. In a markerless context some solutions are proposed, usually not at interactive execution time [69]. Benjamin Petit started in 2007 a Ph.D. to better understand the relation between input data and interactions, in particular in the 3D modeling context, augmented of extra data if required.

Chapter 8

Conclusion

This document synthesized 8 years of my research activity corresponding to a major change of focus. After a Ph.D. in parallel programming language design and a Postdoc on parallel machine performance measure, my research effort headed towards the association of parallel computing and virtual reality.

First contributions answer to the challenge of powering an active stereo multi-projector environment with a PC cluster equipped with commodity graphics cards. It led to Net Juggler and SoftGenLock. Today, using PC clusters for such environments is the norm. Most of VR libraries support PC clusters.

Based on this experience and preliminary coupling experiences of VR and parallel simulations, we designed FlowVR, a hierarchical component oriented middleware based on a data-flow paradigm. The goal was to support coupling various heterogeneous codes, possibly parallel, to build large interactive applications requiring multiple input, output and computing units. Experiences with large applications show that FlowVR meets its goals. It enforces a modular programming that leverages software engineering issues while enabling high performance executions on parallel architectures. It assists users by offering an environment that separates module development, assembly, parameterization and instantiation for a given target architecture. Besides coupling, FlowVR also proved relevant to develop native parallel codes, like the FlowVR Render distributed protocol for parallel 3D rendering, a parallel physical simulation engine or the EPVH parallel 3D modeling algorithm.

Parallel EPVH ensures load balancing through the preprocessing steps occurring at each pipe-line stage. It assumes the evenly distributed tasks all require a similar work load. For octree carving, we adopted a different approach based on work stealing. It ensures an efficient load balancing at a finer granularity, enabling to stop the computation at anytime. We are now using work stealing for the parallelization of the SOFA physical simulation framework. The context is different. Work stealing is used to correct the work unbalance that could occur between iterations (new collisions between objects for instance). We switch from a classical recursive work stealing scheme to an iterative one, requiring to revisit the algo-

rithm.

The Grimage platform plays a central role in this research work. It enabled to validate our approaches on realistic applications and to evaluate their limits. This is essential for interactive applications where the human has a central role. Even if we did not conduced rigorous user studies, experiencing ourselves, observing and questioning visitors was a very valuable source of inputs to foresee upcoming relevant research directions. Grimage was also the cornerstone of the fruitful collaboration with the Perception and Evasion project-teams. The Grimage lab is the room where people meet, work together, exchange skills, imagine, discuss and mature new ideas. It has been the catalyst of a very creative collaboration. Our contributions are diffused through scientific publications, but also open source software libraries, on-site and off-site demos, and technology transfer to private companies.

This work goes beyond the initial VR oriented research. What I would call *High Performance Interactive Computing (HPIC)* is at the intersection between VR, scientific and information visualization, computational steering and parallel computing. Involving humans changes the architecture and goals of the applications. Interactions require input and output devices. Interaction should be immersive and multi-modal, going a way beyond the classical keyboard, mouse and display devices. Interaction means a performance adapted to human sensory-motor and cognitive abilities. From a system/middleware point of view, it translates into latencies, refresh rates, coherency and level of details management concerns.

Adapted computing platforms are needed. Grimage relies on a dedicated cluster (30 nodes, gigabit Ethernet network). Because high performance architectures have evolved towards more commodity architectures and improved software administration tools increasing management flexibility, we can consider a more polyvalent architecture. The coming Digitalis computing platform of the INRIA and LIG lab will be a large cluster shared for HPIC as well as HPC. Some nodes will be available to support I/O devices like cameras and projectors. GPUs will be available for both computing and rendering. The OAR scheduler will enable to reserve interactive computing time. Because costs are shared with the HPC community, it provides access to a machine with a unified high performance network gathering a large number of nodes (from 1024 to 2048 CPU cores expected). Such architecture will ease quality model texturing with high resolution multi-video streams. It will also enable to couple larger parallel codes, like Gromacs molecular dynamics simulations. This machine will be part of the Grid'5000[1] grid infrastructure, favoring on-going telepresence and interactive grid experiments.

Grimage leans towards a large number of high resolution cameras to increase the 3D model quality and the volume of the acquisition space. All interactive experiments on Grimage have been performed at the third-person. We are integrating a head mounted display and held cameras for testing first-person interactions. Beside reducing latency, improving 3D modeling geometrical and photometric qual-

[1] http://www.grid5000.fr

ity, improving refresh rates to better sample fast movements, we are also studying how to extract higher level data, like velocity. Interaction capabilities depend on the quality and type of information extracted. For instance computing the force a virtual ball is kicked requires the acceleration of the user's foot. Grabbing virtual objects is an other example of difficult interaction issue. Springs could be created between the hand and the object when detecting a collision. But changes of the 3D model geometry between iterations can affect the stability of the connection with the virtual object. Grabbing in the real world involves frictions and pressures. Can we infer such data using cameras?

The multi and many core shift in processor architecture gives parallelism a new emphasis even for small configurations. FlowVR has been tested on SMP machines, but as the number of cores grows, we may have to revisit the daemon architecture. To take advantage of GPUs we are studying a FlowVR extension to simplify and optimize GPU access for Cuda modules. We tested how work stealing could be used to dynamically balance octree carving on multiple CPUs, one of them using the GPU as a co-processor. In the SOFA context, we expect to couple our work stealing parallelization with the Cuda code developped by Jérémie Allard.

When developing large applications, users are facing an explosion of the number of design choices and parameters to tune. Changing execution context can also affect the behavior of the application, requiring on-line adjustments. FlowVR work focused on structuring the application development and instantiation. A long term research goal is to study how middleware algorithms could take in charge some aspects of resource allocation. Target criteria are multiple and can be antagonist, like latency, frequency, level of details. Algorithm behaviors should be consistent with the user expectations or provide high level tuning parameters to control trade-off between criteria. On-going work focuses on on-line frequency control. We intend to develop local adaptive algorithms that require sparse and local monitoring data about the current state of execution.

Bibliography

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

[2] J. Ahrens, C. Law, W. Schroeder, K. Martin, and Michael Papka. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. http://www.acl.lanl.gov/Viz/papers/pvtk/pvtkpreprint/.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, January 2006.

[4] Jérémie Allard. *FlowVR: calculs interactifs et visualisation sur grappe*. PhD thesis, INPG, 2005.

[5] Jérémie Allard, Edmond Boyer, Jean-Sébastien Franco, Clément Ménier, and Bruno Raffin. Marker-less Real Time 3D Modeling for Virtual Reality. In *Immersive Projection Technology Symposium (IPT'04)*, Ames, Iowa, May 2004.

[6] Jérémie Allard, Marcio C. Cabral, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 03, Course 18*, California, July 2003.

[7] Jérémie Allard, Stephane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Herve Delingette, and Laurent Grisoni. SOFA: an Open Source Framework for Medical Simulation. In *Medicine Meets Virtual Reality (MMVR)*, 2007. http://www.sofa-framework.org.

[8] Jérémie Allard, Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. The GrImage Platform: A Mixed Reality Environment for Interactions. In *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*, pages 46–52, New York, January 2006.

[9] Jérémie Allard, Valérie Gouranton, Gilles Lamarque, Emmanuel Melin, and Bruno Raffin. Softgenlock: Active Stereo and Genlock for PC Cluster. In *IPT & EGVE Workshop 2003*, pages 255–260, Zurich, Switzerland, May 2003.

[10] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Sébastien Limet, Emmanuel Melin, Bruno Raffin, and Sophie Robert. FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 497–505, Pisa, Italia, August 2004.

[11] Jérémie Allard, Valérie Gouranton, Loic Lecointre, Emmanuel Melin, and Bruno Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE Virtual Reality Conference*, pages 275–276, Orlando, USA, March 2002.

[12] Jérémie Allard, Valérie Gouranton, Emmanuel Melin, and Bruno Raffin. Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster. In *Immersive Projection Technology Symposium (IPT)*, Orlando, USA, March 2002.

[13] Jérémie Allard, Clément Ménier, Edmond Boyer, and Bruno Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In *IPT & EGVE Workshop 2005*, Denmark, October 2005.

[14] Jérémie Allard, Clément Ménier, Bruno Raffin, Edmond Boyer, and François Faure. Grimage: Markerless 3D Interactions. In *Proceedings of ACM SIGGRAPH 07*, San Diego, USA, August 2007. Emerging Technologies.

[15] Jérémie Allard and Bruno Raffin. A Shader-Based Parallel Rendering Framework. In *IEEE Visualization Conference*, pages 127–134, Minneapolis, USA, October 2005.

[16] Jérémie Allard and Bruno Raffin. Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[17] Jérémie Allard, Bruno Raffin, and Florence Zara. Coupling Parallel Simulation and Multi-display Visualization on a PC Cluster. In *Euro-par 2003*, Klagenfurt, Austria, August 2003.

[18] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent 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.

[19] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[20] Pilippe Augerat, Camille Goudeseune, Hank Kaczmarski, Bruno Raffin, Benjamin Schaeffer, Luciano Soares, and Marcelo K. Zuffo. Commodity Clusters for Immersive Projection Environments. In *Proceedings of ACM SIGGRAPH 02, Course 47*, Texas, July 2002.

[21] Françoise Baude, Denis Caromel, and Matthieu Morel. From Distributed Objects to Hierarchical Grid Components. In *CoopIS/DOA/ODBASE*, pages 1226–1242, 2003.

[22] Bruce G. Baumgart. *Geometric Modeling for Computer Vision*. PhD thesis, CS Dept, Stanford U., Oct. 1974. AIM-249, STAN-CS-74-463.

[23] 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.

[24] M. A. Bender and M. O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems Special Issue on SPAA '00*, 35(3):289–304, 2002.

[25] 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.

[26] A. Bierbaum1, P. Hartling, P. Morillo, and C. Cruz-Neira. Implementing immersive clustering with vr juggler. In *Computer Graphics and Geometric Modeling (TSCG 2005) Workshop*, volume 3482/2005 of *LNCS*, pages 1119–1128, 2005.

[27] 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.

[28] 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.

[29] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *IEEE Visualization 2005*, pages 191–198, October 2005.

[30] J. Clyne and J. Dennis. Interactive Direct Volume Rendering of Time-Varying Data. In *Eurographics Data Visualization '99 Conference*, pages 109–120, 1999.

[31] Muray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[32] D. Cotting, M. Waschbusch, M. Duller, and M. Gross. Winsgl: synchronizing displays in parallel graphics using cost-effective software genlocking. *Parallel Computing*, 33(6):420–437, 2007.

[33] C. Cruz-Neira, C. Just, K. Meinert, A. Bierbaum, P. Hartling, and B. Raffin. Open Source Virtual Reality. IEEE VR 2002 Tutorial, Florida, March 2002.

[34] 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.

[35] Isabelle Debled-Rennesson, Salvatore Tabbone, and Laurent Wendling. Fast Polygonal Approximation of Digital Curves. In *Proceedings of the 17th International Conference on Pattern Recognitio)*, volume I, pages 465–468, 2004.

[36] 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.

[37] Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM: an Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19(4):575–585, 2003.

[38] Eric Bruneton and Thierry Coupaye and Matthieu Leclercq and Vivien Quéma and Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java: Experiences with Auto-Adaptive and Reconfigurable Systems. *Software Practice & Experience*, 36(11-12):1257–1284, 2006.

[39] A. Esnard. *Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles*. Informatique, Université de Bordeaux 1, décembre 2005.

[40] David Foulser. IRIS Explorer: a Framework for Investigation. *Journal of ACM SIGGRAPH 95*, 29(2):13–16, 1995.

[41] Jean-Sebastien Franco and Edmond Boyer. Exact Polyhedral Visual Hulls. In *Proceedings of the British Machine Vision Conference, Norwich (UK)*, pages 329–338, Septembre 2003.

[42] Jean-Sébastien Franco, Clément Ménier, Edmond Boyer, and Bruno Raffin. A Distributed Approach for Real Time 3D Modeling. In *Conference on Computer Vision and Pattern Recognition Workshop (CVPRW) 2004*, pages 31–38, Washington, USA, July 2004.

[43] J.S. Franco and E. Boyer. Efficient Polyhedral Modeling from Silhouettes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[44] N. Férey, O. Delalande, G. Grasseau, and M. Baaden. A VR Framework for Interacting with Molecular Simulations. In *ACM VRST 2008*, Bordeaux, October 2008.

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[46] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23, New York, NY, USA, 2007. ACM.

[47] 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.

[48] R. L. Graham. Bound for certain multiprocessing anomalies. *Bell System Tech. J.*, pages 1563–1581, 1966.

[49] M. Gross, S. Wuermlin, M. Naef, E. Lamboray, C. Spagno, A. Kunz, E. Koller-Meier, T. Svoboda, L. Van Gool, K. Strehlke S. Lang, A. Vande Moere, and O. Staadt. Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 03*, San Diego, 2003.

[50] Jean-Marc Hasenfratz, Marc Lapierre, and François Sillion. A real-time system for full body interaction with virtual worlds. *Eurographics Symposium on Virtual Environments*, pages 147–156, 2004.

[51] Alan Heirich and James Arvo. A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. *The Journal of Supercomputing*, 12(1–2):57–68, 1998.

[52] Adrian Hilton and Jonathan Starck. Multiple view reconstruction of people. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 357–364, Washington, DC, USA, 2004. IEEE Computer Society.

[53] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of ACM SIGGRAPH 2001*, 2001.

[54] Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. In *Proceedings of ACM SIGGRAPH 02*, pages 693–702, 2002.

[55] Sylvain Jubertie, Emmanuel Melin, Jeremie Vautard, and Arnaud Lallouet. Mapping heterogeneous distributed applications on clusters. In *Europar*, August 2008.

[56] Takeo Kanade, Peter Rander, and P.J. Narayanan. Virtualized reality: Constructing virtual worlds from real scenes. *IEEE Multimedia, Immersive Telepresence*, 4(1):34–47, January 1997.

[57] Mustafa Karaman, Lutz Goldmann, Da Yu, and Thomas Sikora. Comparison of static background segmentation methods. In *Visual Communications and Image Processing (VCIP '05)*, Beijing, China, July 2005.

[58] 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.

[59] Myron W. Krueger, Thomas Gionfriddo, and Katrin Hinrichsen. Videoplace – an artificial reality. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 35–40. ACM Press, 1985.

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

[61] Jean-Denis Lesage and Bruno Raffin. A Hierarchical Component Model for Large Parallel Interactive Applications. *Journal of Supercomputing*, July 2008. Extended version of NPC 2007 article.

[62] Jean-Denis Lesage and Bruno Raffin. High Performance Interactive Computing with FlowVR. In *IEEE VR 2008 SEARIS workshop*, pages 13–16, Reno, USA, March 2008. Shaker Verlag.

[63] Ming Li, Marcus Magnor, and Hans-Peter Seidel. A hybrid hardware-accelerated algorithm for high quality rendering of visual hulls. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 41–48, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[64] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14:59–68, July 1994.

[65] 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.

[66] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *A Pattern Language for Parallel Programming*. Addison Wesley, 2004.

[67] W. Matusik and H. Pfister. 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. In *Proceedings of ACM SIGGRAPH 04*, 2004.

[68] Douglas B. Maxwell. Linux Cluster Powers Four-Wall 3D Display. *Linux Journal*, December 2002.

[69] Clément Ménier. *Systéme de vision temps-réel pour les interactions*. PhD thesis, INPG, 2007.

[70] Clément Ménier, Edmond Boyer, and Bruno Raffin. 3D Skeleton-Based Body Pose Recovery. In *International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'06)*, pages 389–396, 2006.

[71] 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.

[72] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics (ACM SIGGRAPH 97)*, pages 293–303. ACM Press, August 1997.

[73] Object Management Group. CORBA components specification, v. 3.0, June 2002. Document formal/02-06-65.

[74] OMG CCM Implementers Group. CORBA component model tutorial, April 2002. Document ccm/2002-04-01.

[75] Bruno Raffin and Luciano Soares. PC Clusters for Virtual Reality. In *IEEE Virtual Reality Conference*, pages 215–222, Alexandria, USA, March 2006.

[76] N. Richart, A. Esnard, and O. Coulaud. Toward a Computational Steering Environment for Legacy Coupled Simulations. In *Proceedings of 6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 319–326, Hagenberg, Austria, July 2007. IEEE Press.

[77] S. Robert and S. Limet. FlowVR-VRPN: a Generic VRPN/FlowVR Coupling for Interactive Applications. In *ACM VRST 2008*, Bordeaux, October 2008.

[78] B. Schaeffer and C. Goudeseune. Syzygy: Native PC Cluster VR. In *IEEE VR Conference*, 2003.

[79] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3), April-June 2002.

[80] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition*. Kitware, Inc., 2003.

[81] Jocelyn Serot and Dominique Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, December 2002.

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

[83] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, and Philip M. Papadopoulos. The optiputer. *Commun. ACM*, 46(11):58–67, 2003.

[84] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch. Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling. In *Eurographics 2008 Symposium on Parallel Graphics and Visualization (EGPGV'08)*, pages 273–274, Lugano,Switzerland, May 2007.

[85] Luciano P. Soares, Bruno Raffin, and Joaquim A. Jorge. PC Clusters for Virtual Reality. *The International Journal of Virtual Reality*, 7(1):67–80, March 2008. Extended Version of IEEE VR 20006 survey.

[86] R. Szeliski. Rapid Octree Construction from Image Sequences. *Computer Vision, Graphics and Image Processing*, 58(1):23–32, 1993.

[87] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *Super Computing*, 2006.

[88] 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.

[89] Jesus Alberto Verduzco. *Environnement X Window pour mur d'images*. PhD thesis, INPG, 2005.

[90] A. Wierse, U. Lang, and R. Rühle. Architectures of distributed visualization systems and their enhancements. In *Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, 1993.

[91] Xiaojun Wu, Osamu Takizawa, and Takashi Matsuyama. Parallel pipeline volume intersection for real-time 3d shape reconstruction on a pc cluster. In *Proceedings of the Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*, Washington, DC, USA, 2006. IEEE Computer Society.

[92] Uwe Wössner and Martin Aumüller. Software-based genlock for active stereo nvidia cards. http://www.hlrs.de/organization/vis/people/aumueller/genlock/.

[93] Keming Zhang, Kostadin Damevski, Venkatanand Venkatachalapathy, and Steven 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.

[94] Li Zhang and M. Parashar. Enabling efficient and flexible coupling of parallel scientific applications. In *Parallel and Distributed Processing Symposium (IPDPS) 2006.*, April 2006.

Appendix A

Selected Articles

A.1 Net Juggler IEEE VR 2002 Article

Net Juggler and SoftGenLock: Running VR Juggler with Active Stereo and Multiple Displays on a Commodity Component Cluster

Jeremie Allard Valérie Gouranton Loïck Lecointre Emmanuel Melin
Bruno Raffin
Université d'Orléans
Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
F-45067 Orleans Cedex 2, France
gouranton@lifo.univ-orleans.fr, melin@lifo.univ-orleans.fr, raffin@lifo.univ-orleans.fr

Abstract

High performance commodity components are available today to build clusters powerful enough to run high performance virtual reality applications. But cluster programming is time consumming and error prone. Virtual reality platforms that hide the complexity of the cluster architecture and provide automatic parallelization schemes are required.

We present in this paper two open source softwares, Net Juggler and SoftGenLock. The association of these libraries makes possible to run a VR Juggler application with active stereo and multi displays on a commodity component PC cluster. Net Juggler turns a cluster running the VR Juggler platform on each node into a single VR Juggler image cluster. Application parallelization is transparent to the user. SoftGenLock provides software genlock and quad buffer page flipped stereo, without requiring specific hardware or graphics card driver source code. The performed tests show these softwares do no introduce a significant overhead.

1 Introduction

Advanced visualization environments are increasingly used for immersive virtual reality, complex data visualization and collaborative work. These environments use multiple projectors to obtain high resolution and large display surfaces. For example, Caves [10] uses from 3 to 6 projectors to display images on some or all the walls of a cubic room, while the l-shaped display surface of a Workbench requires 2 projectors. Stereo head mounted displays can also be used in collaborative environments making 2 projectors per user. A large range of input devices can be used to interact with the application. The user being mobile in such environments, his head position has to be tracked to update the view of the displayed 3D scene accordingly. 3D mice or data gloves may be used to move or bend objects in the scene. The computing task includes collecting input data, updating the 3D scene and then rendering the scene, possibly in stereo, with several projectors. Computational load may be very important. Updating object properties may involve complex behaviors, when two objects are colliding for example. Latency between input events and image projections should be kept small and constant for a smooth interaction, imposing a real-time constraint. Thus, a high performance computer is required.

Today's advanced visualization environments are mainly driven by high-end specific architectures, which cost is prohibitive for most university labs or private companies. The cost of developing proprietary hardware and software, together with the relatively small size of the market partly explain such high prices. For example the SGI Onyx, probably the most used system, delivers high performance by making an intensive use of parallelism. The Onyx 2 [12] has multiple nodes, each node having 2 processors sharing a common memory. Nodes exchange date using a high bandwidth low latency CrayLink interconnect. The machine integrates specialized hardware to provide a global address space with a cache coherency protocol. The OS supports automatic thread distribution easing multi thread programming. The Onyx is equipped with one or more InfiniteReality graphics pipes [20], each one supporting multiple video outputs. A specialized network ensures the synchronization of the frame buffer swaps and the video retraces for the different video outputs. These two synchronization levels, usually called swaplock and genlock, are required in a multi display

context for a high quality image projection without drifting or tearing, and to make possible active stereo rendering.

Starting in the early 90's intensive research has been performed to decrease costs of parallel computers using commodity components and open source softwares. Today large computers built with PCs and gigabit interconnects compete with proprietary parallel machines [18] and are widely used for high performance scientific computing. Such PC clusters can be equipped with high performance graphics cards, based on NVIDIA or ATI graphics processors for example, which are now available at low cost. Nevertheless, using such an architecture to drive an advanced visualization environment faces several limitations.

Common PC graphics cards are not designed to have their video signal swaplocked and genlocked through an external synchronization signal. Known approaches to circumvent this limitation use specific high-end PC graphics cards supporting genlock an swaplock through a specialized synchronization network. We are aware of three of these approaches, the 3Dlabs Wildcat graphics card, the Quantum3D AAlchemy system, and the recently introduced SGI ImageSync technology associated with Vpro graphics cards. Some graphics clusters implement only a software swaplock using a classical communication network [7, 23, 1]. But active stereo rendering using shutter glasses synchronized on video retrace is not possible. A different technology must be used, like passive stereo, generally at the price of a lower quality.

Developing an application for a graphics cluster requires to distribute data and computation load on the different processors. The different video outputs must be coherent to have the different projectors displaying complementary parts of a large image. But parallel programming is time consuming and diverts the programmer from its main goal that is to develop innovative interactive graphics applications. Thus, the programming environment should hide the complexity of the underlying architecture and provide automatic parallelization schemes.

An other approach consists in duplicating the application on each node [21, 7, 8]. Every frame, time and input device data are broadcasted to each copy to guaranty data coherency. This parallelization is handled automatically by the execution environment, without requiring the application developer to be aware of the cluster architecture. Because data are duplicated and some computations are redundant, cluster resources are not optimally used. But compared to the previous approach the network is not the boottleneck. The required network bandwidth is limited due to the small amount of data communicated. Duplicating the application is efficient even for real-time applications with frequent updates of the scene. Note that existing environments require input devices to be on the same node, putting all the stress of input data acquisition on a single node.

In this paper, we extend this approach to turn a cluster running a VR system on each node into a single VR system image cluster. Our work is based on the VR Juggler. The open source VR Juggler library [6, 3] defines a execution platform for virtual reality applications. It provides an abstraction of the underlying system, while giving direct access to various graphics API for maximum control over applications. The application is independent of the displays, the input and output devices. System components are configured with a set of files when launching the application. VR Juggler also integrates dynamic reconfiguration mechanisms, allowing to add a display during the execution for example. Currently, VR Juggler can run on machines like multi-processor multi-pipe SGI Onyx machines, or single (possibly SMP) Linux or Windows boxes, but it does not support cluster configurations. VR Juggler does not support swaplock and genlock. It assumes the system will take care of it if required.

We implemented Net Juggler, a C++ library that turns a cluster running VR Juggler on each node into a single VR Juggler image cluster. Net Juggler does not modify the VR Juggler API. It ensures that all VR juggler operating functionalities are available. VR Juggler configuration mechanism is extended to cluster configuration. The user does not have to contact each node for configuration, which would have been error-prone and fastidious. Instead, the user gives configuration data to the single VR Juggler image that distributes these data to the concerned nodes. In contrast to previous approaches, input devices can be distributed on different nodes for more flexibility and a better use of the connectors available.

Because we first target commodity component clusters, we also developed a software approach for genlock and swaplock that does not require specific hardware. The swaplock synchronization is directly integrated into Net Juggler. The genlock algorithm is included in an external library, called SoftGenLock, currently implemented for Linux. SoftGenLock also implements software quad buffer page flipped stereo.

VR Juggler main features are first described in section 2 before to present Net Juggler in section 3. Section 4 presents the swaplock and genlock algorithms. Section 5 discusses performance results before to conclude in section 6.

2 VR Juggler

VR Juggler is an open source project started in 1997 at Iowa State University [6, 3]. VR Juggler is a virtual reality plateform hiding the underlying machine complexity and specificity. The application is independent of the displays, the input and output devices. For maximum control over applications the developer has direct access to

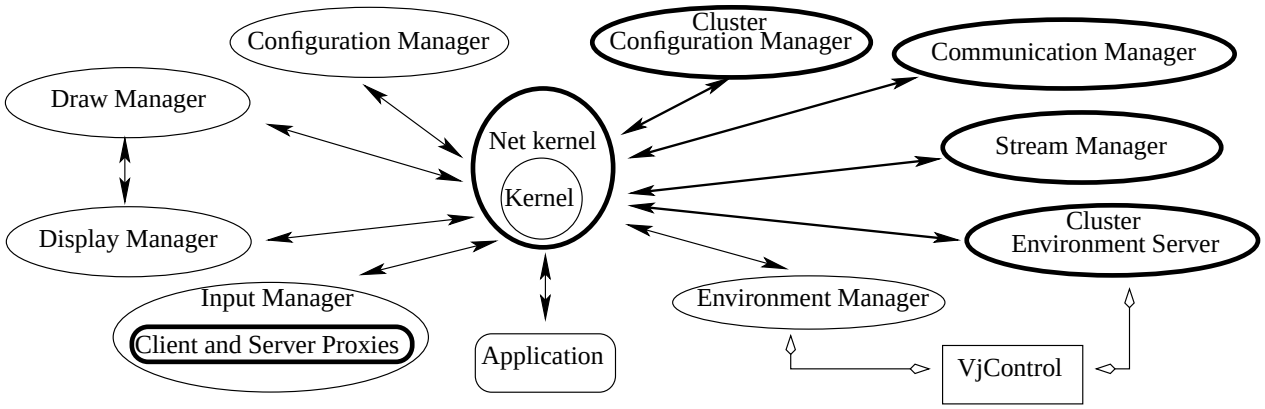


Figure 1. VR Juggler (thin lines) and Net Juggler (thick lines)

various graphics API, like Performer or OpenGL. Special care has been taken to enforce VR Juggler modularity and extensibility to ease addition of new functionalities and portability to different platforms. At this moment, VR juggler has been ported to numerous platforms, like IRIX, Linux, Windows or Solaris. It supports several tracking systems, gloves, and input devices and can probably control any display configuration, like head mounted display or 6 sided Caves.

VR Juggler allows every element of the system to be configured at run-time. For example, an input device can be started, reconfigured or stopped while the application is running. VR Juggler also integrates a tool for collecting performance data and visualize them at run-time.

VR Juggler is a C++ library. Its micro-kernel architecture is organized around a kernel and different components called managers (Fig. 1). Each manager handles a set of specific system details, while the kernel controls the whole run-time system and brokers communications between the different managers. Every input device is controlled by the input manager. When an application requests access to a device, it talks to a proxy. The proxy hides the actual device and tracks the most recent data received from the device. Each graphics API is encapsulated in a draw manager. The kernel instantiates only the draw manager required by the application. The display manager takes care of the windows and displays. The configuration manager handles a data base with configuration informations, like window properties, proxy names and associated devices. The environment manager is the user entry point to talk to VR Juggler. The environment manager collects configuration and performance data from the systems and forwards them to the user through a socket connection. The user also accesses this connection to send reconfiguration orders. A graphics utility, VjControl, is provided with VR Juggler for handling and visualizing data exchanged with the environment manager.

3 Net Juggler

The goal of the Net Juggler project was to design a software harness to run VR Juggler on a cluster. Net Juggler [4] was developed with the following in mind:

- A VR juggler application should not require any modification to run on a cluster.
- Launching the application and configuring the cluster should not require the user to access each node.
- All VR Juggler features, like run-time reconfiguration or performance data collection, should be available on a cluster.
- Net Juggler should be as transparent as possible. Any new feature that could be added in future VR Juggler releases, should be also available on a cluster, ideally with no porting effort.
- Net Juggler should respect VR Juggler micro-kernel architecture. Required modifications to VR Juggler code should be minimal.
- Net Juggler should be scalable and ensure high performance executions. In particular, communication and synchronization costs should be minimized.
- No cluster node should have a master position for better scalability. In particular, it should be possible to scatter input devices on different nodes.
- No specific hardware and driver source code should be necessary. Thus, Net Juggler should include support for a software swaplock. A software genlock and quad buffer page flipped stereo is also required.

3.1 Micro-Kernel Architecture

Adding cluster support to VR Juggler requires new functionalities. Following VR Juggler micro-kernel organization, we implemented new managers. A Net Juggler kernel derives from the VR Juggler kernel to handle them (Fig. 1).

3.2 Parallelization Paradigm

To run a VR Juggler application on a cluster we adopted a simple parallelization paradigm: each node of the cluster runs its own copy of the application with its own local parameters, like the viewport for instance. Obviously, input devices are not duplicated. Thus to ensure data consistency across the different copies, input events must be broadcasted to each node. This parallelization can easily be hidden from the user. It is scalable and ensures the amount of data to communicate is small.

The main drawback is that it can lead to redundant computations. But performance should not be affected for a large range of virtual reality applications. Future works will address this problem for very computation intensive virtual reality applications.

3.3 Sharing Inputs

A given input device is connected to a given node. Net Juggler must get the inputs from the device, and broadcast the collected data to each node of the cluster. VR Juggler manages each input device through a driver. This driver is connected to a proxy that forwards the data to the application. We could develop a server input driver for the node the device is connected to, and a client input driver for the other nodes. This solution was rejected because every single device driver would require a client and a server input driver. Instead, we have client and server proxies (Fig. 1). Proxies provide an abstraction of input drivers and thus their number is limited and should not increase significantly in the future.

3.4 Configuration Management

3.4.1 System Configuration

The VR Juggler system configuration is controlled by files given when starting the program, or by requests sent during the execution from VjControl. Configuration data are organized in chunks, each chunk having some informations about a part of the system. For configuring a Net Juggler cluster we just had to add a host parameter to each chunk pointing out the nodes the chunk should be applied to. We also defined a new type of chunk for client/server proxy couples. In this case, the host parameter has a different semantics: it points out the node that runs the server proxy, all the other nodes having a client proxy.

3.4.2 Processing Configuration Chunks

One cluster configuration manager stores configuration chunks in a data base on each node (Fig. 1). We want each node to know the whole cluster configuration to avoid to centralize configuration informations on one specific node or to have to handle scattered chunks when the user asks for the configuration. Each node processes the chunks to select the ones that must be applied locally and to generate VR Juggler chunks from Net Juggler specific chunks. The VR Juggler configuration manager maintains a data base of local chunks.

3.4.3 Dynamic Configuration

VjControl can connect to any node of the cluster running a cluster environment server (Fig. 1). Configuration requests are intercepted and broadcasted to all nodes before being stored in each local data base and processed to be applied localy. VR Juggler environment manager is still available to retrieve local data, like performance data. This open connection is also useful for debugging.

3.5 Communications

Communications must take place to broadcast configuration requests and input data. For performance purpose these data transfers must be carefully gathered.

3.5.1 Streams

We use and extend the classical stream paradigm to represent data communication between nodes. There is one stream by server proxy and by cluster environment server. A stream is associated to a specific node source and can have several destination nodes. Each stream is identified by a unique id and can be created, deleted or modified at runtime. The abstraction level provided by the streams hides the actual data movements that take place at a lower level. Stream related operations are performed by the stream manager (Fig. 1).

3.5.2 Messages

Data communications take place only once per frame. When a node writes into a stream, it builds a message containing the data and appends it to the buffer of pending messages. When the communication actually takes place each node broadcasts its buffer to each other node (allgather collective communication).

Configuration events can take place at any time and cause buffers to have an unpredictable size. The adopted semantics for the allgather requires all nodes to know the size of the messages they will receive. When the allgather is executed, it sends input data and a special message indicating the size of the reconfiguration data. If this size is different from 0 a second communication step is triggered to send the reconfiguration data.

3.5.3 Communication Implementation

Because only input events are sent over the network, bandwidth should not be a limiting factor. Synchronization barriers are mainly used for swaplock and genlock (see section 4). The genlock requires precise and fast barriers. As we will discuss it in section 5, a gigabit network like Myrinet, or a Fast Ethernet network associated with a specific fast synchronization network provide the required performance.

Net Juggler is designed to ease porting on top of various communication libraries. Communication routines are defined in an abstract communication manager class. Derived classes provide the actual implementation (Fig. 1). For the moment, one communication manager is implemented with the MPI [24] standard. MPI is widely used and many implementations are available, on top of standard protocols like TCP/IP [14] or high performance user-level protocols [5, 17, 13].

Depending on MPI implementations, collective operations may not be optimized for Net Juggler communication requirements. For example the allgather operation is typically implemented by having all processors shifting messages in a ring. This is efficient for large messages, but for small messages a gather followed by a broadcast is generally more efficient [19]. Several implementations of the broadcast and allgather collective operations are provided with Net Juggler so it can be easily tuned up for the network and protocol used.

4 Swaplock, Genlock and Active Stereo

Except for some high-end cards, off-the-shelf graphics cards do not support swaplock and genlock. Especially for Linux, today's graphics card drivers generally do not enable quad buffer page flipped stereo. These are important limitations for virtual reality PC clusters.

We describe in this section software solutions we developed to bypass these limitations. Swaplock support is integrated in Net Juggler. Genlock and active stereo support are gathered in the SoftGenLock library. Note that we developed solutions that do not require the source code of the graphics card driver.

For the moment, SoftGenLock has been ported on Linux. It should work with any graphics cards, but was only tested with Geforce cards.

4.1 Swaplock

Swaplock is obtained with a synchronization barrier that forces the different nodes to wait each other before to swap their frame buffers. This technique is used in other systems [7, 23] and proved efficient.

4.2 Active Stereo Support

Active stereo display requires the graphics card to compute two different images, one for each eye, and to display them alternatively, switching at each video retrace. Shutter glasses are synchronized with the retrace signal to ensure each eye only see its image.

4.2.1 Quad Buffering

XFree86 is set up to have a virtual buffer twice as large as the screen display. The 3D software (NET/VR Juggler) must then be set up to write the left eye image in the left half buffer and the right eye image in the righ half buffer. Each time a vertical retrace is detected, the displayed part of the buffer is changed and a signal is sent to the shutter glasses.

4.2.2 Vertical Retrace Detection

Because proprietary drivers do not let us have access to the vertical retrace interrupt, other approaches to detect and wait for the vertical retrace must be considered.

Most graphics cards are VGA compatible, having a status register and CRTC registers. The status register can be used to detect the vertical retrace. The CRTC registers can be used to modify the video signal.

We could poll the state of the status register to detect the vertical retrace, but this active waiting is CPU time consuming. To free the CPU, we use a real-time timer that is started after each vertical retrace. It is set up to wake up the SoftGenLock thread just before the next vertical retrace.

We use the RT-Linux system [2]. The high precision of RT-Linux timers permits to reduce the active waiting to a few tens of microseconds for each retrace. This overhead could be reduce by dynamically refining the SoftGenLock thread sleeping time.

4.2.3 Page Flipping

To alternate the image displayed, SoftGenLock modifies the display start address in the CRTC registers.

```
wait = false
frame = 0
Loop:
   Wait for vertical retrace
   Barrier
   t = barrier execution time
   set_display_starting address( (frame % 2) *  image_size)
   set_stereo_sync_signal (( frame % 2 ) ? right_eye : left_eye)
   frame = frame + 1
   if (t > too_long   and  wait == false) then
       Slow down video signal
       wait = true
   end
   if (t < small_enough   and wait == true) then
       Go back to normal signal speed
       wait = false
   end
end
```

Figure 2. SoftGenLock main algorithm for stereo and genlock

4.2.4 Stereo Sync Signal

To send the stereo sync signal to shutter glasses we developed two approaches:

- The signal is written to a parallel port register. Generally shutter glasses are not connected to the parallel port but it is easy to brew a home made adaptor.
- The stereo signal is written to the DDC SDA pin of the SVGA video port. For NVIDIA cards the DDC bit is set by writing into the CRTC register `0x3f`. A stereo enabler adaptor is then necessary to extract the signal and forward it to the glasses. Using the ELSA Revelator stereo enabler adaptor allows to connect any glasses that has a VESA standard 3-pin mini-DIN stereo connector.

4.3 Genlock

4.3.1 Algorithm

When a node detects the vertical retrace for its video cards, it starts a synchronization barrier and measures the time taken to proceed this barrier (Fig. 2). If the delay is considered too long the machine slows down its video retraces. When the delay is considered small enough video retrace goes back to its normal speed.

The genlock algorithm requires the following data :

- The time `sync_time` required to execute a barrier when all barrier calls are synchronized.
- The extra time `delay` introduced during one image retrace when the signal is slowed down.

The highest quality genlock is achieved by setting the variables `small_enough` to `sync_time` and `too_long` to `sync_time+delay`.

The synchronization should be short enough to leave enough time to flip the shutters before the next image retrace starts.

Note that active stereo does not require a perfect genlock. The signals should stay synchronized within a range that ensures the shutters flip without an eye seeing the wrong image.

4.3.2 Synchronization Barrier

We estimated a quality stereo display typically requires a synchronization barrier below 100 microseconds. Depending on the cluster size and the barrier implementation, commodity networks like Fast Ethernet or Myrinet, can fulfills this requirement. We also developed a low cost network that provides high performance and scalable synchronization barriers. Executing the synchronization on a dedicated network avoids to overload the communication network and to have the barrier perturbated by other communications.

This network, called ISP_PAPERS, is used for the genlock synchronization barriers, but can also execute the swaplock synchronization barrier. A node connects to the network through the parallel port. A 4 PC synchronization is achieved in less than 5 microseconds. Basic synchronization units can be tree assembled to connect an arbitrary large number of nodes with very high synchronization performance. Each extra tree level only adds a few nanoseconds to the synchronization barrier. ISP_PAPERS is based on

Test	Juggler	Network	Display	Nodes	Frames per second	Sync. and comm. time (pourcentage of the frame time)
Quake	VR Juggler	-	Mono	1	41	-
Quake	Net Juggler	ISP + Fast Ethernet	Mono	4	39.0	3.5%
Quake	Net Juggler	Myrinet	Mono	4	31.4	1.5%
Quake	VR Juggler	-	Stereo	1	31.1	-
Quake	Net Juggler	ISP + Fast Ethernet	Stereo	4	31.4	2.9%
Quake	Net Juggler	Myrinet	Stereo	4	32.6	0.5%
Fluid	VR Juggler	-	Mono	1	26.9	-
Fluid	Net Juggler	ISP + Fast Ethernet	Mono	4	27.6	3.9%
Fluid	Net Juggler	Myrinet	Mono	4	27.8	0.11%
Fluid	VR Juggler	-	Stereo	1	20.5	-
Fluid	Net Juggler	ISP + Fast Ethernet	Stereo	4	19.1	2.8%
Fluid	Net Juggler	Myrinet	Stereo	4	22.2	0.08%

Table 1. Performance results for Quake III Arena and the Fluid applications

the TTL_PAPERS network [11]. But ISP_PAPERS core component can be reprogrammed, adding versatility to the communication, computation and synchronization functions the network can support.

5 Performance

Net Juggler and SoftGenLock were tested on 4 dual Pentium III 800 Mhz PCs equipped with GeForce 2 GTS 64 MB DDR graphics cards. Three networks were available, a 100 Mbits/s Fast Ethernet network associated with the ISP_PAPERS synchronization network, and a 2 Gbits/s Myrinet 2000 network. Each node was running the Linux kernel 2.2.17.

Performance were measured with two different applications, Quake III Arena and an interactive real time fluid flow simulation we developed. These applications were using only one of the two processors present on each node.

For each application we measured the average frame rate and the average communication and synchronization time spent per frame for different configurations, using VR Juggler or Net Juggler, mono or stereo display. As input devices both applications were using a keyboard, a mouse and a time server.

When using VR Juggler, the application ran on a single node driving a single 1024×768 resolution display. When using Net Juggler, the application ran on four nodes, each one driving a 1024×768 resolution display, making four times the resolution of the VR Juggler configuration. Communications and synchronization used either the Fast Ethernet and ISP_PAPERS networks or the Myrinet network. With the Fast Ethernet network Net Juggler used mpich-1.2.4 over TCP/IP and SoftGenLock the ISP_PAPERS network. With Myrinet Net Juggler and SoftGenLock used mpich-1.2..5 over gm-1.4pre51.

Results (Table 1) show no significant performance degradation using Net Juggler alone or with SoftGenLock. The communication and synchronization time represents less than 4% of the total frame time for both network configurations. This time depends on the number of nodes, the number and distribution of input devices, but it does not depend on the scene complexity. Communication performance when using the Fast Ethernet network could be increased using mpich on top of a user-define protocol like Gamma [9] instead of TCP-IP. We did not test this configuration, because Gamma was unstable with the Intel 82559 NIC our nodes were equipped with.

6 Conclusion

High performance commodity components are available today to build clusters powerful enough to run high performance virtual reality applications. But software solutions are required to provide an abstraction level on top of these machines to ease development, portability, cluster configuration and execution control. Platforms like VR Juggler [6] or Maverik [15] are designed with that goal but do not support clusters yet.

We presented in this paper two open source softwares, Net Juggler and SoftGenLock. The association of these libraries makes possible to run a VR Juggler application with active stereo and multi displays on a commodity component PC cluster. Net Juggler turns a cluster running VR Juggler

on each node into a single VR Juggler image cluster. Application parallelization is transparent to the user. Cluster configuration and execution control is done accessing the single image system and not each node individually. Software swaplock support is integrated in Net Juggler. SoftGenLock provides software genlock and quad buffer page flipped stereo, without requiring specific hardware or even graphics card driver source code. The performed tests show these softwares do no introduce a significant overhead.

The parallelization scheme adopted is based on running a copy of the application on each node. Data are duplicated and some computations are redundant. For some applications this can be a memory and performance bottleneck that we will address in future works.

References

[1] MetaVR Web Server. www.metavr.com.

[2] RT Linux Web Server. www.rtlinux.org.

[3] VR Juggler Web Server. www.vrjuggler.org.

[4] J. Allard, L. Lecointre, V. Gouranton, E. Melin, and B. Raffin. Net Juggler Guide. Technical Report 2001-02, Laboratoire d'Informatique Fondamentale d'Orléans, Orléans, France, June 2001.

[5] R. A. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[6] 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.

[7] M. Bues, R. Blach, S. Stegmaier, U. Häfner, H. Hoffmann, and F. Haselberger. Towards a Scalable High Performance Application Platform for Immersive Virtual Environements. In J. D. B. Fröhlich and H.-J. Bullinger, editors, *Immersive Projection Technology and Virtual Environements 2001*, pages 165–174, Stuttgart, Germany, May 2001. Springer.

[8] Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, and K. Li. Software Environments for Cluster-based Display Systems. http://www.cs.princeton.edu/omnimedia/papers.html, 2001.

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

[10] 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.

[11] H. G. Dietz, R. Hoare, and T. Mattox. A Fine-Grain Parallel Architecture Based On Barrier Synchronization. In *Proceedings of the International Conference on Parallel Processing*, pages 247–250, 1996.

[12] J. Fier. *Performance Tuning Optimization for Origin 2000 and Onyx 2*. Silicon Graphics, 1996. http://techpubs.sgi.com.

[13] 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.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. http://www-c.mcs.anl.gov/mpi/mpich/.

[15] R. Hubbold, J. Cook, M. Keates, S. Gibson, T. Howard, A. Murta, A. West, and S. Pettifer. GNU/MAVERIK: A mirco-kernel for large-scale virtual environements. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 66–73, London, UK, December 1999.

[16] G. Humphreys and P. Hanrahan. A Distributed Graphics System for Large Tiled Displays. In *Proceedings of IEEE Visualization'99*, 1999.

[17] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, 1997.

[18] G. R. Luecke, B. Raffin, and J. J. Coyle. Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600. In *Proceedings of the IEEE International Workshop on Cluster Computing (IWCC'99)*, pages 26–35, Melbourne, Australia, December 1999.

[19] G. R. Luecke, B. Raffin, and J. J. Coyle. The Performance of the MPI Collective Communication Routines for Large Messages on the Cray T3E600, the Cray Origin 2000, and the IBM SP. *The Journal of Performance Evaluation and Modelling for Computer Systems*, July 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[20] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics (SIGGRAPH 97)*, pages 293–303. ACM Press, August 1997.

[21] D. Pape, C. Cruz-Neira, and M. Czernuszenko. *CAVE User's Guide*. Electronic Visualization Laboratory, University of Illinois at Chicago, 1997.

[22] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (SIGGRAPH 94)*, pages 381–394. ACM Press, July 1994.

[23] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. In ACM, editor, *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–144, Los Angeles, USA, August, 1999.

[24] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI, The Complete Reference*. Scientific and Engineering Computation. The MIT Press, 1996.

A.2 SoftGenLock IPT 2003 Article

Softgenlock: Active Stereo and GenLock for PC Cluster

Jérémie Allard[1]

Valérie Gouranton[2]

Guy Lamarque[3]

Emmanuel Melin[2]

Bruno Raffin[1]

[1] Laboratoire Informatique et Distribution (ID)
CNRS - INPG - INRIA - UJF
38330 Montbonnot, France
[2] Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Université d'Orléans
F-45067 Orleans Cedex 2, France
[3] Laboratoire d'Electronique, Signaux, Images (LESI)
Université d'Orléans
F-45067 Orleans Cedex 2, France

Abstract

In this paper, we present SoftGenLock, an open source software that enables genlock and active stereo on commodity graphics cards. SoftGenLock is implemented on top of Linux. It does not require any hardware modification of the graphics card. Rather than to gain total control on signal generation, which would make the software deeply dependent on the graphics card specification, SoftGenLock applies continuous small modifications to converge and maintain genlocked video signals. To be properly synchronized with each video retrace, SoftGenLock is executed as a real-time task. The genlock signal is propagated along the different machines using the parallel port, a low latency device present on all PCs. It results in a software that only requires access to few specific registers on a graphics card: it can be ported with minimal effort on potentially any graphics card.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics H.5.1 [Information Interfaces and Presentation]: Artificial, augmented, and virtual realities

Keywords: Active Stereo; Genlock; Real-time; Immersive Projection Environment; PC Cluster.

1. Introduction

Immersive environments [12] are classically powered by dedicated graphics supercomputers like SGI Onyx machines [15]. However, today the anatomy of super-computing is quickly and deeply changing. Clusters of commodity components are becoming the leading choice architecture. They are scalable and modular with a high performance-price ratio. Clusters range from a few low-end machines connected through an Ethernet network to thousands of 64-bit processors using Myrinet or Quadrix interconnects. These architectures proved efficient for classical (non interactive) intensive computations. Recently the availability of low cost high performance graphics cards have foster researches to use clusters to run interactive virtual reality applications in multi-projector immersive environments [6, 8, 14, 16].

One difficulty is to harness the distributed resources (CPU, memory, GPU, projector) such that the images displayed with the different projectors appear as a single high

resolution image. Three levels of synchronization can be identified:

- *DataLock* ensures the images on each node are computed on coherent data sets. For example, the user position should be the same for all nodes.
- *SwapLock* ensures the images computed on each node are released at the same time, i.e. the buffer swaps are synchronized.
- *GenLock* ensures video signals are synchronized. Synchronization can occur at a pixel level, line or frame level. In this paper we consider a frame level genlock.

Several approaches have been developed to ensure proper datalock and swaplock. A master server can ensure datalock by broadcasting graphics primitives to the render nodes [14]. An other approach consists in duplicating data on each node at starting time and broadcasting any subsequent data modification event [6, 11, 17]. A synchronization barrier, executed on a classical Fast Ethernet network just before the buffer swap, is sufficient to ensure a proper swaplock [6, 11, 17]. Genlock, as it concerns video signal generation, is close to the hardware and then much more difficult to ensure with a software only approach. Some vendors, like 3DLabs or nVIDIA (the recently announced quadro FX), develop high-end graphics cards with built-in support for genlock. Other companies like Artabel, Orad or MetaVR, modify existing graphics cards. However, main commodity graphics cards from nVIDIA, ATI or Matrox, do not support genlock. There is no strong technical limitation to provide this feature at a hardware level. It is rather an economical reason as the very small size of the market does not justify the extra cost incurred : extra hardware, extra connector for genlock signal, extra space to locate this connector on the graphics card, and drivers that support it.

A frame level genlock is mandatory for active stereo [8]. Left and right eye images are displayed alternatively. Swap occurs during the vertical blanking, i.e. when no pixel is generated before the next video retrace starts. The user wears glasses with shutters opening alternatively to let him see the right eye images with the right eye only and vice-versa. If genlock is not properly ensured, the user may see from the same eye both, a right and a left eye image displayed by two different projectors. The quality of stereo is then affected.

The goal of the open source SoftGenLock [7] project was to develop a software approach to enable genlock and active stereo on commodity graphics cards. SoftGenLock, developed for Linux, was first released in June 2001. It has been designed to minimize dependencies on graphics cards. It does not require any hardware modification of the graphics card. The genlock signal is propagated along the different machines using the parallel port. Thus no direct wiring of the graphics card is required. Genlocking and active stereo requires a strong synchronization with video retrace. This is usually ensured at a hardware level. SoftGenLock avoids this dependency by taking advantage of real-time systems like RTAI [4] or RTLinux [9]: SoftGenLock is executed as a real-time task. For stereo, SoftGenLock counts on X to setup a virtual screen twice as large as the actual screen, that can contain side by side the left and right eye images.

During video blanking, it just has to swap the address of the half screen to display (the left or right eye image). The switching signal for the shutter glasses is sent through the parallel port. Genlocking is achieved by applying continuous small modifications to the video signal, adding or removing hidden pixels for example, an approach that requires a limited control on the signal generation. As a result, SoftGenLock only requires access to few specific registers on a graphics card: to detect a specific position of the video retrace, to apply small modifications to the video signal and to change the address of the buffer to display. Thus, supporting a new graphics card does not require a full code rewrite but only to support the new set of registers. Given a family of cards, vendors usually ensures an upward compatibility. In this case, SoftGenLock code is identical for all the family. This is for example the case of all cards based on nVIDIA Geforce/Quadro GPUs.

SoftGenLock design and algorithm are discussed first in section 2. Specific features are detailed in sections 3, 4, 5, 6 and 7, before to end with results in section 8.

2. SoftGenLock Design and Algorithm

Rather than to gain total control on signal generation, SoftGenLock applies continuous small modifications to converge and maintain genlocked video signals. This approach requires minimal control over the graphics card. We basically need two features: a way to inflect the signal generation speed, and a way to periodically detect the signal position (once per frame for a frame level genlock). The former can be obtained by modifying the speed of the pixel clock or the number of pixels to generate. The latter, we call it the *sync event*, is implemented using the vertical blanking interrupt or by pooling a CRTC state register [1]. Information must be exchanged between machines to measure the time gap between the different sync events. The video signals will be considered genlocked if the time gaps are small enough (usually 5-40 μs for good stereo quality). Based on this measure each machine can locally decide to slow down or accelerate the video signal to reduce this time gap. Amongst the machines one is considered the reference: the master. It never modifies its video-signal. Each other machine, the slaves, measures the time gap between the master sync event and its own sync event and corrects the video signal speed if required.

To obtain a genlock and stereo of quality, SoftGenLock must be executed in real-time. If glasses switch a few hundreds of microseconds after the expected time, because an interruption suspended SoftGenLock execution for example, the stereo quality is affected (switch occurs during video retrace). As we target a genlock with a discrepancy of about

5-40 μs, the time gap must be measured with a precision of a few microseconds. The Linux kernel [10] does not guarantee a process will not be interrupted or will be granted CPU access with a short response time (few μs). The response time of the Linux Kernel 2.4 can be as large as 10 ms. Thus, Linux does not allow SoftGenLock to measure the time gap with the required precision. To satisfy these real-time constraints we run SoftGenLock as a RTLinux [9] or RTAI [4] task.

Before SoftGenLock starts to genlock the video signals and activate stereo, it goes through a calibration step and a local synchronization step. During the calibration step, SoftGenLock evaluates different times, like the time gap between two consecutive sync events. During the local synchronization step, SoftGenLock calibrates its timers to wake up when needed during the video blanking. The goal is to minimize busy waiting to avoid monopolizing a CPU. Next, once per video retrace, SoftGenLock executes the following tasks (the order may change depending on the implementation):

- The master sends a signal to the slaves a soon as it detects its sync event.
- Each slave detects the time t_m it receives the signal from the master. It subtracts from t_m the expected signal transmission (user given, generally about 5μs).
- Each slave measures the time t_l the local sync event occurred.
- Each slave modifies the video signal generation speed if $|t_m - t_l|$ is considered too large (usually about 5-40 μs).
- The machine driving the shutter glasses sends the switching signal.
- All machines switch the address of the image to display (left or right eye image).

3. Real-time System

SoftGenLock is implemented as a real-time task using RTLinux [9] or RTAI [4]. These add-ons to the Linux kernel ensures real-time tasks are triggered with micro-second response times and are not interrupted without their consent.

RTLinux and RTAI share the same main concepts. The real time system implements a basic kernel that handles two kinds of tasks: the Linux kernel and real-time tasks. The real-time kernel does not implement preemptive multi-tasking. Task scheduling must be programmed at the task level. This is the responsibility of the programmer to ensure two tasks do not try to grab the same CPU at the same time. Classically, real-time tasks are activated through a timer interruption: the real-time kernel intercepts the interruption and activate the real-time task that is waiting for that interruption. The task releases the CPU when it is done. No other task will be scheduled during that period of time. The real-time kernel has been designed to ensure the delay between the hardware interruption and the task activation is as small as possible, generally about a microsecond. The Linux kernel is the lowest priority task. It has access to the CPU only when no real-time task asks for it. Hardware interruptions for the Linux kernel are intercepted and stored by the real-time kernel as long as the Linux kernel does not have access to a CPU.

SoftGenLock is programmed as a real-time task. There is no time sharing difficulty as it is the only real-time task. SoftGenLock reprograms the timer interruption to be activated only when required. As a result, its only uses about 50μs of CPU time per video retrace. The guaranteed microsecond reactivity of real-time tasks and the non-preemptive time sharing ensures SoftGenLock actions are all performed on time.

Real-time execution allows a fine-tuning of event triggering, making it possible to optimize genlock and stereo quality for each configuration. For example shutter latency may vary from one model to the other. SoftGenLock can precisely control when the shutter switching signal is sent to minimize ghosting: glasses should switch as late as possible to avoid the opening shutter let the user see the image just drawn, but not too late to ensure the other shutter is closed when the next video retrace starts.

4. Sync Event Detection

SoftGenLock supports two different approaches. The first one detects the vertical blanking pooling the VGA Input Status #1 register [1] (3DAh). To avoid monopolizing a CPU, SoftGenLock uses a real-time timer to wake up just before the sync event is expected. The second approach uses the vertical blanking interrupt. The interrupt is intercepted by RTAI that wakes-up SoftGenLock within about a microsecond. Next, RTAI forwards the interrupt to the Linux kernel as it may be required by the graphics card driver. This second approach only supports nVIDIA cards for the moment.

5. Communication Network

SoftGenLock requirements for the communication network are the following :

- The master has to send a signal (one bit) to the slaves every video retrace.
- The communication time of the signal should not suffer unpredictable variations. A slave must be able to know the time the signal was sent to deduce the time the sync event was detected on the master slave.
- The network should be accessible from a real-time task.

Using a parallel port based network appeared as the best solution. Parallel port is still present at no extra cost on all PCs. The parallel port can be accessed from a real-time task. The signal (a bit) is directly written on a parallel port register and transformed into an electrical signal, without having to go through a protocol layer [2]. It ensures fast transmission times (about 5μs to send a bit) and low variations. Notice that classical high performance networks (Giga Ethernet, Myrinet or SCI for example) are not accessible from

RTAI or RTLinux tasks. This would require specific real-time drivers that are not available.

The parallel port has 8 pins (pins 2-9) that the master can use to send data [2]. On one pin, it is possible to connect several slaves. We did not make intensive tests, but it seems reasonable to connect 4 slaves per pin without requiring any signal amplifier. Thus, we can easily build a cheap and electronics free network for 33 nodes (one master, 8×4 slaves). By default the master writes the signal on all 8 output pins. This is useful as it allows to add or remove slaves online.

We also developed an advanced network based on the TTL_PAPERS design [13]. Using a reprogramable chip, it can implements more complex tasks than just to propagate a signal. The idea behind this advanced network is to have a reconfigurable networks to use SoftGenLock like algorithms for other tasks, like swaplocking or clock synchronization for example.

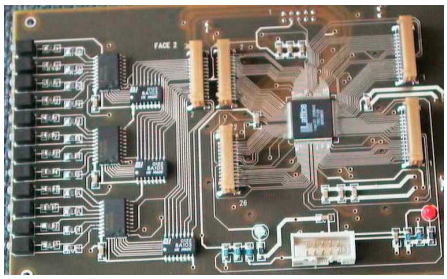


Figure 1: *The basic board with a 2096VE device (front and back side)*

The basic circuit board (Fig. 1) is four layered, the two internal layers are reserved for ground and power planes. The other layers are fully available for signal routing. The architecture is based around two CPLD (Complex Programmable Logic Device) ispLSI 2096VE, one on each side of the board. These devices have 96 general purpose Input/Output pins. These non volatile CPLD are electrically erasable and system programmable. The basic unit of logic on the ispLSI 2096VE devices is the Generic Logic Block (GLB). There are a total of 24 GLBs in the 2096VE device. Each GLB has a programmable AND/OR/Exclusive OR array. Each outputs can be configured to be either combinatorial or registered (96 registers). Even though these devices have a 3.3V low voltage architecture, the signal levels are TTL compatible with standard 5V TTL devices: they are compliant with the IEEE 1284 standard for parallel port. It is possible to interconnect several boards for large clusters. Each additional board only increases communication times by a few nanoseconds.

The hardware design can be programmed with tools like Very high speed Integrated Circuit (VHSIC) or Hardware Description Language (VHDL) for example. Once the program written, a file is generated and loaded to the non volatile Electrically Erasable Programmable Read Only Memory (EEPROM) architecture of the ispLSI.

6. Video Signal Control

Once we determined the time gap between the master sync event and the local sync event, corrections must be applied to accelerate or slow-down the video signal generation. The alteration of the video signal must be done carefully, otherwise the projector detects it : during a few milliseconds it does not display anything to reinitialize its state (the phenomena is classically experienced when changing the resolution of a CRT screen).

Depending on the characteristics of the graphics card, two approaches are supported.

6.1. VGA compatible cards

The VGA standard [1] specifies the registers that control the video signal. Some registers control the number of lines and columns of pixels (CRTC registers 00/06). SoftGenLock accesses these registers to add or remove pixels. When small modifications are required, hidden pixels are added/removed by adding/removing a few columns to some hidden lines. For larger modifications an extra blank line is added (hidden pixels too).

This approach can be used for any graphics card VGA-compatible on register level. It has been successfully used on nVIDIA and S3 cards. Unfortunately, several graphics cards are not compliant (Voodoo, ATI).

As VGA is an old standard, it has several limitations. It only supports one video head on multi-head cards, and some high-resolution modes are not supported. Another important draw-back is the possible interference with the video driver. As SoftGenLock is designed no to require access to the video driver's source code, mutual exclusion between the driver and SoftGenLock cannot be ensured. Accessing a VGA register is not atomic. It is a 2-step operation: first setting the index of the register and then reading/writing the value. Conflicts can occur when SoftGenLock and the driver are concurrently accessing the registers. This may lead to occasional corruptions of the VGA registers, which can lead to a corrupted display, X server crashes, or even hard lock of the entire system. Depending on the hardware environment, these problems can be quite rare (only after several hours of operation), or they can occur after only a few minutes. Resolving this issue require either more knowledge of the graphics card hardware (see next section) or to modify the driver to implement a lock that would guarantee mutual exclusion.

6.2. Pixel Clock Access

The pixel clock is the hardware component used to activate the output of each pixel of the video signal. Depending on

the graphics card model, it is possible to alter its speed. This requires knowledge on the hardware, but provides a very efficient way to control the video signal. For the moment only nVIDIA cards are supported, but this approach can easily be adapted to other models.

The pixel clock is based on a static base clock divided by a programmable factor. On nVIDIA cards, a register (offset 0x680508 in the MMIO area) contains 3 parameters M, N and P, which are used to specify the frequency using the following formula:

$$\text{Pixel Frequency} = \frac{\text{BaseFreq} * N}{M * 2^P}$$

To sync the master sync event and the local sync event, SoftGenLock modifies this register for a small period of time during the vertical blanking.

This approach supports any resolution. Both video heads can be controlled. Writing in this register is atomic so there is no concurrency issues.

7. Active Stereo

Figure 2: *A parallel cable to genlock 2 machines. Beside the two DB25 connectors, we can distinguish a female mini-din 3 connector to plug the IR emitter of the shutter glasses.*

Active stereo requires displaying alternatively the left eye and right eye images with shutter glasses masking the user's opposite eye. Image and shutter switching must occur before each video retrace:

- *Glasses Control:* The signal to control the shutter glasses is written on a pin of the parallel port and forwarded to the glasses with an adapted connector (Fig 2). It can be written at a different address if, for example, the graphics card has a dedicated output for the glasses. The time the signal is sent is critical to ensure the wrong image is not seen while shutters are switching. A microsecond precision is reached using RTAI or RTLinux. Experience shows that stereo signal should be sent about 700 μs before the next video retrace starts (this may vary depending on the glasses).
- *Image Switching:* SoftGenLock provides left/right image switching without requiring any specific hardware or driver support. XFree86 is set to have a virtual screen twice as large as the actual screen (a simple modification in the XF86Config file). We thus have a double size frame buffer. The 3D software must be configured to write the left eye image on the left half screen and the right eye image on the right half screen. This can be done through a simple modification of configuration files with Net Juggler [6]. During the vertical retrace, SoftGenLock switches the starting address of the half screen to display. This is done writing into a VGA register (CRTC registers 0C/0D) or a nVIDIA-specific register.

8. Results

The current implementation supports all cards having VGA registers. SoftGenLock is mainly used with nVIDIA cards (Geforce 2, Quadro 2, Geforce 4, Quadro 4), but was also tested with other VGA cards from S3 for example. It has been used for genlocking and active stereo with up to 6 machines. It requires about 50μs of CPU time per video retrace. SoftGenLock can be adapted to use non VGA registers, assuming the required hardware specification is available. We led developments to support specific nVIDIA registers. Only one output is VGA compliant on nVIDIA cards (the DVI one on GeForce 4 models). Using nVIDIA registers, SoftGenLock proved more reliable and can use the second output (one at a time for the moment, both at the same time in the future). Other non VGA cards, in particular from ATI, Matrox or 3Dlabs, are not supported for the moment.

Hardware specifications required to port SoftGenLock on a new graphics card are basics features that can be found in the 2D open source drivers that are usually available on Linux. We ported SoftGenLock on nVIDIA graphics cards using the informations from the nv driver of The XFree86 Project [5] and the RivaTV [3] developer resources.

9. Conclusion

Today's commodity graphics cards have reached a level of quality that allow them to be used in immersive environments. However, multi-projector displays and active stereo require genlocked video outputs, a feature only supported by some high-end graphics cards. In opposite to these dedicated hardware approaches, SoftGenLock minimizes hardware dependencies by implementing a genlock algorithm on top of real-time Linux systems. It results in a software that can enable genlock and active stereo for potentially any graphics card, assuming the required hardware specification is available.

Future works will investigate possibilities to support directly OpenGL stereo mode. Two directions will be investigated: to associate SoftGenLock with the stereo mode some drivers support, or to emulate quad-buffer stereo using an OpenGL wrapper library.

References

1. Hardware Level VGA and SVGA Video Programming Information Page. http://web.inter.nl.net/hcc/S.Weijgers/FreeVGA/home.htm.
2. Interfacing the Standard Parallel Port. http://www.beyondlogic.org/spp/parallel.htm.
3. RivaTV. http://rivatv.sourceforge.net/.
4. The RTAI Manual. http://www.aero.polimi.it/ rtai/.
5. The XFree86 Project. http://www.xfree86.org/.
6. 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.
7. J. Allard, V. Gouranton, E. Melin, and B. Raffin. Soft-GenLock Manual: Software Active Stereo and Genlock for Linux, 2002. http://netjuggler.sourceforge.net.
8. 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.
9. M. Barabanov and V. Yodaiken. Real-Time Linux, 1996. http://www.fsmlabs.com.
10. D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.
11. M. Bues, R. Blach, S. Stegmaier, U. Häfner, H. Hoffmann, and F. Haselberger. Towards a Scalable High Performance Application Platform for Immersive Virtual Environements. In *Immersive Projection Technology and Virtual Environements 2001*, pages 165–174, Stuttgart, Germany, May 2001. Springer.
12. 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.
13. H. G. Dietz, R. Hoare, and T. Mattox. A Fine-Grain Parallel Architecture Based On Barrier Synchronization. In *Proceedings of the International Conference on Parallel Processing*, pages 247–250, 1996.
14. G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of ACM SIGGRAPH 01*, 2001.
15. J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of ACM SIGGRAPH 97*, pages 293–303. ACM Press, August 1997.
16. R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.
17. B. Schaeffer. Networking Management Framewoks for Cluster-Based Graphics. http://www.isl.uiuc.edu/ClusteredVR/ClusteredVR.htm, 2002.

A.3 FlowVR Render IEEE Vis 2005 Article

A Shader-Based Parallel Rendering Framework

Jérémie Allard* Bruno Raffin†

ID-IMAG, CNRS/INPG/INRIA/UJF
Grenoble - France

ABSTRACT

Existing parallel or remote rendering solutions rely on communicating pixels, OpenGL commands, scene-graph changes or application-specific data. We propose an intermediate solution based on a set of independent graphics primitives that use hardware shaders to specify their visual appearance. Compared to an OpenGL based approach, it reduces the complexity of the model by eliminating most fixed function parameters while giving access to the latest functionalities of graphics cards. It also suppresses the OpenGL state machine that creates data dependencies making primitive re-scheduling difficult.

Using a retained-mode communication protocol transmitting changes between each frame, combined with the possibility to use shaders to implement interactive data processing operations instead of sending final colors and geometry, we are able to optimize the network load. High level information such as bounding volumes is used to setup advanced schemes where primitives are issued in parallel, routed according to their visibility, merged and re-ordered when received for rendering. Different optimization algorithms can be efficiently implemented, saving network bandwidth or reducing texture switches for instance.

We present performance results based on two VTK applications, a parallel iso-surface extraction and a parallel volume renderer. We compare our approach with Chromium. Results show that our approach leads to significantly better performance and scalability, while offering easy access to hardware accelerated rendering algorithms.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

Keywords: Distributed Rendering; Shaders; Volume Rendering

1 INTRODUCTION

Context. In recent years, graphics clusters have become a platform of choice for high performance scientific visualization. The goal is to interconnect multiple PCs through a dedicated network to aggregate the power of their disks, memories, CPUs and GPUs. Such cluster can drive from a single screen to a display wall of several projectors allowing a higher brightness, resolution and display size. The difficulty is then to develop software solutions to efficiently take advantage of such platforms.

Graphics applications, like scientific visualization ones, are often structured as a pipeline. A raw data set is read either from disk or from a live simulation application. It is processed to obtain a graphical representation. This scene is then rendered to an image that is finally displayed. Each of these stages can be distributed differently on a cluster. For example, the final display can be connected to a single host, or each host can drive one video projector of a large display wall. Each host can render the pixels it displays, or it can render pixels to be displayed by different hosts. In this case pixels have to be communicated on the network and recomposed to form the final images [20]. This scheme is usually called a sort-last approach [18]. The graphics objects can also be produced with a different distribution scheme than the one adopted for rendering (sort-first approach). When the higher-level stage is distributed, multiple data streams have to be merged, and similarly scatter operations are required when distributing lower-level stages. The scalability of the system depends on the overhead introduced by these operations and the volume of communications.

Implementing higher level communications is generally more dependent on the application. Sort-last solutions are highly generic, as pixel format does not change. Sort-first frameworks depend on the rendering API used in the application to describe the scene. Most applications use OpenGL.

Humphreys et al. [10] proposes a framework called Chromium. Chromium uses a protocol for distributing OpenGL commands. However, due to OpenGL's history and the requirement to support legacy applications, these commands are numerous and often redundant. For example, OpenGL supports both immediate-mode primitives and retained-mode objects (using several concepts such as display lists, vertex array, buffer objects). Immediate-mode rendering does not allow to easily detect changes between frames, introducing duplicated communications or a high computational overhead to detect unchanged primitives. Similarly, high level informations such as bounding boxes are not available. It makes it difficult to perform efficient scatter operations (i.e. frustum culling) to different render hosts. Moreover, as OpenGL is based on a sequential state machine, commands must respect a strict ordering. Merging multiple streams together requires Chromium to track state changes and to use special commands defining the relative ordering of each stream.

All these constraints are driven by the need to support legacy OpenGL applications. In the context of scientific visualization where performance is critical and the final rendering is often handled by a shared toolkit, considering alternative solutions to an OpenGL based protocol is relevant.

In the rest of this paper we will refer to the programs used in the rendering stage of the pipeline as *renderers*, and the programs responsible to create the graphics objects will be called *viewers*.

Contribution. We propose a sort-first retained-mode parallel rendering framework called *FlowVR Render*. Instead of relying on OpenGL commands, we define a *shader based protocol* using independent batches of polygons as primitives. This protocol offer the following benefits:

- Shaders are used to specify the visual appearance of graphics objects. They require only a few parameters and not the full complexity of the fixed-function OpenGL state machine. It leads to a simpler protocol that does not have to manage state tracking.
- Shaders enables to easily take advantage of all features offered by programmable graphics cards.

*e-mail: Jeremie.Allard@imag.fr
†e-mail:Bruno.Raffin@imag.fr

- Primitives are un-ordered unless explicitly stated by viewers (for transparent objects or user-interface overlays for instance). It enables renderers to optimize the local rendering order, reducing shader and texture switches for example. It also eases merging and reordering primitives coming from multiple streams.
- A higher level information such as bounding-boxes or changes since the last frame can be directly specified by viewers, avoiding unnecessary processing and network traffic. In most cases this information is already available (visualization toolkits often detect changes between frames to recompute only the affected data).

We base our design and implementation on FlowVR [3], a middleware for modular data-flow based interactive applications. It provides the environment to develop, launch and execute a distributed application on a cluster. It also allows to express complex collective communications patterns and coupling policies.

Compared to an OpenGL based protocol, our approach requires modifying the applications to issue FlowVR Render primitives. However this drawback is limited as scientific visualization codes usually rely on a reduced set of graphics primitives. Porting their rendering API to FlowVR Render is usually not too difficult. To demonstrate this, we modified the rendering code of VTK [29], one of the most widely used visualization toolkit, to transparently support FlowVR Render.

Rendering on a display wall with a varying number of parallel viewers and renderer for iso-surface extraction and volume rendering was tested both with VTK+Chromium and VTK+FlowVR Render. FlowVR Render leads to a higher performance and a better network scalability as the number of viewers and/or renderers increases. The shader based protocol also enables to implement hardware-accelerated volume rendering algorithms that achieve high quality rendering [7, 13].

Organization. After presenting the related works in section 2, we detail the protocol as well as the basic filtering operations used by FlowVR Render in section 3. Communication patterns for parallel rendering will be discussed in section 4. Their usage is presented and tested using two applications in section 5. Finally results and future works are discussed in section 6.

2 Background and Related Work

In this section, we discuss rendering APIs before giving an overview of existing parallel rendering approaches. The section ends with a discussion on the benefits of using shaders for visualization algorithms.

2.1 Rendering APIs

OpenGL [30] is currently the predominant graphics rendering API and is available for most platforms and graphics cards. It is originally based on a immediate-mode model where the application sequentially specifies the objects in the scene. A large state machine is used to handle rendering parameters such as transformation matrices, lights, material properties, etc. Graphics cards and hardware platforms have evolved significantly since OpenGL introduction in 1992. To match this evolution an extension mechanism is used for hardware vendors to expose additional functionalities corresponding to new features or optimizations. Periodically, a new OpenGL version standardizing useful extensions is released by the Architecture Review Board (ARB). The latest version, OpenGL 2.0, was released in September 2004 and supports recent features such as non-power-of-two textures and high-level programmable shaders.

Pure immediate-mode rendering requires re-issuing all graphics primitives for each frame and thus can not benefit from inter-frame coherence. It introduces a high overhead on the CPU and the CPU to GPU communication bus. As the graphics cards performance improved, it became a serious performance bottleneck. To reduce this bottleneck, OpenGL has evolved to support a number of mechanisms to optimize data uploads to the graphics card, ranging from the original display-list mechanism to compiled vertex arrays and the recent vertex/pixel buffer objects extensions.

Each new OpenGL version is backward compatible with all previous versions. As a consequence many deprecated or duplicate functionalities exist such as the different ways to specify vertex data or the fixed function pipeline superseded by programmable shaders [4]. This significantly increases the complexity of OpenGL implementations. An effort to remove legacy commands and states from OpenGL's core API was originally proposed for OpenGL 2.0. This proposal was not accepted but led to a derived API for embedded systems, OpenGL ES [5].

In opposite to immediate-mode rendering, retained-mode APIs [34, 32] manage a scene description the application creates and then updates at each frame (often in the form of a hierarchical scene graph). This model is today commonly used by scene graph libraries on top of OpenGL.

2.2 Parallel Rendering

Different approaches have been proposed for cluster based parallel rendering [6]. Sort-last parallelism [18] relies on hardware or software compositors to combine pixels computed on different graphics cards [14, 31]. This approach has also been used for load balancing on display walls [28].

Sort-first approaches distribute graphics primitives to several rendering nodes. Sorting is generally based on each node's view frustum. Pixels are either directly displayed or sort-last techniques are used to recombined them. Chromium [10] proposes an OpenGL based protocol that enables sort-first parallel rendering. Legacy OpenGL applications can be executed on a display wall without recompilation. Chromium intercepts primitives issued by the application and sends them to the rendering hosts. It relies on advanced caching, compression and state tracking mechanisms to save network bandwidth. Chromium also proposes a set of OpenGL extensions for ordering several primitive streams in order to enable a parallel primitive generation, but at the price of the OpenGL compatibility.

Other approaches work at a higher level. Scene graphs rely on a partial graph duplication on rendering nodes and coherency protocols [27], to balance network traffic, duplication of data and computations. However, the user control over the parallelization scheme implemented is usually limited. Scalable scientific visualization often requires a finer control on data movements and to combine different parallelization schemes to be efficient.

2.3 Shader-Based Visualization

Procedural shaders have been extensively used in offline software-based rendering [9] to specify the visual appearance of graphics objects. Initially hardware systems only supported fixed functionalities that were programmed through a set of parameters or switches. New generations of graphics cards are now able to execute full programmable shaders [22, 24]. Recent models [12] support high-level shaders [15, 26] with method calls, loops, conditionals, and full 32-bit floating point precision for computation and textures/buffers.

Shaders provide additional flexibility and precision allowing graphics cards to execute algorithms that only the CPU could execute before. Such algorithms include high-quality lighting models (per-pixel evaluation, shadows), tone shading [8], or volume rendering [25]. For instance, obtaining colors from raw data by applying a transfer function can now be done within a shader.

2.4 Summary

Using recent OpenGL buffer objects and hardware shaders, applications can efficiently take advantage of advanced graphic cards features. However, the OpenGL state machine and the numerous duplicated functionalities that OpenGL still supports significantly increase the complexity and hinder the effectiveness of any implementation, in particular parallel ones.

Using shaders, parts of the data processing visualization pipeline is now executed by the graphics card, distributing the load between the CPU and the GPU. This changes the level of information sent to the graphics cards, which can receive general datasets instead of only final colors. Tuning the parameters of the processing implemented by shaders only requires updating a few values instead of re-uploading the complete dataset. In a distributed context where graphics primitives are issued on one machine and rendering is performed on a distant one this can drastically change the performance of the pipeline.

We propose to develop a pure shader based sort-first protocol dedicated to high performance parallel rendering.

3 FlowVR Render Model

In this section we present the FlowVR Render framework and its different components. A viewer program describes *primitives* sent to a renderer program using a specific *protocol*. Complex parallel rendering architectures can be built combining various viewers and renderers mapped on different nodes of a cluster. *Filters* are used to implement advanced routing functions on primitives.

3.1 Graphics Primitives

A scene is described as a set of independent *primitives*. Each primitive contains the description of a batch of polygons and their rendering parameters (see figure 1). To reduce the number of these parameters, as well as to take advantage of the advanced features of recent graphics cards, we use *shaders* to define each primitive's appearance.

Large resources such as textures or vertex attributes are often used by several primitives in the scene. To avoid duplicating these resources in each primitive, we define *resources*. A resource can encapsulate a shader, texture, vertex buffer or index buffer. It is identified by an unique *ID*. Each primitive refers to the IDs of all the resources it uses. Notice that it introduces a one-way dependency between primitives and resources, but not between primitives.

The framework must ensure that IDs are globally unique even in a distributed context. Possible methods include using a specific host, which can introduce a bottleneck for large applications, or using local information unique to each host. In our implementation we use 64-bits IDs by appending an atomic counter local to each host with the IP address of the host.

A *name* can be specified for each primitive for identification in error messages, visual selection feedback, or filters based on name patterns.

Each primitive also stores its *bounding box*. This information is required to optimize communications using frustum culling. It is useful to specify this information in the primitive as it is costly to recover from the other parameters (especially when using shaders that can change vertex positions).

Some rendering algorithms require primitives to be processed in a given order. FlowVR Render provides a mechanism to enforce this ordering by associating each primitive with an *order* value. This number defines a partial ordering between primitives. A primitive with a lower order value must be rendered before a primitive with a higher value. Primitives with the same value can be rendered in any order. Different values will mainly be used when rendering order can affect the final image like for transparent primitives

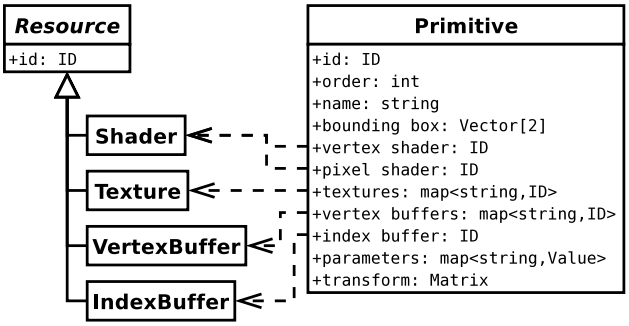


Figure 1: Simplified UML schema of a primitive.

or user-interface overlays. For a given order value, renderers can re-order primitives to improve performance. For instance, primitives can be sorted front-to-back to take advantage of fast Z-culling, primitives with similar parameters such as textures and shaders can be gathered to reduce state switching overheads. This approach enables to easily implement performance optimization compared to the strict ordering defined by an immediate-mode command stream. This strict ordering can however still be achieved by assigning a different order value to each primitive.

Global parameters such as camera position, z clipping distances, or background colors are directly set at the renderer statically or upon reception of user's interaction events. A viewer can also override these parameters using a primitive with the predefined ID *ID_CAMERA*.

3.2 Communication Protocol

In graphics scenes some data are static, i.e. they do not change between frames, while others are dynamic. One important optimization is to describe static information only once, and then transmit changes at each frame. To achieve this goal, renderers maintain the current description of the scene, and viewers describe the changes to this description. Each change is encapsulated in a *chunk*. At each frame, a viewer sends one *message* containing a list of chunks. A renderer waits to receive one message. A chunk can correspond to a:

- Creation of a new resource ;
- Destruction of a resource ;
- Update of a resource ;
- Creation of a new primitive ;
- Destruction of a primitive ;
- Modification of a primitive's parameter.

This protocol is purely one-way, from viewers to renderers. In particular, IDs are generated by the viewers and not the renderers. This property is useful when the viewers and renderers are not tightly synchronized (for example storing messages on disk and replaying them later).

3.3 Filters

Parallel rendering schemes require filtering message streams to distribute data between several viewers and/or renderers. For that purpose we introduce *filters* that implement the processings necessary for advanced routing networks. We define in the following some

common filters. Complex assemblies of viewers, filters and renderers are presented in section 4.

To support scene descriptions distributed on multiple viewer modules, it is necessary to be able to merge several messages together. In our model all primitives are independent and use globally unique IDs. As a consequence this operation consists in a simple gather, appending messages from all streams together.

For sort-first rendering with a single viewer, a simple broadcast can be used to send messages to all renderers. For highly dynamic scenes, messages can be filtered by removing (*cull*) all changes not affecting the local view frustum of each renderer. The bounding box information is readily available to test for intersection with the frustum, enabling to simply discard hidden primitives, without any effects on other primitives as they are independent. The only difficulty concerns resources (textures, vertex data), as they can be shared by several primitives. A simple algorithm consists in sending all resources referenced by each visible primitive, provided it wasn't already sent. For very large textures or meshes, a more complex implementation may only send visible parts of each resource, but recovering this information is costly.

Thanks to the one-way and retained-mode nature of our model, another operation that is very efficient and easy to implement is to allow for different frequencies between viewers and/or renderers. In particular, this enables multiple asynchronous visualization of the scene (two distant displays in a collaborative application, or a low-performance console view not affecting the performance of the main display for instance). Another more advanced application consists for each viewers in having a specific update frequency (low-frequency for large objects or remote viewers, high-frequency for local viewers or interactive updates such as camera manipulation). Filters implementing this strategy simply require either inserting empty messages or appending several messages together.

Other operations can be implemented depending on the applications. It is for instance possible to design filters altering the appearance of the scene for stylized rendering [16, 17], splitting the objects for an exploded view [21], or writing the scene's 3D description in a file [17]. Compared to the traditional approach of modifying OpenGL commands, implementing these operations using our framework is easier as higher-level information is available. Also only a handful set of chunk types are used, compared to the hundreds of different OpenGL commands.

4 System Architecture

Data-flow based architectures, where several data streams are processed through a network of filters, have been successfully applied both in visualization and distributed rendering applications [1, 2, 10]. It can be applied at different stages in the application, from inputs events to final pixels composing. This section presents the design of the data-flow network used to transmit graphics primitives.

4.1 General Design

FlowVR Render architecture follows a data-flow graph. The source nodes of the graph are viewers that produce scene description messages as described in section 3.2. These messages travel through a network of filters that are responsible for implementing the necessary operations as described in section 3.3. Once data have been redistributed and processed by this network, it is used by the destination nodes, the renderers, to render the scene.

We consider modular visualization applications, where objects in the same scene are handled by completely different programs. As FlowVR Render primitves are a-priori independent, merging several data streams has a small overhead. This architecture provides two main advantages. First it favors code reuse, allowing to choose the right visualization toolkit or library for each task instead of reimplementing it in a single environment, and second it permits to choose a different distribution strategy for each object, allowing a fine performance tuning.

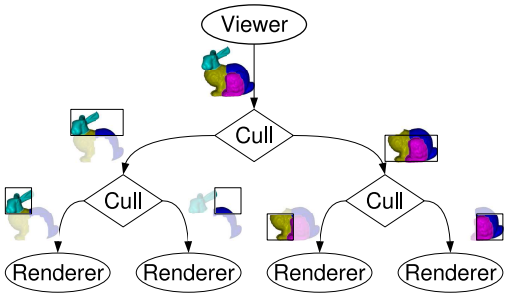


Figure 2: Sort-first distributed rendering. For small or static objects a broadcast tree can be used instead of frustum culling filters.

4.2 Distribution Strategies

Distributed rendering is a typical problem where no generic solution exists. Several factors (size and time dependency of datasets, computational power, network speed, number of pixels) are involved in determining the best distribution scheme between sort-first [18], sort-last, hybrid, or input-event based distribution. Testing different strategies should thus be as simple as possible. Moreover, as the same program can be used in situations requiring different distirbution strategies, changing the distribution scheme should require minimal modifications to the program. Data-flow based architectures propose an elegant solution to this issue as changing of strategy often consists in simply changing the position of each element, eventually adding some filters. The modular design of applications in FlowVR Render lets multiple schemes be simultaneously executed for different parts of the scene. The rest of this section will concentrate on the basic patterns implementing each distribution scheme. Using several schemes in a single application is then possible by combining these patterns together.

In a multiple display system (cave or display wall for example), if one viewer is connected to several renderers through frustum-culling filters as shown in figure 2 we have a sort-first distribution model. Notice that for better scalability we can use a two-level culling filter tree: first splitting horizontally then vertically. Depending on the size and dynamicity of the primitives, the cull filter can either simply cull on an primitive level or split each triangle individually. The first option is often preferred as it is simpler to implement, especially when the frustum is not static, and the spared network and GPU bandwidth might not counter-balance the additional CPU cost of a more precise culling method. To avoid any network communication of the graphics primitive, the viewer can also be replicated locally on each rendering node (figure 3).

When using multiple viewers, each one describing a part of the scene, merge filters are responsible for combining their messages together (figure 4). This scheme is useful for parallel data extraction applications, which we will show in section 5.2. It is also necessary for recombining parts of an application that uses different distribution schemes. In this case, the different streams are merged together before being forwarded to each renderer.

We now consider remote rendering systems. The renderer can be placed at the same location as the viewer, pixels being communicated to the remote site (figure 5(a)). It can alternatively be moved to the display location. In this case, the graphics primitive descriptions are transmitted instead (figure 5(b)). A more interesting case is when both are combined (figure 5(c)). In a scene composed of

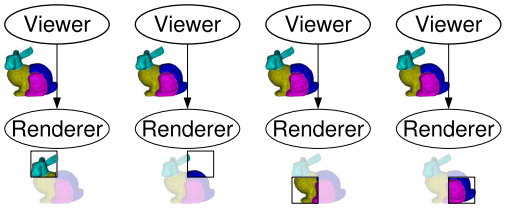


Figure 3: Replication: Viewer application is replicated on each rendering node. If the application is non-determinist then communications will be necessary to keep copies synchronized.

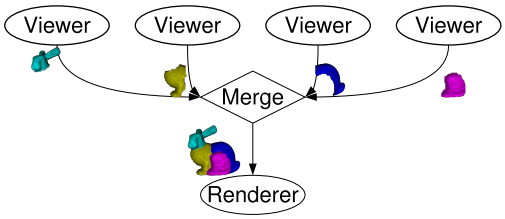


Figure 4: Parallel scene description: To distribute the load in viewers, several viewers can describe parts of the scene which are then merged together.

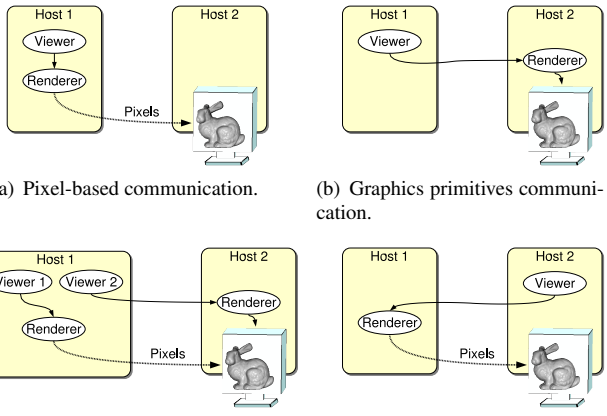
streamlines and volumetric rendering for instance, the streamlines could be transmitted over the network and rendered with a high resolution, while the volume image will be rendered locally and transmitted at a lower resolution. Another combination happens when the viewer and renderer are distant but the final display is on the viewer side (figure 5(d)). This scheme corresponds to the *render server* system, where rendering is provided as a distant shared resource. By using existing image composing tools [14, 20] we can use several servers in parallel for increased performance.

4.3 Interactions and Scene Management

Having a distributed scene description controlled by unrelated programs can lead to difficulties in obtaining a coherent manipulation interface. In monolithic applications, this is often done through the scene graph structure, where the user can change parameters of the nodes (visibility, wireframe, ...) and control transformation nodes with special widgets to move/rotate/scale objects. To implement this within our framework we face two issues: as the communication protocol is strictly one-way, how can viewers get the user's actions; and as the scene description model forbids dependencies between primitives, how can we specify that a group of primitives should be moved through a common transform node.

User inputs depend on the interaction devices (mouse, keyboard, VR tracking devices, ...). Reading inputs data is done by the renderers for mouse/keyboard inputs, or by special tools for other devices. As the data is small, broadcasting the inputs over the network is simple [33, 23]. However interactions are often expressed in terms of objects in the scene (i.e. a user selected object with a specific ID). As we already have a globally unique ID associated with each primitive, renderers can output events containing these IDs. Viewers can then use this information to manage interactions.

As described at the beginning of this section, basic interactions (moving objects, changing rendering attributes, ...) are often not handled by the objects themselves but by other components in the scene (widgets and transform nodes). The same functionality can be achieved by adding custom filters in the data-flow network to transparently change the affected parameters. If required, additional *widgets* viewer programs can be introduced, adding objects



(a) Pixel-based communication.

(b) Graphics primitives communication.

(c) Different communication levels for specific viewers.

(d) Render server scheme: send graphics primitives, get rendered pixel back.

Figure 5: Remote rendering.

in the scene and retrieving interaction events on these objects, eventually sending resulting movements to the transform filters. This architecture can be seen as the data-flow based mapping of the equivalent scene-graph hierarchical structure. Thus while the scene description model does not allow for interdependencies between primitives, we can implement the equivalent functionalities by adding filters in the network.

4.4 Implementation

FlowVR Render architecture can be implemented using many different communication API, such as TCP connections or CORBA objects. We chose to use FlowVR [3] as it provides a clean abstraction to design interactive data-flow applications. At the lowest level it transparently uses either TCP connections for network communications or shared-memory for local communications. It also provides tools to launch and control distributed applications, as well as generating large filters networks using scripts.

Current implementation of FlowVR Render can be downloaded from `http://flowvr.sf.net/render/`.

5 Applications

In this section we present experimental results using FlowVR Render, VTK (version 4.2) and Chromium (version 1.8). VTK and Chromium have been chosen because they are probably the most used and advanced tools publicly available today in their category. We used them without in-depth code tunings and optimizations. A version of VTK has been tuned for Chromium [19], but it is not used here as it is not publicly available.

While we present performance comparisons between Chromium and FlowVR Render, the reader should keep in mind that their conditions of use are different. Chromium supports all OpenGL applications without modification, while FlowVR Render proposes a new shader based rendering framework that requires application to be adapted.

Tests were performed on the GrImage[1] platform composed of a cluster of 16 Dual-Opteron 2.0 GHz having 2 GB of memory each. Cluster nodes are interconnected by a gigabit Ethernet network. Each node uses an NVIDIA Geforce 6800 Ultra graphics cards to drive a 4×4 display-wall with a total resolution of 4096×3072.

[1] `http://www.inrialpes.fr/grimage/`

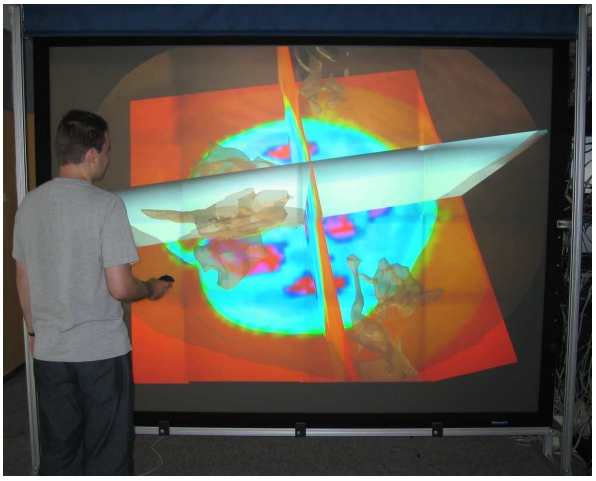


Figure 6: An example VTK application rendering on the display wall with FlowVR Render.

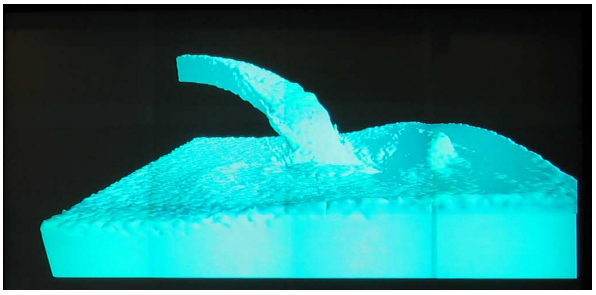
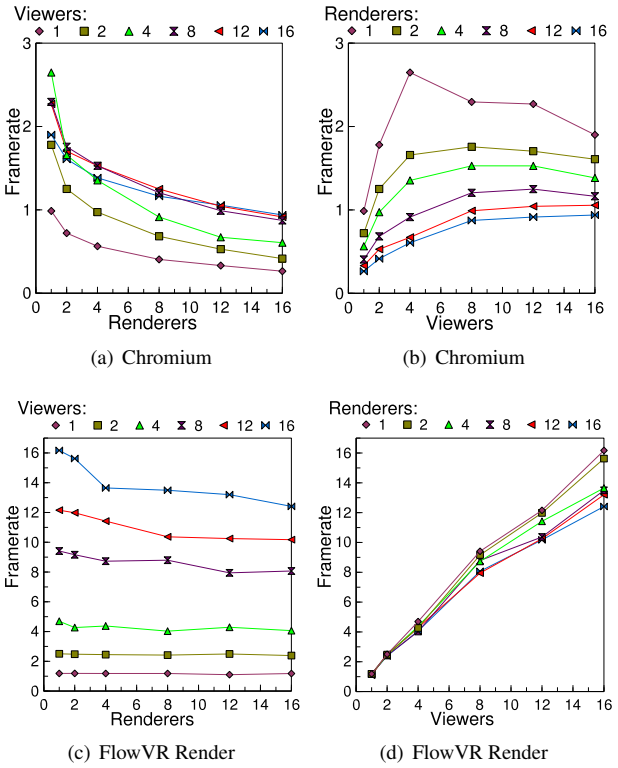


Figure 7: Iso-surface extracted from a frame of a time-varying fluid simulation dataset.



(a) Chromium (b) Chromium

(c) FlowVR Render (d) FlowVR Render

Figure 8: Parallel iso-surface extraction with sort-first rendering, using Chromium (a)-(b) or FlowVR Render (c)-(d). Scalability regarding the number of renderers is presented on the left, while scalability regarding the number of viewers is shown on the right.

5.1 VTK and FlowVR Render Integration

Several powerful open-source visualization toolkits are available such as OpenDX [1] or VTK [29]. We developed a library that transparently replaces VTK rendering classes to use FlowVR Render instead of OpenGL. VTK uses a small set of rendering classes to draw images, meshes (with a list of points, lines and triangles) or volumes. The original rendering classes use OpenGL immediate mode commands or display lists. These are complex classes to handle all possible combinations of vertex data as well as legacy OpenGL drivers. The new FlowVR Render classes are simpler as they often only consist in encapsulating the raw data into FlowVR Render resources and selecting the right shader. Unmodified VTK applications can then be rendered on a display-wall as shown in figure 6. Developing this library only took a couple of weeks for one programmer, most of which was spent learning VTK.

5.2 Parallel Iso-surface Extraction

To compare the performance of our framework with Chromium we implemented a parallel iso-surface extraction application. We used a 3D fluid simulation dataset of $132 \times 132 \times 66$ cells for 900 timesteps (one timestep is shown in figure 7). The application interactively extracts and displays an iso-surface (containing approximately 100000 triangles) for each timestep. The iso-surface extraction is parallelized by splitting the dataset into blocks, each one assigned to a different viewer.

Figure 8 presents Chromium and FlowVR Render performance results depending on the number of renderers as well as the number of iso-surface extraction viewers. FlowVR Render outperforms Chromium and shows a better scalability, both while increasing the number of renderers and the number of viewers. FlowVR Render achieves 12 frames per second with 16 data viewers and 16 renderers to display the result on the 4×4 display-wall. Chromium performance is probably affected by the high overhead related to culling and stream merging operations.

Using pixel shaders can also greatly improve visual quality. For this application, a classic OpenGL-based lighting is evaluated per vertex, while it is computed per pixel with shaders. The resulting surface appears smoother.

5.3 Volume Rendering

The second test pushes further the use of shaders to highlight the benefits of our approach. For that purpose we focus on sort-first parallel volume rendering. Notice that usually sort-last approaches have proved more efficient than sort-first approaches [35]. We show that using hardware shaders can significantly improve performance of sort-first algorithms based on the following points:

- Due to the massively parallel nature of todays GPUs, pixel shaders have access to more important resources, both in terms of memory bandwidth and computing power [12].
- Shaders are able to apply transfer functions to raw volumetric data to obtain the final color and opacity. This allows to send the raw data once, and then only update the transfer function when it is modified by the user. For time-varying datasets

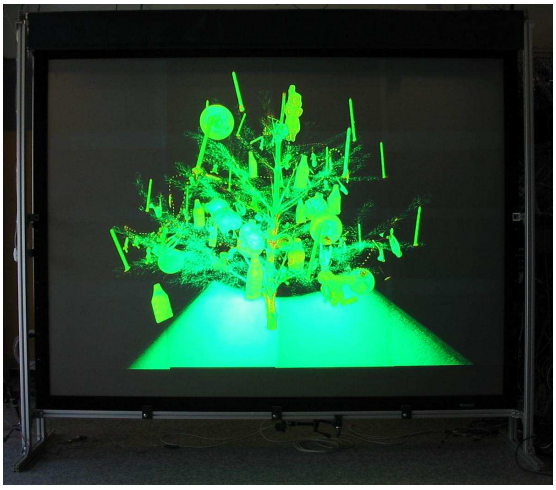


Figure 10: Volume rendering on the display wall.

this is also interesting as the raw data can be 4 times smaller than final colors and opacity data (1 value per voxel versus 4 values).

- Using pre-integrated transfer functions [7] and adaptive sampling steps [25], a shader can create a very high quality image while using fewer steps through the volume, allowing to use larger datasets.

By default VTK uses a slice-based 2D texturing approach for hardware volume rendering. We implemented a new VTK volume rendering class using shaders. A pixel shader is used to cast a ray through the volume and accumulate color and opacity using either blending, additive, or maximum value composing. As this composing is done in temporary registers instead of the frame buffer, it stays in full 32-bit precision and it saves the bandwidth required to write and read-back framebuffer values in traditional multi-pass approaches.

To implement the pre-integrated transfer function, we use a 2D texture that, given the previous and current density value, stores the color and opacity produced by integrating all intermediate densities in the transfer function. This greatly improves visual quality for high frequency transfer functions and allows for much larger sampling steps for smoothly varying datasets.

As this application is only limited by the fill-rate of the graphics cards, we used a simple broadcast distribution scheme where everything is sent to all renderers.

Renderings obtained using the VTK original slice-based 2D texturing and FlowVR Render-based raycasting shader with and without preintegration are shown in figure 9. The data set is a $512 \times 512 \times 512$ Christmas tree [11]. Performance results are presented in table 1. As a comparison, VTK 2D texturing implementation achieved 0.18 frames per second on one display. This is mostly due to the fact that without shaders full-color textures must be used instead of the raw grayscale texture. In our case this means that VTK had to reupload the data at each frame as it does not fit inside the graphics card.

Rendering on the display wall instead of only one display does not introduce significant overhead as the transfered data is small for most frames (camera position and transfer function). We even obtain better performance on the display-wall as coherency between neighbor pixels is higher. It leads to a more efficient texture caching inside the graphics cards.

Notice that a higher framerate may be obtained during interactions (when the camera is moving for example), by decreasing the rendering resolution in addition to the sampling resolution. It permits to obtain fluid movements while keeping a reasonable quality.

6 Conclusion

In this paper, we presented a novel parallel and remote rendering framework using a scene abstraction based on batches of polygons and shaders. This framework proved to be efficient and scalable while using simple enough concepts to be easily extensible. Although not directly compatible with existing applications in opposite to Chromium, the porting effort should usually be limited. In the case of visualization applications using a common toolkit this effort only has to be made once, with the additional benefit of providing access to advanced features using shaders.

The iso-surface extraction test application showed that the FlowVR Render approach outperforms Chromium regarding performance and scalability. The volume rendering application showed that using shaders and a communication protocol based on incremental changes significantly reduces the amount of data communicated over the network. Using raw data sets instead of final colors and geometry also reduces the memory requirements on the graphics card, allowing larger datasets on each node. In particular, we achieved an interactive rendering of a $512 \times 512 \times 512$ dataset on a 4096×3072 display wall.

Notice that the experiments presented focused on direct rendering on a display wall. FlowVR Render can also be used for remote rendering in conjunction with sort-last algorithms.

Future works will address the design of a more complete toolkit to manage user interactions. We will also extend FlowVR Render to support a dynamic level-of-detail. Viewers will send several versions of primitives (either reducing the number of vertices or the shaders quality), and renderers will adapt the rendering resolution and quality of the objects depending on the desired performance.

References

[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] 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.

[3] 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 *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 497–505, Pisa, Italia, August 2004.

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

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

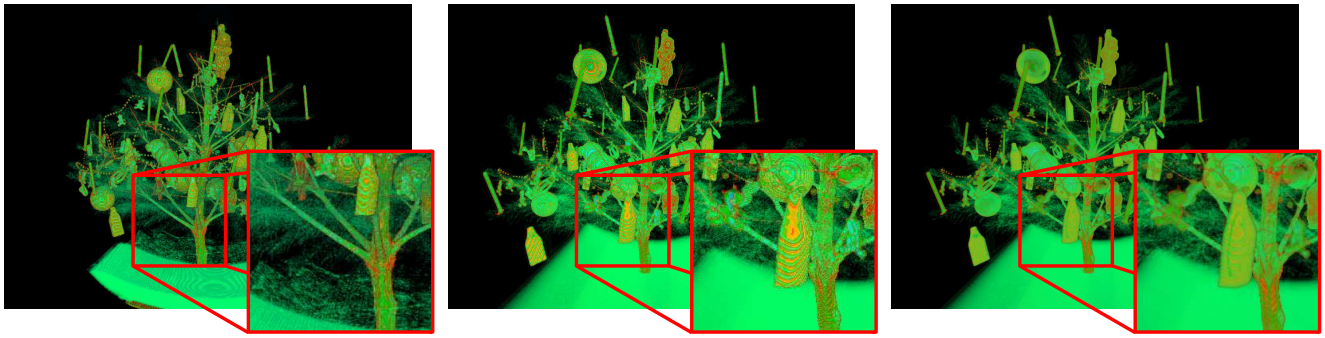
[6] Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, and K. Li. Software Environments for Cluster-based Display Systems. `http://www.cs.princeton.edu/omnimedia/papers.html`, 2001.

[7] 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.

[8] 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.

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

[10] 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.



(a) VTK 2D texture slices (b) Raycasting shader (c) Raycasting shader with preintegration

Figure 9: Christmas tree dataset with different rendering methods (512 sampling steps per pixel).

Method	Sampling steps	Framerate on 1 display Resolution 1024×768	4×4 display-wall 4096×3072	4×4 display-wall 2048×1536	4×4 display-wall 1024×768
Raycast Shader	512	1.16	2.25	5.40	8.21
Pre-Integrated Raycast Shader	512	1.10	2.04	4.97	7.70
Pre-Integrated Raycast Shader	200	2.79	5.14	12.44	19.11

Table 1: Volume rendering performances with a $512 \times 512 \times 512$ dataset.

[11] 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, 2002.

[12] 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.

[13] E. B. Lum, B. Wilson, and K.-L. Ma. Vissym 2004, symposium on visualization. Eurographics Association, May 2004.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] K. Moreland and D. Thompson. From cluster to wall with vtk. In *Proceedings of IEEE 2003 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 25–31, October 2003.

[20] 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.

[21] 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.

[22] 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.

[23] E. Olson. Cluster Juggler–PC cluster virtual reality. Master's thesis, Iowa State University, 2002.

[24] 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, 2001.

[25] 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.

[26] R. J. Rost. *OpenGL® Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[27] M. Roth, G. Voss, and D. Reiners. Multi-threading and clustering for scene graph systems. *Computers & Graphics*, 28(1):63–66, 2004.

[28] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000.

[29] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition*. Kitware, Inc., 2003.

[30] M. Segal and K. Akeley. *The OpenGL™ Graphics System: A Specification, Version 1.0*. Silicon Graphics, 1992.

[31] 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.

[32] P. S. Strauss. Iris inventor, a 3d graphics toolkit. In *OOPSLA '93: Proceedings of the conference on object-oriented programming systems, languages, and applications*, pages 192–200. ACM Press, 1993.

[33] R. M. Taylor 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*, 2001.

[34] A. van Dam. PHIGS+ functional description revision. *SIGGRAPH Computer Graphics*, 22(3):125–220, 1988.

[35] 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 Symposium*, pages 23–30, Grenoble, France, June 2004.

A.4 FlowVR Europar 2004 Article

FlowVR: a Middleware for Large Scale Virtual Reality Applications

Jérémie Allard[1], Valérie Gouranton[2], Loïck Lecointre[1], Sébastien Limet[2], Emmanuel Melin[2], Bruno Raffin[1], and Sophie Robert[2]

[1] Laboratoire ID, CNRS/INPG/INRIA/UJF, Montbonnot, France
[2] LIFO, Université d'Orléans/CNRS, Orléans, France

Abstract. This paper introduces FlowVR, a middleware dedicated to virtual reality applications distributed on clusters or grid environments. FlowVR supports coupling of heterogeneous parallel codes and is component oriented to favor code reuse. While classical communication paradigms focus on either a synchronous approach (FIFO channels) or an asynchronous one (sampling), FlowVR enables a large range of intermediate policies to better balance the application performance between levels of details, latencies and refresh rates.

1 Introduction

Classically, a virtual reality (VR) application features a complex simulation using input and output devices to provide users with a sense of immersion in a synthetic world [7]. Most of today's VR applications only run on machines with a reduced number of processors, like visualization clusters or SGI Onyx. They do not take advantage of the computing power offered by large clusters and grid environments. One main limitation is the difficulty to assemble and distribute the different (potentially parallel) components and to maintain the overall application *coherent* while guaranteeing a good quality interaction with *low latency* and *high refresh rates*. We define the *coherency* as the fact that the information provided to the user senses at a given moment are related to the same simulated time.

To improve latency and refresh rates, VR applications can take advantage of a data exchange model based on sampling. The producer updates data in a shared buffer asynchronously read by the consumer. Some updates may be lost if the consumer is slower than the producer. While asynchronism leads to a performance improvement, the application coherency cannot be maintained. Depending on the context this may be acceptable. It is for example used when coupling haptic and visualization systems that run at very different frequencies (about 1000 Hz and 60 Hz respectively). Distributed virtual environments [9, 11] or VR middlewares like OpenMask [2] use such an approach, but parallel code coupling becomes difficult in this context as no coherency control is offered. The other approach classically used for parallel programming, parallel code coupling [8, 10], or distributed visualization environments [3–5], relies on a classical

FIFO synchronization semantics. It ensures proper application coherency, but it is difficult to efficiently implement a sampling approach.

In this paper, we propose a programming model that eases the implementation of a large range of synchronization policies, from FIFO to sampling. We present FlowVR [1], a middleware dedicated to VR and supporting coupling of heterogeneous parallel codes to build large scale applications. FlowVR reuses and extends the data flow paradigm commonly used for scientific visualization environments [3, 4]. A VR application is seen as a set of possibly distributed modules exchanging data. Each module endlessly iterates, consuming and producing data. From the FlowVR point of view, modules are not aware of the existence of other modules, the FlowVR engine taking care of moving data between producers and consumers. This leads to a simple application programming interface (API) that eases turning an existing code into a FlowVR module (or several modules in case of a parallel code). For data exchange between modules, FlowVR defines an abstract network featuring from simple routing operations to complex message handling operations. Each message is associated with a *list of stamps*, a lightweight data used to route or filter messages. This list can also be routed separately from its message to special network nodes in charge of synchronization policies. Besides predefined FlowVR stamps, others, like a time or a 3D bounding box for instance, may be added to extend the network routing, filtering or synchronization abilities. The FlowVR network enables to build complex collective communications, a desirable feature for efficient parallel code coupling. It is also possible to go beyond the classical synchronization barrier, designing synchronizations waiting for the resolution of complex constraints based on stamps (a data semantically richer than a signal). Different FlowVR networks can be designed without modification of the module codes.

2 The FlowVR Application Model

In this section we introduce the FlowVR application model.

2.1 Running Example

All along this paper, we use a simple yet important example, an interactive VR application where the user can perturbate a fluid flow simulation with its hand. We distinguish three parts :

- A tracker that gives the user's hand position.
- A physical fluid simulation parallelized with MPI. The simulation is based on a 2D grid split in blocks amongst the different MPI processes. MPI communications take place at each iteration to exchange the values of the grid borders between neighbors. Each process should also receive the hand position, which acts as an obstacle for the fluid flow.
- A multi-projector visualization environment. Each projector, driven by its own PC, displays a tile of the entire scene. The distribution paradigm adopted

```
initialization
while not stop
    wait()
    get(position)
    computations
    put(grid)
```

Fig. 1. The algorithm of the simulation module.

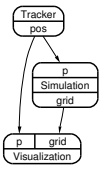


Fig. 2. Interactive fluid simulation with 3 modules.

is simple: all PCs run a copy of the visualization application, each one expecting the coordinates of the hand position and a density grid at each iteration. To ensure a strong coherency of the displayed images, these copies must receive the same input data at each iteration. Next, each copy computes its tile of the global image based on its own viewing frustum (viewing angle). All PCs must then display the new image synchronously, either using a hardware swaplock or a software barrier.

These codes can run independently at very different frequencies. The tracker is certainly the fastest one and the fluid simulation the slowest one. A sampling-based data exchange model will let the codes run independently at their highest frequency, but it may lead to incoherences. For instance, in a given image, the displayed hand position may not correspond to the one used to compute the displayed simulation state. On the opposite, a FIFO communication model will ensure the overall application coherency, but at the price of a lower performance. All codes will run at the same frequency, synchronized on the slowest one. The tracker will produce a new data as soon as room is available in the output channel buffer. The latency will increase by the time such data stay unused in this buffer, the time required by the fluid simulation to consume all data previously stored in this buffer. FlowVR has been designed to let the user specify these different policies and other *intermediate* solutions, without requiring any modification of the codes.

2.2 Modules

We first introduce the API used to program FlowVR *modules*. This API is kept as simple as possible to limit the effort required to convert an existing code into a FlowVR module. For that purpose we explicitly took advantage of the interactive nature of VR applications. A FlowVR module is a computation loop periodically reading input data and producing new results. To improve code reuse, a module cannot directly address another module. This way there is no explicit dependency between modules. Their only knowledge of the FlowVR environment is a list of input and output ports. The module API is based on three main methods:

– The *wait* defines the transition to a new iteration. It is a blocking call that ensures each connected input port holds a new message. Input ports not

connected to any other port will never receive any message. They are *deactivated*.
- The *get* function enables a module to retrieve the message available on a port.
- The *put* function enables a module to write a message on an output port. Only one new message can be written per port and iteration. Each output message is automatically stamped by FlowVR with the current iteration number.

In our example, we would define:

- One module for the tracker with one output port (a position data).
- Each MPI process of the fluid simulation will define a module with one input (a position data) and one output (its block of the fluid density grid) (Fig. 1). To be able to distinguish the different blocks, each process stamps its output messages with the coordinates of its block.
- One module for each visualization process, with two input ports each, one to retrieve the tracker position and the other one to retrieve the whole density grid.

Each module has two additional predefined ports. The *input activation port* is used to lock the module to an external event (fixed frequency trigger for the tracker for instance). The *output activation port* is used to signal other components that the module has started a new iteration (see section 2.5).

2.3 Connections

Once modules are defined, they are assembled connecting their input and output ports. The simplest primitive used to build a FlowVR network is a *connection*. A connection is a typed FIFO channel with one source and one destination. Messages in a connection are numbered. Each message is stamped with this number and the source id.

Let us consider our example. We can build a simple first application with one tracker module, one fluid simulation module and one visualization module (Fig. 2). We add one connection from the tracker to the visualization, another one from the tracker to the simulation and a last one from the simulation to the visualization. This simple application implements a classical communication scheme using FIFO channels. The FIFO connections ensure a strong coherency. At each iteration the visualization module will always retrieve a tracker position and a density grid corresponding to the same simulated time. Therefore the resulting application will be synchronized on the slowest module, presumably the fluid simulation. If the tracker module is faster than the simulation module, there will be a significant lag between between user interactions and their effects on the virtual world. Also notice that adding the connections does not require to modify the code of the modules.

However, having only point to point FIFO connections, it is difficult to loosen the synchronizations imposed by the FIFO model or to express collective communications.

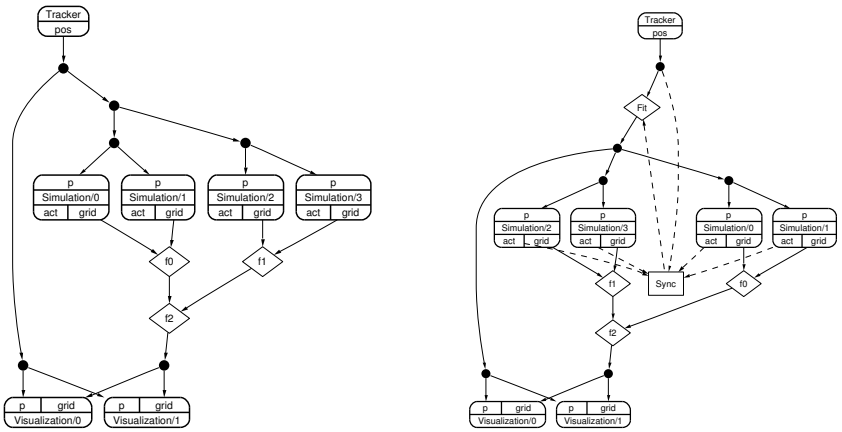


Fig. 3. (a) Fluid simulation with a FIFO network. Modules are represented as round-shaped squares, routing nodes as circles and filters as diamonds. (b) Fluid simulation with a coherent sampling network using one synchronizer (a square). Dashed lines correspond to connections carrying only stamps. The *act* port corresponds to the output activation port.

2.4 Filters

To extend the capabilities of the FlowVR network we introduce a new component, called *filter*.

A filter has typed input and output ports and can perform complex operations on messages. Filters have all the freedom to discard, combine or even generate messages. They are not restricted to receive only one message per port and per iteration like modules. They have free access to incoming buffers. Filters usually handle messages based on the associated lists of stamps. For instance, a filter can discard all incoming messages, which 3D bounding box falls outside of a given volume. Amongst filters, we distinguish the *routing nodes* as the filters that only forward all incoming messages on one or several outputs.

Let extend our example by now using four modules for the simulation and two modules for the visualization (Fig. 3(a)). The tracker messages must be broadcasted to these modules. For that purpose we introduce in our network several routing nodes. To broadcast the data to modules we choose to implement a binary-tree broadcast. The data exchange between simulation and visualization is more complex as we have to ensure that all visualization modules receive the whole density grid while each simulation module sends only one fourth of it. For that purpose we use a filter that combines two blocks of density grids into a larger one. This example implements a network with non trivial collective communications. A strong coherency is still ensured as the filter we use here does not suppress or generate new data (FIFO network).

2.5 Synchronizers

We distinguish a special class of filters, called *synchronizers*, used to implement the resolution of non local constraints. A synchronizer works on stamps. Therefore all incoming and outgoing connections only carry message stamps. Generally a synchronizer activity is triggered by incoming stamps on some selected ports. As synchronizers do not receive the data part of the messages, their output ports are generally connected to filters. These filters typically have 2 input ports, one receiving full messages (the data and its list of stamps) from a module or a filter, and the other one receiving only stamps from a synchronizer. The filter processes the incoming full messages according to incoming stamps. For instance, such a filter can forward to its output only the full messages corresponding to the incoming stamps, discarding the other messages.

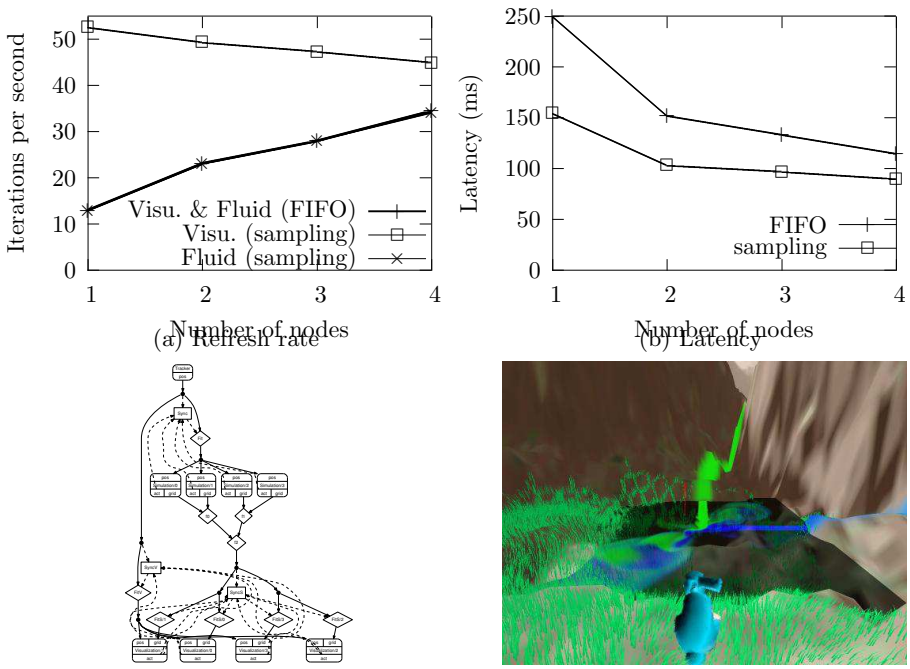
Classical synchronization schemes can often be expressed in term of signal handling. In this case the synchronizer only uses its inputs as signals. A sampling scheme is implemented by selecting the last received message each time an activation signal is received from the destination module (request for another input message). But synchronizers can implement more complex algorithms by taking advantage of the semantically rich information hold by stamps. For example, in VR environments some coherency constraints can be expressed in term of spatial relationships. A strong coherency is required for objects close to the user, while background or unseen parts of the scene require much less attention. A stamp holding a bounding box information can be used to implement such a coherency policy.

In our example, because the simulation will probably be slower than the tracker, we introduce a synchronizer to keep pace with the tracker (Fig. 3(b)). This synchronizer takes as input the stamps from the position messages, and the stamps from the activation output ports of the fluid modules. When all fluid modules request a new data, the synchronizer selects the newest stamp available and sends it to the filter `Fit`. This filter only forwards on its output port the messages having the stamps selected by the synchronizer. A strong coherency is ensured as the visualization and simulation modules receive the same position messages. Similar ideas could be applied to implement a coherent sampling scheme to enable the visualization to run asynchronously from the simulation. Once again, building this network did not require any modification of the module codes.

3 Runtime Engine

FlowVR is open source and currently ported on Linux for IA32, IA64 and Opteron.

The FlowVR runtime engine relies on daemons, one per participating node. Daemons are in charge of FlowVR networks. They act as brokers and relay messages between modules. Filters, including synchronizers, are implemented as dynamically loaded classes (*plugins*) within the daemon. Communications local to a node use a shared memory area. Care is taken to avoid unnecessary



(a) Refresh rate (b) Latency

(c) Coherent sampling network (d) Screenshot of the visualization

Fig. 4. Experimental results with a coherent sampling network and a FIFO network.

data copies and memory allocations by exchanging pointers and reusing allocated buffers. The current implementation of inter-node communications relies on TCP. Networks of heterogeneous nodes are easily exploited, as connections are dynamically created and each daemon can be launched independently. Several applications can safely run concurrently using the same daemons.

Each FlowVR application is managed by one special module called a *controller*, automatically loaded at starting time. The controller first starts the application's modules using their own launching command, `ssh` or `mpirun` for instance. Once the modules launched, they register themselves to their local daemon that sends an acknowledgment to the controller. Then, the controller sends to each daemon the list of plugins to load to implement the FlowVR network.

FlowVR integrates tools to generate the module launching commands and the list of plugins to load. It uses as input an XML description of the syntax of the launching command associated with each module code, as well as an XML description of the FlowVR network with an explicit placement of all components on target nodes. Ongoing work focuses on developing automatic and semi-automatic FlowVR network generation tools.

3.1 Experimental Results

We implemented the running example porting an existing fluid simulation code. The fluid simulation is parallelized with MPI, while the multi-projector visualization is handled by Net Juggler (also based on MPI) [6]. From the FlowVR point of view, each MPI fluid process and each Net Juggler process is seen as a module. Note that all fluid modules (respectively visualization modules) are synchronized through MPI communication calls FlowVR is not aware of. All results presented here run a fluid simulation based on a 2D 512×512 grid. The visualization modules integrate the fluid into a rich virtual environment (See Fig. 4(d)).

Two versions of the network were tested, a FIFO network (similar to Fig. 3(a)), and a coherent sampling network enabling the tracker, the fluid simulation and the visualization to run asynchronously. It extends the network presented in Fig. 3(b) by adding an extra synchronizer between the tracker and the visualization, and another one between the simulation and the visualization (Fig. 4(c)). Tests were performed on a PC cluster with dual Xeon PCs (2.66 GHz) connected through a Gigabit Ethernet network. Each machine was equipped with a GeForce FX 5600 graphics card.

The number of visualization and fluid modules vary from 1 to 4. Each module runs on its own PC. For instance when 8 nodes are used, 4 of them execute a fluid module, while each of the 4 other PCs run a visualization module. Each of these 4 PCs drives a video projector to display the result of its visualization module (1/4 of the global image).

We measured the refresh rate, i.e. number of iterations per second, for the visualization and the fluid simulation (see Fig. 4(a)). The FIFO networks impose the same refresh rate for the visualization and the fluid modules. For the coherent sampling network, the visualization and the fluid run asynchronously. It enables the visualization to run significantly faster than the simulation. The fluid simulation keeps the same performance as in the FIFO case. It shows that the communications induced by synchronizers do not significantly affect the performance. As the number of nodes allocated to the fluid simulation increases, the fluid performance increases too. For the sampling approach this decreases the refresh rate of the visualization modules as they must upload to the graphics card new data from the fluid modules more frequently.

We also measured the overall latency, i.e. the time lag between the time a new tracker position is available and the end of the iteration of the visualization modules using this tracker position (see Fig. 4(b)). Allocating more nodes to the simulation also improves latency. Sampling leads to a better latency than FIFO, because sampling uses the more recent data available while FIFO uses the older one. Note that the FIFO was executed with intermediate buffers of size 2.

The synchronizers used for the sampling approach can be extended to enable a finer control over dependencies between modules. For instance, the synchronizer between the fluid modules and the visualization modules could take into account a user position data to know for each visualization module if the fluid is visible or not. If not, it could block the transmission of fluid grid to the visualization

module, to let the visualization and network resources fully available for objects that are in the user field of view.

4 Conclusion

We introduced FlowVR, a middleware dedicated to distributed interactive applications. FlowVR distinguishes two main parts in an application, the modules and the network. Modules are endless loops reading and writing data on input and output ports. Modules are assembled in a network with advanced features for message handling. It enables parallel code coupling and the design of complex communication and synchronization schemes. First experiences show that FlowVR eases the development and deployment of interactive distributed applications, while leading to high performance executions.

Acknowledgment

This work is partly funded by the RNTL project Geobench.

References

1. FlowVR. http://flowvr.sf.net.
2. OpenMASK. http://www.irisa.fr/siames/OpenMASK.
3. Scirun: A scientific computing problem solving environment. http://software.sci.utah.edu/scirun.html.
4. *Covise Programming Guide*, 2001. http://www.hlrs.de/organization/vis/covise.
5. J. Ahrens, C. Law, W. Schroeder, K. Martin, and Michael Papka. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. http://www.acl.lanl.gov/Viz/papers/pvtk/pvtkpreprint/.
6. 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.
7. 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.
8. A. Denis, C. Pérez, and T. Priol. Padicotm: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 2003.
9. E. Frécon and M. Stenius. Dive: A scalable network architecture for distributed virtual environments. *Distributed Systems Engineering Journa*, 5:91–100, 1998.
10. 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.
11. K. Watsen and M. Zyda. Bamboo - a protable system for dynamically extensible, real-time, networked, virtual environments. In *IEEE Virtual Reality Annual Internationnal Symposium*, Georgia, USA, 1998.

A.5 FlowVR Supercomputing Journal 2008 Article

A Hierarchical Component Model for Large Parallel Interactive Applications

Jean-Denis Lesage and Bruno Raffin

INRIA
Laboratoire d'Informatique de Grenoble (LIG)
Email: jean-denis.lesage@imag.fr and bruno.raffin@imag.fr

Abstract. This paper focuses on parallel interactive applications ranging from scientific visualization, to virtual reality or computational steering. Interactivity makes them particular on three main aspects: they are endlessly iterative, use advanced I/O devices, and must perform under strong performance constraints (latency, refresh rate). A data flow graph is a common approach to describe such applications. Edges represent data streams while vertices are nodes processing incoming data streams and producing new data streams. When applications become large this approach shows its limits in terms of maintainability and portability. In this paper, we propose to use the composite design pattern to extend this model for supporting hierarchies of components. The component hierarchy is traversed to instantiate the application and extract the data flow graph required for the execution. This approach has been implemented for the FlowVR middleware. It enables to define parametric composite components, commonly called skeletons, that can be reused in various applications. This approach proved to significantly leverage application modularity as presented in different case studies.

Keywords: Interactive Applications; Parallelism; Components; Composite Design Pattern

1 Introduction

An interactive application involves a program and a user interacting in an endless iterative process through input and output devices. It is often referred to a "human in the loop simulation". Today, an emerging class of interactive applications intends to associate virtual reality, scientific visualization, simulation and application steering. It leads to very complex applications coupling advanced I/O devices, large data sets, various parallel codes. To be interactive, these applications must perform under strong performance constraints, often measured in terms of latency and refresh rate.

For example, the Hercules system couples an earthquake simulation and an on-line visualization using 2000 processors to reach the frequency of 2Hz on a

1200 billions elements simulation [1]. Other initiatives intend to design cross-continental interactive applications relying on the performance of optical networking [2]. A number of virtual reality applications are relying on parallel machines to provide the required I/O and computing resources. Blue-C [3] and Grimage [4] are good examples of high performance immersive platforms relying on parallel machines to process in real time data acquired through a network of cameras.

In this paper, we focus on two issues faced when designing such applications:

- Software engineering issues where multiple pieces of codes (simulation codes, graphics rendering codes, device drivers, etc.), developed by different persons, during different periods of time, have to be integrated in the same framework to properly work together.
- Hardware performance limitations bypassed by multiplying the units available (disks, CPUs, GPUs, cameras, video projectors, etc.), but introducing at the same time extra complexity. In particular it often requires to introduce parallel algorithms and data redistribution strategies, that should be generic enough to minimize human intervention when the target execution platform changes.

Most iterative applications can be seen as an assembly of static tasks endlessly processing incoming data and forwarding results to other tasks. Many scientific visualization tools use this data flow graph model to specify the applications [5]. But the graph tends to quickly become complex as the application size grows, impairing the modularity.

In this paper, we propose to rely on the composite design pattern to extend the data flow graph model. Edges are components that can recursively contain other components. Vertices link sibling component ports or parent/child ports. To enforce the genericity of the described application, components defer introspection and auto-configuration processes to controllers. A controller is local to a given component, but it may get extra data consulting the state of the neighbor components or through external data repositories. These controllers, that can generate new components for instance, are called recursively and repeatedly in a traverse process until reaching a fixed point. A traverse either leads to an error (missing data impairs the traverse completion) or a success. For instance a traverse is called to extract the data flow graph required for the execution from this hierarchical application description. This approach enables us to define highly generic composite components, enforcing the application maintainability and portability. In particular, we can define skeletons, i.e. parametric composite components, that encapsulate commonly used and optimized parallel processing patterns. This approach has been implemented for the FlowVR middleware [6].

Section 2 discusses related works. After an overview of FlowVR (section 3), we present the hierarchical component model in section 4. Section 5 presents a collection of skeletons built using our model. Section 6 focuses on 2 case studies to discus the benefits of our approach on real applications, before to conclude in section 7.

2 Related Work

The goal of scientific visualization is to process large data sets to compute images. Interactivity enables for instance users to change their point of view on the data set or the transfer function applied for volume rendering. Applications are developed with visualization environments like OpenDX [7], Iris Explorer [8] or VTK [9]. These environments are usually based on a data flow graph model where processing tasks receive data and generate new ones. Most of them support parallel executions. An application is basically a list of filters applied to the data set before rendering. The first natural level of parallelism is to distribute the different steps of the filter pipeline on different machines. Because the data set is read only, the pipeline can easily be duplicated and executed in parallel on sub parts of the data set [10]. Advanced parallel rendering algorithms exist, based for instance on specific parallel data structures and dynamic work balancing schemes. In this case they are implemented on their own, usually using classical parallel programming languages, because visualization environements do not provide the necessary constructs [11].

Attempts to associate virtual reality, scientific visualization and simulations push forward the complexity of interactive applications. They involve various simulation codes that may generate large data sets, advanced I/O devices, like network of cameras, projector arrays, haptic devices. Pipeline must be used with care. It improves the application frequency, but also increases the latency. So to ensure a good trade-off between frequency and latency multiple forms of parallelism are associated, from pipelines or data parallelism to dynamic task parallelism.

In virtual reality, to ensure an efficient data redistribution between parallel algorithms that may run at different and varying frequencies, complex coupling schemes associating data re-sampling and collective communications are required. Dedicated environments like FlowVR [6], OpenMask [12] or COVISE [13] propose different approaches to support such features. However, the resulting application code tends to be difficult to be maintained when reaching a certain size. Connectivity between processing tasks (communication channels) are expressed by direct links between the corresponding elements: it requires the concerned elements be directly visible one from each other, preventing attempts to strongly structure the code by encapsulating patterns in methods or functions.

Component models, like CCA (Common Component Architecture) or CCM (Corba Component Model), provide Architecture Description Languages for distributed applications. SCIRun, an environment dedicated to scientific visualization, is based on the CCA model [14]. Some extensions intend to enforce the support of parallel components and the associated coupling patterns [15]. But these models suffer from the same limitations as the systems mentioned earlier (FlowVR, COVISE) regarding the modularity of parallel component coupling. Fractal [16] is a hierarchical component model. We are aware of one implementation of Fractal for parallel (grid) applications: ProActive [17]. A ProActive composite component can be a parallel component. But redistribution patterns

are coded into the ports of the parallel components. A pattern cannot be modified without modifying the component, limiting the application modularity.

The skeleton model proposes a pattern language for parallel programming [18, 19]. A program is written from the composition of predefined parallel patterns. Various environments rely on this model like ASSIST [20] for grid computing or Skipper [21] for vision applications. Skeletons have a clear semantics, can be associated to a cost model and hide their implementation details to the application developer. Given the target architecture, the application is compiled down to a specialized parallel code. Hierarchies of skeletons are supported by some environments like Skipper-D. ‘ With the emergence of multicore architectures and GPU programming, some programming environments propose to focus on a stream paradigm, like StreamIT [22], Brook [23] or Cg [24]. They target streaming applications like video, voice or DSP programming. A program is usually a set of iterative modules that communicate via FIFO data channels. Parallelism is expressed by the composition of a reduced number of skeletons. For example, in StreamIT, developers are allowed to use 3 kinds of skeletons: Pipeline, SplitJoin and FeedBack Loop. By limiting the available skeletons, it constrains the program to simple data parallel access patterns, enabling to write efficient compilers for the targeted architecture. It is however too restrictive to ensure efficient executions on a general purpose and potentially heterogeneous parallel machine.

3 FlowVR

We present in this section FlowVR [6]. Our component model relies on this middleware. FlowVR is dedicated to parallel interactive applications. It is based on the data flow model also used by other scientific visualization tools. A FlowVR application is a network of static iterative processes connected by data flow channels. The main target applications include virtual reality and scientific visualization.

FlowVR has been used for developing various large interactive applications [25, 26]. FlowVR is open source[1]. It is distributed with extensions like FlowVR-Render that enables distributed rendering or VTK-FlowVR that encapsulates VTK[9] applications into FlowVR Modules [27].

3.1 FlowVR Run-time

An application is composed of modules exchanging data through a FlowVR network. A module is an endless iterative code that defines input and output ports. At each iteration it reads incoming data from input ports, processes these data and writes the results on output ports. A module runs in its own independent process or thread, thus reducing the effort required to turn an existing code into a module. For instance an MPI program can be modified to define one module per process.

[1] FlowVR is available at http://flowvr.sf.net

The FlowVR network is handled at run-time by a FlowVR daemon running on each host of the target machine. Daemons act as brokers. They relay messages between modules. Modules are not aware of the existence of other modules. A module only exchanges data with the daemon that runs on the same host. If the destination module runs on the same host, the daemon gives this module a pointer to the data (messages are stored in shared memory segments). If the destination runs on a distant host, the module sends the data to the daemon of this host using TCP. At reception the daemon stores the message in a shared memory segment and handles a pointer to the destination module.

The role of the daemon is not limited to data forwarding. It can load plugins to process data, duplicate, merge or split messages for instance. The user can define its own plugins if required. Notice that plugins have a less restricted access to the shared memory than modules, enabling to implement more efficient message handling actions.

Each FlowVR application is managed by a special module, called a controller, automatically loaded at starting time. The controller first starts the application's modules using their own launching command, `ssh` or `mpirun` for instance. Once launched, modules register to their local daemon that sends an acknowledgment to the controller. Then, the controller sends to each daemon the routing table and list of plugins to load to implement the FlowVR network.

3.2 Flat Data Flow Graph

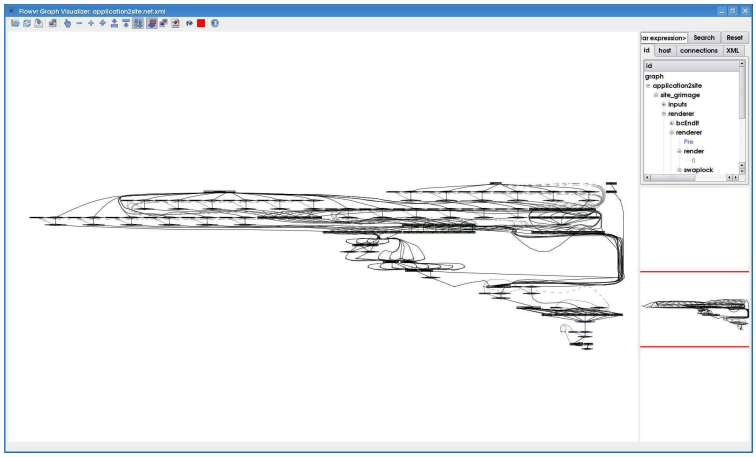


Fig. 1. The flat data flow graph of a large FlowVR application. Edges represent processing tasks and vertices data channels.

At low level a FlowVR application is modeled by a flat data flow graph composed of:

- Modules with input and output ports, each one is mapped on a given host,
- Filters that are daemon plugins. Like modules, filters have input and output ports, and are mapped on a given host.
- Connections that represent FIFO data channels. A connection connects one source input port to a destination output port.
- Routing nodes that have one input port and one or more output ports. They are assigned to a given host and model message routing actions.

In the first versions of FlowVR, the application developer had to specify its application describing this graph. He was assisted by a library of Perl functions that encapsulated some commonly used patterns. However large applications proved difficult to debug and maintain, motivating the adoption of a hierarchical approach to further enforce the application modularity (Fig. 1).

4 Component Model

We adopt a hierarchical component model to describe a FlowVR application. It is based on the composite design pattern [28].

4.1 Hierarchical Components

A component has an interface defined by a set of ports. We distinguish two kinds of components:

Primitive components. A primitive component is a base component that cannot contain an other component. Primitive components are modules, filters, routing nodes and connections.

Composite components. A composite component contains other components (composite or primitive). It has input and output ports. A port is visible from both, the outside and the inside of the component. It identifies the data that can cross a component boundary. Component encapsulation is strict. A component can not be directly contained into two parent components.

4.2 Links

A link connects two component ports. It cannot directly cross a component membrane. A link between 2 ports is allowed only for the 2 following cases:

- A descendant link connects a port of a parent composite component to a port of one of its child component. Such links must always connect an input/input or output/output pair of ports.
- A sibling link connects two ports of two components having the same parent component. Such a link must always connect an input/output pair of ports.

Port typing can be enforced if required, putting more constraints on the ports that can be linked. For instance, link could be restrained to connect only ports corresponding to the same data type.

4.3 Example

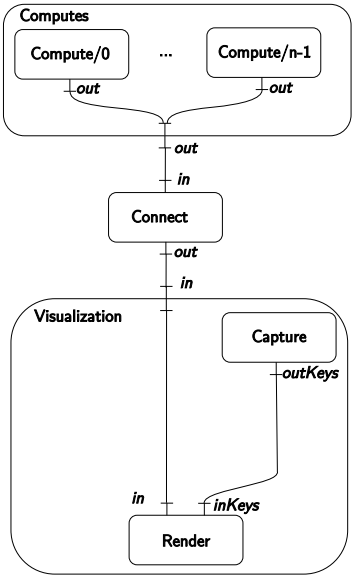


Fig. 2. Application example. *Computes* simulates the dynamics of a ball falling into a water tank. Results transit up to *Render* for rendering. *Capture* forwards mouse positions to *Render* that uses them to render the simulation scene with the point of view requested by the user.

Throughout this paper, we use a simple example (Fig. 2). It shows the classical structure of a basic interactive application. In this iterative simulation, component *Computes* publishes its state at each iteration. We can for instance consider that this simulation computes the dynamics of a ball falling into a water tank. Each simulation state is received by a *Render* component. For a given point of view, this component computes an image giving a view on the simulation scene. The user can control this point of view with a mouse. A *Capture* component is in charge of reading the mouse position and forwarding it to *Render*.

For sake of simplicity, we keep this application synchronous, i.e. the *Render* component can only start the next iteration if it receives data from *Computes* and *Capture*. Often real applications loose this synchronization by introducing data sampling components (a sampling pattern is presented in section 5).

Using the hierarchical component model, the example is structured as follows (Fig. 2):

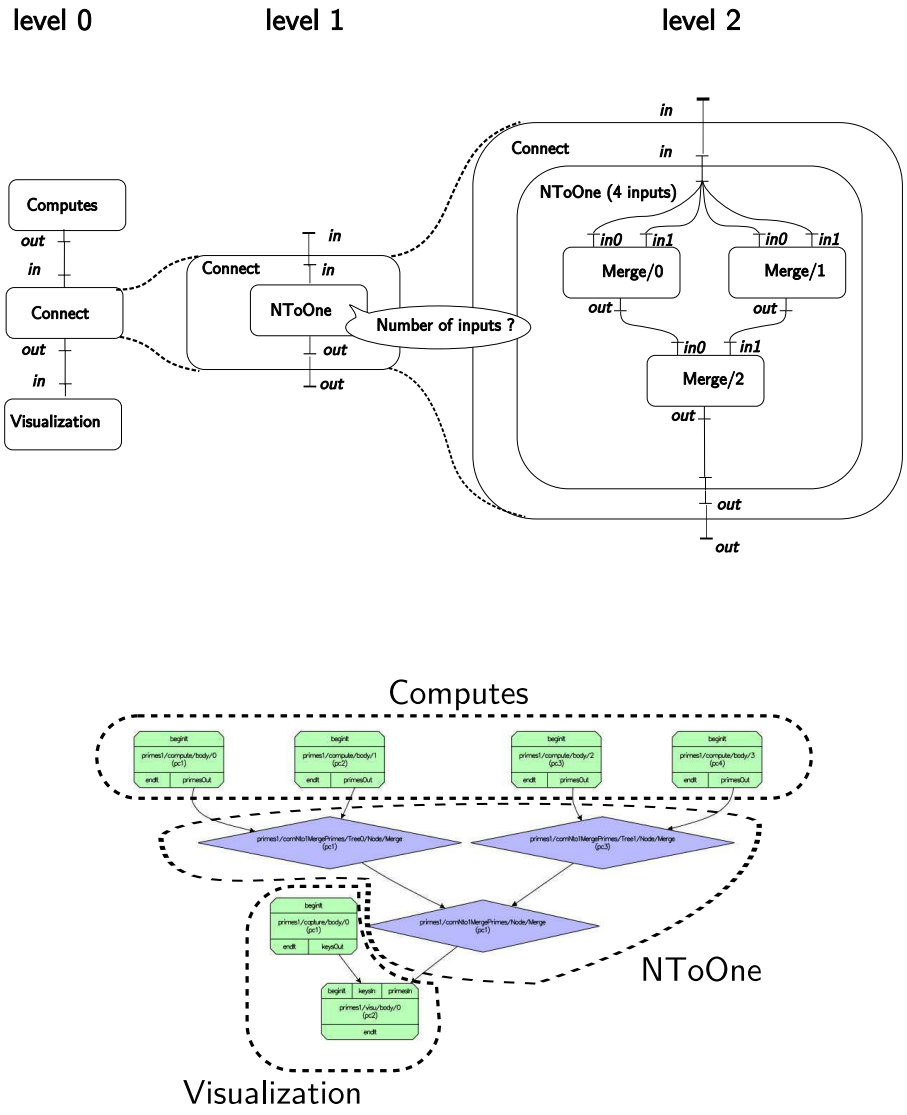


Fig. 3. a) Two levels of hierarchy for the *Connect* component. The skeleton defined by *NtoOne* is generated according to the number of *Computes* primitive components. b) The flat data flow graph for the application. Dashed sets show the composite components the graph elements are related to (connections are arrows, modules are in green and filters in blue).

– As the *Capture* and *Render* components are closely related, they are stored in a composite component called *Visualization*. This encapsulation is a commodity that enables to easily reuse this assembly having just to handle the *Visualization* component.
– The *Computes* component is actually a parallel application that spawns n processes. The goal is to be able to speed-up the simulation involving more processors if available. *Computes* is modeled as a composite component with one output port *out* to send its simulation state at each iteration. It contains n child components *Compute/0,...*, *Compute/n-1*. These are primitive components, each one having an output port *out* linked to the *out* port of *Computes*. The value n and where these n processes are mapped on a target architecture is unknown at the time of the application design. They will be instantiated later when traversing the application to call configuration controllers. Notice that communications can take place between the different parallel processes, but they are not modeled here. We consider that they are under the responsibility of the programming environment used to parallelized the application, MPI for instance.
– *Computes* being a parallel component, each process spawned computes one part of the simulation state. The *Visualization* component is not designed to received partial results. We could modify the *Visualization* component, but we actually prefer to manage this issue outside of this component. Application modularity is enforced by delegating data redistribution issues to specialized components. We use an extra component, called *Connect*, between *Computes* and *Visualization*. *Connect* is in charge of gathering the partial results from the various *Compute/i* processes to forward a single message containing a full simulation state to *Visualization*. *Connect* is a composite component (Fig. 3.a). It is built from the *NtoOne* component. This component encapsulates a generic tree pattern for data redistribution. *Connect* just set the parameters of *NtoOne*: the arity of the tree (2) and the type of the component used for the tree nodes (*Merge*). The actual content of *NtoOne* is only known once *Computes* is properly instantiated. Only at this point *NtoOne* knows how many pieces of data it has to gather to set the tree depth. The *NtoOne* configuration controller must be executed after the *Computes* configuration controller. We see here that the traverse algorithm in charge of executing the configuration controllers has to respect a given processing order. A possible traverse order is: *Computes, Visualization, Connect, NtoOne, Merge/0, Merge/1, Merge/2, Compute/0, Compute/1, Compute/2, Compute/3, Capture, Render*. *Merge* is a primitive component that builds one message sent on its *out* port from the 2 messages it reads on its *in/0* and *in/1* ports.

Notice that if the application is configured with only one component *Compute/0*, *Merge* becomes a simple point-to-point connection between *Compute/0* and *Render*.

The model we propose first target applications with static components, i.e. without components created while the application is running. Because of their

iterative nature, interactive applications tend to be mostly static. However, if required for some parts of the application, a component can dynamically create or kill threads or processes as long as it implements a proxy that hides this dynamic behavior. We are also working on extending the model to support some level of run-time reconfiguration.

4.4 Controllers

To improve the application genericity and thus its portability, instantiation of some component aspects are deferred to controllers. A controller is local to a component. It can only modify the state of its component. It can read the state of other components its owner is linked to (directly or not). A component can have several controllers. It usually enforces modularity to have multiple specialize controllers. We distinguish 2 types of controllers:

- An introspection controller just get data from its component. For instance an introspection controller can be dedicated to print its component name in a file.
- A configuration controller modifies its component state. In the example application, the child components of *Computes* are generated by such a controller.

Controllers are called during an application traverse. Usually one traverse just calls one controller per component. During a traverse, parameters can be exchanged between controllers. It enables for instance to exchange a file descriptor where each controller appends the name of its component. The final result of the traverse is a list of all application components. The result may of course depend on the execution order of the different traverses.

Our model imposes one configuration controller, called *execute*. This controller creates child components. For example, in the *Computes* component, the *execute* controller creates all *Compute/i* primitive components and links them to *Computes*. Data distribution components usually have *execute* controllers that need to get data from the neighbor components. For instance, the *execute* controller of *NtoOne* needs to get the number of *Compute/i* components to create the merging tree.

Developers can create controllers dedicated to a given aspect. A controller can be in charge of mapping primitive components to the target architecture processors. Implementing mapping in a controller enables to keep the application description independent of the mapping. In FlowVR, the application is first traversed to call the *execute* controller, then a mapping controller is called, and a third controller generates the flat data flow graph.

An other example of introspection controller used for FlowVR is the command line generator. The construction of command lines to launch modules is delegated to an introspection controller. This controller builds a command line using data related to the FlowVR network (hosts list, number of processes), configuration files (target architecture description) or user parameters (application

specific parameters). This specific controller is embedded into composite components called metamodules. A metamodule handles modules that are logically related, in particular when they are all started from a single command. This is for instance the case for a MPI code that uses `mpirun` to start all its processes.

Notice that a controller can be seen as an aspect (in the Aspect Oriented Programming way). Nevertheless, we do not have code weaving. Controllers are embedded in components by the programmer.

4.5 Traverse Algorithm

As seen for the example (Section 4.3), in a traverse the execution of controllers may need to obey a certain order to respect data dependencies. We propose a simple algorithm that guarantees to complete the traverse when possible or return the list of misprogrammed components if some data dependencies cannot be solved whatever the execution order is.

The traverse algorithm is a greedy process. The algorithm manages a queue of non-executed components, initialized with the top-level components of the application. For each component in this queue, the algorithm tries to execute the associated controller. If the controller is successfully executed, then all of its children are pushed in the queue. Otherwise, the algorithm restores the component initial state and push it at the end of the queue. The traverse ends successfully when the queue is empty. If no controller can be called on the rest of the components in the list, then the algorithm stops in a fail state. The controller of the remaining components cannot be executed either because at least one of these components is misconfigured (a parameter is not instantiated for instance), or because a cycle of dependencies has been introduced when assembling the components.

4.6 Traverse Proof

We prove the traverse algorithm always ends, with success if a solution exists, and that the number of controller calls, successful or not, is at most quadratic in the number of components.

Let C be the set of all components in an application and N_{comp} the size of C. The goal of the algorithm is to iterate on all components in C with a consistant order. We put in the non-executed queue a marker that denotes the starting point. Each time the marker comes back to the front of the queue, it is appended at the end of the queue. We count the number of times the marker has reached the front since the algorithm started. It denotes what we call in the following the *number of iterations*.

Let $NonExecuted_k = \{c \in C /$ the controller of c has not been executed at the iteration k $\}$ and $Executed_k = \{c \in C /$ the controller of c has been successfully executed during the iteration k $\}$. We call N the iteration that reaches a fixed point, i.e. the first iteration where $Executed_N = \emptyset$. In this case, the algorithm stops. The misconfigured components or dependency cycles are contained in $NonExecuted_N$.

Let $E_k = \bigcup_{i=1}^{k} (Executed_i)$ be the set of components successfully executed from the first to the k^{th} iteration. Let $\overline{E_k} = \bigcup_{i=k}^{\infty} (NonExecuted_i)$ be the set of components that have to be executed after the iteration k.

We call $\overline{E_\infty} = \bigcap_{i=1}^{\infty} \overline{E_i}$ the set of components that cannot be executed.

Thus we have:

- $\forall k, C = E_k \oplus \overline{E_k}$
- $\overline{E_0} = C$ and $E_0 = \emptyset$

We first prove the algorithm always ends.

Proposition 1. *The traverse algorithm reaches a fixed-point with $N \leq N_{comp}$ and $NonExecuted_N = \overline{E_\infty}$*

Proof. During execution of traverse, we are assured that $Executed_N \neq \emptyset$, so for all k we have $\overline{E_{k+1}} = \bigcup_{i=k+1}^{\infty} (NonExecuted_i) \subset \bigcup_{i=k}^{\infty} (NonExecuted_i) \subset \overline{E_k}$. So $\overline{E_N}$ decreases to $\overline{E_\infty}$. The algorithm reaches a fixed-point where $lim_{k\to\infty} \overline{E_k} = \overline{E_\infty}$.

As $C = E_k \oplus \overline{E_k}$, if at the iteration k we have $\overline{E_k} = \overline{E_{k+1}}$ then $E_k = E_{k+1}$. So the algorithm reaches the fixed-point at k.

Consequently $\overline{E_k}$ strictly decreases to $\overline{E_\infty} \Rightarrow N \leq |\overline{E_0}| = N_{comp}$.

We now focus on the complexity of the algorithm.

Proposition 2. *The traverse algorithm performs at most N_{comp}^2 calls to controllers.*

Proof. Let $Calls$ be the total of calls to controller. $Calls = \sum_{k \leq N} |NonExecuted_k|$. Previously, we proved:

- $N \leq N_{comp}$
- $\forall k, NonExecuted_k \subset C \Rightarrow |NonExecuted_k| \leq N_{comp}$

So $Calls \leq N_{comp}^2$

The overhead due to unsuccessful controller calls can be significant. But implementing an algorithm that solves all constraints to identify an acceptable execution order would be complex or it would require the application developer to encode extra information into its program to help that algorithm. Our solution is a good trade-off between scalability and complexity of the implementation. We experimented applications with 200 components. Traverse computation time is about one second only.

We now characterize $\overline{E_\infty}$, the set of remaining components. Let $Data = \{c \in C$ / c cannot be executed because a data is missing $\}$ and $Dep = \{c \in C$ / c cannot be executed because it depends on a component that has not been executed yet $\}$. No other reason can lead to a controller call failure. So we have $\overline{E_\infty} = Data \cup Dep$.

Proposition 3. *If $Data = \emptyset$ and $\overline{E_\infty} = Dep \neq \emptyset$ then there is at least one dependency cycle in $\overline{E_\infty}$*

Proof. Assume there is no dependence cycle in Dep. So there is a longest dependency path. Let c and d be the components at the extremities of one of the longest dependency paths.

But because c belongs to Dep and not to $Data$ ($Data = \emptyset$), there exists e in Dep such as c depends on e. So the path from e to d is longer than the longest path from c to d. It contradicts the assumption: there is a dependence cycle in Dep.

This proposition shows the traverse algorithm can help debugging an application. If the traverse fails, the user should first fix the components with missing data. Usually such flaws are detected when the controller fails if error raising has been properly programmed. Next, if the algorithm still fails, the user should look at suppressing the cyclic dependencies. In our implementation we rely on exceptions to signal when controller fail.

4.7 The FlowVR Front-end

The hierarchical component model only affects the front-end of FlowVR (Fig. 4). The run-time engine is not modified. Components are written in C++ and compiled into shared libraries. An application is also a composite component compiled into a shared library. It can thus be reused in other applications without being recompiled. The FlowVR front-end loads the application and applies a sequence of several traverses to produce the list of commands to start the modules and the instructions to forward to the different daemons to implement the application network. The flat data flow graph is usually saved as it is useful for debugging purpose.

5 Skeletons

We present four base parametric composite components, i.e. skeletons, that proved to be very useful for developing interactive applications. These skeletons provide users an easy way to handle parallel processing patterns or complex communication schemes. These skeletons fully take advantage of the component hierarchy and modularity provided by the controller based approach. They are templated to enforce their genericity. Their instantiation is deferred to their *execute* controller (Sect. 4.4):

Pipeline This is a very simple skeleton modeling a sequence of precessing steps. It is modeled by a composite component containing an arbitrary sequence of linked components (primitives or composite).

Parallel This skeleton creates N instances of a component passed as a template. The skeleton creates the same ports than the template component. Once the internal components created, their ports are linked to their equivalent skeleton ports. The *Computes* component in our example could have been alternatively designed by encapsulating a *Parallel* component pattern using *Compute* as template component. This skeleton can be used as a shell

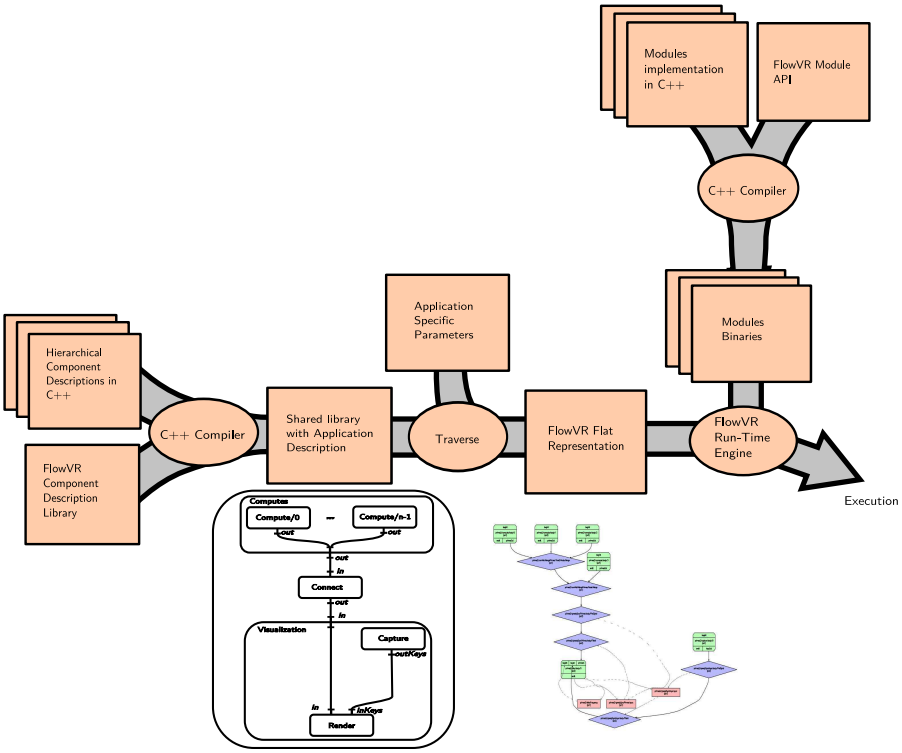


Fig. 4. The FlowVR front-end. Components (left to righ) are compiled, loaded and traversed to provide the module launching commands and the instructions for deamons. Once compiled, modules (top to bottom) are started as requested by the application.

for duplicating a given component. It can also be used to encapsulate a static parallel program. In this case communications due to parallelization are not visible from the component point of view. We consider the parallel programming environment used for the parallelization takes care of these communications.

Tree A tree skeletons has two ports, the root and the leaves. The number of leaves in the tree is defined a traverse time according to the number of neighbors connected to the leaves port. We distinguish 2 specializations of the tree depending on the data propagation direction, either from root to leaves (the *OnetoN* component) or from leaves to root (the *NtoOne* component). The arity of the tree is a parameter to be instantiated. The node type used to build the tree is a template pattern. Here are some examples of components pattern built from *Tree*:

Broadcast The simplest skeleton that can be built from the tree. It uses the *OnetoN* skeleton instantiated with a primitive component, a routing node, that forwards the messages it receives on its input to each of its outputs. The arity of the broadcast tree depends on the number of outputs of the template component.

Scatter Similar to the Broadcast except that the template component splits the input message into sub-messages forwarded on its outputs. For instance, a classical 3D rendering parallelization approach, called sort-first [29], consists in having a task responsible for one area of the screen (Fig. 5.a). Thus, a task only requires to execute the graphics primitives that will contribute to its screen area. To distribute the graphics primitive, we can use a *Scatter* with a *Culling* component that uses a fast method to test if a graphics primitive contributes to a given screen area [27].

Gather A *OneToN* tree that uses a message merging template pattern. Using a template component that sorts the integers it receives, it creates a distributed merge sort (Fig. 5.b). This skeleton is also used by the *Connect* component of our example (Fig. 3).

Sampling This skeleton is specific to interactive applications where tasks may run at different frequencies. For example, a physical simulation has to run at high frequency to be stable, while graphics rendering usually runs between 30 and 60 Hz. If the two tasks are directly connected with a FIFO connection, it will force both tasks to run a the frequency of the slowest one, the rendering task in this case. To avoid this issue a common approach is to sample the incoming signal. This sampling could be performed by the rendering task, making the rendering task less generic. To enforce the modularity, we design a special skeleton that samples data streams under the control of their destination tasks. With this approach neither the source neither the destination tasks need to be modified or even recompiled. The sampling skeleton is an assembly of 2 composite components (Fig. 5.c):

- The *Filter* composite component analyzes and samples the incoming data stream according to an external policy. It has four ports : *in* receives the

incoming data stream, *out* produces the sampled signal, *freq* sends the frequency of the incoming stream and *order* receives sampling orders.
- The *Sampler* composite component controls the sampling policy. It has two ports : *freq* receives the frequency of the incoming stream and *order* sends orders about the stream sampling. Using the incoming stream frequency, it decides the messages that have to be discarded and the ones to replay.

By changing the template components *Sampler* and *Filter* different sampling strategies can be implemented.

Because there is no discontinuity from primitive components to high-level composite ones, the developer can freely choose to combine, extend, specialize or simply ignore these skeletons. In a sense the approach we propose is very close to the one of the C++ Standard Template Library. This skeletons can also be seen as a derivative of Cools skeletons [18] for a specific application domain. One of the main difference is the absence of cost model.

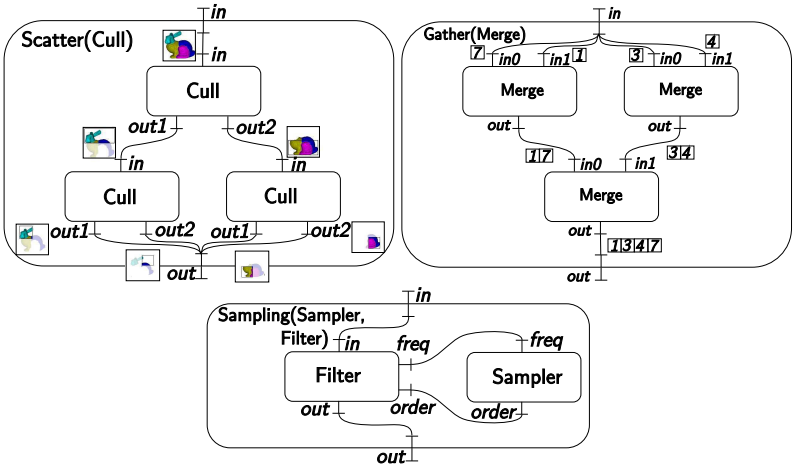


Fig. 5. a) Sort-first scatter pattern. The Culling components route the graphics primitive for rendering the bunny according to the screen area they project onto. b) Integer merge-sort scatter pattern. c) Sampling pattern. *Filter* is the operative part of the communication: it processes sampling on incoming messages flow. *Sampler* is the control part: it decides the sampling policy.

6 Case Studies

In this section, we present two case studies taking advantage of the component hierarchy and the skeletons presented in previous sections. The first application

shows an example of coupling MPI and FlowVR. The second example is an interactive 3D modeling application using a camera network.

6.1 Case Study 1: MPI Fluid Simulation

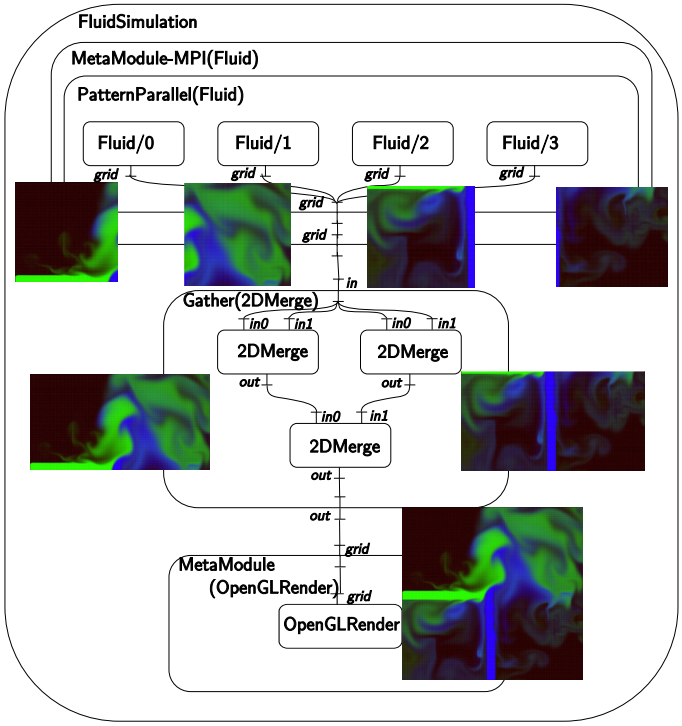
We implemented a fluid simulation algorithm [30] using MPI (Fig. 6.a). This application shows how to integrate a MPI code. The fluid simulation is based on a 2D grid of cells. At each new iteration, a new state is computed for each grid cell. This state depends on the state at the previous iteration of the considered cell and its four neighbors. The simulation is parallelized by splitting the grid cell into blocks distributed amongst the different MPI processes. Data exchange between blocks are MPI communications, transparent to FlowVR. The MPI code is modified so that each process is a FlowVR module with one output port to send the result of each iteration. These modules are called *Fluid/0 .... Fluid/N*, the number being assigned based on the rank provided by MPI. Beside the actual MPI code of the modules, a *Fluid* primitive component is written. A metamodule *Metamodule-MPI* implements the controller to generate the launching command using *mpirun*. It also contain a parallel skeleton that creates the correct number of instances of the *Fluid* module. It is important here that the ranking be the same as the one assigned by MPI. The *Metamodule-MPI* component is linked to a Gather skeleton (Fig. 5.b) using a *2DMerge* filter as template. The goal here is to gather the results of each MPI process into one full 2D grid forwarded to an OpenGL renderer. For more implementation details refer to the fluid example provided with the FlowVR source code.

A possible traverse order to generate the flat data flow graph and the launching commands is:

1. *FluidSimulation* instantiates the 3 components: *MetaModule-MPI(Fluid)*, *MetaModule(OpenGLRender)* and *Gather(2DMerge)*.
2. *MetaModule-MPI(Fluid)* creates the *PatternParallel(Fluid)* component.
3. *PatternParallel(Fluid)* instantiates the 4 *Fluid* modules and set their ranks.
4. *Gather(2DMerge)* detects the 4 *Fluid* and creates the gather tree with 3 *2DMerge* filters. Each *Fluid* is connected to one leave of the gather tree according to their rank.
5. *MetaModule(OpenGLRender)* creates the *OpenGLRender* module.
6. *MetaModule-MPI(Fluid)* generates the MPI command line with the appropriate list of hosts and ranks.
7. *MetaModule(OpenGLRender)* generates the UNIX command line to launch the rendering in the appropriate X-server.

6.2 Case 2: Real-Time 3D Modeling

We ported a parallel real-time 3D modeling application. It consists in computing in real-time a 3D model of a scene from the various 2D video-streams acquired by multiple video cameras surrounding the scene [31] (Fig. 7.a). Real-time 3D




```
void FluidSimulation::execute()
{
 // Components instantiation
       Component* fluid = addObject(
                                  MetaModule-MPI<Fluid>());
       Component* gather = addObject(
                                  Gather<2DMerge>());
       Component* render = addObject(
                                  MetaModule<OpenGLRender>());

 // Link components
       link(fluid->getPort("grid"), gather->getPort("in"));
       link(gather->getPort("out"), render->getPort("grid"));
}
```

Fig. 6. a) Fluid application. Four MPI processes compute a fluid simulation on a 2D grid. A gather skeleton merges results from MPI processes and sends the full grid to an OpenGL renderer. b) C++ description of the FluidSimulation component that encapsulates the application.

modeling enables full body interactions into a virtual environment [4]. 3D modeling is both I/O and computation intensive. We typically use between 6 and

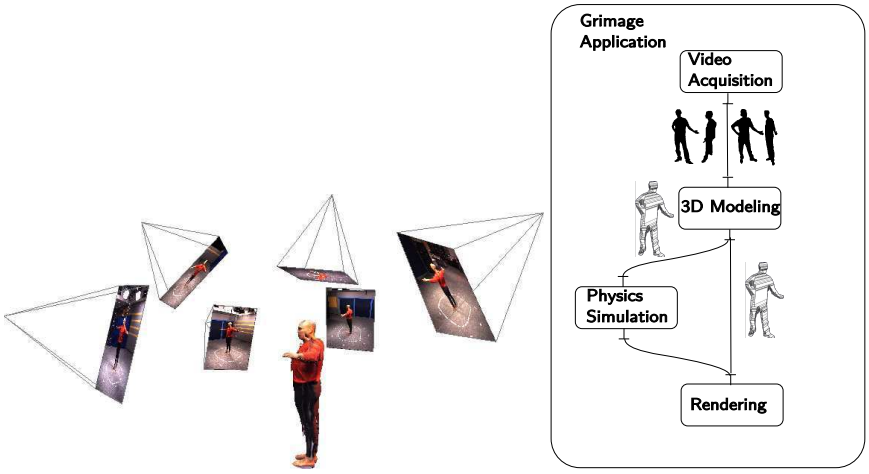


Fig. 7. a) A 3D model of a person computed from 6 cameras. b) Description of the application. This composite contains 4 composite components.

15 cameras, each one acquiring 30 images per seconds. The computation of 3D models must match the camera frequency and run in less than 100 ms per 3D model to keep the overall latency small enough to enable interactions. A parallelization is thus required. Parallelization is based on several steps. For sake of conciseness we just give an overview of the parallel algorithm here. Refer to [32] for details. First, the video stream of each camera is acquired and filtered to subtract the background and compute the image silhouette. This pipeline is executed in parallel on each machine a camera is connected to. Next silhouettes have to be redistributed for computing the 3D model. 3D modeling is implemented with 3 parallel processing steps separated by data redistributions.

This application contains 4 composite components (Fig. 7.b): a video acquisition component, a 3D modeling component, a physical component and a rendering one. The video acquisition component is a parallel pattern containing a pipeline of the different steps from acquisition to silhouette extraction. The 3D modeling component is another hierarchy of components making an intensive use of various skeletons. Physical simulations are computed by SOFA [33], an external framework. This framework computes collisions between virtual objects and user 3D model. The rendering component renders all meshes (virtual objects and user 3D model) and the virtual environment. The application designed is independent from the number of processors available on the target machine, and from the number of cameras and their mapping on the machines.

This application represents a significant development effort involving several developers over several years. Developments started in 2002 using MPI. It was quickly abandoned as MPI proved not to offer a sufficient level of modularity for this type of interactive application. A computer vision specialist should be able to work on the acquisition pipeline without having to worry about the MPI code

or the overall coherency of the communication schemes. We switched to FlowVR that better separates the code of the tasks (the modules) from the task coordination issues. But as the applications grew, for instance texturing of the 3D model started in 2006 and SOFA was only added in 2007, the FlowVR network became very complex and bugs difficult to track and solve. Switching to the hierarchical component approach increased significantly the application maintainability, scalability and portability. It did not directly modify the flat data flow graph and so the performance. But because the modularity improved, performance enhancements proved easier to implement. Several videos are available at http://flowvr.sf.net showing the evolution of the application and the level of performance reached.

Let now focus on the acquisition component. The full pipeline from camera to image silhouette is implemented in a composite component. Using the parallel skeleton, we are able to instantiate this pipeline for all cameras (Fig. 8). These pipelines can be driven from a user interface for on-line tuning of some parameters. To implement this new feature we used 3 parallel skeletons and 1 sampling skeleton (Fig. 9).

Controllers ease extensions of this basic implementation of the acquisition component. For instance, we developed a controller that adds a supervision interface to control these pipelines. This supervision consists in a graphic user interface to set some parameters for the different modules in the pipeline. For example, the user can set the acquisition rate. This interface also displays the outputs from several stages of the pipeline. We use this interface to control and debug the acquisition algorithms. We implemented this controller using a new component that encapsulates the graphics interface. This controller also adds several asynchronous communications that send parameters to pipeline components (Fig. 9). These communications use the sampling skeleton. This implementation enables to separate the main implementation of the pipeline from this supervision aspect. It improves the modularity and provides a simple solution to extend the application.

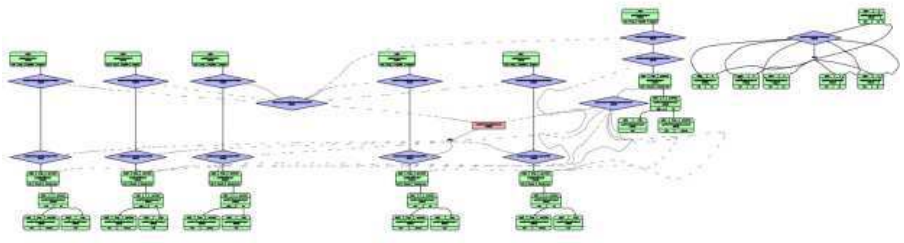


Fig. 8. Flat data flow graph of the acquisition component for 6 cameras (50 nodes and 68 edges).

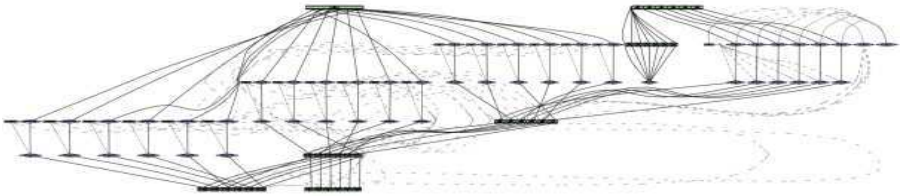


Fig. 9. Flat data flow graph of the acquisition component for 6 cameras and a supervision interface (105 nodes and 176 edges).

7 Conclusion

We presented a framework to use a hierarchical component model for interactive applications. Our main goal was to ensure a high level of modularity for large applications involving parallel components and advanced coupling schemes. Configuration of components is deferred to controllers. It enables us to separate some aspects of a component from its core functional nature. Controllers are called in a traverse process. We presented a traverse algorithm that calls the controllers in an appropriate order or produce an error if completion is not possible due to cycles or missing data. This approach was implemented for the FlowVR middleware and proved effective to leverage the modularity of applications.

Acknowledgment

This work was partly funded by Agence Nationale de la Recherche contract ANR-06-MDCA-003.

References

1. Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.L., O'Hallaron, D.R.: From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In: Super Computing. (2006)
2. Smarr, L.L., Chien, A.A., DeFanti, T., Leigh, J., Papadopoulos, P.M.: The OptIPuter. Communication of the ACM **46**(11) (2003) 58–67
3. Gross, M., Wuermlin, S., Naef, M., Lamboray, E., Spagno, C., Kunz, A., Koller-Meier, E., Svoboda, T., Gool, L.V., S. Lang, K.S., Moere, A.V., Staadt, O.: Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In: Proceedings of ACM SIGGRAPH 03, San Diego (2003)
4. Allard, J., Ménier, C., Raffin, B., Boyer, E., Faure, F.: Grimage: Markerless 3D Interactions. In: Proceedings of ACM SIGGRAPH 07, San Diego, USA (August 2007) Emerging Technology.
5. Brodlie, K., Duce, D.A., Gallop, J.R., Walton, J.P.R.B., Wood, J.: Distributed and Collaborative Visualization. Computer Graphics Forum **23**(2) (2004) 223–251

6. Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a Middleware for Large Scale Virtual Reality Applications. In: Proceedings of Euro-par 2004, Pisa, Italia (August 2004)
7. Lucas, B., Abram, G.D., Collins, N.S., Epstein, D.A., Gresh, D.L., McAuliffe, K.P.: An Architecture for a Scientific Visualization System. In: IEEE Visualization Conference, Los Alamitos, CA, USA, IEEE Computer Society Press (1992) 107–114
8. Foulser, D.: IRIS Explorer: a Framework for Investigation. Journal of ACM SIGGRAPH 95 **29**(2) (1995) 13–16
9. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition. Kitware, Inc. (2003)
10. Ahrens, J., Law, C., Schroeder, W., Martin, K., Papka, M.: A Parallel Approach for Efficiently Visualizing Extremely Large Time-Varying Datasets. Technical report, Los Alamos National Laboratory, Los Alamos National Laboratory (2000)
11. Wang, C., Gao, J., Shen, H.W.: Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. Parallel Computing **31**(2) (February 2005) 185–204
12. D.Margery, B.Arnaldi, A.Chauffaut, S.Donikian, T.Duval: OpenMASK: Multi-Threaded or Modular Animation and Simulation Kernel or Kit : a General Introduction. In Richir, S., Richard, P., Taravel, B., eds.: VRIC 2002 Proceedings. (2002) 101–110
13. A.Wierse, U.Lang, Rhle, R.: Architectures of Distributed Visualization Systems and their Enhancements. In: Eurographics Workshop on Visualization in Scientific Computing, Abingdon (1993)
14. Keming Zhang and Kostadin Damevski and Venkatanand Venkatachalapathy and Steven G. Parker: SCIRun2: A CCA Framework for High Performance Computing. In: Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS. Volume 00., Los Alamitos, CA, USA, IEEE Computer Society (2004) 72–79
15. Denis, A., Pérez, C., Priol, T.: PadicoTM: an Open Integration Framework for Communication Middleware and Runtimes. Future Generation Computer Systems **19**(4) (2003) 575–585
16. Eric Bruneton and Thierry Coupaye and Matthieu Leclercq and Vivien Quéma and Jean-Bernard Stefani: The FRACTAL Component Model and its Support in Java: Experiences with Auto-Adaptive and Reconfigurable Systems. Software Practice & Experience **36**(11-12) (2006) 1257–1284
17. Baude, F., Caromel, D., Morel, M.: From Distributed Objects to Hierarchical Grid Components. In: CoopIS/DOA/ODBASE. (2003) 1226–1242
18. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
19. Mattson, T.G., Sanders, B.A., Massingill, B.L.: A Pattern Language for Parallel Programming. Addison Wesley (2004)
20. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a research framework for high-performance Grid programming environments. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software environments and Tools. Springer (January 2006)
21. Serot, J., Ginhac, D.: Skeletons for parallel image processing: an overview of the skipper project. Parallel Computing **28**(12) (December 2002) 1685–1708
22. William Thies and Michal Karczmarek and Saman P. Amarasinghe: StreamIt: A Language for Streaming Applications. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 179–196

23. Ian Buck and Tim Foley and Daniel Horn and Jeremy Sugerman and Kayvon Fatahalian and Mike Houston and Pat Hanrahan: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transaction on Graphics **23**(3) (2004) 777–786
24. William R. Mark and R. Steven Glanville and Kurt Akeley and Mark J. Kilgard: Cg: a System for Programming Graphics Hardware in a C-like Language. In: Proceedings of ACM SIGGRAPH 03, New York, NY, USA, ACM Press (2003) 896–907
25. Arcila, T., Allard, J., Ménier, C., Boyer, E., Raffin, B.: Flowvr: A framework for distributed virtual reality applications. In: 1ière journées de l'Association Française de Réalité Virtuelle, Augmentée, Mixte et d'Interaction 3D, Rocquencourt, France (November 2006)
26. Allard, J., Raffin, B.: Distributed Physical Based Simulations for Large VR Applications. In: IEEE Virtual Reality Conference, Alexandria, USA (March 2006)
27. Allard, J., Raffin, B.: A Shader-Based Parallel Rendering Framework. In: IEEE Visualization Conference, Minneapolis, USA (October 2005)
28. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
29. Molmar, S., Cox, M., Ellsworth, D., Fuchs, H.: A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications **14**(4) (1994) 23–32
30. Jos Stam: Stable fluids. In: Proceedings of ACM SIGGRAPH 99, New York, NY, USA, ACM Press (1999) 121–128
31. Franco, J., Boyer, E.: Exact Polyhedral Visual Hulls. In: Proceedings of BMVC2003. (2003)
32. Allard, J., Boyer, E., Franco, J.S., Ménier, C., Raffin, B.: Marker-less Real Time 3D Modeling for Virtual Reality. In: Proceedings of the Immersive Projection Technology Workshop, Ames, Iowa (May 2004)
33. Allard, J., Cotin, S., Faure, F., Bensoussan, P.J., Poyer, F., Duriez, C., Delingette, H., Grisoni, L.: SOFA: an Open Source Framework for Medical Simulation. In: Medicine Meets Virtual Reality (MMVR). (2007)

A.6 Parallel EPVH ICVS 2006 Article

The GrImage Platform: A Mixed Reality Environment for Interactions

Jérémie Allard Jean-Sébastien Franco Clément Ménier Edmond Boyer
Bruno Raffin

Laboratoire Gravir, Laboratoire ID
CNRS/INPG/INRIA/UJF
INRIA Rhône-Alpes
655 avenue de l'Europe, 38334 Saint Ismier, France

E-mail: firstname.lastname@inrialpes.fr

Abstract

In this paper, we present a scalable architecture to compute, visualize and interact with 3D dynamic models of real scenes. This architecture is designed for mixed reality applications requiring such dynamic models, tele-immersion for instance. Our system consists in 3 *main parts: the acquisition, based on standard firewire cameras; the computation, based on a distribution scheme over a cluster of PC and using a recent shape-from-silhouette algorithm which leads to optimally precise 3D models; the visualization, which is achieved on a multiple display wall. The proposed distribution scheme ensures scalability of the system and hereby allows control over the number of cameras used for acquisition, the frame-rate, or the number of projectors used for high resolution visualization. To our knowledge this is the first completely scalable vision architecture for real time 3D modeling, from acquisition to visualization through computation. Experimental results show that this framework is very promising for real time 3D interactions.*

1 Introduction

Interactive and mixed reality environments generally rely on the ability to retrieve 3D information about users, in real time, in an interaction space. Such information is used to make real and virtual worlds consistent with one another. Traditional solutions to this problem usually consist in tracking positions of sensors by means of various technologies including electromagnetic waves, infrared sensors or accelerometers. However, this requires users to wear invasive equipment and usually specific body suits. Furthermore it does not lead to a shape description, as required for many applications such as tele-immersion for example. In this paper, we consider a more flexible class of methods based on digital cameras. These methods can compute 3D shape models in real-time, and without any markers or any specific equipment. We propose a framework in this context, from acquisition to visualization and interactions. Our objective is to provide a flexible solution which especially focuses on issues that are critical in such systems: precision of the 3D model, precision of the visualization and process speed.

Several multi-camera systems for dynamic modeling have been proposed. Stereo based systems were first proposed [16] for *virtualization*, but most recent systems use image silhouettes as input data to compute 3D shapes. They can be classified according to the fact that they work offline or in real-time, and also by the type of 3D models which they build. Offline systems allow complex and precise models to be built [6, 5], in particular articulated models, however they do not allow real-time interaction as intended in this work. Most real-time systems, such as [7, 10], that have been proposed in the past, compute voxel models, i.e. discrete 3D models made of elementary parallelepipedic cells. Interestingly, several systems in this category [4, 12, 3, 18] use a distribution scheme over a PC cluster to speed up computations and hence, provide some kind of control over the model precision and the process speed. However, voxel based methods are still imprecise unless a huge number of voxels is used. Furthermore they require post-processing, typically a marching cubes approach, to produce surface shapes. This is computationally expensive, and generates very small-scale geometry whenever precision is required.

Another class of real time, but non-parallel, approaches directly render new viewpoint images [17] using possibly

graphic cards for computations[14]. Based on the *Image Based Visual Hull* method [15], these approaches efficiently focus on the desired 2D image, but they still rely on a single PC for computations, limiting the number of video-streams or the frame-rate, and they do not provide explicit 3D shapes as required by many applications.

In contrast to the aforementioned systems, ours directly computes watertight and manifold surface models. These surface models are exact with respect to the input silhouette information available and, as such, are optimal and equivalent to voxel grids with infinite resolutions. A particular emphasis has been put on the system scalability to ensure flexibility and to address performance and hardware cost efficiency issues. To this aim, the system is composed of multiple commodity components: FireWire cameras distributed on multiple PCs interconnected through a standard Ethernet network, as well as multiple projectors for a wall display. To reach real time performance, a careful distribution of the work load on the different resources is achieved. For that purpose we rely on a middleware library [1], dedicated to the distribution of interactive applications.

Section 2 outlines the global approach. Section 3 discusses issues related to image acquisition. The 3D modeling algorithm and its parallel implementation is then explained in section 4. In section 5, interactions and visualization are described. Section 6 details the distributed framework for our system. Section 7 presents some experimental results before concluding in section 8.

2 Outline

Our goal is to compute 3D shapes of users in an acquisition space surrounded by several cameras in real time (see figure 1). Such models are subsequently used for interaction purposes, including display. In order to achieve this, several processes must be coupled.

Acquisition Fixed cameras are set to surround the scene. Their calibration is obtained offline through off-the-shelf libraries such as OpenCV. Each camera is handled by a dedicated PC. Each acquired image is locally analyzed to extract regions of interest (the foreground) which are then vectorized, i.e. their delimiting polygonal contours are computed.

3D modeling A geometric model is then computed from the silhouettes using an efficient method to compute the *visual hull* [13]. Obtained visual hull polyhedrons are sufficient for numerous VR applications including collision detection or virtual shadow computation for instance. To reach a real time execution, their computation is distributed among different processors.

Interactions and Visualization The 3D mesh is asynchronously sent to the interaction engines and to the visualization PCs. Multi-projector rendering is handled by a mixed replicated/sort-first approach.

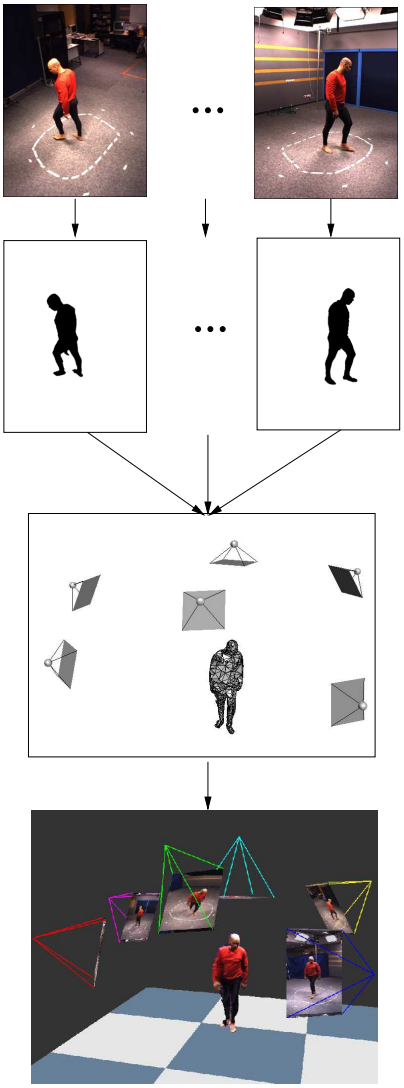


Figure 1. From multi-camera videos to dynamic textured 3D models

3 Acquisition

Acquisition takes place on a dedicated set of PCs, each connected to a single camera. These PCs perform all necessary preliminary image processing steps: color image acquisition, background subtraction and silhouette polygonalization (see figure 2). All cameras are standard firewire cameras, capturing images at 30 fps with a resolution of 780x580 in YUV color space.

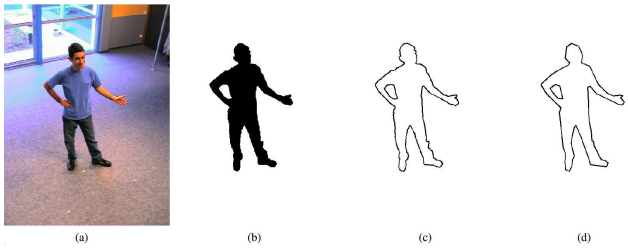


Figure 2. The different steps in the acquisition process: (a) the original image; (b) the binary image of the silhouette; (c) the exact silhouette polygon (250 vertices); (d) a simplified silhouette polygon (55 vertices).

3.1 Synchronization

Dealing with multiple input devices raises the problem of data synchronization. Indeed, our applications rely on the assumption that the input data chunks received from different sources are coherent, i.e. that they relate to the same scene event. We use an hardware synchronization where image acquisition is triggered by externally gen-locking the cameras, ensuring a delay between images below $100\mu s$.

3.2 Background Subtraction

Regions of interest in the images, i.e. the foreground or silhouette, are extracted using a background subtraction process. As most of the existing techniques [11, 7], we rely on a per pixel color model of the background. For our purpose, we use a combination of a Gaussian model for the chromatic information (UV) and an interval model for the intensity information (Y) with a variant of the method by Horprasert *et al.* [11] for shadow detection. A crucial remark here is that the quality of the produced 3D model highly depends on this process since the modeling approach is exact with respect to the silhouettes. Notice that a high quality background subtraction can easily be achieved by using a dedicated environment (blue screen). However, for prospective purposes, we do not limit ourself to such specific environments in our setup.

3.3 Silhouette Polygonalization

Since our modeling algorithm computes a surface and not a volume, it does not use image regions as defined by silhouettes, but their delimiting polygonal contours. We extract such silhouette contours and vectorize them using the method of Debled *et al.* [8]. Each contour is decomposed into an oriented polygon, which approximates the contour to a given approximation bound. With a single-pixel bound, obtained polygons are strictly equivalent to the silhouettes in the discrete sense (see figure 2-c). However in case of noisy silhouettes this leads to numerous small segments. A higher approximation bound results in significantly fewer segments (see figure 2-d). This enables to control the model complexity, and therefore the computation time, in an efficient way.

4 3D Modeling

The visual hull is a well studied geometric shape [13] which is obtained from a scene object's silhouettes observed in n views. It is the maximum shape consistent with all silhouettes. As such, it can be seen as the intersection of the images' *viewing cones*, the volumes that backproject from each view's silhouette (see figure 3).

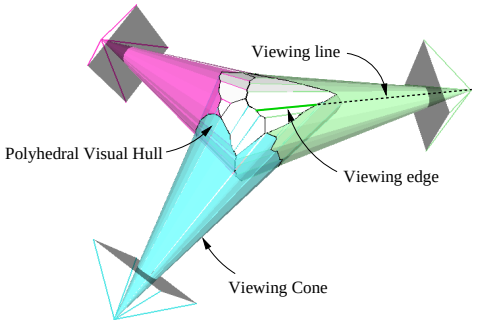


Figure 3. Visual hull of a sphere with 3 views.

We use a distributed surface-based method we have developed [9]. It recovers the exact polyhedral visual hull from the input silhouette polygons in three steps. First, a subset of the polyhedron edges – the viewing edges – is computed. Second, starting from this partial description of the polyhedron's mesh, all other edges and vertices are recovered by a recursive series of geometric deductions. Third, the shape's faces are recovered by traversing the obtained mesh. The following paragraphs briefly detail these steps, and their distribution over p CPUs.

4.1 Computing the Viewing Edges

Viewing edges are intervals along viewing lines associated from silhouette contours' vertices. They are obtained by computing the set of intervals along such a viewing line that project inside all silhouettes. The distribution of this computation uses the fact that each viewing line's contributions can be computed independently. Viewing lines are partitioned into p identical cardinality sets and each batch is

distributed to a different CPU. The final set is obtained by gathering partial results.

4.2 Computing the Visual Hull Mesh

The viewing edges give us an initial subset of the visual hull geometry. The missing chains of edges, are then recovered recursively starting from the viewing edges set. To allow concurrent task execution, the 3D space is partitionned into p slices. Slice width is adjusted by attributing a constant number of viewing edge vertices per slice for workload balancing. Each CPU computes the missing edges in its assigned slice. Partial meshes are then gathered and carefully merged across slice borders.

4.3 Computing the Faces

Faces of the polyhedron surface are extracted by walking through the complete oriented mesh while always taking left turns at each vertex, so as to identify each face's contours. Each CPU independently computes a subset of the face information, the complete mesh being previously broadcasted to each CPU.

5 Interactions and Visualization

5.1 Real-Time interactions

We experimented two different interactions. The first one consists in a simple object carving (see figure 4(a)). The user can sculpt an object using any part of his body. This is done with octree-based boolean operations to update the object where it intersects with the user's model. Update operations include removal, addition of matter and change in sculpture color. The object can be rotated to simulate a potter's wheel.

The second interaction results from the integration of the user's model inside a rigid body simulation (see figure 4(b)). Several dynamic objects where added in the scene, and the system handles collisions with the user's body. This interaction requires all available information about the user's 3D surface, which is not available using classical tracking methods. Using our surface modeling approach, such fine level collision detection is something our system can achieve.

5.2 Multi-projector Visualization

To provide the user with a wide field of view while preserving image details, as necessary in semi-immersive and immersive applications, we have chosen to use a multi-projector display. The most scalable approach to implement

(a) Carving

(b) Collision

Figure 4. Interaction experiments.

this setup is to use one PC to drive each projector. To obtain a coherent image, each PC will have to synchronously render the same scene with a different view point, corresponding to the position of the related projector.

Several methods are available to implement parallel visualization, depending on the level of the primitives exchanged. We use a new framework [2], allowing to use a different scheme for each part of the scene. Large static objects, such as the landscape, use a replicated scheme so that they are sent locally on each PC. Other objects, such as the reconstructed mesh, are created on specific PCs and then sent to all visualization PCs, possibly culling invisible data (sort-first scheme).

The rendering of the 3D mesh itself is quite simple as it is already a polygonal surface. We can optionally compute averaged normal vectors at each vertex to produce a smoothly shaded rendering. It is relatively small (approximately 10000 triangles) so it can be broadcasted to all visualization PCs.

6 Implementation

6.1 The middleware library

To provide the I/O and computing power necessary to run our applications in real time, we use a PC cluster. However, coupling all pieces of code involved, distributing them

on the PCs and insuring data transfers can be cumbersome. To get a high performance and modular application, we use a tool we developed [1] to manage distributed interactive applications. It relies on an data-flow model. Computation and I/O tasks are encapsulated into modules. Each module endlessly iterates, consuming and producing data. Modules are not aware of the existence of other modules. A module only exchanges data with the middleware daemon that runs on the same host. The set of daemons running on a PC cluster are in charge of implementing the data exchange network that connects modules. Daemons use TCP connections for network communications or shared-memory for local communications. The middleware network defined between modules can implement simple module-to-module connections as well as complex message handling operations like synchronizations, data filtering operations, data sampling, broadcasts, etc. This fine control over data handling enables to take advantage of both the specificity of the application and the underlying cluster architecture to optimize the latency and refresh rates.

6.2 Data-flow Graph

We propose for our application the following distributed data-flow graph from acquisition to rendering (see figure 5).

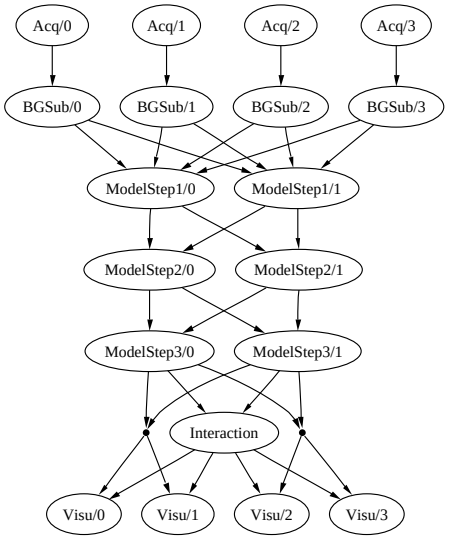


Figure 5. Data-flow graph from 4 cameras acquisitions to 4 video projectors rendering.

Each dedicated acquisition PC locally performs the data acquisition to obtain the silhouettes which are then broadcasted to the PCs in charge of the first modeling step, the viewing edge computation step. Follows the two other modeling steps, the global mesh recovery and the surface extraction. The resulting reconstructed surface is broadcasted to the PCs in charge of interaction computation and to the visualization hosts. These PCs also receive data from the interaction modules of the VR environment.

To obtain good performance and scalability it is necessary to setup specific coupling policies between the different parts of the application so they can run at different frequencies. The acquisition part typically runs at the frequency of the cameras while interactions run at more than $100Hz$. The visualization stage runs independently, allowing to change the viewpoint without waiting for the computation of the next 3D model. To implement these coupling policies we use two dataflow control policies: *FIFO* connections between modules running at the same frequency and *greedy sampling* connections (receivers always use last available data) between modules running asynchronously.

7 Results

We present the results obtained with our platform. It gathers 11 dual-Xeon 2.6 GHz PCs and 16 dual-Opteron PCs connected together by a gigabit Ethernet network. 6 FireWire Cameras are connected to the dual-Xeon machines. 16 projectors are connected to the dual-Opteron machines through NVIDIA 6800 Ultra graphics cards. The projectors display images on a flat screen of 2.7×2 meters. The acquisition space where the cameras are focused is located 1 meter away from the screen.

To evaluate the potential of 3D modeling for interaction purposes, we identified the following classical criteria as being relevant:

- Latency: it is the delay between a user's action and the perception of this action on the displayed 3D model. It is the most important criterion. A large latency can significantly impair the interaction experience. For most experiments on our system the overall latency, including all stages from video acquisition to visualization, was around $100ms$. This can be noticed by the user but is small enough to maintain a high level of interactivity. The quality of the background subtraction step as well as the simplification threshold applied to the resulting contours have a high impact on the latency as they determine the computational cost of the 3D modeling.

- Update frequency (modeling framerate): in our experiments, using 4 CPUs was enough to provide an update frequency of 30 Hz with 6 cameras when one user was in the interaction space.

- Quality (model's level of detail): in our experiments, the user was able to use its hands to carve virtual objects, and, depending on the angle relative to the cameras, it was possible to distinguish his fingers.

- Robustness to acquisition noise: our modeling algorithm is exact with respect to provided input silhouettes however noisy. The resulting 3D model is always watertight (no holes) and manifold (no self intersections). These properties are very important as many interaction applications or visualization (shadows, ...) rely on them. Moreover the approximation of silhouette contours removes most of the background subtraction noise.

- Model Content (the type of information available, surfaces, and textures in our case). When texturing the 3D models with the images obtained from the cameras, this property enables to avoid artefacs (see figure 6). Notice that in the applications presented the model is not textured. Real-time texturing is a challenging issue as the amount of data to handle in a distributed context is important. This is an ongoing work with very promising preliminary results.

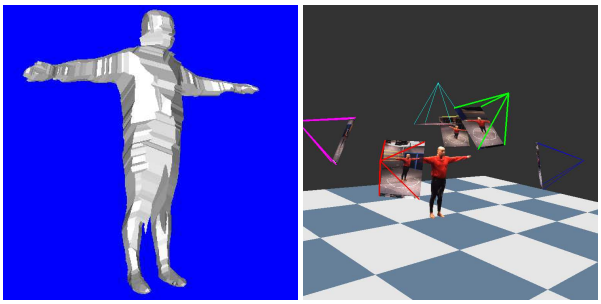


Figure 6. Details of a 3D model and its textured version (off-line).

8 Conclusion

We presented a marker-less 3D shape modeling approach which optimally exploits all the information provided by standard background subtraction techniques and produces watertight 3D models. The shape can easily be used for visual interactions, like rendering, shading, object occlusion, as well as mechanical interactions, like collision detection with other virtual objects. I/O devices and computing units are commodity components (FireWire cameras, PCs, gigabit Ethernet network, classroom projectors). They provide a scalable and efficient environment. The aggregation of multiple units and an adequate work-load distribution enable us to achieve real time performance.

Future works investigate two directions. One is to focus on data quality, in particular background subtraction and temporal consistency. The other is to focus on recovering semantic information about scene objects. The goal is to identify parts of the user's body for motion tracking, gesture recognition and more advanced interactions with the virtual world.

References

[1] FlowVR. `http://flowvr.sf.net`.

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

[3] D. Arita and R.-I. Taniguchi. Rpv-ii: A stream-based real-time parallel vision system and its application to real-time volume reconstruction. In *Proceedings of ICVS, Vancouver (Canada)*, 2001.

[4] Eugene Borovikov and Larry Davis. A Distributed System for Real-time Volume Reconstruction. In *proceedings of CAMP-2000, IEEE*, 2000.

[5] J. Carranza, C. Theobalt, M. Magnor, and H.P. Seidel. Free-viewpoint video of human actors. In *Proc. of ACM SIGGRAPH, San Diego*, pages 569–577, 2003.

[6] G. Cheung, S. Baker, and T. Kanade. Visual Hull Alignment and Refinement Across Time: A 3D Reconstruction Algorithm Combining Shape-From-Silhouette with Stereo. In *Proceedings of CVPR, Madison*, 2003.

[7] G. Cheung, T. Kanade, J.Y. Bouguet, and M. Holler. A real time system for robust 3d voxel reconstruction of human motions. In *Proceedings of CVPR, Hilton Head Island*, volume 2, pages 714 – 720, June 2000.

[8] I. Debled-Rennesson, S. Tabbone, and L. Wendling. Fast Polygonal Approximation of Digital Curves. In *Proceedings of ICPR*, volume I, pages 465–468, 2004.

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

[10] J.-M. Hazenfratz, M. Lapierre, J.-D. Gascuel, and E. Boyer. Real Time Capture, Reconstruction and Insertion into Virtual World of Human Actors. In *Vision, Video and Graphics Conference*, 2003.

[11] T. Horprasert, D. Harwood, and L.S. Davis. A Statistical Approach for Real-time Robust Background Subtraction and Shadow Detection . In *IEEE ICCV'99 frame-rate workshop*, 1999.

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

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

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

[15] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image Based Visual Hulls. In *Proceedings of ACM SIGGRAPH*, pages 369–374, 2000.

[16] P.J. Narayanan, P.W. Rander, and T. Kanade. Constructing Virtual Wolrds Using Dense Stereo. In *Proceedings of ICCV, Bombay, (India)*, pages 3–10, 1998.

[17] W. Stephan, L. Edouard, and G. Markus. 3D Video Fragments: Dynamic Point Samples for Real-time Free-Viewpoint Video. *Computers & Graphics*, 28(1):3–14, 2004.

[18] X. Wu and T. Matsuyama. Real-Time Active 3D Shape Reconstruction for 3D Video. In *Proceedings of the 3rd International Symposium on Image and Signal Processing and Analysis, Rome (Italy)*, pages 186–191, September 2003.

A.7 Parallel Octree Carving EGPGV 2007 Article

Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling

Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch.

INRIA, Laboratoire d'Informatique de Grenoble - LIG, Grenoble, France

Abstract

This paper introduces a dynamic work balancing algorithm, based on work stealing, for time-constrained parallel octree carving. The performance of the algorithm is proved and confirmed by experimental results where the algorithm is applied to a real-time 3D modeling from multiple video streams. Compared to classical work stealing, the proposed algorithm enforces a relaxed width first octree carving that enables to stop computations at anytime while ensuring a balanced carving.

Categories and Subject Descriptors (according to ACM CCS): C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors) I.4.5 [Image Processing and Computer Vision]: Reconstruction

1. Introduction

Mastering parallelism is a major challenge when developing computationally intensive interactive applications, but it can enable to reach the targeted low latencies and high refresh rates.

This article presents a dynamic load balancing algorithm, based on work stealing, that enables to stop computations at any time. This algorithm is applied to 3D modeling. 3D Modeling consists in building a 3D model of people or objects being filmed by a set of calibrated cameras. This 3D model must be computed in real time from the different video streams before it is injected into the virtual world to enable interactions [MP04, GWN*03]. Many different algorithms exist for 3D modeling. A classical approach, the one our experimental results rely on, is to "carve" an octree [Sze93]. 3D modeling has been selected as it is an interesting case study where interactivity is critical and the amount of computations to perform can be significant.

Parallelizing such octree carving algorithm raises two main isssues:

- load-balancing: the shape of the octree is irregular and depends on the input data. Thus a static load balancing scheme fails to guarantee an efficient use of the available computing resources. An efficient dynamic load balancing is required.
- time-constraint: when the timeout occurs we expect the octree to be balanced, i.e. that computing resources coordinate their efforts to avoid having a branch deeply explored while an other is seldom tested. It requires extra synchronizations between processors to enforce a relaxed width first octree carving.

To achieve this goal we propose a modified work-stealing technique. The efficiency of this approach is validated formally and experimentally.

Our parallel algorithm dynamically balances the processor work load and ensures it can be stopped at anytime while guaranteeing a balanced space carving. The implementation on a 16 cores architecture (8 dual-core processors) speeds up the computations up to 14.4 times in comparison to the sequential execution. We also present early experiences using one GPU as a co-processor. The performance increases by 30% compared to using only one CPU, and fades as the number CPUs involved increases.

The paper is organized as follow. Section 2 presents the octree based algorithm for 3D Modeling. Work stealing and the associated theoretical results are detailed in Section 3. The parallel octree algorithm is presented in Section 4, its proof in Section 5, its implementation in Section 6 and the experimental results in Section 7. Section 8 discusses the GPU based tests before conclusions are draw in Section 9.

2. Octree Based Voxel Carving

We present in the following the sequential octree based voxel carving algorithm. The algorithm takes as input data video streams from a network of cameras (Fig. 1). To ensure a high quality modeling, cameras must be properly calibrated (brightness, color and position) and synchronized. From each image a 2D silhouette is extracted by background substraction. Various methods exist [HHD99]. we rely on [KGYS05]. Pixels outside the silhouettes are set to white, while the others are set to black (Fig. 2). The octree algorithm is executed on each set of silhouette images taken at the same time. Starting from one initial voxel corresponding to the acquisition space, the algorithm probes each voxel to compute if it lies outside or inside the visual hull, i.e. if the voxel projects at least outside one silhouette or inside all silhouettes. Uncertain voxels (intersecting a silhouette contour) are split in 8 smaller voxels. A stopping condition can be set to bound the execution time, based on a timer or a maximum depth level for instance.

Figure 1: *6 Cameras filming a user.*

Figure 2: *Silhouettes from 3 cameras.*

There are several approaches to test if a voxel is full (inside the visual hull) or not. We are using an exhaustive method where a voxel is subdivided into a regular 3D grid. The algorithm iterates on each grid vertex, projecting it into the silhouette. Computations for this voxel are stopped as soon as the voxel is known to be full, empty or uncertain. The number of grid vertices contained in a voxel is proportional to its size.

This algorithm is interesting on different aspects. First, as all octree based approaches, it enables to reduce the amount of computations, compared to a space partitioning with a fixed voxel size. Second, it provides a volumetric model in the form of a list of cubes, making it easy for computing collisions with other objects. Finally, it easily enables to control the amount of time allotted to carving by having the stop condition waiting for a timeout or an external event. In this case it is important to underline that the tree carving should be performed width first rather than depth first. If performed depth first, the octree will be very detailed in some areas, while seldom tested in others (Fig. 3). For a sequential execution, the only consequence of a width first carving is an extra memory consumption.

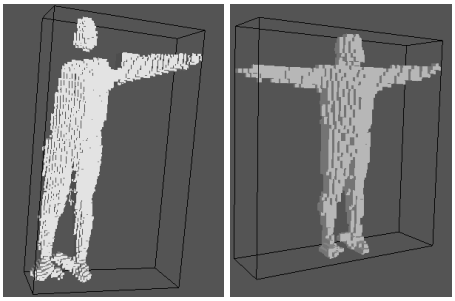


Figure 3: *A depth first(left) and width first(right) octree carving with the same elapsed time. The former leads to an unbalanced space carving, the latter to a balanced one.*

3. Work Stealing

We recall the principle of work stealing, the main associated results and analyze why work stealing is well adapted for octree carving.

Work stealing is a classical approach for dynamic load balancing. It has been used for various computations, including parallel graphics [HA98, CD99]. It extends the Graham list scheduling principle [Gra66] for programs that create tasks recursively. The principle is simple. When starting the execution, a first processor is assigned all source tasks (the initial ready tasks). At runtime, each processor maintains a local list where it stores the ready tasks it has locally created. A task becomes ready when all its predecessor tasks, i.e. the tasks it depends on, have been executed. The tasks are organized in the list according to a total sequential depth first order. When a processor P completes a task, it pops the first one t (according to depth first order) in its local ready list if non empty. Else, P is idle and becomes a thief: it randomly selects another processor until finding one victim processor V that owns ready tasks. Then it picks-up the oldest ready task t in the ready list of V. In both cases, P starts the execution of t.

Work stealing achieves a provable performance with respect to the *work* and *depth* of the parallel algorithm. The work W_1 is the total number of elementary operations performed during the execution of the algorithm. An instruction

may be a standard operation or a task creation. The depth W_∞ is the critical path in number of operations for an execution on an unbounded number of processors, i.e. the number of instructions along the longest dependency path. Let T_p be the execution time on p identical processors with execution speed Π (in number of instructions per time unit). An execution takes a time $T_1 = \frac{W_1}{p\Pi}$ on a single processor and a time $T_\infty = \frac{W_\infty}{\Pi}$ on an unbounded number of processors. On p processors, work stealing ensures that with a high probability [ABP01]:

$$T_p \leq \frac{W_1}{p\Pi} + O\left(\frac{W_\infty}{\Pi}\right) \quad (1)$$

and the number of steals is small, $O(W_\infty)$ per processor.

Thus, if the depth W_∞ is small compared to the total amount of work W_1 the parallel execution time is close to the lower bound $\frac{W_1}{p\Pi}$. This motivates the use of work stealing to schedule parallel programs having a small depth W_∞.

The octree shows properties making it very well adapted to work stealing. We consider a task the computation associated with one voxel. The dependency graph corresponds to the octree graph, as a given voxel can be computed as soon as its parent has been treated. In the worst case where no pruning occurs, a tree of depth n leads to $W_1 = \frac{8^{n+1}-1}{7}$ tasks, while the critical path is $W_\infty = O(n)$. Thus, having $W_1 \gg W_\infty$, work stealing should lead to optimal parallel executions. Even when pruning occurs, the ratio is usually very favorable to work stealing. For instance, our test data set, consisting of a full human body (Fig. 4), has $W_1 = 162799$ voxels when going up to depth level $W_\infty = 8$. The following properties also contribute to keep the overhead of work stealing small:

- All tasks perform the same computation. The only difference between two tasks is the coordinates and size of the considered voxel. Thus stealing work only consists in stealing a list of a voxel coordinates and size. It leads to light memory transfers.
- Task dependencies are simple: a task only depends on its parent. When a processor splits a voxel, it can directly add its 8 children voxels to its ready list. There is no other dependency to be solved that could require to wait for another processor.

Close to the octree root, the amount of parallelism is reduced (1 voxel at the root, 8 at depth 1, 64 at depth 2), and the cost of projecting a voxel into the silhouettes is higher (proportional to the voxel size). This can impair the performance of work stealing. To soften this effect, usually octree carving starts at a higher depth level (level 2 for our tests).

4. Adaptive Octree Algorithm

The goal of this work stealing algorithm is to dynamically schedule the work load to better use the processors available. We assume the target parallel machine supports a global address space with a shared (or virtually shared) memory access and manages data locality.

Each voxel is represented as a quadruple of its coordinates and level (i, j, k, d). To manage the workload the voxels are organized in ready lists. Each ready list consists of a vector of voxels and pointers to the first, last and current voxel. We call a task, the computations required to test a voxel status.

4.1. Initialization

The algorithm starts at depth level n with 8^n initial voxels. The number n is usually the smallest number to have more initial voxels than processors. These voxels are split into p ready lists, where p is the number of processors. The goal is to avoid the performance bottleneck of the first depth levels that do not provide enough parallelism.

The ready lists are organized in a singly-circularly-linked list. There is one list of ready lists per depth level, called a level list. All level lists, except the ones from the starting depth level, are initially empty. Each processor is assigned a first ready list from the starting level.

One processor has a manager role. It takes care of initializing the first ready lists and signal all other processors when computations must be stopped (because the timeout occurred).

4.2. Octree Level Computation Based on Work Stealing

Each processor tests the voxels of its ready list. If a voxel needs to be split, its 8 child are inserted into the ready list of the next level.

Each processor cycles twice through the level list. A processor scans the level list from a randomly chosen position, looking for a free ready list in the level list until it completes one cycle. If it finds one, it takes it. Else, since there is no more ready list to grab, it performs a second loop, but this time the processor becomes a thief. It traverses the ready lists trying to get part of the remaining voxels. For a target ready list, the thief processor locks the current working pointer of the victim processor (the owner of the list). It grabs half of the remaining voxels, leaving a minimum number of voxels (fixed by a threshold) to avoid voxels to be stolen back and forth. The thief creates a new ready list containing these voxels. This operation just involves pointer settings and does not lead to voxel copies. The working pointer of the victim is unlocked as soon as it can safely restart processing its ready list.

Finally, when a processor ends its second cycle through the current level list, it starts working on the next level, processing the voxels of its ready list if not empty.

4.3. Overlapping Level Synchronization

Let $L_i(t)$ be the level of the voxel being computed by processor i at time t, and $\sigma = \max_t \max_{i \neq j} |L_i(t) - L_j(t)|$ be the maximal level synchronization, i.e. the largest distance in term of levels between two processors. To enforce a (relaxed) width first like octree carving it is required that σ is always kept small. This is achieved by introducing extra synchronizations. Setting a global barrier after each level guarantees a synchronization $\sigma = 0$ at any time, but it prevents level overlapping and then restricts parallelism. A synchronization $\sigma \leq 1$ enables to overlap synchronization overhead while limiting the octree unbalance. To ensure $\sigma \leq 1$, we implemented it as follows. To each level d corresponds a shared counter $C[d]$ initialized to the number of processors p. When a processor completes all its ready tasks at level d and if $C[d-1] = 0$, it starts stealing. If it does not succeed to get voxels from other processors, then it decreases $C[d]$ by 1 and starts the computation of its local voxels at level $d+1$. Once completed, it waits until $C[d-1] = 0$ before starting new steal requests.

4.4. Time Control

A time control routine was integrated in the algorithm to bound the time spent to carve the octree. The main objective is to enforce a real-time behaviour for the final application, i.e. a minimum latency and maximum refresh rate.

The manager processor is in charge of checking the time elapsed and signal other processors that the time is over.

One difficulty is to define the time check frequency, to limit the overhead while being frequent enough to enable a good time control. A good choice is to have the manager control the time elapsed each time it completes a ready list. Because synchronizations are present into the code to enforce a width first tree traversal, it enables to stop the algorithm at any time keeping a well balanced space carving.

5. Provable Performance of the Adaptive Octree Algorithm

This section deals with the proof of the performance of the adaptive octree algorithm. It is analyzed with respect to the reference sequential algorithm.

For the sake of simplicity, we consider that the adaptive algorithm implements a maximal level synchronization $\sigma = 0$, with a synchronization barrier after each octree level. This prevents level overlapping: level d is computed only when level $d-1$ is completed, like in the sequential computation. Then parallelism occurs only at each level. Furthermore, we will consider that the sequential computation starts with the same voxels as the parallel algorithm. So if we consider p processors, we consider the sequential computation to be initialized at depth $\lceil \log_8 p \rceil$ with $8^{\lceil \log_8 p \rceil}$ voxels.

Under those assumptions, when there is no time limit, the following theorem states that the adaptive algorithm is almost p times faster than the sequential one if the depth of the octree is small w.r.t. its number of voxels.

Theorem Let T_s be the time of the reference sequential algorithm to compute an octree with n nodes and depth d on a processor with speed Π. The adaptive algorithm running on p identical processors with speed Π computes the same octree in time:

$$T_p =_{n \to \infty} \frac{T_s}{p} + O\left(\frac{d \log n}{\Pi}\right) \tag{2}$$

Proof. The state of each voxel being deterministically computed, both algorithms compute the same octree. Let $T_s(i)$ (resp. $T_p(i)$) be the time of the reference sequential algorithm (resp. adaptive algorithm) to compute level i, $1 \leq d \leq i$, and n_i be its number of nodes. At each successful steal, a processor steals half the ready voxels of a non idle processor. Then, on an infinite number of processors, the parallel algorithm has critical depth $W_\infty = \log n_i$. The operations performed by the parallel algorithm are either voxels computations, i.e. $T_s(i).\Pi$ unit operations, or overhead instructions to manage parallelism (locks, steals, list management). Due to work stealing, the number of steal requests is $O(W_\infty)$ per processor, i.e. $O(\log n_i)$. Except steals, the only other overhead operations are when a processor access its own local ready list to extract voxels. This requires a lock to avoid contention with possible thieves. However, if there are v voxels in the ready list, then $\log v$ voxels are extracted at the price of only one lock. Then, following [RTB06], the number of lock operations is $O\left(\frac{n_i}{\log n_i}\right)$. Then, from the work stealing fundamental theorem (Section 1), we have $T_p(i) \leq \frac{T_s(i)}{p} + \frac{1}{\Pi} O\left(\frac{n_i}{p.\log n_i} + \log n_i\right) =_{n_i \to \infty} \frac{T_s(i)}{p}$. Summing for all the levels concludes the proof. △

However, due to real time interactive constraints, the depth of the octree is truncated at a given unknown time limit. The computation time of the algorithm is fixed and the objective is to maximize the level of details, i.e. the number of voxels computed. Indeed, taking benefit of parallelism, the adaptive octree algorithm not only computes faster but also more details. The next theorem states that in a fixed time t, the adaptive algorithm on p processors computes almost the same precision as the reference sequential algorithm in a time $p.t$.

Theorem

Let n_p be the number of voxels computed by the adaptive algorithm in a time limit t on p identical processors. Let n_s be the number of voxels computed by the sequential reference algorithm in a time limit $p.t$ on 1 processor. Let d_p (resp. d_s) be the last fully completed level of the adaptive (resp. sequential) algorithm.

Then:

$$n_p = n_s - O(\log n_s) \text{ and } |d_p - d_s| \leq 1 \quad (3)$$

Proof. The proof is also based on the work stealing theorem, applied to each level. Let d be the maximal level of a voxel computed by the adaptive algorithm. Since this algorithm performs a barrier after each level, clearly $d \leq d_p + 1$. From previous theorem, in a time t, the adaptive algorithm performs $W_p = p.t.\Pi$ operations among which at most $O\left(d \log n_p + \frac{n_p}{\log n_p}\right)$ are overhead instructions with respect to voxels computation. Then, if $T_s(n_p)$ denotes the sequential time to compute the corresponding voxels, $p.t.\Pi = T_s(n_p) + O\left(d \log n_p + \frac{n_p}{\log n_p}\right)$. Then, asymptotically for n_p large enough w.r.t. d, $p.t.\pi \simeq T_s(n_p)$. Moreover, all those n_p nodes are at most on two levels, d_p and $d_p + 1$. Then, since $p.t.\Pi = T_s(n_s)$, $n_p \simeq n_s$ and, due to barrier, $d_s = d_p$ or $d_s = d_p + 1$. △

Asymptotically for a large number of nodes w.r.t. the depth of the octree, both theorems generalize to the practical case where $\sigma = 1$ (then at most two levels may differ between the sequential and the adaptive algorithm).

6. Implementation

The algorithm was implemented in C++ using Posix Threads. As stated earlier, we target parallel computers supporting a global address space with shared (or virtually shared) memory. In this context Posix threads provide a well adapted programming environment.

For a better performance the use of mutex like semaphores was eliminated. Instead assembly atomic operations like compare_and_swap "cmpxchg" and atomic_add_return "xadd" combined with the LOCK prefix were used. These atomic operations are supported by most modern CPUs. We noticed a performance increase of about 20% compared to a mutex based implementation.

The "yield" instruction("sched_yield" systems call) was used to better manage waiting times. It improves the performance in the waiting loops informing the kernel to schedule other processes.

The application is launched with one thread per processor. The first thread is the manager. We consider we are the only users of the computer and no other application is running. To prevent the migration of threads during execution, which would impact performance, each thread was locked on a given processor. For that purpose, we used the "pthread_setaffinity_np" instruction. This technique also improves the frame memory control, avoiding cache misses and sparse memory allocations.

To better balance the work load into the initial ready lists, the voxels are distributed in a round-robin fashion. The goal is to give each working list voxels from different space regions.

To reduce contention, a thread does not wait to steal from a locked ready list. If one thief fails to lock a working list, it does not try a second time. It just steps to the next ready list in the chain. It avoids waiting for a lock release.

We relied on the GCC compiler to make an efficient use of SIMD parallel instructions available on the processor. A careful manual code optimization could probably further improve performance.

7. Results

The computer used for the tests has 8 dual Core AMD™2.2GHz Opteron processors, 32 GB of memory, and is running Linux kernel 2.6.17. This is a CC-NUMA (Cache coherency Non Uniform Memory Access) architecture with a virtually shared memory using the HyperTransport™inter-processors communication layer.

7.1. Off-line Cameras

Tests were first performed with two off line series of images. The first one is a sequence of a full body person filmed with 8 cameras, called the Ben benchmark, freely available at `https://charibdis.inrialpes.fr/`. Each camera image has a resolution of 780x582 pixels. The second benchmark, called Al Capone, is a synthetic 3D model, from which we computed 64 images (resolution of 300x300 pixels) from different view points. This model enables to test our algorithm with a very large number of silhouettes. Though today marker-less motion capture environments have usually less than 64 cameras, the trend is to increase this number as it improves the quality of the obtained 3D model. Notice that both image sets fit in the 1MB L2 cache available per core. Ben requires 456KB and Al Capone 713KB.

We first compared a pure sequential implementation of the octree carving algorithm with our parallel code launched with only one thread. The overhead due to the extra code introduced into the algorithm for work stealing is small. It is about 4% for the Ben model and below 1.3% for Al Capone.

We ran the algorithm for both benchmarks with varying numbers of CPUs, without time limit but with various max depth levels. All results are averages over 100 runs. The execution times include the time to load the images from the local disk. We plot (Fig. 6 and Fig. 7) the execution times (logarithmic scale on the y-axis) and the speed up ($s = \frac{T_1}{T_n}$). The gain of using 16 CPUs is very significant with an efficient use of the resources (high speed-ups). For instance Ben at max depth level 8 is computed on 1 processor in about 234.2 ms. The same model takes only 16.82 ms on 16 processors.

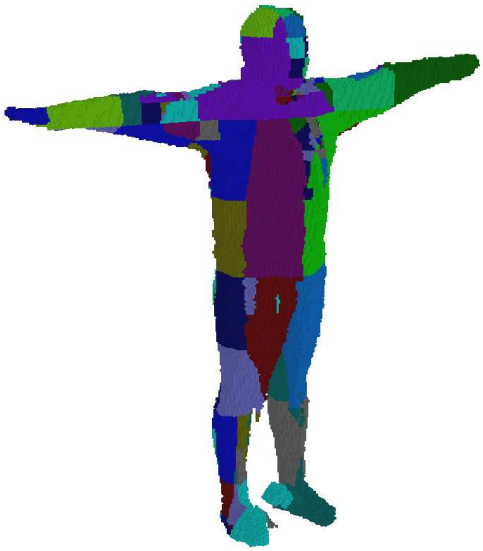


Figure 4: *Ben. Max depth level set to 8.*

Figure 5: *Al Capone. Depth level set to 5 (left) and 7 (right).*

At max depth level 7, the Al Capone goes from about 441.1 ms with 1 CPU to about 31.15 ms with 16 CPUs. Notice that the reconstruction at level 5 does not scale well, since at this low level the execution time is dominated by the image loading – sequential – step. As the amount of parallelism increases while going deeper in the tree, the speed-up increases with the max depth level. Al Capone was also tested with work stealing turned off (Fig. 7). The performance is significantly affected. A static load balancing is inefficient as the shape of the octree, and thus the work load, cannot be predicted.

The number of steals is low in comparison to the amount of cells as predicted by the theory. Table 1 presents the average of all voxels computed against the number of full voxels, and the relative number of steals. About 60% of steal attempts are successful. A side effect from this low number of steals is the good space locality of voxel distribution. By associating a color per CPU, we notice that large contiguous area are processed by the same thread (Fig. 4 and Fig. 5).

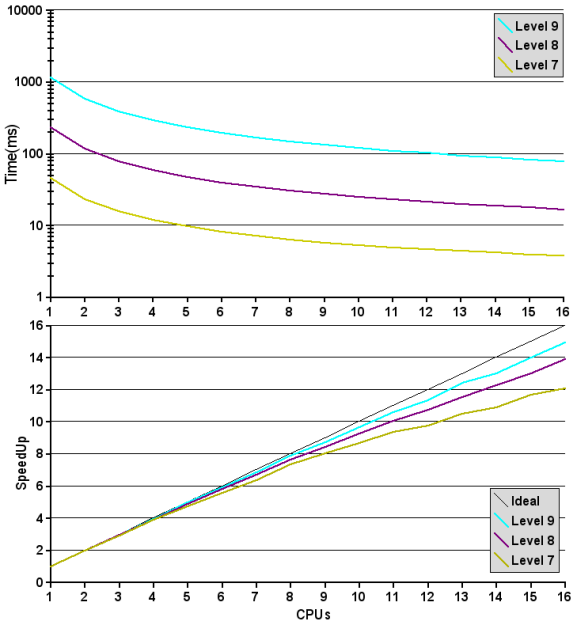


Figure 6: *Execution time and speed up for Ben.*

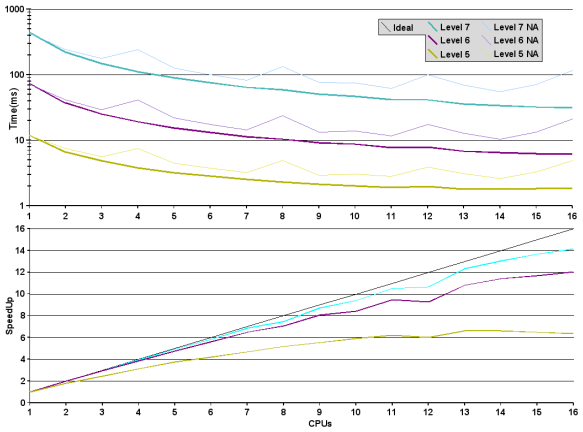


Figure 7: *Execution time and speed up for Al Capone, with work stealing enabled or disabled (NA for Non-Adaptive)*

We tested the time control routine with Ben (Fig. 8) and a 30 ms deadline. The simulation is set to go up to depth level 8. With just one processor it is not even possible to complete level 5, making the model unrecognizable. The execution time is significantly larger than 30 ms, because the processor does not check the elapsed time before it completes the first ready list. Up to 8 processors, the time control is effective: the execution stops before all voxels of level 8 are computed. Notice that the measured execution time is usually slightly higher than 30 ms because after the timeout occurs all processors apply a fast test algorithm to guess

		Voxels		Steals/threads	
Dataset	Level	Computed	Full	Tries	Success
Ben	8	162799	67398	42.59	25.26
Al Capone	7	44840	34601	26.21	16.18

Table 1: *For each data set computed up to a given max level, the number of voxels computed is given with the number of voxels identified as full, the number of steal attempts and successful steals per thread.*

if each pending voxel is full or empty. With 8 processors, the 30 ms limit enables to reach the max depth level. Next, as the number of processors further increase, the extra computing resources available enable to decrease the execution time, ending below 20 ms (the number of voxels computed at level 8 stops to increase).

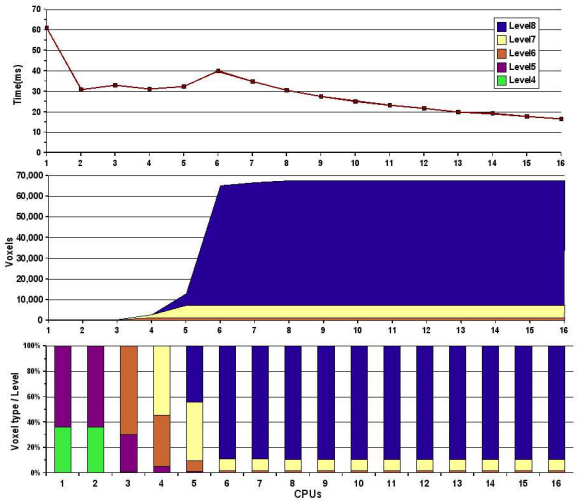


Figure 8: *Ben modeling with a 30 ms time limit. The graph plots the total execution time, the middle graph plots the amount of voxels produced per level, and the lower graph the percentage of voxels types.*

7.2. On-line Cameras

We tested the algorithm in a live environment with 5 FireWire cameras (image resolution 780x580) filming a person in real time. Cameras are genlocked through a specific network. Each one is connected to one computer (dual xeon), processing the incoming video stream to remove the background and compute the silhouette images. Then, the silhouettes are forwarded to the 16 cores computer. It computes the octree and sends the list of full voxels to 16 dual Opteron computers powering a 16 projectors high resolution display wall. These computers render the voxels. All computers are connected through a gigabit Ethernet network.

This application was developed on top of FlowVR [AR06]

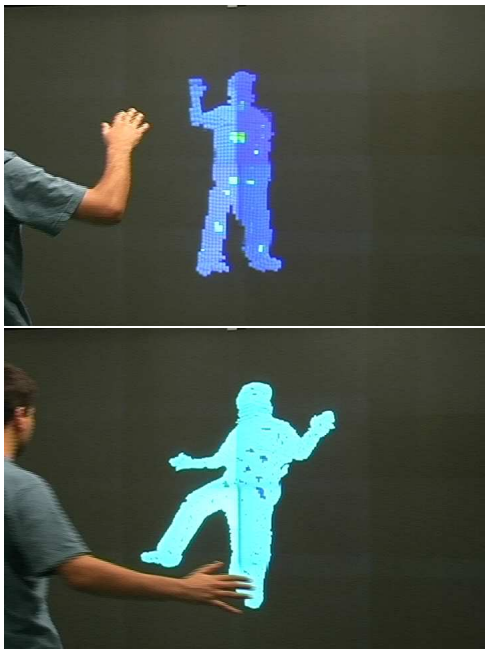


Figure 9: *Live tests with 5 on-line cameras with max depth level 6 (top) and 8 (bottom).*

for coupling and distributing the different software components. FlowVR Render [AR05] was used for the distributed rendering on the display wall.

Refer to the video associated with the article for the results. Notice the resolution of the video is lower than the display wall resolution, making it difficult to distinguish the smaller voxels while they are clearly visible on the display wall. When rendering, the voxels are colored according to their depth level. Tests were performed with and without time control, with various numbers of processors and different levels of max depths. The quality significantly increases with the max depth (Fig. 9). Fingers become visible at level 8. Some artifacts (ghost leg) are visible in some situations. This is due to the accumulation of small errors from camera calibration, background subtraction and voxel projection tests. The time control enables to keep the latency low and the frame rate stable. Some momentaneous performance drops are visible in the video. Though the cause of these drops are not yet clearly identified, it is probably related to network issues (we suspect the linux network driver).

8. Involving the GPU

The implementation was modified to use a GPU as a co-processor for one thread. The work stealing algorithm is not modified. The only difference comes from the way a GPU

processes a voxel. As stated earlier, to test a voxel, different points contained in the voxel are projected back onto the silhouettes. On a CPU, the result of each point projection is probed to detect if the status of the voxel can be defined. If so the CPU skips to the next voxel. Due to the SIMD nature of a GPU, making so many probing tests is highly inefficient. To bypass this limitation, the CPU provides to the GPU a list of points to project back onto the silhouettes. The GPU performs all these projections and the CPU gets back the results to define the status of the voxels. The GPU becomes faster than the CPU (compared to the case where the CPU performs all the projections) only if the number of points to test is large enough to hide the overhead of transferring the data back and forth between the CPU and the GPU. So the use of the GPU is triggered only if the number of tests to perform reaches a certain threshold. To reach that threshold several voxels can be tested at once if available in the ready list.

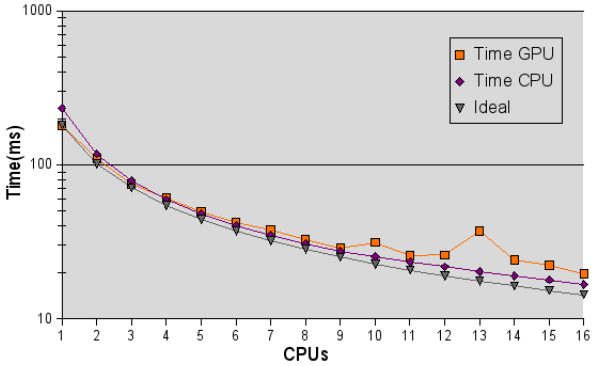


Figure 10: *GPGPU x Pure CPU.*

Experiments were performed on the 16 core machine equipped with one Nvidia Geforce 7900 graphics card (Fig. 10). Involving the GPU instead of relying only on the CPU increases the performance by 30% from 234.20ms to 180.17ms. Using these numbers as the reference sequential execution times ($T_1^{GPU} = 180.17$ms and $T_1^{CPU} = 234.20$ms), we can compute a lower bound for the execution time when $p-1$ CPUs and one CPU/GPU couple are involved in the computation:

$$T_{ideal}(p) = \frac{1}{\frac{(p-1)}{T_1^{CPU}} + \frac{1}{T_1^{GPU}}} \quad (4)$$

Experimental results shows that our implementation usually fails to stick to this ideal case, often a pure CPU based execution being more efficient. The CPU is in fact often faster than the GPU because it can bypass many projection tests while the GPU will always perform all tests even if the voxel status can be defined after just a few tests. The main interest of this early experiment is to show that a GPU can be involved in the computation without having to deeply revisit the work stealing algorithm. Future experiments will focus on involving more GPUs and improving the GPU implementation, targeting a Nvidia G80 GPU programmed with the CUDA library.

Notice that the theoretical results (Section 5) does not apply to computing units running at different speeds. However we should be able to extend our result to this case by relying on Bender and Rabint's proof of work-stealing for heterogeneous processors of different and possibly changing speeds [BR02].

9. Conclusion

This paper introduced a work stealing algorithm for a time-constrained octree carving. The algorithm enables to dynamically balance the work load while ensuring a relaxed width first octree carving, required to get a balanced octree carving when the timeout occurs.

The algorithm was validated theoretically as well as experimentally by applying it to 3D modeling. The algorithm is general enough to be applied to other problems, for instance from computer graphics where octrees are common. It can also be applied to different tree structures. Just notice that the smaller the tree arity, the smaller the ratio $\frac{W_1}{W_\infty}$.

Future work will focus on improving the GPU implementation to efficiently involve multiple CPUs as well as multiple GPUs into the computation.

10. Acknowledgements

The authors wish to thank Thomas Arcila, Everton Hermann and Florian Geffray for their help with the experiments.

This work is partly funded by ANR grant BGPR/SafeScale.

References

[ABP01] ARORA N. S., BLUMOFE R. D., PLAXTON C. G.: Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst. 34*, 2 (2001), 115–144.

[AR05] ALLARD J., RAFFIN B.: A shader-based parallel rendering framework. In *IEEE Visualization Conference* (Minneapolis, USA, October 2005).

[AR06] ALLARD J., RAFFIN B.: Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference* (Alexandria, USA, March 2006).

[BR02] BENDER M. A., RABIN M. O.: Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems Special Issue on SPAA '00 35*, 3 (2002), 289–304.

[CD99] CLYNE J., DENNIS J.: Interactive direct volume rendering of time-varying data. In *Eurographics Data Visualization '99 Conference* (1999), pp. 109–120.

[Gra66] GRAHAM R. L.: Bound for certain multiprocessing anomalies. *Bell System Tech. J.* (1966), 1563–1581.

[GWN*03] GROSS M., WUERMLIN S., NAEF M., LAMBORAY E., SPAGNO C., KUNZ A., KOLLER-MEIER E., SVOBODA T., GOOL L. V., S. LANG K. S., MOERE A. V., STAADT O.: Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 03* (San Diego, 2003).

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1–2 (1998), 57–68.

[HHD99] HORPRASERT T., HARWOOD D., DAVIS L. S.: A Statistical Approach for Real-time Robust Background Subtraction and Shadow Detection . In *IEEE ICCV'99 frame-rate workshop* (1999).

[KGYS05] KARAMAN M., GOLDMANN L., YU D., SIKORA T.: Comparison of static background segmentation methods. In *Visual Communications and Image Processing (VCIP '05)* (Beijing, China, July 2005).

[MP04] MATUSIK W., PFISTER H.: 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. In *Proceedings of ACM SIGGRAPH 04* (2004).

[RTB06] ROCH J.-L., TRAORE D., BERNARD J.: On-line adaptive parallel prefix computation. In *EUROPAR'2006* (Dresden, Germany, August 2006), Springer-Verlag L. ., (Ed.), pp. 843–850.

[Sze93] SZELISKI R.: Rapid Octree Construction from Image Sequences. *Computer Vision, Graphics and Image Processing 58*, 1 (1993), 23–32.