I

Etat de l'art

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

Logiciels pour la réalité virtuelle 2

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

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

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

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

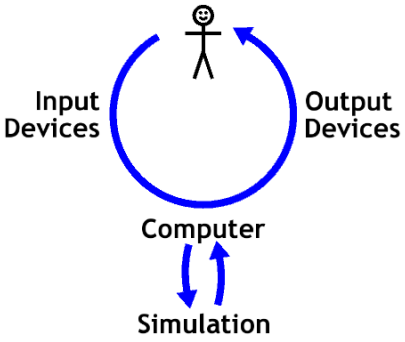


Figure 2.1 *Boucle principale d'une application interactive.*

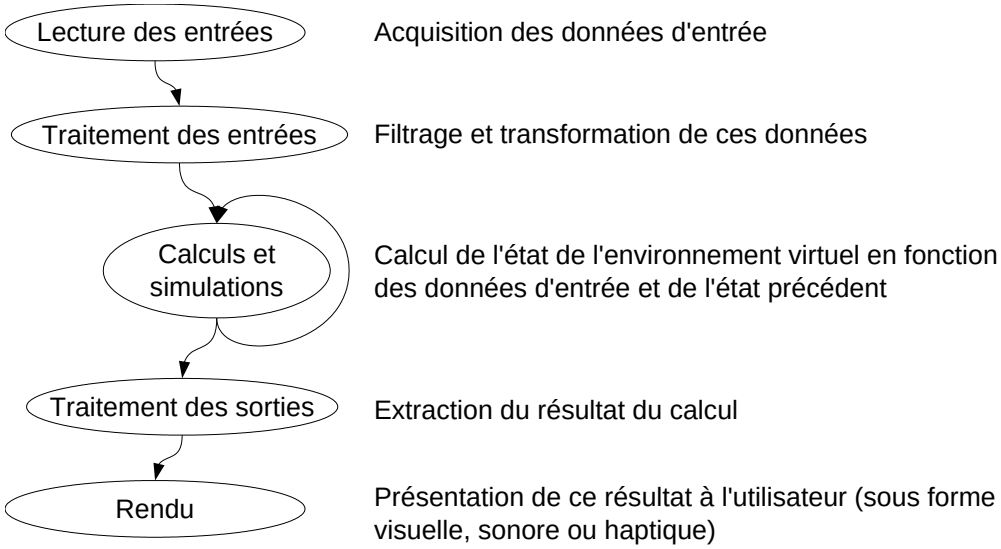


Figure 2.2 *Structure d'une application de réalité virtuelle.*

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

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

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

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

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

Figure 2.3 *Périphériques classiques de capture de mouvement.*

2.1 Lecture des entrées

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

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

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

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

2.2 Traitement des entrées

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

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

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

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

2.3 Calculs et simulations

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

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

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

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

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

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

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

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

2.3.1 Systèmes multi-agents

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

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

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

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

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

2.3.2 Simulations interactives

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

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

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

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

2.3.3 Simulations externes

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

2.4 Traitement des sorties

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

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

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

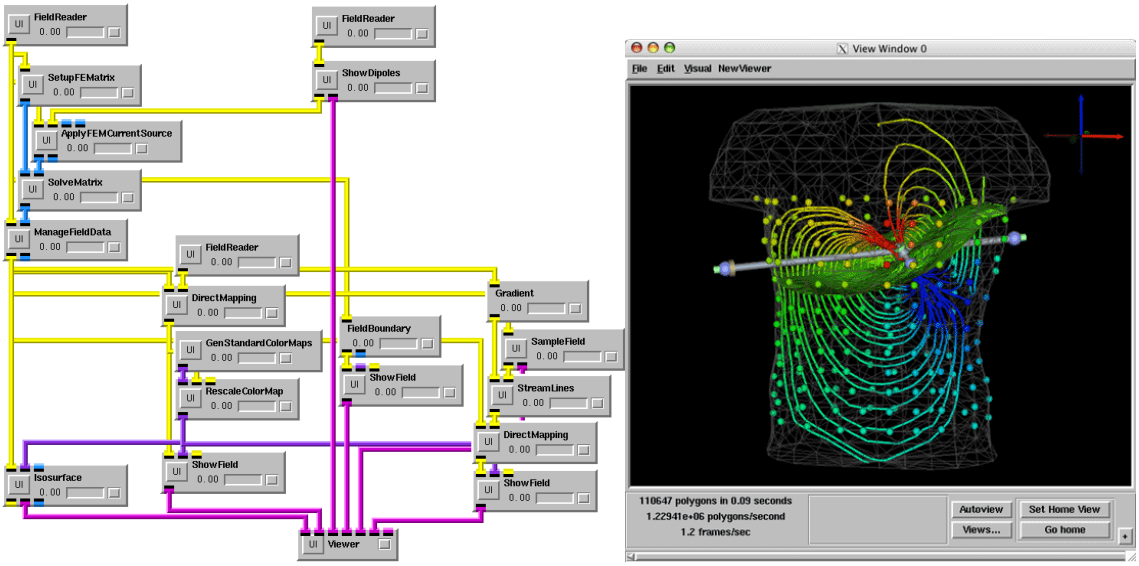


Figure 2.4 *Application utilisant l'environnement de visualisation interactive SCIRun [139].*

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

2.5 Rendu

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

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

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

Figure 2.5 *Systèmes classiques d'affichage.*

(a) (b)

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

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

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

2.6 Bilan

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

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

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

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

Communications et couplage de codes 3

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

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

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

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

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

des machines d'architectures différentes dans plusieurs lieux distants.

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

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

3.1 API de communication

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

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

3.1.1 Passage de messages

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

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

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

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

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

3.1.2 Appel de procédure à distance

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

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

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

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

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

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

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

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

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

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

3.1.4 PadicoTM : Communications multi-protocoles

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

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

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

3.2 Couplage de codes

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

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

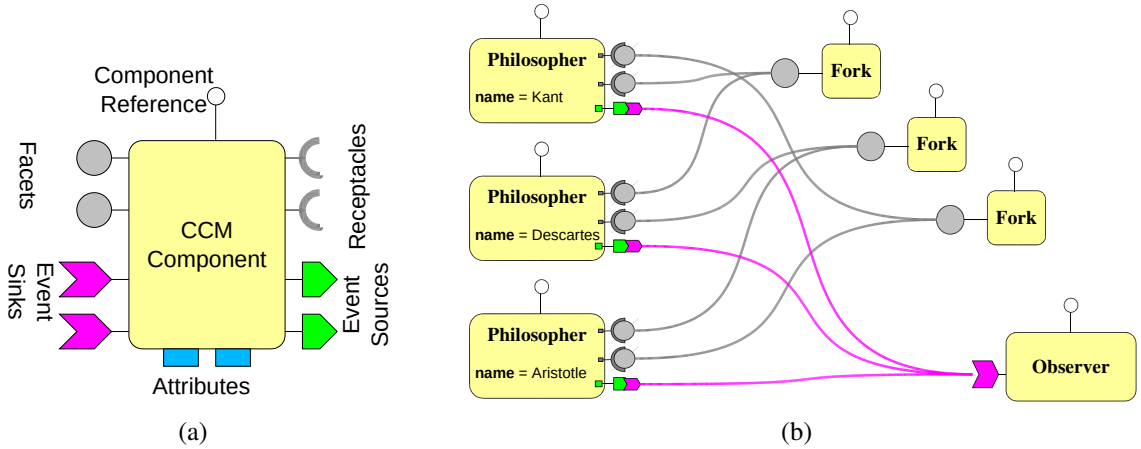


Figure 3.1 *(a) Modèle d'un composant CCM. Les ports de gauche constituent les ports de réceptions d'appels ou d'évènements, alors qu'à droite se trouvent les ports complémentaires d'envoi. (b) Exemple d'application CCM pour le problème du dîner des philosophes [135].*

3.2.1 Composants logiciels

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

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

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

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

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

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

Event sinks : port de réception d'évènements.

Attributes : paramètres ou états du composant.

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

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

3.2.2 Couplage de codes parallèles

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

3.2.2.1 Adaptation de modèles existants

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

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

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

3.2.2.2 Conception de nouveaux modèles

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

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

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

composants distribués mais non parallèles.

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

3.3 Bilan

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

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

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

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

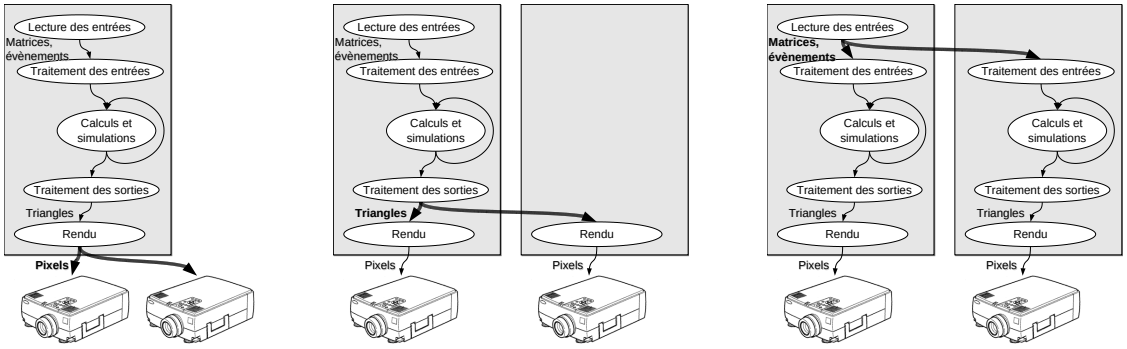
La construction d'applications interactives nécessite un support des différentes fréquences de fonctionnement des modules, liées aux périphériques externes et aux besoins

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

Parallélisation pour la réalité virtuelle 4

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

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



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

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

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

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

4.1 Rendu multi-projecteurs

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

4.1.1 Sort-last

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

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

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

4.1.2 Sort-first

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

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

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

4.1.3 Graphes de scènes distribués

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

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

4.1.4 Duplication

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

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

4.2 Cadres applicatifs de réalité virtuelle

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

4.2.1 CAVELib

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

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

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

4.2.2 VR Juggler

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

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

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

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

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

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

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

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

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

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

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

4.2.3 Syzygy

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

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

4.2.4 OpenMASK

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

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

4.3 Bilan

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

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

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