IV

Couplage de simulations distribuées interactives

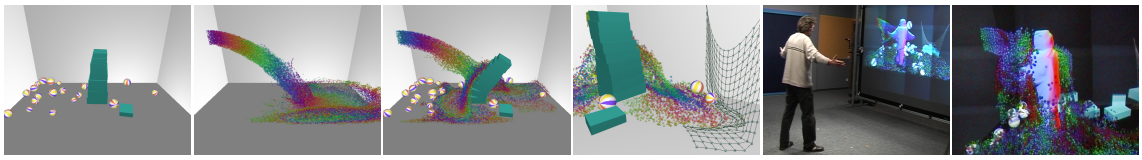
L'aboutissement des travaux sur le couplage de codes parallèles interactifs (partie II) et la modularisation de la visualisation (partie III) permettent d'envisager de nouvelles applications. C'est l'objet de cette dernière partie qui présente une expérimentation en ce sens. Bien que ce travail soit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

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

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

Introduction 15

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



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

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

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

15.1 Simulations physiques

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

15.1.1 Simulation pour les applications graphiques interactives

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

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

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

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

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

15.1.2 Simulations parallèles et distribuées

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

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

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

Architecture de couplage 16

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

16.1 Conception générale

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

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

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

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

16.2 Objets et animators

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

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

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

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

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

16.3 Interactors et filtrage des données

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

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

16.4 Protocole de communication

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

16.4.1 Objects

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

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

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

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

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

16.4.2 Evènements

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

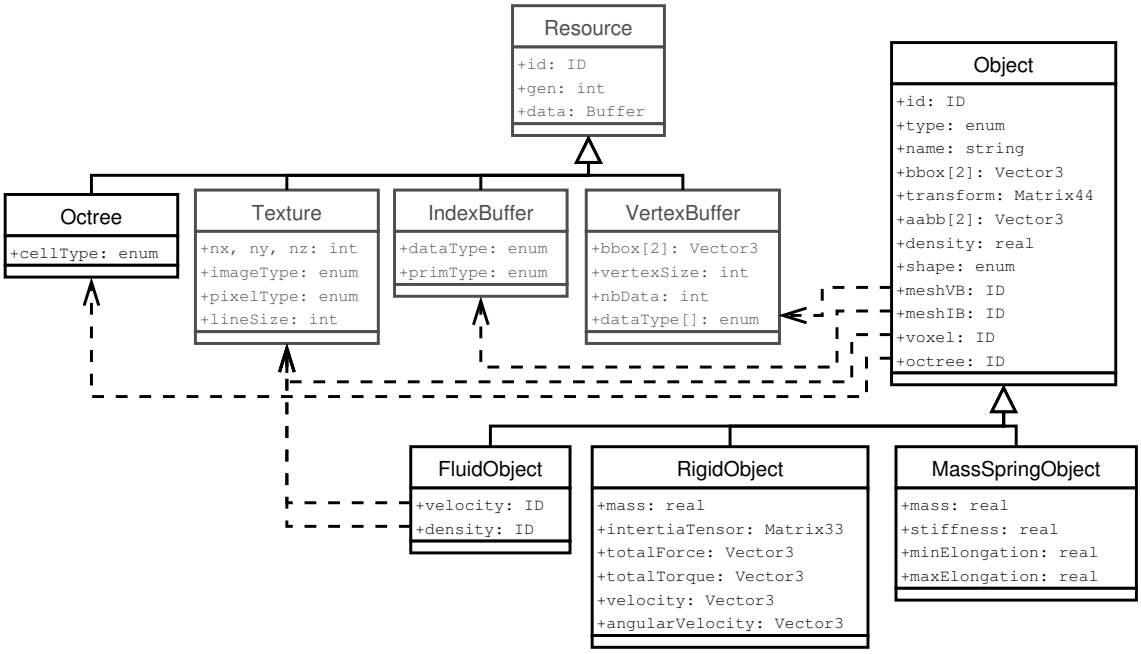


Figure 16.1 *Schéma UML des objets utilisés dans la description de la scène. Les classes grisées sont reprisent de FlowVR Render.*

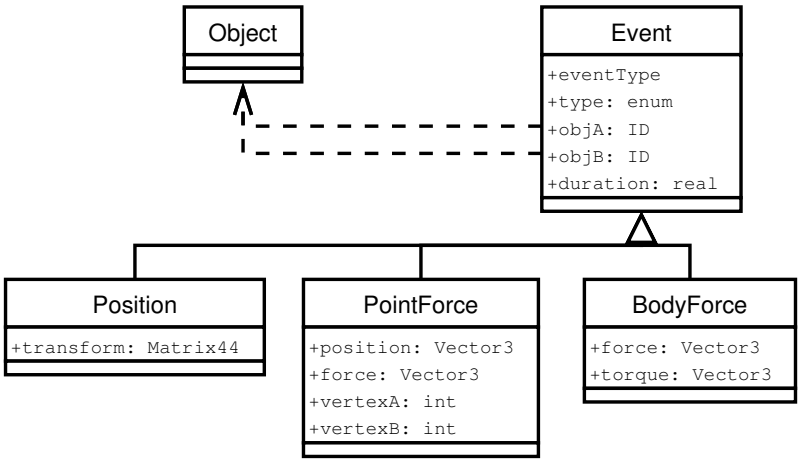


Figure 16.2 *Schéma UML des classes d'évènements.*

16.5 Construction de l'application

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

16.5.1 Exemple

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

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

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

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

16.5.2 Application séquentielle synchrone

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

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

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

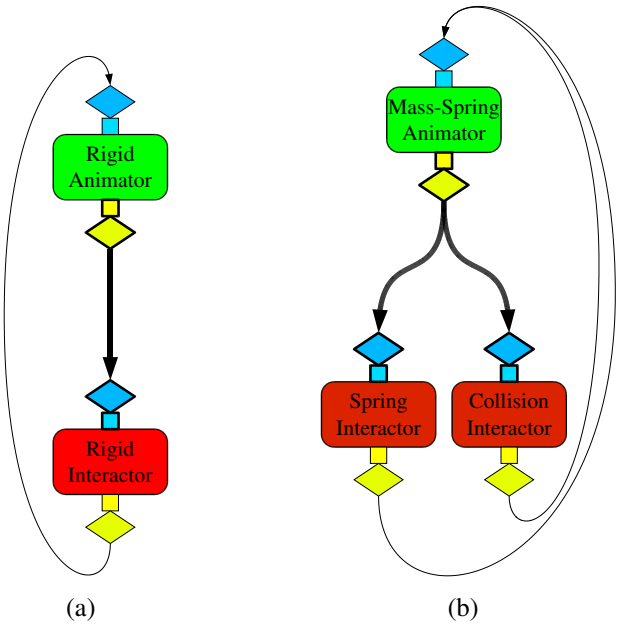


Figure 16.3 *Une simulation d'objet rigides (a) et une simulation de tissus (b). Les ports d'objets et leurs filtres associés sont en gras, les ports d'évènement en traits fins.*

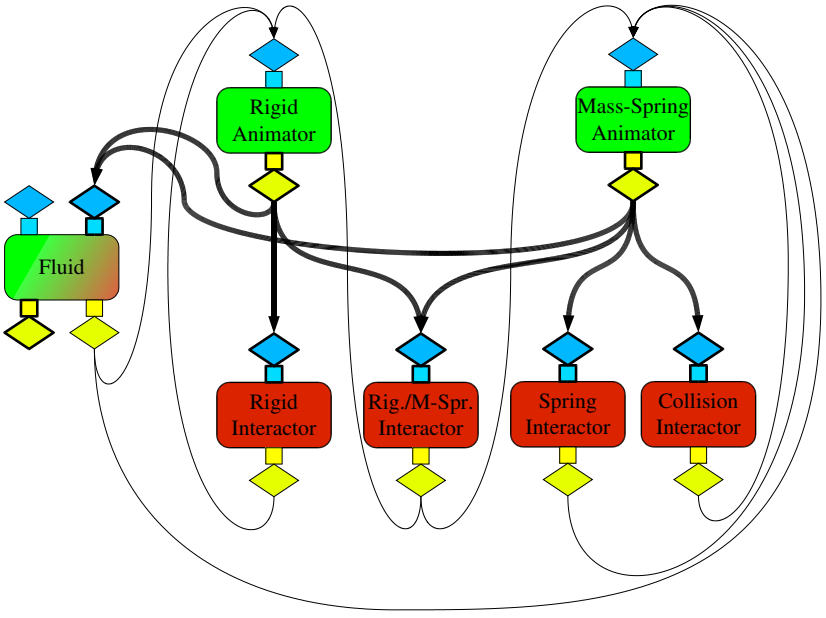


Figure 16.4 *Une application comportant des interactions bidirectionnelles entre des objets rigides, des tissus et un fluide. Le graphe de flux de données relie les composants interdépendants en supprimant les connexions inutiles (par exemple les forces issues des ressorts ne concernent que l'animator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

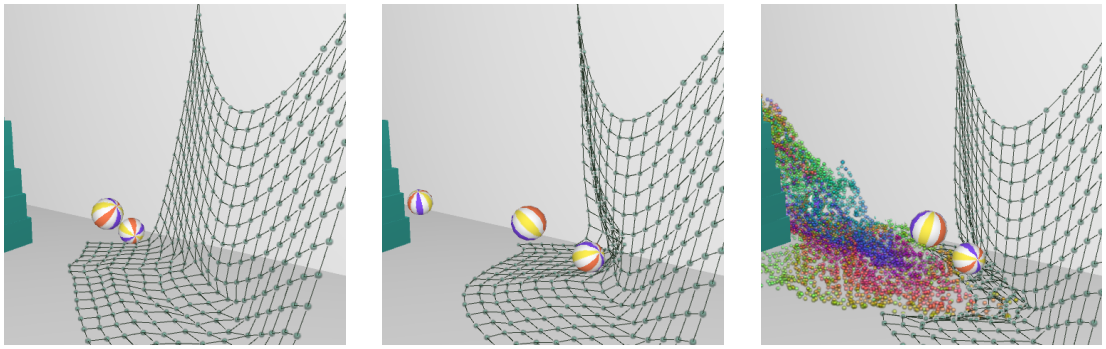


Figure 16.5 *Séquence présentant successivement les interactions d'un filet avec un objet rigide puis avec un fluide.*

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

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

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

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

16.5.3 Applications distribuées

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

16.5.3.1 Filtrage des données en sous-ensembles

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

16.5.3.2 Module parallèle

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

16.5.4 Multi-fréquences

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

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

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

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

Implantation et résultats 17

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

17.1 Implantation

17.1.1 Objets rigides

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

17.1.2 Fluide

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

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

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

17.1.3 Filet masses-ressorts

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

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

17.1.4 Interactions avec l'utilisateur

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

17.1.5 Visualisation

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

17.2 Résultats

17.2.1 Animation séquentielle hors-ligne

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

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

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

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

17.2.2 Parallélisation

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

17.2.3 Exécution interactive

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

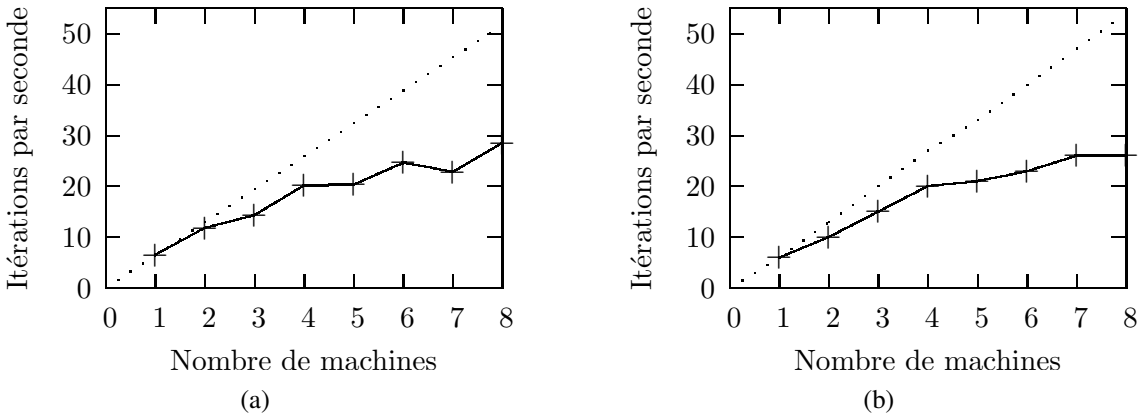


Figure 17.1 *Performance de la parallélisation du fluide en exécution synchrone non interactive avec une grille de* $32 \times 64 \times 32$. *(a) Simulation du fluide seule. (b) Fluide couplé avec les autres objets.*

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

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

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

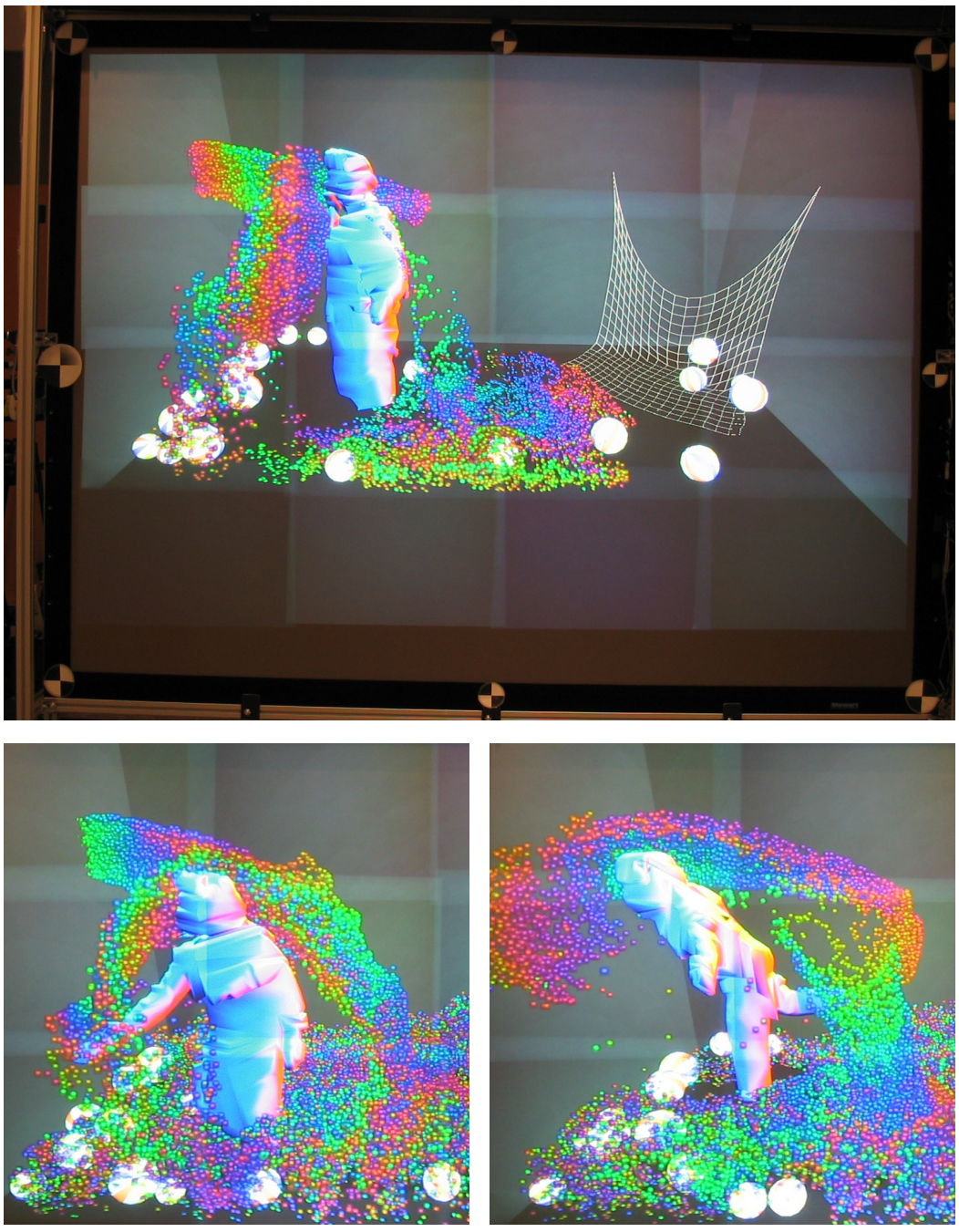


Figure 17.2 *Exécution interactive sur mur d'image avec une interaction utilisateur basée sur la reconstruction multi-caméras.*

Bilan 18

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

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

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

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

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

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