N° D'ORDRE : 2954 ANNEE : 1998

THESE

présentée devant

L'UNIVERSITE PAUL SABATIER DE TOULOUSE (SCIENCES)

en vue de l'obtention

DU GRADE DE DOCTEUR DE L'UNIVERSITE PAUL SABATIER

spécialité : INFORMATIQUE

par Patrice TORGUET

VIPER : Un modèle de calcul réparti pour la gestion d'environnements virtuels

Soutenue le 17 février 1998 devant le jury composé de :

Directeur de recherche :	R. CAUBET	Professeur à l'Université Paul Sabatier
Rapporteurs :	P. COIFFET	Directeur de Recherche CNRS au Laboratoire de Robotique de Paris
	D. THALMANN	Professeur à l'Ecole Polytechnique Fédérale de Lausanne
Examinateurs :	C. BETOURNE	Professeur à l'Université Paul Sabatier
	Y. DUTHEN	Professeur à l'Université des Sciences Sociales de Toulouse
	G. FISSE	Responsable Imagerie, CISI
	J.P. JESSEL	Maître de Conférences à l'Université Paul Sabatier

Institut de **R**echerche en **I**nformatique de **T**oulouse
Université **P**aul **S**abatier – 118, Route de Narbonne – 31062 TOULOUSE Cedex

Résumé

La réalité virtuelle se propose de changer les interfaces homme-machine. L'avènement de stations de travail en réseau dotées de très fortes capacités graphiques 3D couplées à de nouveaux périphériques de visualisation et d'interaction intuitifs et hautement expressifs permet de fournir à plusieurs utilisateurs toutes les informations sensorielles nécessaires pour les convaincre de leur présence dans un monde synthétique. De plus, la possibilité de manipuler certains aspects de ces mondes virtuels quasiment comme dans la vie réelle offre aux intervenants la possibilité d'utiliser leur expérience et leurs capacités naturelles pour travailler de façon coopérative.

Néanmoins, la conception et la réalisation d'environnements virtuels distribués sont des tâches longues et complexes à mettre en œuvre. En effet, pour développer de tels environnements, le concepteur doit être compétent en programmation d'applications réparties, d'applications graphiques ainsi qu'en programmation d'interfaces utilisateurs. De plus, les programmes répartis sont fondamentalement plus difficiles à développer et à mettre au point que des programmes séquentiels. Pour simplifier ces diverses tâches nous avons conçu et développé VIPER (VIrtuality Programming EnviRonment).

VIPER est une plate-forme générique orientée-objet permettant la gestion d'environnements virtuels multi-utilisateurs et, plus généralement, le développement d'applications de réalité virtuelle collecticielles. Ce système gère en temps réel plusieurs utilisateurs munis de périphériques spécialisés grâce à une architecture répartie. Cette architecture est générique vis à vis des applications qu'elle peut gérer et du matériel utilisé (machines et périphériques). De plus, la principale originalité de cette architecture est de proposer deux niveaux de programmation au développeur d'environnements virtuels. Le premier niveau masque totalement l'aspect réparti de l'application qui apparaît ainsi séquentielle. Le second niveau permet de choisir et/ou de redéfinir des mécanismes de répartition proposés par VIPER en optimisant la plate-forme pour une classe d'application.

Abstract

Virtual Reality plans to change computer-human interfaces. The advent of networked workstations with very strong 3D graphic capacities coupled to new visualisation devices and intuitive interaction ones enables to provide several users all the necessary sensory information to make them believe that they are really present in a synthetic world. Moreover, the possibility to manipulate some aspects of these worlds nearly as in real life offers the participants the opportunity to use their experience and their natural capacities to work co-operatively.

Nevertheless, developing distributed virtual environments is a complex time-consuming task. In order to develop such environments the programmer has to be proficient in network, graphics and user interface programming. Furthermore, network-based programmes are inherently more difficult to program and debug than sequential ones. In order to simplify this task, we have developed VIPER (VIrtuality Programming EnviRonment).

VIPER is a generic object-oriented platform enabling multi-user virtual environment management and, more generally, the development of collaborative Virtual Reality applications. This system manages several users in real time thanks to a distributed architecture. This architecture is generic regarding applications that can be built with it along with the equipment it supports (computers and devices). In addition, the main originality of this architecture is to propose two programming levels to the developer of distributed virtual environments. The first level totally hides the distributed aspects of the application that therefore appears sequential. The second level enables the developer to choose and/or to redefine distribution schemes proposed by VIPER. In this way, s/he may optimise the platform for his/her specific class of applications.

Remerciements

Je tiens à exprimer mes sincères remerciements à Monsieur le professeur René CAUBET qui m'a accueilli au sein de l'équipe synthèse d'images de l'IRIT, qui m'a proposé ce sujet de thèse des plus intéressants, qui a aiguillé mon travail de recherche et qui a accepté de présider le jury de thèse.

J'exprime mes vifs remerciements à Monsieur Philippe COIFFET, directeur de recherche CNRS au Laboratoire de Robotique de Paris, et à Monsieur Daniel THALMANN, professeur à l'Ecole Polytechnique Fédérale de Lausanne, pour avoir accepté d'être rapporteurs de cette thèse et pour avoir eu l'amabilité de participer au jury.

Je remercie également Messieurs Claude BETOURNE, professeur à l'Université Paul Sabatier, Yves DUTHEN, professeur à l'Université des Sciences Sociales de Toulouse, Georges FISSE, Responsable Imagerie, CISI et Jean-Pierre JESSEL, maître de conférences à l'Université Paul Sabatier, pour avoir accepté de participer à ce jury.

Je tiens à remercier les membres passés et présents de l'équipe synthèse d'images ainsi que ceux du département réalité virtuelle de CISI avec qui j'ai pris plaisir à travailler. Je tiens à remercier tout particulièrement Monsieur Roger PUJADO pour son amabilité, sa disponibilité et sa patience.

Enfin je tiens à remercier ma famille et mes amis qui m'ont supporté (dans tous les sens du terme) pendant toutes ces années et plus particulièrement Anne pour tout ce qu'elle a fait pour l'aboutissement de cette thèse.

Table des matières

RÉSUMÉ .... **I**

ABSTRACT .... **III**

REMERCIEMENTS .... **V**

TABLE DES MATIÈRES .... **VII**

TABLE DES FIGURES .... **XI**

INTRODUCTION **1**

CHAPITRE 1 : LA RÉALITÉ VIRTUELLE DISTRIBUÉE – ETAT DE L'ART **5**

1.1. TAXONOMIE DES SYSTÈMES DE RÉALITÉ VIRTUELLE DISTRIBUÉE 6

1.1.1. Généricité du système 6

1.1.2. Distribution géographique et multi-utilisateur 6

1.1.3. Modèles d'architecture de communication 8

1.1.4. Définition et gestion de mondes virtuels dynamiques : les comportements 17

1.2. PREMIERS SYSTÈMES À ARCHITECTURE CENTRALISÉE 20

1.2.1. Les Chatworlds 20

1.2.2. RB2 21

1.3. SYSTÈMES À ARCHITECTURE DISTRIBUÉE DÉDIÉS À LA SIMULATION MILITAIRE 24

1.3.1. SIMNET 24

1.3.2. DIS 29

1.3.3. NPSNET 31

1.4. SYSTÈMES GÉNÉRIQUES À ARCHITECTURE CLIENT-SERVEUR 39

1.4.1. dVS 39

1.4.2. WAVES 43

1.4.3. MASSIVE 48

1.4.4. VLNET 55

1.5. SYSTÈMES GÉNÉRIQUES À ARCHITECTURE ÉGAL À ÉGAL 61

1.5.1. DIVE 61

1.5.2. AVIARY 67

1.5.3. MASSIVE-2 71

1.6. TENTATIVES DE NORMALISATION DE LA RÉALITÉ VIRTUELLE DISTRIBUÉE 76

1.6.1. VRML2 76

1.6.2. HLA 78

1.7. CONCLUSION SUR LES SYSTÈMES DE RVD EXISTANTS 81

CHAPITRE 2 : PRÉSENTATION DU SYSTÈME VIPER **87**

2.1. STRUCTURE D'UNE APPLICATION 87

2.1.1. Structure logique de l'environnement virtuel 87

2.1.2. Mécanisme de communication inter-entités 88

2.1.3. Structure logique d'une entité ... *89*
2.1.4. Structure logique et rôle d'un capteur ... *90*
2.1.5. Structure logique et rôle d'un effecteur ... *91*
2.2. CHOIX TECHNIQUES ... 92
2.2.1. Une architecture distribuée égal à égal ... *92*
2.2.2. Une approche orientée objet ... *92*
2.2.3. Un modèle de programmation inspiré du SPMD ... *93*
2.2.4. Deux niveaux de programmation ... *93*
2.2.5. Prise en compte de l'hétérogénéité ... *94*
2.2.6. Adhésion au standard VRML ... *94*
2.3. LE MODÈLE EN COUCHE ... 94
2.4. SPÉCIFICATION DE L'ENVIRONNEMENT VIRTUEL ... 96
2.4.1. Les entités ... *98*
2.4.2. Les composants comportementaux ... *101*
2.4.3. L'état interne d'une entité ... *106*
2.4.4. Les stimuli et les différents types d'interactions ... *106*
2.4.5. Les capteurs et les effecteurs ... *111*
2.4.6. Le cache de composants, états internes, capteurs, effecteurs et stimuli ... *116*
2.4.7. Exemple d'entité : un avatar gérant un char d'assaut ... *116*
2.5. DISCUSSION ... 118
2.6. CONCLUSION ... 118
CHAPITRE 3 : LES ASPECTS RÉPARTIS DE VIPER ... 121
3.1. LA PLATE-FORME RÉPARTIE ... 121
3.1.1. Gestion d'un site de la simulation ... *122*
3.1.2. Gestion de la communication ... *122*
3.1.3. Gestion des processus ... *127*
3.2. LES OBJETS PARALLÈLES ... 128
3.2.1. Un modèle de programmation parallèle à objets ... *128*
3.2.2. Structures de données distribuées mises en œuvre ... *136*
3.3. LE SERVEUR DE CONNEXIONS ... 136
3.4. LES UNIVERS VIRTUELS DISTRIBUÉS ... 138
3.4.1. Les univers virtuels distribués passifs ... *139*
3.4.2. Les univers virtuels distribués actifs ... *139*
3.4.3. Les univers virtuels dupliqués ... *143*
3.5. LES ESPACES DE STIMULI DISTRIBUÉS ... 144
3.5.1. Les espaces de stimuli dupliqués actifs ... *145*
3.5.2. Les espaces de stimuli dupliqués passifs ... *153*
3.5.3. Les espaces de stimuli partitionnés ... *154*
3.6. CONCLUSION SUR LES ASPECTS RÉPARTIS ... 156
CHAPITRE 4 : EVALUATIONS ET APPLICATIONS ... 159

4.1. EVALUATION DU MODÈLE....159
4.1.1. Comparaison avec les autres systèmes....159
4.1.2. Originalités du modèle proposé....161
4.1.3. Mesures sur les aspects répartis du modèle....162
4.2. PREMIÈRE APPLICATION : MODÉLISATION DE FORMES....165
4.2.1. Un éditeur collecticiel de formes virtuelles....165
4.2.2. Outils virtuels et déformations de formes libres virtuelles....166
4.2.3. Définition des outils....170
4.2.4. Aspects multi-utilisateurs spécifiques à l'application....181
4.2.5. Conclusions sur cette application....182
4.3. GRS (*GENERIC RAILWAY SYSTEM*)....183
4.3.1. Modes d'interaction avec la simulation....184
4.3.2. Les comportements....185
4.4. PROVIS (PROTOTYPAGE VIRTUEL DE SYSTÈMES)....185
4.4.1. But du projet PROVIS....186
4.4.2. L'architecture de PROVIS....187
4.4.3. Les entités de PROVIS....187
4.4.4. Les périphériques spécialisés utilisés....191
4.4.5. Travailler avec PROVIS....192
4.4.6. Conclusion sur l'application....194
4.5. CONCLUSION....195
CONCLUSION ET PERSPECTIVES....195
RÉFÉRENCES BIBLIOGRAPHIQUES....201

Table des figures

Figure 1 : Différents modes de communication à travers un réseau 9
Figure 2 : Architecture client-serveur centralisée 11
Figure 3 : Architecture égal à égal distribuée point à point 13
Figure 4 : Structure d'un ORB de CORBA 16
Figure 5 : Architecture de RB2 de la société VPL Research 23
Figure 6 : Structure d'un simulateur SIMNET 25
Figure 7 : Exemple d'utilisation du *dead-reckoning* 26
Figure 8 : Le pipeline applicatif à trois processus 34
Figure 9 : Structure des processus et flot de données dans NPSNET-IV 35
Figure 10 : Zone d'intérêt d'un véhicule exprimée par un ensemble de cellules 37
Figure 11 : Les acteurs de dVS 40
Figure 12 : Architecture de dVS en réseau 42
Figure 13 : L'architecture logicielle de WAVES 44
Figure 14 : Auras, foci, nimbi et conscience pour deux objets 50
Figure 15 : Exemples de vues graphiques et textuelles de MASSIVE 51
Figure 16 :le *trading* spatial 54
Figure 17 : Un site serveur de VLNET avec plusieurs clients connectés 56
Figure 18 : Architecture d'un client VLNET 57
Figure 19 : Architecture logique de DIVE 62
Figure 20 : Architecture logicielle d'AVIARY 68
Figure 21 : Architecture distribuée d'AVIARY 69
Figure 22 : Les objets tiers 72
Figure 23 : L'activation des objets tiers 73
Figure 24 : Structure d'un environnement virtuel 87
Figure 25 : Mécanisme de communication inter-entités 89
Figure 26 : Schéma structurel d'une entité 90
Figure 27 : Structure logique d'un capteur 91
Figure 28 : Structure logique d'un effecteur 91
Figure 29 : Architecture en couches de VIPER 95
Figure 30 : Modèle comportemental 96
Figure 31 : Ensemble de classes qui gère la définition des comportements 97
Figure 32 : Extrait de l'interface de la classe Entité 98
Figure 33 : Extrait de l'interface de la classe ComposantComportemental 102
Figure 34 : Exemple de graphe de classe mixte C++/ObjectTcl 103
Figure 35 : Définition dans le langage CDL de la classe ComposantComportemental 103
Figure 36 : Un composant comportemental écrit en ObjectTcl 104

Figure 37 : Un composant comportemental à lier dynamiquement 105
Figure 38 : Graphe de la scène et portions privées de deux formes 107
Figure 39 : Equations d'extrapolation de formes 109
Figure 40 : Les classes qui encapsulent la communication point à point............ 123
Figure 41 : Classes abstraites qui encapsulent la communication par groupe 125
Figure 42 : Classes concrètes qui encapsulent la communication par groupe de RMP 126
Figure 43 : Evitement de l'interblocage grâce aux serveurs locaux 130
Figure 44 : Graphe de classes gérant un objet parallèle 131
Figure 45 : Résumé des caractéristiques des différentes classes d'objets parallèles............ 135
Figure 46 : Cinq entités gérées par trois univers virtuels distribués 138
Figure 47 : première étape de la migration............ 141
Figure 48 : La seconde étape de la migration d'une entité 142
Figure 49 : Un espace de stimuli dupliqué 146
Figure 50 : Espace des formes............ 146
Figure 51 : Un bâtiment et la zone de perception réelle d'une entité 150
Figure 52 : Choix d'un comportement réparti grâce à une simple allocation 152
Figure 53 : Un espace d'ordres 154
Figure 54 : Visite interactive multi-utilisateurs d'un bâtiment virtuel 163
Figure 55 : Prototype d'un simulateur distribué de véhicules militaires............ 164
Figure 56 : Un exemple de treillis 168
Figure 57 : Un type d'outils 169
Figure 58 : Un coup de marteau 170
Figure 59 : Illustration de la manipulation via un outil 170
Figure 60 : Le coup de marteau............ 171
Figure 61 : Deux utilisateurs munis de marteaux 171
Figure 62 : Extrait de l'interface du composant comportemental Ordres............ 173
Figure 63 : Extrait du code traitant un ordre du type Attraper 174
Figure 64 : Algorithme du composant comportemental Collision............ 175
Figure 65 : Initialisation de l'état interne de l'entité Marteau 176
Figure 66 : Empaquetage de l'entité Marteau............ 176
Figure 67 : Classe Martèlement............ 177
Figure 68 : Traitement des ordres reçus par un objet déformable 178
Figure 69 : La classe FormeFFD 179
Figure 70 : Déformation par crochet 180
Figure 71 : Un autre outil de déformation : le crochet............ 181
Figure 72 : Un verrou posé sur un objet 181
Figure 73 : Sessions de modélisation de paysage multi-utilisateurs 182
Figure 74 : Exemple de terrain modélisé............ 182
Figure 75 : La « Gare du Nord » 183
Figure 76 : Intérieur du TGV à deux niveaux............ 184

Figure 77 : Une foule dans la « Gare du Nord »..184
Figure 78 : Une vue simplifiée des composants comportementaux du constructeur 188
Figure 79 : Construction d'une commande multi-utilisateur 189
Figure 80 : Construction d'un satellite avec un utilisateur distant..190
Figure 81 : (a) Une IHM 2D avec les *widgets* du constructeur (b) l'IHM contient maintenant les *widgets* d'un composant de satellite...191
Figure 82 : Les périphériques choisis...192
Figure 83 : Sélecteur de liaison..193
Figure 84 : Manipulation directe d'un faisceau semi-rigide (20 articulations) à l'intérieur de la plate-forme d'un satellite 193
Figure 85 : (a) Définition des paramètres de la tâche, (b) La solution calculée 194

Introduction

La réalité virtuelle se propose de changer les interfaces homme-machine. L'avènement de stations de travail en réseau dotées de très fortes capacités graphiques 3D couplées à de nouveaux périphériques de visualisation et d'interaction intuitifs et hautement expressifs permet de fournir à plusieurs utilisateurs toutes les informations sensorielles nécessaires pour les convaincre de leur présence dans un monde synthétique. De plus, la possibilité de manipuler certains aspects de ces mondes virtuels quasiment comme dans la vie réelle offre aux intervenants la possibilité d'utiliser leur expérience et leurs capacités naturelles pour travailler de façon coopérative.

Un système de réalité virtuelle distribuée (RVD) est une architecture matérielle et logicielle qui gère un univers hybride fait de données réelles (provenant de périphériques spécialisés) ajoutées à un monde synthétique, que des applications utilisent pour simuler des expériences, avec des fonctionnalités annexes pour créer de nouvelles applications ou de nouveaux environnements. Les mondes synthétiques créés et gérés par un système de RVD peuvent être virtuellement peuplés par de nombreux utilisateurs qui interagissent avec ces mondes synthétiques à travers des périphériques spécialisés (gants de données, casques de visualisation, costumes dotés de capteurs de mouvements…). La RVD permet grâce aux nombreuses modalités de dialogue entre utilisateurs (son localisé dans l'espace, perception spatiale des autres utilisateurs et de leurs actes, dialogue gestuel…) introduites par les périphériques et les matériels spécialisés la création d'applications coopératives très réalistes et très intuitives.

Le volume de données nécessaire pour représenter un environnement virtuel est très important. En effet, la simulation d'environnements virtuels est une application temps réel qui doit réagir en un temps fixé très court (de l'ordre du $1/25^{eme}$ de seconde). De plus pendant ce court intervalle, le système de réalité virtuelle doit gérer de nombreuses opérations différentes : la gestion des périphériques spécialisés, l'application construite sur l'environnement virtuel, le rendu visuel, le rendu sonore, les interactions avec le monde virtuel... D'autre part, la base de données permettant de représenter un monde virtuel dynamique réaliste devrait contenir de un à plusieurs millions de polygones élémentaires

texturés ainsi que de nombreux comportements très complexes simulant les actions et réactions des objets virtuels dans leur environnement. De plus, les utilisateurs qui peuplent ces environnements virtuels peuvent être physiquement éloignés de plusieurs centaines de kilomètres voire même être situés sur des continents différents. Enfin, la conception et la réalisation d'environnements virtuels distribués sont des tâches longues et complexes à mettre en œuvre. En effet, pour développer de tels environnements, le concepteur doit être compétent en programmation d'applications réparties, d'applications graphiques ainsi qu'en programmation d'interfaces utilisateurs. De plus, les programmes répartis sont fondamentalement plus difficiles à développer et à mettre au point que des programmes séquentiels.

Le but de nos recherches lors de cette thèse a été de créer un modèle de calcul réparti pour résoudre plusieurs des problèmes énoncés dans le précédent paragraphe. La répartition de certaines des opérations nécessaires à la gestion d'environnements virtuels permet, en augmentant la puissance de calcul, d'accélérer globalement la simulation. De plus, il existe des processeurs spécialisés qui sont plus à même de résoudre certains types d'opérations (processeurs graphiques par exemple). Enfin, étant donné qu'un environnement virtuel distribué comporte un nombre important d'objets et d'utilisateurs qui agissent indépendamment les uns des autres, un parallélisme sur la simulation du comportement des objets et des utilisateurs s'impose (un groupe de machines s'occupant de la gestion d'un groupe d'objets et/ou d'utilisateurs).

VIPER (VIrtuality Programming EnviRonment) [Torguet 95a] est une plate-forme générique orientée-objet permettant la gestion d'environnements virtuels multi-utilisateurs et, plus généralement, le développement d'applications de réalité virtuelle collecticielles. Ce système gère en temps réel plusieurs utilisateurs munis de périphériques spécialisés grâce à une architecture répartie. Cette architecture est générique vis à vis des applications qu'elle peut gérer et du matériel utilisé (machines et périphériques). De plus, la principale originalité de cette architecture est de proposer deux niveaux de programmation au développeur d'environnements virtuels. Le premier niveau masque totalement l'aspect réparti de l'application qui apparaît ainsi séquentielle. Le second niveau permet de choisir et/ou de redéfinir des mécanismes de répartition proposés par VIPER en optimisant la plate-forme pour une classe d'application.

Dans le premier chapitre de cette thèse nous présenterons un état de l'art de la réalité virtuelle distribuée. Nous définirons, tout d'abord, un certain nombre de caractéristiques des systèmes de réalité virtuelle distribuée qui peuvent être utilisées pour les classer. Ensuite, un classement selon l'architecture distribuée utilisée servira de fil conducteur à une présentation de quelques systèmes existants choisis parmi les plus significatifs. Enfin, après une discussion sur les tentatives actuelles de standardisation de la RVD, nous conclurons cette présentation de la réalité virtuelle distribuée par des tableaux comparants les systèmes présentés et nous identifierons les limites communes à ces systèmes.

Nous détaillerons, dans le second chapitre, la structure logique qui permet de concevoir une application de réalité virtuelle avec VIPER. Ensuite, après une présentation des choix techniques que nous avons fait pour réaliser notre système de RVD, nous décrirons la structure en couche de l'architecture logicielle de VIPER et les deux niveaux de programmation évoqués plus haut. Nous présenterons enfin en détail les mécanismes qui permettent de définir les comportements des objets virtuels et leurs interactions.

Le troisième chapitre détaillera les aspects répartis de VIPER. Nous commencerons par présenter les couches basses de son architecture qui permettent de gérer la distribution des données et des calculs. La première de ces couches, la plate-forme répartie, encapsule les aspects dépendants des systèmes d'exploitation et de communication de façon à assurer la portabilité de VIPER. La seconde couche permet de développer simplement des applications réparties grâce à des objets parallèles qui cachent tous les aspects répartis de telles applications. Enfin, nous montrerons, en fin de chapitre, comment les univers virtuels et les espaces de stimuli permettent de résoudre les problèmes de répartition des applications ciblées.

Dans le quatrième chapitre, après une évaluation de notre architecture tant au niveau des originalités proposées qu'au niveau de son efficacité de gestion de la répartition, nous présenterons quelques applications développées ou en cours de développement avec VIPER. La première application est un modeleur d'objets et de scènes qui permet de sculpter des objets virtuels dans des formes simples grâce à des outils de déformations. Les deux autres applications présentées sont des applications industrielles de notre modèle de calcul. La première de ces applications industrielles, GRS (*Generic Railway System*), est un simulateur

de gare et la seconde, PROVIS (PROtoypage VIrtuel de Systèmes), permet le prototypage virtuel coopératif de satellites.

Enfin, nous conclurons ce document en insistant sur l'originalité du modèle de calcul réparti proposé et nous présenterons des perspectives de recherche.

Chapitre 1. La réalité virtuelle distribuée – Etat de l'art

Les premiers environnements virtuels distribués sont apparus aux alentours de 1986. Le tout premier monde virtuel commercial « en ligne » (*on-line*) fut Habitat [Morningstar 90]. Créé en 1986 par Lucasfilm Games en collaboration avec Quantum Computer Services (qui est devenu depuis America Online), cet environnement, fonctionnant au départ sur des Commodore 64, était représenté par des scènes animées en deux dimensions dans lesquelles interagissaient des utilisateurs connectés via des modems à 300 bauds. Habitat a réussi à drainer, en tout, 15 000 utilisateurs sur une base totale de 100 000 utilisateurs de ce type de service « en ligne » (à l'époque). Les créateurs d'Habitat ont ensuite créé une société appelée Electric Communities qui en collaboration avec Fujitsu a produit WorldsAway qui est actuellement l'un des mondes virtuels 2D les plus utilisés au monde (WorldsAway est accessible depuis Compuserve).

Depuis ces premières expérimentations, le domaine des environnements virtuels distribués a « explosé » aussi bien au niveau industriel (essentiellement dans les industries des loisirs et dans le domaine militaire) qu'au niveau de la recherche avec l'apparition de nombreux systèmes et architectures distribués spécifiques.

Dans ce chapitre nous allons présenter un certain nombre de caractéristiques des systèmes de réalité virtuelle distribuée qui peuvent être utilisées pour les classer. Ensuite, un classement selon l'architecture distribuée utilisée servira de fil conducteur à une présentation des systèmes existants les plus significatifs. Ainsi, nous commencerons par présenter les premiers systèmes qui étaient dotés d'une architecture centralisée simple. Nous parlerons ensuite des systèmes à architecture distribuée dédiés à la simulation. Puis nous décrirons des systèmes génériques dotés d'une architecture client-serveur. La partie suivante détaillera des systèmes génériques à architecture égal à égal (*peer to peer*). Enfin, après une discussion sur les tentatives actuelles de standardisation de ce domaine, nous conclurons cette présentation de la réalité virtuelle distribuée par des tableaux comparants les systèmes présentés.

1.1. Taxonomie des systèmes de réalité virtuelle distribuée

Il existe de nombreux critères pour classer les systèmes de réalité virtuelle distribuée. Nous en avons choisi quatre qui nous paraissent importants pour pouvoir comparer le système que nous proposons aux systèmes existants.

1.1.1. Généricité du système

Un système de réalité virtuelle distribuée peut être générique vis à vis des plates-formes matérielles gérées par le système. Par exemple, un système comme WAVES (cf. 1.4.2) permet de gérer des environnements virtuels distribués sur des stations de travail UNIX et sur des compatibles PC. D'autres systèmes comme NPSNET (cf. 1.3.3) sont limités à un seul type de plate-forme matérielle : les stations de travail Silicon Graphics.

Un système de réalité virtuelle peut être soit dédié à un type d'application (par exemple la simulation militaire) soit adapté à de nombreux types d'application. On dira dans ce dernier cas que le système est générique vis à vis des différents types d'applications supportées.

Enfin, un système de réalité virtuelle distribué peut gérer en même temps de nombreux mondes virtuels disposant chacun de caractéristiques propres (lois physiques, décors…). On dira dans ce cas que le système est générique vis à vis des types de mondes qu'il peut gérer simultanément.

1.1.2. Distribution géographique et multi-utilisateur

La réalité virtuelle distribuée (RVD) a deux buts principaux : augmenter la puissance de calcul du système de façon à accélérer globalement la simulation d'une part et permettre à plusieurs utilisateurs, éloignés géographiquement, de se retrouver dans le même monde virtuel (pour réaliser des tâches en coopération) d'autre part. Le nombre d'utilisateurs et la répartition géographique (ou non) du système sont des critères importants pour juger de l'extensibilité du système et des applications qui peuvent être mises en œuvre grâce au système.

1.1.2.1. Mono-site et mono-utilisateur

Cette première catégorie correspond au premier but de la RVD, l'augmentation de la puissance de calcul. Les systèmes de ce type datent des débuts de la RVD immersive. En effet pour procurer une immersion convenable il faut une puissance de calcul non négligeable qu'il était difficile de posséder à cause du coût des stations de travail graphiques de l'époque (1989-1993). Il n'était pas rare alors de voir des systèmes disposant d'une station graphique (moyenne gamme) par œil en plus de stations s'occupant de la gestion du monde virtuel. Les problèmes dû à cette répartition de la simulation (répartition des données, synchronisation entre les différentes machines…) ont amené à la création des premiers systèmes génériques qui évitent le développement d'une architecture répartie pour chaque application.

Des exemples de systèmes de cette première catégorie sont l'IHM de l'IPA Stuttgart [Strommer93], la première version du MR Toolkit [Shaw 92] et DIVER [Gossweiler 93].

1.1.2.2. Mono-site et multi-utilisateur

Le second type de système est apparu à la même période pour les mêmes raisons mais permettait, en outre, de gérer plusieurs utilisateurs sur un même site. C'est le cas de RB2 [Blanchard90] ainsi que des premières versions de dVS [Pountain 91] et d'AVIARY [West92] (sur réseau de Transputer). Ce sont en général des systèmes de transition qui sont depuis devenus des systèmes de la catégorie suivante.

1.1.2.3. Multi-sites gérant quelques dizaines d'utilisateurs

Cette catégorie est historiquement la première à être apparue avec des systèmes non immersifs tels que les MUDs (qui peut signifier : *Multiple User Dimensions, Multiple User Dungeons* ou *Multiple User Dialogues*) [Pulkka 95]. Depuis les systèmes immersifs sont eux aussi parvenus à ce niveau et des systèmes tels que DIVE [Hagsand 96] ou MASSIVE [Greenhalgh 95] ont été utilisés de façon effective pour permettre à une dizaine d'utilisateurs répartis géographiquement de travailler dans un même monde virtuel.

1.1.2.4. Multi-sites à large échelle

Les systèmes de cette catégorie sont essentiellement des systèmes militaires qui permettent de simuler des opérations d'entraînement comportant plusieurs centaines d'utilisateurs. Nous pouvons par exemple citer SIMNET [Calvin 93] et NPSNET [Macedonia 94] que nous présentons dans la partie 1.3.

1.1.3. Modèles d'architecture de communication

Avant de décrire les architectures de communication utilisées par les systèmes de RVD il nous semble utile de présenter quelques généralités sur les réseaux et leurs protocoles.

1.1.3.1. Les différents types de réseaux et leurs protocoles

Les réseaux d'interconnexions sont principalement de deux types : locaux et grande distance. Les réseaux locaux (LAN : *local area network*) relient directement des systèmes informatiques entre eux sur un même site géographique. Ils sont caractérisés par une latence faible (durée de transit d'un paquet sur le réseau), une bonne fiabilité (les paquets sont très rarement perdus et sont toujours reçus dans l'ordre d'émission) et une bande passante moyenne (quelques dizaines de mégabits par seconde). Les réseaux grandes distances (WAN : *wide area network*) relient en général des réseaux locaux à d'autres réseaux locaux souvent sur de longues distances. Ils sont caractérisés par une latence importante, une fiabilité en général mauvaise (pertes de paquets et ordre perturbé) et une bande passante faible (quelques mégabits par seconde). Les technologies réseau récentes, comme ATM (*Asynchronous Transfer Mode*), tendent à estomper les différences entre ces deux types de réseau mais elles ne sont pas encore largement disponibles.

Pour que deux ordinateurs puissent communiquer sur un même réseau il faut qu'ils utilisent une même suite de protocoles. La suite de protocole la plus utilisée actuellement est TCP/IP (*Transmission Control Protocol/Internet Protocol).* L'ensemble de tous les ordinateurs et de tous les réseaux mondiaux connectés gérés par les protocoles TCP/IP forme l'Internet. Sans entrer dans le détail nous allons présenter brièvement les protocoles de TCP/IP qui sont utilisés pour réaliser des applications de RVD. Pour une présentation plus détaillée de TCP/IP nous dirigeons le lecteur vers l'ouvrage : [Stevens 94].

1.1.3.2. Les différents modes de communication offerts par TCP/IP

Lorsque plusieurs applications communiquent à travers un réseau géré par TCP/IP elles peuvent utiliser plusieurs méthodes (Figure 1) : des communications point à point, la diffusion totale (*broadcasting*) et la diffusion sur des groupes de communications (*multicasting*).

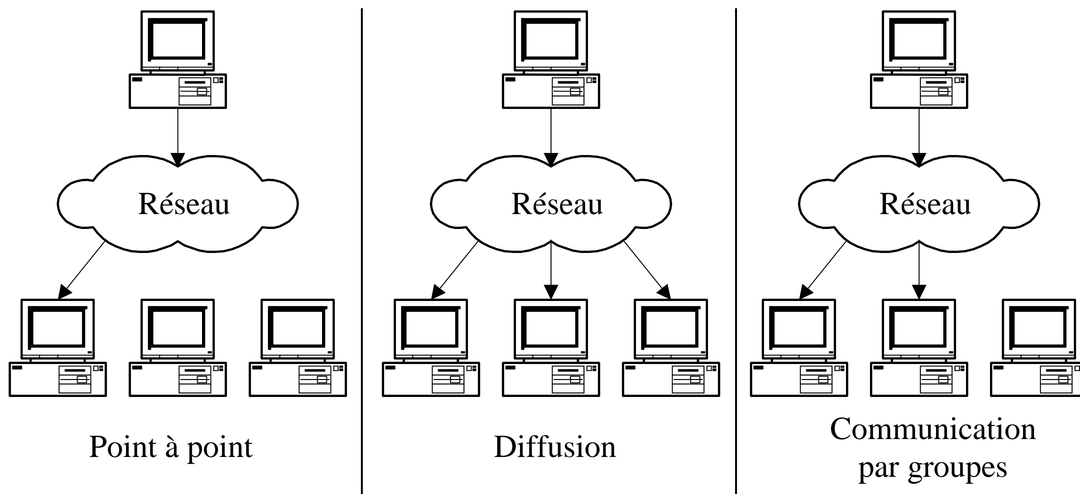


Figure 1 : Différents modes de communication à travers un réseau

La communication point à point consiste à envoyer un paquet à une machine précise identifiée par son adresse sur le réseau. C'est le mode de communication le plus simple et le plus utilisé actuellement. TCP/IP propose deux protocoles gérant des communications de ce type : UDP (*User Datagram Protocol*) et TCP (*Transmission Control Protocol*). UDP est le protocole le plus simple, il offre un service non connecté (le programmeur doit préciser l'adresse du destinataire à chaque message) et sans garantie de fiabilité. TCP offre, lui, un service connecté et met en œuvre des mécanismes permettant d'améliorer la fiabilité de bout en bout sur des réseaux peu fiables. Néanmoins ces mécanismes ont un impact important sur la performance de TCP et ne sont pas très adaptés aux applications de RVD. En conséquence, la plupart des systèmes de RVD utilisant des communications point à point choisissent UDP et lui ajoutent des mécanismes, pour améliorer la fiabilité, en adéquation avec leurs besoins.

La diffusion totale consiste à envoyer directement un paquet à tous les ordinateurs connectés à un réseau. Ce qui est évidemment beaucoup moins coûteux en ressources réseau (bande passante) que d'envoyer le même paquet à toutes les machines en utilisant le mode de communication précédent. UDP propose un service de ce type en utilisant comme adresse de destination d'un message une adresse spécifique qui n'identifie pas une machine mais un

réseau. Pour éviter de submerger de nombreuses machines, qui ne sont pas intéressées par les paquets qui sont diffusés, ce type de communication est en général limité à des réseaux locaux et n'est pas disponible globalement sur Internet. D'autre part, la plupart des réseaux étant utilisés simultanément pour plusieurs applications, ce type de communication est en général évité pour ne pas déranger les autres utilisateurs. En fait, la diffusion totale n'est utilisée en RVD que sur des réseaux spécifiques à une application par des organismes gouvernementaux (l'armée américaine notamment).

La diffusion sur des groupes de communications consiste à envoyer en une seule opération un paquet à plusieurs machines sur un réseau. UDP propose un service de ce type en utilisant comme adresse de destination d'un message une adresse spécifique qui n'identifie pas une machine, ni un réseau, mais un groupe de communications. Les machines (en fait leurs applications) qui souhaitent recevoir les paquets d'une diffusion demandent leur inscription au groupe correspondant. Par la suite tout message envoyé au groupe est reçu par toutes les machines inscrites. Ainsi, seules les machines qui sont intéressées reçoivent le paquet ce qui est un très gros avantage par rapport à la diffusion totale. Il est, aussi, intéressant de noter que les machines peuvent s'inscrire et se retirer d'un groupe dynamiquement.

La diffusion sur des groupes de communications est disponible globalement sur l'Internet grâce au MBone. Il s'agit d'un réseau virtuel pour la diffusion sur groupe qui a été conçu par Cassner et Deering [Cassner 92]. Ce réseau est virtuel car il utilise le même support de communication que l'Internet. Les réseaux locaux qui utilisent la diffusion sur groupe sont reliés via des routeurs spécialisés (les mrouteurs : *multicast routers*) qui sont soit des routeurs réels mis à jour pour gérer le *multicasting* ou des stations de travail couplées à des routeurs standards. La grande majorité des routeurs qui forment l'Internet ne permettant pas le *multicasting*, le MBone est augmenté par un mécanisme appelé « *tunneling* » qui consiste à encapsuler des paquets *multicast* dans des paquets standards et à les échanger entre deux mrouteurs (via des routeurs standards). Petit à petit ce mécanisme devrait disparaître car les nouveaux routeurs sont en général capables de gérer la diffusion par groupes et au fur à mesure qu'ils remplaceront les anciens routeurs d'Internet (ou que ceux-ci seront mis à jours) le MBone s'agrandira et deviendra plus efficace.

Le problème de ce mode de communication est lié à son avantage par rapport à la diffusion. Pour éviter que toutes les machines connectées au MBone ne reçoivent tous les

paquets envoyés sur tous les groupes de communications, les notifications d'inscription et de retrait d'une machine à un groupe de communications sont remontées de mrouteur en mrouteur dans le réseau. Ainsi, lors d'une diffusion, chaque mrouteur sait s'il doit ou non envoyer le paquet au mrouteur suivant. Le problème est la latence introduite par ces remontées dans le réseau. Cette latence implique que l'inscription et le retrait de groupes ne peuvent être très dynamiques.

Dans la suite de cette partie, différentes architectures réparties utilisées dans les systèmes de RVD vont être présentées.

1.1.3.3. Architecture client-serveur centralisée

Dans ce type d'architecture, la base de données représentant le monde virtuel est entièrement gérée par un seul serveur. Chaque client envoie les ordres de son utilisateur (déplacement, communication textuelle, saisie d'objets…) au serveur central et reçoit en retour l'information perçue par l'utilisateur. Ainsi, il n'y a qu'un seul utilisateur qui peut agir à la fois.

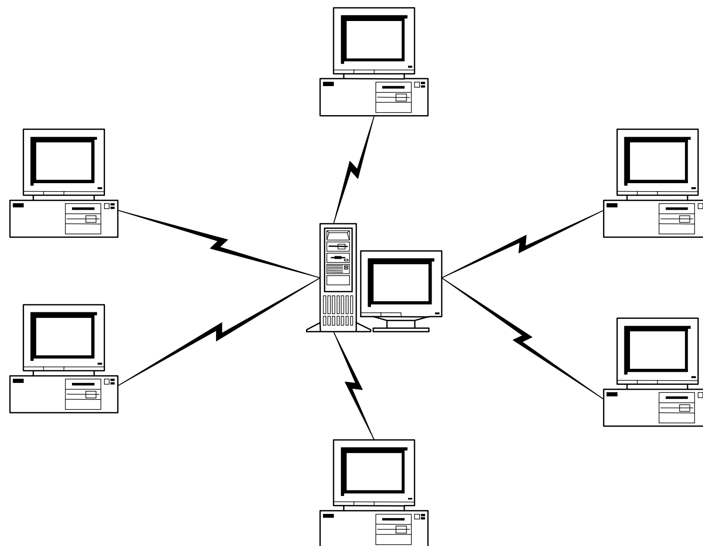


Figure 2 : Architecture client-serveur centralisée

L'architecture de communication est en forme d'étoile, chaque client étant connecté directement au serveur central (Figure 2). Ce modèle d'architecture n'est évidemment pas très extensible et permet de gérer au mieux une cinquantaine d'utilisateurs en même temps dans le cas de communications purement textuelles (MUD par exemple). Nous reparlerons de ce type

d'architecture dans la partie 1.2.1 qui présente les ChatWorlds qui sont essentiellement construits sur des architectures de ce type.

1.1.3.4. Architecture client-serveur distribuée

Ce modèle d'architecture ressemble au précédent avec la différence suivante : la base de données représentant le monde virtuel est répartie sur les différents clients et, donc, seules les communications entre les clients sont gérées par le serveur. Ce modèle est en particulier utilisé par BrickNet [Singh 94].

En général, le serveur gère divers filtres pour éviter que tous les messages qu'il reçoit ne soient diffusés à tous les autres clients. Ces filtres sont en général spatiaux : le message d'un client est envoyé à tous les clients dont l'avatar (entité graphique qui représente un utilisateur) est proche dans le monde virtuel.

Dans le cas de mondes virtuels importants (en nombre d'entités présentes) et dynamiques le serveur peut vite devenir un goulet d'étranglement.

1.1.3.5. Architecture client-serveur distribuée avec plusieurs serveurs

Pour pallier ce problème, les concepteurs de systèmes de RVD ont eu l'idée d'utiliser plusieurs serveurs. Ces serveurs peuvent être organisés de diverses façons.

Le plus souvent il s'agit d'une organisation fonctionnelle : chaque serveur s'occupant d'un type de communication. Ainsi on peut avoir, par exemple, un serveur pour gérer les collisions et un autre pour gérer les interactions distantes (voix, vision…). Dans ce cas, on peut retrouver un goulet d'étranglement si de nombreuses entités interagissent simultanément.

Pour éviter ce problème, il est possible de hiérarchiser des serveurs s'occupant d'un même type de communication. Chaque client communique directement avec le serveur le plus proche (en terme de distance dans le réseau), qui envoie les mises à jour aux autres serveurs (les serveurs peuvent aussi dialoguer entre eux pour gérer des filtres), qui à leur tour communiquent chacun avec leurs clients. Ce modèle augmente la complexité pour maintenir une base de données cohérente, mais diminue l'impact de l'ajout de nouveaux clients (tant qu'il y a suffisamment de serveurs). D'autre part, les messages des clients pouvant traverser plusieurs serveurs pour aboutir à d'autres clients, la latence augmente ce qui diminue le

niveau d'interactivité. Ce type d'architecture est utilisé par WAVES (cf. partie 1.4.1), par MASSIVE (cf. partie 1.4.3) et par RING [Funkhouser 95].

1.1.3.6. Architecture égal à égal (peer to peer) distribuée point à point

Dans ce type d'architecture, tous les ordinateurs ont le même rôle. La base de données représentant le monde virtuel est dupliquée sur les différents sites, et les machines communiquent les modifications locales à tous les autres sites (Figure 3).

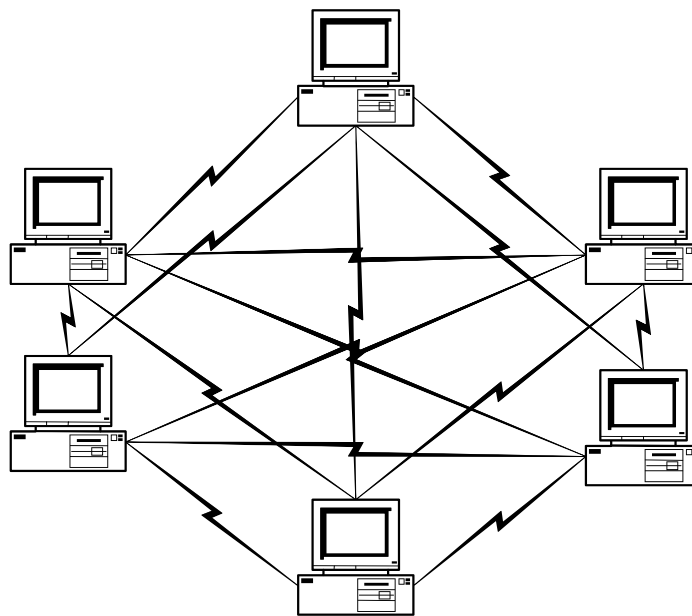


Figure 3 : Architecture égal à égal distribuée point à point

Le principal problème de ce type d'architecture, du point de vue de l'extensibilité, est que le nombre de messages de mises à jour augmente beaucoup avec le nombre de sites. Si chaque site envoie une mise à jour, à chaque cycle de simulation, n*(n-1) messages vont circuler sur le réseau à chaque cycle [Gossweiler 94]. Pour pallier ce type de problème il est possible de diminuer la fréquence des messages de mise à jour grâce, par exemple, à la technique du *dead-reckoning* que nous présentons dans la partie 1.3.1. Néanmoins la meilleure technique pour éviter d'envoyer autant de messages est d'utiliser les modes de communications permettant de diffuser un message à plusieurs sites en une seule opération. C'est ce qui est mis en œuvre dans les deux modèles d'architecture suivants.

Un autre problème de ce type d'architecture est le maintient de la cohérence des copies de la base de données représentant le monde virtuel. En effet, pour éviter que plusieurs sites

puissent modifier la même partie de la base de données un mécanisme de protection doit être mis en œuvre. De nombreux mécanismes, issus des systèmes d'exploitation distribués, existent pour gérer ce problème, du plus simple : à un instant donné, un seul site est autorisé à modifier la base de données et ce droit circule de site en site (grâce à un anneau à jeton logique par exemple) ; au plus complexe : tous les sites font les modifications qu'ils veulent en précisant leur date et lorsqu'un conflit ce produit la modification la plus ancienne est prise en compte et les autres sont annulées (ceci implique que l'on puisse réellement annuler les autres modifications…).

1.1.3.7. Architecture égal à égal distribuée en mode diffusion

Pour réduire le nombre de messages circulant sur le réseau dans les architectures égal à égal distribuées on peut utiliser d'autres types de communication que les communications point à point. Le premier type de communication qui a été utilisé dans ce but à été la diffusion. Ce type d'architecture est utilisé par SIMNET [Calvin 93].

1.1.3.8. Architecture égal à égal distribuée avec groupes de communications

Cependant, comme nous l'avons vu précédemment, la diffusion totale est très gênante pour les autres utilisateurs du réseau, donc les concepteurs se sont rapidement tournés vers la diffusion sur des groupes de communications qui apporte à peu près les mêmes avantages sans cet inconvénient. D'autre part, l'utilisation de nombreux groupes de communications peut augmenter l'extensibilité d'un système en diminuant le nombre de machines en communication directe à tout instant. Ce type de modèle est utilisé par NPSNET [Macedonia 95a], Spline [Barrus 95] et MASSIVE-2 [Greenhalgh 97].

Ce type d'utilisation des groupes de communications s'apparente à l'utilisation de multiples serveurs dans une architecture client-serveur distribuée avec plusieurs serveurs (cf. partie 1.1.3.5). Le rôle joué par chaque serveur est assuré en partie par un groupe de communications. L'avantage de ce type d'architecture par rapport à une architecture à serveurs multiples est de minimiser la latence due à la traversée du ou des serveurs (serveurs organisés de façon hiérarchique) et de minimiser le nombre de messages émis sur le réseau. L'inconvénient dans ce cadre d'un groupe de communications est que celui-ci ne peut pas assurer un filtrage intelligent des messages (ce qui peut être fait par un serveur). D'autre part le temps de latence d'inscription/retrait d'un groupe implique une dynamique plus faible que

dans le cas d’une architecture à multiples serveurs. Récemment des architectures mêlant client-serveur et groupes de communications sont apparues pour essayer de concilier les avantages des deux techniques en limitant leurs inconvénients. Smallview [Broll 97] est une architecture de ce type.

1.1.3.9. CORBA

Etant donné que les environnements virtuels distribués peuvent être représentés par des systèmes à objets distribués on pourrait croire qu'une architecture générique de ce domaine pourrait être directement utilisée. Nous allons présenter une architecture de ce type : CORBA et nous montrerons que ces architectures ne sont en fait pas vraiment adaptées à notre domaine.

CORBA (*Common Object Request Broker Architecture*) a été défini par un organisme international, l'OMG (*Object Management Group*), qui propose de normaliser l'interopérabilité d'applications orientées objets réparties. CORBA représente un modèle objet concret dérivé d'un modèle abstrait défini par l'OMG dans le guide de l'OMA (*Object Management Architecture*). L'architecture est générique du point de vue du langage, du système d'exploitation et de la répartition.

Dans les systèmes orientés objets, deux objets communiquent via l'envoi d'un message. Un objet (le client) envoie un message à un autre objet (le serveur) pour lui demander d'effectuer une opération. C'est la réception du message qui déclenche l'exécution de l'opération (ou méthode).

CORBA propose d'insérer entre ces deux objets, un intermédiaire (l'ORB : *Object Request Broker*) qui reçoit le message (appelé requête) et se charge de trouver l'objet serveur (appelé *object implementation*), d'acheminer, de traduire les données paramètres de la méthode appelée, d'activer la méthode proprement dite et de convoyer si nécessaire les résultats de l'exécution de la méthode. C'est donc l'ORB qui va gérer la répartition et l'hétérogénéité du système.

Un ORB est structuré en un certain nombre de parties (Figure 4). Ce schéma montre les différentes interfaces de l'ORB.

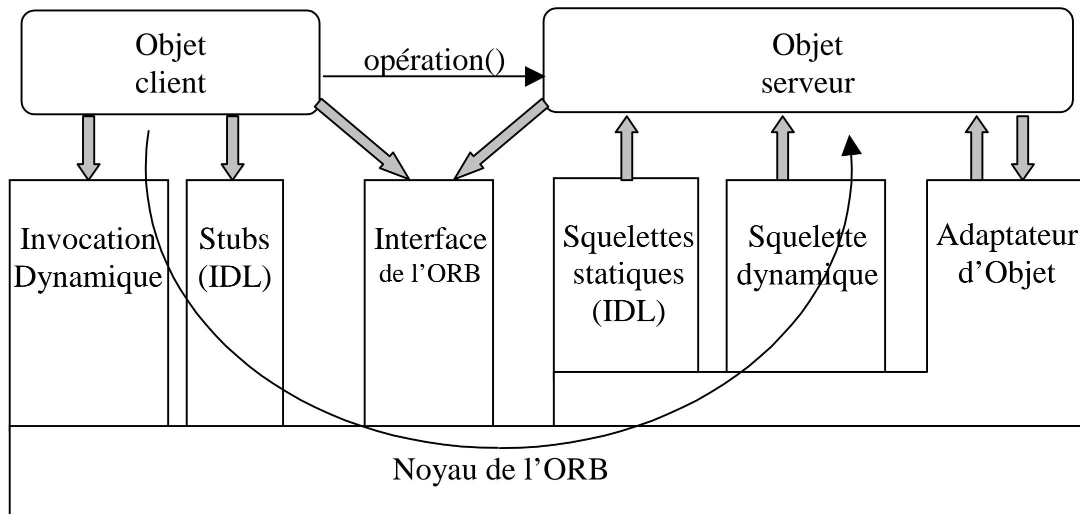


Figure 4 : Structure d'un ORB de CORBA

Pour envoyer une requête le client peut utiliser soit des souches IDL (fonctions *stubs*) spécifiques à chaque objet cible (les souches sont générées à partir de la définition de l'interface de l'objet dans un langage appelé IDL : *interface definition language*), soit une interface d'invocation dynamique, générique, qui construit petit à petit la requête en fonction des données spécifiées par le client (référence à l'objet cible, nom de la méthode appelée et valeur des paramètres).

L'objet serveur reçoit la requête soit via des fonctions « squelettes » générées par l'IDL (qui sont donc spécifiques à l'objet) ou via un squelette dynamique (générique). Lors du traitement de la requête, l'objet serveur peut obtenir un certain nombre de services de l'ORB via un adaptateur d'objet. Enfin, lorsque la requête est traitée les éventuels résultats sont renvoyés via l'ORB à l'objet client qui reprend son exécution.

Le problème actuel de CORBA du point de vue de la Réalité Virtuelle Distribuée est que, bien que l'architecture prévoit une adaptation à plusieurs types de couches transport, seul TCP est actuellement utilisé par les implémentations de CORBA. Or, TCP n'est pas adapté à nos applications (cf. 1.1.2.2). TASC, une société américaine, a proposé, il y a quelques temps, un portage de CORBA sur leur couche transport de type communication par groupe fiable (RAMP *reliable adaptative multicast protocol*) [Smith 96], néanmoins cette version n'est pas encore totalement générique (elle a été écrite pour une base de données répartie d'images satellites) et n'a été qu'examinée par l'OMG. D'autre part, des travaux de recherches en cours ont pour but d'étendre le modèle en incluant la gestion de média continu (tels que le son et la

vidéo numériques) et des abstractions pour gérer les mécanismes de qualité de service [Coulson 96].

Au vu de ces différents problèmes on comprend pourquoi aucuns des systèmes de RVD actuels n'utilise directement une architecture standard gérant des objets distribués. Les auteurs des systèmes que nous présentons dans la suite de ce chapitre ont conçu des solutions plus ou moins spécifiques à la réalité virtuelle de façon à créer des architectures les plus efficaces possibles pour ce domaine. Cependant certains des concepts des architectures génériques à objets distribués ont été adaptés par un certain nombre de systèmes. Par exemple, AVIARY (cf. partie 1.5.2) utilise un concept proche des ORB de CORBA pour permettre à des objets de découvrir d'autres objets. Nous verrons aussi dans la partie 1.4.3 que MASSIVE reprend et adapte le concept de *trader* qui viens de ODP [ODP 95], une architecture concurrente de CORBA. Enfin, bien que les premiers prototypes de la RTI de HLA (cf. partie 1.6.2) aient été réalisés en utilisant une implémentation de CORBA (Orbix), l'implémentation réalisée pour le programme STOW (*Synthetic Theatre Of War*) de l'armée américaine, pour gérer des simulations militaires réelles, est directement mise en œuvre au-dessus de UDP pour des raisons d'efficacité [Calvin 97].

1.1.4. Définition et gestion de mondes virtuels dynamiques : les comportements

Un système de réalité virtuelle distribuée doit permettre de définir la structure des mondes virtuels qu'il gère. Cette structure est en général constituée de deux parties : la présentation et le comportement. La présentation est en général mise en œuvre par un graphe de scène qui comporte des informations multimédias (formes géométriques, fichiers sonores…). Cette présentation provient souvent d'un simple fichier de description de scène défini dans un format spécifique au système ou dans un format générique comme VRML.

La définition de comportements est plus complexe. Les comportements peuvent être définis de diverses façons. Ils peuvent être compilés avec le système, compilé dans une application externe, interprétés par le système ou liés dynamiquement avec le système.

1.1.4.1. Comportements compilés avec le système

Les comportements compilés avec le système sont les plus simples à mettre en œuvre. Dans ce cas, les comportements sont définis dans le même langage que le système (en général C ou C++) et compilés dans des modules qui sont liés statiquement avec le programme principal qui gère le système.

Les avantages de ce type de comportements sont d'être réalisés dans des langages très utilisés et donc très connus. Les comportements sont, de plus, mis en œuvre de façon très efficace. Néanmoins, l'inconvénient de cette solution est que le système doit être modifié et recompilé à chaque introduction ou modification d'un comportement.

Ce type de comportement est utilisé par les systèmes de réalité virtuelle militaires comme NPSNET dont les comportements simulent les caractéristiques de véhicules qui ne changent pas très souvent. Ce type de comportement est aussi utilisé par la première version de MASSIVE qui est quasiment dédiée à des applications de conférence virtuelle ne nécessitant pas de nombreux comportements.

1.1.4.2. Comportements compilés dans des applications externes

Une solution un peu plus complexe à mettre en œuvre est de créer une application qui gérera le comportement d'un ou plusieurs objets du monde virtuel et qui communiquera avec le système pour permettre aux actions de ces objets d'être perceptibles dans tout l'environnement. Dans ce cas, l'application est souvent écrite dans le même langage que le système (bien que ce ne soit pas nécessaire) et est liée avec une librairie qui permet de gérer la communication.

L'avantage de cette solution est de ne pas avoir à recompiler l'ensemble du système à chaque introduction ou modification de comportement tout en conservant un bon niveau d'efficacité pour le comportement. L'inconvénient est que les communications à mettre en œuvre pour les échanges entre l'environnement d'un objet et son comportement entraînent une augmentation de la latence et de la complexité du système (avec une augmentation du risque d'erreurs de programmation). D'autre part, la compilation et l'édition de liens des modules de l'application avec la librairie du système (et éventuellement d'autres librairies) entraînent un délai lors des modifications du comportement qui peut être assez long.

Cette solution est utilisée par des systèmes comme dVS (cf. partie 1.4.1), DIVE (cf. partie 1.5.1) et AVIARY (cf. partie 1.5.2).

1.1.4.3. Comportements interprétés

Cette solution est la plus complexe à mettre en œuvre car elle nécessite qu'un interpréteur du langage en question soit intégré au système. Les comportements sont récupérés sous forme de fichiers textes et sont interprétés pendant l'exécution du système.

Les avantages de cette solution sont multiples. Les comportements peuvent être utilisés directement sur n'importe quelle machine même si le système est hétérogène. En conséquence, les comportements peuvent être envoyés d'une machine à une autre pendant l'exécution du système (migration de comportements). Les modifications d'un comportement peuvent être rapidement prises en compte (sans recompilation ni édition de liens). Ceci pouvant être fait dynamiquement pendant l'exécution. On pourra ainsi réaliser un prototypage rapide d'une application de RVD. L'inconvénient de cette solution est que l'interprétation est souvent peu efficace. Cependant, des langages comme Java qui sont mi-compilés/mi-interprétés voire même compilés à la volée (*just in time compiling*), ont tendance à devenir quasiment aussi efficaces que les langages compilés. L'autre inconvénient est la différence de langage qui peut exister entre l'implémentation du système et des comportements. Cette différence peut poser des problèmes d'interface et ainsi restreindre les possibilités du développeur de comportements. De plus, si le langage de définition de comportements n'est pas un langage standard il nécessitera une étape d'apprentissage.

Ce type de comportement existe dans DIVE grâce à son interface DIVE/Tcl (cf. 1.5.1.4) et dans BrickNet [Singh 95] (qui dispose d'un langage spécifique non standard appelé Starship).

1.1.4.4. Comportements liés dynamiquement

Cette dernière solution consiste à compiler des comportements dans une librairie qui sera liée dynamiquement au système pendant son exécution. Ce mécanisme que nous détaillons dans le chapitre 2 n'est, à notre connaissance, utilisé que par notre système. Il possède quasiment les mêmes avantages que les comportements interprétés (possibilité de faire migrer les comportements, rapidité de prise en compte des modifications du comportement), tout en étant aussi efficace que le premier type de comportement. Son seul inconvénient est la non

portabilité des comportements dans un système hétérogène. Mais nous tentons d'y apporter une solution (cf. partie **2.4.2.2**).

Après cette présentation de quelques critères permettant de classer les systèmes de réalité virtuelle distribuée, nous allons décrire un certain nombre de systèmes existants parmi les plus typiques. Pour chaque système nous avons essayé, quand cela est possible, de décrire : la distribution géographique du système, le nombre d'utilisateurs pouvant être pris en compte par le système, l'architecture distribuée du système et le modèle choisi pour la définition des comportements. Nous décrivons aussi pour chaque système ses spécificités et ses caractéristiques les plus intéressantes.

1.2. Premiers systèmes à architecture centralisée

1.2.1. Les Chatworlds

Les mondes virtuels 2D comme Habitat sont communément appelés des ChatWorlds (littéralement : mondes où l'on bavarde). Ces premiers environnements dérivent des MUDs (*Multiple User Dimensions)*, mondes virtuels généralement en mode texte qui sont apparus au début des années 80 avec les premiers systèmes « en ligne », les BBS (*Bulletin Board Systems*), pour permettre à leurs utilisateurs de discuter.

Le premier MUD a été créé à l'Université de Essex (Angleterre) en 1979 dans le but de réaliser une version multi-utilisateurs d'un jeu d'aventure en mode texte [Bartle 90]. Depuis ce précurseur, de nombreux MUDs ont été créés et il en existe à l'heure actuelle plus de 250. Habitat a rajouté au mode texte initial des graphiques 2D et a ainsi augmenté le niveau de « présence » (l'utilisateur croit qu'il est réellement présent dans le monde virtuel) ce qui le place comme le précurseur en terme de réalité virtuelle distribuée (ceci est discutable et dépend de la définition du terme réalité virtuelle qui est changeante…).

Ces environnements se sont rapidement structurés de façon à permettre aux utilisateurs de s'organiser en petites communautés virtuelles et de « bâtir » leur monde virtuel idéal. Chaque MUD a généralement un thème qui est souvent issu d'un ouvrage de science fiction. Les

MUDs sont organisés en « pièces » (qui peuvent aussi représenter des portions de territoire en extérieur) comprenant une description (au départ en mode texte), des objets aux comportements plus ou moins complexes qui sont souvent des représentations d'objets de la vie courante (tableaux d'affichage, boîtes à lettres, mais aussi distributeurs de boissons virtuelles, taverniers…) et des issues permettant d'accéder à d'autres pièces. Les comportements des objets sont définis dans des langages spécifiques très simples qui sont interprétés par le serveur.

Ces environnements sont caractérisés par une architecture client-serveur centralisée très simple. Le serveur gère tous les aspects du monde virtuel en recevant des informations de chacun des clients et en diffusant ensuite ces informations aux clients qui peuvent avoir accès à cette information. Le rôle central du serveur pose évidemment de nombreux problèmes. D'une part, en cas de panne du serveur le monde virtuel est totalement inopérant. D'autre part, toutes les communications transitant par le serveur, celui ci devient rapidement un goulet d'étranglement. Ainsi ces systèmes ne gèrent en général pas plus de quelques dizaines d'utilisateurs. Ce nombre relativement important étant atteint grâce à la faible interactivité de ces systèmes.

1.2.2. RB2

RB2 (*Reality Built for two*) [Blanchard90], de la société VPL Research, fut, dès 1990, le premier système, grand public, commercialisé permettant de définir des environnements virtuels tridimensionnels sur lesquels on peut interagir avec des outils de réalité virtuelle. Le système a été conçu pour gérer deux utilisateurs sur un même site géographique. En effet le monde virtuel est géré par un seul système informatique. Le système permet la conception et la gestion de réalités virtuelles.

1.2.2.1. Décomposition logique

Le système se décompose en trois modules :

- Un modeleur, RB2 Swivel, qui supporte des liens hiérarchiques entre objets, des textures et des contraintes sur les mouvements. Il sert à définir l'aspect visuel des objets du monde virtuel.

- Un outil, Body Electric, permettant la définition puis la gestion et la modification en temps réel du comportement de ces objets dans le monde virtuel. Les comportements sont définis et modifiés interactivement grâce à Flex, un environnement de programmation graphique de type « Boîtes et Flèches ». Les flots de données suivent les flèches et les boîtes contiennent chacune une fonction capable de modifier les données ou de générer des événements. Les modifications de comportement peuvent intervenir en cours d'utilisation du monde virtuel et en temps réel.

- Le dernier module, Isaac, s'occupe du rendu temps réel du monde virtuel. Un sous module permet de pré-calculer la photométrie du monde virtuel en utilisant la technique de radiosité.

1.2.2.2. Première réalisation

La décomposition physique du système (Figure 5) comprend :

- Un Macintosh II d'Apple pour la conception et la gestion du monde virtuel (sur lequel s'exécutent Swivel et Body Electric), et sur lequel tous les capteurs sont connectés via le DATU (*Data Acquisition and Transmission Unit* : unité d'acquisition et de transfert des données) et le STS (*Spatial Tracking System* : système capteur de mouvement).

- Une ou deux stations de travail Silicon Graphics pour le rendu (Sur lesquelles s'exécute Isaac).

- Eventuellement un périphérique, appelé Audiosphere, qui gère le son tridimensionnel en temps réel.

La liaison entre le Macintosh et les deux Stations est réalisée via des connections Ethernet point à point et une ligne spéciale utilisée pour la synchronisation.

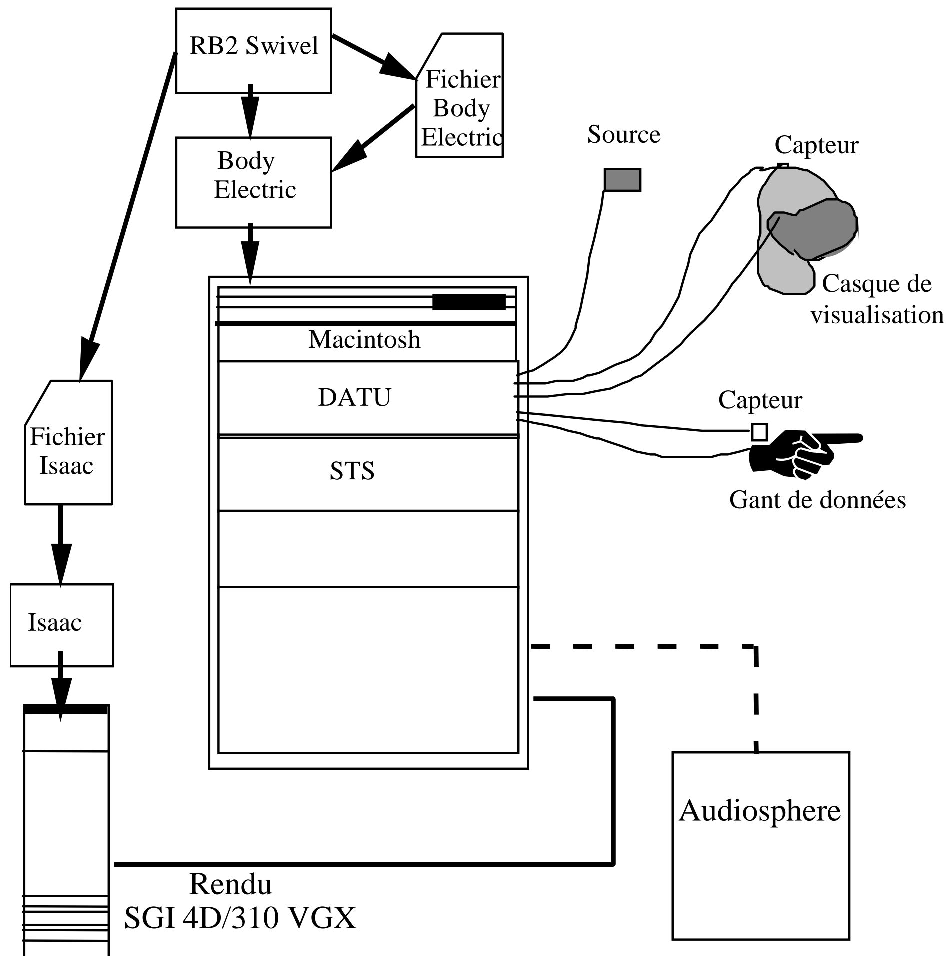


Figure 5 : Architecture de RB2 de la société VPL Research

1.2.2.3. Conclusions sur cette architecture

Le système RB2 fut le premier système commercialisé utilisant des périphériques de réalité virtuelle. Il prônait déjà l'utilisation du parallélisme bien que celui-ci soit faible dans ce système. En effet, seul le rendu est fait en parallèle avec le reste de l'application. Le principal problème étant justement que le reste de l'application est très coûteux en temps de calcul pour des mondes virtuels intéressants. Le Macintosh en effet gère tout seul tous les périphériques de réalité virtuelle, le comportement de tous les objets et les interactions entre les objets et les utilisateurs. En fait, ce système a surtout été utilisé pour réaliser quelques démonstrations « preuves de concept » pour la réalité virtuelle. Dans ce cadre, un simple Macintosh suffisait pour gérer les quelques objets peuplant une scène simple.

1.3. Systèmes à architecture distribuée dédiés à la simulation militaire

Les premières applications de la réalité virtuelle distribuée (bien avant que le nom existe) ont été militaires. Les protocoles SIMNET (*SIMulation NETworking*) et DIS (*Distributed Interactive Simulation*) sont à la base de la plupart des simulateurs en réseau construit pour l'armée américaine. Nous allons décrire dans cette partie SIMNET, DIS et NPSNET, leurs origines, leurs caractéristiques et les problèmes et limitations de ces systèmes.

1.3.1. SIMNET

SIMNET a été à l'origine développé pour D-ARPA (*Defense Advanced Research Projects Agency*) et l'armée américaine par BBN (Bolt Beranek and Newman), Perceptronics et Delta Graphics en 1983 [Calvin 93] [Macedonia 95b]. Le but du système est de permettre à de petites unités militaires (équipage d'un véhicule, groupe de véhicules ou compagnie) d'apprendre à s'organiser et à combattre en équipe. SIMNET a eu le mérite de prouver aux militaires que l'entraînement collectif qui est très important pour les militaires est possible à moindre coût (par rapport aux exercices grandeur nature) grâce aux simulateurs en réseau. On peut donc considérer que SIMNET est la première réussite industrielle de la réalité virtuelle distribuée.

1.3.1.1. Spécifications du système

Les spécifications initiales du système étaient :

- Entre 100 et 100 000 entités doivent pouvoir être simulées.
- Les entités et leurs équipages sont répartis géographiquement.
- Les simulations sont hétérogènes.
- Le système est peu coûteux (comparé au coût d'une simulation grandeur nature).
- Le système est totalement réparti (il n'y a pas de site central).
- Le système fonctionne en temps réel.

1.3.1.2. Structure d'un simulateur SIMNET

Les principaux composants de SIMNET (Figure 6) sont : la maquette du véhicule à simuler avec les différents postes de l'équipage, un générateur d'image de synthèse, une base de

données statique représentant le terrain, un système informatique qui gère la simulation et fourni des données au générateur d’images et un réseau Ethernet.

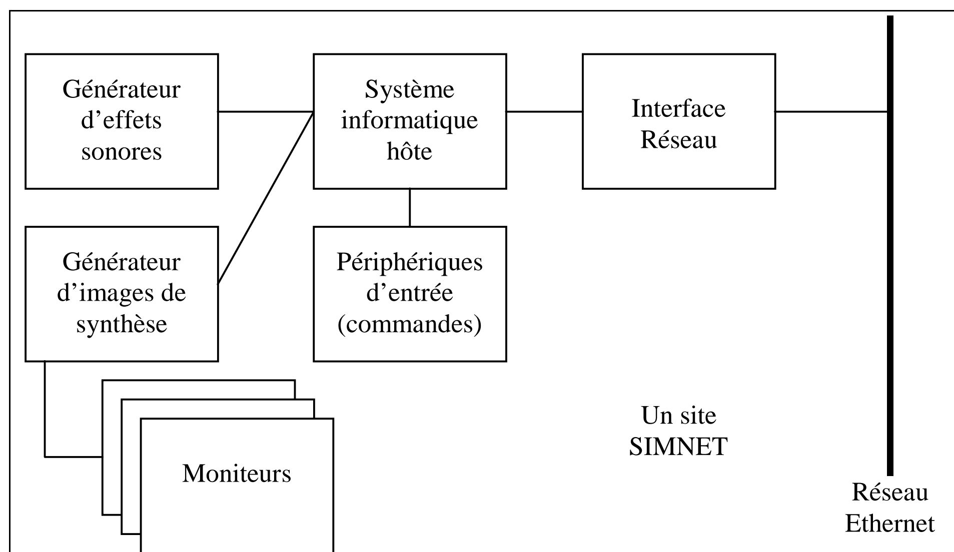


Figure 6 : Structure d’un simulateur SIMNET

1.3.1.3. Principes de fonctionnement

Les principes de fonctionnement d’un simulateur SIMNET sont :

- Le modèle est fondé sur des **objets** et des **événements**. Les objets simulés interagissent entre eux via une série d’événements. Les objets statiques sont connus de tous les objets dynamiques (c’est à dire les objets représentant les participants de la simulation). Ils sont en fait dupliqués sur chaque site. Les objets dynamiques se tiennent mutuellement informés de leur position et de leur statut grâce à un ensemble d’événements prédéfinis.

- Les objets sont **autonomes**. Les événements sont diffusés à tous les objets dynamiques. L’objet qui crée l’événement ne détermine pas les objets qui peuvent être intéressés par l’événement ou comment l’objet qui recevra l’événement peut être affecté par lui. C’est de la responsabilité de chaque objet de déterminer s’il est intéressé par l’événement et comment il doit réagir. L’intérêt de gérer des objets autonomes est que n’importe quel objet peut entrer ou sortir de la simulation à n’importe quel moment.

- Chaque objet transmet la **vérité absolue** à propos de son état. Les objets qui reçoivent les événements doivent déterminer s’ils peuvent percevoir l’événement. Les objets

doivent donc transformer cette vérité absolue de façon à modéliser ce qu'il se passerait dans le monde réel avant de présenter l'information à un être humain ou à un système de capteur. Par exemple, la vérité absolue permet de savoir qu'un véhicule est caché derrière une colline néanmoins lorsque la scène sera affichée le véhicule ne devra pas être visible.

- Les objets transmettent des informations uniquement à propos des **changements** de leur état. Ceci permet de limiter la transmission et la réception d'informations redondantes.

- Les objets utilisent des algorithmes de ***dead-reckoning*** pour extrapoler leur état. Chaque objet extrapole, par exemple, sa position grâce aux derniers événements qu'il a reçus. C'est de la responsabilité des objets d'émettre des événements avant que les extrapolations deviennent trop mauvaises. En fait, chaque objet maintient un modèle extrapolé qui le représente et qui est exactement le même que celui disponible sur les autres sites. Lorsque ce modèle devient trop éloigné de la réalité (d'après un seuil fixé pour l'objet) un événement contenant la mise à jour de cet objet est diffusé. Cet événement contient la position, la vitesse et l'orientation du véhicule ainsi que la position de ces composants (comme les tourelles et les canons d'un char d'assaut), les nuages de poussières et de fumée créés par le véhicule, sa signature thermique et d'autres émissions électromagnétiques. La technique du dead-reckoning est très efficace pour limiter les échanges entre les sites. Elle a d'ailleurs été reprise par de nombreux systèmes de réalité virtuelle distribués militaires ou civils.

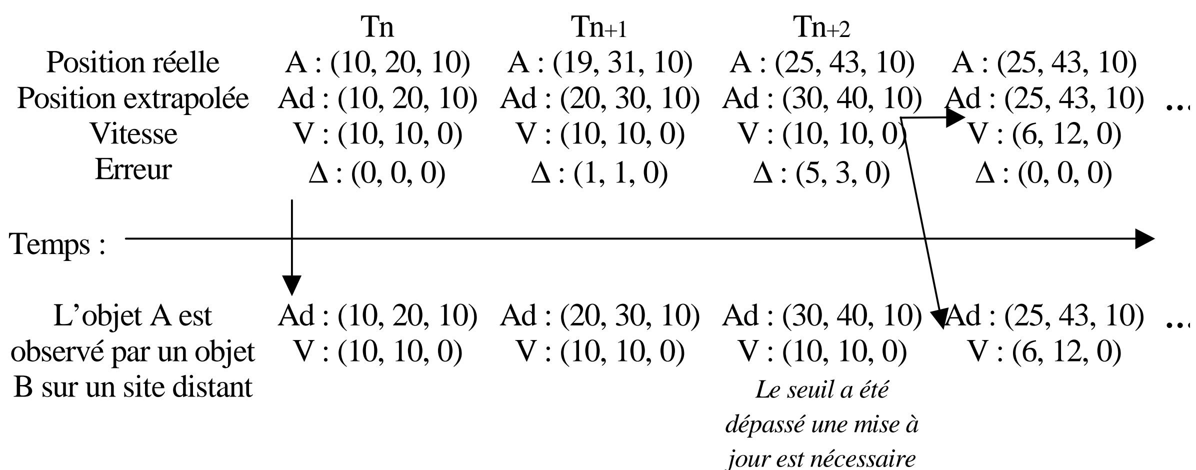


Figure 7 : Exemple d'utilisation du *dead-reckoning*

Sur l'exemple précédent (Figure 7), la position de l'objet A est observé au temps Tn, Tn_{+1} et Tn_{+2}. L'exemple utilise l'algorithme du *dead-reckoning* avec un seuil de 2.5 m sur chaque axe (c'est à dire que dès que l'erreur observée sur l'un des axes dépasse 2.5 le seuil est dépassé). La ligne position réelle reflète la position réelle du véhicule à chaque instant. La ligne position extrapolée est la position du véhicule extrapolée grâce à l'algorithme du dead-reckoning avec une position de départ égale à (10, 20, 10) et une vitesse initiale de (10, 10, 0). Ces deux paramètres sont envoyés au site de l'objet B à l'instant Tn et l'algorithme d'extrapolation est mis en route. Lorsque la différence, entre la position réelle et la position extrapolée, excède le seuil, un nouveau message contenant la position réelle et la nouvelle vitesse sont envoyés sur le réseau et l'algorithme est remis en route avec ces nouveaux paramètres.

Lors de la réception d'un message de mise à jour dû à un dépassement de seuil, le simulateur a deux possibilités : il peut modifier brusquement la position de l'objet (ceci n'est acceptable que si le seuil est très faible) ou il peut modifier petit à petit la position du véhicule jusqu'à la bonne position (position extrapolée à partir des nouveaux paramètres).

Une autre technique intéressante développée par SIMNET est la technique dite des SAF (*Semi-Automated Forces*). Cette technique permet à un seul opérateur humain de diriger les actions d'une centaine d'objets dans le monde virtuel. Ceci permet de simuler des confrontations importantes de troupes à un coût réduit. Ceci peut être fait en donnant des ordres de haut niveau comme : « dirigez-vous vers un endroit géographique particulier en déterminant vous-même la route » ou un peu plus explicite comme : « faites feu sur cette cible ». De plus, les SAF peuvent réagir de façon autonome à certaines situations (par exemple ils peuvent riposter lorsqu'ils sont attaqués par des véhicules ennemis). L'intérêt de cette technique est que l'on ne peut pas distinguer, a priori, les véhicules SAF des véhicules pilotés par des êtres humains.

SIMNET comprend trois classes de protocoles : un protocole de simulation qui transmet les informations entre les différents simulateurs, un protocole de collecte de données pour la gestion de la simulation et un protocole d'association qui fournit des services de niveau transport et session (selon la terminologie du modèle OSI) au-dessus d'Ethernet. Le protocole de simulation est constitué d'un ensemble d'unités de données de protocole (PDU) qui transportent des informations d'état ou des événements. Par exemple, le PDU apparence d'un

véhicule contient la position, l'orientation et le statut d'un véhicule. Les PDUs feu, feu indirect, collision et impact correspondent, quant à eux, aux événements les plus communs sur un champ de bataille.

1.3.1.4. Conclusions sur cette architecture

Le succès de SIMNET est essentiellement dû aux différentes techniques originales mises en œuvre pour gérer au mieux les capacités machines et réseau. La première de ces techniques est le *dead-reckoning* que nous venons de décrire et qui réduit l'utilisation de la bande passante du réseau de façon substantielle au prix d'un surcoût en traitement (en général faible) sur chaque site. Une autre technique tient à l'existence d'une valeur booléenne à l'intérieur du PDU apparence d'un véhicule pour indiquer qu'un véhicule est à l'arrêt. Ce qui permet de ne pas lui appliquer l'algorithme du *dead-reckoning* et ainsi d'économiser la puissance des machines. Enfin, le protocole d'association exploite les capacités de communication par groupe (*multicasting*) d'Ethernet en associant chaque exercice militaire à une adresse multicast Ethernet. Ainsi, plusieurs exercices peuvent coexister sur un même Ethernet sans que l'application ait à traiter elle-même le flux réseau (l'interface Ethernet pouvant distinguer les trames Ethernet intéressant la machine).

Néanmoins, SIMNET présente de nombreux inconvénients liés à ses avantages. Par exemple, l'utilisation du *multicasting* Ethernet implique que SIMNET est lié à cette technologie et est difficilement portable sur d'autres types de réseaux (Token ring, ATM, FDDI…). D'autre part, les PDU sont très spécifiques à l'application (simulation militaire de char d'assaut). Pour étendre SIMNET à d'autres types de simulation il est donc nécessaire de définir de nouveaux PDU ce qui peut poser problème au niveau de la standardisation des simulateurs et donc de leurs capacités à interopérer.

D'autre part, les algorithmes de *dead-reckoning* définis par SIMNET sont hélas beaucoup moins efficaces lorsqu'on les applique à des entités non militaires (véhicules civils en agglomérations, êtres humains au comportement en général imprévisible…). En effet les entités militaires terrestres modernes (principales entités de SIMNET) se déplacent en petits groupes disciplinés (colonnes de chars par exemple) et conservent la même allure et la même direction pendant de longues périodes.

1.3.2. DIS

Le standard DIS essaie de pallier les inconvénients de SIMNET en proposant des PDUs plus nombreux et plus génériques et en se démarquant d'un protocole de trop bas niveau comme Ethernet. Le but de ce standard est de permettre à plusieurs simulateurs (plates-formes logicielles et matérielles) développés par des entreprises différentes d'interopérer. Ainsi un environnement virtuel à large échelle peut être développé plus efficacement et plus rapidement. Nous allons présenter dans cette partie DIS, ces points communs et divergences avec SIMNET, ces avantages et ces inconvénients.

1.3.2.1. Les unités de données de protocole

DIS reprend les principaux types de PDU de SIMNET. DIS comprend 27 PDUs qui sont définis par la norme IEEE 1278. Seuls trois PDUs servent à l'interaction des entités, les autres servent à transmettre des informations sur les opérations de support (par exemple lorsqu'un véhicule reçoit du carburant ou des munitions), à gérer les émanations électroniques (ondes radar) et à contrôler la simulation. Le PDU état de l'entité (*Entity State PDU* : ESPDU) est utilisé pour transmettre l'état courant d'un véhicule (position, orientation, vitesse et apparence). Le PDU tir (*Fire PDU*) est émis lorsqu'une arme tire et contient des données sur le type de munitions, la position et l'orientation initiale du tir… Le PDU détonation est émis lorsque des munitions ou des entités explosent.

1.3.2.2. Principe de fonctionnement

Le fonctionnement d'un simulateur conforme au standard DIS repose sur le paradigme « joueurs et fantômes » (*players and ghosts*). Ce paradigme indique que chaque objet de la simulation est géré sur sa propre station de travail hôte par un objet logiciel appelé un Joueur, alors que sur toutes les autres stations participant à la simulation une version simplifiée du Joueur est modélisée dynamiquement par un objet logiciel appelé Fantôme.

Les objets Fantômes mettent à jour leur position à chaque itération de la boucle de simulation grâce à un algorithme de *dead-reckoning*. Ce principe permet d'économiser énormément la bande passante sans trop pénaliser le réalisme de la simulation. En effet il a été prouvé que même avec un seuil d'erreur de 3 mètres sur la position, des algorithmes de *dead-*

reckoning du premier ordre (ne travaillant que sur la vitesse) étaient suffisants pour modéliser un avion de combat moderne F16 avec un gain de 66% au niveau du trafic réseau (les mises à jours de l'entité passant d'une fréquence de 50 Hz à 17 Hz) [Macedonia 95b].

DIS a de plus hérité de deux autres concepts de SIMNET qui influencent la bande passante et la fiabilité. Tout d'abord, chaque entité est présumée « honnête » et donc il n'y a aucune négociation entre les entités à propos de leur état (mis à part lors d'opération de réapprovisionnement). Par exemple lorsqu'un véhicule tire, c'est son entité qui détermine qui a été touché et communique cette information à toutes les autres entités. L'entité touchée calcule ensuite les dégâts sur son véhicule et communique cette nouvelle information à toutes les autres entités. Ainsi un seul calcul balistique est effectué pour toute la simulation et une seule entité calcule les dégâts.

D'autre part, il n'y a pas de base de données ou de serveur central. Ainsi une entité doit déterminer l'état de son environnement grâce aux PDUs d'états des autres entités. Ceci implique qu'une entité doit régulièrement (toutes les 5 secondes d'après la norme) émettre un ESPDU pour permettre aux entités entrant dans la simulation de connaître son état.

1.3.2.3. Conclusions sur DIS

Tout comme SIMNET, DIS est limité du fait même de sa spécificité aux simulations militaires de taille moyenne (quelques centaines d'entités). En effet, les seules interactions possibles sont, ici aussi, très orientées militaire (PDU feu et PDU détonation).

L'extensibilité du protocole aux simulations de grande taille (plusieurs centaines de milliers d'entités) est limitée. Dans [Macedonia 95b] Macedonia liste un nombre important de problèmes qui limitent DIS dans ce sens, nous présentons ici les principaux :

- La consommation de la bande passante réseau et des ressources de calcul locales sont prohibitives. Une simulation comprenant 100 000 utilisateurs a besoin au maximum (dans le pire cas) d'une bande passante de 375 mégabits par seconde vers chaque ordinateur participant à la simulation, ce qui est vraiment énorme compte tenu de la technologie actuelle. D'autre part, le maintien de l'état des entités distantes grâce à des algorithmes de *dead-reckoning* nécessite d'importantes ressources de calcul sur chaque ordinateur de la simulation.

- DIS impose que le multiplexage/démultiplexage de différents médias (PDU de la simulation, audio et vidéo) soit réalisé par la couche application et non par les couches réseau ou transport (en terminologie OSI). Les applications de simulation utilisant DIS sont d'autant plus délicates à concevoir et mettre en œuvre. Par exemple, les paquets de données représentant un signal audio doivent être lus à des périodes fixes qui ne correspondent pas nécessairement avec la boucle de simulation. De plus ces paquets sont de taille importante et retardent donc le traitement des PDUs de la simulation.

- La gestion des objets statiques n'est pas efficace. Un grand nombre d'objets statiques comme des ponts ou des bâtiments peuvent être modifiés à la suite d'un événement (une explosion par exemple). Ceux-ci ainsi que les objets devenus immobiles doivent envoyer des messages à intervalle régulier pour informer les simulateurs de leur état actuel. Par exemple, un char d'assaut qui a été détruit doit constamment informer les autres simulateurs qu'il est mort de façon à ce que les nouveaux simulateurs (ainsi que ceux qui auraient ratés tous les autres messages) sachent que ce char est détruit. Ainsi de nombreux messages quasi inutiles encombrent le réseau.

- Tous les modèles et toute la base de données représentant le monde simulé doivent être répliqués dans chaque simulateur. Il n'y a pas de mécanisme dans DIS qui permette une distribution des objets à la demande. Pour des simulations à large échelle cela devient nécessaire surtout si les simulateurs en réseau appartiennent à de nombreuses organisations qui ne peuvent toutes se concerter pour s'échanger les modèles physiques et graphiques des véhicules qui peuvent prendre part à une simulation. D'autre part, il n'est pas possible ni efficace que tous les simulateurs doivent posséder chaque modèle et base de données d'une simulation comprenant 100 000 entités. De toute façon, le simulateur d'un soldat à terre n'est, par exemple, normalement pas intéressé par les mouvements de satellites ni ceux de porte-avions qui croisent au large.

1.3.3. NPSNET

NPSNET (*Naval Postgraduate School's Network vehicle simulator*) peut être vu comme une plate-forme de recherche visant à faire évoluer le standard DIS. Il existe de nombreuses plates-formes de ce type que l'on qualifie globalement de systèmes DIS++ (par analogie avec C++ vu comme le « successeur » du C). NPSNET est la plate-forme de ce type qui a été la

plus décrite dans la littérature scientifique. La version actuelle de NPSNET (NPSNET-IV) est donc basée essentiellement sur DIS dont elle diverge sur un certain nombre de points que nous allons présenter dans cette partie.

NPSNET est développé au département d'informatique de la *Naval Postgraduate School* (NPS) à Monterey (Californie). NPSNET-IV est unique en simulation militaire distribuée car il incorpore toutes les techniques suivantes dans un simulateur visuel opérationnel :

- Le standard *Distributed Interactive Simulation* (DIS version 2.0.3) pour permettre des communications avec d'autres simulateurs (comme des simulateurs de vols et des véhicules réels dotés de capteurs).
- Le *multicasting* IP, le standard d'Internet pour la communication par groupes qui permet de gérer des simulations distribuées à grande échelle.
- Un parallélisme de haut niveau qui permet de créer des pipelines applicatifs.

NPSNET-IV peut être configuré par l'utilisateur comme un simulateur de véhicules terrestres, marins (de surface ou submersibles), aériens ou encore d'êtres humains (soldats). L'utilisateur contrôle son véhicule en utilisant divers périphériques comme un « manche à balai », une Spaceball (périphérique à six degrés de libertés) ou encore un ensemble clavier et souris. D'autres véhicules contrôlés par d'autres utilisateurs sur des stations de travail distantes peuvent participer à la simulation. Les utilisateurs peuvent être des participants humains réels, des entités autonomes gérées par un ensemble de règles (comme les forces semi-automatisées introduites par SIMNET) ou des entités rejouant un script (déplacements préalablement enregistrés provenant d'un exercice réel ou simulé). Enfin, des objets statiques et dynamiques dotés de propriétés multimédias peuvent exister dans l'environnement.

1.3.3.1. Gestion des PDUs

Un processus asynchrone scrute continuellement le réseau pour récupérer l'information qui contient les PDUs reçus par le simulateur. Le filtrage qui permet de ne récupérer que les PDUs d'une simulation particulière est réalisé par ce processus. Les PDUs de la simulation courante sont passés à l'application via un tampon alloué en mémoire partagée.

La boucle principale de l'application consiste à lire un PDU (depuis la file en mémoire partagée) et à le traiter suivant son type. Dans la version actuelle de NPSNET, seul les PDUs

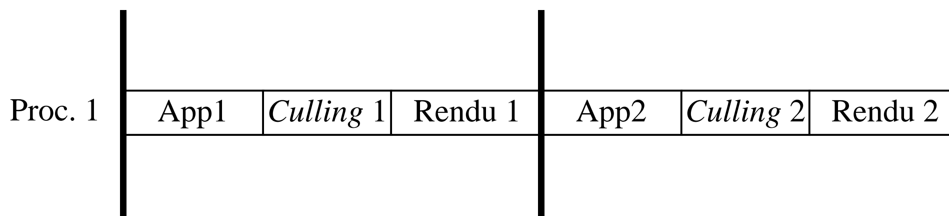
DIS de type Etat d'Entité, Feu et Détonation sont traités. Après son traitement éventuel, la mémoire utilisée par le DPU est libérée.

A chaque itération de la boucle principale, chaque objet entité actif (aussi bien local que distant) invoque son opération `moveDR` (déplacement utilisant un algorithme de *dead-reckoning*). Ensuite, les entités distantes se déplacent. L'entité locale, se déplace en fonction des ordres donnés par l'utilisateur et lorsque son modèle extrapolé (par le *dead-reckoning*) n'est plus valide (à un seuil près) ou lorsqu'une collision est intervenue, un PDU Etat d'Entité est émis.

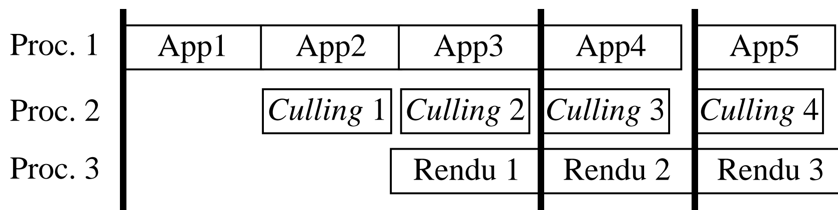
La réception d'un PDU Etat d'Entité indique donc que l'état d'une entité distante, qui en accord avec la philosophie de DIS est gérée localement par la technique du *dead-reckoning*, a changé. Le traitement des PDUs Feu et Explosion déclenche divers effets visuels ou sonores comme l'apparition de cratères sur le terrain, des fumées et des bruits d'explosion ainsi que l'éventuelle destruction d'entités qui en avertissent les autres simulateurs grâce à des PDUs Etat d'Entité.

1.3.3.2. Le parallélisme des traitements dans NPSNET-IV

Pour arriver à conserver une latence globale du système la plus basse possible tout en assurant un rafraîchissement égal en moyenne à 10 images par secondes, NPSNET-IV tire partie de l'architecture multiprocesseur des calculateurs Onyx de SGI dotés de cartes graphiques comme le Reality Engine2 ainsi que de la librairie graphique IRIS Performer.



Modèle à un seul processeur



Modèle à trois processeurs

Figure 8 : Le pipeline applicatif à trois processus

Le système utilise un pipeline applicatif hérité de la librairie graphique IRIS Performer [Rohlf 94] qui décompose une application de simulation en trois processus (Figure 8) : l'application proprement dite qui gère la simulation, un processus d'élagage (*culling*) qui permet d'éviter l'envoi de la base de données complète à la carte graphique, un processus de rendu qui envoie les ordres graphiques de bas niveau qui permettent d'afficher (idéalement) la partie visible de la base de données graphique. Pendant que le processus de rendu envoie les ordres permettant de calculer l'image N, le processus d'élagage travaille sur les données qui seront affichées à l'image N+1 et l'application réalise des modifications de la base de données graphique qui seront visibles dans l'image N+2. En associant chacun de ces processus à un processeur unique (dans un calculateur Onyx à quatre processeurs par exemple) on obtient un véritable pipeline qui permet de réaliser de nombreuses opérations en parallèle et augmente la vitesse de rafraîchissement (nombre d'images par seconde).

NPSNET-IV rajoute à ce pipeline un quatrième processus (qui s'exécute idéalement sur le quatrième processeur d'une Onyx) pour gérer les données venant du réseau (Figure 9). Ceci est important car pour une simulation comprenant un grand nombre d'entités, la gestion du réseau nécessite d'importantes ressources.

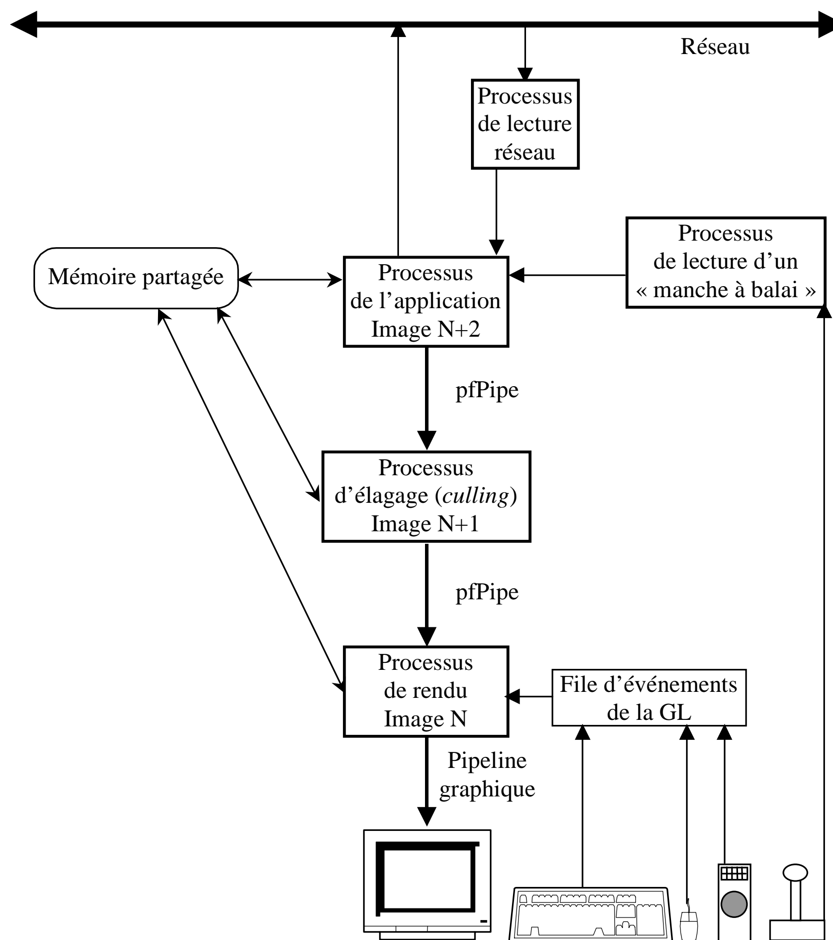


Figure 9 : Structure des processus et flot de données dans NPSNET-IV

Il y a deux avantages à placer le processus de lecture réseau sur son propre processeur : d'une part, la gestion du réseau ne pénalise ni l'application, ni l'élagage, ni le rendu, d'autre part, ce processus récupère immédiatement les PDUs quand ils sont offerts par la carte réseau ce qui permet de réduire la probabilité qu'ils soient rejetés dans les couches basses de la gestion réseau à cause de tampons pleins. Par exemple, pour une simulation affichée à la vitesse de 10 Hz et un rythme d'arrivée de 1000 PDUs par secondes, environ 15 kilooctets de données doivent être mis dans le tampon par le processus de lecture réseau pendant que le processus application réalise un cycle de sa boucle principale (mise à jour de la base de données graphique, gestion des algorithmes d'extrapolation, gestions des commandes utilisateurs…) [Macedonia 95b].

Cependant il y a des inconvénients à cette approche. Tout d'abord, la réalisation du système est spécifique aux machines de SGI car elle utilise des appels systèmes spécifiques à IRIX (système d'exploitation des stations Silicon Graphics) et la librairie Performer qui

n'existe que sur ces machines. Ensuite, le processus réseau oblige à réaliser une recopie d'information qui aurait pu être évitée. Enfin, dans le cas d'une machine à un seul processeur, la latence globale du système (entre le moment où l'on reçoit un PDU et où la modification est visible) augmente à cause des différents étages du pipeline. Ces inconvénients semblent cependant être plus que compensés par l'accélération globale du système dans le cas d'une station multiprocesseur. Ce qui implique que NPSNET-IV est un système qui reste très coûteux financièrement pour les organismes qui l'utilisent (même si ce coût semble très inférieur à celui d'un simulateur militaire dédié).

1.3.3.3. Evolutions vers la communication par groupes

Pour tenter de résoudre les problèmes du protocole DIS dans le cas de simulations de grande taille (plusieurs milliers d'entités), le groupe de recherche de NPSNET a proposé une approche originale utilisant de nombreux groupes de communications IP [Macedonia 95a].

Cette approche s'appuie sur le fait, observé depuis longtemps dans de nombreuses études [Benford 93] [Kazman 93d] [Torguet 93], que les entités ont une zone d'interaction limitée. Par exemple, dans le cadre militaire, un char d'assaut sur un champ de bataille ne peut agir sur et observer d'autres entités que si celles-ci sont situées dans un rayon de 4 km. Cette zone d'interaction est appelée zone d'intérêt (AOI : *Area Of Interest*) par les auteurs de NPSNET.

Dans l'approche proposée, les entités dont les zones d'intérêt s'interpénètrent sont membres d'une même **classe spatiale** dans l'environnement virtuel et chaque classe spatiale est associée à un groupe de communications. Ainsi les entités d'une même classe peuvent échanger leurs PDUs en « privé » sur leur groupe de communications. Les groupes de communications sont associés aux classes spatiales de la façon suivante : le terrain est découpé en un ensemble de cellules hexagonales et chaque cellule est associée à une adresse *multicast*. Un véhicule est associé à sept cellules (en gris sur la figure) qui représentent sa zone d'intérêt (Figure 10) et donc à sept groupes de communications. La machine de l'entité écoute les sept groupes de communications mais n'émet de PDUs que dans le groupe associé à la cellule dans laquelle le véhicule est situé.

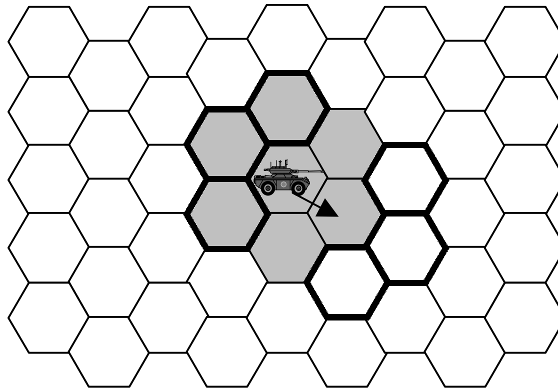


Figure 10 : Zone d'intérêt d'un véhicule exprimée par un ensemble de cellules

L'intérêt d'utiliser des hexagones est que, grâce à leur structure d'agencement régulière, un véhicule qui se déplace d'une cellule à une autre ajoute et enlève le même nombre de cellules (et donc de groupes de communications) (par exemple, dans la Figure 10 le véhicule va enlever les trois cellules grisées entourées de noir et ajouter à sa zone d'intérêt les trois cellules blanches entourées de noir). De plus, la zone d'intérêt d'un véhicule étant en général définie par un cercle elle s'associe naturellement à un ensemble d'hexagones.

Le modèle introduit en plus deux autres critères pour établir d'autres groupes de communications indépendants du terrain.

Les entités peuvent appartenir à une même **classe fonctionnelle**. Par exemple, toutes les entités qui sont intéressées par les événements aériens et par les entités qui volent peuvent faire partie d'un même groupe de « contrôle aérien ». Cette classe est intéressante car les entités aériennes sont typiquement en faible nombre par rapport aux entités terrestres dans une simulation militaire. Les quelques avions qui sont intéressés par les unités terrestres peuvent, de plus, focaliser leur attention sur une zone du terrain en devenant temporairement membre du groupe de communications associé.

Les entités peuvent aussi appartenir à une même **classe temporelle**. Ainsi, certaines entités qui n'ont pas besoin de connaître les modifications de l'état d'autres entités en temps réel, peuvent faire partie d'un groupe de communications dans lequel les entités représentant des véhicules émettront leurs PDU Etat d'Entité à une fréquence très faible (de l'ordre d'une fois par minute ou par heure). Un exemple d'entité de ce type peut être un simulateur de satellite « espion » qui observe toute une opération militaire et qui peut focaliser son attention (selon le

besoin exprimé par son opérateur) sur une zone particulière en devenant temporairement membre du (des) groupe(s) de communication associé(s).

Les changements de groupes sont de plus utilisés pour mettre à jour la base de données représentant l'environnement de façon à éliminer l'envoi régulier de PDU Etat d'Entité. Ceci est fait de façon distribuée en utilisant, simplement, « l'âge » d'une entité dans son groupe. Les entités sont soit passives, soit actives dans leur groupe. Les entités passives ne font qu'écouter le groupe alors que les entités actives envoient en plus des PDUs au groupe.

Quand une entité rejoint un nouveau groupe son site note sa date d'entrée dans le groupe et envoie un PDU Requête de Connexion (*Join Request PDU*) au groupe. Ce PDU contient un booléen indiquant que l'entité va être passive dans le groupe. Le site de l'entité chef de groupe (il s'agit de l'entité active la plus ancienne dans le groupe) répond par un PDU Pointeur (*Pointer PDU*) qui référence la requête de connexion et contient un pointeur vers le site d'une entité active. Le site qui rejoint le groupe envoie ensuite un PDU Requête de Données (*Data Request PDU*) directement au site référencé par le PDU Pointeur. Ce site lui renvoie un PDU Données (*Data PDU*) qui contient les PDUs Etat d'Entité concernant toutes les entités actives du groupe agrégées dans le même message. Ces deux derniers PDU sont envoyés sur une connexion point à point fiabilisée (cette communication fiabilisée permet d'éviter les messages envoyés à intervalle réguliers par les entités statiques – cf. la partie 1.3.2.3). Lorsque l'entité devient active le site envoie un nouveau PDU Requête de Connexion avec le booléen d'activité positionné à vrai. Enfin, les retraits de groupes sont annoncés par des PDUs Requête de Déconnexion (*Leave Request PDU*).

1.3.3.4. Conclusions sur NPSNET

Le système reste très lié à son actuelle application (la simulation militaire). Ceci est dû essentiellement aux types d'interaction entre entités qui sont disponibles et à la non-extensibilité vers d'autres types d'interaction sans reconsidérer en grande partie le système (cf. le traitement des PDU). D'autre part, comme tout système dérivé de DIS, NPSNET n'est réellement efficace que pour la gestion de véhicules terrestres dont les changements de comportements (accélération, décélération, changements de direction) sont peu fréquents (cf. *dead-reckoning*). En comparaison, la gestion d'un être humain entraîne de nombreux problèmes. Dans l'article [Pratt 94] qui décrit l'introduction de la gestion réaliste d'un être

humain dans NPSNET, les auteurs nous indiquent que la complexité des mouvements humains implique que le système doit envoyer des informations à chaque itération de la boucle de simulation et ainsi, un seul être humain génère autant de trafic réseau que cinq ou huit chars d'assaut ou trois avions performants.

L'évolution utilisant les zones d'intérêt est très intéressante mais reste, de l'aveu même de son concepteur, très liée aux simulations militaires et dépend énormément de la vitesse de déplacement des entités. En effet, étant donné la latence qui existe lorsqu'un site rejoint un groupe, il est important de ne pas changer trop souvent de groupes. Les paramètres qui influent sur ces changements sont la taille des cellules et la vitesse des entités. Plus la vitesse des véhicules est importante plus ils ont besoin de changer de groupes. Inversement, plus les cellules sont grandes, moins les véhicules changent de cellules. La taille des cellules permet, donc, de compenser la vitesse des véhicules. Cependant, si les cellules sont trop grandes il y a un risque de se retrouver avec un trop grand nombre de véhicules dans le même groupe. En conclusion on peut dire que ce mécanisme n'est applicable de façon réellement efficace qu'aux véhicules terrestres qui se déplacent lentement.

1.4. Systèmes génériques à architecture client-serveur

1.4.1. dVS

La société DIVISION commercialise un système de réalité virtuelle distribuée, appelé dVS, depuis 1990. La première version de ce système a été décrite en détail lors d'une communication [Grimsdale 90] et dans un article de la revue BYTE [Pountain 91]. Depuis cette version initiale, l'architecture a un peu évoluée mais les concepts de base de cette architecture n'ont pas changés. Nous allons décrire dans cette partie la version 3.0 de dVS qui est décrite dans la documentation technique du système [DIVISION] et dans l'ouvrage [Burdea 93].

1.4.1.1. Description générale du système

Le système dVS (*distributed Virtual environment System*) est un modèle parallèle fondé sur la notion d'**acteur**. Les acteurs sont des processus qui se partagent une base de données. Les

acteurs peuvent être distribués sur les différentes machines participant à la simulation d'un environnement virtuel. Plusieurs acteurs ont été développés pour gérer les différentes tâches de la simulation d'un environnement virtuel : le rendu, la synthèse de son, les capteurs de données (gant de données, capteurs de position…), les collisions entre objets, l'application (définition du comportement des objets virtuels)… (Figure 11). Chaque acteur dispose d'une copie partielle de la base de données et un acteur spécial appelé **directeur** assure la cohérence entre les différentes copies.

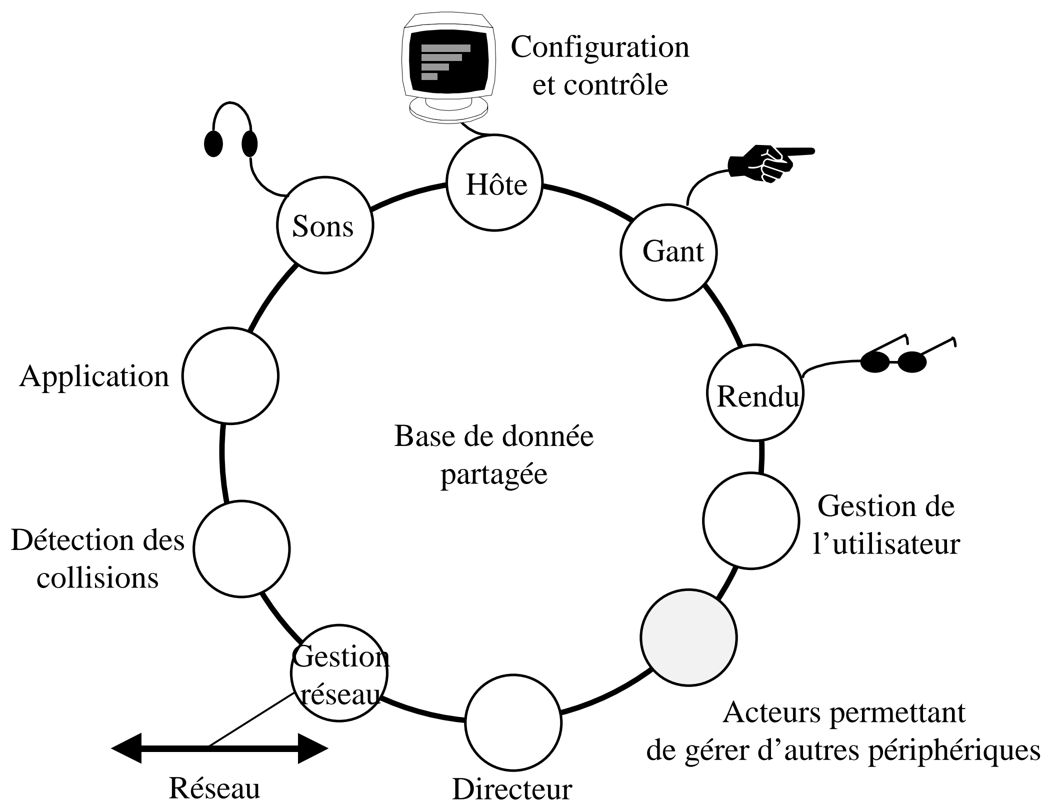


Figure 11 : Les acteurs de dVS

1.4.1.2. Définition de l'environnement

La base de données représentant l'environnement virtuel est une collection d'**objets** possédant des attributs appelés **éléments** (position, couleur, texture...). Cette base de données est appelée l'**environnement**. Les objets peuvent être groupés et organisés de façon hiérarchique. D'autre part, tous les objets sont créés par des acteurs.

Les éléments sont considérés comme des types abstraits. On peut citer les éléments prédéfinis suivants : *Visual* (géométrie de l'objet), *Position* (position et orientation de l'objet),

Audio (fichier son associé à un objet), *Boundary* (boîte englobante servant à détecter les collisions) et *Dynamics* (contrainte de mouvement pour l'objet : gravité, force de friction…). En fait, un objet est défini, à un instant donné, par les **instances** des éléments qui le composent (c'est à dire les valeurs de ces éléments). De nouveaux éléments spécifiques à une application peuvent être créés via un pré-processeur d'élément (*epp*) qui génère, à partir d'une définition en langage C, des structures de donnée au format de dVS et des fonctions d'accès.

Les acteurs peuvent être conceptuellement classés en trois types : les capteurs, les contrôleurs et les afficheurs. Les **capteurs** ont pour rôle la récupération de données en provenance du monde réel (des acteurs *VCINPUT* existent pour chacun des périphériques standards : gant, capteur de positions…). Ces données sont placées dans l'environnement sous la forme d'éléments qui peuvent être récupérés par d'autres acteurs. Les **contrôleurs** ont pour rôle de gérer le comportement des objets et leurs interactions (exemple : l'acteur *VCCOLLIDE* surveille en permanence l'environnement pour détecter les collisions). Ce sont ces acteurs qui récupèrent les éléments placés par les capteurs. Ainsi, l'acteur *VCBODY* gère tout ce qui concerne un utilisateur (récupération d'informations fournies par des acteurs *VCINPUT*, modification des éléments *Position* des objets correspondants à la main et à la tête d'un utilisateur virtuel…). Les **afficheurs** sont chargés de fournir un retour sensoriel aux utilisateurs (visuel grâce à l'acteur *VCVISUAL* et sonore grâce à l'acteur *VCAUDIO*).

1.4.1.3. Gestion de la cohérence

Les acteurs communiquent entre eux de façon indirecte via la base de données partagée (l'environnement). Ainsi, certains acteurs seront intéressés par certains éléments particuliers. L'acteur *VCVISUAL*, qui s'occupe du rendu visuel, surveillera par exemple les éléments *Visual* et *Position* de l'environnement. Chaque acteur qui a besoin d'une instance d'un élément en demande une copie au directeur. Si un acteur veut modifier les caractéristiques d'une instance d'élément dont il possède une copie, il envoie une requête de modification au directeur qui ensuite propage la modification aux autres acteurs qui possèdent eux aussi une copie de cette instance.

Cette technique de gestion de la cohérence est étendue dans le cas d'une distribution sur plusieurs machines reliées par un réseau. Dans ce cas, sur chaque site, un acteur spécial nommé **agent** gère la propagation des modifications aux sites intéressés. Les agents se

communiquent les diverses modifications d'instances selon le besoin respectif des acteurs de leur machine. Ce besoin est géré localement sur chaque machine par le directeur. Sur l'un des sites, le directeur est défini comme le **maître** de la simulation. Il gère les autres directeurs et l'ensemble des instances. Un directeur **esclave** ne pourra modifier une instance que si il en a reçu l'autorisation du directeur maître. Ainsi le directeur maître joue le rôle d'un **serveur** pour les autres directeurs et indirectement pour tous les acteurs.

La figure suivante (Figure 12) présente deux environnements (existants sur deux machines distinctes) qui ont des instances d'éléments en commun. Lors d'une modification d'une instance, l'agent de la machine où l'événement s'est produit va propager ou non la modification suivant les informations fournies par le directeur local.

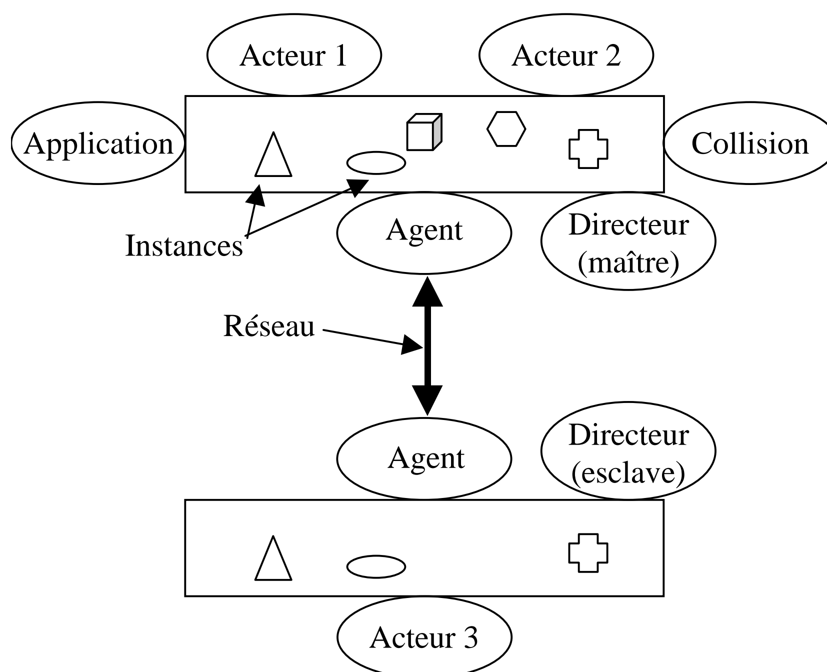


Figure 12 : Architecture de dVS en réseau

Il est important de noter que les acteurs application (*DVISE*) et collision (*VCCOLLIDE*) sont centralisés sur une seule machine.

1.4.1.4. Conclusions sur cette architecture

Le problème principal de cette architecture est la centralisation des acteurs qui gèrent l'application et la détection des collisions ainsi que le rôle du directeur maître. L'acteur application doit gérer les comportements de tous les objets de l'environnement. Ce qui limite

donc dVS à la gestion d'environnements simples. D'autre part, l'acteur collision doit recevoir toutes les modifications des positions et orientations des objets virtuels pour pouvoir détecter leurs collisions. Le directeur maître peut lui aussi poser des problèmes si à chaque modification les directeurs esclaves doivent lui demander une autorisation. Or un système de réalité virtuelle multi-utilisateur est par nature très dynamique, et donc ces modifications sont très nombreuses, beaucoup sont simultanées et se renouvellent très souvent, noyant, a priori, ces acteurs sous un très grand nombre de requêtes.

Ces problèmes sont typiques des architectures qui ont été construites pour gérer un système mono-utilisateur (c'est aussi le cas de MR Toolkit [Shaw 92] et de VUE [Appino 92] [Codella 92]) et qui pour cette raison ne sont pas extensibles simplement au-delà de trois ou quatre utilisateurs.

Une solution pour résoudre ces problèmes est le filtrage des messages basé sur les limites spatiales des utilisateurs. En effet, les messages provenants d'un utilisateur n'intéressent pas en général les utilisateurs qui sont trop éloignés dans le monde virtuel. Ce type de filtrage a été mis en œuvre dans les systèmes que nous décrivons dans la suite de cette partie.

1.4.2. WAVES

Cette partie va présenter les travaux de recherches de Rick Kazman commencés à l'Université Carnegie Mellon (Etats-Unis) [Kazman93a] sous le nom de HIDRA, (*Highly Interactive Distributed Real-time Architecture*) et poursuivis à l'Université de Waterloo (Canada) sous le nom de WAVES (*WAterloo Virtual Environment System*) [Kazman 93b,c,d].

L'architecture de WAVES a été conçue pour résoudre un certain nombre de problèmes présents dans les systèmes précédents comme l'extensibilité à des environnements gérants de nombreux utilisateurs, la simplification de la distribution, l'indépendance vis à vis de la plate-forme d'exécution et la séparation de certains traitements dans des processus spécialisés.

1.4.2.1. Les « objoids »

Pour WAVES un environnement virtuel est composé d'objets autonomes qui sont appelés objoids (le terme d'objet ne plaisant pas aux concepteurs du système à cause de sa trop forte connotation venant de la programmation orientée objet). Les objoids encapsulent leur état,

possèdent un modèle explicite de leur comportement et peuvent simuler ce comportement de façon autonome en temps réel. Idéalement, le modèle comportemental doit pouvoir servir à prédire l'état de l'objoid dans un futur plus ou moins proche. Cette capacité étant utilisée par WAVES comme un algorithme de *dead-reckoning* amélioré.

1.4.2.2. L'architecture logicielle de WAVES

L'architecture est composée d'un certain nombre d'**hôtes** connectés via un ou plusieurs **gestionnaires de messages** (Figure 13). Les hôtes sont des processus qui gèrent chacun une partie de la simulation d'un monde virtuel. Chaque gestionnaire de messages gère la communication entre différents hôtes.

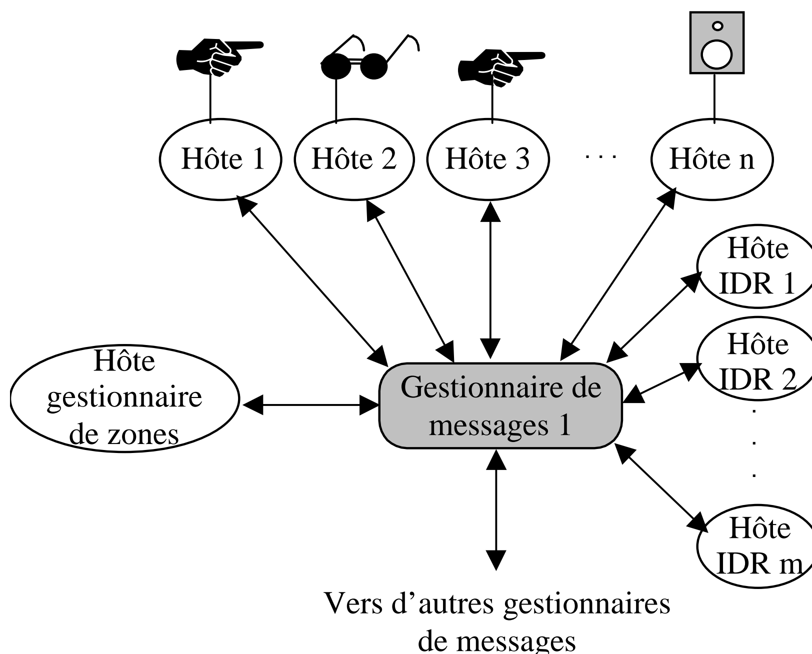


Figure 13 : L'architecture logicielle de WAVES

Les gestionnaires de messages permettent d'éviter à chaque hôte de devoir connaître chacun des autres hôtes. Ceci permet de réaliser des environnements virtuels distribués flexibles dans lesquels des hôtes peuvent entrer et sortir dynamiquement. En plus de gérer les communications entre les hôtes, un gestionnaire de messages peut filtrer les messages qu'un hôte particulier reçoit. Ceci permet de minimiser globalement le volume de communication dans un environnement virtuel complexe. Pour profiter de cette fonctionnalité, un hôte doit envoyer un ensemble de filtres à son gestionnaire de messages.

Un gestionnaire de messages gérant au plus 64 hôtes, si le système comprend un nombre plus important d’hôtes ou si le gestionnaire de messages est submergé, d’autres gestionnaires peuvent être introduits dynamiquement. Un hôte communique avec un seul gestionnaire à la fois et ce sont les gestionnaires qui communiquent entre eux. Une communication directe entre deux hôtes peut être déléguée par un gestionnaire lorsque le volume d’échange entre ces deux hôtes devient trop important ou s’ils en font la demande.

La plupart des hôtes gèrent la description physique (présentation) et comportementale (simulation) des objoids ainsi que leur création et leur destruction. Toutes les entrées/sorties avec le monde réel passent par ces hôtes. En effet ils peuvent gérer des périphériques spécialisés (gants, casques…).

La gestion des interactions est réalisée dans des hôtes spécialisés : les hôtes IDR (*Interaction Detection and Resolution*). La détection des interactions dans WAVES peut être définie de deux façons :

- Comme la gestion de l'utilisation d'une ressource (par exemple l'espace physique). Pour cela chaque hôte IDR contient une carte du « théâtre d'opérations » qui stocke la position de chaque objet et lève une exception si deux objoids tentent d'occuper la même place.
- Comme une contrainte sémantique sur les attributs des objoids. Pour ce faire chaque hôte IDR contient un système de production de règles. Ce qui donne la possibilité, au concepteur de l’environnement, de définir des règles modélisant les contraintes sur l'état d'un objet.

Une fois détectée, une interaction peut être résolue par certaines règles du système de production de règles ou par un mécanisme de résolution générique utilisant la cinématique.

Les hôtes et le gestionnaire maintiennent chacun un cache qui contient les objoids qui leurs sont utiles à un moment donné. Les objoids placés dans un cache sont aussi appelés des clones. Grâce au modèle de comportement qui est présent dans chaque objet, les hôtes n'ont besoin de communiquer entre eux que lors d'une modification du comportement d'un objoid. Lors d’une telle modification, le gestionnaire envoie cette modification à tous les hôtes qui ont le clone à modifier dans leur cache (cette modification peut transiter par un ou plusieurs gestionnaires de messages suivant la topologie courante de WAVES).

De plus, le système permet de faire migrer les objoids d'un hôte trop chargé vers un hôte moins chargé. Cette fonctionnalité est rendue plus aisée par l'encapsulation de l'état et du comportement dans les objoids. Une utilisation intéressante de cette migration qui n'est qu'effleurée dans l'article [Kazman 93b] consiste à tenter de rassembler les objoids proches dans l'environnement sur le même hôte.

1.4.2.3. Gestion de l'espace

En partant de l'idée que dans des environnements virtuels très complexes de larges portions de l'environnement n'intéressent pas un utilisateur donné à un instant donné, le concepteur de WAVES a décidé d'inclure au système des mécanismes permettant de filtrer ces informations inutiles. Par exemple, un utilisateur présent dans un bâtiment architectural ne sera intéressé que par la pièce dans laquelle il se trouve et par les pièces adjacentes reliées par des portes ou des fenêtres.

Comme pour la détection des interactions, le concepteur de WAVES a décidé de placer ce mécanisme dans des processus spécifiques en l'intégrant dans des hôtes appelés **gestionnaires de zones**. En général il existe un gestionnaire de zones par gestionnaire de messages. Si l'environnement devient trop important pour être géré par un seul gestionnaire de messages ou par un seul gestionnaire de zones alors il y aura duplication des deux gestionnaires s'il existe suffisamment de ressources libres dans le réseau géré par WAVES.

Lors de son initialisation ou à n'importe quel autre moment pendant son exécution, un gestionnaire de zones va recevoir les spécifications d'un ensemble de zones découpant l'environnement virtuel, de leurs relations entre elles et de leur contenu. Le gestionnaire de zones va alors construire un graphe de visibilité qui va contenir les relations de visibilité directe entre les zones. Toutes les zones et leur contenu sont des objoids. Parmi les objoids présents dans les zones certains correspondent à des points de vue associés à des utilisateurs et le but du gestionnaire de zones est de calculer pour chacun de ces **objoids points de vue** quelles sont les zones visibles. Les zones ainsi calculées deviennent par la suite des filtres fournis au gestionnaire de messages pour limiter le volume de messages reçu par l'hôte qui gère l'objoid point de vue.

1.4.2.4. Conclusions sur cette architecture

Le système WAVES gère les objets du monde virtuel un peu comme dVS. Un objet peut exister dans plusieurs hôtes (soit sous sa forme originale soit dans un cache d'objets). Le gestionnaire de messages s'occupe du maintien de la cohérence entre toutes les manifestations de cet objet. Néanmoins grâce au modèle de comportement inclus dans chaque objet les manifestations d'un objet ne sont incohérentes que lors d'un changement de comportement. De plus, les filtres placés dans les gestionnaires de messages (et en particulier ceux fournis par les gestionnaires de zones) limitent les échanges. C'est un avantage par rapport au système dVS dans lequel les incohérences apparaissent dès qu'une instance d’élément est modifiée et les messages de mise à jour envoyés à tous les acteurs qui sont intéressés par les instances de ce type.

Malgré cela le système peut avoir les mêmes problèmes que celui de Division si les comportements des objets se modifient souvent ou s’ils sont indéterministes comme c'est le cas pour celui d'un objet géré par un utilisateur. Il existe en fait des systèmes de prédiction pour prédire les mouvements de la tête d'un utilisateur à partir des positions précédentes (Filtrage de Kalman) mais ces prédictions sont de courte durée [Liang 91].

Les idées qui sous-tendent les gestionnaires de zones ont été reprises par un système plus récent (WAVES étant apparemment abandonné à l’heure actuelle) appelé RING. Ce système est basé sur une architecture client-serveur développée au laboratoire AT&T Bell de Murray Hill (New Jersey) [Funkhouser 95]. La nouveauté majeure de ce système, dans sa toute dernière version, est que les zones sont distribuées par régions aux serveurs (qui jouent le double rôle des gestionnaires de message et des gestionnaires de zones de WAVES) et que les clients (les hôtes de WAVES) changent de serveurs quand les entités qu’ils gèrent changent de régions [Funkhouser 96]. L’intérêt de ce mécanisme est de limiter les échanges entre les serveurs. Cependant le serveur gérant la zone d’un client pouvant a priori se trouver n’importe où sur le réseau, la latence peut être très importante ce qui n’est pas le cas dans un système comme WAVES (ou la première version de RING) où le serveur choisi est généralement le plus proche (en terme de latence moyenne).

1.4.3. MASSIVE

MASSIVE (*Model, Architecture and System for Spatial Interaction in Virtual Environments*) [Greenhalgh 94] [Greenhalgh 95], conçu à l'Université de Nottingham, utilise une architecture mixte entre client-serveur et égal à égal. Cependant, la partie la plus originale de ce système, le modèle spatial d'interaction est réalisé par des connexions client-serveur c'est pourquoi nous avons classé ce système dans la partie 1.4.

1.4.3.1. Le modèle spatial d'interaction

Avant de décrire MASSIVE nous allons décrire le modèle logique sur lequel il est fondé. Ce modèle a été développé au sein du projet COMIC (projet européen ESPRIT sur le travail coopératif) [Benford 93].

Le concept de base du modèle est la notion d'**espace**. Les espaces virtuels peuvent avoir n'importe quel nombre de dimensions du moment qu'une mesure de distance est possible. L'espace est habité par des **objets** qui peuvent représenter des utilisateurs, de l'information ou des artefacts informatiques (outils immergés dans l'environnement virtuel).

Les objets interagissent via plusieurs **médias** d'interaction. Le modèle gère ces interactions grâce à deux composantes ayant chacune un objectif différent : faciliter l'extensibilité du système en terme de nombre maximal d'utilisateurs et contrôler les interactions spatiales entre les différents objets présents dans l'environnement virtuel.

La première composante est fondée sur le concept d'**aura**. Chaque objet présent dans un monde virtuel possède une aura pour chacun de ses médias d'interaction (graphique, audio, texte…). Ces auras définissent la zone dans laquelle une interaction avec un autre objet est possible. Cette zone est exprimée par une fonction mathématique ayant pour paramètre la position et éventuellement d'autres attributs de l'objet. L'interaction entre deux objets n'est possible que lorsque leurs auras indiquent chacune la possibilité d'une interaction. Cet état de fait est appelé dans le modèle une collision d'auras. Ainsi lorsqu'un objet pénètre dans un monde virtuel pour la première fois il ne possède aucune connaissance sur les autres objets du monde. Il doit déclarer ses auras à un gestionnaire d'auras qui teste les collisions entre les auras d'un même type et notifie les objets concernés. Lors de cette notification, le gestionnaire donne suffisamment d'information pour que les objets concernés puissent communiquer entre

eux directement pour gérer l’interaction. Dans la plupart des applications les auras représentent une région fermée de l’espace. L’utilisation d’auras facilite l’extensibilité du système à de nombreux utilisateurs en limitant le nombre d’interactions à gérer à chaque instant (et sur chaque site).

La seconde composante contrôle les interactions entre deux objets qui ont été mis en contact à la suite d’une collision de leurs auras respectives. Le principal concept dans ce cadre est le concept de **conscience** (*awareness*). La conscience d’un objet envers un autre objet quantifie l’importance subjective de l’autre objet par rapport à un type de communication. En général, le système prêtera d’autant plus attention (et donc offrira plus de bande passante et de ressources de calcul) à l’interaction entre deux objets que leur conscience respective est importante.

Ce concept est géré via deux sous-concepts : le **focus** et le **nimbus** qui permettent à deux objets de modifier leur niveau de conscience mutuelle. Le focus permet à un objet (l’observateur) de diriger son attention sur une (ou plusieurs) zone(s) précise(s) ou sur un (ou plusieurs) type(s) d’objet précis. Le nimbus permet à un objet (observé) de gérer sa « visibilité » ou son accessibilité de façon à ce que certains objets puissent avoir un plus haut niveau de conscience de lui-même. Le niveau de conscience d’un objet observateur envers un objet observé dépend donc du focus de l’observateur et du nimbus de l’objet observé.

Les auras, les foci et les nimbi peuvent être des fonctions multi-valuées, peuvent avoir une étendue et une forme précise et sont spécifiques à un type de communication (ce sont des sortes de champs de potentiel). La figure suivante (Figure 14) illustre ces différents concepts pour un seul type de communication et pour un seul objet observé et un seul observateur.

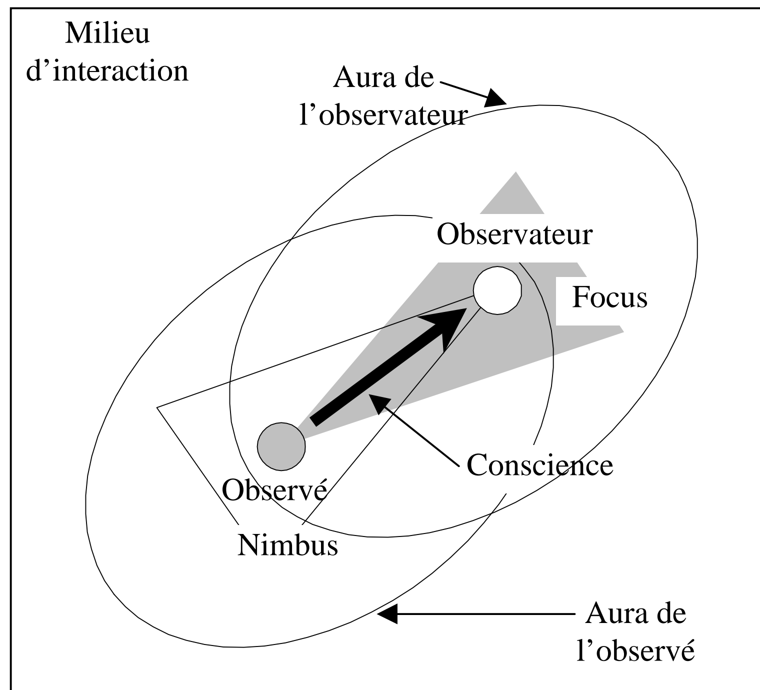


Figure 14 : Auras, foci, nimbi et conscience pour deux objets

Les auras, le focus et le nimbus (et donc la conscience) peuvent être manipulés de trois façons. De manière **implicite** via des actions spatiales comme des déplacements ou des rotations. Par exemple, quand un objet tourne sur lui-même son aura, son focus et son nimbus tournent de la même manière ce qui modifie son niveau de conscience des autres objets et le niveau de conscience que les autres ont de lui. De manière **explicite** en choisissant parmi différentes formes et étendues. Par exemple, en passant d'un focus large à un focus resserré. Enfin, des **objets adaptateurs** transforment les auras, foci et nimbi d'autres objets. Ces objets peuvent représenter des outils de communication comme un podium, un porte-voix ou encore une table de conférence. Par exemple, un objet adaptateur représentant un podium augmente l'aura et le nimbus de son utilisateur (quand sa représentation est placée au-dessus de la forme graphique du podium). Un autre type d'objet adaptateur peut correspondre à des frontières qui segmentent un monde virtuel en diverses régions et qui contrôlent comment la conscience les traverse. Par exemple, un objet adaptateur de ce type pourra spécifier que le son émis dans une pièce (dont l'adaptateur est la frontière) ne peut en sortir.

1.4.3.2. Fonctionnalités apportées par MASSIVE

Après avoir présenté les concepts qui sous-tendent le système nous allons présenter les fonctionnalités apportées par la dernière version de MASSIVE.

MASSIVE gère plusieurs mondes virtuels connectés par des « portails ». Chaque monde définit un espace virtuel infini et disjoint peuplé par plusieurs utilisateurs qui peuvent interagir via une combinaison de graphiques, de sons et de textes. Les portails permettent aux utilisateurs de passer d’un monde à un autre (via une sorte de téléportation).

L’interface graphique affiche les objets visibles dans une vue 3D du monde et permet aux utilisateurs de se déplacer dans l’espace 3D. L’interface sonore permet aux utilisateurs de communiquer en temps réel et d’entendre les fichiers sons qui peuvent être attachés aux objets. L’interface textuelle apporte un plan du monde (vue textuelle de dessus ressemblant à ce qui existe pour certains MUDs) sur lequel sont figurés les utilisateurs (représentés par un caractère) et permet des communications textuelles. MASSIVE permet de combiner ces interfaces de façon arbitraire et en fonction des capacités des équipements informatiques de chaque utilisateur (Figure 15). MASSIVE permet ainsi à un utilisateur ne disposant que d’un terminal ASCII de participer à une session de réalité virtuelle en même temps qu’un utilisateur doté d’une station de travail multimédia qui pourra, lui, utiliser les trois interfaces.

Pour permettre des interactions entre des interfaces de type différent, un utilisateur ne disposant que de l’interface textuelle peut exporter une forme graphique, une aura et un nimbus le représentant dans le média graphique (même s’il ne peut pas s’y voir lui-même). De même, un utilisateur disposant de l’interface graphique peut exporter une forme textuelle, une aura et un nimbus le représentant dans le média textuel. En d’autres termes, les utilisateurs textuels peuvent être incarnés dans le média graphique et les utilisateurs graphiques peuvent être incarnés dans le média textuel.

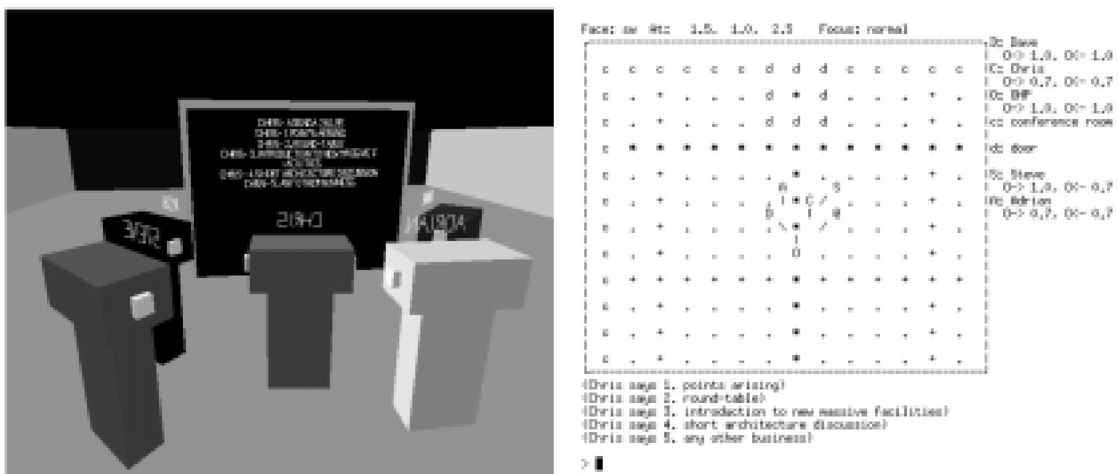


Figure 15 : Exemples de vues graphiques et textuelles de MASSIVE

Le modèle spatial régit les interactions entre les interfaces d'un même type de différents utilisateurs. En effet, les interactions d'un type donné ne sont pas possibles tant qu'il n'y a pas eu de collision d'auras de ce type. Ainsi, un objet ne peut être visible tant que son aura graphique n'est pas entrée en collision avec une autre aura graphique. Les interactions sont ensuite contrôlées par le niveau de conscience et donc par un focus et un nimbus. Par exemple, le niveau de conscience sonore donne le volume sonore du canal de communication et est dépendant de la distance et des orientations respectives des utilisateurs concernés (en fait des objets qui les représentent).

Dans la version actuelle de MASSIVE, les utilisateurs peuvent choisir parmi trois réglages pour leurs foci et nimbi : normal, resserré (utilisé lors de communications privées à deux utilisateurs) et large (utilisé pour rechercher d'autres utilisateurs/objets dans l'environnement). De plus, quatre objets adaptateurs sont disponibles : le podium que nous avons décrit précédemment, une table de conférence qui remplace les auras, foci et nimbi de ses utilisateurs par de nouvelles instances qui englobent uniquement la table (ceci permet des discussions privées autour de la table), un traducteur « texte vers parole » qui utilise la synthèse vocale pour traduire des messages du média textuel en des messages sonores et un traducteur « texte vers graphique » qui affiche des messages du média textuel sur un tableau noir présent dans le média graphique (Figure 15). Ces adaptateurs sont eux aussi gérés par le modèle spatial de sorte qu'ils ne deviennent actifs que lorsqu'un utilisateur est suffisamment proche d'eux. Par exemple, lorsqu'un utilisateur textuel s'approche suffisamment près d'un traducteur (au point que leurs auras s'interpénètrent) ce dernier devient actif et commence à traduire automatiquement les messages d'un média vers un autre. De plus, plusieurs utilisateurs peuvent utiliser le traducteur (ou tout autre adaptateur) simultanément.

MASSIVE qui fonctionne sur des stations SUN et SGI est utilisé en continu par le DEVRL (*Distributed Extensible Virtual Reality Laboratory*) un laboratoire virtuel étendu sur trois sites (l'université de Nottingham, de Lancaster et les collèges Queen Mary et West-field de Londres) qui sert de banc d'essai pour la RVD en Angleterre. Plusieurs rencontres de groupes comportant jusqu'à huit utilisateurs ont été réalisées avec succès en utilisant le réseau grande distance SuperJANET.

1.4.3.3. Architecture logicielle de MASSIVE

L'architecture de MASSIVE est fondé sur trois éléments importants :

- Une architecture de communication utilisant des connexions en mode point à point typées. Ces connexions utilisent une combinaison de RPC (*Remote Procedure Call*), d'attributs partagés (et synchronisés) et de flux multimédias. Ces connexions sont établies dynamiquement et existent aussi bien entre des clients et des serveurs qu'entre deux clients. C'est pour cela que l'architecture est mixte client-serveur et égal à égal.

- Des serveurs qui mettent en contact les interfaces de leurs clients lorsque des conditions spatiales sont atteintes (collision d'auras plus spécifiquement).

- Des interactions dynamiques entre les clients utilisant différents médias.

Chaque objet dispose d'un certain nombre d'interfaces qui lui permettent de communiquer avec le reste du système. Ces interfaces correspondent aux différents médias à travers lesquels l'objet peut interagir avec d'autres objets. Les objets ne peuvent pas directement connecter leurs interfaces pour échanger des informations. L'interaction entre deux objets n'est possible que si deux conditions sont satisfaites. Premièrement, les deux objets doivent posséder des interfaces compatibles. Deuxièmement, les objets doivent être suffisamment près l'un de l'autre de façon à ce que leurs auras entrent en collision. Ces deux pré-conditions définissent ce que les auteurs de MASSIVE appellent le ***trading* spatial**. Ce concept combine la détection de collisions, qui est une des techniques de la réalité virtuelle, avec le concept de *trading*, qui est issu des systèmes distribués. Ce dernier concept vient de systèmes comme ODP (*Open Distributed Processing*) [ODP 95] ou ANSA (*Advanced Network Systems Architecture*) [ANSA 89] qui définissent un *trader* comme un serveur permettant à des objets de publier les services qu'ils proposent et de découvrir des services proposés par d'autres objets. En fait, un *trader* peut être vu comme un serveur qui permet d'obtenir le nom d'un ou plusieurs objets (en fait une référence vers ces objets) à partir d'un service qu'ils offrent (et qu'ils ont préalablement publié auprès du *trader*). Pour expliquer le *trading* spatial nous allons voir ce qu'il se passe lorsque deux objets pénètrent dans le même monde, se déplacent l'un vers l'autre et commencent à interagir. Les différentes étapes de ce scénario sont présentées sur la figure suivante (Figure 16) et décrites dans les paragraphes suivants.

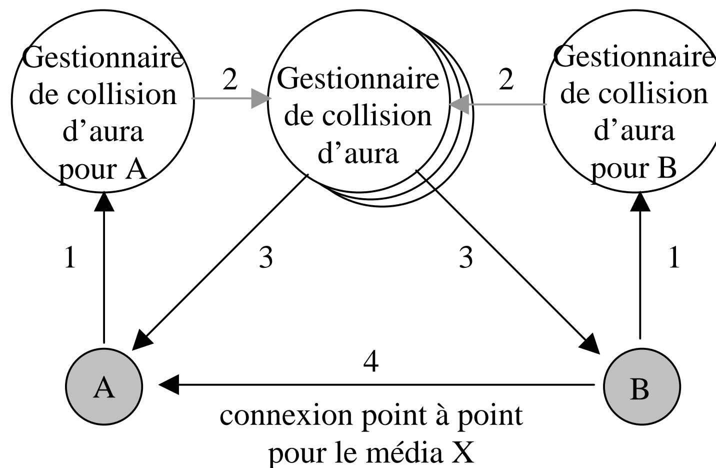


Figure 16 :le *trading* spatial

Lorsqu'un objet entre dans un monde virtuel, il contacte le gestionnaire de collision d'aura local en précisant le monde dans lequel il veut pénétrer et les médias de communication que l'objet gère (étape 1). Pour pouvoir entrer dans un monde virtuel local ou dans un monde accessible depuis un monde local (via des portails), l'objet n'a besoin de connaître initialement que l'adresse de son gestionnaire de collision d'aura local. Un gestionnaire de collision d'aura est chargé de détecter les collisions entre aura pour chaque média d'un ou plusieurs monde. Chaque gestionnaire a une liste partielle des autres gestionnaires d'aura et des mondes qu'ils gèrent qui est configurée localement. Ainsi, un gestionnaire peut confier la gestion d'un objet à un autre gestionnaire lorsque l'objet change de monde (étape 2). Lorsqu'un deuxième objet entre dans le monde, il suit les mêmes étapes que le premier objet et il est donc géré par le même gestionnaire d'auras.

Lorsqu'un gestionnaire détecte la collision de deux auras, il envoie à chaque objet l'adresse de l'autre objet concerné (étape 3). Ainsi les objets peuvent établir une connexion pour échanger des informations (étape 4).

Une fois connectés, le calcul du niveau respectif de conscience est laissé à la charge des deux objets. Ce calcul est réalisé en suivant un protocole simple qui permet à deux objets quelconques d'échanger des informations décrivant leurs positions, orientations, foci et nimbi. Le protocole de communication pour chaque média (graphique, son ou texte) est dérivé de ce protocole simple de façon à gérer des informations spécifiques à chaque média.

Les foci et nimbi utilisés par MASSIVE sont des fonctions mathématiques qui prennent leurs valeurs dans l'intervalle [0..1]. La fonction permettant d'obtenir la conscience à partir du

focus et du nimbus est multiplicative. C’est à dire que les foci et nimbi sont simplement multipliés pour obtenir la conscience. Cela donne un niveau de contrôle équivalent à l’observateur et à l’objet observé, permet à chaque intervenant de forcer un niveau de conscience nul et leur interdit de forcer un niveau de conscience totale contre le souhait de l’autre objet.

1.4.3.4. Conclusions sur MASSIVE

Ce système permet à une dizaine d'utilisateurs d'interagir grâce à des interfaces graphiques, sonores ou textuels (ou combinaison). Ceci est dû au concept d’aura qui limite les interactions qui peuvent se produire dans l’environnement virtuel. Cette fonctionnalité est similaire à celle apportée par le gestionnaire de zones de WAVES mais plus générale (elle n’est pas limitée à la vision).

Le problème de ce système est que la gestion des collisions d’auras par des serveurs augmente la latence lors d’interactions entre deux entités (cela dit, cette augmentation de latence est moins pénalisante que dans un système comme WAVES pour lequel tous les messages transitent par le serveur). D’autre part, l’omniprésence du protocole TCP au cœur de l’architecture de communication implique aussi que le niveau de dynamisme du système doit être réduit (le temps de connexion d’une liaison point à point TCP est relativement important).

MASSIVE est, de plus, limité dans le type d’application qu’il peut gérer car il ne permet pas de manipulations directes des objets virtuels par les avatars. Il n’y a, d’autre part, aucun mécanisme permettant de définir aisément des comportements pour les différents objets (apparemment les comportements doivent être définis directement dans le système) ce qui limite aussi énormément les types d’application supportés par ce système. Les applications ciblées par MASSIVE sont, en effet, des conférences multi-utilisateurs utilisant les outils de communication pré-définis (table de conférences, podium et traducteurs).

1.4.4. VLNET

VLNET (Virtual Life Network) est un système développé conjointement par les équipes de recherches des laboratoires MIRALab de l'Université de Genève et le Laboratoire d'Informatique Graphique de l'Ecole Polytechnique Fédérale de Lausanne en Suisse. Les

auteurs de VLNET ont choisi une approche différente de celle suivie par la grande majorité des architectes des autres systèmes que nous présentons dans ce chapitre. En effet, l'originalité de ce système est de simuler des humains virtuels sophistiqués permettant des mouvements d'articulations réalistes en accord avec l'anatomie humaine, des déformations de la peau et l'animation du visage [Capin 97].

Gérer de nombreux êtres humains virtuels aussi complexes dans le cadre d'un environnement virtuel distribué multi-utilisateur est une tâche complexe à développer et à réaliser en temps-réel. C'est pourquoi, VLNET offre une architecture modulaire dont les nombreux modules peuvent être gérés, par exemple, par les différents processeurs d'un multiprocesseur.

VLNET possède une architecture client-serveur avec, pour l'instant, un seul serveur. Nous allons tout d'abord détailler l'architecture du serveur puis nous présenterons l'architecture du client [Pandzic 97].

1.4.4.1. L'architecture du serveur

Un site serveur pour VLNET est composé initialement d'un processus serveur HTTP (*HyperText Transfer Protocol*) et d'un processus serveur de connexion. Ces deux serveurs peuvent gérer plusieurs mondes virtuels qui sont définis soit dans des fichiers spécifiques à VLNET soit dans des fichiers au format VRML 1.0 (*Virtual Reality Modeling Language*).

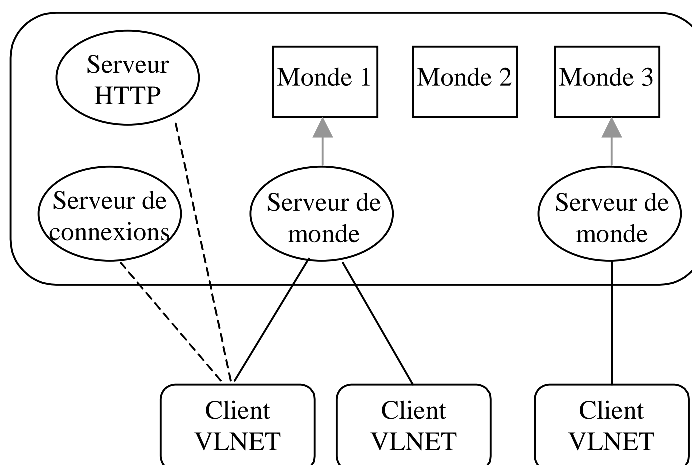


Figure 17 : Un site serveur de VLNET avec plusieurs clients connectés

Quand un client veut se connecter à un monde particulier désigné par son URL, c'est le serveur HTTP qui fourni le fichier décrivant le monde. Une fois le fichier récupéré, le client communique avec le serveur de connexion. S'il n'existait pas déjà, un processus serveur de monde est créé sur ce site serveur (Figure 17). Puis, dans tous les cas, le numéro de port correspondant au serveur de monde est envoyé au client. Une fois la connexion établie avec le serveur de monde, toutes les communications entre les clients d'un même monde transitent par le serveur de monde. Les clients peuvent se déconnecter à tout instant. Enfin, lorsqu'il n'y a plus aucun client connecté au monde qu'il gère, le serveur de monde se détruit.

Pour réduire le trafic sur le réseau, le serveur de monde filtre les messages échangés entre les clients en tenant compte de la pyramide de vision dans le monde virtuel de l'utilisateur de chaque client.

1.4.4.2. L'architecture d'un client

L'architecture d'un client est composée de nombreux processus qui communiquent via plusieurs zones mémoires partagées. L'architecture est extensible grâce à un certain nombre d'interfaces qui permettent à un programmeur de développer des modules, appelés pilotes, communiquant avec le cœur d'un client à travers ces interfaces (Figure 18).

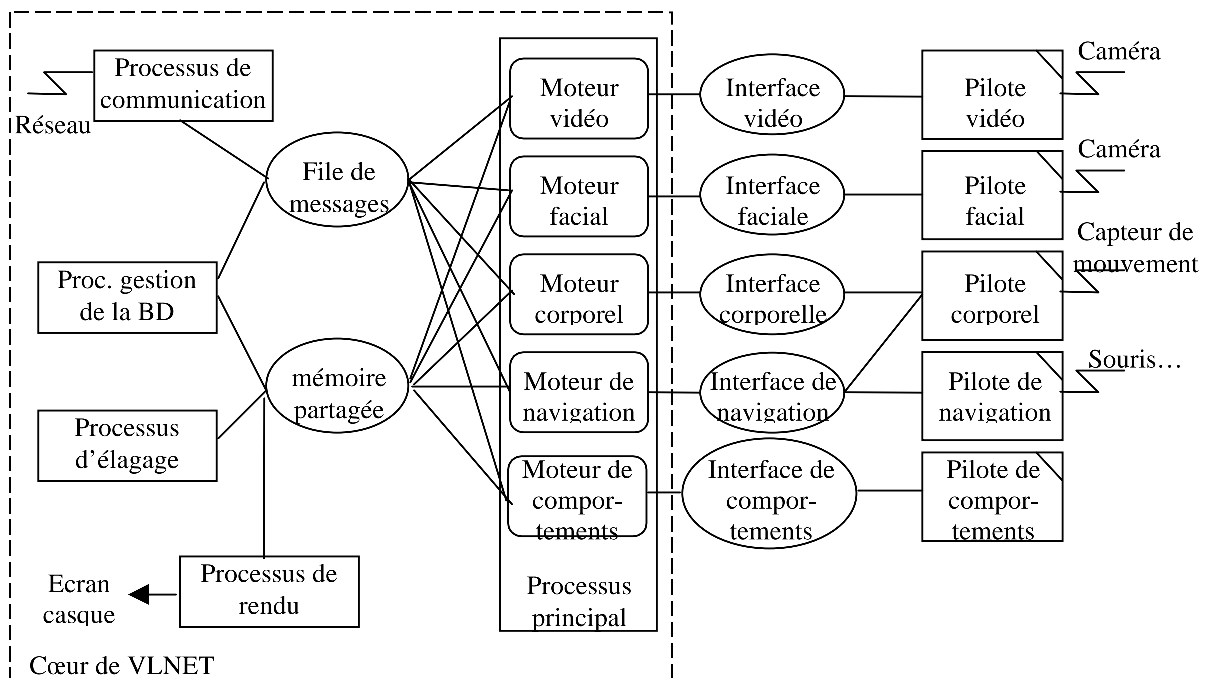


Figure 18 : Architecture d'un client VLNET

Le cœur d'un client est un ensemble de cinq processus qui mettent en œuvre les fonctions essentielles de VLNET. Le processus principal gère des tâches de haut niveau comme la manipulation d'objets et les déformations des humains virtuels. Le processus de communication gère les interactions avec le réseau. Le processus de la base de données gère le chargement de la scène représentant le monde. Enfin, les processus de rendu et d'élagage gèrent l'affichage de la scène. Ces deux derniers processus sont des processus standards de la librairie graphique Performer de SGI.

Le processus principal crée les autres processus. Il comprend cinq entités logiques appelées moteurs. Chaque moteur est équipé d'une interface qui permet aux pilotes de se connecter. Le moteur de comportements d'objets prend en compte les comportements prédéfinis (comme diverses rotations et la gestion de la pesanteur) et permet la programmation de divers comportements externes grâce à des pilotes. Le moteur de navigation et de manipulation d'objets gère les ordres de base de l'utilisateur (déplacements dans le monde virtuel, saisie et déplacements d'objets...). Il gère l'interface avec un pilote de manipulation et propose un pilote par défaut qui utilise la souris si aucun pilote n'est actif. Le moteur de représentation de corps gère les déformations de chaque corps en fonction de sa posture courante. Il gère une interface qui permet de changer la posture d'un corps. Le pilote par défaut utilise l'interface de navigation pour choisir la posture du corps de l'utilisateur. Le moteur de représentation faciale permet la modification de l'expression ou de la texture dynamique d'un visage synthétique. L'interface d'expression faciale est utilisée dans ce cadre. Enfin, le moteur vidéo permet d'associer une texture dynamique, correspondant à un flux vidéo, à un objet quelconque. La texture faciale est un cas particulier qui est traité en association avec le moteur précédent. Les moteurs utilisent le résultat du processus d'élagage pour éviter de réaliser des calculs concernant des corps ou des objets non visibles.

Le processus de communication utilise une file de messages qui permet aux autres processus et aux moteurs de placer des messages à envoyer. Le processus envoie ces messages au serveur qui les renvoie après filtrage aux autres clients. Les messages reçus par le processus de communication sont eux aussi placés dans la file de message pour être consommés par les autres processus ou les moteurs. Ce processus permet de gérer des algorithmes de dead-reckoning pour extrapoler les informations contenues dans certains types de messages (les positions des objets, les postures des utilisateurs et les expressions faciales).

Le processus de la base de données permet la récupération et le chargement de la scène et des représentations des utilisateurs. Ces opérations coûteuses sont réalisées par un processus spécifique pour ne pas bloquer l'exécution du système.

1.4.4.3. Extension des clients : les pilotes

Les pilotes permettent le contrôle des fonctionnalités de VLNET. Les pilotes sont connectés directement aux moteurs. Chaque pilote est programmé en utilisant une interface simple qui consiste essentiellement à échanger des données avec un ou plusieurs moteurs par l'intermédiaire de zones mémoires partagées.

La plupart des pilotes sont optionnels et VLNET propose une fonctionnalité par défaut pour les remplacer. Les pilotes sont générés par le processus principal au lancement du client. Ils peuvent être créés sur le site local ou sur un site distant. Dans ce dernier cas, des processus gèrent une interface transparente avec le réseau. Pour chaque moteur il existe un type de pilote.

Un type de pilote intéressant est celui qui permet d'accéder à un interpréteur de L-systèmes. Les systèmes Lindenmayer (L-systèmes en abrégé) sont des systèmes de production de règles prenant en compte le temps qui ont été conçus pour modéliser le comportements d'objets statiques, de plantes et de créatures autonomes. L'interpréteur de L-systèmes à été utilisé pour modéliser les comportements complexes d'humains virtuels autonomes [Noser 96].

1.4.4.4. Les humains virtuels

VLNET utilise pour chaque être humain virtuel un modèle articulé présentant 75 degrés de libertés auxquels s'ajoutent 30 degrés de libertés pour chaque main. Le squelette est représenté par une hiérarchie d'articulations dotées chacune de limites réalistes. Des métaboules (*metaballs*) sont placées, autour du squelette, pour représenter les muscles et la peau. L'utilisation de métaboules permet de recouvrir le corps avec un minimum de primitives. Les métaboules ne sont en fait utilisées que pour la création d'un corps. Il en effet impossible de gérer ces primitives en temps-réel. Les points qui représentent l'enveloppe extérieure des métaboules sont stockés dans un fichier de description du corps.

De nombreux paramètres permettent de définir les postures et les expressions faciales des êtres humains virtuels : les paramètres de positionnement global (positions et orientations de certains points du corps humain exprimées dans le repère global au corps), les paramètres angulaires des articulations, les paramètres des mains et des doigts (ils sont séparés de ceux du reste du corps car ils sont beaucoup plus nombreux), les paramètres faciaux.

Pour contrôler ces divers paramètres trois types de modèles d'humains virtuels ont été créés :

- Les humains virtuels contrôlés directement qui permettent à l'utilisateur de modifier directement les paramètres faciaux et des articulations (grâce à des capteurs de positions par exemple).
- Les humains virtuels guidés par l'utilisateur qui permettent un contrôle au niveau de la tâche à accomplir (par exemple, l'utilisateur peut indiquer à son humain virtuel de marcher ou de s'asseoir).
- Les humains virtuels autonomes qui possèdent un état interne bâti à partir de capteurs et de buts à accomplir. L'utilisateur n'intervient dans ce cas qu'à un très haut niveau en définissant les motivations de l'humain virtuel.

1.4.4.5. Conclusions sur VLNET

Ce système est très intéressant car il se concentre sur aspect peu envisagé par les autres systèmes : les avatars. Pour toutes les applications où le sens de présence (l'utilisateur croit qu'il se trouve réellement dans le monde virtuel) est prédominant, il est clair que les avatars utilisés par les autres systèmes de RVD (main seule, simulacre de corps humain fait de primitives simples, corps humain virtuel statique…) sont insuffisants. Dans ce cadre, VLNET offre les avatars les plus réalistes possibles compte tenu des contraintes temps-réel. Evidemment, une telle complexité a un impact non négligeable sur les traitements et les échanges réseaux nécessaires pour gérer des environnements virtuels multi-utilisateurs. Néanmoins, les auteurs de ce système étudient des algorithmes pour gérer au mieux la bande passante et les ressources de calcul disponibles [Capin 97]. Ils ont même proposé un algorithme dérivé du *dead-reckoning* pour prendre en compte les humains virtuels.

Cependant, actuellement, l'architecture répartie utilisée pose un problème d'extensibilité au niveau du nombre maximal d'utilisateurs que le système peut gérer. Ceci est dû à la gestion

d'un monde virtuel par un seul serveur. Bien que ce serveur gère des filtres spatiaux pour éviter que tous les messages soient envoyés à tous les sites, il doit prendre en compte tous les utilisateurs présents dans un monde virtuel et il reçoit et filtre tous leurs messages. Plusieurs serveurs pourraient être utilisés pour répartir la charge de l'ensemble des utilisateurs à l'exemple de ce qui est mis en œuvre par WAVES. Une autre solution mise en œuvre par les systèmes que nous présentons dans la partie suivante, consiste à utiliser une architecture égal à égal et des groupes de communications. Cette solution est d'ailleurs envisagée par les architectes de VLNET qui ont isolé la gestion du réseau au niveau du client dans un seul processus de façon à migrer facilement vers d'autres architectures distribuées.

1.5. Systèmes génériques à architecture égal à égal

1.5.1. DIVE

DIVE (*Distributed Interactive Virtual Environment*) est un système de réalité virtuelle distribuée développé par l'institut suédois d'informatique (SICS) [Hagsand 96]. Il a été développé dans le cadre de MultiG, un programme de recherche suédois sur la communication à haute vitesse et les applications réparties.

1.5.1.1. Modèle général de DIVE

DIVE est fondé sur le concept de groupes de processus [Carlsson 93]. Pour DIVE les utilisateurs et les applications sont des processus qui se partagent des **mondes virtuels** (considérés comme des mémoires partagées). Pour simplifier, on peut dire que chaque participant possède une copie locale d'une base de données distribuée dont les modifications sont propagées aux autres participants grâce à des groupes de communications fiables (Figure 19). Ce modèle de distribution est appelé « **réplication active** » par les concepteurs du système.

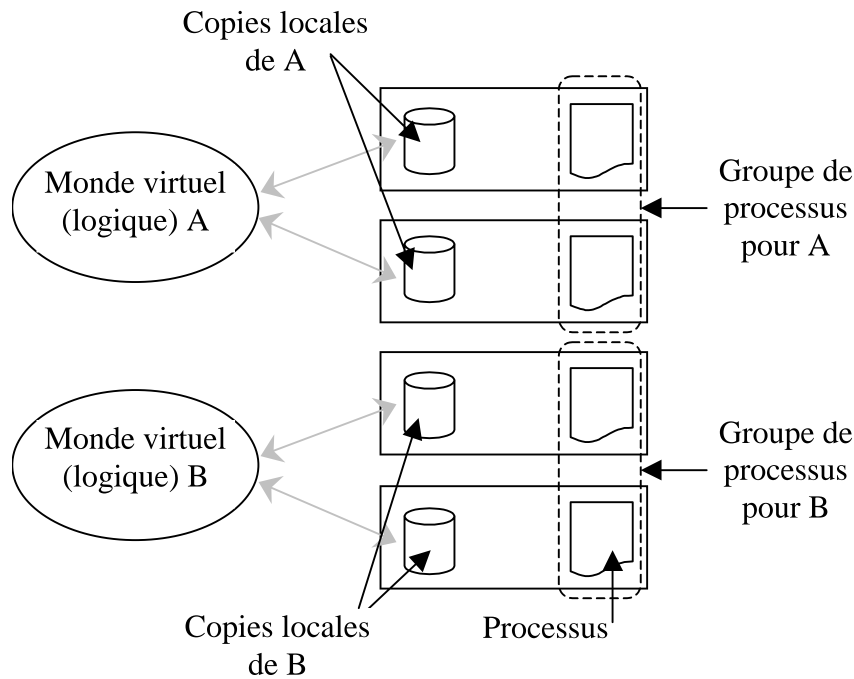


Figure 19 : Architecture logique de DIVE

Un participant représente soit un utilisateur soit une application inclue dans l'environnement virtuel. Les utilisateurs naviguent dans l'espace 3D et peuvent voir, rencontrer et collaborer avec des utilisateurs et des applications. Les utilisateurs sont représentés par des objets graphiques appelés *body-icons*.

Les applications utilisent l'environnement virtuel pour construire leur IHM (interface homme-machine) en créant et en introduisant des objets graphiques dans l'environnement virtuel. Par la suite, le processus de l'application va observer les événements qui se produisent dans le monde virtuel. Quand un événement est reçu, il réagit en accord avec un algorithme prédéfini (réalisé par un programme écrit en langage C par exemple).

Les utilisateurs et applications peuvent se connecter et se déconnecter dynamiquement d'un monde virtuel.

1.5.1.2. Modélisation des mondes virtuels

Les mondes virtuels de DIVE sont composés d'entités. L'entité est l'unité de répartition qui peut être adressée, demandée et envoyée. DIVE utilise une hiérarchie de classes comprenant, à la racine, la classe entité. On peut noter que cette hiérarchie de classes est purement conceptuelle puisque DIVE est écrit en langage C. Une entité possède un identificateur unique

globalement utilisé pour l'adressage, un nom, une description de son comportement et un ensemble de propriétés où sont placées des données spécifiques à une application. Les classes entités et vue (objets graphiques) sont abstraites c'est à dire qu'aucune instanciation de ces classes n'est possible. Les instances d’entités sont structurées hiérarchiquement en arbre. Un monde en est la racine, les nœuds sont les dive_objets et les vues et lumières sont les feuilles.

Les dive_objets comportent la plupart des informations logiques, d'interaction et de dynamique d'un monde. Ceci inclut les matrices de transformations, les descriptions de matériaux et un ensemble de variables qui contrôlent l'interaction et le rendu.

Les vues sont des formes graphiques passives qui peuvent être crées dynamiquement. Lorsqu'un utilisateur interagit avec l'une d'entre elle (en la désignant par exemple), c'est le dive_objet le plus proche dans la hiérarchie qui gère l'interaction.

Les mondes virtuels de DIVE représentent des espaces disjoints comme pour MASSIVE. Un monde comporte des informations qui sont valables pour toutes les entités présentes dans le monde (lumière globale, brouillard et limites spatiales). Un groupe de communications est associé à chaque monde. Seul les sites qui font parti de ce groupe peuvent recevoir les événements qui se produisent dans le monde. Pour connaître les entités qui appartiennent au monde (lorsqu'on vient d'y pénétrer par exemple) il suffit d'en faire la requête sur le groupe de communications.

Un acteur représente un utilisateur ou un processus automatisé (une application inclue dans le monde virtuel). L’acteur est mis en œuvre par une entité liée à un processus (et donc a un site) qui agit dans un monde virtuel. Il modifie les objets et envoie des messages à d'autres entités dans le même monde. Les messages sont soit le résultat de modifications concrètes de la base de données (le monde), comme quand un objet est déplacé, ou représentent des informations plus abstraites telles que : « un utilisateur a saisi un objet ».

Un acteur peut changer de monde en entrant dans des portails (*gateway*) qui sont représentés par des objets graphiques. Lorsque l'acteur traverse l'objet, un gestionnaire de collision signale la collision, un serveur de nom est interrogé pour obtenir l'adresse multicast du monde, l'acteur s'abonne au groupe de communications (et se retire du groupe précédent) et sa représentation graphique est transférée au nouveau monde.

1.5.1.3. Gestion de la cohérence des modifications

Etant donné que les objets peuvent être modifiés par n'importe quel acteur (pas seulement le créateur), d'éventuelles requêtes de modifications simultanées doivent être gérées. DIVE assume que les acteurs « possèdent » les objets pendant « longtemps » et que des requêtes concurrentes se produisent très rarement. Les entités sont donc protégées par un algorithme simple de passage de jeton (il existe un jeton par entité). Si le jeton n’est pas disponible, l'acteur attend de le recevoir pour effectuer la modification.

1.5.1.4. Définition de comportements

Les mondes de DIVE ne sont pas passifs. Les entités peuvent être considérées comme autonomes : elles réagissent à des stimuli, se déplacent, se transforment et s’adaptent aux changements de leur environnement. Ce niveau d’autonomie est réalisé en associant des scripts Tcl [Osterhout 94] aux entités. En général c'est un événement qui déclenche le script, ce qui résulte en l’exécution d’actions sur le monde. Grâce à Tcl qui est interprété et portable, les comportements sont répliqués, sur les différentes machines, avec les entités et peuvent être exécutés sans compilation. Tcl est complété par des fonctions de DIVE le tout formant DIVE/Tcl. L’exemple suivant (quelque peu simplifié) présente un script DIVE/Tcl qui réalise un mouvement simple lorsque l’objet associé est sélectionné :

```
proc move_up {id type actor srcid} {
   for {set i 0}{$i < 100}{incr i}{
        dive_sleep 100
        dive_move $id 0 0.5 0 LOCAL_C
   }
}
dive_register select move_up
```

La procédure `move_up` est associée (grâce à l'appel de la procédure `dive_register`) aux événements `select` qui peuvent se produire sur l'objet qui contient le script. Quand la procédure `move_up` est appelée, les paramètres identifient l'acteur qui a sélectionné l'objet, ainsi que l'objet lui-même. La procédure `dive_move` déplace l'objet de 0.5 mètres dans le repère local toutes les 100 ms. Des comportements complexes déclenchés par des événements ou par des chronomètres peuvent être réalisés grâce aux fonctionnalités de DIVE et aux structures de contrôle de Tcl.

Un autre moyen pour définir des comportements dans un monde virtuel de DIVE est de coder ceux-ci dans une application externe. Dans la précédente version de DIVE, la seule manière pour mettre en œuvre cette méthode était de lier l'application externe avec les librairies de fonctions de DIVE.

L'interface client de DIVE (DCI : *DIVE Client Interface*) [Frécon 95] offre une alternative pour les applications externes écrites dans un autre langage que C/C++ ou pour les systèmes complexes qui nécessitent une séparation nette avec DIVE. Pour simplifier, on peut dire que toute application liée avec les librairies de DIVE (processus qui gèrent des acteurs) peut agir comme un serveur TCP, répondant aux demandes de connexion et offrant les services de DIVE/Tcl à tout client connecté. Les clients peuvent ainsi envoyer des commandes DIVE/Tcl, recevoir des réponses à ces commandes et être prévenus si des événements se produisent (s'ils en ont manifesté le besoin).

1.5.1.5. Gestion des collisions

La gestion des collisions est très importante pour DIVE car elle est à la base de nombreux services comme l'utilisation des portails entre mondes virtuels et la détection des interactions grâce au modèle spatial d'interaction (cf. 1.4.3.1). Comme MASSIVE, DIVE utilise un gestionnaire de collisions. Les processus manifestent leur intérêt pour certains objets et acteurs auprès du gestionnaire de collisions. Par la suite, ce dernier génère des événements de collision (pour les entités qui intéressent au moins un processus) lorsque leurs volumes

s’interpénètrent. Des serveurs mettent en œuvre les gestionnaires de collisions et la gestion des événements de collision.

1.5.1.6. Le modèle de répartition

La première version de DIVE utilisait ISIS pour gérer les groupes de communications et la distribution des données. ISIS est un système qui permet de construire des applications réparties grâce à des protocoles de communication par groupe fiable. Cette version de DIVE était limitée au niveau de l’extensibilité pour deux raisons. L’utilisation d’acquittements positifs par le protocole de communication par groupe fiable (ces acquittements sont utilisés pour déterminer si un paquet a été perdu et s’il doit donc être renvoyé à nouveau) souffre d’un problème appelé « l’explosion des acquittements » (bien qu’un paquet soit émis sur un groupe de communications, les acquittements sont envoyés sur des connections point à point depuis chaque site récepteur ce qui induit une augmentation du nombre de paquets échangés lorsque le nombre de site augmente). D’autre part, DIVE utilisait alors la réplication totale. Dans ce modèle, toutes les modifications d’une copie doivent être envoyées aux autres sites ce qui génère un grand nombre de messages qui de plus doivent être acquittés. Ces problèmes impliquaient une limite de dix sites, sur un réseau local, se partageant le même monde.

La nouvelle version de DIVE utilise une librairie spécifique appelée SID. SID gère des groupes de communications IP *multicast*. Il fournit un service fiable en utilisant des acquittements négatifs (lorsqu’un site s’aperçoit qu’un paquet qui lui était destiné a été perdu il le redemande en envoyant un acquittement négatif). D’autre part, cette version utilise une version simplifiée de *dead-reckoning* (cf. 1.1.2.3). Enfin, un serveur de noms spécifique à DIVE assigne les adresses *multicast* aux différents mondes et gère les recherches d'adresses à partir d'un nom de monde.

1.5.1.7. Conclusions sur cette architecture

DIVE est le système de RVD le plus complet et le plus utilisé à l’heure actuelle. En effet, il est disponible gratuitement ce qui fait de lui un outil très utilisé dans les établissements éducatifs. De plus, il dispose, depuis peu, d’une architecture distribuée extensible qui devrait lui permettre de gérer des environnements partagés par de nombreux utilisateurs. Enfin, il offre de nombreux mécanismes pour développer les comportements des objets virtuels.

Néanmoins, le choix de Tcl comme principal langage de spécification du comportement, semble peu intéressant compte tenu du faible pouvoir expressif de ce langage du à la pauvreté des structures de données (la seule structure évoluée étant la chaîne de caractère). D'autre part, l'algorithme de passage de jeton de DIVE est très critiquable car il empêche toute manipulation coopérative d'un objet. Ce qui est très dommage pour un système multi-utilisateur.

Enfin, le passage de la version de DIVE basée sur ISIS à la version basée sur le *multicasting* d'IP ne s'est pas fait facilement à cause du fort couplage de DIVE avec son système de communication.

1.5.2. AVIARY

AVIARY [West93] est un système conçu à l'Université de Manchester qui gère un environnement virtuel permettant à plusieurs applications et plusieurs utilisateurs d'interagir dans un même monde virtuel. Les applications pouvant être gérées par AVIARY couvrent un vaste domaine allant des simples outils utilisables dans l'environnement jusqu'à de larges activités couvrant la majeure partie de cet environnement.

Nous allons décrire dans la suite de cette partie, le modèle conceptuel introduit par AVIARY et les détails de son architecture distribuée.

1.5.2.1. Modèle conceptuel

Le modèle comprend une hiérarchie de mondes [Snowdon93]. Un monde définit un ensemble d'attributs, que tous les objets existants dans ce monde doivent posséder, et des lois qui agissent sur ces attributs. Les objets simulés, appelés artefacts, sont représentés et gérés par des instances concrètes d'objets informatiques, appelées démons. En fait un artefact est composé d'un démon et d'un objet contenant les attributs spécifiques au monde et les lois qui s'appliquent sur ces attributs. Ce sont les classes de ces derniers objets qui forment la hiérarchie de mondes qui est basée sur une relation d'héritage (au sens programmation orientée objet). Les démons possèdent des attributs et des comportements spécifiques (indépendants du monde dans lequel ils se trouvent).

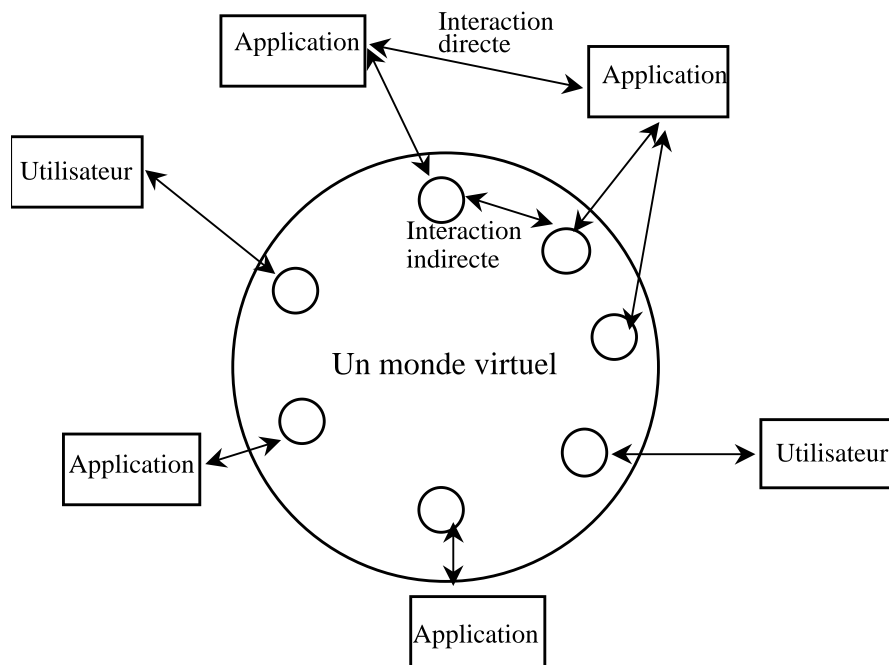


Figure 20 : Architecture logicielle d'AVIARY

Les applications, tout comme les utilisateurs, interagissent avec le monde par l'intermédiaire d'artefacts qui peuvent être vus comme des manifestations de l'application ou de l'utilisateur dans le monde. Plusieurs applications et utilisateurs peuvent être présents dans un même monde (Figure 20).

De plus, plusieurs mondes virtuels peuvent coexister et des « portails » permettent aux artefacts de passer d'un monde à un autre. Il est important de noter que dans ce cas, les attributs et les lois qui sont liées au monde changent. C'est l'objet contenant les attributs du monde qui change lors d'un changement de monde. A ce moment, les attributs communs aux deux mondes sont recopiés et les attributs inexistants dans le précédent monde reçoivent des valeurs par défaut qui pourront être modifiées par le démon. Ainsi, lorsqu'un artefact passe d'un monde qui ne possède pas de lois simulant la gravité à un monde qui la possède il verra un attribut poids ajouté à ses attributs.

1.5.2.2. Architecture distribuée

L'architecture distribuée est constituée de deux types d'objets : les objets lourds (*heavyweight objects*) et les objets légers (*lightweight objects*) [Snowdon 94]. Ces deux termes sont utilisés en analogie avec les processus de poids léger des systèmes d'exploitation. Néanmoins, alors que les objets lourds sont mis en œuvre par des processus du système

d'exploitation d'un site, les objets légers sont implantés par des objets similaires à ceux de langages orientés objets.

L'architecture comprend généralement les objets lourds présentés sur la figure suivante (Figure 21) :

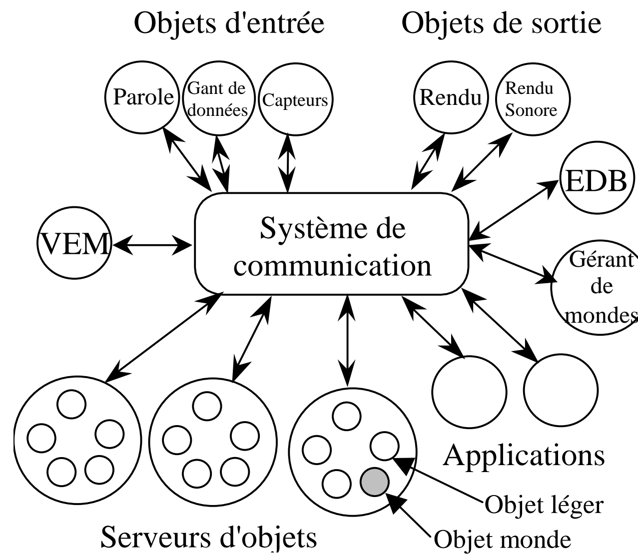


Figure 21 : Architecture distribuée d'AVIARY

Des **objets d'entrée** gèrent les périphériques (par exemple des gants de données et des systèmes de reconnaissance de la parole) et produisent des données pour les différents objets intéressés par ces périphériques. Symétriquement, des **objets de sortie** produisent des sorties pour les utilisateurs (par exemple du graphique et du son) à partir de l'état du monde virtuel qui contient l'utilisateur.

Un objet nommé gestionnaire d'environnement virtuel (**VEM** : *virtual environment manager*) maintient la cohérence de l'environnement virtuel. Une de ses tâches est de générer sur demande un numéro unique pour chaque objet du monde virtuel. Il sert aussi de serveur de nom permettant de déterminer à partir de l'identificateur d'un objet, le site qui le gère.

Un ou plusieurs objets appelés **serveurs d'objets** gèrent les objets légers (création, destruction, exécution des comportements et acheminement des messages) et contiennent un interpréteur d'un langage, appelé langage de contrôle de l'environnement (ECL : *environment control language*). L'ECL sert à décrire et à modifier en cours d'exécution les lois des mondes virtuels et les comportements des objets présents dans l'environnement virtuel. Les serveurs

d'objets peuvent faire migrer des objets légers d'un serveur à un autre. Ainsi ils gèrent l'équilibre des charges du système. Il existe pour chaque monde virtuel un objet léger qui lui correspond et qui est géré par un serveur d'objet. Cet objet, appelé **objet monde**, sert de point de connexion lors de changements de mondes. Il sert aussi à diffuser des messages à tous les objets légers du monde (il suffit pour cela d'envoyer le message à l'objet monde). Les utilisateurs sont, comme les artefacts, gérés par des démons (objets légers). Ces démons sont un peu particuliers car ils communiquent avec des objets d'entrée et avec des objets de sortie.

Un objet appelé base de données de l'environnement virtuel (**EDB** : *environment database*) s'occupe de la détection des collisions grâce à des boîtes englobantes. La détection est centralisée pour éviter à chaque serveur d'interroger tous les autres serveurs à chaque fois qu'un objet est déplacé. Dans ce cas le serveur demande à l'EDB s’il y a collision ; si c'est le cas l'EDB en averti chaque objet concerné en envoyant des messages à leurs serveurs respectifs. Si le premier EDB devient trop surchargé de travail, un autre EDB peut être créé. Si plusieurs EDB existent, l'environnement est découpé en zones confiées chacune à un EDB qui détecte les collisions entre les artefacts présents dans la zone.

Un objet appelé **gérant de mondes** permet de déterminer le site d'un objet lourd qui fournit un service particulier (entrée, sortie, serveur d'objet, EDB...)

Enfin, une ou plusieurs **applications** utilisent l'environnement virtuel comme une interface graphique en présentant leurs résultats et en récupérant des données. Un module permettant la connexion de l'application au système de communication a été écrit sous la forme d'une librairie de fonctions.

1.5.2.3. Réalisation

Le système a été mis en œuvre sur un réseau de stations de travail SUN et sur une machine à 64 Transputers T800 qui possède un réseau d'interconnexion configurable dynamiquement et des capacités d'entrées/sorties.

1.5.2.4. Conclusions sur cette architecture

Ce système est intéressant à plus d'un titre, premièrement il permet la gestion de plusieurs applications qui apportent un comportement aux objets de l'environnement virtuel, de plus il

gère les utilisateurs de la même façon que les applications, enfin le système peut gérer plusieurs mondes virtuels dotés de lois différentes en même temps.

Le seul inconvénient de ce système est que toutes les communications entre les objets lourds sont réalisées par des communications point à point (en utilisant UDP sur les stations de travail). En utilisant la diffusion de groupe une importante bande passante pourrait être économisée. Les auteurs ont bien compris cette limitation puisqu'ils se sont associés aux auteurs de MASSIVE pour créer la deuxième version de ce système qui utilise la diffusion sur groupe.

1.5.3. MASSIVE-2

Les auteurs de MASSIVE et d’AVIARY sont en train de concevoir et de réaliser un nouveau système de réalité virtuelle distribuée appelé MASSIVE-2 [Greenhalgh 96]. Ce nouveau système reprend le modèle spatial d’interaction qui a été étendu par la notion d’objets tiers [Benford 97]. MASSIVE-2 a une approche différente de MASSIVE puisqu’il utilise à la place des gestionnaires de collisions d’auras, des groupes de communications IP. Le modèle d’architecture passe, donc, d’un mélange client-serveur et égal à égal à un modèle purement égal à égal [Greenhalgh 97].

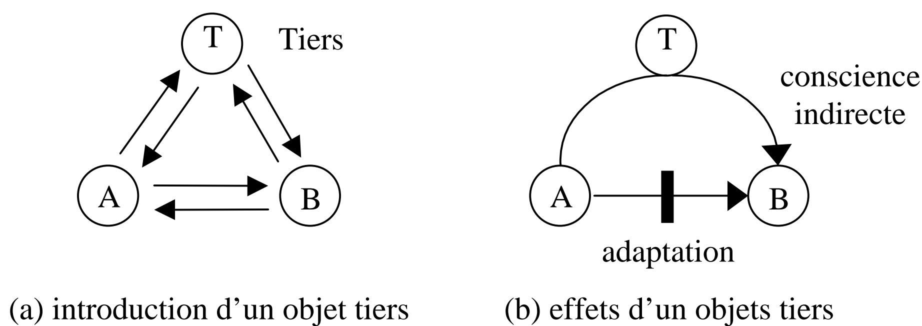
Cette partie va présenter MASSIVE-2 en insistant sur les différences avec MASSIVE et l’ancien modèle spatial d’interaction (tous deux présentés en partie 1.4.3).

1.5.3.1. Les objets tiers

La principale raison pour laquelle le modèle d’interaction a été étendu est la gestion des foules. En effet, les auteurs pensent que pour gérer des environnements virtuels à grande échelle (plusieurs centaines d’utilisateurs) il faut que le système puisse prendre en compte la formation de foules et en profiter pour économiser les ressources dont il dispose (réseau et calcul). De plus, cette prise en compte est importante pour éviter la présentation à chaque participant de toutes les données multimédias provenant d’une foule (visualisation détaillée de chaque participant et brouhaha formé par plusieurs dialogues simultanés). Enfin, cette prise en compte peut permettre de créer une sensation d’audience (qui est importante pour des

événements tels que les pièces de théâtre, les concerts, les événements sportifs…) et peut ainsi amener de nouvelles formes d'interaction sociale.

Le précédent modèle d'interaction était limité par son approche bilatérale des interactions (calcul de la conscience entre un seul objet observé et un seul objet observateur) qui n'est pas aisément extensible pour un très grand nombre de participants. De plus, les auteurs pensent que le concept d'objet adaptateur n'est pas suffisamment général pour gérer les facteurs contextuels intervenants dans une interaction (effet de l'environnement entourant les deux objets par exemple). Le concept d'**objet tiers** (*third party object*) a été introduit pour résoudre ces différents problèmes. Un objet tiers est un objet indépendant qui influence le niveau de conscience respective établi entre d'autres objets. Ainsi le modèle gère maintenant des scénarios d'interaction de base concernant trois objets et leurs relations de conscience (Figure 22 a).



(a) introduction d'un objet tiers (b) effets d'un objets tiers

Figure 22 : Les objets tiers

Il est important de noter que les objets tiers peuvent opérer différemment dans chacun des médias qu'ils gèrent. De plus, étant donné que ce sont des objets à part entière ils peuvent être visibles, mobiles ou fixes. Enfin, même s'ils sont pour l'instant utilisés pour gérer des relations spatiales, ils peuvent aussi travailler sur des relations non liées à l'espace.

Les objets tiers sont caractérisés par deux aspects : leurs effets sur le niveau de conscience entre deux objets et leur activation.

Les objets tiers peuvent avoir deux types d'effets qui peuvent être combinés de diverses façons (Figure 22 b). L'**adaptation** implique la manipulation d'une relation de conscience préexistant entre deux objets. C'est une généralisation de la notion d'objet adaptateur. La **conscience indirecte** (le terme anglais est : *secondary sourcing* qui est difficilement traduisible) implique l'introduction de nouvelles relations de conscience indirectes entre les

deux objets. Ce concept est à la base de la gestion des foules et implique l'absorption d'informations provenant d'un groupe d'objets, leur transformation et leur retransmission sous une nouvelle forme qui représente une vue globale du groupe. Il s'agit de créer une seule vue agrégée ou un seul flux d'information à partir de multiples sources.

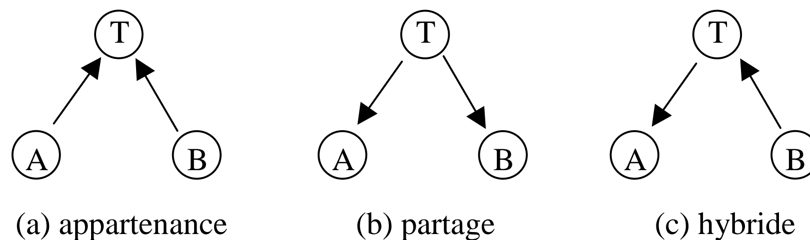


Figure 23 : L'activation des objets tiers

L'activation des objets tiers dépend des niveaux de conscience existant entre l'objet A et l'objet T d'une part et entre l'objet B et l'objet T (ceci représente quatre relations qui sont visibles sur la Figure 22 a). Parmi toutes les combinaisons possibles trois cas sont plus intéressants (Figure 23). L'**appartenance** correspond au cas où l'objet tiers est activé en fonction du niveau de conscience qu'il a des deux objets. La relation indique le niveau d'appartenance de l'objet (A ou B) à l'objet tiers. Par exemple, quelqu'un peut devenir membre d'une pièce d'un bâtiment (et donc lui appartenir) en traversant l'une de ces ouvertures. Le **partage** correspond au cas où l'objet tiers est activé en fonction du niveau de conscience que les deux autres objets ont de lui. On peut faire une analogie avec deux objets en partageant un troisième qui modifie leurs niveaux de conscience respectifs. Les tables de conférence de MASSIVE sont un bon exemple de partage. Le cas **hybride** implique que l'activation dépend de la conscience qu'un objet à de l'objet tiers et de la conscience que l'objet tiers à d'un autre objet. C'est le cas pour une foule qui pour un objet se trouvant à l'extérieur (ne faisant pas partie de la foule) est vue comme une forme agrégée des objets qui la composent (qui sont à l'intérieur). Les objets à l'intérieur sont visibles des autres objets qui sont à l'intérieur mais invisibles de l'extérieur (à la place, la vue agrégée est présentée).

Certains objets tiers (les pièces et les foules) forment une hiérarchie basée sur l'inclusion d'un objet (tiers ou normal) dans un objet tiers. Par exemple, un objet tiers représentant un bâtiment contiens différents objets tiers représentant les pièces, à l'intérieur de celles-ci on peut trouver des foules qui contiennent des objets représentant des participants.

1.5.3.2. Mise en œuvre du modèle spatial avec des groupes de communications

Dans MASSIVE-2, la hiérarchie d'objets tiers que nous venons de présenter est mise en œuvre par une hiérarchie de groupes de communications. Chaque groupe permet de gérer les communications (selon différents médias) des objets (normaux et tiers) inclus dans un objet tiers (typiquement une pièce ou une foule) qui est associé au groupe. Des mécanismes permettent à des objets qui émettent de l'information de rejoindre et de quitter des groupes pendant leurs déplacements respectifs (des groupes et des objets) dans le monde virtuel.

Les groupes de communications utilisés par MASSIVE-2 sont une généralisation des groupes de communications IP et peuvent être mis en œuvre par plusieurs adresses *multicast*. L'effet d'agrégation des objets tiers est mis en œuvre en étendant les groupes de communications IP de façon à ce qu'un groupe puisse émettre de l'information dans son groupe père (dans la hiérarchie).

L'architecture comprend quatre types de composants fondamentaux : Le **monde** qui a le même rôle que dans MASSIVE et qui est, de plus, associé au groupe racine de la précédente hiérarchie. Les **artefacts** qui jouent le rôle des objets mais qui sont essentiellement considérés comme des émetteurs d'information. Les **foci** qui dans cette version sont, a priori, indépendants des artefacts et sont essentiellement considérés comme des récepteurs d'information. Enfin, les **groupes**, qui représentent les objets tiers du modèle. Les groupes servent à structurer l'espace du monde. Ils possèdent une forme tridimensionnelle, et gèrent l'adaptation du niveau de conscience et l'agrégation. Ils comprennent deux groupes de communications IP qui servent à transmettre de l'information entre les artefacts et les foci qui sont associés au groupe. Il y a deux classes de membres de groupe : les membres émetteurs (les artefacts) et les membres récepteurs (les foci). Le premier groupe sert à envoyer l'information des artefacts aux foci (par exemple les modifications entraînées par leur déplacement) et donc seul les foci en sont membres. Le second sert aux foci pour demander aux artefacts une vue instantanée de leur état lors de leur connexion (de façon à pouvoir utiliser par la suite les mises à jours reçues sur l'autre groupe). Seul les artefacts sont membres de ce groupe.

L'appartenance des artefacts et des foci à un groupe sont définies comme suit. Un artefact n'appartient qu'à un seul groupe à la fois. C'est le groupe le plus bas dans la hiérarchie qui le

contient **totalement**. Cette relation stricte est très importante pour assurer la cohérence du modèle. D'autre part les groupes utilisent la même stricte relation pour s'organiser en hiérarchie. Aussi bien les groupes que les artefacts se déplacent dans la hiérarchie en se déplaçant dans le monde (lorsqu'un groupe se déplace dans la hiérarchie, tous ses fils se déplacent avec lui). Chaque groupe a la possibilité d'envoyer un flux d'information agrégée à son groupe père. L'appartenance des foci est gérée différemment. Un focus appartient à tous les groupes pour lesquels il y a intersection entre la forme du groupe et celle du focus. Ainsi un focus appartient toujours au groupe racine représentant le monde et potentiellement à de nombreux autres groupes.

Pour gérer l'adaptation de la conscience, chaque artefact possède un attribut qui liste les groupes dont il est membre (directement ou indirectement via la hiérarchie) ainsi qu'une définition des effets de l'adaptation. Ces effets peuvent être de trois types : un multiplicateur (qui amplifie ou atténue le niveau de conscience), une limite maximale du niveau de conscience et un seuil qui spécifie que les observateurs ne prennent conscience des artefacts présents dans un groupe que si leur niveau de conscience du groupe atteint cette valeur seuil.

MASSIVE utilise un protocole *multicast* fiable pour toutes les communications non temps réel (audio et vidéo). Le protocole est basé sur des acquittements négatifs et des réémissions (en mode point à point) en cas de perte de paquet (détectée grâce à des numéros de séquence inclus dans les paquets).

1.5.3.3. Conclusions sur cette architecture

Cette nouvelle version de MASSIVE est typique de la tendance actuelle des systèmes de RVD à évoluer vers la communication par groupe. Ceci est certainement dû à l'augmentation régulière du MBone (réseau mondial qui gère la communication par groupe) et à l'amélioration constante des routeurs et des protocoles qui le gèrent.

D'autre part, le concept de foule semble très intéressant pour gérer des environnements comportant de nombreux utilisateurs. Cependant, sa mise en œuvre semble un peu trop dynamique pour être efficacement gérée à l'heure actuelle par la communication par groupe d'IP.

1.6. Tentatives de normalisation de la Réalité Virtuelle Distribuée

Actuellement, un certain nombre d'organismes commencent à s'intéresser à la normalisation de la réalité virtuelle distribuée. Les initiatives principales sont VRML2 qui s'intéresse à la description d'environnements virtuels dynamiques distribués ou non et HLA qui a pour but de standardiser le domaine de la simulation distribuée. Nous allons, ici, décrire ces deux initiatives.

1.6.1. VRML2

Le but de VRML (*Virtual Reality Modeling Language*) est de standardiser tous les langages de description de scène (il en existe des dizaines) en proposant un format le plus général possible qui doit être importable et exportable dans tous les modeleurs, les systèmes de réalité virtuelle, les simulateurs… La première version de ce langage n'est donc qu'un simple langage de description de scène statique dans lequel quelques mécanismes hypermédias ont été ajoutés. La nouvelle version du langage, appelée VRML2 ou VRML97, a un but plus ambitieux puisque le langage doit être évolutif de façon à décrire n'importe quelle scène virtuelle dynamique [VRML97 97].

1.6.1.1. Structure du graphe de la scène

Le graphe de la scène contient des nœuds qui décrivent des objets et leurs propriétés. Le graphe contient des objets géométriques organisés hiérarchiquement qui forment une représentation audiovisuelle des objets de la scène, ainsi que des nœuds qui expriment le comportement de ces objets. Chaque type de nœud comprend : des attributs qui sont exportés (*exposedField*), des attributs privés (*field*), des événements qu'il peut recevoir (*eventIn*) et des événements qu'il peut émettre (*eventOut*). A chaque attribut exporté d'un nœud correspond implicitement un événement qu'il peut recevoir (qui permet de modifier cet attribut) et un événement qu'il émet lors de la modification de l'attribut. L'application qui affiche et gère une scène VRML est appelée un ***browser***.

1.6.1.2. Le routage des événements et les nœuds Script

Des nœuds VRML appelés **capteurs** (*sensors*) génèrent des événements en réponse à des actions de l'utilisateur, à des changements de la scène ou encore en fonction de l'écoulement du temps. Le routage des événements est un mécanisme, séparé de la hiérarchie du graphe de la scène, par lequel ces événements peuvent être propagés et effectuer des changements dans d'autres nœuds. A la fin du fichier VRML, un certain nombre de routes indiquent chacune un couple (nœud, événement) de départ et un couple (nœud, événement) d'arrivé sous la forme suivante :

```
ROUTE Objet1.événement1 TO Objet2.événement2.
```

Les événements sont typés et les routes ne peuvent être établies qu'entre deux événements du même type. Une fois générés, les événements sont acheminés via une route et traités par le nœud d'arrivée. Ce traitement peut changer l'état du nœud, générer des événements supplémentaires ou changer la structure de la scène.

Les nœuds de type **Script** permettent de définir des traitements arbitraires lors de la réception d'un événement. Un événement reçu par un nœud Script déclenche l'exécution d'une fonction du nœud qui peut envoyer des événements soit par le mécanisme normal de routage soit directement à tout nœud dont il possède une référence. Les scripts peuvent, de plus, ajouter ou effacer dynamiquement des routes et ainsi changer la topologie du mécanisme de routage. Actuellement, les fonctions peuvent être écrites au moyen des langages Java et JavaScript, cependant les *browsers* peuvent ne gérer qu'un des deux langages voire aucuns…

1.6.1.3. Conclusions sur VRML2

L'effort fourni par les auteurs de VRML2 est louable car il est indispensable de standardiser les différents aspects de la réalité virtuelle pour convaincre le monde industriel de sa maturité.

La version actuelle offre un langage de description de scène très complet par rapport aux autres langages existants. En ce qui concerne la partie dynamique il est plus difficile de savoir si le langage permettra de définir tout environnement virtuel dynamique, ceci est dû à la courte existence de langages de description de ce type (relativement à l'existence des langages de description de scène). Néanmoins, on peut dire à l'heure actuelle que l'aspect le plus pauvre de VRML2 est la prise en compte de mondes virtuels multi-utilisateurs. En effet, la seule

méthode décrite dans le standard est, pour les *browsers* qui supportent ce langage, d'écrire en Java des nœuds scripts qui vont se charger de répliquer les actions des utilisateurs sur les autres sites. Cette méthode n'est pas aisée à mettre en œuvre et est certainement peu efficace.

Une prise en compte au niveau même de l'architecture du *browser* est certainement la meilleure méthode pour permettre de créer des mondes virtuels multi-utilisateurs avec VRML2. Il existe actuellement un certain nombre de prototypes permettant d'étendre VRML aux aspects multi-utilisateurs. On peut citer : Community Place de Sony [Lea 97], CyberHub de Black Sun Interactive [CyberHub] et Smallview de GMD [Broll 97a].

Le système de GMD est actuellement le plus complet et le plus intéressant. Il utilise une architecture de communication basée sur la diffusion de groupe qui est fiabilisée grâce à un serveur (il envoie des acquittements positifs pour chacun des messages sur le groupe de communications) qui joue aussi le rôle de point d'entré dans le système (les nouveaux clients se connectent au serveur pour récupérer la version courante du monde virtuel) [Broll 97b]. Pour gérer l'extensibilité à de nombreux utilisateurs, le système permet de gérer des zones d'intérêt (similaires à celles de MASSIVE-2). Enfin, la spécificité la plus intéressante de ce système est de permettre des interactions multi-utilisateurs (deux ou plusieurs utilisateurs peuvent agir ensembles sur un même objet) [Broll 95]. C'est actuellement, avec notre système (cf. partie 4.2), le seul système qui permet cette fonctionnalité indispensable pour gérer des environnements virtuels collaboratifs.

1.6.2. HLA

HLA (*High Level Architecture*) [Miller 96] est une architecture distribuée développée, par de nombreux organismes militaires, universitaires et des sociétés impliquées dans le développement de simulateurs militaires, pour le Département de la Défense américain (DOD). HLA a pour but de supporter toutes les simulations utilisées dans le cadre militaire et au-delà toute simulation dans n'importe quel domaine. De plus, HLA doit permettre de simplifier la réutilisation de composants d'un simulateur à un autre (pour réduire les coûts de développement) et l'interopération entre différents simulateurs. HLA a été créée pour remédier à différents problèmes de DIS (cf. 1.3.2) et pour l'étendre à d'autres types de simulations (les simulations non temps réel basées sur des événements discrets et un temps logique notamment).

La philosophie qui sous-tend HLA est de spécifier le minimum de concepts et de services de façon à imposer le moins de contraintes possibles aux concepteurs de simulateurs.

Dans la terminologie HLA, une **fédération** correspond à un ensemble d'applications, appelées **fédérés**, qui peuvent interagir. Un fédéré est un membre d'une fédération. Cela peut être, une application de simulation, un gestionnaire de simulation, un collecteur de données, un simulateur servant d'interface à un être humain ou encore un outil passif de visualisation.

1.6.2.1. Présentation de l'Architecture

HLA définie trois composants : les **règles** qui sont des principes généraux qui doivent être appliqués lors du développement d'une fédération compatible avec HLA, le formalisme **OMT** (*Object Model Template*) qui est un modèle orienté objet générique qui permet de définir les composants d'une fédération et enfin la RTI (*Runtime Infrastructure*) une infrastructure d'exécution distribuée qui gère toutes les communications dans une fédération. Nous allons détailler dans cette partie, l'OMT et les services offerts part la RTI.

1.6.2.2. Le modèle orienté objet générique OMT

L'OMT [Lutz 96] permet de définir d'une façon orientée objet les entités (appelées **objets**) peuplant la fédération, les **interactions** possibles entre ces entités et les attributs publics des entités (qui peuvent être exportés vers d'autres fédérés). C'est une spécification de la fédération et des fédérés qui est utilisée comme un support d'échange entre les différents concepteurs intervenants dans la création de la fédération.

L'OMT définit deux types de modèles : le modèle de la fédération (FOM : *Federation Object Model*) et celui de la simulation (SOM : *Simulation Object Model*). Le FOM décrit les objets, interactions et attributs qui sont partagés dans une fédération. C'est une sorte de contrat régissant les échanges entre tous les fédérés. Le SOM décrit les objets, interactions et attributs qu'une application de simulation offre à une fédération (devenant ainsi un fédéré). C'est une sorte de fiche signalétique de l'application qui permet de savoir rapidement si elle peut être réutilisée dans une autre fédération. Les SOMs sont donc destinés à faciliter la réutilisation d'applications existantes.

1.6.2.3. La RTI

L'infrastructure d'exécution de HLA (RTI) [Calvin 96] propose un ensemble de services communs qui sont généralement requis par la plupart des systèmes de simulation distribués. Ces services communs sont répartis en six catégories : la gestion de la fédération, la gestion des déclarations, la gestion des objets, la gestion de la propriété, la gestion du temps et la gestion de la distribution des données.

La **gestion de la fédération** permet la création, le contrôle et la destruction d'une exécution répartie d'une fédération. La **gestion des déclarations** permet à un fédéré de spécifier les types d'objets, d'interactions et d'attributs qu'il va fournir pendant l'exécution de la fédération ainsi que ceux qui l'intéressent. Ces déclarations explicites permettent à la RTI de limiter les échanges de données sur le réseau. La **gestion des objets** permet aux fédérés de créer, modifier et détruire des objets et des interactions. La **gestion de la propriété** permet de transférer le droit de propriété d'un attribut (d'un objet) d'un fédéré à un autre. Ce droit de propriété permet à un fédéré de fournir des mises à jours de la valeur de l'attribut en question. Les services de la **gestion du temps** intéressent les simulations non temps réels qui peuvent participer à une simulation. Ils permettent par exemple de gérer un temps logique qui avance avec la réception d'événements discrets. La **gestion de la distribution des données** permet de mettre en œuvre des mécanismes de filtrage évolués comme ceux définis par Macedonia dans NPSNET-IV (cf. la partie 1.3.3.3).

1.6.2.4. Conclusions sur HLA

La définition de HLA est certainement une initiative importante pour le domaine de la réalité virtuelle distribuée. Les mécanismes définis sont très intéressants car ils sont génériques et pourront donc être a priori appliqués à tous les domaines d'application de la RVD. C'est la plus grande différence avec le protocole DIS défini pour le même organisme mais spécifique au domaine militaire.

Cependant quelques critiques peuvent être faites à la version actuelle de HLA. En effet, les possibilités d'interactions entre applications sont statiques puisque le FOM est défini, typiquement, avant la conception des fédérés. Les fédérés peuvent, il est vrai, être réutilisés mais il n'existe pas de mécanisme permettant de faire interopérer des simulations qui utilisent

des SOM incompatibles. Des mécanismes de ce type, définis dans [Gông 96], semblent intéressants car plus généraux et plus dynamiques.

1.7. Conclusion sur les systèmes de RVD existants

En conclusion de cette présentation des systèmes existants, nous allons reprendre chaque critère que nous avons défini au début de ce chapitre, puis nous présenterons un tableau récapitulatif indiquant les choix faits par chacun des systèmes. Enfin, nous tenterons d'analyser les choix les plus suivis actuellement.

Nous allons, tout d'abord, aborder la généricité des systèmes de RVD. On peut distinguer trois types de généricité : vis à vis des plates-formes matérielles gérées par le système, vis à vis des différents types d'applications supportées et enfin des types de mondes gérés par le système au même moment. Le tableau suivant récapitule pour chacun des systèmes le niveau de généricité pour chacun de ces trois aspects.

	Plates-formes	Applications	Mondes
Chatworlds	Stations UNIX et PC	Mono-application	1
RB2	Macintosh + SGI	Démonstrations	1
SIMNET	Spécifique	Mono-application	1
DIS	-	Mono-application	1
NPSNET	Stations SGI	Mono-application	1
dVS	Stations UNIX	Multi-applications	1
WAVES	Stations UNIX et PC	Multi-applications	Plusieurs
MASSIVE	Stations UNIX	Conférences	Plusieurs
VLNET	Stations SGI	Multi-applications	Plusieurs
DIVE	Stations UNIX	Multi-applications	Plusieurs
AVIARY	Stations UNIX et multiprocesseur	Multi-applications	Plusieurs
MASSIVE-2	Stations UNIX	Multi-applications	Plusieurs

On peut remarquer qu'à l'heure actuelle il y a deux types de systèmes. Des systèmes dédiés à une application (en général de simulation militaire) qui permettent de gérer de nombreux utilisateurs de façon très efficace et des systèmes plus génériques qui sont comparativement moins performants mais plus fiables pour permettre le travail coopératif. Cependant, le fossé

existant au niveau des performances et de l'extensibilité est en train de se combler grâce, essentiellement, à de nouveaux algorithmes répartis (filtrage de messages et *dead-reckoning* général).

Nous allons nous intéresser, maintenant, aux aspects répartis de ces systèmes en examinant les critères correspondant aux nombres de sites et d'utilisateurs et surtout à l'architecture répartie. Ainsi, le tableau suivant rappelle pour chacun des systèmes que nous avons abordés :

- si le système est multi-sites et quel est le nombre approximatif d'utilisateurs que peut gérer le système ;
- quel est le type d'architecture répartie (client-serveur ou égal à égal) ;
- comment est gérée la base de donnée représentant le monde (est-elle centralisée sur un seul site, est-elle dupliquée sur tous les sites ou partitionnée sur les sites) ;
- quel modèle de communication est utilisé (liaisons point à point, diffusion ou communication par groupes) ;
- si un filtrage de messages est mis en œuvre et si c'est le cas selon quels critères ;
- enfin, le tableau indique si une migration est possible à l'intérieur de l'architecture répartie.

	Multi-site multi-utilisateur	Architecture	Base de données	Modèle de communication	Filtrage des messages	Migration
Chatworlds	Multi-site 50 util.	Client-serveur	Centralisée	Point à point	Parfois	Non
RB2	Mono-site 2 util.	Client-serveur	Centralisée	Point à point	Non	Non
SIMNET	Multi-site 1000 util.	Egal à égal	Dupliquée totalement[1]	Diffusion	Non	Non
DIS	Multi-site 1000 util.	Egal à égal	Dupliquée totalement	Diffusion	Non	Non
NPSNET	Multi-site 100 000 util.	Egal à égal	Dupliquée[2]	Groupes de communications	Spatial, fonctionnel et temporel	Non
dVS	Multi-site 10 util.	Client-serveur hiérarchique	Dupliquée/ Centralisée[3]	Point à point	Fonctionnel	Non
WAVES	Multi-site 10 util.	Client-serveur avec plusieurs serveurs	Distribuée[4]	Point à point	Spatial et fonctionnel	Oui
MASSIVE	Multi-site 10 util.	Client-serveur avec plusieurs serveurs	Distribuée	Point à point	Spatial	Non
VLNET	Multi-site 10 util.	Client-serveur	Distribuée	Point à point	Spatial	Non
DIVE	Multi-site 10 util.	Egal à égal	Dupliquée	Groupes de communications	Spatial simple	Non
AVIARY	Multi-site 10 util.	Egal à égal	Distribuée	Point à point	Spatial pour les collisions	Oui
MASSIVE-2	Multi-site 100 util.	Egal à égal	Distribuée	Groupes de communications	Spatial	Non

La quasi-totalité des systèmes présentés qui subsistent actuellement sont des systèmes multi-sites qui gèrent une dizaine d’utilisateurs et qui tentent d’atteindre à moyen terme une centaine d’utilisateurs grâce aux progrès des réseaux et aussi à une architecture répartie extensible.

A ce propos, les architectures qui sont les plus utilisées sont des architectures extensibles de deux types : client/serveur avec plusieurs serveurs et égal à égal utilisant la diffusion sur groupe. Cependant, nous pouvons remarquer que la tendance actuelle est de venir à des architectures utilisant les groupes de communications que ce soit dans des architectures égal à égal ou même en client/serveur. Ceci est dû à l’extension et à l’amélioration constante du MBone. En effet, de plus en plus de services en lignes, tels que des téléconférences, des

[1] Comportements et géométrie sont dupliqués sur tout les sites.

[2] La duplication est partielle.

[3] Les comportements des objets sont centralisés sur un seul site, les autres informations sont dupliquées.

[4] Les comportements sont distribués à plusieurs sites et les autres informations sont dupliquées.

émissions radiophoniques et télévisuelles, proposent une version accessible via le MBone qui est plus efficace. Ainsi le MBone est de plus en plus populaire et apparaît comme une des meilleures solutions aux problèmes de bande passante de l'Internet.

Cette migration vers des architectures extensibles est liée à la prise en compte de diverses techniques de filtrage de messages, dont la plus efficace est sans doute la gestion de zones d'intérêt. Cette idée qui est en gestation depuis quatre ans maintenant semble s'imposer comme incontournable dans les systèmes de RVD qui se veulent extensibles. D'autre part, des extensions de l'algorithme de *dead-reckoning* fleurissent aussi dans la plupart des systèmes et s'imposent comme une très bonne méthode pour limiter les échanges sur le réseau.

Enfin, quelques systèmes ajoutent à toutes ces techniques, des possibilités de migration permettant d'équilibrer les charges de calcul présentes dans un système de RVD. Ce type de technique semble être important pour gérer des environnements virtuels peuplés par de nombreuses entités autonomes.

Nous allons aborder le dernier critère que nous avons défini : le modèle comportemental des systèmes de RVD. Nous présentons, ici aussi, dans un tableau un certain nombre de sous-critères pour chacun des systèmes présentés. Tout d'abord, il est important de rappeler quels sont les systèmes qui permettent l'ajout de comportements nouveaux en plus de ceux fournis par le système. Puis nous rappellerons pour chaque système quel type de comportement est géré (compilés avec le système, définis dans des applications externes ou interprétés). Enfin, une caractéristique intéressante pour gérer des environnements virtuels évolutifs est la possibilité de changer dynamiquement le comportement d'un objet virtuel pendant une session de simulation.

	Ajout de comportements possible	Comp. compilés avec le syst.	Comp. dans des applications externes	Comp. interprétés	Changement dynamique de comportement pendant l'exécution
Chatworlds	oui	non	non	oui	oui
RB2	oui	non	non	oui	oui
SIMNET	non	oui	non	non	non
DIS	non	oui	non	non	non
NPSNET	non	oui	non	non	non
DVS	oui	oui	non	non	non
WAVES	oui	oui	non	non	non
MASSIVE	oui	oui	non	non	non
VLNET	oui	non	oui	oui (L-systèmes)	oui
DIVE	oui	oui	oui	oui (DIVE-Tcl)	oui
AVIARY	oui	non	oui	oui (ECL)	oui
MASSIVE-2	oui	oui	non	non	non

Pour ce dernier aspect des systèmes de RVD, la définition de comportements interprétés par le système, semble en passe de s'imposer. En effet, la plupart des systèmes annoncent à court ou moyen terme leur adhésion au standard VRML qui prône cette solution. L'intérêt de ce type de comportement est de pouvoir être facilement accessible via le WWW dans une forme utilisable par de nombreuses plates-formes. Néanmoins, le principal inconvénient dans ce cadre étant la relative efficacité due à l'interprétation d'un langage, nous proposons dans ce document une alternative basée sur la liaison dynamique qui nous semble intéressante pour conserver un bon niveau d'efficacité.

Enfin, le problème commun aux différents systèmes que nous avons décrit est d'imposer toutes les solutions choisies pour tous les environnements gérés par les systèmes. Or il n'est pas du tout évident que toutes les solutions soient suffisamment générales. Par exemple, la gestion de zones d'intérêt dans un environnement très dynamique comportant peu de sites n'est pas intéressante et peut même entraîner une augmentation des transmissions sur le réseau si les entités changent très fréquemment de zones (à chaque fois, l'état instantané de la zone doit être transmis). Le système que nous proposons tente de résoudre ce problème en proposant des techniques permettant de choisir parmi différents modèles de distribution ceux qui sont les plus adaptés à un domaine d'application.

Chapitre 2. Présentation du système VIPER

Le chapitre précédent nous a permis de présenter les caractéristiques générales d'un système de réalité virtuelle distribuée. Nous avons, de plus, présenté un certain nombre de systèmes parmi les plus représentatifs. Nous allons maintenant décrire le modèle qui sous-tend le système de réalité virtuelle distribuée que nous proposons : VIPER (*VIrtuality Programming EnviRonment*).

Tout d'abord, nous allons présenter la structure logique qui permet de concevoir une application de réalité virtuelle avec VIPER. Nous présenterons ensuite, les choix techniques que nous avons fait pour réaliser le système. Puis, nous décrirons la structure en couche de l'architecture logicielle de VIPER. Nous détaillerons ensuite les mécanismes qui permettent de définir les comportements des objets virtuels et leurs interactions. Enfin, nous terminerons ce chapitre par une discussion sur les termes utilisés par notre modèle.

2.1. Structure d'une application

2.1.1. Structure logique de l'environnement virtuel

Notre but est de proposer un modèle de calcul réparti générique permettant de gérer n'importe quelle application pouvant être modélisée en terme d'échanges (symbolisés par des **stimuli**) entre des **entités**, dans un **univers virtuel** (Figure 24).

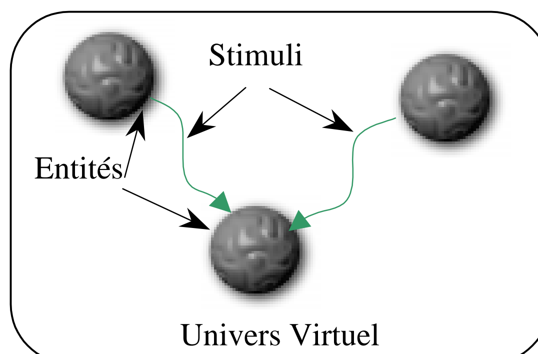


Figure 24 : Structure d'un environnement virtuel

• La notion d'**entité** nous permet de gérer de façon uniforme les décors des mondes virtuels, les objets virtuels mais aussi des **clones** [Mouli 93] (objet associé à un utilisateur, à une application ou à un robot, qui leur permet de participer à la simulation comme s'ils étaient réellement présents dans l'univers virtuel). Le clone d'un utilisateur est généralement appelé **avatar**[5] dans la littérature. Les entités sont autonomes et possèdent un ensemble de propriétés et de comportements. Elles sont conceptuellement regroupées en familles (population des entités ayant les mêmes propriétés et les mêmes comportements). Globalement les systèmes simulés sont homogènes (beaucoup d'objets regroupés dans peu de familles).

• Les **stimuli** sont les véhicules des interactions entre les entités.

• L'**univers virtuel** représente le milieu tridimensionnel dans lequel les entités coexistent et interagissent.

Pour décrire de façon plus détaillée la structure d'une application nous allons maintenant présenter le mécanisme qui permet à plusieurs entités de communiquer. Nous parlerons ensuite plus en détail des entités et de leur structure logique.

2.1.2. Mécanisme de communication inter-entités

Les interactions entre les entités transitent par un média qui permet des communications entre plusieurs entités simultanément. Une entité reçoit les modifications de son environnement grâce à des capteurs et peut agir sur cet environnement grâce à des effecteurs.

Nous proposons, pour chaque type d'interaction, une modélisation par quatre types d'éléments : capteur, effecteur, stimulus et espace de stimuli (Figure 25).

---

[5] Avatar vient du sanskrit *avatãra* (descente de Vishnu sur terre) qui est le nom donné à chacune des incarnations de Vishnu dans la religion hindoue.

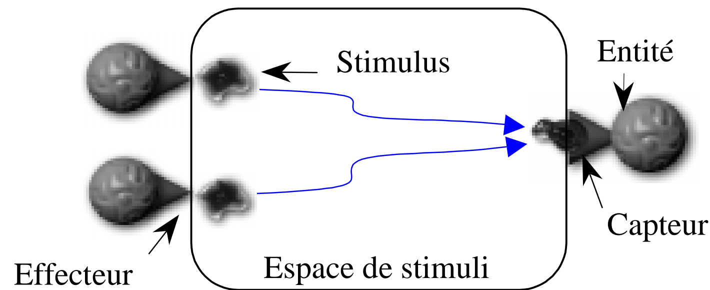


Figure 25 : Mécanisme de communication inter-entités

Un stimulus est un phénomène ou un événement perceptible par une entité (par exemple une forme graphique, un son ou encore une collision). Un espace de stimuli est un milieu dans lequel s'échangent les stimuli d'un certain type. On peut le définir comme une projection de l'univers virtuel selon un axe propre à un stimulus (espaces des formes graphiques, espaces des sons...).

Un capteur reçoit les stimuli que l'entité est capable de percevoir (formes visibles par l'entité, sons proches...). Par l'exécution d'une action, un effecteur produit un stimulus que d'autres entités percevront.

En formalisant, on peut dire qu'un stimulus est un n-uplet comportant au moins une étiquette temporelle T et autant que possible pour chaque valeur dépendante du temps une dérivée du premier ordre (par rapport au temps), de façon à permettre une extrapolation de la valeur dans le cas où le stimulus capté n'est pas suffisamment récent. Soit S l'ensemble de tous les stimuli d'un certain type, un espace de stimuli est définit par : $E_S = \{ s / s \in S \}$. De plus, un tel espace peut disposer d'une fonction d'évolution (EV) de signature EV : $E_S \rightarrow E_S$, qui modifie les stimuli en fonction du temps.

2.1.3. Structure logique d'une entité

Dans cette partie nous allons décrire la structure des entités présentes dans l'environnement virtuel en exposant les mécanismes associés à la définition et à la prise en compte du comportement de ces entités (Figure 26).

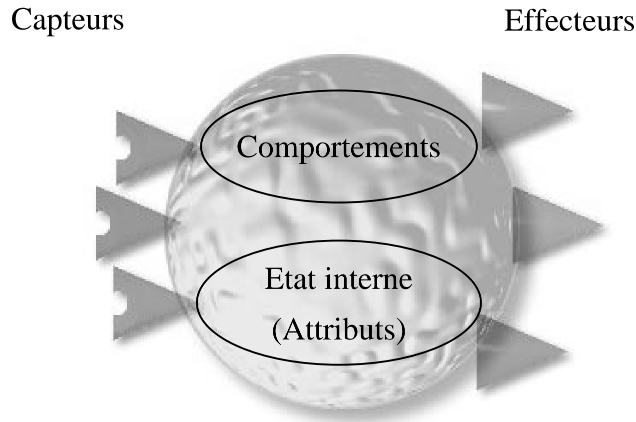


Figure 26 : Schéma structurel d'une entité

Une entité possède un ensemble de capteurs et d'effecteurs qui lui permettent d'interagir avec son environnement (univers virtuel et autres entités). Une entité est, de plus, dotée d'un certain nombre de comportements qui modifient son état interne (ensemble de ses attributs) et commande des actions à ses effecteurs.

2.1.4. Structure logique et rôle d'un capteur

Un capteur récupère à une fréquence donnée, sur demande, ou automatiquement lors d'une modification de l'environnement de l'entité, un ensemble de stimuli présents dans un espace de stimuli. Cet ensemble de stimuli est filtré pour ne conserver que ceux qui sont pertinents en fonction des paramètres du capteur (comme par exemple sa position et son orientation dans l'environnement). L'ensemble de stimuli obtenu après filtrage est interprété pour fournir une information pertinente pour l'entité. Le résultat de l'interprétation va ensuite être envoyé à l'entité qui possède le capteur. Le résultat pourra, d'autre part, être stocké dans un tampon prévu à cet effet et consommé par le comportement de l'entité selon ses besoins (Figure 27).

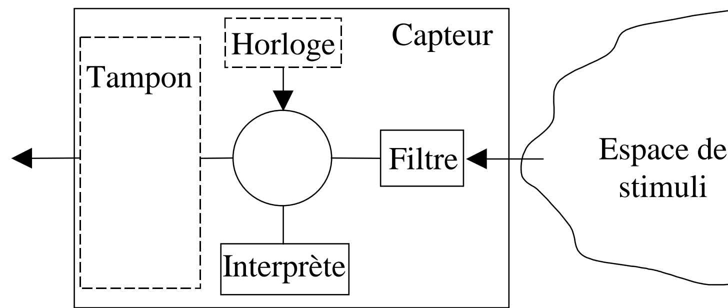


Figure 27 : Structure logique d'un capteur

La signature du fonctionnement d'un capteur est : E_S x C $\rightarrow F_C(E_S)$ où $F_C(E_S)$ est l'ensemble des interprétations de l'espace de stimuli E_S et C l'ensemble des capteurs d'un même type.

2.1.5. Structure logique et rôle d'un effecteur

Un effecteur exécute, soit sur demande soit à une fréquence donnée, une ou plusieurs actions en produisant des stimuli qui sont placés dans un espace de stimuli. Les actions à réaliser peuvent être stockées dans un tampon. Dans ce cas, elles seront réalisées à la fréquence de l'horloge de l'effecteur.

Un effecteur peut posséder un filtre qui permettra de limiter l'émission de stimuli dans l'environnement (Figure 28).

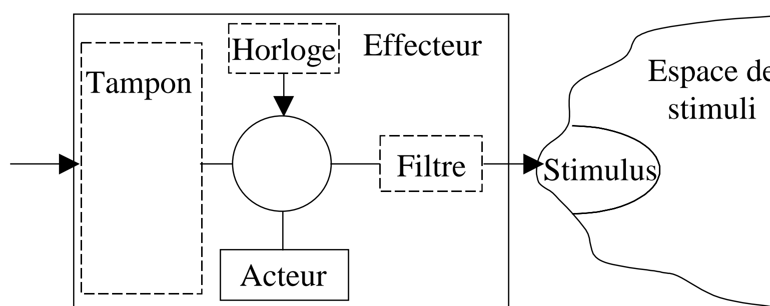


Figure 28 : Structure logique d'un effecteur

La signature du fonctionnement d'un effecteur est : A x EF $\rightarrow$ S où A est un ensemble d'actions possibles dans l'environnement et EF l'ensemble des effecteurs d'un certain type.

2.2. Choix techniques

Pour implanter les applications cibles sous la forme précédemment décrite, nous avons choisi de définir une architecture distribuée égal à égal. Cette architecture permet deux niveaux de programmation de difficulté croissante grâce à un modèle SPMD orienté-objet. L'architecture permet de gérer une distribution sur de nombreux types de machines. Enfin, nous avons décidé de nous conformer le plus possible aux divers standards émergeants. Avant de décrire cette architecture nous allons justifier ces différents choix.

2.2.1. Une architecture distribuée égal à égal

Nous avons choisi le modèle le plus général et le plus extensible. Nous utilisons la communication par groupe partout où cela est possible.

Néanmoins, l'architecture comprend un serveur qui est utilisé lors de la connexion pour obtenir des informations sur les groupes de communications utilisés par un monde virtuel. D'autre part, ce serveur permet d'obtenir des identificateurs uniques pour les types de stimuli qui peuvent être échangés dans les mondes virtuels. Ce serveur pourra aussi servir, par la suite, à gérer une procédure d'authentification pour protéger l'accès à un environnement virtuel.

2.2.2. Une approche orientée objet

Une approche orientée objet semble bien adaptée compte tenu de la structure des applications qui est, intrinsèquement, elle-même fortement orientée objet (les entités envoient/reçoivent des stimuli vers/depuis d'autres entités). Les avantages classiques des langages à objets (encapsulation, réutilisabilité...) et plus particulièrement l'héritage semblent des plus intéressants, notamment pour implanter la notion de famille d'entités.

Le choix du langage C++ est justifié par sa large diffusion (C++ existe en effet sur la plupart des architectures matérielles) et sa reconnaissance comme standard de facto en programmation orientée objet de systèmes distribués.

2.2.3. Un modèle de programmation inspiré du SPMD

La plupart des architectures réparties que nous avons décrites sont fondées sur un modèle globalement MIMD[6] (les sites exécutent des instructions différentes sur des données différentes). Dans ce cadre, une application est le plus souvent décomposée en un ensemble de processus qui communiquent (comme l'illustre DIVE). Cependant, dans ce cas, le développeur doit gérer ce découpage en processus et s'expose donc aux problèmes classiques de mise au point, d'équilibrage des charges, d'interblocage...

Notre approche est inspirée du modèle SPMD[7] (tous les sites exécutent le même programme ou le même ensemble de tâches mais de manière asynchrone et sur des données différentes). A terme, le développeur devrait pouvoir écrire l'application comme si elle était purement séquentielle en indiquant uniquement le schéma de répartition des données.

Pour atteindre cet objectif, nous avons choisi d'offrir au développeur un ensemble de classes génériques qui lui permettent de créer et d'utiliser des objets parallèles. Un objet parallèle est un agrégat (ensemble d'objets en général identiques réunis sous un même nom – exemple : un tableau est un agrégat) qui encapsule la répartition de ses données et les mécanismes gérant les accès distants éventuels.

Des classes gérant divers modes de répartition ont été implantées (réplication active, partitionnement…).

2.2.4. Deux niveaux de programmation

Pour répondre aux spécificités des diverses applications, de nombreux choix peuvent être faits parmi les optimisations proposées par VIPER. Ainsi nous mettons à la disposition des développeurs deux niveaux de programmation. Le premier niveau masque totalement l'aspect réparti de l'application qui apparaît ainsi totalement séquentielle. Le second niveau permet de redéfinir les mécanismes de répartition.

---

[6] MIMD signifie Multi*ple Instructions Multiple Data.*

[7] SPMD signifie *Simple Program Multiple Data.*

2.2.5. Prise en compte de l’hétérogénéité

Nous avons décidé de gérer de nombreuses architectures matérielles : du PC à la station SGI haut de gamme. Nous pensons, en effet, qu’un système de réalité virtuelle distribuée ne doit pas être spécifique à quelques stations graphiques très puissantes mais doit plutôt être disponible sur de nombreuses plates-formes pour être accessible à un public très large. Pour cela nous avons essayé de choisir quand c’était possible des librairies existant sur de nombreuses plates-formes.

Ainsi pour le rendu graphique et sonore nous utilisons la librairie WorldToolKit de la société Sense8 qui existe sur de nombreuses plates-formes comme Windows95, Windows NT (dans les deux cas il existe une version DirectX et une version OpenGL), Sun, DEC Alpha (sous Windows NT) et Silicon Graphics.

De même, pour la gestion de la répartition, nous avons décidé d’utiliser PVM (*Parallel Virtual Machine*) qui existe sur un très grand nombre de plates-formes. En effet, ce système de communication existe sur PC, sur la quasi totalité des stations de travail existantes et aussi sur de nombreuses machines parallèles. Nous utilisons aussi RMP (*Reliable Multicast Protocol*) qui existe sur PC et sur de nombreuses stations de travail.

2.2.6. Adhésion au standard VRML

Nous avons décidé d’adhérer au standard émergeant VRML pour pouvoir profiter du nombre important d’objets et de mondes virtuels définis dans ce standard. De plus, ce format de fichier est en passe de devenir le plus connu des formats grâce à sa large présence sur le WWW.

Ainsi, les mondes virtuels de VIPER, les avatars et les autres objets que peuvent apporter un site lorsqu’il rejoint le monde virtuel sont définis dans des fichiers VRML.

2.3. Le modèle en couche

L'architecture de VIPER est constituée de quatre couches (Figure 29) :

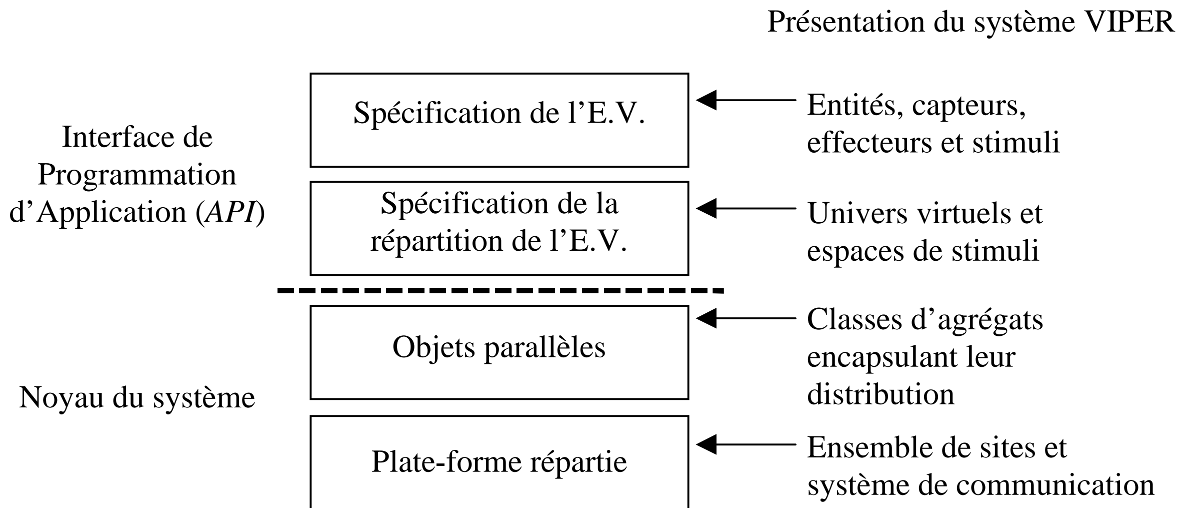


Figure 29 : Architecture en couches de VIPER

- La couche la plus basse, la plate-forme répartie, correspond au système de communication utilisé par VIPER (actuellement PVM [Sunderam 90] et RMP [Whetten 95]), qui est encapsulé dans un groupe de classes, de façon à permettre un portage rapide vers d'autres systèmes de communication (MPI[8], RAMP[9]...).

- Au dessus de cette couche, se trouve un environnement de programmation parallèle SPMD à objets développé dans notre équipe de recherche [Moisan 93a,b] qui propose différentes classes d'objets parallèles.

- La troisième couche permet d'exprimer les mécanismes de répartition de l'environnement virtuel. Les espaces de stimuli et les univers virtuels sont des agrégats, gérés par des objets parallèles, contenant des stimuli et des entités. Les mécanismes de répartition sont définis par héritage et spécialisation de classes préexistantes d'objets parallèles existants.

- La dernière couche permet la définition d'entités (comportements et attributs), de capteurs, d'effecteurs et de stimuli spécifiques à l'application considérée en spécialisant des classes existantes.

[8] MPI (*Message Passing Interface*) est un système de communication par passage de messages.

[9] RAMP (*Reliable Adaptative Multicast Protocol*), tout comme RMP, est un système de communication par groupe fiabilisé.

La troisième et la quatrième couches mettent à la disposition des développeurs deux niveaux de programmation. Le premier niveau masque totalement l'aspect réparti de l'application. Le programmeur définit de nouvelles entités (en définissant leurs comportements et leurs attributs), de nouveaux stimuli, de nouveaux capteurs et de nouveaux effecteurs comme s'il travaillait avec une hiérarchie de classes normale utilisée pour réaliser un programme séquentiel. La distribution des données est cachée dans les espaces de stimuli et les univers virtuels utilisés.

Le second niveau permet de redéfinir les mécanismes de répartition. A ce niveau, le développeur peut créer de nouveaux espaces de stimuli et de nouveaux univers virtuels qui permettent de gérer la répartition de stimuli et d'entités. Le programmeur est ainsi libre d'optimiser l'application à sa convenance [Torguet 95a].

Nous allons présenter dans la suite de ce chapitre le premier niveau de programmation et nous détaillerons, dans le chapitre suivant, le second niveau et les couches basses de VIPER qui sont plus particulièrement utilisées par la troisième couche.

2.4. Spécification de l'environnement virtuel

Chaque entité possède un comportement plus ou moins complexe (qui peut être changé dynamiquement) constitué par des composants comportementaux connectés (Figure 30) [Balet 96]. Un capteur ou tout autre composant peut déclencher un composant via des communications internes.

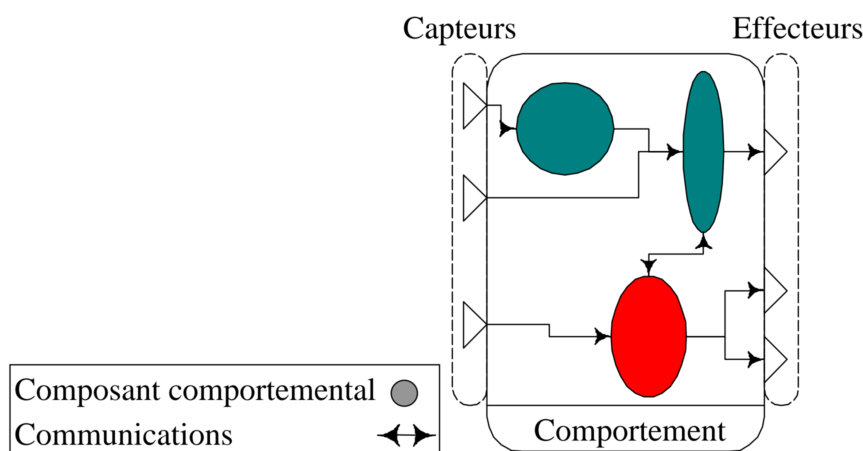


Figure 30 : Modèle comportemental

Les composants comportementaux peuvent commander des actions aux effecteurs en fonction de la connaissance de l'environnement (acquise via les capteurs), d'une mémoire (représentée par les attributs de son état interne) et d'un mécanisme de réflexion qui peut être modélisé de diverses façons (fonction dans un langage conventionnel, automate à états finis [Carlsson 93], moteur d'inférence PROLOG [Mouli 92]...). Le mécanisme de réflexion d'une entité peut aussi être modélisé par un réseau de Petri orienté objet [Balet 97]. Actuellement VIPER permet de mettre en œuvre des composants soit dans un langage interprété (ObjectTcl[10]) soit directement en langage C++ (ces composants peuvent être liés dynamiquement). Les composants communiquent par l'intermédiaire d'appels de méthodes (ce qui correspond conceptuellement à l'envoi d'un message pour un langage à objets).

La figure suivante (Figure 31) présente l'ensemble des classes qui gèrent, au plus haut niveau, la définition de comportements.

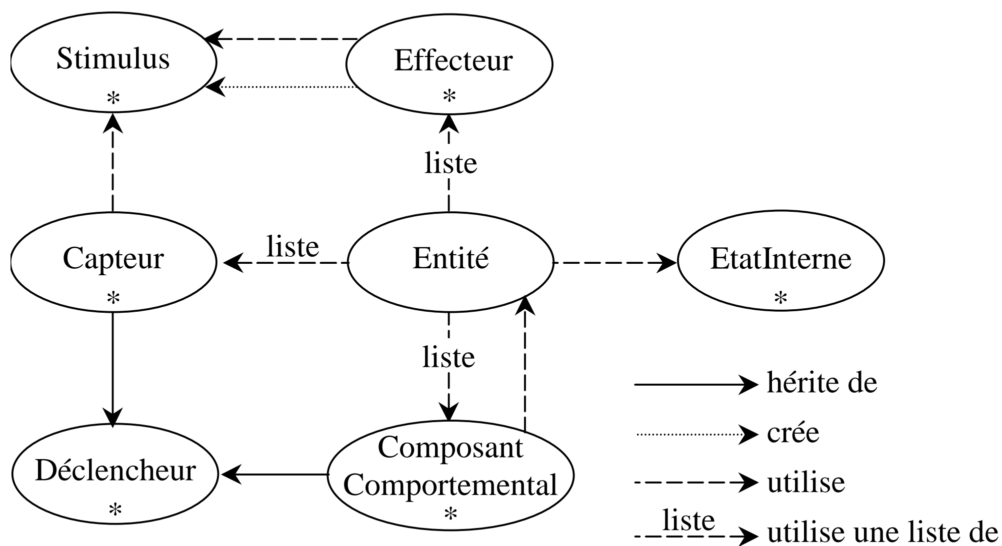


Figure 31 : Ensemble de classes qui gère la définition des comportements

Nous allons présenter dans ce chapitre chacune de ces classes ainsi que les mécanismes que nous avons définis pour rendre le système le plus extensible et le plus dynamique possible.

[10] Ce langage interprété est fondé sur le langage Tcl [Osterhout 94] qu'il étend.

2.4.1. Les entités

Les entités sont définies comme des éléments d'agrégat de façon à pouvoir être réparties (Figure 32). Chaque entité possède un identificateur qui est unique sur la plate-forme distribuée. Les entités possèdent chacune des capteurs et des effecteurs pour communiquer, un état interne et des composants comportementaux. Les classes Capteur et Composant-Comportemental héritent toutes deux d'une classe appelée Déclencheur qui permet de les traiter de manière uniforme lorsqu'ils déclenchent un composant comportemental.

```
#ifndef HENTITE
#define HENTITE
...
class Entité : public ElémentD'agrégat
{
protected :

     // Identificateur unique de l'entité
     IdD'Entité MonId;

     Liste<ComposantComportemental*> MesComposants;

     Liste<Capteur*> MesCapteurs;

     // Déclencheurs (contient tous les capteurs
// et les composants)
Liste<Déclencheur*> MesDéclencheurs;

     Liste<Effecteur*> MesEffecteurs;

     EtatInterne *MonEtatInterne;

public :
     ...
     // Appelle les méthodes «action» des composants
     // qui doivent s'exécuter à chaque itération
     // de la boucle de simulation
     virtual void Action();
     ...
};
#endif
```

Figure 32 : Extrait de l'interface de la classe Entité

Les entités d'un monde virtuel sont déclarées directement dans les fichiers VRML qui décrivent la scène (en général comme fils du nœud qui deviendra leur forme). Cette référence est placée dans un nœud spécifique à VIPER appelé : ViperEntité. Ce nœud spécifique est

défini, en accord avec la norme d'extension de VRML 1.0 [VRML1.0 96], de la façon suivante :

```
ViperEntité {
   fields [  MFString systemes,
             MFString composantsComportementaux,
             MFString capteurs,
             MFString effecteurs,
             MFString nouveauxStimuli,
             MFString etatInterne,
             MFString parametres ]
   systemes [ ]
   composantsComportementaux [ ]
   capteurs [ ]
   effecteurs [ ]
   nouveauxStimuli [ ]
   etatInterne [ ]
   parametres [ ]
}
```

On peut noter, que cette déclaration définit la structure du nœud ViperEntity et déclare une entité ViperEntity vide (qui n'a pas de sens et n'est placée ici que pour présenter la structure). Le premier champ définit une liste de chaînes de caractères (type MFString de VRML) qui permet d'indiquer sur quels systèmes l'entité peut s'exécuter (exemple : IRIX6, WIN32…). Par convention, si la liste est vide l'entité peut s'exécuter sur n'importe quel système. Les trois champs suivants définissent chacun une liste de chaînes de caractères qui permettent d'identifier les composants comportementaux, les capteurs et les effecteurs dont disposera l'entité. Le cinquième champ permet de préciser les types de stimuli non standard que l'entité peut produire et/ou recevoir. L'avant-dernier champ définit une liste de chaînes de caractères qui identifie le fichier (ou les fichiers si l'entité est exécutable sur plusieurs plates-formes) qui définit l'état interne de l'entité. Enfin, le dernier champ définit une liste de chaînes de caractères qui peuvent être utilisés pour paramétrer l'entité en indiquant, par exemple, le nom du nœud représentant sa forme graphique, l'URL[11] d'une icône sonore...

Les chaînes de caractères des seconds, troisièmes, quatrièmes, cinquièmes et sixièmes champs peuvent contenir soit le nom d'une classe standard définie par VIPER (à l'exception du champ `nouveauxStimuli`) soit une URL qui permet d'accéder à la définition d'une nouvelle classe. La nouvelle classe pourra être définie soit dans un fichier ObjectTcl soit dans une librairie dynamique (cf. 2.4.2). Dans le premier cas (ObjectTcl), la chaîne de caractère

[11] Une URL (*Uniform Resource Locator*) permet d'accéder à un document sur le World Wide Web.

sera composée de l'URL du fichier, suivi du signe #, suivi du nom de la classe. Dans le second cas, la chaîne sera composée de l'URL du fichier, suivi de #, suivi du type du système, suivi de #, suivi du nom de la classe. De plus, pour les nouveaux stimuli, l'URL sera terminée par les signes >> suivi du nom d'une classe standard de VIPER ancêtre de la nouvelle classe.

Ainsi, un composant comportemental gérant la réponse à une collision peut être défini par la chaîne : "http://virus.irit.fr/VIPER/Comportements.so#IRIX6#Collision".

Pour une entité qui peut être exécutée sur différents systèmes, il conviendra de fournir pour chaque composant comportemental de l'entité et pour chaque type de système une URL différente. D'autre part, pour qu'une entité puisse être exécutée sur n'importe quel système, il faut que ses capteurs, effecteurs, composants et son état interne soient de type standard ou qu'ils soient définis en ObjectTcl.

2.4.1.1. Les entités migrantes

Pour qu'une entité puisse être déplacée d'un site à un autre, par exemple pour se rapprocher d'une entité avec laquelle elle interagit, il faut qu'elle puisse s'empaqueter et se dépaqueter d'un message. Il faut, de plus, que ces composants comportementaux puissent s'exécuter sur le site de destination.

Pour gérer de telle entités, nous avons défini une classe EntitéMigrante qui hérite de la classe Entité en définissant deux méthodes : empaqueter et dépaqueter. Ces méthodes sont héritées par la classe Entité depuis la classe ElémentD'agrégat. Une entité migrante doit posséder un composant comportemental spécifique qui ne définit pas une méthode Action mais qui doit définir une méthode Empaquetage et une méthode Dépaquetage. Les signatures de ces méthodes sont les suivantes : `boolean Empaquetage(Message*)`et `boolean Dépaquetage(Message*)`. Le paramètre est un pointeur sur le message dans (ou depuis) lequel l'entité doit être empaquetée (ou dépaquetée) et le paramètre de retour indique si il y a eu un problème. Ce sont ces méthodes qui vont être appelées par les méthodes empaqueter et dépaqueter de la classe EntitéMigrante.

Une entité migrante est définie dans un fichier VRML grâce à la structure suivante :

```
ViperEntitéMigrante {
   fields [  MFString systemes,
             MFString composantsComportementaux,
             MFString capteurs,
             MFString effecteurs,
             MFString nouveauxStimuli,
             MFString etatInterne,
             MFString empaquetage,
             MFString parametres ]
   systemes [ ]
   composantsComportementaux [ ]
   capteurs [ ]
   effecteurs [ ]
   nouveauxStimuli [ ]
   etatInterne [ ]
   empaquetage [ ]
   parametres [ ]
}
```

Dans cette structure la seule différence est la présence de l'attribut empaquetage qui contient une URL (ou plusieurs URLs si le composant comportemental est définit sur plusieurs plates-formes) pour le composant comportemental décrit ci-dessus.

2.4.2. Les composants comportementaux

Les composants comportementaux héritent tous de la classe ComposantComportemental (Figure 33). Ils définissent une méthode Action qui doit être appelée pour que l'algorithme de comportement s'exécute. Cette méthode peut être invoquée de deux façons : soit par un capteur ou un autre composant comportemental, soit par VIPER à chaque itération de la boucle de simulation. Un composant comportemental invoqué de façon directe ou indirecte par un capteur est appelé un composant **réactif** ou réflexe. Alors qu'un composant comportemental invoqué directement ou indirectement dans la boucle de simulation est appelé un composant **actif** ou réfléchi. Les comportements réactif qui doivent être invoqués par un capteur doivent s'inscrire lors de leur création auprès du capteur et les comportements actifs qui doivent être invoqués directement à chaque pas de simulation doivent s'inscrire auprès de l'entité pour être appelés lors de l'invocation de sa méthode Action.

```
#ifndef HCCOMPORTEMENAL
#define HCCOMPORTEMENAL
...
// les composants comportementaux et les capteurs sont
// des Déclencheurs
class ComposantComportemental : public Déclencheur
{
protected :

     // Entité qui possède le composant
     Entité      *MonEntité;

public :
     // le paramètre de construction est une référence
// vers l'entité qui possède le composant
     ComposantComportemental(Entité &);

     ~ComposantComportemental();

     Entité&     AccésAEntité()
     { return *MonEntité; }

     // le paramètre indique la position dans la liste de
      // l'entité de l'objet (capteur ou composant) qui a
      // déclenché l'exécution de la méthode
     virtual void Action(int IdDéclencheur);
     ...
};
#endif
```

Figure 33 : Extrait de l'interface de la classe ComposantComportemental

2.4.2.1. Composants comportementaux interprétés

Un composant comportemental peut être défini dans une classe ObjectTcl. Le principal intérêt du langage ObjectTcl est que ce langage est orienté objet et facilement intégrable à une application définie en C++. En effet, grâce à ObjectTcl une classe de ce langage peut hériter d'une classe C++ et redéfinir des méthodes virtuelles (méthodes polymorphes). Par exemple, si il existe une classe C++ ObjetGraphique (gérant des listes de facettes) dotée de deux méthodes virtuelles (afficher et volume) on peut définir une classe ObjectTcl Sphère qui pourra redéfinir la méthode volume. Par la suite, on pourra disposer d'une liste polymorphe d'objets graphiques (contenant des Sphères, des Cubes… – Figure 34) et lorsqu'on appellera la méthode volume sur un des éléments, ce sera soit le code C++ soit le code ObjectTcl qui sera appelé (le choix est réalisé en fonction du type statique de l'objet sur lequel la méthode est appelée).

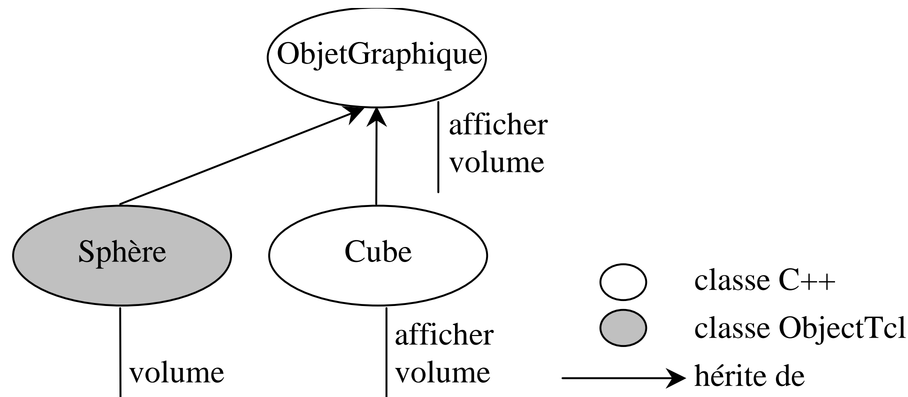


Figure 34 : Exemple de graphe de classe mixte C++/ObjectTcl

Parallèlement à la classe ComposantComportemental on a écrit, dans un langage de liaison, appelé CDL (*class definition language*), une description de la classe (ainsi que de toutes les classes qui pourront être utilisées depuis ObjectTcl : Entité, EtatInterne…) comme suit :

```
pass {
// Cette partie sera incluse telle quelle dans les classes
// C++ de liaison

// Generé depuis ComposantComportemental_cdl.cdl
#include "ComposantComportemental.hxx"
#include " ComposantComportemental_cdl.hxx"
}

class -isA Déclencheur ComposantComportemental {
   constructor {obref Entité}

   method AccésAEntité -static {void} {obref Entité}

   method Action -dynamic {int} {void}

}
```

Figure 35 : Définition dans le langage CDL de la classe ComposantComportemental

Un composant comportemental Collision peut ensuite être défini en ObjectTcl comme suit :

```
otclInterface Collision -isA ComposantComportemental {
  constructor {obref Entité}

  # Redéfini la méthode "Action" de la classe ancêtre
  method Action {int}
}

otclImplementation Collision {
  # Le constructeur passe son argument au
  # constructeur de la classe ancêtre
  constructor {ent} {{ComposantComportemental $ent}} {
       # initialisation des attributs de la classe
       set explose 0
  }
  destructor {
       # Destructeur
  }
  # La nouvelle version de la méthode action
  method Action {cc} {
       # Le comportement proprement dit est défini ici
  }
  # Définition d’attribut
  attribute explose
}
```

Figure 36 : Un composant comportemental écrit en ObjectTcl

2.4.2.2. Composants comportementaux liés dynamiquement

La définition de composants comportementaux liés dynamiquement est semblable à celle des composants interprétés. Cependant les composants liés dynamiquement sont définis en C++. Ils sont ensuite compilés sur chaque type de machine qui doit pouvoir exécuter le comportement et stockés dans des librairies dynamiques (ces librairies sont appelées DSO dans le monde UNIX, DLL dans le monde Windows).

L’édition de lien dynamique dépendant du type de système nous l’avons encapsulée dans une classe qui permet aux classes appelantes de rester indépendantes du matériel. Dans les deux cas (UNIX et Windows) il y a deux étapes : l’ouverture du fichier contenant la librairie dynamique (dlopen pour UNIX et LoadLibrary pour Windows) et la recherche d’une fonction dans la librairie (dlsym pour UNIX et GetProcAddress pour Windows).

Un composant comportemental à lier dynamiquement est défini de la façon suivante :

```
class Traitement : public ComposantComportemental
{
public :

     Traitement(Entité &);

     ~Traitement();

     virtual void Action(DéclencheurId);
};

// définition du comportement proprement dit
void Traitement::Action(DéclencheurId pId) {
     ...
}
...

ComposantComportemental* NouveauTraitement(Entité& pEntité) {
     return new Traitement(pEntité);
}
```

Figure 37 : Un composant comportemental à lier dynamiquement

La fonction **`NouveauTraitement`** permet de créer un nouvel objet de la classe liée dynamiquement. C'est la fonction qui est recherchée lors de l'édition de lien dynamique. Le nom de la fonction est créé en accolant le mot Nouveau au nom de la classe du composant comportemental. La fonction est ensuite appelée pour allouer l'objet qui sera associé à l'entité et qui exécutera le comportement proprement dit lorsque la fonction virtuelle **`Action`** sera appelée. Ce mécanisme permet d'allouer dynamiquement un objet dont on ne connaît pas le type lors de la compilation. Nous verrons dans la suite du document que ce mécanisme est beaucoup utilisé pour permettre l'extension de VIPER.

Le grand avantage de ce type de composant comportemental est la rapidité à l'exécution due à la compilation préalable. L'inconvénient est qu'il faut compiler ce comportement sur toutes les machines sur lesquelles on veut pouvoir l'exécuter.

L'idéal serait certainement un composant écrit en Java, compilé en *bytecode* et associé à un *Just in Time Compiler* qui pourrait être intégré à VIPER. Cependant, l'héritage entre une classe C++ et une classe Java n'est pas possible actuellement. En attendant, étant donné que les toutes dernières versions de Tcl (Tcl 8.x) incluent un *Just in Time Compiler,* une version d'ObjectTcl possédant un tel dispositif devrait pouvoir être disponible dans quelques temps et être utilisée par VIPER pour mettre en œuvre des composants comportementaux indépendants du système.

2.4.3. L'état interne d'une entité

Comme nous l'avons présenté dans la partie précédente, les attributs d'une entité (masse, vitesse…) peuvent être définis dans des composants comportementaux. Néanmoins, lorsque des attributs doivent être utilisés par plusieurs composants, on doit les déclarer dans un objet particulier appelé état interne. L'état interne d'une entité est défini par une classe C++ ou ObjectTcl, héritant de la classe EtatInterne (classe très simple servant d'ancêtre commun à tout état interne), qui ne contient que des attributs et des méthodes permettant de les interroger et de les modifier. Tout composant comportemental de l'entité peut avoir accès à l'état interne et peut ainsi y placer des informations qui pourront être récupérées par les autres composants.

2.4.4. Les stimuli et les différents types d'interactions

Nous avons défini un certain nombre de classes de stimuli qui permettent de gérer la plupart des types d'interactions intéressants pour gérer des mondes virtuels complexes. D'autres types d'interactions peuvent, cependant, être définis en héritant des classes de stimuli existantes.

Toutes ces classes héritent d'une classe abstraite appelée Stimulus. Cette classe hérite de la classe ElémentD'agrégat pour que les stimuli puissent être échangés sur le réseau. En accord avec la définition de cette classe, toutes les classes de stimuli devront définir une méthode permettant de les empaqueter dans un message et une méthode pour les dépaqueter. La classe Stimulus contient de plus un attribut qui reçoit une étiquette temporelle correspondant à la date de création ou de dernière modification du stimulus.

Enfin, la classe Stimulus définit une méthode Action qui est appelée à chaque pas de la simulation, par la fonction d'évolution d'un espace de stimuli, pour modifier automatiquement le stimulus en fonction de l'écoulement du temps. Cette méthode pourra servir à réaliser une extrapolation et ainsi permettre la réalisation de fonctionnalités équivalentes à celles d'un algorithme de *dead-reckoning*.

2.4.4.1. Les formes

Les formes sont les stimuli qui permettent des communications visuelles entre entités. Une entité peut avoir une forme graphique qui peut être vue par les utilisateurs. La scène virtuelle est représentée par une hiérarchie de formes géométriques gérée par la librairie graphique utilisée par VIPER (actuellement WTK de Sense8) et décrite dans un ou plusieurs fichiers VRML.

Concrètement, une forme correspond à une portion de la hiérarchie qui est considérée comme privée à l'entité. Cette portion est établie de deux façons. Soit en précisant, lors de la création de la forme, le nom d'un nœud de la scène VRML qui sera l'ancêtre commun à la portion privée (c'est à dire que la portion privée correspond à un arbre dont la racine est le nœud en question. C'est le cas pour la forme 2 de la Figure 38). Soit en précisant une URL permettant d'atteindre un fichier VRML et un nœud de la scène courante qui deviendra le père de la forme. Dans ce cas, un nouveau nœud sera créé, le fichier en question sera récupéré et tous les nœuds racines définis dans ce fichier seront placés sous le nouveau nœud qui deviendra la racine de l'arbre représentant la portion privée à l'entité (cas de la forme 1 de la Figure 38).

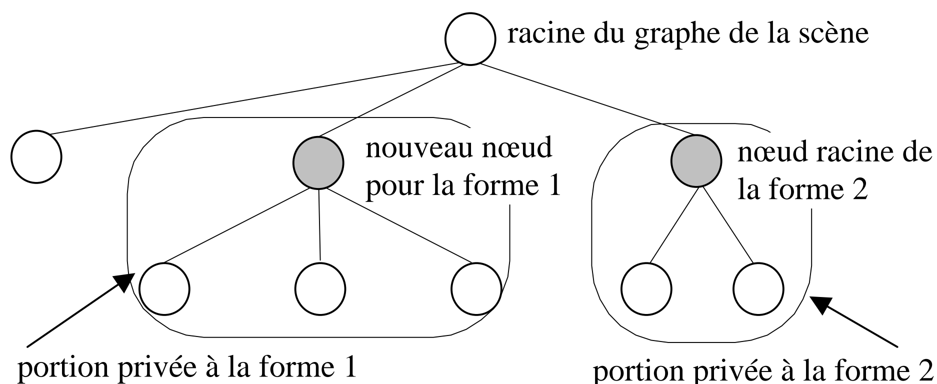


Figure 38 : Graphe de la scène et portions privées de deux formes

En plus de l'identificateur du nœud racine de la portion privée, une forme possède l'identificateur de l'entité qui la possède. Ainsi, lors d'une sélection de la forme par l'utilisateur ou lors d'une collision, on pourra connaître l'entité concernée et interagir directement avec elle. Dans ce cadre, on peut dire que l'espace de stimuli qui gère les formes joue le même rôle qu'un serveur de noms dans un système distribué.

Après sa création par un effecteur spécialisé, la forme pourra être modifiée en précisant quel est le nœud qui est modifié, quels sont les paramètres qui sont modifiés et quelles sont les nouvelles valeurs de ces paramètres. Les paramètres d'un nœud qui peuvent être modifiés dépendent du type du nœud. Par exemple, un nœud Transform qui représente une matrice de transformation a comme paramètres modifiables : sa translation, sa rotation (exprimée par un quaternion) et ses nœuds fils. Un nœud IndexedFaceSet, qui représente une forme géométrique constituée par un ensemble de facettes a comme paramètres modifiables : les sommets de ses facettes et les normales de ses sommets.

La classe Forme définit, de plus, des fonctionnalités permettant de réaliser un algorithme de *dead-reckoning* qui permet de limiter les échanges sur le réseau. Cet algorithme ne s'applique pour l'instant qu'au nœud racine de la forme. La forme contient des attributs permettant de choisir l'algorithme d'extrapolation, de choisir le mode de mise à jour en cas d'erreur (simple téléportation ou interpolation entre la valeur entachée d'erreur et la vraie valeur) et de préciser les paramètres de l'algorithme d'extrapolation et les valeurs correspondant au seuil maximum de l'erreur acceptée. La forme définit aussi la méthode Action qui réalise l'extrapolation en fonction des divers attributs. De plus, si la forme est une copie sur un site distant, Action met à jour le nœud racine de la forme (ainsi la forme va se modifier automatiquement sans que des messages soient envoyés). Lors d'une modification de la position ou de l'orientation de la forme (sur le site d'origine de celle-ci) la nouvelle valeur sera comparée à la valeur extrapolée et si la différence est plus grande que le seuil alors la forme précisera à son effecteur qu'il doit y avoir un message de mise à jour.

Pour l'instant, nous avons défini trois algorithmes tirés de [Pratt 94]. Le premier est un algorithme très simple de degré zéro qui consiste à ne pas extrapoler la position et l'orientation de la forme (Figure 39 a) et qui peut être utilisé par les formes généralement immobiles. Le second est un algorithme du premier degré qui extrapole la position en fonction du vecteur vitesse de la forme (Figure 39 b). Le troisième algorithme, lui aussi du premier degré, consiste à extrapoler, en plus de la position, l'orientation (exprimée par des angles d'Euler) en tenant compte de la vitesse angulaire de rotation correspondant à l'azimut (angle de rotation α exprimé autour de l'axe vertical – axe Y pour WTK) du nœud principal de la forme (Figure 39 b – β est l'angle de tangage c'est à dire l'angle de rotation autour de l'axe X et γ est l'angle de roulis c'est à dire l'angle de rotation autour de l'axe Z). Ce dernier

algorithme est intéressant pour les véhicules qui peuvent tourner avec une vitesse angulaire quasiment constante.

$$a)\quad \begin{cases} P' = P \\ O' = O \end{cases}$$

$$b)\quad \begin{cases} P' = P + \Delta t \times \dfrac{dP}{dt} \\ O' = O \end{cases}$$

$$c)\quad \begin{cases} P' = P + \Delta t \times \dfrac{dP}{dt} \\ O' = \left(\alpha' = \alpha + \Delta t \times \dfrac{d\alpha}{dt}, \beta' = \beta, \gamma' = \gamma \right) avec\ O = (\alpha, \beta, \gamma) \end{cases}$$

Figure 39 : Equations d'extrapolation de formes

Nous n'avons pas défini d'algorithmes basés sur des équations du second degré car ils ne semblent pas très intéressants. En effet, plus l'algorithme d'extrapolation est complexe et plus il nécessitera de ressources de calcul sur chacun des sites participants à la simulation. De même, il est possible d'appliquer les algorithmes d'extrapolation sur n'importe quel nœud privé à la forme.

D'autres classes de stimuli héritant de la classe Forme peuvent être définies pour permettre de réaliser des algorithmes d'extrapolation plus spécifiques et pour permettre de mettre en œuvre des optimisations pour limiter la taille des messages échangés lors des mises à jours. Ce type d'optimisation est mis en œuvre par la première application que nous avons réalisé, pour éviter d'échanger trop d'information lors de la déformation d'un objet.

2.4.4.2. Les icônes sonores

Les icônes sonores sont des stimuli qui permettent à une entité de jouer divers sons lors de l'exécution de son comportement. Ainsi, un véhicule peut jouer un son de moteur lorsqu'il est en marche et un son spécifique lorsque son avertisseur sonore est activé.

Une icône sonore est définie par un nom qui peut représenter l'URL d'un fichier son, le nom d'un nœud WWWAnchor, pointant sur un fichier son, d'un des fichiers VRML de la scène ou encore un échantillon MIDI. De plus, un certain nombre de paramètres peuvent

modifier le son produit. Par exemple, il est possible de positionner le son dans l'espace, de modifier son volume et de le répéter en continu.

Après sa création par un effecteur spécialisé, l'icône sonore pourra être modifiée en précisant de nouveaux paramètres ou en indiquant que le son doit être démarré ou arrêté.

2.4.4.3. Gestion des interactions directes : les ordres

Les interactions directes entre entités sont des interactions qui ne concernent qu'un petit nombre d'entités. On peut citer par exemple la sélection d'un objet par un avatar, le déplacement d'un objet par un avatar, la collision entre deux objets…

Toutes ces interaction sont gérées, dans VIPER, par des classes de stimuli qui héritent de la classe **Ordre**. Un ordre est un type de stimuli spécifique composé de l'identificateur de l'entité qui envoie l'ordre, du nombre d'entités qui doivent le recevoir, des identificateurs de ces entités et de divers paramètres qui dépendent de la classe d'ordre.

Actuellement, nous avons défini les sous-classes d'Ordre suivantes :

- **Attraper** permet de saisir un objet (par exemple avec une main virtuelle). Les paramètres de ce type d'ordre sont : l'identificateur d'un nœud de la hiérarchie, une position et une orientation. Le comportement par défaut lors de la réception d'un tel ordre est de déplacer la forme de l'objet, qui reçoit l'ordre, dans la hiérarchie de la scène de façon à ce qu'il se retrouve parmi les fils du nœud dont l'identificateur est passé en paramètre de l'ordre. La position et l'orientation qui sont en paramètre de l'ordre Attraper seront, de plus, affectés au nœud principal de la forme de l'objet attrapé de façon à ce que l'objet ne change pas instantanément de position et d'orientation réelle (en effet, la position et l'orientation d'un nœud dépendent de la position et de l'orientation des nœuds ancêtres). Par la suite, l'objet va se déplacer automatiquement lorsque la position ou l'orientation d'un de ces ancêtres sera modifiée.
- **Relâcher** est l'ordre dual, qui permet d'annuler le lien précédemment établi. La position et l'orientation qui sont les paramètres de cet ordre permettent de replacer le nœud à sa position ancienne dans la hiérarchie sans modifier la position apparente de l'objet graphique (c'est le comportement par défaut des entités).

- **Sélectionner** est un ordre qui permet de sélectionner et déselectionner un ou plusieurs objets (par exemple avec une souris ou avec un gant). Le comportement par défaut de l'entité consiste à afficher la boite englobante de l'objet avec une couleur passée en paramètre de l'ordre.
- **DéplacerObjetSélectionné** est un ordre qui permet de modifier la position et l'orientation d'un objet précédemment sélectionné. Ce déplacement est ici unique et instantané par opposition aux déplacements qui sont occasionnés par l'ordre Attraper.
- **Acquittement** est un ordre qui permet d'indiquer à une entité que l'ordre qu'elle a précédemment envoyé a été accepté.
- **AcquittementNégatif** permet de refuser un ordre. Ainsi, une entité qui ne veut pas être attrapée peut renvoyer un AcquittementNégatif lors de la réception d'un ordre Attraper. De même, une entité déjà sélectionnée par une autre entité renverra un tel ordre lors d'une autre tentative de sélection.

2.4.4.4. Définition de nouveaux stimuli

Pour définir de nouvelles classes de stimuli il y a, comme pour la définition des classes de composants comportementaux, deux solutions : on peut les définir en ObjectTcl ou en C++ dans une librairie dynamique.

Dans tous les cas, la nouvelle classe hérite de la classe Stimulus ou d'une de ses descendantes. Elle définit un certain nombre d'attributs correspondant à l'information transportée par les stimuli. Elle définit, de même, des méthodes pour modifier ces attributs. Elle peut, en outre, définir la méthode Action. Enfin, la classe doit définir les méthodes pour empaqueter (et dépaqueter) le stimulus dans (et depuis) un message.

2.4.5. Les capteurs et les effecteurs

Comme nous l'avons décrit dans la partie 2.1, les capteurs et les effecteurs peuvent disposer d'un tampon, d'une horloge, d'un filtre et d'un mécanisme spécifique pour interpréter une information ou agir. Deux classes génériques ont été définies pour préciser l'interface de ces mécanismes. La classe Capteur a pour paramètres de généricité : le type des stimuli captés et le type de l'information interprétée. Le filtre est mis en œuvre par une méthode retournant une valeur booléenne qui, appliquée à un stimulus, indique s'il est

récupérable par l'entité. Cette méthode sera appelée par l'espace de stimuli. Le mécanisme d'interprétation est défini dans une méthode Action (la classe Capteur hérite de Déclencheur qui définit cette méthode). La classe Effecteur a pour paramètres de généricité : le type des stimuli créés et le type des actions à exécuter. Le filtre est mis en œuvre par une méthode retournant une valeur booléenne qui, appliquée à un stimuli, indique si celui-ci doit être envoyé dans l'espace de stimuli. Par défaut cette méthode renvoie toujours vrai (dans ce cas il n'y a pas de filtrage). De plus, les deux classes peuvent disposer d'un tampon qui est mis en œuvre par une file bornée (le nombre d'éléments qu'elle peut contenir est limité). Lors de la création du capteur ou de l'effecteur, un paramètre permet d'indiquer si le tampon est utile et doit donc être créé. Enfin, pour gérer les horloges, nous avons défini une classe Horloge qui permet de préciser le capteur qui doit être appelé (sa méthode action sera alors invoquée) et l'intervalle de temps entre deux appels. Cet intervalle n'est qu'indicatif puisqu'il est impossible, au vu du nombre de capteurs qui peut être géré par site, d'utiliser une primitive de synchronisation temporelle (comme les *interval timers* d'UNIX) par capteur. Dans la version actuelle de VIPER, nous utilisons en fait un simple algorithme appelé à chaque pas de la simulation qui détermine si l'intervalle est écoulé et qui appelle dans ce cas la méthode action du capteur. Les capteurs peuvent aussi être appelés explicitement par un composant comportemental de l'entité ou encore être abonnés auprès d'un espace de stimuli qui les invoquera dès qu'une partie de l'espace sera modifié (en fonction du filtre du capteur).

Nous avons, ici aussi, défini des classes de capteurs et effecteurs standards pour gérer la plupart des interactions intéressantes dans le cadre de la réalité virtuelle. Néanmoins, comme pour les stimuli et les composants comportementaux, il est possible de définir de nouvelles classes, en C++ (dans des librairies dynamiques) ou en ObjectTcl, en héritant de la classe Capteur ou de la classe Effecteur.

Les capteurs et les effecteurs servent aussi à la communication entre le monde réel et l'environnement virtuel. Ces communications transitent nécessairement via des clones qui représentent chacun une entité réelle dans le monde virtuel. Ainsi les avatars (clones d'utilisateurs) permettent à des utilisateurs d'interagir dans le monde virtuel. De même des clones d'applications, de robots ou d'autres systèmes électroniques permettent l'échange d'information avec un environnement virtuel.

Il existe, dans ce cadre, deux grands types de capteurs : ceux qui encapsulent des périphériques spécialisés et ceux qui présentent à un utilisateur une « vue » de l'environnement virtuel qui entoure son avatar. Les premiers récupèrent de l'information provenant du monde réel pour la fournir à l'avatar et les seconds présentent à l'utilisateurs des informations provenant de l'environnement virtuel.

Nous allons tout d'abord présenter ces capteurs spécifiques puis nous présenterons des capteurs et effecteurs plus conventionnels qui sont offerts en standard par VIPER.

2.4.5.1. Capteurs encapsulant des périphériques spécialisés

Nous avons défini des capteurs de ce type pour tous les périphériques de réalité virtuelle gérés par WTK. Il existe, par exemple, des capteurs qui gèrent les périphériques servant à calculer la position d'une partie du corps d'un utilisateur (comme le Fastrak de Polhemus) ainsi que des capteurs gérant les divers gants de données existants. Etant donné que WTK gère lui-même ces périphériques, les capteurs que nous avons définis sont très simples et ne font que faire appel aux fonctions de WTK. Néanmoins ces différents capteurs nous permettent de rester le plus indépendant possible de la librairie utilisée. Ainsi, pour la version précédente de VIPER qui utilisait la librairie Performer de SGI, les capteurs géraient directement les périphériques tout en disposant de la même interface.

2.4.5.2. Le capteur « caméra »

Ce capteur sert principalement à préciser la position et l'orientation du point de vue de l'avatar dans le monde virtuel. De plus, il peut permettre de modifier le type d'affichage choisi par l'utilisateur : affichage en mode fil de fer ou face pleine, choix du modèle d'ombrage (plat ou de Gouraud), affichage texturé ou non… Les fonctionnalités supportées, dans ce cadre, dépendent de la librairie graphique utilisée par VIPER.

Ce capteur permet, en outre, pour les formes ayant plusieurs représentations d'en choisir une. Ainsi, une forme figurant un ensemble de données (issues d'un tableur par exemple) pourra définir plusieurs représentations géométriques comme un nuage de point, un camembert ou encore un histogramme parmi lesquelles l'utilisateur pourra choisir celle qui lui convient le mieux.

2.4.5.3. Le capteur d'icônes sonores

De façon similaire, ce capteur sert à préciser la position et l'orientation des récepteurs de sons de l'avatar dans le monde virtuel (c'est à dire des « oreilles virtuelles » de l'avatar). De plus, ce capteur permet de limiter la distance maximale de perception d'un son, de modifier globalement le volume sonore des sons perçus et même d'arrêter momentanément le rendu sonore.

2.4.5.4. Les effecteurs gérant les formes et les icône sonores

Ces deux types d'effecteurs sont très simples et permettent essentiellement de créer initialement une forme ou une icône sonore et de les modifier par la suite. L'effecteur qui gère la forme d'une entité permet, en plus, de prendre en compte un algorithme de *dead-reckoning* pour filtrer l'envoi des messages de mise à jours.

Pour l'instant, un seul stimulus existe pour un effecteur de formes ou d'icône sonore donné. Ce n'est pas un problème pour les formes puisque grâce au nœud *Switch* de VRML on peut combiner plusieurs représentation géométriques pour la même forme. Par contre, pour les sons cela peut paraître limitatif. En effet, une entité peut potentiellement émettre divers sons en fonction de son état interne. Pour permettre cela, avec la version actuelle, il suffit pour l'entité de posséder plusieurs effecteurs d'icône sonore.

2.4.5.5. Les capteurs volumiques

Les capteurs volumiques détectent la présence d'une entité dans leur volume de détection. Le volume de détection du capteur est une forme géométrique qui fait partie de la forme de l'entité et qui possède un nom spécial (commençant obligatoirement par : VCap). Ce nom permet lors de la création d'une entité d'identifier automatiquement ces capteurs, de créer les instances qui vont les gérer et de rendre leur forme géométrique invisible. Ainsi, il est inutile de les déclarer explicitement dans la liste des capteurs et effecteurs d'une entité définie dans le fichier VRML.

N'importe quel composant comportemental peut s'abonner à un capteur volumique. Ainsi, lors de la détection d'une entité, le capteur volumique appellera la méthode Action du composant comportemental qui pourra récupérer des informations fournies par le capteur

volumique (identificateur de l'entité détectée, nœud de la forme de l'entité qui a pénétré à l'intérieur du capteur…) pour réagir à cette détection. Ainsi, ces capteurs peuvent, par exemple, servir à gérer l'ouverture automatique d'une porte lorsqu'une entité s'en approche.

2.4.5.6. Le capteur de détection des collisions

Ce capteur permet d'indiquer au système que l'entité souhaite être prévenue si une collision de sa forme avec une autre forme ou avec le décor est détectée. Dans ce cadre, comme pour les capteurs volumiques, un composant comportemental peut s'inscrire auprès du capteur pour être invoqué dès qu'une collision intervient.

Une entité peut aussi utiliser ce capteur pour éviter que sa forme entre en collision avec le décor ou une autre forme lors d'un déplacement. Dans ce cas, le composant comportemental qui gère les déplacements de l'entité peut simuler un déplacement de la forme, demander au capteur si il y a collision et choisir ou non d'effectuer ce déplacement.

Ce capteur peut aussi être utilisé pour détecter s'il y a intersection entre une demi-droite et la scène. Ce mécanisme peut, par exemple, servir à mettre en œuvre un algorithme de suivi de terrain pour un véhicule terrestre.

Dans tous les cas, lors de la détection d'une collision, le capteur permet d'obtenir l'identificateur de l'entité qui possède la forme intersectée ou un identificateur invalide si on est entré en collision avec le décor. De plus, il permet d'obtenir le nœud intersecté, et pour la détection d'intersection avec une demi-droite, les points d'intersections sur la demi-droite (ordonnés) et la facette intersectée.

2.4.5.7. Le capteur et l'effecteur d'ordres

L'effecteur dépose simplement un ordre dans l'espace de stimuli correspondant. Les ordres qui ont été déposés par un effecteur ne peuvent pas être modifiés par la suite. En fait, tout se passe comme si l'ordre était consommé lors de cette opération.

Un capteur d'ordre reçoit un ordre dès que celui-ci est disponible. Le filtre de réception des ordres permet de ne récupérer que les ordres envoyés à l'entité qui possède le capteur. En fait, le filtrage est optimisé car l'espace de stimuli conserve une liste d'ordres pour chaque capteur (et donc pour chaque entité).

N'importe quel composant comportemental peut s'inscrire auprès d'un capteur d'ordre pour que sa méthode Action soit invoquée lors de la réception d'un ordre. Dans ce cas, le capteur est abonné auprès de l'espace de stimuli. Si aucun composant comportemental n'est inscrit, on peut, tout de même, récupérer les ordres en interrogeant activement le capteur d'ordre depuis un composant comportemental actif.

2.4.5.8. Le capteur de temps

Ce capteur est très simple, il ne possède qu'une horloge et il invoque les composants comportementaux qui sont inscrits auprès de lui. Ce capteur permet donc de définir des comportements actifs disposant d'une fréquence de fonctionnement quasiment fixe. Les autres comportements actifs ont une fréquence variable qui est la plus faible possible compte tenu de la fréquence d'observation du comportement (ils sont appelés à chaque pas de la simulation).

2.4.6. Le cache de composants, états internes, capteurs, effecteurs et stimuli

VIPER gère un cache de tous les fichiers ObjectTcl et de toutes les librairies dynamiques, précédemment utilisées, de façon à accélérer la création et la migration d'entités de même type. En effet, dans ce cas, seule la première création ou migration nécessitera une récupération de fichiers (qui peuvent être importants en taille et donc être lents à récupérer), les opérations suivantes utiliseront les fichiers déjà présents sur la machine (s'il ne s'agit pas de la même session) ou en mémoire (s'il s'agit de la même session).

2.4.7. Exemple d'entité : un avatar gérant un char d'assaut

Dans le but de préciser notre modèle nous allons présenter dans cette partie une entité qui permet à un utilisateur de piloter un char d'assaut et de tirer sur d'autres chars. Cette entité est utilisée dans une petite application qui nous a permis de valider le modèle comportemental de VIPER.

2.4.7.1. Son état interne

L'état interne de l'entité est défini comme suit :

- Une valeur entière gère les points de structure du char. Si la valeur devient égale à zéro, le char explose.
- Un booléen permet de savoir si le char est en fonctionnement ou s'il est détruit.
- Un booléen permet de savoir si le char est en train de tirer.
- Une valeur réelle stocke la vitesse de déplacement du char (c'est une valeur scalaire positive ou négative). Cette valeur est limitée par une autre valeur qui stocke la vitesse maximale (en valeur absolue).

2.4.7.2. Ses capteurs et effecteurs

Cette entité dispose d'un capteur Console qui lui permet de récupérer les ordres de l'utilisateur qui proviennent du clavier et de la souris. Elle possède, de plus, un capteur « caméra » pour permettre à son utilisateur de voir les autres chars ainsi que le décor. Elle possède aussi un capteur de détection de collision pour pouvoir gérer l'algorithme de suivi de terrain et pour pouvoir gérer les tirs. Elle possède un effecteur qui gère sa forme en utilisant un algorithme de *dead-reckoning*. Enfin, elle possède un capteur et un effecteur d'ordres.

2.4.7.3. Ses composants comportementaux

L'entité possède trois composants : un module de tir et de gestion des dégâts, un module de suivi de terrain et un module de pilotage. Le module de pilotage permet à l'utilisateur d'accélérer, de décélérer, de freiner brusquement (la vitesse devient égale à zéro), de tourner et de tirer. C'est ce module qui gère les déplacements de la forme de l'entité.

Le module de suivi de terrain détecte les collisions entre une demi-droite verticale dirigée vers le bas et la scène pour connaître la position du sol. De plus, il utilise la normale à la facette intersectée pour modifier l'assiette du véhicule. Ce module peut être réutilisé par n'importe quelle entité gérant un véhicule.

Enfin, le module de tir détermine, lorsque l'utilisateur décide de tirer, si un char est dans la ligne de tir. Il utilise, lui aussi, le capteur de détection de collision pour réaliser cette opération. Si un char est dans la ligne de tir, le module va envoyer un ordre Tir (une nouvelle sous-classe d'Ordre) à l'entité en question. Cet ordre contient le type de munition utilisé et la distance de tir. Symétriquement, lors de la réception d'un tel ordre, le module va déterminer en fonction des paramètres quels sont les dégâts réels affectés au char et gérer son explosion

éventuelle. Lors de l'explosion, le nœud géométrique représentant l'enveloppe du char sera remplacé par un nœud montrant un char détruit (ceci est réalisé en choisissant parmi deux nœuds placés sous un nœud Switch de VRML).

2.5. Discussion

Les termes que nous utilisons ont été choisis pour ne pas risquer de confusions avec des termes plus généraux. Ainsi nous avons choisi le terme « Entité » plutôt que le terme plus général d'objet car ce dernier a une trop forte connotation venant des langages à objets. D'autre part, comme nous l'avons vu lorsque nous avons abordé la définition des comportements, une entité est en fait mis en œuvre par de nombreux objets C++. Enfin, le terme Entité est déjà utilisé par DIS et SIMNET pour signifier sensiblement la même chose que dans notre système.

De même, nous avons choisi d'utiliser le terme « Stimuli » plutôt que le terme de message car dans notre système, les stimuli peuvent représenter une information persistante. Dans ce cas, la création, les modifications et la destruction des stimuli va potentiellement occasionner l'envoi de nombreux messages qui ne contiendrons, en fait, que des portions d'un ou plusieurs stimuli. Il est clair dans ce cas que le terme de message aurait pu entraîner de nombreuses confusions.

2.6. Conclusion

Nous avons présenté dans ce chapitre, la structure logique qui permet de concevoir une application de réalité virtuelle avec VIPER. La structure proposée est inspiré des modèles définis pour la simulation comportementale et en particulier du modèle In VitrAm [Rainjonneau 92] défini dans notre équipe de recherche. En effet, de nombreux comportements complexes et faisant preuve « d'intelligence » ont été mis en œuvre avec ces différents modèles, ce qui est un gage de généralité et de puissance pour VIPER.

Nous avons aussi décrit, dans ce chapitre, la structure en couche de l'architecture logicielle de VIPER. Cette architecture est générique vis à vis du matériel et du système de communication utilisé. En effet, d'une part, nous avons encapsulé dans la plus basse couche la

mise en œuvre spécifique à un système d'exploitation et à un système de communication, et d'autre part, les diverses bibliothèques de fonctions utilisées par VIPER ont été choisies pour leur existence sur de nombreuses plates-formes. Après avoir étudié les différents modèles de communication des architectures existantes, nous avons choisi le modèle qui nous semble être le plus extensible : une architecture distribuée égal à égal utilisant la communication par groupe. De plus, la principale originalité de notre architecture est de proposer deux niveaux de programmation aux développeurs d'environnements virtuels. Le premier niveau masque l'aspect réparti de l'application qui apparaît ainsi comme séquentielle. Le second niveau permet de choisir et/ou de redéfinir des mécanismes de répartition proposés par VIPER en optimisant la plate-forme pour une classe d'application.

Le modèle proposé pour spécifier un environnement virtuel, en utilisant le premier niveau de programmation offert par VIPER, est aisément extensible car il permet non seulement la définition de nouveaux comportements, mais aussi la définition de nouveaux types d'interactions et de capteurs et effecteurs pour les gérer. Dans ce cadre, nous avons proposé un mécanisme original qui permet de gérer des comportements, stimuli, capteurs et effecteurs mis en œuvre dans des librairies liées dynamiquement au système VIPER. Ce mécanisme permet un bon niveau de dynamisme pour la définition de comportements et leur prototypage rapide tout en conservant une efficacité maximale à l'exécution. VIPER permet, en outre, la définition d'entités indépendantes du matériel grâce au langage interprété ObjectTcl qui est intégré à VIPER.

VIPER permet de gérer simplement la répartition d'un environnement sur une plate-forme distribuée et ceci de façon quasi implicite. Le seul inconvénient dans ce cadre est la nécessité de définir des méthodes d'empaquetage et de dépaquetage de message. Cependant, nous pensons qu'à terme cet inconvénient pourra être évité grâce à un pré-processeur qui générera automatiquement ces méthodes (les méthodes générées seront alors soit compilées avec la librairie dynamique, soit simplement ajoutées au fichier ObjectTcl).

Dans le chapitre suivant nous allons décrire de façon détaillée les aspects répartis de VIPER. Ces aspects, définis dans les trois premières couches de notre architecture, permettent d'offrir la répartition implicite disponible au niveau de la dernière couche que nous venons de présenter.

Chapitre 3. Les aspects répartis de VIPER

Dans le chapitre précédent, nous avons décrit l'architecture générale de VIPER et les choix techniques faits lors de sa conception. Puis nous avons présenté les mécanismes qui permettent de définir un monde virtuel avec VIPER. Ces mécanismes permettent de définir les comportements des entités qui existent dans le monde virtuel et les types d'interactions entre entités gérés par le monde virtuel. Ces mécanismes cachent le plus possible les aspects répartis d'un environnement virtuel distribué. En effet, ils offrent un premier niveau de programmation qui masque la répartition d'un environnement virtuel pour que son développement soit aussi simple que la programmation d'une application orientée-objet séquentielle.

Dans ce chapitre nous allons décrire la répartition d'environnements virtuels avec VIPER. Nous commencerons par présenter les couches basses de VIPER qui permettent de gérer la distribution des données et des calculs. La première de ces couches, la plate-forme répartie, encapsule les aspects dépendants des systèmes d'exploitation et de communication de façon à conserver un bon niveau de généricité pour VIPER. La seconde couche permet de développer simplement des applications réparties grâce à des objets parallèles qui encapsulent la gestion distribuée des applications. Enfin, nous montrerons, dans la suite de ce chapitre, comment les univers virtuels et les espaces de stimuli permettent de résoudre les problèmes de répartition des applications ciblées.

3.1. La plate-forme répartie

La plate-forme répartie encapsule de manière stricte le système de communication en assurant l'évolution depuis la toute première version sur réseau de transputer [Moisan 93a,b] en passant par la version actuelle écrite au-dessus de PVM [Geist 94] et de RMP [RMP 96] vers d'autres plates-formes comme MPI[12], RAMP[13] ou même vers des systèmes répartis

[12] MPI (*Message Passing Interface*) est une interface de programmation d'applications réparties utilisant le passage par messages explicite.

comme Chorus. Cette plate-forme encapsule aussi la gestion des processus utilisés par VIPER. La plate-forme générique est réalisée par un ensemble de classes C++.

3.1.1. Gestion d'un site de la simulation

Chaque site de la simulation est géré par une instance de la classe NœudDuRéseau. Cette classe permet d'affecter à chaque site un identificateur unique qui est composé de l'adresse IP de la machine et de l'identificateur (*pid*) du processus principal qui gère VIPER sur ce site. De plus, cette classe appelle les fonctions d'initialisation des systèmes de communication utilisés par VIPER (actuellement PVM et RMP). Enfin, elle gère la traduction entre l'identificateur de site de VIPER et l'identificateur de site géré par le système de communication (par exemple le task_id pour PVM).

La création de l'instance de cette classe est la première (macro)instruction exécutée par tout programme principal développé avec VIPER.

3.1.2. Gestion de la communication

VIPER utilise deux modes de communication : la communication point à point et la communication par groupes. La communication par groupe est la plus utilisée car elle est très efficace pour limiter les échanges sur les réseaux.

3.1.2.1. La communication point à point

La communication en mode point à point est définie dans sept classes (Figure 40) provenant du précédent prototype de l'environnement de programmation parallèle à objet (qui était réalisé sur un réseau de Transputers). Elles sont basées sur la notion de canal de communication entre deux sites. Les canaux peuvent être orientés ou permettre une communication dans les deux sens. Les classes offrent un contrôle strict à la compilation qui permet d'empêcher, par exemple, les tentatives d'émission sur un canal déclaré pour recevoir des données.

---

[13] RAMP (*Reliable Adaptative Multicast Protocol*) est une interface de programmation d'applications réparties permettant des diffusions sur des groupes de communication IP *multicast* fiabilisées.

Il est important de noter que la notion de canal est indépendante de l'implémentation réelle. Ce qui veut dire que les canaux ne sont pas nécessairement réalisés par des communications point à point en mode connecté. En effet, dans la version actuelle de VIPER, les canaux sont mis en œuvre par PVM qui utilise dans la version TCP/IP le mode non connecté (c'est à dire UDP qui est, de plus, fiabilisé). Ceci est très important pour l'extensibilité du système puisque l'utilisation d'un mode connecté entraînerai la création d'autant de points de connexion (*sockets*) que de canaux alors que la version actuelle ne nécessite qu'un point de connexion par site.

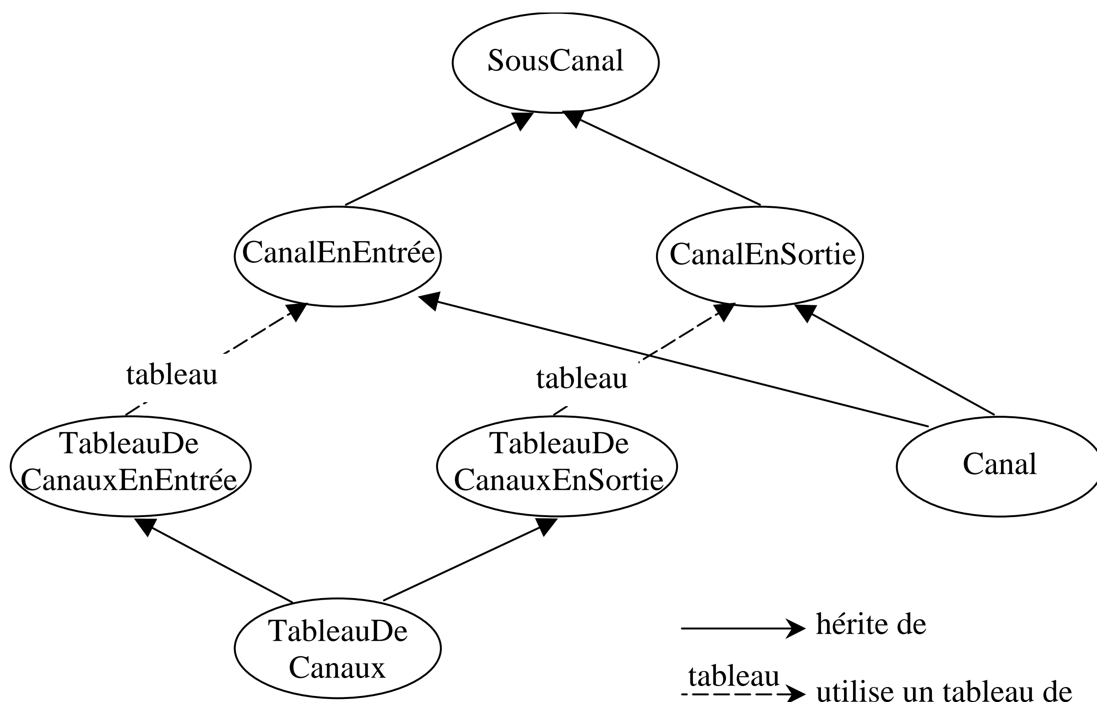


Figure 40 : Les classes qui encapsulent la communication point à point

La classe *SousCanal* contient l'implémentation commune de ses deux sous-classes. Elle comprend, dans la version actuelle, les identificateurs des deux sites reliés par le canal de communication point à point et un identificateur unique de canal qui sert à différencier les messages des différents canaux qui transitent sur la connexion de bas niveau (pour PVM il s'agit d'un *tag* de message). Cette classe permet, de plus, de comptabiliser les opérations effectuées sur le canal (nombre d'octets émis, nombre d'opérations de réception, nombre d'opérations d'émission…). Cette fonctionnalité peut être désactivée par un mécanisme de compilation conditionnelle.

La classe *CanalEnEntrée* définie des méthodes permettant de recevoir des messages de façon bloquante ou non bloquante. Elle permet la réception de valeurs de types de base (char, int, float…) ou de valeurs de types complexes (composés de type de base). Pour la réception bloquante on peut utiliser l'opérateur >>. Ainsi la réception d'un entier sur un canal en entrée « Ce » pourra être écrite : `int Valeur; Ce >> Valeur;` ce qui correspond à une lecture sur un flot C++ (*stream*). De façon symétrique la classe *CanalEnSortie* permet d'envoyer des valeurs de types de base et de types complexes en utilisant l'opérateur <<. Ces classes héritent virtuellement de la classe *SousCanal* pour éviter la duplication de leurs données internes dans la classe *Canal* qui permet la déclaration de canaux utilisables dans les deux sens.

La classe *TableauDeCanauxEnEntrée* permet de réaliser une attente multiple gardée sur N canaux (réception bloquante ou non, contrainte ou non par un délai). Cette fonctionnalité correspond en partie à celle qui est proposée à plus bas niveau par la fonction *select*. L'attente s'effectue par l'intermédiaire de l'opérateur >>, et d'un ensemble de méthodes permettant de définir le *mode de scrutation.* Des statistiques sont ici aussi disponibles (nombre de réponses par canal, nombre d'échecs, nombre de délais expirés…). Des canaux peuvent être rajoutés dynamiquement au tableau qui est en fait mis en œuvre par un vecteur extensible (il s'agit d'une liste de blocs comportant un nombre fixe de canaux, si tous les blocs sont pleins on alloue un nouveau bloc).

La classe *TableauDeCanauxEnSortie* permet, de manière duale, de diffuser un message sur plusieurs canaux. Les deux classes disposent d'un opérateur d'accès (opérateur []) qui permet d'accéder aux fonctionnalités de chacun de leurs canaux internes. Comme pour les canaux, une classe supplémentaire hérite des deux classes précédentes et permet donc d'accéder à l'ensemble de leurs fonctionnalités.

3.1.2.2. La communication par groupe

La communication par groupes a été rajoutée à l'environnement de programmation initial pour permettre des communications plus efficace lorsque cette fonctionnalité est gérée par les protocoles réseaux sous-jacents. A la différence de la communication point à point, nous proposons plusieurs mises en œuvre de ce type de communication. Pour gérer de façon

uniforme ces différentes implémentations nous avons défini trois classes abstraites qui présentent l'interface commune aux différentes implémentations (Figure 41).

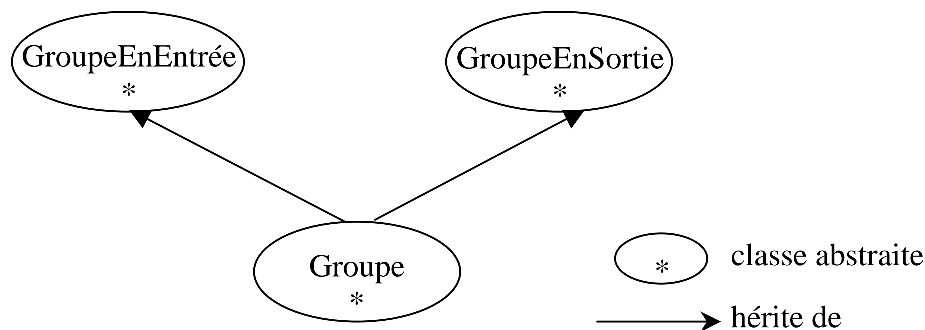


Figure 41 : Classes abstraites qui encapsulent la communication par groupe

La première mise en œuvre utilise une fonctionnalité de PVM qui permet de réaliser une *pseudo* communication par groupes en utilisant plusieurs communications point à point et un serveur qui gère l'appartenance aux différents groupes. Lors d'une émission sur un groupe nommé, une requête est envoyée au serveur de groupes pour demander quels sites appartiennent au groupe. Dès la réception de cette information, le message sera envoyé individuellement à chaque site. Cette fonctionnalité est intéressante car elle permet de mettre en œuvre une communication par groupe dans des architectures ou celle-ci n'est pas prévue. C'est surtout le cas pour des multiprocesseurs à mémoire locale comme les réseaux de Transputers.

Néanmoins, pour permettre une communication plus efficace sur le MBone, nous proposons aussi une mise en œuvre utilisant les groupes de communication d'UDP. Ce type de communication étant à l'origine non fiable nous avons choisi RMP pour permettre différentes qualités de service (fiabilité, conservation de l'ordre d'émission depuis chaque site, conservation d'un ordre total d'émission sur le groupe…).

Enfin, pour permettre une communication la plus efficace possible sans garantie de qualité de service nous proposons une mise en œuvre directement au dessus des groupes *mulitcast* UDP. En effet, les mécanismes définis par RMP ne sont malheureusement pas totalement débrayables et la connexion à un groupe RMP est trop lente pour être réalisée dynamiquement et en temps réel pendant l'exécution du système.

Pour pouvoir gérer des architectures hétérogènes dans les deux derniers types de groupes (PVM gère directement l'hétérogénéité), on utilise la librairie XDR (*eXternal Data Representation*) [XDR 87] qui définit une représentation standard pour des données scalaires (entiers, flottants, caractères…) et pour des données structurées (tableaux, chaînes de caractères…). Les données sont codées sur le site d'émission en suivant cette représentation pour être échangées sur le réseau et sont décodées sur le site de réception. XDR permet notamment de gérer les problèmes d'ordre des octets dans un mot (*little endian* versus *big endian*).

Pour garder une écriture générique des classes d'objets parallèles nous avons choisi d'utiliser un mécanisme présenté dans la figure suivante (Figure 42). La classe d'objet parallèle qui doit utiliser des groupes de communications va utiliser (par référence) les classes abstraites ainsi qu'un créateur de groupes. A chaque fois que l'objet parallèle à besoin d'un groupe de communication il va demander une nouvelle instance à son créateur de groupe. Une instance d'une classe concrète (par exemple *CréateurDeGroupesRMP*) héritant de la classe *CréateurDeGroupes* sera passée en paramètre du constructeur de l'objet parallèle et ainsi créera les instances voulues (par exemple des *GroupesRMP*).

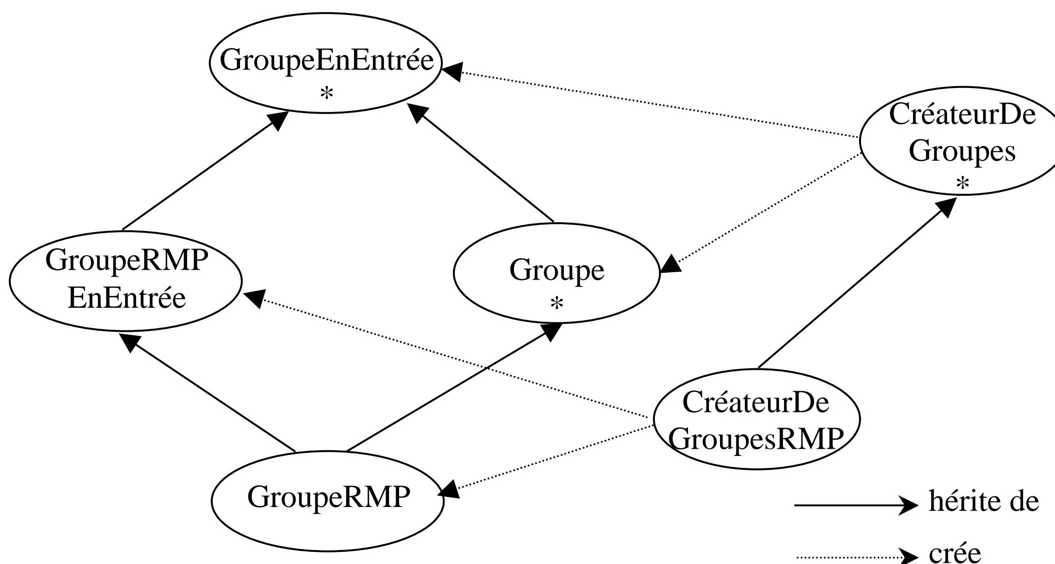


Figure 42 : Classes concrètes qui encapsulent la communication par groupe de RMP

3.1.3. Gestion des processus

Nous utilisons deux types de processus : les processus standards et des processus de poids léger (*threads*). Les processus normaux représentent les sites (ou nœuds de calcul) sur lesquels le système s'exécute. Chaque processus normal peut contenir des processus de poids léger qui ont pour particularité de se partager un même espace mémoire. Les deux classes suivantes encapsulent la gestion de ces deux processus qui dépend du système d'exploitation. Ainsi, on garde une bonne indépendance par rapport au système utilisé.

3.1.3.1. Les processus standards

La classe *Processus* qui encapsule ce premier type de processus permet la création d'un nouveau processus sur la machine locale ou sur une autre machine sur laquelle le système de communication (actuellement PVM) s'exécute. Cette classe permet aussi de détruire un processus local ou distant.

L'intérêt de cette classe est de permettre un contrôle de l'application répartie depuis un seul site. Cette fonctionnalité est très intéressante pour les applications qui sont gérées par une seule personne sur un site géographique comportant plusieurs machines (exemple : une batterie de simulateurs de vols en réseau gérée par un seul administrateur).

3.1.3.2. Les processus de poids léger

La classe *Ppl* encapsule la gestion de ce type de processus et permet la création, la destruction et le blocage d'un processus de poids léger. On ne peut créer de processus de poids léger que dans le processus courant. De même, la destruction et le blocage sont limités aux processus de poids léger du processus courant.

3.2. Les objets parallèles

3.2.1. Un modèle de programmation parallèle à objets

Le modèle de programmation parallèle à objets développé dans l'équipe a été fortement modifié et étendu pour être plus général et s'adapter aux nombreux besoins de la réalité virtuelle distribuée. Nous allons présenter dans cette partie le nouveau modèle ainsi défini.

Le but du modèle reste sensiblement le même : simplifier le développement d'applications réparties grâce à des agrégats qui gèrent de façon implicite la distribution de leurs éléments et l'accès distant à un élément. Néanmoins, nous nous sommes quelque peu éloignés du modèle SPMD initial. En effet, le modèle actuel est beaucoup plus dynamique et permet l'ajout et le retrait dynamique de sites à l'application.

Pour gérer la distribution implicite des données, le modèle utilise deux notions importantes : le **domaine** et la **fonction de répartition**.

Le domaine représente l'ensemble des sites du réseau sur lesquels les données d'un agrégat sont réparties. Le concept de domaine correspondait, dans le modèle original, à une sorte de machine virtuelle dotée d'un réseau de communication privé totalement maillé. Il correspond maintenant à un groupe dynamique de sites qui participent à la gestion de l'agrégat reliés soit par un maillage de communication point à point soit par un ou plusieurs groupes de communication.

La fonction de répartition décrit l'algorithme exact de répartition des données sur le domaine. C'est une méthode virtuelle qui est concrètement définie dans chacune des classes d'agrégats génériques (vecteurs, matrices, listes...). Ce choix d'implantation permet une excellente encapsulation de la répartition des données tout en permettant de définir incrémentalement de nouveaux types de répartition en définissant de nouvelles classes d'agrégats par héritage.

Le modèle définit, de plus, le concept de **fonction parallèle**. Il s'agit d'une méthode applicable sur un agrégat qui prend en paramètre un nombre quelconque d'agrégats du même type (et ayant typiquement le même domaine) et qui s'applique simultanément à chacun des

éléments de l'agrégat résultat. Néanmoins, bien que cette fonction s'applique parallèlement sur tous les sites de l'agrégat, elle est appliquée séquentiellement sur tous les éléments locaux sur un site donné.

Les agrégats dotés de fonctions parallèles sont appelés **objets parallèles**. Ce concept unifie donc, la répartition implicite des données avec un parallélisme, également, implicite. Il y a plusieurs types de fonctions parallèles qui sont chacune de complexité différente :

- Dans le cas le plus simple, l'application sur chacun des éléments se fait de façon indépendante (par exemple, la remise à zéro des éléments d'un tableau).

- La fonction doit s'appliquer de façon sérialisée (exemple : sauvegarde des éléments d'un vecteur qui doit conserver l'ordre des éléments). Dans ce cas, il est nécessaire de disposer de primitives de synchronisation globale sur l'agrégat. Pour gérer ce type de fonction parallèle nous avons défini un anneau à jeton logique qui utilise des canaux de communications.

- Il existe des dépendances entre les éléments et il faut éviter les effets de bords indésirables. Dans ce cas, la solution choisie est de considérer l'agrégat résultat comme différent des agrégats paramètres de l'opération (il y a création d'un nouvel agrégat).

De plus, pour permettre la réalisation de fonctions parallèles qui ont besoin, lors de leur application, d'accéder aux éléments non locaux nous avons défini la notion d'objet parallèle **actif**. Ce type d'objet parallèle permet d'accéder aux objets en lecture ou en écriture. Pour permettre ces accès, ces objets parallèles disposent d'un ensemble d'outils de communication (des canaux ou des groupes) et d'un serveur, mis en œuvre par un processus de poids léger, qui sur chaque site gère les accès distants. L'intérêt de ce serveur est triple : il permet de gérer les requêtes en lecture des sites distants dès qu'elles arrivent sur le site, il permet d'éviter des situations d'interblocage dans le cas d'accès synchrones en lecture sur des sites distants (par exemple, si un site A demande la valeur d'un élément au site B qui lui même demande une valeur au site A, sans serveur les deux sites se retrouveraient bloqués en attente de message alors qu'avec le serveur ce problème n'existe pas. Cf. Figure 43) et il permet, enfin, d'éviter des pertes de messages dues à des débordements de file d'attentes. De plus, les éléments d'un objet parallèle devront être communiquables. C'est à dire qu'ils devront hériter d'une classe

appelée ElémentD'agrégat qui leur permet de définir une méthode pour les « empaqueter » dans un message et une méthode pour les retirer d'un message (les « dépaqueter »).

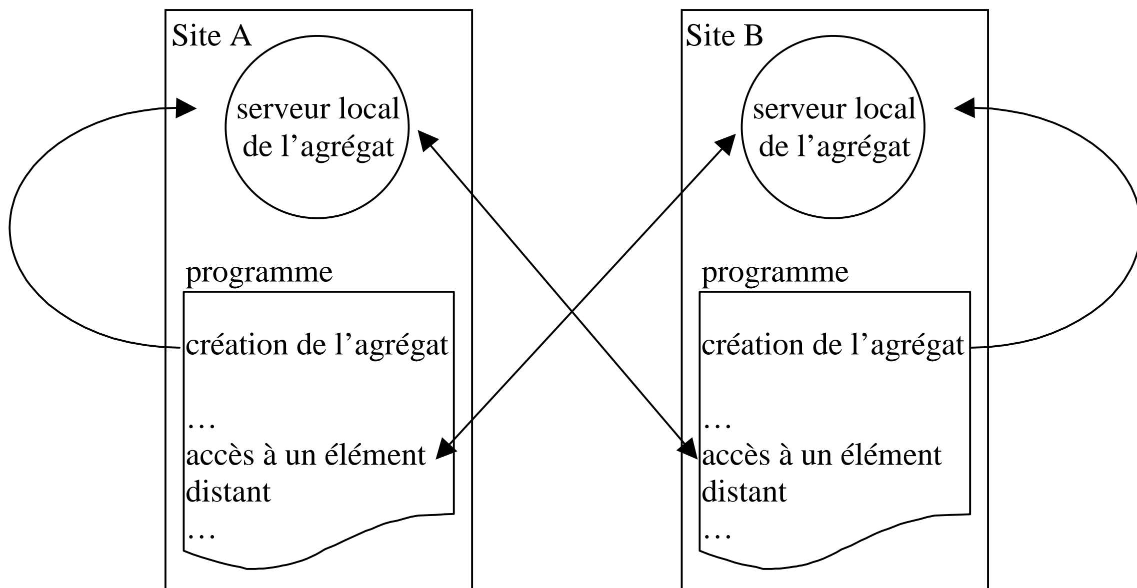


Figure 43 : Evitement de l'interblocage grâce aux serveurs locaux

Symétriquement, les objets parallèles **passifs** ne permettent pas d'accéder aux éléments distants et sont donc plus simple à mettre en œuvre et moins coûteux à l'exécution (pas de serveur local).

Les différentes fonctionnalités des objets parallèles que nous avons développés peuvent se résumer sur le graphe de classes suivant (Figure 44) :

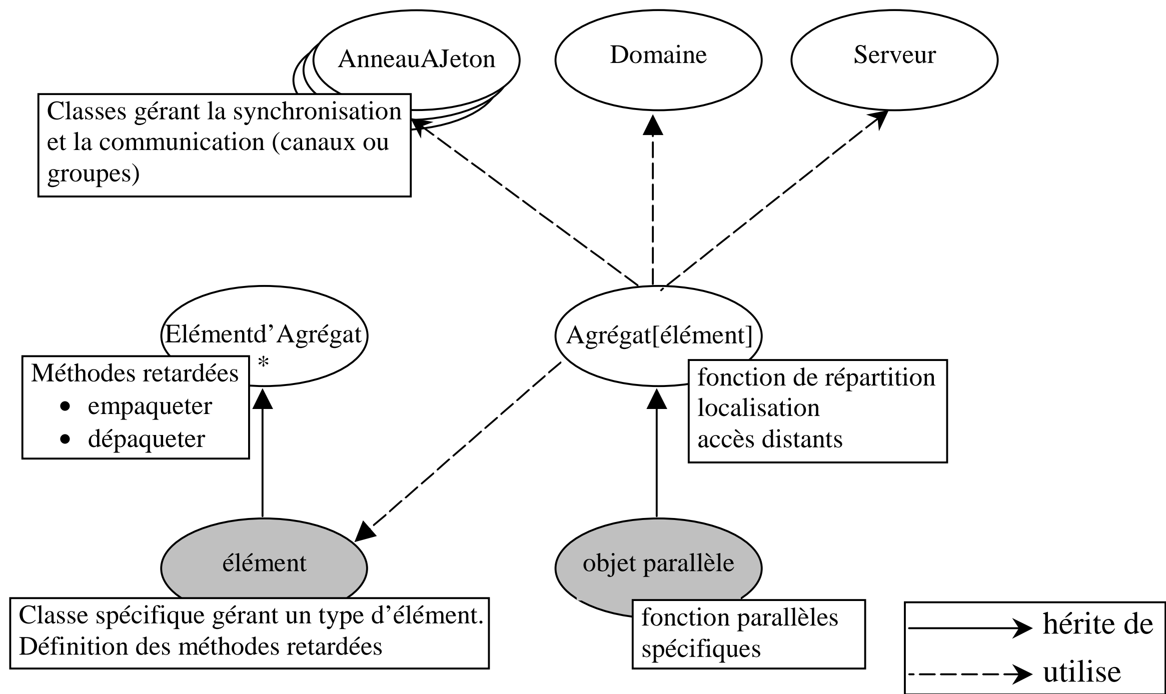


Figure 44 : Graphe de classes gérant un objet parallèle

La classe Agrégat[élément] est construite automatiquement à partir de la classe d'agrégat générique gérant la structure de données souhaitée. Les seules classes spécifiques à un type d'élément sont la classe de l'objet parallèle et, bien évidemment, la classe correspondant aux éléments.

Le travail le plus important est réalisé dynamiquement à la création d'un objet parallèle. L'appel au constructeur alloue la mémoire nécessaire pour stocker les éléments locaux de l'objet sur le site courant (ceci est réalisé avec un coût mémoire minimum). Les outils de communications sur le domaine sont créés et, pour les objets actifs, un serveur local est lancé pour assurer l'accès distant à partir des autres sites.

Par la suite, chaque méthode appelée sur un objet est filtrée par un processus de contrôle d'accès (en tenant compte de la fonction de répartition interne) qui indique la marche à suivre pour la localisation des données, le calcul et la conversion des adresses et l'accès distant si nécessaire.

Pour définir un comportement réparti plus spécifique, avec de simples mécanismes d'héritage, nous avons dérivé trois classes d'objets parallèles : les objets partitionnés, les objets dupliqués et les objets centralisés.

3.2.1.1. Les objets partitionnés

Dans ce cas l'ensemble des données est partitionné de telle manière que chacune des partitions se trouve sur un site donné du domaine. La création des partitions est réalisée en suivant une méthode de répartition virtuelle (qui sera définie par les classes descendantes de la classe des objets partitionnés). De plus, des éléments peuvent être créés et détruits dynamiquement sur chaque site.

La variante active de ces objets permet d'accéder à n'importe quel élément local ou distant, depuis n'importe quel site du domaine de l'objet parallèle. Ces accès sont réalisés de manière transparente. En effet, on utilise une même méthode pour accéder à un élément que ce soit en local ou à distance. Par exemple, pour les accès en écriture la méthode ressemblera à : `Mat[i,j].Ecrire(element)` pour une matrice et, suivant l'adresse (c'est à dire son identificateur de site) de l'élément repéré par les indices i et j, l'accès sera soit local soit distant. De plus, lors de la création ou de la destruction d'un élément local, les autres sites seront prévenus de façon à ce qu'ils soient, au minimum, au courant du nombre total d'éléments de l'agrégat.

La mise en œuvre des accès distants, des créations et des destructions est assurée par plusieurs tableaux de canaux et par un serveur. L'objet parallèle va posséder, sur chaque site, un serveur et un tableau de canaux en sortie pour communiquer avec les serveurs de tous les autres sites. Ces canaux sont utilisés pour envoyer les requêtes d'accès, la valeur de l'élément modifié dans le cas d'un accès en écriture et les avis de création et de destruction. L'objet va, de plus, posséder sur chaque site un tableau de canaux en entrée pour recevoir le résultat d'une lecture distante.

Symétriquement, le serveur va posséder un tableau de canaux en entrée pour recevoir les diverses requêtes, les avis et les valeurs à modifier. Il va aussi posséder un tableau de canaux en sortie pour envoyer la valeur demandée par un accès distant en lecture. La boucle de fonctionnement du serveur consiste à réaliser une attente bloquante sur l'ensemble des canaux, à gérer une requête dès qu'elle est reçue et à retourner à l'attente bloquante.

Lorsqu'un site se retire de la simulation, le serveur et l'objet parallèle sont prévenus et ils notent que les canaux correspondants peuvent être réutilisés (ceci est réalisé sur chaque site). De même, ils sont prévenus lorsqu'un nouveau site entre dans la simulation. Dans ce cas, soit

il y a réutilisation de canaux existants qui serviront par la suite à communiquer avec le nouveau site, soit il y a création de quatre nouveaux canaux (deux en entrée, deux en sortie) qui sont rajoutés aux tableaux de canaux.

On peut remarquer que dans le cas d'une simulation comportant de nombreux utilisateurs ce mécanisme va nécessiter un nombre très important de canaux par site. Néanmoins, il est important de rappeler que ces canaux ne nécessitent pas chacun un *socket* et ne sont en fait là que pour stocker l'adresse des différents sites participant à la simulation et éventuellement pour gérer quelques statistiques de communication. Malgré cela, nous pensons, à terme pour gérer de très grands nombres d'utilisateurs, concevoir des tableaux de canaux qui géreront l'émission et la réception de messages sans nécessiter de canaux pour chaque site. Dans ce cas, il faudra préciser à chaque émission, l'adresse du site de réception.

3.2.1.2. Les objets dupliqués

Les objets dupliqués sont une restriction des objets partitionnés. Leurs données internes sont dupliquées sur chaque nœud du domaine et, dans le cas d'objets actifs, les changements sont répercutés au reste du domaine de manière transparente. Il est important de noter que chaque élément de l'agrégat est assigné à un site de référence (par la fonction de répartition) et que, dans le cas d'un objet actif, seul ce site est autorisé à modifier directement l'élément. Si un site non autorisé veut modifier l'élément, il devra en faire la requête au site de référence. Cette restriction, qui a été introduite pour simplifier la gestion des objets dupliqués actifs, est le comportement par défaut qui peut être modifié dans les classes descendantes. De plus, nous montrerons dans la suite de ce chapitre que tous les agrégats utilisés à l'heure actuelle par VIPER ont été conçus de façon à modifier chaque élément uniquement sur son site de référence.

Les objets dupliqués actifs sont mis en œuvre grâce à un groupe de communication et à des serveurs qui gèrent la communication par groupe (la classe de ces serveurs, appelée ServeurPourGroupe, hérite de la classe serveur que nous avons précédemment décrite). L'objet parallèle va posséder un groupe en sortie pour envoyer les requêtes d'accès en écriture et les avis de création et de destruction d'élément (les accès en lecture sont réalisés en local puisque les informations sont dupliquées). Dans le cas, d'un objet dupliqué actif, on peut noter que les requêtes d'accès en écriture servent aussi bien à synchroniser les éléments

dupliqués qu'à demander la modification d'un élément dont le site de référence n'est pas local. Néanmoins, cette dernière utilisation est coûteuse puisque la requête est envoyée à chaque site et pas uniquement au site de référence. Ce n'est pas un problème à l'heure actuelle puisque nous n'utilisons pas cette fonctionnalité qui n'a été conçue que pour rester le plus général possible.

Le serveur va posséder un groupe en entrée pour recevoir les requêtes d'accès en écriture et les avis de création et de destruction d'élément. Le fonctionnement du serveur est quasiment identique à celui des serveurs d'objets partitionnés actifs. La seule différence est que l'attente bloquante est réalisée sur un groupe de communication.

Comme nous l'avons décrit dans la partie 3.1.2.2, les objets dupliqués actifs utilisent un créateur de groupes qui leur est fourni via leur constructeur. Ce créateur permet de ne pas lier le fonctionnement d'un objet dupliqué à une mise en œuvre particulière de la communication par groupe.

3.2.1.3. Les objets centralisés

Les objets centralisés ont leurs données internes centralisées sur un nœud unique. Ces objets n'existent qu'en variante active qui permet d'accéder aux éléments depuis n'importe quel site du domaine (en lecture et en écriture). Ce type d'objet parallèle est utile pour implanter des accès à des ressources ou à des périphériques de manière transparente.

Nous avons ajouté à ces objets, une requête d'abonnement qui permet à n'importe quel site de demander que les valeurs de certains éléments lui soient automatiquement envoyés quand ils sont modifiés. Les données sont dans ce cas, stockées dans une file d'attente, sur le site abonné, jusqu'à ce quelles soient consommées. Il est important dans ce cadre, que ces données soient consommées le plus rapidement possible pour ne pas encombrer les files d'attentes du système de communication.

Ce mécanisme est utile dans le cadre de la RVD pour gérer l'accès transparent à un périphérique spécialisé tel qu'un gant de données qui se trouve sur une machine différente de celle qui gère l'avatar d'un utilisateur. Dans ce cas, le site de l'avatar va s'abonner aux mises à jour des données du gant (position de la main, flexion des doigts…) qui lui seront

automatiquement envoyées et qui seront appliquées à chaque itération de la boucle de simulation.

La mise en œuvre de ces objets utilise un seul serveur avec ses tableaux de canaux (sur le site de référence) et pour chacun des autres sites une paire de canaux (un en entrée, l'autre en sortie). La paire de canaux sert à envoyer les requêtes de lecture, d'écriture et d'abonnement (et la valeur à écrire) et à recevoir le résultat d'une requête de lecture ou les données émises après un abonnement. Le serveur (qui est une instance de la classe ServeurCentral qui hérite de la classe Serveur) fonctionne quasiment de la même façon que celui d'un objet partitionné en notant en plus les sites abonnés à chaque élément dans un ensemble de listes spécifiques (il peut y en avoir, au maximum, une par élément). Enfin, lorsqu'une requête d'écriture est traitée par le site de référence (par l'objet ou par son serveur), la liste de site abonnés (si elle existe) est consultée et des messages contenant la nouvelle valeur sont envoyés aux différents sites. En fait, nous n'utilisons pas dans ce cas de groupes de communication car le nombre de sites abonnés est typiquement faible et donc ce serait dommage de « gaspiller » une ou plusieurs adresses *multicast*.

La figure suivante (Figure 45) résume les caractéristiques des différentes sous-classes d'objets parallèles que nous avons réalisées.

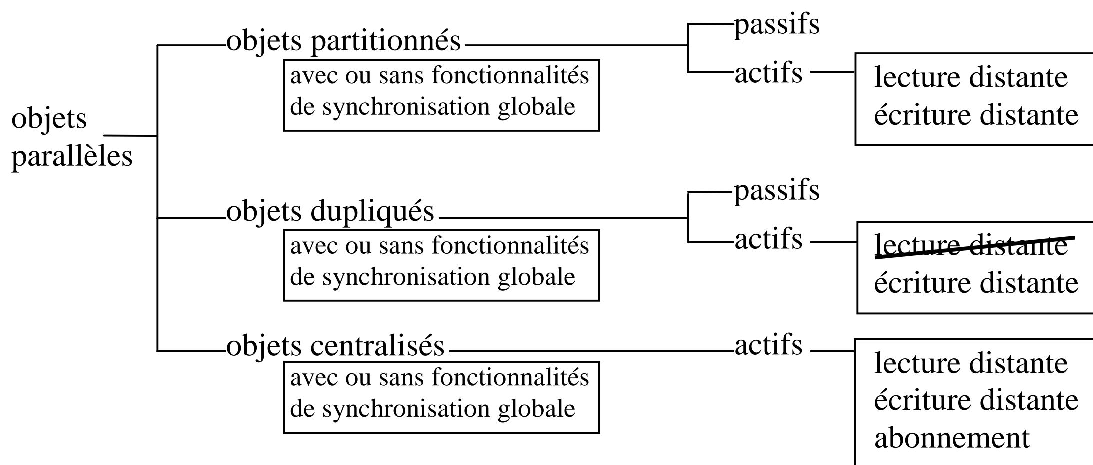


Figure 45 : Résumé des caractéristiques des différentes classes d'objets parallèles

3.2.2. Structures de données distribuées mises en œuvre

Nous avons défini un ensemble de classes dérivées des classes d'objets partitionnés, dupliqués ou centralisés qui permettent de répartir diverses structures de données. Pour la plupart des structures que nous avons mises en œuvre, il existe cinq sous-classes correspondant aux différentes fonctionnalités des objets parallèles. Ainsi pour la structure de données vecteur, nous avons défini des vecteurs partitionnés actifs, des vecteurs partitionnés passifs, des vecteurs dupliqués passifs, des vecteurs dupliqués actifs et des vecteurs centralisés.

Les structures de données simples mises en œuvre sont, en plus des vecteurs, des matrices et des matrices 3D. Nous avons, de plus, défini des structures de données évoluées comme des vecteurs extensibles (on peut leur ajouter des éléments), des vecteurs de listes, des vecteurs de files, des vecteurs de files gérant une priorité et des dictionnaires (sortes de tables de *hashing*).

Nous détaillerons l'implémentation de chacun des agrégats utilisés par VIPER dès qu'ils seront introduits pour gérer un type précis d'élément.

3.3. Le serveur de connexions

Comme nous l'avons précisé dans le chapitre précédent, les univers virtuels gérés par VIPER sont définis dans des fichiers VRML. Pour qu'un utilisateur puisse « pénétrer » dans un de ces univers virtuels, il lui faut connaître l'URL de son principal fichier VRML. Il doit de plus connaître le nom d'une machine qui fait fonctionner un serveur de connexions spécifique à VIPER. En effet, pour qu'un site puisse participer à une session de simulation de VIPER il lui faut connaître un certain nombre d'adresses de groupes de communication permettant d'échanger des messages utiles à la simulation d'un univers virtuel. Ces informations auraient pu être placées dans le principal fichier VRML d'un univers virtuel mais, dans ce cas, on n'aurait pas pu gérer plusieurs sessions distinctes utilisant le même univers virtuel. C'est en partie pour cette raison que nous avons choisi d'utiliser un serveur de connexion pour VIPER. De plus, si cela est nécessaire, le serveur peut permettre de limiter les accès à un ou plusieurs univers virtuels grâce à des mots de passe ou à un filtrage sur les adresses IP des machines des utilisateurs.

Un serveur peut gérer un ou plusieurs univers virtuels. Pour stocker toutes les informations spécifiques à ces univers virtuels, le serveur utilise un dictionnaire dont la clé (utilisée par la fonction de *hashing*) est une URL et les données sont polymorphes et dépendent du type de l'URL. En fait, le serveur et la partie gérant la connexion de chaque application construite avec VIPER sont en fait mis en œuvre par un objet parallèle héritant de la classe dictionnaire centralisé. Ce dictionnaire est un peu particulier puisque son domaine recouvre automatiquement tous les sites participant à toutes les sessions de simulation gérées par le serveur de connexion et même les sites en cours de connexion.

Le premier type d'information contenu dans le dictionnaire est une structure identifiée par l'URL du fichier VRML principal de l'univers virtuel qui contient un préfixe de 16 bits partagé par toutes les adresses IP des groupes de communication de l'univers virtuel. Ainsi chaque univers virtuel peut posséder au maximum 65 536 groupes de communications ce qui devrait être amplement suffisant. Le préfixe de 16 bits doit être attribué de manière unique probablement par l'organisme qui gère la machine serveur de connexion (l'attribution des adresses IP *multicast* n'est pas, à notre connaissance, traitée de façon officielle par les organismes qui gèrent l'Internet). Le suffixe de 16 bits est lui géré directement par le serveur de connexion.

Lors de la première requête de connexion à un univers virtuel, une entrée spécifique est créée dans le dictionnaire. Un fichier de description de l'univers virtuel qui décrit les objets parallèles (univers virtuels distribués et les espaces de stimuli distribués) utilisés est chargé (ce fichier qui possède le même nom que le fichier VRML principal de l'univers virtuel et une extension différente est stocké localement sur la machine qui gère le serveur de connexion). Ensuite, pour chaque groupe de communication utilisé par l'univers virtuel, une adresse IP *multicast* et un numéro de port sont choisis (séquentiellement en commençant par 0.0 pour les 16 bits de l'adresse IP et par 8000 pour le numéro de port). De même, pour les objets parallèles qui utilisent des canaux de communication, un identificateur de canal est choisi (cet identificateur – appelé *tag* de message par PVM – sert à multiplexer/démultiplexer les messages concernant plusieurs canaux sur une même connexion UDP/IP). Toutes ces informations sont ensuite stockées dans une structure qui est placée dans le dictionnaire (avec pour clé l'URL principale du monde virtuel) et envoyées au premier site. Par la suite, lorsqu'un autre site se connecte au même univers virtuel, la structure lui est envoyée. Chaque

site utilise les informations fournies dans un ordre précis de façon à ce que les adresses IP, les numéros de port et les identificateurs de canaux soient identiques sur tous les sites.

Le dictionnaire du serveur de connexion gère aussi l'affectation d'un identificateur unique aux nouveaux types de stimuli définis dans des fichiers ObjectTcl ou dans des librairies dynamiques. Les identificateurs (représentés par un entier sur 16 bits) sont alloués séquentiellement en commençant par la valeur 1024 (les valeurs inférieures sont réservées pour les types de stimuli standards de VIPER).

3.4. Les univers virtuels distribués

Pour distribuer un environnement virtuel on peut commencer par envisager la répartition de ses entités. VIPER utilise pour cela des univers virtuels distribués (UVD). Chaque type d'UVD est un agrégat d'entités. Il définit une fonction de nommage d'entité, qui attribue à chaque entité un identificateur unique sur le réseau. Actuellement, les identificateurs d'entités sont composés d'un identificateur d'UVD, suivi de l'identificateur du site de création de l'entité suivi d'un nombre séquentiel. Un UVD définit, de plus, la fonction de répartition qui permet, à partir d'un identificateur d'entité, de trouver le site qui gère l'entité à un instant donné. D'autre part, certains UVD offrent des méthodes permettant d'émettre et de recevoir des entités via le réseau. La figure suivante (Figure 46) présente trois classes d'univers virtuels distribués que nous allons détailler dans la suite de cette partie.

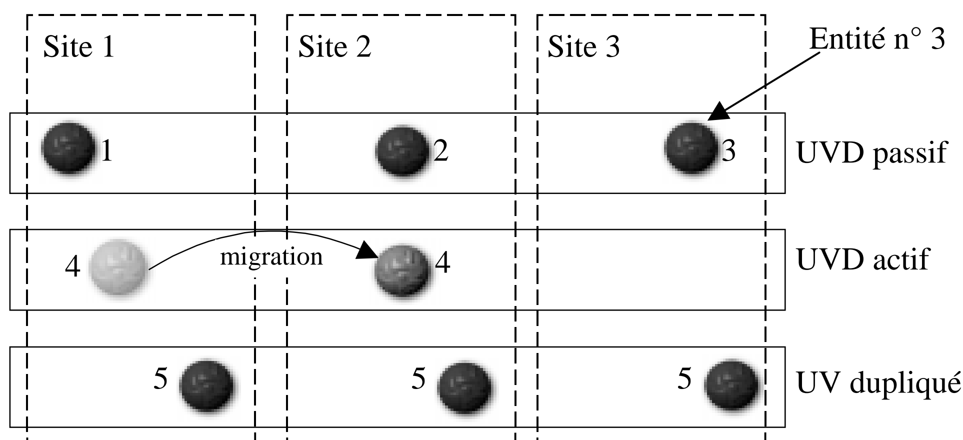


Figure 46 : Cinq entités gérées par trois univers virtuels distribués

3.4.1. Les univers virtuels distribués passifs

La classe la plus simple d'UVD est l'univers virtuel distribué passif. Cet UVD contient des entités qui sont liées à leur site de création et ne peuvent pas migrer. Il permet la création dynamique d'entités sur les sites de son domaine et l'ajout/le retrait de sites de son domaine. Les avatars appartiennent à ce type d'UVD car ils sont obligés de rester sur la station de travail de l'utilisateur qu'ils représentent.

Ces UVDs sont mis en œuvre par des listes partitionnées passives d'entités. En fait, sur chaque site il y a une liste d'entités qui évolue indépendamment des autres listes. Ces UVD offrent une fonction de répartition très simple qui extrait l'identificateur de site de l'identificateur d'entité.

3.4.2. Les univers virtuels distribués actifs

Lorsque deux entités d'UVD passifs interagissent, des communications au travers du réseau peuvent être mises en œuvre (si elles sont localisées sur des sites différents). Ceci peut poser un problème si la latence introduite par ces communications est importante. En effet, ceci peut entraîner une interactivité réduite. Pour résoudre ce problème nous avons introduit une classe d'UVD appelée les UVD actifs [Torguet 96]. Ce type d'UVD permet aux entités de migrer d'un site à l'autre. Pour qu'une entité puisse migrer d'un site à un autre il faut que ce soit une instance de la classe **EntitéMigrante**.

Ainsi, la migration peut être utilisée pour « rapprocher » deux entités qui interagissent. Par exemple, si un utilisateur manipule un objet virtuel, l'UVD va faire migrer l'entité représentant l'objet sur le site de l'avatar de l'utilisateur. De cette façon l'interactivité du point de vue de l'utilisateur sera grandement augmentée.

Les UVD actifs offrent une fonction de répartition qui permet à partir d'un identificateur d'entité de localiser son site actuel. Cette fonction est réalisée grâce à un dictionnaire. Il s'agit d'une sorte de table de *hashing* qui permet d'obtenir, à partir d'un identificateur d'entité, l'identificateur du site actuel de l'entité. Ce dictionnaire d'entité hérite de la classe parallèle, dictionnaire dupliqué, de façon à ce que les modifications de site d'une entité soient automatiquement synchronisées sur chaque site. Cette synchronisation est réalisée grâce à un

groupe de communication privé. De plus, lors d'une modification de site d'une entité précédemment locale, le dictionnaire prévient son UVD actif. Ce comportement constitue une spécialisation de la classe dictionnaire dupliqué qui est utilisée pendant la migration.

Les entités sont, elles, placées dans une liste partitionnée active d'entités. En fait, l'UVD actif hérite de la classe parallèle liste partitionnée active. La mise en œuvre de cette classe parallèle implique qu'une liste existe sur chaque site et que ces listes peuvent évoluer différemment. Néanmoins, pour l'UVD actif, nous avons modifié le comportement par défaut de la liste partitionnée active. En effet, les requêtes de lecture et d'écriture on été remplacées par des requêtes de migration. De plus, le serveur a été modifié pour gérer ces différentes requêtes.

3.4.2.1. Migration d'entités

La migration des entités est réalisée selon le protocole suivant. On distingue deux étapes : la création d'une nouvelle entité, et la synchronisation de l'ancienne entité et de la nouvelle. Nous allons présenter ces deux étapes dans la suite de cette partie. Pour cela, on considère qu'une entité située sur un site que nous appellerons A doit migrer sur un site B.

La première étape comporte 5 sous étapes (Figure 47) :

- La définition de l'entité (structure ViperEntitéMigrante située dans un fichier VRML) est empaquetée dans un message (1).
- Le site A envoie le message au site B (2).
- Après réception du message, le site B construit la nouvelle entité. Cette construction peut avoir une durée importante puisqu'il faudra éventuellement récupérer les fichiers ObjectTcl ou les librairies dynamiques contenant la définition des capteurs, effecteurs, composants comportementaux et état interne de l'entité. Pour éviter de bloquer le serveur de l'UVD actif situé sur le site B, un processus de poids léger est créé lors de cette opération (3 et 4).
- Si l'entité a pu être construite, le site B envoie un acquittement au site A et on peut passer à la seconde étape, sinon le site B envoie un acquittement négatif au site A et l'opération de migration est annulée (5).

En fait, même si la construction d'une entité nécessite la récupération de fichiers ObjectTcl ou de librairies dynamiques, la sous étape de construction peut être de courte durée si tous les

fichiers en question ont déjà été récupérés antérieurement et sont conservés dans un cache ou sont déjà chargés en mémoire.

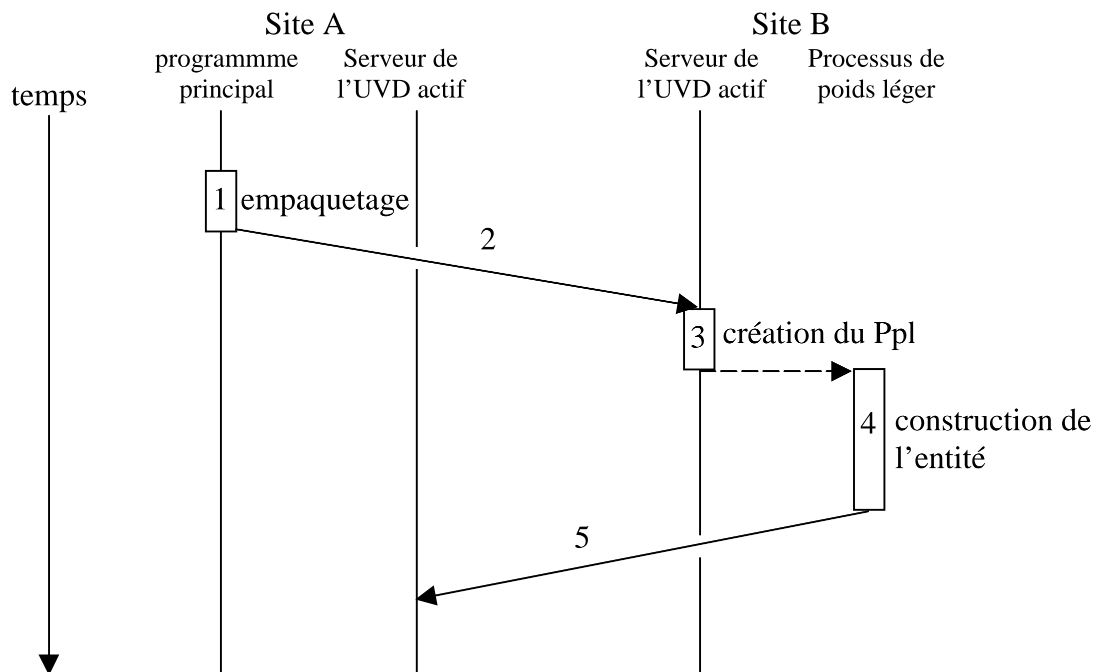


Figure 47 : première étape de la migration

La seconde étape comporte, elle, un nombre variable de sous étapes (Figure 48) :

- Le comportement de l'entité est inhibé sur le site A (1).
- Le site A demande à l'entité de s'empaqueter dans un message (2). Pour s'empaqueter dans un message, l'entité y place les données de son état interne (ceci est réalisé par le composant comportemental d'empaquetage de l'entité).
- Le site A envoie le message au site B (3).
- Le site B reçoit le message et dépaquette le message dans la nouvelle entité (créée dans la précédente étape) (4).
- Si l'opération précédente s'est déroulée sans problème le serveur du site B met à jour la valeur du site de l'entité. Cette mise à jour va, grâce au dictionnaire dupliqué, être envoyée à tous les sites sur lesquels l'UVD est défini (5) ; le site A, lorsqu'il a connaissance de cette mise à jour, détruit la copie locale de l'entité (6) ; le site B, immédiatement après l'envoi de la mise à jour permet à l'entité d'exécuter son comportement (7).

- Si l'opération échoue, le site B envoie un acquittement négatif au site A (8) ; celui ci, dès réception de cet acquittement, va permettre à l'entité de reprendre l'exécution de son comportement (9).

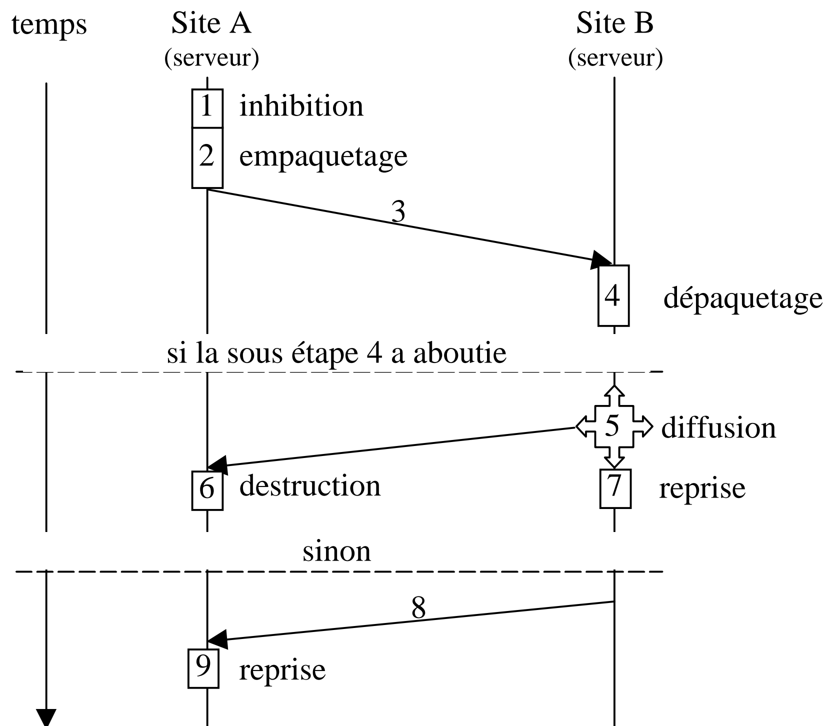


Figure 48 : La seconde étape de la migration d'une entité

Dans cette seconde étape, toutes les étapes sont gérées par les serveurs de l'UVD actif. Ainsi, le dépaquetage de la nouvelle entité, le mécanisme réparti de mise à jour du site de référence et l'éventuelle reprise du comportement sont gérées par le serveur de l'objet parallèle qui existe sur le site B. De même, l'inhibition, l'empaquetage, la destruction de l'entité ou la reprise du comportement sont gérés par le serveur de l'objet parallèle du site A.

Il est à noter que le programmeur d'une entité migrante n'a à écrire qu'une fonction qui place et retire d'un message les données de son entité. Toutes les autres étapes de la migration sont réalisées implicitement par l'univers virtuel distribué et par la classe entité migrante. De plus, nous pensons qu'à terme il suffira au développeur de décrire les données de l'état interne de l'entité et qu'un pré-compilateur générera automatiquement la fonction d'empaquetage/dépaquetage des données. Ce qui correspond à ce qui est fait par l'IDL de CORBA [CORBA 96].

3.4.2.2. Prise de décision pour la migration

La décision de faire migrer une entité peut être prise par divers éléments et pour diverses raisons. Comme nous l'avons décrit, sur l'exemple d'utilisation de la migration, une entité peut décider elle même de changer de site. Dans ce cas, il lui suffira d'en faire la requête auprès de son UVD actif. D'autre part, un UVD peut décider de faire migrer un certain nombre d'entités soit pour gérer un équilibre des charges dans le système réparti soit lorsqu'un site se retire de la simulation (à la demande de l'utilisateur local).

La fonction assurant l'équilibre des charges qui existe actuellement consiste, lors de l'arrivée d'un nouveau site dans la simulation, à faire migrer un certain nombre d'entités vers ce nouveau site de façon à ce que tous les sites aient, à peu près, le même nombre d'entités. Cet algorithme très simple pourra être redéfini par la suite, en héritant de la classe UVD actif, pour mettre en œuvre des algorithmes plus complexes comme ceux définis dans WAVES [Kazman 93b].

Dans tous les cas, la migration sera commencée dès que le comportement de l'entité aura été simulé c'est à dire dès que l'exécution de la fonction Action de l'entité sera terminée. En effet, pour l'instant, les entités migrantes n'ont pas la possibilité de disposer de comportements réactifs. Cette limitation n'existera plus à terme car nous pensons utiliser un processus de poids léger pour gérer la migration. Ce processus pourra, après avoir spécifié l'inhibition de tous les composants comportementaux, attendre la fin de l'exécution en cours des comportements et ensuite commencer la migration. Un processus de poids léger sera créé pour chaque migration et terminera son exécution dès l'acquittement positif ou négatif de la migration (Figure 48 étapes 7 et 10).

3.4.3. Les univers virtuels dupliqués

Pour des raisons d'efficacité similaires à celles qui nous ont conduit à réaliser les UVD actifs, nous avons réalisé des univers virtuels dupliqués simples pour lesquels toutes les entités sont dupliquées sur chaque site [Balet 96]. Les comportements sont exécutés sur chaque site et il n'y a pas de synchronisation entre les états des entités dupliquées.

La mise en œuvre de ces univers virtuels est simple. Elle utilise des listes dupliquées passives d'entités. Lors de la récupération d'un monde virtuel (décrit par un fichier VRML), chaque site va créer toutes les entités duplicables présentes dans le monde et les placer dans la liste. Ces entités sont décrites par une structure identique à celle d'une entité normale mais comportent le mot clé DUPLICABLE comme premier paramètre de l'entité (cf. chapitre précédent). Par la suite, les comportements seront appelés normalement à la différence près que les stimuli créés ou modifiés par ces entités ne nécessiteront pas de messages de mise à jour distante. L'UVD utilisera pour cela des espaces de stimuli passifs (cf. 3.5.1.2 et 3.5.3.2).

Ce type d'UVD est intéressant pour simuler des objets virtuels dont le comportement ne dépend que du temps qui passe et avec lesquels il est impossible d'interagir directement. C'est le cas par exemple d'horloges fixes dans le monde virtuel, ou encore d'astres virtuels (la lune par exemple) dont le mouvement est imperturbable.

D'autre part, ces UVDs peuvent aussi être utilisés pour réaliser des objets mondes similaires à ceux du système AVIARY [Snowdon 93]. Dans ce cadre, ce type d'objet peut être doté de lois (mises en œuvre par des composants comportementaux) qu'il applique à toutes les entités du site local présentes dans le monde qu'il représente. Ainsi, une telle entité peut faire descendre vers le sol toutes les entités soumises à la pesanteur à chaque itération de la boucle de simulation.

3.5. Les espaces de stimuli distribués

L'introduction d'un espace de stimuli, qui joue un rôle d'intermédiaire dans la communication, permet de simplifier la gestion des interactions dans un cadre distribué. L'espace de stimuli devient alors un objet parallèle qui gère les interactions de façon autonome. De plus, il traite uniformément les interactions locales à un site et les interactions impliquant des entités qui sont gérées par des sites distants.

Les espaces de stimuli distribués sont des agrégats de stimuli qui ont la particularité d'être polymorphes. En effet, il est possible de définir de nouveaux types de stimuli qui pourront être gérés par des espaces de stimuli qui gèrent leur classe ancêtre. Dans ce cadre, le problème est la création distante de stimuli. Il faut effectivement allouer un stimuli du bon type. Pour gérer ce problème, tous les espaces de stimuli disposent d'un dictionnaire dont la clé (utilisée par la

fonction de *hashing*) est un identificateur unique de type de stimuli qui permet d'accéder à un pointeur de fonction qui alloue un stimuli du bon type. Le dictionnaire est rempli lors de l'initialisation de VIPER sur chaque site avec les types standards gérés par l'espace de stimuli. Ensuite, dès qu'une entité utilisant un nouveau type de stimuli est créée par un UVD, le fichier contenant l'implémentation du type de stimuli est récupéré. L'espace de stimuli qui gère ce type de stimuli (en fait un type correspondant à la classe ancêtre du nouveau stimuli) est aussitôt notifié, le serveur de connexion de l'univers virtuel est contacté pour connaître l'identificateur unique de stimuli et le dictionnaire reçoit une entrée de plus. Toutes ces opérations sont réalisées par un processus de poids léger créé spécialement pour éviter de bloquer la simulation.

Nous allons montrer, maintenant, comment les types d'espaces de stimuli distribués que nous avons défini actuellement peuvent résoudre les problèmes soulevés par les aspects répartis des applications ciblées par VIPER.

3.5.1. Les espaces de stimuli dupliqués actifs

Le premier problème apparaît dès que plusieurs avatars évoluent ensemble dans un même monde virtuel. Un utilisateur doit avoir conscience de la présence des autres utilisateurs. Pour qu'il puisse en être ainsi, l'utilisateur doit pouvoir voir et entendre (et éventuellement toucher) les avatars représentant les autres utilisateurs. D'autre part, il faut que l'utilisateur puisse percevoir les modifications du monde virtuel réalisées par les autres utilisateurs. Le problème dans ce cadre est que, comme nous l'avons précisé précédemment, les avatars sont en général gérés par des machines différentes.

Pour résoudre ce problème on définit un type particulier d'espace de stimuli distribué, appelé espace dupliqué. C'est un agrégat qui lors de la modification de l'un de ses stimulus maintient la cohérence entre la version originale et ses copies figurant sur chaque site (Figure 49). Ce maintient de la cohérence est assuré par l'envoi de messages de mise à jour de façon transparente. Ces envois de messages sont réalisés grâce à un groupe de communication privé à l'espace de stimuli dupliqué.

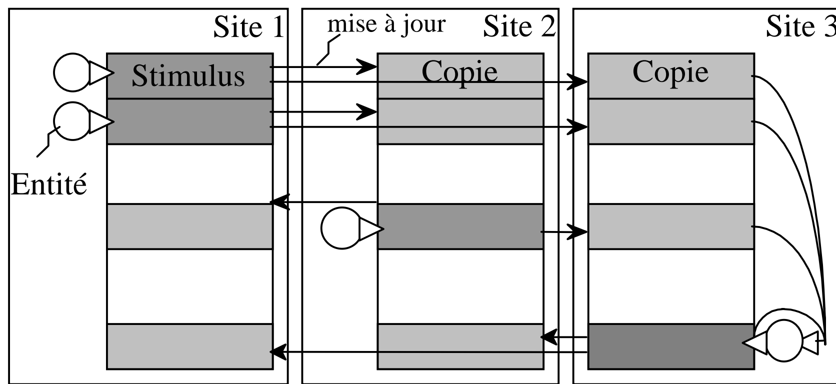


Figure 49 : Un espace de stimuli dupliqué

3.5.1.1. Les espaces de formes

La perception visuelle illustre bien le principe de fonctionnement des espaces de stimuli dupliqués. Chaque entité peut être pourvue d'une **forme** qui la représente. Un espace de stimuli dupliqué, appelé **espace des formes**, contient toutes les formes. Ainsi lors de la modification de l'une d'elles (déplacement, rotation ou déformation de l'objet), l'espace de stimuli maintient la cohérence entre les différentes copies (Figure 50). Par la suite, chaque entité disposant d'un capteur intéressé par les formes (par exemple : un capteur « caméra » ou un capteur volumique) a accès à une version mise à jour de chaque forme.

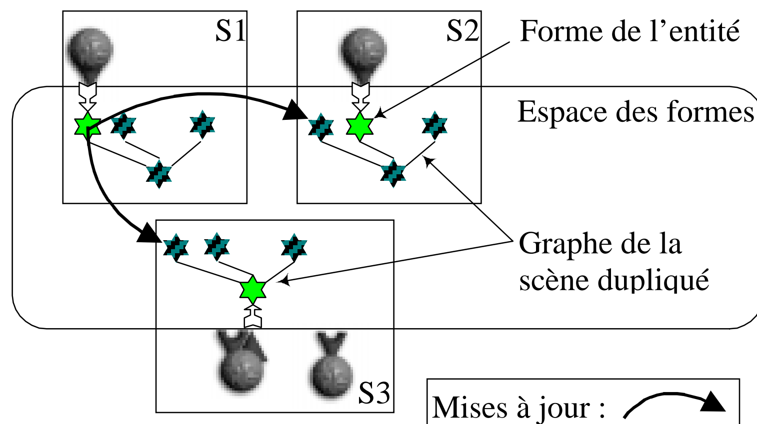


Figure 50 : Espace des formes

En fait, il existe un espace des formes pour chaque monde virtuel. Cet espace de stimuli encapsule un graphe de scène dupliqué sur chaque site qui gère des entités présentes dans le monde virtuel. Comme nous l'avons vu dans le chapitre précédent, à chaque forme correspond une portion privée du graphe de la scène. Cette portion est composée d'un certain nombre de

nœuds VRML. Chacun de ces nœuds peut être modifié si il possède un nom (précisé par le mot clé DEF de VRML).

Néanmoins, avant la première modification, un identificateur unique de nœud doit être créé pour que la modification puisse être identique sur tous les sites. Pour le nœud racine de la portion privée, la création de l'identificateur est réalisée lors de la création de la forme. L'identificateur contenant un identificateur de site et un numéro séquentiel, il est unique dans le monde virtuel.

Lors de la création d'une forme, une requête de création est envoyée à chacun des sites. Cette requête contient : l'identificateur unique du type de stimuli (soit l'identificateur standard de la classe forme, soit l'identificateur d'une de ses descendantes), l'identificateur unique du nœud racine de la forme (qui est créé sur le site local), l'identificateur de l'entité qui possède la forme, d'éventuels paramètres permettant de gérer une extrapolation de la forme et soit un nom de nœud existant déjà dans la scène (qui deviendra le nœud racine), soit l'URL du fichier VRML qui contient la définition de la forme. Ainsi, une création de forme peut impliquer le téléchargement d'un fichier VRML. Pour ne pas bloquer la simulation pendant une telle récupération, un processus de poids léger est créé pour mettre en œuvre cette opération.

Si on veut modifier un autre nœud que le nœud racine, il faudra aussi lui affecter un identificateur unique. Nous avons appelé cette opération, une promotion. On dira qu'un nœud est promu au rang de nœud modifiable. La promotion s'accompagne d'une requête de promotion qui est envoyée à tous les sites du monde virtuel (via le groupe de communication). Sur chaque site, le nœud est ajouté dans un dictionnaire de dictionnaires de nœuds dupliqué (en fait, la classe espace des formes hérite de la classe qui met en œuvre les dictionnaires de dictionnaires dupliqués actifs). L'entrée dans le premier dictionnaire est obtenue en précisant comme clé l'identificateur de site et l'entrée dans le second dictionnaire (correspondant à la première clé) est obtenue en précisant comme clé la partie restante de l'identificateur de nœud. Nous utilisons des dictionnaires de dictionnaires au lieu de simples dictionnaires pour pouvoir détruire plus rapidement un dictionnaire lorsqu'un site quitte le monde virtuel.

Par la suite, pour modifier une forme, il suffit de préciser à l'effecteur de forme, en plus de la modification à apporter, l'identificateur du nœud à modifier. Une requête de modification sera émise et chaque site qui va recevoir la requête modifie la forme. En fait, la requête de modification est préparée par la forme, lorsque la méthode de modification est invoquée par

l'effecteur. Cette requête est ensuite placée dans un message, par la forme, lorsque l'effecteur appelle sa méthode d'empaquetage et ce dernier n'a plus qu'à transmettre ce message à l'espace des formes qui s'occupe de l'émission sur le réseau. D'autre part, lors de la réception du message par l'un des serveurs de l'espace des formes, la requête est extraite du message lorsque l'espace des formes invoque la méthode de dépaquetage de la forme. Dès que la requête est extraite, la forme recherche le nœud correspondant à l'identificateur (en utilisant l'un des dictionnaires de nœuds de l'espace des formes) et applique la modification.

3.5.1.2. Les espaces de formes à zones d'intérêt

Pour gérer des environnements virtuels complexes comportant de nombreuses entités nous avons mis en œuvre, à l'intérieur d'espaces de formes, des techniques permettant de limiter le nombre de messages de mise à jour émis sur le réseau ainsi que le nombre de sites interagissants. En effet, nous pouvons noter que deux entités ne peuvent se voir que si elles sont assez proches. L'exploitation de cette propriété de localité de la perception peut résoudre certains des problèmes induits par la présence d'un grand nombre d'entités dans un même monde virtuel.

Un découpage spatial logique de l'univers virtuel est une solution qui paraît intéressante lorsque le monde virtuel est constitué d'un ensemble de pièces reliées par des portails (portes, ouvertures, sas...). Dans ce cas, chaque pièce va devenir une région de l'espace. Un découpage spatial régulier (2D ou 3D) peut lui aussi être envisagé. Dans cet autre cas, on va découper l'espace du monde virtuel en régions hexagonales (applications pour lesquelles la hauteur importe peu : simulation automobile par exemple) ou cubiques (applications fortement tridimensionnelles : simulation de vol par exemple) qui peuvent être de tailles équivalentes ou de tailles variées dans une structure récursive (par exemple un octree). Ensuite, dans les deux cas, on va associer les formes aux régions dans lesquelles elles sont spatialement présentes.

Ensuite, on peut imaginer une répartition de l'espace de stimuli consistant à partitionner l'ensemble des régions sur différents sites. Ainsi, le site gère les interactions visuelles entre les entités se trouvant dans la région qui lui a été attribuée. Le problème apporté par cette solution, qui ressemble à ce qui est fait par MASSIVE [Greenhalgh 95] et par la seconde version de RING [Funkhouser 96], est la latence introduite par le transit obligatoire des messages de modification de forme par le site intermédiaire qui gère la portion du monde.

Dans le cas d'une application à grande échelle, répartie sur un réseau longue distance (WAN), cette latence peut être critique.

Nous avons donc choisi d’utiliser une autre solution, initiée par Macedonia pour le NPSNET [Macedonia 95a], qui consiste à associer à chaque région du monde, un groupe de communication. Par la suite, les modifications de chaque forme sont envoyées sur un seul groupe de communication à un moment donné (celui correspondant à la région dans laquelle l’entité est localisée) et chaque entité qui veut recevoir des stimuli s'abonne aux groupes de communication correspondant à la zone qui l’intéresse (c’est à dire qu’elle peut observer).

Dans ce cadre général, une fois que l’on a choisi la méthode de découpage, il reste à choisir l’algorithme qui permet de déterminer la zone d’intérêt (ensemble de régions) d’une entité.

Le premier espace de formes à zones d’intérêt que nous avons mis en œuvre gère le découpage logique d’un bâtiment. L’unité de découpage est évidemment la pièce. Il existe de nombreux algorithmes permettant de déterminer une zone d’intérêt ou de visibilité dans des bâtiments architecturaux [Airey 90] [Teller 92]. Celui que nous utilisons a été défini par Luebke et Georges [Luebke 95] pour implanter un algorithme d’élagage (*culling*) intelligent. L’intérêt de cet algorithme est qu’il est utilisable dynamiquement et facilement intégrable dans des librairies graphiques existantes comme Performer et WTK. De plus, nous avons adapté cet algorithme pour qu’il gère, en plus, la limitation des échanges entre les différents sites.

L’algorithme consiste à découper un bâtiment en cellules (correspondant au diverses pièces) reliées par des portails. Chaque cellule est représentée par un volume polyédrique convexe (qui doit être disjoint des polyèdres représentant les autres cellules et qui doit permettre de déterminer si un point est à l’intérieur du volume) et chaque portail est représenté par un polygone transparent quelconque (qui peut être concave). Une cellule ne peut être « vue » depuis une autre cellule qu’au travers de portails qui correspondent aux diverses portes et fenêtres qui les relient (directement ou indirectement).

Par la suite, pour trouver la zone d’intérêt d’un point de vue, on considère l’origine du point de vue et la pyramide de vision. On commence par déterminer la pièce qui contient l’origine du point de vue. Puis, pour chaque portail de la pièce, on projette dans l’espace écran les différents sommets du polygone qui le représente. On détermine un rectangle parallèle aux

axes englobant tous les points projetés (rectangle min-max). Ce rectangle, qui est appelé un rectangle d'élagage, représente un englobant par excès du portail. C'est à dire qu'on peut être certain que les portails qui ne sont pas visibles à travers ce rectangle d'élagage sont invisibles à travers le portail.

L'algorithme consiste à accumuler récursivement (et en profondeur d'abord) les différents rectangles d'élagages des portails des cellules visibles en conservant leur intersection. Lorsque l'intersection entre le rectangle d'élagage d'un portail et le rectangle accumulé est vide, on peut être sûr que la cellule qui se trouve derrière le portail est invisible à travers les différents portails accumulés.

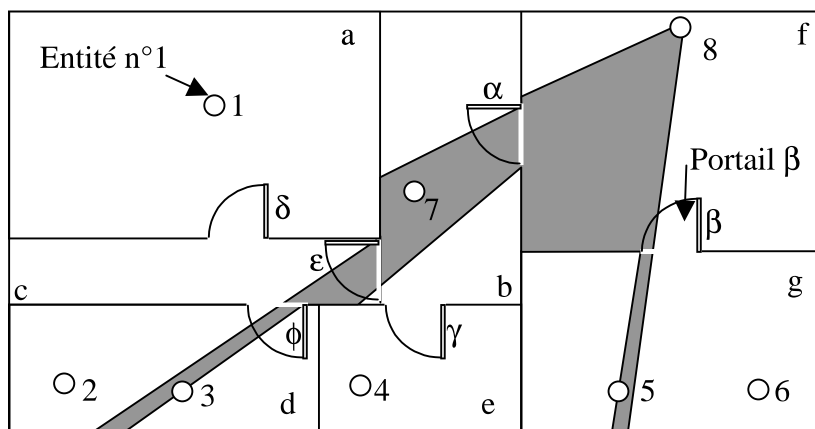


Figure 51 : Un bâtiment et la zone de perception réelle d'une entité

Par exemple, pour le bâtiment de la figure précédente (Figure 51), pour déterminer la zone de visibilité par excès de l'entité numéro 8, on commence par traiter la cellule f. On va considérer tout d'abord le portail α, on va calculer son rectangle d'élagage que l'on va conserver directement. Puis on va considérer les portails de la pièce b en commençant, par exemple, par le portail γ. Dans ce cas, l'intersection entre le rectangle d'élagage du portail γ et le rectangle accumulé (ici le rectangle d'élagage du portail α) est nulle et donc la cellule e est élaguée. Puis, en traitant le portail ε, on va trouver une intersection non nulle et on va traiter recursivement les portails de la cellule c. Seul le rectangle d'élagage du portail φ a une intersection non nulle avec le rectangle accumulé (rectangle correspondant à l'intersection des rectangles d'élagages des portails α et ε) et donc la cellule d est considérée comme visible. Il ne reste plus alors qu'à traiter le portail β qui permet de considérer la cellule g comme visible. En conclusion, seules les cellules f, b, c, d et g sont conservées et seront rendues avec leur contenu.

Le traitement rajouté pour gérer la limitation des échanges entre les différents sites, consiste à déterminer entre deux étapes quelles sont les cellules qui sont désormais visibles (et qui étaient invisibles à l'étape précédente) et celles qui sont désormais invisibles (et visibles à l'étape précédente). Puis, pour toute les cellules désormais visibles, le site gérant l'entité considérée s'abonne aux groupes de communication des cellules. Symétriquement, le site se retirera des groupes de communication des cellules désormais invisibles. Etant donné que les messages de mise à jour de la forme d'une entité sont diffusés sur le groupe de la cellule qui la contient, seules les mises à jour des formes visibles par les entités locales seront reçues par un site donné.

De plus, lorsque le site s'abonne à un groupe, il va demander (en envoyant une requête au groupe) l'état actuel de toutes les formes de la cellule. En effet, même si le site était précédemment connecté à ce groupe, depuis la dernière connexion les formes ont pu évoluer. Les réponses à cette demande sont envoyées par les sites qui gèrent les entités possédant les formes concernées. Actuellement, ces réponses sont envoyées sur le groupe correspondant à la cellule et permettent ainsi une remise à jour de tous les sites connectés (ceci peut être utile si un précédent message de mise à jour a été perdu). Cependant, pour éviter cette diffusion systématique dans un environnement très dynamique, nous pensons à terme envoyer les réponses directement au site demandeur. De même, pour récupérer uniquement l'état des formes modifiées depuis la dernière connexion on peut imaginer, à terme, d'envoyer la date de dernière connexion qui sera comparée à la date de dernière mise à jour de chaque forme.

En fait, seules les mises à jour de la position et de l'orientation des nœuds représentant une forme sont envoyées sur le groupe de sa cellule courante. Les autres mises à jour (promotion d'un nœud, changement de couleur, déformation d'objet…), qui sont moins fréquentes, sont émises sur un groupe global à l'espace de formes. Ainsi il est plus simple, lorsqu'un groupe est ajouté, de récupérer l'état actuel des formes présentes dans sa cellule puisqu'il n'y a que la position et l'orientation courante de chaque nœud promu à envoyer.

Il est important de noter que nous utilisons des groupes de communication non fiables (*multicasting* IP) pour les cellules et un groupe de communication fiable pour le groupe global (un groupe RMP actuellement). En effet, la connexion/déconnexion à un groupe non fiable est plus rapide. Pour gérer une fiabilité minimale, les sites envoient à une fréquence faible (toutes

les cinq secondes actuellement) un message de mise à jour sur le groupe global pour les nœuds qui ont été modifiés depuis le dernier message fiable.

Le problème induit par la solution choisie (utilisation de groupes de communication) est le temps pris pour s'abonner et se désabonner d'un groupe de communication. Ce temps dans un réseau *multicast* à grande échelle comme le MBone est assez long (en comparaison avec la latence maximale requise par les applications de réalité virtuelle). Pour limiter ce problème nous proposons d'agréger plusieurs pièces contiguës en une seule cellule. Ainsi, les différentes pièces seront gérées par le même groupe.

Le choix d'un espace de forme est difficile à faire. En fait, ce choix dépend beaucoup du type d'application visée. Pour un simulateur de véhicules, par exemple, le choix risque de porter sur un découpage spatial ressemblant à celui défini par Macédonia pour le NPSNET, alors que pour un système de visite de bâtiments virtuels il semble plus à propos de choisir le découpage spatial logique utilisant les cellules et les portails.

VIPER simplifie ce choix, car tous les espaces de formes que nous avons défini s'utilisent de la même façon et il suffit donc d'allouer dans le programme principal un espace de forme différent pour obtenir un comportement réparti différent (Figure 52).

```
{
  ...
  // Déclaration d'un pointeur d'espace de forme
  EspaceDeForme*  monEspaceDeForme;

  // Allocation d'un espace de forme simple
  // (les paramètres sont le noeud racine de la scène et
  // le domaine sur lequel l'espace de forme est réparti)
  monEspaceDeForme = new EspaceDeForme(noeudRacine, domaine);

  // ou

  // Allocation d'un espace de forme gérant des zones
  // d'intérêt grâce à des cellules et portails
  // (qui sont recherchés dans le graphe de la scène
  // sous le noeud racine)
  monEspaceDeForme =
   new EspaceDeFormeACellulesEtPortails(noeudRacine, domaine);

  ...
}
```

Figure 52 : Choix d'un comportement réparti grâce à une simple allocation

3.5.1.3. Les espaces d'icônes sonores

Les espaces d'icônes sonores sont eux aussi des espaces de stimuli dupliqués mis en œuvre par un dictionnaire de dictionnaires dupliqué. Leur fonctionnement est similaire à celui d'un espace de formes.

Lors de la création de l'icône sonore, l'entité précise à son effecteur un pointeur qui permet d'identifier et d'atteindre le son qui devra être joué. Un identificateur d'icône sonore est alloué par l'espace des sons. Cet identificateur est constitué de façon similaire à l'espace des formes par l'identificateur du site de l'entité et un numéro séquentiel. Une requête de création, contenant l'identificateur d'icône sonore et le pointeur, est ensuite envoyée à tous les sites (par l'intermédiaire du groupe de l'espace d'icônes sonores) qui vont lors de sa réception, télécharger le fichier son s'il n'est pas déjà présent en mémoire et placer une référence au son dans le dictionnaire correspondant au site de création de l'icône sonore.

Par la suite, lors d'une modification de l'icône sonore par l'entité sur son site de création, un message de mise à jour sera envoyé au groupe qui gère l'espace de stimuli et les modifications seront répercutées sur chacun des sites.

Il est intéressant de noter que pour cet espace de stimuli nous n'avons quasiment rien eu à développer car son fonctionnement est purement et simplement le fonctionnement par défaut de l'objet parallèle utilisé. La seule chose à mettre en œuvre était la récupération du fichier son à partir de son URL, mais ce transfert de fichier est identique à celui mis en œuvre par l'espace des formes.

3.5.2. Les espaces de stimuli dupliqués passifs

Ces espaces de stimuli sont utilisés par les univers virtuels dupliqués et sont nettement plus simples puisqu'ils ne nécessitent pas de mises à jours distantes. Nous avons défini des espaces de formes passifs et des espaces d'icônes sonores passifs qui ne gèrent que les modifications locales de formes et d'icônes sonores.

3.5.3. Les espaces de stimuli partitionnés

3.5.3.1. Les espaces d'ordres

Les interactions directes (manipulations) créent un nouveau problème : l'entité effectrice agissant sur une entité peut être située sur un autre site.

Dans un contexte distribué, un espace de stimuli dupliqué pourrait résoudre ce problème. On peut néanmoins remarquer qu'un ordre donné ne concerne que l'entité manipulatrice et les entités manipulées. Il n'est donc pas nécessaire de le communiquer à tous les sites prenant part à la simulation. La fonction de répartition d'un UVD permettant de trouver le site d'une entité à partir de son identificateur, nous pouvons limiter la transmission d'un ordre aux sites intéressés (Figure 53).

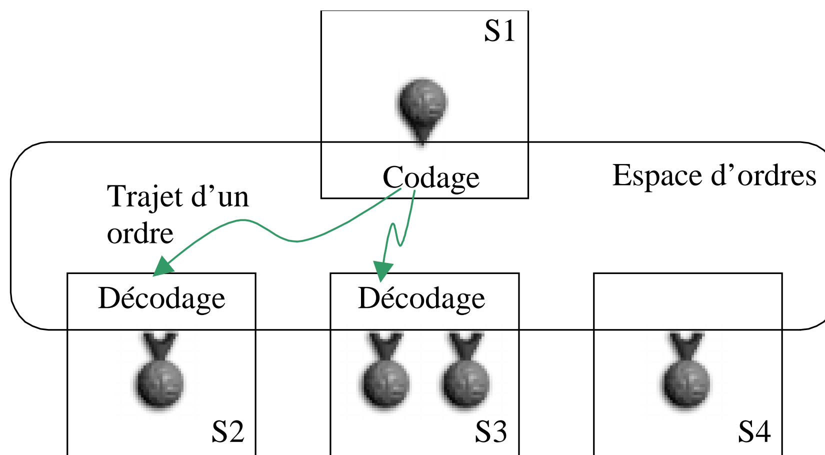


Figure 53 : Un espace d'ordres

Ainsi, l'espace d'ordre est mis en œuvre par des objets partitionnés. Nous utilisons un vecteur partitionné de files gérant une priorité pour les entités qui sont gérées sur le site où elles ont été créées. De plus, nous utilisons un dictionnaire partitionné de files (gérant une priorité) pour les entités qui ont migré de site. Les files gérant des priorités permettent de récupérer les stimuli dans leur ordre d'émission car la priorité gérée consiste à comparer les estampilles horaires des stimuli.

Lorsqu'une entité veut émettre un ordre, elle fournit à son effecteur d'ordre : l'identificateur unique du type de stimuli (soit l'identificateur d'une classe standard héritant de la classe ordre, soit l'identificateur d'une classe descendante de celles-ci), les identificateurs

des entités qui doivent recevoir l'ordre et les paramètres de l'ordre. L'effecteur va affecter la date courante à l'estampille horaire du stimulus, rajouter l'identificateur de l'entité qui émet l'ordre et faire passer le stimulus à l'espace d'ordres. L'espace questionne localement les UVD des entités réceptrices pour connaître leurs sites actuels. Puis l'espace va envoyer à chaque site différent du site de départ de l'ordre un message contenant l'entité. Si l'ordre est envoyé à une entité locale (soit créée localement soit ayant migré sur ce site), il sera directement placé dans la file correspondant à l'entité (qui sera créée si elle n'existait pas déjà). De plus, sur tous les sites distants qui recevront le message, l'ordre sera dépaqueté du message par le serveur de l'espace d'ordre et il sera placé dans les files correspondant aux entités réceptrices.

En fait, si un capteur est abonné à l'espace d'ordre, lors de chaque réception d'un ordre, le capteur sera activé et on lui délivrera l'ordre en question qu'il fera passer à un composant comportemental. Dans ce cas, la file d'ordres n'est pas utilisée. Néanmoins, un capteur d'ordre peut être utilisé pour récupérer depuis un composant comportemental actif les ordres reçus depuis la dernière récupération. C'est dans ce dernier cas que la file à priorité est utile.

3.5.3.2. Les espaces d'ordres passifs

Ces espaces d'ordres sont une restriction des espaces précédents utilisés par les univers virtuels dupliqués. Ils sont beaucoup plus simples puisqu'ils ne nécessitent pas d'accès distants.

Ils sont mis en œuvre par des vecteurs partitionnés de files à priorité. Lorsqu'une entité dupliquée veut envoyer un ordre, son effecteur d'ordre le fera passer à un espace d'ordre passif. L'espace ne traitera que les ordres envoyés aux entités locales qu'il délivrera instantanément (cas d'un capteur d'ordre abonné) ou qu'il placera dans une file. Etant donné que le comportement des entités dupliquées est déterministe, l'ordre sera traité sur chaque site du domaine de l'univers virtuel dupliqué et donc il sera délivré à toutes les entités qui doivent le recevoir et qui sont situées sur des sites de ce domaine. Les ordres envoyés à des entités situées sur des sites en dehors du domaine sont ignorés.

3.6. Conclusion sur les aspects répartis

Dans ce chapitre nous avons décrit les mécanismes qui permettent de répartir un environnements virtuel avec VIPER. Nous avons, tout d'abord, présenté la première couche de notre architecture : la plate-forme répartie. Cette plate-forme permet d'assurer la généricité vis à vis des systèmes d'exploitation et de communication utilisés par VIPER. Dans ce cadre, la portabilité de notre système est assurée par les librairies de bas niveau qu'utilise VIPER. En effet nous utilisons des librairies qui existent sur de nombreuses plates-formes : depuis les compatibles PC, en passant par les stations de travail UNIX les plus répandues et même, pour PVM, jusqu'à une large gamme de multiprocesseurs. Tous les aspects spécifiques à une architecture physique ou à un système de communication sont encapsulés dans un petit nombre de classes. Ainsi, même dans le cas où une nouvelle architecture physique ou un nouveau système de communication devrait être utilisé, les modifications à apporter à VIPER seraient circonscrites à cet ensemble de classes.

Nous avons ensuite présenté la seconde couche qui permet de développer simplement des applications réparties grâce à des objets parallèles. Ces objets sont des agrégats qui gèrent tous les aspects distribués liés à la répartition de leurs données sur un ensemble de sites. Ces objets parallèles ont été initialement conçus et développés par un autre membre de notre équipe de recherche pour gérer des applications de rendu réaliste sur des réseaux de Transputers. Notre apport personnel a consisté, pour adapter cet environnement de programmation parallèle à objets à la réalité virtuelle distribuée, d'une part à redéfinir certains concepts comme le domaine, pour rendre plus flexibles les objets parallèles, et d'autre part à intégrer des concepts comme la communication par groupe à l'environnement pour le rendre plus efficace. De plus, nous avons du réécrire de nombreuses classes existantes pour assurer leur exécution sur les architectures physiques (stations de travail UNIX et compatibles PC) et les systèmes de communication utilisés par VIPER.

Enfin, nous avons montré comment les univers virtuels distribués et les espaces de stimuli distribués permettent de résoudre les problèmes de répartition des applications ciblées. Nous avons, tout d'abord, défini de nouveaux objets parallèles utiles pour répartir des environnements virtuels. Ces objets parallèles sont utilisés par les UVD et les espaces de stimuli distribués pour gérer de façon transparente la répartition de leurs données (entités et

stimuli). De plus, pour certains types d'espaces de stimuli distribués nous avons proposés plusieurs modèles de répartition qui peuvent être choisis simplement en déclarant telle ou telle classe gérant des stimuli d'un même type. Ceci est très intéressant pour pouvoir optimiser notre système à un type d'application.

Chapitre 4. Evaluations et applications

Dans les deux chapitres précédents nous avons présenté le système de réalité virtuelle distribuée VIPER, son architecture, son modèle comportemental et ses aspects répartis.

Dans ce chapitre nous allons tenter d'évaluer VIPER tant au niveau des originalités proposées par ce système qu'au niveau de son efficacité de gestion de la répartition. Ces évaluations préliminaires ont été menées sur deux applications simples : une visite interactive multi-utilisateur d'un bâtiment architectural virtuel et une maquette de simulateur de véhicules militaires multi-utilisateur. Après ces évaluations nous présenterons trois applications. La première application est un modeleur d'objets et de scènes qui permet de sculpter des objets virtuels dans des formes simples grâce à des outils de déformations. La seconde application, GRS (*Generic Railway System*), est un simulateur de gare réalisé par la société CISI pour la SNCF (Société Nationale des Chemins de Fer). Cette application est basée sur le modèle comportemental et d'interaction de VIPER mais elle a été développée de façon à ne fonctionner que sur une seule machine. Enfin, la troisième application, PROVIS (PROtoypage VIrtuel de Systèmes), permet le prototypage virtuel de satellites par plusieurs utilisateurs et a été conçue et développée en collaboration avec CISI pour le CNES (Centre National d'Etudes Spatiales).

4.1. Evaluation du modèle

4.1.1. Comparaison avec les autres systèmes

Pour comparer VIPER avec les systèmes que nous avons présentés dans le premier chapitre, nous allons reprendre les différents critères qui nous ont servi à les comparer. Pour chaque critère nous présentons un tableau équivalent à ceux définis dans la conclusion du premier chapitre.

Le premier critère défini était la généricité. Comme nous l'avons montré dans les deux précédents chapitres, notre système est générique par rapport aux plates-formes matérielles et

par rapport aux applications qu'il peut gérer. De plus, notre système peut gérer plusieurs types de mondes possédant diverses lois physiques.

	Plates-formes	Applications	Mondes
VIPER	Stations UNIX et PC	Multi-application	Plusieurs

Le second critère concernait les aspects répartis des systèmes de RVD. A ce sujet, notre système appartient à la catégorie des systèmes multi-sites permettant de gérer une dizaine d'utilisateurs et probablement plus grâce au mécanismes de filtrage de messages spatiaux que nous avons développé (cf. chapitre 3). Dans ce cadre, il serait intéressant de tester des applications permettant l'interaction d'une centaine d'utilisateurs. L'architecture adoptée est une architecture essentiellement égal à égal. La base de donnée représentant l'environnement virtuel est distribuée : l'ensemble des entités est partitionné sur les différents sites et les formes et les icônes sonores sont, elles, dupliquées. Le modèle de communication utilise aussi bien des diffusions sur groupe de travail (pour les formes, les icônes sonores…) que des communications point à point (pour l'échange des ordres et pour la migration des entités). Nous avons conçus et développé un espace de formes à zones d'intérêt qui nous permet de gérer un filtrage spatial des messages de mise à jour des formes. Enfin, notre système permet la migration d'entités pour améliorer l'interactivité et pour gérer un équilibre des charges.

	Multi-site multi-utilisateur	Architecture	Base de données	Modèle de communication	Filtrage des messages	Migration
VIPER	Multi-site 10 util.	Egal à égal	Distribuée	Groupes de communications et Point à point	Spatial	Oui

Le dernier critère que nous avons défini concerne la définition de comportements. Notre système propose aussi bien l'ajout de comportements au niveau du système (avec recompilation totale) que la définition de comportements avec un langage interprété (ObjectTcl) . De plus, nous proposons un mécanisme original qui permet de définir des comportements dans des librairies dynamiques. Les comportements définis en ObjectTcl et ceux définis dans des librairies dynamiques peuvent être changés dynamiquement pendant l'exécution du système. Nous ne proposons pas, actuellement, la définition de comportements dans des applications externes puisque les librairies dynamiques offrent sensiblement les mêmes avantages avec plus d'efficacité (on n'utilise pas de communications explicite au niveau du système d'exploitation ni de zones mémoire partagées).

	Ajout de comportements possible	Comp. Compilés avec le syst.	Comp. dans des applications externes	**Comp. compilés dans des librairies dynamiques**	Comp. interprétés	Changement dynamique de comportement pendant l'exécution
VIPER	oui	Oui	non	**oui**	Oui (ObjectTcl)	oui

4.1.2. Originalités du modèle proposé

Après cette comparaison générale nous allons présenter un certain nombre d'originalités introduites par notre système.

4.1.2.1. Aspect génie logiciel

Cet aspect n'est qu'effleuré par la plupart des systèmes de RVD. Le plus en avance de ce côté est sans doute WAVES, cependant ce système a été soit mis en sommeil soit abandonné et d'autre part, même si il y a eu une volonté de la part des concepteurs de travailler l'aspect génie logiciel ils ne donnent pas énormément d'outils aux développeurs d'entités (objoids).

Nous proposons, comme WAVES, des entités autonomes. Nous proposons, de plus, deux niveaux de programmation qui permettent soit de développer le contenu d'environnements virtuels (entités et modèles d'interaction) soit de développer des modèles de répartition utilisables dans des environnements virtuels distribués.

Pour parvenir à ce résultat nous avons fondé VIPER sur de nombreux outils offerts par la programmation par objets. En particulier, l'héritage permet de définir de nouvelles classes d'entités et de stimuli sans se préoccuper de la gestion de leur répartition qui est définie dans une des classes ancêtres. D'autre part la généricité permet de définir des modèles de répartition qui pourront être appliquées grâce à l'héritage ou à l'instanciation générique.

4.1.2.2. Aspect modèle de répartition

Nous adoptons une approche hybride entre les systèmes qui privilégient la vitesse et l'extension vers des environnements virtuels très importants et ceux qui privilégient l'interaction entre les utilisateurs.

Ainsi, nous incorporons aussi bien des mécanismes d'extrapolation du mouvement des formes des entités (*dead reckoning* – cf. 2.4.4.1), des découpages de l'environnement permettant de tenir compte de la localité des interactions (découpage en cellules et portails – cf. 3.5.1.2) mais aussi un niveau de fiabilité permettant d'adresser des applications coopératives (utilisation de systèmes de communication fiabilisés : PVM et RMP).

Nous avons aussi conçu et mis en œuvre un mécanisme qui permet à des entités de migrer de site pour permettre une interactivité accrue entre entités et un équilibre des charges de calcul sur la plate-forme répartie.

4.1.2.3. Aspect Extensibilité du Système

Le système est extensible via la définition de nouveaux modèles de répartition (par exemple, des découpages spatiaux en cellules hexagonales comme ceux du NPSNET peuvent être définis pour gérer des espaces de formes), de nouvelles entités (en plus des entités fournies avec le système comme les avatars, il est possible de définir n'importe quelles entités : des outils virtuels, des objets dotés de comportements simples comme des tiroirs ou des portes et des entités plus complexes comme des modeleurs déclaratifs – cf. 4.2 et 4.4) et de nouveaux stimuli (il est par exemple possible de définir de nouveaux ordres ou de nouvelles formes et même de tous nouveaux stimuli permettant de gérer des communications orales – cf. 4.2 et 4.4) en définissant de nouvelles classes qui sont ajoutées au système.

De plus, le système peut être étendu par des modules externes (librairies dynamiques ou fichiers interprétés) définissant des comportements et des attributs d'entités ainsi que des stimuli, ce qui permet une extensibilité sans recompilation totale du système.

4.1.3. Mesures sur les aspects répartis du modèle

Nous allons présenter, ici, un certain nombre de mesures préliminaires sur notre système de RVD. Ces mesures ne peuvent être que préliminaires compte tenu du faible nombre de machines dont nous disposions pour les réaliser. Il serait intéressant de pouvoir réaliser des mesures à plus grande échelle sur le MBone.

4.1.3.1. Comparaison des différents espaces de formes

Les mesures, pour comparer les espaces de formes actuellement développés, ont été réalisées sur une application de visite interactive d'un bâtiment architectural virtuel (Figure 54). Les machines des utilisateurs étaient reliées par un réseau local Ethernet avec un débit maximal de 10 Mégabits/s. Les utilisateurs ont suivi, lors de ces tests, des chemins aléatoires avec pour unique obligation de traverser toutes les pièces du bâtiment au moins une fois. De plus, ils ont pu lors de la visite déclencher un certain nombre de comportements, modifiant la forme d'autres entités, comme l'ouverture de tiroirs, de portes…

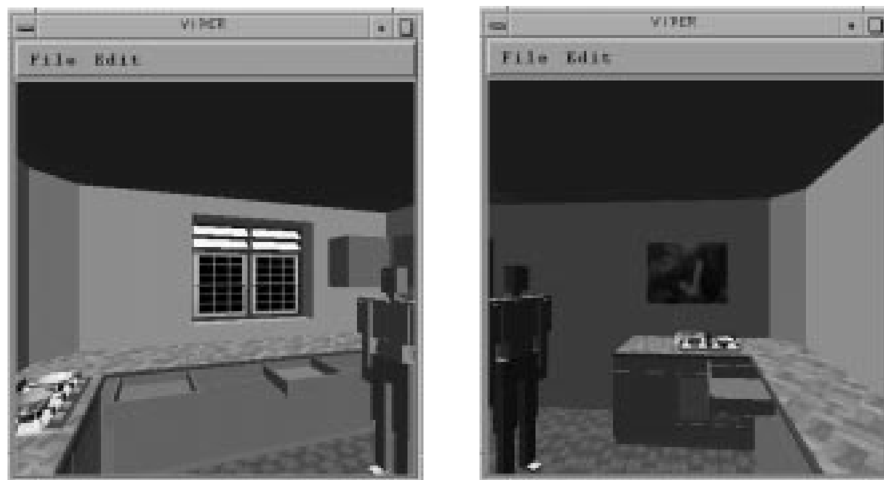


Figure 54 : Visite interactive multi-utilisateurs d'un bâtiment virtuel

Le tableau suivant compare le nombre total de messages émis et reçus, par l'ensemble des machines participant à la simulation, dans le cas d'un espace de formes simple et d'un espace gérant des zones d'intérêt (pour réaliser un filtrage spatial des mises à jours). Pour chaque ligne du tableau, le chemin suivi par les utilisateurs et leurs interactions avec l'environnement virtuel sont identiques. Le tableau présente, de plus, la quantité de données échangée sur le réseau (en octets). Enfin, le tableau présente le gain, exprimé en pourcentage, sur le nombre de messages reçu par chaque site. Ce gain se situe aux alentours de 60 % avec une faible baisse lorsque le nombre d'utilisateurs augmente. La différence (autour de 3 %) existant entre le nombre de messages émis par les espaces de formes est due aux messages qui sont émis lorsqu'un utilisateur entre dans une pièce (ces messages contiennent la position et l'orientation courantes des entités présentes dans la pièce).

Nombre d'utilisateurs	Espace de formes simples				Espace de formes à zones d'intérêt				Gain (%)
	Réception		Emission		Réception		Emission		
	Messages	Octets	Messages	Octets	Messages	Octets	Messages	Octets	
2	5758	546756	5758	546756	2269	217812	5889	556600	**60.59**
3	15036	1424184	7518	712092	5513	529632	7768	732284	**63,34**
4	32394	3061200	10798	1020400	13800	1317928	11112	1044912	**57,40**

4.1.3.2. Efficacité des algorithmes de « dead-reckoning »

Pour évaluer l'efficacité des algorithmes de *dead-reckoning* mis en œuvre dans les espaces de formes de VIPER nous utilisons l'application de simulation de véhicules militaires présentée à la fin du deuxième chapitre de ce document (Figure 55).

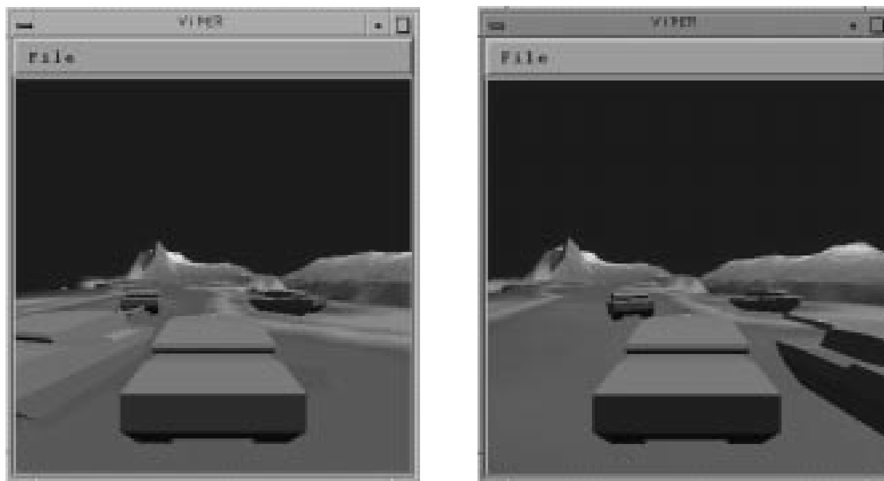


Figure 55 : Prototype d'un simulateur distribué de véhicules militaires

Le tableau suivant présente le nombre de messages de mise à jour qui auraient été envoyés si le dead-reckoning n'était pas utilisé, le nombre de messages réellement envoyés et le gain (exprimé en pourcentage) en nombre de message. Ce gain se situe aux alentours de 84 % et il est à peu prés indépendant du nombre d'utilisateurs. Le gain dépends par contre du seuil utilisé (ce seuil défini l'erreur acceptée entre la valeur réelle de la position du véhicule et la valeur extrapolée). Cependant, même avec un seuil d'un mètre le gain reste supérieur à 77 %.

Nombre d'utilisateurs	Mises à jour demandées	Mises à jour envoyées	Gain (%)	Mises à jour de « recalage »	Rapport (%)
2 (seuil : 4 m)	8375	1308	**84,39**	197	**15,06**
3 (seuil : 4 m)	14133	1578	**88,84**	340	**21,55**
4 (seuil : 4 m)	20516	3630	**82,31**	915	**25,20**
4 (seuil : 1m)	25257	5796	**77,05**	1935	**33,38**

4.2. Première Application : Modélisation de formes

Le collecticiel couvre de nombreux domaines d'activité tels que la conception, l'aide à la décision, l'apprentissage, l'information… Il a fait l'objet de nombreuses études de définition de modèles d'architecture, de formalisations des modèles d'interaction ainsi que d'études spécifiques à des problèmes donnés (édition collective de textes par exemple). Nous pensons néanmoins que certaines voies de recherche, riches de promesses, n'ont pas été complètement exploitées. Ainsi nous avons étudié les apports du collecticiel dans la modélisation interactive dans un environnement virtuel. Nous pensons que la modélisation interactive d'objets peut largement bénéficier des apports de la réalité virtuelle distribuée en terme d'ergonomie, de convivialité et d'efficacité [Torguet 96].

4.2.1. Un éditeur collecticiel de formes virtuelles

Nous avons donc créé un système permettant à des utilisateurs de modeler des objets à partir d'une forme préexistante ou d'un quelconque bloc de matière [Torguet 95b]. Nous avons défini deux types d'usagers qui peuvent évoluer dans l'espace de modélisation. Chacun est doté d'une forme géométrique (une simple main pour ne pas surcharger l'affichage) et d'un point de vue, de plus on leur assigne un rôle spécifique.

La première catégorie d'utilisateur concerne les visiteurs. Un visiteur est un participant passif de la simulation. Il peut se déplacer, visiter les différentes pièces, mais il ne peut pas intervenir dans une tâche de modélisation. Toute personne qui rejoint une session en cours reçoit les droits d'un visiteur.

Le second type d'usager définit les sculpteurs. Un sculpteur participe activement à la modélisation. Il déforme les objets par l'intermédiaire de l'outil qu'il détient (outils qui seront décrits par la suite). A tout instant, un visiteur peut devenir un sculpteur. Il lui suffit de saisir un outil. Ces outils sont actuellement présentés sur des tables disséminées dans le monde virtuel. Les objets constituant la scène sont soit créés lors de la construction de l'environnement virtuel, soit créés par un utilisateur.

Notre objectif est de rendre la tâche de modélisation plus intuitive. Ainsi nos outils ont un rôle double : leur forme illustre non seulement leur fonction (marteau, crochet…) mais aussi la façon de les utiliser (taper avec un marteau, tirer avec un crochet…). La métaphore du sculpteur est réalisée : il y a correspondance directe entre le geste effectué par l'utilisateur dans le monde réel et l'action réalisée par son avatar dans le monde virtuel.

L'aspect collecticiel de cette application induit des risques de conflit quand deux sculpteurs désirent travailler sur le même objet. Nous avons, donc, mis en place trois systèmes de verrous. Le premier niveau est le moins contraignant : la seule restriction est que plusieurs utilisateurs n'ont pas le droit de déformer simultanément la même partie d'un objet. Le second niveau permet, si un objet est défini par une hiérarchie de formes géométriques, à des utilisateurs de travailler sur des éléments distincts (pied et plateau d'une table par exemple). Enfin le dernier niveau est le plus strict : un sculpteur peut se réserver l'exclusivité de l'objet. Pour verrouiller un objet, l'utilisateur doit se munir d'un verrou qu'il trouve sur la table des outils et qu'il place sur l'objet concerné. La pose d'un verrou est signifiée aux autres utilisateurs par l'apparition d'une boîte en fil de fer qui englobe l'objet.

4.2.2. Outils virtuels et déformations de formes libres virtuelles

Nous allons maintenant présenter le modèle mathématique associé à chacun de nos outils de déformation.

Il existe deux grandes catégories d'opérateurs dans les techniques de déformation.

- La déformation par manipulation directe : L'utilisateur transforme l'objet en modifiant directement sa structure géométrique [Barr 84]. C'est l'approche choisie par Delingette et al. [Delinguette 94] qui déforment des structures osseuses définies par des maillages

simplexes en appliquant des forces sur les nœuds du réseau. Le problème dans ce cadre est que la technique de déformation est liée au type de modèle géométrique des objets.

- La déformation par manipulation indirecte : Cette catégorie d'opérateurs est totalement indépendante des modèles géométriques. L'utilisateur n'agit plus directement sur la forme de l'objet. Il passe par le biais d'une entité de déformation qui l'englobe. Il s'agit pour l'utilisateur de déplacer les points de contrôle définissant le volume englobant (qui est appelé un treillis). D'autre part, ce modèle permet de faciliter les déformations locales d'un objet (le treillis n'englobant, dans ce cas, que la partie à déformer).

C'est ce type de déformation que nous avons choisi notamment en raison de son indépendance de tout modèle géométrique et de sa souplesse d'utilisation. Le modèle que nous avons retenu est plus connu sous le nom de FFD (*Free-Form Deformation*) [Sederberg 86]. Déformer grâce aux FFD consiste à enfermer l'objet (ou une partie de celui-ci) dans un bloc de plastique souple. Dès lors, les déformations ne sont plus appliquées directement à l'objet mais à l'espace qui l'englobe. Les transformations subies par l'objet ne sont qu'une interpolation de la déformation du volume englobant.

Les FFD ayant fait l'objet de nombreux travaux et extensions [Coquillard 90], nous ne rappellerons que brièvement leurs fondements mathématiques. Le volume englobant un objet (un parallélépipède, par exemple) définit une partition de l'espace en deux sous-ensembles : les points intérieurs et les points extérieurs. Ces derniers ne seront pas affectés par le processus de déformation qui se décompose en plusieurs étapes. Il s'agit tout d'abord de définir le volume englobant en indiquant son origine, ses axes directeurs et le nombre de points de contrôle selon chacun de ces axes. On crée ainsi un treillis de points dont les coordonnées sont exprimées dans le repère propre au treillis (Figure 56). Dans une seconde phase, il faut calculer les nouvelles coordonnées des points de l'objet dans le repère local. A partir de ce moment, toute déformation du treillis (déplacement d'un ou de plusieurs points de contrôle) peut être répercutée à l'objet par une interpolation de type Bézier, B-Spline, ...

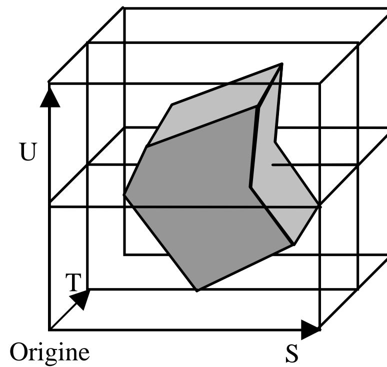


Figure 56 : Un exemple de treillis

L'exemple suivant illustre une interpolation à base de polynômes de Bernstein :

$$P_{ffd} = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i \left[\sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j \left[\sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^k P'_{ijk} \right] \right]$$

où (s, t, u) sont les coordonnées locales du point traité, et P'_{ijk} sont les points de contrôle après déplacement.

Cet outil illustre parfaitement la métaphore du sculpteur : du point de vue de l'utilisateur, il suffit de modeler le bloc de plastique pour déformer l'objet, ce qui est très intuitif et puissant. Néanmoins, dans la réalité, on s'aperçoit que la technique n'est pas toujours facile à maîtriser pour obtenir certains effets (comme aplatir une bosse).

Nous avons donc été amenés à concevoir un système qui masquerait totalement la notion de fonctions mathématiques pour la modélisation d'objets. Nous proposons à l'utilisateur un ensemble d'outils ne nécessitant que des paramètres ayant une signification réelle (partie de l'objet qui sera modifiée, taille de la tête d'un marteau…) et communiqués de la manière la plus naturelle (désignation graphique, choix du marteau…). Dans cette optique, la réalité virtuelle nous fournit la possibilité de mettre en œuvre ces concepts. Ainsi à un outil est associé un geste qu'il est facile d'analyser : une main qui tient un marteau et qui tape avec force une surface est un événement facilement interprétable (du moins dans le contexte d'un éditeur de formes). A cet effet, nous avons étudié un ensemble d'outils basés sur les FFD tels que des marteaux, des poinçons, des pinces… [Rubio 95].

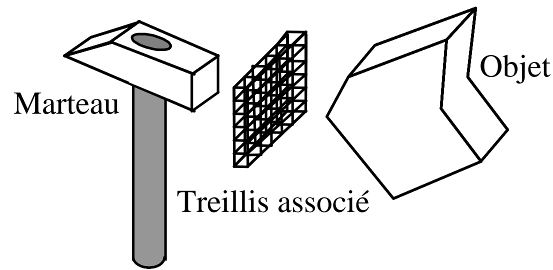


Figure 57 : Un type d'outils

Nous allons maintenant présenter un des outils réalisés : le marteau. La réalité virtuelle imposant des temps de réponse extrêmement brefs, nous avons défini des treillis aussi simples que possible. La hauteur et la largeur du treillis dépendent de la taille de la tête du marteau. La profondeur est calculée en fonction de la rigidité de l'objet. Précisons ici que cette rigidité n'a rien de physique mais elle permet de varier les effets. Cette même rigidité nous indique la densité de points de contrôle du treillis. Ainsi pour un objet donné et un type de marteau, nous créons un treillis (Figure 57).

Un coup de marteau est traité de la façon suivante : dès que la tête du marteau (en fait un point de la tête) pénètre dans un objet, le treillis est orienté selon la trajectoire de l'outil (deux dernières positions) et déplacé en translation au point d'impact. Les coordonnées des points de l'objet sont calculées dans le repère associé au treillis. Ce dernier est modifié (avec prise en compte de la pseudo-rigidité) puis l'objet est déformé. La figure suivante (Figure 58) illustre cet exemple en deux dimensions.

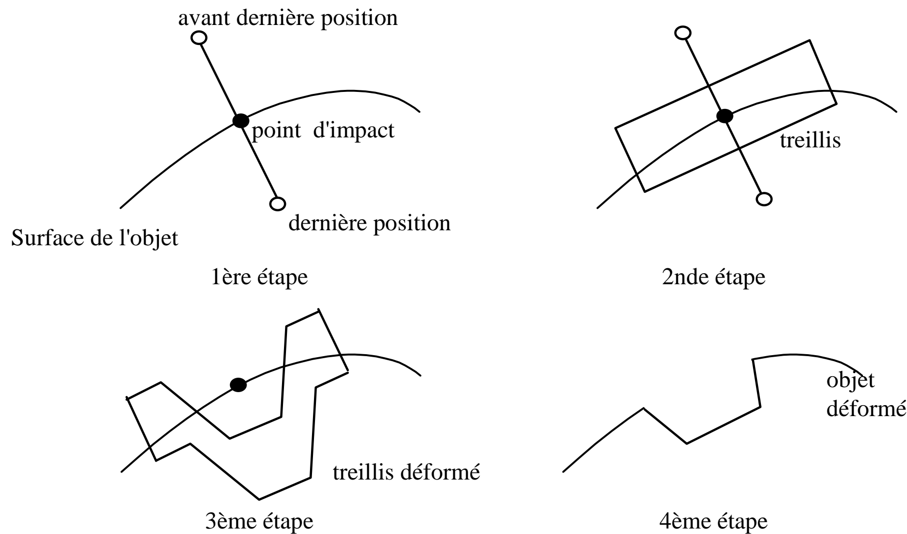


Figure 58 : Un coup de marteau

4.2.3. Définition des outils

Un outil dans le modèle de VIPER est représenté par un type particulier d'entité manipulée par une ou plusieurs entités et qui, à son tour, manipule des entités (Figure 59).

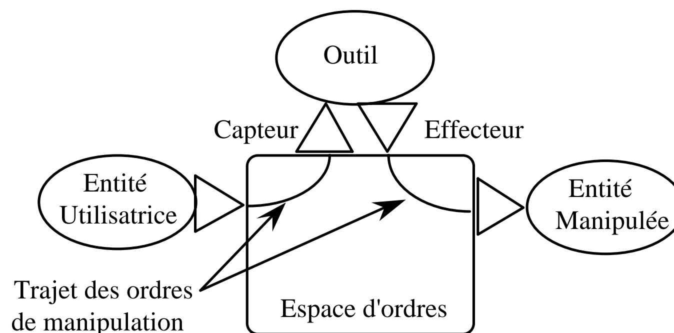


Figure 59 : Illustration de la manipulation via un outil

Un outil possède au moins un capteur qui reçoit les ordres qui lui sont transmis ainsi qu'un effecteur d'ordres qui envoie une commande de manipulation aux entités contrôlées par l'outil. Le comportement d'une telle entité est composé d’un composant comportemental réactif déclenché par la récupération d'un ordre reçu par l'outil et qui, en fonction du résultat de l'analyse, va produire, si nécessaire, un ordre de manipulation. Chaque entité utilisatrice d'un outil doit posséder un effecteur d'ordre. Symétriquement, chaque entité manipulée dispose d'un capteur d'ordre et d'un composant comportemental qui lui est associé et qui gère les réactions aux ordres reçus.

4.2.3.1. Un premier outil : le marteau

Nous allons illustrer ces principes par la mise en œuvre du marteau (Figure 60).

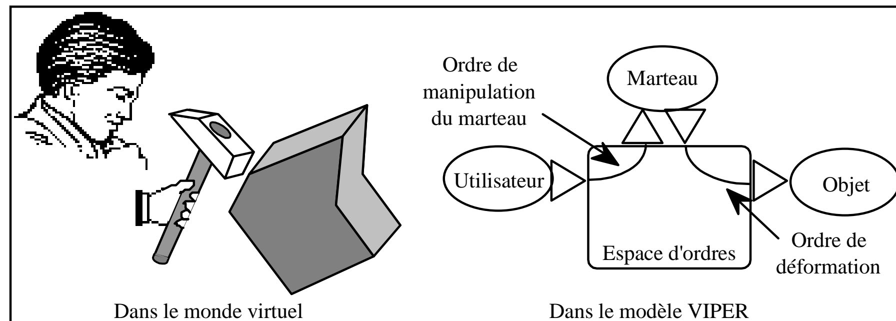


Figure 60 : Le coup de marteau

Dans cet exemple, l'entité représentant l'utilisateur (avatar) contrôle l'entité marteau en lui envoyant des ordres de manipulation (attraper et relâcher). Lorsque sa forme est saisie par un utilisateur (Figure 61), l'entité marteau commence à détecter les collisions avec son milieu pour déterminer si la tête du marteau heurte effectivement un objet. Pour cela, elle utilise un capteur de collisions auquel est associé un composant comportemental qui, lorsqu'une collision est détectée, envoie un ordre de déformation (appelé **martèlement**) à l'objet touché. Martèlement est un nouveau type d'ordre qui comprend comme paramètres les deux dernières positions de la tête du marteau, son diamètre et l'identificateur de l'avatar qui manipule le marteau. Dès la réception de cet ordre, l'entité qui gère l'objet vérifie si l'utilisateur à le droit de modifier sa forme (cf. 4.2.4) et si tel est le cas, demande à son effecteur de modifier sa forme.

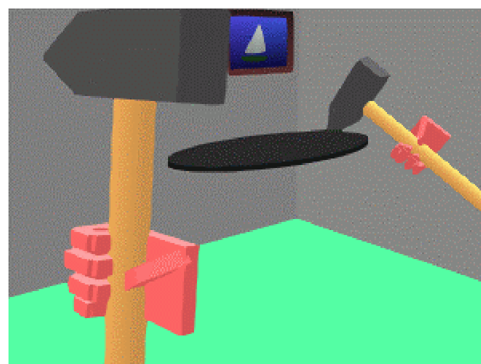


Figure 61 : Deux utilisateurs munis de marteaux

Pour qu'elle puisse être déformée, une entité doit posséder un composant comportemental permettant de traiter les ordres de déformation et une forme **déformable par FFD**. Cette dernière est un nouveau type de stimuli (nommé **FormeFFD**) qui hérite de la classe forme et qui possède des fonctions de déformation par FFD. Lorsqu'un effecteur veut déformer une telle forme, il appelle une des fonctions de déformation (dans le cas du marteau, ce sera la méthode **Marteler)** qui crée un treillis, le modifie conformément aux paramètres de l'ordre de déformation (positions et diamètre de la tête du marteau) et calcule sa nouvelle forme. De plus, l'espace des formes va envoyer, de façon transparente, une requête de mise à jour de la forme sur tous les sites participant à la simulation. Cette requête de mise à jour, qui contient le type de déformation et les paramètres de la déformation, est préparée par l'instance de la classe FormeFFD lorsque la méthode de déformation est invoquée et l'effecteur n'a qu'à la transmettre à l'espace des formes. Ensuite, lorsque la requête de mise à jour est reçue par un des sites, la méthode de déformation correcte est invoquée avec les bons paramètres.

4.2.3.2. Détails de la mise en œuvre

Le marteau est une entité migrante (de façon à permettre une meilleure interactivité) définie de la façon suivante :

```
DEF Marteau1 ViperEntitéMigrante {
fields [ MFString systemes,
         MFString composantsComportementaux,
         MFString capteurs,
         MFString effecteurs,
         MFString nouveauxStimuli,
         MFString etatInterne,
         MFString empaquetage,
         MFString parametres ]

systemes ["IRIX6", "WIN32" ]

composantsComportementaux [
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Ordres",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#Ordres",
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Collision",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#Collision" ]

capteurs [ "CapteurOrdre", "CapteurCollision" ]
effecteurs [ "EffecteurOrdre", "EffecteurForme" ]

nouveauxStimuli [
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Martelement>>Ordre",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#Martelement>>Ordre" ]

etatInterne [
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#EtatMarteau",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#EtatMarteau" ]


empaquetage [
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Empaquetage",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#Empaquetage" ]
```

```
parametres [ "FormeMarteau1" ]
}
```

L'entité est exécutable sur deux systèmes : IRIX6 et WIN32 et elle est définie dans deux librairies dynamiques : Marteau.so et Marteau.dll. Elle possède deux composants comportementaux. Le premier composant, qui a pour nom de classe **Ordres** (Figure 62), traite les ordres de manipulation envoyés par un avatar.

```
class Ordres : public ComposantComportemental
{
protected :
     EtatMarteau*      MonEtat;
Effecteurd'Ordre* MonEffecteurD'Ordre;
EffecteurDeForme*      MonEffecteurDeForme;

public :
     Ordres(Entité &);

     ~Ordres();

     virtual void Action(DéclencheurId);
};

ComposantComportemental* NouveauOrdres(Entité& pEntité) {
     return new Ordres(pEntité);
}
```

Figure 62 : Extrait de l'interface du composant comportemental Ordres

Le traitement d'un ordre du type Attraper par le composant comportemental est effectué par le code qui suit :

```
// définition du comportement proprement dit
void Ordres::Action(DéclencheurId pId) {
 // récupération du capteur d'ordre qui a appelé le composant
 CapteurOrdre* lCapteurOrdre
                  = (CapteurOrdre&)MonEntité->MesDeclencheurs(pId);

 // traite l'ordre qui viens d'être reçu
 Ordre* UnOrdre = lCapteurOrdre->OrdreLePlusRécent();

 if (UnOrdre->Type() == TAttraper) {
  // une entité veut attraper le marteau
  if (! MonEtat->Attrapé) {
      // le marteau n'est pas déjà saisi
      IdNoeud lParent; // identificateur d'un nœud du graphe de la scène
      WTp3 lPos; // vecteur de 3 flottants contenant une position
      WTq  lOri; // vecteur de 4 flottants contenant un quaternion

      UnOrdre->Paramètres(&lParent, lPos, lOri);

      // enlève le nœud racine de la forme du graphe de la scène
      MonEtat->MaForme.EnlèveNoeud(MonEtat->MaForme.MonIdNoeud);
      MonEffecteurDeForme->Envoyer(MonEtat->MaForme);

      // place le nœud racine de la forme sous le nœud paramètre de
      // l'ordre avec les positions et orientations précisées
      ...
      MonEtat->Attrapé = TRUE;

      // acquittement de l'ordre
      Acquittement* Réponse = new Acquittement;
      Réponse->IdRécepteurs(UnOrdre->IdEmetteur);
      Réponse->IdEmetteur = MonEntité->MonId;
      MonEffecteurD'Ordre->Envoyer(Réponse);
  } else {
      // le marteau est déjà saisi
      // on envoie un acquittement négatif
      AcquittementNégatif* Réponse = new AcquittementNégatif;
      Réponse->IdRécepteurs(UnOrdre->IdEmetteur);
      Réponse->IdEmetteur = MonEntité->MonId;
      MonEffecteurD'Ordre->Envoyer(Réponse);
  }
 }
 // traitement des autres types d'ordres
 ...
}
```

Figure 63 : Extrait du code traitant un ordre du type Attraper

Cet extrait de code présente l'utilisation des espaces d'ordres et de formes. Il est intéressant de noter que, grâce à VIPER, le développeur du comportement d'une telle entité n'a pas à gérer la construction et l'émission des messages de mise à jour de sa forme.

Collision, le second composant comportemental, envoie, lors d'une collision entre la tête du marteau et une forme déformable, un ordre de déformation à l'entité qui possède cette forme. La figure suivante (Figure 64) présente l'algorithme de haut niveau de ce composant comportemental.

```
// définition du comportement proprement dit
void Collision::Action(DéclencheurId pId)
Début

 Récupérer le capteur de collision qui a appelé le composant;

 Récupérer l'identificateur du nœud racine de la forme touchée;

 Si la forme touchée est déformable alors

     Récupérer l'identificateur de l'entité qui possède la forme;

     Allouer un ordre de déformation du type martèlement;

     Préciser les paramètres de l'ordre (deux dernières positions
     du marteau et diamètre de sa tête);

     Envoyer l'ordre de déformation à l'entité qui possède la forme;
 Fin Si;

Fin
```

Figure 64 : Algorithme du composant comportemental Collision

L'état interne du marteau comporte l'identificateur de l'entité manipulatrice, une valeur booléenne indiquant si l'objet est manipulé, la position de la pointe du marteau, le diamètre de sa tête, l'identificateur du nœud principal de l'entité et le nom de ce nœud. L'état interne possède, de plus, un attribut qui conserve la date du début de la manipulation du marteau par une entité et un délai. Lorsque la date courante est supérieure à la date de début de manipulation à laquelle on ajoute le délai alors, l'entité va migrer sur le site de son entité manipulatrice de façon à améliorer l'interactivité de la manipulation. La prise de décision de migrer est gérée automatiquement par un composant comportemental standard que possède les entités migrantes. L'extrait de code suivant (Figure 65) présente l'initialisation de certains attributs et du composant comportemental standard.

```
EtatMarteau::EtatMarteau(Entité* MonEntité) {
  // recupère certains pointeurs présents dans l'entité
  ...
  Manipulation = FAUX;
  // rècupère le noeud correspondant
  // à la pointe du marteau
  WTnode* NoeudPointe =
    MonEspaceDeFormes->RécupèreNoeudParNom(MaForme, "POINTE");
  PositionPointe = WTnode_getposition(NoeudPointe);
  // calcule le diamètre de la tête à partir de la
  // boîte englobante de la tête
  ...
  DiamètreTête = ...;
  IdNoeudForme = MaForme->MonIdNoeud;
  // copie le nom de la forme
  strcpy(NomNoeudForme, MaForme->MonNom);

  DélaiAvantMigration = 1000.0; // 1 seconde
  DébutManipulation.RemiseAZéro();

  MonGestionnaireDeMigration->DonneLesParamètres(
    &DébutManipulation, &DélaiAvantMigration);
}
```

Figure 65 : Initialisation de l'état interne de l'entité Marteau

La figure suivante (Figure 66) présente le code d'empaquetage de l'état interne. Le code de dépaquetage est obtenu en remplaçant les opérateurs << par des opérateurs >>.

```
boolean Empaquetage::Empaquetage(Message* M) {
  M << MonEtatInterne->IdManipulateur;
  M << MonEtatInterne->Manipulation;
  M << MonEtatInterne->PositionPointe;
  M << MonEtatInterne->DiamètreTête;
  M << MonEtatInterne->IdNoeudForme;
  M << MonEtatInterne->NomNoeudForme;
}
```

Figure 66 : Empaquetage de l'entité Marteau

L'ordre Martèlement est défini par la classe suivante (Figure 67) :

```
class Martelement : public Ordre
{
protected :
     float       DiamètreTête;
     WTp3        DernièrePosition; // WTp3 est un float[3]
     WTp3        AvantDernièrePosition;
     IdEntité    Avatar;

public :
     Martelement(Entité &);

     ~Martelement();

     // permet de préciser la valeur des paramètres
     // de l'ordre
     void DonnerLesParamètres(float, WTp3, WTp3, IdEntité);
     // permet de récupérer la valeur des paramètres
     void Paramètres(float&, WTp3, WTp3, IdEntité&);

     // fonctions d'empaquetage et de dépaquetage dans un
     // message
     boolean Empaquetage(Message*);
     boolean Dépaquetage(Message*);
};
```

Figure 67 : Classe Martèlement

Les objets déformables sont gérés par des entités migrantes définies de la façon suivante :

```
DEF ObjetDeformable1 ViperEntitéMigrante {
fields [     MFString systemes,
MFString composantsComportementaux,
MFString capteurs,
MFString effecteurs,
MFString nouveauxStimuli,
MFString etatInterne,
MFString empaquetage,
MFString parametres ]
systemes ["IRIX6", "WIN32" ]

composantsComportementaux [
"http://virtus.irit.fr/VIPER/Deformable.so#IRIX6#Ordres",
"http://virtus.irit.fr/VIPER/Deformable.dll#WIN32#Ordres"]

capteurs [ "CapteurOrdre" ]
effecteurs [ "EffecteurForme" ]

nouveauxStimuli [
"http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Martelement>>Ordre",
"http://virtus.irit.fr/VIPER/Marteau.dll#WIN32#Martelement>>Ordre",
"http://virtus.irit.fr/VIPER/Deformable.so#IRIX6#FormeFFD>>Forme",
"http://virtus.irit.fr/VIPER/Deformable.dll#WIN32#FormeFFD>>Forme"]

etatInterne [
"http://virtus.irit.fr/VIPER/Deformable.so#IRIX6#EtatObjet",
"http://virtus.irit.fr/VIPER/Deformable.dll#WIN32#EtatObjet"]

empaquetage [
"http://virtus.irit.fr/VIPER/Deformable.so#IRIX6#Empaquetage",
"http://virtus.irit.fr/VIPER/Deformable.dll#WIN32#Empaquetage"]

parametres [ "FormeObjetDeformable1" ]
}
```

Ces entités possèdent un composant comportemental qui gère les ordres qui leur sont envoyés. Ce composant, appelé Ordres, a comme méthode action le code suivant (Figure 68) :

```
// définition du comportement des objets déformables
void Ordres::Action(DéclencheurId pId) {
 // récupération du capteur d'ordre qui a appelé le composant
 CapteurOrdre* lCapteurOrdre
          = (CapteurOrdre&)MonEntité->MesDeclencheurs(pId);

 // traite l'ordre qui viens d'être reçu
 Ordre* UnOrdre = lCapteurOrdre->OrdreLePlusRécent();

 // récupère l'identificateur du type de l'ordre de martèlement
 // à partir de l'une de ces URLs
 TMartèlement = MonEspaceDOrdres->TypeDe(
     "http://virtus.irit.fr/VIPER/Marteau.so#IRIX6#Martelement>>Ordre");

 if (UnOrdre->Type() == TMartèlement) {
     // une entité marteau a frappé sur l'objet

     // récupérer les paramètres de l'ordre
     Martèlement* UnMartèlement = (Martèlement*)UnOrdre;
     float       DiamètreTête;
     WTp3        DernièrePosition, AvantDernièrePosition;
     IdEntité    Avatar;
     UnMartèlement->Paramètres(    DiamètreTête, DernièrePosition,
                                   AvantDernièrePosition, Avatar);

     // si l'objet est vérouillé par un autre avatar
     if (Vérouillé && (IdAvatarVérouillé != Avatar)) {
           // envoyer un acquittement négatif
           ...
     } else {
           // si l'objet est composé de sous-parties déterminer
           // laquelle a été frappée et si elle est vérouillée
           // par un autre avatar
           ...
           // dans tous les autres cas déformer la forme
           MaForme->Marteler(      DiamètreTête, DernièrePosition,
                                   AvantDernièrePosition);
           MonEffecteurDeForme->Envoyer(MaForme);
     }
 }
    // traitement des autres types d'ordres
    ...
}
```

Figure 68 : Traitement des ordres reçus par un objet déformable

Enfin, la classe FormeFFD est définie comme suit (Figure 69) :

```
class FormeFFD : public Forme
{
protected :
     // pseudo-rigidité de l'objet
     float Rigidité;

     Treillis    MonTreillis;
     // nombre de sommets intérieurs au treillis
     int         NombreDeSommetsIntérieurs;
     // sommets intérieurs au treillis
     Sommet*     SommetsIntérieurs;

     // Pour les déformations par marteau
     float       DiamètreTête;
     WTp3        DernièrePosition;
     WTp3        AvantDernièrePosition;

     // d'autres attributs pour d'autres types de
     // déformations (crochet par exemple)
     ...

public :
     FormeFFD (Entité &);

     ~FormeFFD ();

     void Marteler(float, WTp3, WTp3);

     // autres fonctions de déformations
     ...

     // fonctions d'empaquetage et de dépaquetage dans un message
     boolean Empaquetage(Message*);
     boolean Dépaquetage(Message*);
};

Forme* NouveauFormeFFD() {
     return new FormeFFD ;
}
void FormeFFD::Marteler(float Diamètre, float DernièrePosition,
                      float AvantDernièreposition) {
     DiamètreTête = Diamètre;
     ::DernièrePosition = DernièrePosition;
     ::AvantDernièrePosition = AvantDernièreposition;

     // crée un treillis et le déforme selon les paramètres
     // du martèlement tout en tenant compte de la rigidité
     MonTreillis.InitialiseDéformationAvecMarteau(DiamètreTête,
           DernièrePosition, AvantDernièrePosition,
           Rigidité);

     // calcule les points intérieurs
     MonTreillis.CalculPointsInterieurs(MonNoeudGraphique,
           NombreDeSommetsIntérieurs, SommetsIntérieurs);

     // déforme l'objet
     MonTreillis.Déforme(MonNoeudGraphique,
           NombreDeSommetsIntérieurs, SommetsIntérieurs);
}
```

Figure 69 : La classe FormeFFD

4.2.3.3. Un autre outil : le crochet

Le deuxième outil que nous avons mis en œuvre est un crochet qui permet de déformer un objet interactivement. Il se présente sous la forme d'un stylet doté d'une pointe à l'une de ses extrémités. Dès que le crochet touche la surface d'un objet, nous créons un treillis (Figure 70, première et seconde étapes). A chaque déplacement de l'outil, nous calculons la déformation à appliquer au treillis (Figure 70, troisième étape). L'étendue de cette déformation sur la surface dépend des caractéristiques définissant le crochet. Puis nous déformons la partie de l'objet affectée (Figure 70, quatrième étape). L'utilisateur a ainsi l'impression que la surface de l'objet s'est « collée » à l'outil et qu'elle suit tous ses déplacements. Enfin, lorsque l'utilisateur lâche le stylet, celui-ci se décolle de l'objet.

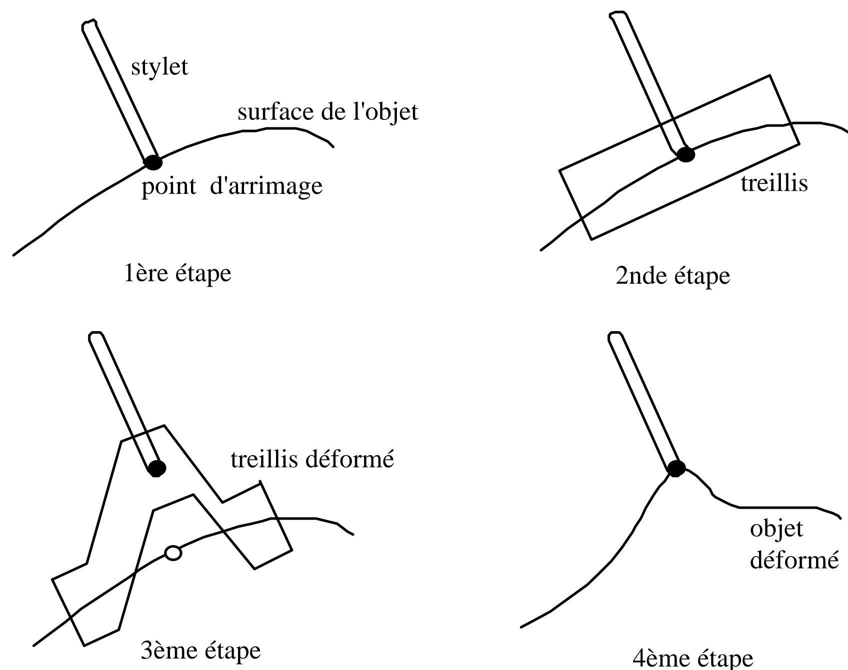


Figure 70 : Déformation par crochet

Pour ne pas trop ralentir la simulation, seuls les points sont déformés pendant la déformation interactive et la forme est affichée en filaire (Figure 71 – sur cette figure nous avons affiché avec une couleur différente la partie de l'objet qui se déforme pour pouvoir visualiser l'étendue de cette déformation).

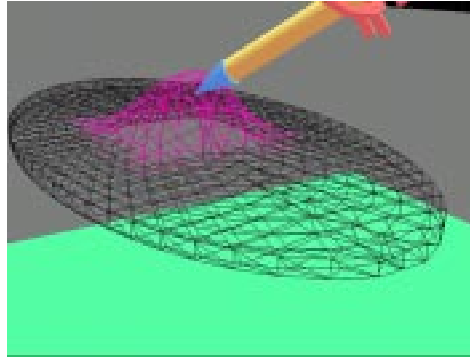


Figure 71 : Un autre outil de déformation : le crochet

Les nouvelles normales de l'objet sont calculées lorsque le crochet se détache de la forme car c'est un calcul très lourd et il est réaffiché normalement.

4.2.4. Aspects multi-utilisateurs spécifiques à l'application

Les aspects multi-utilisateurs entraînent des problèmes liés aux conflits risquant de survenir lorsque deux utilisateurs tentent de réaliser des déformations antagonistes. Ces conflits peuvent intervenir à plusieurs niveaux correspondant à la granularité de l'information manipulée.

Si l'unité de manipulation est l'objet, deux sculpteurs devront obligatoirement travailler sur des objets différents. Dans ce cas, nous proposons la pose explicite d'un verrou graphique sur l'objet. Ce verrou devient alors un outil qui ne peut être manipulé que par un seul utilisateur qui va le saisir, l'amener sur un objet et le lâcher (*drag and drop* des interfaces 2D). L'objet sera désormais verrouillé (Figure 72) jusqu'à ce que ce même utilisateur retire le verrou.

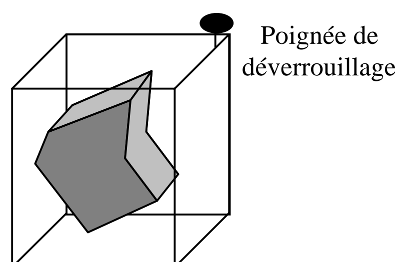


Figure 72 : Un verrou posé sur un objet

Si l'unité de manipulation est une sous partie de l'objet (cas d'un objet défini par une hiérarchie de formes géométriques) nous proposons la pose d'un verrou sur cette sous partie.

Si aucun verrou n'est placé, on autorise deux manipulations sur un même objet en vérifiant qu'il n'y ait pas de recouvrement spatio-temporel des différents treillis mis en jeu. Ce concept est intéressant car il nous permettra à terme de faire évoluer notre application vers la collaboration de plusieurs sculpteurs à une même déformation (torsion de barres, aplatissement, étirements...).

4.2.5. Conclusions sur cette application

Nous avons décidé d'illustrer les idées présentées précédemment en réalisant une application de modélisation de terrain. Nous fournissons aux utilisateurs la matière première (terrain), des outils et des éléments (présentés sur une table) pour aménager le paysage qu'ils souhaitent créer. Les utilisateurs creusent des lacs et des rivières ou élèvent des montagnes. Ils peuvent aussi disposer les éléments du décor (arbres, maisons, …). La présence de plusieurs utilisateurs permet d'assigner à chacun une tâche bien précise (creuser des vallées, des lacs…). Les images qui suivent (Figure 73 et Figure 74) donnent un aperçu d'une session de modélisation.

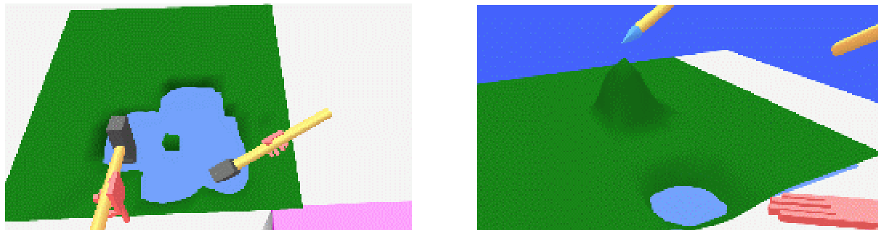


Figure 73 : Sessions de modélisation de paysage multi-utilisateurs

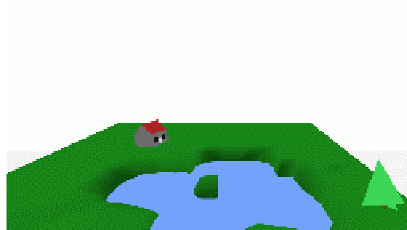


Figure 74 : Exemple de terrain modélisé

4.3. GRS (*Generic Railway System*)

Ce projet a été réalisé par la société CISI avec qui notre équipe de recherche travaille sur de nombreux projets en réalité virtuelle. GRS a été développé pour la SNCF avec pour objectif de lui permettre d'estimer les avantages de la réalité virtuelle. Le logiciel comprend : une boîte à outils réutilisable dont le modèle est dérivé de VIPER qui permet de gérer des entités virtuelles quelconques et un ensemble d'entités et de comportements qui permettent de gérer différentes applications dans le domaine de la simulation en relation avec les chemins de fer. Il est important de noter qu'à l'heure actuelle le logiciel est mono-utilisateur (en accord avec la spécification fournie par la SNCF) mais que le modèle peut permettre son extension aisée vers des simulations partagées par plusieurs utilisateurs.

Figure 75 : La « Gare du Nord »

La première version du simulateur a lieu dans la « Gare du Nord » (Figure 75). La gare a été reproduite fidèlement à partir de plans, de photographies et de vidéos. Le simulateur présente aussi une maquette virtuelle d'un TGV à deux niveaux qui peut, lui aussi, être visité (Figure 76).

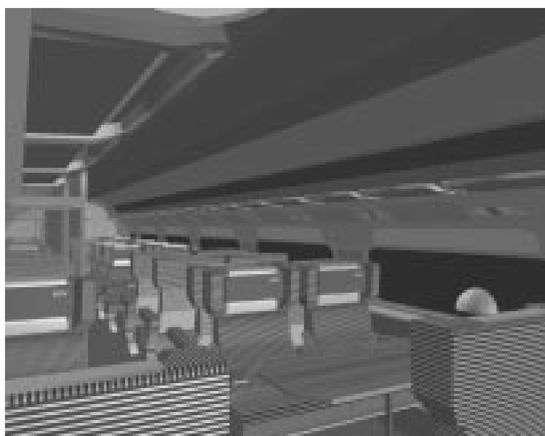


Figure 76 : Intérieur du TGV à deux niveaux

4.3.1. Modes d’interaction avec la simulation

L'utilisateur interagit avec la simulation soit avec une interface standard comprenant une souris 3D, un écran et si nécessaire des lunettes stéréoscopiques, soit avec un *joystick* 3D et un casque de réalité virtuelle.

Ces deux interfaces offrent les mêmes fonctionnalités permettant l’accès aux modes de déplacement au sol et en vol, à l’édition de scène, à la création de visites guidées de la gare ou encore au contrôle de la densité d’une foule virtuelle qui évolue dans la station (Figure 77).

Figure 77 : Une foule dans la « Gare du Nord »

4.3.2. Les comportements

De plus, plusieurs objets de la scène possèdent un comportement : des horloges affichent l'heure exacte, des escaliers mécaniques sont activés lorsque l'utilisateur avance sur leur première marche, des portes peuvent être ouvertes, des lumières allumées ou éteintes et il est même possible de poinçonner un titre de transport. L'architecture utilisée dans ce cadre reprend la plupart des fonctionnalités du modèle de définition des comportements de VIPER et il est ainsi possible d'ajouter dynamiquement de nouveaux comportements et de modifier les comportements existants grâce à l'utilisation de librairies chargées dynamiquement.

Pour compléter le réalisme de la simulation, divers effets sonores sont diffusés dans la gare. Ces effets on été produits à partir de prises de son réelles effectuées dans la Gare du Nord. Ainsi, les annonces diffusées aléatoirement, le bruit des escaliers mécaniques et les bruits du mécanisme de poinçon des titres de transport améliorent la sensation d'immersion.

Pour le département de recherche de la SNCF, cette plate-forme est un système évolutif permettant l'étude du potentiel de la réalité virtuelle dans le domaine des chemins de fer. Les domaines d'application sont : la modélisation de nouvelles gares et de trains, l'amélioration des services pour la clientèle et à terme la formation des techniciens aux systèmes de maintenance.

4.4. PROVIS (PROtotypage VIrtuel de Systèmes)

PROVIS [Balet 97], initiée par le CNES est réalisée par CISI, en collaboration avec l'IRIT pour la partie distribuée, permettra la visualisation et l'interaction multi-utilisateur avec des satellites en phase de conception. Une première version de cette application industrielle a déjà été réalisée par Olivier Balet. Avec qui, nous sommes, actuellement, en train de perfectionner les aspects multi-utilisateur [Torguet 97] que nous allons présenter dans cette partie.

4.4.1. But du projet PROVIS

De nos jours, les maquettes physiques sont encore employées dans une large gamme d'activités (fabrication automobile, industrie aérospatiale ou architecture). Elles restent encore la seule solution pour tester l'intégration, l'accessibilité et les exigences d'espace de leurs équipements. Néanmoins, leur construction est longue, fastidieuse et, la plupart du temps, coûteuse. Enfin, les maquettes physiques, une fois réalisées, ne sont ni flexibles ni faciles à manipuler.

Considérant ces observations, le CNES et CISI ont lancé le projet de recherche PROVIS en 1995. L'objectif principal de ce projet est de trouver des solutions, qui permettraient à des concepteurs de satellite de créer, manipuler et étudier leurs modèles en utilisant des maquettes numériques appelées prototypes.

Afin d'augmenter l'efficacité de cette expérience et d'offrir une interface aussi conviviale que possible, nous employons une interface de réalité virtuelle « sur le bureau » (*Desktop VR*) associée à des ordres de haut niveau. Le but est essentiellement d'éviter aux utilisateurs d'apprendre un nouveau système pour réaliser leur tâche.

Actuellement, la plupart des systèmes qui gèrent des environnements 3D sont contrôlés par des interfaces homme-machine (IHM) en 2D. Le manque de corrélation entre les manipulations et leurs effets, d'une part, et la distance importante entre l'image mentale qu'ont les utilisateurs et les modèles édités, d'autre part, sont les désavantages majeurs de cette solution. D'un autre côté, les interfaces 3D totalement immersives sont le plus souvent inadaptées à des applications industrielles. Dans ce cas, le système de menu est parfois remplacé par des menus 3D [Butterworth 92] flottants dans l'environnement virtuel sans fournir plus d'efficacité que leur équivalent en 2D. De plus, activer un bouton 3D (avec l'aide d'un capteur 3D) n'apporte rien car c'est une opération au moins aussi longue et plus fatigante que l'activation de son équivalent 2D avec une souris conventionnelle. Pour toutes ces raisons nous avons choisi de créer une architecture logicielle ouverte qui permettra la création d'interfaces mixtes (2D/3D) basées sur le concept d'entités autonomes [Balet 96].

De plus, nous souhaitons permettre à plusieurs équipes, situées dans des locaux distants, d'étudier et de construire des prototypes virtuels d'une façon collaborative afin de soutenir et d'améliorer les procédés existants d'ingénierie concurrente.

4.4.2. L'architecture de PROVIS

PROVIS a été conçu pour permettre à des utilisateurs non experts de créer des prototypes virtuels qui peuvent, par exemple, être employés pour tester les exigences d'espace et l'intégration d'équipements ou pour définir et mettre en œuvre des mécanismes dynamiques entre les parties mobiles d'un système, tout en explorant et en accroissant la documentation « en ligne » du prototype.

En plus d'un emploi intuitif qui dérive d'une forte interactivité, l'apport principal de ce logiciel est qu'il permet une évaluation qualitative des composants du prototype (intégration, exigences d'espace, poids, consommation électrique, etc.). La représentation graphique 3D est en fait la partie visible d'un composant offrant d'autres attributs : des attributs physiques (poids, longueur, largeur…), des attributs fonctionnels (par exemple l'appartenance de l'objet à un sous-système), des attributs documentaires (texte et commentaires vocaux, liens hypertextes…). L'enveloppe graphique est une métaphore d'accès à d'autres données, qui sont atteintes en désignant physiquement tel ou tel composant

PROVIS est basé sur l'architecture de VIPER de façon à permettre le travail coopératif et la répartition des données des prototypes C'est pourquoi, grâce à PROVIS, plusieurs concepteurs situés sur des sites éloignés peuvent travailler sur le même prototype partagé et confronter leurs idées.

4.4.3. Les entités de PROVIS

Dans sa version la plus simple, PROVIS est défini par 3 entités principales :

- Le constructeur (*builde*r) : l'objectif de cette entité est de fournir à l'utilisateur des commandes de haut niveau pour bâtir des prototypes 3D. Elle gère la bibliothèque de modèles, l'interaction avec les prototypes et des fonctions spécifiques à l'application

(assemblage de composant, ajout de documentation…). En fait, il précise la fonctionnalité d'application.

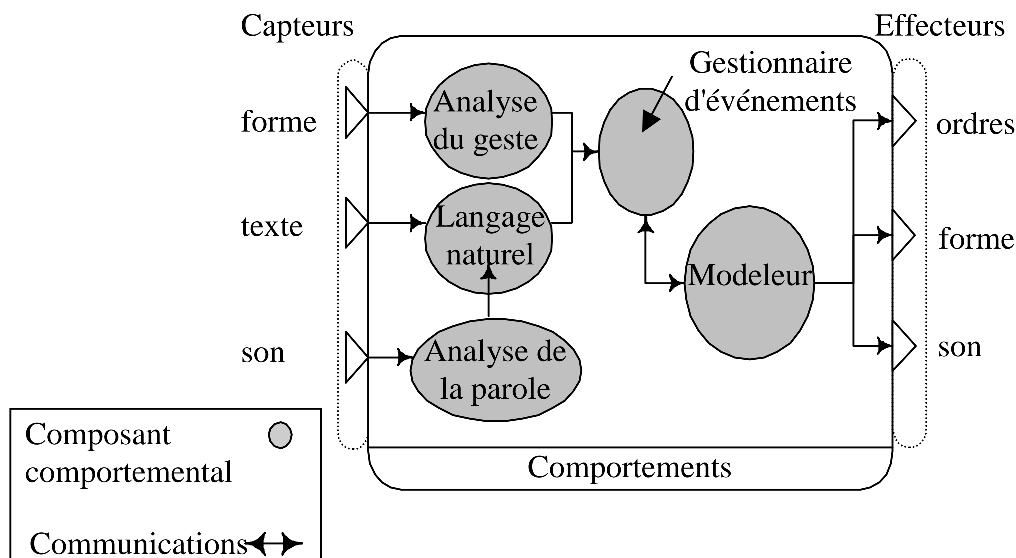


Figure 78 : Une vue simplifiée des composants comportementaux du constructeur

La figure précédente (Figure 78) présente une vue simplifiée de cette entité : plusieurs composants comportementaux (modules d'entrée) analysent et contrôlent les stimuli captés par l'entité, tels que des gestes ou des sons, pour les convertir en des événements normalisés. Ces événements sont envoyés au gestionnaire d'événements qui les fusionne en une commande unique qui est ensuite envoyée au modeleur. Grâce à son gestionnaire d'événements multimodal, PROVIS permet la combinaison de plusieurs dispositifs d'entré (gant de données, système de reconnaissance de la parole, *spaceball*, souris 3D, souris...) pour construire une même commande. Par exemple, l'utilisateur peut dire au système : « Met cela sur la table », où **cela** remplace une bouteille (précédemment sélectionnée avec une souris 3D) qui est récupérée dans l'historique du dialogue grâce à ses attributs sémantiques et syntaxiques.

De plus, le gestionnaire d'événements multimodal permet de combiner en une même commande des données provenant d'utilisateurs différents (Figure 79). Nous employons une description homogène pour définir la signature de chaque commande de notre système :

Par exemple, la commande de déplacement est précisée comme suit:

```
DEPLACER (Objet, Force, ...)
[
   MULTI-UTILISATEUR;
   DELAI = 100;
]
```

Ainsi, `DEPLACER` peut recevoir plusieurs forces comme paramètres. Cette commande est `MULTI-UTILISATEUR`, c'est à dire qu'elle peut recevoir ses paramètres de plusieurs utilisateurs (ce qui est très utile pour simuler les procédures d'entretien quand un élément nécessite plusieurs utilisateurs pour être déplacé). L'attribut `DELAI` précise le délai (donné en millisecondes) entre l'instant de réception et l'exécution de la commande. Cet attribut est employé par le gestionnaire d'événements pour définir la durée d'attente d'événements nouveaux avant l'envoi de la commande complète. De plus, le composant gestionnaire d'événements multimodal est un composant comportemental par défaut de chaque élément de prototype. Cela permet à ces éléments de pouvoir être manipulé par plusieurs utilisateurs ou par un utilisateur unique utilisant ses deux mains.

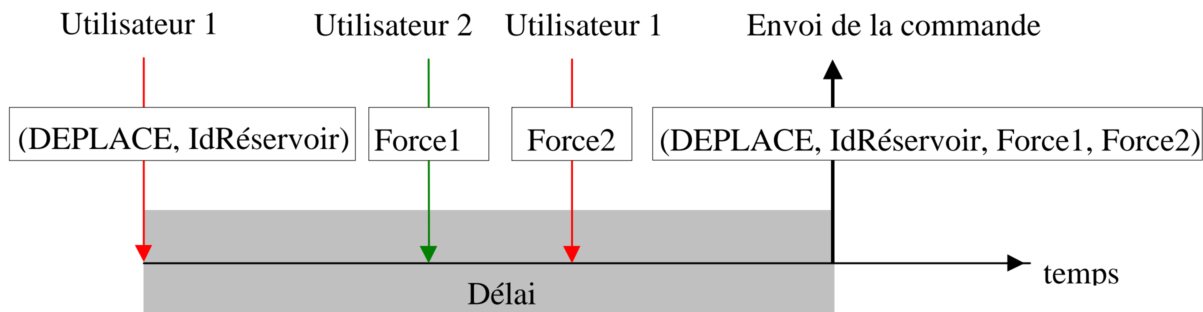


Figure 79 : Construction d'une commande multi-utilisateur

Le composant modeleur contient des procédures qui affectent la scène telles que des procédures géométriques ou des fonctions déclaratives. L'objectif des fonctions déclaratives [Gaildrat 93] est d'offrir à l'utilisateur la possibilité de décrire des scènes sans préciser de données numériques. Il permet à l'utilisateur d'exprimer son image mentale de la scène dans un langage quasi-naturel, incluant des commandes de haut niveau, des deictiques et des paramètres flous : « Place la bouteille sur la table », « Déplace la vers la gauche »…

A la suite d'études ergonomiques, nous avons décidé de limiter notre langage gestuel à 3 types de commandes : sélectionner, déplacer en translation et déplacer en rotation. Pour clarifier l'affichage de la scène, nous avons aussi décidé d'attribuer une forme simple aux avatars qui représentent les autres utilisateurs (un pointeur 3D avec une couleur spécifique à chaque utilisateur). De plus, les avatars peuvent aussi être représentés par une fenêtre 2D (qui présente une vidéo temps-réel du visage de l'utilisateur) et par la voix digitalisée de l'utilisateur (Figure 80).

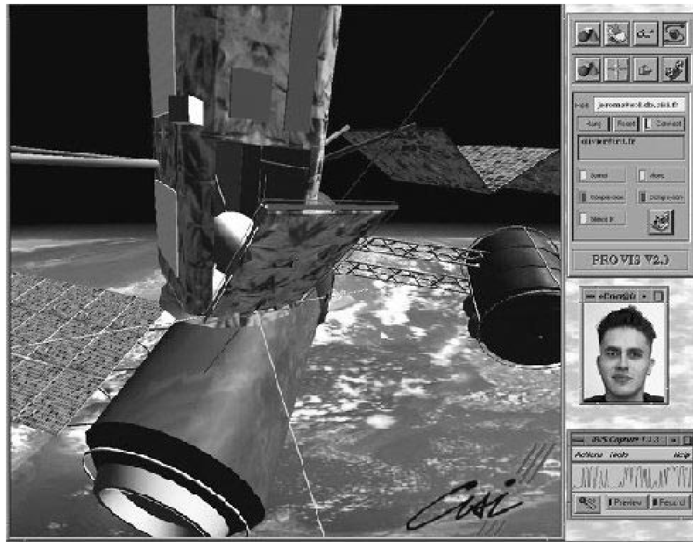


Figure 80 : Construction d'un satellite avec un utilisateur distant

- L'expert : l'objectif de cette entité est de vérifier la cohérence d'un prototype : évaluation du poids, consommation électrique et intégrité structurelle. Cette entité doit contenir la connaissance des experts. Elle emploie divers attributs (par exemple des attributs sémantiques) des composants du prototype afin d'évaluer la validité du prototype.

- Les IHM 2D : ces interfaces homme machine sont un cadre où d'autres entités peuvent ajouter ou enlever leurs propres « *widgets* » (Figure 81a). Une entité peut ajouter ses *widgets*, en envoyant des ordres à une IHM 2D (il y en a une par site), lorsque sa forme est sélectionnée par un utilisateur (Figure 81b). Par la suite, l'entité recevra des stimuli, via son capteur d'ordre, à chaque fois que l'une de ses *widgets* sera activée. Les IHM 2D ont été conçues de façon à pouvoir être affichées indifféremment sur une station de travail exécutant un serveur X Window ou sur un PC exécutant Windows 95 ou Windows NT.

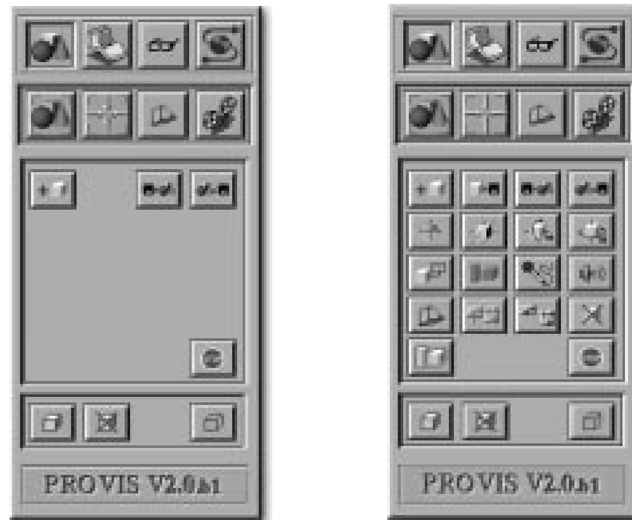


Figure 81 : (a) Une IHM 2D avec les *widgets* du constructeur (b) l'IHM contient maintenant les *widgets* d'un composant de satellite

Grâce à l'architecture distribuée de VIPER, l'entité IHM 2D peut s'exécuter sur une station de travail tandis que le concepteur travaille sur une autre station. Cette possibilité est très intéressante lorsque la station de travail employée par l'utilisateur affiche une vue plein écran de la scène. En effet, certaines stations de travail ne peuvent afficher en même temps, une vue stéréoscopique et des *widgets* 2D sur le même écran.

4.4.4. Les périphériques spécialisés utilisés

L'interaction avec des mondes virtuels tridimensionnels ne diffère pas fondamentalement de l'interaction avec des mondes virtuels bi-dimensionnels. Dans les deux environnements, l'utilisateur doit sélectionner des objets, entrer des valeurs, positionner, orienter ou mettre à l'échelle des modèles. Cependant, l'utilisation de périphériques classiques (tels qu'une souris) pour réaliser des manipulations en 3D ralentit souvent le travail de l'utilisateur. C'est pourquoi nous proposons une *spaceball* (périphérique incrémental à six degrés de libertés rappelant un trackball) et un gant de données pour interagir le plus naturellement possible avec les entités 3D. De plus, nous avons décidé de fournir une vue stéréoscopique de l'environnement virtuel à l'utilisateur (soit grâce à des lunettes stéréoscopiques soit grâce à un casque de visualisation) afin de tirer parti, au mieux, de la puissance ajoutée par la troisième dimension. Pendant la phase d'évaluation de PROVIS, les utilisateurs ont exprimé leur préférence pour la *spaceball* et les lunettes stéréoscopiques. C'est pour cette raison que nous avons décidé d'employer une

interface *desktop VR* (Figure 82) qui permet à l'opérateur de travailler pendant plusieurs heures sans l'inconfort introduit par un gant de donnée (fatigue musculaire) et un casque de visualisation (tension oculaire et poids excessif). Dans l'interface choisie, l'affichage stéréoscopique est réalisé grâce à des lunettes stéréoscopiques dotées de capteurs 3D qui permettent de coupler les mouvements du point de vue en fonction de ceux de la tête de l'opérateur.

Figure 82 : Les périphériques choisis

De plus, un microphone connecté à un PC (équipé d'une carte de reconnaissance vocale) permet d'analyser la parole de l'utilisateur. Les phrases reconnues sont ensuite envoyées à la station hôte. Des évaluations de PROVIS ont été réalisées sur des stations de travail SGI Onyx/RE^2 et Indigo Maximum Impact reliées à des PC pour la reconnaissance de la parole.

4.4.5. Travailler avec PROVIS

Grâce à l'architecture décrite précédemment, PROVIS permet à des utilisateurs de manipuler naturellement des objets 3D. PROVIS offre, de plus, plusieurs *widgets* 3D [17] qui accélèrent la tâche de prototypage. Par exemple, nous fournissons un sélecteur cyclique 3D (Figure 83) qui permet de relier interactivement des éléments en sélectionnant la représentation 3D normalisée du type de liaison choisie.

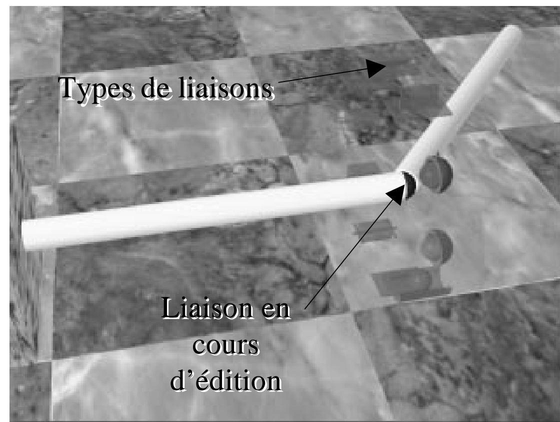


Figure 83 : Sélecteur de liaison

Nous avons aussi introduit des fonctionnalités plus avancées pour manipuler des objets complexes tels que des faisceaux électriques ou des tubes flexibles. Acheminer ce type d'objet dans une structure complexe est un des problèmes principaux qui doit être résolu avant d'utiliser réellement le prototypage virtuel dans un projet industriel. Dans notre système, ces objets sont modélisés comme des chaînes articulées définies par des tubes, des liaisons et des contraintes (tel la flexibilité).

Nous offrons aux utilisateurs deux types d'interactions :

1. La manipulation directe : l'utilisateur se sert d'un pointeur 3D (Figure 84) pour interagir avec un objet articulé, en saisissant une partie de l'objet.

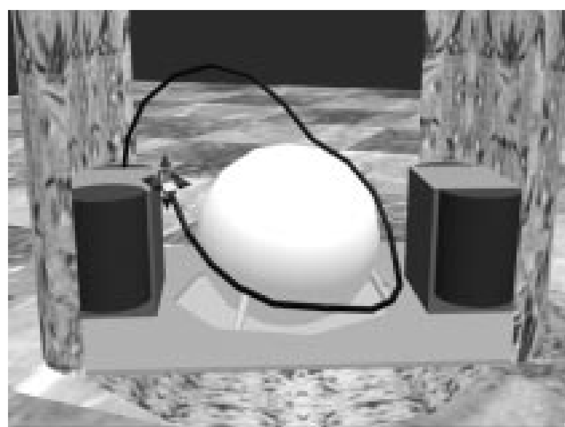


Figure 84 : Manipulation directe d'un faisceau semi-rigide (20 articulations) à l'intérieur de la plate-forme d'un satellite

2. La manipulation au niveau de la tâche : l'utilisateur choisit un type de faisceau, définit une source et une cible, les zones favorites, les zones interdites et lance la résolution de la tâche. Cette résolution, planifie un chemin, exempt de collision, qui respecte les contraintes choisies. La figure suivante (Figure 85 a) présente les différents paramètres précisés interactivement par l'utilisateur. La sphère rouge représente une zone qui doit être évitée en planifiant le chemin. La pince représente une zone par laquelle le faisceau doit passer. En fait, la pince présentée dans cette figure est une contrainte en position. De plus, nous fournissons aux utilisateurs un autre type de pince qui représente une contrainte sur la rotation et sur la position. Sur la figure suivante(Figure 85 b) on peut observer que, bien que la solution présentée ait impliquée la rotation de la pince, elle respecte la contrainte de position.

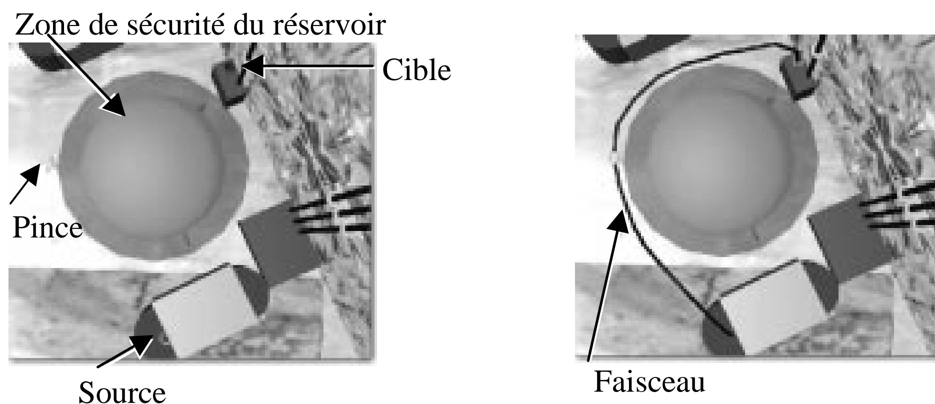


Figure 85 : (a) Définition des paramètres de la tâche, (b) La solution calculée

Plusieurs méthodes ont été proposées pour gérer des chaînes articulées. Ces méthodes, qui sont fondées soit sur la cinématique inverse soit sur la dynamique inverse, n'offrent jamais une façon intuitive pour définir les contraintes globales et internes. Notre équipe de recherche a proposé une application originale des algorithmes génétiques pour résoudre ce problème [Balet 97]. Ainsi nous pouvons gérer de façon homogène, les interactions avec des objets articulés contraints.

4.4.6. Conclusion sur l’application

Dans cette partie, nous avons présenté PROVIS, une solution logicielle qui a l’avantage de promouvoir le prototypage virtuel collaboratif dans les diverses phases de chaque projet, en

allégeant et en accélérant le travail des concepteurs. PROVIS sera employé pour la conception de la prochaine génération de satellites du CNES.

4.5. Conclusion

Nous avons présenté dans ce chapitre des évaluations préliminaires de VIPER. Nous avons, tout d'abord, comparé notre système à ceux qui ont été présentés dans le premier chapitre. Puis, nous avons précisé les originalités apportées par notre modèle de calcul réparti. Dans ce cadre la principale originalité par rapport aux autres systèmes de RVD est sans aucun doute l'existence de deux niveaux de programmation qui permettent d'une part de développer le contenu d'un environnement virtuel et d'autre part de choisir ou de définir sa répartition. D'autre part, notre modèle réparti est très générique et il peut ainsi s'adapter à de nombreuses architectures et systèmes de communication et être utilisé pour gérer une large gamme d'applications (de la simulation de véhicules militaires ou civils, qui nécessite essentiellement une forte extensibilité en nombre d'utilisateurs maximums, à la modélisation collecticielle qui demande un bon niveau de fiabilité). Enfin, notre système est extensible en définissant de nouvelles classes de stimuli, d'entités, d'univers virtuels distribués et d'espace de stimuli. Il est intéressant de noter dans ce cadre que les stimuli, les attributs et les comportements des entités peuvent être définis dans des modules externes qui sont soit interprétés soit liés dynamiquement.

Avant de présenter les premières applications développées avec VIPER ou avec son modèle de calcul, nous avons présenté quelques mesures préliminaires dans le but de juger de l'efficacité de VIPER. Nous avons, ensuite, présenté un éditeur collecticiel de formes qui permet de modeler des objets avec l'aide d'outils utilisant un modèle de déformation appelé les FFDs. Puis nous avons présenté une première application industrielle de notre modèle de calcul : le simulateur de gare GRS. Enfin, nous avons présenté la toute dernière application que nous avons développée et qui continue d'être étendue à l'heure actuelle : PROVIS.

Conclusion et Perspectives

A partir de l'étude de nombreux systèmes de réalité virtuelle distribuée, nous avons mis en évidence leurs caractéristiques ainsi que leur faiblesse commune actuelle : imposer toutes les solutions (à la gestion répartie de l'environnement) choisies pour tous les environnements virtuels. A notre avis, il n'est pas du tout évident que toutes ces solutions soient suffisamment générales. Par exemple, la gestion de zones d'intérêt dans un environnement très dynamique comportant peu de sites n'est pas intéressante et peut même entraîner une augmentation des transmissions sur le réseau si les entités changent très fréquemment de zones (à chaque fois, l'état instantané de la zone doit être transmis).

Cette constatation est la motivation principale qui nous a conduit à définir un nouveau système de réalité virtuelle distribuée que nous avons appelé VIPER. Notre système est très ouvert et il propose, entre autres, des techniques permettant de choisir parmi différents modèles de distribution ceux qui sont le plus adaptés à un domaine d'application. De plus, de façon à rester le plus générique possible vis à vis des applications développables avec VIPER, nous avons choisi le modèle d'architecture distribuée qui nous semble être le plus extensible : une architecture distribuée égal à égal utilisant la communication par groupe fiabilisée si nécessaire. Ainsi, notre système peut potentiellement gérer des applications très coûteuses en trafic réseau et nécessitant une interactivité maximale comme les simulateurs militaires distribués ainsi que des applications qui nécessitent un haut degré de fiabilité (souvent au détriment de l'interactivité) comme la téléopération médicale.

D'autre part, la structure permettant de définir les aspects comportementaux d'un environnement virtuel est inspirée des modèles définis pour la simulation comportementale. En effet, de nombreux comportements complexes et faisant preuve « d'intelligence » ont été mis en œuvre avec ces différents modèles, ce qui est un gage de généralité et de puissance pour notre système.

VIPER est fondé sur une architecture composée de quatre couches. La première couche, la plate-forme répartie, comprend des classes qui encapsulent la partie dépendante des matériels et des systèmes de communications utilisés de façon à assurer une portabilité aisée vers

d'autres environnements matériels et d'autres systèmes de communication. La seconde couche est un environnement de programmation parallèle à objets. Cet environnement permet la définition de classes d'objets parallèles qui sont des agrégats encapsulant la gestion de la répartition de leurs données sur un ensemble de sites appelé domaine.

Au-dessus de ces deux premières couches sont situées deux interfaces de programmation d'applications différentes. La première interface qui correspond à la plus haute couche de VIPER permet de définir des environnements virtuels distribués comme s'ils n'étaient pas répartis. En effet, cette couche cache le plus possible les aspects répartis. Le développeur peut définir les comportements d'entités virtuelles présentes dans l'environnement virtuel et des stimuli qui permettent l'interaction entre entités. C’est cette couche qui est inspirée des modèles utilisés en simulation comportementale. Cette couche introduit, de plus, un mécanisme intéressant qui permet de définir des entités et des stimuli dans des librairies qui sont liées dynamiquement à VIPER pendant son exécution. Notre système permet aussi la définition d’entités et de stimuli à l’aide d’un langage interprété : Object-Tcl.

La deuxième interface qui correspond à la troisième couche permet de définir le modèle de répartition d'environnements virtuels en choisissant parmi des modèles de répartition prédéfinis ou en en définissant de nouveaux. Nous avons défini pour cela des univers virtuels distribués qui permettent de gérer la répartition des entités sur la plate-forme et des espaces de stimuli distribués qui gèrent les interactions entre entités dans un contexte réparti.

Dans ce cadre, nous avons conçu et implémenté un mécanisme qui permet de limiter le trafic réseau en gérant des zones d’intérêt pour chaque utilisateur. Ce mécanisme est totalement géré par un espace de stimuli spécifique. Ainsi, il suffit d’indiquer que l’on veut l’utiliser et automatiquement, de façon transparente, le filtrage des messages réseau est réalisé. Néanmoins, ce filtrage est dédié aux environnements virtuels représentant des bâtiments architecturaux.

Actuellement, nous travaillons en collaboration avec le Laboratoire Central de Recherche de Thomson-CSF pour définir une structure spatiale 3D basée sur un *octree* pour limiter les échanges sur le réseau [Farcet 98]. L’intérêt de cette structure est qu’elle permet de gérer des environnements virtuels 3D quelconques (pas seulement des bâtiments ou des environnements essentiellement 2D comme ceux utilisés par les simulations militaires du NPSNET).

Pour évaluer notre système de réalité virtuelle distribuée nous avons réalisé deux petites applications (un prototype de simulateur de véhicules et une visite architecturale multi-utilisateurs) ainsi qu'un éditeur collecticiel de formes utilisant les FFDs comme modèle de déformation. Par la suite, le simulateur de gare GRS, nous a permis de confronter nos idées (surtout celles ayant trait à la définition de comportements) avec la réalité industrielle. D'autre part, PROVIS nous permet de valider les aspects répartis de VIPER dans le cadre d'une application industrielle de prototypage collaboratif. Des évaluations de performance, en cours de réalisation, sur cette dernière application vont nous permettre de mieux connaître les limites de notre système en terme, par exemple, de nombre maximal d'utilisateurs.

En perspective de ce travail de thèse, nous pensons qu'il est intéressant de continuer à créer de nouveaux univers virtuels distribués et espaces de stimuli de façon à constituer une bibliothèque de mécanismes de distribution parmi lesquels un développeur pourra choisir ceux qui semblent les plus appropriés à l'application qu'il souhaite construire.

Enfin, eu égard au grand nombre de systèmes de réalité virtuelle distribués existants, il semble très important de prévoir des possibilités d'interaction entre deux ou plusieurs systèmes. Ce domaine de recherche est appelé interopérabilité des systèmes de réalité virtuelle [Gong 96]. Cette problématique de recherche fait partie des perspectives de notre travail de recherche. En particulier, nous nous intéressons actuellement à l'architecture HLA (*High Level Architecture*) qui répond à une grande part de ces impératifs d'interaction entre systèmes de réalité virtuelle et qui semble donc être la voie à suivre la plus intéressante actuellement. C'est pourquoi nous pensons à court terme rendre VIPER compatible avec HLA de façon à ce qu'il soit possible d'utiliser notre système pour développer des fédérés HLA. Néanmoins HLA reste assez spécifique au domaine qu'elle vise (simulations militaires et civiles) et est trop statique (toutes les possibilités d'interopérabilité doivent être définies de façon très précise avant la réalisation des simulateurs fédérés). Il nous semble donc important de travailler à plus long terme et en collaboration avec d'autres architectes de systèmes de RVD sur l'interopérabilité dans un cadre plus générique et plus dynamique.

Références bibliographiques

[Airey 90] AIREY J., ROHLF J.H., BROOKS JR F.P., « *Towards Image Realism with Interactive Update Rates in Complex Virtual Buildings Environments* », Actes du Symposium on Interactive 3D Graphics, ACM SIGGRAPH, 1990, p. 41-50.

[ANSA 89] « *The Advanced Network Systems Architecture (ANSA) Reference Manual* », Castle Hill, Cambridge, Angleterre, Architecture Project Management.

[Appino 92] APPINO P., BRYAN LEWIS J., KOVED L., LING D., RABENHORST D., CODELLA C., « *An Architecture for Virtual Worlds* », Presence, Vol. 1, N° 1, Hiver 1992.

[Astheimer 93] ASTHEIMER P., FELGER W., MÜLLER S., « *Virtual Design: a Generic VR System for Industrial Applications* », Computer & Graphics, Vol. 17, N° 6, Novembre/Décembre 1993, p. 671-678.

[Balet 96] BALET O., TORGUET P., GAILDRAT V., CAUBET R., « *Autonomous Entities in Distributed Virtual Environments* », MMM'96, Actes de la Third International Conference on MultiMedia Modeling, Toulouse, Novembre 1996, publiés dans l'ouvrage : Multimedia Modeling'96, Courtiat et al. (Editeurs), World Scientific.

[Balet 97] BALET O., LUGA H., DUTHEN Y., CAUBET R., « *PROVIS: A platform for virtual prototyping and maintenance tests* », Actes de Computer Animation'97, Genève, Suisse, Juin 1997, p. 39-47.

[Barr 84] BARR H., « *Global and Local deformations of solid primitives* », Actes du SIGGRAPH'84, Vol 18, N° 3, ACM Computer Graphics, Juillet 1984.

[Barrus 95] BARRUS J. W., WATERS R. C., ANDERSON, D. B., « *Locales and Beacons : Efficient and Precise Support For Large Multi-User Virtual Environments* », Actes de IEEE Virtual Reality Annual International Symposium 1995, (VRAIS'95), IEEE Computer Society Press, p. 204-213.

[Bartle 90] BARTLE R., « *Interactive Multi-User Computer Games* », rapport technique non publié, disponible sur le site ftp : ftp.ccs.neu.edu dans le fichier : /pub/mud/docs/papers/mudreport.ps.gz

[Benford 93] BENFORD S., FAHLEN L., « *A Spatial Model of Interaction in Large Virtual Environments* », Actes de la Third European Conference on Computer Supported Cooperative Work (ECSCW'93), Milan, Italie, Septembre 1993.

[Benford 97] BENFORD S., GREENHALGH C.M., LLOYD D., « *Crowded Collaborative Virtual Environments* », Actes de ACM CHI'97, Atlanta, Géorgie, Mars 1997.

[Blanchard 90] BLANCHARD C., BURGESS S., HARVILL Y., LANIER J., LASKO A., OBERMAN M., TEITEL M., « *Reality Built for Two : A virtual reality tool* », Actes de ACM SIGGRAPH Symposium on Interactive 3D Graphics, 1990, p. 35-36.

[Broll 95] BROLL W., ENGLAND D., « *Bringing Worlds Together : Adding Multi-User Support to VRML* », Actes du VRML'95 Symposium, San Diego, Décembre 1995.

[Broll 97a] BROLL W., « *Populating the Internet : Supporting Multiple Users and Shared Applications with VRML* », Actes du VRML'97 Symposium, Monterey, Californie, Février 1997.

[Broll 97b] BROLL W., « *Distributed Virtual Reality for Everyone – a Framework for Networked VR on the Internet* », Actes de IEEE Virtual Reality Annual International Symposium 1997 (VRAIS'97), IEEE Computer Society Press, p. 121-128.

[Burdea 93] BURDEA G., COIFFET P., « *La Réalité Virtuelle* », HERMES.

[Butterworth 92] BUTTERWORTH J., DAVIDSON S., HENCH S. and OLANO T.M., « *3DM: A Three Dimensional Modeler Using a Head-Mounted Display* », Actes du 1992 Symposium on Interactive 3D Graphics, Cambridge, Massachusetts, March 29-April 1 1992, p. 135-138.

[Calvin 93] CALVIN J., DICKENS A., GAINES B., METZGER P., MILLER D., OWEN D., « *The SIMNET Virtual World Architecture* », Actes de VRAIS'93, p. 450-455.

[Calvin 96] CALVIN J.O., WEATHERLY R., « *An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)* », Actes du 14th Workshop on the Standards for the Interoperability of Distributed Simulation, Orlando, Floride, Mars 1996, p. 705-715.

[Calvin 97] CALVIN J.O., CHIANG C.J., MCGARRY S.M., RAK S.J., VAN HOOK D.J., SALISBURY M., « *Design, Implementation and Performance of the STOW RTI Prototype (RTI-s)* », Actes du Simulation Interoperability Workshop (SIW) Spring 97, Mars 1997.

[Capin 97] CAPIN T., NOSER H., THALMANN D., PANDZIC I., MAGNENAT THALMANN N., « *Virtual Human Representation and Communication in VLNet* », IEEE Computer Graphics and Applications, Numéro spécial 3D and Multimedia on the Information Superhighway, IEEE Computer Society Press, Mars-Avril 1997, p. 42-53.

[Carlsson 93] CARLSSON C., HAGSAND O., « *DIVE - a Platform for Multi-User Virtual Environments* », Computer & Graphics, Vol. 17, N° 6, Novembre-Decembre 1993, p. 663-669.

[Cassner 92] CASSNER S., DEERING S., « *First IETF Internet Audiocast* », ACM SIGCOMM Computer Communication Review, Juillet 1992, p. 92-97.

[Codella 92] CODELLA C., JALILI R., KOVED L., BRYAN LEWIS J., LING D., LIPSCOMB J., RABENHORST D., WANG C., NORTON A., SWEENEY P., TURK G., « *Interactive Simulation in a Multi-Person Virtual World* », Actes de CHI '92, p. 329-334, 1992.

[Coquillard 90] COQUILLARD S., « *Extended Free-Form Deformation : A Sculpturing Tool for 3D Geometric Modeling* », Actes du SIGGRAPH'90, Dallas, Texas, ACM, 6-10 Août 1990.

[CORBA 96] « *Corba 2.0 Specification* », Document technique de l'OMG, Mars 1996.

[Coulson 96] COULSON G., WADDINGTON D.G., « *A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks* », Actes de International Workshop TreDS'96 on Trends in Distributed Systems: CORBA and Beyond, Springer LNCS 1161, Aachen, Octobre1996, p. 14-29.

[CyberHub] « *CyberHub, Multi-User VR browser* », Black Sun Interactive, Documentation accessible via l'URL : http://www.blacksun.com.

[Delinguette 94] DELINGETTE H., SUBSOL G., COTIN S., PIGNON J. « *A Craniofacial Surgery Simulation Testbed* », Actes de Visualization for Biomedical Computing (VBC'94) Rochester, USA, Octobre 1994.

[DIVISION] « *dVISE Developper 3.0 reference manual* », Documentation technique de dVISE et de DVS, DIVISION.

[Farcet 98] FARCET N., TORGUET P., « *Space-scale Structure for Information Rejection in Large-scale Distributed Virtual Environments* », A paraître dans les actes du IEEE Virtual Reality Annual International Symposium 1998 (VRAIS'98), Mars 1998, Atlanta, Géorgie.

[Frécon 95] FRECON E., HAGSAND O., « *The DIVE Client Interface* », Documentation de référence de DIVE, SICS, Novembre 1995.

[Funkhouser 95] FUNKHOUSER T.A., « *RING : A Client-Server System for Multi-User Virtual Environments* », Actes du 1995 Symposium on Interactive 3D Graphics, Monterey, Californie, 1995, p. 85-92.

[Funkhouser 96] FUNKHOUSER T.A., « *Network Topologies for Scalable Multi-User Virtual Environments* », Actes du IEEE Virtual Reality Annual International Symposium 1996 (VRAIS'96), Mars 1996, p. 222-228.

[Gaildrat 93] GAILDRAT V., CAUBET R., RUBIO F., « *Declarative Scene Modelling with Dynamic Links and Decision Rules Distributed Among the Objects* », Actes de IFIP International Conference on Computer Graphics ICCG93 Bombay, Février 1993, p. 165-178.

[Geist 94] GEIST A., BEGUELIN A., DONGARRA J., JIANG W, MANCHECK R., SUNDERAM V., « *PVM : Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing* », MIT Press, Scientific and Engineering Computation, 1994.

[Gobetti 95] GOBBETI E., BALAGUER J.F., « *An Integrated Environment to Visually Construct 3D Animations* », Actes de SIGGRAPH 95, Août 1995, ACM SIGGRAPH, p. 395-398.

[Gông 96] LE VAN GONG H., « *Paradigmes pour l'interopérabilité entre environnements virtuels* », Thèse de Doctorat de l'Université Paris VI, Décembre 1996.

[Gossweiler 93] GOSSWEILER R., LONG C., KOGA S., PAUSCH R., « *DIVER : A Distributed Virtual Environment Research Platform »,* Actes du IEEE Symposium on Research Frontiers in Virtual Reality, San Jose, Californie, 25-26 Octobre 1993.

[Gossweiler 94] GOSSWEILER R., LAFERRIERE R.J., KELLER M.L., PAUSCH R., « *An Introductory Tutorial for Developing Multi-User Virtual Environments »,* Presence: Teleoperators and Virtual Environments, Vol. 3, Num. 4, p. 255-264, Décembre1994.

[Greenhalgh 94] GREENHALGH C., « *An experimental implementation of the spatial model* », Actes du 6th ERCIM Workshop, Stockholm, Juin, 1994, Swedish Institute of Computer Science, Stockholm, Suéde.

[Greenhalgh 95] GREENHALGH C., BENFORD S., « *MASSIVE : a Distributed Virtual Reality System Incorporating Spacial Trading*», Actes de 15th International Conference on Distributed Computing Systems (ICDCS'95), Vancouver, Canada, Mai-Juin 1995, p. 27-34.

[Greenhalgh 96] GREENHALGH C., « *Dynamic, embodied multicast groups in MASSIVE-2* », Rapport technique numéro : NOTTCS-TR-96-8 soumis à ICDCS'97 (International Conference on Distributed Computing Systems), Department of Computer Science, The University of Nottingham, UK, 1996.

[Greenhalgh 97] Greenhalgh C., Benford S., « *A Multicast Network Architecture for Large Scale Collaborative Virtual Environments* », Multimedia Applications, Services and Techniques – ECMAST '97, Lecture Notes in Computer Science, Vol. 1242, Springer-Verlag, 1997.

[Grimsdale 90] GRIMSDALE C., GHEE S., « *Virtual realities – artificial environments* », Actes de la conférence Computer Graphics, Londres, Novembre 1990, p. 1-8.

[Hagsand 96] HAGSAND O., « *Interactive Multiuser VEs in the DIVE System* », IEEE MultiMedia Magazine, Vol. 3, N° 1, Printemps 1996, p. 30-39.

[Kazman 93a] KAZMAN R., « *HIDRA : An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations* », International Journal in Computer Simulation, 1993.

[Kazman 93b] KAZMAN R., « *Load balancing and latency management in a distributed virtual world* », Actes de 3rd International Conference on Cyberspace, Mai 1993.

[Kazman 93c] KAZMAN R., « *Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments* », Actes de la First IEEE Conference on Virtual Reality Technology, 1993.

[Kazman 93d] KAZMAN R., « *Problems of Scale: Moving Beyond Toy Virtual Worlds* », Actes de UIST '93, 1993.

[Lea 97] LEA R., HONDA Y., MATSUDA K., MATSUDA S., « *Community Place: architecture and performance* », Actes du VRML'97 Symposium, Monterey, Californie, Février 1997.

[Liang 92] LIANG J., SHAW C., GREEN M., « *On Temporal-Spatial Realism in the Virtual Reality Environment* », IUST'91: Actes du ACM Symposium on User Interface Software and Technology, p. 321-328, 1992.

[Liang 93] LIANG J., GREEN M., « *Geometric Modeling Using Six Degrees of Freedom Input Devices* », Actes de 3rd International Conference on CAD and Computer Graphics Proceedings, Beijing, China, Août 1993, p. 217-222.

[Luebke 95] LUEBKE, D., GEORGES, C. « *Portals and Mirrors : Simple, Fast Evaluation of Potentially Visible Sets* », Actes du Symposium on Interactive 3D Graphics, ACM, Avril 1995, p. 105-106.

[Lutz 96] LUTZ R.R., « *HLA Object Model Template (OMT) Status* », Actes du 14th Workshop on the Standards for the Interoperability of Distributed Simulation, Orlando, Floride, Mars 1996.

[Macedonia 94] MACEDONIA M.R., ZYDA M.J., PRATT D.R., BARHAM P.T., ZESWITZ S. *« NPSNET: A Network Software Architecture For Large Scale Virtual Environments »*, Presence Vol. 3, N° 4, Automne 1994.

[Macedonia 95a] MACEDONIA M.R., ZYDA M.J., PRATT D.R., BARHAM P.T., *« Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments »*, Actes de VRAIS'95. IEEE Computer Society Press, Los Alamitos, CA, Mars 1995.

[Macedonia 95b] MACEDONIA M.R., *« A Network Software Architecture for Large Scale Virtual Environments »*, Thèse de Doctorat (PhD) de la Naval Postgraduate School, Monterey, CA, Juin1995.

[Miller 96] MILLER D.C., « T*he DOD High Level Architecture and the Next Generation of DIS* », Actes du 14th Workshop on the Standards for the Interoperability of Distributed Simulation, Orlando, Floride, Mars 1996.

[Moisan 93a] MOISAN B., DUTHEN Y., CAUBET R., « *Tools for SPMD object-oriented programming* », Actes de EUROMICRO'93, Barcelone, Espagne, Septembre 1993.

[Moisan 93b] MOISAN B., « *Un modèle de programmation parallèle à objets appliqué à la synthèse d'images* », Thèse de Doctorat de l'Université Paul Sabatier, Décembre 1993.

[Morningstar 90] MORNINGSTAR C., FARMER F.R., « *The Lessons of Lucasfilm's Habitat* », Cyberspace: First Steps, Michael Benedikt (editeur), 1990, MIT Press, Cambridge, Massachusetts.

[Mouli 93] MOULI R., DUTHEN Y., CAUBET R., « *In VitrAm (In Vitro Animats, a behavioural simulation model)* », Actes du 2nd IEEE International Workshop RO-MAN'93, Tokyo, Novembre 1993.

[Noser 96] NOSER H., PANDZIC I.S., CAPIN T.K., MAGNENAT THALMANN N., THALMANN D., « *Playing Games through the Virtual Life Network* », Actes de Artificial Life 96, Chiba, Japon, 1996, p. 114-121.

[ODP 95] « *ODP Trading Function* », Final Draft – ISO/IEC DIS 13235, Juin 1995.

[Osterhout 94] OSTERHOUT J.K., « *Tcl and the Tk toolkit* », Addison-Wesley, Reading, Massachusetts, 1994.

[Pandzic 97] PANDZIC I., CAPIN T., LEE E., MAGNENAT THALMANN N., THALMANN D., « *A flexible architecture for Virtual Humans in Networked Collaborative Virtual Environments* », Actes de Eurographics'97, Budapest, Hongrie, 1997.

[Plémenos 93] PLEMENOS D., « *De la modélisation classique à la modélisation déclarative* », Premières Journées Nationales AFIG-GROPLAN, Bordeaux, 1-3 Décembre 1993, p. 31-37.

[Pountain 91] POUNTAIN D., « *PROVISION : The packaging of Virtual Reality* », BYTE, Octobre 1991.

[Pulkka 95] PULKKA A.K., « *Spatial Culling of Interpersonal Communication within Large-Scale Multi-User Virtual Environments* », Thèse de Master of Science de l'Université de Washington, Seattle, Washington, 1995.

[Pratt 94] PRATT D.R., BARHAM P.T., LOCKE J., ZYDA M.J., « *Insertion of an Articulated Human into a Networked Virtual Environment* », Actes de 1994 AI, Simulation and Planning in High Autonomy Systems Conference, Gainesville, Floride, Décembre 1994.

[Quéau 93] QUEAU P., « *Televirtuality: the merging of telecommunications and virtual reality* », Computer & Graphics, Vol. 17, N° 6, Novembre/Décembre 1993, p. 691-693.

[Rainjonneau 92] RAINJONNEAU S. « *Un modèle orienté objet pour la simulation comportementale* », Thèse de Doctorat de l'Université Paul Sabatier, Décembre 1992.

[RMP 96] « *Programming Functional Specification for the Reliable Multicast Protocol Version 3.1* », Rapport technique de GlobalCast, Septembre 1996.

[Rohlf 94] ROHLF J., HELMAN J., « *IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics* », Actes du SIGGRAPH 94, Orlando, Floride, Juillet 1994, ACM SIGGRAPH, p. 381-394.

[Rubio 95] RUBIO F., TORGUET P., GAILDRAT V., CAUBET R., « *Déformations de formes libres à l'aide d'outils de réalité virtuelle* », Actes de l'Interface des Mondes Réels et Virtuels, Montpellier, Juin 1995, p. 139-149.

[Sachs 91] SACHS E., ROBERTS A. and STOOPS D., « *3-Draw: A Tool for Designing 3D Shapes* », IEEE Computer Graphics and Applications, 11, November 1991, p. 18-24.

[Sederberg 86] SEDERBERG T.W., PARRY S.R., « *Free-Form Deformation of Solid Geometric Models* », Actes du SIGGRAPH 86, Vol 20, N° 14, p. 151-160.

[Shaw 92] SHAW C., LIANG J., GREEN M., SUN Y., « *The Decoupled Simulation Model for Virtual Reality Systems* », Actes de ACM SIGCHI Human Factors in Computer Systems Conference, Mai 1992, p. 321-328.

[Singh 94] SINGH G., SERRA L., PNG W., NG H., « *BrickNet: A Software Toolkit for Network-Based Virtual Worlds* », Presence, Vol. 3, N° 1, Hiver 1994, p. 19-34.

[Singh 95] SINGH G., SERRA L., PNG W., WONG A., NG H., « *BrickNet: Sharing Object Behaviors on the Net* », Actes de VRAIS'95, Mars 1995, p. 19-25.

[Smith 96] SMITH W.G., KOIFMAN A., « *A Distributed Interactive Simulation Intranet Using RAMP, a Reliable Adaptative Multicast Protocol* », Actes du 14th Workshop on Standards for the Interoperability of Distrbuted Simulations, Orlando, Floride, Mars 1996.

[Snowdon 93] SNOWDON D.N., WEST A.J., HOWARD T.L.J., « *Towards the next generation of Human-Computer Interface* », Actes de l'Interface des Mondes Réels et Virtuels, Montpellier, Mars 1993, p. 399-408.

[Snowdon 94] SNOWDON D.N., WEST A.J., « *The AVIARY VR System: A Prototype Implementation* », Actes du 6th Workshop ERCIM, Stockholm, Suède, Juin 1994.

[Stevens 94] STEVENS W.R., « *TCP/IP Illustrated : the protocols* », Addison-Wesley Publishing Company, 1994.

[Strommer 93] STROMMER W.M., NEUGEBAUER J.G., FLAIG T., « *Transputer-based virtual reality workstation as implemented for the example of industrial robot control* », Actes de l'Interface des Mondes Réels et Virtuels, 22-26 mars 1993, p. 137-146.

[Sunderam 90] SUNDERAM V., « *PVM : A framework for Parallel Distributed Computing* », Concurrency: Practice & Experience, Vol 2, N° 4, Décembre 1990.

[Teller 92] TELLER S., « *Visibility Computation in Densely Occluded Polyhedral Environments* », Thèse de Ph.D., UC Berkeley CS Department, rapport technique numéro 92/708, 1992.

[Torguet 93] TORGUET P., « *Etude d'un modèle d'architecture pour la simulation d'environnements virtuels multimodaux* », Rapport de D.E.A., I.R.I.T., Juin 1993.

[Torguet 95a] TORGUET P., CAUBET R., « *VIPER (VIrtuality Programming EnviRonment): A virtual reality applications design platform* », Actes du 2nd Eurographics Workshop on Virtual Environments, Janvier 1995.

[Torguet 95b] TORGUET P., RUBIO F., CAUBET R., « *Atelier de sculpture virtuelle multi-utilisateurs* », Actes de Interaction Homme Machine 95 (IHM'95), Toulouse, Octobre 1995.

[Torguet 96] TORGUET P., RUBIO F., GAILDRAT V., CAUBET R., « *Multi-user Interactions in the context of concurrent virtual world modelling* », Actes du 3rd Eurographics Workshop on Virtual Environments, Monaco, Février 1996, publiés dans l'ouvrage : Virtual Environments and Scientific Visualization' 96, Göbel et al. (Editeurs), Springer Computer Science, p. 121-130.

[Torguet 97] TORGUET P., BALET O., CAUBET R., « *A Software Architecture for Collaborative Virtual Prototyping* », Actes de COMPUGRAPHICS'97, Vilamoura, Algarve, Portugal, Décembre 1997, p. 310-319.

[VRML1.0 96] « *The Virtual Reality Modeling Language Version 1.0C Spécification* », document technique accessible via l'URL : http://vag.vrml.org/vrml10c.html, Janvier 1996.

[VRML97 97] « *The Virtual Reality Modeling Language* », ISO/IEC DIS (Draft for International Standard) 14772-1, document technique accessible via l'URL : http://www.vrml.org/Specifications/VRML97/DIS/index.html, Avril 1997.

[West 93] WEST A. J., HOWARD T. L. J., HUBBOLD R. J., MURTA A. D., SNOWDON D. N., BUTLER D. A., « *AVIARY - A Generic Virtual Reality Interface for Real Applications* », Virtual Reality Systems, Earnshaw, Gigante et Jones (éditeurs), Academic Press, chapitre 15, p. 213-236, Mars 1993.

[Whetten 95] WHETTEN B., MONTGOMMERY T., KAPLAN S., « *A High Performance Totally Ordered Multicast Protocol* », Theory and Practice in Distributed Systems, Springer Verlag Lecture Notes in Computer Science 938, 1995.

[XDR 87] « *XDR: External Data Representation* », RFC1014, Juin 1987.