FRÉDÉRIC DROLET

COHÉRENCE ET SYNCHRONISATION DANS UN ENVIRONNEMENT VIRTUEL MULTI-SENSORIEL RÉPARTI

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en génie électrique
pour l'obtention du grade de maître en génie électrique (M. Sc.)

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
FACULTÉ DES SCIENCES ET GÉNIES
UNIVERSITÉ LAVAL
QUÉBEC

2008

© Frédéric Drolet, 2008

Résumé

La réalité virtuelle est une technologie qui permet à un utilisateur d'interagir avec une scène générée par ordinateur. L'environnement virtuel dans lequel évolue le participant doit lui fournir des informations sensorielles pour qu'il puisse effectuer des tâches de façon naturelle. Plusieurs outils logiciels et matériels sont actuellement disponibles sur le marché pour tenir compte des différents aspects d'un environnement virtuel: visualisation immersive, simulation des lois de la physique, émissions sonores, etc. Cependant, aucune application n'a été développée pour intégrer ces différentes technologies dans un seul et même système. Ce projet présente une approche modulaire et flexible qui utilise les principes du *multi-threading* afin de synchroniser les données communes de position, d'orientation et de facteurs d'échelle entre plusieurs sous-systèmes intégrés dans un environnement virtuel multi-sensoriel et distribué. L'application finale comporte cinq sous-systèmes qui gèrent tous un aspect de l'environnement virtuel : la synchronisation, la visualisation, la gestion des lois physiques, la distribution sur un réseau et l'intégration des sous-systèmes.

Tout d'abord, un module de synchronisation forme le cœur du système en protégeant l'accès asynchrone aux données partagées entre les modules. Un processus de synchronisation en trois étapes permet alors de maintenir la cohérence entre les différents modules. Ensuite, un module de visualisation permet de produire un rendu graphique en trois dimensions de l'environnement virtuel en affichant les géométries et les textures des différents éléments virtuels composant la scène. De plus, un module haptique est utilisé pour appliquer les lois physiques sur les objets. Puis, afin de distribuer les modules sur plusieurs machines d'un réseau ou rendre l'application multi-utilisateurs, un module de distribution sera également ajouté au système final. Enfin, un module d'intégration permet d'interconnecter les modules entre eux en encapsulant les fonctionnalités du module de synchronisation. Ce dernier module permettra de gérer la transmission d'information entre les modules et d'ajouter des comportements de haut niveau au système final comme la navigation, la manipulation d'objets, des interfaces graphiques, etc.

Abstract

Virtual reality is a technology that enables a user to interact with a computer-generated scene. The virtual environment in which the participant evolves must provide sensory information to enable him to perform tasks in a natural way. Several software and hardware tools are currently available on the market to take care of the different aspects of a virtual environment: immersive visualization, simulation of physical laws, audio effects, and so on. However, no application has been developed to integrate these technologies into a single system. Therefore this project presents a flexible and modular approach using the principles of multithreading to synchronize common data such as position, orientation and scale factors among several subsystems into a distributed multi-sensory virtual environment. The final application consists of five subsystems managing every aspect of the virtual environment: synchronization, visualization, applications of physical laws, distribution over a network and integration of subsystems.

First of all, a synchronization module forms the core of the system by protecting asynchronous access to shared data between modules. It defines a three-stage synchronization process that maintains consistency between the different modules. Then, a visualization module produces a three-dimensional graphic rendering of the virtual environment by displaying geometries and textures of the various elements composing the scene. In addition, a haptic module is used to apply physical laws on objects. Then, in order to distribute the modules on several machines on a network or to make the application multi-users, a distribution module will also be added to the final system. Finally, an integration module is used to interconnect modules with each others by encapsulating functionalities from the synchronization module. This module will manage information transmission between modules and add high-level behaviours to the final system like navigation, object manipulation, graphical interfaces, and so on.

À mon amour Christine, ma famille, mes amis et mes collègues.
Merci pour votre soutien constant.

Table des matières

Résumé i
Abstract ii
Table des matières iv
Liste des figures vi
Chapitre 1 - Introduction 9
1.1 Contexte du projet 10
1.2 Théorie et vocabulaire 12
1.2.1 Définitions et mesures de performance 13
1.2.2 Représentation des objets dans l'environnement virtuel 16
1.2.3 Évaluation de la profondeur 19
1.2.4 Technologies de rendu graphique 26
1.2.5 Technologies de positionnement 35
1.2.6 Métaphores de manipulation 40
1.2.7 Métaphores de navigation 49
1.3 Description du projet 61
Chapitre 2 - Synchronisation des modules 68
2.1 Définition d'un module 68
2.2 Principes du *multi-threading* 71
2.2.1 Exécution parallèle avec les *threads* 71
2.2.2 Protection par les *mutex* pour l'accès asynchrone aux données partagées 74
2.3 Module de synchronisation 76
2.3.1 Structure générale 77
2.3.2 Format de données neutre 78
2.3.3 Processus de synchronisation des données 80
2.3.4 Priorité des modules 84
2.3.5 Verrouillage des objets 87
2.3.6 Gestion temporelle 90
2.3.7 Émission d'événements 91
2.3.8 Ajout flexible de modules 93
Chapitre 3 - Visualisation d'une scène virtuelle 94
3.1 Principe de l'arbre de rendu dans OpenSceneGraph 95
3.2 Principe du rendu stéréoscopique *off-axis* dans VR Juggler 99
3.3 Encapsulation des propriétés dans la classe VisualObject 108
3.4 Protection à l'accès asynchrone 114
3.5 Intégration au module de synchronisation 117
Chapitre 4 - Distribution sur un réseau 120
4.1 Communication réseau hybride 122
4.1.1 Mises à jour et synchronisation rapides par le protocole UDP 122
4.1.2 Cohérence assurée par le protocole TCP 123
4.2 Organisation réseau hybride 124
4.2.1 Synchronisation des clients 126
4.2.2 Enregistrement des clients 129
4.3 Création et classement des objets 130
4.4 Fonctionnalités et propriétés de haut niveau 133

Chapitre 5 - Intégration des modules et interface utilisateur 136
5.1 Définition de l'univers virtuel 138
5.1.1 Définition des objets virtuels 142
5.1.2 Définition du monde virtuel 154
5.1.3 Définition des *proxies* virtuels 155
5.1.4 Définition de l'avatar 156
5.1.5 Topologies de l'univers virtuel 162
5.2 Interface utilisateur 164
5.2.1 Interface utilisateur de configuration 166
5.2.2 Interface utilisateur graphique 3D 167
5.2.3 Interface utilisateur graphique distribuée 171
5.2.4 Traqueurs et autres périphériques 172
5.3 Simulation du système de locomotion 174
Chapitre 6 - Évaluations et expérimentations 178
6.1 Latence de synchronisation 178
6.2 Évaluation de la performance 181
6.2.1 Taux de rafraîchissement par rapport au nombre d'objets 182
6.2.2 Taux de rafraîchissement par rapport au nombre de modules 183
6.2.3 Taux de rafraîchissement par rapport au nombre d'instances réseau 185
6.3 Évaluation empirique 186
6.3.1 Évaluation du réalisme de l'environnement 187
6.3.2 Évaluation de l'aspect intuitif de l'interface 188
Conclusion 190
Bibliographie 193

Liste des figures

Figure 1-1: Modèle de la plate-forme de locomotion NELI 11
Figure 1-2: Traqueur de main d'InterSense, gant CyberGlove et mécanisme CyberGrasp utilisés au LIV 12
Figure 1-3: Quelques configurations des écrans déployables du système FLEX 13
Figure 1-4: Matrices de rotation indépendante autour des axes X, Y et Z 19
Figure 1-5: Rotations résultantes différentes avec les conventions d'Euler XYZ et ZYX... 20
Figure 1-6: Principes d'accommodation et de convergence 21
Figure 1-7: Principe de l'affichage stéréoscopique 22
Figure 1-8: Deux contraintes du rendu on-axis 24
Figure 1-9: Contradiction entre la convergence et l'accommodation des yeux dans un environnement virtuel 25
Figure 1-10: Système FLEX utilisé au LIV fabriqué par Fakespace Systems Inc. 28
Figure 1-11: Workbench modèle M1 Desk de Fakespace Systems Inc. 29
Figure 1-12: Affichage hémisphérique VisionStation de Elumens Corporation 30
Figure 1-13: Head-Mounted Display ProView SR80 de la compagnie Rockwell Collins... 31
Figure 1-14: Arm-Mounted Display nommé "Binocular Omni-Orientation Monitor" (BOOM) de Fakespace Systems Inc. 32
Figure 1-15: Affichage auto-stéréoscopique lenticulaire au plasma de 50 po de la compagnie NewSight 33
Figure 1-16: Affichage auto-stéréoscopique volumétrique Perspecta de la compagnie Actuality Systems 34
Figure 1-17: Affichage auto-stéréoscopique holographique HelioDisplay M2 de la compagnie IO2 35
Figure 1-18: Traqueur Fastrak de la compagnie Polhemus 36
Figure 1-19: Traqueur et module sans-fil du système IS-900 39
Figure 1-20: Traqueur de pupilles EyeLink II de SR Research, Ltd. 40
Figure 1-21: Approche des plaques pour la technique de l'aperture 44
Figure 1-22: Technique Go-Go 46
Figure 1-23 - Variables utilisées pour la définition des techniques de steering 52
Figure 1-24: Wand du système IS-900 d'InterSense Inc 55
Figure 1-25: Exemple de fonction gaussienne pour une rotation virtuelle non-isomorphique 60
Figure 1-26: Structure générale de VirtualUniverse 64
Figure 2-1: Exécution des threads selon la méthode round-robin avec calculs CPU rapides 72
Figure 2-2: Exécution des threads avec accès au disque ou à la mémoire 73
Figure 2-3: Accès asynchrone à des données partagées sans mutex et avec mutex 75
Figure 2-4: Structure générale du module de synchronisation 79
Figure 2-5: Processus de synchronisation simple 81
Figure 2-6: Processus de synchronisation optimisé 83
Figure 2-7: Incohérence lecture/écriture lors d'une mise à jour critique à la cohérence de la scène sans gestion de la priorité des modules 85
Figure 2-8: Incohérence corrigée grâce aux priorités 87

Figure 2-9: Artefact visuel dû à la vitesse relative des modules 88
Figure 2-10: Artefact visuel dû à la vitesse relative des modules corrigé 90
Figure 2-11: Processus de synchronisation optimisé avec émission et traitement des événements 92
Figure 3-1: Structure générale d'un arbre de rendu 96
Figure 3-2: Principales classes de nœuds de l'arbre de rendu d'OpenSceneGraph 97
Figure 3-3: Projection de la scène virtuelle sur l'écran selon la position des yeux de l'utilisateur 99
Figure 3-4: Référentiels utilisés pour définir les paramètres extrinsèques 102
Figure 3-5: Référentiels utilisés pour définir les paramètres intrinsèques 104
Figure 3-6: Plans formant le champ de vision de l'utilisateur imposé par la surface de projection 107
Figure 3-7: Deux niveaux d'abstraction de VisualObject 109
Figure 3-8: Représentation UML de l'objet visuel 110
Figure 3-9: Suite des opérations pour l'ajout d'un fils à un VisualObject. 112
Figure 3-10: Principe de conservation des modèles 3D dans une mémoire cache 113
Figure 3-11: Mécanisme d'émission et d'exécution des tâches 116
Figure 3-12: Adaptation du processus de synchronisation au module visuel 118
Figure 4-1: Représentation générale du module de distribution 121
Figure 4-2: Modèle de connexions de la structure réseau hybride 125
Figure 4-3: Algorithme de Cristian 126
Figure 4-4: Algorithme de Cristian modifié 128
Figure 4-5: Propagation de l'adresse UDP pour mises à jour en broadcast 130
Figure 4-6: Transmissions de messages TCP via serveur 131
Figure 4-7: Création et classement des objets 132
Figure 4-8: Accès à module spécialisé quelconque par un proxy 135
Figure 5-1: Structure générale de VirtualUniverse avec comportement de haut niveau 137
Figure 5-2: Représentation de l'univers virtuel 139
Figure 5-3: Représentation UML de l'univers virtuel 141
Figure 5-4: Représentation de l'objet virtuel 142
Figure 5-5: Échelle du niveau de virtualité 143
Figure 5-6: Incohérence lors de l'utilisation du référentiel absolu avec une structure en arbre 147
Figure 5-7: Conservation de la cohérence grâce au référentiel relatif avec une structure en arbre 148
Figure 5-8: Représentation UML de l'objet virtuel 151
Figure 5-9: Proxies virtuels 156
Figure 5-10: Représentation de la main virtuelle 159
Figure 5-11: Représentation du pied virtuel 161
Figure 5-12: Topologie mono-utilisateur de base de l'univers virtuel 163
Figure 5-13: Topologie mono-utilisateur distribuée 164
Figure 5-14: Topologie multi-utilisateurs avec serveur haptique 165
Figure 5-15: Window3D et sa représentation UML 167
Figure 5-16: TextBox3D et sa représentation UML 169
Figure 5-17: Menu3D comprenant plusieurs objets Button3D et sa représentation UML. 170
Figure 5-18: Topologie de l'interface utilisateur graphique distribuée 172

Figure 5-19: Métaphore de navigation ski-pole pour la simulation du système de locomotion NELI 176
Figure 6-1: Taux de rafraîchissement par rapport au nombre d'objets virtuels 183
Figure 6-2: Taux de rafraîchissement par rapport au nombre de modules 184
Figure 6-3: Taux de rafraîchissement par rapport au nombre d'instances réseau 186

Chapitre 1 - Introduction

Depuis les dernières années, la réalité virtuelle (RV) a pris un essor important dans plusieurs domaines de recherche. Simulant un environnement virtuel (EV) immersif où un utilisateur peut interagir de façon plus naturelle avec la machine, cette nouvelle technologie semble une voie prometteuse pour l'apprentissage et l'entraînement. En médecine, une opération délicate pourra être répétée autant de fois qu'il le faudra sans faire appel à des cobayes. Dans le domaine militaire, des simulations de combat permettront de former les soldats sur un champ de bataille virtuel et de leur apprendre à formuler des tactiques et à prendre des décisions. L'industrie du jeu vidéo commence également à faire son entrée dans l'ère du virtuel en révolutionnant la façon dont le joueur participe à l'expérience[1].

Une multitude de technologies existent actuellement sur le marché afin de rendre les simulations virtuelles plus réalistes. Pour cela, plusieurs aptitudes sensorielles du participant doivent être stimulées simultanément. Par exemple, certains systèmes de rendu graphique possèdent plusieurs écrans permettant d'élargir le champ de vision de l'utilisateur. D'autres appareils sont installés directement sur la tête de l'usager, lui permettant d'observer le monde dans toutes les directions. De plus, différentes technologies d'affichage ont été développées pour simuler la profondeur dans une scène en trois dimensions. Afin de permettre une interaction naturelle avec l'environnement et les objets virtuels, des gants munis de différentes technologies de capteurs peuvent être utilisés pour numériser les mouvements des mains de l'utilisateur. D'autres appareils simulent les retours de force lorsqu'il y a collision avec un objet. Finalement, toute une gamme de systèmes de positionnement permet de traquer l'emplacement et l'orientation d'un objet dans l'espace avec une précision plus ou moins grande selon le prix. Grâce à tous ces éléments, l'utilisateur peut observer et manipuler naturellement les objets virtuels qui forment la scène.

Cependant, la principale limitation de la RV est l'environnement physique dans lequel évolue le participant. En effet, comme le système de rendu graphique est parfois imposant,

---

[1] Wii, la nouvelle console de Nintendo, propose un mécanisme de contrôle révolutionnaire pour le marché du jeu vidéo où le joueur doit effectuer un déplacement physique de la manette dans l'espace afin d'interagir avec la partie.

la zone d'interaction où l'utilisateur peut se déplacer demeure statique. Dans le cas d'un casque masquant complètement la vue, l'usager fait plutôt face à des problèmes de longueur de câbles et de sécurité car les obstacles du monde réel ne sont alors plus visibles. De plus, la qualité visuelle est souvent réduite à cause de la petite taille des écrans. Plusieurs techniques de navigation ont été développées afin de remédier à ce problème. L'approche virtuelle propose des méthodes de déplacement où l'utilisateur contrôle sa position de façon détournée avec le clavier, une manette, des symboles ou d'autres métaphores. L'approche physique comprend plutôt des méthodes où il devra lui-même exécuter le déplacement dans l'espace. Certains casques projettent les éléments virtuels sur une surface semi-transparente. La scène réelle demeure donc toujours visible et l'utilisateur peut alors se déplacer de façon sécuritaire dans l'espace à la fois réel et virtuel qui l'entoure. Cependant, ces systèmes encombrants et énergivores font généralement face à un problème de portabilité. D'autres technologies ont plutôt été mises sur pied pour simuler la marche. Cependant, la plupart d'entre elles ne permettent pas d'atteindre les objectifs de réalisme désirés. En effet, lourds et bruyants, ces systèmes de locomotion forcent généralement le participant à s'y adapter plutôt que de lui offrir une méthode de navigation naturelle.

Dans cette optique, le département de génie électrique et de génie informatique en collaboration avec le département de génie mécanique ont lancé le projet « Network Enabled Locomotion Interface » (NELI) afin de développer une interface de marche plus performante basée sur un mécanisme à câbles. L'interface comprend trois sous-systèmes distincts: le mécanisme à câbles, le contrôleur et l'environnement virtuel. L'aboutissement du projet dépendra donc du travail de recherche collaboratif de plusieurs étudiants et professeurs dans ces trois domaines.

1.1 Contexte du projet

Le système de locomotion proposé par le projet NELI sera constitué d'un mécanisme à câbles contrôlant la position et l'orientation de deux plates-formes sur lesquelles l'utilisateur prendra place. De plus, un système de harnais permettra de ramener le corps de celui-ci au centre de l'appareil afin de compenser pour le déplacement engendré par la

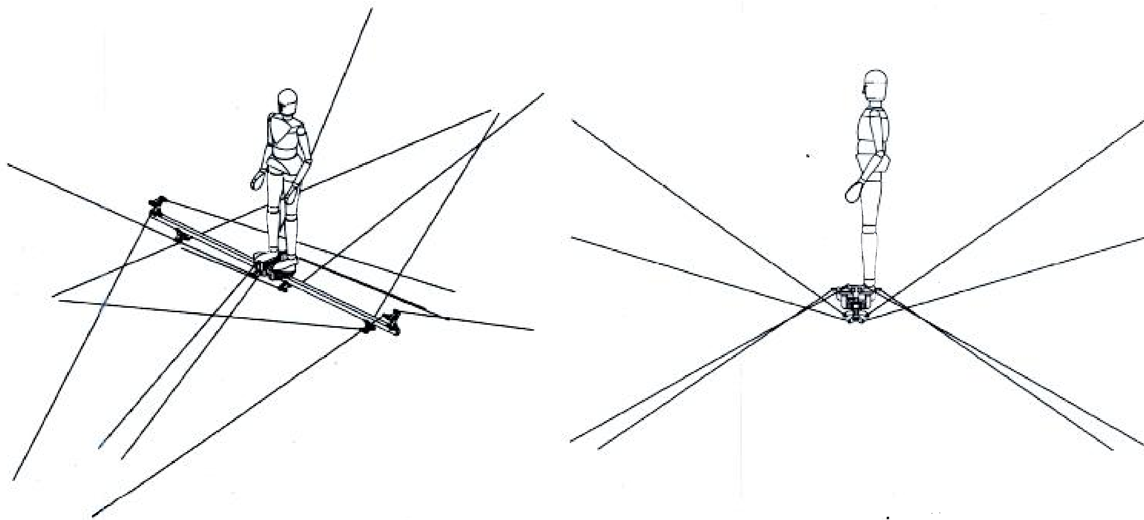


Figure 1-1: Modèle de la plate-forme de locomotion NELI

marche tout en soutenant une partie de son poids. La faible inertie des câbles et des plates-formes permettra une réaction très rapide aux déplacements des pieds du participant afin de simuler le mouvement naturel de la marche. Le contrôleur permettra ensuite d'imposer une position et une orientation aux plates-formes selon les forces appliquées sur chaque pied lors du déplacement de l'utilisateur sur le sol du monde virtuel. Les principes de cette interface de locomotion sont illustrés dans la Figure 1-1.

Concurremment au démarrage de ce projet de recherche, le centre de Recherche et Développement pour la Défense Canada à Valcartier (RDDC - Valcartier) mettait sur pied le Laboratoire d'immersion virtuelle (LIV) dont le but est d'étudier le potentiel des différentes technologies de RV dans un contexte militaire. C'est donc en collaboration avec ce centre que l'environnement virtuel présenté dans cet ouvrage a été développé. En effet, le LIV s'est muni de plusieurs outils de haute technologie permettant la détection de l'utilisateur dans l'espace et la génération d'un rendu graphique réaliste. Le système de positionnement IS-900 de la compagnie InterSense Inc. détermine la position et l'orientation d'un objet dans l'espace grâce à une technologie inertielle et ultrasonique. De plus, deux gants de réalité virtuelle CyberGlove® d'Immersion Corporation permettent de connaître la configuration de chaque main à l'aide de capteurs très précis qui mesurent la flexion et l'abduction des différentes phalanges, de la paume ainsi que du poignet. Ensuite, un exosquelette CyberGrasp™, également de l'entreprise Immersion Corporation, est un autre mécanisme à câbles qui simule l'application de forces sur le bout des doigts à l'aide

Figure 1-2: Traqueur de main d'InterSense, gant CyberGlove et mécanisme CyberGrasp utilisés au LIV

d'anneaux reliés par un câble à des actionneurs. Présentés dans la Figure 1-2, ces trois périphériques combinés permettent de simuler la plate-forme NELI. En effet, la position du pied est déterminée à l'aide du système IS-900 et du gant CyberGlove®. Puis, le retour de force induit par la collision avec le sol peut être simulé à l'aide du mécanisme CyberGrasp™.

Pour le rendu, le LIV utilise un système FLEX™ de FakeSpace Systems Inc. qui est muni de quatre écrans (au centre, à gauche, à droite et au sol) dont les dimensions sont de 3,25 m (10 pi 8 po) en largeur et de 2,44 m (8 pi) en hauteur. Les deux écrans latéraux sont déployables permettant ainsi une disposition flexible des surfaces de projection (fermée, ouverte, en L, à 45° [mode théâtre], etc.) comme le démontre la Figure 1-3. Une projection arrière sur tous les écrans à l'exception de celui du sol permet d'éviter l'apparition de l'ombre des participants dans la scène virtuelle. Tout ce matériel est mis à la disposition du projet NELI. En échange, l'application finale devra être exploitable dans le contexte militaire du LIV.

Avant de définir le projet, la prochaine section présente la théorie et le vocabulaire sur lesquels reposent les principes généraux de la RV. Les termes techniques et les définitions propres à ce domaine seront présentés afin de fournir au lecteur les bases théoriques utilisées tout au long de ce document.

1.2 Théorie et vocabulaire

Avant de commencer la description du travail de recherche, il est important de bien comprendre les notions de base de la RV qui ont guidé la réalisation de ce projet. Les aspects théoriques et mathématiques du rendu graphique et de la représentation des objets

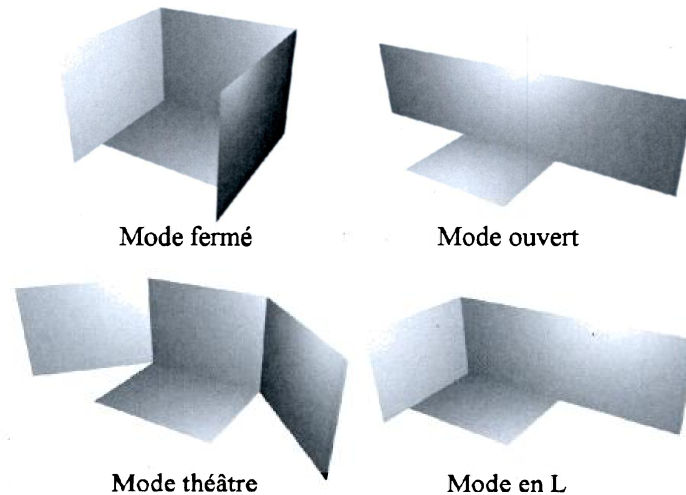


Figure 1-3: Quelques configurations des écrans déployables du système FLEX

virtuels seront présentés. Puis, un bref état de l'art permettra de comparer les technologies actuellement utilisées pour remplir les différentes fonctionnalités présentées plus tôt.

1.2.1 Définitions et mesures de performance

Il existe plusieurs termes spécifiques à la RV qui doivent être énumérés avant de poursuivre la présentation du projet. En effet, ce vocabulaire permettra de définir les bases théoriques du domaine et les mesures de performance servant à l'évaluation de l'application finale.

Tout d'abord, la **virtualité** en soi semble le terme le plus important à définir en premier. La virtualité est l'état d'un objet qui se trouve à mi-chemin entre le réel et l'imaginaire. Il possède donc des propriétés du monde réel qui sont perceptibles comme une forme, un son, un poids, etc. Cependant, cet objet n'existe pas en réalité, il est plutôt virtuel. La **réalité virtuelle** implique donc la simulation d'une scène quelconque dans laquelle un utilisateur peut percevoir et interagir en temps réel avec des objets virtuels à l'aide des sens du toucher, de la vue et de l'ouïe.

Puisque la RV est un domaine relativement nouveau, plusieurs évaluations demeurent parfois subjectives. En effet, la qualité principale que doit avoir un environnement virtuel est d'être immersif. Cependant, ce critère n'est pas facile à mesurer. L'**immersion** signifie que l'utilisateur a l'impression de réellement faire partie du monde virtuel. La scène doit être réaliste et fluide. De plus, l'usager doit pouvoir interagir avec des composantes le plus

naturellement possible. Le participant ne doit pas non plus être gêné par son entourage physique lors des manipulations. La présence de câbles et le poids des appareils nuisent donc à l'effet d'immersion.

Puisque l'immersion est difficile à évaluer, d'autres mesures de performance plus objectives peuvent être utilisées pour y arriver. Premièrement, le **réalisme** de la scène détermine à quel point l'environnement virtuel semble « vrai ». C'est le critère qui contribue le plus à l'immersion. Cependant, il peut être défini par plusieurs facteurs : la résolution des textures et de l'image, la largeur du champ de vision, le nombre d'objets dans la scène et leur complexité, les effets spéciaux (flammes, fumée, etc.) et les filtres (anticrénelage[1], anisotrope[2]) appliqués sur les images, etc. Il faudra donc évaluer le réalisme de l'application finale sur plusieurs aspects.

Ensuite, un environnement virtuel immersif doit être **intuitif**. L'utilisateur doit donc pouvoir y manipuler les objets et s'y déplacer de façon naturelle. De plus, l'interface lui permettant d'interagir avec l'application devra également être facile d'utilisation. Ainsi, la transition entre le monde réel et l'environnement virtuel pourra se faire plus facilement. Bien que l'aspect intuitif d'une application soit plutôt subjectif, on peut tout de même évaluer ce critère en mesurant la vitesse d'exécution de certaines tâches spécifiques comme un déplacement ou la manipulation d'objets dans l'environnement.

Finalement, deux autres mesures de performance objectives sont importantes à considérer pour évaluer l'immersion: le taux de rafraîchissement et la latence. Le **taux de rafraîchissement** de l'application détermine le nombre de fois que la scène visuelle est mise à jour par seconde avec la métrique fps (images par seconde ou *frame per second* en anglais). Plus le taux de rafraîchissement sera élevé, plus l'affichage de la scène et l'ajustement du point de vue seront fluides. Évidemment, comme le monde réel fournit

[1] Filtre anticrénelage (*antialiasing* en anglais) : Type de filtrage qui permet d'éliminer les artefacts visuels en forme d'escalier introduits par les hautes fréquences dans une image. Les objets éloignés couvrant moins de pixels sur l'écran (fréquence d'échantillonnage moins élevée) et les textures de hautes résolutions (hautes fréquences) appliquées sur ceux-ci sont donc plus difficiles à afficher correctement. Ce type de filtrage permet d'atténuer le problème.

[2] Filtre anisotrope (*anisotropic* en anglais) : Lorsqu'une texture est observée de loin avec un grand angle, elle apparaît sous forme de trapèze écrasé au lieu d'un rectangle et semble floue. Ce problème augmente davantage avec la distance. Ce type de filtrage permet d'améliorer la représentation des textures observées dans de telles conditions.

toujours une image sans discontinuité, l'effet d'immersion sera donc atteint en augmentant le plus possible cette valeur. À l'inverse, une mise à jour saccadée forme une énorme source de distraction et d'inconfort et rend donc l'expérience peu satisfaisante. Par contre, le taux de rafraîchissement est affecté par le niveau de réalisme appliqué à l'environnement. En effet, il diminuera en fonction de la complexité de la scène et des textures. Il faut donc trouver un compromis entre réalisme et taux de rafraîchissement. Ce compromis dépend des performances de la machine utilisée pour l'affichage. Il ne faut pas confondre le taux de rafraîchissement de l'application avec celui de l'écran qui détermine plutôt le nombre de fois par seconde que la carte graphique présente le contenu de sa mémoire à l'écran en Hertz (Hz). En effet, lorsque la scène est mise à jour, l'image résultante est écrite dans un tampon qui sera ensuite lu pour l'affichage par la carte vidéo. Ces opérations de lecture et d'écriture n'ont pas à être à la même fréquence. La fréquence de rafraîchissement de l'écran est un élément important pour l'affichage stéréoscopique qui sera présenté dans les prochaines sections.

Enfin, la **latence** détermine le délai entre un événement quelconque et son effet dans la scène virtuelle. Lorsque l'utilisateur effectue une commande ou une action, le résultat obtenu confirmant l'exécution de la tâche s'appelle une rétroaction. La latence s'applique à plusieurs niveaux. Tout d'abord, la *latence visuelle* concerne surtout le délai entre le déplacement de la tête et la mise à jour du point de vue à l'écran. Pour améliorer l'effet d'immersion, il faudra une latence courte et constante. Une longue latence sera tolérable, mais une latence variable sera très inconfortable. Encore une fois, ce délai varie en fonction du niveau de réalisme de la scène. La *latence haptique* définit le temps d'attente pour avoir un retour de force lors de la manipulation d'objets. Bien que le système tactile humain soit très sensible (il détecte des vibrations à plus de 1 KHz), la constance de ce type de latence est moins critique pour l'effet d'immersion car il demeure plus adaptatif que le système visuel. Finalement, les *latences de synchronisation* concernent plutôt les délais réseau et logiciel qui sont inévitables dans le développement d'une application modulaire et distribuée. Il faut évidemment tenter de diminuer le plus possible ces valeurs, mais il est impossible de les éliminer complètement.

L'ensemble de ces mesures de performance sera évalué plus en détails au chapitre 6 pour l'application finale. Comme mentionné dans la description du projet, l'application doit être modulaire et distribuée. Pour l'environnement virtuel développé, un **module** désigne un sous-système qui définit et contrôle une propriété sensorielle spécifique d'un objet virtuel (visuelle, haptique, etc.). Il doit également partager des paramètres communs avec les autres modules comme la position, l'orientation et des facteurs d'échelle. Chaque module doit posséder sa propre boucle d'exécution indépendante qui met à jour l'information sensorielle et qui synchronise les propriétés communes avec les autres modules. L'approche du *multi-threading* permet d'intégrer un tel comportement aux modules. Ainsi, la boucle de mise à jour des objets sera exécutée en parallèle par un *thread* contenu dans chacun des modules. Puisque le traitement de chaque module est indépendant des autres, il faudra par contre développer un processus pour partager l'information qui leur est commune de façon sécuritaire et cohérente. Les méthodes de synchronisation et de *multi-threading* seront présentées au chapitre 2.

1.2.2 Représentation des objets dans l'environnement virtuel

Un objet peut être positionné dans l'espace à l'aide d'une position et d'une orientation. La position est définie par un vecteur à trois dimensions $p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T$. L'orientation peut être représentée à l'aide d'une matrice de rotation 3x3 R ou d'un quaternion Q :

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad Q = [q_0 \ q_1 \ q_2 \ q_3]^T$$

Le quaternion est plus compact que la matrice car il possède quatre paramètres au lieu de neuf. Reposant sur la théorie des nombres complexes, on peut également définir un quaternion sous la forme suivante pour pouvoir l'interpréter de façon vectorielle à l'aide d'un scalaire et d'un vecteur:

$$Q = (s, \vec{v}) = \left(\cos\left(\frac{\varphi}{2}\right), \vec{\boldsymbol{u}} \sin\left(\frac{\varphi}{2}\right)\right)$$

Le paramètre φ représente alors l'angle de rotation autour d'un axe défini par le vecteur unitaire $\vec{\boldsymbol{u}}$.

Bien que le quaternion soit plus compact, la matrice de rotation peut être combinée facilement à l'information de translation en formant une matrice de transformation homogène 4x4. En effet, soit :

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{ et } R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

En combinant la position et l'orientation en matrice de transformation homogène 4x4, on obtient :

$$T_H = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1/s_g \end{bmatrix}$$

La valeur de s_g définit un facteur d'échelle global (sur tous les axes) appliqué sur l'objet. Cette valeur affecte également la translation définie dans la matrice et elle est égale sur tous les axes. Pour enrichir davantage la transformation appliquée à un objet, on peut ajouter des facteurs d'échelle individuels et une réflexion sur chacun des axes à l'aide d'une autre matrice. Soit la matrice:

$$S = \begin{bmatrix} \pm s_x & 0 & 0 & 0 \\ 0 & \pm s_y & 0 & 0 \\ 0 & 0 & \pm s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

On définit finalement une transformation T à partir de la matrice homogène:

$$T = ST_H$$

Les valeurs s_x, s_y et s_z déterminent le facteur d'échelle appliqué sur chacun des axes et le signe détermine s'il y a réflexion (-) ou non (+) sur chacun d'entre eux. Cette réflexion inverse les valeurs sur un axe par rapport au plan formé par les deux autres axes. Par exemple, si la valeur de s_x est négative, alors les valeurs sur l'axe des X seront inversées selon le plan YZ. Une fois combinées à l'information d'orientation et de position dans une seule matrice, les valeurs des facteurs d'échelle ne sont par contre pas plus faciles à extraire. Il faut donc les conserver à part si on veut y accéder sans augmenter le temps de calcul.

Avec les matrices, la transformation peut donc être appliquée en une seule opération contrairement aux quaternions qui ne contiennent que l'information de rotation. De plus, certains processeurs graphiques appelés généralement GPU[1] sont optimisés pour le calcul matriciel.

La matrice d'orientation est obtenue facilement en combinant trois matrices représentant les rotations θ_x, θ_y et θ_z autour de trois axes orthogonaux. La représentation de ces rotations ainsi que les matrices correspondantes sont illustrées à la Figure 1-4. On obtient une orientation quelconque en multipliant les trois matrices dans un ordre arbitraire mais fixe. Cet ordre détermine la convention d'Euler utilisée (XYZ, ZYX, XYX, etc.). Il faut porter une attention particulière à cet ordre car les matrices d'orientation résultantes ne seront pas nécessairement identiques même si les angles sont égaux sur les trois axes. En effet, soit :

$$R_{xyz} = R_x R_y R_z \text{ et } R_{zyx} = R_z R_y R_x$$

En général :

$$\boldsymbol{R}_{xyz} \neq \boldsymbol{R}_{zyx}$$

La première rotation (multiplication) s'effectue sur le repère original de l'objet. Par la suite, les rotations subséquentes sont appliquées au repère modifié par les précédentes (post-multiplications). Comme le démontre la Figure 1-5, avec des angles de $\theta_x = \pi/2$,

[1] GPU : signifie « *Graphics Processing Units* » en anglais.

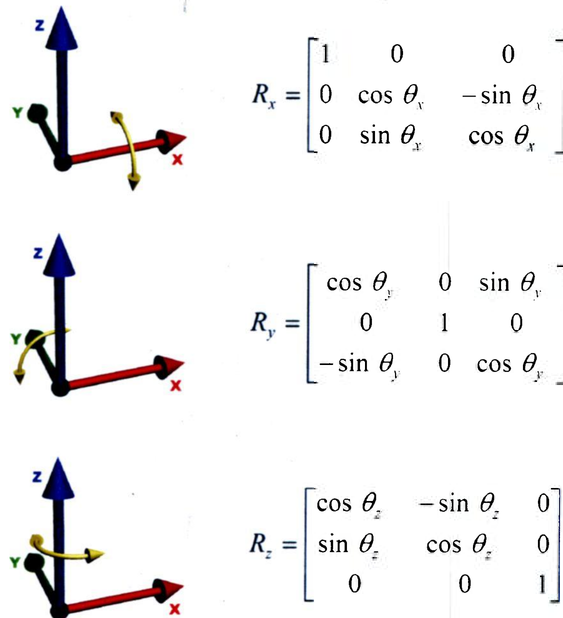


Figure 1-4: Matrices de rotation indépendante autour des axes X, Y et Z

$\theta_y = -\pi/2$ et $\theta_z = \pi/2$, le fait d'avoir les mêmes angles de rotation sur chacun des axes avec des conventions différentes ne garantit pas d'avoir une orientation identique. Cependant, lorsque la matrice finale est construite, l'information d'orientation contenue dans celle-ci est unique et ne dépend d'aucune convention. Le quaternion possède également cette propriété, mais il n'est pas construit de la même façon avec les trois angles.

La convention d'Euler varie selon les systèmes et les domaines. Dans les librairies développées au cours de ce projet, la convention utilisée est XYZ quand on veut construire une matrice à l'aide de trois angles. Le système de positionnement IS-900 fournit ces valeurs selon la convention ZYX.

1.2.3 Évaluation de la profondeur

Puisque l'affichage de la scène virtuelle s'effectue toujours en deux dimensions sur une surface quelconque, on doit faire appel à différentes astuces afin de simuler la troisième dimension et de permettre à l'utilisateur d'estimer la distance des objets par rapport à son propre point de vue. Ce sont les indices de profondeur qui rendent les interactions avec l'environnement plus faciles pour le participant, surtout lorsqu'il s'agit de la sélection, de la manipulation ou de la navigation. En effet, ces indices lui fournissent de l'information sur

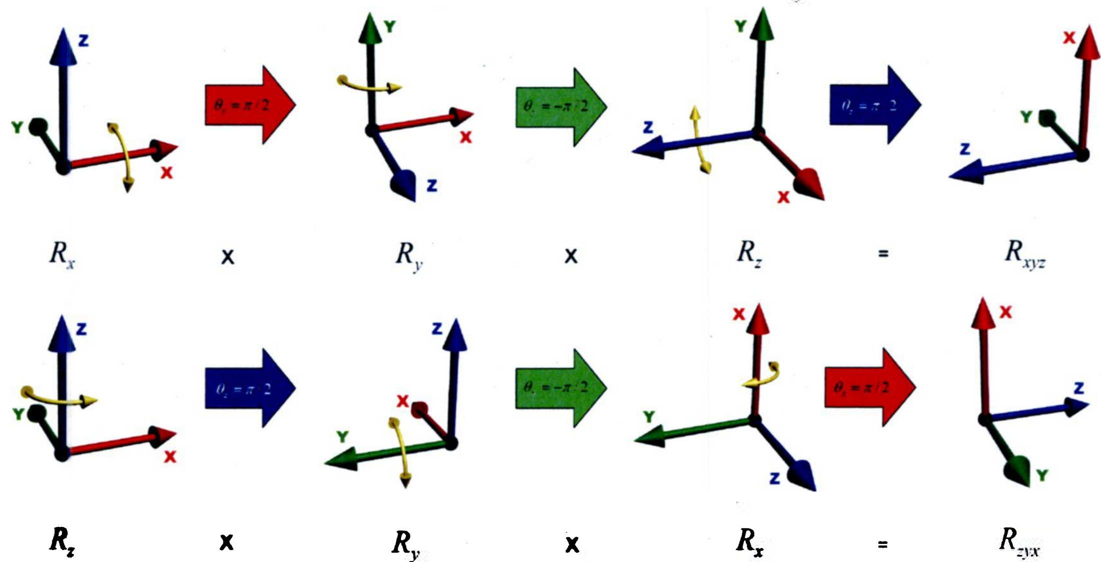


Figure 1-5: Rotations résultantes différentes avec les conventions d'Euler XYZ et ZYX

la position relative des objets et sur son propre emplacement dans l'environnement. Il existe quatre catégories d'indices de profondeur : les indices monoculaires ou statiques, les indices oculomoteurs, la parallaxe du mouvement et la disparité binoculaire qu'on appelle également l'affichage stéréoscopique.

1.2.3.1 Indices de profondeur monoculaires

Les indices monoculaires procurent de l'information simple sur la distance à l'aide d'un seul œil. Tout d'abord, la *taille* et la *hauteur relative des objets par rapport à l'horizon* permettent de savoir si un objet est plus éloigné qu'un autre. Ensuite, l'*occlusion* permet de savoir si un objet est derrière un autre si celui-ci le cache partiellement. La *perspective linéaire* fait en sorte que des droites parallèles convergent avec la distance. La *perspective aérienne* suppose que la lumière est absorbée par l'atmosphère avec la distance. Les objets rapprochés apparaîtront donc plus saturés en couleur. De plus, la *projection des ombres* sur la scène permet également d'avoir une idée de la position des objets. Enfin, la *densité de la texture* des objets augmentera avec la distance ce qui fournit une information supplémentaire sur la profondeur.

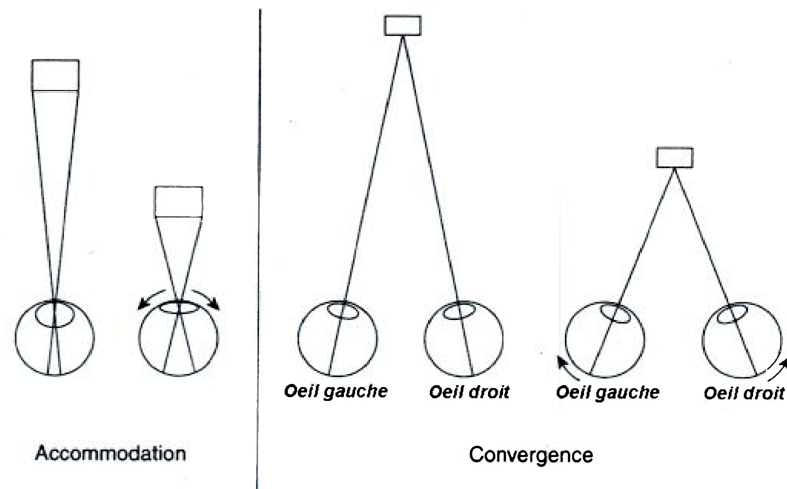


Figure 1-6: Principes d'accommodation et de convergence

1.2.3.2 Indices de profondeur oculomoteurs

Les indices oculomoteurs sont fournis directement par les muscles des yeux. Lorsqu'on regarde un objet à une certaine distance, le relâchement ou la contraction du cristallin permet à l'œil d'avoir un bon focus. Ce phénomène s'appelle l'*accommodation*. De plus, en regardant ce même objet, les yeux vont pivoter pour pointer tous les deux vers celui-ci. Ce phénomène s'appelle la *convergence*. Le cerveau est alors capable d'approximer la distance grâce à l'information fournie par les muscles. La Figure 1-6 présente ces deux indices oculomoteurs.

1.2.3.3 Parallaxe du mouvement

La parallaxe du mouvement fait en sorte que les objets rapprochés se déplacent plus rapidement dans le champ de vue de l'utilisateur que les objets éloignés. Cet indice est dynamique car il implique que les objets de la scène soient en mouvement pour pouvoir déduire l'information de profondeur relative.

1.2.3.4 Disparité binoculaire et affichage stéréoscopique

La disparité binoculaire concerne la différence qui existe entre les images observées par chacun des yeux de l'utilisateur. Plus un objet est proche, plus grande est la disparité. À l'inverse, un objet très éloigné sera perçu sensiblement de la même manière par les deux

yeux. L'affichage stéréoscopique est une particularité très intéressante de certains outils de rendu qui permet de générer cet indice sur un écran à 2 dimensions en générant une image pour l'œil droit et une autre pour l'œil gauche. On peut ainsi ajouter beaucoup de réalisme aux applications développées. De plus, le principe de base est assez simple et les résultats sont très satisfaisants.

Comme on le sait, l'humain possède deux yeux afin de voir en trois dimensions et ainsi d'évaluer la profondeur. Lorsqu'on regarde une scène réelle en trois dimensions, chaque œil ne perçoit pas la même image. Chacune de ces images est en deux dimensions, mais le cerveau est capable d'en extraire l'information pour évaluer la troisième.

On peut donc appliquer le même concept à la RV en projetant chaque point de la scène sur l'écran deux fois, une fois pour chaque œil. Il suffit de déterminer le point d'intersection entre l'écran et la droite « œil/point virtuel ». Ainsi, on obtient deux images et le cerveau peut extraire l'information comme dans le monde réel. La Figure 1-7 démontre le principe de projection de la scène sur l'écran. Dans cette image, l'objet virtuel est le cube vert. L'œil droit et son image perçue sont représentés par la couleur bleue. L'œil gauche et son image perçue quant à eux, sont en rouge. Les droites rouges et bleues déterminent l'intersection sur l'écran des segments entre tous les points de l'objet virtuel et chaque œil. Évidemment,

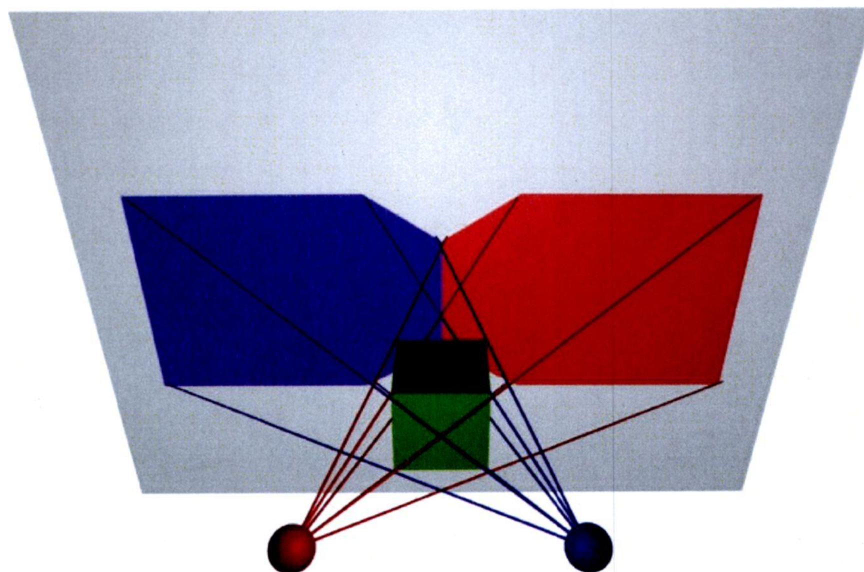


Figure 1-7: Principe de l'affichage stéréoscopique

les faces du cube sur chaque projection devront posséder des propriétés lumineuses afin d'accentuer le réalisme.

Il ne manque qu'un détail pour que la méthode fonctionne. Si les deux images sont affichées simultanément à l'écran, la scène sera floue car il y a en fait deux images disjointes. Il faut donc que chaque œil perçoive son image sans voir l'autre. Pour se faire, il existe deux approches : la stéréoscopie active ou passive. Dans la *stéréoscopie active*, les images pour chaque œil sont affichées alternativement à l'écran. On utilise des lunettes stéréoscopiques et un périphérique infrarouge pour les contrôler. Les projecteurs rafraîchissent l'image 96 fois par seconde et les lunettes sont munies de deux volets qui bloquent la vue à l'œil qui ne correspond pas à l'image projetée. Ainsi, chaque œil obtient une image distincte à un taux de rafraîchissement de 48 Hz. Le sens de la vue ne perçoit plus les transitions d'image à cette fréquence et les volets semblent alors transparents et diminuent simplement la luminosité. La scène observée semble donc fluide et en trois dimensions. Un dispositif envoie aux lunettes un signal infrarouge indiquant quel volet fermer pour percevoir la bonne image à l'écran. Chaque œil perçoit alors au bon moment l'image de son point de vue et l'impression de profondeur apparaît. Dans la *stéréoscopie passive*, les images sont polarisées et affichées simultanément. Les lunettes doivent alors dépolariser les images pour les faire correspondre à chaque œil. Il existe deux modes d'affichage stéréoscopique : le rendu *on-axis* et le rendu *off-axis*.

Affichage stéréoscopique on-axis

Le rendu *on-axis* est la manière la plus simple d'afficher une scène en trois dimensions. Tout d'abord, le point de vue de l'utilisateur est supposé fixe et souvent positionné au centre de l'écran. En déplaçant la tête autour du point de vue, la perception de la scène ne changera donc pas.

La simplicité de ce type de rendu amène un autre problème lorsqu'on utilise l'affichage stéréoscopique. En effet, le rendu *on-axis* suppose également que les yeux de l'utilisateur se trouvent sur un axe fixe, souvent l'axe gauche/droit. Ainsi, si on pivote la tête, les images perçues par chaque œil ne correspondent plus et l'effet de profondeur disparaît.

Ce type de rendu est utilisé pour la plupart des jeux vidéo actuels. Puisque la dimension de l'écran est raisonnable et que l'utilisateur ne bouge pas beaucoup, ce modèle suffit amplement d'autant plus qu'il simplifie beaucoup les calculs et le traitement du rendu. Cependant, l'affichage sur les écrans d'un système tel le FLEX™ demande une technique plus complexe afin d'obtenir le niveau de réalisme désiré.

Les deux contraintes du rendu *on-axis* sont représentées dans la Figure 1-8. Dans cette image, on voit d'abord l'erreur de perception lorsque l'utilisateur déplace la tête par rapport à un objet virtuel situé en avant de l'écran. On voit également que l'axe des yeux est supposé fixe même si l'utilisateur pivote la tête.

Affichage stéréoscopique off-axis

Pour atteindre un niveau de réalisme adéquat avec une plus grande surface de rendu, il faut plutôt utiliser l'affichage *off-axis*. Dans ce type de rendu, la position de la tête de l'utilisateur n'est pas supposée fixe. Ainsi, grâce à un traqueur de tête, celui-ci pourra se déplacer dans l'environnement et l'examiner sous tous ses angles. La scène ainsi perçue à l'écran sera sans cesse mise à jour afin de représenter le point de vue de l'utilisateur.

Il n'y a pas non plus de problème avec l'affichage stéréoscopique. En effet, puisque l'orientation de la tête n'est pas supposée fixe, l'axe des yeux devient également dynamique et la disparité entre les images s'effectue dans toutes les directions. Les deux types d'erreur de perception de la Figure 1-8 introduits par le modèle *on-axis* disparaissent donc.

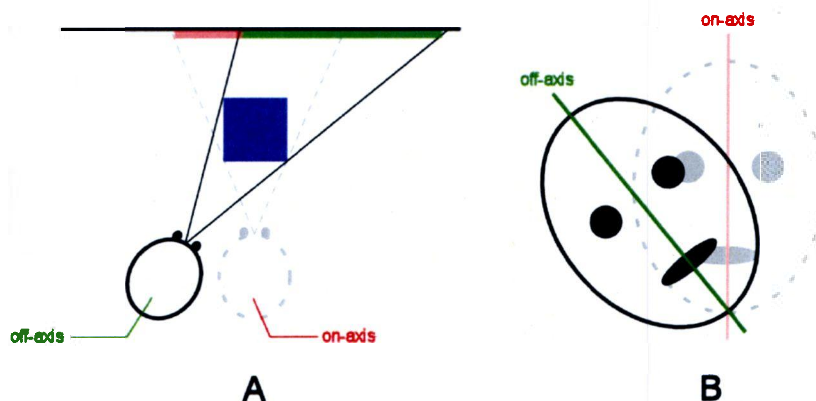


Figure 1-8: Deux contraintes du rendu on-axis
a) Le on-axis suppose une position fixe de la tête de l'utilisateur
b) Le on-axis suppose une orientation fixe de la tête de l'utilisateur

Cette méthode nécessite évidemment beaucoup plus de calculs, mais donne un aspect beaucoup plus réaliste à l'expérience virtuelle. Il est donc primordial d'introduire la stéréoscopie *off-axis* à l'affichage car le développement d'un environnement immersif et intuitif est l'un des critères les plus importants de ce projet.

Malgré l'utilisation du rendu *off-axis*, il existe encore quelques indices de profondeur qui demeurent problématiques. Tout d'abord, la position de la pupille est supposée fixe. Ainsi, même si l'utilisateur bouge ses yeux sans déplacer la tête, il n'y aura aucun changement à l'écran. De plus, il y a une disparité entre l'information fournie par l'accommodation des yeux (focus) et leur convergence. En effet, dans une scène réelle, la convergence et l'accommodation des yeux s'effectue à la même profondeur dépendant de la distance où se trouve l'objet regardé. Cependant, dans une scène virtuelle, la mise au foyer se fait toujours au niveau de l'écran, mais la convergence se fait sur l'objet virtuel. Le cerveau peut se réadapter à cette différence, mais certaines personnes auront plus de difficulté que d'autres. En effet, chez certaines personnes, cette disparité pourra causer des maux de tête ou la nausée. La Figure 1-9 montre la contradiction entre ces deux indices de profondeur oculomoteurs. Bien qu'il existe en ce moment des prototypes permettant de traquer les pupilles et de simuler un focus artificiel, ces systèmes sont encore beaucoup trop coûteux et peu performants pour être facilement exploitables. Il faudra donc développer l'application finale en tenant compte de ces limitations.

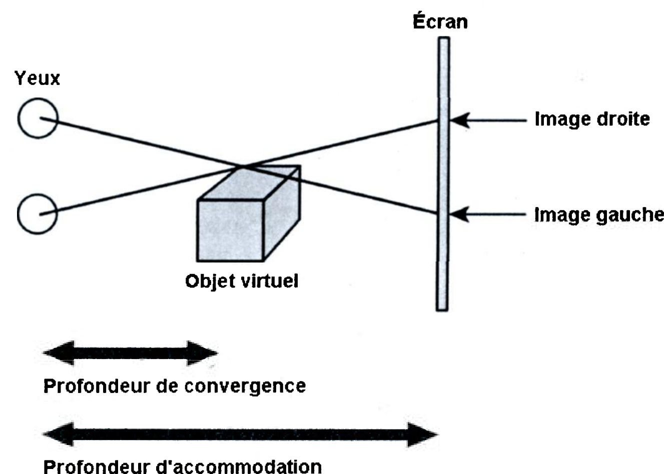


Figure 1-9: Contradiction entre la convergence et l'accommodation des yeux dans un environnement virtuel

1.2.4 Technologies de rendu graphique

Il existe actuellement sur le marché plusieurs types d'affichage visuel qui varient en performance, en technologie et en prix. Pour comparer ces périphériques, (Bowman, Kruijff et al. 2005) présente plusieurs propriétés pouvant être évaluées, notamment : le champ de regard et le champ de vision, la résolution spatiale, la géométrie de l'écran, le mécanisme de transfert lumineux, le taux de rafraîchissement et l'ergonomie. On peut également vérifier la disponibilité des indices de profondeur présentés plus tôt. Avant de présenter les différentes technologies de rendu graphique, voici d'abord une brève définition de ces propriétés.

Le **champ de regard** (FOR pour *Field Of Regard* en anglais) décrit la quantité d'espace physique qui entoure l'utilisateur sur laquelle des images peuvent être affichées. Il est mesuré en degrés par rapport à l'angle visuel. Le **champ de vision** (FOV pour *Field Of View* en anglais) décrit plutôt l'angle maximal (en degrés) qui peut être observé simultanément sur le périphérique d'affichage. Il y a une distinction majeure entre ces deux caractéristiques. Le FOR décrit l'espace du monde réel où l'utilisateur peut regarder la scène virtuelle tandis que le FOV décrit l'espace du monde virtuel qui peut être observé à un moment précis. Le FOV varie selon la distance entre l'œil et la surface de rendu et ne pourra jamais dépasser le FOV du système visuel humain (environ 200 degrés). De plus, le FOV sera toujours inférieur ou égal au FOR.

La **résolution spatiale** est reliée à la taille des pixels et à celle de l'écran. Elle fournit une mesure de la qualité de l'affichage souvent donnée en dpi (*dots per inch*). Il faut ici faire la distinction avec la résolution d'affichage utilisée pour les écrans d'ordinateur comme 800x600 et 1024x768 par exemple. À une même résolution d'affichage, la résolution spatiale peut changer si la taille de l'écran varie. En effet, l'augmentation de la taille de l'écran élargira les pixels affichés et la qualité de l'image obtenue sera moindre. Il faut donc augmenter la résolution d'affichage en même temps que la taille de l'écran si on veut conserver une résolution spatiale constante. La distance de l'utilisateur avec l'écran peut aussi faire varier la résolution spatiale perçue par celui-ci. En effet, plus il se trouvera près de l'écran, plus il pourra distinguer les pixels et la qualité de l'image diminuera en

conséquence. C'est d'ailleurs le principal problème des *head-mounted displays* (HMD) présentés plus loin.

La **géométrie de l'écran** est importante à considérer pour certains systèmes dont la surface de projection n'est pas plane et rectangulaire. Il faut alors utiliser des algorithmes de projection non standards qui peuvent affecter la qualité visuelle. Dans certains cas, on verra apparaître de la distorsion sur les bords de l'écran. De plus, la résolution spatiale ne sera pas toujours uniforme sur toute la surface.

Pour le **mécanisme de transfert lumineux**, on parle ici de l'emplacement et de la technologie de la source lumineuse. Par exemple, on peut utiliser une projection en arrière ou en avant de l'écran. D'autres systèmes utilisent des technologies plus complexes comme la projection d'un laser directement sur la rétine. On voit rapidement les limites imposées par un environnement utilisant une projection avant dans lequel l'utilisateur peut projeter son ombre sur l'écran.

Le **taux de rafraîchissement** donne en images par seconde (Hz) la vitesse à laquelle le périphérique rafraîchit l'image à partir d'un tampon en mémoire. Un taux de rafraîchissement inférieur à 50-60 Hz peut causer des saccades selon la sensibilité du système visuel de l'utilisateur, surtout si l'affichage stéréoscopique est utilisé car le taux de rafraîchissement est alors diminué de moitié pour chaque œil.

Finalement, l'**ergonomie** dans les systèmes de rendu graphique prend surtout en compte le confort de l'utilisateur. On veut idéalement que le périphérique ne soit pas porté par celui-ci ou qu'il soit léger. Il faut aussi avoir un minimum d'obstruction pour les mouvements.

Dans les systèmes actuellement disponibles sur le marché, on trouve notamment les moniteurs standards, les affichages à écrans *surround*, les *workbenches*, les affichages hémisphériques, les *head-mounted displays* (HMD), les *arm-mounted displays* (AMD), les affichages à projection rétinienne et les affichages auto-stéréoscopiques.

1.2.4.1 Moniteurs standards

L'utilisation de moniteurs standards est une manière peu onéreuse pour créer un environnement virtuel 3D. Ils peuvent fournir les indices de profondeur monoculaires et la parallaxe du mouvement. Une stéréoscopie *on-axis* plus ou moins efficace peut également être obtenue à l'aide d'une carte graphique spéciale. Le moniteur possède une excellente résolution spatiale et permet facilement l'utilisation des périphériques de bureau. Cependant, la mobilité de l'utilisateur est réduite à cause du FOR très petit. De plus, le bord de l'écran peut faire de l'occlusion à la scène virtuelle et entrer en contradiction avec les indices de profondeur.

1.2.4.2 Écrans *surround*

Un système d'affichage à écrans *surround* est composé de larges surfaces de projection entourant l'utilisateur. Ces écrans géants ont habituellement une hauteur et une largeur comprises entre 8 et 12 pieds et on en compte souvent plus de trois. La plupart du temps, une projection arrière est utilisée pour s'assurer que l'utilisateur ne projette pas d'ombre sur la scène observée. Un des principaux avantages est son large FOR et un FOV équivalent. L'utilisateur peut donc se déplacer dans l'environnement et utiliser sa vision périphérique

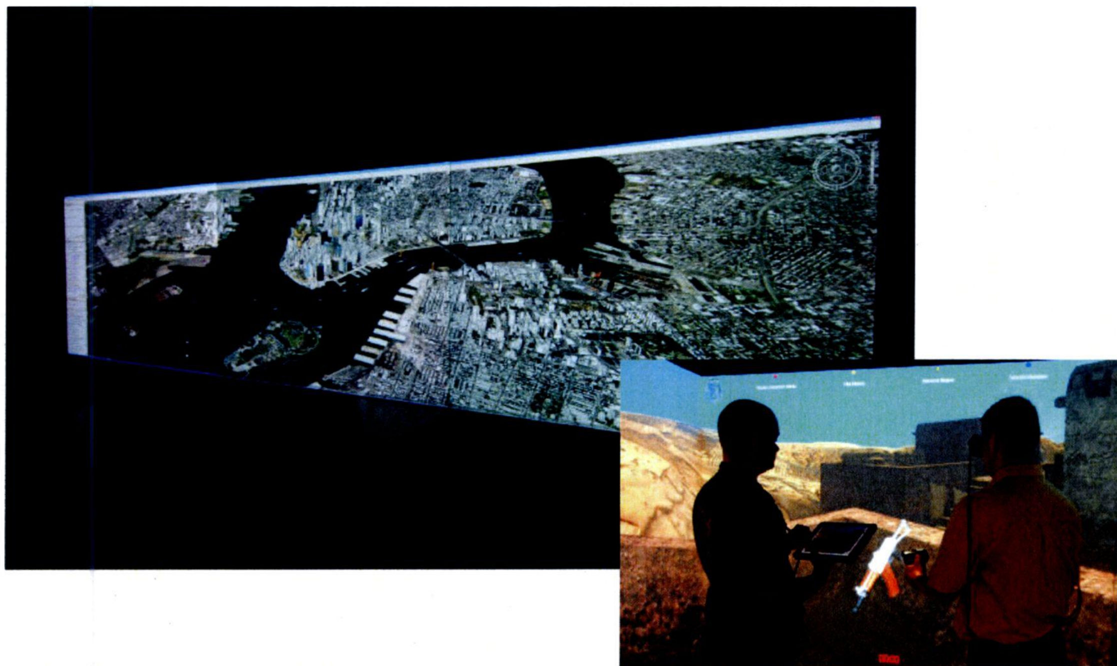


Figure 1-10: Système FLEX utilisé au LIV fabriqué par Fakespace Systems Inc.

pour observer le monde virtuel. Les indices monoculaires, la parallaxe du mouvement et la stéréoscopie *off-axis* sont également disponibles. Les principaux désavantages sont le prix et l'occupation d'espace. De plus, il est difficile d'avoir une application à plusieurs participants à moins d'utiliser autant de lunettes stéréoscopiques et d'augmenter la fréquence pour faire correspondre le point de vue pour chacun d'entre eux. Cependant, on atteint vite la limite de performance du matériel avec une telle approche.

Le système FLEX™, utilisé pour ce projet et illustré à la Figure 1-10, entre dans cette catégorie. Comme mentionné plus tôt, il possède quatre projecteurs et offre la possibilité de reconfigurer la surface de rendu grâce aux écrans latéraux amovibles. Selon la disposition choisie, il peut offrir un FOR jusqu'à plus de 200 degrés avec un FOV équivalent si l'utilisateur regarde dans la bonne direction. Le fait de n'apercevoir que la scène virtuelle augmente alors grandement le réalisme. Évidemment, le champ de vision varie selon la position et l'orientation de la tête du participant car il n'y a pas de surface de rendu à l'arrière ni au dessus.

1.2.4.3 Workbenches

Les *workbenches* possèdent pratiquement les mêmes particularités que les moniteurs standards. Cependant, ce périphérique a été conçu pour simuler le travail sur une table ou sur un bureau. Ils sont utiles entre autres pour le dessin et la modélisation. On travaille

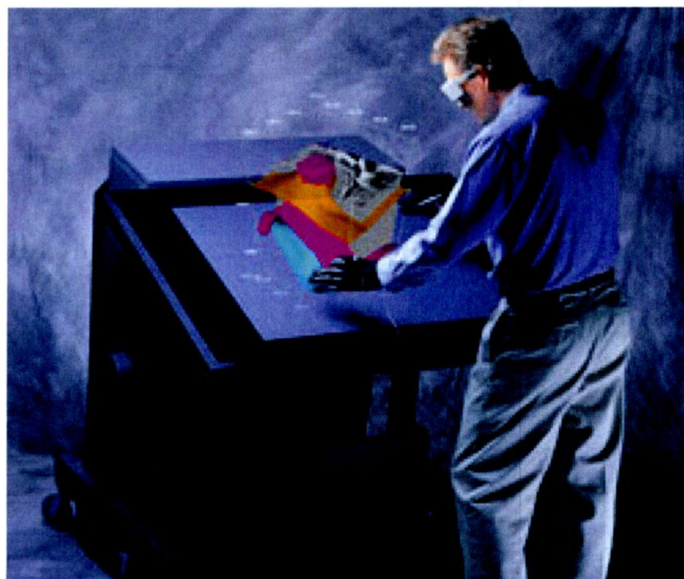


Figure 1-11: Workbench modèle M1 Desk de Fakespace Systems Inc.

souvent directement sur la surface de projection plutôt que d'utiliser d'autres périphériques pour entrer les commandes indirectement. Évidemment, ils sont un peu plus dispendieux que les écrans standards. La Figure 1-11 montre le modèle M1 Desk fabriqué par la compagnie Fakespace Systems Inc.

1.2.4.4 Affichages hémisphériques

Les affichages hémisphériques utilisent une lentille à grand angle à la sortie du projecteur et des fonctionnalités logicielles afin de fournir une image avec un FOV de 180 degrés sur une surface hémisphérique de diamètre variable. La partie logicielle impose une distorsion à l'image à la sortie de la carte graphique et la lentille élimine cet effet pour que l'image soit reconnaissable sur l'écran non planaire. Ce type d'affichage offre également les indices de profondeur monoculaire et binoculaire ainsi que la parallaxe du mouvement. Cependant, ce genre de système utilise une projection avant et l'utilisateur peut donc projeter son ombre sur l'écran lors de certaines tâches. De plus, la résolution spatiale ne sera pas constante sur toute la surface de projection. La qualité de l'image sera meilleure au centre et se dégradera en s'approchant du bord de l'écran. La Figure 1-12 montre le système VisionStation de Elumens Corporation.

Figure 1-12: Affichage hémisphérique VisionStation de Elumens Corporation

1.2.4.5 Head-Mounted Displays

Le *Head-Mounted Display* (HMD) ou casque de réalité virtuelle en français est attaché directement sur la tête de l'utilisateur. Il permet de placer les images devant ses yeux avec un (affichage monoscopique) ou deux (affichage stéréoscopique) petits écrans. La technologie LCD est la plupart du temps utilisée car les écrans sont moins lourds. Cependant, les écrans CRT offrent une meilleure résolution. Une autre approche utilise des projecteurs montés sur la tête et une surface rétroréfléchissante située dans l'environnement permet le rendu. Ce matériel a la propriété de réfléchir un rayon lumineux dans la direction opposée peu importe l'angle incident. Les HMD bloquent la vue de l'utilisateur sur le monde réel. Bien que cet aspect soit un avantage dans certaines situations, il soulève par contre quelques problématiques au niveau de la sécurité. Le principal avantage des HMD est son FOR de 360 degrés. En effet, l'utilisateur peut tourner la tête dans toutes les directions pour observer le monde virtuel. De plus, ils permettent d'avoir plusieurs utilisateurs simultanément car le point de vue généré est local à chacun d'entre eux en comparaison avec les techniques mentionnées plus haut. À l'opposé, le FOV d'un HMD est habituellement limité ce qui diminue grandement la qualité d'immersion. De plus, la

Figure 1-13: Head-Mounted Display ProView SR80 de la compagnie Rockwell Collins

résolution spatiale est faible à cause de la taille des écrans et de la proximité des yeux de l'utilisateur. Enfin, les indices oculomoteurs d'accommodation et de convergence sont davantage en conflit. Les périphériques de sortie mentionnés plus tôt possèdent également cette limitation, mais cette confusion est encore plus apparente avec les HMD puisque la mise au foyer se fait tout près des yeux. Finalement, la latence est généralement plus remarquable sur ces systèmes en raison du FOV limité. La Figure 1-13 montre le modèle ProView SR80 de la compagnie Rockwell Collins.

1.2.4.6 Arm-Mounted Displays

Le *Arm-Mounted Display* (AMD), présenté à la Figure 1-14, ressemble beaucoup au HMD, mais ces dispositifs ne sont pas portés sur la tête. Ils sont plutôt soutenus par une armature avec un contrepoids qui permet de les déplacer facilement avec les mains. L'utilisation des CRT plus lourds mais de meilleure qualité est alors envisageable. Le principal avantage des AMD par rapport aux HMD est donc l'ergonomie car l'utilisateur n'est plus encombré par le système de rendu. Cependant, la mobilité devient alors limitée par l'armature et l'usager devra toujours garder au moins une main sur le périphérique afin de le déplacer avec lui. Cet outil serait idéal pour simuler des observations quelconques qui demandent une bonne

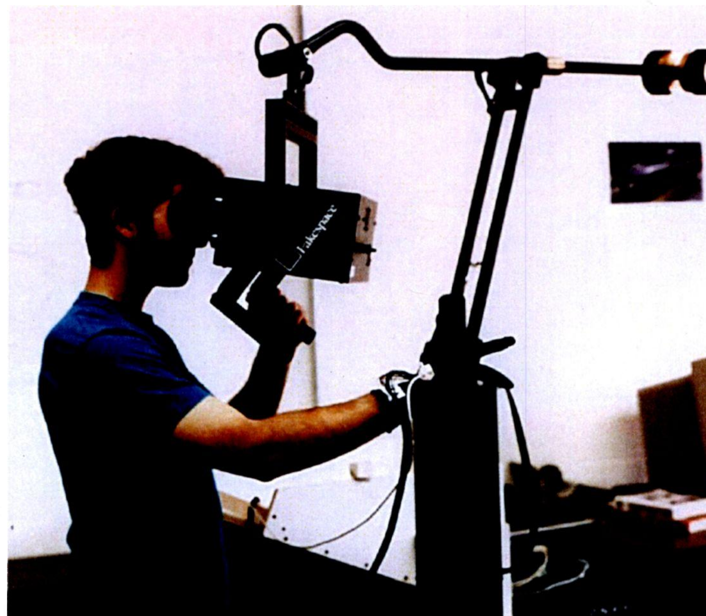


Figure 1-14: Arm-Mounted Display nommé "Binocular Omni-Orientation Monitor" (BOOM) de Fakespace Systems Inc.

résolution sans toutefois nécessiter beaucoup de mouvement : une opération médicale, une observation au microscope ou au binoculaire, etc.

1.2.4.7 Affichages à projection rétinienne

Les affichages à projection rétinienne (VDR pour *Virtual Retinal Display* en anglais) reprennent le principe du HMD mais utilisent plutôt la rétine de l'œil comme surface de rendu. Trois lasers (rouge, bleu et vert) sont utilisés pour créer une image en couleur. Le rayon lumineux balaie la rétine à une fréquence élevée afin d'y positionner chaque point. Bien que la plupart de ces systèmes soient encore au stade de prototypes, le FOV et le FOR correspondraient en théorie à ceux du monde réel donnant ainsi une immersion totale. Il existe cependant deux problèmes majeurs. Tout d'abord, le système suppose que la rétine demeure à une position fixe. En déplaçant la pupille, l'utilisateur peut complètement perdre l'image. Pour remédier à ce problème, il faudrait incorporer un traqueur pour les yeux. Ensuite, le système souffre également de la contradiction accommodation/convergence mentionnée plus tôt. Des prototypes de miroirs déformables sont présentement en développement. Ils permettront de changer le plan focal et d'éliminer ce problème.

Figure 1-15: Affichage auto-stéréoscopique lenticulaire au plasma de 50 po de la compagnie NewSight

1.2.4.8 Affichages auto-stéréoscopiques

Enfin, les affichages auto-stéréoscopiques permettent de produire les indices binoculaires sans l'aide de lunettes spéciales. Il existe trois technologies : lenticulaire, volumétrique et holographique.

Les *affichages lenticulaires*, représentés à la Figure 1-15, produisent deux images sur le même écran. Il existe deux approches : une grille verticale où les pixels pairs et impairs sont associés à un œil différent ou la lentille cylindrique qui envoie les deux images avec un angle différent. Dans les deux cas, l'utilisateur doit se placer à une position précise devant l'écran afin d'avoir un bon affichage stéréoscopique limitant ainsi sa mobilité.

Ensuite, les *affichages volumétriques*, présentés à la Figure 1-16, créent de « vraies » images 3D en illuminant des points dans l'espace à leur position réelle. Des lasers perpendiculaires parcourent le volume et l'intersection des rayons forme un *voxel* (un pixel en 3D).

Enfin, les *affichages holographiques*, dont un exemple est donné à la Figure 1-17, génèrent également de vraies images 3D. Une portion logicielle convertit la scène en hologramme en enregistrant l'information sur la lumière provenant de multiples directions. Une portion

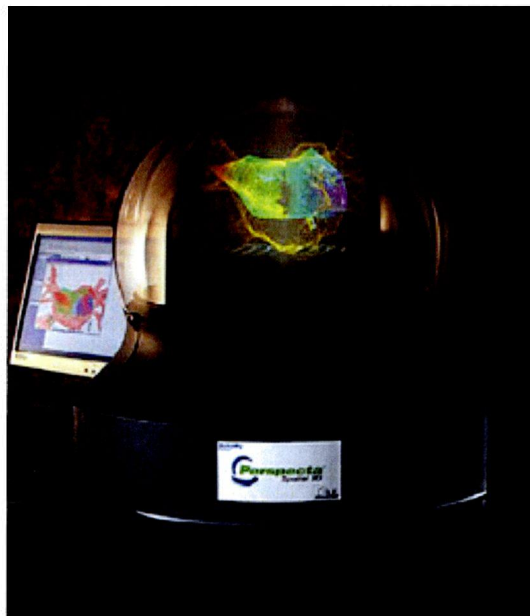


Figure 1-16: Affichage auto-stéréoscopique volumétrique Perspecta de la compagnie Actuality Systems

Figure 1-17: Affichage auto-stéréoscopique holographique HelioDisplay M2 de la compagnie IO2

optique permet ensuite de transformer l'hologramme en objet 3D selon le point d'observation.

Bien que les deux dernières technologies permettent à plusieurs utilisateurs de regarder la scène de points de vue différents, il est difficile de produire des effets d'occlusion et d'ombrage. De plus, le volume de rendu est généralement petit ce qui ne permet pas de créer des environnements immersifs.

1.2.5 Technologies de positionnement

Le but principal du positionnement est d'établir une correspondance entre les objets physiques et ceux de l'environnement virtuel. On obtient cette information grâce à des traqueurs de position. Ceux-ci possèdent différentes caractéristiques critiques comme la portée, la latence, la stabilité (sensibilité au bruit) et la précision qui limitent le choix pour certaines applications. Il existe actuellement cinq technologies pour développer des traqueurs de position : magnétique, mécanique, acoustique, inertielle et optique. Chacune de ces technologies possède ses avantages et ses inconvénients. Des traqueurs hybrides, combinant deux technologies ou plus à la fois, ont donc également été développés pour remédier à ces lacunes.

Il y a toujours deux éléments dans un traqueur : un émetteur qui transmet de l'information selon la technologie utilisée et un récepteur qui détermine la position et l'orientation d'un objet selon la mesure reçue par l'émetteur. Il y a deux approches pour disposer de ces deux éléments. L'approche *outside-in* dispose les émetteurs sur l'objet traqué et les récepteurs sont fixés dans l'environnement physique. L'approche *inside-out* utilise plutôt des émetteurs dans l'environnement et des récepteurs attachés à l'objet traqué.

1.2.5.1 Traqueurs magnétiques

Les traqueurs magnétiques utilisent un émetteur de champ magnétique à basse fréquence et un récepteur qui détermine sa position et son orientation relatives à la source. Ils fonctionnent généralement bien à des distances inférieures à 10 m car la précision du champ magnétique diminue rapidement en s'éloignant de la source. Le principal désavantage de cette technologie est la distorsion du champ magnétique en présence d'objets métalliques conducteurs ou de champs magnétiques parasites dans l'environnement ce qui réduit davantage la précision. Cependant, des procédures de calibrage et des opérations de filtrage peuvent alors améliorer la situation. La compagnie Polhemus produit des traqueurs utilisant cette technologie dont le modèle Fastrak, illustré à la Figure 1-18.

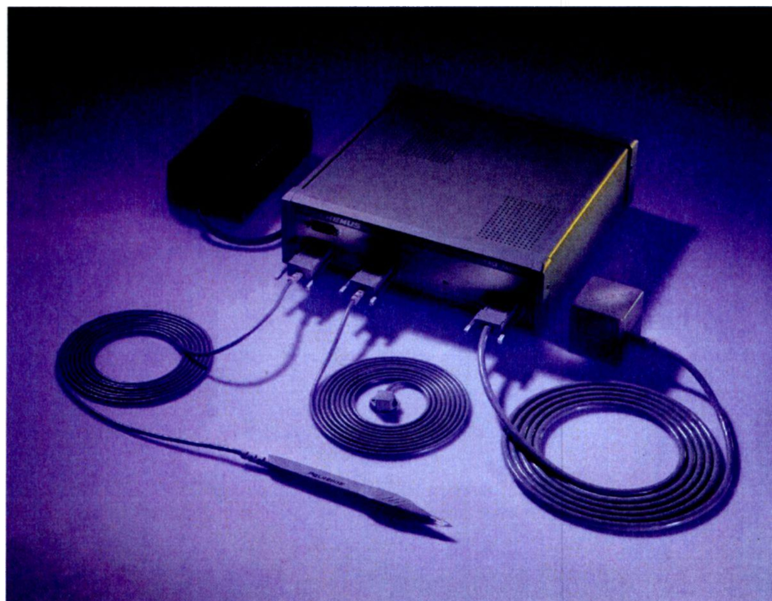


Figure 1-18: Traqueur Fastrak de la compagnie Polhemus

1.2.5.2 Traqueurs mécaniques

Les traqueurs mécaniques possèdent une structure rigide établissant un lien physique (câble, bras articulé, etc.) entre l'objet traqué et un senseur quelconque utilisé pour déterminer la position et l'orientation de l'objet. L'utilisateur peut donc déplacer librement l'objet dans l'espace pendant que l'autre extrémité du lien est fixée sur la structure du traqueur. Le lien physique doit évidemment suivre le mouvement de l'objet traqué ce qui apporte généralement des délais dus à l'inertie du système. De plus, la mobilité de l'utilisateur est limitée par les dimensions du traqueur. Cependant, le lien physique offre une mesure très précise de la position et de l'orientation.

Ce type de traqueur offre la possibilité d'appliquer du retour de force aux systèmes pour rendre la sélection et la manipulation d'objets plus naturelles. C'est d'ailleurs le cas du mécanisme de locomotion développé dans le cadre de projet NELI. En effet, la longueur des câbles peut être utilisée pour déterminer la position et l'orientation des plates-formes dans l'espace alors que les moteurs peuvent induire des forces en réaction à l'environnement virtuel pour déplacer les deux plates-formes. Le simulateur du système NELI développé dans le cadre de ce projet permettra de reproduire la fonctionnalité de positionnement seulement.

1.2.5.3 Traqueurs acoustiques

Les traqueurs acoustiques utilisent des ultrasons pour déterminer la position et l'orientation d'un objet traqué. Un microphone sur le récepteur est utilisé pour capter ces sons et déterminer la distance de l'objet selon le temps de vol du message sonore. Une méthode de triangulation détermine la position et l'orientation de l'objet à l'aide de plusieurs mesures. Les principales limitations de ce type de traqueur sont sa sensibilité aux bruits extérieurs, aux surfaces qui réfléchissent le son et la possibilité d'occultation de l'émetteur. Par contre, ils offrent une solution légère et peu dispendieuse au problème de positionnement.

1.2.5.4 Traqueurs inertiels

Les traqueurs inertiels sont généralement composés de gyroscopes angulaires et d'accéléromètres linéaires. Pendant que le gyroscope calcule la vitesse angulaire lors des

rotations, l'accéléromètre détermine l'accélération pendant les translations. En intégrant ces deux valeurs sur l'échelle du temps, on peut déterminer la position et l'orientation de l'objet traqué. Puisque l'information désirée est obtenue par intégration, il y a accumulation de l'erreur et la précision diminue rapidement avec le temps pour les accéléromètres linéaires. Il existe cependant des techniques avec les gyroscopes pour corriger cette erreur et c'est pourquoi les traqueurs purement inertiels sont souvent à 3 DOF d'orientation seulement. Les 3 autres DOF pour la position seront la plupart du temps définis par une autre technologie et le traqueur devient alors hybride (voir section 1.2.5.6).

1.2.5.5 Traqueurs optiques

Les traqueurs optiques utilisent des techniques de vision numérique et des senseurs optiques pour mesurer la position et l'orientation d'un objet par rapport à des sources lumineuses ou des cibles facilement repérables. Le nombre de degrés de liberté de l'objet traqué dépend directement du nombre de cibles et de caméras dans l'environnement. En augmenter le nombre permet d'améliorer la qualité du traqueur au prix d'une plus grande complexité. En effet, la calibration d'un traqueur optique est relativement complexe et l'algorithme utilisé doit être performant pour demeurer en temps réel. Un trop grand nombre de caméras ou de cibles peut nuire à la performance. Le système souffre également d'un problème d'occultation. L'utilisateur peut facilement cacher une cible ou obstruer le champ de vision des caméras réduisant ainsi l'efficacité du système. Cependant, les traqueurs optiques permettent de dissocier complètement l'utilisateur de la machine ce qui constitue un avantage précieux au niveau du confort et de la mobilité.

1.2.5.6 Traqueurs hybrides

Il est également possible de combiner plusieurs de ces technologies dans un traqueur hybride pour augmenter la performance en réduisant les limitations individuelles de chaque type de traqueur. Par exemple, le système InterSense IS-900 présenté dans sa version sans-fil à la Figure 1-19 utilise un traqueur inertiel/acoustique afin de déterminer l'orientation avec des gyroscopes et approxime la position à l'aide d'accéléromètres. L'accumulation d'erreur due aux accéléromètres est ensuite corrigée par un système à ultrasons. Le système

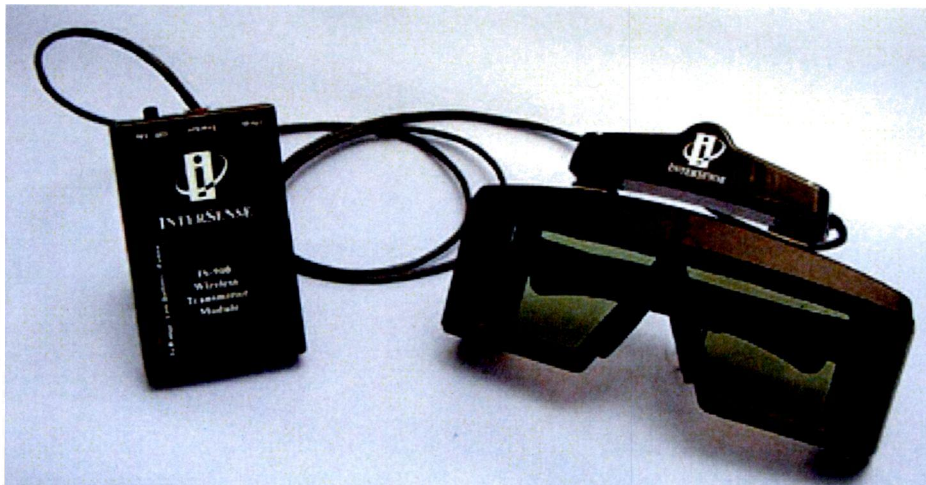


Figure 1-19: Traqueur et module sans-fil du système IS-900

GPS est aussi souvent utilisé en conjonction avec des gyroscopes pour traquer des objets dans un environnement plus vaste.

1.2.5.7 Traqueurs de pupilles

Les travaux sur ce type de traqueur sont encore jeunes, mais déjà quelques techniques ont été développées afin d'obtenir la position de la pupille de l'utilisateur et de déterminer la direction exacte de son regard. En effet, lorsque la tête de l'utilisateur est traquée dans les systèmes actuels, la position des pupilles est souvent fixée au centre des yeux ce qui diminue l'effet d'immersion lors de l'exploration visuelle de l'environnement. De plus, connaître la position des pupilles permet d'améliorer la performance et le confort lorsque des techniques d'interaction basées sur le regard sont utilisées. Ces techniques seront présentées dans les sections suivantes. Il existe actuellement trois approches pour traquer les pupilles: un système de caméra détectant la position de la pupille dans l'œil, des électrodes mesurant le potentiel électrique dans les muscles autour de l'œil et des systèmes de référence miniatures (généralement magnétiques) embarqués sur des verres de contact. Bien que plusieurs projets de recherche soient présentement en cours, seule la première technologie est actuellement disponible sur le marché. Le modèle avec système de caméra haute vitesse EyeLink II de SR Reseach, Ltd. est présenté à la Figure 1-20.

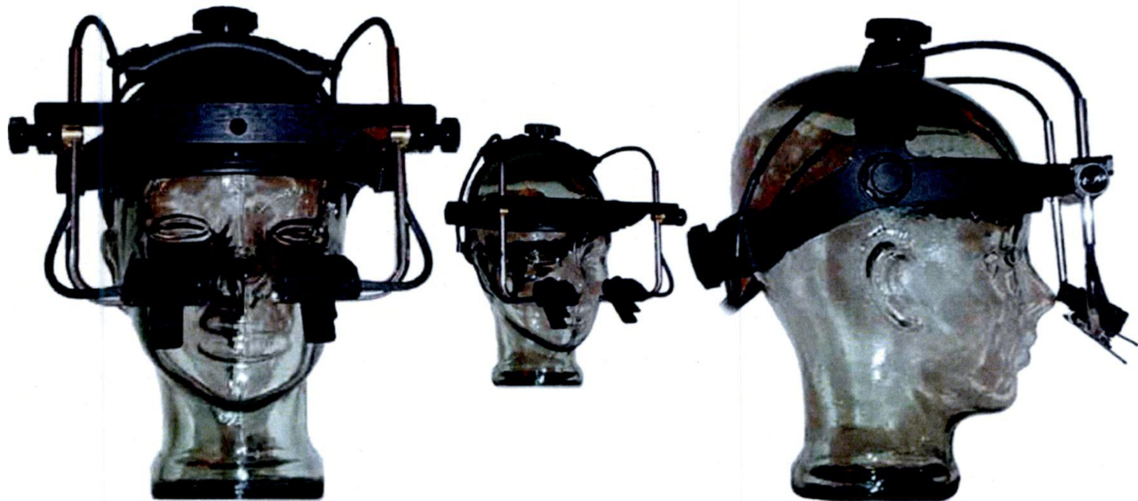


Figure 1-20: Traqueur de pupilles EyeLink II de SR Research, Ltd.

1.2.6 Métaphores de manipulation

La tâche de manipulation est définie comme la composante qui associe les entrées de l'utilisateur comme la trajectoire de la main ou l'activation d'un bouton à une action particulière effectuée sur un objet virtuel comme son déplacement et sa rotation.

On peut diviser la tâche de manipulation en trois tâches plus basiques: la sélection, le positionnement et la rotation. La *sélection* concerne l'identification et l'acquisition d'un objet virtuel particulier dans la scène. Le *positionnement* est la tâche qui définit la position en 3D de l'objet dans l'espace. Finalement, la *rotation* permet de changer son orientation.

Plusieurs paramètres peuvent affecter la performance de ces tâches dont la distance et la direction de la cible, la grosseur de l'objet et la densité d'éléments autour de celui-ci. Il existe également deux caractéristiques importantes à propos des périphériques utilisés pour les techniques de manipulation. Le *nombre de dimensions de contrôle* définit le nombre de degrés de liberté (DDL ou DOF pour *degrees of freedom* en anglais) que le composant peut contrôler. L'*intégration* concerne plutôt la quantité de ces degrés de liberté qui peuvent être contrôlés simultanément lors d'un seul mouvement. Les périphériques les plus performants pour la manipulation 3D seront ceux qui permettent un grand nombre de dimensions de contrôle avec une intégration complète. Les traqueurs de position présentés à la section 1.2.5 remplissent généralement bien ce critère de performance.

La forme et l'emplacement des périphériques sur le corps sont également des caractéristiques importantes. En effet certains périphériques demanderont un plus grand effort de la part de l'utilisateur s'ils sont attachés sur la main car plusieurs muscles seront sollicités pour la manipulation en plus de devoir soutenir un poids supplémentaire. De plus, à cause des limites physiques des bras et des poignets, on verra apparaître un phénomène appelé le *clutching* où les déplacements imposés à un objet ne peuvent pas toujours être définis avec un seul mouvement. Par exemple, il sera difficile d'effectuer des rotations de plus de 180 degrés avec le poignet.

Enfin, une technique de manipulation peut être isomorphique ou non. L'*approche isomorphique* établit un rapport direct entre les déplacements physique et virtuel. Ces techniques sont souvent plus naturelles, mais elles sont également moins efficaces à cause des limitations physiques de l'humain (phénomène de *clutching*). L'*approche non-isomorphique* permet de manipuler les objets autrement que dans le monde réel en faisant varier le rapport entre les distances réelles et virtuelles. Ces techniques de manipulation seront souvent plus performantes et pourront être utilisées si le réalisme n'est pas un critère essentiel.

Les sections suivantes présenteront les différentes approches pour la manipulation d'objets virtuels soit: le pointage, la manipulation directe avec une main virtuelle, le monde miniature, la combinaison de techniques et les rotations non-isomorphiques.

1.2.6.1 Techniques de pointage

Les techniques de pointage permettent à l'utilisateur de sélectionner des objets qui sont hors de sa portée simplement en les pointant. Lorsqu'un objet est intersecté par le vecteur de direction du pointage, on peut confirmer la sélection avec une commande quelconque (bouton, signe, etc.) selon le périphérique utilisé.

Pour définir une technique de pointage, on a besoin de deux variables. La première indique comment la position et l'orientation du périphérique d'entrée utilisé affectent la direction du pointage. La seconde concerne la forme du volume de sélection. Cette deuxième

variable définit également l'information visuelle présentée à l'utilisateur et la quantité d'objets pouvant être sélectionnés en même temps.

En général, les techniques de pointage offriront une meilleure performance de sélection que les techniques de manipulation directe avec main virtuelle à cause de la grande portée. Cependant, le positionnement et la rotation sont limités à cause des contraintes de mouvement imposées par le rayon.

On distingue six techniques de pointage qui seront présentées dans les prochains paragraphes: le *ray-casting* simple, le pointage à deux mains, la lampe de poche, l'aperture, le plan image et la canne à pêche.

1.2.6.1.1 Ray-casting

Dans cette technique, l'utilisateur pointe vers les objets qu'il veut sélectionner. Un rayon virtuel est attaché à la main. Sa direction est représentée par le vecteur unitaire $\vec{p}$ et on définit ensuite le segment de droite $p(\alpha)$ qui traverse les objets virtuels à partir de la position de la main h :

$$p(\alpha) = h + \alpha \vec{p}, \quad 0 < \alpha < +\infty$$

Plusieurs objets peuvent être intersectés par le segment de droite $p(\alpha)$. Le plus près de la main sera la plupart du temps sélectionné par défaut. Cette technique sera la plus simple et la plus performante lorsque les objets à sélectionner sont rapprochés de l'utilisateur. Par contre, les objets éloignés seront difficiles à pointer à cause de l'amplification du bruit des mouvements de la main avec la distance.

1.2.6.1.2 Pointage à deux mains

Cette technique ressemble beaucoup au *ray-casting* à l'exception près que les deux mains sont utilisées pour définir la droite qui intercepte les objets. La main située la plus près de l'utilisateur définit l'origine du rayon. En supposant que la main droite est la plus rapprochée, l'équation de la droite $p(\alpha)$ devient :

$$p(\alpha) = h_d + \alpha(h_g - h_d), \quad 0 < \alpha < +\infty$$

Évidemment, cette technique impose l'utilisation de deux traqueurs pour chacune des mains. Cependant, de nouvelles fonctionnalités peuvent être introduites pour rendre la sélection plus flexible. Par exemple, la distance entre les mains pourrait définir la profondeur de sélection. Une autre approche utilise les rotations du poignet pour créer des courbes dans la droite qui intercepte les objets.

1.2.6.1.3 Lampe de poche et aperture

Les techniques de pointage avec un rayon possèdent un gros désavantage: la précision n'est pas suffisante pour les objets petits ou éloignés. La technique de la lampe de poche transforme le rayon virtuel du *ray-casting* en cône de sélection. Puisque plusieurs objets peuvent maintenant être sélectionnés simultanément, le problème devient alors d'éliminer cette ambiguïté. Deux règles sont utilisées. La première stipule que les objets les plus rapprochés du rayon central seront sélectionnés. Si une ambiguïté persiste avec des objets dont la distance est égale, la deuxième règle sélectionne l'objet le plus rapproché de la main. Malgré ces règles, il devient parfois difficile de sélectionner un objet particulier dans un groupe dense.

La technique d'aperture tente d'éliminer ce problème en permettant à l'utilisateur de changer le volume de sélection à tout moment. L'origine du cône est définie par la position des yeux de l'utilisateur e. La direction de pointage s'établit avec la main. La droite $p(\alpha)$ devient alors :

$$p(\alpha) = e + \alpha(h - e), \quad 0 < \alpha < +\infty$$

En changeant la distance entre les yeux et la main de l'utilisateur, on peut changer le rayon du cône dynamiquement. Une autre approche utilise la rotation du poignet associée à deux plaques qui bouchent le faisceau virtuel de la lampe de poche afin de limiter davantage le volume de sélection. Cette approche est illustrée à la Figure 1-21.

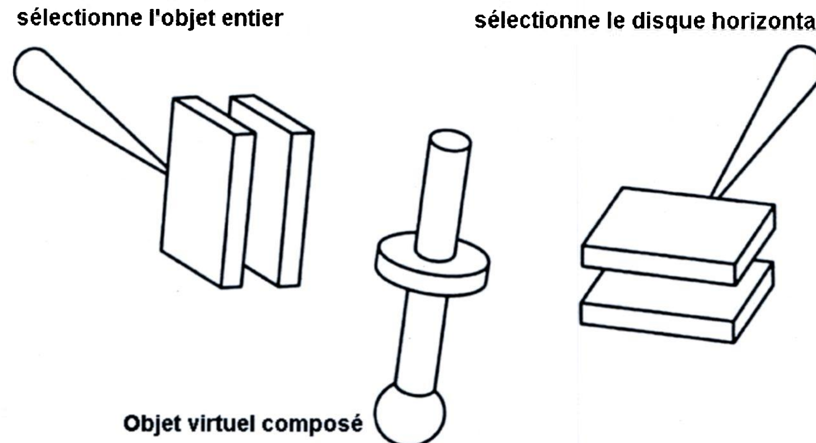


Figure 1-21: Approche des plaques pour la technique de l'aperture

1.2.6.1.4 Plan image

Cette technique simplifie la tâche de sélection en permettant à l'utilisateur de contrôler deux DOF seulement. La sélection et la manipulation des objets 3D s'effectuent sur leur projection 2D dans un plan image virtuel présenté devant l'utilisateur. Il existe deux variations. L'*approche sticky-finger* reprend le principe du *ray-casting* et trace un vecteur entre les yeux et la main de l'utilisateur. L'intersection avec un objet sur le plan image détermine la sélection. L'*approche head-crusher* demande à l'utilisateur de positionner le pouce et l'index pour saisir la projection de l'objet sur le plan image. Dans les deux cas, l'objet est miniaturisé et transporté près de l'utilisateur pour la manipulation. L'orientation est donc facile à modifier, mais le changement de position est moins intuitif à cause de la téléportation de l'objet.

1.2.6.1.5 Canne à pêche

Cette technique utilise un périphérique supplémentaire pour contrôler la longueur du rayon virtuel. Il est donc possible de choisir un objet particulier lorsque plusieurs sont intersectés par la droite. De plus, l'utilisateur pourra approcher et éloigner l'objet avec la nouvelle composante de la même manière qu'une canne à pêche. Cependant, puisque l'intégration des dimensions n'est pas complète, on observera une diminution de la performance.

1.2.6.2 Manipulation directe avec main virtuelle

Dans les techniques de manipulation avec main virtuelle, l'utilisateur déplace un curseur 3D dans l'espace afin de sélectionner les objets. Lorsque ce curseur intercepte l'élément désiré, l'utilisateur n'a qu'à lancer une commande pour fixer l'objet à la main. Ensuite, le changement de position et la rotation s'effectuent de façon naturelle. Pour relâcher l'objet, une autre commande doit être transmise à l'application. Afin de fournir un indice de profondeur supplémentaire lors de la sélection, la main virtuelle sera généralement transparente. Il existe deux techniques avec main virtuelle : la main virtuelle simple et la technique Go-Go.

1.2.6.2.1 Main virtuelle simple

Cette technique associe linéairement la position réelle $\vec{p}_r$ et la position virtuelle $\vec{p}_v$ de la main par un facteur d'échelle α. Les rotations isomorphiques sont la plupart du temps utilisées, mais certaines applications pourraient également utiliser un changement d'échelle pour la rotation. On a donc en général :

$$\vec{p}_v = \alpha \vec{p}_r, \; R_v = R_p$$

Cette méthode est évidemment très intuitive puisqu'elle reproduit les interactions quotidiennes de l'humain avec son entourage. Cependant, pour accéder à des objets en dehors de la portée des bras, il faudra utiliser une technique de navigation (voir section 1.2.7) ce qui réduira la performance.

1.2.6.2.2 Technique Go-Go

La technique Go-Go reprend le principe de la méthode précédente, mais permet une relation non-proportionnelle entre les positions réelle et virtuelle de la main à partir d'un certain seuil D. Si on définit les vecteurs $\vec{r}_r$ et $\vec{r}_v$ tels qu'illustrés dans la Figure 1-22, on peut utiliser la relation suivante pour établir la position de la main virtuelle :

$$\vec{r}_v = \begin{cases} \vec{r}_r & si \; \|\vec{r}_r\| < D \\ \vec{r}_r + \alpha(\|\vec{r}_r\| - D)^2 \vec{u}_r & sinon \end{cases}$$

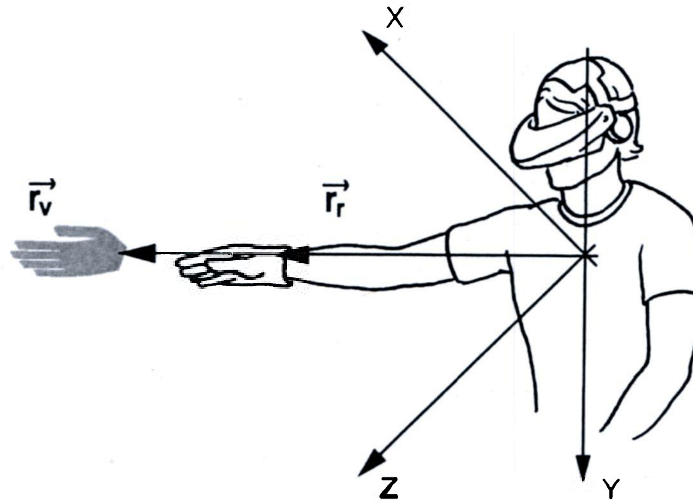


Figure 1-22: Technique Go-Go

Dans cette équation, le vecteur unitaire $\vec{u}_r$ donne la direction de $\vec{r}_r$. Lorsque le seuil D est dépassé, le bras virtuel s'allongera de façon non linéaire permettant à l'utilisateur d'atteindre des objets plus éloignés. Cependant, il est à noter que la portée est encore finie et qu'une métaphore de navigation sera tout de même nécessaire si l'objet dépasse la nouvelle limite du bras virtuel.

1.2.6.3 Monde miniature

Une alternative à l'allongement du bras virtuel est de créer une version miniature de l'environnement où l'utilisateur pourra sélectionner et déplacer la copie des objets « réels ». Dans la technique du monde miniature (WIM pour *World-In-Miniature* en anglais), on manipule donc les objets indirectement via leur représentation miniaturisée. Lors du rendu, il faudra utiliser des techniques spéciales pour éliminer les murs et les plafonds du modèle réduit qui pourraient boucher la vue afin de pouvoir voir à l'intérieur des salles. Une des principales limites de cette approche est la taille de l'environnement. Si le monde virtuel est trop gros, un facteur d'échelle correspondant réduira tellement la grosseur de la copie que les objets seront trop petits pour être manipulés facilement.

1.2.6.4 Combinaison de techniques

Puisque les techniques de manipulation présentées dans les sections précédentes ne sont pas adéquates dans toutes les situations, des combinaisons de ces méthodes ont été développées

afin de remédier à certains problèmes. Il existe deux approches pour combiner les techniques. Tout d'abord, l'agrégation permet à l'utilisateur de choisir la technique de sélection désirée selon la situation parmi un ensemble prédéterminé, une boîte à outils. L'autre approche, l'intégration des techniques, laisse plutôt le système changer lui-même la méthode de façon transparente selon le contexte de la tâche à effectuer. Chacune des étapes de la manipulation (sélection, positionnement, rotation) peut être réalisée avec une technique différente. Selon cette approche émergent trois techniques : HOMER, Scaled-world Grab et les poupées Voodoo.

1.2.6.4.1 HOMER

La technique HOMER (pour *Hand-centered Object Manipulation Extending Ray-Casting* en anglais) utilise d'abord le *ray-casting* pour sélectionner l'objet virtuel. Ensuite, au lieu d'attacher la cible au rayon, la main virtuelle est plutôt transportée à la position de l'objet et une manipulation directe peut être exécutée. Le mouvement de la main virtuelle $\vec{r}_v$ est alors amplifié par rapport au déplacement réel $\vec{r}_r$ en fonction des distances initiales de l'objet (D_o) et de la main (D_h) avec l'utilisateur lors de la sélection :

$$\vec{r}_v = \alpha_h \vec{r}_h, \quad \alpha_h = \frac{D_o}{D_h}$$

La rotation de l'objet virtuel se fait également de manière naturelle avec la main. Cependant, le changement de position est limité et dépend du facteur α_h défini lors de la sélection.

1.2.6.4.2 Scaled-world Grab

Dans cette technique, la sélection s'effectue avec la méthode du plan image. Lorsque cette opération est exécutée, la phase de manipulation utilise également la main virtuelle. Cependant, au lieu de changer l'échelle du déplacement de la main comme dans la technique HOMER, l'approche *Scaled-world Grab* va plutôt miniaturiser l'environnement par rapport au point de vue de l'utilisateur pour que l'objet virtuel désiré se trouve à sa portée. En considérant la distance D_v entre le point de vue et la main virtuelle et la distance

D_o entre ce même point de vue et l'objet virtuel sélectionné, le facteur d'échelle α_s recherché est:

$$\alpha_s = \frac{D_v}{D_o}$$

1.2.6.4.3 Poupées Voodoo

La technique des poupées Voodoo utilise deux mains pour la sélection et la manipulation des objets virtuels. Tout d'abord, la sélection s'effectue encore une fois grâce à la méthode du plan image. Lorsque l'objet est choisi, une copie miniature, appelée poupée, apparaît alors dans les mains de l'utilisateur et la manipulation est exécutée indirectement. Lorsque l'utilisateur tient la poupée dans sa main non-dominante, l'objet virtuel correspondant ne bouge pas. La main non-dominante définit donc le référentiel stationnaire de l'objet. Par contre, la main dominante effectue des changements de position et d'orientation relatifs au référentiel obtenu.

1.2.6.5 Rotations non-isomorphiques

Les techniques présentées jusqu'à maintenant ont toujours utilisé des rotations isomorphiques puisque cette approche est la plus naturelle pour la plupart des utilisateurs. Par contre, amplifier les rotations permet de diminuer l'effort et d'éliminer le *clutching*. À l'inverse, on peut également atténuer les rotations afin d'effectuer des mouvements précis. Si on représente l'orientation virtuelle à l'aide d'un quaternion q_v, on peut amplifier de manière absolue l'orientation du périphérique q_p par un facteur k selon la relation suivante :

$$q_v = q_p^{\,k}$$

On peut également amplifier une rotation de manière relative à partir d'une orientation de base q_0 avec cette équation :

$$q_v = (q_p q_0^{-1})^k q_0$$

Cette relation amplifie d'abord la rotation entre l'orientation du périphérique et celle de base. Puis, on applique la rotation amplifiée à q_0 pour obtenir l'orientation virtuelle désirée.

1.2.7 Métaphores de navigation

La tâche de navigation est définie comme une interaction avec l'environnement virtuel où l'utilisateur se déplace d'un point initial à une position cible ou dans la direction désirée. La navigation comprend deux aspects dont l'utilisateur doit tenir compte pour bien effectuer cette tâche : un aspect moteur et un aspect cognitif.

Tout d'abord, l'aspect moteur définit la façon dont l'utilisateur doit interagir avec l'environnement pour effectuer un déplacement virtuel quelconque. Cette sous-tâche est appelée le *travel* en anglais. Puisque le déplacement physique ne peut se faire que dans un espace restreint, des techniques ont été développées pour pouvoir effectuer des mouvements virtuels. Il est très important que ces techniques soient intuitives puisque la navigation est sans aucun doute la tâche la plus universelle dans les environnements virtuels. Il existe cependant une limite commune aux systèmes actuels qui simulent le mouvement : les indices visuels de mouvement sont fournis mais pas les indices vestibulaires. L'aspect moteur est celui qui représente le plus d'intérêt pour le projet car il englobe toutes les techniques présentées dans cette section ainsi que plusieurs autres mécanismes de locomotion dont la plate-forme NELI fait partie.

La portion cognitive de la navigation comprend plutôt la notion de compréhension de l'environnement. Pour y arriver, l'utilisateur doit utiliser les indices visuels qui lui sont fournis comme des cartes ou un compas et se déplacer dans le monde virtuel pour connaître les trajectoires entre les différents lieux et identifier les particularités de l'environnement (édifice ou paysage particuliers, couleurs, panneaux, etc.). Cette sous-tâche de la navigation, appelée *wayfinding* en anglais, permet au participant d'acquérir des connaissances spatiales pour construire une représentation mentale du monde virtuel qui l'entoure. Cette représentation mentale est appelée la *carte cognitive*. Lors de l'apprentissage, cette carte s'installe peu à peu dans la mémoire à long terme du participant et elle lui permet finalement d'établir un lien constant entre l'information visuelle reçue, sa

position et son orientation dans l'environnement ainsi que la prochaine opération de *travel* à effectuer.

Comme le *travel* est le seul aspect de la navigation qui sera étudié pour le projet, le terme navigation aura ici la même définition que ce dernier. Pour le reste du document, la navigation désignera donc la tâche motrice définie dans cette présente section où l'utilisateur doit interagir avec l'environnement virtuel pour s'y déplacer.

Les techniques de navigation permettent d'exécuter trois tâches de base : l'exploration, la recherche et la manœuvre.

Tout d'abord, l'*exploration* consiste à parcourir l'environnement sans but particulier. L'objectif est d'obtenir de l'information sur l'emplacement des objets dans le monde virtuel. Cette tâche s'effectue typiquement au début de l'interaction avec un nouvel environnement afin d'en découvrir les particularités. Les techniques d'exploration doivent être naturelles et intuitives pour que l'utilisateur puisse se concentrer sur l'apprentissage de son entourage et non pas sur l'exécution du mouvement.

La *recherche* implique plutôt un déplacement vers une cible précise dans l'environnement. On distingue deux catégories de recherche. La *recherche naïve* s'effectue lorsque l'emplacement de la cible ou la trajectoire vers celle-ci n'est pas connu. Il est à noter que l'exploration est une forme de recherche naïve extrême. Dans la *recherche amorcée* par contre, l'utilisateur connaît la position de la cible à atteindre et le chemin pour s'y rendre.

Finalement, la *manœuvre* concerne les petits mouvements précis qui sont effectués localement pour exécuter une tâche particulière. Par exemple, se cacher derrière un mur pour regarder de l'autre côté du coin de l'œil pour ne pas être vu n'est pas une tâche facile à exécuter avec n'importe quelle interface. Souvent un déplacement physique du corps et de la tête sera la façon la plus naturelle d'aborder ce genre de problèmes.

Il existe différentes caractéristiques pour classer les techniques de navigation. Tout d'abord, une méthode peut être *active* ou *passive*. Dans le mode actif, l'utilisateur contrôle directement le mouvement du point de vue. Lorsque la technique est passive, c'est plutôt le système qui contrôle automatiquement le mouvement selon le contexte.

On distingue également les tâches de navigation *physiques* ou *virtuelles*. Dans une méthode physique, l'utilisateur se déplace dans le monde réel afin de changer le point de vue dans l'environnement. Dans l'approche virtuelle, le corps du participant demeure fixe ou presque. Plusieurs applications utilisent une combinaison des deux.

Finalement, on peut classer les techniques selon la métaphore utilisée. Les prochaines sections présenteront ces différentes approches.

1.2.7.1 Techniques de locomotion physique

Ces techniques utilisent l'effort physique du participant afin de le transporter dans l'environnement. Elles imitent plus ou moins bien les méthodes naturelles de locomotion dans le monde réel. On distingue quatre approches : la marche normale, la marche sur place, les simulateurs de marche et les cycles.

1.2.7.1.1 Marche

Cette méthode est une approche naturelle et familière qui produit les indices vestibulaires du mouvement ce qui aide à comprendre davantage les proportions physiques de l'environnement. Cependant, à cause de l'espace souvent limité, la marche n'est souvent pas possible complètement. De plus, la longueur de fil des périphériques est une autre limitation. Par contre, les technologies sans fil sont prometteuses. La marche est une technique très utile pour la réalité augmentée où le participant doit se déplacer dans un environnement réel dans lequel des éléments virtuels sont ajoutés.

1.2.7.1.2 Marche sur place

La technique de la marche sur place est similaire à la précédente. Cependant, le problème d'espace est résolu puisque l'utilisateur n'a qu'à simuler la marche sans se déplacer. Cependant, les indices vestibulaires sont supprimés et l'utilisateur devra contrôler son mouvement ce qui rend la tâche moins naturelle. De plus, la marche sur place sera plus fatigante pour parcourir de longues distances virtuelles.

1.2.7.1.3 Simulateurs de marche

Ces appareils utilisent des technologies mécaniques diverses pour simuler le mouvement lors de la marche. Il existe actuellement des systèmes à tapis roulants bidirectionnels et d'autres à plates-formes articulées par des bras mécaniques. Cependant, ces dispositifs ne sont pas assez rapides pour bien suivre les déplacements de l'utilisateur. Bien que ces appareils soient plus ergonomiques que la marche sur place, le participant devra quand même généralement adapter son mouvement au système.

1.2.7.1.4 Cycles

Si l'exercice musculaire est désiré mais que la marche n'est pas nécessaire, on peut utiliser une métaphore de véhicule comme la bicyclette avec un mécanisme à pédales. Ces appareils seront moins coûteux que les simulateurs de marche et l'entraînement physique sera encore possible. Cette approche est parfois utilisée dans le domaine militaire.

1.2.7.2 Techniques de *steering*

Le *steering* implique le contrôle continuel de la direction du mouvement par l'utilisateur. Ces techniques de navigation virtuelles utilisent généralement l'orientation de certaines parties du corps de l'utilisateur pour déterminer le déplacement. On compte notamment la

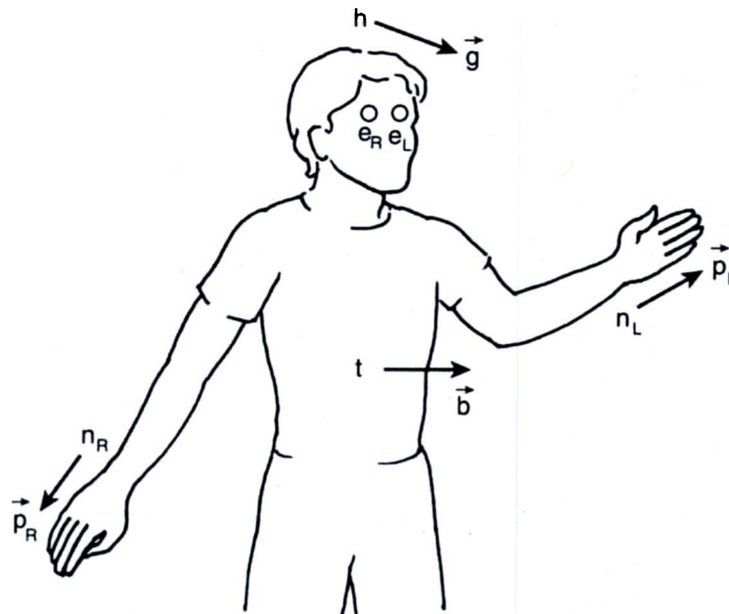


Figure 1-23 - Variables utilisées pour la définition des techniques de steering

direction par le regard, le pointage, la direction par le torse, la technique de la caméra en main, les *props* physiques, les contrôleurs de mouvement virtuel et le *steering* semi-automatisé. La Figure 1-23 montre les variables qui seront utilisées dans la définition des méthodes suivantes.

1.2.7.2.1 Direction par le regard

Cette technique utilise un traqueur de position sur la tête de l'utilisateur pour approximer la direction de son regard, le vecteur unitaire $\vec{g}$. La nouvelle position t_{new} du participant à chaque itération dépendra de $\vec{g}$, du temps écoulé depuis la dernière itération Δt et de la vitesse v qui peut être modifiée dynamiquement :

$$t_{new} = t + v \cdot \Delta t \cdot \vec{g}$$

On peut également ajouter des fonctionnalités à cette technique afin d'augmenter les directions permises à l'utilisateur lors des déplacements. Cependant, il faut faire attention si on veut donner la possibilité de « voler » dans l'environnement. En effet, si la direction verticale est continuellement affectée par le regard, le déplacement sur un plan horizontal devient difficile. Par contre, si elle change seulement aux orientations extrêmes de la tête (regarder en haut ou en bas), la technique devient vite inconfortable. De plus, la méthode de direction par le regard ne permet pas de regarder dans une autre direction pendant le déplacement.

1.2.7.2.2 Pointage

Similaire à la technique précédente, le pointage utilise un traqueur de position attaché à la main pour contrôler la direction vers laquelle l'utilisateur pointe, le vecteur unitaire $\vec{p}$:

$$t_{new} = t + v \cdot \Delta t \cdot \vec{p}$$

On peut aussi étendre cette méthode en utilisant deux mains. La distance entre les mains, normalisée par le facteur α, permettra de contrôler la vitesse :

$$t_{new} = t + v \cdot \Delta t \cdot \partial \cdot \frac{(n_R - n_L)}{\|n_R - n_L\|}$$

Cette technique est très appropriée pour l'exploration de l'environnement car l'utilisateur peut alors observer le monde virtuel tout en se déplaçant dans une autre direction.

1.2.7.2.3 Direction par le torse

La technique de la direction par le torse utilise cette fois le vecteur $\vec{b}$ pour déterminer la direction. Elle découple également le regard et le mouvement mais d'une manière plus naturelle puisque la tête sera tout de même un peu affectée par l'orientation du corps comme dans un déplacement réel. Cependant, cette technique ne s'applique qu'aux mouvements horizontaux car pointer vers le haut ou vers le bas avec le torse n'est pas une tâche aisée. De plus, déterminer la position du torse dans l'espace exige un traqueur supplémentaire sur la poitrine ou à la taille de l'utilisateur pour dissocier le torse de la tête.

1.2.7.2.4 Caméra en main

Dans cette technique, la position et l'orientation du point de vue sont directement contrôlées par la main ($\vec{n}$ et $\vec{p}$). Bien que le contrôle du déplacement devienne alors très flexible, cette méthode n'est par contre pas la plus intuitive puisque la main perçue d'un point de vue troisième personne par le participant lui présentera une scène 3D à la première personne.

1.2.7.2.5 Props physiques

Cette approche utilise des objets physiques appelés *props* en RV pour simuler une méthode de déplacement qui existe dans le monde réel. Par exemple, un simulateur d'avion utilisera un manche à balai et un simulateur de voiture, un volant et des pédales. Ces appareils seront souvent spécialisés pour une application particulière et l'utilisateur devra parfois être assis pour les manœuvrer. D'autres périphériques comme la *wand* du système IS-900, présentée à la Figure 1-24, sont plutôt génériques et s'apparentent simplement à des manettes de jeu conventionnelles. Malgré leur flexibilité, ces accessoires n'apportent pas une interaction très réaliste avec le monde virtuel.

Figure 1-24: Wand du système IS-900 d'InterSense Inc.

1.2.7.2.6 **Steering** *semi-automatisé*

Cette technique est grandement utilisée dans les attractions virtuelles où le système déplace automatiquement le participant vers un but éventuel tout en lui permettant un certain contrôle. La métaphore utilisée est celle d'une chaîne dont l'une des extrémités est attachée à l'utilisateur et l'autre, à une ancre dont la trajectoire est définie par l'application. L'utilisateur peut alors profiter du voyage pour observer l'environnement avec une certaine liberté de mouvement.

1.2.7.3 Techniques de planification de route

Ce type de techniques permet à l'utilisateur de spécifier d'abord la route dans l'environnement et ensuite de se déplacer en suivant le chemin défini. C'est un processus en deux étapes : l'utilisateur planifie puis le système exécute le plan. Pour augmenter la flexibilité, ces techniques peuvent également permettre un changement dynamique de la trajectoire au fur et à mesure que le déplacement évolue. On distingue trois approches : le dessin de trajectoire, les points de relais et la manipulation d'une représentation de l'usager.

1.2.7.3.1 Dessin de trajectoire

Dans cette approche, l'utilisateur doit tracer une ligne continue qui définit la trajectoire dans l'environnement. On suppose généralement que la caméra se déplacera à une certaine

hauteur au dessus du sol. Des algorithmes plus sophistiqués qu'une simple projection du trait en 2D sur la scène utilisent la forme de la courbe et la géométrie des éléments du monde virtuel pour maximiser la continuité de la trajectoire. Cependant, une méthode pour dessiner doit être développée dans l'application pour effectuer le tracé.

1.2.7.3.2 Points de relais

La technique des points de relais permet plutôt à l'utilisateur de définir seulement quelques points de la trajectoire. Le système s'occupe ensuite de trouver la route continue. La méthode est beaucoup plus simple pour le participant, mais la définition du trait par l'application peut varier en complexité selon l'algorithme utilisé. Au plus simple, des segments droits sont tracés entre les points de relais. Cependant, la définition de courbes permettra un déplacement plus naturel, plus fluide et potentiellement plus optimal.

1.2.7.3.3 Manipulation d'une représentation de l'usager

Avec cette approche, l'utilisateur manipule une version miniaturisée de lui-même pour décrire une trajectoire avec une technique WIM (voir section 1.2.6.3). Il enregistre d'abord le déplacement, puis le système l'exécute. Le principal avantage de cette méthode est que l'utilisateur peut également définir l'orientation du point de vue lors de la définition du déplacement et pas seulement la position comme avec les deux approches précédentes.

1.2.7.4 Techniques *Target-Based*

Dans ce genre de techniques, le but de l'utilisateur est de définir seulement le point d'arrivée lors d'un déplacement. La téléportation n'est alors pas la bonne solution puisque le changement brusque de position désorientera le participant. Le déplacement en continu est généralement recommandé. La trajectoire vers la cible doit donc être déterminée par le système. Deux techniques utilisent cette approche : la spécification de cible sur une carte et la méthode *ZoomBack*.

1.2.7.4.1 Spécification de cible sur une carte

Cette technique simple incorpore à l'interface une carte ou un WIM de l'environnement avec lequel l'utilisateur peut définir une destination cible. Une fois cette position identifiée, l'application génère la trajectoire que le point de vue suivra ensuite automatiquement.

1.2.7.4.2 ZoomBack

L'approche *ZoomBack* utilise une métaphore de *ray-casting* pour sélectionner un objet duquel l'utilisateur peut s'approcher ou s'éloigner par la suite. La position de départ est également enregistrée afin de permettre un retour rapide à la trajectoire initiale. Cette technique est donc appropriée pour observer temporairement un objet de plus près pour ensuite retourner facilement à la tâche précédente.

1.2.7.5 Techniques utilisant les métaphores de manipulation

Ces techniques utilisent une métaphore de manipulation avec main virtuelle comme HOMER et Go-Go pour manipuler le point de vue au lieu d'objets virtuels. Elles seront très efficaces si l'application utilise les mêmes méthodes pour la manipulation d'objets car un simple changement de mode permettra à l'utilisateur d'effectuer les deux opérations de la même manière. On distingue deux approches : *grabbing the air* et la manipulation d'objets fixes.

1.2.7.5.1 Grabbing the air

Dans cette approche, le monde virtuel en entier est considéré comme un objet manipulable. Lorsque l'utilisateur exécute le geste d'attraper quelque chose n'importe où dans l'espace, l'environnement est déplacé pendant que le point de vue reste fixe. La nouvelle position w du monde correspond à la translation de la main n :

$$w = w_{past} + (n - n_{past})$$

Cette technique demande par contre beaucoup de mouvements des bras. On peut l'améliorer en utilisant deux mains (métaphore de tirer sur une corde) ou en permettant une extension du bras pour parcourir plus de distance avec la méthode Go-Go (voir section 1.2.6.2.2).

1.2.7.5.2 Manipulation d'objets fixes

Pour cette technique, l'utilisateur choisit d'abord un objet qui servira ensuite d'ancre pour déplacer le point de vue. La force appliquée avec la main sur l'objet fixe permet donc un mouvement relatif autour de celui-ci. Puisque la tâche de manipulation des objets est très similaire à celle du point de vue, l'utilisateur devra cependant porter une attention particulière au mode de sélection en cours.

1.2.7.6 Techniques *Travel-by-Scaling*

Cette méthode permet à l'utilisateur de changer l'échelle du monde virtuel pour qu'un pas dans l'environnement physique soit égal à une plus grande distance virtuelle. Ainsi, le participant peut utiliser la technique naturelle de la marche dans un espace restreint pour parcourir des kilomètres virtuels. Cependant, il faut pouvoir fournir à l'usager une référence de la grandeur réelle de l'environnement. Pour ce faire, un avatar est généralement inclus dans la scène pour représenter sa taille. Cette technique souffre également d'un manque de précision pour les déplacements si la distance parcourue à chaque fois est grande. Enfin, le changement d'échelle répété de l'environnement peut augmenter les effets du *cybersickness*[1]. Cette méthode devrait donc être utilisée avec modération.

1.2.7.7 Techniques pour l'orientation du point de vue

La plupart des techniques présentées précédemment permettent uniquement de déplacer la position du point de vue. Le changement de l'orientation est également important pour pouvoir observer la scène virtuelle sous tous ses angles. On distingue trois méthodes : le *tracking* de la tête, la vue orbitale et la rotation non-isomorphique.

---

[1] Cybersickness : Décrit l'ensemble des malaises pouvant être ressentis lors d'une expérience virtuelle. Plusieurs facteurs influencent le cybersickness dont la contradiction entre l'accommodation et la convergence mentionnée plutôt, le manque d'indice vestibulaire lors de la navigation, un taux de rafraîchissement peu élevé, etc. Les symptômes varient d'un participant à l'autre et dépendent des facteurs en cause. On compte notamment les maux de tête, les nausées et les vomissements. Certains participants n'en ressentiront tout simplement pas les effets.

1.2.7.7.1 **Tracking** ***de la tête***

Cette approche est probablement la plus efficace et la plus naturelle de toutes. Un traqueur est installé sur la tête de l'utilisateur et l'orientation est définie directement. Cette méthode offrira au participant un meilleur sens de l'orientation que les virages virtuels.

1.2.7.7.2 Vue orbitale

Cette méthode utilise également un traqueur et permet de déplacer le point de vue autour d'un objet en changeant simplement l'orientation de la tête. Par exemple, si l'utilisateur veut regarder l'objet d'en haut, il regarde en bas. S'il veut le regarder de la gauche, il regarde à droite. Cette technique est moins naturelle, mais elle permet l'étude locale d'un objet dans l'environnement avec peu d'effort.

1.2.7.7.3 Rotation non-isomorphique

Cette technique utilise un traqueur attaché à la taille de l'utilisateur. Elle permet de régler le problème du mur manquant dans les systèmes d'affichage à écrans *surround* tel que celui utilisé pour le projet. Lorsque le participant pivote, l'angle de rotation virtuel θ_v doit correspondre à l'angle physique θ_p si celui-ci est petit. Lorsque l'angle physique dépasse un certain seuil s, la rotation physique θ_p est amplifiée par un facteur $(1+\phi)$ pour obtenir la rotation virtuelle θ_v. Ainsi, l'utilisateur pourra voir la scène virtuelle tout autour de lui sans devoir faire face au mur absent. Donc, en résumé :

$$\theta_v = \begin{cases} \theta_p, & \text{si } \theta_p \leq s \\ \theta_p \cdot (1+\phi), & \text{sinon} \end{cases}$$

Pour définir le paramètre ϕ qui affecte l'angle de rotation virtuel, on utilise généralement une fonction gaussienne puisque la courbe qu'elle définit est toute indiquée pour le type d'opération que l'on veut effectuer sur la rotation virtuelle:

$$\phi = f(\theta_p) = e^{-\frac{(|\theta_p| - \pi/2)^2}{\sigma_1^2}}$$

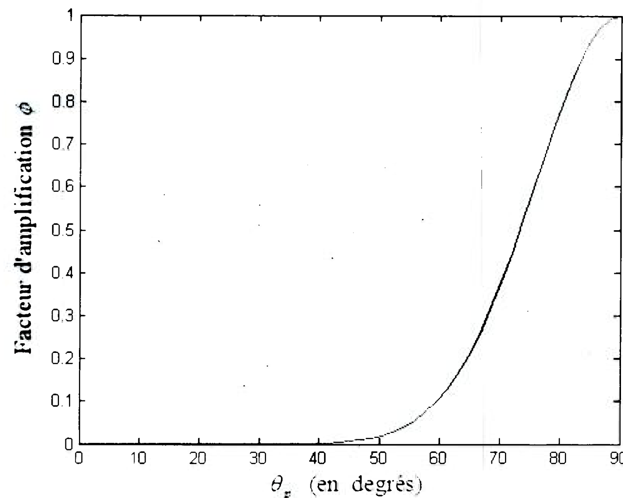


Figure 1-25: Exemple de fonction gaussienne pour une rotation virtuelle non-isomorphique

Cette équation, tient compte de l'orientation θ_p de la taille de l'utilisateur. À $\theta_p = \pm\pi/2$, l'utilisateur fait face aux écrans de côtés et la rotation virtuelle sera donc maximale ($\phi = 1, \theta_v = \pm\pi$). Le facteur σ_1 permet de déterminer la forme de la courbe gaussienne. Une valeur inférieure à 1 sera généralement souhaitable afin d'augmenter graduellement la valeur de ϕ au début et d'accélérer davantage au fur et à mesure que la valeur de la rotation physique augmente.

Un exemple d'une telle courbe avec $\sigma_1 = 0{,}35$ est présenté à la Figure 1-25. On voit que la courbe monte d'abord lentement au début puis accélère par la suite avec l'augmentation de l'angle physique pour finalement converger plus lentement vers la valeur maximale ($\phi = 1$) aux environs de $\theta_p = \pm\pi/2$. Dans cet exemple, la valeur du seuil s devrait être fixée entre 0 et $\pi/9$ (40 degrés) pour être conséquent avec la forme de la courbe et ainsi éviter l'augmentation discontinue de la valeur de la rotation virtuelle.

Une forme plus complexe de cette équation existe également pour tenir compte de la distance d de l'utilisateur avec l'arrière du système. En effet, plus le participant avancera vers l'avant, plus la valeur limite de θ_p avant de faire face à l'écran manquant sera grande. Cependant, un nouveau facteur L devra également être ajouté à l'équation afin de normaliser la distance. Ce facteur arbitraire aura également un impact sur la forme de la

courbe. À la limite, lorsque $L \to \infty$, on retrouve la forme simplifiée de l'équation présentée plus tôt :

$$\phi = f(\theta_p) = e^{-\frac{(|\theta_p| - \frac{\pi(1+d/L)}{2})^2}{\sigma_1^2}}$$

1.2.7.8 Techniques de spécification de vitesse

Certaines techniques de navigation permettent de changer dynamiquement la vitesse de déplacement. En effet, une vitesse constante sera toujours trop lente pour certaines situations et trop rapide pour d'autres. L'approche *lean-based* est adéquate lorsque des méthodes basées sur le regard sont utilisées. La vitesse peut alors être déterminée selon la distance de la tête par rapport au corps en pliant le cou vers l'avant. On peut également utiliser la distance de la main par rapport au corps dans les techniques de pointage. Des *approches discrètes* utilisent deux boutons sur une manette : un pour accélérer et l'autre pour freiner. Enfin, un *contrôle automatisé* de la vitesse peut être associé aux techniques *Target-Based.* Dans cette approche, lors d'un nouveau déplacement, le système accélérera lentement au départ jusqu'à une vitesse de croisière constante. À l'approche de la cible, le mouvement ralentira légèrement jusqu'à l'arrêt.

Évidemment, l'ensemble des méthodes et des technologies présentées dans cette section n'a pas été intégré en totalité dans le cadre du projet. Cependant, cette brève revue de littérature permet de présenter les différents aspects de la RV et d'introduire l'étendue de ce jeune domaine de recherche. La section suivante présente une description sommaire du projet qui sera défini plus en détails dans les chapitres suivants.

1.3 Description du projet

Le prototype du mécanisme à câbles étant en cours de développement, le projet présenté propose donc de développer un environnement virtuel servant à la fois d'interface de test et de simulateur pour le système de locomotion. L'application finale doit posséder les fonctionnalités suivantes qui définissent le sujet des prochains chapitres de ce mémoire.

Tout d'abord, afin de rendre l'application flexible et extensible, une approche modulaire doit être employée. Dans le projet actuel, on définit un module comme étant une représentation du monde virtuel qui encapsule des fonctionnalités et des propriétés spécifiques (synchronisation, intégration, visualisation, haptique[1], émission sonore, réseau, etc.) et qui partage des propriétés communes avec les autres modules (position, orientation, facteurs d'échelle, symétrie, dimensions, etc.). Chaque module possède une boucle de traitement indépendante des autres qui peut s'exécuter en parallèle et/ou être contrôlée par un module d'intégration central. Puisque l'accès aux propriétés communes sera effectué de façon asynchrone par les différents modules, un processus de synchronisation devra être mis en place pour assurer la cohérence entre les modules. Cette fonctionnalité étant l'objectif primaire du projet, le chapitre 2 définit d'abord les notions de l'exécution en parallèle et présente le module de synchronisation développé afin d'assurer la cohérence entre les modules du système final.

Suite au développement du processus de synchronisation, plusieurs modules ont été développés afin d'étendre les fonctionnalités sensorielles de l'application. Premièrement, il est impératif de produire une représentation visuelle de la scène afin de permettre à l'utilisateur de s'y retrouver facilement et de percevoir le résultat de la marche simulée. Le chapitre 3 présente les notions du rendu graphique permettant l'ajout de propriétés visuelles à l'environnement virtuel. Cette représentation visuelle forme un module où les objets possèdent des couleurs, des textures et des formes. Ensuite, (duTremblay à venir) présente un module haptique qui permet l'intégration de propriétés physiques dans l'environnement. Ainsi, il sera possible de détecter les collisions et de simuler l'effet des forces pour la préhension et la manipulation des objets virtuels de façon naturelle avec les mains. D'autres modules sensoriels permettant la diffusion de sons, de goûts et d'odeurs pourront également être ajoutés au système, mais ces derniers n'ont pas été développés pour le projet.

Ensuite, l'environnement virtuel doit être distribué sur un réseau. Le processus de communication utilisé doit permettre de rendre l'application multi-usagers et de distribuer les modules sur plusieurs machines afin d'augmenter davantage le parallélisme de l'exécution. Avec une approche distribuée, il sera possible d'utiliser plusieurs plates-formes

[1] Haptique : qui se rapporte au sens du toucher et à l'application de forces physiques.

à câbles situées dans des lieux physiques différents, mais prenant part au même environnement virtuel. Le chapitre 4 présente ce mécanisme de distribution réseau qui forme en fait un autre module intégré à l'environnement virtuel final.

Finalement, l'application doit intégrer tous les modules disponibles et aussi offrir une interface utilisateur de haut niveau qui soit intuitive et qui permette de configurer l'application et de simuler le système de locomotion avec d'autres périphériques de réalité virtuelle. Les fonctionnalités de l'application finale seront présentées au chapitre 5. Avant de conclure, le chapitre 6 présentera les résultats des différents tests de performance ainsi qu'une revue des propriétés de l'environnement développé.

En résumé, l'environnement virtuel cible doit posséder une représentation visuelle et offrir la possibilité d'y introduire de nouvelles propriétés sous forme de modules. De plus, le tout doit être distribué sur un réseau afin que plusieurs utilisateurs puissent interagir entre eux en partageant un monde virtuel commun. L'intégration des modules dans l'application finale forme un système complet appelé `VirtualUniverse`. La structure générale de l'environnement virtuel développé dans ce mémoire est présentée à la Figure 1-26.

Le diagramme présente l'ensemble des modules qui ont été intégrés dans le projet : module d'intégration, module de synchronisation, module visuel, module haptique et module de distribution. Chaque module définit sa propre classe d'objet spécialisée qui représente une propriété sensorielle ou fonctionnelle particulière d'un objet virtuel. Lors de la création d'un objet virtuel, ce dernier se voit attribuer un certain nombre de propriétés sensorielles et fonctionnelles. Les objets correspondants seront créés dans les différents modules et seront associés avec l'objet virtuel via le module de synchronisation afin de partager les données communes de position, d'orientation et de facteurs d'échelle.

Tout d'abord, le module de synchronisation présenté au chapitre 2 forme le noyau du module d'intégration. Situé au centre du diagramme de la Figure 1-26, il interconnecte les objets des différents modules afin de partager entre eux les données communes de façon sécuritaire et cohérente. Le module de synchronisation définit la classe `InterModuleObject` (IMO) qui représente un verrou central protégeant l'accès à un tampon partagé `InterModuleBuffer` (IMB) qui contient les données communes de

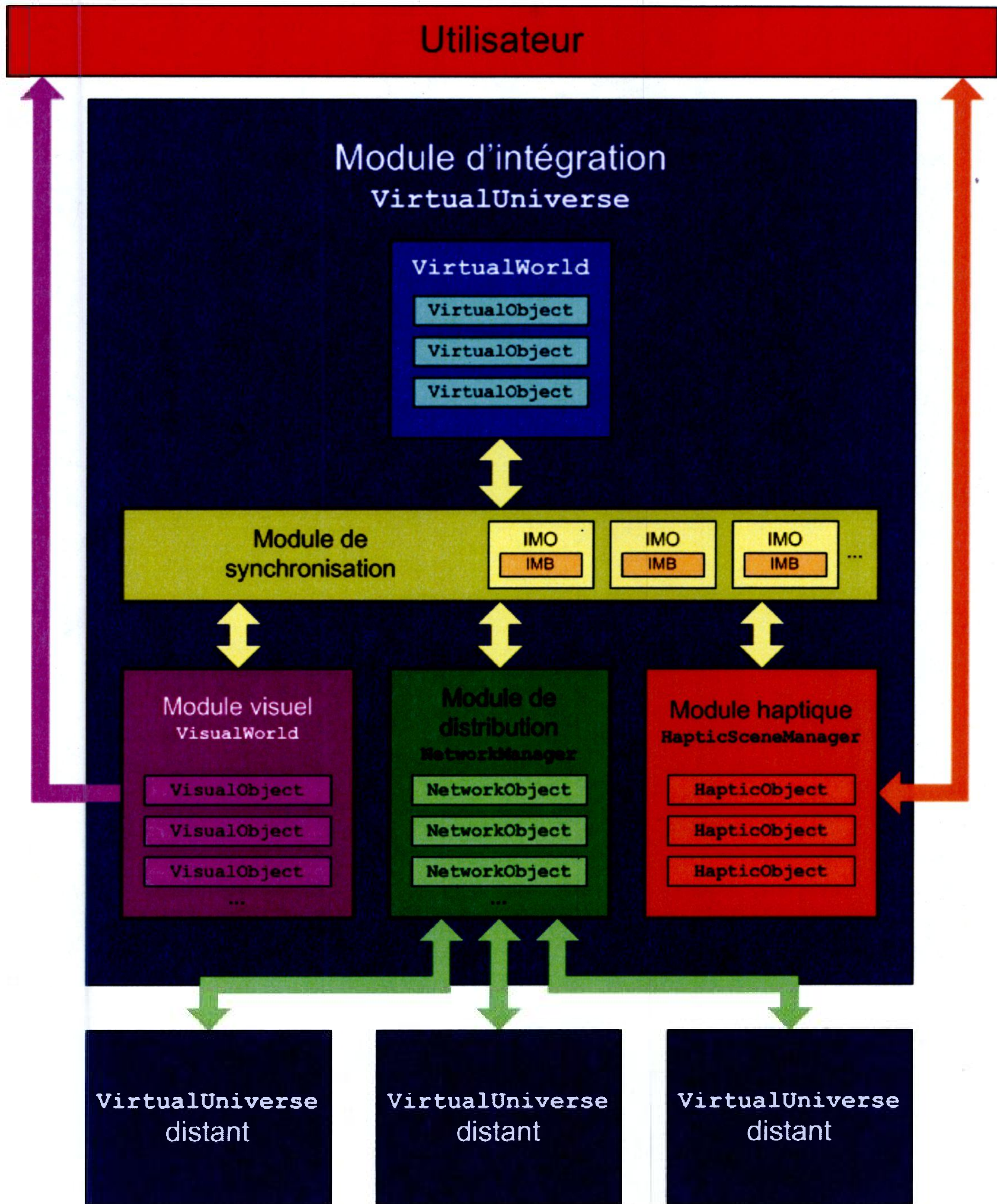


Figure 1-26: Structure générale de VirtualUniverse

position, d'orientation et de facteurs d'échelle. Les objets des autres modules partageant ces données devront faire des requêtes à l'IMO pour accéder au tampon IMB pour la lecture et l'écriture.

Ensuite, les modules visuel et haptique remplissent les fonctionnalités sensorielles du système. Ces deux modules permettent de fournir une rétroaction sensorielle à l'utilisateur sous forme d'affichage graphique pour le module visuel et par retour de force avec le module haptique. En plus de la rétroaction sensorielle, le module haptique permet également d'interagir sur le système avec une main virtuelle en manipulant les différents objets virtuels contenus dans la scène. Les modules visuel et haptique possèdent leur représentation respective de l'objet virtuel qui devra être synchronisée entre eux et avec le reste du système. Le module visuel définit la classe `VisualObject` qui encapsule les géométries et les paramètres visuels de l'objet affiché à l'écran. Le module haptique définit quant à lui la classe `HapticObject` qui calcule les collisions avec les autres objets haptiques de la scène et qui pourra être manipulé par l'utilisateur. Le module visuel sera présenté au chapitre 3. Plus de détails sur le module haptique sont également fournis dans (duTremblay à venir).

Puis, le module de distribution présenté au chapitre 4 permet la communication entre plusieurs instances de `VirtualUniverse` pour former une application multi-usagers ou pour distribuer les modules d'une application mono-utilisateur sur le réseau afin d'augmenter le parallélisme du système. Dans la Figure 1-26, le module de distribution permet au module `VirtualUniverse` principal (serveur) de communiquer avec les instances clientes (représentées par les trois blocs « Remote VirtualUniverse »). Le terme « instance » sera toujours utilisé tout au long de ce document pour désigner une instance réseau de l'application finale formant un processus indépendant démarré sur une machine pouvant potentiellement communiquer avec d'autres machines d'un même réseau.

Il est à noter que chaque instance (serveur ou client) de l'univers virtuel forme en fait une application complète qui charge ses propres modules localement. Par conséquent, les modules chargés ne seront pas nécessairement les mêmes sur chacune des instances d'un même système. En effet, comme on le verra plus loin, mis à part les modules d'intégration et de synchronisation qui forment le cœur du système, chacun des autres modules est facultatif au bon fonctionnement de l'application. Selon les fonctionnalités désirées et la topologie du système, il sera possible de déterminer les modules à charger avant le démarrage de l'expérience virtuelle. Une instance de l'univers virtuel ne possède pas

nécessairement d'utilisateur. Dans ces circonstances, elle aura généralement la responsabilité de gérer certaines fonctionnalités sensorielles pour l'ensemble du réseau. Suite au développement de l'application finale, il sera également possible d'étendre les fonctionnalités du système en y introduisant de nouveaux modules développés selon les principes de synchronisation et la structure flexible de l'architecture qui seront présentés tout au long de ce document.

Le module de distribution représente les objets virtuels à l'aide d'une structure XML encapsulée dans la classe `NetworkObject`. Ainsi, grâce à cette structure en format texte facile à manipuler, il sera aisé de changer les paramètres d'un objet et de distribuer cette mise à jour aux autres instances sur le réseau.

Finalement, le module d'intégration `VirtualUniverse` présenté au chapitre 5 encapsule les fonctionnalités du module de synchronisation pour intégrer et synchroniser l'ensemble des modules et fournir une interface permettant la configuration du système. De plus, le module d'intégration définit la classe `VirtualObject` qui représente l'objet virtuel auquel est associé l'ensemble des objets sensoriels et fonctionnels provenant des autres modules. Pour cela, la classe `VirtualObject` dérive simplement de la classe `InterModuleObject` afin d'encapsuler la gestion de la synchronisation entre les objets des différents modules. Finalement, des classes héritant de `VirtualObject` permettent également l'ajout de comportements de haut niveau aux objets virtuels et l'utilisation de métaphore de navigation et de manipulation. Ces fonctionnalités seront exécutées pendant la boucle d'exécution du module d'intégration.

Les principaux défis techniques du projet portent sur la synchronisation et la cohérence entre les modules de même que sur la robustesse de chacun. En effet, chaque module effectuant sa mise à jour de façon indépendante des autres, les principes de l'exécution en parallèle devront être intégrés au système afin d'assurer une cohérence complète. De plus, d'autres modules devront être développés et intégrés à l'environnement pour fournir de l'information multi-sensorielle à l'utilisateur et permettre la synchronisation sur un réseau. Le bon fonctionnement du simulateur de marche dépendra de la qualité de la synchronisation entre les différents modules et de la performance de chacun d'eux.

Finalement, le module d'intégration `VirtualUniverse` devra être ajouté au système afin de gérer le fonctionnement de l'ensemble des modules, de fournir une interface utilisateur au participant et de produire un comportement plus complexe dans l'environnement virtuel.

Chapitre 2 - Synchronisation des modules

L'une des caractéristiques essentielles que doit posséder l'application finale est d'être modulaire. De façon générale, cet objectif est atteint en séparant et en encapsulant les différentes fonctionnalités d'un système complexe à l'intérieur de sous-systèmes plus spécialisés appelés *modules*. Pour chaque module, les entrées, les sorties et les interactions avec le reste du système sont définies clairement afin de favoriser l'intégration avec les autres modules.

Comme mentionné plus tôt dans la description du projet, l'environnement virtuel contient plusieurs modules qui doivent s'exécuter en parallèle lors d'une simulation afin de fournir différentes fonctionnalités à l'utilisateur. Pour y arriver, un module de synchronisation basé sur les principes du *multi-threading* a été développé afin d'assurer la synchronisation entre tous les autres modules.

La prochaine section présente de manière formelle la définition d'un module qui sera utilisée tout au long du projet. Puis, les sections suivantes décriront les principes du *multi-threading* et le module de synchronisation développé qui se base sur ceux-ci.

2.1 Définition d'un module

Dans l'application développée, un module est défini comme un sous-système indépendant qui encapsule et exécute en parallèle une portion des différentes fonctionnalités du monde virtuel et des comportements des objets qui s'y trouvent : synchronisation, visualisation, manipulation, réaction aux lois physiques, distribution sur un réseau, navigation, émission sonore, etc. Un module génère et met à jour une liste d'objets partageant des propriétés communes qui doivent être synchronisées avec les objets des autres modules qui sont associés au même objet virtuel. Parmi ces propriétés, on compte notamment la position, l'orientation, des facteurs d'échelle, des descriptions, etc. Un niveau de priorité pour l'accès aux propriétés partagées peut être appliqué à un objet à l'intérieur d'un module pour lui donner une importance plus ou moins grande par rapport aux autres. Pour interagir avec l'utilisateur, chacun de ces modules utilise un média et une technique unique pour représenter la scène et interagir avec celle-ci : images, forces, sons, texte, etc.

Les modules peuvent être cruciaux ou optionnels au bon déroulement d'une simulation. Un sous-système est crucial lorsqu'un autre en dépend et utilise ses fonctionnalités. Ces modules forment généralement la couche logicielle de bas niveau qui sert à gérer l'exécution d'une application. À l'opposé, les modules optionnels définissent souvent les fonctionnalités plus abstraites qui forment la couche de haut niveau. Ces sous-systèmes permettent plutôt d'interagir avec l'environnement lors de l'exécution d'une application.

L'environnement virtuel développé dans le cadre de ce mémoire comprend plusieurs modules qui remplissent tous une fonctionnalité spécifique : synchronisation des modules, visualisation de la scène, gestion des forces, distribution sur un réseau et intégration des modules. Le module de synchronisation présenté plus loin dans ce chapitre utilise les principes de *multi-threading* afin de partager les propriétés communes entre les modules de façon sécuritaire. Ensuite, le module visuel présenté au chapitre 3 utilise les librairies OpenSceneGraph (OSG) et VR Juggler (VRJ) pour faire le rendu graphique sur plusieurs écrans. Puis, le module haptique, présenté dans (duTremblay à venir), se base sur Virtual Hand Toolkit (VHT) et sur l'engin physique Newton afin d'obtenir la numérisation complète des angles entre chacune des parties de la main pour déterminer la configuration de chacune des phalanges de pour ensuite sélectionner et manipuler les objets virtuels de façon quasi-naturelle dans l'espace. Ce module permet également la détection de collisions et l'application des lois de la physique aux éléments de l'environnement virtuel, qu'ils soient visibles ou non. Dans le même ordre d'idées, des modules auditif, olfactif et gustatif seraient également envisageables afin d'ajouter davantage de réalisme à l'environnement. En plus des fonctionnalités sensorielles, on peut introduire d'autres modules qui rempliront les tâches de distribution réseau et d'intégration. Ces deux derniers sous-systèmes seront présentés aux chapitres 4 et 5 respectivement.

Les modules sensoriels sont tous optionnels au bon déroulement d'une expérience virtuelle. En effet, bien qu'il faille au moins l'un d'entre eux pour que l'utilisateur puisse extraire de l'information de l'environnement, aucun n'est crucial à l'exécution de l'expérience virtuelle. Des objets virtuels sans propriétés sensorielles peuvent exister sans problème, mais aucune interaction ne sera possible avec ceux-ci. Il est également possible d'avoir un environnement où seulement quelques sens de l'utilisateur sont stimulés.

Le module de distribution réseau est également optionnel pour des raisons plus évidentes car une expérience virtuelle peut être locale ou distribuée. Une application distribuée met davantage en évidence le fait que les modules sensoriels soient optionnels. En effet, les instances de l'application peuvent alors se partager la gestion des différentes propriétés sensorielles de l'environnement. La distribution sur le réseau permet également à plusieurs utilisateurs de participer simultanément à l'expérience virtuelle.

Il n'y a donc que deux modules cruciaux pour l'exécution de l'environnement virtuel développé : le module de synchronisation et le module d'intégration. Le premier, présenté dans ce chapitre, encapsule le processus de partage cohérent d'information entre les différents modules basé sur les principes de *multi-threading*. Le second, présenté au chapitre 5, forme l'interface de haut niveau par laquelle l'utilisateur contrôle la création et la gestion des différents modules. C'est également à partir de ce dernier que d'autres fonctionnalités plus abstraites pourront être développées par la suite.

Qu'il soit crucial à l'exécution ou non, un module doit toujours être indépendant et s'exécuter en parallèle avec les autres. Il doit donc pouvoir former à lui seul une application complète qui n'a pas nécessairement à faire partie du système entier pour fonctionner. Ces applications serviront alors de plate-forme de test pour les différents modules. Dans le cas où un seul des modules optionnels est intégré au système, il peut s'exécuter de façon indépendante sans utiliser les principes de synchronisation et d'intégration alors inutiles.

Finalement, il est à noter que ce n'est pas le module qui possède des propriétés mais bien chacun des objets à l'intérieur de celui-ci. Par exemple, selon la fonctionnalité que remplit un objet particulier dans l'environnement virtuel, sa configuration ne sera pas toujours la même à l'intérieur d'un même module. Cependant, tous les objets d'un même module possèderont le comportement commun qui consiste à synchroniser les données partagées de position, d'orientation et de facteurs d'échelle avec les objets des autres modules. En fait, un module contient plusieurs objets aux propriétés spécifiques et les met à jour dans sa boucle d'exécution tandis que les objets forment les différentes portions sensorielles et fonctionnelles associées à un objet virtuel complet qui sont mises à jour par le module dans sa boucle d'exécution indépendante. La mise à jour du module concerne donc la mise à jour de toute la liste des objets créés par ce module tandis que la mise à jour d'un objet consiste

simplement en une étape de la mise à jour du module. C'est pour cette raison que les expressions « module » et « objet » seront utilisées sans distinction lors de la description du module de synchronisation et du processus de mise à jour qui seront présentés dans la dernière section de ce chapitre. Mais tout d'abord, la section suivante explique les principes du *multi-threading* qui ont servi de base au développement du module de synchronisation présenté par la suite.

2.2 Principes du *multi-threading*

Lorsqu'un ordinateur charge un programme en mémoire, les instructions contenues dans celui-ci sont exécutées une à une par le processeur. Certaines opérations arithmétiques élémentaires comme les additions et les multiplications sont effectuées en quelques cycles par la machine mais d'autres instructions comme les accès à la mémoire vive ou au disque dur sont beaucoup plus longues à traiter. Les principes du *multi-threading*, présentés dans les prochaines sections, permettront d'exploiter ces périodes d'attente afin d'optimiser l'application finale, quelle qu'elle soit.

2.2.1 Exécution parallèle avec les *threads*

Afin d'ajouter du parallélisme à l'exécution, on peut lui intégrer des *threads*. Un *thread* est une partie de code dans un programme qui forme généralement une boucle et qui, une fois activé, s'exécute « simultanément » avec les autres *threads* jusqu'à sa désactivation ou sa destruction. En fait, à moins d'avoir une machine munie de plusieurs processeurs, il est impossible avec la technologie actuelle d'exécuter deux opérations simultanément. Cependant, on peut augmenter grandement les performances globales d'une application en simulant ce parallélisme de deux façons. Premièrement, les *threads* doivent pouvoir accéder au processeur à tour de rôle pendant une courte période de temps afin d'exécuter partiellement leur boucle de traitement. Ainsi, tous les sous-systèmes de l'application évoluent environ au même rythme. Deuxièmement, un *thread* doit céder l'accès au processeur lorsqu'il doit attendre longtemps pour la réponse d'un périphérique quelconque. Par exemple, les accès à la mémoire, au disque dur ou à la carte réseau peuvent prendre plusieurs cycles où le processeur n'a rien à faire. On peut alors optimiser ces moments de repos en exécutant les autres *threads* qui ont besoin de faire des calculs immédiatement.

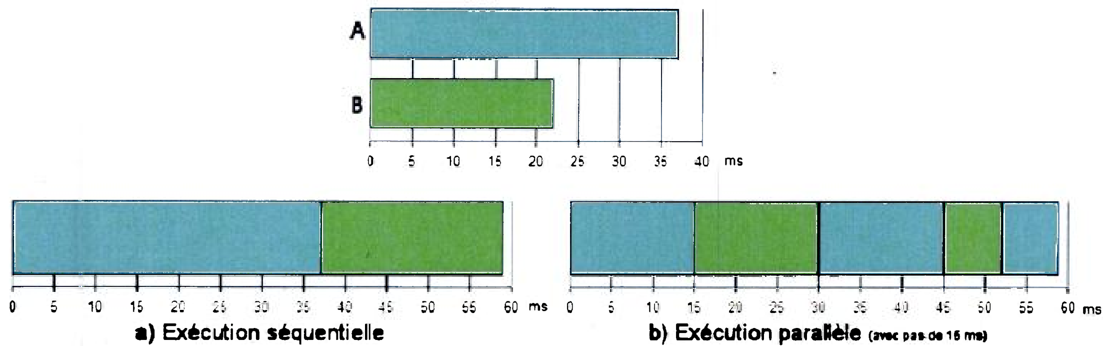


Figure 2-1: Exécution des threads selon la méthode round-robin avec calculs CPU rapides

Dans un même processus, il peut y avoir un ou plusieurs *threads*. C'est le système d'exploitation qui s'occupe de gérer l'ordonnancement des *threads* pour leur attribuer du temps de processeur. La méthode la plus simple pour ordonner l'exécution des *threads* s'appelle *round-robin*. Il suffit simplement de placer chaque *thread* dans une file d'attente. Puis, à intervalle régulier ou lorsqu'un *thread* doit attendre un périphérique, le système d'exploitation le fait passer à la fin de la file et donne l'accès au processeur au *thread* suivant dans la liste pour qu'il s'exécute.

Les figures Figure 2-1 et Figure 2-2 comparent les approches séquentielle et parallèle pour l'exécution de deux *threads* qui se terminent d'eux-mêmes (ne forment pas de boucle). Dans la première figure, on suppose que chacun d'entre eux ne fait que des calculs sur le processeur (zone bleue pour le *thread* A, zone verte pour le *thread* B). On constate qu'il n'y a pas de gain sur la durée malgré l'exécution en parallèle car il n'y a pas d'optimisation à faire pour des accès aux périphériques. Cependant, l'évolution de chacun des *threads* est mieux distribuée. Dans la seconde figure, on suppose cette fois que le *thread* A fait un accès à un périphérique quelconque (mémoire, disque dur, port, etc.) lors de son exécution (zone rouge). C'est lors des moments d'attente pendant l'accès à ces périphériques qu'on peut grandement optimiser l'utilisation du processeur et on constate nettement ce gain de performance dans la Figure 2-2.

Dans les exemples ci-dessus, on suppose que les *threads* se terminent d'eux-mêmes. Cette particularité fait en sorte qu'il est possible de comparer les approches séquentielle et parallèle. Cependant, dans le cas où chacun des *threads* formerait une boucle d'exécution

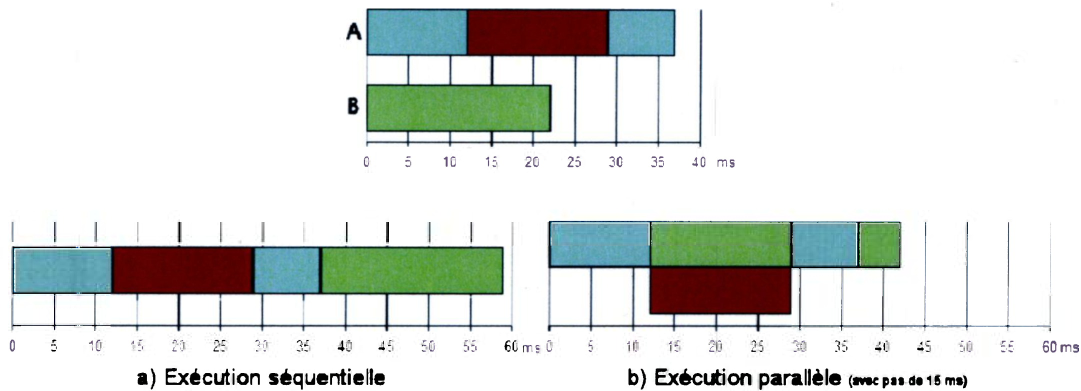


Figure 2-2: Exécution des threads avec accès au disque ou à la mémoire

indépendante qui ne se termine pas d'elle-même, il serait impossible d'utiliser l'approche séquentielle.

Pour le développement d'une application, il existe donc deux approches pour l'exécution d'un processus : l'approche *single-threading* et l'approche *multi-threading*. La nomenclature des deux techniques évoque clairement ce qu'elles impliquent.

La première approche forme la façon la plus simple de créer une application. Le *single-threading* correspond à l'approche séquentielle mentionnée plus tôt. Il ne peut alors y avoir qu'une seule boucle d'exécution car il n'y a qu'un seul *thread*. Cette caractéristique permet une gestion plus facile, mais c'est également sa principale limitation. En effet, l'exécution de la totalité de l'application se faisant de façon séquentielle, le temps d'attente pour l'accès aux périphériques sera perdu. Cette technique peut être acceptable dans le cas d'une application très simple.

L'approche *multi-threading* permet d'ajouter du parallélisme à l'application. Plusieurs *threads* sont exécutés à tour de rôle selon un ordonnancement quelconque défini par le système d'exploitation. On peut alors optimiser l'utilisation du processeur comme expliqué plus tôt. Cette technique devient rapidement plus complexe à mesure que le nombre de *threads* et de données qu'ils partagent entre eux augmente. En effet, puisque chaque *thread* n'exécute pas sa boucle nécessairement à la même vitesse que les autres, l'accès à l'information commune se fait de façon asynchrone et imprévisible.

C'est d'ailleurs le cas pour les modules intégrés à l'environnement virtuel développé dans ce projet. En effet, chaque sous-système est constitué d'un *thread* qui gère une fonctionnalité particulière de l'application comme la visualisation de la scène ou la gestion des forces. Cependant, les types de calculs impliqués pour chaque fonctionnalité n'étant pas les mêmes, le traitement de la boucle d'exécution prendra un temps variable et l'accès aux propriétés partagées, comme la position et l'orientation d'un objet virtuel, doit être protégé pour assurer la cohérence entre ces fonctionnalités.

C'est le *mutex*, qui remplira le rôle de protéger l'accès asynchrone aux données partagées entre plusieurs *threads*. Cette propriété essentielle du *multi-threading* forme la base du module de synchronisation présenté plus loin. La section suivante définit ce qu'est un *mutex* et en démontre l'utilité pour l'environnement virtuel développé dans ce projet.

2.2.2 Protection par les *mutex* pour l'accès asynchrone aux données partagées

Lorsque plusieurs *threads* partagent des propriétés (ou des données) communes, il faut absolument éviter les situations où plusieurs d'entre eux accèdent « simultanément » à l'information. En effet, puisque l'exécution leur est attribuée rapidement à tour de rôle par le système d'exploitation, l'accès aux données se fait de façon asynchrone et imprévisible si aucun processus de protection n'est mis en place. Les opérations de lecture et d'écriture peuvent donc être interrompues à tout moment et des données incohérentes ou incomplètes seront parfois générées. Pour les modules de l'application développée qui partagent l'information de position et d'orientation sous forme d'une matrice à seize paramètres comme celle présentée dans la section 1.2.2, il se pourrait qu'une modification du contenu de cette matrice soit interrompue par un accès en lecture et que seules quelques composantes aient été changées. Dans ces situations, la matrice résultante sera incohérente car la mise à jour n'aura été effectuée que partiellement.

Afin d'assurer la cohérence lors des modifications, il est donc impératif que les accès aux données partagées en lecture et en écriture s'effectuent en entier et de façon exclusive pour chaque *thread*. Pour y arriver, on utilise le *mutex*. On définit cette propriété du *multi-threading* comme un verrou logiciel qui permet de limiter l'accès à une portion de code à

un seul *thread* à la fois. Ainsi, avant d'effectuer des accès asynchrones sur les données partagées, chaque *thread* devra d'abord verrouiller le *mutex*. Cette étape implique deux opérations. Tout d'abord, il faut vérifier si le verrou est déjà activé par un autre *thread*. Si c'est le cas, il faudra attendre qu'il soit libéré. Une fois déverrouillé, le *thread* en question prend possession du *mutex* et accède ensuite de façon exclusive et sécuritaire aux données partagées. Une fois le traitement terminé, il faut s'assurer de libérer le *mutex* afin d'en permettre l'accès à d'autres *threads*.

La Figure 2-3 compare les approches avec et sans *mutex* pour l'accès en lecture et en écriture pour deux *threads*. Les périodes en bleu et en vert représentent les moments où il n'y a que des calculs sur le processeur pour les *threads* A et B respectivement. Il n'y a alors aucun accès à l'information partagée. Les zones jaunes illustrent les opérations de lecture et les rouges, les opérations d'écriture. Les opérations partielles sont reliées par des pointillés pour illustrer leur continuité malgré l'arrêt momentané dû au changement de *thread* par le système d'exploitation. On constate rapidement que l'approche sans *mutex* introduit plusieurs incohérences dues aux accès interrompus aux données communes. Cependant, les opérations de protection de l'accès asynchrone avec *mutex* sont entières mais elles imposent des délais d'attente qui rendent le temps d'exécution globalement plus long. En effet, dans l'exemple avec *mutex* de la Figure 2-3, on voit que le *thread* B est retardé de 5 ms au début pendant la fin de l'écriture du *thread* A. Le *thread* B ne fait alors plus rien et attend simplement que le *thread* A libère le *mutex*. Dans ces brèves séquences, l'approche avec *mutex* semble plus performante. Cependant, dans un programme plus complexe où plus de deux *threads* accèdent à des données partagées à des moments plus aléatoires, les périodes d'attente sont plus fréquentes et la durée globale des boucles d'exécution de chaque *thread*

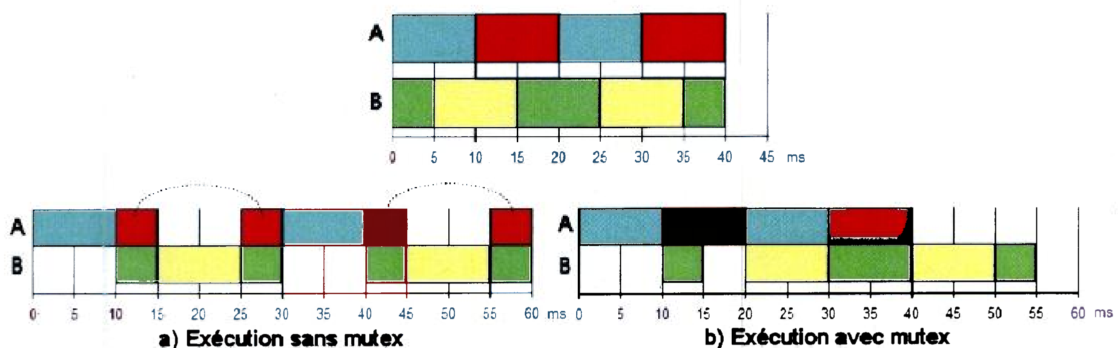


Figure 2-3: Accès asynchrone à des données partagées sans mutex et avec mutex

est alors plus longue car le parallélisme de l'application est grandement diminué pendant ces périodes d'attente.

Pour qu'un environnement virtuel soit stable, il faut évidemment prioriser la cohérence de la scène. C'est pourquoi l'approche *multi-threading* avec *mutex* servira de base au module de synchronisation présenté dans la prochaine section. Ce choix permettra d'ajouter du parallélisme à l'application finale et d'assurer une protection robuste des données partagées.

2.3 Module de synchronisation

Comme discuté plus tôt, l'approche modulaire a été choisie pour le développement des différentes fonctionnalités de l'environnement virtuel. Chacun des modules du système possède des propriétés locales qui sont indépendantes des autres. Par exemple, le module visuel présenté au prochain chapitre utilise des formes géométriques, des textures, des couleurs et d'autres éléments graphiques pour représenter les objets. Le module réseau, expliqué au chapitre 4, représente plutôt les objets par des message XML qui peuvent être transmis via les protocoles UDP et TCP pour rendre l'application distribuée. Le module haptique de (duTremblay à venir) utilise également des formes géométriques pour définir les forces et les interactions entre objets. Finalement, le module virtuel présenté au chapitre 5 définit des propriétés de plus haut niveau afin d'ajouter des fonctionnalités plus complexes aux objets. Ces fonctionnalités permettront à l'utilisateur d'interagir avec l'environnement et d'y ajouter des sources d'information qui ne sont généralement pas disponibles dans le monde réel.

Bien que chaque module possède des propriétés indépendantes, chacun d'entre eux doit pouvoir disposer les objets dans l'espace virtuel. Pour cela, on utilise des matrices de transformation pour représenter les informations de position, d'orientation et de facteur d'échelle. Un objet virtuel peut posséder les caractéristiques de plusieurs modules à la fois. Cependant, les modules impliqués deviennent alors complémentaires et il devient impératif de partager et de synchroniser les différentes matrices de transformation entre eux. Par exemple, lorsqu'un objet entre en collision avec un autre, c'est le module haptique qui prend en charge cette interaction. Cependant, c'est le rendu graphique qui permettra

d'observer l'événement. À l'inverse, le sens de la vue permet de localiser rapidement les objets à manipuler. Par contre, c'est le module haptique qui s'occupe de la sélection et de la préhension. Dans le cas d'une application distribuée, la synchronisation devient primordiale afin de s'assurer que chaque participant perçoive une scène virtuelle identique.

C'est le module de synchronisation nommé `InterModuleSynchro` (IMS) qui remplit la fonctionnalité de faire partager les informations de position, d'orientation et de facteur d'échelle entre les modules du système de façon cohérente et sécuritaire. Conformément à la définition de module explicitée plus tôt, l'IMS possède la propriété locale d'encapsuler les principes du *multi-threading* afin de représenter les objets comme un verrou central protégeant les données. Les propriétés partagées correspondent évidemment aux données de position, d'orientation et de facteurs d'échelle qui doivent être synchronisées.

Le module de synchronisation fait par contre exception à la définition originale en ce qui concerne la boucle d'exécution en parallèle, car il ne forme pas un *thread* indépendant comme les autres. En fait, comme il sera expliqué dans les prochaines sections, chaque sous-système greffe sa propre boucle au module de synchronisation en faisant appel à celui-ci pour synchroniser les données. Ainsi, ce dernier fait partie intégrante de l'exécution de chacun des modules plutôt que de s'exécuter en concurrence. Le module de synchronisation sert donc en quelque sorte de routeur entre les modules. Les prochaines sections donnent de plus amples détails sur la structure générale et les particularités du module de synchronisation ainsi que sur le processus de synchronisation que ce dernier définit afin de partager les données entre tous les modules.

2.3.1 Structure générale

Le module de synchronisation encapsule les priorités communes des objets de l'environnement virtuel comme la position, l'orientation et les facteurs d'échelle qui doivent être partagées entre tous les autres modules du système. À l'intérieur du module de synchronisation, ces objets virtuels sont alors représentés comme un verrou central protégeant l'accès aux données partagées. Les autres modules du système encapsulent ensuite les autres propriétés sensorielles et fonctionnelles des objets virtuels dans une classe d'objets particulière (visuel, haptique, etc.). Selon la configuration d'un objet virtuel

particulier, certaines de ces propriétés seront associées au verrou central pour former un objet virtuel multi-sensoriel complet. Les différents objets décrivant les propriétés de l'objet virtuel partagent ensuite les données communes via le verrou central auquel ils sont associés. L'ensemble de ses propriétés définit les fonctionnalités et les caractéristiques sensorielles de l'objet virtuel (visuel, haptique, réseau, etc.) qui sont gérées par les différents modules dans leurs boucles d'exécution respectives.

À l'intérieur du module de synchronisation, on définit ce verrou générique dans la classe `InterModuleObject` (IMO). Cette dernière contient un tampon, nommé `InterModuleBuffer` (IMB), qui conserve les données de position, d'orientation et de facteur d'échelle mises à jour dont l'accès asynchrone est protégé par un *mutex*.

Puisque les modules à synchroniser ne sont pas connus *a priori*, il importe de créer un processus général pour les associer de façon flexible à l'IMS. Effectivement, le module de synchronisation se trouve dans une couche logicielle de bas niveau à l'intérieur du système et le fait de le corréler étroitement à d'autres objets de plus haut niveau ne constitue pas une bonne approche pour le développement logiciel. Ainsi, la classe `Synchronizable` a été définie afin de représenter une instance générale d'un objet à synchroniser. Il est alors possible d'y encapsuler le processus de synchronisation et les objets des autres modules développés n'auront qu'à être dérivés de celle-ci pour ainsi hériter de son comportement de façon transparente. Un objet IMO est donc associé à un ensemble d'objets de la classe `Synchronizable` qui sont spécialisés à l'intérieur de chacun des modules développés. La Figure 2-4 illustre cette structure générale dont le comportement sera présenté dans les prochaines sections.

2.3.2 Format de données neutre

Le format de données utilisé pour conserver les informations de position, d'orientation et de facteurs d'échelle des objets n'est pas nécessairement connu *a priori* lors de l'ajout d'un module à synchroniser. Pour contrer ce problème, quelques structures mathématiques comme des matrices et des vecteurs ont été développées pour conserver les données à l'intérieur du IMB. Afin d'assurer une portabilité maximale, ces classes ne dépendent

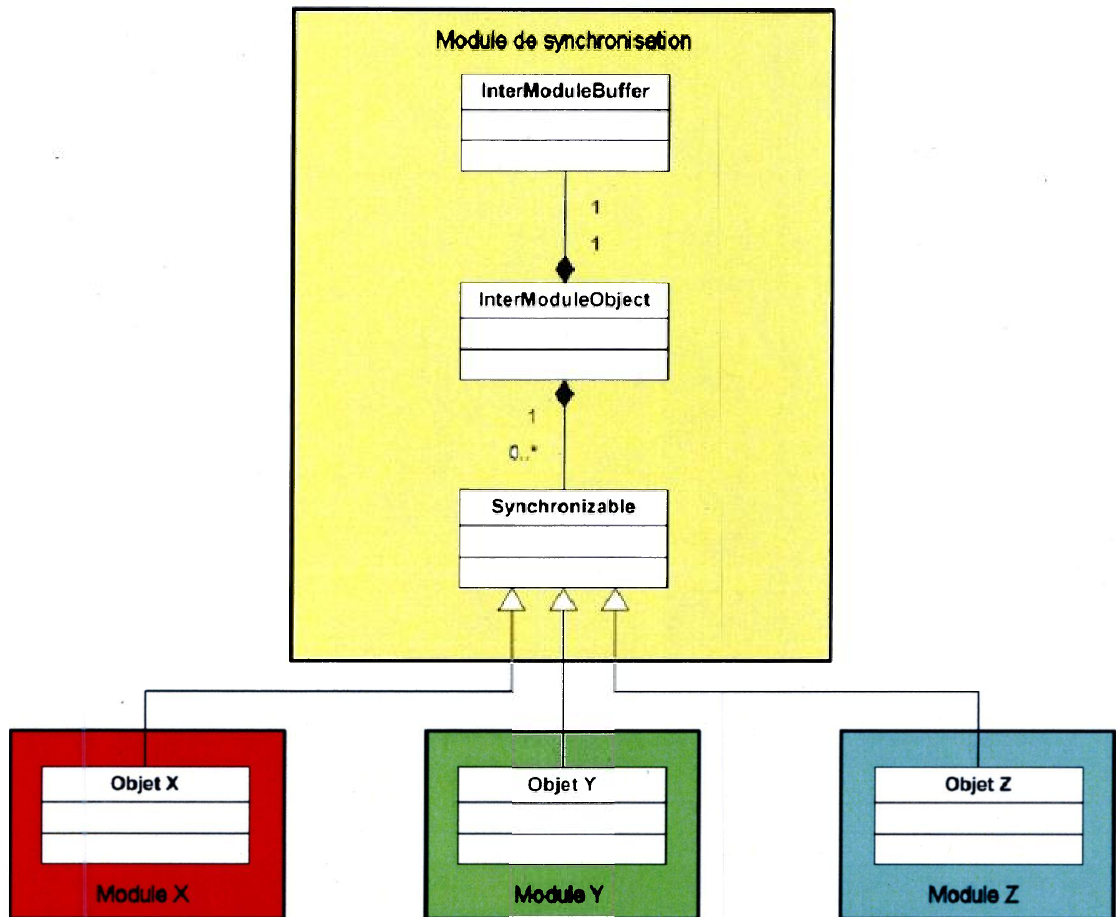


Figure 2-4: Structure générale du module de synchronisation

d'aucune librairie externe. Ainsi, on obtient un format de données neutre qui peut être facilement utilisé et converti par les sous-systèmes.

Comme mentionné précédemment, on peut représenter une orientation grâce à une matrice de rotation 3x3 ou un quaternion. La librairie mathématique développée dans ce projet définit ces deux structures respectivement par les classes `RotationMatrix` et `Quat`. On représente une position grâce à un vecteur 3D encapsulé dans la classe `Vector3D`. On peut combiner l'information de position et d'orientation dans une matrice de transformation homogène 4x4 appelée `TransformMatrix`. Finalement, on représente les couleurs avec quatre paramètres : le rouge (R), le vert (G), le bleu (B) et le canal alpha (A). Ce dernier paramètre définit le niveau de transparence de la couleur. La classe `Vector4D` est alors utilisée pour paramétrer une couleur dans un format neutre.

Le tampon `InterModuleBuffer` contenu dans les objets de la classe `InterModuleObject` conserve donc l'information de position, d'orientation et de facteurs d'échelle dans ce format de données neutre. Une matrice 4x4 de type `TransformMatrix` contient l'information de position et d'orientation. Deux vecteurs 3D permettent de conserver les facteurs d'échelle locaux et de groupe sur chacun des axes. Le premier type de facteur d'échelle s'applique à l'objet uniquement tandis que celui de groupe s'applique à l'objet et sa hiérarchie. En effet, comme on le verra plus loin, il est possible d'organiser les éléments d'une scène virtuelle de façon hiérarchique selon un principe parent-fils. Ainsi, grâce à ce facteur d'échelle global, il sera possible de modifier la taille d'un groupe en une seule étape.

Lorsqu'un objet applique ou reçoit une mise à jour, il faudra donc procéder à une étape de conversion entre le format de données neutre présenté ici et celui utilisé spécifiquement dans le module. Cette opération doit être spécialisée pour chacun des modules car il est impossible de prévoir un processus générique pour y arriver.

2.3.3 Processus de synchronisation des données

Lors de l'exécution en parallèle des modules dans des *threads* séparés, les objets qu'ils contiennent sont sans cesse mis à jour en boucle selon le comportement particulier de chacun des modules. Par exemple, le module visuel met à jour le rendu graphique, le module haptique calcule les nouvelles forces imposées par la physique appliquée à l'environnement, le module réseau envoie des messages XML qui décrivent les modifications aux autres instances de l'application, etc.

Pour harmoniser l'information partagée entre tous les sous-systèmes, on doit appliquer les trois étapes illustrées à la Figure 2-5 pendant la boucle de mise à jour des modules et ce pour chacun des objets contenus à l'intérieur de ceux-ci. Tout d'abord, l'objet à synchroniser (dérivant de la classe `Synchronizable`) doit faire une requête de mise à jour à l'IMO auquel il est associé. Cette première étape permet d'importer les données les plus récentes contenues en format de données neutre à l'intérieur du tampon IMB dans l'IMO et de les convertir dans le format de données utilisé dans le module pour effectuer un traitement quelconque.

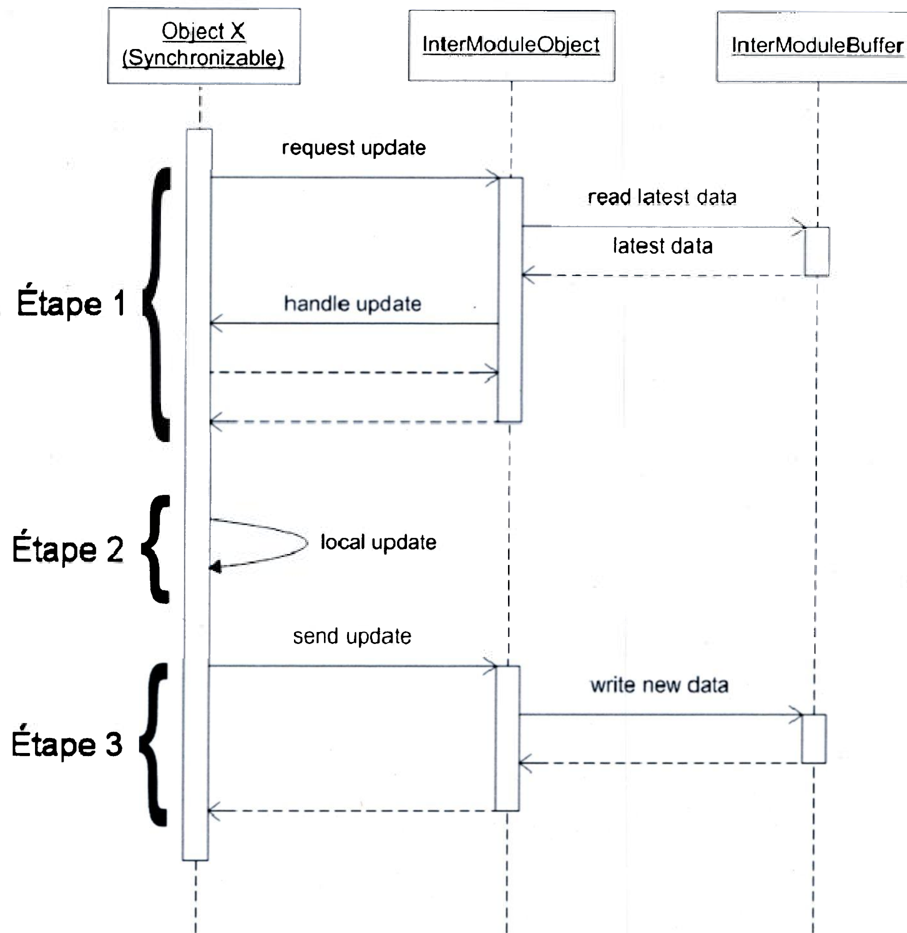


Figure 2-5: Processus de synchronisation simple

Puis, la deuxième étape du processus de synchronisation consiste à exécuter la mise à jour locale des données de l'objet `Synchronizable` qui est spécialisée dans chaque module à partir de cette récente information. Les opérations effectuées lors de cette mise à jour dépendront des fonctionnalités ou des caractéristiques sensorielles de l'objet virtuel que le module contrôle. Par exemple, un module visuel utilisera l'information de position, d'orientation et de facteur d'échelle pour mettre à jour la représentation visuelle des différents objets virtuels. Un module haptique utilisera cette même information pour mettre à jour les forces appliquées sur la totalité des éléments de la scène.

Finalement, la troisième étape consiste simplement à transmettre à l'IMO les nouvelles valeurs mises à jour résultant des traitements effectués localement sur l'objet pour que les autres modules puissent les recevoir lors de leur phase de synchronisation.

Les étapes 1 et 3 correspondent donc aux opérations de lecture et d'écriture dans le tampon partagé. Ce tampon est protégé par un *mutex* à l'intérieur de la classe `InterModuleObject` pour qu'un seul module à la fois puisse y accéder. Ainsi, on assure la cohérence et l'intégrité des données qui y sont contenues. Ces fonctionnalités de synchronisation sont encapsulées dans les classes `Synchronizable` et `InterModuleObject` et sont donc disponibles de façon transparente aux modules spécialisés.

Lors de l'étape 1, les données contenues à l'intérieur du tampon partagé peuvent avoir été modifiées ou non par un autre module depuis la dernière itération. Il est donc impératif de s'assurer que les données les plus récentes sont utilisées lors de la mise à jour locale des modules. L'étape 2 doit être spécialisée pour chaque module en fonction du comportement particulier de chaque sous-système. Finalement, l'étape 3 permet de transmettre les données de position, d'orientation et de facteurs d'échelle modifiées lors de l'étape 2 aux autres modules. Ainsi, ces derniers auront également l'information la plus récente lors de leur mise à jour locale.

Il arrive rarement que toutes les données partagées d'un objet soient modifiées à chaque itération. En effet, les facteurs d'échelle d'un objet demeurent généralement fixes et il se peut également qu'un objet demeure immobile pendant plusieurs itérations. La mise à jour systématique de toutes les valeurs du tampon partagé n'est donc pas l'approche la plus optimale. Effectivement, il est possible d'améliorer les étapes 1 et 3 du processus de synchronisation pour limiter la transmission inutile de l'information inchangée comme le démontre la Figure 2-6. Cet avantage sera particulièrement utile lors de l'envoi de messages sur le réseau comme on le verra plus loin.

Tout d'abord, lorsqu'un module envoie une mise à jour pendant l'étape 3, il ne modifie pas nécessairement la totalité des paramètres. En général, seule la position et l'orientation seront mises à jour régulièrement. Les facteurs d'échelle seront plutôt définis lors de la création de l'objet et ne seront modifiés que rarement. Ainsi, on peut conserver une trace des paramètres qui ont changé ou non à l'intérieur du tampon. Lorsqu'un objet à synchroniser émet une requête de mise à jour, on peut identifier les valeurs qui ont

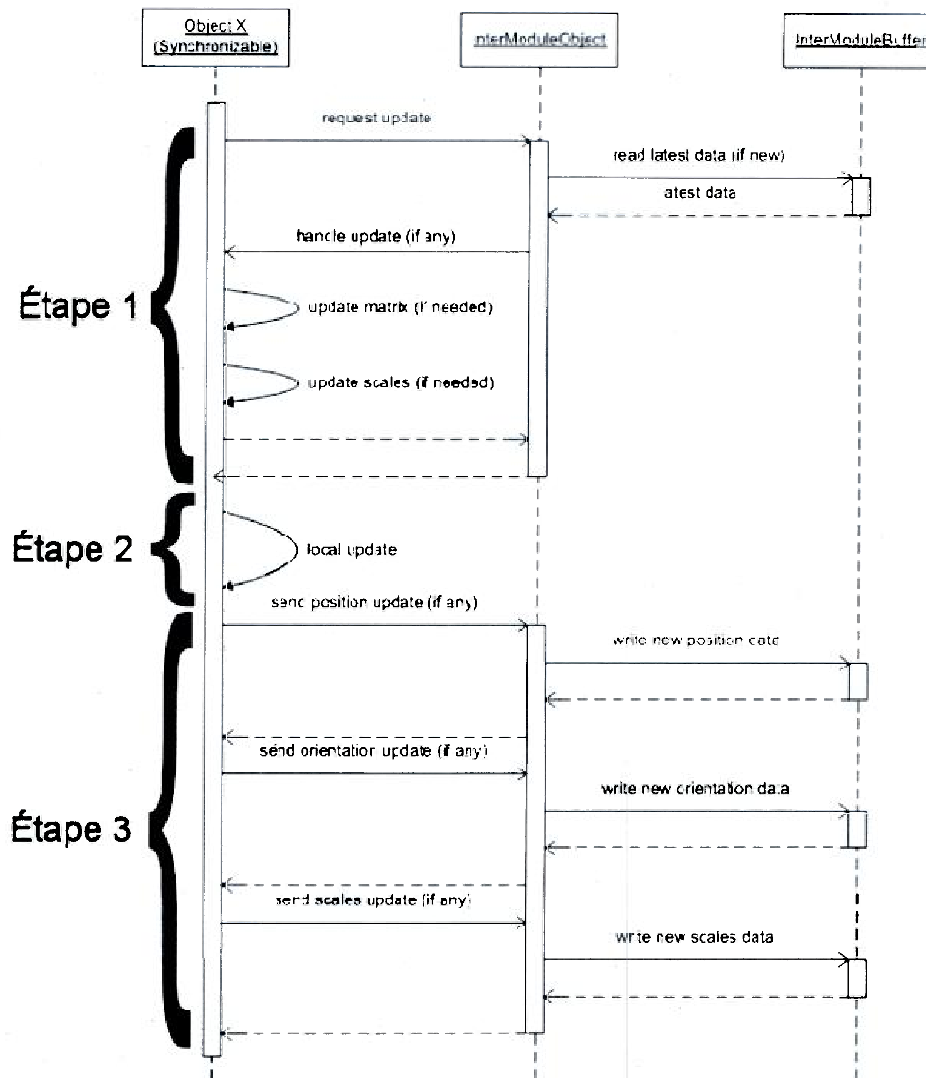


Figure 2-6: Processus de synchronisation optimisé

réellement changé depuis la dernière itération et ainsi ne transmettre que celles-ci pour limiter la bande passante et les modifications locales dans le module.

Ensuite, afin d'empêcher qu'un objet ne se transmette de nouvelles valeurs à lui-même ou qu'un module reçoive les mêmes données plus d'une fois, on peut ajouter un paramètre booléen indiquant le besoin d'une mise à jour dans la classe `Synchronizable`. Lorsqu'un module transmet une modification à l'IMO à l'étape 3 de la Figure 2-6, il suffit simplement que ce dernier active ce paramètre pour tous les objets lui étant associés sauf pour celui qui émet la mise à jour. Ainsi, lors de l'étape 1, l'objet faisant la requête de mise à jour la recevra une seule fois. L'objet IMO désactivera ensuite son booléen empêchant

ainsi la réception de valeurs identiques à la prochaine itération. L'objet en question ne recevra plus de mise à jour tant que de nouvelles informations ne seront pas transmises au tampon. Lorsque l'IMO envoie les données à un objet et qu'il désactive son état de besoin de mise à jour, il vérifie la totalité des objets `Synchronizable` lui étant associés. Si aucun d'entre eux n'a besoin d'une mise à jour, le tampon IMB est réinitialisé. La réinitialisation consiste simplement à indiquer qu'aucune nouvelle valeur n'est contenue dans le tampon (toujours à l'aide d'un booléen) tout en conservant la dernière mise à jour à l'intérieur de celui-ci.

Finalement, puisqu'un objet nouvellement associé à un IMO n'a encore reçu aucune mise à jour contenue dans le tampon lors de sa création, il faut intégrer un dernier paramètre booléen à la classe `Synchronizable` qui lui permet de détecter cette situation lors de sa première mise à jour. Ainsi, lorsque le module ayant créé le nouvel objet fera la requête, toute l'information lui sera transmise et ce, même si le tampon avait préalablement été réinitialisé. Grâce à l'ajout de ces valeurs binaires dans le tampon et la classe `Synchronizable`, on peut donc optimiser le transfert des données partagées en n'effectuant que les opérations essentielles. On évite ainsi d'appliquer des valeurs redondantes à chaque itération.

2.3.4 Priorité des modules

Bien que le processus de synchronisation optimisé fonctionne bien en théorie, il existe tout de même des situations problématiques où la cohérence ne sera pas assurée en raison de la nature du *multi-threading*. En effet, une fois qu'un objet a terminé la transmission de sa mise à jour à l'`InterModuleObject`, le *mutex* du tampon partagé est déverrouillé et n'importe quel autre module peut ensuite modifier l'information. *A priori*, si le nouvel objet accédant au tampon fait d'abord une lecture, puis sa mise à jour locale et finalement l'écriture sur les données communes, le tout demeurera cohérent. Cependant, pour des raisons évidentes de performance, le tampon ne peut pas demeurer verrouillé pendant la mise à jour locale de l'objet à l'intérieur du module. Ainsi, on peut imaginer une situation comme celle de la Figure 2-7 où un premier objet X accède au tampon partagé en lecture et le libère ensuite pour faire sa mise à jour locale. Puis, un second objet Y accède également

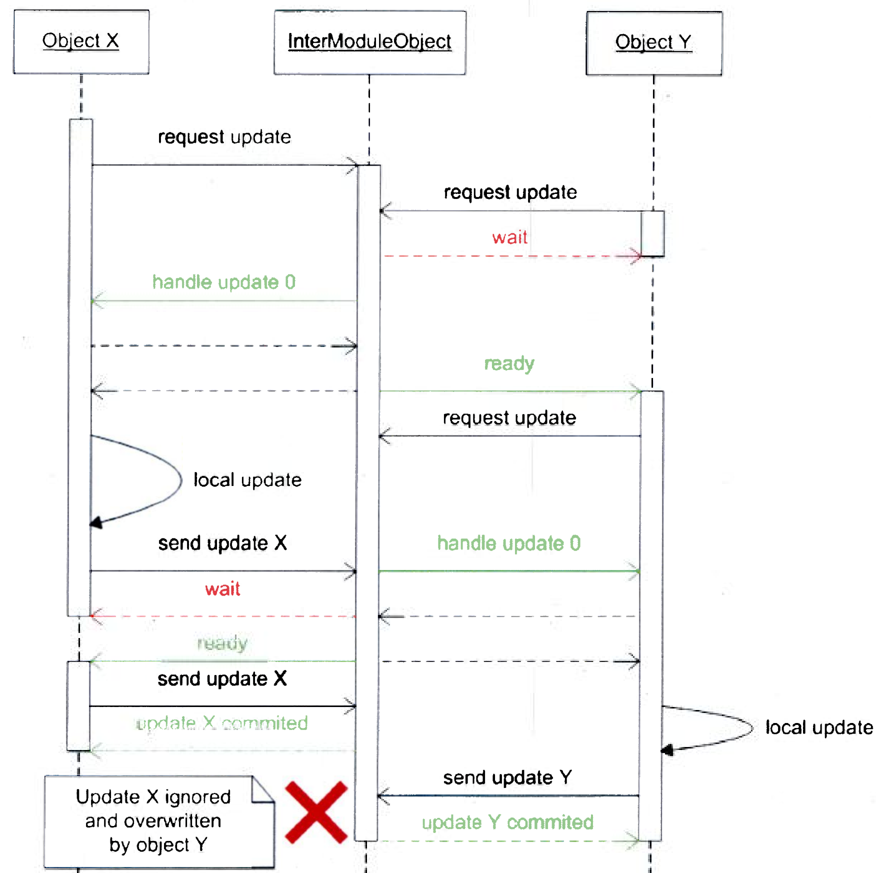


Figure 2-7: Incohérence lecture/écriture lors d'une mise à jour critique à la cohérence de la scène sans gestion de la priorité des modules

au tampon en lecture et le libère par la suite. Une fois la mise à jour locale de l'objet X terminée, celui-ci verrouille à nouveau le tampon pour l'opération d'écriture. Une fois libéré, ce dernier sera à nouveau accédé en écriture par l'objet Y qui n'aura pas pris en compte la dernière mise à jour. Selon les circonstances, cette situation peut être catastrophique pour la cohérence de la scène virtuelle.

Dans cette dernière figure, on peut considérer le cas où un objet X à l'intérieur d'un module visuel peut être téléporté[1] sporadiquement dans l'espace (dans le cas où l'utilisateur pourrait changer la position des objets via une interface graphique par exemple) et où un second module haptique fait tomber selon la loi de la gravité l'objet Y, associé au même IMO que l'objet X. Si la téléportation d'un objet correspond à la mise à jour du module visuel, on constate que le module haptique ne recevra jamais cette information et que la

[1] Téléporter : transporter d'un point à un autre instantanément, sans transition ni déplacement apparent.

chute de l'objet continuera à l'endroit initial. Lorsque le module visuel exécutera une nouvelle itération, il recevra le nouveau positionnement de l'objet émis par le module haptique et la téléportation de l'objet aura été ignorée.

Pour remédier à ce problème, il faut introduire le principe de priorité dans les mises à jour. Chaque objet `Synchronizable` à l'intérieur des modules se voit attribuer une importance particulière pour l'accès au tampon partagé. Lorsqu'une opération d'écriture est effectuée, il faut conserver cette priorité à l'intérieur du tampon pour indiquer à quel point la cohérence de la scène dépend de la mise à jour en question. Lorsqu'un deuxième `Synchronizable` tentera ensuite d'accéder au tampon, il ne pourra le modifier que si sa priorité est supérieure ou égale à la priorité minimale conservée à l'intérieur du tampon lors de la dernière mise à jour. Cette nouvelle fonctionnalité ajoute une nouvelle étape dans la réinitialisation du tampon partagé lorsque tous les objets ont reçus la dernière mise à jour. En effet, en plus d'indiquer qu'aucune nouvelle information n'est maintenant disponible, il faut également remettre la priorité du tampon au minimum afin que d'autres objets `Synchronizable` puisse y accéder par la suite.

Grâce à l'optimisation effectuée sur les trois étapes de synchronisation, il est possible de savoir à quel moment les objets associés à un `InterModuleObject` ont tous reçu une mise à jour. Lorsque ces données d'une priorité quelconque ont été transmises à tous les éléments, le tampon est réinitialisé. En plus d'indiquer qu'il n'y a plus de nouvelles informations, la priorité est également remise au niveau minimal. Ainsi, lorsque tous les objets auront reçu la mise à jour plus prioritaire, il sera ensuite possible d'accéder au tampon en écriture sans restriction pour n'importe lequel d'entre eux.

La Figure 2-8 reprend l'exemple présenté précédemment en donnant à l'objet X du module visuel une plus grande priorité pour la mise à jour. On constate que l'objet Y du module haptique, de plus basse priorité, ne peut pas accéder au tampon en écriture tant qu'il n'aura pas reçu la nouvelle mise à jour. Évidemment, cette approche introduit nécessairement des délais potentiels, mais assure par contre la cohérence de la scène.

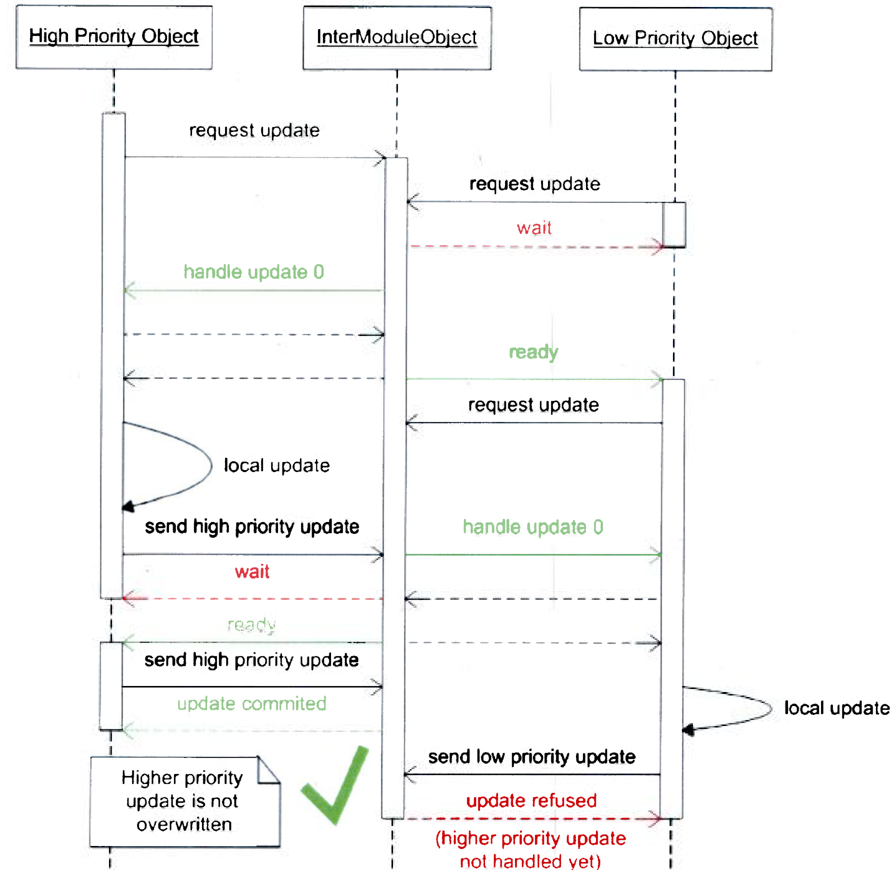


Figure 2-8: Incohérence corrigée grâce aux priorités

2.3.5 Verrouillage des objets

Malgré l'optimisation apportée par le principe de priorité qui assure la cohérence de la scène, il reste un problème moins critique qui est relié à la vitesse d'exécution de chacun des modules. En effet, jusqu'à présent on a supposé que chaque module s'exécutait à la même cadence. Cependant, selon la complexité des calculs de l'étape 2 du processus de synchronisation, ce ne sera généralement pas le cas. Dans ces circonstances, il se peut qu'un module soit plus rapide qu'un autre mais avec une priorité moins élevée. Ainsi, lorsque le module moins prioritaire aura reçu une mise à jour, le tampon sera réinitialisé et ce dernier sera accessible en écriture par la suite par les modules de moindre priorité. Jusqu'à maintenant, tout semble bien fonctionner. Cependant, dans le cas où le module prioritaire voudrait faire suivre une trajectoire quelconque à un objet, la continuité de ce déplacement ne serait alors plus assurée en raison des mises à jour moins prioritaires entre chaque itération du module ayant la priorité élevée.

Pour bien illustrer la situation, on peut considérer l'exemple suivant où un module de haut niveau prioritaire dont le taux de rafraîchissement est peu élevé tente cette fois-ci de faire suivre à un objet une trajectoire en ligne droite d'un point A à un point B à une distance quelconque au dessus du sol. Un module haptique, qu'on suppose plus rapide et moins prioritaire, tente de faire tomber les objets selon la loi de la gravité. Pour observer le déplacement de l'objet, on utilise également un module visuel qui n'interagit pas en écriture avec le tampon. Ce module ne contribue donc pas au problème mais il permet de l'observer. La Figure 2-9 démontre ce qui se passe. Tout d'abord, le module de haut niveau, lent mais prioritaire, applique sa première mise à jour et impose sa priorité élevée sur le tampon. Par la suite, le module haptique tente d'écrire sur le tampon et n'y arrive pas car sa

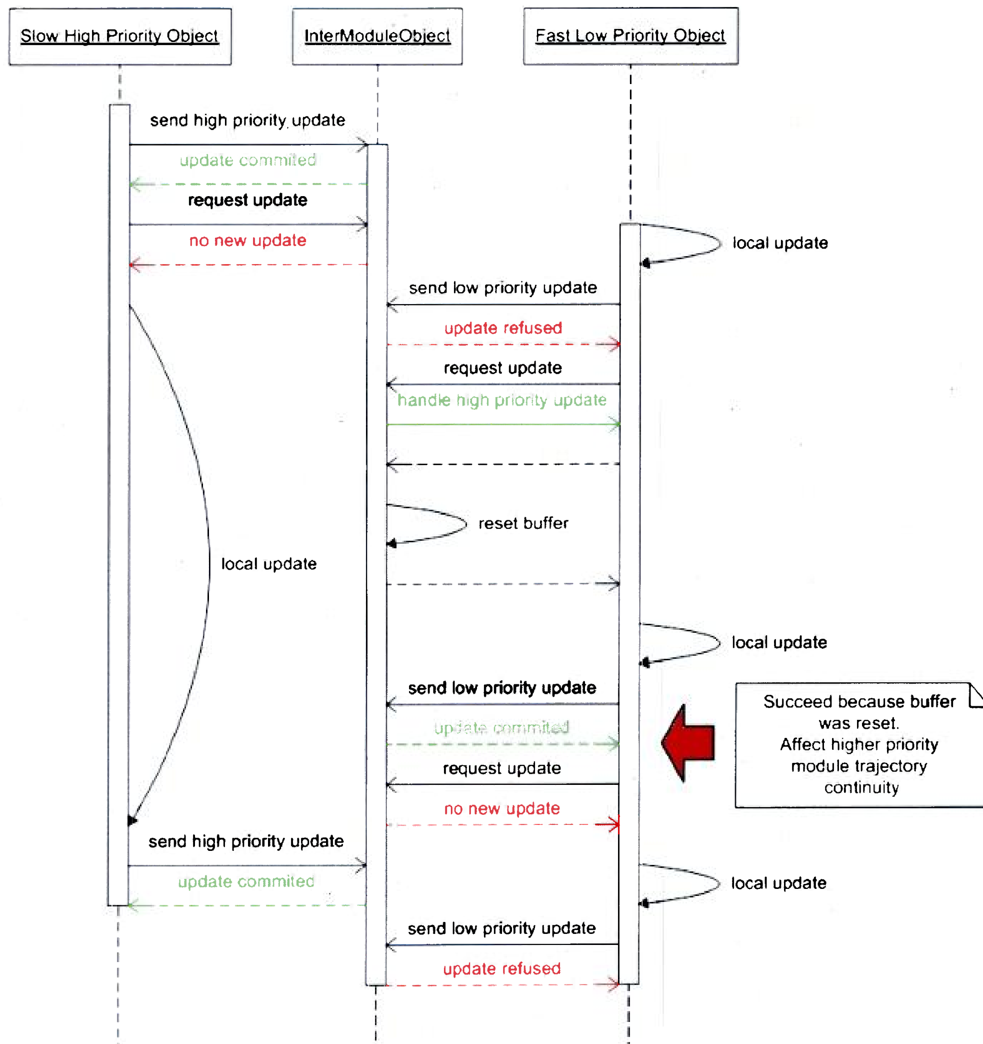


Figure 2-9: Artefact visuel dû à la vitesse relative des modules

priorité n'est pas suffisante. Par contre, lors de la prochaine itération, il aura lu la nouvelle information et le tampon sera réinitialisé. L'objet haptique de basse priorité pourra donc modifier les données contenues dans le tampon ce qui fera tomber légèrement l'objet. Selon la vitesse relative du module haptique par rapport au module de haut niveau gérant la trajectoire, il se peut que plus d'une itération comme celle-ci ait lieu dans le module haptique avant que l'objet ne soit à nouveau modifié par le module de haut niveau. Ainsi, il se peut que l'objet tombe pendant une certaine période de temps. Lorsque le module de haut niveau applique à nouveau une mise à jour il y aura deux possibilités. La première consiste à ignorer les modifications du module haptique et à reprendre la trajectoire initiale. Dans cette situation, l'objet sera téléporté du point où il en est rendu dans sa chute jusque sur la ligne droite à suivre. Sur toute la trajectoire, on observera donc un effet de tremblement de l'objet car il ne demeurera pas sur la trajectoire désirée à tout moment. La seconde possibilité consiste à reprendre le déplacement au point où l'objet est rendu. La trajectoire sera alors altérée. Cette dernière possibilité peut être acceptable selon la situation. Cependant, il sera généralement préférable de confier la gestion de la trajectoire continue d'un objet à un seul module à la fois pour assurer la cohérence.

Pour remédier à ce nouveau problème, on introduit la notion de verrou d'objet à l'intérieur du module de synchronisation. Lorsqu'un objet prioritaire prévoit la mise à jour du tampon partagé pendant un certain nombre d'itérations (de sa boucle d'exécution locale), il devra verrouiller l'IMO lui étant associé. Ainsi, seul un module plus prioritaire pourra déverrouiller le tampon et modifier les données s'y trouvant pendant la durée du verrou. De cette façon, les modules rapides et moins prioritaires n'altéreront pas la suite de mises à jour effectuées sur plusieurs itérations par un module de priorité supérieure. La version corrigée de l'exemple de la Figure 2-9 est illustrée à la Figure 2-10. On voit que le module plus rapide mais moins prioritaire ne peut plus faire sa mise à jour pendant la période où le module plus prioritaire a verrouillé l'objet.

Il est à noter que le problème de vitesse relative des modules présenté ici n'engendre aucune incohérence dans la scène. Seuls des artefacts visuels désagréables comme des effets de tremblement apparaîtraient si le processus de verrou n'existait pas. Cependant,

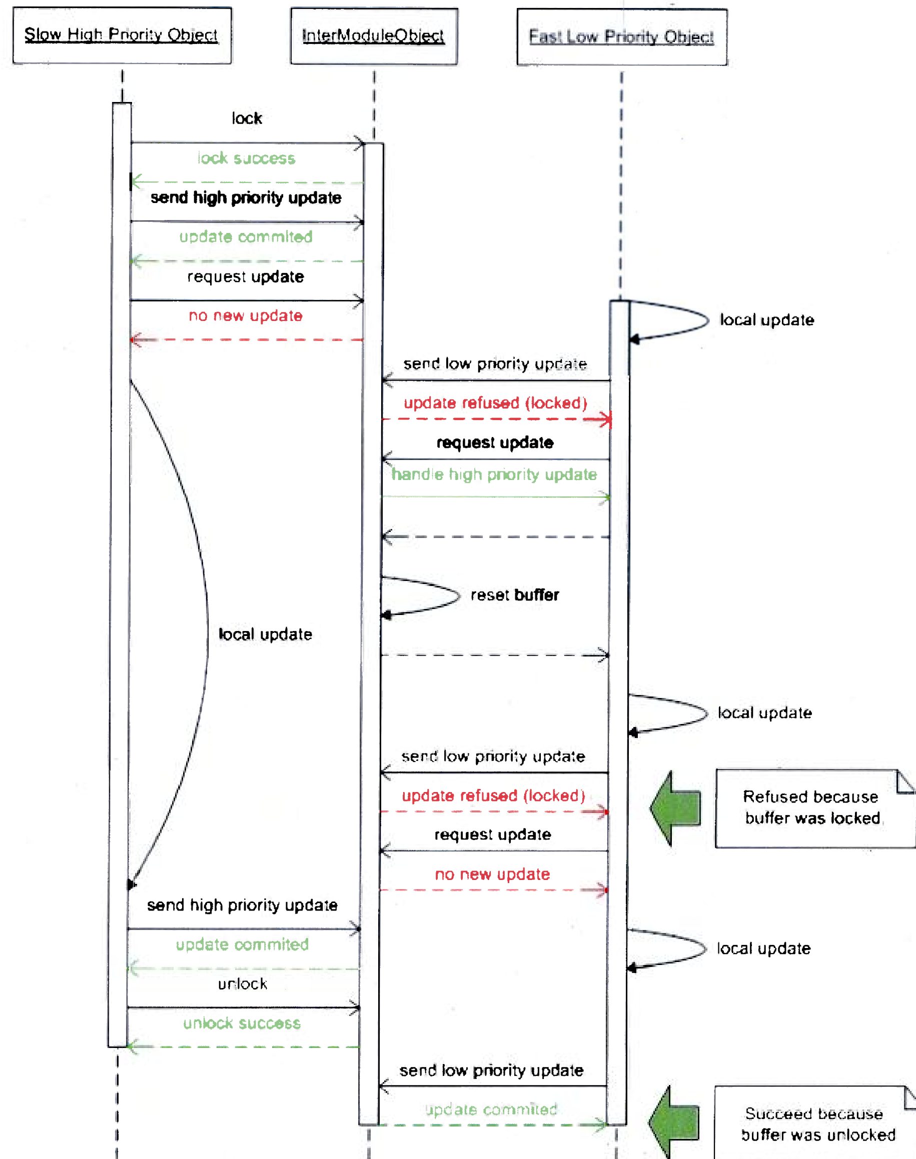


Figure 2-10: Artefact visuel dû à la vitesse relative des modules corrigé

pour assurer un minimum de réalisme, il est primordial d'intégrer cette fonctionnalité au module de synchronisation.

2.3.6 Gestion temporelle

Afin d'assurer la cohérence et la chronologie des mises à jour, surtout lors de l'utilisation d'un module pour la distribution sur un réseau, un processus de gestion temporelle doit être intégré au système. Le module de synchronisation utilise l'approche du *timestamp* pour ordonner les mises à jour dans le temps. Lorsqu'un objet `Synchronizable` transmet une

mise à jour à l'`InterModuleObject`, celui-ci lui associe une valeur numérique qui correspond au nombre de millisecondes écoulées depuis le début de l'expérience virtuelle. Ainsi, si une mise à jour arrive en retard, elle sera ignorée afin de ne pas introduire de saccades indésirables dans la scène. Si deux mises à jour sont transmises pendant le même *timestamp*, c'est d'abord leur priorité respective qui départagera leur accès au tampon. Si la priorité est égale, l'écriture sera exécutée selon les principes du *multi-threading*.

2.3.7 Émission d'événements

Bien que le partage des données de position, d'orientation et de facteurs d'échelle soit maintenant robuste, les interactions entre les modules demeurent tout de même relativement limitées. En effet, il se peut que certaines actions effectuées pendant l'étape 2 du processus de synchronisation aient un impact quelconque sur les autres modules. Par exemple, on peut vouloir donner une rétroaction visuelle à l'utilisateur lorsque le module haptique détecte la sélection ou la préhension d'un objet avec la main. Pour communiquer entre les modules de façon transparente, un mécanisme d'émission d'événements a été introduit au processus de synchronisation.

L'approche est relativement simple : il suffit d'ajouter deux nouvelles étapes aux processus de synchronisation comme le montre la Figure 2-11. Tout d'abord, on divise la phase 1 en deux sous-tâches distinctes : la lecture des données partagées dans le tampon (1.1) et de la liste des événements qui ont été classés en ordre chronologique (1.2). Puis, l'étape d'émission de nouveaux événements dépend de ce qui se passe pendant la phase 2 du processus de synchronisation. Ces événements seront transmis un à un à l'`InterModuleObject` selon le comportement de la mise à jour locale de chaque module.

À l'aide de la gestion temporelle présentée à la section précédente, on peut savoir à quel moment les événements se sont produits et ainsi les classer en ordre chronologique dans une liste. La résolution du classement est d'un millième de seconde. Si deux événements sont émis pendant le même *timestamp*, ce sont les algorithmes du *multi-threading* qui prendront la relève pour les ordonner. Chaque objet `Synchronizable` possède une telle liste d'événements qui peut être lue lors de l'étape 1.2 du processus de synchronisation.

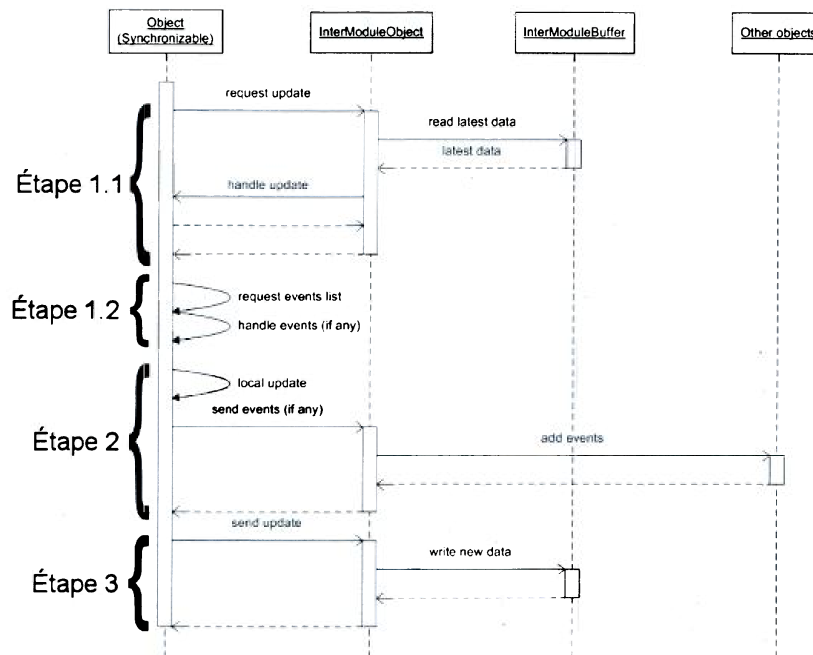


Figure 2-11: Processus de synchronisation optimisé avec émission et traitement des événements

Lorsqu'un nouvel événement est transmis à l'`InterModuleObject`, celui-ci l'envoie à tous les objets lui étant associés pour qu'ils l'ajoutent à leur liste sauf à celui qui l'a émis.

Pendant la boucle d'exécution de chaque sous-système, l'étape 1.2 consiste à lire et à traiter les événements de la liste un à un en ordre chronologique. Tous les événements n'auront pas nécessairement un impact sur le module en question. En effet, ces événements prennent la forme d'une valeur numérique constante qui leur est associée. C'est cette valeur qui sera transmise à l'`InterModuleObject` avec un *timestamp*. Selon la concordance des événements avec les modules, certains d'entre eux seront simplement ignorés et n'auront aucun effet sur leur comportement. D'autres généreront un comportement quelconque à l'intérieur des modules et permettront de diffuser des informations de plus haut niveau que la position, l'orientation et les facteurs d'échelle. Par exemple, dans le système développé, le module haptique émet des événements de sélection et de préhension des objets. Cette information est captée par le module visuel qui produit un résultat graphique à l'écran permettant de guider l'utilisateur dans ses tâches de manipulation. Le module haptique produit également d'autres événements comme des symboles effectués avec la main et la collision des objets avec le sol qui sont traités par un module de plus haut niveau.

Cette approche est bien sûr assez limitée puisqu'il est impossible de paramétrer les événements. Ces comportements plus complexes seront gérés par le module d'intégration de plus haut niveau présenté au chapitre 5. Pour le moment, cette méthode permet de conserver une grande flexibilité au niveau du module de synchronisation qui est de plus bas niveau. En effet, il suffit d'ajouter de nouvelles valeurs à la liste qui seront émises par un module spécifique. Ensuite, les autres modules ayant une affinité avec ces événements n'auront qu'à capter les valeurs numériques pertinentes et agir en conséquence.

2.3.8 Ajout flexible de modules

Grâce à l'encapsulation des fonctionnalités de synchronisation, l'ajout de nouveaux modules peut se faire de façon très flexible. En effet, pour définir un nouveau type d'objet, il suffit d'abord de le faire hériter de la classe `Synchronizable`. Les fonctions `requestUpdate()` et `requestEventList()` remplissent les étapes 1.1 et 1.2 du processus de synchronisation. Il faut ensuite surdéfinir les fonctions `handleUpdate()` et `handleEvent()` pour réagir aux mises à jour et aux événements. La phase 2 doit être spécialisée pour chaque module. Les événements et les mises à jour peuvent être transmis à l'`InterModuleObject` via plusieurs fonctions comme `sendPosition()`, `sendOrientation()`, `sendEvent()`, etc. Pour verrouiller et déverrouiller l'accès au tampon, il suffit de générer les événements « *lock* » et « *unlock* » respectivement. Les opérations de synchronisation sont exécutées de façon transparente.

Dans les prochaines sections, les différents modules développés au cours du projet seront présentés en détails. Ils sont tous basés sur les principes du module de synchronisation. Au chapitre 3, il sera d'abord question du module visuel qui permet de représenter les objets à l'écran sous forme graphique. Puis, le module réseau qui rend l'application multi-usagers sera présenté au chapitre 4. Finalement, le chapitre 5 explique le fonctionnement du module virtuel de plus haut niveau qui permet d'encapsuler l'ensemble des fonctionnalités de l'application finale. Le module haptique présenté dans (duTremblay à venir) utilise également le module de synchronisation pour communiquer avec le reste du système.

Chapitre 3 - Visualisation d'une scène virtuelle

Comme le sens de la vue est celui qui apporte le plus d'information à l'humain, il est impératif que l'environnement produise une représentation visuelle du monde virtuel. Ainsi, l'utilisateur peut facilement repérer les différents objets et les fonctionnalités qui s'offrent à lui. Plusieurs outils de rendu existent sur le marché pour générer rapidement une scène. OpenSceneGraph (OSG) disponible sur (Osfield 2007) est l'engin qui a été choisi pour ce projet. En plus d'être gratuit, il utilise la métaphore de l'arbre de rendu qui permet d'appliquer facilement les principes matriciels présentés au chapitre 1. Il propose également une panoplie d'effets spéciaux qui peuvent être ajoutés à la simulation. Il offre aussi la possibilité de charger plusieurs formats de modèles 3D dont OpenFlight et 3D Studio ce qui rend la création de la scène très flexible.

Afin d'implanter la stéréoscopie *off-axis* et le rendu sur plusieurs écrans, une seconde librairie, VR Juggler (VRJ), a été intégrée à OSG. En plus de modifier dynamiquement la projection de l'image de l'engin de rendu selon la position des yeux de l'utilisateur pour générer un point de vue réaliste, ce deuxième outil permet également de lire la position et l'orientation d'un très large éventail de traqueurs dont le système IS-900 d'InterSense Inc. Il offre aussi la possibilité de créer et d'intégrer des pilotes personnalisés pour de nouveaux périphériques ou pour modifier le comportement de ceux qui existent. Enfin, VRJ permet également de distribuer le rendu sur plusieurs machines en formant un *cluster*. Cette fonctionnalité permettrait par exemple d'utiliser quatre ordinateurs distincts pour produire l'affichage sur les quatre faces du système FLEX™ et de synchroniser les images grâce à un principe de *timestamp* semblable à celui présenté au chapitre 2. Cependant, cette dernière fonctionnalité n'a pas été utilisée pour le projet puisque les machines disponibles possèdent la configuration matérielle nécessaire pour produire le rendu sur quatre écrans directement et ce à partir d'un seul ordinateur.

Un module visuel nommé `VisualWorld` a été développé afin d'encapsuler les fonctionnalités des librairies graphiques OSG et VRJ. Il propose la métaphore de l'objet visuel qui réunit plusieurs éléments de l'arbre de rendu dans une même classe : `VisualObject`. Comme on le verra dans la prochaine section, il faut utiliser plusieurs

nœuds dans l'arbre de rendu d'OSG pour placer les géométries aux endroits désirés dans la scène virtuelle. La métaphore de l'objet visuel permet d'encapsuler plusieurs nœuds OSG dans une seule et même classe afin de déterminer facilement la position, l'orientation et les facteurs d'échelle des géométries représentées alors comme des objets qu'on peut disposer dans l'espace. Ainsi, l'arbre de rendu d'OSG n'est plus constitué d'un ensemble de nœuds aux propriétés diverses mais plutôt d'une hiérarchie d'objets visuels possédant une position, une orientation, des facteurs d'échelle et des géométries affichées à l'écran.

Le module visuel respecte la définition de module présenté au chapitre 2. En effet, il possède la propriété locale de représenter les objets virtuels comme des éléments observables à l'écran. Comme pour les autres sous-systèmes intégrés à l'application finale, la portion visuelle devra partager et synchroniser les informations de position, d'orientation et de facteurs d'échelle encapsulées à l'intérieur des objets visuels avec les autres modules.

Avant de définir plus en détails le comportement du module visuel, la prochaine section présente d'abord les principes de l'arbre de rendu qui sont utilisés dans OSG. Par la suite, les principes du rendu stéréoscopique *off-axis* proposés par VRJ seront expliqués plus en détails. Finalement, la structure et le comportement du module `VisualWorld` pour la gestion des propriétés des objets visuels et de leur synchronisation avec les autres modules seront présentés à la fin de ce chapitre.

3.1 Principe de l'arbre de rendu dans OpenSceneGraph

L'une des façons les plus populaires d'organiser les objets dans la majorité des librairies graphiques est l'arbre de rendu ou *scene graph* en anglais. Cette structure est en fait une hiérarchie de nœuds. Chaque nœud, à part la racine, possède un(des) parent(s) duquel (desquels) il hérite de certaines propriétés comme la position, l'orientation, le facteur d'échelle, la transparence, etc. Un nœud peut aussi posséder des frères dont les propriétés communes sont seulement celles des parents. Enfin, un nœud peut aussi posséder des fils auxquels il transmet ses propriétés et celles des générations précédentes. Un nœud n'ayant pas de fils est appelé une feuille. La Figure 3-1 montre la structure générale d'un arbre de rendu. Dans cette image, le nœud C est fils de A et frère de B et D. Il est également père

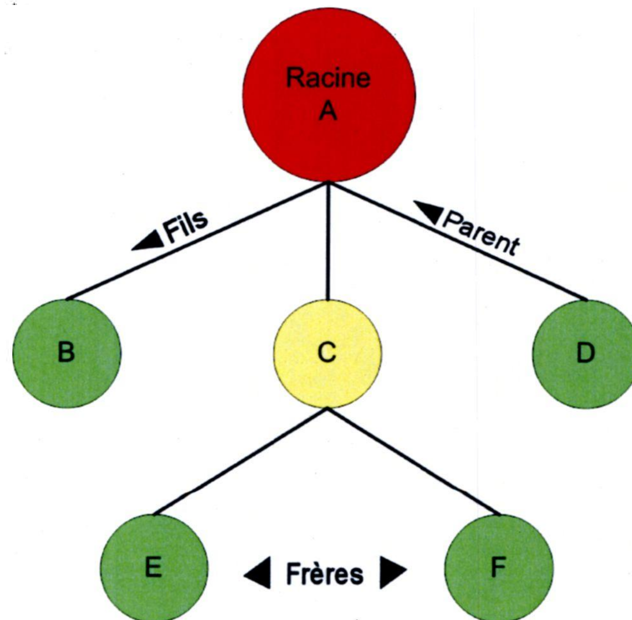


Figure 3-1: Structure générale d'un arbre de rendu

des noeuds E et F qui forment des feuilles tout comme B et D puisque ceux-ci n'ont pas de descendance.

Dans OSG, les nœuds de l'arbre de rendu peuvent donc regrouper d'autres nœuds, transformer la position, l'orientation ou la taille d'un objet, afficher des géométries, illuminer la scène, etc. La connaissance des différents éléments de l'arbre permettra au développeur de générer une hiérarchie d'objets pouvant être manipulée de façon très flexible. Les différents types de nœuds sont représentés par des classes dans OSG. L'utilisation de l'héritage permet de donner à la hiérarchie des fonctionnalités générales pendant que certains nœuds seront plus spécialisés.

Les classes de nœuds principales utilisées pour le projet sont présentées à la Figure 3-2. Tout d'abord, la classe `osg::Node` est la plus générale de l'arbre de rendu. Elle représente tout simplement un nœud générique dans l'arbre. Une spécialisation est nécessaire afin que le nœud ait un comportement particulier comme l'affichage de géométries ou la possession de fils.

Puis, certains nœuds représentent un regroupement de formes géométriques visibles qui sont appelés géodes et qui sont représentés par la classe `osg::Geode`, dérivée de

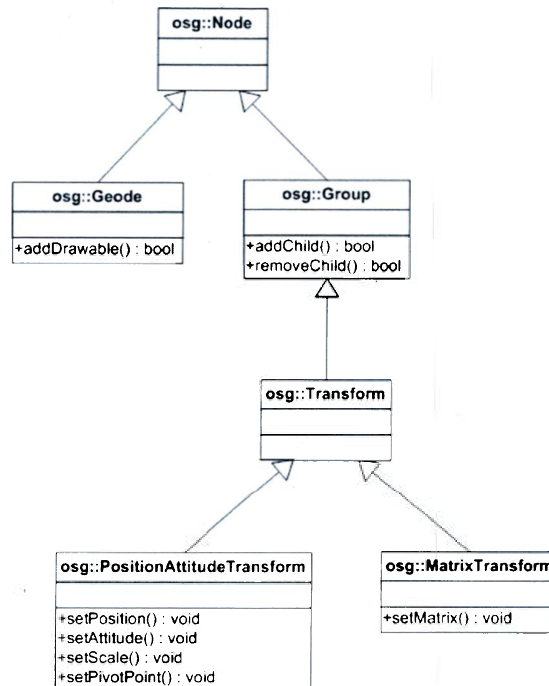


Figure 3-2: Principales classes de nœuds de l'arbre de rendu d'OpenSceneGraph

`osg::Node`. Ces nœuds forment le résultat graphique affiché à l'écran et contiennent les paramètres qui sont utilisés par OpenGL[1] pour le rendu.

Ensuite, la classe `osg::Group`, dérivée également de la classe `osg::Node`, permet l'ajout, la suppression et la manipulation de fils associés à un noeud. Cependant, cette classe demeure tout de même assez générique car elle ne possède aucun comportement particulier. Pour définir une transformation sur un groupe d'objets dans l'espace, on utilisera plutôt la classe `osg::Transform` qui encapsule une matrice de transformation 4x4 pour établir la position, l'orientation et les facteurs d'échelle du nœud et de sa hiérarchie. En fait, ce sont plutôt les classes spécialisées `osg::MatrixTransform` et `osg::PositionAttitudeTransform` qui possèdent les fonctions nécessaires à la modification de cette matrice. Pendant que la première classe spécialisée modifie directement les valeurs de la matrice, la seconde utilise plutôt deux vecteurs 3D pour la position et le facteur d'échelle et un quaternion pour l'orientation. Ces trois paramètres, plus simples conceptuellement, seront utilisés pour modifier les différentes portions de la

[1] OpenGL : librairie graphique qui forme la couche logicielle de bas niveau d'OpenSceneGraph pour le rendu. Les commandes OpenGL sont généralement compatibles avec la plupart des cartes graphiques.

matrice déterminant ainsi la transformation. Puisque la classe `osg::MatrixTransform` modifie directement la matrice de transformation, on peut donc effectuer des transformations non-homogènes plus complexes comme des biais (*skews* en anglais) et des symétries. Par contre, la classe `osg::PostionAttitudeTransform` ne peut définir que des transformations homogènes.

Une fois la matrice appliquée à un nœud de transformation, celle-ci définit le nouveau référentiel à partir duquel les fils de ce nœud sont positionnés. Par la suite, de nouveaux référentiels peuvent être appliqués cumulativement dans la hiérarchie de l'arbre, d'un niveau à l'autre. Par exemple, soit les nœuds A, C et E de la Figure 3-1 qui définissent respectivement les matrices de transformation T_A, ${}_AT_C$ et ${}_CT_E$ relatives à leur parent s'il y en a un. La matrice T_A représente la transformation du nœud A par rapport à l'origine absolue du monde virtuel. Ensuite, ${}_AT_C$ représente la transformation du nœud C par rapport à A. Puis, ${}_CT_E$ détermine la transformation du nœud E par rapport à C. Les coordonnées des fils incluront toujours implicitement le déplacement des générations précédentes. Ainsi, on peut déterminer à quel endroit se trouve un noeud par rapport à l'origine absolue en multipliant l'ensemble des matrices se retrouvant le long de la hiérarchie entre la racine et ce nœud. Pour terminer cet exemple, la matrice de transformation du nœud E par rapport à l'origine du monde virtuel serait obtenue ainsi :

$$T_E = T_A\,{}_AT_C\,{}_CT_E$$

Ce type de nœud est très utile pour appliquer une transformation commune à un groupe d'objets indépendants. Une hiérarchie de tous ces nœuds permet rapidement la conception d'un environnement graphique où les objets possèdent des dimensions, des positions et des orientations dans l'espace virtuel. D'autres propriétés comme l'invisibilité et la transparence sont également transmises des parents aux fils.

Finalement, OSG permet de modifier dynamiquement la structure de l'arbre. On peut donc facilement ajouter ou enlever des nœuds (ou des branches) à la hiérarchie de l'arbre de rendu et modifier les paramètres de chacun d'eux. La fonction de rendu graphique de l'application parcourt l'arbre à partir de la racine jusqu'aux feuilles en appliquant à chaque

nœud et à ses fils les transformations et en affichant les géodes à l'écran selon les paramètres géométriques fournis.

3.2 Principe du rendu stéréoscopique *off-axis* dans VR Juggler

Comme mentionné plus tôt, c'est la librairie VR Juggler (VRJ), disponible sur (Infiscape 2007), qui permet de produire la stéréoscopie *off-axis* en s'intégrant aux fonctionnalités de rendu d'OSG. Tout d'abord, on obtient la position de chacun des yeux de l'utilisateur dans l'espace grâce à la position et à l'orientation d'un traqueur. Ensuite, en connaissant d'avance la disposition des écrans qui demeurent fixes pendant l'expérience virtuelle, on peut faire la projection de toute la scène sur ceux-ci afin d'ajuster dynamiquement le point de vue de l'utilisateur de façon réaliste. Comme le démontre la Figure 3-3, le principe de base de la stéréoscopie *off-axis* consiste à tracer une droite entre chaque œil et tous les points de la scène. Le point d'intersection de ces droites avec l'écran correspond à l'endroit où le point correspondant dans la scène virtuelle 3D doit être affiché sur la surface 2D. L'ensemble des projections pour chaque œil forme deux images distinctes pour simuler la troisième dimension et donner un effet de profondeur. En fait, chaque écran forme en quelque sorte une fenêtre sur le monde virtuel par laquelle on peut observer la scène.

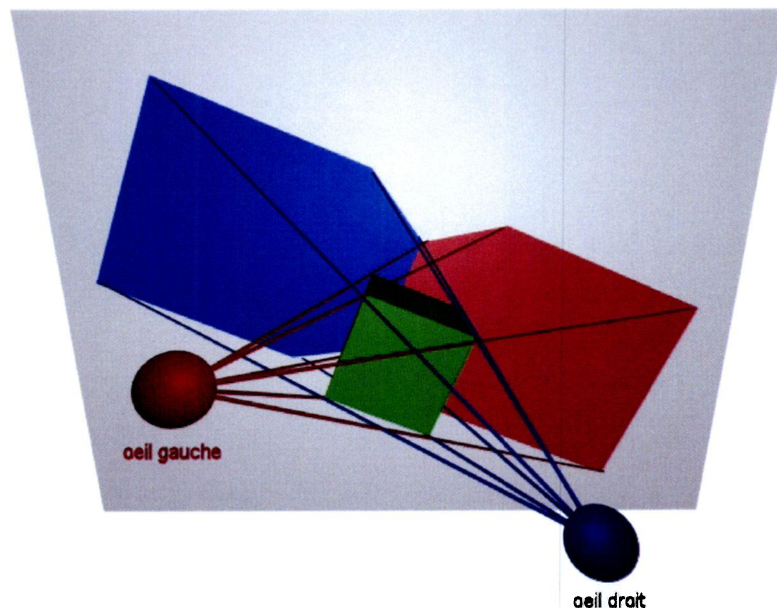


Figure 3-3: Projection de la scène virtuelle sur l'écran selon la position des yeux de l'utilisateur

Comme pour une fenêtre réelle à l'intérieur d'un établissement qui donne sur l'extérieur, l'information visuelle observable au travers de la surface ne sera pas la même selon l'endroit où l'on se positionne pour regarder.

Évidemment, calculer le point d'intersection avec l'écran pour tous les éléments de la scène serait très inefficace. Il est possible de calculer une matrice de projection pour chaque œil en connaissant la position de celui-ci et les coordonnées des quatre coins de l'écran dans l'espace. Cette matrice, calculée par VRJ, est ensuite directement appliquée sur chaque point 3D de la scène virtuelle pour obtenir la coordonnée correspondante sur l'écran où la projection doit être faite. Par la suite, en connaissant la résolution de l'écran, on peut interpoler la couleur de chaque pixel selon le filtre utilisé (*antialiasing*, anisotropique, etc.). La technique présentée ici suppose que les écrans sont plans. Bien sûr, d'autres approches existent afin de projeter la scène sur des surfaces hémisphériques ou de formes quelconques. Cependant, comme les écrans utilisés pour le projet sont rectangulaires, seule la méthode utilisée pour les écrans plans sera présentée ici.

Avant de présenter les équations, il faut tout d'abord définir les référentiels qui sont en cause pour le calcul de la projection des éléments de la scène virtuelle sur les surfaces physiques. Premièrement, le **référentiel physique** représente le point d'origine à partir duquel sont définis tous les éléments réels devant être positionnés dans l'espace : les coins des écrans, la tête et les mains de l'utilisateur, la *wand*, les différents émetteurs d'ultrasons du système IS-900 permettant de déterminer la position des traqueurs, etc.

Ensuite, le **référentiel de la tête** détermine la position et l'orientation de la tête de l'utilisateur par rapport au référentiel physique à l'aide d'un traqueur. Puis, on détermine les deux **référentiels des yeux** (gauche et droit) grâce à une translation fixe relative au référentiel de la tête. C'est à partir de ces deux points d'origine que les droites sont projetées sur les objets virtuels en traversant la surface de projection pour former les deux images stéréoscopiques.

Puisque l'orientation des yeux n'a pas d'importance pour la projection de droites sur la surface de l'écran, on peut aligner le référentiel de chacun des yeux avec le plan que forme l'écran pour que l'un des trois axes soit orthogonal à celui-ci. On obtient ainsi huit

référentiels de caméra (un pour chaque écran et pour chaque oeil). Le fait d'avoir une transformation supplémentaire pour aligner le référentiel de l'oeil avec l'écran facilite simplement les calculs sans toutefois altérer les résultats.

Finalement, le **référentiel virtuel** représente le point d'origine du monde virtuel. C'est le référentiel absolu du système à partir duquel tous les objets virtuels sont positionnés. Même le référentiel physique possède une transformation relative à ce dernier afin de faire correspondre les éléments physiques avec leurs représentations virtuelles. En effet, le référentiel physique peut se déplacer virtuellement dans l'environnement afin de modifier dynamiquement le champ de vue qu'offrent les surfaces de projection sur la scène. Ainsi, en effectuant cette transformation, on obtiendra la position « réelle » des objets physiques à l'intérieur du monde virtuel.

Les Figure 3-4 et Figure 3-5 illustrent l'ensemble des référentiels qui seront utilisés pour le calcul de la matrice de projection. Dans la Figure 3-4(a), on illustre d'abord l'origine des référentiels virtuel, physique et celui de la tête de l'utilisateur définis respectivement par les valeurs O_V, O_P et O_H. Puis, la Figure 3-4(b) montre les référentiels des deux yeux, O_{E_L} et O_{E_R}, définis par rapport au référentiel de la tête. Ensuite, la Figure 3-4(c) illustre les référentiels de caméra pour les quatre écrans du système. Finalement, la Figure 3-5 montre le référentiel 2D O_S formé par le plan d'un écran sur lequel est projeté l'espace 3D défini par l'une des caméras.

Pour décharger quelque peu la Figure 3-4(c), il est à noter qu'un seul référentiel de caméra par écran y est représenté. En effet, pour chacun des quatre écrans, il y aura une caméra par œil soit un total de huit référentiels de caméra en tout. Par la suite, toujours par souci de simplifier les équations et les explications, le calcul de la matrice de projection sera présenté en n'utilisant qu'un seul œil et un seul écran. Le référentiel de l'œil sera défini par O_E et celui de la caméra, par O_C.

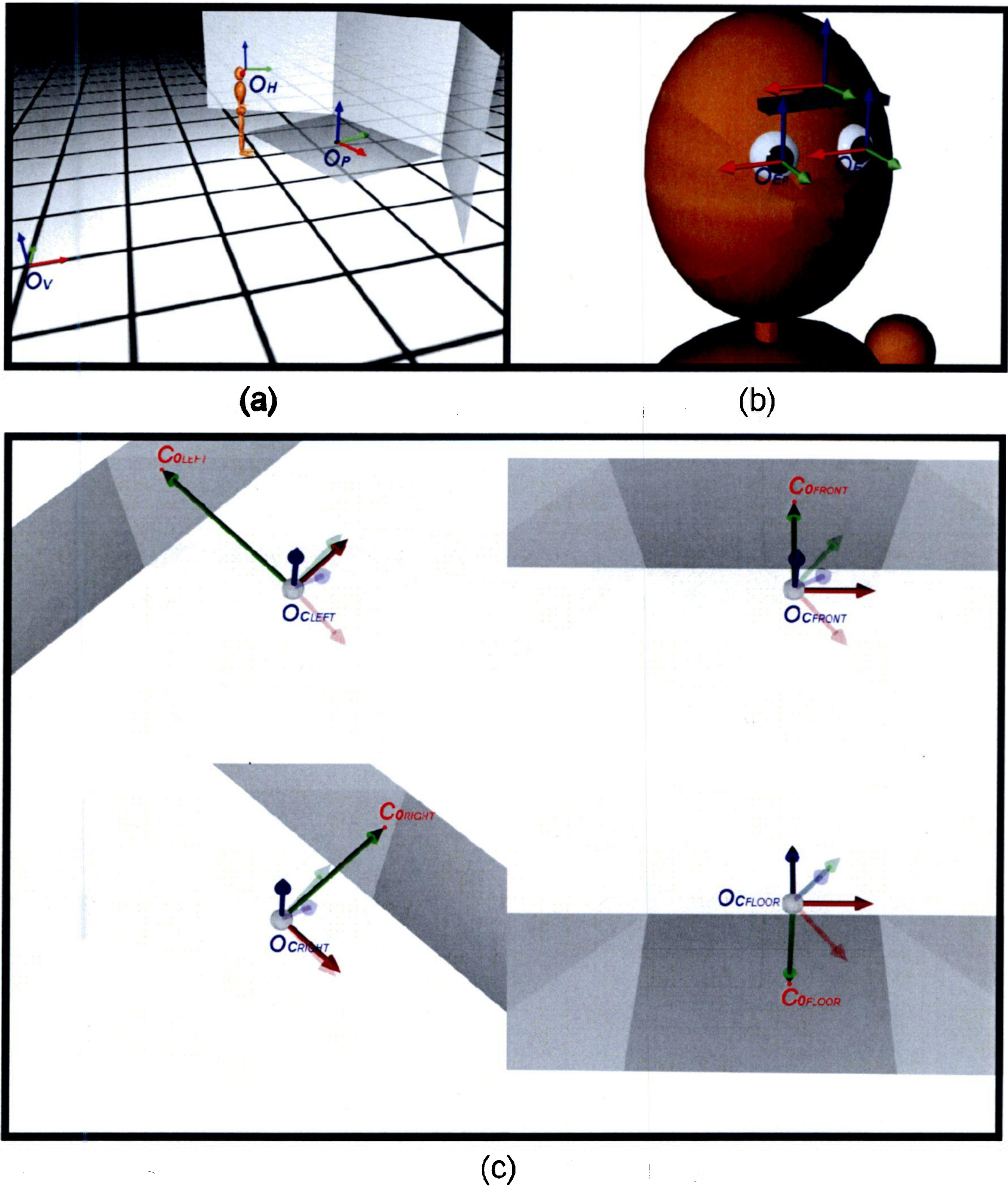


Figure 3-4: Référentiels utilisés pour définir les paramètres extrinsèques
a) Référentiels virtuel (OV), physique (OP) et de la tête (OH)
b) Référentiels de la tête (OH) et des yeux (OE)
c) Quatre référentiels de caméra

La méthode proposée se base sur les notions utilisées en vision artificielle pour le calibrage d'une caméra présentées dans (Forsyth and Ponce 2003). Chaque caméra virtuelle est constituée d'une surface de projection et d'un centre optique. La surface de projection

correspond ici à chacun des quatre plans définis par les écrans du système de rendu. Le *centre optique* est défini par le référentiel de la caméra dont l'un des trois axes, nommé l'*axe optique*, est orthogonal avec la surface de projection. Le point d'intersection C_0 entre l'axe optique et la surface de projection dans la Figure 3-4(c) est appelé le *point principal.* Puisqu'il y a quatre surfaces de projection et deux yeux, il faudra déterminer huit modèles de caméra pour obtenir huit matrices de projection.

Tout comme pour le calibrage d'une caméra réelle, il faudra déterminer les paramètres intrinsèques et extrinsèques de la caméra virtuelle. Les *paramètres extrinsèques* représentent la rotation et la translation requises pour convertir les coordonnées des objets virtuels données selon le référentiel virtuel en coordonnées relatives au référentiel de la caméra. Puisqu'il y a cinq référentiels impliqués dans la Figure 3-4 (O_V , O_P , O_H , O_E , et O_C), il faudra quatre matrices de transformation homogène 4x4 pour convertir les coordonnées 3D dans l'espace d'un référentiel à l'autre, ${}_PT_V$ (virtuel → physique), ${}_HT_P$ (physique → tête), ${}_ET_H$ (tête → oeil) et ${}_CT_E$ (oeil → caméra). Ainsi, les coordonnées d'un point $p_V = [x_v \quad y_v \quad z_v]^T$ données dans le référentiel du monde virtuel pourront être converties en coordonnées d'un point $p_c = [x_c \quad y_c \quad z_c]^T$ relatives au référentiel de la caméra grâce à la matrice des paramètres extrinsèques E :

$$E = {}_CT_E\,{}_ET_H\,{}_HT_P\,{}_PT_V$$

$$\begin{bmatrix} p_C \\ 1 \end{bmatrix} = E \begin{bmatrix} p_V \\ 1 \end{bmatrix}$$

Chacune de ces quatre matrices de transformation peut-être modifiée dynamiquement pendant l'exécution puisque l'utilisateur peut se déplacer physiquement et virtuellement dans l'espace 3D. Seule la matrice ${}_ET_H$ qui représente la transformation entre la tête et l'œil demeure généralement constante pendant l'exécution. Cette valeur changera plutôt d'un utilisateur à l'autre selon la physiologie de son visage.

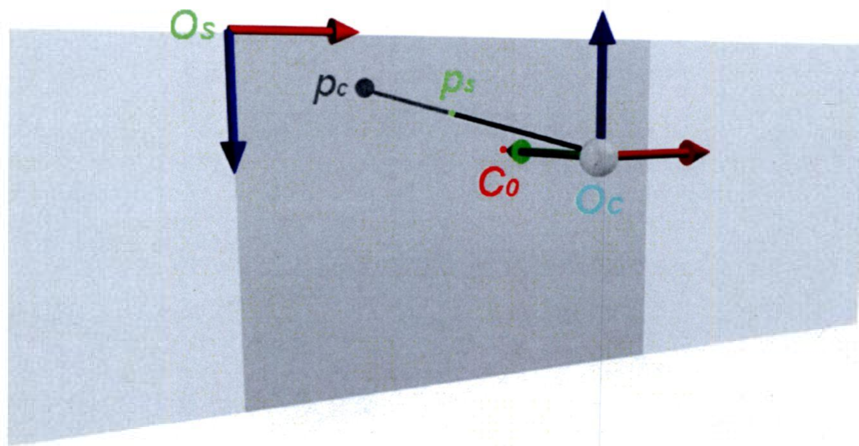


Figure 3-5: Référentiels utilisés pour définir les paramètres intrinsèques

Les *paramètres intrinsèques* définissent la matrice de projection qui transforme les coordonnées 3D d'un point donné selon le référentiel de la caméra en coordonnées 2D relatives au référentiel formé par le plan orthogonal à l'axe optique et parallèle à l'écran dont l'origine est le centre optique comme illustré à la Figure 3-5. Les paramètres intrinsèques tiennent compte de la conversion des unités de mesure, de la distance focale, de la translation entre l'origine du référentiel de l'écran O_S et le point principal C_0, ainsi que des potentiels défauts de fabrication mineurs d'une caméra réelle généralement dus à la lentille.

Tout d'abord, la matrice des paramètres intrinsèques doit convertir les unités du référentiel de la caméra (mètres) en unités pour le référentiel de l'écran (pixels). Comme les pixels sont généralement rectangulaires, il y aura deux facteurs d'échelle, k et l, pour convertir les unités de mesure sur les deux axes de l'écran : horizontal ($\vec{u}$) et vertical ($\vec{v}$). Dans le cas du LIV, les projecteurs utilisent une résolution de 1400 x 1050 pixels sur une surface de projection de 3,25 m (10 pi 8 po) par 2,44 m (8 pi). Par une simple division, on obtient ici une valeur égale pour les deux facteurs, soit 430,5 pixels/m.

Ensuite, la distance focale f est un autre paramètre qui affecte la projection de l'image. Sur une caméra réelle, cette valeur est affectée par l'objectif et la lentille utilisés avec l'appareil. Cependant, la caméra virtuelle présentée ici utilise plutôt un modèle de lentille « *pinhole* ». Selon ce modèle, l'image sera toujours au foyer peu importe la distance entre l'objectif et la surface de projection puisque la lumière passe par un trou infiniment petit situé sur le centre optique. La distance focale varie alors dynamiquement selon la position des yeux de l'utilisateur. Elle correspond simplement à la distance entre le centre optique et le point principal C_0. Évidemment, ce type de lentille est réalisable virtuellement mais il n'est pour le moment pas possible de reproduire une telle lentille pour une caméra réelle car il n'y aurait alors pas assez de lumière qui pourrait passer par l'objectif pour projeter l'image sur la pellicule ou le CCD.

Les paramètres intrinsèques présentés jusqu'à maintenant permettent de calculer les coordonnées en pixels par rapport au point C_0 sur la surface de l'écran. Afin d'obtenir des coordonnées relatives à l'origine de l'écran, il faut simplement ajouter la translation du point principal C_0 représenté par la paire (u_0, v_0) dans le référentiel 2D O_S.

Grâce à l'ensemble de ces paramètres, on peut calculer la projection sur le plan de l'écran pour tout point dans l'espace 3D défini selon le référentiel de la caméra virtuelle. En effet, soit un point $p_c = [x_c \quad y_c \quad z_c]^T$ défini dans le référentiel de la caméra. On obtient la projection 2D de ce point aux coordonnées $p_s = [u \quad v]^T$ grâce à l'équation suivante :

$$\begin{cases} u = u_0 + \dfrac{kf}{y_c} x_c \\ v = v_0 - \dfrac{lf}{y_c} z_c \end{cases}$$

La coordonnée z_c du point 3D est inversée dans le calcul de la coordonnée v car la direction de ces axes est opposée comme le montre la Figure 3-5. La division par y_c permet de normaliser les valeurs de x_c et z_c par rapport à la distance focale f. Sous forme

matricielle, on définit la matrice des paramètres intrinsèques I qui permet de projeter les points 3D sur la surface 2D de l'écran :

$$I = \begin{bmatrix} kf & u_0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & v_0 & -lf & 0 \end{bmatrix}$$

$$p_s = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u \\ 1 \\ v \end{bmatrix} = \frac{1}{y_c} \cdot I \begin{bmatrix} p_c \\ 1 \end{bmatrix}$$

Lorsqu'une caméra réelle est utilisée, celle-ci peut avoir des défauts de fabrication. Les axes du système de coordonnées de la surface de projection peuvent par exemple avoir un angle différent de 90°. On peut alors tenir compte de cet angle θ en complexifiant quelque peu la matrice des paramètres intrinsèques :

$$I = \begin{bmatrix} kf & u_0 & kf \cot\theta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & v_0 & -lf/\sin\theta & 0 \end{bmatrix}$$

Cependant, comme la surface de projection utilisée est rectangulaire, le sinus et la cotangente s'annulent et on retrouve alors la forme simple de la matrice des paramètres intrinsèques. En combinant les paramètres intrinsèques aux paramètres extrinsèques, on obtient la matrice de projection finale qui permet de convertir tout point défini dans le référentiel 3D du monde virtuel en projection sur la surface 2D de l'écran :

$$E = {}_C T_E \, {}_E T_H \, {}_H T_P \, {}_P T_V$$

$$I = \begin{bmatrix} kf & u_0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & v_0 & -lf & 0 \end{bmatrix}$$

$$p_s = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u \\ 1 \\ v \end{bmatrix} = \frac{1}{y_c} \cdot IE \begin{bmatrix} p_V \\ 1 \end{bmatrix}$$

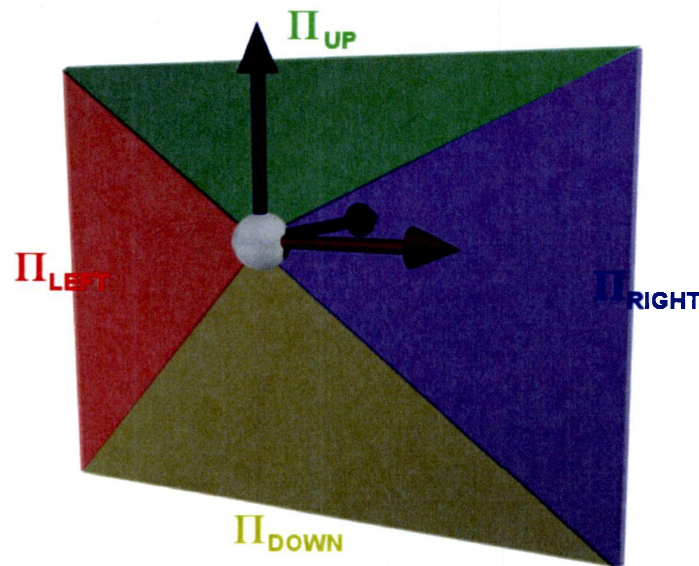


Figure 3-6: Plans formant le champ de vision de l'utilisateur imposé par la surface de projection

Finalement, pour déterminer si un point dans l'espace est affiché ou non sur la surface de rendu, il faut déterminer les FOV vertical et horizontal de la caméra en tenant compte des quatre coins de l'écran rectangulaire. En utilisant les deux coins adjacents à chacune des arêtes du rectangle et le centre optique, on peut former quatre plans qui déterminent le champ de vision imposé par l'écran comme le montre la Figure 3-6.

Soit les quatre plans Π_{LEFT}, Π_{UP}, Π_{RIGHT} et Π_{DOWN} dont la direction de la normale est toujours vers l'intérieur du champ de vision. Alors, le point virtuel 3D sera affiché sur l'écran si :

$$\Pi_n(P_V) \geq 0, \quad \forall\, n = LEFT, UP, RIGHT, DOWN$$

À l'aide du FOV, l'application pourra sauver beaucoup de temps lors du rendu de la scène. En effet, il sera possible d'éliminer tous les calculs de profondeur et de transparence pour la portion du monde virtuel qui n'a pas à être affichée lors de la phase de *culling* qui sera présentée plus tard dans ce chapitre. Cette astuce permettra donc d'améliorer grandement le taux de rafraîchissement de l'application.

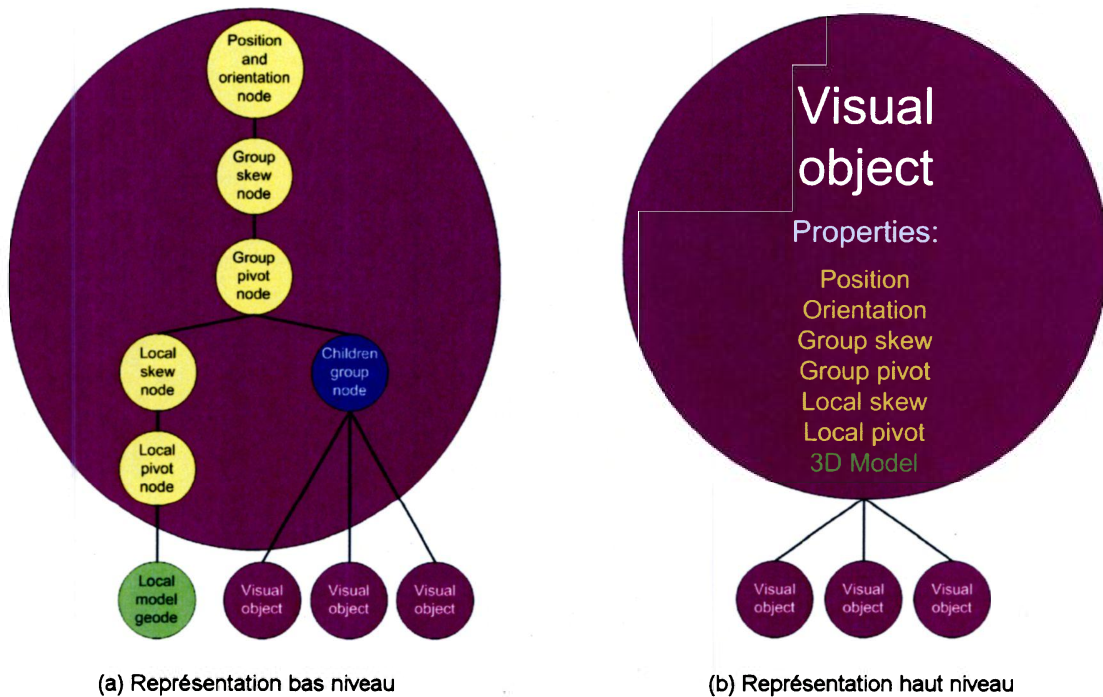
3.3 Encapsulation des propriétés dans la classe VisualObject

Le module `VisualWorld` regroupe les différentes fonctionnalités de l'engin graphique qui servent à construire l'arbre de rendu de l'environnement virtuel. La métaphore de l'objet visuel est utilisée afin d'encapsuler certains types de nœuds OSG à l'intérieur de la classe `VisualObject`. Celle-ci permettra donc de définir rapidement les différentes composante de la matrice de transformation homogène via chacun des nœuds de transformation de la structure de l'arbre de rendu: position, orientation, facteurs d'échelle, symétries et points pivots.

Un pivot permet de changer l'origine à partir duquel les rotations sont appliquées. En effet, le modèle 3D chargé dans un objet visuel possède une origine par défaut autour de laquelle les rotations sont effectuées. Les géométries du modèle tournent alors autour de ce point pivot. Les nœuds déterminant les points pivots de l'objet permettent d'appliquer une translation à l'origine du modèle 3D ce qui affectera la rotation. Ainsi, on peut modifier le point autour duquel les géométries pivotent dans l'espace. Pour une même matrice de rotation, le résultat graphique affiché à l'écran sera donc différent selon la position du point pivot sur le modèle 3D.

En plus des nœuds de transformation, on ajoute une géode à l'objet visuel pour afficher le modèle 3D qui le représente. Finalement, un nœud de type groupe est ajouté à l'objet visuel afin d'ajouter d'autres objets à sa hiérarchie comme fils. Ces deux derniers nœuds sont également encapsulés à l'intérieur de la classe `VisualObject` et peuvent donc être modifiés de façon transparente grâce à la métaphore de l'objet visuel utilisée ici.

Les objets visuels peuvent être représentés à deux niveaux d'abstraction comme le démontre la Figure 3-7. Au plus bas niveau dans la Figure 3-7(a), ils sont formés d'une série de nœuds `osg::MatrixTransform` (nœuds jaunes) qui définissent chacun des éléments de la transformation comme mentionné plus tôt. La classe `VisualObject`, qui dérive de la classe `osg::MatrixTransform` définit donc un sous-arbre et forme la racine de cette structure. La racine détermine la position et l'orientation de l'objet visuel et de ses fils dans l'espace. Les deux nœuds suivant la racine déterminent le *skew* de groupe (combine symétrie, facteur d'échelle et d'autres déformations quelconque de l'objet) et le



(a) Représentation bas niveau

(b) Représentation haut niveau

Figure 3-7: Deux niveaux d'abstraction de VisualObject

point pivot de groupe. Le terme « de groupe » signifie que ces deux paramètres affectent également le groupe d'objets visuels fils contenu dans le nœud `osg::Group` (nœud bleu) en plus du modèle 3D local Par contre, sur la branche gauche du nœud déterminant le point pivot de groupe, les deux derniers nœuds de transformation définissent le *skew* et le point pivot locaux qui n'affectent que le modèle local.

Ce sous-arbre forme une hiérarchie identique d'un objet visuel à l'autre où chacun des nœuds contrôle les différents paramètres de la transformation (position, orientation, symétries, facteurs d'échelle et points pivots) de façon transparente. L'abstraction de bas niveau illustrée à la Figure 3-7(a) représente bien l'ordre dans lequel la multiplication des matrices des différents nœuds de transformation doit être effectuée. En effet, comme présenté dans la section précédente, il suffit de multiplier les matrices de transformation relatives de la racine à la feuille pour obtenir la transformation globale. Ainsi, le rendu final du modèle local à l'objet contenu dans la géode (nœud vert) et du groupe d'objets fils (nœud bleu) est affecté par les nœuds de transformation (nœuds jaunes) qui définissent la position, l'orientation, les symétries, les facteurs d'échelle et les points pivots locaux et globaux de l'objet.

La représentation haut niveau de la Figure 3-7(b) présente un objet visuel comme un seul nœud possédant différentes propriétés (position, orientation, etc.) qui peuvent être modifiées pendant l'exécution. En effet, puisque la classe `VisualObject` encapsule un sous-arbre dont la structure demeure constante, on pourra modifier les différentes composantes de la transformation via les méthodes de la classe `VisualObject` plutôt que d'accéder directement aux différents nœuds de la structure qui sont privés. L'accès aux différents nœuds de transformation se fait donc de façon transparente via la classe `VisualObject`. Pour conserver un accès privé au nœud de groupe lors de l'ajout de fils à la hiérarchie de l'objet visuel, `VisualObject` surdéfinit la méthode `addChild()` de la classe `osg::MatrixTransform` afin d'ajouter les objets fils directement en dessous du nœud de groupe.

Grâce à l'encapsulation du sous-arbre dans la classe `VisualObject`, la structure interne de l'objet visuel ne sera jamais affectée par les manipulations effectuées sur l'arbre de rendu. Des objets visuels fils peuvent donc être ajoutés à la hiérarchie d'un objet visuel de façon transparente puisqu'ils sont en réalité constitués des différents types de nœuds

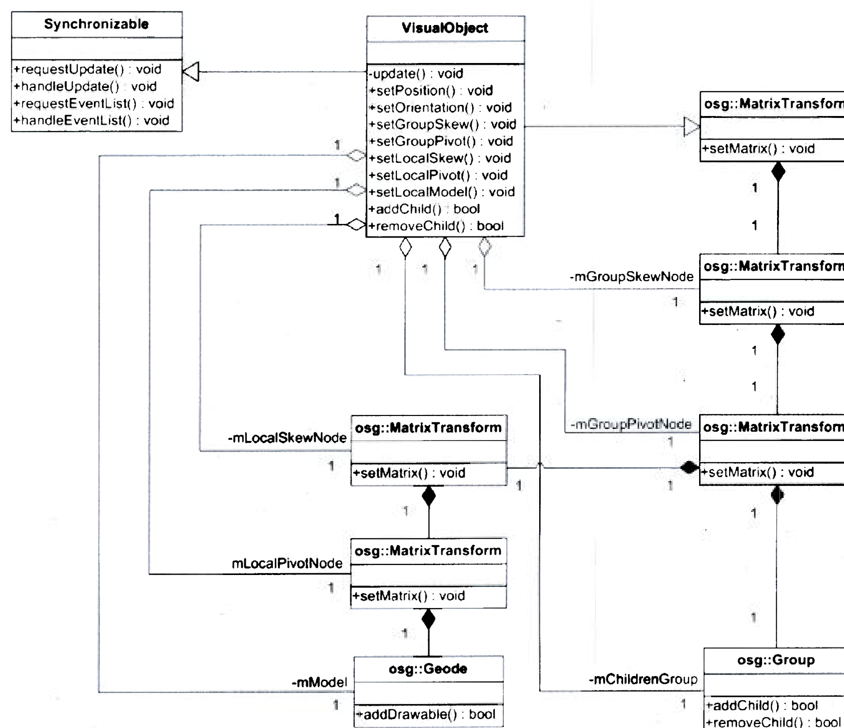


Figure 3-8: Représentation UML de l'objet visuel

d'OSG. Cependant, puisque le sous-arbre encapsulé par l'objet visuel est inséré au complet à chaque appel de la fonction `addChild()`, on peut faire abstraction de la structure interne de l'objet visuel qui ne changera pas et considérer l'objet comme un seul nœud dans l'arbre de rendu. En résumé, la classe `VisualObject` offre une interface qui permet de lire et de modifier les paramètres de chacun des nœuds de l'objet visuel comme s'il s'agissait en fait d'une propriété quelconque à l'intérieur d'un seul et unique nœud.

Finalement, la Figure 3-8 illustre une représentation de l'objet visuel selon le modèle UML. On y retrouve les différentes classes de nœuds OSG qui sont encapsulées dans la classe `VisualObject` et les méthodes utilisées pour la modification des composantes de la transformation, la gestion du modèle 3D affiché à l'écran et la manipulation d'objets fils de façon transparente.

Pour l'ajout de fils à un objet visuel, une modification mineure à la gestion de la structure de l'arbre de rendu a été définie afin d'éviter qu'un objet visuel existe en double dans l'environnement. En effet, par défaut, OSG offre la possibilité d'avoir plus d'un parent pour un même nœud. Cette particularité offre l'avantage de ne charger qu'une seule fois le nœud en question dans la mémoire et de l'afficher plusieurs fois dans la scène à des endroits différents. Lorsque l'arbre est parcouru de la racine aux feuilles pour le rendu, le nœud sera atteint plusieurs fois via ses parents et les transformations résultantes pourront être différentes selon les changements de référentiel appliqués par les générations antérieures. Ce comportement n'est par contre pas désirable dans le cas des objets visuels. En effet, puisqu'on veut synchroniser ces éléments avec les autres modules, ils se doivent d'être uniques et de ne posséder qu'une seule transformation absolue par rapport à l'origine du monde virtuel. Donc, lorsqu'un objet est ajouté comme fils à la hiérarchie d'un autre, il se doit d'éliminer la liaison avec son parent actuel. Ainsi, une seule copie de chaque objet visuel existera à la fois dans le monde virtuel. Cette particularité ne change en rien l'intégration de ces éléments à l'arbre de rendu d'OpenSceneGraph. Seules quelques opérations sont effectuées de façon transparente sur la structure lors de l'ajout d'un objet fils à la hiérarchie comme le montre la Figure 3-9.

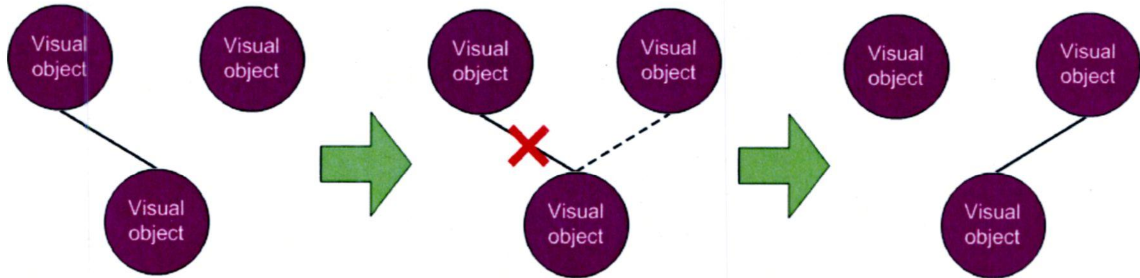


Figure 3-9: Suite des opérations pour l'ajout d'un fils à un VisualObject.

Bien que chaque objet visuel soit unique dans l'arbre de rendu, on peut tout de même exploiter la possibilité d'avoir plusieurs parents pour les géodes qui contiennent les modèles 3D afin d'optimiser l'utilisation de la mémoire. Dans la Figure 3-7(a), on voit bien que la géode n'appartient pas vraiment à l'objet visuel. En effet, même si la transformation globale de chacun des objets doit être unique, la représentation géométrique de ceux-ci peut par contre être partagée entre eux. Ainsi, deux objets distincts peuvent avoir une position et une orientation différentes et faire apparaître le modèle 3D d'une même géode deux fois à des endroits différents dans la scène. Puisque le chargement des modèles 3D peut être plus ou moins long selon la complexité de leur géométrie, la classe `VisualWorld` s'assure de ne charger qu'une seule fois le même modèle. Par la suite, lorsqu'un objet visuel est construit, il consulte d'abord la liste des géométries conservées en mémoire et ne crée une nouvelle géode que si le modèle demandé est introuvable. Ce processus est représenté à la Figure 3-10. Chaque élément de la classe `VisualObject` existe donc de façon distincte dans l'arbre de rendu afin d'éviter les confusions lorsqu'on veut obtenir leur transformation à partir de l'origine absolue pour la synchronisation. Par contre, les géodes contenant les modèles ayant déjà été chargés sont disponibles à l'ensemble des objets visuels créés par la suite afin d'optimiser le temps de chargement et l'utilisation de la mémoire.

Dans la Figure 3-10, on remarque que certains modèles ne sont en relation avec aucun objet visuel. Ce détail sert à illustrer le fait que la classe `VisualWorld` peut conserver plusieurs modèles dans une mémoire tampon même si aucune instance de ces modèles n'est affichée dans la scène. Cette fonctionnalité permettra d'augmenter la vitesse de création des objets visuels qui utiliseront ultérieurement les modèles déjà chargés. Par exemple, l'attente pour le chargement des modèles pourrait être effectuée lors du démarrage de l'application plutôt que de bloquer l'expérience virtuelle en cours d'exécution.

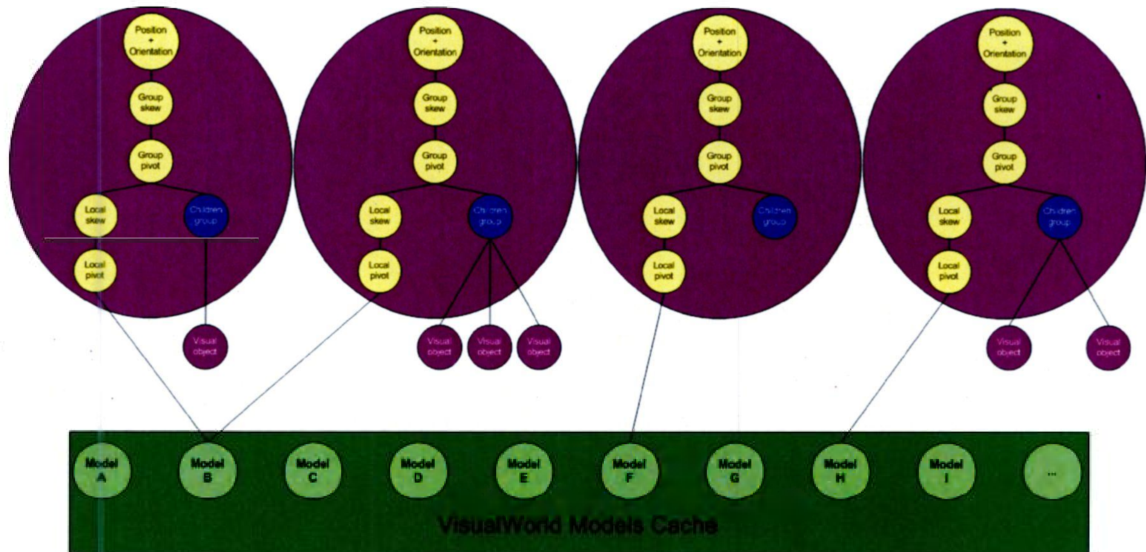


Figure 3-10: Principe de conservation des modèles 3D dans une mémoire cache

En plus du modèle 3D, d'autres éléments graphiques peuvent être ajoutés à la géode afin d'enrichir l'information visuelle disponible via l'objet. La boîte d'encadrement, appelée plus communément *bounding box* en anglais, forme le prisme rectangulaire de volume minimal qui englobe la totalité d'un ensemble de formes géométriques 3D quelconques. À partir de cette boîte, on peut facilement extraire les dimensions maximales d'un modèle complexe pour la longueur, la largeur et la hauteur. Ajoutée aux objets visuels, la *bounding box* permet d'obtenir rapidement une représentation grossière du modèle si la complexité géométrique n'est pas nécessaire pour une opération quelconque. Par exemple, le module haptique permet de créer des boîtes de collision simples afin d'alléger le traitement pour les calculs physiques plutôt que de charger la géométrie complexe des objets. Après la création d'un objet visuel, il sera donc possible de créer sa correspondance haptique sous une forme plus simple à l'aide de cette *bounding box*. De plus, on peut afficher cette boîte autour de l'objet avec une transparence pour offrir une rétroaction à l'utilisateur lors d'un événement comme la sélection ou la préhension (voir section 2.3.7). La classe `VisualObject` ajuste automatiquement les dimensions de la *bounding box* lorsque le modèle 3D est changé. De plus, une seconde boîte permet de regrouper l'objet visuel et l'ensemble de ses fils. Cette deuxième boîte est également ajustée lorsque la hiérarchie est modifiée. On peut donc facilement distinguer les groupes et les objets indépendants dans la scène.

C'est la représentation de haut niveau de l'objet visuel qui sera utilisée pour la suite du document. Bien que le modèle bas niveau permette de bien comprendre la structure et le comportement des objets, il n'offre cependant pas la vue d'ensemble qui sera nécessaire à la définition des concepts présentés dans les prochaines sections. Pour partager l'information à synchroniser entre les modules, cette représentation offre un format plus général qui peut être appliqué à n'importe quelle structure. En effet, les objets ne seront pas nécessairement organisés sous forme d'arbre de rendu dans tous les modules et ce choix facilitera donc la compréhension lors de l'intégration des modules dans l'application finale. L'intégration des modules sera présentée au chapitre 5.

3.4 Protection à l'accès asynchrone

Puisque le module visuel s'intègrera à un système plus complexe, il faut s'attendre à ce que la structure de l'arbre de rendu et les propriétés de chacun des objets puissent être modifiées à tout moment lors de l'exécution. Comme mentionné plus tôt, chaque module possède sa propre boucle de mise à jour indépendante des autres. La classe `VisualWorld` encapsule les fonctionnalités de VRJ et OSG pour former un *thread* qui s'occupe de gérer le rendu *off-axis* ainsi que l'affichage et la mise à jour de l'arbre de rendu. Puisque la librairie OSG ne permet pas à l'utilisateur d'apporter à tout moment des changements de façon sécuritaire à la structure de l'arbre, des mécanismes de protection à l'accès asynchrone ont été mis en place dans le module visuel.

La boucle d'exécution de VRJ (qui encapsule celle d'OSG) comprend trois phases : la mise à jour (*update*), la cueillette (*cull*) et l'affichage (*draw*). Pendant la phase d'*update*, l'arbre de rendu peut être modifié par l'utilisateur afin de changer dynamiquement l'apparence de la scène. Pendant la phase de *cull*, le processeur calcule différents éléments pour produire le rendu de façon ordonnée comme les effets de la lumière sur les surfaces, la transparence, l'obstruction, l'ordre d'affichage des objets, etc. Finalement, la phase *draw* permet de transmettre la projection 2D de la scène à la mémoire graphique pour être ensuite transmise au périphérique de rendu. Il est important de s'assurer que des *threads* externes à OSG ne viennent pas modifier l'arbre de rendu pendant les phases de *cull* et de *draw*. Sinon,

l'application générera des erreurs d'accès à la mémoire. C'est donc uniquement pendant la phase d'*update* que l'arbre pourra être modifié.

Afin de modifier dynamiquement le contenu de l'arbre de rendu, l'engin graphique fait appel à la méthode des fonctions *callback*. En général, une telle fonction est passée en paramètre à un *thread* et est appelée à un moment quelconque en réaction à un événement. Dans le cas d'OSG, des fonctions *callback* sont associées à chacun des nœuds et celles-ci seront exécutées une à une pendant le parcours de l'arbre lors de la phase d'*update*. Cette approche permet donc d'attribuer un comportement particulier à chaque type de nœud. Cependant, cette fonctionnalité n'ajoute aucune protection à l'accès asynchrone. Elle garantit par contre que les opérations contenues dans la fonction *callback* seront effectuées pendant la phase d'*update* de façon sécuritaire.

Ainsi, il suffit d'intégrer un mécanisme qui permet de conserver les paramètres d'une opération effectuée sur l'objet de façon asynchrone et de reporter l'exécution de cette opération au moment de l'appel à la fonction *callback*. Pour ce faire, une approche par émission et exécution de tâches a été développée et encapsulée à l'intérieur de la classe `VisualObject`. Cette méthode simple et performante est illustrée à la Figure 3-11. Tout d'abord, les modules externes à `VisualWorld` ont accès à l'interface publique des objets visuels afin de générer des tâches à tout moment pendant l'exécution. Pour un objet visuel, une tâche consiste par exemple à ajouter ou retirer un fils à sa hiérarchie, à modifier les différentes matrices des nœuds qui le constituent ou à modifier sa géométrie. Au lieu d'exécuter ces opérations directement, celles-ci sont plutôt insérées sous forme de tâches dans une liste protégée par un *mutex*. Un seul *thread* à la fois pourra donc ajouter ou retirer une tâche de cette liste. Lors de la phase d'*update* d'OSG, la fonction *callback* de chacun des objets est appelée et toutes les tâches lui étant associées sont exécutées dans le même ordre qu'elles ont été émises précédemment. Ce sont alors les méthodes privées de `VisualObject` qui appliquent les changements de façon sécuritaire. À cette étape, il faut s'assurer de bien respecter la chronologie des événements afin de conserver la cohérence dans l'arbre de rendu. Cette approche a été intégrée dans les classes du module `VisualWorld` et la modification des propriétés des objets peut donc être effectuée de façon asynchrone et transparente.

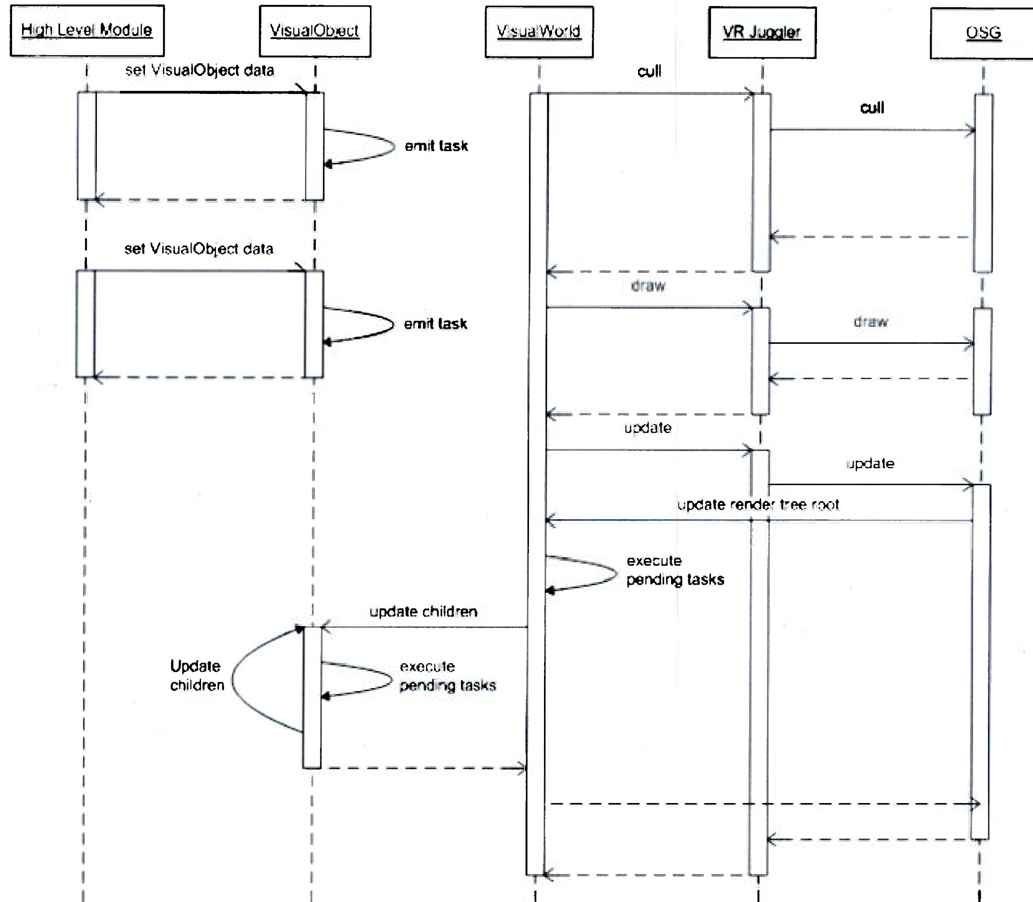


Figure 3-11: Mécanisme d'émission et d'exécution des tâches

L'approche par émission et exécution de tâches possède un seul désavantage majeur : son manque de flexibilité. En effet, pour chacune des opérations effectuées sur l'arbre, il faudra développer un nouveau type de tâche car les paramètres d'une tâche ne sont pas nécessairement les mêmes que ceux des autres tâches. De plus, chaque tâche définit une action précise à exécuter sur l'objet visuel qui correspond à une méthode spécifique dans la classe `VisualObject`. Par exemple, pour changer les facteurs d'échelle d'un objet, on a besoin d'un vecteur 3D et pour la couleur, d'un vecteur en quatre dimensions (rouge, vert, bleu et transparence). Par contre, pour changer le modèle à afficher, on utilise une chaîne de caractères spécifiant le répertoire et le fichier à charger. Finalement, une matrice de transformation homogène 4x4 est nécessaire pour définir une position et une orientation. D'autres tâches comme rendre l'objet visible ou non ne nécessitent aucun paramètre. Pour l'instant, le module visuel regroupe l'ensemble des tâches jugées importantes à la manipulation des objets visuels et leur structure de bas niveau présentée plus tôt. Pour

étendre les fonctionnalités de `VisualWorld`, il faudra par contre créer de nouvelles tâches. Cette limitation n'est cependant pas critique. En effet, malgré le manque de flexibilité, cette approche demeure simple, extensible et très rapide lors de l'exécution.

Une autre approche moins performante consiste à conserver une version tampon de l'arbre de rendu sur laquelle les changements sont effectués directement de façon asynchrone et sécuritaire. Lors de la phase d'*update*, la structure modifiée est d'abord copiée en entier pour remplacer l'originale et le parcours de chaque nœud est ensuite entamé pour l'exécution des fonctions *callback* et la mise à jour. Puisque l'arbre est copié à chaque fois, tout type de changement apporté à celui-ci sera tenu en compte. Cependant, cette approche rencontre rapidement un problème de performance lorsque la scène devient plus complexe. En effet, si le nombre d'objets virtuels augmente, l'arbre de rendu contient inévitablement plus de nœuds et l'opération de copie devient alors plus longue à effectuer. En général, il est très rare que tous les nœuds soient mis à jour en totalité dans une même itération. Les modifications sont souvent locales et la plupart des éléments demeurent donc intacts. La copie entière de l'arbre de rendu à chaque itération de la boucle d'OSG utiliserait donc beaucoup trop de ressources inutilement et la performance de l'application diminuerait rapidement en fonction du nombre d'objets dans la scène. Pour cette raison, l'approche par émission et exécution de tâches a été adoptée pour le projet. Cette technique permet de conserver la localité des modifications dans l'arbre de rendu en ne modifiant que les nœuds qui ont été mis à jour pendant l'itération. Ainsi, la performance de l'application finale sera maximisée au coût d'une faible perte en flexibilité.

3.5 Intégration au module de synchronisation

L'adaptation du processus de synchronisation dans le module visuel est relativement simple. En effet, comme présenté au chapitre 2, les objets visuels, en plus d'hériter des fonctionnalités du nœud de transformation dans l'arbre de rendu d'OSG, doivent également hériter de la classe `Synchronizable`. Ainsi, c'est pendant l'appel à la fonction *callback* de chaque objet visuel que la synchronisation sera exécutée. La boucle d'exécution complète du module de visualisation incluant l'affichage, l'exécution des tâches, la mise à

jour de la hiérarchie d'objets et la synchronisation est illustrée dans le diagramme séquentiel de la Figure 3-12.

Tout d'abord, les tâches en suspens dans la liste sont exécutées une à une. Ensuite, la première étape du processus de synchronisation, la lecture du tampon partagé, est effectuée afin d'obtenir la dernière mise à jour de la part des autres modules. Puis, pour la deuxième étape de synchronisation, si le parent de l'objet a été modifié pendant la même itération, ce dernier met à jour localement sa transformation globale avant d'effectuer la troisième étape, l'écriture dans le tampon partagé de l'`InterModuleObject` auquel il est associé. Afin de savoir si un parent a été modifié, ce dernier doit simplement activer un booléen lorsqu'une tâche ou le processus de synchronisation met à jour un paramètre de la

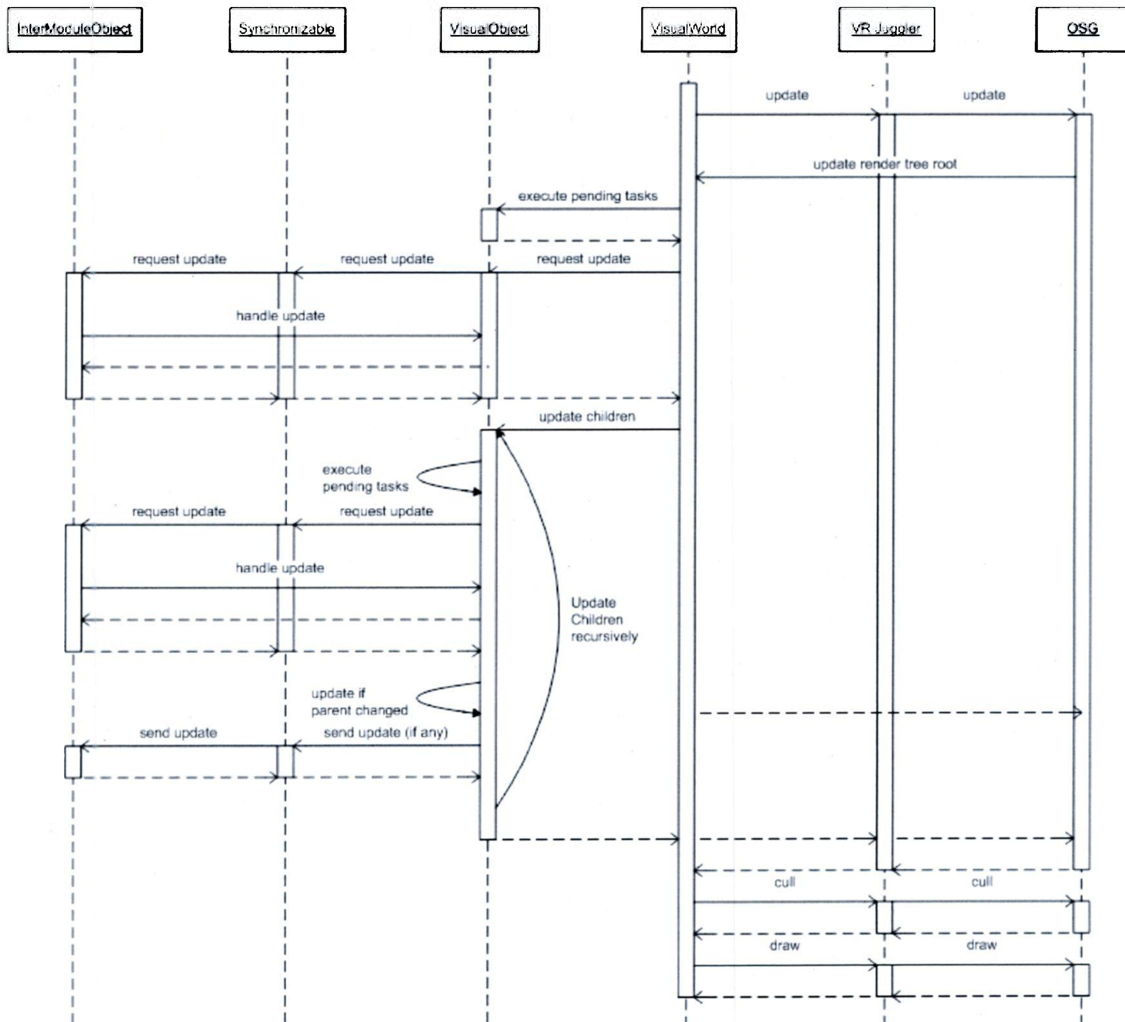


Figure 3-12: Adaptation du processus de synchronisation au module visuel

transformation. Ensuite, l'objet visuel fils possède un pointeur vers son parent ce qui lui permet de savoir s'il a été modifié pendant l'itération en cours. En fait, le module `VisualWorld` ne fera des opérations d'écriture dans le tampon partagé que s'il y a une hiérarchie d'objets visuels. Sinon, il ne fera que des lectures. En effet, la position des objets visuels dépend de la transformation de leur parent. Ainsi, si un parent est déplacé, ses fils sont également affectés par la modification. Cependant, les autres modules n'implémentent pas nécessairement une hiérarchie d'objets comme c'est le cas ici pour le module visuel et les objets peuvent être indépendants entre eux. Lorsqu'un objet visuel possédant des fils est mis à jour, ces derniers doivent donc déterminer leur nouvelle position et synchroniser cette information avec les autres modules afin de conserver la cohérence de la scène. Une fois la phase *update* terminée, les phases *cull* et *draw* de VRJ et OSG sont exécutées normalement.

Chapitre 4 - Distribution sur un réseau

En plus des propriétés visuelle et haptique, un module de distribution doit maintenant être intégré au système afin de rendre l'application distribuée sur un réseau. Cette nouvelle propriété permettra d'ajouter deux fonctionnalités au système : rendre l'environnement multi-utilisateurs et augmenter le parallélisme de l'application. Afin de rendre l'application multi-usagers, il suffit de répliquer les modules sensoriels sur les différentes machines utilisées par les utilisateurs. Pour augmenter le parallélisme du système, il faudra plutôt distribuer les modules sur plusieurs machines qui partageront la tâche d'exécution des différents modules sensoriels pour un ou plusieurs utilisateurs. Ces deux fonctionnalités permettront d'ajouter une grande flexibilité pour la configuration de la topologie de l'application finale.

Comme il a été mentionné dans la description du projet au chapitre 1, chaque instance de l'univers virtuel forme une application complète qui charge ses propres modules localement pour remplir les fonctionnalités nécessaires selon le contexte. Les modules de synchronisation et d'intégration sont essentiels au fonctionnement du système et seront donc obligatoirement chargés pour toutes les instances. Lorsque plusieurs instances sont intégrées au système, le module de distribution devient alors également nécessaire pour établir la communication entre elles. Chaque instance qui prend part au monde virtuel pourra donc charger les modules sensoriels et fonctionnels nécessaires et définir leurs paramètres localement. À l'aide des deux fonctionnalités du module de distribution, il sera alors possible de distribuer les modules sur un réseau et de centraliser la gestion de certaines fonctionnalités sur une seule machine. Par exemple, le module visuel sera généralement associé à toute instance où un utilisateur prendra place. Par contre, le module haptique sera généralement sur une seule machine pour assurer la cohérence lors de l'application des lois de la physique sur l'ensemble du système. De plus, il sera possible pour l'utilisateur d'interagir à distance avec l'environnement virtuel via une interface utilisée dans une autre instance de l'application. Ces fonctionnalités seront décrites plus en détails au chapitre 5. La Figure 4-1 montre une représentation générale du module de distribution et son intégration à l'univers virtuel final.

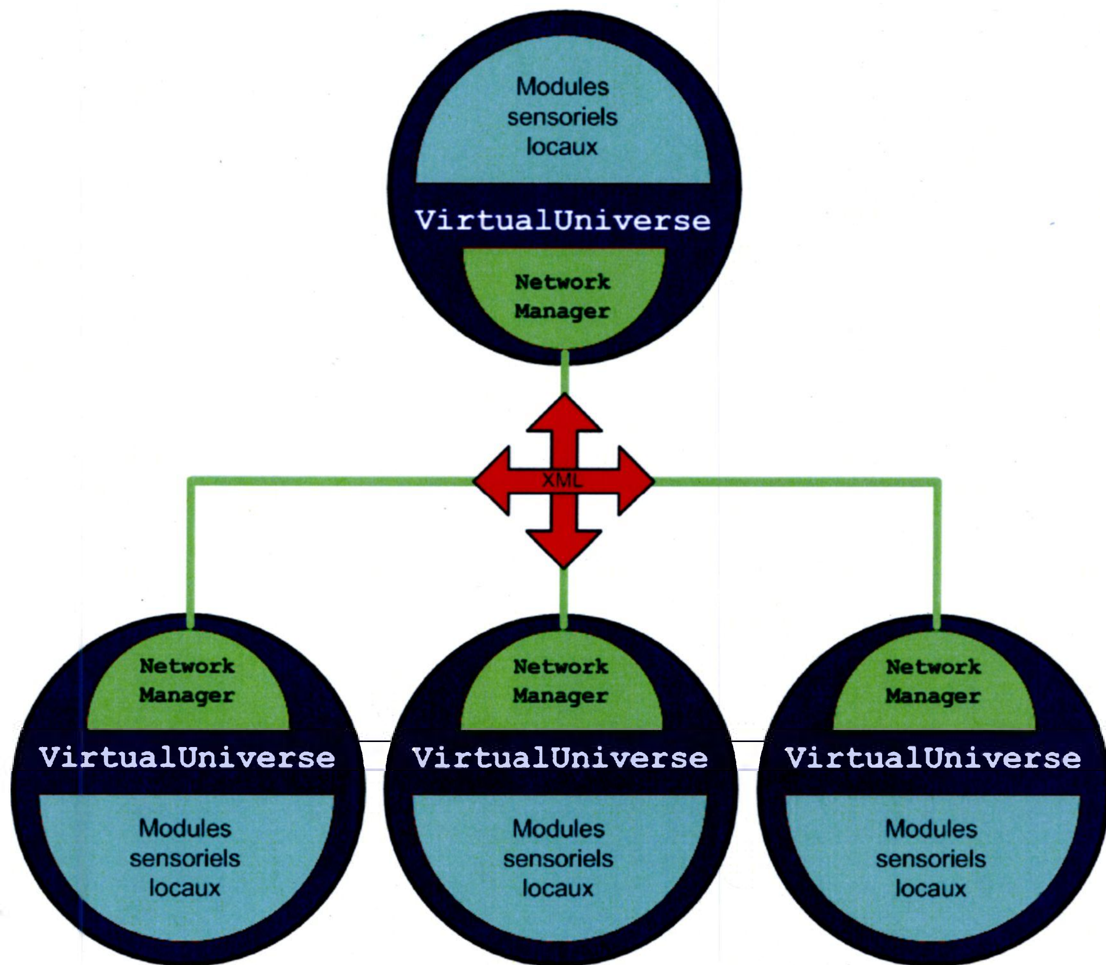


Figure 4-1: Représentation générale du module de distribution

De façon similaire aux autres sous-systèmes, le module de distribution, nommé `NetworkManager`, adhère aux principes de synchronisation présentés au chapitre 2. En effet, dans son instance locale, il partage avec les autres modules les données de position, d'orientation et de facteurs d'échelle provenant des autres instances du réseau selon le même processus. Cependant, en plus de la synchronisation locale, le module de distribution procure également la fonctionnalité de partager cette information sur le réseau en représentant les objets et leurs mises à jour par des messages en format XML qui sont expédiés aux autres instances du système via les protocoles « *transmission control protocol* » (TCP) et « *user datagram protocol* « (UDP). Cette communication hybride, présentée à la prochaine section, permettra de tirer les avantages des deux approches de transmission. Puis, les deux sections suivantes expliquent les techniques utilisées pour

l'enregistrement et la propagation de l'adresse des différentes instances de l'application ainsi que pour la création et la classification des objets pour un accès optimal. Finalement, puisque certaines opérations comme la création et la destruction d'objets peuvent avoir des répercussions sur les autres modules du système, le module de distribution doit nécessairement pouvoir faire appel à des fonctionnalités de plus haut niveau pour effectuer ces actions. Pour ce faire, une approche de design par *proxy* a été utilisée pour permettre au module de distribution d'utiliser certaines fonctionnalités du module d'intégration sans connaître à l'avance la définition de ce dernier. Cette méthode, présentée à la dernière section de ce chapitre, conserve donc l'encapsulation de bas niveau du module de distribution tout en permettant un accès transparent à un sous-système quelconque de plus haut niveau qui joue le rôle du module d'intégration dans le cas de l'application développée dans le cadre de ce projet.

4.1 Communication réseau hybride

Il existe deux protocoles très répandus pour la communication réseau : le TCP et le UDP. Les deux méthodes possèdent des avantages et des inconvénients et leur utilisation dépend donc de l'application qu'on veut en faire. Puisque l'environnement virtuel développé peut distribuer une panoplie d'information de tous genres, une approche hybride a été employée pour le projet. En effet, le TCP possède l'avantage d'être fiable car il assure l'arrivée des paquets à destination et leur ordonnancement temporel. Cependant, il sera généralement plus lent à cause de cette propriété. En contrepartie, le protocole UDP est beaucoup plus rapide, mais la réception et la chronologie des paquets ne sont pas assurées. Le TCP sera donc utilisé pour les messages critiques dont la cohérence de la scène dépend directement et le UDP servira pour les mises à jour fréquentes qui sont moins cruciales et qui proviennent du processus de synchronisation. Afin de représenter les messages de façon générique, le format XML a été intégré au module réseau. Cette approche apporte encore une fois de la flexibilité au système mais aussi une excellente portabilité car le XML est très répandu.

4.1.1 Mises à jour et synchronisation rapides par le protocole UDP

Les données de position et d'orientation partagées avec le module de synchronisation sont transmises pratiquement sans interruption sur le réseau. En effet, le monde haptique met à

jour tous les objets sur lesquels une force est appliquée. Par exemple, l'utilisateur peut déplacer des objets avec ses mains, ces objets peuvent également effectuer une chute lorsqu'ils sont lâchés ou se collisionner entre eux. Il faut donc utiliser un mécanisme de communication rapide pour transmettre ces fréquentes informations : le protocole UDP.

Lors du processus de synchronisation, la requête de mise à jour permet d'obtenir l'ensemble des transformations et de créer des paquets UDP qui seront expédiés aux autres instances de l'application pour qu'elles puissent procéder à la mise à jour locale de cette information. Ces paquets contiennent des messages XML dont la structure avec balises (ou *tags* en anglais) semblable au langage HTML permet de bien organiser l'information et d'établir un format standard pour la représentation des données entre l'expéditeur et le destinataire.

Afin d'ordonner temporellement les paquets avec ce protocole qui n'assure pas la chronologie lors de la réception, on doit accompagner chaque message d'un *timestamp* fourni par le module de synchronisation. Insérée au message XML, cette valeur d'identification temporelle permettra aux récipiendaires des paquets d'ignorer ceux dont la valeur du *timestamp* est périmée par rapport à celle de paquets déjà reçus et ainsi d'ordonner les événements correctement. Même si certains messages sont perdus et n'arrivent pas à destination pendant la transmission de la trajectoire d'un objet, les mises à jour subséquentes permettront de maintenir une synchronisation acceptable malgré une brève saccade de quelques itérations.

4.1.2 Cohérence assurée par le protocole TCP

Certains événements sont critiques à la cohérence de la scène pour tous les utilisateurs et ils doivent donc être reçus sans faute par le destinataire. Par exemple, lorsqu'un objet termine sa trajectoire, il n'y aurait aucun moyen de récupérer la dernière position si celle-ci était perdue lors de la transmission via le protocole UDP. Puisque certains événements comme la création ou la destruction d'un objet particulier ne se produisent qu'une seule fois pendant une expérience virtuelle, il est impératif que ces événements soient envoyés via un protocole de communication sécuritaire et fiable : le TCP.

Ainsi, même si cette méthode ajoute quelques délais pour assurer et ordonner la réception de l'information grâce à un processus de confirmation et de partitionnement des paquets, ces tâches supplémentaires permettent de conserver la cohérence de la scène à tout moment lorsque cela s'avère nécessaire. Dans l'application développée, la plupart des événements du module de synchronisation sont jugés critiques. De plus, les opérations et les paramètres pour la création et la destruction d'objets et les changements sur la structure des objets (arbre, liste, etc.) sont également nécessaires à la cohérence du monde virtuel partagé. Finalement, lorsqu'un élément termine une trajectoire ou se téléporte dans l'environnement, sa position doit aussi être envoyée de façon sécuritaire par le protocole TCP. En fait, seules les mises à jour fréquentes comme celles du module haptique sont envoyées par UDP car la cohérence demeurera assurée même si la continuité de l'information est perdue momentanément.

4.2 Organisation réseau hybride

Il y a plusieurs façons d'organiser la structure réseau des différentes instances de l'application. Tout d'abord, l'approche *par serveur* consiste à confier le rôle de gestionnaire de la scène virtuelle à une seule machine. Cette méthode est pertinente si on veut conserver la cohérence de l'information car c'est le serveur qui aura la responsabilité de distribuer celle-ci.

Cependant, si tous les messages transigent par le serveur, celui-ci pourra devenir surchargé et des délais dus au processus de relais se manifesteront. L'autre approche pour structurer le réseau, le *broadcast*, consiste à propager les messages directement entre tous les nœuds du système. Cette méthode implique que chaque instance connaisse l'existence des autres instances qui sont connectées. Ainsi lorsque l'une d'entre elles envoie un message, ce dernier sera transmis à toutes les autres instances sans transiger par un serveur. Cette technique est idéale pour des envois fréquents car elle n'engorge pas une seule machine devant relayer toutes les entrées et sorties du système.

Puisque l'application développée implique de conserver certaines données et de faire des mises à jour fréquentes, une approche hybride a encore une fois été choisie pour l'organisation de la structure du réseau. Les messages critiques comme la création d'objets

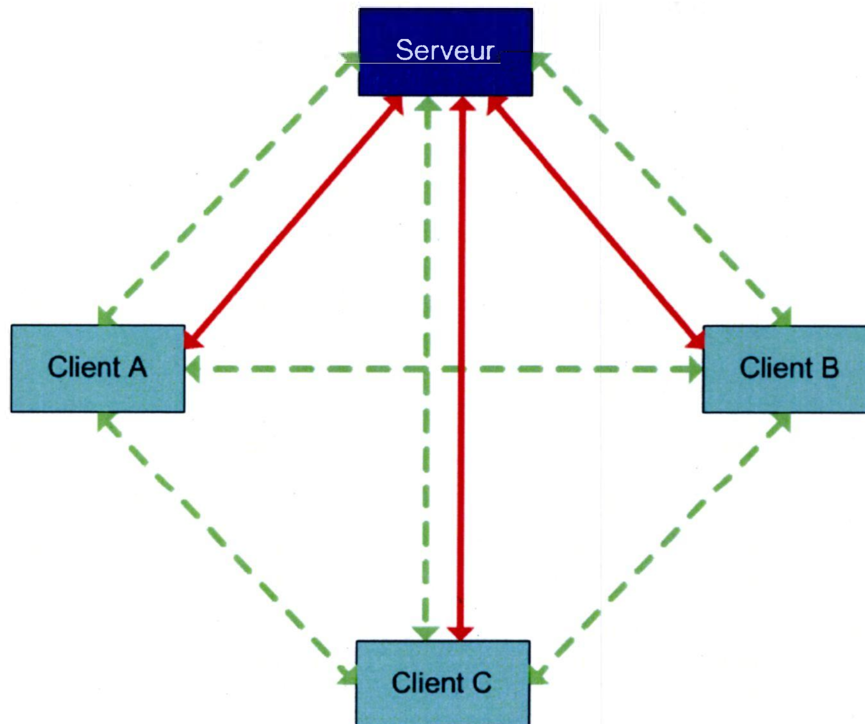


Figure 4-2: Modèle de connexions de la structure réseau hybride

et l'émission d'événements qui sont envoyés par TCP doivent transiger par un serveur pour être distribués à toutes les instances du système. Ainsi, la gestion du temps de simulation et de l'organisation des objets virtuels est centralisée sur une seule machine pour assurer la cohérence. Par contre, les mises à jour fréquentes du module de synchronisation qui sont expédiées via le protocole UDP sont plutôt transmises avec la méthode du *broadcast* pour ne pas engorger le serveur. En fait, le gestionnaire devient alors un destinataire comme les autres. La Figure 4-2 montre un exemple de réseau à trois clients avec un serveur. Les lignes pleines en rouge représentent les liens TCP transportant les messages critiques via le serveur. Les segments pointillés verts qui interconnectent directement tous les nœuds entre eux définissent les connexions UDP.

Avant de se connecter à l'environnement virtuel, une nouvelle instance doit d'abord synchroniser la valeur de son *timestamp* avec le serveur et y enregistrer ses adresses TCP et UDP. La confirmation de création du nouveau lien UDP est ensuite transmise à tous les nœuds actuellement enregistrés afin que chacun puisse l'ajouter à sa liste d'envoi pour le *broadcast* des mises à jour. Les autres messages plus critiques seront plutôt relayés à toutes

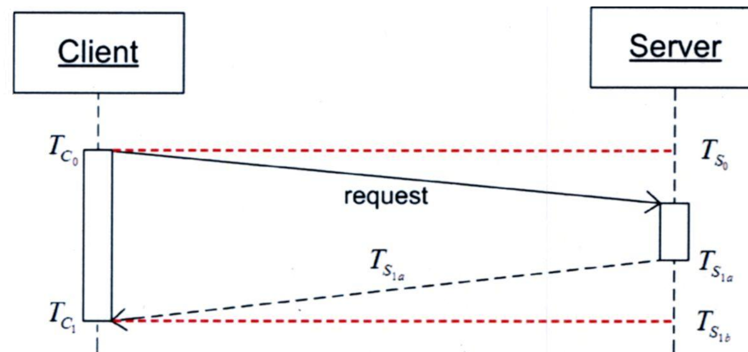


Figure 4-3: Algorithme de Cristian

les instances par la méthode du serveur via le protocole TCP. Ces trois aspects seront présentés plus en détails dans les prochains paragraphes.

4.2.1 Synchronisation des clients

Pour distribuer l'application sur un réseau, il faut d'abord démarrer une première instance de l'univers virtuel sur une machine qui sera dédiée comme serveur. Par la suite, en connaissant d'avance les adresses TCP et UDP de ce serveur, des clients pourront se joindre à l'environnement virtuel. Cependant, avant de s'enregistrer auprès du serveur, les clients doivent d'abord synchroniser la valeur du *timestamp* contenue dans le module de synchronisation. Ce processus est essentiel afin d'assurer la cohérence et l'ordonnancement des événements.

L'approche qui a été choisie pour la synchronisation du *timestamp* est une version modifiée de l'algorithme de Cristian présenté dans (Amir 2006). Dans sa version originale présentée à la Figure 4-3, l'algorithme de Cristian consiste simplement à faire une requête au serveur de temps qui retourne la valeur demandée. À partir des valeurs temporelles T_{C_0} et T_{C_1} qui correspondent respectivement aux moments d'envoi et de réception des messages, le client peut approximer une nouvelle valeur de T_{C_1} qui correspondra davantage au temps « réel » grâce au temps $T_{S_{1a}}$ obtenu du serveur:

$$T_{C_1} := T_{S_{1a}} + \frac{T_{C_1} - T_{C_0}}{2}$$

Le résultat de $T_{C_1} - T_{C_0}$ donne le délai de transmission exact pour cet aller-retour spécifique. En divisant par deux cette valeur, on estime le délai de transmission $T_{S_{1b}} - T_{S_{1a}}$ pour le retour du message, du serveur au client. La valeur de $T_{S_{1b}}$ correspond au moment réel sur le serveur où le message de réponse est reçu par le client. Cependant, cette valeur est impossible à déterminer pour le client car les délais de transmission sur un réseau sont variables selon la topologie et le niveau de trafic et donc imprévisibles. Ainsi, par l'équation précédente, la nouvelle valeur de T_{C_1} ne donne qu'une approximation de $T_{S_{1b}}$ et non pas la valeur exacte. L'algorithme de Cristian suppose une transmission symétrique. C'est-à-dire que les délais de transmission pour l'aller et le retour du message sont supposés égaux. Comme expliqué ici, ce ne sera généralement pas le cas, mais l'approximation du temps de transmission du serveur vers le client en divisant par deux le temps de l'aller-retour est le meilleur estimé à faire. Évidemment, répéter plusieurs fois l'algorithme de Cristian ne permet pas une meilleure synchronisation car on calcule à chaque fois une nouvelle valeur temporelle qui ne tient pas compte des précédentes.

Afin d'obtenir une meilleure synchronisation, la version modifiée de cet algorithme, illustrée à la Figure 4-4, propose plutôt d'estimer le décalage (*offset* en anglais) entre le serveur et le client en considérant plusieurs itérations de l'algorithme de Cristian de base. En supposant l'application de M itérations de l'algorithme de Cristian, on peut estimer à chaque itération N le temps de transmission entre le serveur et le client en calculant la moitié du temps de l'aller-retour :

$$\frac{T_{C_N} - T_{C_{N-1}}}{2} \cong T_{S_{Nb}} - T_{S_{Na}}$$

À partir de cette équation, on calcule à chaque itération une approximation de la valeur du décalage entre le *timestamp* du serveur et celui du client :

$$Offset_N = T_{C_N} - \left(T_{S_{Na}} + \frac{T_{C_N} - T_{C_{N-1}}}{2} \right) \cong T_{C_N} - T_{S_{Nb}}$$

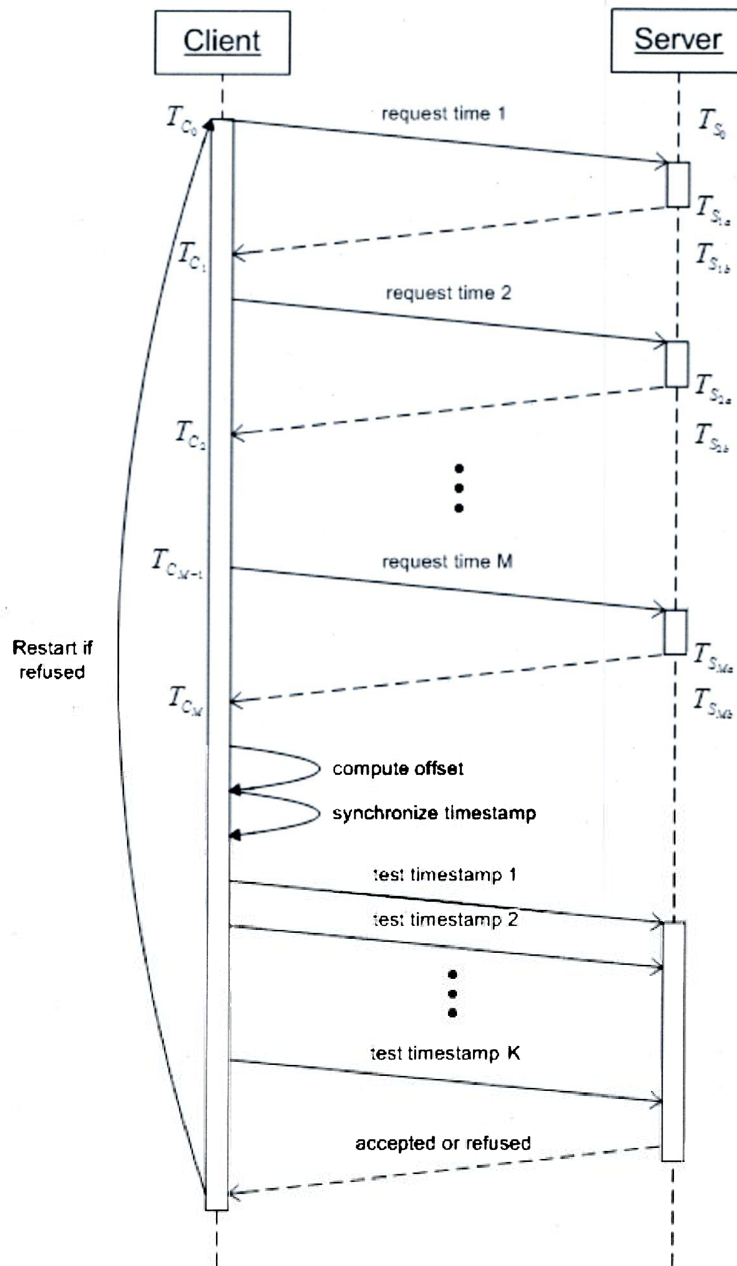


Figure 4-4: Algorithme de Cristian modifié

On conserve également le temps de transmission aller-retour (*round-trip time* en anglais) de la requête afin d'ordonner en ordre croissant, selon cette valeur, les estimations du décalage correspondantes obtenues pendant la totalité des itérations :

$$RoundTrip_N = T_{C_N} - T_{C_{N-1}}$$

Après les M itérations de l'algorithme, on obtient une liste d'*offsets* ordonnée selon l'ordre croissant des valeurs des différents temps de *round-trip* correspondants. On retranche ensuite la moitié supérieure de cette liste afin d'ignorer les itérations où le temps de transmission comportait trop de latence. Finalement, on estime par la valeur $\hat{T}_S$ le temps « réel » du serveur T_S en calculant la moyenne des décalages qui restent et en ajoutant cette valeur au temps actuel du client T_C :

$$\hat{T}_S = T_C + \frac{\sum_{N=0}^{M/2-1} Offset_N}{M/2}$$

$$\hat{T}_S \cong T_S$$

Puis, afin de s'assurer que la synchronisation a été effectuée correctement, une dernière étape a été ajoutée à l'algorithme de Cristian modifié afin de tester les nouvelles valeurs temporelles maintenant émises par le client. Pour y arriver, il suffit au client d'envoyer K paquets de test au serveur contenant la nouvelle valeur du *timestamp*. Si une majorité des *timestamps* provenant du client ont une différence inférieure à un certain seuil avec les *timestamps* du serveur, le client sera accepté et il pourra prendre part à l'expérience virtuelle partagée. Dans le cas contraire, il devra recommencer à nouveau l'algorithme de Cristian modifié.

4.2.2 Enregistrement des clients

Une fois synchronisé avec le serveur, le client doit s'enregistrer auprès de ce dernier en lui transmettant ses adresses UDP et TCP. Le serveur propage ensuite la nouvelle adresse UDP à tous les autres clients déjà connectés et il transmet la liste des adresses UDP de ces derniers à la nouvelle instance. Ainsi, le *broadcast* des mises à jour pourra être effectué entre toutes les machines sans transiger par le serveur qui devient alors un destinataire comme les autres. Le processus de *broadcast* et la propagation de la nouvelle adresse UDP sont montrés à la Figure 4-5.

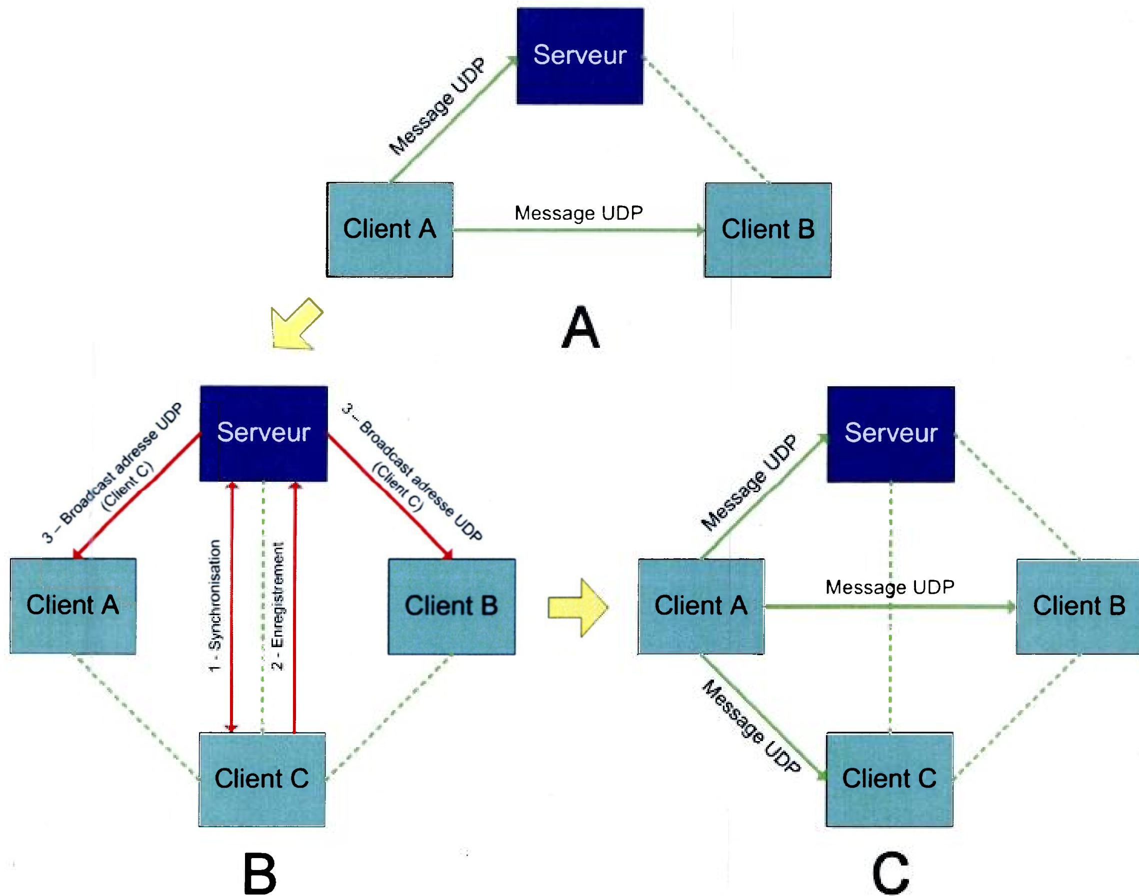


Figure 4-5: Propagation de l'adresse UDP pour mises à jour en broadcast

Il est inutile pour les clients de connaître l'adresse TCP des autres instances de l'application mis à part le serveur car les messages transmis par ce protocole de communication sont toujours relayés par le serveur. L'adresse du serveur est déjà connue au démarrage car elle est nécessaire pour la connexion des clients. Ainsi, malgré l'ajout de nouvelles instances à l'environnement virtuel, la transmission de messages critiques se fera toujours de façon transparente via le serveur comme le montre la Figure 4-6.

4.3 Création et classement des objets

Comme pour les autres sous-systèmes, le module réseau possède sa propre représentation des objets du monde virtuel encapsulée dans la classe `NetworkObject`. Ce nouveau type d'objet partage également les données de position, d'orientation et de facteurs d'échelle avec les autres modules et il permet de transmettre ces informations sur le réseau sous forme de message XML via le protocole UDP. Seules les mises à jour non-critiques d'un

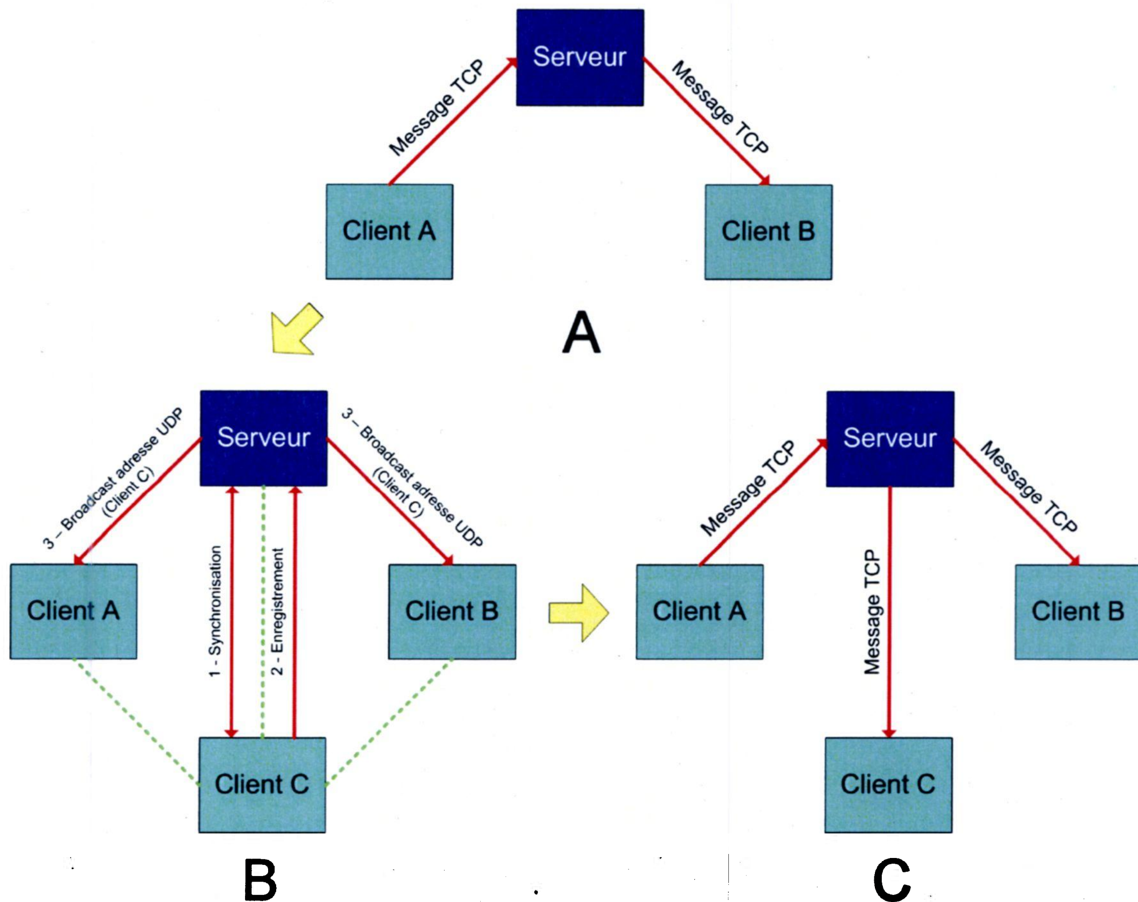


Figure 4-6: Transmissions de messages TCP via serveur

objet sont transmises par le protocole UDP. Les autres événements comme la création et la destruction d'objets réseau sont critiques à la cohérence de la scène et de tels messages doivent nécessairement transiger par le serveur via le protocole TCP.

En plus d'assurer la cohérence de la scène, la centralisation des événements critiques sur un serveur permet également d'implanter facilement la méthode de classement des objets qui a été choisie pour le module réseau. Le module de distribution (plus précisément la classe `NetworkManager` qui permet la gestion du réseau) contient une liste d'objets `NetworkObject` mise à jour en boucle pendant l'exécution de son *thread*. Chaque instance de l'application possède une copie identique de cette liste, mais c'est le serveur qui possède la responsabilité exclusive de gérer l'ordre dans lequel sont classés les différents objets dans cette liste. En effet, lorsqu'un objet est créé, le serveur lui attribue un index particulier et cette valeur détermine la position qu'occupe l'objet dans la liste. Puisqu'on ne

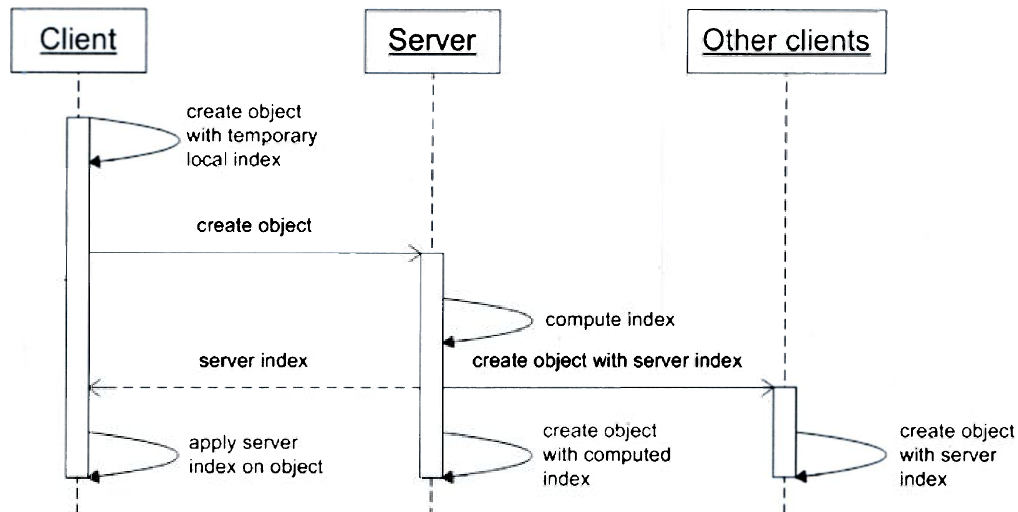


Figure 4-7: Création et classement des objets

connaît pas d'avance l'adresse qu'occupera un objet en mémoire, c'est l'index qui sert de référence pour déterminer l'objet sur lequel on veut effectuer une opération. Lors de l'envoi d'un message XML pour effectuer une action quelconque sur un objet particulier, cet index, contenu dans l'une des balises du message, permettra un accès rapide à l'objet `NetworkObject` dans la liste pour la mise à jour, l'ajout de fils ou la destruction. L'ordonnancement des objets dans la liste doit donc être identique sur chacune des instances de l'application afin de permettre un accès rapide et cohérent aux objets concernés lors de la réception d'un message. Ce sera la responsabilité du serveur de déterminer l'ordre de la liste des objets `NetworkObject`.

Le processus de création et de classement des objets adopté par un client est illustré à la Figure 4-7. Tout d'abord, le client crée localement l'objet avec un index temporaire également déterminé localement. Par la suite, le message XML critique annonçant l'événement de création d'objet est envoyé au serveur via le protocole TCP. Le serveur crée alors à son tour une instance locale de l'objet et lui attribue un index selon l'état actuel de la liste d'objets centrale. Le serveur retourne ensuite cet index au client qui vient tout juste de créer l'objet et ce dernier doit l'appliquer à l'objet et modifier sa position dans sa liste locale en conséquence. Finalement, le serveur envoie un message XML annonçant la création de l'objet avec l'index modifié aux autres clients qui participent à l'expérience virtuelle. L'index et la position de l'objet dans la liste sont alors imposés par le serveur. Par la suite, l'index assigné par le serveur pourra être utilisé comme référence à l'objet pour un

accès rapide à la liste locale plutôt que de faire une recherche exhaustive à chaque itération. Lorsque c'est le serveur qui crée un objet, celui-ci décide simplement de la valeur de l'index et envoie le message de création à tous les clients sans intermédiaire.

Il est à noter que ce processus de réplication des index est géré de façon transparente dans le module de distribution afin d'établir une correspondance entre les objets partagés entre plusieurs instances. Lorsqu'un message XML est transmis à la classe `NetworkManager`, cette dernière transmet la mise à jour à l'objet `NetworkObject` correspondant grâce à l'index contenu dans le message. Ensuite, lors de la boucle d'exécution, l'objet en question extrait les données des différentes balises du message XML et les convertit dans le format neutre présenté au chapitre 2 pour ensuite transmettre cette information au module de synchronisation qui se chargera de distribuer le tout aux autres modules. Le processus inverse fonctionne sous le même principe. Lorsqu'une mise à jour est reçue par le `NetworkObject` à partir du module de synchronisation, un message XML contenant l'index de l'objet et les données de la mise à jour sera généré et expédié aux autres instances du réseau qui pourront à leur tour accéder rapidement à l'objet dans leur liste locale qui sera identique.

4.4 Fonctionnalités et propriétés de haut niveau

Les propriétés communes comme la position, l'orientation, les facteurs d'échelle, les symétries et les points pivot à l'intérieur des objets `NetworkObject` sont partagées et synchronisées directement dans le module réseau via le module de synchronisation présenté au chapitre 2. En effet, puisque `NetworkObject` dérive de la classe `Synchronizable`, les instances de cette classe doivent par conséquent partager avec tous les modules locaux l'information qu'ils reçoivent par réseau. Cependant, la création d'objets avec ces propriétés communes demeure très générique et peu flexible. Il serait intéressant de pouvoir partager des propriétés quelconques comme des propriétés sensorielles et d'autres propriétés de haut niveau. En effet, les objets `Synchronizable` et `InterModuleObject` présentés au chapitre 2 implémentent un processus d'événement qui peut facilement être distribué sur un réseau. De plus, les objets `VisualObject` du module visuel présentés au chapitre 3, les objets virtuels du module

d'intégration qui sera présenté au prochain chapitre ainsi que les objets haptiques du module haptique présenté dans (duTremblay à venir) possèdent plusieurs autres propriétés sensorielles et de haut niveau locales à chacun de ces modules. Afin d'étendre davantage la flexibilité de l'application il serait intéressant de pouvoir partager ces propriétés de haut niveau entre les différentes instances. Il deviendrait alors possible de modifier et de synchroniser toutes les propriétés d'un objet comme son modèle, sa transparence, ses fils (dans le cas où le module utiliserait une structure de classement semblable à un arbre de rendu), son comportement, etc.

Pour arriver à partager les propriétés de haut niveau des différents modules, il suffit d'ajouter des balises contenant des informations de haut niveau dans le message XML. Le module d'intégration, module de haut niveau permettant l'encapsulation du module de distribution dans le système final, devra être conçu pour gérer ces nouveaux *tags*. Le module d'intégration sera donc responsable de l'insertion de *tags* de haut niveau dans les messages envoyés via le module de distribution lors de la création d'un objet ou de tout autre événement modifiant une propriété de haut niveau. Il devra également analyser les *tags* de haut niveau lors de la réception de messages et effectuer le changement local approprié sur les propriétés de haut niveau.

Lors de la transmission et de la réception de messages contenant des balises de haut niveau, le module de distribution ignore le contenu de ces *tags* car il n'en connaît pas la nature (ni la structure). Par contre, il relaie cette information au module de haut niveau et il ne sert alors que d'intermédiaire pour la transmission de commandes quelconques entre deux modules de haut niveau appartenant à deux instances différentes sur le réseau. Les détails sur la manipulation des balises XML de haut niveau par le module d'intégration seront présentés dans le chapitre 5.

Afin de s'assurer que le module de distribution demeure générique, il faut utiliser une approche de design logiciel qui permettra une communication avec le module d'intégration sans toutefois en connaître la définition lors de la compilation. Puisque le module d'intégration doit déjà être développé en tenant compte des *tags* de haut niveau dans les messages XML, l'approche du *proxy* semble toute désignée pour rendre le module de distribution générique. En effet, selon cette technique, il suffit de définir une classe

Figure 4-8: Accès à module spécialisé quelconque par un proxy

abstraite qui sert de *proxy* dans le module à généraliser pour communiquer avec un module spécialisé quelconque dont la définition est inconnue. Lors de la conception du module spécialisé, il suffira de définir une classe qui dérive du *proxy* générique et qui spécialise son comportement face aux messages de haut niveau selon les fonctionnalités particulières du module en question.

La Figure 4-8 montre la structure UML qui représente de façon générale la technique du *proxy* appliquée au projet. Tout d'abord, le module de distribution définit deux *proxies* pour la communication avec le module spécialisé : un pour la classe `NetworkObject`, le `HighLevelObjectProxy`, et un autre pour le `NetworkManager`, le `HighLevelManagerProxy`. Puis, un module quelconque voulant s'interfacer avec le module de distribution devra définir deux *proxies* spécialisés pour transmettre les propriétés de haut niveau de façon transparente par le réseau. Une application de cette approche de design logiciel spécifique au module d'intégration sera présentée au prochain chapitre.

Chapitre 5 - Intégration des modules et interface utilisateur

Dans les chapitres précédents, plusieurs modules indépendants mais permettant la synchronisation des propriétés communes de position, d'orientation et de facteurs d'échelle ont été présentés. Tout d'abord, le module de synchronisation, présenté au chapitre 2, protège l'accès aux données partagées dans un tampon central pour assurer la cohérence entre les modules. Puis, le module visuel, présenté au chapitre 3, permet l'affichage des géométries et des textures des différents modèles 3D sur la surface de rendu. Ensuite, le module réseau du chapitre 4 permet le partage des propriétés communes et celles de haut niveau entre les instances d'une même application via une connexion réseau hybride. Finalement, le module haptique, développé dans (duTremblay à venir), permet quant à lui de simuler l'application des lois de la physique sur les objets du monde virtuel et d'intégrer différents périphériques comme le CyberGlove pour la manipulation d'objets dans la scène virtuelle.

Malgré l'utilisation du modèle de synchronisation qui permet une configuration flexible pour la communication entre les modules, le comportement et les fonctionnalités de l'ensemble des modules utilisés demeurent très différents. Par exemple, le module de distribution présenté dans le chapitre précédent exige, pour demeurer générique, qu'un module de haut niveau qui l'utilise spécialise les classes abstraites `HighLevelObjectProxy` et `HighLevelManagerProxy` et qu'il ajoute au besoin les balises XML de haut niveau pour communiquer avec les autres instances du réseau qui utilisent également ce même module de haut niveau.

Le module de haut niveau qui gère le module de distribution et l'ensemble des autres modules développés dans le cadre de ce projet fera l'objet de ce chapitre. En effet, afin d'intégrer la totalité des modules présentés jusqu'à maintenant dans un seul système tout en conservant une certaine flexibilité, un module d'intégration accompagné d'une interface utilisateur intuitive pour la configuration de l'application a été développé.

Dans les prochaines sections, les concepts généraux et des détails plus techniques sur la conception logicielle du module d'intégration développé pour le projet, nommé `VirtualUniverse`, seront tout d'abord présentés. Comme pour les autres modules, ce dernier contient une liste locale d'objets qui sont cette fois-ci virtuels et désignés par la classe `VirtualObject`. Ces objets de haut niveau permettent d'intégrer les différentes

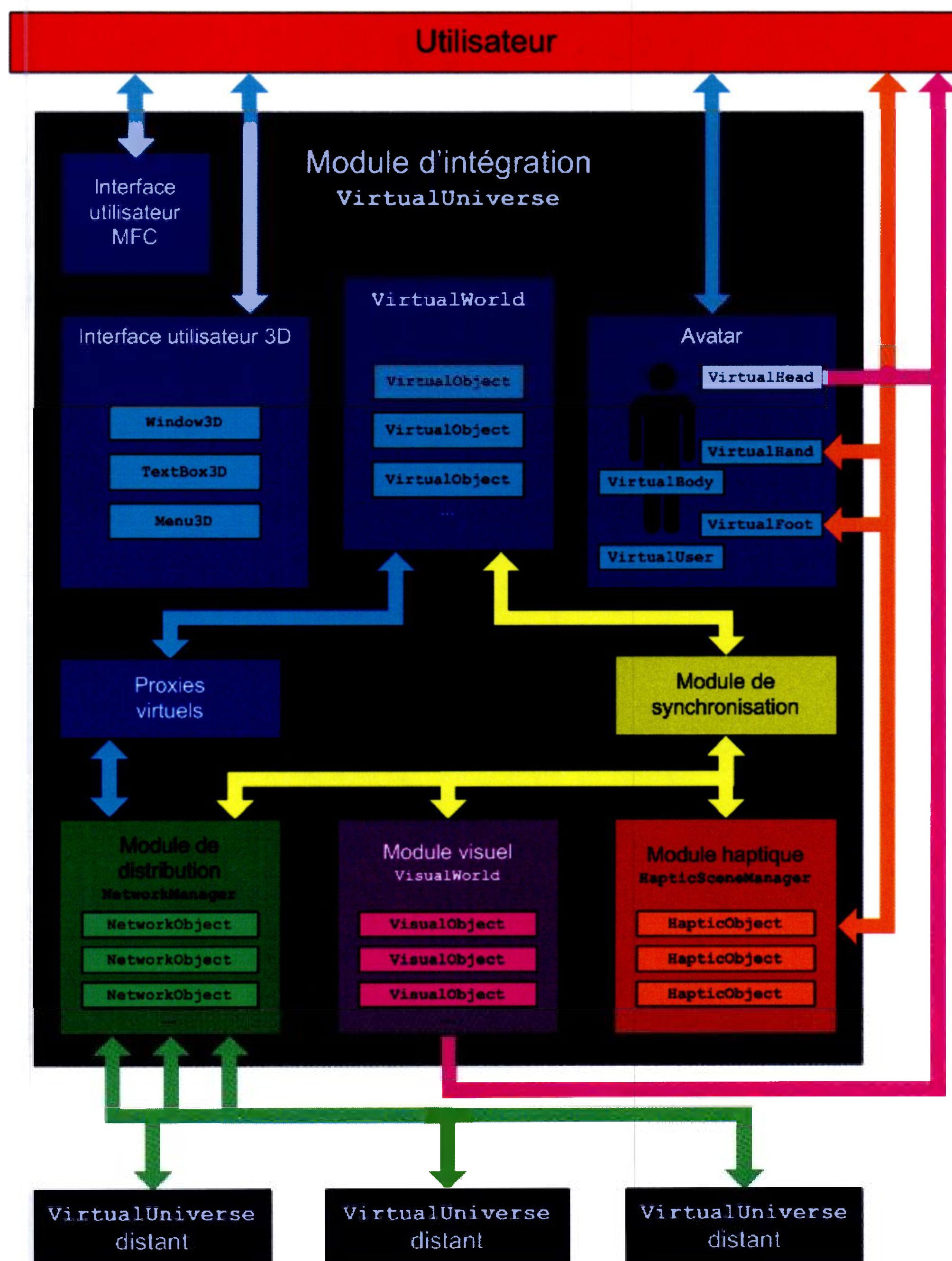


Figure 5-1: Structure générale de VirtualUniverse avec comportement de haut niveau

propriétés (visuelles, haptiques, réseau, etc.) des objets définis dans les autres modules. Puis, l'interface utilisateur qui permettra aux usagers de configurer le module `VirtualUniverse` avant et pendant l'exécution et d'interagir avec le monde virtuel sera présentée dans la section suivante. Finalement, la métaphore utilisée pour la simulation du système de locomotion NELI qui permettra à l'utilisateur de naviguer dans l'environnement virtuel sera présentée à la dernière section.

La Figure 5-1 reprend la structure générale de l'univers virtuel présentée au chapitre 1 en ajoutant les différents éléments permettant la gestion du comportement de haut niveau du module d'intégration qui seront présentés dans les prochaines sections. Tout d'abord, `VirtualWorld` contient la liste de tous les objets `VirtualObject` définissant la scène virtuelle qui seront mis à jour par le *thread* de `VirtualUniverse`. Ensuite, les classes `VirtualUser`, `VirtualBody`, `VirtualHand`, `VirtualFoot` et `VirtualHead` définissent les différentes portions de l'avatar[1] qui permettront de définir des métaphores d'interaction pour la manipulation et la navigation et d'autres fonctionnalités de haut niveau. Puis, les *proxies* virtuels permettent la communication haut niveau entre le module de distribution et le module d'intégration. Finalement, deux types d'interfaces permettent à l'utilisateur d'interagir avec l'environnement : une interface utilisateur 3D et une interface utilisateur basée sur les classes *Microsoft Foundation Classes* (MFC).

5.1 Définition de l'univers virtuel

Le module d'intégration `VirtualUniverse` développé pour le projet peut être représenté comme un ensemble qui regroupe les différents modules qui définissent les propriétés sensorielles et les fonctionnalités de l'univers virtuel. Plus formellement, un univers virtuel est défini ici comme un système interactif où plusieurs utilisateurs situés à des endroits physiques géographiquement différents peuvent partager et modifier de l'information multi-sensorielle commune. Cette définition est illustrée à la Figure 5-2. L'univers virtuel encapsule donc les fonctionnalités qui permettent aux utilisateurs de percevoir les différentes informations sensorielles et d'interagir avec la scène et les autres utilisateurs.

---

[1] Avatar : Représentation virtuelle de l'utilisateur

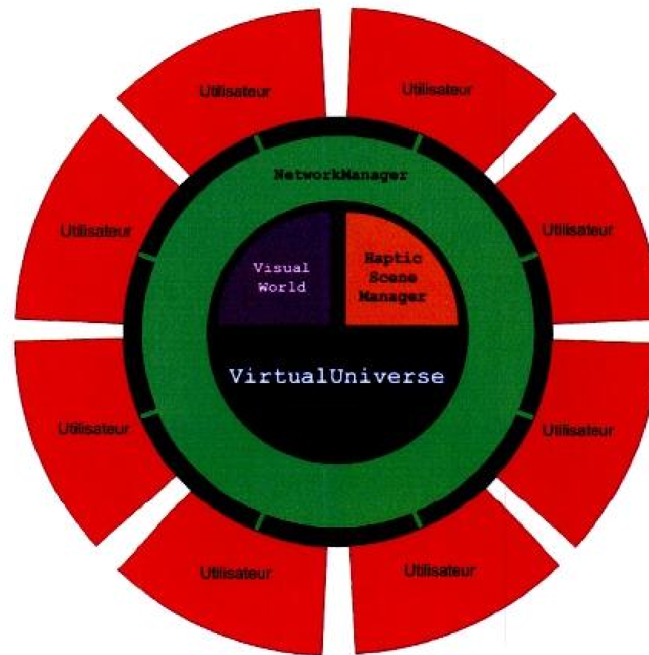


Figure 5-2: Représentation de l'univers virtuel

Le terme « univers virtuel » est utilisé ici par analogie avec l'univers réel dans lequel l'humain évolue. En effet, la perception de l'univers réel qui nous entoure est limitée par l'information accessible via nos cinq sens. Tout d'abord, le sens de la vue permet d'observer les formes, les textures et les couleurs des différents objets de l'univers en trois dimensions. Ensuite, le sens du toucher permet une rétroaction constante lors du contact avec les objets pour ressentir les formes, les textures, la température, etc. Puis, le sens de l'ouïe permet de percevoir les sons pour l'identification d'objets bruyants et la communication vocale. Finalement, les sens du goût et de l'odorat enrichissent l'éventail de perception sensorielle en permettant la perception des saveurs et des odeurs.

De façon similaire, l'univers virtuel englobe une multitude de propriétés sensorielles qui peuvent être transmises à l'utilisateur par le biais de ses cinq sens. De plus, le participant peut interagir de façon plus ou moins naturelle avec l'univers virtuel grâce à des périphériques spécialisés. Finalement, le module réseau présenté au chapitre précédent permet de distribuer cette information sensorielle à tous les utilisateurs faisant partie de l'univers virtuel. Ainsi, grâce à cette dernière fonctionnalité, chacun des participants pourra partager l'information sensorielle commune afin que tous fassent partie de la même expérience virtuelle.

Le module d'intégration ne fait pas exception à la définition de module présentée au chapitre 2. En effet, comme pour tous les autres modules du projet, une représentation locale des objets, appelée `VirtualObject`, doit synchroniser les propriétés communes de position, d'orientation et de facteurs d'échelle avec les objets des autres modules. De plus, l'univers virtuel possède la fonctionnalité spécifique d'être l'intégrateur et le gestionnaire du système. C'est donc `VirtualUniverse` qui sera responsable de la création et de la configuration des différents modules et des objets à l'intérieur de ceux-ci. C'est aussi le module d'intégration qui permettra le transfert de messages haut niveau via le module de distribution à l'aide des *proxies* et qui définira la topologie lors de la distribution de l'application sur le réseau. Il aura également la responsabilité de définir la configuration des écrans pour le rendu graphique lors de l'utilisation du module visuel. Finalement, il devra définir l'avatar ainsi que les méthodes d'interaction qui s'offriront à ce dernier (pour la navigation et la manipulation d'objets entre autres).

Afin d'encapsuler les fonctionnalités pour réaliser ces différentes tâches, plusieurs classes ont été définies à l'intérieur du module d'intégration. Tout d'abord, la classe `VirtualUniverse` est sans aucun doute la plus importante. Elle sert d'interface pour la création et la configuration initiale de la totalité des éléments de l'univers virtuel : modules, objets, périphériques, utilisateurs, etc. Ensuite, la classe `VirtualObject`, présentée à la section suivante, regroupe les fonctions qui permettront de modifier les propriétés sensorielles des objets virtuels pendant l'exécution. De façon similaire aux autres modules, cette classe permettra également la modification des propriétés communes de position, d'orientation et de facteurs d'échelle qui devront par la suite être synchronisées. Puis, la classe `VirtualWorld`, également présentée plus en détails plus loin, représente le terrain virtuel sur lequel les objets reposent et encapsule les propriétés des différents modules intégrés à l'application. Avant de présenter l'ensemble de ces classes, la Figure 5-3 présente d'abord une représentation sommaire de l'univers virtuel sous forme de diagramme UML afin d'illustrer les relations entre chacun des éléments. La classe `VirtualUniverse` est un *thread* dont la boucle d'exécution, implémentée dans la fonction `run()`, met à jour continuellement la liste des objets virtuels créés. L'univers virtuel peut également intégrer différents modules sensoriels et fonctionnels (visuel,

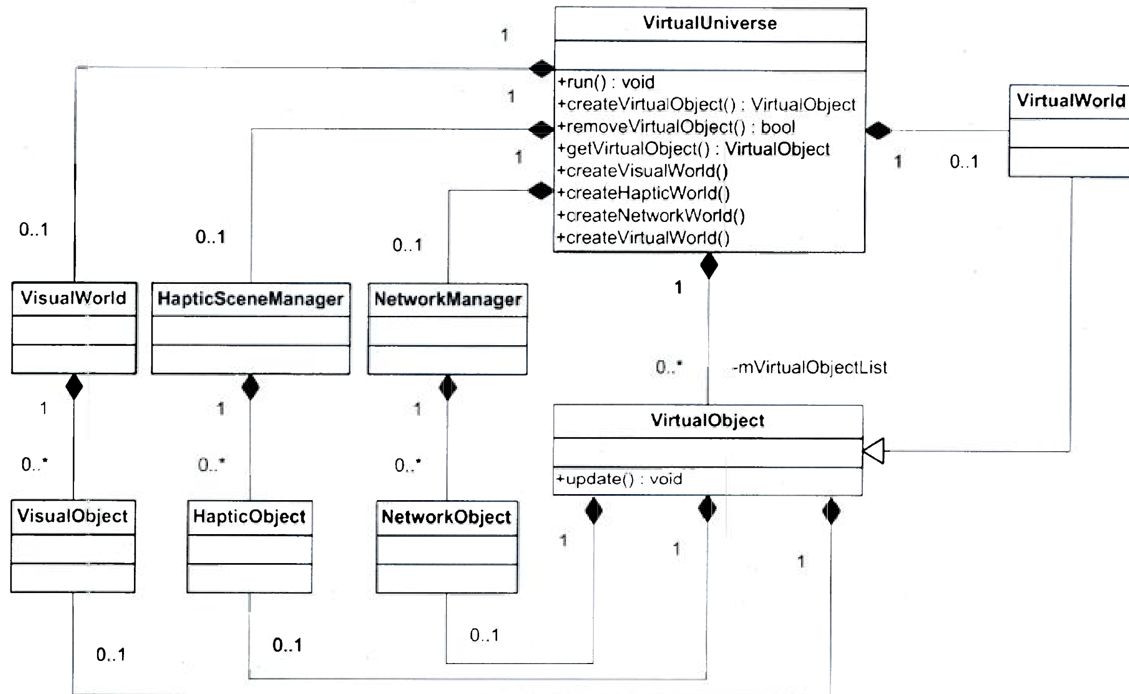


Figure 5-3: Représentation UML de l'univers virtuel

haptique et réseau). Ces modules gèreront les portions sensorielles et fonctionnelles qui composent l'objet virtuel.

Par la suite, de plus amples informations seront données sur les *proxies* virtuels permettant la transmission de propriétés de haut niveau via le module de distribution. Ensuite, la description de l'ensemble des classes qui composent l'avatar sera présentée. Finalement, l'énumération des configurations possibles pour la topologie des instances de l'univers virtuel sur le réseau terminera cette section pour démontrer la flexibilité du système.

Il est à noter que la classe `VirtualUniverse` porte un nom identique à celui du module d'intégration présenté dans ce chapitre. Jusqu'à maintenant, seule une vue d'ensemble du module d'intégration a été présentée. Cependant, les prochaines sections définiront plus en détails les fonctionnalités du module d'intégration regroupées dans les différentes classes qui le composent. Par conséquent, et afin d'éviter toute confusion, le terme « `VirtualUniverse` » sera utilisé à partir de ce point pour désigner uniquement la classe et non pas le module d'intégration en entier. Dans le même ordre d'idées, le terme « univers virtuel » ne désignera plus strictement le module d'intégration mais plutôt le

système en entier comprenant le module d'intégration et l'ensemble des modules chargés pour définir l'application.

5.1.1 Définition des objets virtuels

Comme pour les autres modules, le module d'intégration, plus précisément la classe `VirtualUniverse`, contient une liste d'objets virtuels qu'il mettra à jour localement et en parallèle dans sa propre boucle d'exécution. Ces objets sont représentés par la classe `VirtualObject` comme mentionné plus tôt. De façon générale, on définit un objet virtuel comme une entité logicielle au comportement de haut niveau possédant un nombre quelconque de propriétés sensorielles et des propriétés de haut niveau, lesquelles peuvent être distribuées ou non sur un réseau. La représentation d'un tel objet est illustrée à la Figure 5-4.

Un objet virtuel intègre les fonctionnalités sensorielles des différents modules présentés jusqu'à maintenant ainsi que la fonctionnalité de distribution réseau présentée au chapitre 4 dans une seule et même entité. Pour maximiser la flexibilité, un objet virtuel pourra posséder une portion ou la totalité de ces fonctionnalités et ainsi s'apparenter davantage à un objet réel ou encore n'en posséder aucune et n'avoir qu'un comportement purement virtuel.

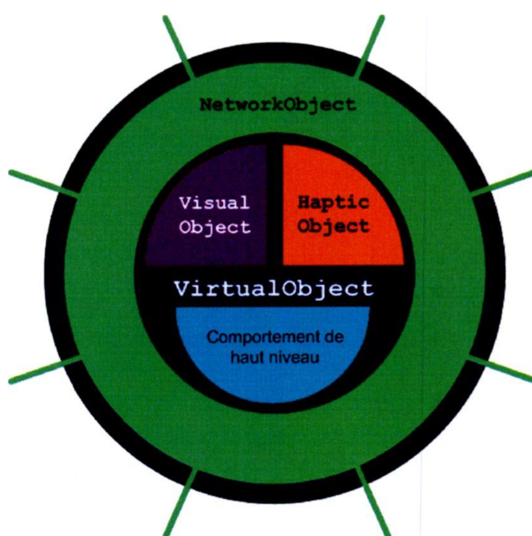


Figure 5-4: Représentation de l'objet virtuel

Comme pour l'univers virtuel, le terme « objet virtuel » est ici utilisé par analogie avec les objets réels. En effet, chaque objet du monde réel possède des propriétés pouvant être observées par les cinq sens. En ajoutant davantage de fonctionnalités sensorielles aux objets virtuels, ces derniers représenteront de plus en plus fidèlement les objets réels et pourront ainsi être perçus et manipulés de façon plus naturelle par l'utilisateur. La fonctionnalité de distribution permet de partager l'objet virtuel lorsque celui-ci doit faire partie de la réalité commune de plusieurs utilisateurs.

À l'inverse, si l'objet virtuel ne possède aucune propriété sensorielle, son comportement devient purement virtuel. En spécialisant ce type d'objet, on peut alors définir des fonctionnalités qui seraient impossibles à réaliser par un objet réel. Malgré tout, un tel objet pourra être distribué sur un réseau et partager ses fonctionnalités avec l'ensemble des utilisateurs.

On peut situer le niveau de virtualité d'un objet virtuel sur une échelle dont les extrémités sont « purement virtuel » et « réel » comme le montre la Figure 5-5. Afin qu'un objet virtuel soit défini comme étant réel, il faudrait qu'un utilisateur ne puisse pas faire la différence entre les instances réelle et virtuelle d'un même objet. Or, en raison des limitations technologiques auxquelles la réalité virtuelle fait maintenant face, ce niveau de virtualité est pour le moment inatteignable et ce, même si la stimulation des cinq sens est implantée dans l'environnement virtuel. De plus, dans l'état actuel des technologies de RV, les sens du goût et de l'odorat sont loin d'être simulés de façon réaliste et satisfaisante. À l'inverse, un objet purement virtuel est facilement réalisable mais le réalisme ne doit pas être l'objectif recherché dans cette situation. Évidemment, il est pratiquement impossible de quantifier le niveau d'importance relative d'une information sensorielle par rapport à une autre. Cependant, l'échelle présentée à la Figure 5-5 énonce clairement que les sens de la

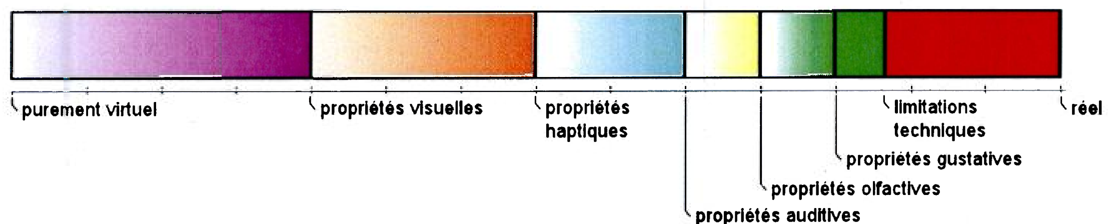


Figure 5-5: Échelle du niveau de virtualité

vue et du toucher seront la plupart du temps les éléments les plus critiques pour le réalisme. Par contre, la perte de réalisme due aux limitations technologiques est difficile à évaluer. Ce qui est probable, c'est que les nouvelles recherches dans le domaine de la RV permettront de réduire cette marge de plus en plus au cours des prochaines années.

Dans les deux prochaines sections, le processus de création et de configuration des objets virtuels sera d'abord présenté afin d'expliquer la façon dont l'intégration des propriétés des différents modules a été effectuée dans la classe `VirtualObject`. Par la suite, une vue d'ensemble des fonctionnalités de l'objet virtuel sera présentée.

5.1.1.1 Création et configuration d'un objet virtuel

Lors de la création d'un objet virtuel via la classe `VirtualUniverse`, l'utilisateur doit fournir une structure de configuration qui permet d'initialiser les propriétés de l'objet virtuel. Cette structure, dont la classe de base est appelée `VirtualObjectConfig`, contient un ensemble de paramètres qui permet de définir les différents aspects de l'objet virtuel dont :

- le type d'objet (permet de déterminer le constructeur à utiliser lorsqu'on objet virtuel spécialisé est créé comme on le verra plus loin);
- les propriétés sensorielles (visuel, haptique, etc.) et fonctionnelles (distribué, traqué);
- le nom de l'objet (utilisé pour la recherche dans une liste);
- le référentiel utilisé pour la synchronisation (absolu ou relatif au parent)
- l'activation de la synchronisation entrante et sortante (étapes 1 et 3 du processus de synchronisation) et
- les paramètres spécifiques qui définissent les propriétés sensorielles et fonctionnelles associées à l'objet.

Le premier paramètre permet de déterminer le type d'objet virtuel créé par `VirtualUniverse`. Puisque `VirtualObject` n'est qu'une classe de base à partir de laquelle d'autres classes devront être définies pour remplir des fonctionnalités spécialisées, ce paramètre essentiel permet de déterminer la classe dérivée de `VirtualObject` qui doit être instanciée lors de la création de l'objet virtuel. En effet, comme on le verra plus loin, d'autres classes dérivées de `VirtualObject` ont été ajoutées au module

d'intégration afin de remplir des fonctionnalités de haut niveau. Par exemple, la classe VirtualUser définit la représentation virtuelle de l'utilisateur. Aussi, la classe VirtualHand permettra de manipuler des objets virtuels. D'autres classes spécialisées seront présentées tout au long du chapitre. Avec le type d'objet, VirtualUniverse pourra donc déterminer le constructeur adéquat qui doit être appelé lors de la création d'un objet virtuel. Si la classe dérivée de VirtualObject exige de nouveaux paramètres pour la configuration, cette dernière devra également définir une nouvelle classe de configuration dérivée de VirtualObjectConfig pour décrire l'information à transmettre à l'objet créé.

Le référentiel utilisé pour la synchronisation de la position d'un objet virtuel est un paramètre très important également. En effet, puisque le module visuel utilise la librairie OSG, il exploite une structure en arbre pour l'organisation des objets dans le monde virtuel. Les données de position, d'orientation et de facteurs d'échelle sont exprimées dans un référentiel relatif au parent dans l'arbre de rendu. Cependant, puisque les modules n'implémentent pas nécessairement tous une telle structure en arbre, il faut pouvoir déterminer si l'information contenue dans le tampon du module de synchronisation réfère à un référentiel absolu ou relatif au parent afin d'assurer la cohérence et d'établir un référentiel standard entre les modules. Le module visuel, utilisant l'arbre de rendu pour l'organisation des objets visuels, peut simuler l'utilisation d'un référentiel absolu en multipliant récursivement les matrices de la racine à l'objet. Le module haptique classe plutôt les objets dans une liste et le référentiel utilisé est donc absolu. Le référentiel utilisé dans le module de distribution n'a aucune importance car ce dernier n'effectue aucune opération locale sur les données partagées par les objets réseau. En effet, les données sont simplement transmises aux autres instances du système qui connaîtront également le référentiel standard à utiliser pour la mise à jour. Cependant, les objets virtuels contenus dans le module d'intégration devront connaître le référentiel utilisé pour l'exécution de leur comportement de haut niveau.

Il est important de noter que si un seul des modules intégrés au système n'implémente pas la structure en arbre et ne la simule pas (n'utilise que le référentiel absolu sans possibilité de conversion) pour la classification des objets, il sera nécessaire d'utiliser un référentiel

absolu comme référentiel standard entre les modules. Dans le même ordre d'idées, si un seul des modules utilisant une structure en arbre ne permet pas la conversion vers le référentiel absolu, il faudra utiliser le référentiel relatif au parent comme référentiel commun entre les modules. Pour demeurer flexible, il faudra donc que les modules implémentent un processus de conversion entre les référentiels relatif et absolu. Pour les modules utilisant une structure en arbre, il suffit de multiplier récursivement les matrices de transformations relatives de la racine à l'objet pour obtenir la matrice de transformation absolue. Pour les modules organisant les objets dans une liste et utilisant le référentiel absolu, la conversion impliquera de connaître les liens entre les objets formant une hiérarchie pour pouvoir déduire la matrice de transformation relative.

Tel que démontré ci-dessous, puisque l'utilisation du référentiel absolu peut entraîner des effets désastreux sur la cohérence de la scène lorsqu'une structure en arbre est utilisée dans un des modules, il sera plutôt recommandé d'utiliser le référentiel relatif au parent comme référentiel standard pour des raisons de cohérence et de performance.

Par exemple, soit le module visuel `VisualWorld` qui contient un ensemble d'objets organisés selon une structure en arbre avec un référentiel relatif au parent et le module d'intégration `VirtualUniverse` qui contient une liste des mêmes objets dans un référentiel absolu comme le montre la Figure 5-6. Pour simplifier la figure, ces objets virtuels ne possèdent qu'une propriété de facteur d'échelle de groupe (propagée sur le reste de la hiérarchie). Le référentiel absolu est utilisé comme référentiel standard. Le module visuel doit donc faire la conversion entre le référentiel relatif et absolu avant de faire l'écriture dans le tampon partagé lors du processus de synchronisation. Les valeurs de R donnent le facteur d'échelle relatif au parent utilisé dans le module visuel. Les différentes valeurs de A représentent le facteur d'échelle de l'objet donné selon le référentiel absolu. Sur la Figure 5-6, l'itération 1 présente l'état initial des deux structures. À l'itération 2 du module d'intégration, ce dernier modifie le facteur d'échelle de deux objets différents dont l'un est le parent de l'autre dans le module visuel. Cependant, puisque le taux de rafraîchissement des différents modules n'est pas le même, il peut arriver des situations comme celle illustrée dans la Figure 5-6 où l'itération du module visuel soit déjà commencée au moment de la synchronisation du module d'intégration. Dans l'exemple

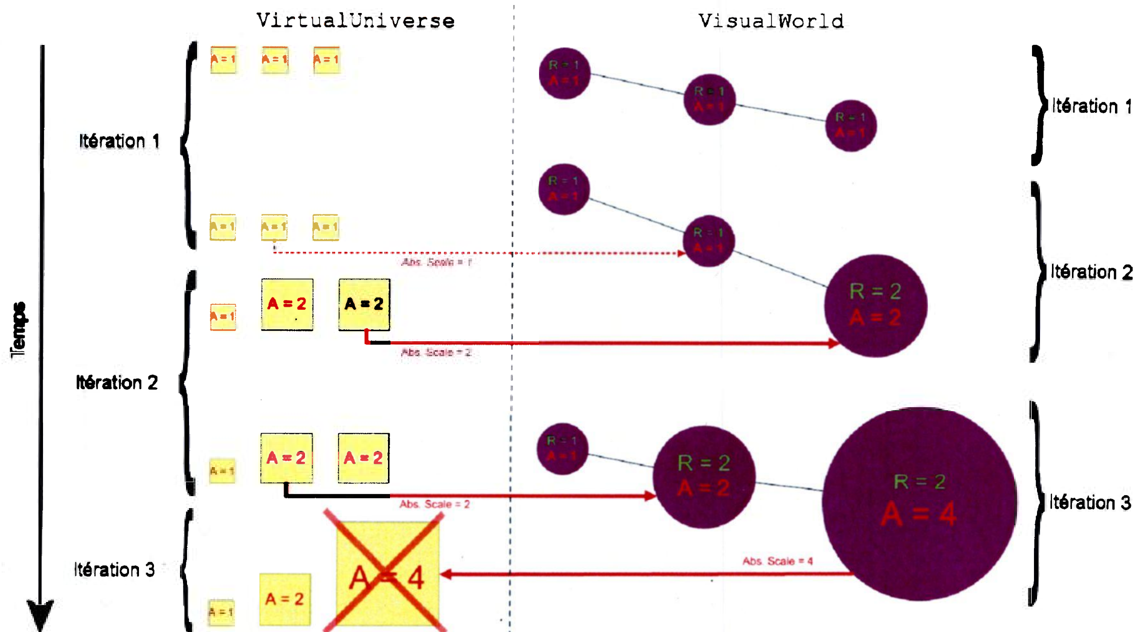


Figure 5-6: Incohérence lors de l'utilisation du référentiel absolu avec une structure en arbre

présenté ici, le deuxième objet dans l'arbre de rendu sera mis à jour en tenant compte des données de l'itération précédente du module d'intégration (flèche pointillée). En fait, il n'y a alors pas de mise à jour puisque le tampon partagé n'aura pas changé dans le module de synchronisation. Ensuite, dans la même itération du module visuel, le troisième objet, fils du second, est également mis à jour en tenant compte des modifications apportées à l'objet pendant l'itération 2 du module d'intégration. Le fils convertit alors les données absolues selon les matrices contenues dans le parent à cet instant. À l'itération 3 du module visuel, le parent recevra sa mise à jour et effectuera la conversion nécessaire. Cependant, cette modification aura une répercussion sur l'objet fils qui devra également mettre à jour la valeur absolue. En effet, puisque le fils a tenu compte des modifications du module d'intégration avant le parent, la valeur relative calculée par rapport à celui-ci était alors erronée. Lorsque le parent est ensuite mis à jour à l'itération suivante et transmet les nouvelles matrices de transformation à l'objet fils, le résultat de cette modification synchronisé avec le module d'intégration crée une incohérence car le facteur d'échelle absolu qui est obtenu par l'objet fils n'a alors plus la même valeur qu'à l'itération précédente.

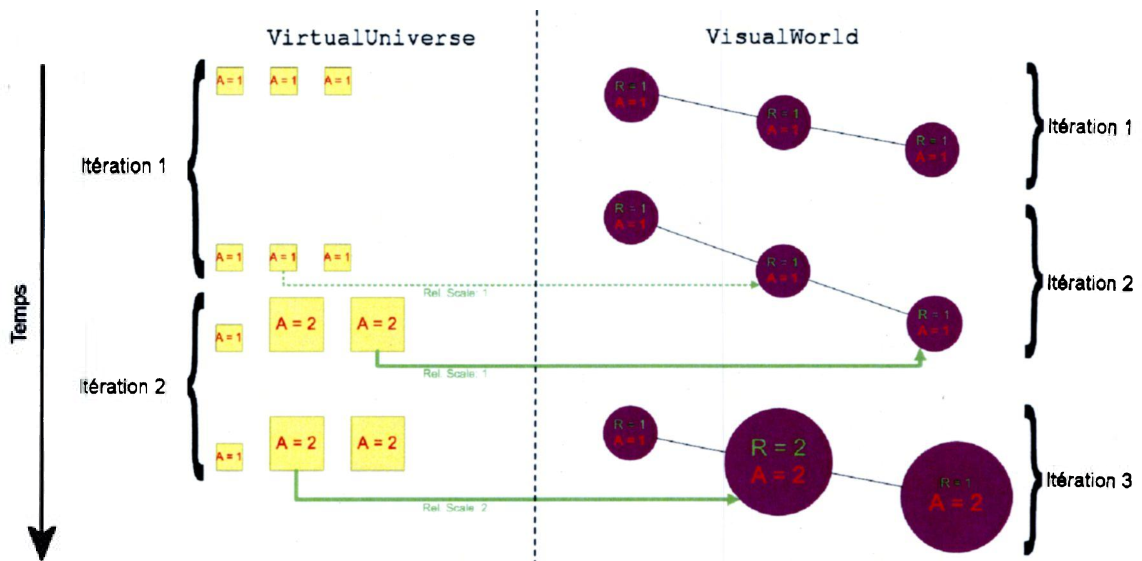


Figure 5-7: Conservation de la cohérence grâce au référentiel relatif avec une structure en arbre

L'approche du référentiel relatif au parent permet d'assurer une synchronisation cohérente lors de l'utilisation d'une structure en arbre. En effet, la Figure 5-7 reprend l'exemple précédent. Cette fois-ci, c'est le module virtuel qui doit faire la conversion entre le référentiel absolu et le référentiel relatif au parent utilisé pour la synchronisation. À l'itération 2 du module visuel, la valeur absolue du facteur d'échelle du troisième objet sera erronée pendant une itération puisque le parent n'aura pas encore été mis à jour et que le facteur d'échelle relatif au parent n'aura pas changé. Cependant, puisque la valeur absolue du facteur d'échelle n'est pas synchronisée, elle n'aura aucun impact sur le système. À l'itération suivante, le parent sera mis à jour et l'arbre converge vers la structure cohérente.

En plus d'assurer la cohérence, le référentiel relatif au parent augmente également la performance de l'application en limitant le nombre de mises à jour nécessaires pour synchroniser une transformation commune à une hiérarchie d'objets avec le tampon central. En effet, si on veut effectuer une transformation quelconque sur un objet visuel et l'ensemble de ces fils, il suffira d'appliquer la mise à jour au parent seulement et cette modification sera alors implicite pour toute la hiérarchie de l'objet visuel. Par contre, pour effectuer cette même opération avec le référentiel absolu, il faudra faire une opération d'écriture et de lecture sur le tampon central pour chacun des objets visuels composant la hiérarchie. Selon la profondeur de l'arbre de rendu, le référentiel relatif au parent sera donc en général plus performant que le référentiel absolu.

Si l'utilisation du référentiel absolu est vraiment nécessaire, on peut contourner le problème en définissant un arbre avec une racine et un seul niveau hiérarchique dans le module visuel. Ainsi, les données seront toujours cohérentes. Cependant, il ne sera alors plus possible de former une hiérarchie complexe pour les objets visuels ce qui diminuerait grandement la flexibilité. L'utilisation du référentiel relatif au parent sera donc l'approche conseillée dans la plupart des cas.

Finalement, les derniers paramètres de la structure `VirtualObjectConfig`, les paramètres spécifiques qui définissent les propriétés sensorielles et fonctionnelles associées à l'objet virtuel, prennent également la forme de structures contenant à leur tour plusieurs paramètres. Ces structures de bas niveau ne peuvent par contre pas être spécialisées et seront donc communes à toutes les configurations virtuelles, même celles dérivées de `VirtualObjectConfig`. Suite à la réalisation du projet, on compte trois de ces structures pour configurer les propriétés visuelles et haptiques ainsi qu'un éventuel dispositif de *tracking*. Les objets `NetworkObject` du module de distribution ne nécessitent aucun paramètre de configuration.

Pour la propriété visuelle, la structure comporte les paramètres suivants :

- le type d'objet visuel (« fantôme[1] », modèle 3D complexe, boîte, sphère, etc.);
- le nom de fichier du modèle 3D (le cas échéant);
- la couleur (pour la boîte ou la sphère);
- la dimension (pour le mode « fantôme », la boîte ou la sphère);
- l'origine du référentiel de coordonnées à synchroniser (origine par défaut du modèle ou centre de la *bounding box* qui entoure l'objet) et
- l'activation de la synchronisation entrante et sortante (étapes 1 et 3 du processus de synchronisation).

Pour la propriété haptique, les paramètres suivants sont intégrés à la structure :

- le type d'objet haptique (complexe, boîte, sphère, etc.);
- le nom du modèle 3D (pour un objet complexe);
- la dimension (pour la boîte ou la sphère);

[1] Existe dans le monde visuel (possède une position, une orientation, des facteurs d'échelle, des dimensions, etc.), mais la géode définissant le modèle 3D de l'objet visuel ne contient aucune géométrie à afficher.

- l'activation des interaction avec les mains (aucune, droite ou gauche, les deux);
- l'activation de la gravité;
- l'activation des collisions et
- l'activation de la synchronisation entrante et sortante (étapes 1 et 3 du processus de synchronisation).

Finalement, pour définir le traqueur lorsqu'un objet virtuel est traqué, on utilise les paramètres suivants :

- le type de traqueur (InterSense, souris, VRJ, etc.);
- le numéro de la station de *tracking*[1];
- le nom du *proxy* dans VRJ (lorsque le type de traqueur correspondant est utilisé) et
- le gestionnaire du traqueur.

Le gestionnaire du traqueur permet de déterminer le module parmi tous ceux associés à l'objet virtuel qui aura accès aux données du traqueur pour définir la position et l'orientation de l'objet en question. Bien que le choix de ce paramètre n'affecte en rien la synchronisation ou la cohérence de la scène, il aura par contre une incidence sur la performance des différents modules. En effet, l'objet ayant accès directement aux données du traqueur pourra mettre à jour sa matrice de transformation sans délai lors de la boucle d'exécution du module et ces nouvelles données seront ensuite transmises aux autres modules via le processus de synchronisation ce qui leur imposera un délai pour la réception. Ce paramètre aura donc un impact différent selon la valeur choisie.

Par exemple, si le gestionnaire du traqueur est l'objet haptique, la rétroaction des forces sera effectuée sans délai mais l'objet visuel suivra la trajectoire du traqueur avec un peu de retard. À l'inverse, si l'objet visuel gère le traqueur, le rendu sera mis à jour directement lors d'un déplacement du traqueur, mais la réaction aux forces appliquées sera effectuée après un court délai. De plus, puisque le résultat de la simulation haptique a des répercutions sur le rendu visuel, il y aura un délai supplémentaire pour observer les effets de l'application des lois de la physiques sur les objets haptiques affectés par l'objet traqué.

---

[1] En effet, un même système de *tracking* peut traquer plusieurs dispositifs à la fois. Ces dispositifs sont appelés station de *tracking*. Il faudra alors déterminer l'index de la station pour la différencier des autres stations lors de la mise à jour. C'est le cas entre autres pour le système IS-900 d'InterSense dont le numéro de la station détermine également le numéro du port sur le système de *tracking*.

Un compromis consiste à mettre l'objet virtuel comme gestionnaire. Ainsi, il y aura un délai pour tous les modules, mais celui-ci sera uniforme. Cependant, cette situation n'est pas toujours souhaitable selon l'application. Dans tous les cas, plus la fréquence de mise à jour des différents modules est élevée, moins ces délais seront observables.

La classe `VirtualObject` dérive de `InterModuleObject` et `Synchronizable` toutes deux définies dans le module de synchronisation présenté au chapitre 2. L'héritage multiple permet aux objets virtuels d'être à la fois l'objet central qui contient les données communes à tous les modules et l'objet à synchroniser via le processus de synchronisation lors de la boucle de mise à jour de haut niveau. Lors de la création de l'objet virtuel, la configuration initiale déclenche la création des objets sensoriels et fonctionnels dérivés de la classe `Synchronizable` qui lui sont associés selon les paramètres de la configuration. Ensuite, par le biais de la classe `InterModuleObject`, il interconnecte ces différents

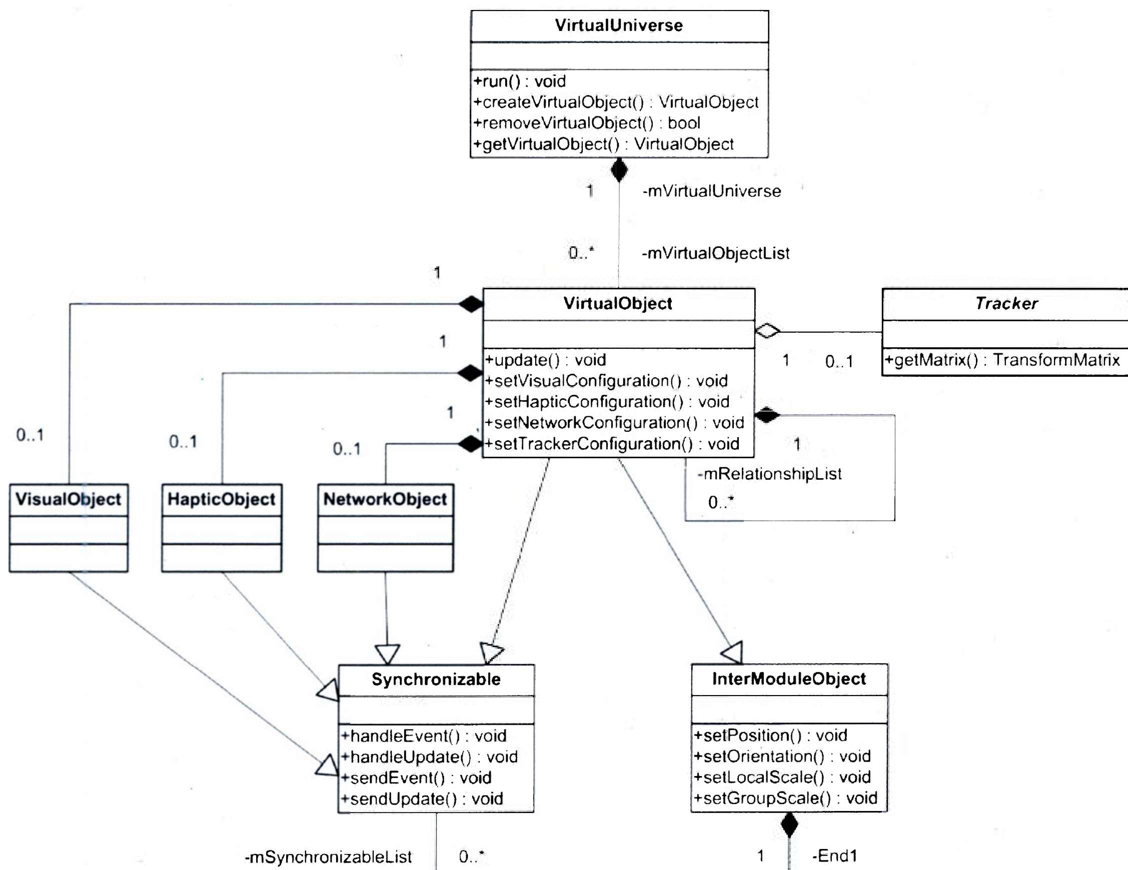


Figure 5-8: Représentation UML de l'objet virtuel

objets entre eux pour enclencher le processus de synchronisation. Il faut ici noter un détail important : le module d'intégration ne s'occupera pas de la synchronisation en tant que tel car cette tâche est attribuée au module de synchronisation. En effet, chaque objet `Synchronizable` situé à l'intérieur des modules accèdera aux fonctionnalités de l'`InterModuleObject` qui lui sera attribué de façon transparente pour synchroniser les propriétés partagées lors de sa propre mise à jour. On peut donc raffiner la représentation de l'objet virtuel de la Figure 5-4 en utilisant les diagrammes UML comme le montre la Figure 5-8. Une fois l'objet virtuel créé, ce dernier est ajouté à la liste d'objets contenue dans `VirtualUniverse` et ensuite mis à jour pendant la boucle d'exécution du module d'intégration.

5.1.1.2 Fonctionnalités des objets virtuels

Afin d'incorporer un comportement de haut niveau aux objets virtuels, trois fonctionnalités ont été ajoutées à la classe `VirtualObject` en plus des méthodes permettant la modification des données communes de position, d'orientation et de facteurs d'échelle. Il s'agit de la gestion haut niveau des événements, de la mise à jour spécialisée et des liens virtuels entre deux objets.

Tout d'abord, la gestion haut niveau des événements sert simplement à améliorer la distribution d'un événement aux différents objets `Synchronizable` spécialisés (objets visuel, haptique, réseau, etc.) associés à l'objet virtuel qui encapsule les fonctionnalités de synchronisation de l'`InterModuleObject`. En effet, par défaut dans le module de synchronisation, un événement quelconque émis par un module sera transmis à l'`InterModuleObject` qui fera ensuite un simple *broadcast* à tous les autres modules. Bien que la réception d'un événement inutile ne soit pas catastrophique, la classe `VirtualObject` surdéfinit la méthode `applyEvent()` de la classe `InterModuleObject` ce qui permettra d'expédier les événements aux modules adéquats selon le contexte et les fonctionnalités de l'objet virtuel pour optimiser la transmission d'information entre les modules. Par exemple, lorsque le pied de l'utilisateur touche le sol dans le module haptique, cela génère un événement qui est envoyé à l'objet `VirtualObject` associé au pied haptique. Cet événement est crucial pour la métaphore

de navigation qui sera expliquée à fin de ce chapitre. Par contre, il n'est d'aucune utilité pour les autres modules du système actuel qui n'ont besoin que de la position et de l'orientation du pied déterminées pas le processus de synchronisation. D'autres événements comme la préhension d'un objet avec la main haptique seront cependant significatifs pour plusieurs modules.

Ensuite, la mise à jour spécialisée permet de définir le comportement haut niveau des objets virtuels. En effet, le module d'intégration encapsule lui aussi les fonctionnalités du *multi-threading* dans `VirtualUniverse` pour mettre à jour en boucle et en parallèle la liste des objets virtuels qui ont été créés en appelant la fonction `update()` contenue dans la classe `VirtualObject`. Ainsi, selon le type d'objet virtuel qui a été ajouté, on peut surdéfinir la fonction `update()` afin d'exécuter sans arrêt un comportement de haut niveau quelconque de façon itérative. Par défaut, la classe `VirtualObject` intègre le comportement d'un traqueur pour déterminer sa position dans l'espace comme le montre le diagramme UML de la Figure 5-8.

Finalement, les liens virtuels permettent d'établir une relation entre deux objets virtuels quelconque. Cette relation implique toujours deux objets virtuels dont l'un est le propriétaire de la relation et l'autre est le client. C'est le propriétaire qui engendre le lien virtuel avec le client qui accepte simplement la relation. Chacun des partis est alors informé de l'existence de l'autre et peut donc accéder aux propriétés de l'autre lors de l'exécution de la fonction `update()`. Il devient alors possible d'établir un comportement de haut niveau commun quelconque entre les deux objets virtuels.

Il existe trois types de lien virtuel. Tout d'abord, le lien de type *child-tree* indique que les deux objets virtuels impliqués seront organisés selon une structure en arbre semblable à celle de l'arbre de rendu présentée au chapitre 3. En effet, le propriétaire de la relation devient alors le parent et le client devient l'un de ses fils. Si le parent est détruit, toute la hiérarchie sous-jacente sera également détruite. De plus, la position de l'objet virtuel fils sera affectée par les déplacements de son parent. Généralement, la structure de l'arbre du module d'intégration sera identique à celle de l'arbre de rendu du module visuel. Ensuite, le second type de lien virtuel est le *owned-link*. La structure en arbre n'est alors pas utilisée et

la position du client n'est donc pas affectée par celle du propriétaire. Cependant, lors de sa destruction, l'objet virtuel propriétaire conserve la fonctionnalité qui consiste à détruire également le client. Finalement, le dernier type de lien virtuel est le *free-link*. Comme son nom l'indique, le client d'un tel lien ne sera pas détruit lors de la destruction du propriétaire de la relation. Ce type de lien ne sert donc qu'à informer les deux objets virtuels qu'une relation existe entre eux.

La relation établie par le lien virtuel peut avoir deux objectifs. Premièrement, elle peut servir à unir deux objets physiquement afin de définir un objet virtuel composé. Le lien virtuel *child-tree* est alors tout indiqué. Ou alors, la relation peut unir deux objets de façon à établir un comportement de haut niveau commun. Les trois types de liens peuvent alors être utilisés selon le contexte. Dépendant du type d'objet virtuel impliqué dans la relation, il suffira de définir une mise à jour spécialisée dans la fonction `update()` pour collaborer avec un autre objet et réaliser une tâche commune de haut niveau particulière. En plus de leur application évidente pour la gestion de l'arbre de rendu du module visuel, les liens virtuels seront aussi particulièrement utiles pour les interactions de la main virtuelle avec l'interface utilisateur 3D comme on le verra plus loin dans le chapitre.

5.1.2 Définition du monde virtuel

Afin d'encapsuler les comportements de haut niveau reliés à la gestion de la scène virtuelle à l'intérieur de laquelle l'utilisateur peut naviguer, la classe `VirtualWorld`, dérivée de `VirtualObject` a été ajoutée au module d'intégration. Cette classe représente un monde virtuel possédant des propriétés sensorielles et fonctionnelles. En plus d'avoir une représentation visuelle aux géométries généralement complexes, le monde virtuel implémente également certaines fonctionnalités de haut niveau comme le calcul de la hauteur Z du terrain aux coordonnées X et Y. Cette valeur est facilement obtenue en calculant le point d'intersection avec le sol d'une droite verticale passant par ses coordonnées. Ces fonctionnalités seront très utiles afin de détecter la collision d'un objet quelconque par rapport au sol ou pour ancrer l'utilisateur au sol virtuel lors des déplacements. Le monde virtuel doit également gérer l'objet haptique qui est associé au terrain virtuel car la classe décrivant cet objet dans le module haptique n'hérite pas des

fonctionnalités de la classe `Synchronizable` permettant le processus de synchronisation normal. Finalement, il peut être interfacé avec les pilotes VRJ comme on le verra plus loin.

5.1.3 Définition des *proxies* virtuels

Comme il a été mentionné au chapitre 4, le module de distribution réseau a défini un mécanisme de *proxy* pour la communication transparente avec un module de haut niveau. Puisque le module d'intégration encapsule les fonctionnalités du module de distribution pour les intégrer au système, il joue donc le rôle de module de haut niveau. Pour permettre la transmission d'information de haut niveau et la réplication du comportement des différents objets virtuels par le biais du module de distribution, le module d'intégration doit définir deux *proxies* : `VirtualObjectProxy` et `VirtualUniverseProxy`. Ces derniers permettront aux objets `NetworkObject` et `NetworkManager` de transmettre des messages de haut niveau aux instances de `VirtualObject` et de `VirtualUniverse`. Ainsi, il sera possible pour le module de distribution d'exécuter des tâches telles que la création et la configuration d'objets virtuels et la création de liens virtuels via les *proxies*.

Comme le montre la Figure 5-9, lors de la création de `VirtualUniverse`, la configuration initiale du système permettra de déterminer s'il y a un module de distribution ou non et le *proxy* correspondant sera créé dans l'affirmative. De la même façon, au moment de la création d'un objet virtuel, celui-ci créera un objet réseau et un *proxy* si la configuration l'exige. L'instance du *proxy* est alors transmise à l'objet réseau correspondant afin de permettre la communication entre les deux modules. Lorsque l'objet réseau fera des requêtes via les différentes fonctions du *proxy*, il pourra accéder à l'objet virtuel de façon transparente. Il est à noter que l'objet virtuel ne possèdera pas nécessairement un objet réseau et un *proxy* si le module de distribution est chargé dans l'univers virtuel. En effet, certains objets virtuels pourront être créés localement afin d'être utilisés dans une seule instance réseau.

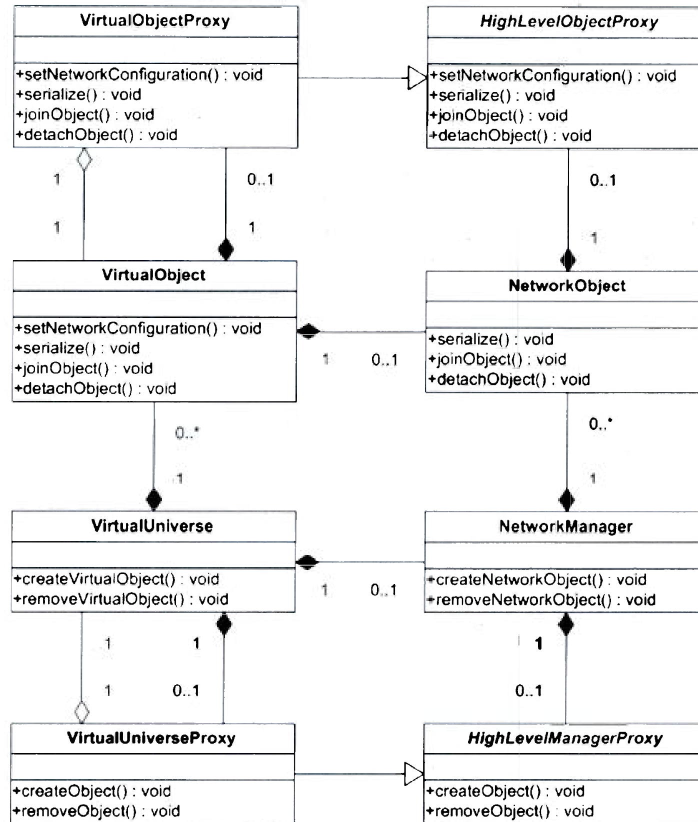


Figure 5-9: Proxies virtuels

5.1.4 Définition de l'avatar

On définit l'avatar comme la représentation virtuelle de l'utilisateur. Ce dernier doit pouvoir observer la scène, naviguer dans l'environnement et interagir avec les objets. Pour permettre ces actions, on peut définir les différentes parties du corps virtuel de l'utilisateur : sa tête, son corps, ses mains et ses pieds. Ces éléments forment un utilisateur virtuel dont les fonctionnalités sont encapsulées à l'intérieur de la classe `VirtualUser`, également dérivée de `VirtualObject`. L'utilisateur sera généralement traqué afin de pouvoir naviguer dans l'environnement virtuel.

Le comportement de `VirtualUser` est fortement lié à celui de VRJ. En effet, puisque la position de la caméra virtuelle est gérée par VRJ, il faut faire correspondre la position de la tête de l'utilisateur avec celle-ci. Pour y arriver, on utilise les référentiels définis au chapitre 3. Le référentiel physique permet de déterminer la position de l'utilisateur dans l'environnement virtuel. Effectivement, on suppose que l'objet `VirtuelUser` sera

toujours situé à l'origine du référentiel physique. Ainsi, il sera facile d'utiliser le référentiel de la tête pour déterminer la position relative de celle-ci par rapport à l'utilisateur. La caméra virtuelle de VRJ possèdera la même position et la même orientation que la tête de l'utilisateur.

Un utilisateur définit donc de nouveaux paramètres pour la configuration de l'objet `VirtualUser`. Ces paramètres sont encapsulés dans la classe `VirtualUserConfig` :

- le nom de l'utilisateur;
- le mode de déplacement de `VirtualUser` (référentiel physique) et
- le mode de déplacement de la tête (référentiel de la tête et de la caméra).

Tout d'abord, le nom de l'utilisateur permet de déterminer les fichiers de configuration à charger pour obtenir les paramètres spécifiques à un utilisateur. Par exemple, la largeur entre les yeux et la calibration d'un CyberGlove peut varier d'une personne à l'autre. Puis, les modes de déplacement permettent de déterminer le traqueur à utiliser pour la position et l'orientation de l'utilisateur et de sa tête. Puisque VRJ doit recevoir la même information pour la mise à jour de la caméra, l'approche la plus simple consiste à utiliser les traqueurs définis dans les fichiers de configuration de VRJ. Les modes de déplacement sont prédéfinis dans `VirtualUser`. Il ne s'agit en fait que d'un simple index qui permet de déterminer le fichier de configuration VRJ à charger. Le clavier, la souris, la *wand* et les différents traqueurs InterSense pourront alors être utilisés via VRJ pour la navigation. Plus de détails sur les pilotes VRJ seront donnés dans la section sur l'interface utilisateur.

Les prochaines sections présentent les particularités pour la création et la configuration des différentes parties du corps de l'utilisateur virtuel. Il est à noter que la création de ces éléments est facultative au bon fonctionnement de l'application.

5.1.4.1 Définition de la tête

La position de la tête est reliée à celle de la caméra VRJ. Les valeurs obtenues dépendront donc du mode de déplacement de la tête déterminé dans la configuration de l'utilisateur. La tête de l'utilisateur est représentée par la classe `VirtualHead` qui dérive également de

`VirtualObject`. La classe `VirtualHead` ne définit aucun nouveau paramètre et agit donc comme un simple objet virtuel traqué.

5.1.4.2 Définition du corps

Comme mentionné plus tôt, la position de `VirtualUser` est ancrée à l'origine du référentiel physique. En effet, `VirtualUser` n'est qu'une entité qui permet de représenter l'utilisateur dans l'univers virtuel. Afin de donner une représentation visuelle et haptique à l'utilisateur, il faut lui associer un corps virtuel. C'est la classe `VirtualBody` qui représente le corps de l'utilisateur. Dans un environnement virtuel comme le FLEX, l'utilisateur peut se déplacer librement dans l'espace physique qui l'entoure. Il peut également déplacer sa tête par rapport à son corps pour effectuer certains mouvements. Si le torse du participant est traqué, il pourra reproduire virtuellement les déplacements de l'utilisateur et sa position relative par rapport à la tête et à `VirtuelUser`. Si le corps n'est pas traqué, comme c'est le cas dans le système actuel, on peut utiliser directement la position de `VirtualUser` ou approximer la position du corps en supposant qu'il se situe toujours au dessous de la tête.

5.1.4.3 Définition des mains

La main est sans aucun doute la partie du corps la plus importante pour l'utilisateur virtuel. En effet, elle lui permet de manipuler les objets virtuels qui ont une propriété haptique et de générer des événements à l'aide de signes et de gestes. Les fonctionnalités de la main virtuelle sont encapsulées dans la classe `VirtualHand`. Cette dernière est constituée d'une paume ainsi que de quinze objets `VirtualFingerPart` qui définissent chacune des trois phalanges des cinq doigts de la main comme le démontre la Figure 5-10. Les différentes parties de la main sont organisées selon une structure en arbre grâce aux liens virtuels *child-tree*.

Les classes `VirtualHand` et `VirtualFingerPart` définissent toutes les deux de nouveaux paramètres de configuration. Tout d'abord, pour la main virtuelle, on retrouve les éléments suivants dans la classe `VirtualHandConfig` :

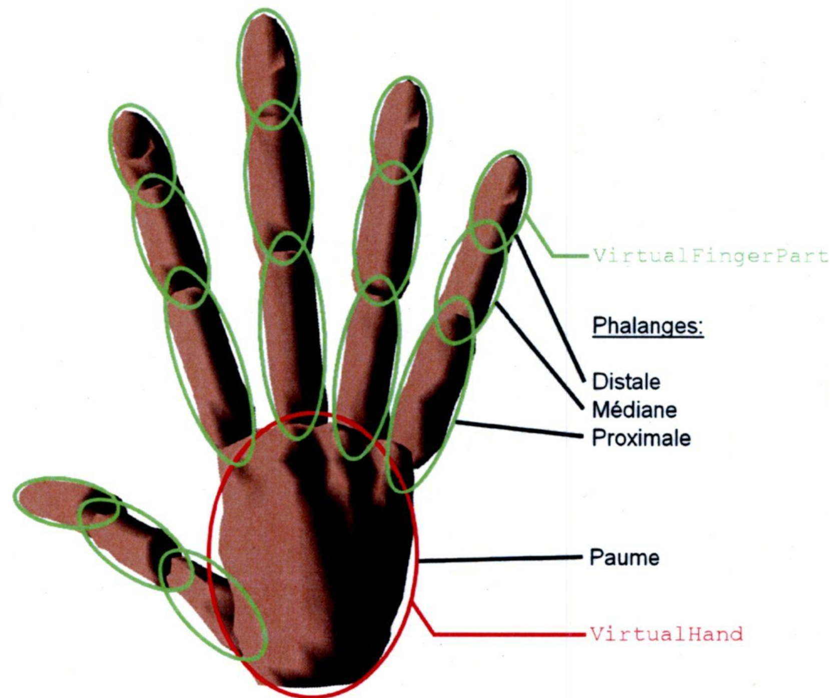


Figure 5-10: Représentation de la main virtuelle

- la latéralité (main gauche ou droite);
- l'adresse IP pour la communication avec le serveur CyberGlove;
- le numéro de port pour la communication avec le serveur CyberGlove;
- l'activation du CyberGrasp et
- le mode d'interaction avec les objets haptiques (sélectionner ou pousser).

L'adresse IP et le numéro de port permettent la communication avec le serveur CyberGlove afin d'obtenir la valeur des différents senseurs de flexion pour la représentation de la main virtuelle. Le mode d'interaction permet de déterminer si les objets peuvent être sélectionnés et saisis par la main ou s'ils sont plutôt simplement déplacés lors de l'application de forces par les différentes phalanges.

Les instances de la classe `VirtualFingerPart` définissent les nouveaux paramètres de configuration suivants dans la classe `VirtualFingerPartConfig` :

- le nom du doigt (pouce, index, majeur, annulaire et auriculaire);
- le nom de la partie du doigt (métacarpien, proximal et distal);
- l'activation d'un pointeur et
- la longueur du pointeur (s'il y a lieu).

Les deux premiers paramètres ne servent qu'à distinguer les différentes parties de la main lors d'une recherche par exemple. Par contre, l'activation du pointeur et sa longueur sont deux paramètres très utiles pour l'interaction avec les objets virtuels. En effet, le pointeur est un simple segment de droite qui est ajouté au bout de l'objet `VirtualFingerPart`. Il sera utilisé pour déterminer si un objet virtuel est pointé par la main en détectant les collisions du segment de droite avec celui-ci. Évidemment, c'est généralement le distal de l'index qui possèdera un tel pointeur. Cette fonctionnalité sera particulièrement utile pour les interactions avec l'interface utilisateur 3D comme on le verra plus loin.

Finalement, comme tout autre objet `VirtualObject`, la main virtuelle peut faire la gestion haut niveau des événements qui transigent via le module de synchronisation. Plus particulièrement, la classe `VirtualHand` permet de réagir aux différents symboles qui peuvent être reproduits par la main de l'utilisateur. C'est le module haptique qui a la responsabilité de détecter les symboles effectués par la main et de générer les événements correspondants. Quelques symboles dont une main plate, un signe « *thumb's up* » et un poing fermé peuvent être détectés pour générer un comportement de haut niveau quelconque dans la fonction `applyEvent()`, surdéfinie dans la classe `VirtualHand`.

5.1.4.4 Définition des pieds

Bien que le déplacement de `VirtualUser` à l'aide d'un traqueur soit une approche efficace et facile à développer pour la navigation dans un environnement virtuel, cette technique est loin d'être la plus réaliste. En effet, dans le monde réel, l'humain se déplace la plupart du temps en marchant pour parcourir de courte distance. De plus, l'interface de locomotion NELI nécessite la détection des collisions du pied de l'utilisateur avec le sol virtuel pour la simulation des forces de contact. Il devient donc impératif de définir une représentation virtuelle du pied de l'utilisateur avec les propriétés visuelle et haptique.

Les fonctionnalités du pied virtuel sont encapsulées dans la classe `VirtualFoot`. Comme pour la main, le pied peut posséder plusieurs parties définies par la classe `VirtualFootPart`. Pour le projet, le pied est simplement divisé en deux morceaux. La classe `VirtualFoot` englobe le talon qui inclut la base du pied; et un seul `VirtualFootPart` représente la pointe qui inclut les orteils. Les deux objets sont

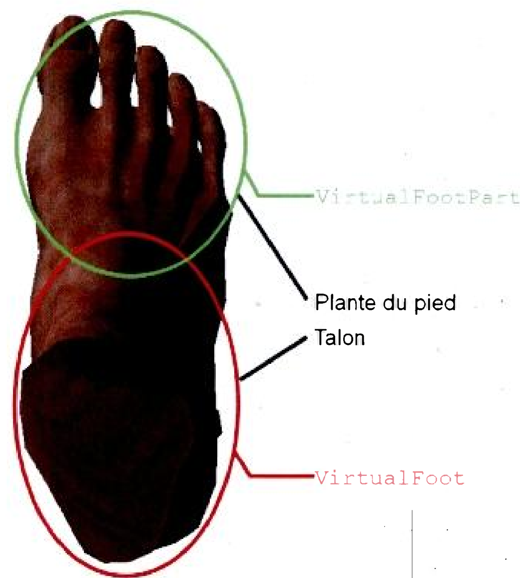


Figure 5-11: Représentation du pied virtuel

encore une fois unis par un lien virtuel *child-tree*. Cette représentation du pied est illustrée à la Figure 5-11. Évidemment, d'autres parties pourraient être ajoutées pour simuler indépendamment les cinq orteils par exemple. Cependant, puisque des périphériques semblables au CyberGlove pour les pieds n'existent pas encore, il n'est pas nécessaire d'en tenir compte pour le moment. De plus, la plate-forme NELI et le module haptique ne nécessitent que ces deux morceaux pour simuler les collisions avec le sol lors de la marche. Le pied virtuel peut donc plier à la jonction entre les deux parties. On peut également utiliser les pieds virtuels pour simuler le fonctionnement du système de locomotion comme on le verra plus loin.

Tout comme la main, la classe `VirtualFoot` doit définir de nouveaux paramètres pour sa configuration. C'est la classe `VirtualFootConfig` qui encapsule cette fois les nouvelles valeurs :

- la latéralité (pied gauche ou droit);
- la métaphore de navigation et
- les paramètres de la métaphore (s'il y a lieu).

La métaphore de navigation permet de déterminer la façon dont les pieds peuvent déplacer l'objet de la classe `VirtualUser` pour la navigation. Pour le projet, seulement deux

métaphores ont été intégrées à `VirtualFoot`. Tout d'abord, la plate-forme NELI réelle permet d'obtenir directement la position de chacun des pieds et de déterminer la position de l'utilisateur virtuel. Ensuite, l'autre métaphore consiste à simuler cette plate-forme comme on le verra à la fin de ce chapitre. Finalement, le pied peut ne posséder aucune métaphore. Il agira alors comme un simple objet virtuel et ne déplacera pas l'utilisateur. La classe `VirtualFootPart` ne détermine qu'un seul nouveau paramètre pour la configuration dans la classe `VirtualFootPartConfig` : le nom de la partie du pied. Comme pour les objets `VirtualFingerPart`, cette valeur pourrait servir à la recherche s'il y avait plusieurs objets `VirtualFootPart` associés au pied virtuel.

Comme pour la main virtuelle, la classe `VirtualFoot` définit une nouvelle gestion haut niveau pour les événements. Grâce à cette fonctionnalité, le pied virtuel pourra déterminer s'il y a collision ou non avec le sol. En effet, puisque le module haptique doit déjà communiquer cette information à la plate-forme NELI lorsque celle-ci est utilisée, il suffit donc au module haptique de propager un événement lorsque le pied touche ou quitte le sol virtuel. Grâce à cette information, le pied virtuel pourra déterminer le comportement de l'utilisateur si la métaphore servant à simuler la plate-forme NELI est utilisée.

5.1.5 Topologies de l'univers virtuel

Maintenant que les modules utilisés pour le projet et la plupart des types d'objets virtuels ont été présentés, on peut intégrer tous ces éléments dans l'univers virtuel pour former l'application finale. Comme il a été mentionné plus tôt, les modules sont tous facultatifs au bon fonctionnement de l'application sauf le module de synchronisation qui forme le cœur du système. De plus, le module d'intégration doit y être ajouté pour faciliter le chargement des modules et gérer l'exécution de l'application. Par contre, ces deux modules réunis ne produisent aucune information perceptible par l'utilisateur. Lors de l'exécution, il faut donc ajouter les modules qui rempliront les fonctionnalités nécessaires pour l'expérience virtuelle. Dans sa topologie la plus simple, dite de base, l'univers virtuel comprend les modules visuel et haptique avec un seul utilisateur comme le montre la Figure 5-12.

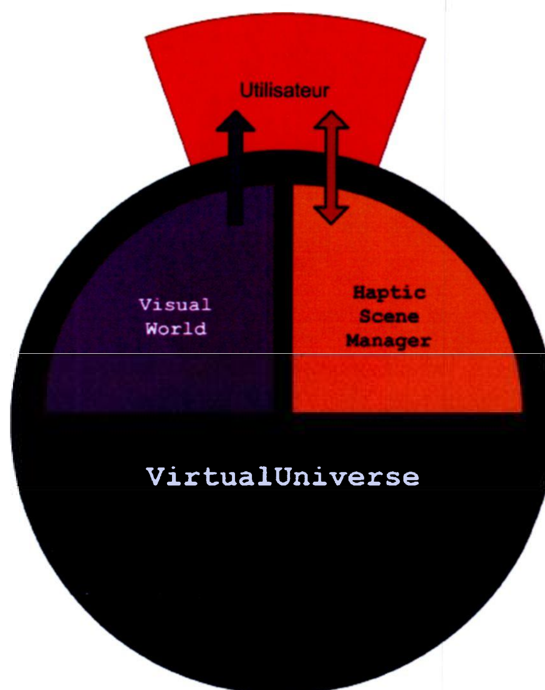


Figure 5-12: Topologie mono-utilisateur de base de l'univers virtuel

Généralement, s'il n'y a qu'un seul utilisateur, il ne sera pas nécessaire d'ajouter le module de distribution. Par contre, on peut distribuer les modules sensoriels sur plusieurs machines pour un même utilisateur afin d'augmenter le parallélisme. En effet, puisque les modules sensoriels sont susceptibles d'utiliser une bonne partie des ressources de la machine, la performance de l'approche par *multi-threading* peut devenir insuffisante si la complexité des modules devient trop grande. Par exemple, dans un engin physique, le nombre d'opérations pour le calcul des collisions augmente exponentiellement selon le nombre d'objets actifs dans la scène. Donc, afin d'éviter une trop grande dégradation de la performance, on peut distribuer les modules sur plusieurs machines afin de répartir la charge de calcul. Ainsi, chaque machine du système devient le gestionnaire d'un seul module sensoriel. La synchronisation de ces sous-systèmes sera assurée par le module de distribution. Dans un système à écrans multiples, on peut également créer plusieurs modules visuels qui seront distribués sur des machines différentes. Chacune de ces machines ne gèrera qu'une seule fenêtre de rendu et cela permettra donc d'augmenter les performances visuelles. Cette topologie mono-utilisateur distribuée est présentée à la Figure 5-13.

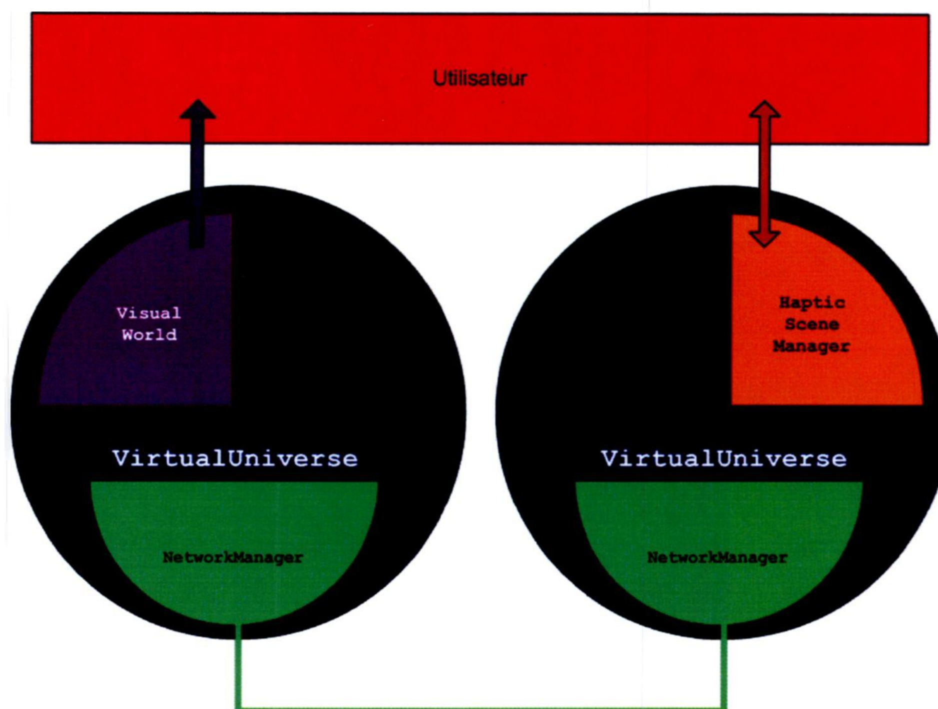


Figure 5-13: Topologie mono-utilisateur distribuée

Lorsque l'application est multi-usagers, il suffit simplement d'unir plusieurs instances des deux topologies précédentes via le module de distribution. Par contre, puisque le module haptique est celui qui met le plus régulièrement à jour les objets virtuels, il sera préférable d'en avoir un seul pour l'ensemble du système afin d'éviter une communication inutile entre les instances de l'application via le module de distribution. Une autre solution consisterait à faire la simulation haptique localement seulement. Cependant, cette approche entraînera nécessairement des incohérences. Lors de la configuration de l'application, on peut donc déterminer une machine qui jouera le rôle de serveur haptique. Chaque utilisateur aura évidemment son instance du module visuel afin d'observer la scène de son propre point de vue. Cette dernière topologie est présentée à la Figure 5-14. Il est à noter que l'approche avec plusieurs instances du module haptique fonctionnera correctement avec un nombre restreint d'objets virtuels.

5.2 Interface utilisateur

Afin de configurer l'univers virtuel au démarrage et pendant l'exécution, il faut intégrer à l'application une interface qui permettra à l'utilisateur d'envoyer une série de commandes

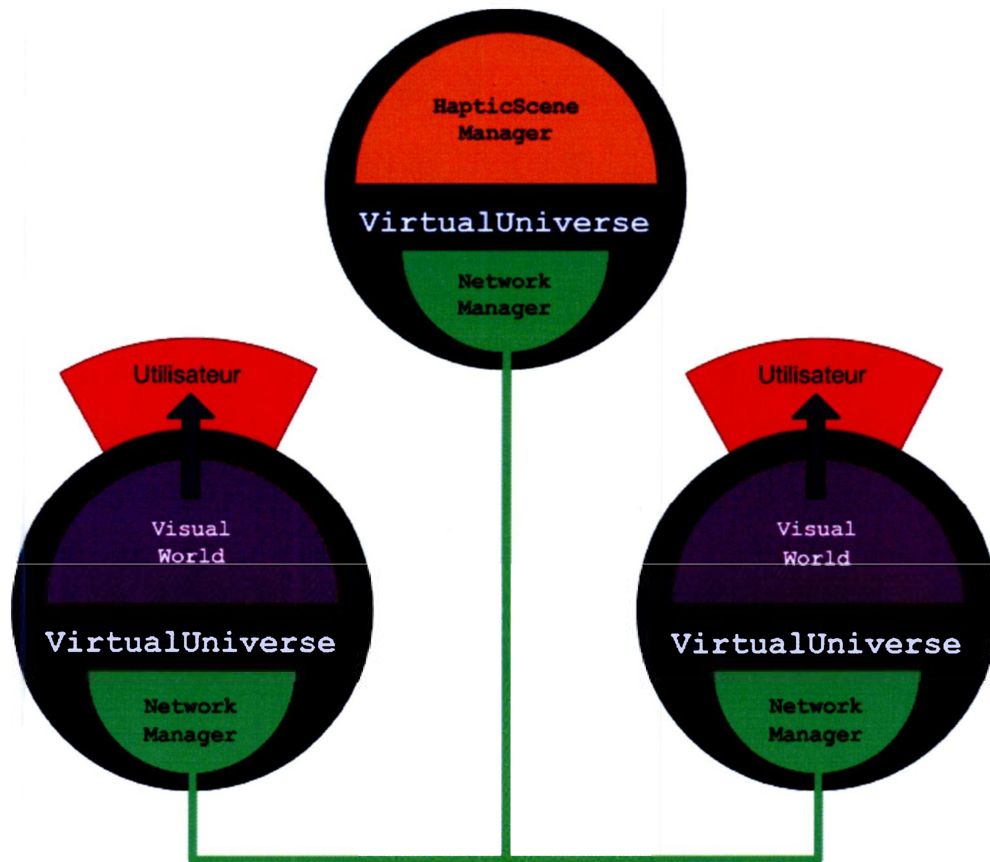


Figure 5-14: Topologie multi-utilisateurs avec serveur haptique

au système de façon transparente et naturelle. Ces commandes consistent par exemple à créer des objets virtuels via la classe `VirtualUniverse` ou encore à modifier les paramètres d'un objet spécifique dans l'environnement. Bien que l'utilisation des classes « Microsoft Foundation Classes » (MFC) permette la création rapide d'une interface utilisateur selon le standard Windows, cette approche ne sera pas suffisante pour tous les aspects de l'application comme on le verra dans les prochaines sections.

Tout d'abord, l'interface utilisée pour la configuration initiale de l'application sera présentée dans la prochaine section. Puis, l'interface graphique 3D, intégrée directement à l'environnement virtuel pour une configuration dynamique, sera le sujet de la section suivante. Ensuite, une autre approche de configuration dynamique exploitant les fonctionnalités du module de distribution sera également présentée. Finalement, un bref aperçu des traqueurs et des autres périphériques permettant l'interfaçage avec l'application conclura cette section sur l'interface utilisateur.

5.2.1 Interface utilisateur de configuration

Afin de paramétrer aisément l'application lors du démarrage, il sera nécessaire de présenter à l'utilisateur une interface de configuration lui permettant d'initialiser l'univers virtuel. Puisque l'expérience virtuelle ne sera pas encore commencée à cette étape de l'exécution, l'utilisation des classes « Microsoft Foundation Class » (MFC) est toute indiquée pour ce genre d'interface car l'utilisateur pourra ainsi configurer rapidement et facilement l'état initial de l'application à l'aide de la souris.

Tout d'abord, puisque le chargement des différents modules est facultatif, il faut pouvoir déterminer lesquels d'entre eux seront utilisés pendant la simulation. Évidemment, comme il a été mentionné plus tôt, les modules d'intégration et de synchronisation sont essentiels au bon fonctionnement de l'application et ne peuvent donc pas être retirés du système. Mis à part le module de distribution qui doit déterminer s'il est serveur ou non et l'adresse du serveur s'il ne l'est pas, aucun autre paramètre n'est nécessaire pour la configuration des modules du système actuel.

Ensuite, l'interface de configuration peut également être utilisée pour initialiser les différents objets formant la scène dès le début de la simulation. La position, l'orientation, les facteurs d'échelle ainsi que les différents paramètres de la structure `VirtualObjectConfig` sont déterminés au démarrage. On peut alors déterminer les propriétés sensorielles et fonctionnelles de chacun des objets virtuels en plus des paramètres de haut niveau selon leur type.

Aussi, l'interface de configuration peut servir à configurer l'utilisateur virtuel et ses différentes composantes. On peut déterminer facilement le nom de l'utilisateur ainsi que les modes de déplacement de `VirtualUser`, de sa tête, de ses mains et de ses pieds. De plus, le participant peut choisir et paramétrer les différentes parties du corps qui doivent être ajoutées à la scène et définir les doigts qui possèderont un pointeur sur le distal.

Finalement, un principe de sauvegarde exploitant une structure XML semblable à celle du module de distribution permet de faciliter la configuration initiale de la scène. En effet, en permettant à l'utilisateur la sauvegarde et le chargement de fichiers en format texte contenant la description de la totalité de la scène dans un arbre XML, on peut extraire de

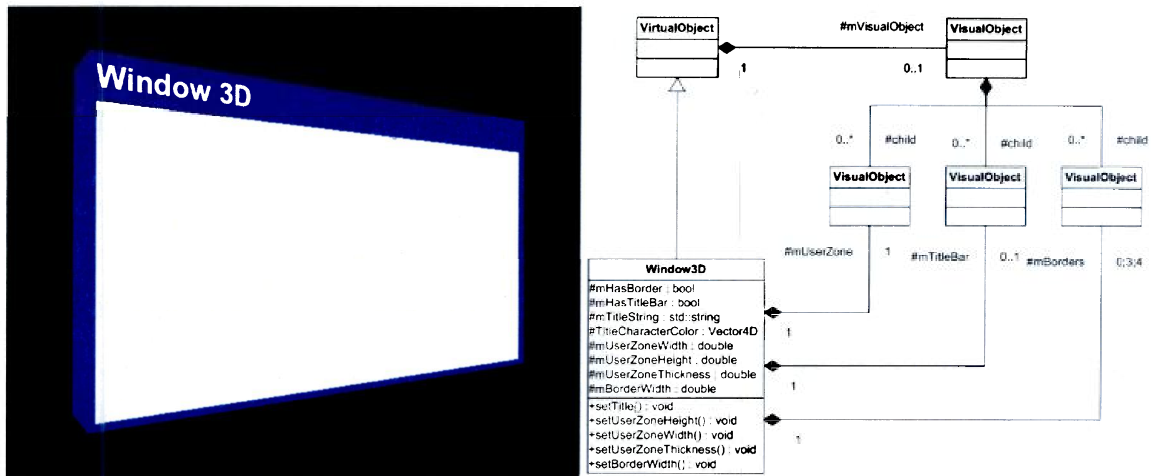


Figure 5-15: Window3D et sa représentation UML

cet arbre les tags qui décrivent la configuration et la hiérarchie des objets virtuels et ainsi conserver en mémoire ou configurer rapidement l'univers virtuel. Ainsi, il sera possible d'enregistrer le résultat d'une expérience virtuelle pour repartir de cet état ultérieurement.

5.2.2 Interface utilisateur graphique 3D

Bien que l'interface précédente soit très utile pour établir la configuration initiale du monde virtuel, elle ne permet pas de mettre à jour dynamiquement la scène lorsque la simulation est démarrée. Il faut donc fournir à l'utilisateur une interface graphique 3D qui s'intègre naturellement à l'environnement virtuel auquel il prend part. Pour y arriver, il suffit de définir de nouvelles classes dérivées de `VirtualObject` qui auront un comportement de haut niveau plus spécialisé.

Tout d'abord, la classe `Window3D` définit une fenêtre 3D semblable à celle de Windows comme le montre la Figure 5-15. Elle possède plusieurs éléments : une barre de titre, une bordure et une zone utilisateur centrale vide. Ce nouvel objet virtuel ne définit aucun nouveau comportement de haut niveau dans la fonction `update()`. Par contre, lors de sa création, la fenêtre 3D définira un sous-arbre dans le module visuel afin de définir les éléments qui la composent. En effet, comme le démontre la représentation UML de la Figure 5-15, une fenêtre 3D possède plusieurs objets visuels qui représentent la barre de titre, les bordures et la zone utilisateur. De plus, un objet visuel invisible forme la racine du sous-arbre pour regrouper l'ensemble de ces éléments visuels. Lors de la synchronisation

avec les autres modules, seul l'objet visuel à la racine est synchronisé. Les autres objets visuels formant la fenêtre 3D ne sont tout simplement pas associés à un `InterModuleObject` et ne participent donc pas au processus de synchronisation. Leur position et leur orientation seront toujours relatives à l'objet visuel à la racine. Ainsi, la fenêtre 3D représente un seul objet dans l'environnement virtuel. Par conséquent, un seul objet haptique sera associé à la fenêtre 3D et les dimensions de cet objet seront déterminées par la *bounding box* entourant la totalité des éléments visuels de la fenêtre.

La création des bordures et de la barre de titre est facultative. Au minimum, une fenêtre 3D peut donc être constituée uniquement de la zone utilisateur. La taille initiale et le volume de chacun des éléments est unitaire (1 mètre dans OSG) et il suffit d'utiliser le facteur d'échelle local des objets visuels correspondants pour déterminer les dimensions. La couleur des bordures et de la zone utilisateur ainsi que le titre de la fenêtre peuvent également être définis via la classe `Window3D`. Il est à noter que l'accès aux paramètres des différents éléments visuels se fait de façon transparente via les méthodes de la classe `Window3D`. Pour redimensionner la totalité de la fenêtre et les éléments qui la composent, il faudra par contre utiliser le facteur d'échelle de groupe (via la classe mère `VirtualObject`) qui sera synchronisé avec l'objet visuel à la racine.

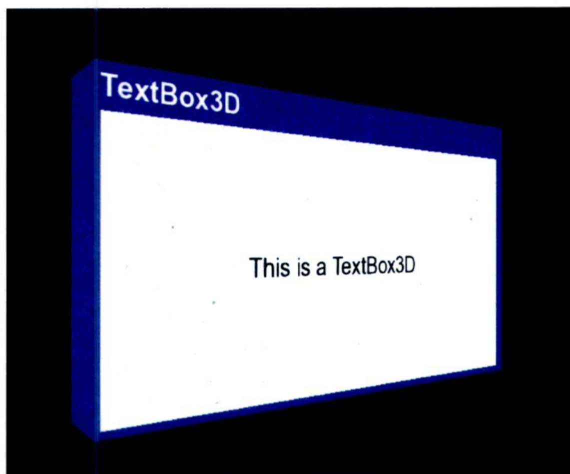
En résumé, la classe `Window3D` ne définit aucun nouveau comportement de plus haut niveau par rapport à `VirtualObject` dans la fonction `update()`. Par contre, une fenêtre 3D possède une structure visuelle plus complexe dans le module visuel afin de définir les dimensions et les autres paramètres des différents éléments visuels qui la composent. La classe `Window3D` définit plusieurs méthodes pour permettre une modification transparente de ces paramètres. Cette fonctionnalité sera transmise aux autres classes de fenêtres 3D spécialisées qui hériteront de la classe `Window3D`.

La configuration initiale des dimensions et des autres paramètres de la fenêtre 3D pourra être effectuée via la classe `Window3DConfig` grâce à ces nouveaux paramètres :

- l'activation des bordures;
- l'activation de la barre de titre (s'il y a des bordures);
- la largeur et l'épaisseur des bordures;

- la couleur des bordures (et de la barre de titre s'il y a lieu);
- la largeur, la hauteur et l'épaisseur de la zone utilisateur;
- les marges horizontale et verticale de la zone utilisateur;
- la couleur de la zone utilisateur;
- le titre de la fenêtre;
- la grosseur des caractères du titre;
- la couleur du titre et
- la largeur de la marge du titre (espacement avec la bordure).

Afin d'enrichir la composition visuelle de la fenêtre 3D, il est possible d'ajouter du texte à celle-ci dans la zone utilisateur centrale. Cette fonctionnalité permettra de transmettre de l'information à l'utilisateur. C'est la classe `TextBox3D` qui encapsule les fonctionnalités d'une fenêtre de texte 3D comme celle illustrée à la Figure 5-16. Ces fonctionnalités comprennent la définition du contenu de la zone de texte (zone utilisateur) ainsi que l'alignement (gauche, centré, droit), la grosseur et la couleur du texte. La classe `TextBox3DConfig` surdéfinit la classe de configuration de la fenêtre 3D afin d'ajouter ces quatre paramètres pour la configuration de la boîte de texte. Lorsque le texte de la zone utilisateur est modifié, l'objet `TextBox3D` doit ajuster la hauteur de la zone centrale ainsi que la position des bordures en conséquence. La largeur de la zone utilisateur demeure par contre toujours fixe à moins que celle-ci soit modifiée explicitement via la méthode correspondante dans la classe `Window3D`. Le choix de modifier la hauteur plutôt que la largeur selon le contenu du texte est totalement arbitraire. Cependant, il sera plus naturel



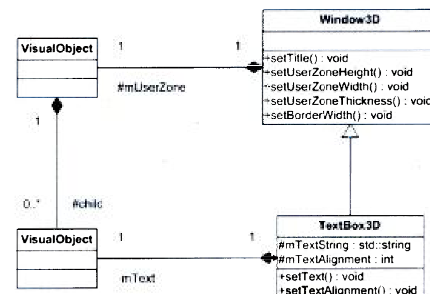


Figure 5-16: TextBox3D et sa représentation UML

pour l'utilisateur de lire un long texte sur plusieurs lignes plutôt que d'avoir une très large boîte de texte qu'il faudra déplacer pour lire le contenu disposé sur une seule ligne.

Finalement, afin de permettre à l'utilisateur d'interagir avec la scène, il est également possible d'augmenter les fonctionnalités de la classe `Window3D` en ajoutant à la zone utilisateur centrale une série de boutons sur lesquels l'utilisateur peut appuyer pour générer un comportement quelconque. Ce nouveau type de fenêtre est représenté par la classe `Menu3D` et illustré à la Figure 5-17. L'ensemble des boutons du menu est représenté par une liste d'objets `Button3D`. La classe `Button3D` est simplement dérivée de `TextBox3D`, mais la configuration par défaut n'active pas les bordures. Pour que l'utilisateur puisse interagir avec le menu 3D, il faut d'abord unir les mains virtuelles décrites à la section 5.1.4.3 avec la totalité des objets `Menu3D` contenu dans la scène à l'aide d'un lien virtuel de type *free-link*. En ayant accès aux fonctionnalités de `VirtualHand`, la classe `Menu3D` peut définir une nouvelle fonctionnalité de haut niveau dans la fonction `update()` qui permet de déterminer si le pointeur d'un des doigts d'une main entre en collision avec l'un des boutons. Si c'est le cas, le bouton 3D pointé sera activé et l'utilisateur pourra aller toucher le bouton avec le doigt virtuel pour exécuter la commande associée au bouton.

Il y a deux façons d'associer un comportement de haut niveau aux boutons contenus dans un menu 3D. Tout d'abord, on peut dériver de la classe `Button3D` et surdéfinir la

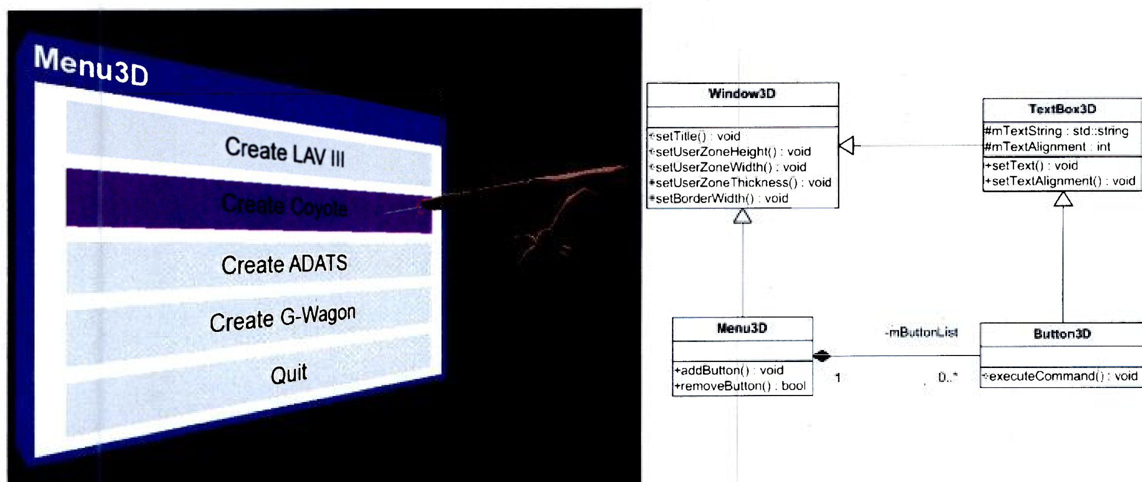


Figure 5-17: Menu3D comprenant plusieurs objets Button3D et sa représentation UML

méthode `executeCommand()` qui est appelée par défaut par la classe `Menu3D` lorsque celle-ci détecte la collision du doigt avec le bouton. L'autre approche consiste plutôt à définir une nouvelle classe de menu spécialisé dérivée de `Menu3D`. En surdéfinissant la méthode `update()`, on peut alors utiliser les boutons de façon arbitraire.

Grâce aux objets `Menu3D`, il sera possible entre autres de créer et de détruire dynamiquement des objets virtuels et d'effectuer plusieurs autres tâches générales sur l'environnement virtuel comme le chargement d'une scène, l'affichage de l'aide, etc. De plus, des menus contextuels spécialisés pourront également être associés à des objets virtuels quelconques pour effectuer des tâches plus spécifiques qui permettront de modifier leurs propriétés de haut niveau en dérivant simplement de la classe `Menu3D` et en surdéfinissant la méthode `update()`.

5.2.3 Interface utilisateur graphique distribuée

Bien que l'approche proposée par l'interface utilisateur 3D permette une mise à jour dynamique de la scène, la gamme d'opérations pouvant être effectuées demeure assez limitée dans le système actuel : créer et détruire des modèles 3D, afficher de l'information. Il serait évidemment possible d'étendre davantage les fonctionnalités de ce type d'interface, mais cela demande beaucoup de temps de développement.

Puisque l'ensemble des classes MFC procure rapidement une panoplie de possibilités d'interactions pour l'utilisateur, il serait intéressant de pouvoir intégrer une telle interface dans l'environnement virtuel. Cependant, à moins de sacrifier une portion de la surface de rendu du système immersif, ces fonctionnalités demeurent difficiles à intégrer dans l'instance locale de l'application. Par contre, le module de distribution apporte une solution à cette problématique. En effet, il suffit de créer deux instances de l'application. La première comprend les modules visuel, haptique et celui de distribution. La seconde instance ne charge que le module de distribution et intègre une interface utilisateur graphique dont l'exécution des commandes est distribuée sur le réseau. Il n'y a donc pas de rendu graphique 3D pour l'instance qui fournit l'interface utilisateur. L'application forme alors une interface semblable à l'interface de configuration qui gère l'exécution de l'univers virtuel local de façon dynamique. Lorsque l'utilisateur activera à distance les différentes

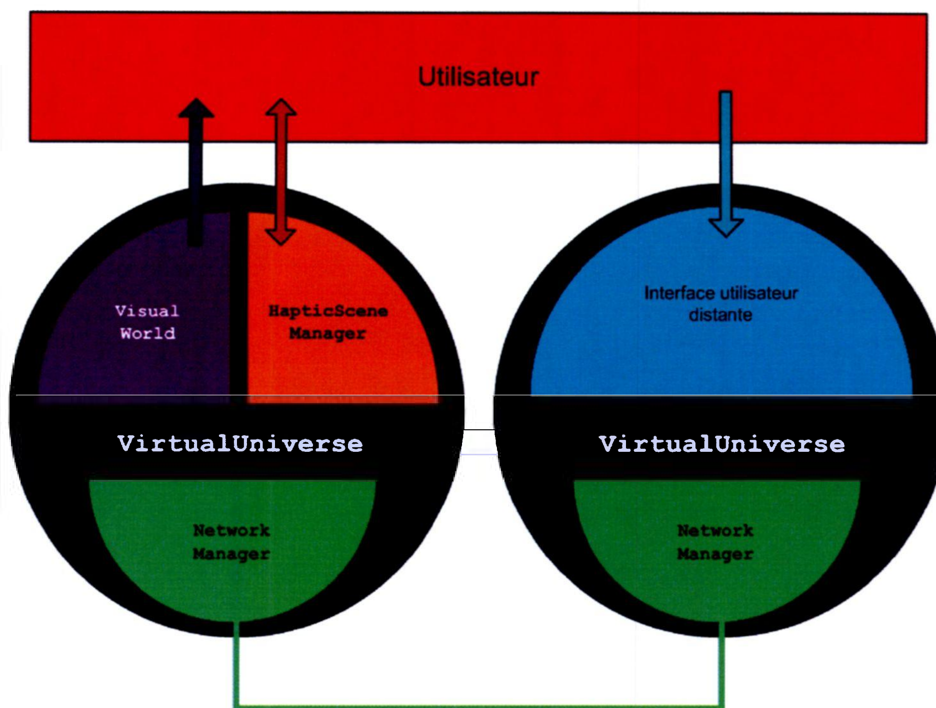


Figure 5-18: Topologie de l'interface utilisateur graphique distribuée

fonctions de l'interface pendant l'exécution, les commandes correspondantes seront transmises aux autres instances de l'application via le module de distribution. Cette topologie est illustrée à la Figure 5-18.

À l'aide d'un périphérique comme le TabletPC, cette interface graphique distribuée pourra être intégrée directement dans l'environnement virtuel. L'utilisateur pourra ensuite activer les différentes fonctionnalités de l'interface du bout des doigts.

5.2.4 Traqueurs et autres périphériques

La plupart des interactions effectuées par l'utilisateur qui ont été présentées jusqu'à maintenant impliquent l'utilisation des mains : manipulation d'objets, interaction avec un menu 3D, reproduction de symboles, etc. Tout d'abord, afin de déterminer la position et l'orientation de la main, un traqueur de position doit être utilisé. Pour la configuration des doigts de la main, un périphérique semblable au CyberGlove peut servir à reproduire des symboles qui devront être reconnus par la portion logicielle. En collaboration avec le traqueur, un tel périphérique donnera également la possibilité à l'utilisateur d'effectuer des gestes plus complexes dans l'espace 3D pour générer des comportements quelconques.

La gestion du CyberGlove est une responsabilité exclusive du module haptique. Par contre, les données du traqueur doivent être accessibles par tous les modules. C'est donc la librairie `Tracker` qui encapsule le comportement de l'ensemble des traqueurs qui sont utilisés pour le projet. On retrouve évidemment la classe `ISTracker` pour le traqueur InterSense, mais aussi les classes `PTracker` pour le traqueur Polhemus et `MouseTracker` qui représente un traqueur simulé par la souris pour effectuer des tests rapides sur l'environnement. Chacune de ces classes définit un comportement spécifique pour la gestion du traqueur. Par contre, les données de position et d'orientation de l'ensemble de ces traqueurs peuvent être obtenues de façon transparente via la classe `TrackerWrapper`. Cette classe procure une interface générique pour accéder aux données sans nécessairement connaître le type de traqueur qui lui est associé.

5.2.4.1 Traqueurs VR Juggler

En plus des traqueurs physiques comme InterSense et Polhemus, la librairie `Tracker` intègre également un autre type de traqueur qui encapsule certaines fonctionnalités de VR Juggler pour la définition de périphériques virtuels : `VRJTracker`. En effet, la librairie VRJ ajoute une grande flexibilité au système en rendant possible le développement de nouveaux pilotes ce qui permet l'ajout de périphériques virtuels pouvant décrire un comportement de haut niveau quelconque en plus de fournir des informations de position et d'orientation. Une fois le pilote développé tel que décrit dans la documentation fournie sur (Infiscape 2007), la configuration des pilotes de VRJ est ensuite déterminée facilement par le biais de fichiers de configuration chargés au démarrage de l'application. Encore une fois, c'est la structure XML qui sera employée par VRJ. Cependant, le format des différentes balises est différent des messages XML présentés au chapitre 4. À l'intérieur de ces fichiers, on détermine également un nom spécifique pour identifier le périphérique, un *alias*. Ce nom sera ensuite utilisé afin d'obtenir l'instance du traqueur virtuel lors du lancement de l'application. Ces fonctionnalités sont encapsulées à l'intérieur de la classe `VRJTracker` qui peut être utilisée pour obtenir les données de position et d'orientation de façon transparente via `TrackerWrapper`. Cependant, la configuration du comportement de haut niveau devra être effectuée par le module d'intégration. On pourra par exemple

attribuer un comportement quelconque aux différents boutons de la *wand* fournie avec le système InterSense.

Pour obtenir plus d'information sur ce sujet, (Drolet 2006) et (Infiscape 2007) présentent de plus amples détails sur la définition des fichiers de configuration et la création de nouveaux pilotes VRJ.

5.3 Simulation du système de locomotion

Comme mentionné lors de l'introduction, la plate-forme NELI est un système de locomotion à mécanismes à câbles permettant la simulation du mouvement de la marche pour une navigation naturelle dans l'environnement virtuel. Tout d'abord, l'utilisateur dépose ses pieds sur deux plates-formes dont la position dans l'espace dépend du mouvement effectué. Selon la forme et les propriétés physiques du plancher virtuel sur lequel l'utilisateur se déplace, comme la friction et la rigidité, les plates-formes devront réagir en conséquence lors de la collision du pied avec le sol. L'inertie du système est minimale grâce à l'utilisation de câbles. De plus, les plates-formes peuvent se déplacer selon six degrés de liberté afin de simuler le mieux possible la topologie du terrain virtuel.

À des fins de sécurité, un harnais retient solidement l'utilisateur au niveau du torse et des hanches pour éviter toute chute en cas de perte d'équilibre. De plus, ce harnais procure une meilleure stabilité à l'utilisateur lors de l'application de l'algorithme de rappel. L'algorithme de rappel est le processus qui permet de ramener l'utilisateur vers le centre du système lors d'un déplacement. En effet, puisque l'environnement physique qui entoure l'utilisateur demeure tout de même restreint, un mouvement compensatoire doit être appliqué dans la direction opposée aux déplacements effectués par l'utilisateur. La vitesse du mouvement compensatoire dépendra de la vitesse du déplacement de l'utilisateur dans une direction donnée. Idéalement, l'algorithme de rappel doit être le moins perceptible possible pour ne pas générer d'indice vestibulaire pour les mouvements compensatoires. Avec l'application de l'algorithme de rappel, le déplacement total de l'utilisateur dans l'espace physique sera en moyenne nul. Lorsque l'utilisateur effectue un pas dans une direction, le pied suit donc une trajectoire quasi-circulaire qui le ramène à son point de

départ. Le mouvement compensatoire effectué par le système de locomotion aura par contre induit une translation à l'utilisateur virtuel, ici représenté par la classe `VirtualUser`.

Puisque l'application développée au cours du projet consiste entre autres à fournir un environnement de test et de simulation pour la plate-forme NELI, la métaphore développée pour la navigation sans le système de locomotion physique doit également être naturelle et doit tenter d'imiter le mouvement circulaire de l'algorithme de rappel.

L'approche proposée définit une métaphore de navigation, nommée *ski-pole*, permettant le déplacement de l'utilisateur dans l'environnement virtuel sans l'intervention des pieds réels. Le pied virtuel est alors représenté par une tige verticale d'une longueur quelconque. Cette tige possède une portion haptique et peut donc être manipulée à l'aide de la main virtuelle et entrer en collision avec le sol. Puis, à l'aide d'un mouvement de propulsion circulaire effectué avec le bras, l'utilisateur pourra se déplacer dans l'environnement virtuel d'une façon similaire au ski de fond en ayant préalablement enfoncé la tige dans le plancher virtuel. Ce principe de propulsion virtuelle est illustré à la Figure 5-19.

Comme le démontre la Figure 5-19, avant d'initier le déplacement, l'utilisateur se trouve d'abord à la position initiale u_0. Puis, à l'itération $i-1$, le pied virtuel entre en collision avec le sol au point f_{i-1}. L'itération i est ensuite exécutée en deux étapes. Tout d'abord, on calcule la translation effectuée depuis l'itération $i-1$ par le pied virtuel, maintenant à la coordonnée f_i. Puis, cette translation est appliquée à l'utilisateur dans la direction opposée. Ce processus sera répété pour toutes les itérations subséquentes tant que le pied virtuel touchera le sol. Évidemment, la translation présentée dans la Figure 5-19 ne sert qu'à illustrer le principe de base de la métaphore de navigation *ski-pole*. En effet, il sera impossible pour l'utilisateur d'effectuer de si large mouvement en une seule itération lors de l'exécution de l'application. Le mouvement global effectué par l'utilisateur sera donc la somme de plusieurs petites translations effectuées par le pied virtuel pendant plusieurs itérations consécutives si ce dernier demeure en contact avec le sol.

Une équation semblable à la méthode *grabbing the air* présentée à la section 1.2.7.5.1 est employée pour calculer la translation induite à l'utilisateur par le pied virtuel. En général,

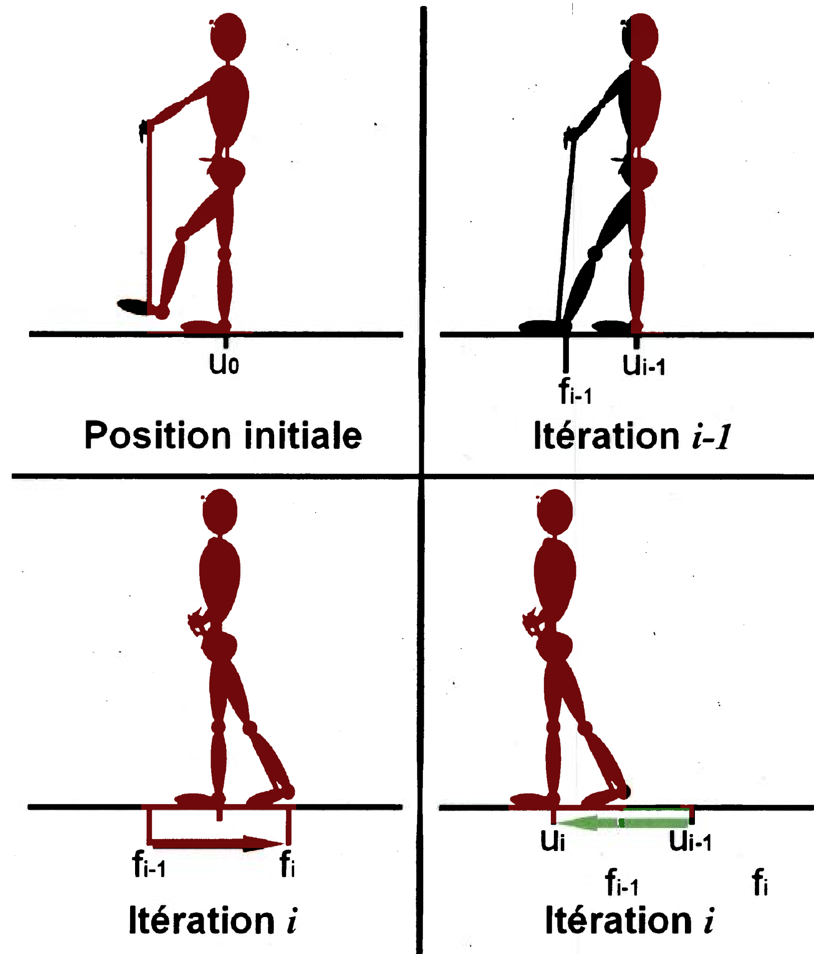


Figure 5-19: Métaphore de navigation ski-pole pour la simulation du système de locomotion NELI

soit la position de l'utilisateur, u_i, et celle du pied, f_i, à l'itération i. Lorsque le pied virtuel touche le sol, la translation opposée à celle effectuée par le pied sera appliquée à l'utilisateur comme l'exprime l'équation suivante :

$$u_i = u_{i-1} + (f_i - f_{i-1})$$

Bien que l'utilisation des mains ne simule pas le mouvement de la marche avec les pieds, le mouvement circulaire induit par l'algorithme de rappel est par contre reproduit avec cette approche. De plus, l'utilisation d'une tige de longueur quelconque pour représenter le pied virtuel évite à l'utilisateur de s'accroupir au sol pour effectuer les déplacements. Finalement, la détection des collisions du pied virtuel avec le sol et l'application des forces correspondantes pourront être simulées par le module haptique afin de reproduire plus fidèlement le comportement de la plate-forme réelle en dessous du pied de l'utilisateur.

Pour utiliser cette métaphore, l'univers virtuel devra intégrer au moins une main et un pied virtuel utilisant la métaphore *ski-pole* pour la navigation. Ainsi, lorsque la tige représentant le pied virtuel entre en collision avec le sol, le module haptique génèrera l'événement correspondant. Avec cette information, la classe `VirtualFoot` appliquera au référentiel de l'utilisateur virtuel une translation opposée à celle effectuée par le pied pour simuler l'effet de propulsion au sol. Lorsque la tige quitte le sol, un nouvel événement est généré et le pied virtuel ne mettra plus à jour la position de l'utilisateur. En utilisant deux mains et deux pieds virtuels, on pourra alterner entre les deux tiges pour le déplacement afin de simuler grossièrement le mouvement du pied effectué lors de la marche sur le système de locomotion NELI.

Finalement, une approche similaire peut être employée pour l'orientation de l'utilisateur virtuel. En effet, il suffira de faire pivoter la tige sur l'axe des z pour induire un changement de rotation à `VirtualUser`. Cependant, afin d'éviter les rotations involontaires lors des translations, le changement d'orientation ne devra être appliqué à l'utilisateur que si la rotation de la tige dépasse un certain seuil (en degrés) après la collision avec le sol. Seule la rotation sur l'axe des z peut être modifiée avec cette technique.

Chapitre 6 - Évaluations et expérimentations

Suite au développement de l'application finale, il est nécessaire d'effectuer une phase d'évaluation et d'expérimentation afin de tester les performances et la qualité de l'environnement virtuel.

Tout d'abord, le problème de latence de synchronisation sera expliqué dans la prochaine section. Comme on le verra, cette limitation du système ne nuit en rien à la cohérence de la scène. Cependant, elle introduira quelques artefacts visuels. Ensuite, la section suivante décrira les tests effectués pour mesurer l'évolution du taux de rafraîchissement du module visuel en fonction de différents facteurs. Finalement, les deux dernières sections présenteront une évaluation empirique du réalisme de l'environnement virtuel et de l'utilisation intuitive de l'interface utilisateur.

6.1 Latence de synchronisation

Comme il a été mentionné dans le chapitre précédent, les modules visuel, haptique et virtuel possèdent tous les trois une boucle de mise à jour qui s'exécute en parallèle. Plus précisément, le module visuel s'intègre à la boucle de VRJ qui met d'abord à jour les différents traqueurs VRJ avant de parcourir l'arbre de rendu OSG. Puisque la caméra est également gérée par VRJ, les mises à jour effectuées dans le module visuel seront perceptibles sans délai par l'utilisateur sur la surface de rendu. Cependant, si un autre module met à jour la position d'un objet virtuel, le processus de synchronisation induira un délai pour la transmission de cette modification vers le module visuel. Par exemple, la chute des objets due à la gravité calculée par le module haptique sera toujours observée avec un court délai dans le module visuel[1].

Donc, il n'y aura aucune latence de synchronisation pour le module qui effectue une mise à jour puisque ce module aura d'abord tenu compte de cette modification localement avant de la transmettre au tampon central. Par contre, pour les modules qui reçoivent par la suite

[1] La latence de synchronisation est toujours présente aussitôt qu'il y a transfert d'une mise à jour d'un module vers les autres. Cependant, dans le système actuel, seul le module visuel permet à l'utilisateur d'observer ces délais directement. Un module sonore ou l'intégration d'un périphérique de retour de force au module haptique permettrait également d'observer la latence de synchronisation.

cette mise à jour, la latence de synchronisation se limitera à une seule itération de délai au maximum. En effet, si l'objet modifié par l'autre module a déjà été mis à jour localement pendant l'itération actuelle, la modification de cet objet sera obtenue à la prochaine itération, induisant ainsi une itération de latence. De façon temporelle, ce délai sera plus ou moins long selon la performance du module en question. L'augmentation du taux de rafraîchissement d'un module diminuera donc la perception de la latence par l'utilisateur si ce module produit une rétroaction sensorielle.

Malgré l'augmentation de la performance des modules pour diminuer les délais de synchronisation, une problématique demeure constante : puisque la durée de la boucle de mise à jour varie d'un module à l'autre et fluctue dynamiquement pendant l'exécution[1], il y aura toujours de la latence de synchronisation et son occurrence sera imprévisible. Cette problématique n'affecte pas la cohérence mais elle peut causer des artefacts de synchronisation et des événements quasi-stochastiques. Elle est également la cause du problème d'incohérence présenté au chapitre précédent qui se produit lors de l'utilisation du référentiel absolu avec une structure en arbre (voir section 5.1.1.1). Puisque l'utilisation d'un référentiel absolu ne ferait qu'empirer les effets de la latence de synchronisation, on supposera l'utilisation du référentiel relatif pour la représentation des données partagées à partir de ce point.

On définit un *artefact de synchronisation* comme une incohérence momentanée qui n'affecte pas le résultat final du déroulement d'un événement. Ce problème apparaît généralement avec l'utilisation d'une structure en arbre pour l'organisation des objets dans un module. Si plusieurs modules se partagent la mise à jour de différents objets contenus dans un arbre pour la définition d'une trajectoire continue, la configuration exacte des différents objets de l'arbre l'un par rapport à l'autre à chaque itération sera imprévisible. Cependant, le mouvement global et le résultat final seront cohérents grâce à l'utilisation du référentiel relatif.

---

[1] Par exemple, le taux de rafraîchissement du module visuel varie en fonction de la complexité de la scène visible dans le champ de vue de l'utilisateur. Aussi, le taux de rafraîchissement du module haptique varie selon le nombre de contacts simultanés dans une même itération.

Généralement, ces artefacts ne seront pas perçus par l'utilisateur car le mouvement des objets entre chaque itération peut être infime. Cependant, une manifestation évidente du problème d'artefact de synchronisation dans l'application finale peut se produire lors du déplacement de l'utilisateur dans l'environnement virtuel. En effet, l'utilisateur virtuel utilise un traqueur VRJ pour définir la position et l'orientation de sa tête dans l'espace 3D. Pour calculer la stéréoscopie *off-axis* et obtenir une latence minimale entre les mouvements de la tête et le rendu visuel, il est impératif que la position et l'orientation de la tête de l'utilisateur soient connues en tout temps par VRJ. De son côté, le module haptique présenté dans (duTremblay à venir) nécessite le contrôle du traqueur pour le déplacement des mains virtuelles afin d'avoir une latence minimale lors de l'application des lois de la physique sur la scène.

La problématique devient alors de déterminer le module qui gèrera le déplacement de l'utilisateur en entier car la position et l'orientation de ce dernier servent de référentiel relatif pour la tête, les mains et les autres parties du corps. Si le module visuel contrôle l'utilisateur, la caméra ne subit aucun artefact de synchronisation puisqu'il n'y a pas de latence. Cependant, l'application des forces de la main virtuelle subira un délai dans le module haptique. À l'inverse, si le module haptique gère la position et l'orientation de l'utilisateur, il n'y aura plus de délai pour l'application des forces avec la main. Cependant, la caméra subit alors les effets des artefacts de synchronisation. En effet, puisque la position de la caméra peut se déphaser momentanément par rapport à celle de l'utilisateur virtuel de façon imprévisible, le participant pourra facilement observer les déplacements en saccades de la main virtuelle qui sont les manifestations des artefacts de synchronisation. Pour éliminer ces artefacts tout en permettant une mise à jour sans délai dans les deux modules, une approche simple consiste à donner le contrôle du traqueur aux deux modules, mais en attribuant à l'un d'entre eux une priorité supérieure à l'autre. Ainsi, chacun des modules pourra obtenir l'information la plus récente du traqueur à chaque itération sans latence. Par contre, l'information partagée dans le tampon central sera celle déterminée par le module de plus haute priorité.

L'autre conséquence de la latence de synchronisation, les *événements quasi-stochastiques*, se définit comme l'ensemble des interactions induites par une suite déterministe d'actions

effectuées par différents modules qui mène à un résultat réaliste mais imprévisible. Cette problématique se manifeste également lors de l'utilisation d'une structure en arbre s'il y a interaction entre les objets de l'arbre et que ceux-ci sont gérés dans différents modules. Selon l'occurrence de la latence de synchronisation à chaque itération, la position et l'orientation des objets pourront être légèrement différentes d'une simulation à l'autre induisant ainsi une variabilité sur le résultat final. Cependant, l'impact sur ce résultat final sera généralement négligeable et la suite d'événements demeurera tout de même réaliste. Afin d'éliminer l'occurrence d'événements quasi-stochastiques, il faudra confier la gestion de l'arbre (ou d'un sous-arbre) en entier à un seul module. Il est à noter que l'artefact de synchronisation est également une forme d'événement quasi-stochastique. Cependant, il n'affecte pas le résultat final qui demeure toujours prévisible.

6.2 Évaluation de la performance

Afin de mesurer les performances du système final, l'application développée doit subir une série de tests qui comparent l'évolution du taux de rafraîchissement du module visuel selon différents paramètres. C'est le taux de rafraîchissement visuel qui a été choisi comme critère de performance car il mesure directement la qualité de l'immersion virtuelle. En effet, puisque le taux de rafraîchissement visuel est facilement observable, ce critère de performance aura un impact majeur sur le degré d'immersion et de satisfaction de l'utilisateur lors des interactions avec le système.

Les machines utilisées pour les différentes expérimentations possèdent la configuration suivante :

- MS Windows XP Professional SP2
- Processeur Dual Core AMD Opteron version 275 à 2.2 GHz
- 2 Go de mémoire vive DDR2 à 667 Mhz
- 2 cartes graphiques nVidia Quadro FX 4500 avec 1 Go de mémoire vive DDR3
- Carte réseau nVidia nForce Networking Controller à 1 Gbps

Dans le cas où l'application n'est pas distribuée sur le réseau, le module visuel sera configuré pour utiliser les quatre écrans du système FLEX et produire un rendu stéréoscopique. Avec une topologie réseau, seul le serveur définira une configuration à

quatre écrans. Les clients utiliseront une fenêtre de résolution inférieure et un rendu 2D et ne serviront que d'observateurs pour la réplication de la scène via le réseau.

La séquence de test consiste simplement à créer et positionner les objets l'un derrière l'autre et à les maintenir en mouvement selon une trajectoire prédéfinie par le module virtuel. Cette trajectoire est définie par les quatre coins d'un carré sur lesquels les objets sont téléportés à toutes les secondes par le module virtuel afin de conserver une charge sur le processeur pour la gestion de la synchronisation entre les modules. Le délai d'une seconde permettra à d'autres modules (comme le module haptique) d'affecter la position des objets virtuels pendant un court instant afin de charger davantage le module de synchronisation.

Les trois sections suivantes comparent le taux de rafraîchissement du module visuel par rapport au nombre d'objets dans la scène, au nombre de modules intégrés au système et au nombre d'instances de l'application sur le réseau avec cette séquence.

6.2.1 Taux de rafraîchissement par rapport au nombre d'objets

Ce premier test de performance consiste à établir la relation entre le taux de rafraîchissement et le nombre d'objets contenus dans la scène. Pour cette évaluation, seul le module visuel a été chargé et le même modèle 3D a été répliqué plusieurs fois pour chacun des objets.

Le graphique de la Figure 6-1 montre donc la relation entre le taux de rafraîchissement et le nombre d'objets virtuels contenus dans la scène. On peut voir que le taux de rafraîchissement diminue exponentiellement jusqu'à 150 objets où l'on retrouve un bref plateau. La baisse du taux de rafraîchissement reprend ensuite une courbe exponentielle jusqu'à 300 objets. Le plateau à 150 objets n'est pas une fluctuation du taux de rafraîchissement comme on en retrouve sur le reste de la courbe. En effet, lors de la séquence de test, le mécanisme de création des objets virtuels dispose les différents éléments l'un derrière l'autre jusqu'au $150^{ième}$. Par la suite, les 150 suivants sont positionnés à nouveau près de l'utilisateur sur une trajectoire qui forme un plus petit carré à l'intérieur du premier. Ce moment correspond à une seconde chute rapide du taux de

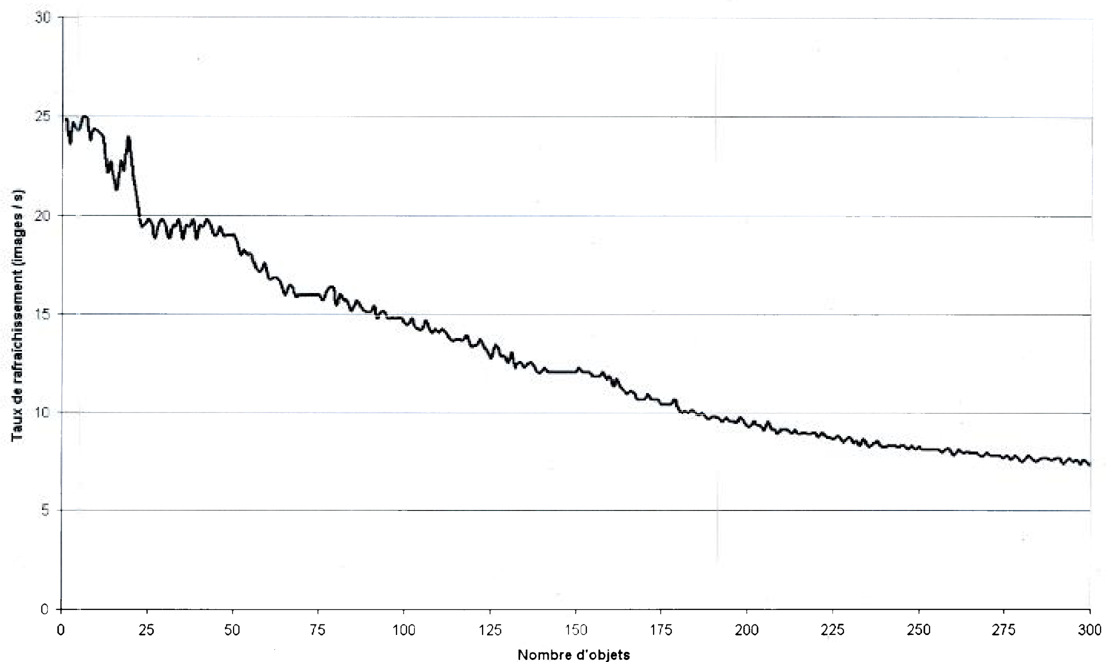


Figure 6-1: Taux de rafraîchissement par rapport au nombre d'objets virtuels

rafraîchissement ce qui suggère que les objets créés plus près de l'utilisateur affectent davantage les performances du module visuel. Pour vérifier cette hypothèse, la même séquence de test a été effectuée en retirant les objets du champ de vue de l'utilisateur (en regardant de l'autre côté). La performance du module visuel n'est alors pratiquement pas affectée par le nombre d'objets présents dans la scène. Les objets éloignés ou en dehors du champ de vue de l'utilisateur affectent moins significativement le taux de rafraîchissement du module visuel. Malgré tout, globalement, le taux de rafraîchissement du module visuel diminue donc en fonction du nombre d'objets virtuels présents dans la scène.

Dans un autre ordre d'idées, on peut constater que le taux de rafraîchissement initial de l'application est relativement faible (environ 25 images par seconde). Cette performance limitée est due à la synchronisation des images nécessaire pour le rendu stéréoscopique. Lorsque l'application ne simule par la troisième dimension, un taux de plus de 60 images par seconde a été observé pour la même séquence de test.

6.2.2 Taux de rafraîchissement par rapport au nombre de modules

Cette seconde série d'expérimentations consiste simplement à refaire le test de performance précédent en ajoutant un module à la fois au système. Ces tests permettront d'analyser le

comportement du taux de rafraîchissement en fonction du nombre de modules intégrés au système. Encore une fois, les objets sont créés l'un derrière l'autre et suivent la même trajectoire. Cependant, les objets sont disposés d'une telle façon que les volumes de collision s'interpénètrent. Lors de l'ajout du module haptique, cette approche aura pour effet de créer une forte charge de calcul pour la réaction aux collisions en plus de simuler la chute des objets due à la gravité pendant le court délai d'une seconde.

Le graphique de la Figure 6-2 reprend donc la courbe précédente comme base de comparaison et comprend également deux courbes additionnelles qui mesurent l'évolution du taux de rafraîchissement par rapport au nombre d'objets virtuels après l'ajout du module haptique et du module de distribution (avec un seul client).

Selon cette deuxième série de tests, on peut voir que le taux de rafraîchissement évolue de la même façon en ajoutant des modules : le plateau à 150 objets est toujours présent et le taux de rafraîchissement diminue en fonction du nombre d'objets. On remarque également que la courbe du taux de rafraîchissement descend au fur et à mesure que des modules sont

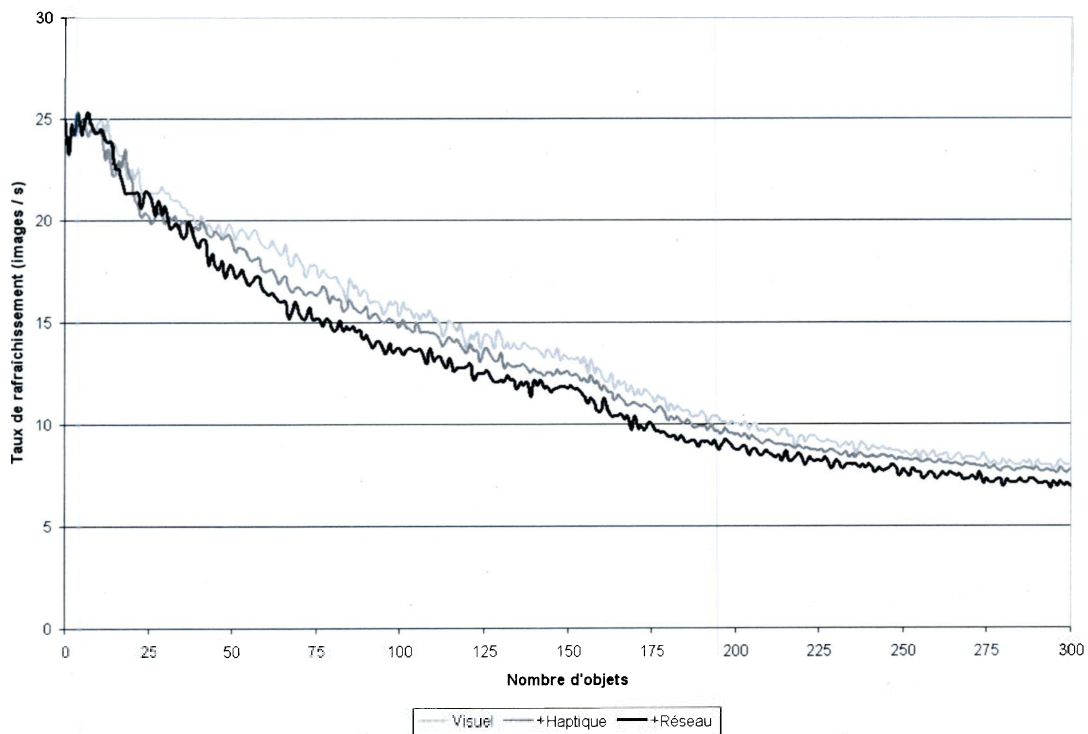


Figure 6-2: Taux de rafraîchissement par rapport au nombre de modules

ajoutés au système. Cette baisse de performance est due au partage des ressources du processeur entre les différents modules et au mécanisme de synchronisation qui représente une charge de travail de plus en plus importante pour l'application. Cependant, on constate que l'écart du taux de rafraîchissement entre les différents essais n'augmente pas avec le nombre d'objets. Ce résultat suggère que les latences de synchronisation augmentent proportionnellement au nombre d'objets et de modules et non pas exponentiellement. Cette observation démontre que le module de synchronisation ne forme pas le « *bottleneck* » du système. Le fait de verrouiller le tampon central contenant les données partagées à chaque mise à jour des objets à l'intérieur des différents modules semble donc une approche très performante. Par contre, une autre approche consistant à verrouiller la totalité des modules sauf celui qui fait la mise à jour aurait certainement entraîné une dégradation beaucoup plus rapide du taux de rafraîchissement car les avantages du *multi-threading* n'auraient alors pas été exploités au maximum.

6.2.3 Taux de rafraîchissement par rapport au nombre d'instances réseau

Finalement, cette dernière série d'expérimentations permet d'observer l'évolution du taux de rafraîchissement selon le nombre d'instances réseau formant l'application distribuée. Le graphique de la Figure 6-3 reprend d'abord les deux dernières courbes de la section précédente comme base de comparaison : celle avec les modules visuel et haptique seulement (aucun client) et celle avec le module de distribution à un seul client. Trois nouvelles courbes sont ensuite ajoutées au graphique pour mesurer l'évolution du taux de rafraîchissement après l'ajout d'un deuxième, d'un troisième et d'un quatrième client au système. Encore une fois, la même séquence de test a été utilisée pour ces expérimentations. Dans tous les cas, le module haptique est intégré au système afin de maximiser le nombre de mises à jour et l'utilisation de la bande passante sur le réseau.

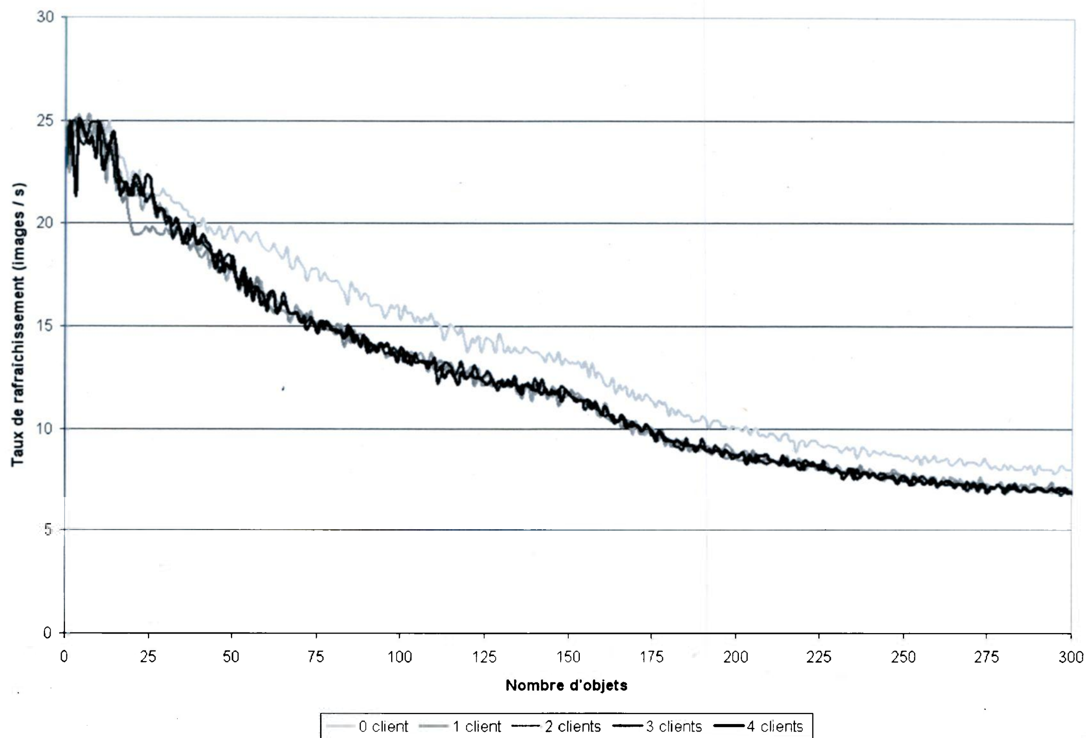


Figure 6-3: Taux de rafraîchissement par rapport au nombre d'instances réseau

Selon ce dernier graphique, on remarque que les quatre courbes qui mesurent la performance du système avec le module de distribution sont toutes confondues. L'évolution du taux de rafraîchissement demeure donc sensiblement la même malgré l'ajout des clients suivants. Ce résultat démontre que la bande passante du réseau n'est jamais utilisée au maximum malgré le nombre grandissant d'objets et de clients. En effet, puisque la vitesse de transmission disponible sur le réseau est de 1 Gbps et que les messages de mise à jour UDP transmis sont relativement petits (moins de 1 Ko), il sera difficile de saturer le module de distribution avec les fonctionnalités simples et limitées du système actuel.

6.3 Évaluation empirique

Suite aux expérimentations précédentes qui fournissent une évaluation objective de l'application finale, ces deux dernières sections couvriront des aspects plus empiriques de l'évaluation du système. Tout d'abord, la première section présente une évaluation du réalisme de l'application et la seconde décrira l'ergonomie de l'interface utilisateur.

6.3.1 Évaluation du réalisme de l'environnement

Puisque le but premier du projet n'était pas le réalisme, des fonctionnalités sensorielles de base ont été ajoutées au système afin de développer et de tester les fonctionnalités de synchronisation, de cohérence et de distribution qui forment les objectifs prioritaires du projet.

Tout d'abord, le module de visualisation procure un rendu 3D stéréoscopique qui permet d'immerger l'utilisateur dans une représentation en trois dimensions de l'environnement virtuel à l'intérieur duquel la profondeur et la distance des différents objets virtuels peuvent facilement être évaluées. Grâce à l'information contenue dans la troisième dimension, l'utilisateur pourra facilement identifier la position des objets par rapport aux autres et déterminer la distance à parcourir pour atteindre un point lors de la manipulation et la navigation.

Par contre, le module de visualisation exploite à peine l'ensemble des fonctionnalités qu'offre la librairie OSG. En effet, l'ombrage n'est pas présent dans la scène, les effets de lumière ne sont pas dynamiques, l'application des textures et les effets de transparence sont simples, il n'y a pas de réflexion de la lumière, etc. Bref, seule la méthode de rendu de base est utilisée et les modèles 3D chargés en mémoire sont affichés tel quel sans modifications. Malgré ces fonctionnalités de visualisation très simples, le module résultant demeure suffisant pour la compréhension de l'environnement et les interactions que l'utilisateur doit effectuer.

Bien que le module de visualisation soit le seul à pouvoir souffrir d'un manque de réalisme dans tous les modules développés dans le cadre de ce projet[1], le système final possède tout de même certaines lacunes au niveau des autres sens. Tout d'abord, le module haptique présenté dans (duTremblay à venir) offre un retour de force peu réaliste à l'utilisateur avec le CyberGrasp. L'utilisation du CyberGlove permet quant à lui de manipuler les différents objets virtuels, mais la présence du module de visualisation est alors obligatoire afin de pouvoir identifier les objets à manipuler. De plus, la forme des objets composant la scène

[1] Les modules de synchronisation, de distribution et d'intégration sont des modules fonctionnels qui ne fournissent aucune information sensorielle à l'utilisateur. Le réalisme n'est donc pas un critère d'évaluation envisageable pour ces modules. Le module haptique a été développé au cours d'un autre projet.

est généralement approximée par un volume simple comme une boîte ou une sphère afin d'obtenir une performance accrue lors de la simulation des lois de la physique.

Un autre module sensoriel intéressant à ajouter au système serait un module auditif. Grâce à cette nouvelle information sensorielle, l'utilisateur pourrait identifier la provenance et l'emplacement de certains objets sonores sans les voir directement sur la surface de projection. Ce nouveau module augmenterait grandement l'effet d'immersion et le réalisme de l'environnement virtuel. Pour le goût et l'odorat, la principale limitation demeure toujours la technologie qui n'est pas très avancée dans ces deux domaines.

6.3.2 Évaluation de l'aspect intuitif de l'interface

Beaucoup d'utilisateurs sont familiers avec les métaphores de fenêtres et de boutons pour effectuer des actions offertes par des suites de systèmes d'exploitation comme Windows car ces logiciels sont utilisés dans le monde entier. Il est donc avantageux de développer des interfaces qui sont basées sur les classes MFC. Évidemment, d'autres outils logiciels implémentent également des métaphores de fenêtres et de boutons et certains outils sont portables d'un système d'exploitation à l'autre.

L'interface de configuration et l'interface utilisateur 3D sont deux exemples différents de l'application des métaphores employées dans les classes MFC. Tout d'abord, l'interface de configuration présentée dans la section 5.2.1 utilise directement les fenêtres et les boutons 2D de Windows. L'interface est donc facile à comprendre et à utiliser de façon intuitive. L'interface utilisateur 3D quant à elle applique les fonctionnalités des fenêtres et des boutons dans l'environnement virtuel en les représentant en 3D. L'utilisation de la souris n'est alors plus nécessaire car les traqueurs de position et le gant CyberGlove permettront à l'utilisateur de déplacer les fenêtres avec les mains et de cliquer directement sur les différents boutons du bout des doigts. Malgré leurs différences fondamentales, ces deux interfaces offrent une façon simple et intuitive à l'utilisateur d'effectuer des tâches prédéfinies comme la création d'objets ou l'activation d'un événement.

Pour la navigation, la *wand* est très facile à utiliser. Les deux axes analogiques permettent les déplacements à l'intérieur du monde virtuel et différents boutons peuvent être utilisés

pour exécuter des actions spécifiques. Cependant, cette approche simple n'est pas la plus naturelle et elle oblige l'utilisateur à tenir un objet en main continuellement. Le système de locomotion NELI tentera de remédier à ces contraintes de navigation en offrant un mécanisme permettant de se déplacer naturellement en marchant dans un environnement virtuel. Dans le cadre de ce projet, la métaphore *ski-pole* a été développée pour simuler le mécanisme de locomotion. Bien que l'utilisateur doive utiliser ses mains pour se déplacer, il peut tout de même conserver les mains libres. De plus, le mouvement circulaire qu'il doit effectuer pour induire la propulsion demeure assez naturel.

L'utilisation de symboles et de gestes effectués avec les gants de réalité virtuelle est une approche intéressante pour effectuer rapidement des actions sans faire appel à une interface graphique 3D. Puisque la manipulation d'un menu et les interactions avec les boutons demandent un certain temps à l'utilisateur, des symboles et des gestes permettront d'effectuer des actions qui doivent souvent être exécutées. Par contre, la détection des symboles devient alors la partie la plus difficile. En effet, plus il y aura de symboles, plus il sera difficile de les distinguer car la marge d'erreur due à la calibration du gant devient plus significative. De plus, cette approche ne sera pas nécessairement naturelle si le nombre de symboles est restreint car il devient alors difficile de trouver un lien entre la signification du symbole et celle de l'action. Des systèmes complexes permettent de reconnaître le langage des signes utilisés par les sourds et muets. Cependant, cette façon de communiquer n'est malheureusement pas naturelle pour tout le monde et demande un temps d'apprentissage non négligeable.

Finalement, l'intégration d'un système de reconnaissance vocale serait également très intéressant pour rendre les interactions avec l'application encore plus naturelles pour l'utilisateur. Cependant, les outils logiciels permettant une « conversation » naturelle et fluide avec le système sont très rares et le développement d'une telle interface peut facilement devenir assez complexe.

Conclusion

Plusieurs technologies matérielles et logicielles ont été étudiées et développées au cours de ce projet afin d'atteindre l'objectif principal qui consistait à produire un environnement virtuel multi-sensoriel modulaire et distribué qui soit cohérent et synchronisé.

Tout d'abord, les principes du *multi-threading* encapsulés dans la librairie OpenThreads ont été intégrés au module de synchronisation pour le partage de l'information commune de position, d'orientation et de facteurs d'échelles entre les différents modules. C'est un tampon central, protégé par un *mutex*, qui sert de point d'accès commun à tous les modules pour la lecture et l'écriture de ces données partagées. Un processus de synchronisation en trois étapes (lecture, mise à jour locale et écriture) a également été défini afin d'effectuer la tâche de synchronisation de façon transparente. De plus, un principe de priorité et la fonctionnalité de verrou permettront de contrôler davantage la cohérence du système selon le rôle accordé à chacun des modules. Finalement, un mécanisme d'événements permet d'ajouter un peu de flexibilité pour une communication limitée entre les modules. Le module de synchronisation forme donc le cœur de système en assurant la cohérence et la synchronisation entre les différents modules qui seront intégrés par la suite.

Ensuite, les librairies OSG et VRJ forment la base du module de visualisation qui permet à l'utilisateur d'observer la scène virtuelle et les différents objets qui la composent en trois dimensions. Bien que les fonctionnalités exploitées lors du rendu soient assez simples (pas d'ombrage, pas de lumière dynamique, etc.), le module visuel demeure tout de même suffisant pour immerger l'utilisateur et lui permettre de reconnaître les différents éléments de la scène et de s'y retrouver d'un seul coup d'œil. Le rendu stéréoscopique semble être la principale cause de la baisse du taux de rafraîchissement. Cependant, la performance du module visuel demeure toutefois suffisante pour l'application finale.

Puis, les protocoles UDP et TCP sont utilisés conjointement dans le module de distribution pour former une méthode de communication hybride entre les différentes instances de l'application. En effet, les événements critiques à la cohérence de la scène comme la création, la destruction et la téléportation d'objets seront envoyés par TCP afin d'assurer la réception. Cependant, les mises à jour fréquentes lors du déplacement des objets seront

plutôt expédiées via le protocole UDP. Dans les deux cas, une structure à balises XML sera utilisée pour définir les messages car ce format permettra une extraction facile et rapide des différents paramètres. Finalement, grâce au module de distribution, la topologie de l'application distribuée devient très flexible. Par exemple, les différents modules utilisés par un même utilisateur pourront être distribués sur plusieurs machines pour former une application distribuée mono-utilisateur et ainsi augmenter le parallélisme de l'application. Une approche multi-utilisateurs classique est également envisageable.

Finalement, le module d'intégration, appelé également univers virtuel, encapsule les fonctionnalités du module de synchronisation et permet l'intégration des différents modules et périphériques. Le module d'intégration définit l'objet virtuel qui se compose des différentes propriétés sensorielles et fonctionnelles des modules ajoutés au système. Un objet virtuel peut également être traqué et implémenter un comportement de haut niveau quelconque. C'est également le module d'intégration qui définit l'utilisateur virtuel et les différentes parties de son corps (tête, mains et pieds). Afin de permettre à l'utilisateur de manipuler les objets virtuels avec les mains, le module haptique de (duTremblay à venir) a été intégré au système final. Le module haptique permet également la simulation des lois de la physique pour les collisions et la gravité sur les différents objets. L'univers virtuel propose également à l'utilisateur une série d'interfaces qui lui permettront de configurer et d'interagir avec le système.

Le but secondaire du projet étant de développer un simulateur pour le mécanisme de locomotion à câbles NELI, d'autres fonctionnalités ont également du être ajoutées à l'application finale. Tout d'abord, le module haptique permet d'interfacer le mécanisme NELI, détecte les collisions du pied avec le sol et calcule les forces appliquées sur les pieds à tout moment lors d'un contact avec le sol. La direction et l'amplitude de ces forces varieront selon le type et la topologie du terrain. Puis, afin de simuler le fonctionnement du système de locomotion grâce aux périphériques disponibles, une métaphore de navigation nommée *ski-pole* a été développée. Cette métaphore consiste à propulser l'utilisateur dans la direction inverse au mouvement du pied lorsque ce dernier est en contact avec le sol. Afin de rendre la navigation plus confortable, le pied virtuel peut être déplacé avec les

mains. Il suffit alors de faire un mouvement circulaire semblable à celui du ski de fond à l'aide des deux mains pour propulser l'utilisateur dans la direction désirée.

En conclusion, grâce à l'environnement virtuel et aux fonctionnalités développés au cours de ce projet, il sera possible de produire une plate-forme de test et de simulation plus complète pour le mécanisme de locomotion NELI. Les paramètres et les contraintes mécaniques pourront être intégrés au système pour obtenir un comportement simulé plus réaliste. L'approche modulaire flexible et le processus de synchronisation performant assurant la cohérence du système deviennent donc les deux éléments clés qui permettront d'étendre les fonctionnalités de l'application dans le futur. En effet, d'autres modules pourront être développés afin d'étendre les aspects sensoriels et fonctionnels du système.

La synchronisation et la cohérence sont deux objectifs très importants à atteindre dans un environnement virtuel multi-sensoriel afin d'obtenir une application stable et robuste. Le réalisme demeure sans aucun doute l'élément le plus difficile à atteindre. En effet, pour ce qui est du réalisme visuel, les techniques de rendu ont beaucoup évolué dans les dernières années grâce à l'industrie du jeu vidéo, très active dans ce domaine. Même si plusieurs heures de travail sont nécessaires pour obtenir une scène visuelle complexe et réaliste, la technologie évolue rapidement dans cette direction. Par contre, les technologies qui amènent un réalisme au niveau des autres sens comme le goût et l'odorat sont encore très limitées. Améliorer le réalisme sensoriel tout en conservant une performance acceptable sera donc certainement un défi technologique de taille pour la réalité virtuelle au cours des prochaines années. Finalement, ces nouvelles avancées permettront d'obtenir un effet d'immersion inégalé où l'utilisateur ne pourra presque plus faire la différence entre le virtuel et la réalité!

Bibliographie

(2003). "Elumens VisionStation VS/V3." from http://www.est-kl.com/projection/elumens/vs.html.

(2005). "Actuality Systems." from http://www.actuality-medical.com/indexAS.html.

(2006). Advanced Visualization Systems, Fakespace Systems Inc.

(2007). "IO2 Technologies." from http://www.io2technology.com/.

(2008). "Immersion Corporation." from http://www.immersion.com/.

(2008). "InterSense, Inc." from http://www.intersense.com/.

(2008). "Mechdyne Corporation: 3D and Advanced Visualization Solutions." from http://www.mechdyne.com/.

(2008). "Motion Tracking, 3D Scanning, and Eye Tracking Solutions from Polhemus." from http://www.polhemus.com/.

(2008). "NewSight - Advanced Display Solutions." from http://www.newsight.com/.

(2008). "ProView SR80." from http://www.rockwellcollins.com.

(2008). "SR Research EyeLink: Fast, Accurate, Reliable Eye Tracking." from http://www.eyelinkinfo.com.

Amir, Y. (2006). "Distributed Systems." from http://www.cs.jhu.edu.

Baker, M. (2007). "EuclideanSpace - Building a 3D World." from http://www.euclideanspace.com.

Bowman, D. A., E. Kruijff, et al. (2005). 3D User Interfaces: Theory and Pratice. Boston, Addison Wesley.

Cruz-Neira, C., D. J. Sandin, et al. (1993). Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. SIGGRAPH 93. Anaheim, California, USA.

Drolet, F. (2006). Réalité virtuelle dans un environnement immersif à projecteurs multiples: exploration et familiarisation. Québec, DRDC-RDDC Valcartier.

duTremblay, C. (à venir). Interactions haptiques et applications de simulation à l'aide de CyberGloves. Département de génie électrique et de génie informatique. Québec, Université Laval. **M. Sc.**

Forsyth, D. A. and J. Ponce (2003). Computer Vision: A Modern Approach. New Jersey, Prentice Hall.

Gosselin, C. NELI: A Network-Enabled Locomotion Interface. Québec, Canada, Département de génie mécanique, Université Laval.

Infiscape. (2007). "The VR Juggler Suite." from http://www.vrjuggler.org/.

Infiscape. (2008). "The VR Juggler Suite." from http://www.vrjuggler.org/.

Laurendeau, D., A. Branzan-Albu, et al. (2003). Survey of the State-of-the-Art on Synthetic Environments, Sensori-Motor Activities in Synthetic Environments, Simulation Frameworks and Real-World Abstraction Models. Val-Bélair (QC), Canada, DRDC-RDDC Valcartier.

Li, W.-J., C.-C. Chang, et al. (2001). "A PC-based distributed multiple display virtual reality system." Elsevier, from www.elsevier.com.

Osfield, R. (2007). "OpenSceneGraph." from http://www.openscenegraph.org.

Osfield, R. (2008). "OpenSceneGraph." from http://www.openscenegraph.org.

Perreault, S. (2007). Conception mécanique d'une plate-forme de marche entraînée par câbles. Département de génie mécanique. Québec, Université Laval. **M. Sc.**

Perreault, S. and C. M. Gosselin (2007). Cable-Driven Parallel Mechanisms: Application to a Locomotion Interface. IDETC/CIE 2007, Las Vegas, Nevada, ASME.

Sherman, W. R. and A. B. Craig (2003). Understanding Virtual Reality: Interface, Application, and Design. San Francisco, Morgan Kaufmann.