OpenMASK: {Multi-threaded | Modular} Animation and Simulation {Kernel | Kit}: un bref survol

David Margery, Bruno Arnaldi, Alain Chauffaut, Stéphane Donikian et Thierry Duval

Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France
David.Margery@irisa.fr

Résumé : *Dans cet article, nous présentons OpenMASK, un noyau d'animation et de simulation multi activités qui est aussi une plate-forme pour le développement et l'exécution d'applications modulaires dans le domaine de l'animation, de la simulation et de la réalité virtuelle. OpenMASK est le résultat d'un compromis original entre la performance, l'abstraction et la modularité permettant de répondre aux besoins d'une grande variété d'applications de réalité virtuelle, y compris dans des contextes d'exécution distribuée. Cet article présente les objectifs, les concepts, les notions de base et la performance de la version publique* [1] *d'OpenMASK.*

Mots-clés : Simulation interactive, réalité virtuelle distribuée, boite à outil pour la réalité virtuelle

1 Introduction

Les applications des technologies de la réalité virtuelle sont nombreuses et prometteuses. Néanmoins, leur usage reste à ce jour encore relativement limité à cause de la complexité d'intégration dans un même logiciel de tous les composants logiciels et matériels nécessaires. En effet, pour une application donnée, le savoir faire, et donc le code, provenant de différente spécialités doit être assemblé sans nuire à la performance globale. En effet, les utilisateurs des technologies de la réalité virtuelle demandent toujours plus de réalisme et d'interactivité parce que les deux promesses de la réalité virtuelle sont celles d'une grande richesse de retour sensoriel et d'outils d'interaction puissants.

Cependant, afin d'être utiles, ce retour sensoriel et cette interaction doivent être calculés dans un intervalle de temps contraint, ce qui n'est pas toujours compatible avec le temps de calcul des simulations sous-jacentes. En effet, la réalité virtuelle est souvent utilisée pour interagir avec des simulations qui sont elles-mêmes coûteuses en temps de calcul, ce qui conduit à l'utilisation du parallélisme ou de la distribution pour réduire celui-ci. Ce facteur doit être pris en compte dès la conception d'un environnement pour la programmation et l'exécution d'applications de réalité virtuelle.

1.1 Problématique et résultats présentés

Un problème fondamental en réalité virtuelle est donc de concilier performance, multi-résolution des temps de calcul et abstraction. La performance est centrale à la richesse d'interaction et du retour sensoriel. La multi-résolution est importante parce que les applications de réalité virtuelle doivent coordonner des cycles de calcul de granularité différente. Mais on ne peut négliger l'abstraction, parce qu'intégrer dans la même application des composants logiciels d'origines différentes implique une abstraction commune à ces logiciels. Dans cet article nous présentons une telle abstraction : OpenMASK. OpenMASK est une plate-forme pour le développement et l'exécution d'applications modulaires dans le domaine de l'animation, de la simulation et de la réalité virtuelle. Elle est déjà utilisée au sein du projet Siames de l'Irisa, pour des applications de travail collaboratif (fig.2), de simulation mécanique et d'animation comportementale (fig.1). L'utilisation d'une même plate-forme pour ces programmes aux contraintes différentes est à notre sens un résultat significatif, puisqu'il démontre la possibilité de réutilisation de composants logiciels et donc qu'il abaisse le coût logiciel lié à l'utilisation de la réalité virtuelle. De plus, en utilisant la distribution non seulement à des fins de collaboration mais aussi pour améliorer la performance, la richesse des environnements virtuels produits s'en trouve augmentée.

1. voir http://www.openmask.org pour les détails

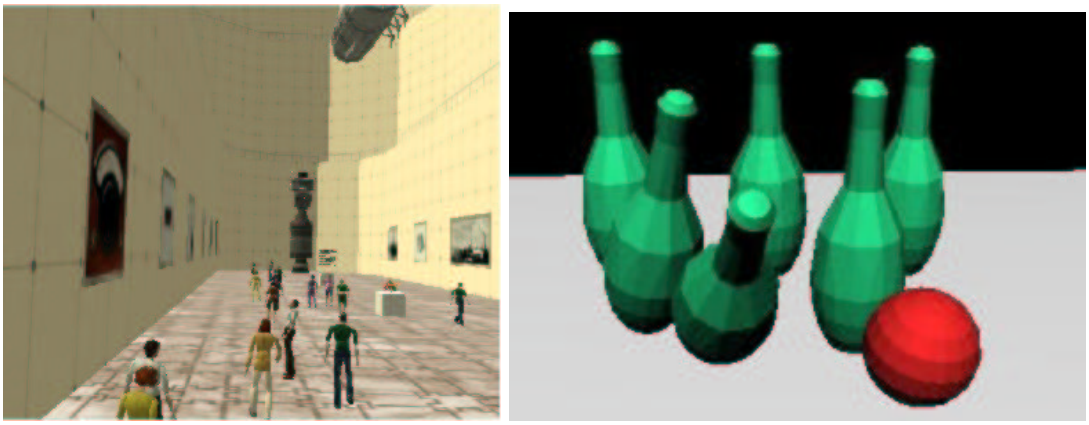

FIG. 1 – *Une simulation comportementale et une simulation mécanique utilisant OpenMASK*

1.2 Travaux connexes

Il y a de nombreuses boîtes à outils pour la réalité virtuelle. Que ce soit les premières boîtes à outils telles que MR-Toolkit[SG93] ou des plus récentes telles que VR-Juggler[JBBCN98], CAVERNSoft[KYN+00], Avango[Tra99], ou Maverick[RJM+99], l'objectif principal semble toujours d'abstraire le matériel pour le programmeur, permettant ainsi le changement de matériel, soit pendant l'exécution soit entre deux exécutions du programme. Ce travail est nécessaire, mais en général il ne fournit pas au programmeur un cadre conceptuel, forçant soit une reconception de la structure de l'application à chaque nouvelle application, soit l'utilisation du cadre conceptuel très basique fourni avec la boite à outils. Par conséquent, ces boîtes à outils, bien qu'utiles pour un développement plus rapide, n'aident pas à structurer l'application en composants qui pourraient facilement être réutilisés.

Dans le domaine de la réalité virtuelle distribuée, des cadres conceptuels plus complexes ont été proposés, parce qu'ils deviennent nécessaires pour pouvoir assurer la cohérence à travers le réseau avec de bonnes performances. Ceux-ci, qu'ils soient indépendants (Aviary[WHH+92], VEOS[BC94] ou Spline[BWA96]) ou basés sur une des boîtes à outils présentées ci-dessus (Environment Manager au-dessus de MR-Toolkit[WGS95] ou Deva[SJJA00]), se préoccupent davantage de la structuration de l'environnement virtuel que de la structuration logicielle de l'application. C'est pourquoi les cadres conceptuels proposés ne sont adaptés qu'à une classe limitée d'environnements virtuels. De plus, ces projets favorisent l'abstraction à la performance. Néanmoins, ces abstractions ont grandement influencées notre travail parce qu'elles cherchent toutes à définir les abstractions de base nécessaires dans une application de réalité virtuelle, et dont certaines sont déjà présentes dans GASP[SATR98, TD00b, TD00a], le prédécesseur d'OpenMASK.

Dans le même contexte de réalité virtuelle distribuée, des recherches visant la performance ont concentré leur attention sur l'infrastructure logicielle sous-jacente nécessaire à une distribution performante d'un environnement virtuel. La performance de ces systèmes distribués (la série MASSIVE [GB95, CJD00], la série NPSNET [MDM+95, CMBZ00] ou Community Place[LHMM97]) est basée sur un relâchement de la cohérence des différentes copies de l'environnement virtuel et sur la limitation du nombre d'objets impliqués dans les algorithmes de maintient de la cohérence. Là encore, les résultats obtenus ne sont applicables qu'à condition d'avoir prévu les objets explicitement pour la distribution. Dans OpenMASK, nous utilisons la cohérence relâché contrainte (décrite dans [Mar01]), mais en utilisant des informations systèmes pour limiter le nombre d'objets, une stratégie ne limitant pas le nombre possible d'applications.

Un dernier domaine connexe est le vaste domaine de la réutilisation de code, et plus spécifiquement de la construction d'une infrastructure logicielle capable de charger, décharger, connecter et découvrir des composants logiciels. Cette approche est relativement récente dans le cadre de la réalité virtuelle avec des prototypes tels que JADE[MJM00], Bamboo[Wat00] ou NPSNET-V[CMBZ00]. Une telle dynamicité n'est pas possible avec OpenMASK, qui doit avoir la connaissance a priori de tous les composants logiciels utilisés par une application donnée. Cependant, la notion de composants logiciels d'OpenMASK est beaucoup plus structurée que l'approche très fondamentale des projets cités précédemment, et ce parce qu'ils se placent à un niveau plus abstrait.

OpenMASK est donc notre contribution à la recherche du meilleur compromis possible lors de la conception

d'un environnement de développement et d'exécution pour la réalité virtuelle qui prenne en compte les problématiques de performances, d'abstraction et de distribution. Cet article est structuré de la façon suivante. Dans la prochaine section, les concepts de base d'OpenMASK sont présentés. Dans la section 3, nous présentons les politiques d'ordonnancement des objets de simulation d'OpenMASK (l'unité de modularité, composant logiciel d'OpenMASK) puis dans la section 4 les outils pour permettre la communication et donc l'interaction entre ces objets. La gestion de ces objets est ensuite présentée dans la section 5 avant que soient expliqués les principes et les performances du noyau d'exécution distribué.

2 Présentation générale d'OpenMASK

2.1 Objectifs de conception

L'objectif principal d'OpenMASK est de fournir un noyau d'animation et de simulation :

1. indépendant du niveau d'animation (descriptive, générative or comportementale) utilisé ;
2. indépendant du style de programmation de l'animation (réactive, orienté agent, objets actifs . . .) utilisé ;
3. indépendant de la bibliothèque de rendu utilisée ;
4. capable d'utiliser plusieurs activités en parallèle pour le calcul de la simulation ;
5. indépendant du type d'exécution multi-activité utilisé (calcul distribué ou calcul parallèle).

Ces deux derniers points ont particulièrement influencés la conception du noyau d'OpenMASK, puisque notre objectif a été de permettre un parallélisme et une distribution performante. De plus, définir les objets manipulés par le noyau revient à privilégier une certaine granularité de l'animation et donc certains niveaux et styles d'animation.

Cependant, il convient de noter que les choix de conception sont davantage biaisés en faveur de l'exécution multi-activité qu'en faveur d'un niveau ou d'un style d'animation particulier. De plus, les outils fournis avec OpenMASK ont pour objectif de permettre à un programmeur l'expression de ses algorithmes d'animation dans le style naturel à ceux-ci, afin de permettre au noyau de manipuler des objets ayant une sémantique forte, et donc de faire des optimisations pertinentes.

Dans cet article, les termes simulation et animation sont utilisés avec un sens légèrement différent. Le terme simulation est utilisé lorsque le résultat produit (qui peut ne pas être visuel) est plus important que le temps mis à le produire, alors que le terme animation est utilisé pour des simulations interactives pour lesquelles l'interactivité est un aspect clé de l'application. Nous croyons que les mêmes composants logiciels devraient pouvoir être utilisés pour ces deux types d'applications, avec des optimisations différentes effectuées par le noyau en fonction du contexte d'utilisation. Cependant, dans cet article, nous ne nous intéresserons que très peu au contexte d'exécution et c'est pourquoi nous y discutons de la simulation d'objets de simulation sans présumer du contexte de leur utilisation.

FIG. 2 – *Une application de réalité virtuelle coopérative*

2.2 Concepts de base d'OpenMASK

Dans OpenMASK, la brique de base pour construire une application est l'objet de simulation. C'est au sein de l'objet de simulation que tout le code décrivant l'évolution de l'objet et son interaction avec les autres objets est

localisé. Les deux questions qui se posent sont donc les suivantes:

1. Quand le code de l'objet est-il exécuté? C'est la question de l'activation de l'objet.
2. Comment les objets communiquent-ils entre eux? C'est la question de la communication.

Une troisième question se pose : quelle est la granularité de l'objet de simulation? Ou encore, qu'anime un objet de simulation? Il s'agit d'une question à laquelle OpenMASK ne répond pas, puisque que l'objectif est de réaliser un noyau d'animation et de simulation qui soit le plus général possible. L'expérience d'utilisation d'OpenMASK montre que la granularité d'un objet de simulation varie d'un humanoïde virtuel complet avec ses comportements à une sphère inerte et ce dans la même application. Le meilleur exemple de la flexibilité et de la généralité du concept d'objet de simulation est que le rendu des environnements virtuels conçus au sein du projet Siames est réalisé à travers un objet de simulation appelé OpenMASK-3DVis [2] et qui gère le rendu sur dispositif immersif (workbench ou reality center) aussi bien que sur une station de travail, avec ou sans stéréo-vision.

3 Activation de l'évolution des objets

Chaque objet de simulation peut avoir le calcul de son évolution déclenché par deux méthodes. La première, la méthode `compute` est appelée à une certaine fréquence pour tous les objets dans l'état actif (voir fig 3 pour un diagramme des états possibles) ayant une fréquence non-nulle. La seconde méthode est liée au traitement des événements. Pour les objets dans l'état suspendu, le traitement des événements se fait au rythme de réception des événements, soit au maximum à la fréquence du contrôleur. Pour les objets dans l'état actif, le traitement des événements a lieu à la fréquence d'activation de l'objet par défaut, mais il est possible de le faire à la fréquence du contrôleur.

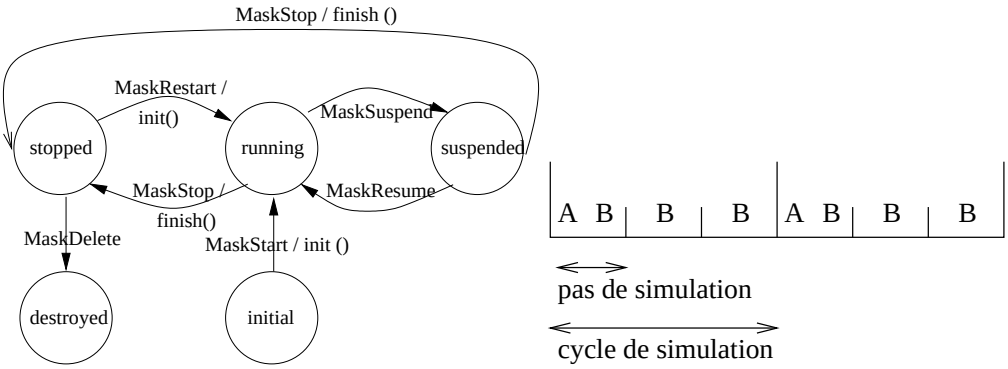

FIG. 3 – *Diagramme des états possibles d'un objet de simulation et cycles d'activation dans OpenMASK*

3.1 Politique d'ordonnancement

L'unité minimal d'ordonnancement d'OpenMASK est le pas de simulation (voir fig 3, où la fréquence de A est de 10Hz et celle de B 30Hz). Toute tâche ordonnancée est exécutée dans les limites d'un pas de simulation. Du point de vue de l'ordonnancement, OpenMASK est un système synchrone : aucun objet ne peut être ordonnancé 2 fois pendant le calcul d'un autre objet. Ce dernier point est souvent mal compris, parce que les objets peuvent avoir des fréquences différentes.

Pour calculer les structures d'ordonnancement, le contrôleur calcule le pgcd et le ppcm des fréquences non nulles de tous les objets de simulation. Le pgcd est alors la fréquence d'un cycle de simulation et le ppcm la fréquence d'un pas de simulation. On obtient alors, du point de vue de l'ordonnanceur un certain nombre de pas de simulation dans un cycle de simulation. Pour un objet ayant la fréquence d'un cycle de simulation, il sera ordonnancé sur le premier pas de simulation [3] de chaque cycle (la cas de l'objet A de la figure 3). Cependant, un

2. Aussi disponible à l'adresse www.openmask.org
3. Un choix arbitraire d'implémentation

pas de simulation ne peut commencer que lorsque le calcul de tous les objets ordonnancés sur le pas de simulation précédant ont terminé leur calcul.

La conséquence la plus gênante de cette stratégie d'ordonnancement est de faire varier le temps de calcul pour chaque pas de simulation en fonction du nombre et de la nature des objets ordonnancés à chaque pas. Même si du point de vue des objets simulés le temps s'écoule régulièrement, du point de vue d'un observateur extérieur à la simulation le temps peut s'écouler par à coups. C'est pourquoi il faut écrire les applications utilisant des objets fonctionnant à différentes fréquences de manière prudente, sinon le résultat obtenu pourrait être différent de celui attendu.

Dans un avenir proche, ce problème doit être réduit à cause du besoin de faire coopérer dans une même application des objets ayant des fréquences d'ordre de grandeur différent (retour haptique et visuel par exemple). Cependant, à cause des racines synchrones d'OpenMASK, la résolution de ce problème n'est pas triviale.

4 Gestion des objets

4.1 Les classes de base pour la gestion des objets

Plusieurs classes sont utilisées par OpenMASK pour la gestion des objets. Les objets de simulation sont connus entre eux grâce au descripteur d'objet (*objectDescriptor*), qui est une donnée qui n'évolue pas et qui contient le nom de l'objet, sa classe, et ses paramètres d'ordonnancement et de configuration.

Le contrôleur *PsController* est l'objet responsable pour toutes les fonctions globales à la simulation, en particulier pour la création, l'ordonnancement et la destruction des objets. Par ailleurs, le contrôleur est l'objet qui maintient une date de simulation globale à la simulation.

Le contrôleur ne manipule pas directement les objets de simulation, sauf lors de leur création et de leur destruction. Pour toutes les autres opérations, le contrôleur manipule un *PsObjectHandle* dans lequel sont stockés toutes les données d'un objet liées à l'exécution. Lors de l'utilisation d'un noyau distribué, un référentiel est utilisé pour manipuler la version de référence de l'objet de simulation (celle qui effectue les calculs), et un miroir est utilisé lorsqu'un contrôleur manipule une copie de l'objet pour permettre la communication entre objets simulé sur des nœuds différents.

4.2 L'arbre de simulation

Pour une application donnée, les objets de simulation sont structurés dans un arbre de simulation. L'objet racine de cet arbre est le contrôleur, mais le reste de la sémantique de l'arbre de simulation est laissé au concepteur d'application. Il y a deux fonctionnalités d'OpenMASK dont le comportement change en fonction de la structuration de l'arbre de simulation.

1. la création d'objet, puisque l'interprétation de la chaîne de caractères représentant la classe de l'objet à créer est faite récursivement à partir de l'objet père de l'objet à créer. Les prochaines versions d'OpenMASK pourrait faire la distinction entre le créateur d'un objet et le père de celui ci pour l'interprétation de cette chaîne de caractère, influant sur le type exact de l'objet créé.
2. les fonctions de recherche d'un objet. L'arbre de simulation permet de faire des recherches globales ou limités à un sous-arbre, aux frères ou encore au fils dans l'arbre de simulation.

Il convient pour finir de noter que l'arbre de simulation est une notion totalement différente de l'arbre d'héritage ou du graphe de scène.

4.3 La création d'objets

Les objets de simulation d'OpenMASK peuvent être créés soit :

1. statiquement : leur description est faite dans l'arbre de simulation paramètre du contrôleur à la création ;
2. dynamiquement, à l'initiative du programmeur ;

3. dynamiquement, en fonction des besoins des autres objets. Ceci est vrai en particulier dans le cas de l'exécution distribuée, puisque des objets miroirs sont créés au besoin pour communiquer avec les objets de simulation de référence.

Le C++ ne fournit pas de méthodes pour créer un objet à partir de la chaîne de caractère décrivant sa classe. C'est pourquoi nous utilisons un mécanisme spécifique : tous les objets, y compris le contrôleur ont une liste des classes d'objets qu'ils sont capables de créer. Pour chacune de ces classes, l'objet possède un pointeur sur un objet de type *PsObjectCreator* dont l'appel à une méthode spécifique provoque l'instanciation d'un nouvel objet.

5 La communication entre les objets de simulation

Il y a de nombreux outils sous OpenMASK pour faire communiquer les objets entre eux. Seulement, il faut se limiter aux outils fournis par le noyau d'OpenMASK afin de profiter des propriétés de calcul parallèle d'OpenMASK. En particulier, ceci implique que l'appel de méthodes entre objets de simulation est à proscrire, sauf dans certains contextes particuliers.

La raison de ce choix est que l'exécution multi-activité introduit des problèmes d'intégrité des données. Un noyau tel que celui d'OpenMASK peut gérer ces problèmes en protégeant les données de façon à éviter au programmeur d'objet de simulation de devoir comprendre le paradigme d'exécution multi-activité utilisé. Par contre, le noyau ne peut intercepter les appels de méthodes fait entre les objets de simulation (à moins d'introduire une indirection par précompilation du programme) et c'est pourquoi ceux-ci sont interdits.

Ce choix constitue le compromis de conception fondamental d'OpenMASK. De ce choix dérive les bonnes propriétés pour le calcul parallèle d'une simulation, mais aussi les quelques contraintes de programmation. D'un point de vue théorique, ce compromis s'exprime de la façon suivante : un objet ne peut avoir son exécution interrompue par l'attente du résultat d'une requête faite auprès d'un autre objet. De plus, il ne peut y avoir 2 flots d'exécution actifs simultanément au sein d'un même objet sans être programmés explicitement.

OpenMASK distingue plusieurs styles de communication entre les objets de simulation. La communication standard permet à un objet de lire les attributs d'un autre objet de manière régulière. Un objet de simulation rend ses attributs lisibles par les autres objets de simulation en les plaçant dans des sorties ou des paramètres de contrôle, et il sont généralement lus à travers des entrées. L'autre moyen de communication se fait à travers des signaux émis par un objet, et reçus soit directement, soit à travers des auditeurs d'événements et sont conçus pour une communication sporadique. De ces notions est dérivée celle d'événement qui permet la communication entre deux objets spécifiés à l'avance et permet l'échange de requêtes entre les objets de simulation.

5.1 Les sorties

Une sortie est utilisée pour rendre public un attribut (une position par exemple) d'un objet de simulation dont la valeur a toujours un sens : quel que soit le moment ou cet attribut sera lu, la valeur lue doit avoir un sens. C'est pourquoi l'interpolation et l'extrapolation des valeurs d'une sortie sont légitimes, même si la sémantique de la valeur rendue publique dans la sortie ne se prête pas bien aux méthodes numériques classiques d'interpolation ou d'extrapolation fournit par le noyau d'OpenMASK. Par conséquent, il est possible d'associer un *polateur* (un objet réalisant l'extrapolation et l'interpolation) dédié à chaque sortie. Il convient de voir la déclaration d'une sortie comme une déclaration d'interface. Alors que les méthodologies de programmation objet recommandent de protéger les données d'un objet et de ne rendre public que ses attributs, avec OpenMASK c'est l'opposé qui est recommandée. Un objet rend publiques certaines de ces données, et protège les méthodes qui sont utilisées pour calculer son évolution. Pour cette raison, la création de sortie ne doit être faite que dans le constructeur de l'objet de simulation, le noyau supposant que l'interface d'un objet est constante après sa création. De plus, cette contrainte garantit que cette interface est totalement préservée par héritage. La construction d'une sortie nécessite 3 paramètres :

1. le type de valeurs de la sortie (c'est un paramètre template)
2. le nom de la sortie
3. (optionnel) un polateur. Si aucun polateur n'est spécifié, le polateur par défaut associé au type est utilisé.

Le polateur le plus simple est appelé polateur naïf, et est pertinent pour tout les types de données, puisqu'il ne peut calculer que des valeurs de la file de valeur conservant l'historique de la sortie. Les autres polateurs se servent de cet historique pour calculer les valeurs demandées.

La liste de toutes les sorties d'un objet de simulation est stockée dans une table des sorties qui est accessible par tous les autres objets afin de permettre la découverte dynamique des propriétés d'un objet. Cette forme simple de réflexivité est importante pour intégrer dans une même application des objets conçus pour des applications différentes et donc avec des graphes d'héritage indépendants.

5.2 Les paramètres de contrôle

Un paramètre de contrôle est une sortie spéciale pour 2 raisons. La première, c'est que tout objet de simulation peut tenter de changer la valeur d'un paramètre de contrôle, et la seconde est qu'un paramètre de contrôle n'est pas référencé dans la table des sorties mais dans une table annexe, la table de paramètre de contrôle. Cependant, il est tout a fait possible de brancher une entrée à un paramètre de contrôle.

Lorsqu'un objet autre que le propriétaire du paramètre de contrôle tente de changer la valeur de celui-ci, un événement valué est envoyé au propriétaire. Cette événement est interprété par défaut par un auditeur d'événement qui va remplacer la valeur du paramètre de contrôle par la nouvelle valeur. Mais il est bien-sûr possible de changer ce comportement par défaut en surchargeant l'auditeur d'événement (voir le paragraphe 5.5).

5.3 Les entrées

Une entrée est utilisée pour établir un chemin de données entre une sortie d'un autre objet et l'objet de simulation propriétaire de l'entrée. Une fois établi, ce chemin de donnée donne accès à la valeur de la sortie à laquelle l'entrée est branchée. Il y a deux type d'entrées :

1. les entrées privées : le branchement de ces entrées aux sorties d'un autre objet peut seulement être fait à la demande du propriétaire de l'entrée.
2. les entrées publiques : elles ont les même propriétés que les entrées privées, mais acceptent aussi des branchement fait à l'initiative du propriétaire de la sortie sur laquelle elles sont branchés. Par le même mécanisme que celui utilisé pour changer la valeur d'un paramètre de contrôle, lorsqu'une sortie prend l'initiative d'un branchement sur une entrée publique, cela provoque l'envoi d'un événement demandant le branchement qui est par défaut accepté par un auditeur d'événement associé à l'entrée, mais qui peut être surchargé.

5.3.1 Lecture d'une entrée

La méthode par défaut pour lire une entrée est d'utiliser la méthode `get()`. Cette méthode accepte un argument optionnel qui correspond à un retard entre la date de simulation courante et la date de la valeur renvoyée par la méthode `get`. Par exemple, `get(0)` renverra une valeur calculée par la sortie à l'aide de son polateur et correspondant à une valeur pour la date courante. Il est possible que cette valeur soit une valeur exacte ou une extrapolation. De même, `get(20)` renverra une valeur calculée pour correspondre à la valeur de la sortie 20 ms secondes avant la date courante. S'il est important que la valeur lue par l'entrée corresponde à une valeur produite par le propriétaire de la sortie, il faut utiliser la méthode `getLastExactValue()`.

En utilisant une entrée sensible, un service de plus haut niveau est fourni : la détection de nouvelles valeurs produites sur la sortie à laquelle l'entrée est branchée. Il devient ainsi possible de savoir si une nouvelle valeur a été produite sans la lire et la manipuler (pour la comparer à l'ancienne valeur lue par exemple). En utilisant une entrée sensible signalante, une évolution de l'entrée sensible, un événement est envoyé au propriétaire de l'entrée lorsque la valeur de la sortie change.

5.4 Les signaux

Un signal est une information émise par un objet (ou par le noyau pour les signaux systèmes) dans l'environnement. Les signaux sont différenciés entre eux par des identifiants (la signature du signal) et il est possible de leur

associer une valeur.

Les objets voulant être informé de l'émission d'un signal donné doivent s'enregistrer pour recevoir le signal. Si l'objet ne s'intéresse qu'aux signaux émis par un objet particulier, l'enregistrement à lieu auprès de cet objet, sinon il faut s'enregistrer auprès du contrôleur de la simulation. Lorsqu'un signal est émis par un objet, celui est transformé en événement envoyé à tous ceux qui ont manifesté de l'intérêt dans un signal avec la même signature. Par défaut la signature de l'événement envoyé est celle du signal émis, mais il est possible de donner un prototype d'événement à envoyer lors de l'enregistrement afin de spécifier la signature de l'événement reçu en réaction au signal émis.

Ainsi, en utilisant des signaux et des entrées sensibles signalante, il est possible d'utiliser un mode de programmation réactif pour faire communiquer les objets d'OpenMASK.

5.5 Événements, événements valués et auditeurs d'événement

Un événement est une structure de donnée composée de l'émetteur, le destinataire, la signature et la date d'émission de l'événement. Un événement valué est un événement auquel un champ de donnée supplémentaire (de n'importe quel type compatible avec OpenMASK) a été ajouté permettant à un événement de porter une valeur.

Le traitement des événements est fait au niveau de l'*object handle* (see 4.1) qui est la structure de données permettant au contrôleur de la simulation de manipuler des objets de simulation. Les événements reçus sont triés à leur arrivée selon leur date d'émission puis leur ordre d'arrivée.

Le fait pour un objet de simulation de réagir d'une façon prédéterminée à un certain nombre d'événements peut faire partie de son interface, et doit donc être préservé à travers l'héritage et être visible par réflection. Les auditeurs d'événements sont des objets remplissant ce rôle. Ils encapsulent des fragments de code qui sont associés à certains événements et qui sont automatiquement appelés lorsque ces événements sont reçus par l'objet. En tant qu'éléments de l'interface d'un objet ils doivent être construits dans le constructeur de l'objet de simulation.

6 Distribution et Parallélisme

Tout dans la conception d'OpenMASK a été fait pour permettre une distribution aisé des calculs, puisque toutes les interaction entre les objets on lieu à travers le noyau ou en utilisant des objets construit par le noyau. En particulier, les objets de simulation n'ont pas à être retouchés pour pouvoir être utiliser par un contrôleur distribué, puisque tous les problèmes d'intégrité des données sont à gérer au niveau des outils fournit par le contrôleur.

En supposant que le nombre d'objets de simulation à distribuer soit au moins d'une ordre de grandeur supérieur au nombre de processeurs disponibles, l'algorithme d'équilibrage de charge le plus simple (round robin) produit des résultats tout à fait acceptables.

C'est pourquoi les problèmes à résoudre lors de la distribution et de la parallélisation sont la cohérence de l'environnement virtuel et l'intégrité des données. L'intégrité des données est un problème devant être résolu par le contrôleur parallèle et la cohérence par le contrôleur distribué.

6.1 Le contrôleur parallèle

Le contrôleur parallèle (utile pour les machines multiprocesseur) est une adaptation directe du contrôleur classique. Ce contrôleur ordonnance les objets de simulation sur un des processeurs [4], en utilisant une stratégie d'allocation équilibrant le nombre d'objets ordonnancé sur chaque processeur. Puisque les sortie gèrent une file d'historique des valeurs, l'exclusion mutuelle entre l'écriture d'une nouvelle donnée et la lecture des valeurs de la sortie pour garantir l'intégrité des données est réglée de manière triviale.

Les résultats obtenus avec le contrôleur parallèle sont très dépendant de l'utilisation de la mémoire dynamique utilisée par l'application et ces différents modules, puisque l'allocation est un point de contention au niveau du système. Les résultats présenté dans le tableau 1, un cinquième des objets utilisent fortement l'allocation dynamique

4. En utilisant un thread par processeur

pour leur structures de données internes. L'application test est ici une simulation urbaine simulant 30 véhicules, avec leur simulation mécanique et comportementale associée. 5 objets de simulation sont utilisés pour chaque véhicule, et la simulation complète utilise 187 objets de simulation une fois l'animation du décor (en particulier des feux de circulation) prise en compte.

nombre d'activités	temps d'exécution en s	accélération
1	23,378	1
2	13,387	1,75
3	10,605	2,20
4	8,941	2,61

TAB. 1 – *Accélération obtenue sur une machine SMP à 4 processeurs*

6.2 Le contrôleur distribué

Les principes utilisés pour le contrôleur distribué ont déjà été présenté dans [SATR98, TD00b, Mar01] et peuvent être résumé en 2 points :

1. Chaque fois qu'un objet de simulation a besoin d'accéder aux données publiques d'un objet simulé sur un autre nœud, une copie locale appelée miroir est crée. Elle est alors synchronisée à chaque pas de temps avec la version originale appelé référentiel.
2. Le cohérence et la synchronisation de tous les nœuds est réalisée à l'aide d'un algorithme original qui permet la parallélisation du calcul d'un pas de simulation et le transfert sur le réseau des informations de mises à jour. Un paramètre de cet algorithme mettant en œuvre une cohérence relâchée contrainte, la latence, permet de borner le nombre de pas de simulation qu'un nœud peut faire sans avoir reçu des informations de mise à jour suffisamment récentes.

nombre de nœuds	temps d'exécution (ms)	accélération
1	91 882	1
2	64 500	1.42
3	51 800	1.77
4	46 000	1.99
5	40 000	2.29

TAB. 2 – *Accélération obtenu en utilisant le contrôleur distribué au-dessus de PVM*

7 Conclusion

Dans cet article, nous avons présenté un survol d'OpenMASK, une plate-forme pour la conception modulaire d'applications d'animation et de simulation pouvant utiliser des noyaux d'exécution parallèle ou distribué, disponible en téléchargement. Cette plate-forme pour les applications de réalité virtuelle cherche à combiner abstraction, performance, distribution et réutilisation de composants logiciels dans le domaine de la réalité virtuelle, sans limiter le domaine d'application.

Nous croyons qu'OpenMASK représente un compromis intéressant, puisqu'il est déjà utilisé comme plate-forme logicielle pour différents programme de recherche. Ces programmes incluent des applications de réalité virtuelle coopérative, d'interaction utilisateur en environnement virtuel, de retour haptique, de simulation comportementale, de capture de mouvement et de contrôle de mouvement. OpenMASK est en particulier utilisé au sein de Perf-RV.

Références

[BC94] W. Bricken and G. Coco. The VEOS project. *Presence*, 3(2):111–129, 1994.

[BWA96] J. Barrus, R. Waters, and D. Anderson. Locales and beacons: Precise and efficient support for large multi-user virtual environments. *Proceedings of VRAIS'96, Santa Clara CA*, pages 204–213, 1996.

[CJD00] C. Greenhalg, J. Purbrick, and D. Snowdon. Inside MASSIVE-3: Flexible support for data consistency and world structuring. In *Collaborative Virtual Environments*, number 1-58113-303-0, pages 139–146. ACM, september 2000.

[CMBZ00] Michael Capps, Don McGregor, Don Brutzman, and Michael Zyda. Projects in VR: NPSNET-V: A new beginning for dynamically extensible virtual environments. *IEEE Computer Graphics and Applications*, 20(5):12–15, September/October 2000.

[GB95] C. Greenhalgh and S. Benford. MASSIVE: A distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 27–35, Los Alamitos, CA, USA, May30 June–2 1995. IEEE Computer Society Press.

[JBBCN98] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira. Vr juggler: A framework for virtual reality development. In *Proceedings of the 2nd International Immersive Projection Technology Workshop*, 1998.

[KYN+00] K. Park, Y. Cho, N. Krishnaprasad, C. Scharver, M. Lewis, J. Leigh, and A. Johnson. Cavernsoft g2: A toolkit for high performance tele-immersive collaboration. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology 2000*, pages pp. 8–15, Oct 22-25 2000.

[LHMM97] Rodger Lea, Yasuaki Honda, Kouichi Matsuda, and Satoru Matsuda. Community place: Architecture and performance. In Rikk Carey and Paul Strauss, editors, *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, February 1997. ACM SIGGRAPH / ACM SIGCOMM, ACM Press. ISBN 0-89791-886-x.

[Mar01] David Margery. *Environnement logiciel temps-réel distribué pour la simulation sur réseau de PC*. PhD thesis, Université de Rennes 1, 2001.

[MDM+95] M. R. Macedonia, D. P. Brutzmann, M. J. Zyda, D. R. Pratt, P. T. Barham, J. Falby, and J. Locke. NPSNET: A multi-player 3D virtual environment over the internet. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 93–94. ACM SIGGRAPH, apr 1995. ISBN 0-89791-736-7.

[MJM00] M. Oliveira, J. Crowcroft, and M. Slater. Componant framework infrstructure for virtual environments. In *Collaborative Virtual Environments*, number 1-58113-303-0, pages 139–146. ACM, september 2000.

[RJM+99] R. Hubbold, J. Cook, M. Keates, S. Gibson, T. Howard, A. Murta, A. West, and S. Pettifer. Gnu/maverik a micro-kernel for large-scale virtual environments. In *VRST99*, December 1999.

[SATR98] S. Donikian, A. Chauffaut, T. Duval, and R. Kulpa. Gasp: from modular programming to distributed execution. In *Computer Animation'98, IEEE, Philadelphia, USA*, pages 79–87, june 1998.

[SG93] C. Shaw and M. Green. The MR toolkit peers package and experiment. *Proceedings of VRAIS'93*, pages 463–469, 1993.

[SJJA00] S. Pettifer, J. Cook, J. Marsh, and A. West. Deva3: Architecture for a large-scale distributed virtual reality system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology 2000*, Oct 22-25 2000.

[TD00a] T. Duval and D. Margery. Building objects and interactors for collaborative interactions with gasp. In *Proceedings of the Third International Conference on Collaborative Virtual Environments (CVE'2000)*, pages 129–138. ACM, September 2000.

[TD00b] T. Duval and D. Margery. Using gasp for collaborative interactions within 3d virtual worlds. In *Proceedings of the Second International Conference on Virtual Worlds (VW'2000)*, Paris, France, july 2000. Springer LNCS/AI.

[Tra99] Henrik Tramberend. Avango: A distributed virtual reality framework. In *Proc. Of the IEEE Virtual Reality 1999*. IEEE, march 1999.

[Wat00] Kent Watsen. Bamboo: A platform and language independant mechanism enabling dynamically reconfigurable applications. présenté au 4th Annual Workshop on Distributed System Aspects of Sharing a Virtual Reality, September 2000.

[WGS95] Q. Wang, M. Green, and C. Shaw. EM - an environment manager for building networked virtual environments. *Proceedings of VRAIS'95*, 1995.

[WHH+92] A. West, T. Howard, R. Hubbold, A. Murta, D. Snowdon, and D. Butler. AVIARY - A generic virtual reality interface for real applications. *Proceedings of Virtual Reality Systems, London*, 1992.